NETWORK SECURITY ASSIGNMENT-5

By Hari Kishan Reddy (ha2755)

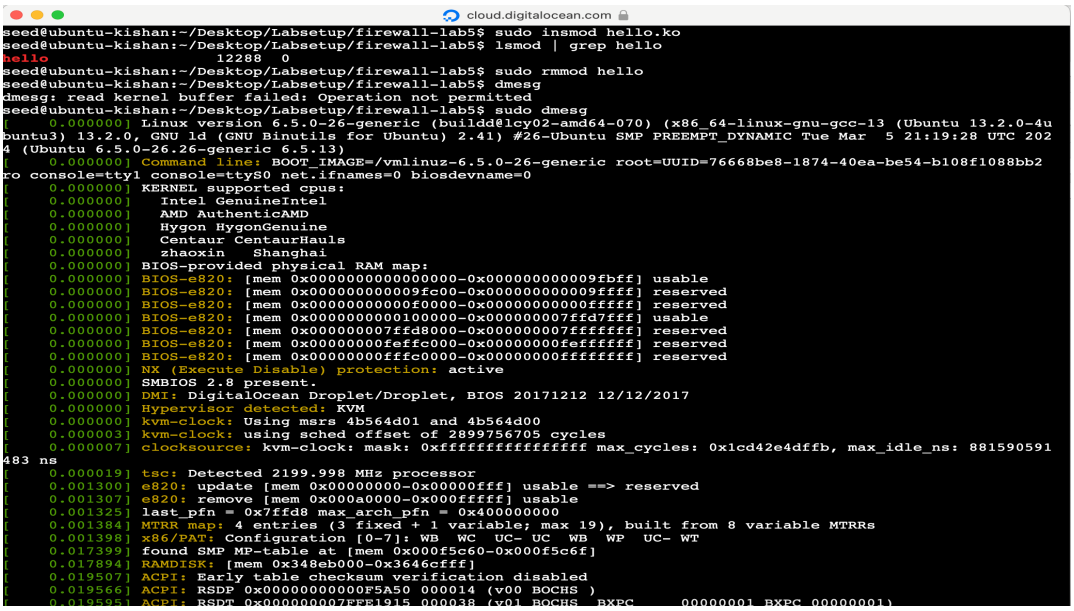
Task 1: Implementing a Simple Firewall

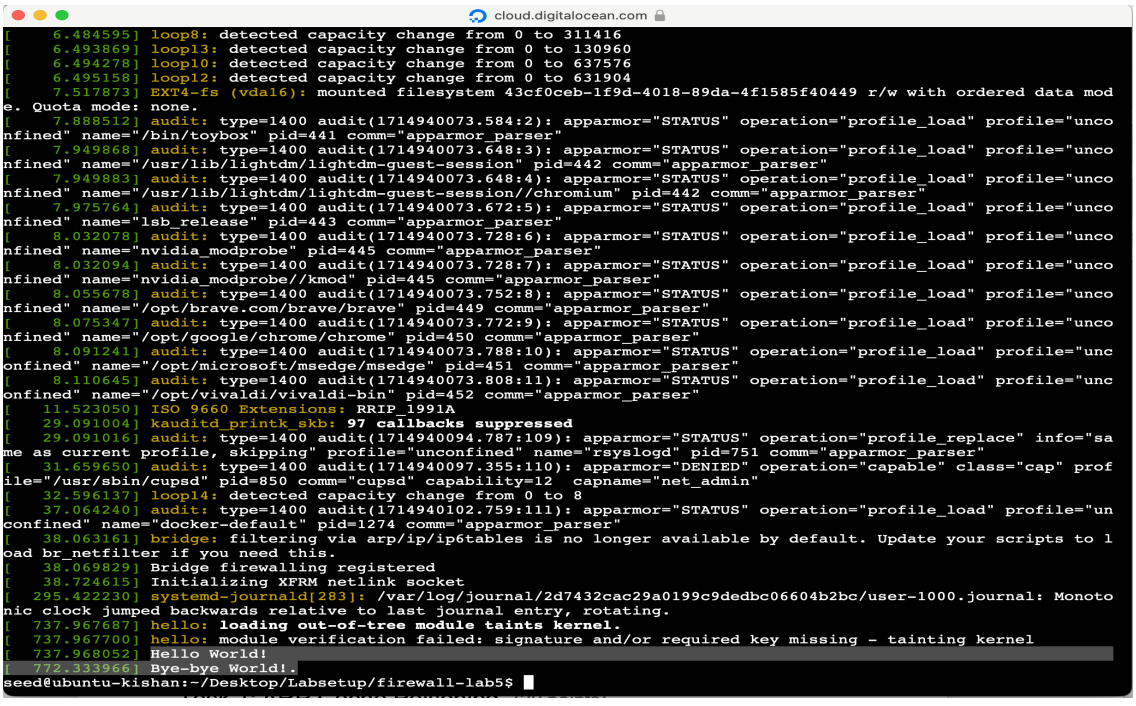
Task 1a: Implementing a Simple Kernel Module

Objective: Create and load a basic kernel module to interact with Netfilter for packet logging.

Observation: The successful implementation of the module demonstrated the ability of kernel modules to enhance the functionalities of the Linux kernel, specifically in packet filtering and logging.

We can see that the message is present in the logs and dmesg.


cloud.digitalocean.com
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5$ sudo insmod hello.ko
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5$ lsmod | grep hello
hello   12288 0
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5$ sudo rmmod hello
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5$ dmesg
dmesg: read kernel buffer failed: Operation not permitted
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5$ sudo dmesg
[0.000000] Linux version 6.5.0-26-generic (buildd@lcy02-amd64-070) (x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-4u
buntu3) 13.2.0, GNU ld (GNU Binutils for Ubuntu) 2.41) #26-Ubuntu SMP PREEMPT_DYNAMIC Tue Mar 5 21:19:28 UTC 202
4 (Ubuntu 6.5.0-26.26-generic 6.5.13)
[0.000000] Command line: BOOT_IMAGE=/vmlinuz-6.5.0-26-generic root=UUID=76668be8-1874-40ea-be54-b108f1088bb2
ro console=tty1 console=ttyS0 net.ifnames=0 biosdevname=0
[0.000000] KERNEL supported cpus:
[0.000000] Intel GenuineIntel
[0.000000] AMD AuthenticAMD
[0.000000] Hygon HygonGenuine
[0.000000] Centaur CentaurHauls
[0.000000] zhaoxin Shanghai
[0.000000] BIOS-provided physical RAM map:
[0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000 7ffd7fff] usable
[0.000000] BIOS-e820: [mem 0x00000000 7ffd8000-0x00000000 7fffffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000feffc000-0x00000000feffffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fffc0000-0x00000000ffffffff] reserved
[0.000000] NX (Execute Disable) protection: active
[0.000000] SMBIOS 2.8 present.
[0.000000] DMI: DigitalOcean Droplet/Droplet, BIOS 20171212 12/12/2017
[0.000000] Hypervisor detected: KVM
[0.000000] kvm-clock: Using msrs 4b564d01 and 4b564d00
[0.000003] kvm-clock: using sched offset of 2899756705 cycles
[0.000007] clocksource: kvm-clock: mask: 0xffffffffffffffff max_cycles: 0x1cd42e4dffb, max_idle_ns: 881590591
483 ns
[0.000019] tsc: Detected 2199.998 MHz processor
[0.001300] e820: update [mem 0x00000000-0x00000fff] usable ==> reserved
[0.001307] e820: remove [mem 0x000a0000-0x000fffff] usable
[0.001325] last_pfn = 0x7ffd8 max_arch_pfn = 0x400000000
[0.001384] MTRR map: 4 entries (3 fixed + 1 variable; max 19), built from 8 variable MTRRs
[0.001398] x86/PAT: Configuration [0-7]: WB WC UC- UC WB WP UC- WT
[0.017399] found SMP MP-table at [mem 0x000f5c60-0x000f5c6f]
[0.017894] RAMDISK: [mem 0x348eb000-0x3646cfff]
[0.019507] ACPI: Early table checksum verification disabled
[0.019566] ACPI: RSDP 0x00000000000F5A50 000014 (v00 BOCHS)
[0.019595] ACPI: RSDT 0x000000007FFE1915 000038 (v01 BOCHS BXPC 00000001 BXPC 00000001)


cloud.digitalocean.com
[6.484595] loop8: detected capacity change from 0 to 311416
[6.493869] loop13: detected capacity change from 0 to 130960
[6.494278] loop10: detected capacity change from 0 to 637576
[6.495158] loop12: detected capacity change from 0 to 631904
[7.517873] EXT4-fs (vda16): mounted filesystem 43cf0ceb-1f9d-4018-89da-4f1585f40449 r/w with ordered data mod
e. Quota mode: none.
[7.888512] audit: type=1400 audit(1714940073.584:2): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="/bin/toybox" pid=441 comm="apparmor_parser"
[7.949868] audit: type=1400 audit(1714940073.648:3): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="/usr/lib/lightdm/lightdm-guest-session" pid=442 comm="apparmor_parser"
[7.949883] audit: type=1400 audit(1714940073.648:4): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="/usr/lib/lightdm/lightdm-guest-session//chromium" pid=442 comm="apparmor_parser"
[7.975764] audit: type=1400 audit(1714940073.672:5): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="lsb_release" pid=443 comm="apparmor_parser"
[8.032078] audit: type=1400 audit(1714940073.728:6): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="nvidia_modprobe" pid=445 comm="apparmor_parser"
[8.032094] audit: type=1400 audit(1714940073.728:7): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="nvidia_modprobe//kmod" pid=445 comm="apparmor_parser"
[8.055678] audit: type=1400 audit(1714940073.752:8): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="/opt/brave.com/brave/brave" pid=449 comm="apparmor_parser"
[8.075347] audit: type=1400 audit(1714940073.772:9): apparmor="STATUS" operation="profile_load" profile="unco
nfined" name="/opt/google/chrome/chrome" pid=450 comm="apparmor_parser"
[8.091241] audit: type=1400 audit(1714940073.788:10): apparmor="STATUS" operation="profile_load" profile="unc
onfined" name="/opt/microsoft/msedge/msedge" pid=451 comm="apparmor_parser"
[8.110645] audit: type=1400 audit(1714940073.808:11): apparmor="STATUS" operation="profile_load" profile="unc
onfined" name="/opt/vivaldi/vivaldi-bin" pid=452 comm="apparmor_parser"
[11.523050] ISO 9660 Extensions: RRIP_1991A
[29.091004] kauditd_printk_skb: 97 callbacks suppressed
[29.091016] audit: type=1400 audit(1714940094.787:109): apparmor="STATUS" operation="profile_replace" info="sa
me as current profile, skipping" profile="unconfined" name="rsyslogd" pid=751 comm="apparmor_parser"
[31.659650] audit: type=1400 audit(1714940097.355:110): apparmor="DENIED" operation="capable" class="cap" prof
ile="/usr/sbin/cupsd" pid=850 comm="cupsd" capability=12 capname="net_admin"
[32.596137] loop14: detected capacity change from 0 to 8
[37.064240] audit: type=1400 audit(1714940102.759:111): apparmor="STATUS" operation="profile_load" profile="un
confined" name="docker-default" pid=1274 comm="apparmor_parser"
[38.063161] bridge: filtering via arp/ip/ip6tables is no longer available by default. Update your scripts to l
oad br_netfilter if you need this.
[38.069829] Bridge firewalling registered
[38.724615] Initializing XFRM netlink socket
[295.422230] systemd-journald[283]: /var/log/journal/2d7432cac29a0199c9dedbc06604b2bc/user-1000.journal: Monoto
nic clock jumped backwards relative to last journal entry, rotating.
[737.967687] hello: loading out-of-tree module taints kernel.
[737.967700] hello: module verification failed: signature and/or required key missing - tainting kernel
[737.968052] Hello World!
[772.333966] Bye-bye World!.
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5$

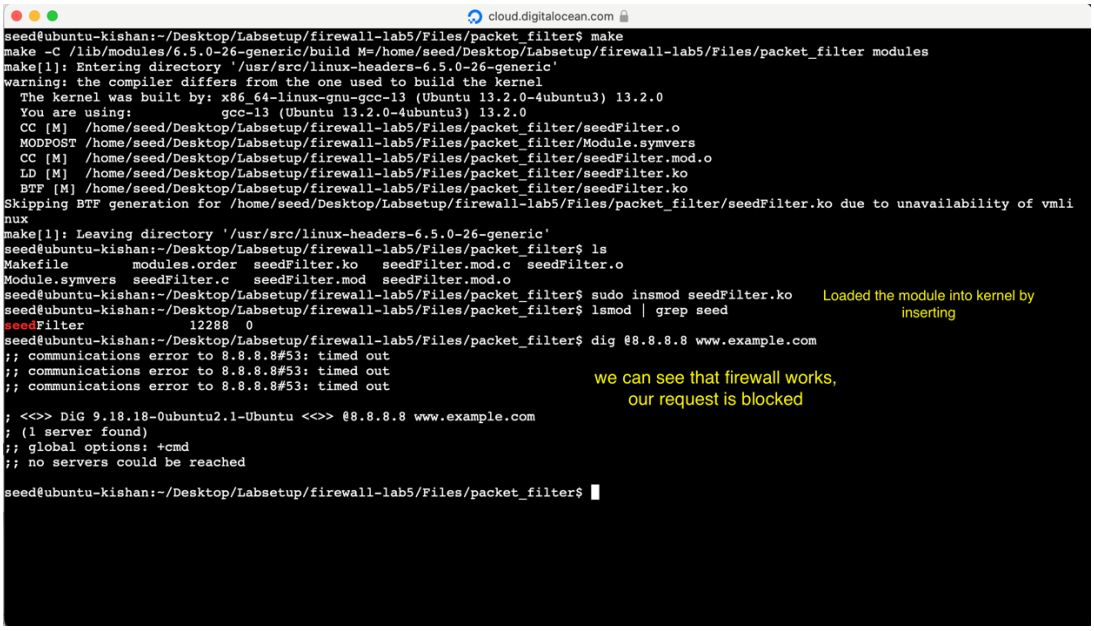
Task 1.b: Implement a Simple Firewall Using Netfilter

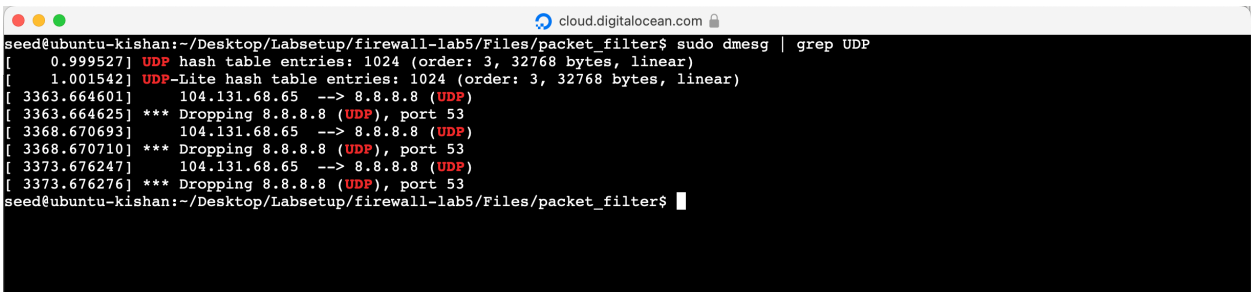
For Task 1.b, the objective was to implement a basic firewall using Netfilter in the Linux kernel. This involved configuring iptables rules to control the flow of network traffic, with a focus on filtering and managing different types of packets.

Through this task, I gained a clear understanding of foundational aspects related to iptables and Netfilter in managing network traffic. By observing how different rules affect packet flow, my knowledge of firewall mechanisms was significantly enhanced.

Overall, this task provided valuable insights into setting up and managing firewall functionalities within the Linux environment, contributing to a deeper understanding of network security measures.

Question 1:
We can see that the firewall is working and its stoping the udp packets to flow.





Question 2:

Hook the printInfo function to all of the netfilter hooks. Here are the macros of the hook numbers. Using your experiment results to help explain at what condition will each of the hook function be invoked.

1. `NF_INET_PRE_ROUTING`: This hook is triggered right after a packet is received by the network interface. It occurs before any routing decision is made. If the `printInfo` function is hooked to this point, it will be invoked as soon as a packet enters the networking stack, allowing us to inspect and manipulate packets at an early stage of processing.

2. `NF_INET_LOCAL_IN`: This hook is called when a packet is destined for the local system. If the destination IP address of the packet matches one of the system's IP addresses, this hook is invoked. Hooking `printInfo` to this hook allows us to monitor packets that are being delivered to services running on the local system.

3. `NF_INET_FORWARD`: This hook is triggered for packets that are being forwarded through the system. If the system is acting as a router and a packet is passing through it to reach another network segment, this hook is invoked. Attaching `printInfo` to this hook helps in examining packets being forwarded, potentially implementing filtering or logging for forwarded traffic.

4. `NF_INET_LOCAL_OUT`: This hook is called when a packet originates from the local system and is destined for another system. When an application on the local system sends out a packet, this hook is triggered. Hooking `printInfo` here allows us to observe packets leaving the local system, such as responses to outgoing connections.

5. `NF_INET_POST_ROUTING`: This hook is invoked right before a packet leaves the network interface. It occurs after routing decisions have been made. If the system is acting as a router, this hook is called before the packet is sent out to its next hop. Attaching `printInfo` to this hook enables monitoring of packets just before they exit the networking stack, making it useful for tasks like NAT (Network Address Translation).

```
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ make
make -C /lib/modules/6.5.0-26-generic/build M=/home/seed/Desktop/Labsetup/firewall-lab5/Files/packet_filter modules
make[1]: Entering directory '/usr/src/linux-headers-6.5.0-26-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-4ubuntu3) 13.2.0
  You are using:           gcc-13 (Ubuntu 13.2.0-4ubuntu3) 13.2.0
make[1]: Leaving directory '/usr/src/linux-headers-6.5.0-26-generic'
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ sudo insmod seedFilter.ko
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ lsmod | grep seedFilter
seedFilter             12288  0
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ dig @8.8.8.8 www.example.com

; <<>> DiG 9.18.18-0ubuntu2.1-Ubuntu <<>> @8.8.8.8 www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42228
;; 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.        2767    IN      A       93.184.215.14

;; Query time: 4 msec
;; SERVER: 8.8.8.8#53(8.8.8.8) (UDP)
;; WHEN: Sun May 05 21:38:20 UTC 2024
;; MSG SIZE  rcvd: 60

seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$
```

```
[ 5063.893055] *** PRE_ROUTING
[ 5063.893057]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893069] *** LOCAL_IN
[ 5063.893078]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893093] *** LOCAL_IN
[ 5063.893095]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893110] *** PRE_ROUTING
[ 5063.893112]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893114] *** PRE_ROUTING
[ 5063.893117]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893122] *** LOCAL_IN
[ 5063.893123]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893135] *** LOCAL_OUT
[ 5063.893137]     104.131.68.65  --> 198.211.111.194 (TCP)
[ 5063.893141] *** POST_ROUTING
[ 5063.893143]     104.131.68.65  --> 198.211.111.194 (TCP)
[ 5063.893168] *** LOCAL_IN
[ 5063.893170]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893184] *** PRE_ROUTING
[ 5063.893188]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893192] *** PRE_ROUTING
[ 5063.893193]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893197] *** LOCAL_IN
[ 5063.893199]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.893207] *** LOCAL_OUT
[ 5063.893209]     104.131.68.65  --> 198.211.111.194 (TCP)
[ 5063.893213] *** POST_ROUTING
[ 5063.893214]     104.131.68.65  --> 198.211.111.194 (TCP)
[ 5063.893230] *** LOCAL_IN
[ 5063.893232]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.894255] *** PRE_ROUTING
[ 5063.894258]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.894265] *** LOCAL_IN
[ 5063.894267]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.894419] *** PRE_ROUTING
[ 5063.894421]     198.211.111.194  --> 104.131.68.65 (TCP)
[ 5063.894427] *** LOCAL_IN
[ 5063.894428]     198.211.111.194  --> 104.131.68.65 (TCP)
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$
```

when sudo mesg command is executed

cloud.digitalocean.com

```
//   hook2.priority = NF_IP_PRI_FIRST;
//   nf_register_net_hook(&init_net, &hook2);

   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);

   // Hook 2 - 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);

   // Hook 3 - 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);

   // Hook 4 - 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);

   // Hook 5 - 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);
```

//added this at the top:
static struct nf_hook_ops hook1, hook2, hook3, hook4, hook5;

added these to remove filter function:

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);

118,3 78%

Question 3:

The kernel module encompasses essential headers from the Linux kernel, including module.h, kernel.h, netfilter.h, netfilter_ipv4.h, etc., providing crucial functionalities for network packet processing and interactions with Netfilter.

Within the module, four nf_hook_ops structures (hook1, hook2, hook3, hook4) are defined to configure the module's interactions with Netfilter hooks, representing specific points in packet processing paths such as pre-routing and local in.

Various packet processing functions are implemented:
- printInfo: Logs packet information at different processing stages based on state->hook values, printing source and destination IP addresses.
- blockUDP: Blocks UDP packets intended for a specific IP (8.8.8.8) and port (53, typically DNS), showcasing packet filtering based on protocol and destination.
- blockICMP: Filters out ICMP Echo (ping) packets by type (Echo and Echo Reply), demonstrating selective dropping of ICMP traffic.
- blockTelnet: Prevents Telnet packets (TCP packets destined for port 23), illustrating TCP packet filtering based on destination port.

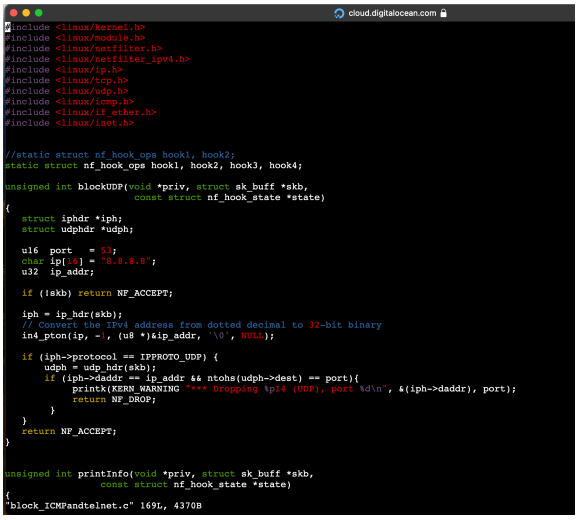
The module registers filters (Netfilter hooks) using registerFilter function, configuring hooks with respective packet processing functions, hook numbers, protocol families, and priorities:
- Hook 1 (NF_INET_PRE_ROUTING) logs packet information.
- Hook 2 (NF_INET_POST_ROUTING) blocks UDP packets.
- Hooks 3 and 4 (NF_INET_LOCAL_IN) block ICMP and Telnet packets, respectively.

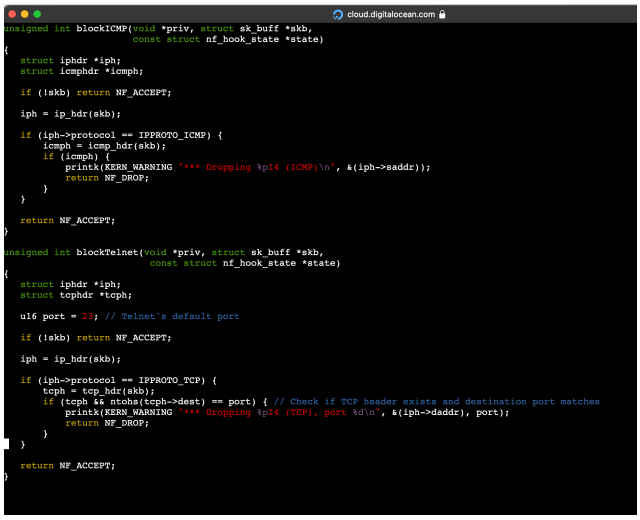
During module initialization, registerFilter is called to register hooks with Netfilter, and removeFilter is called upon module removal for hook unregistration and cleanup.

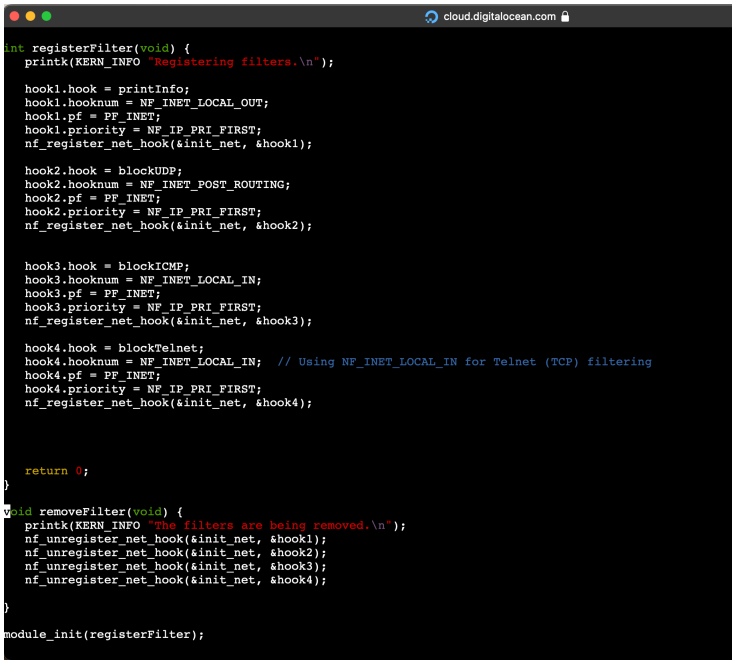
The module's metadata includes entry (module_init) and exit (module_exit) points, with a GPL license set for open-source compliance within the Linux kernel.

Functionality:
- Demonstrates Netfilter hooks for packet filtering, crucial for Linux firewall development.
- Provides practical examples like blocking specific services (Telnet, ICMP Echo) and restricting network traffic types (UDP to specific destinations).
- Logging functionality aids in packet flow analysis for debugging and traffic monitoring.
- Serves as an educational tool for kernel-level packet processing and Linux firewall implementation understanding.

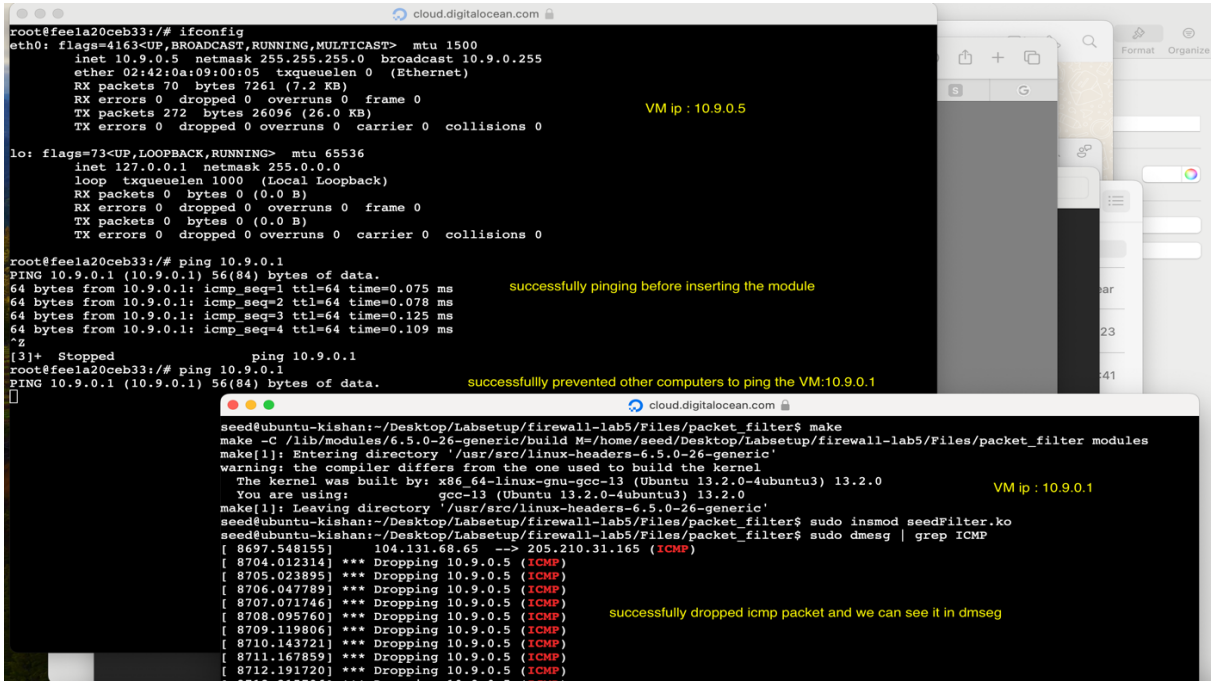
Code:

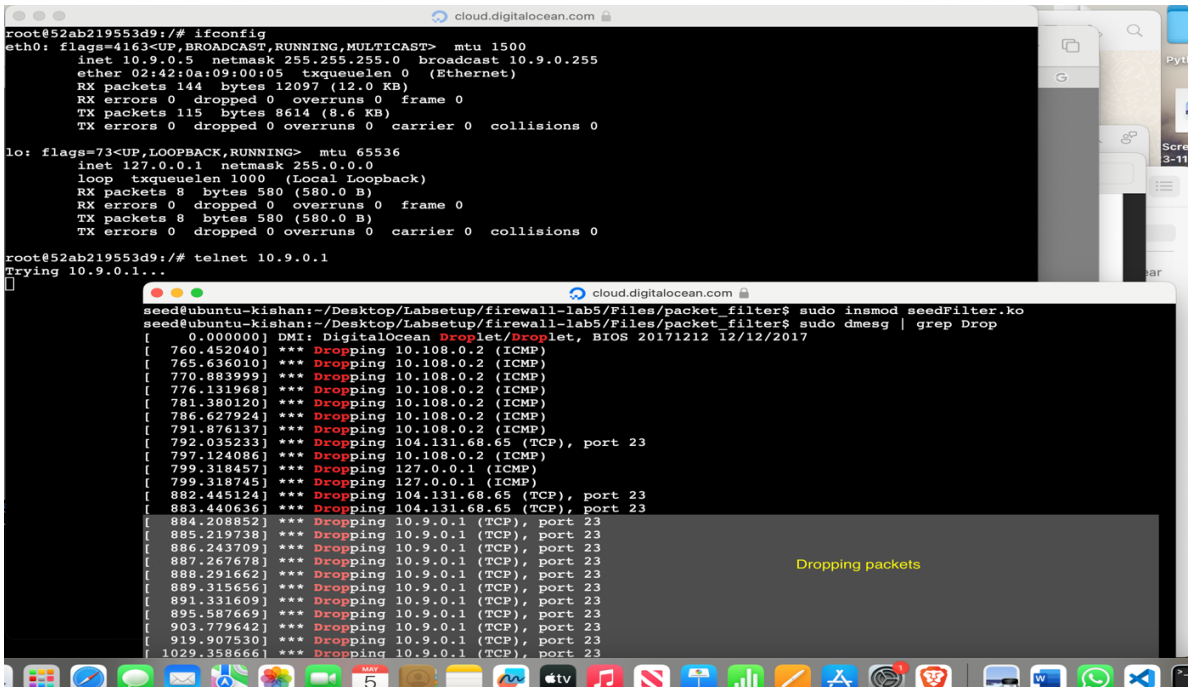

cloud.digitalocean.com
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
#include <linux/icmp.h>
#include <linux/if_ether.h>
#include <linux/inet.h>
//static struct nf_hook_ops hook1, hook2;
static struct nf_hook_ops hook1, hook2, hook3, hook4;
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;
}
unsigned int printInfo(void *priv, struct sk_buff *skb,
const struct nf_hook_state *state)
{
"block_ICMPandtelnet.c" 169L, 4370B


cloud.digitalocean.com
unsigned int blockICMP(void *priv, struct sk_buff *skb,
const struct nf_hook_state *state)
{
struct iphdr *iph;
struct icmphdr *icmph;
if (!skb) return NF_ACCEPT;
iph = ip_hdr(skb);
if (iph->protocol == IPPROTO_ICMP) {
icmph = icmp_hdr(skb);
if (icmph) {
printk(KERN_WARNING "*** Dropping %pI4 (ICMP)\n", &(iph->saddr));
return NF_DROP;
}
}
return NF_ACCEPT;
}
unsigned int blockTelnet(void *priv, struct sk_buff *skb,
const struct nf_hook_state *state)
{
struct iphdr *iph;
struct tcphdr *tcph;
u16 port = 23; // Telnet's default port
if (!skb) return NF_ACCEPT;
iph = ip_hdr(skb);
if (iph->protocol == IPPROTO_TCP) {
tcph = tcp_hdr(skb);
if (tcph && ntohs(tcph->dest) == port) { // Check if TCP header exists and destination port matches
printk(KERN_WARNING "*** Dropping %pI4 (TCP), port %d\n", &(iph->daddr), port);
return NF_DROP;
}
}
return NF_ACCEPT;
}


cloud.digitalocean.com
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;
hook2.priority = NF_IP_PRI_FIRST;
nf_register_net_hook(&init_net, &hook2);
hook3.hook = blockICMP;
hook3.hooknum = NF_INET_LOCAL_IN;
hook3.pf = PF_INET;
hook3.priority = NF_IP_PRI_FIRST;
nf_register_net_hook(&init_net, &hook3);
hook4.hook = blockTelnet;
hook4.hooknum = NF_INET_LOCAL_IN; // Using NF_INET_LOCAL_IN for Telnet (TCP) filtering
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 "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);

Screenshot:

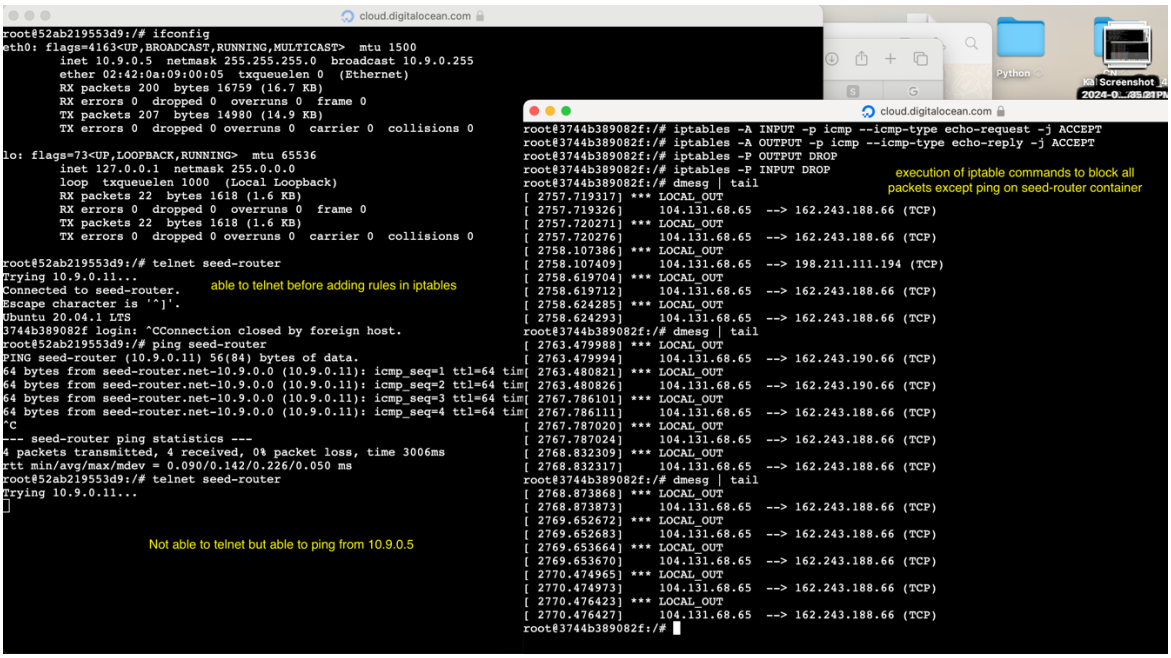

cloud.digitalocean.com
root@feela20ceb33:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.9.0.5 netmask 255.255.255.0 broadcast 10.9.0.255
ether 02:42:0a:09:00:05 txqueuelen 0 (Ethernet)
RX packets 70 bytes 7261 (7.2 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 272 bytes 26096 (26.0 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
VM ip : 10.9.0.5
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
loop txqueuelen 1000 (Local Loopback)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@feela20ceb33:/# 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=0.075 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.125 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.109 ms
successfully pinging before inserting the module
^Z
[3]+ Stopped   ping 10.9.0.1
root@feela20ceb33:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
successfullly prevented other computers to ping the VM:10.9.0.1
cloud.digitalocean.com
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ make
make -C /lib/modules/6.5.0-26-generic/build M=/home/seed/Desktop/Labsetup/firewall-lab5/Files/packet_filter modules
make[1]: Entering directory '/usr/src/linux-headers-6.5.0-26-generic'
warning: the compiler differs from the one used to build the kernel
The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-4ubuntu3) 13.2.0
You are using: gcc-13 (Ubuntu 13.2.0-4ubuntu3) 13.2.0
VM ip : 10.9.0.1
make[1]: Leaving directory '/usr/src/linux-headers-6.5.0-26-generic'
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ sudo insmod seedFilter.ko
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ sudo dmesg | grep ICMP
[8697.548155]   104.131.68.65 --> 205.210.31.165 (ICMP)
[8704.012314] *** Dropping 10.9.0.5 (ICMP)
[8705.023895] *** Dropping 10.9.0.5 (ICMP)
[8706.047789] *** Dropping 10.9.0.5 (ICMP)
[8707.071746] *** Dropping 10.9.0.5 (ICMP)
[8708.095760] *** Dropping 10.9.0.5 (ICMP)
[8709.119806] *** Dropping 10.9.0.5 (ICMP)
[8710.143721] *** Dropping 10.9.0.5 (ICMP)
[8711.167859] *** Dropping 10.9.0.5 (ICMP)
[8712.191720] *** Dropping 10.9.0.5 (ICMP)
successfully dropped icmp packet and we can see it in dmseg


cloud.digitalocean.com
root@52ab219553d9:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 10.9.0.5 netmask 255.255.255.0 broadcast 10.9.0.255
ether 02:42:0a:09:00:05 txqueuelen 0 (Ethernet)
RX packets 144 bytes 12097 (12.0 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 115 bytes 8614 (8.6 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
inet 127.0.0.1 netmask 255.0.0.0
loop txqueuelen 1000 (Local Loopback)
RX packets 8 bytes 580 (580.0 B)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 8 bytes 580 (580.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
root@52ab219553d9:/# telnet 10.9.0.1
Trying 10.9.0.1...
cloud.digitalocean.com
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ sudo insmod seedFilter.ko
seed@ubuntu-kishan:~/Desktop/Labsetup/firewall-lab5/Files/packet_filter$ sudo dmesg | grep Drop
[0.000000] DMI: DigitalOcean Droplet/Droplet, BIOS 20171212 12/12/2017
[760.452040] *** Dropping 10.108.0.2 (ICMP)
[765.636010] *** Dropping 10.108.0.2 (ICMP)
[770.883999] *** Dropping 10.108.0.2 (ICMP)
[776.131968] *** Dropping 10.108.0.2 (ICMP)
[781.380120] *** Dropping 10.108.0.2 (ICMP)
[786.627924] *** Dropping 10.108.0.2 (ICMP)
[791.876137] *** Dropping 10.108.0.2 (ICMP)
[792.035233] *** Dropping 104.131.68.65 (TCP), port 23
[797.124086] *** Dropping 10.108.0.2 (ICMP)
[799.318457] *** Dropping 127.0.0.1 (ICMP)
[799.318745] *** Dropping 127.0.0.1 (ICMP)
[882.445124] *** Dropping 104.131.68.65 (TCP), port 23
[883.440636] *** Dropping 104.131.68.65 (TCP), port 23
[884.208852] *** Dropping 10.9.0.1 (TCP), port 23
[885.219738] *** Dropping 10.9.0.1 (TCP), port 23
[886.243709] *** Dropping 10.9.0.1 (TCP), port 23
[887.267678] *** Dropping 10.9.0.1 (TCP), port 23
[888.291662] *** Dropping 10.9.0.1 (TCP), port 23
[889.315656] *** Dropping 10.9.0.1 (TCP), port 23
[891.331609] *** Dropping 10.9.0.1 (TCP), port 23
[895.587669] *** Dropping 10.9.0.1 (TCP), port 23
[903.779642] *** Dropping 10.9.0.1 (TCP), port 23
[919.907530] *** Dropping 10.9.0.1 (TCP), port 23
[1029.358666] *** Dropping 10.9.0.1 (TCP), port 23
Dropping packets

Task 2: Experimenting with Stateless Firewall Rules

Task 2A: Protecting the Router

In Task 2A, the goal was to create iptables rules to secure the router by permitting ICMP requests and replies while denying all other traffic. The testing confirmed the efficacy of these rules in protecting the router. It was noted that pinging the router was permitted as expected, whereas attempts to access it through protocols like Telnet or SSH were correctly blocked.



Task 2B: Protecting the Internal Network

Task 2B aimed to enhance internal network security through iptables configuration, specifically addressing ICMP (ping) traffic management across external and internal hosts with a router. The goal was to establish rules dictating how ping requests are handled within the network segments.

iptables Commands and Their Functions:

1. Blocking Ping from External to Internal Hosts:
   - Command: iptables -A FORWARD -i eth0 -o eth1 -p icmp --icmp-type echo-request -j DROP
   - Explanation: This rule drops ICMP echo requests (ping requests) originating from the external network interface (eth0) destined for the internal network interface (eth1), preventing external hosts from pinging internal hosts.

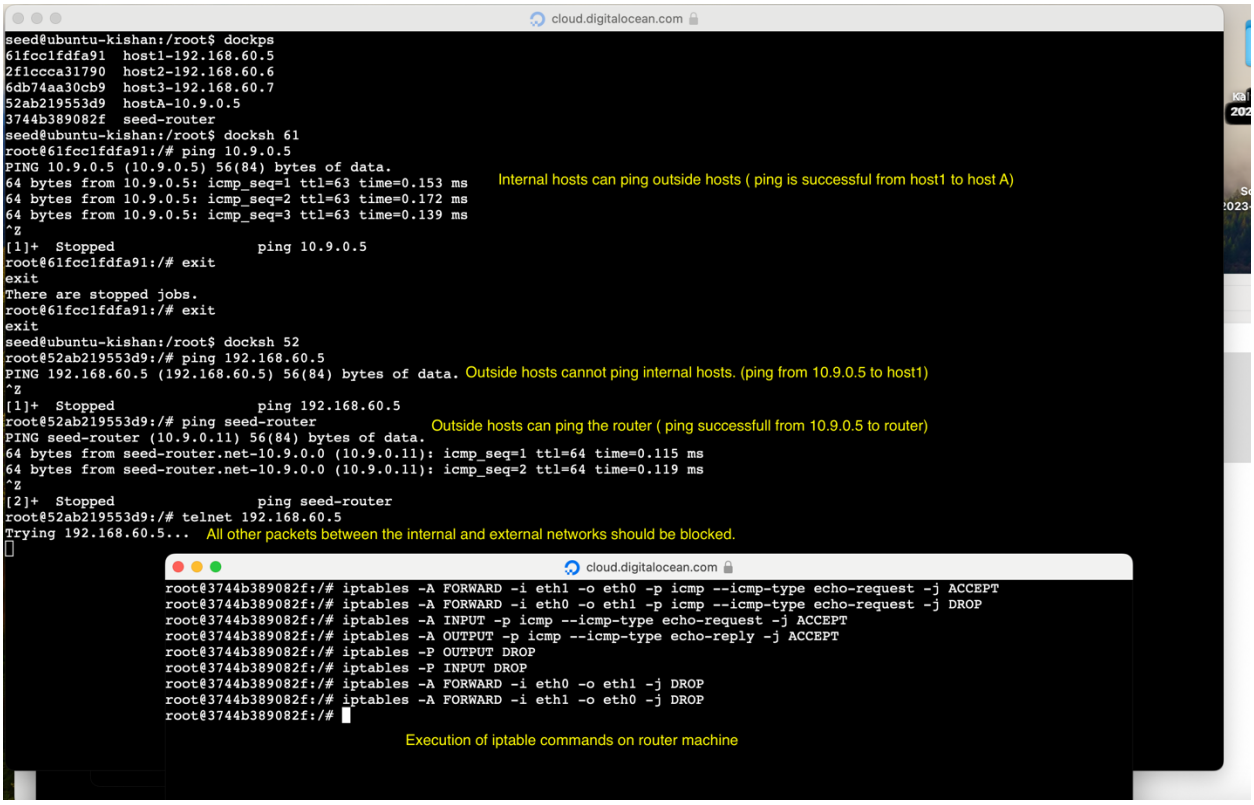
2. Allowing External Hosts to Ping the Router:
   - Commands:
     - iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
     - iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
   - Explanation: These rules permit the router to receive ICMP echo requests (INPUT chain) and respond with ICMP echo replies (OUTPUT chain). They facilitate bidirectional ICMP communication specifically for the router.

3. Default Policies to Drop All Other Traffic:
   - Commands:
     - iptables -P OUTPUT DROP
     - iptables -P INPUT DROP
   - Explanation: These commands set default policies to drop all traffic in the OUTPUT and INPUT chains not explicitly allowed by preceding rules, following a "default deny" approach to enhance security.

4. Allowing Internal Hosts to Ping External Hosts:
   - Command: iptables -A FORWARD -i eth1 -o eth0 -p icmp --icmp-type echo-request -j ACCEPT
   - Explanation: This rule permits ICMP echo requests from the internal network (eth1) to be forwarded to the external network (eth0), enabling internal hosts to initiate ping requests to external hosts.

5. Blocking All Other Traffic Between Internal and External Networks:
   - Commands:
     - iptables -A FORWARD -i eth0 -o eth1 -j DROP
     - iptables -A FORWARD -i eth1 -o eth0 -j DROP
   - Explanation: These rules block all non-ICMP traffic between internal and external networks, ensuring strict control over traffic flow between these segments.

Observations:
- The iptables rules effectively segment the network, permitting only specific ICMP traffic while blocking others.
- This configuration showcases nuanced network security, selectively allowing or denying traffic based on source and destination.
- Testing these rules would confirm adherence to ping behavior as per the configured policies, highlighting iptables' capabilities in traffic management and security enhancement.
- The task emphasizes the importance of precise rule definition in iptables for desired network control and security policies.



Task 2c: Protecting Internal Servers

Task 2C focused on creating complex firewall rules using iptables to regulate access between external and internal hosts, emphasizing controlled access to specific internal servers and managing network traffic flow.

Key iptables Commands and Their Functions:

1. Blocking External Access to All Internal Servers Except 192.168.60.5 for Telnet:
   - Commands:
     - iptables -A FORWARD -i eth1 -o eth0 -p tcp -d 192.168.60.5 --dport 23 -j ACCEPT
     - iptables -A FORWARD -i eth1 -o eth0 -p tcp --dport 23 -j DROP
   - Explanation: The first rule allows external hosts to access the Telnet server on internal host 192.168.60.5, while the second rule blocks external access to Telnet servers on other internal hosts, maintaining strict access control except for 192.168.60.5.

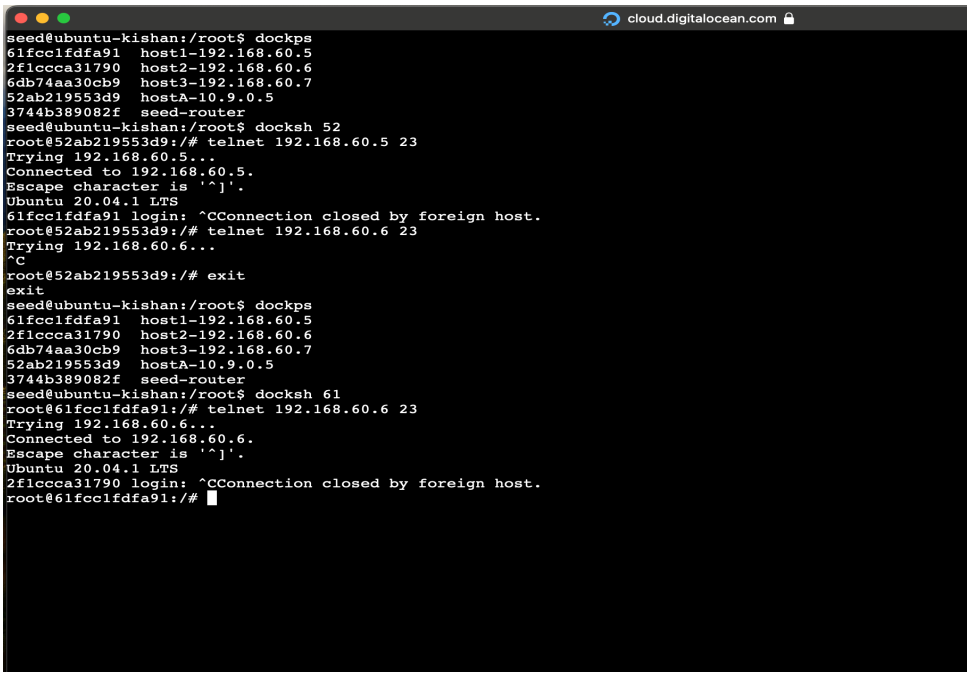
2. Blocking External Access to Other Internal Servers:

- Command: iptables -A FORWARD -i eth1 -o eth0 -d 192.168.60.0/24 -j DROP
- Explanation: This rule denies external hosts from accessing any services on the internal network, effectively isolating the internal network from external access to enhance security.

3. Allowing Internal Hosts to Access All Internal Servers:
- Command: iptables -A FORWARD -i eth1 -o eth0 -s 192.168.60.0/24 -j ACCEPT
- Explanation: Permits traffic originating from the internal network, enabling internal hosts to communicate freely within the network, ensuring seamless connectivity.

4. Blocking Internal Hosts from Accessing External Servers:
- Command: iptables -A FORWARD -i eth1 -o eth2 -s 192.168.60.0/24 -j DROP
- Explanation: Blocks outgoing connections from the internal network to the external network, preventing internal hosts from accessing external servers, crucial for preventing unauthorized communications.

Observations:
- Implementation of these rules demonstrated the capability to create sophisticated firewall configurations meeting specific security needs.
- Careful crafting of rules balanced security requirements with access necessities, allowing or blocking connections based on source, destination, and service type.
- Practical application of iptables showcased in network segmentation and traffic control between network zones.
- Testing these rules would confirm their effectiveness in achieving desired network isolation and access control, highlighting the importance of understanding network protocols and iptables syntax for effective firewall policies.
Screenshot:



Task 3: Connection Tracking and Stateful Firewall

Task 3a: Experiment with the Connection Tracking

Task 3A focused on exploring the connection tracking mechanism in the Linux kernel to support stateful firewalls. It aimed to provide hands-on experience and understanding of how connection tracking works, particularly for protocols like ICMP and UDP that don't inherently manage connections like TCP does.

Here's a rewritten version of the task description:

Task 3A delved into the connection tracking mechanism within the Linux kernel, crucial for supporting stateful firewalls. The objective was to give students practical experience and insight into how connection tracking operates, especially for protocols such as ICMP and UDP, which lack inherent connection management like TCP.

Procedure and Observations:

ICMP Experiment:

The task involved sending ICMP packets from one container (10.9.0.5) to another (192.168.60.5) using the ping command. Monitoring the connection tracking information on the router using the **conntrack -L** command was part of the experiment.

Observation: This experiment aimed to illustrate how the conntrack mechanism tracks ICMP packets, commonly used for diagnostic or control purposes. Key observations included understanding the duration of ICMP connection states maintained in the conntrack table.

UDP Experiment:

Setting up a netcat UDP server on one internal host (192.168.60.5) and sending UDP packets from an external host (10.9.0.5) to this server was the focus.The command nc -lu 9090 was used to start the UDP server, and nc -u 192.168.60.5 9090 was used to send UDP packets.
Observation: This experiment aimed to observe connection tracking behavior for UDP, a connectionless protocol. It demonstrated how the conntrack mechanism handles transient UDP interactions and the duration of UDP connection states in the tracking table.

TCP Experiment:

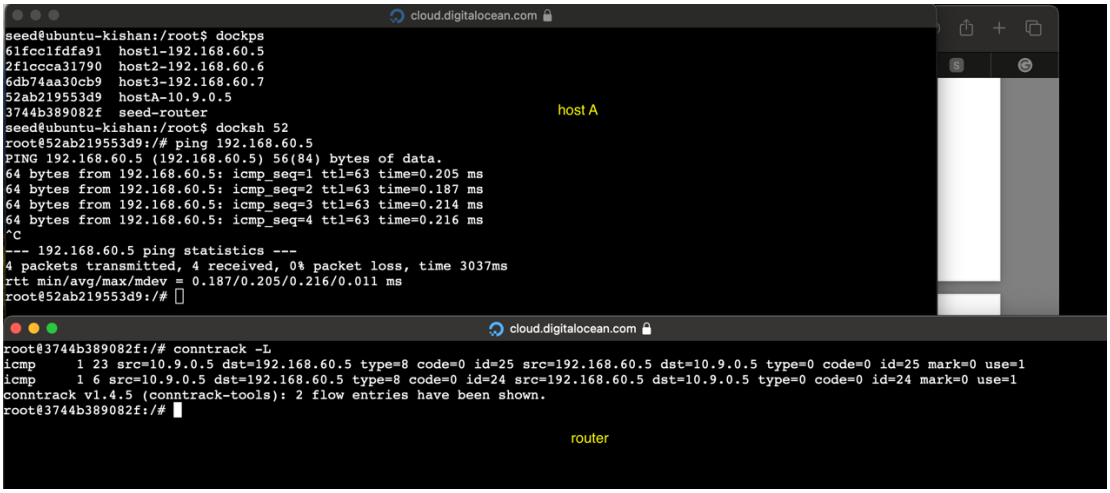
Similar to UDP, the TCP experiment involved setting up a TCP server and client communicating over TCP to observe TCP state tracking.
Observation: This experiment highlighted the contrast in connection tracking between TCP (a connection-oriented protocol) and UDP/ICMP. It focused on studying the duration of TCP connection states in the conntrack table.

Summary:

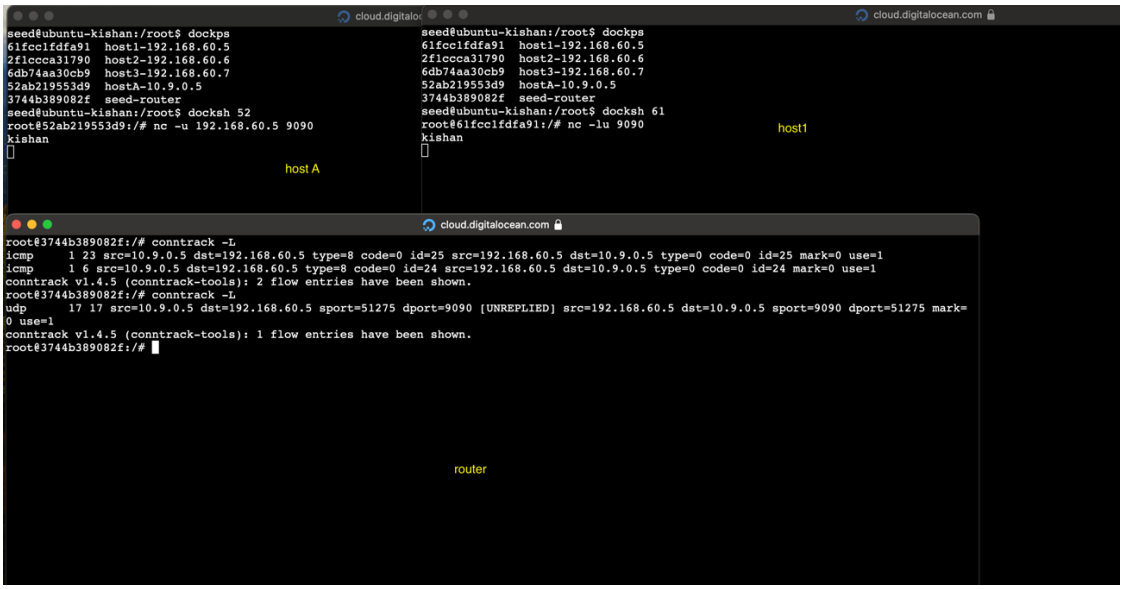
These experiments provided practical insights into how the conntrack mechanism operates for different protocol types.Students gained understanding of how stateful firewalls track connections and sessions, essential for implementing secure and context-aware firewall rules. The task emphasized the differences in handling state information between connection-oriented (TCP) and connectionless (UDP and ICMP) protocols.This foundational knowledge is crucial for setting up advanced firewall configurations based on connection states, enabling the management of established connections while blocking unauthorized ones.

Screenshots:

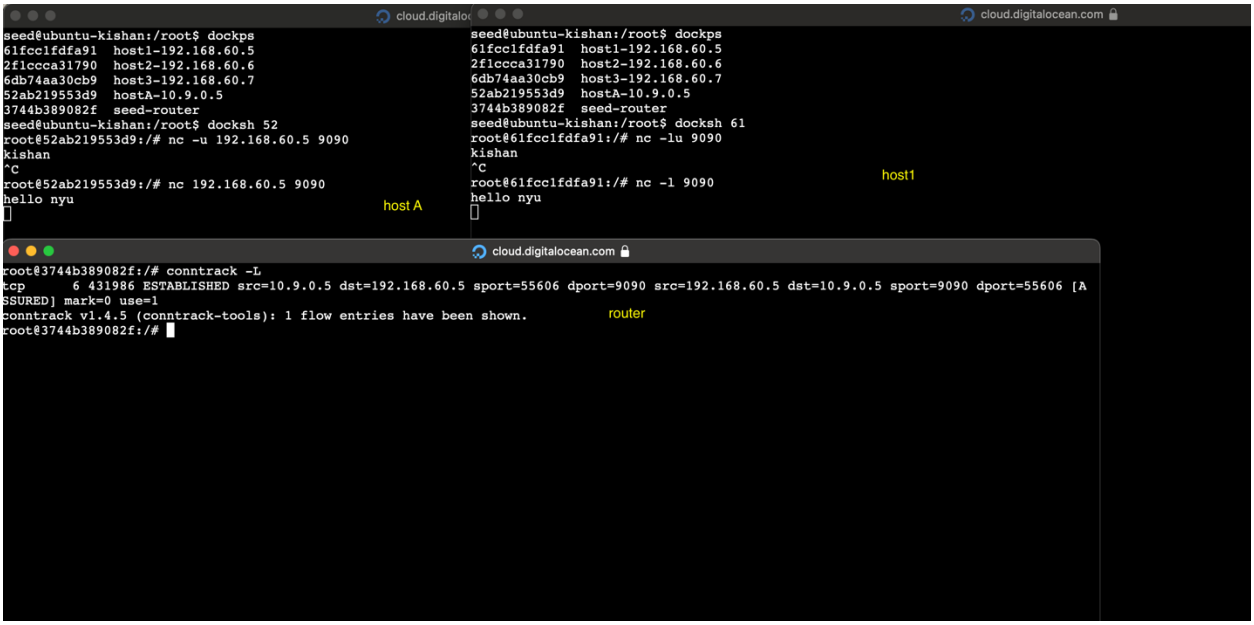
ICMP:



UDP:



TCP:



Task 3b: Setting Up a Stateful Firewall

Task 3B aimed to demonstrate the effectiveness of utilizing connection states in iptables to create a stateful firewall. By leveraging the conntrack module, the task illustrated how to build more intelligent and context-aware firewall rules. The comparison between stateful and stateless approaches underscored the trade-offs between complexity, resource usage, and security. The task was a practical application of advanced firewall concepts, providing insights into the nuances of network traffic management and security.

1. Allow Established and Related Connections:
   Command: iptables -A FORWARD -p tcp -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
   - Explanation: This rule permits TCP packets that are part of an already established connection or related to such a connection to pass through the FORWARD chain, ensuring ongoing and legitimate connections are not interrupted by the firewall.

2. Allow Incoming SYN Packets for New Connections:
   - Command: iptables -A FORWARD -p tcp -i eth0 --dport 8080 --syn -m conntrack --ctstate NEW -j ACCEPT

- Explanation: This rule allows incoming SYN packets (which initiate TCP connections) on port 8080 from the external interface (eth0), crucial for establishing new connections as SYN packets are the first step in the TCP three-way handshake.

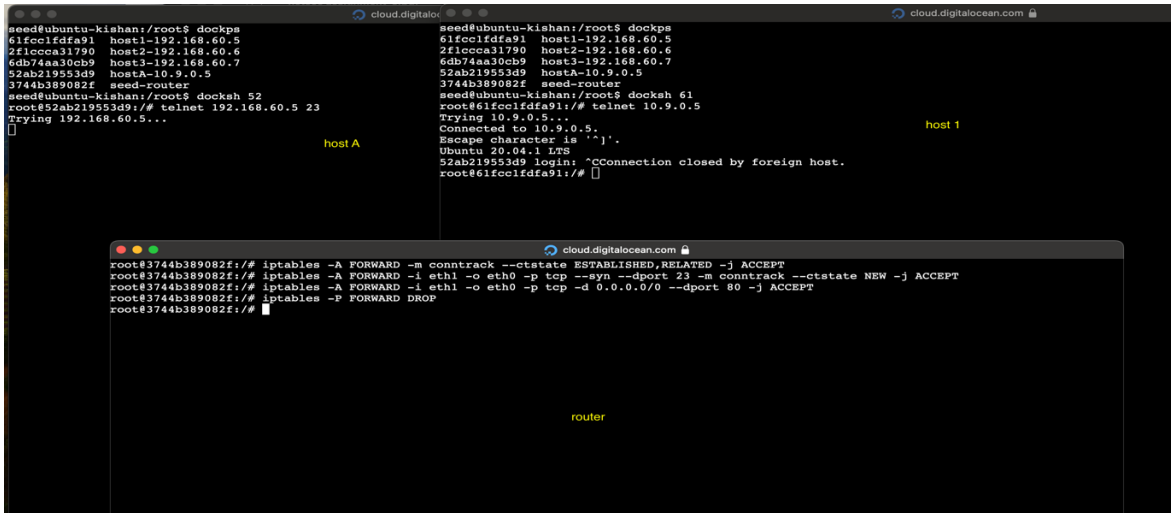
3. Set Default Policy to Drop Everything Else:
- Command: iptables -P FORWARD DROP

Explanation: Setting the default policy for the FORWARD chain to DROP ensures that any packets not explicitly accepted by the above rules are denied, adding an additional layer of security.

4. Comparison with Stateless Firewall Rules:

Advantage of Stateful Rules: Stateful rules simplify managing and securing network traffic by allowing the firewall to make decisions based on connection context, providing better security by only permitting established and related connections, reducing the risk of unauthorized access. We can use syn and other details in the command to filter out the packet.
Disadvantage: Stateful tracking requires more system resources as it involves maintaining a table of all current connections and their states.
Stateless Rules: The alternative, stateless rules, might be more resource-efficient but would require more complex rule definitions to mimic stateful behavior, potentially making them less secure if not configured correctly.

Summary:
Task 3B demonstrated the effectiveness of using connection states in iptables for creating a stateful firewall. It provided insights into the nuances of network traffic management and security, highlighting the trade-offs between complexity, resource usage, and security when choosing between stateful and stateless firewall rules.



Task 4 Limiting Network Traffic

Task 4 in the Firewall Exploration Lab aimed to implement rate limiting for network traffic using the limit module in iptables. The primary objective was to control the number of packets allowed to pass through the firewall from a specific source, demonstrating iptables' versatility in managing network traffic.

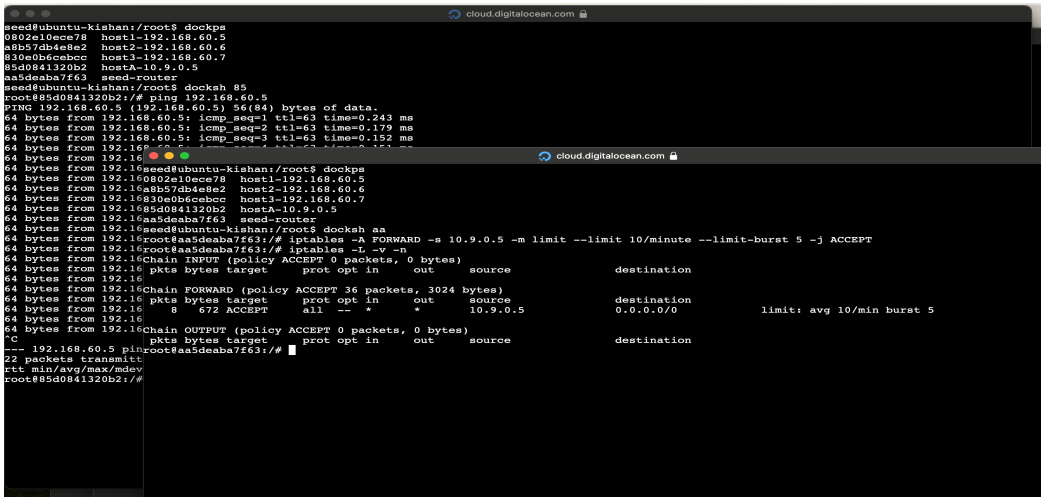
The setup involved a network configuration with internal and external hosts, where the router (seed-router) functioned as the firewall. The internal host with IP address 10.9.0.5 was designated to generate network traffic for the experiment.

Procedure:
1. Configuring iptables for Rate Limiting:
  - Commands executed on the router:
    iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT
    iptables -A FORWARD -s 10.9.0.5 -j DROP

  - Explanation: These commands were designed to first accept packets from the source IP 10.9.0.5 but with a rate limit of 10 packets per minute and a burst limit of 5 packets. The second command was set to drop packets from this source once the limit was reached.

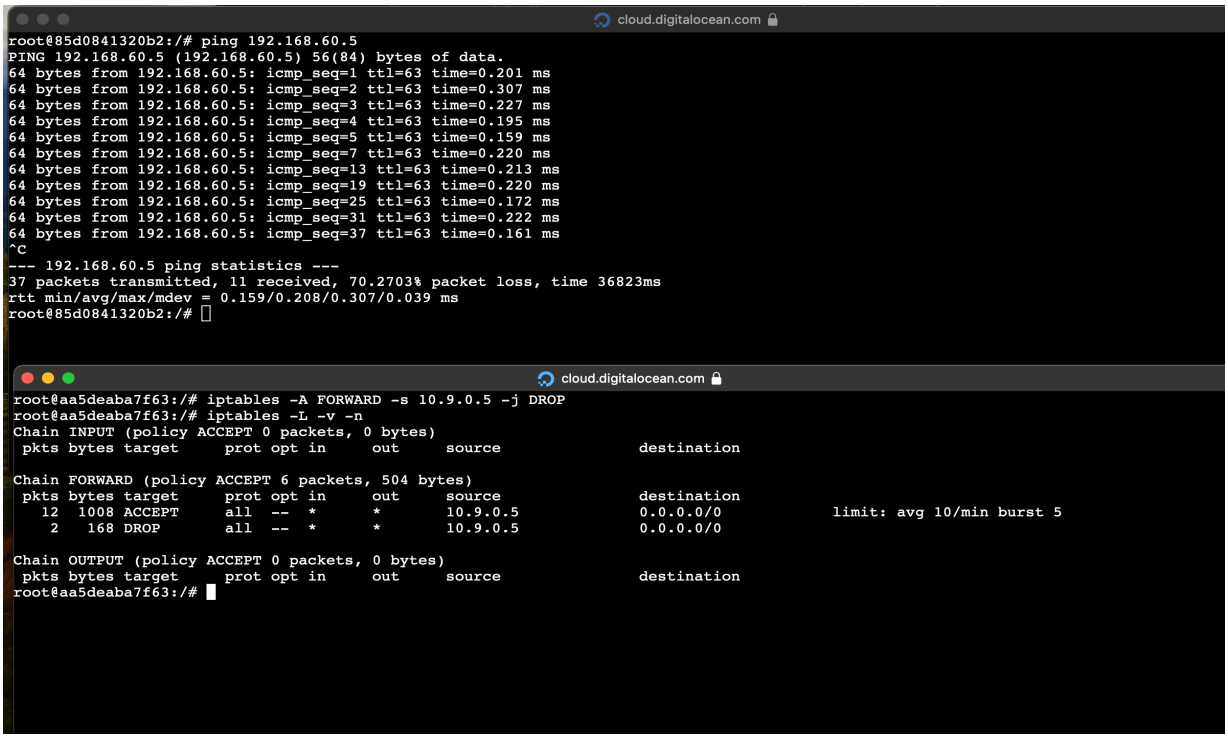
2. Initiating Network Traffic:
Traffic was generated by initiating ping requests from the internal host 10.9.0.5 to another internal host 192.168.60.5.

Observations and Analysis:
- Without Rate Limiting:
  - Initially, all ICMP echo requests (ping packets) from 10.9.0.5 were accepted, and responses were received, resulting in no packet loss.



- With Rate Limiting:
  - After implementing the rate limiting rule, it was observed that ping packets sent from 10.9.0.5 were dropped by the firewall once the rate limit was exceeded, leading to packet loss.
- Rule Analysis:
  - The first iptables rule effectively allowed incoming packets from 10.9.0.5 but only up to the specified rate limit and burst limit.
  - The second rule was responsible for dropping packets from 10.9.0.5 that exceeded the rate limit, ensuring the enforcement of the limit.
- Necessity of the Second Rule:
  - The second rule was crucial in this setup for enforcing the rate limit. Without this rule, packets exceeding the rate limit would not be explicitly dropped, potentially allowing excess traffic to pass through.

Conclusion:
The experiment successfully demonstrated the use of iptables' limit module to control the rate of incoming packets from a specific source. This task highlighted iptables' versatility in managing network traffic, showcasing its ability not only to block traffic but also to manage and shape it according to predefined limits. The experiment provided practical insights into how rate limiting can be used as an effective tool for network traffic management and security.



Task 5: Load Balancing

Task 5 in the Firewall Exploration Lab focused on implementing load balancing for UDP traffic across three internal servers. The task utilized the iptables' static module to demonstrate two load balancing techniques: round-robin (nth mode) and random distribution.

Objective:
The objective of Task 5 was to implement load balancing for UDP traffic across multiple internal servers using iptables' static module. The task aimed to showcase two load balancing techniques: round-robin for deterministic traffic distribution and random distribution for dynamic load balancing.

Experimental Setup:
- Servers: Three internal hosts (192.168.60.5, 192.168.60.6, and 192.168.60.7) were set up with UDP servers listening on port 8080.
- Router Configuration: The seed-router was configured with iptables rules to distribute incoming UDP traffic to these servers.

Procedure and iptables Commands:
1. Using nth Mode for Round-Robin Load Balancing:
  - This method implements a round-robin policy, distributing packets evenly among the servers.
  - Rules implemented on the router:

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

iptables -t nat -A PREROUTING -p udp --dport 8080 \
-m statistic --mode nth --every 3 --packet 1 \
-j DNAT --to-destination 192.168.60.6:8080

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

2. Using Random Mode for Load Balancing:
  - This approach randomly distributes packets among the servers based on specified probabilities.
  - Rules implemented on the router:

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

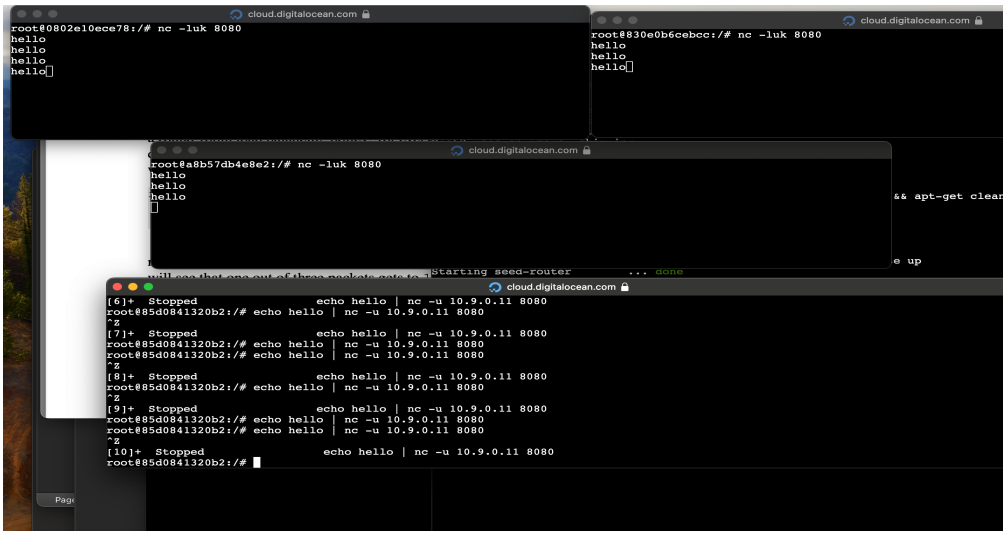
**iptables -t nat -A PREROUTING -p udp --dport 8080 **
**-m statistic --mode random --probability 0.33 **
-j DNAT --to-destination 192.168.60.6:8080

**iptables -t nat -A PREROUTING -p udp --dport 8080 **
**-m statistic --mode random --probability 0.34 **
-j DNAT --to-destination 192.168.60.7:8080

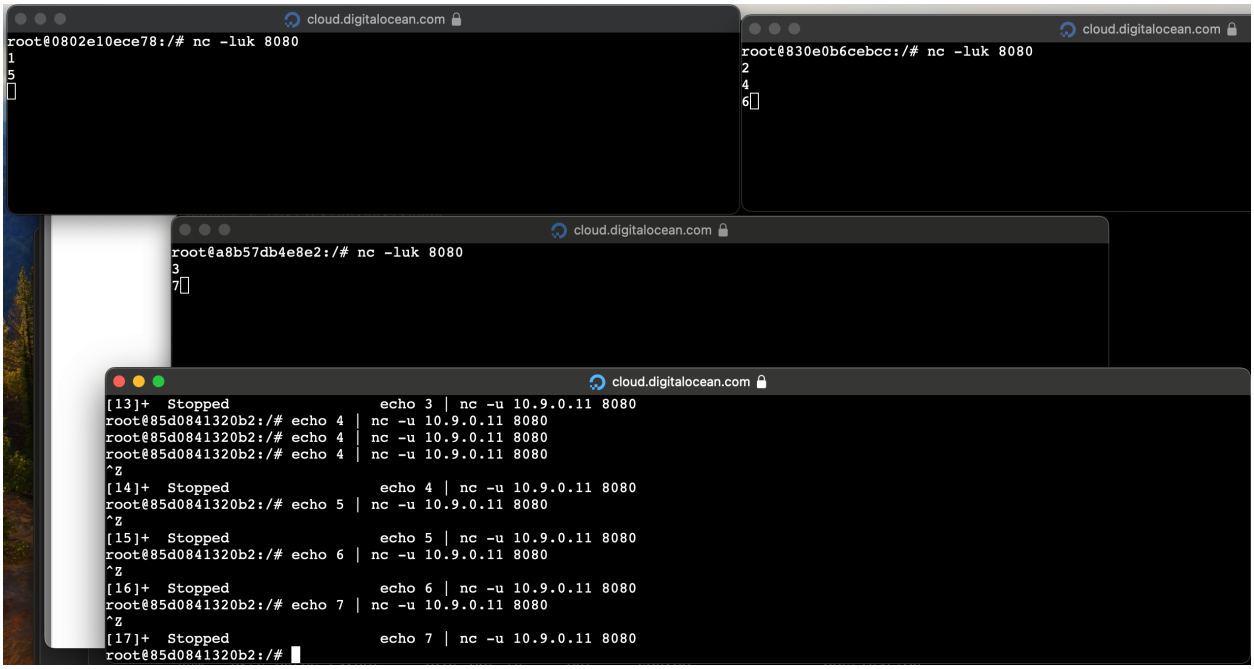
Observations and Analysis:
- nth Mode:
  - The round-robin approach ensured a deterministic and equal distribution of traffic, with each server receiving one out of every three packets.
  - This mode is beneficial in scenarios where a predictable load distribution is critical.
- Random Mode:
  - The random distribution provided a more dynamic approach, where each packet had a certain probability of being directed to a particular server.
  - This method is useful in situations where a non-deterministic, evenly spread load is preferred.

Conclusion:
Task 5 effectively showcased iptables versatility in network traffic management, particularly in load balancing scenarios. The two techniques, round-robin and random distribution offered insights into different strategies for distributing network load among multiple servers. These iptables rules highlight the tool's capability in handling complex network tasks beyond basic firewall functionalities.

Screenshot (round robin):



Screenshot (random):



Write up:

Learning about iptables in the Firewall Exploration Lab was interesting. It went beyond just understanding how firewalls block packets. We delved into connection tracking, rate limiting, and even explored load balancing, which was really interesting.

The lab also taught me about stateful and stateless firewalls. Stateful ones are like smart cookies that remember past connections, making decisions based on that history. On the other hand, stateless firewalls need specific rules for everything, like telling them exactly what to allow or block. It was like learning the difference between having a super-intelligent assistant and having to micromanage every little task.

What really caught my attention were the different behaviors of protocols under firewall rules. TCP, UDP, and ICMP all have their quirks when it comes to how they play with firewalls. TCP, being all about connections, behaved differently from UDP and ICMP, which are more free-spirited. It was fascinating to see how firewall rules shaped their interactions.

I also learned that the order of rules in iptables is crucial. Getting the order right is like solving a puzzle to make sure the firewall behaves just the way we want it to.

Moreover, experimenting with round-robin and random load balancing provided an insightful perspective on traffic management. It was fascinating to see how iptables could be used to distribute network load in a controlled manner.

Of course, it wasn't all smooth sailing. Configuring iptables rules, especially for fancy stuff like connection tracking and rate limiting, was a bit like solving a complex puzzle. It required attention to detail and careful planning to avoid messing things up.

Resource management was another challenge. Making sure the firewall didn't hog all the resources while still keeping everything secure was amazing. Another fascinating aspect was the impact of rule order in iptables. The sequence of rules can completely alter the firewall's behavior, making it crucial to get the order right to achieve the desired security and functionality balance.

Testing and validating the rules was like double-checking your homework before submitting it. We had to simulate different scenarios to see if the rules held up under different conditions. It was like being a detective, uncovering any potential weaknesses in our security setup.

Overall, the lab provided a comprehensive view of network security and the critical role that firewalls play in safeguarding networks. It was a hands-on learning experience that boosted my confidence in managing and securing network infrastructures effectively.