

Firewall Exploration Lab

GITHUB REPOSITORY:

<https://github.com/matteodalgrande/Ethical-Hacking-UniPD-Challenges>

Task 1: Implement a simple firewall

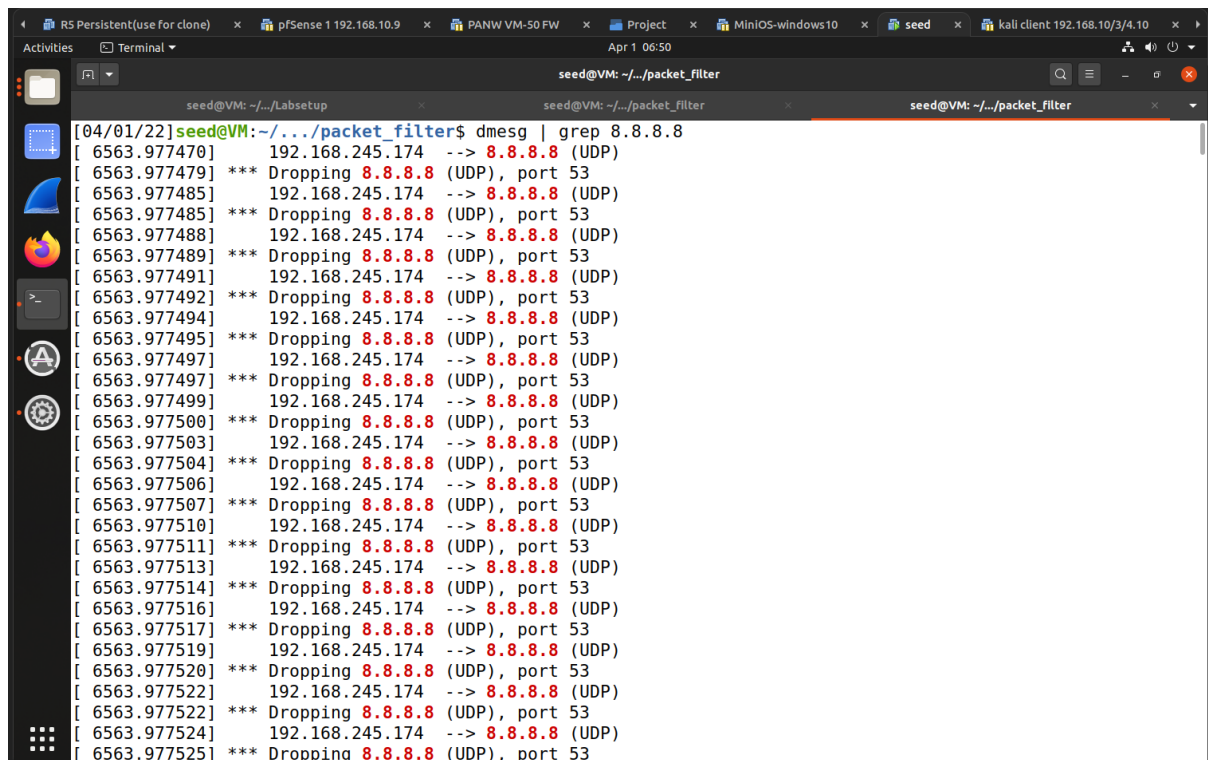
Task 1.B: Implement a Simple Firewall Using Netfilter

We ran the modules named seedFilter.ko

Then by changing the DNS of the virtual machine setting 8.8.8.8. To apply this change we ran the command `sudo service network-manager restart`.

After that we open a terminal and try to `dig google.com` url and we saw that the resolution of the web server was not successful.

By checking the kernel logs using `dmesg | grep 8.8.8.8` we can see how there has been a *Dropping 8.8.8.8 (UDP), port 53*.



```
seed@VM: ~/.../packet_filter$ dmesg | grep 8.8.8.8
[ 6563.977470] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977479] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977485] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977485] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977488] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977489] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977491] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977492] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977494] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977495] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977497] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977497] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977499] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977500] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977503] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977504] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977506] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977507] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977510] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977511] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977513] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977514] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977516] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977517] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977519] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977520] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977522] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977522] *** Dropping 8.8.8.8 (UDP), port 53
[ 6563.977524] 192.168.245.174 --> 8.8.8.8 (UDP)
[ 6563.977525] *** Dropping 8.8.8.8 (UDP), port 53
```

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

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.

3. Implement two more hooks to achieve the following:
 - a. Preventing other computers to ping the VM
 - b. Preventing other computers to telnet into the VM. Please implement two different hook functions, but register them to the same `netfilter` hook. You should decide what hook to use. Telnet's default port is TCP port 23. To test it, you can start the containers, go to 10.9.0.5, run the following commands (10.9.0.1 is the IP address assigned to the VM; for the sake of simplicity, you can hardcode this IP address in your firewall rules)

In order to complete this task is necessary to create a C file program in a way that can be compiled and loaded by the kernel.

The C file used is in

Labsetup/Files/04_preventing_other_ping_to_vm/preventing_ping.c

The most relevant part are:

1. create a function to register the hook
2. create a function to remove the hook
3. create two functions that are called **blockICMP** and **blockTELNET** that check the IP address and the destination port and the protocol in order to match the packets passing through the kernel. After a match we can do some stuff by dropping the packet.

The following script shows how to match a protocol and then look inside the packet checking the destination address and the destination port for TCP protocol(in the case of telnet connection).

```

85  if (iph->protocol == IPPROTO_TCP) {
86      tcp_h = tcp_hdr(skb);
87      if (iph->daddr == ip_addr && ntohs(tcp_h->dest) == port){
88          printk(KERN_WARNING "*** Dropping Telnet: %pI4 (TCP), port %d\n", &(iph->daddr), port);
89          return NF_DROP;
90      }
91  }
92  return NF_ACCEPT;

```

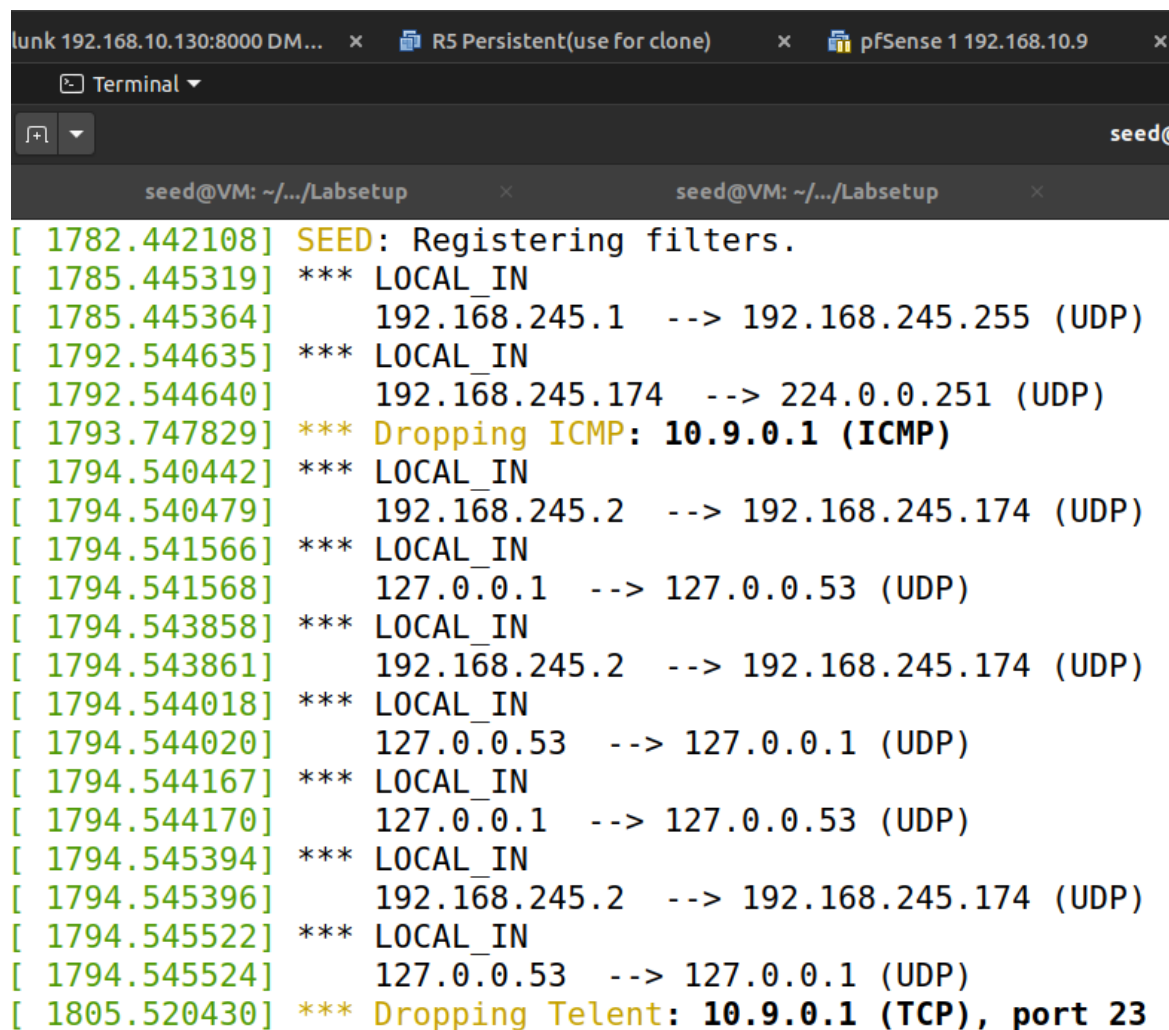
The following script shows how to match a protocol and then look inside the packet checking the destination address and the type of the icmp packets(in the case of icmp connection).

```

58  if (iph->protocol == IPPROTO_ICMP) {
59      icmp_h = icmp_hdr(skb);
60      if (iph->daddr == ip_addr && icmp_h->type == ICMP_ECHO){ //https://docs.huohoo.com/doxygen/linux/kernel/3.7/-uapi\_2linux\_2icmp\_8h\_source.html
61          printk(KERN_WARNING "*** Dropping ICMP: %pI4 (ICMP)", &(iph->daddr));
62          return NF_DROP;
63      }
64  }
65  return NF_ACCEPT;

```

The following script shows the log file of the kernel to verify that the script is running correctly.



```

[ 1782.442108] SEED: Registering filters.
[ 1785.445319] *** LOCAL_IN
[ 1785.445364] 192.168.245.1 --> 192.168.245.255 (UDP)
[ 1792.544635] *** LOCAL_IN
[ 1792.544640] 192.168.245.174 --> 224.0.0.251 (UDP)
[ 1793.747829] *** Dropping ICMP: 10.9.0.1 (ICMP)
[ 1794.540442] *** LOCAL_IN
[ 1794.540479] 192.168.245.2 --> 192.168.245.174 (UDP)
[ 1794.541566] *** LOCAL_IN
[ 1794.541568] 127.0.0.1 --> 127.0.0.53 (UDP)
[ 1794.543858] *** LOCAL_IN
[ 1794.543861] 192.168.245.2 --> 192.168.245.174 (UDP)
[ 1794.544018] *** LOCAL_IN
[ 1794.544020] 127.0.0.53 --> 127.0.0.1 (UDP)
[ 1794.544167] *** LOCAL_IN
[ 1794.544170] 127.0.0.1 --> 127.0.0.53 (UDP)
[ 1794.545394] *** LOCAL_IN
[ 1794.545396] 192.168.245.2 --> 192.168.245.174 (UDP)
[ 1794.545522] *** LOCAL_IN
[ 1794.545524] 127.0.0.53 --> 127.0.0.1 (UDP)
[ 1805.520430] *** Dropping Telnet: 10.9.0.1 (TCP), port 23

```

Task 2: Experimenting with Stateless Firewall Rules

Task 2.A: Protecting the Router

1. Can you ping the router? 2. Can you telnet into the router (a telnet server is running on all the containers; an account called **seed** was created on them with a password **dees**).

Before inserting the INPUT and OUTPUT policies as DROP into the filter table everything it's working fine and we can ping and telnet the router 10.9.0.11 from the host 10.9.0.5. After the insertion of the DROP rules and the rules that accept only echo-request and echo-reply, we can only ping the router from the host container and no more connect using telnet.

Task 2.B: Protecting the Internal Network

In order to perform this task it's necessary to create very simple rules to drop or accept packets passing through the router.

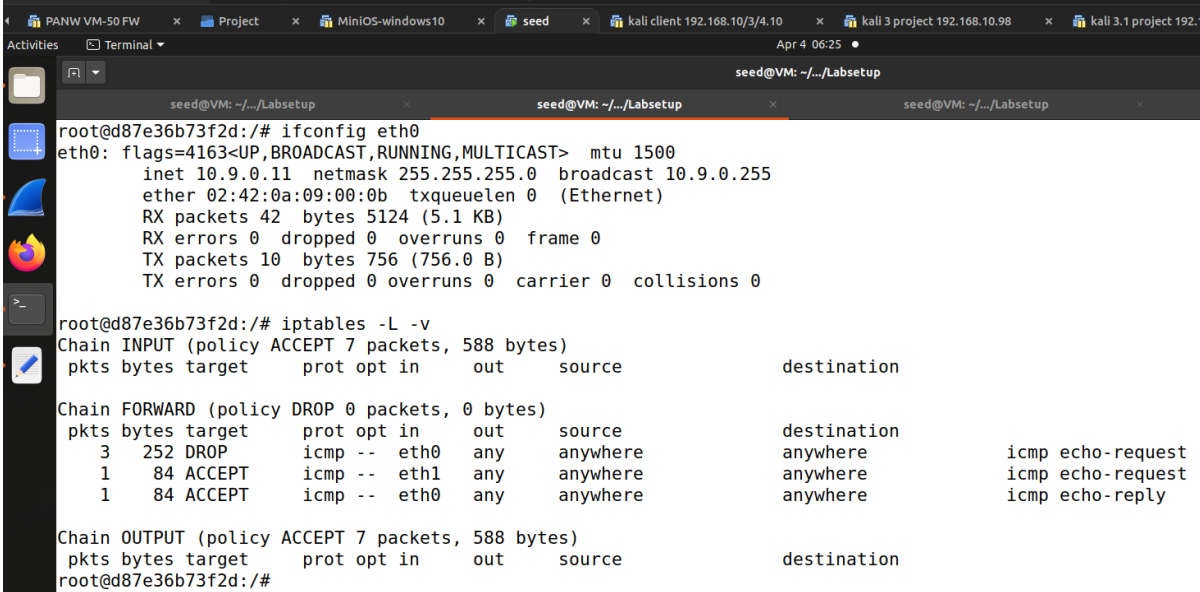
The commands to execute are the below.

```
iptables -t filter -A FORWARD -p icmp --icmp-type echo-request -i eth0 -j DROP &&
```

```
iptables -t filter -A FORWARD -p icmp --icmp-type echo-request -i eth1 -j ACCEPT &&
```

```
iptables -t filter -A FORWARD -p icmp --icmp-type echo-reply -i eth0 -j ACCEPT &&
```

```
iptables -P FORWARD DROP
```



```
root@d87e36b73f2d:/# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.11 netmask 255.255.255.0 broadcast 10.9.0.255
    ether 02:42:0a:09:00:0b txqueuelen 0 (Ethernet)
    RX packets 42 bytes 5124 (5.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 10 bytes 756 (756.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@d87e36b73f2d:/# iptables -L -v
Chain INPUT (policy ACCEPT 7 packets, 588 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
 3    252 DROP      icmp -- eth0    any     anywhere          anywhere          icmp echo-request
 1     84 ACCEPT    icmp -- eth1    any     anywhere          anywhere          icmp echo-request
 1     84 ACCEPT    icmp -- eth0    any     anywhere          anywhere          icmp echo-reply
Chain OUTPUT (policy ACCEPT 7 packets, 588 bytes)
 pkts bytes target    prot opt in     out     source            destination
root@d87e36b73f2d:/#
```

The above screenshot shows the rules inside the seed-router container.

Task 2.C: Protecting Internal Server

The commands that have to be executed inside the seed-router container are the following two.

```
iptables -t filter -A FORWARD -d 192.168.60.0/24 ! 192.168.60.5 -p tcp --sport 1024:65535 -dport 23 -i eth0 -j REJECT &&  
iptables -P FORWARD DROP
```

These two commands allow traffic only for the telnet connection on port 23 from external network to the host 192.168.60.5 and dropping all the others with destination host in the subnetwork 192.168.60.0/24.

Task 3: Connection Tracking and Stateful Firewall (Optional)

1. How long is the ICMP connection state be kept?

The ICMP connection is kept for 30 seconds alive.

2. How long is the UDP connection state be kept?

Also the UDP connection is kept for 30 seconds alive.

3. How long is the TCP connection state be kept?

TCP connection is kept for 431500 seconds.

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

Allow only ESTABLISHED or RELATED CONNECTION to pass:

connections are state related if a new connection is related to an already established connection. For example, the data FTP connection is related to the ftp session.

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

Accept SYN packets to start new connections.

```
iptables -A FORWARD -p tcp -i eth0 --dport 8080 --syn \
-m conntrack --ctstate NEW -j ACCEPT
```

The following command sets the default policy as DROP.

```
iptables -P FORWARD DROP
```

Rewrite the firewall rules in Task 2.C, but this time, we will add a rule allowing internal hosts to visit any external server, using connection tracking mechanism

```
iptables -t filter -A FORWARD -p tcp -i eth0 --syn -d 192.168.60.5 --dport 23 -m conntrack --ctstate NEW -j ACCEPT &&
iptables -t filter -A FORWARD -p tcp -i eth1 --syn -m conntrack --ctstate NEW -j ACCEPT &&
```

```
iptables -t filter -A FORWARD -p tcp -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT &&
iptables -t filter -A FORWARD -p tcp -j DROP &&
iptables -P FORWARD ACCEPT
```

To test these rules we can try to telnet the hosts and see that all the internal hosts can reach 10.9.0.5 and connect to the telnet server but 10.9.0.5 can reach only 192.168.60.5. In addition if we open a nc server in 10.9.0.5 we can reach it by each host inside the 192.168.60.0/24 network.

In the case without the state of the connection we have to keep track of each service looking and decide to accept or drop or packet inspecting the active TCP flag in each packet, in addition we have to keep the POLICY In ACCEPT mode, that it is not a good way to operate speaking in terms of security.

Task 4: Limiting Network Traffic (Optional)

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

Using the upper rules and looking at wireshark we can see that a lot of packets after 5 packets will be sent by 10.9.0.5 to 192.168.60.5 but there is not a reply due to the router has a limit of 10 packets per minute.

Without the second rule each packet reaches the host and there will be a reply(so pinging everything it's working fine).

Task 5: Load Balancing (Optional)

1. Using the `nth` mode (round-robin):

```
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 0 -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 0 -j DNAT
--to-destination 192.168.60.7:8080
```

Using the above commands it will be possible to distribute the load correctly among three servers exploiting iptables command.

```
#!/bin/bash
while [ true ]
do
echo script1 | nc -w 1 -u 10.9.0.11 8080
echo 'done'
done
```

Using the above script we can check that the iptables command are working fine

2. Using the random mode:

```
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.2 -j DNAT  
--to-destination 192.168.60.5:8080 &&  
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.4 -j DNAT  
--to-destination 192.168.60.6:8080 &&  
iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 1 -j DNAT  
--to-destination 192.168.60.7:8080
```

Using the upper commands it is possible to balance the load approximately equally among all the servers. It is important to set different probabilities and maintain the 1 probability to the last server in order to not lose the packet if we have a policy setted as drop by default. Instead if we have ACCEPT as default policy, our router manages the connection internally without losing any packet.

Using the following script inside the host 10.9.0.5 it will be possible to test the rules esely.

```
#!/bin/bash  
while [ true ]  
do  
echo script2 | nc -w 1 -u 10.9.0.11 8080  
echo 'done'  
done
```