Firewall Exploration Lab

Contents

Lab Setup

Please download the Labsetup.zip file from the below link to your VM, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment.

https://seedsecuritylabs.org/Labs_20.04/Files/Firewall/Labsetup.zip

In this lab, we need to use multiple machines. Their setup is depicted in Figure 1. We will use containers to set up this lab environment..

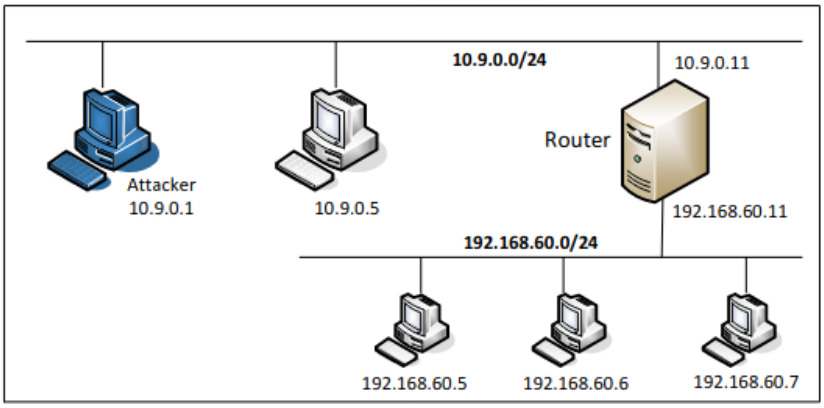

Figure 1: Lab Setup

Lab Overview

The learning objective of this lab is two-fold: learning how firewalls work, and setting up a simple firewall for a network. Students will first implement a simple stateless packet-filtering firewall, which inspects packets, and decides whether to drop or forward a packet based on firewall rules.

Through this implementation task, students can get basic ideas on how a firewall works. Actually, Linux already has a built-in firewall, also based on netfilter. This firewall is called iptables. Students will be given simple network topology, and are asked to use iptables to set up firewall rules to protect the network. Students will also be exposed to several other interesting applications of iptables.

This lab covers the following topics:

- Firewall
- Netfilter
- Loadable kernel module
- Using iptables to set up firewall rules
- Various applications of iptables

Task 1: Implementing a Simple Firewall

In this task, we will implement a simple packet filtering type of firewall, which inspects each incoming and outgoing packet, and enforces the firewall policies set by the administrator. Since the packet processing is done within the kernel, the filtering must also be done within the kernel. Therefore, it seems that implementing such a firewall requires us to modify the Linux kernel. In the past, this had to be done by modifying and rebuilding the kernel. The modern Linux operating systems provide several new mechanisms to facilitate the manipulation of packets without rebuilding the kernel image. These two mechanisms are Loadable Kernel Module (LKM) and Netfilter.

Notes about containers - Since all the containers share the same kernel, kernel modules are global. Therefore, if we set a kernel model from a container, it affects all the containers and the host. For this reason, it does not matter where you set the kernel module. We will just set the kernel module from the host VM in this lab**.**

Another thing to keep in mind is that containers' IP addresses are virtual. Packets going to these virtual IP addresses may not traverse the same path as what is described in the Netfilter document.

Therefore, in this task, to avoid confusion, we will try to avoid using those virtual addresses. We do most tasks on the host VM. The containers are mainly for other tasks.

Task 1.A: Implement a Simple Kernel Module

LKM allows us to add a new module to the kernel at runtime. This new module enables us to extend the functionalities of the kernel, without rebuilding the kernel or even rebooting the computer. The packet filtering part of a firewall can be implemented as an LKM. In this task, we will get familiar with LKM.
The following is a simple loadable kernel module. It prints out "Hello World!" when the module is loaded; when the module is removed from the kernel, it prints out "Bye-bye World!".

The messages are not printed out on the screen; they are actually printed into the /var/log/syslog file. You can use "dmesg" to view the messages.

hello. c - included in the lab setup files

```
#include <linux/module.h>
#include <linux/kernel.h>
int initialization(void)
{
    printk(KERN_INFO "Hello World!\n");
    return 0;
}
void cleanup(void)
{
    printk(KERN_INFO "Bye-bye World!.\n");
}
module_init(initialization);
module_exit(cleanup);
MODULE_LICENSE("GPL");
```

Note - Please use the VM's Terminal and not the containers given in the lab setup for this sub task.

On the VM - Open two Terminal Tabs, one to load the module and the other to view the messages.

- Use one terminal window to view the messages

Command:

- **$ sudo dmesg -k -w**

- The other to Load and Remove the Kernel

(**Change the directory to the kernel_module folder in 'Codes'** (given to the students) before executing the below)

Command:

- **$ make**
  $ sudo insmod hello.ko (inserting a module)
  $ lsmod | grep hello (list modules)
  $ sudo rmmod hello

Provide screenshots of your terminal with the required explanations of the same.

Task 1.B: Implement a Simple Firewall Using Netfilter

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

Netfilter is designed to facilitate the manipulation of packets by authorized users. It achieves this goal by implementing a number of hooks in the Linux kernel. These hooks are inserted into various places, including the packet's incoming and outgoing paths. If we want to manipulate the incoming packets, we simply need to connect our own programs (within LKM) to the corresponding hooks.

Once an incoming packet arrives, our program will be invoked. Our program can decide whether this packet should be blocked or not; moreover, we can also modify the packets in the program.

In this task, you need to use LKM and Netfilter to implement a packet filtering module. This module will fetch the firewall policies from a data structure, and use the policies to decide whether packets should be blocked or not. We would like students to focus on the filtering part, the core of firewalls, so students are allowed to hardcode firewall policies in the program.

Tasks

1. Compile the 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.

On the VM Terminal type the following command to make sure www.example.com is reachable, if it is not consider changing 8.8.8.8 to 8.8.4.4

Command:

$ **dig @8.8.8.8 www.example.com**

Note your observations with screenshots

Open a new terminal to view kernel messages and execute the following -

Command:

$ sudo dmesg -k -w

Now on the earlier Terminal Window execute the following to insert the kernel object.

Note - Please change the make file, 'uncomment' the respective names (seedFilter, seedPrint, seedBlock) based on the task

Uncomment - 'obj-m += seedFilter.o' and comment the other two.

Command:

$ make

$ sudo insmod seedFilter.ko

$ lsmod | grep seedFilter

After inserting execute the below and notice the difference -

Command:

$ **dig @8.8.8.8 www.example.com**

Take screenshots and explain your observations on both Terminal Windows.

Remove the module by executing -

$ sudo rmmod seedFilter

To clear the kernel messages execute -

$ dmesg -C

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 each of the hook functions be invoked.

   NF_INET_PRE_ROUTING
   NF_INET_LOCAL_IN
   NF_INET_FORWARD
   NF_INET_LOCAL_OUT
   NF_INET_POST_ROUTING

Similar to the previous task, open two terminal windows on your VM Host - one for viewing the kernel messages and the other for inserting the module.

On one window execute the following to view kernel messages -

Command:

$ sudo dmesg -k -w

Change the makefile as previously mentioned -

Uncomment - 'obj-m += seedPrint.o' and comment the other two.

Now on the other terminal window, insert the kernel module -

Command:

$ make

$ sudo insmod seedPrint.ko

$ lsmod | grep seedPrint

After inserting execute the below and notice the difference -

Command:

$ **dig @8.8.8.8 www.example.com**

Take screenshots and explain your observations on both Terminal Windows.

Remove the module by executing **-**

$ sudo rmmod seedPrint

3. Implement two more hooks to achieve the following:

   (1) preventing other computers to ping the VM, and

   (2) preventing other computers from telnetting 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):

   For this Task you need the docker container - Open the Host 10.9.0.5

Similar to the previous task,this time we need to open three terminal windows on your VM Host - one for viewing the kernel messages, one for inserting the module and the docker container that acts as an external machine.

On one window execute the following to view kernel messages -

Command:

$ sudo dmesg -k -w

Change the makefile as previously mentioned -

Uncomment - ‘obj-m += seedBlock.o’ and comment the other two.

Now on the other terminal window, insert the kernel module -

Command:

$ make

$ sudo insmod seedBlock.ko

$ lsmod | grep seedBlock

On the Host A - 10.9.0.5 terminal try -

Command:

 # ping 10.9.0.1

 # telnet 10.9.0.1

Take screenshots and explain your observations on allTerminal Windows.

Remove the module by executing **-**

$ sudo rmmod seedBlock

To clear the kernel messages execute -

$ dmesg -C

Task 2: Experimenting with Stateless Firewall Rules

In the previous task, we had a chance to build a simple firewall using netfilter. Actually, Linux already has a built-in firewall, also based on netfilter. This firewall is called iptables. Technically, the kernel part implementation of the firewall is called Xtables, while iptables is a user-space program to configure the firewall. However, iptables is often used to refer to both the kernel-part implementation and the user-space program.

Background of iptables

In this task, we will use iptables to set up a firewall. The iptables firewall is designed not only to filter packets, but also to make changes to packets. To help manage these firewall rules for different purposes, iptables organizes all rules using a hierarchical structure: table, chain, and rules. There are several tables, each specifying the main purpose of the rules as shown in Table 1. For example, rules for packet filtering should be placed in the filter table, while rules for making changes to packets should be placed in the nat or mangle tables.

Each table contains several chains, each of which corresponds to a netfilter hook. Basically, each chain indicates where its rules are enforced. For example, rules on the FORWARD chain are enforced at the NF INET FORWARD hook, and rules on the INPUT chain are enforced at the NF INET LOCAL IN hook. Each chain contains a set of firewall rules that will be enforced. When we set up firewalls, we add rules to these chains. For example, if we would like to block all incoming telnet traffic, we would add a rule to the INPUT chain of the filter table. If we would like to redirect all incoming telnet traffic to a different port on a different host, basically doing port forwarding, we can add a rule to the INPUT chain of the mangle table, as we need to make changes to packets

Task 2.A: Protecting the Router

Before proceeding ahead with this task, make sure to **open all the docker containers**.

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

On the **Router Container (seed-router)**, execute the following Iptables commands.

In order to view the current policies run the below command

Command:

 # iptables -t filter -L -n

Now please execute the following iptables command on the router container, and then try to access it from 10.9.0.5.

On seed-router run -

Command:

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

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

 # iptables -P OUTPUT DROP

 # iptables -P INPUT DROP

 # iptables -t filter -L -n

Now try to access (ping and telnet) the router from Host A - 10.9.0.5

Command:

 # ping seed-router

 # telnet seed-router

Questions :

(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).

Please report your observation with screenshots and explain the purpose for each rule

Cleanup

Before moving on to the next task, please restore the filter table to its original state by running the **following commands:**

 # iptables -F

 # iptables -P OUTPUT ACCEPT

 # iptables -P INPUT ACCEPT

Another way to restore the states of all the tables is to **restart the container.**

You can do it using the **following command (you need to find the container's ID first):**

$ docker restart <container ID>

Task 2.B: Protecting the Internal Network

In this task, we will set up firewall rules on the router to protect the internal network 192.168.60.0/24. We need to use the FORWARD chain for this purpose.

The directions of packets in the INPUT and OUTPUT chains are clear: packets are either coming into (for INPUT) or going out (for OUTPUT). This is not true for the FORWARD chain, because it is bi-directional: packets going into the internal network or going out to the external network all go through this chain. To specify the direction, we can add the interface options using "-i xyz" (coming in from the xyz interface) and/or "-o xyz" (going out from the xyz interface). The interfaces for the internal and external networks are different. You can find out the interface names via the "ip addr" command.

In this task, we want to implement a firewall to protect the internal network. More specifically, we need to enforce the following restrictions on the ICMP traffic:

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

Execute the following iptables commands on the **seed-router container** -

Commands:

 # iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-request -j DROP

 # iptables -A FORWARD -i eth1 -p icmp --icmp-type echo-request -j ACCEPT

 # iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-reply -j ACCEPT

 # iptables -P FORWARD DROP

 # iptables -L -n -v

Now we shall see if these **restrictions have been enforced** in the network.

1. Outside hosts cannot ping internal hosts.

On Host A - 10.9.0.5 execute

Command:

 # ping 192.168.60.5

2. Outside hosts can ping the router.

On Host A - 10.9.0.5 execute

Command:

 # ping seed-router

3. Internal hosts can ping Outside Hosts.

On host1-192.168.60.5 execute

Command:

 # ping 10.9.0.5

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

On host1-192.168.60.5 execute

Command:

 # telnet 10.9.0.5

Please report your observation with screenshots and explain the purpose for each rule

Cleanup

Before moving on to the next task, please restore the filter table to its original state by running the **following commands:**

On seed-router

 # iptables -F

 # iptables -P OUTPUT ACCEPT

 # iptables -P INPUT ACCEPT

Another way to restore the states of all the tables is to **restart the container.**

You can do it using the **following command (you need to find the container's ID first):**

$ docker restart <container ID>

Task 2.C: Protecting Internal Servers

In this task, we want to protect the TCP servers inside the internal network (192.168.60.0/24). More specifically, we would like to achieve the following objectives

1. All the internal hosts run a telnet server (listening to port 23). Outside hosts can only access the telnet server on 192.168.60.5, not the other internal hosts.
2. Outside hosts cannot access other internal servers.
3. Internal hosts can access all the internal servers.
4. Internal hosts cannot access external servers.

Execute the following iptables commands on the **seed-router container** -

Commands:

```
# iptables -A FORWARD -i eth0 -d 192.168.60.5 -p tcp --dport 23 -j ACCEPT
# iptables -A FORWARD -i eth1 -s 192.168.60.5 -p tcp --sport 23 -j ACCEPT
# iptables -P FORWARD DROP
# iptables -L -n -v
```

Now we shall see if these **restrictions have been enforced** in the network.

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

On host A - 10.9.0.5

Command:

```
# telnet 192.168.60.5
```

2. Outside hosts cannot access other internal servers.

On host A - 10.9.0.5

Command:

 # telnet 192.168.60.6

 # telnet 192.168.60.7

3. Internal hosts can access all the internal servers.

On host2 - 192.168.60.6

Command:

 # telnet 192.168.60.5

 # telnet 192.168.60.7

4. Internal hosts cannot access external servers.

On host2 - 192.168.60.6

Command:

 # telnet 10.9.0.5

Please report your observation with screenshots and explain the purpose for each rule

Cleanup

Before moving on to the next task, please restore the filter table to its original state by running the **following commands:**

On seed-router

 # iptables -F

 # iptables -P OUTPUT ACCEPT

 # iptables -P INPUT ACCEPT

Another way to restore the states of all the tables is to **restart the container.**

You can do it using the **following command (you need to find the container's ID first):**

$ docker restart <container ID>

Task 3: Connection Tracking and Stateful Firewall

In the previous task, we have only set up stateless firewalls, which inspect each packet independently. However, packets are usually not independent; they may be part of a TCP connection, or they may be ICMP packets triggered by other packets. Treating them independently does not take into consideration the context of the packets, and can thus lead to inaccurate, unsafe, or complicated firewall rules.

For example, if we would like to allow TCP packets to get into our network only if a connection was made first, we cannot achieve that easily using stateless packet filters, because when the firewall examines each individual TCP packet, it has no idea whether the packet belongs to an existing connection or not, unless the firewall maintains some state information for each connection. If it does that, it becomes a stateful firewall.

Task 3.A: Experiment with the Connection Tracking

To support stateful firewalls, we need to be able to track connections. This is achieved by the conntrack mechanism inside the kernel. In this task, we will conduct experiments related to this module, and get familiar with the connection tracking mechanism. In our experiment, we will check the connection tracking information on the router container.

This can be done using the following command:

```
# conntrack -L
```

The goal of the task is to use a series of experiments to help students understand the connection concept in this tracking mechanism, especially for the ICMP and UDP protocols, because unlike TCP, they do not have connections. Please conduct the following experiments. For each experiment, please describe your observation, along with your explanation.

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?

On host A - 10.9.0.5

Command:

 # ping 192.168.60.5

Let the ping run for sometime, then stop it (Ctrl + C)

Immediately move to the seed router and run

 # conntrack -L

Keep executing the above command, and you shall see the timer decreasing (time remaining for the connection)

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?

On host 1 - 192.168.60.5

Command:

 # nc -lu 9090

On host A - 10.9.0.5

Command:

 # nc -u 192.168.60.5 9090

<type something and hit return>

On seed router

 # conntrack -L

Execute the above command continuously and see if you spot the change in the timer.

Close the UDP connection and execute conntrack -L on the router container.

Take screenshots and explain your observations.

TCP experiment:

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

On host 1 - 192.168.60.5

Command:

 # nc -l 9090

On host A - 10.9.0.5

Command:

 # nc 192.168.60.5 9090

<type something and hit return>

On seed router

 # conntrack -L

Execute the above command continuously and see if you spot the change in the timer.

Close the TCP connection and execute conntrack -L on the router container.

Do you spot any difference?

Take screenshots and explain your observations.

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, which is a very important module for iptables; it tracks connections, and iptables replies on the tracking information to build stateful firewalls. The --ctstate ESTABLISHED,RELATED indicates whether a packet belongs to an ESTABLISHED or RELATED connection. The rule allows TCP packets belonging to an existing connection to pass through

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

On seed-router execute

Commands:

 # iptables -A FORWARD -p tcp -i eth0 -d 192.168.60.5 --dport 23 --syn -m conntrack --ctstate NEW -j ACCEPT

 # iptables -A FORWARD -i eth1 -p tcp --syn -m conntrack --ctstate NEW -j ACCEPT

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

 # iptables -A FORWARD -p tcp -j DROP

 # iptables -P FORWARD ACCEPT

 # iptables -L -n -v

Now we shall see if these **restrictions have been enforced** in the network.

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

On host A - 10.9.0.5

Command:

 # telnet 192.168.60.5

2. Outside hosts cannot access other internal servers.

On host A - 10.9.0.5

Command:

 # telnet 192.168.60.6

 # telnet 192.168.60.7

3. Internal hosts can access all the internal servers.

On host2 - 192.168.60.6

Command:

 # telnet 192.168.60.5

 # telnet 192.168.60.7

4. Internal hosts can access external servers.

On host2 - 192.168.60.6

Command:

 # telnet 10.9.0.5

Please report your observation with screenshots and explain the purpose for each rule

Highlight the differences between stateless and stateful firewalls based on your observation

Cleanup

Before moving on to the next task, please restore the filter table to its original state by running the **following commands:**

On seed-router

 # iptables -F

 # iptables -P OUTPUT ACCEPT

 # iptables -P INPUT ACCEPT

Another way to restore the states of all the tables is to **restart the container.**

You can do it using the **following command (you need to find the container's ID first):**

$ docker restart <container ID>

Task 4: Limiting Network Traffic

In addition to blocking packets, we can also limit the number of packets that can pass through the firewall. This can be done using the limit module of iptables. In this task, we will use this module to limit how many packets from 10.9.0.5 are allowed to get into the internal network. You can use "iptables -m limit -h" to see the manual

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

On seed router execute -

Command:

 # 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

 # iptables -L -n -v

On host A - 10.9.0.5

Command:

 # ping 192.168.60.5

Please report your observation with screenshots and explain the purpose for each rule

Clean the rules and now perform the same task without the second rule -

On seed router execute -

Command:

 # iptables -A FORWARD -s 10.9.0.5 -m limit --limit 10/minute --limit-burst 5 -j ACCEPT

 # iptables -L -n -v

On host A - 10.9.0.5

Command:

 # ping 192.168.60.5

Please report your observation with screenshots and explain the purpose for each rule

Clean the rules before proceeding to the next task

Task 5: Load Balancing

The iptables are very powerful. In addition to firewalls, it has many other applications. We will not be able to cover all its applications in this lab, but we will be experimenting with one of the applications, load balancing. In this task, we will use it to 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 (the -k option indicates that the server can receive UDP datagrams from multiple hosts):

```
# nc -luk 8080
```

We can use the statistic module to achieve load balancing.
There are two modes: random and nth. We will conduct experiments using both of them.

Using the nth mode (round-robin) - On the router container, we set the following rule, which applies to all the UDP packets going to port 8080. The nth mode of the statistic module is used; it implements a round-robin load balancing policy:

In this subtask we shall implement policies to equally divide the incoming packets between the three interval servers.

On the seed-router container:

Command:

```
# 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 2 --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 1 --packet 0 -j DNAT --to-destination 192.168.60.7:8080
#  iptables -L -n -v
```

On host1 - 192.168.60.5, host2 - 192.168.60.6 and host3 - 192.168.60.7 start the server

Command:

 # nc -luk 8080

On host A - 10.9.0.5

Command:

 # nc -u 10.9.0.11 8080

< enter 3 words, wait 30 seconds before entering the next word>

For example :

 # nc -u 10.9.0.11 8080

Hello 1

Wait 30 seconds

Hello 2

Wait 30 seconds

Hello 3

'Hello 1' appears in the host 1 terminal, 'Hello 2' appears in the host 2 terminal etc.

If you do not have a waiting period all the three statements will be considered as a single packet, so please wait for sometime before entering some text.

Students should enter their First Name, Last Name and SRN.

Clean the rules before proceeding to the next task

Using the random mode - Let's use a different mode to achieve load balancing. The following rule will select a matching packet with the probability P. You need to replace P with a probability number.

On the seed-router container:

Command:

 # iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 0.3333 -j DNAT --to-destination 192.168.60.5:8080

 # 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

 # iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode random --probability 1 -j DNAT --to-destination 192.168.60.6:8080

 # iptables -L -n -v

On host1 - 192.168.60.5, host2 - 192.168.60.6, host3 - 192.168.60.7 start the server

Command:

 # nc -luk 8080

On host A - 10.9.0.5

Command:

 # nc -u 10.9.0.11 8080

< enter 3 words, wait 30 seconds before entering the next word>

For example :

 # nc -u 10.9.0.11 8080

Hello 1

Wait 30 seconds

Hello 2

Wait 30 seconds

Hello 3

This time each server **may not get the same amount of traffic**.

Please provide some explanation for the rules with the appropriate screenshots.

Submission

You need to submit a detailed lab report to describe what you have done and what you have observed; you also need to provide explanations for the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation. Simply attaching code without any explanation will not receive credits.