*Lab 4*

*Computer Security 460*

*Basic Network Security (80 Pts + 30 Optional)*

*Sniffing & Spoofing (40)*

*Firewall Configuration (40)*

*Optional Firewall (30)*

For each Task: You must have a **dedicated section named Task [X] (**where X is the corresponding task) in your report that contains: 1) Relevant code/ configuration snippets, 2) a brief description of what the code or configuration does 3) Screenshots showing the relevant part clearly when applicable. For part two (firewall) iptables configuration is to be treated as a code . Submissions without the relevant code or expanations will receive partial credit

Finally, if you use AI tools, you still have to justify (and are responsible) for their output. There are two sections in this assignment:

1) Sniffing and Spoofing
2) Firewall Configuration
    a) Tasks 3.B,4 and 5 are OPTIONAL.

Both Labs are based on corresponding SEED labs but we only study part of them. You still need to download the corresponding arm or intel version for the lab setup as usual.

# PART 1 (40PTS)

# Packet Sniffing and Spoofing Lab



## 1 Overview

Packet sniffing and spoofing are two important concepts in network security; they are two major threats in network communication. Being able to understand these two threats is essential for understanding security measures in networking. There are many packet sniffing and spoofing tools, such as `Wireshark`, `Tcpdump`, `Netwox`, `Scapy`, etc. Some of these tools are widely used by security experts, as well as by attackers. Being able to use these tools is important for students, but what is more important for students in a network security course is to understand how these tools work, i.e., how packet sniffing and spoofing are implemented in software.

The objective of this lab is two-fold: learning to use the tools and understanding the technologies underlying these tools. For the second object, students will write simple sniffer and spoofing programs, and gain an in-depth understanding of the technical aspects of these programs. This lab covers the following topics:

- How the sniffing and spoofing work
- Packet sniffing using the `pcap` library and Scapy
- Packet spoofing using raw socket and Scapy
- Manipulating packets using Scapy

**Readings and Videos.** Detailed coverage of sniffing and spoofing can be found in the following:

- Chapter 15 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 2 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

**Lab environment.** This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

## Environment Setup using Container

In this lab, we will use three machines that are connected to the same LAN. We can either use three VMs or three containers. Figure 1 depicts the lab environment setup using containers. We will do all the attacks on the attacker container, while using the other containers as the user machines.

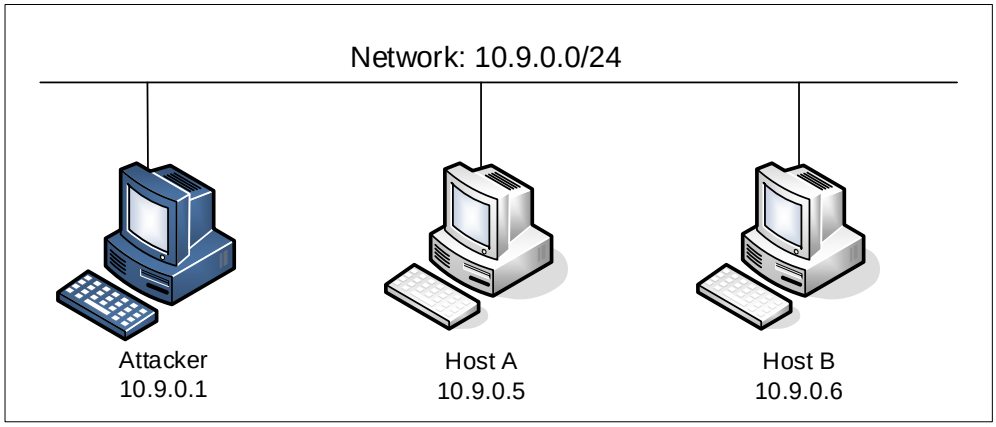



Figure 1: Lab environment setup

### a. Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash
```

```
// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

### b. About the Attacker Container

In this lab, we can either use the VM or the attacker container as the attacker machine. If you look at the Docker Compose file, you will see that the attacker container is configured differently from the other containers. Here are the differences:

- *Shared folder.* When we use the attacker container to launch attacks, we need to put the attacking code inside the attacker container. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker `volumes`. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the `./volumes` folder on the host machine (i.e., the VM) to the `/volumes` folder inside the container. We will write our code in the `./volumes` folder (on the VM), so they can be used inside the containers.

  ```
  volumes:
        - ./volumes:/volumes
  ```

- *Host mode.* In this lab, the attacker needs to be able to sniff packets, but running sniffer programs inside a container has problems, because a container is effectively attached to a virtual switch, so it can only see its own traffic, and it is never going to see the packets among other containers. To solve this problem, we use the `host` mode for the attacker container. This allows the attacker container to see all the traffics. The following entry used on the attacker container:

  ```
  network_mode: host
  ```

  When a container is in the `host` mode, it sees all the host's network interfaces, and it even has the same IP addresses as the host. Basically, it is put in the same network namespace as the host VM. However, the container is still a separate machine, because its other namespaces are still different from the host.

**Getting the network interface name.** When we use the provided Compose file to create containers for this lab, a new network is created to connect the VM and the containers. The IP prefix for this network is

`10.9.0.0/24`, which is specified in the `docker-compose.yml` file. The IP address assigned to our VM is `10.9.0.1`. We need to find the name of the corresponding network interface on our VM, because we need to use it in our programs. The interface name is the concatenation of `br-` and the ID of the network created by Docker. When we use `ifconfig` to list network interfaces, we will see quite a few. Look for the IP address `10.9.0.1`.

```
$ ifconfig
br-c93733e9f913: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        ...
```

Another way to get the interface name is to use the "`docker network`" command to find out the network ID ourselves (the name of the network is `seed-net`:

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
a82477ae4e6b        bridge              bridge              local
e99b370eb525        host                host                local
df62c6635eae        none                null                local
c93733e9f913        seed-net            bridge              local
```

## Lab Task Set 1: Using Scapy to Sniff and Spoof Packets

Many tools can be used to do sniffing and spoofing, but most of them only provide fixed functionalities. Scapy is different: it can be used not only as a tool, but also as a building block to construct other sniffing and spoofing tools, i.e., we can integrate the Scapy functionalities into our own program. In this set of tasks, we will use Scapy for each task.

To use Scapy, we can write a Python program, and then execute this program using Python. See the following example. We should run Python using the root privilege because the privilege is required for spoofing packets. At the beginning of the program (Line ①), we should import all Scapy's modules.

```
# view mycode.py
#!/usr/bin/env python3

from scapy.all import *    ①

a = IP()
a.show()

# python3 mycode.py
###[ IP ]###
  version   = 4
  ihl       = None
  ...


// Make mycode.py executable (another way to run python programs)
# chmod a+x mycode.py
# mycode.py
```

We can also get into the interactive mode of Python and then run our program one line at a time at the

Python prompt. This is more convenient if we need to change our code frequently in an experiment.

```
# python3
>>> from scapy.all import *
>>> a = IP()
>>> a.show()
###[ IP ]###
  version   = 4
  ihl       = None
  ...
```

### (20pts) Task 1.1: niffing Packets

Wireshark is the most popular sniffing tool, and it is easy to use. We will use it throughout the entire lab. However, it is difficult to use Wireshark as a building block to construct other tools. We will use Scapy for that purpose. The objective of this task is to learn how to use Scapy to do packet sniffing in Python programs. A sample code is provided in the following:

```
#!/usr/bin/env python3
from scapy.all import *

def print_pkt(pkt):
  pkt.show()

pkt = sniff(iface='br-c93733e9f913', filter='icmp', prn=print_pkt)
```

The code above will sniff the packets on the `br-c93733e9f913` interface. Please read the instruction in the lab setup section regarding how to get the interface name. If we want to sniff on multiple interfaces, we can put all the interfaces in a list, and assign it to `iface`. See the following example:

```
iface=['br-c93733e9f913', 'enp0s3']
```

**Task 1.1A.** In the above program, for each captured packet, the callback function `print_pkt()` will be invoked; this function will print out some of the information about the packet. Run the program with the root privilege and demonstrate that you can indeed capture packets. After that, run the program again, but without using the root privilege; describe and explain your observations.

```
// Make the program executable
# chmod a+x sniffer.py

// Run the program with the root privilege
# sniffer.py

// Switch to the "seed" account, and
// run the program without the root privilege
# su seed
$ sniffer.py
```

**Task 1.1B.** Usually, when we sniff packets, we are only interested certain types of packets. We can do that by setting filters in sniffing. Scapy's filter use the BPF (Berkeley Packet Filter) syntax; you can find the

BPF manual from the Internet. Please set the following filters and demonstrate your sniffer program again (each filter should be set separately):

- Capture only the ICMP packet
- Capture any TCP packet that comes from a particular IP and with a destination port number 23.
- Capture packets comes from or to go to a particular subnet. You can pick any subnet, such as `128.230.0.0/16`; you should not pick the subnet that your VM is attached to.

### (10 pts) Task 1.2: poofing CMP Packets

As a packet spoofing tool, Scapy allows us to set the fields of IP packets to arbitrary values. The objective of this task is to spoof IP packets with an arbitrary source IP address. We will spoof ICMP echo request packets, and send them to another VM on the same network. We will use Wireshark to observe whether our request will be accepted by the receiver. If it is accepted, an echo reply packet will be sent to the spoofed IP address. The following code shows an example of how to spoof an ICMP packets.

```
>>> from scapy.all import *
>>> a = IP()                ①
>>> a.dst = '10.0.2.3'      ②
>>> b = ICMP()              ③
>>> p = a/b                 ④
>>> send(p)                 ⑤
.
Sent 1 packets.
```

In the code above, Line ① creates an IP object from the IP class; a class attribute is defined for each IP header field. We can use `ls(a)` or `ls(IP)` to see all the attribute names/values. We can also use a.show() and IP.show() to do the same. Line ② shows how to set the destination IP address field. If a field is not set, a default value will be used.

```
>>> ls(a)
version    : BitField (4 bits)       = 4               (4)
ihl        : BitField (4 bits)       = None            (None)
tos        : XByteField              = 0               (0)
len        : ShortField              = None            (None)
id         : ShortField              = 1               (1)
flags      : FlagsField (3 bits)     = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField (13 bits)      = 0               (0)
ttl        : ByteField               = 64              (64)
proto      : ByteEnumField           = 0               (0)
chksum     : XShortField             = None            (None)
src        : SourceIPField           = '127.0.0.1'     (None)
dst        : DestIPField             = '127.0.0.1'     (None)
options    : PacketListField         = []              ([])
```

Line ③ creates an ICMP object. The default type is echo request. In Line ④, we stack `a` and `b` together to form a new object. The `/` operator is overloaded by the IP class, so it no longer represents division; instead, it means adding `b` as the payload field of `a` and modifying the fields of `a` accordingly. As a result, we get a new object that represent an ICMP packet. We can now send out this packet using `send()` in Line ⑤. Please make any necessary change to the sample code, and then demonstrate that you can spoof an ICMP echo request packet with an arbitrary source IP address.

### (10 pts) Task 1.3: Traceroute

The objective of this task is to use Scapy to estimate the distance, in terms of number of routers, between your VM and a selected destination. This is basically what is implemented by the `traceroute` tool. In this task, we will write our own tool. The idea is quite straightforward: just send an packet (any type) to the destination, with its Time-To-Live (TTL) field set to 1 first. This packet will be dropped by the first router, which will send us an ICMP error message, telling us that the time-to-live has exceeded. That is how we get the IP address of the first router. We then increase our TTL field to 2, send out another packet, and get the IP address of the second router. We will repeat this procedure until our packet finally reach the destination. It should be noted that this experiment only gets an estimated result, because in theory, not all these packets take the same route (but in practice, they may within a short period of time). The code in the following shows one round in the procedure.

```
a = IP()
a.dst = '1.2.3.4'
a.ttl = 3
b = ICMP()
send(a/b)
```

If you are an experienced Python programmer, you can write your tool to perform the entire procedure automatically. If you are new to Python programming, you can do it by manually changing the TTL field in each round, and record the IP address based on your observation from Wireshark. Either way is acceptable, as long as you get the result.

Part 2 (70 pts: 40 Pts + 30 pts the OPTIONAL part)

# Firewall Exploration Lab



## 1 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 the basic ideas on how firewall works.

Actually, Linux already has a built-in firewall, also based on `netfilter`. This firewall is called `iptables`. Students will be given a 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`

**Readings and videos.** Detailed coverage of firewalls can be found in the following:

- Chapter 17 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 9 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

**Lab environment.** This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

## 2 Environment Setup Using Containers

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.

### 2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your VM from the lab's website, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. Detailed explanation of the content in this file and all the involved `Dockerfile` can be found from the user manual, which is linked

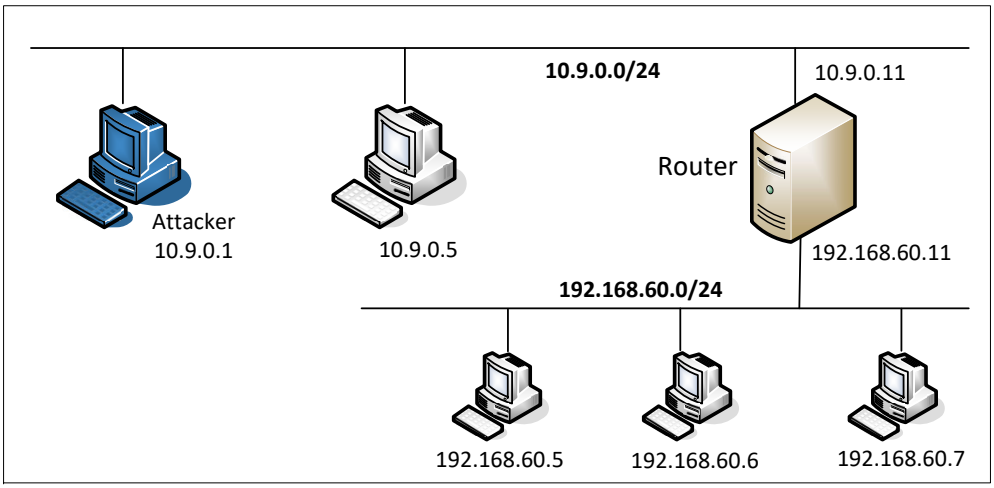



Figure 1: Lab setup

to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container images
$ docker-compose up     # Start the containers
$ docker-compose down   # Shut down the containers

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "`docker ps`" command to find out the ID of the container, and then use "`docker exec`" to start a shell on that container. We have created aliases for them in the `.bashrc` file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
```

# 4 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.

## 4.1 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.

## 4.2 Using `iptables`

To add rules to the chains in each table, we use the `iptables` command, which is a quite powerful command. Students can find the manual of `iptables` by typing `"man iptables"` or easily find many tutorials from online. What makes `iptables` complicated is the many command-line arguments that we

Table 1: `iptables` Tables and Chains

| Table | Chain | Functionality |
|---|---|---|
| filter | `INPUT`<br>`FORWARD`<br>`OUTPUT` | Packet filtering |
| nat | `PREROUTING`<br>`INPUT`<br>`OUTPUT`<br>`POSTROUTING` | Modifying source or destination network addresses |
| mangle | `PREROUTING`<br>`INPUT`<br>`FORWARD`<br>`OUTPUT`<br>`POSTROUTING` | Packet content modification |

need to provide when using the command. However, if we understand the structure of these command-line arguments, we will find out that the command is not that complicated.

In a typical `iptables` command, we add a rule to or remove a rule from one of the chains in one of the tables, so we need to specify a table name (the default is `filter`), a chain name, and an operation on the chain. After that, we specify the rule, which is basically a pattern that will be matched with each of the packets passing through. If there is a match, an action will be performed on this packet. The general structure of the command is depicted in the following:

```
iptables -t <table> -<operation> <chain>  <rule>   -j <target>
         ---------- --------------------  -------  -----------
            Table          Chain           Rule      Action
```

The rule is the most complicated part of the `iptables` command. We will provide additional information later when we use specific rules. In the following, we list some commonly used commands:

```
// List all the rules in a table (without line number)
iptables -t nat -L -n

// List all the rules in a table (with line number)
iptables -t filter -L -n --line-numbers


// Delete rule No. 2 in the INPUT chain of the filter table
iptables -t filter -D INPUT 2

// Drop all the incoming packets that satisfy the <rule>
iptables -t filter -A INPUT <rule>  -j DROP
```

**Note.** Docker relies on `iptables` to manage the networks it creates, so it adds many rules to the `nat` table. When we manipulate `iptables` rules, we should be careful not to remove Docker rules. For example, it will be quite dangerous to run the "`iptables -t nat -F`" command, because it removes all the rules in the `nat` table, including many of the Docker rules. That will cause trouble to Docker containers. Doing this for the `filter` table is fine, because Docker does not touch this table.

### (10 pts) Task 2.A: Protecting the Router

In this task, we will set up rules to prevent outside machines from accessing the router machine, except ping. Please execute the following `iptables` command on the router container, and then try to access it from `10.9.0.5`. (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 and explain the purpose for each rule.

```
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     ← Set default rule for OUTPUT
iptables -P INPUT  DROP     ← Set default rule for INPUT
```

**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>
```

### (10 pts) 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.

You will need to use the `"-p icmp"` options to specify the match options related to the ICMP protocol. You can run `"iptables -p icmp -h"` to find out all the ICMP match options. The following example drops the ICMP echo request.

```
iptables -A FORWARD -p icmp --icmp-type echo-request -j DROP
```

In your lab report, please include your rules and screenshots to demonstrate that your firewall works as expected. When you are done with this task, please remember to clean the table or restart the container before moving on to the next task.

### (10 pts) 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.
5. In this task, the connection tracking mechanism is not allowed. It will be used in a later task.

You will need to use the "`-p tcp`" options to specify the match options related to the TCP protocol. You can run "`iptables -p tcp -h`" to find out all the TCP match options. The following example allows the TCP packets coming from the interface `eth0` if their source port is `5000`.

```
iptables -A FORWARD -i eth0 -p tcp --sport 5000  -j ACCEPT
```

When you are done with this task, please remember to clean the table or restart the container before moving on to the next task.

## 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.

### (10 pts) 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 is the ICMP connection state be kept?

```
// On 10.9.0.5, send out ICMP packets
# ping 192.168.60.5
```

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

```
// On 192.168.60.5, start a netcat UDP server
# nc -lu 9090

// On 10.9.0.5, send out UDP packets
# nc -u 192.168.60.5 9090
<type something, then hit return>
```

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

```
// On 192.168.60.5, start a netcat TCP server
# nc -l 9090

// On 10.9.0.5, send out TCP packets
# nc 192.168.60.5 9090
<type something, then hit return>
```

### (Optional 10pts) 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 `--ctsate ESTABLISHED,RELATED` indicates that whether a packet belongs to an `ESTABLISHED` or `RELATED` connection. The rule allows TCP packets belonging to an existing connection to pass through.

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

The rule above does not cover the SYN packets, which do not belong to any established connection. Without it, we will not be able to create a connection in the first place. Therefore, we need to add a rule to accept incoming SYN packet:

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

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

```
iptables -P FORWARD DROP
```

Please 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). After you write the rules using the connection

tracking mechanism, think about how to do it without using the connection tracking mechanism (you do not need to actually implement them). Based on these two sets of rules, compare these two different approaches, and explain the advantage and disadvantage of each approach. When you are done with this task, remember to clear all the rules.

## (Optional 10 pts) 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.

```
$ iptables -m limit -h
limit match options:
--limit avg             max average match rate: default 3/hour
                        [Packets per second unless followed by
                        /sec /minute /hour /day postfixes]
--limit-burst number    number to match in a burst, default 5
```

Please run the following commands on 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.

```
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
```

## (Optional 10pts) Task 5: Load Balancing

The `iptables` is 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 experimenting with one of the applications, load balancing. In this task, we will use it to load balance three UDP servers running in the internal network. Let's first start the server on each of the hosts: `192.168.60.5`, `192.168.60.6`, and `192.168.60.7` (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. You can type the following command to get its manual. You can see there are two modes: `random` and `nth`. We will conduct experiments using both of them.

```
$ iptables -m statistic -h
statistic match options:
 --mode mode          Match mode (random, nth)
 random mode:
[!] --probability p   Probability
 nth mode:
[!] --every n         Match every nth packet
 --packet p           Initial counter value (0 <= p <= n-1, default 0)
```

**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: for every three packets, pick the packet 0 (i.e., the first one), change its destination IP address and port number to `192.168.60.5` and `8080`, respectively. The modified packets will continue on its journey.

```
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
```

It should be noted that those packets that do not match the rule will continue on their journeys; they will not be modified or blocked. With this rule in place, if you send a UDP packet to the router's `8080` port, you will see that one out of three packets gets to `192.168.60.5`.

```
// On 10.9.0.5
echo hello | nc -u 10.9.0.11 8080
<hit Ctrl-C>
```

Please add more rules to the router container, so all the three internal hosts get the equal number of packets. Please provide some explanation for the rules.

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

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

Please use this mode to implement your load balancing rules, so each internal server get roughly the same amount of traffic (it may not be exactly the same, but should be close when the total number of packets is large). Please provide some explanation for the rules.

# 8 Submission and Demonstration

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

Alexander Crespo

# Part 1

a. Container setup and commands

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dcbuild
attacker uses an image, skipping
hostA uses an image, skipping
hostB uses an image, skipping
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dcup
Creating network "net-10.9.0.0" with the default driver
Pulling attacker (handsonsecurity/seed-ubuntu:large-arm)...
large-arm: Pulling from handsonsecurity/seed-ubuntu
82d728d38b98: Pulling fs layer
c06a3dc7ba9f: Pulling fs layer
82d728d38b98: Downloading [>                                                  ]  272.6kB/25.97MBiting

82d728d38b98: Pull complete
c06a3dc7ba9f: Pull complete
1622d55204e9: Pull complete
04e16e3db838: Pull complete
408a37ef18eb: Pull complete
768c10fd7ef1: Pull complete
c79874963658: Pull complete
Digest: sha256:dd7d27f1596068634a79cf7801d080147732f8d9913d25937e9e15f9f6e765ef
Status: Downloaded newer image for handsonsecurity/seed-ubuntu:large-arm
Creating hostA-10.9.0.5 ... done
Creating seed-attacker  ... done
Creating hostB-10.9.0.6 ... done
Attaching to seed-attacker, hostA-10.9.0.5, hostB-10.9.0.6
hostA-10.9.0.5 |  * Starting internet superserver inetd                [ OK ]
hostB-10.9.0.6 |  * Starting internet superserver inetd                [ OK ]
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dockps
41b272137b95 seed-attacker
6a3e62d8916b hostB-10.9.0.6
5c4ceee0fdd3 hostA-10.9.0.5
```

b. About the attack container

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ ifconfig
br-c75bea6b929d: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.1  netmask 255.255.255.0  broadcast 10.9.0.255
        inet6 fe80::42:d1ff:fe24:3adc  prefixlen 64  scopeid 0x20<link>
        ether 02:42:d1:24:3a:dc  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 36  bytes 4803 (4.8 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

# 1.1 A (Sniffing Packets)

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dockps
41b272137b95 seed-attacker
6a3e62d8916b hostB-10.9.0.6
5c4ceee0fdd3 hostA-10.9.0.5
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ docksh 5c4ceee0fdd3
root@5c4ceee0fdd3:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.327 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.264 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.273 ms
^C
--- 10.9.0.6 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
rtt min/avg/max/mdev = 0.264/0.288/0.327/0.027 ms
root@5c4ceee0fdd3:/#
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 53024
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x576c
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x34e4
        id        = 0x2
        seq       = 0x1
###[ Raw ]###
           load      = '\xe9\xc8\x0bh\x00\x00\x00\x00\x08\x15\x07\x00\x00\x00\x00\x00\x10\x11\x12\x13\x1
4\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ ./sniffer.py
Traceback (most recent call last):
  File "/home/seed/Desktop/Labs/Labsetup-arm/./sniffer.py", line 6, in <module>
    pkt = sniff(iface='br-c75bea6b929d', filter='icmp', prn=print_pkt)
  File "/home/seed/.local/lib/python3.10/site-packages/scapy/sendrecv.py", line 1424, in sniff
    sniffer._run(*args, **kwargs)
  File "/home/seed/.local/lib/python3.10/site-packages/scapy/sendrecv.py", line 1273, in _run
    sniff_sockets[_RL2(iface)(type=ETH_P_ALL, iface=iface,
  File "/home/seed/.local/lib/python3.10/site-packages/scapy/arch/linux/__init__.py", line 218, in __ini
t__
    self.ins = socket.socket(
  File "/usr/lib/python3.10/socket.py", line 232, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

It ran as expected. Sniffer just needs to be run with sudo to have root privileges. After running with sudo, I entered container A and pinged container B, and the sniffer caught it. I tried running without root privileges and it told me the operation was not permitted.

Alexander Crespo

# 1.1 B (Sniffing Packets)

```python
#!/usr/bin/env python3

from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface='br-c75bea6b929d', filter='icmp', prn=print_pkt)
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dockps
41b272137b95 seed-attacker
6a3e62d8916b hostB-10.9.0.6
5c4ceee0fdd3 hostA-10.9.0.5
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ docksh 5c4ceee0fdd3
root@5c4ceee0fdd3:/# ping 10.9.0.6
PING 10.9.0.6 (10.9.0.6) 56(84) bytes of data.
64 bytes from 10.9.0.6: icmp_seq=1 ttl=64 time=0.327 ms
64 bytes from 10.9.0.6: icmp_seq=2 ttl=64 time=0.264 ms
64 bytes from 10.9.0.6: icmp_seq=3 ttl=64 time=0.273 ms
^C
--- 10.9.0.6 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
rtt min/avg/max/mdev = 0.264/0.288/0.327/0.027 ms
root@5c4ceee0fdd3:/#
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 53024
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x576c
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x34e4
        id        = 0x2
        seq       = 0x1
###[ Raw ]###
           load      = '\xe9\xc8\x0bh\x00\x00\x00\x00\x08\x15\x07\x00\x00\x00\x00\x00\x10\x11\x12\x13\x1
4\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567'

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:0a:09:00:06
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
```

Original filter used with ICMP, so this was already done in the previous task. It worked as expected and caused no issues.

```python
#!/usr/bin/env python3

from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface='br-c75bea6b929d', filter='tcp and src host 10.9.0.5 and dst port 23', prn=print_pkt)
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dockps
41b272137b95 seed-attacker
6a3e62d8916b hostB-10.9.0.6
5c4ceee0fdd3 hostA-10.9.0.5
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ docksh 5c4ceee0fdd3
root@5c4ceee0fdd3:/# telnet 10.9.0.6
Trying 10.9.0.6...
Connected to 10.9.0.6.
Escape character is '^]'.
Ubuntu 20.04.6 LTS
6a3e62d8916b login: seed
Password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-138-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@6a3e62d8916b:~$
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ sudo ./sniffer.py
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x10
     len       = 60
     id        = 27568
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = tcp
     chksum    = 0xbadf
     src       = 10.9.0.5
     dst       = 10.9.0.6
     \options   \
###[ TCP ]###
        sport     = 55710
        dport     = telnet
        seq       = 3090711129
        ack       = 0
        dataofs   = 10
        reserved  = 0
        flags     = S
        window    = 64240
        chksum    = 0x144b
        urgptr    = 0
        options   = [('MSS', 1460), ('SAckOK', b''), ('Timestamp', (1771566897, 0)), ('NOP', None), ('WScale', 7)]

###[ Ethernet ]###
  dst       = 02:42:0a:09:00:06
  src       = 02:42:0a:09:00:05
  type      = IPv4
```

This worked once I figured out what to run, telnet was on destination port 23. After doing this the sniffer caught the packets of the new filter.

```python
#!/usr/bin/env python3

from scapy.all import *
def print_pkt(pkt):
    pkt.show()
pkt = sniff(iface='br-c75bea6b929d', filter='net 128.230.0.0/16', prn=print_pkt)
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ dockps
41b272137b95 seed-attacker
6a3e62d8916b hostB-10.9.0.6
5c4ceee0fdd3 hostA-10.9.0.5
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ docksh 41b272137b95
root@seedvm2004:/# ls
bin  boot  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var  volumes
root@seedvm2004:/# python3
Python 3.8.10 (default, May 26 2023, 14:05:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from scapy.all import *
>>> a = IP()
>>> a.dst = '128.230.1.1'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>> from scapy.all import *
>>> a = IP()
>>> a.src = '128.230.1.1'
>>> a.dst = '10.9.0.5'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
>>>
>>>
```

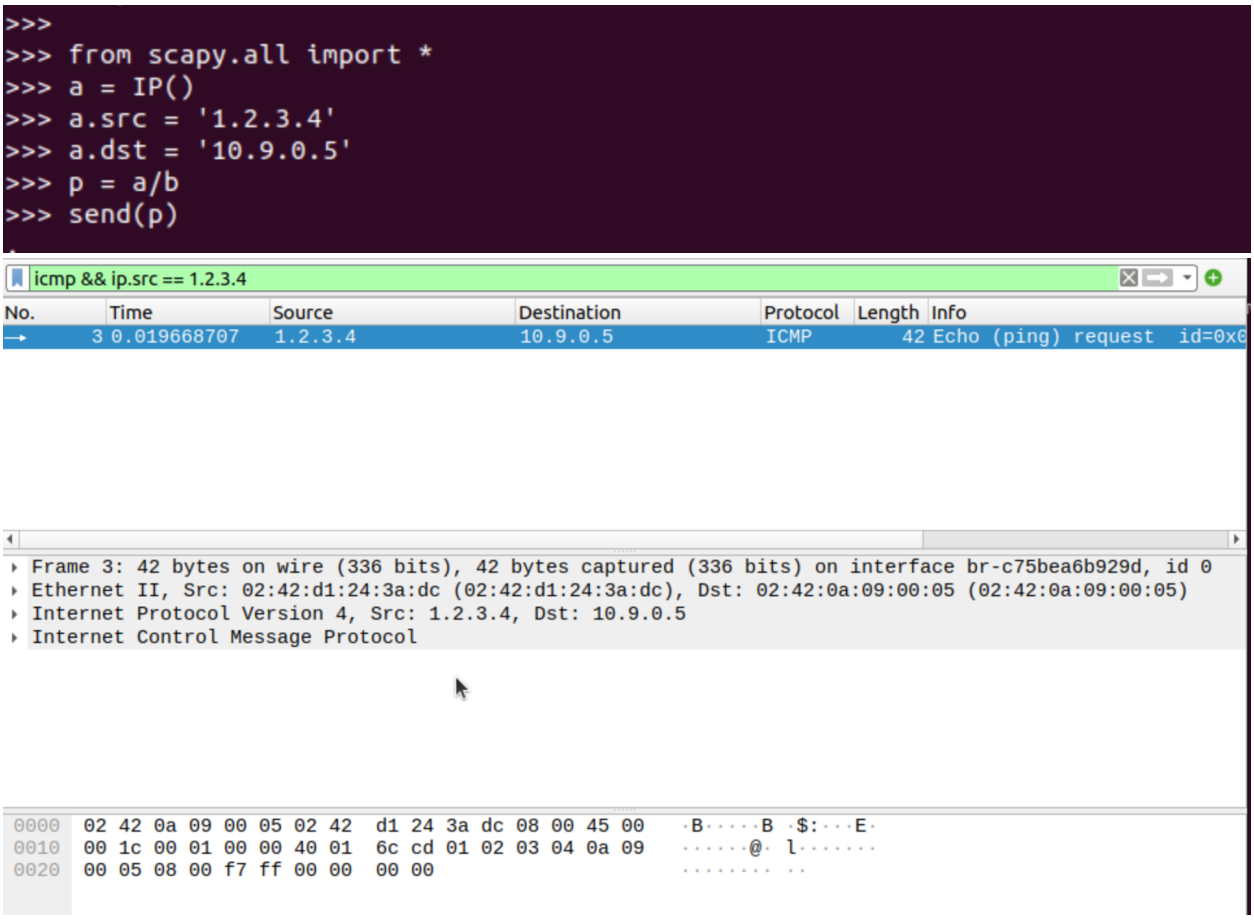
```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ sudo ./sniffer.py
[sudo] password for seed:
###[ Ethernet ]###
  dst       = 02:42:0a:09:00:05
  src       = 02:42:d1:24:3a:dc
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 28
     id        = 1
     flags     =
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0xeeeb
     src       = 128.230.1.1
     dst       = 10.9.0.5
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0xf7ff
        id        = 0x0
        seq       = 0x0

###[ Ethernet ]###
  dst       = 02:42:d1:24:3a:dc
  src       = 02:42:0a:09:00:05
  type      = IPv4
```

It was not working as expected but I wrote some code wrong. I put the destination as what should have been the source. After changing this it worked and the packets were successfully snipped.

# Task 1.2: Spoofing

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ sudo wireshark
 ** (wireshark:8216) 18:26:09.179321 [GUI WARNING] -- QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
 ** (wireshark:8216) 18:26:17.396408 [Capture MESSAGE] -- Capture Start ...
 ** (wireshark:8216) 18:26:17.436236 [Capture MESSAGE] -- Capture started
 ** (wireshark:8216) 18:26:17.436272 [Capture MESSAGE] -- File: "/tmp/wireshark_br-c75bea6b929dUYBN52.pcapng"
 ** (wireshark:8216) 18:26:55.692881 [Capture MESSAGE] -- Capture Stop ...
 ** (wireshark:8216) 18:26:55.724413 [Capture MESSAGE] -- Capture stopped.
 ** (wireshark:8216) 18:26:55.763101 [Capture MESSAGE] -- Capture Start ...
 ** (wireshark:8216) 18:26:55.804634 [Capture MESSAGE] -- Capture started
 ** (wireshark:8216) 18:26:55.804657 [Capture MESSAGE] -- File: "/tmp/wireshark_br-c75bea6b929d7KPE52.pcapng"
```




I used the attacker container to send a spoofed packet. It worked as expected and wireshark was able to capture it. I used the filter feature to find this specific packet and once sent the filter caught it.

## Task 1.3: Trace Route

```
1 #!/usr/bin/env python3
2
3 from scapy.all import *
4 a = IP()
5 a.dst = '10.9.0.5'
6 b = ICMP()
7
8 #See if 10 loops is enough
9 for i in range(1,10):
10     a.ttl=i
11     p = a/b
12     print(f"Sent with TTL {i}")
13     send(p)
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm$ sudo ./spoofedip.py
[sudo] password for seed:
Sent with TTL 1
.
Sent 1 packets.
Sent with TTL 2
.
Sent 1 packets.
Sent with TTL 3
.
Sent 1 packets.
Sent with TTL 4
.
Sent 1 packets.
Sent with TTL 5
.
Sent 1 packets.
Sent with TTL 6
.
Sent 1 packets.
Sent with TTL 7
.
Sent 1 packets.
Sent with TTL 8
.
Sent 1 packets.
Sent with TTL 9
.
Sent 1 packets.
```

```
10.9.0.5        ICMP    42 Echo (ping) request  id=0x0000, seq=0/0, ttl=1 (reply in 4)
10.9.0.1        ICMP    42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in 3)
10.9.0.5        ICMP    42 Echo (ping) request  id=0x0000, seq=0/0, ttl=2 (reply in 6)
10.9.0.1        ICMP    42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in 5)
10.9.0.5        ICMP    42 Echo (ping) request  id=0x0000, seq=0/0, ttl=3 (reply in 8)
10.9.0.1        ICMP    42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in 7)
10.9.0.5        ICMP    42 Echo (ping) request  id=0x0000, seq=0/0, ttl=4 (reply in 10)
10.9.0.1        ICMP    42 Echo (ping) reply    id=0x0000, seq=0/0, ttl=64 (request in 9)
10.9.0.5        ICMP    42 Echo (ping) request  id=0x0000, seq=0/0, ttl=5 (reply in 12)
```

It worked as expected but no packets were dropped from the first TTL. They all received replies as shown above.

# Part 2

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm-firewall$ dcbuild
HostA uses an image, skipping
Host1 uses an image, skipping
Host2 uses an image, skipping
Host3 uses an image, skipping
Building Router
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM handsonsecurity/seed-ubuntu:large-arm
 ---> fdeca9c488e3
Step 2/2 : RUN apt-get update     && apt-get install -y kmod      && apt-get clean
 ---> Using cache
 ---> 313b7bebf825
Successfully built 313b7bebf825
Successfully tagged seed-router-image:latest
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm-firewall$ dcup
Starting host3-192.168.60.7 ... done
Starting host1-192.168.60.5 ... done
Starting seed-router        ... done
Creating hostA-10.9.0.5     ... done
Starting host2-192.168.60.6 ... done
Attaching to seed-router, host1-192.168.60.5, host2-192.168.60.6, hostA-10.9.0.5, host3-192.168.60.7
host1-192.168.60.5 |  * Starting internet superserver inetd              [ OK ]
host2-192.168.60.6 |  * Starting internet superserver inetd              [ OK ]
host3-192.168.60.7 |  * Starting internet superserver inetd              [ OK ]
hostA-10.9.0.5 |  * Starting internet superserver inetd                  [ OK ]
seed-router |  * Starting internet superserver inetd                     [ OK ]
```

Containers were built and ran perfectly. They are ready to be used.

# Task 2.A: Protecting the Router

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm-firewall$ docksh 8c5c09354643
root@8c5c09354643:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@8c5c09354643:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@8c5c09354643:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@8c5c09354643:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@8c5c09354643:/# iptables -P OUTPUT DROP
root@8c5c09354643:/# iptables -P INPUT DROP
root@8c5c09354643:/#
```

```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm-firewall$ docksh 62d268ca2a24
root@62d268ca2a24:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.491 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.274 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.250 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.099 ms
64 bytes from 10.9.0.11: icmp_seq=5 ttl=64 time=0.160 ms
^C
--- 10.9.0.11 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4094ms
rtt min/avg/max/mdev = 0.099/0.254/0.491/0.133 ms
root@62d268ca2a24:/# telnet 10.9.0.11
Trying 10.9.0.11...
```

```
root@8c5c09354643:/# iptables -F
root@8c5c09354643:/# iptables -P OUTPUT ACCEPT
root@8c5c09354643:/# iptables -P INPUT ACCEPT
```

The commands were run inside the server container for the router. After running them we can see that it worked as expected. You are able to ping the router but telnet does not work. This is due to the rules that were put in place. The first rule was meant to allow ICMP echo requests which allowed the ping. Second rule allows ICMP echo replies which allows the response of the ping from the router. Third rule drops all outgoing packets that were not specified, meaning that the router does not send out any traffic unless explicitly allowed. Fourth rule is to drop all incoming packets that were not specified, not allowing the router to receive traffic except for the ones explicitly allowed. After I finished , I cleaned up the rules.

# Task 2.B: Protecting the Internal Network

Rules to protect internal network

```
root@8c5c09354643:/# iptables -A FORWARD -i eth0 -o br-34a9341d1ba9 -p icmp --icmp-type echo-request -j DROP
root@8c5c09354643:/#
root@8c5c09354643:/# iptables -A FORWARD -i br-34a9341d1ba9 -o eth0 -p icmp --icmp-type echo-request -j ACCEPT
root@8c5c09354643:/#
root@8c5c09354643:/# iptables -A FORWARD -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
root@8c5c09354643:/# iptables -L FORWARD -v -n --line-numbers
Chain FORWARD (policy DROP 18 packets, 1512 bytes)
num   pkts bytes target     prot opt in     out     source               destination
1        0     0 DROP       icmp --  eth0   br-34a9341d1ba9  0.0.0.0/0            0.0.0.0/0            icmptype 8
2        0     0 ACCEPT     icmp --  br-34a9341d1ba9 eth0    0.0.0.0/0            0.0.0.0/0            icmptype 8
3        0     0 ACCEPT     all  --  *      *       0.0.0.0/0            0.0.0.0/0            ctstate RELATED,ESTABLISHED
root@8c5c09354643:/#
```

Host A Container

```
root@62d268ca2a24:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.422 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.269 ms
^C
--- 10.9.0.11 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1016ms
rtt min/avg/max/mdev = 0.269/0.345/0.422/0.076 ms
root@62d268ca2a24:/# ping 192.168.60.11
PING 192.168.60.11 (192.168.60.11) 56(84) bytes of data.
64 bytes from 192.168.60.11: icmp_seq=1 ttl=64 time=0.195 ms
64 bytes from 192.168.60.11: icmp_seq=2 ttl=64 time=0.246 ms
64 bytes from 192.168.60.11: icmp_seq=3 ttl=64 time=0.239 ms
^C
--- 192.168.60.11 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2038ms
rtt min/avg/max/mdev = 0.195/0.226/0.246/0.022 ms
root@62d268ca2a24:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7164ms

root@62d268ca2a24:/#
```

Host B Container

```
root@0cd676cf2d22:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
^C
--- 10.9.0.5 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1002ms

root@0cd676cf2d22:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=64 time=0.143 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.153 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.123 ms
^C
--- 192.168.60.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2033ms
rtt min/avg/max/mdev = 0.123/0.139/0.153/0.012 ms
root@0cd676cf2d22:/#
```

Everything worked as expected after figuring out the rules to put in. Hardest part was figuring out how they worked but once it was known it was not too bad. I was able to get things running once I asked ai to help me find my mistakes, and it pointed out that I had my internal interface wrong. Once the internal interface was corrected it worked and the rules provided what was needed.

## Task 2.C: Protecting Internal Servers

```
root@8c5c09354643:/# iptables -A FORWARD -i eth0 -d 192.168.60.5 -p tcp --dport 23 -j ACCEPT
root@8c5c09354643:/# iptables -A FORWARD -o eth0 -s 192.168.60.5 -p tcp --sport 23 -j ACCEPT
root@8c5c09354643:/#
root@8c5c09354643:/# iptables -A FORWARD -i eth0 -d 192.168.60.0/24 -p tcp --dport 23 -j DROP
root@8c5c09354643:/# iptables -A FORWARD -o eth0 -s 192.168.60.0/24 -p tcp --sport 23 -j DROP
root@8c5c09354643:/# iptables -A FORWARD -i br-34a9341d1ba9 -o br-34a9341d1ba9 -p tcp -j ACCEPT
root@8c5c09354643:/# iptables -A FORWARD -i br-34a9341d1ba9 -o eth0 -p tcp -j DROP
```

```
root@62d268ca2a24:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.6 LTS
0cd676cf2d22 login: seed
Password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-138-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@0cd676cf2d22:~$ exit
logout
Connection closed by foreign host.
```

```
root@62d268ca2a24:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
```

```
root@3a67b48b89ab:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.6 LTS
0cd676cf2d22 login: seed
Password:
Welcome to Ubuntu 20.04.6 LTS (GNU/Linux 5.15.0-138-generic aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Fri Apr 25 21:50:22 UTC 2025 from 10.9.0.5 on pts/2
seed@0cd676cf2d22:~$ exit
logout
Connection closed by foreign host.
```

```
root@3a67b48b89ab:/# telnet 10.9.0.5
Trying 10.9.0.5...
^C
```

Firewall rules were configured to achieve the objectives of this task. After implementing it worked as expected and is shown in the snippets above. Telnet from outside to 192.168.60.5 worked. Telnet from outside to any of the internal hosts was blocked. Internal hosts could telnet to each other. Internal hosts were not able to telnet to external hosts.

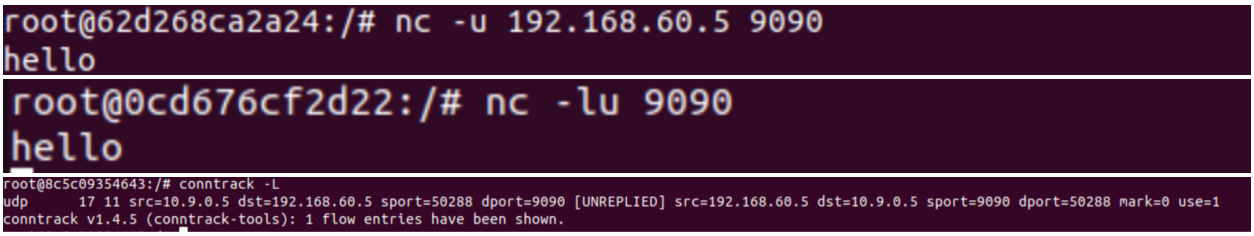
## Task 3.A: Experiment with the Connection Tracking

```
root@62d268ca2a24:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.200 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.308 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.291 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.341 ms
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3064ms
rtt min/avg/max/mdev = 0.200/0.285/0.341/0.052 ms
root@62d268ca2a24:/# conntrack -L
icmp     1 5 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=35 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=35 mark=0 use=1
icmp     1 14 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=36 src=192.168.60.5 dst=10.9.0.5 type=0 code=0 id=36 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 2 flow entries have been shown.
```

For the first experiment 4 packets were sent. I used the conntrack -L command and it had two entries. The ttls shown on the entries were 5 and 14 showing a short time kept.

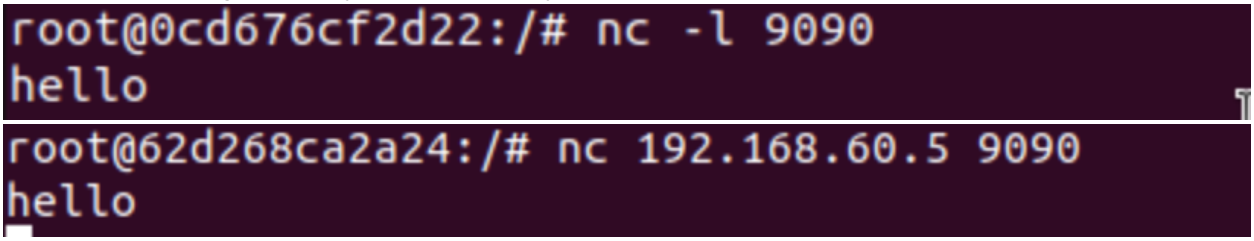
```
seed@seedvm2004:~/Desktop/Labs/Labsetup-arm-firewall$ docksh 0cd676cf2d22
root@0cd676cf2d22:/# nc -lu 9090
```

Alexander Crespo




```
root@8c5c09354643:/# conntrack -L
udp      17 11 src=10.9.0.5 dst=192.168.60.5 sport=50288 dport=9090 [UNREPLIED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=50288 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

After the second experiment the conntrack table only had one entry. This entry's ttl was only 11 seconds, showing the entry was temporary.




```
root@8c5c09354643:/# conntrack -L
tcp      6 431993 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=36216 dport=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=36216 [ASSURED] mark
=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
```

After the last experiment the conntrack table had one entry again. It showed that both sides communicated. The ttl was 431993 showing that tcp connections are kept long term, not like the other experiments.

All the experiments worked as expected, and they showed the differences between each protocol.