Firewall Exploration Lab

Ethical Hacking 2022/23, University of Padua

Eleonora Losiouk, Alessandro Brighente, Gabriele Orazi, Francesco Marchiori

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`

2 Environment Setup

For this lab, the network you will run with Docker will be composed of some machine as follows:

```
                     Net. 10.9.0.0/24
----------------------------------------------------------
       |                |                |  10.9.0.11
    Attacker        10.9.0.5          Router
    10.9.0.1                             |  192.168.60.11
                                         |
                Net. 192.168.60.0./24    |
                ------------------------------------------
                    |               |                |
               192.168.60.5    192.168.60.6   192.168.60.7
```

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file to your machine (or in you VM, if you are using it) from Moodle, unzip it, enter the `Labsetup` folder, and use the `docker-compose.yml` file to set up the lab environment. You can find more details and ways to resolve some problems in this manual.

In the following, we list some of the commonly used commands related to docker-compose. Since we are going to use these commands very frequently, we have created aliases for them in the `.bashrc` file (already configured in our provided VM, but you can easily add them to your local favourite `rc` file).

```
$ 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 that 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 want to use your local `rc` file, you can simply paste at the end the following:

```
# Aliases for the Docker Compose
alias dcbuild='sudo docker-compose build' # Alias for: docker-compose build
alias dcup='sudo docker-compose up' # Alias for: docker-compose up
alias dcdown='sudo docker-compose down' # Alias for: docker-compose down
alias dockps='sudo docker ps --format "{{.ID}} {{.Names}}"'
alias docksh='f(){ sudo docker exec -it $1 /bin/bash;  unset -f f; }; f'
```

Then, remember to relaunch the terminal or source your `rc` file.

---

3 Task 1: Implement a simple Firewall

In this task, we will implement a simple packet filtering type of firewall, which inspects each incoming and outgoing packets, 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. In this lab, we will just set the kernel module from the host VM.

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 the other tasks.

3.1 Task 1.A: Implement a Simple Kernel Module

`LKM` allows us to add a new module to the kernel at the 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.

```
/* Listing 1: hello.c */
#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);
```

We now need to create `Makefile`, which includes the following contents (the file is included in the lab setup files). Just type `make` (with `sudo` if needed), and the above program will be compiled into a loadable kernel module (if you copy and paste the following into `Makefile`, make sure *replace the spaces before the* `make` *commands with a tab*).

```
obj-m += hello.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

The generated kernel module is in `hello.ko`. You can use the following commands to load the module, list all modules, and remove the module. Also, you can use "`modinfo hello.ko`" to show information about a Linux Kernel module.

```
$ sudo insmod hello.ko  # inserting a module
$ lsmod | grep hello    # list modules
$ sudo rmmod hello      # remove the module
$ dmesg                 # check the messages
```

Task. Please compile this simple kernel module on your VM, and run it on the VM. Remember that for this task, we will not use any containers.

Notes. If you prefer to use your own Ubuntu machine, make sure to install all the needed packets (`sudo apt install build-essentials gcc make flex` and all the other requirements). You may have to debug some errors while compiling the kernel module, so for this lab it is advised to use the VM. Another reason to use the VM is to avoid problems with your machine since, in the following tasks, we will mess with the kernel.

3.2 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 in into the packet processing path inside the kernel. This cannot be easily done in the past before the `netfilter` was introduced into the 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 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.

Hooking to Netfilter. Using `netfilter` is quite straightforward. All we need to do is to hook our functions (in the kernel module) to the corresponding `netfilter` hooks. Here we show an example (the code is in `Labsetup/packet_filter`, but it may not be exactly the same as this example).

The structure of the code follows the structure of the kernel module implemented earlier. When the kernel module is added to the kernel, the `registerFilter()` function in the code will be invoked. Inside this function, we register two hooks to `netfilter`. To register a hook, you need to prepare a hook data structure, and set all the needed parameters, the most important of which are a function name (Line (1)) and a hook number (Line (2)). The hook number is one of the 5 hooks in `netfilter`, and the specified function will be invoked when a packet has reached this hook. In this example, when a packet gets to the `LOCAL IN` hook, the function `printInfo()` will be invoked (this function will be given later). Once the hook data structure is prepared, we attach the hook to `netfilter` in Line (3)).

```
/* Listing 2: Register hook functions to netfilter */
static struct nf_hook_ops hook1, hook2;

int registerFilter(void) {
    printk(KERN_INFO "Registering filters.\n");

    // Hook 1
    hook1.hook = printInfo;                     // (1)
    hook1.hooknum = NF_INET_LOCAL_IN;           // (2)
    hook1.pf = PF_INET;
    hook1.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook1);    // (3)
    // Hook 2
    hook2.hook = blockUDP;
    hook2.hooknum = NF_INET_POST_ROUTING;
    hook2.pf = PF_INET;
    hook2.priority = NF_IP_PRI_FIRST;
    nf_register_net_hook(&init_net, &hook2);
    return 0;
}
void removeFilter(void) {
    printk(KERN_INFO "The filters are being removed.\n");
    nf_unregister_net_hook(&init_net, &hook1);
    nf_unregister_net_hook(&init_net, &hook2);
}
module_init(registerFilter);
module_exit(removeFilter);
```

Note for Ubuntu 16.04 VM: If you are using the 16.04 VM, you have to make some changes in the hook registration and un-registration APIs. See the difference in the following:

```
// Hook registration:
nf_register_hook(&nfho);                      // For Ubuntu 16.04 VM
nf_register_net_hook(&init_net, &nfho);       // For Ubuntu 20.04 VM

// Hook unregistration:
nf_unregister_hook(&nfho);                    // For Ubuntu 16.04 VM
nf_unregister_net_hook(&init_net, &nfho);     // For Ubuntu 20.04 VM
```

Hook functions. We give an example of hook function below. It only prints out the packet information. When `netfilter` invokes a hook function, it passes three arguments to the function, including a pointer to the actual packet (`skb`). In the following code, Line (1) shows how to retrieve the hook number from the state argument. In Line (2), we use `ip_hdr()` function to get the pointer for the IP header, and then use the `%pI4` format string specifier to print out the source and destination IP addresses in Line (3).

```
/* Listing 3: An example of hook function */
unsigned int printInfo(void *priv, struct sk_buff *skb,
                        const struct nf_hook_state *state) {
    struct iphdr *iph;
    char *hook;
    switch (state->hook) {                                    // (1)
        case NF_INET_LOCAL_IN:
            printk("*** LOCAL_IN"); break;
        // .. (code omitted) ...
    }
    iph = ip_hdr(skb);                                        // (2)
    printk(" %pI4 --> %pI4\n", &(iph->saddr), &(iph->daddr));   // (3)
    return NF_ACCEPT;
}
```

If you need to get the headers for other protocols, you can use the following functions defined in various header files. The structure definition of these headers can be found inside the `/lib/modules/5.4.0-54-generic/build/include/uapi/linux` folder, where the version number in the path is the result of "`uname -r`", so it may be different if the kernel version is different.

```
struct iphdr *iph = ip_hdr(skb)          // (need to include <linux/ip.h>)
struct tcphdr *tcph = tcp_hdr(skb)       // (need to include <linux/tcp.h>)
struct udphdr *udph = udp_hdr(skb)      // (need to include <linux/udp.h>)
struct icmphdr *icmph = icmp_hdr(skb)   // (need to include <linux/icmp.h>)
```

Blocking packets. We also provide a hook function example to show how to block a packet, if it satisfies the specified condition. The following example blocks the UDP packets if their destination IP is `8.8.8.8` and the destination port is `53`. This means blocking the DNS query to the nameserver `8.8.8.8`.

Listing 4: Code example: blocking UDP

```
unsigned int blockUDP(void *priv, struct sk_buff *skb,
                        const struct nf_hook_state *state) {
    struct iphdr *iph;
    struct udphdr *udph;
    u32 ip_addr;
    char ip[16] = "8.8.8.8";
    // Convert the IPv4 address from dotted decimal to a 32-bit number
    in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);              // (1)
    iph = ip_hdr(skb);
    if (iph->protocol == IPPROTO_UDP) {
        udph = udp_hdr(skb);
        if (iph->daddr == ip_addr && ntohs(udph->dest) == 53){  // (2)
            printk(KERN_DEBUG "****Dropping %pI4 (UDP), port %d\n",
        &(iph->daddr), port);
        return NF_DROP;                                         // (3)
        }
    }
    return NF_ACCEPT;                                           // (4)
}
```

In the code above, Line (1) shows, inside the kernel, how to convert an IP address in the dotted decimal format (i.e., a string, such as `1.2.3.4`) to a 32-bit binary (`0x01020304`), so it can be compared with the binary number stored inside packets. Line (2) compares the destination IP address and port number with the values in our specified rule. If they match the rule, the `NF_DROP` will be returned to `netfilter` (Line (3)), which will drop the packet. Otherwise, the `NF_ACCEPT` will be returned (Line (4)), and `netfilter` will let the packet continue its journey (`NF_ACCEPT` only means that the packet is accepted by this hook function; it may still be dropped by other hook functions).

Tasks. The complete sample code is called `seedFilter.c`, which is included in the lab setup files (inside the `Files/packet_filter` folder). Please do the following tasks (do each of them separately):

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

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

   ```
   NF_INET_PRE_ROUTING
   NF_INET_LOCAL_IN
   NF_INET_FORWARD
   NF_INET_LOCAL_OUT
   NF_INET_POST_ROUTING
   ```

3. Implement two more hooks to achieve the following:

   1. Preventing other computers to ping the VM

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

```
ping 10.9.0.1
telnet 10.9.0.1
```

Important note: Since we make changes to the kernel, there is a high chance that you would crash the kernel. Make sure you back up your files frequently, so you don't lose them. One of the common reasons for system crash is that you forget to unregister hooks. When a module is removed, these hooks will still be triggered, but the module is no longer present in the kernel. That will cause system crash. To avoid this, make sure for each hook you add to your module, add a line in removeFilter to unregister it, so when the module is removed, those hooks are also removed.

---

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.

Table 1: iptables Tables and Chains

Table	Chain	Functionality
filter	INPUT	Packet filtering
	FORWARD	
	OUTPUT	
nat	PREROUTING	Modifying source or destination
	INPUT	network addresses
	OUTPUT	
	POSTROUTING	
mangle	PREROUTING	Packet content modification

Table	Chain	Functionality
	INPUT FORWARD OUTPUT POSTROUTING	

5 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 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 1. Remember that `iptables` entries are applied in order: the first specified rule will be the first one to be taken into consideration. If there is no match, the following entry will be checked. In case no other entries are present, the default behavior will be applied. If you think about a simple table, imagine that when you add a row you append it at the bottom and when you receive a packet you always start checking from the top.

Moreover, remember that as soon as there is a match for an entry, the specified action will be taken and no other entries are checked anymore since the packet is basically considered as processed.

Note 2. 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.

5.1 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>
```

5.2 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
```

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

5.3 Task 2.C: Protecting Internal Server

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 -o eth1 -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.

---

6 Task 3: Connection Tracking and Stateful Firewall (Optional)

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.

6.1 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.

1. 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
```

2. 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>
```

3. 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>
```

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

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.

---

7 Task 4: Limiting Network Traffic (Optional)

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
```

---

8 Task 5: Load Balancing (Optional)

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)
```

1. 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.

2. 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.

---

9 Extra

Since Ubuntu 8.04 LTS, the `UncomplicatedFirewall` (`ufw`) was added as a frontend to the more complete but complex `iptables`. You can take a look at the documentation on the Ubuntu Wiki page or at the manual. You can decide to try again some of the tasks of this lab using `ufw`.