Lab 6: IPTABLES Firewall

> You must start the lab computer using the **Linux partition**. Your instructor will provide you with a **username and password**, which will be **different from your usual credentials**.

The **TCP/IP protocol stack** is part of virtually every modern operating system, including **GNU/Linux, Windows, macOS, Android**, and **iOS**. Since it is the **operating system** that manages the communication requests made by applications, it is quite natural to consider the possibility of adjusting the behaviour of those communications to define policies deemed appropriate by the system administrator.

Each operating system includes its own way of specifying those preferences. In the case of **GNU/Linux**, the tool used for this purpose is **iptables**.

You are probably already familiar with the term **firewall** in a computer networking context. Firewalls are devices (traditionally hardware-based) that allow certain types of network traffic to be filtered in order to **protect systems and internal networks from external threats**.

Many companies place a firewall between their **local network and the Internet connection**. This device includes rules that filter specific packets to improve **internal network security**.

Iptables is a very powerful **software tool** that allows administrators to **consult and modify network traffic management policies** on each system. Among other functions, it supports typical firewall features (such as **blocking unwanted traffic**), **NAT protocol management on routers**, **event logging**, and **modifying the contents of IP datagram headers**. In this lab, we will explain and put into practice some of the **basic uses** of this powerful tool.

One of the main drawbacks of iptables is that its **configuration is not persistent**—when the system is shut down or restarted, the configuration is lost. The administrator must manually create a **script** to be executed during the system's startup in order to **load the desired iptables configuration**.

Due to the complexity of managing iptables, and the fact that its configuration doesn't persist after shutdown, modern Linux systems often include a tool called **ufw (Uncomplicated Firewall)**. This tool allows **easier and more user-friendly management** of common firewall options. Ufw internally uses iptables, so it can be seen as a **convenient frontend** for iptables. Additionally, **ufw configures the operating system's startup process** so that the **latest configuration is loaded automatically** on subsequent reboots.

However, it is possible that the Linux system we need to configure **does not include any additional tools** for managing iptables, meaning we will need to work directly with iptables itself. For this reason, **basic knowledge** of its functionality is **essential** for any administrator.

Basic Concepts

In **iptables**, datagrams arriving at or leaving the system through any network interface—including **virtual ones such as loopback**—are processed following the **flow diagram** below

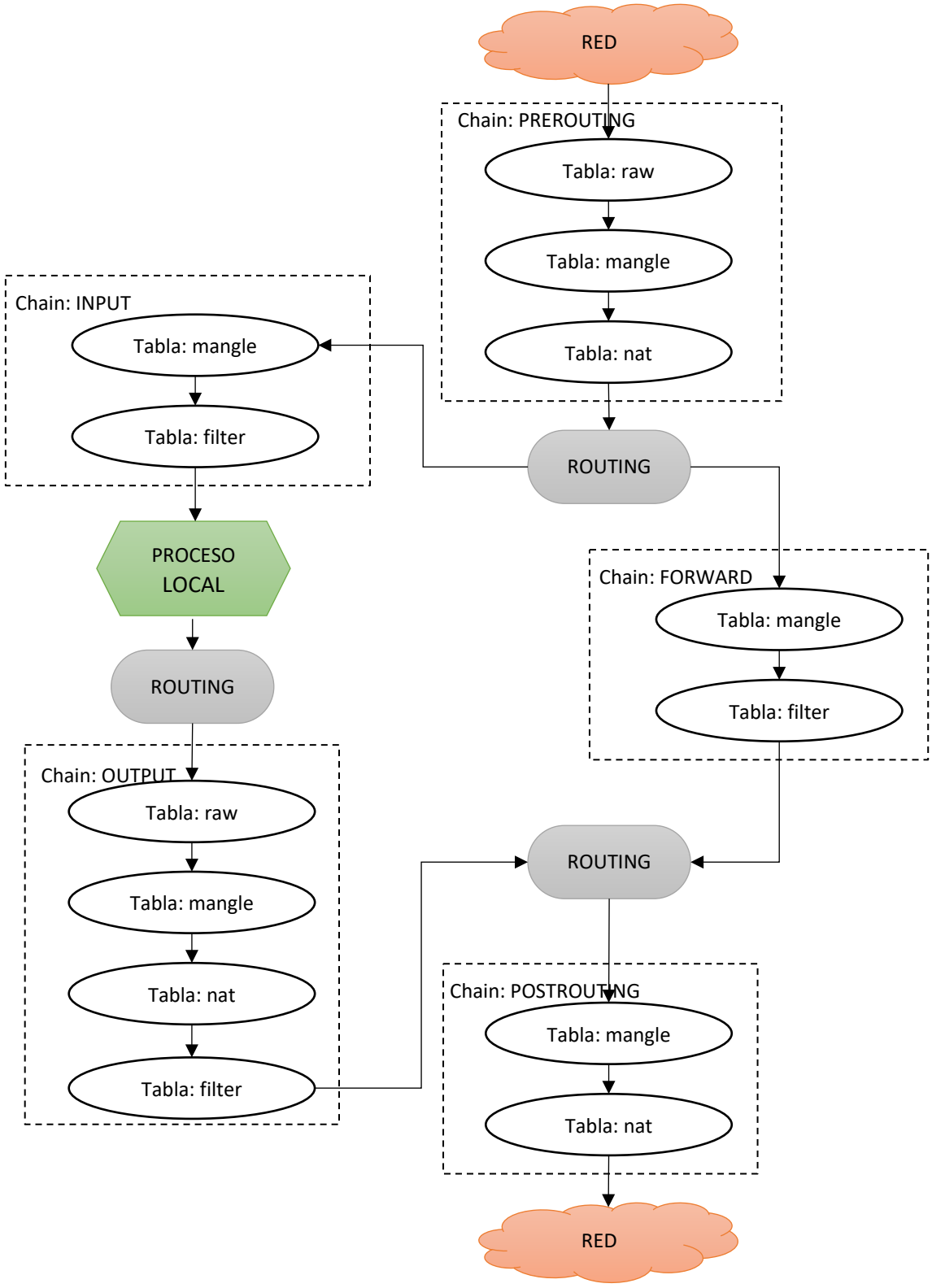

Figure 1: iptables Flow diagram.

To understand this flow diagram, it's important to note that **iptables** is organized into a set of **tables**. Each table contains a set of **predefined chains**, and each chain contains **rules** that are evaluated in sequential order.

In the diagram, we can identify **three possible paths** that IP datagrams can take in iptables. On a **host**, **incoming traffic** follows the path from the **network** (shown at the top) to the **local process**. **Outgoing traffic** follows the path from the local process to the network (shown at the bottom). In **routers**, transit traffic follows the path from the top network to the bottom one, passing through the **FORWARD chain** (shown on the right).

For example, an **incoming datagram** on a personal computer will go through the following stages in iptables.

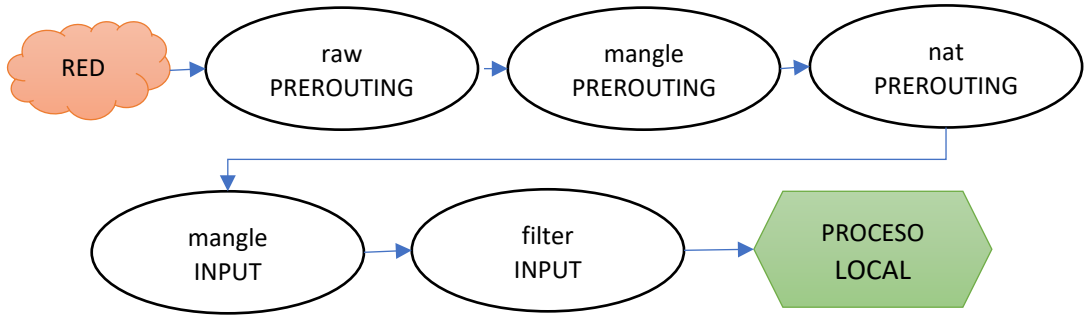

Figure 2: iptables Flow diagram for incoming traffic.

Note: The lowercase name is the name of the table, and the uppercase name is the name of the rule chain.

As shown, **not all rules of an entire table** are processed at each stage, but rather only the rules in the **relevant chain**. You'll see in the diagram that the **mangle table** processes two different chains at different points: **PREROUTING** and **INPUT**.

An **outgoing datagram** on a personal Linux computer follows this flow:

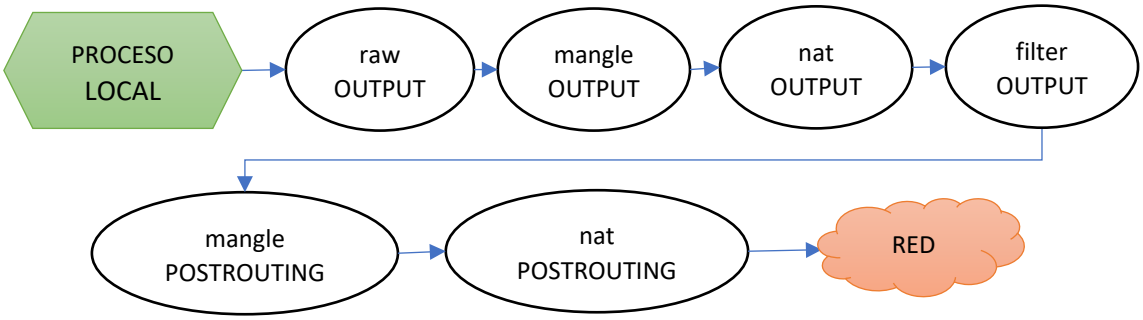

Figure 3: iptables Flow diagram for outgoing traffic.

We can observe that **incoming and outgoing datagrams** traverse **different chains** in each table. Therefore, when adding rules that affect traffic, we must be aware of whether we want to act on **incoming** or **outgoing** traffic.

iptables Tables

The **iptables** tool includes five tables:

- **raw**: Filters packets **before any other table**. In the previous diagrams, the first chain that a datagram passes through in both directions belongs to the raw table.

- **mangle**: Contains rules to **modify fields in TCP/IP headers**, such as **ToS or TTL** in IP, or **SYN, RST flags**, etc., in TCP.
- **nat**: Used to perform **Network Address Translation (NAT)**.
- **security**: Used for **Mandatory Access Control (MAC)** rules.
- **filter**: This is the **default table**. It contains **filtering rules**. Through this table, for example, you can **drop or accept IP datagrams** based on various conditions.

The **filter table** is the **most commonly used**, as most **firewall operations** are performed on it. Within this table, we will work primarily with the **INPUT** and **OUTPUT** chains.

Rule Chains in *iptables*

As mentioned earlier, each table contains several **chains of rules**. In general, a **rule** consists of a **condition** and an **action**. When evaluating a rule, **iptables checks whether the condition matches the datagram**, and if so, it performs the **specified action**.

The main actions we will use in this lab are **ACCEPT** and **DROP**:

- A rule with the **ACCEPT** action means the datagram is allowed.
- A rule with **DROP** will silently discard the datagram.

Instead of **DROP**, you can also use **REJECT**, which sends back an **ICMP "port unreachable"** message (this is the default behaviour). Using **REJECT** blocks access to ports more politely, but also **allows attackers to scan for open ports more easily**.

It's important to understand that rules in a chain are **processed sequentially**, and as soon as one **matches the datagram**, the **corresponding action is applied** (e.g., DROP or ACCEPT), and the rest of the rules in the chain **are not processed**. If **none of the rules match**, the **default policy** for that chain is applied.

Some Useful Commands

There are some **helpful iptables commands** that can get you out of trouble when you have made a mistake while creating rules.

To view the status of a table in iptables, use the following command (square brackets indicate optional parameters):

```
sudo iptables [-t table] -L [chain]
```

The default table is **filter**, so if you omit the -t parameter, it will show the status of that table. If no **chain** is specified, it will show **all chains** in the table.

Examples:

```
sudo iptables -L
```

Shows the status of **all chains in the filter table**.

```
sudo iptables -L INPUT
```

Shows the status of the **INPUT chain** in the filter table.

Sometimes, if you've added several incorrect rules and are unsure how to fix them, the easiest solution is to **flush all rules** from the table and **reinsert them**.

To flush all rules from a table or chain, use the -F parameter:

Examples:

```
sudo iptables -F
```

Flushes **all rules from all chains** in the **filter table**.

```
sudo iptables -t nat -F
```

Flushes all rules from all chains in the **nat table**.

```
sudo iptables -F OUTPUT
```

Flushes all rules from the **OUTPUT chain** in the **filter table**.

Default Policies

The **default policy** of a chain determines the **default behaviour** for datagrams when the chain contains **no rules**, or when a datagram **does not match any rules** within that chain.

This **general behaviour**, which affects **all types of traffic**, can later be **refined** by adding more specific rules that modify the default behaviour for specific packets. For example, you may decide to **initially accept all traffic**, but then add a new rule to **block a specific type of traffic**, such as **SSH connections**.

Using **iptables**, the administrator defines a **default policy** for **incoming or outgoing traffic**, and then uses a **set of additional rules** to allow or block specific types of network traffic. In this process, it is essential to carefully define the **most appropriate default policy**.

If the goal is to create a **highly restrictive system**, the best approach is to **block all traffic** by default, and then **explicitly authorize** only specific communications. In this case, we can start by **blocking all outgoing traffic**, and then **gradually authorize** only the necessary services — such as **DNS server access** or **SSH connections** to specific servers. All other traffic, unless explicitly allowed, will be rejected by the restrictive default policy.

On the other hand, you may only want to **block certain types of "annoying" traffic** without affecting the rest of the system's services. For instance, you might want to prevent users from **printing to a remote printer** from that machine. In this scenario, it makes more sense to **start with a permissive default policy** that **accepts all traffic**, and then add a rule to **specifically block traffic to that printer**.

How to Change the Default Policy Using *iptables*

A **deep understanding** of iptables requires much more time than this lab session allows. Since **traffic management is handled by the OS kernel**, only the **administrator** can access this tool. Because we do not have admin access on the lab machines, we will **use sudo** before every iptables command.

Try executing the following command to **view the current state** of all chains in the **filter table**:

```
$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

In the **initial configuration**, for each of the three chains (**INPUT**, **FORWARD**, and **OUTPUT**), the default policy is to **accept all traffic** (policy ACCEPT), regardless of its **source**, **destination**, or **protocol**. This setup — which we could describe as **fully open** — does **not restrict incoming traffic (INPUT)**, **outgoing traffic (OUTPUT)**, or even **forwarded traffic (FORWARD)**, in case the machine was acting as a **router**. Although this last case is not enabled on lab computers, each machine does have two network interfaces and **could act as a router** if necessary.

Exercise 1

1. Let's begin by modifying the initial configuration to see its effects. We'll use the **-P** (policy) option by typing the following command:

   `sudo iptables -P OUTPUT DROP`

   IMPORTANT: Do not close the terminal window where you run this command.

2. Now, run the following command: **ping 127.0.0.1** Observe what happens[1]. You should stop the ping after a few seconds using **Ctrl+C**.

What happens is that this command changes the **default policy** for all **outgoing traffic**, including traffic that does not leave the system (i.e., to localhost). As a result, the system becomes something like a **black hole** on the network — **no packets can exit**. A similar effect (though not as severe) occurs if you **unplug the network cable** (although in that case, the previous ping **would still work** correctly).

[1] The address "127.0.0.1" corresponds to the IPv4 localhost interface of our machine. If we use "localhost" directly instead, the packet would be sent through the IPv6 loopback interface, whose iptables are different and managed by another command (ip6tables).

Exercise 2

1. Since the previous experience isn't particularly useful (it results in total disconnection), we will **restore the system** to its initial state. In the **same terminal window** where you ran the previous command, type:

   ```
   sudo iptables -P OUTPUT ACCEPT
   ```

2. Confirm that **network services are working again**. Run: **ping 127.0.0.1** Observe what happens now.

This is the **same state as at the beginning**: **all traffic flows without restrictions**.

How to Add and Remove Rules in Chains with the *iptables* Command

In this lab, we can make our computer selectively reject certain types of traffic. To do this, we'll need more specific rules than the one used in the first exercise.

Using the iptables command, you can **add or delete rules** to any chain within any table. You can append a rule to the **end** of a chain with the -A (Append) option, insert it at the **beginning** of a chain with -I (Insert), and delete it with -D (Delete). Of course, when adding a rule, you must specify the **table** (filter, mangle, nat, etc.) and the **chain** (INPUT, OUTPUT, etc.) to which you want to add or delete the rule. If you don't specify the table, the rule will be added to the **filter** table by default.

You can also delete **all rules** from a chain with the -F (Flush) option.

Remember: rules in a chain are processed **sequentially**, and once a rule **matches**, its **action is executed**, and no further rules in that chain are processed. This means that to **undo an effect**, the solution is to **delete the rule that causes it**, rather than trying to add a contradictory one (e.g., adding an ACCEPT after a DROP won't override the earlier DROP).

Exercise 3

1. Let's block **local traffic**, i.e., traffic over the **loopback** device. Type the following command:

   ```
   sudo iptables -A INPUT -i lo -j DROP
   ```

2. Now check if the rule had the desired effect by typing: **ping 127.0.0.1** Do you get a response? Try **ping 158.42.4.23** Does it work? Yes
   No
3. To **restore local traffic**, simply delete the previous rule::

   ```
   sudo iptables -D INPUT -i lo -j DROP
   ```

   (Remember: -D is for **deleting** a rule, while -A is for **adding** one.)

In this exercise, we saw how to use the -i parameter to specify a **network interface** (you can run *ip addr* to list all available interfaces and their current configurations), and the -j parameter to specify **what action** to take on matching traffic.

The **lo** device is not a real network card — it represents internal communication via the **loopback address 127.0.0.1**.

In the example, we determined that all **incoming traffic (INPUT)** on the **lo** device should be **dropped**.

To make many rules useful, it's not enough to specify the interface — it's also necessary to define rules based on **protocols** and/or **ports**.

Exercise 4

1. Verify that you can connect to zoltar.redes.upv.es via **SSH**. Use the username redesXX (where XX is your assigned workstation number) and the password given by your instructor:

   ```
   ssh redesXX@zoltar.redes.upv.es
   ```

2. Open **another terminal** and type:

   ```
   sudo iptables -A INPUT -p tcp --sport 22 -j DROP
   ```

3. Return to the **SSH window**, type 123456. What happens? Why?
   Numbers don't type, we don't receive any information from the server.
4. Go back to the **second terminal**, and delete the rule you just added:

   ```
   sudo iptables -D INPUT -p tcp --sport 22 -j DROP
   ```

   What happens now? Does your SSH session to zoltar still work?
   All messages sent are suddenly received, it still works.
5. End the SSH session by typing: "exit". Close that terminal.

In the previous exercise, we blocked **incoming SSH traffic** (INPUT chain). If you took more than a minute between steps 2, 3, and 4, the SSH connection might have dropped. In that case, try the exercise again a bit faster to observe a different result.

In this example, we created a rule that did not specify a device, but did define a **protocol** (-p tcp) and a **source port** (--sport 22). For SSH connections to zoltar, packets coming from the server have source port **22**, the default port for SSH.

Similarly, you could use this strategy to block access to other services. Rules can be applied to **incoming**, **outgoing**, or **both** directions of traffic.

Rules can also be based on **IP addresses**.

Exercise 5

1. Open a browser and load the page http://www.upv.es/
2. In a terminal window, type[2]:

```
sudo iptables -A OUTPUT -p tcp -d www.upv.es --dport 80 -j DROP
```

3. View the chain status with: **`sudo iptables -L`**
4. Try reloading the page in your browser. What happens? It can't reload.
5. Try visiting another site like: www.ua.es — does it work? Yes
6. Delete the rule you added in step 2.
7. Run **`sudo iptables -L`** to check the current list of rules.

In this exercise, we created a rule to **drop all outgoing TCP traffic** destined for www.upv.es on port **80** (HTTP). This rule does **not affect** similar traffic to other servers — only blocks access to the **UPV's main web server**.

You can create a similar set of rules to block access to a list of web servers. Remember: unless you **delete** this rule, its effect will persist throughout the session.

Sometimes it's useful to use the **negation operator !** to reverse the meaning of certain parameters like: **protocol** (-p), **source** (-s), **destination** (-d), or port (--dport, --sport). For example: **sudo iptables -A OUTPUT ! -d[3] localhost** ..., This rule applies to **all packets** that are sent to destinations **other than** localhost.

[2] Many web servers now use the HTTPS protocol (port 443) in addition to port 80. You may need to block both ports for some exercises.

[3] It is important that there is a space between the "!" and the "-" characters. You should also not use the computer's numeric keypad to enter the "-" character, as this can cause problems..

Exercise 6

1. Make sure you have deleted the rule from **Exercise 5** using the command: `sudo iptables -L`.
2. Add a single rule that allows **only outgoing traffic** to **port 80** destined for http://www.upv.es/, and drops all other traffic to port 80. To specify the port number in a rule, you must also specify the **transport protocol** using the -p option. In our case, use -p tcp. We also have to add the same rule for port 443 to cover https as well.
3. Add another rule to allow **outgoing SSH traffic** only to zoltar.redes.upv.es
4. In a terminal window, try to connect via SSH to zoltar.redes.upv.es. A response from the destination is enough; you don't need to complete the login. Then, from another terminal window, try to SSH into a classmate's computer (e.g., rdcXX.redes.upv.es). Also check that you still have access to the UPV website. With these three steps, you can confirm whether your rule is working as intended.
5. Remove the rule related to SSH traffic using the -D option.
6. Now add these two new rules:

```
sudo iptables -A OUTPUT -p tcp ! -d zoltar.redes.upv.es -j DROP
sudo iptables -A OUTPUT -p tcp -d www.upv.es --dport 80 -j ACCEPT
```

7. Try reloading the www.upv.es page in your browser. Does it work? Use the command: `sudo iptables -L` to view the current rule list and try to **explain the meaning of the three rules** that appear. Consider how **the order** of the rules affects their application. How should they be modified to allow access to www.upv.es ? We have to change the order of the 2 added rules in point 6 of this exercise
8. Let's verify it. Delete the first rule from step 6:

```
sudo iptables -D OUTPUT -p tcp ! -d zoltar.redes.upv.es -j DROP.
```

   Now add it again at the **end** of the chain:

```
sudo iptables -A OUTPUT -p tcp ! -d zoltar.redes.upv.es -j DROP
```

9. Type `sudo iptables -L` to view the current state of the rule list.
10. Check that you've now regained access to the http://www.upv.es server. As you can see, the **order** in which rules are processed is extremely important.
11. Now, clear **all rules** using the command: `sudo iptables -F`

Event Logging

The functionality of iptables not only allows you to specify rules to accept or drop packets (as we've seen in several examples). In addition to these functions, rules can trigger **logging actions** that are recorded in the **system log** (in the /var/log/messages file on the lab machines).

Note: To clear the event log, you can run the command: **`sudo dmesg -C`**

To create rules that generate a log entry when a packet matches the rule, use the -j LOG option. Rules with the LOG action **do not make any decision** about the packet being processed—they neither accept nor reject it. Therefore, when such a rule is triggered, **it does not stop** the processing of the remaining rules in the current chain.

The interest in logging certain network traffic events depends on the context. It can provide the administrator with useful information that may or may not be available elsewhere. For example, if you want to know how many users connect to your SSH server each day, it's likely that the SSH server software already maintains detailed logs. However, a minimal server may not log anything at all. Let's suppose we are in this second case, and have been asked to determine how many clients connect to the server each day.

The first thing we need to do is define what condition qualifies as a new client connection. The simplest (though not necessarily the most accurate) is to consider each new connection to port 22 of the server as a new client.

Exercise 7

1. Check if an SSH server is running on your device by attempting a local connection: **`ssh localhost`**.
2. Clear the event log using: **`sudo dmesg -C`**
3. Create a rule that logs SSH traffic:

   ```
   sudo iptables -A OUTPUT -p tcp --dport 22 -j LOG
   ```

4. Next, try to access the local SSH server using: **ssh 127.0.0.1**. You don't need to complete the login—interrupt it with **Ctrl+C**
5. Now run **sudo dmesg** and review the latest lines. Look for one where the **SYN flag** is set. What do you see? Do you understand what it means?

The rule from Exercise 7 logs every packet destined to the SSH server. However, this approach has a serious downside: the rule matches **every SSH segment**, so a single client session could generate **thousands of log entries**. Filling up log files with low-value data is a bad idea.

To log more intelligently, we should log only the **first segment** of a connection (i.e., the one with the **SYN flag** set).

Exercise 8

1. First, remove the previous rule:

   ```
   sudo iptables -D OUTPUT -p tcp --dport 22 -j LOG
   ```

2. Now add a rule that specifically logs new connection attempts:

   ```
   sudo iptables -A OUTPUT -p tcp --dport 22 --tcp-flags ALL SYN -j LOG
   ```

3. Clear the log again with: **sudo dmesg -C**. Now, if you try connecting again to the local SSH server and check the log using sudo dmesg, you'll see that **only one message** is logged per connection attempt.

It may look like we repeated Exercise 7, but this time we're paying attention to the **TCP flags field**. All bits are checked (that's why we use ALL), and only segments **with the SYN flag active** and targeting port 22 are logged.

Let's now add a rule to log all outgoing **web traffic** from users of this machine. Specifically, we want to log **TCP segments** directed to **ports 80 and 443** that have the SYN flag active.

Exercise 9

1. Add a new rule that logs **outgoing connections to any web server**. Type (note: this is a single command split over two lines):

   ```
   sudo iptables -A OUTPUT -p tcp --match multiport
     --dports 80,443 --tcp-flags ALL SYN -j LOG
   ```

   Note that we've used the multiport module to filter for **multiple destination ports**. This rule will log all connections to any HTTP or HTTPS server made from this computer.
2. Clear the log and then access several websites.
3. Use **sudo dmesg** to check that the web page accesses have been logged. Look for lines with DPT=80 or DPT=443. Did you get more than one message per website? If so, why do you think that happened?
4. Use the following command to view the active rules and how many times they've matched: **sudo iptables -L -v** In this output, the **pkts** column (under "Chain") tells you how many times each rule has been triggered. You can also see the number of **received** (INPUT), **sent** (OUTPUT), and **forwarded** (FORWARD) packets and bytes.
5. Remove the rule you added in this exercise.

But iptables functionality doesn't stop here. The following **optional exercises** introduce even more advanced capabilities of this powerful tool.

Optional Exercises

Modifying Destination IPs and/or Ports: The nat Table

Although due to time constraints we won't cover all the possibilities of iptables, we would like to illustrate some of its features with practical examples. In the following exercise, we'll create a rule to modify the **destination address** of all matching TCP segments.

Exercise 10

1. Open your browser and visit the page: http://www.redes.upv.es
2. Now enter the following rule (the two lines are one single command):

```
sudo iptables -t nat -A OUTPUT -p tcp -d 158.42.0.0/16 --dport 80 -j DNAT
--to 158.42.4.23:80
```

3. Reload the website from step 1. **The prefix http:// is important** to ensure port 80 is used. What happens?
4. Try accessing www.upv.es

Let's review the command. Unlike previous cases where the -j option was followed by LOG or DROP, here it is DNAT. This rule allows rewriting the **destination IP address and/or port** of a segment. In this case, **all TCP segments** headed to **web servers** within the campus network (158.42.0.0/16) are redirected to the web server at 158.42.4.23.

You may notice that this rule specifies a new table: -t nat, whereas previously we used the **default table** (-t filter), which didn't need to be explicitly mentioned. The nat table enables operations like **network address and port translation (NAPT)**, similar to the ones performed by cable or ADSL routers.

We will not go into detail on how to build a full set of rules to implement this functionality.

Exercise 11

1. Use the following command to display the current ruleset: sudo iptables **-L** Can you see the rule you added in Exercise 10? Why not?
2. Try again with: `sudo iptables -t nat -L`
3. Now remove the rule(s) from the nat table.

The -L option shows the contents of a table. By default, it displays the filter table. To view another table, use the -t option, as in -t nat.

Exercise 12

Now let's try a slightly more complex exercise.

1. Think about which rules you'd need to redirect connections **to any UPV web server**, except for www.upv.es, to IP address 193.145.235.30 (the web server for www.ua.es).
2. Check that the redirection works correctly by connecting to:
    - www.redes.upv.es and www.cfp.upv.es. If your redirection is set up correctly, you should be shown the www.ua.es homepage.
    - www.upv.es Which should display the usual UPV web server.
3. Remove the rules you added using: `sudo iptables -t nat -F`

Changing Other Fields

It's also possible to modify the values of other fields in network traffic for different purposes. One example is the **Type of Service (TOS)** field in the IP header. You can set a value that best suits a specific application. Even if the value is ignored beyond your system, it may help **differentiate simultaneous traffic flows**.

Let's look at a simpler example: the **Time to Live (TTL)** field in the IP header. This value is usually preset by your operating system, and most admins never modify it.

Suppose we want to change the **TTL** value for certain types of traffic (but not for all packets). This can be done with a selective rule in iptables.

Exercise 13

1. How can you find out the **TTL value** your system uses when sending traffic?
2. Now add this rule:

    ```
    sudo iptables -t mangle -A POSTROUTING -j TTL --ttl-set 5
    ```

3. Check whether you can access the following websites: www.upv.es, www.etsit.upv.es and www.uji.es
   If you use the ping command to test access, pay close attention to the response messages. Look closely at the IP address shown in the **first response line**, and the IPs shown in subsequent replies.

The issue is that with a TTL value of only 5, the datagram can only make **four hops** before being discarded. This greatly limits the number of reachable networks. A **TTL=1** would prevent crossing even a single router, restricting communication to **only hosts within the same subnet**.