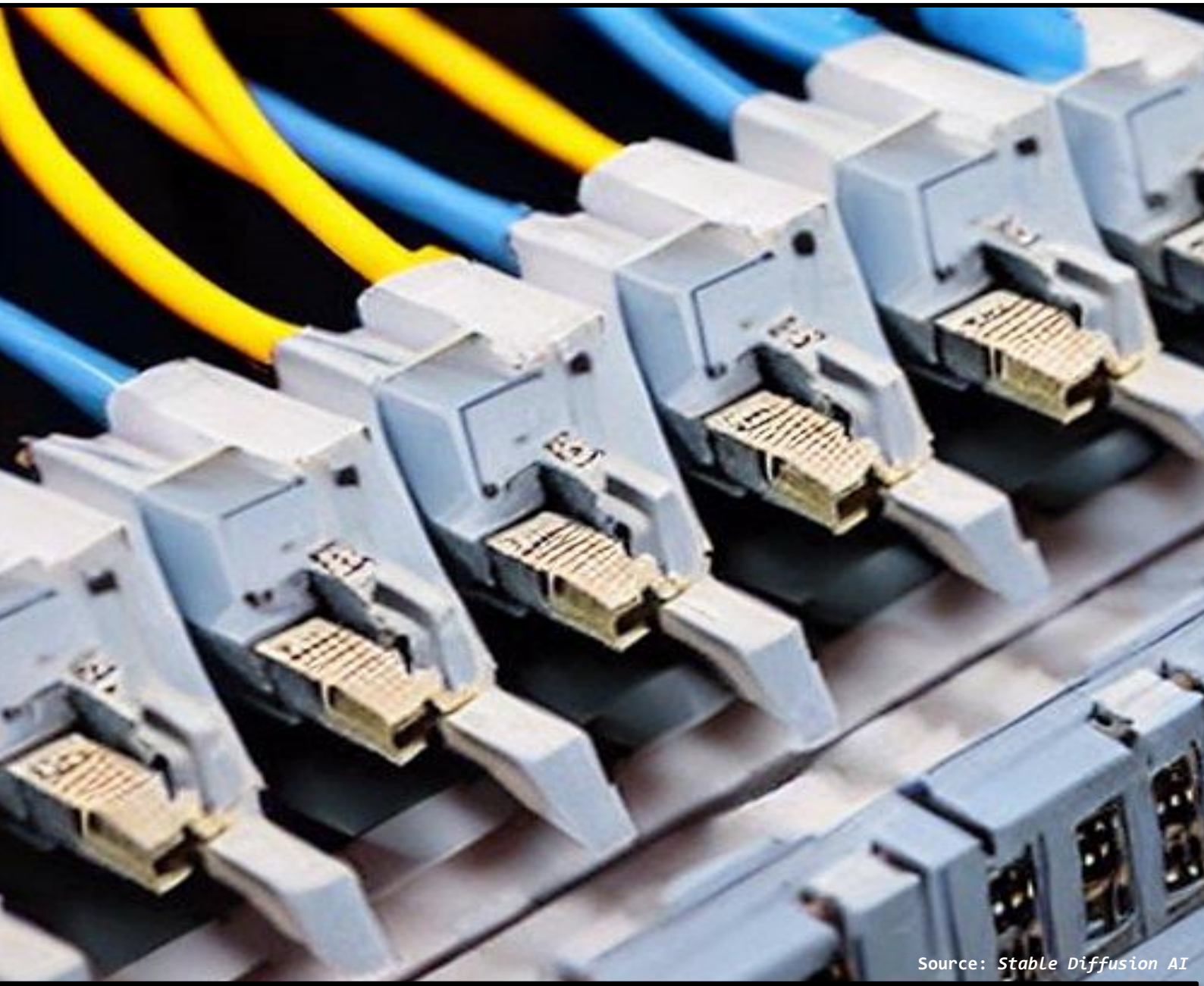


Source: *Stable Diffusion AI*

LABORATORY 7. NETWORK SECURITY

DEPARTMENT OF COMPUTER SCIENCE. UNIVERSITY OF OVIEDO

Computer Security | 2022 – 2023 (V2.7 *"S-81 Isaac Peral"*)

CONTENT

THE INFRASTRUCTURE OF THIS LABORATORY

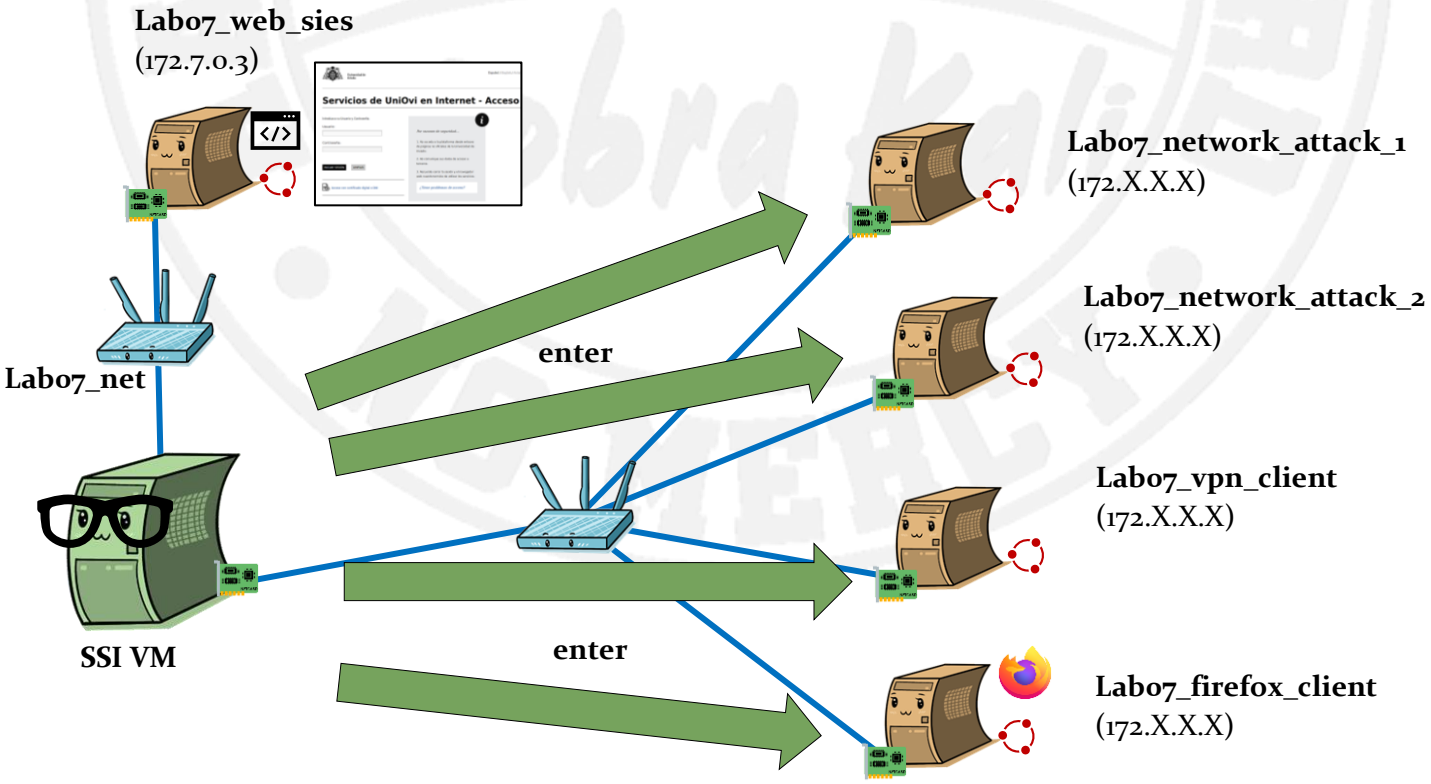
*(**NOTE**: To avoid problems, we strongly encourage you to run this laboratory infrastructure **on the base VM instead on its security hardened versions** that you may have generated from previous labs. In real life you should do the opposite, but real deployments also have a validation time to ensure that everything is working as expected after hardening)*

***WARNING**: Some of you have experienced problems running Firefox from a container, which is something that we will use in this lab. To avoid these problems, follow these steps:*

- *Open the file* `build_lab07.sh`
- *Add to this file the following line:* `xhost + local:root`
- *Open the folder that was inside the zip file of the labs and locate the file* `\images\ubuntus\ubuntu_base_cmd_firefox\Dockerfile`
- *Add the line* `RUN apt-get update` *just below the line* `USER root` *and save this file*
- *Do the same in the file* `\images\ubuntus\ubuntu_ssh_vpn_client\Dockerfile`
- *Build the lab as usual*

The infrastructure of this laboratory is composed by five containers communicated via internal networks with your VM. Most are expected to be used interactively.

- Two containers are "network attack" containers and have the tools to test the firewall and other protection software (`nmap`, ...) we are going to see in this laboratory. Data written in the `/etc/sysctl.d` directory is mapped in the `volume_data/network_data_<N>/sysctl.d/` directory in the host.
- The third one is designed to function as a client for the *Wireguard* VPN. Data written in the `/etc/wireguard` directory is mapped in the `volume_data/vpn_client/wireguard` directory in the host.
- A fourth container is enabled to run GUI applications and has *Firefox* installed.
- Finally, the last container is a web server with the Uniovi intranet front-end page communicated with the host VM with a special private network (172.7.0.0/16) and a fixed IP.



*(**NOTE**: Some of the operations of this laboratory require to know the IP address of the containers and the current network they are using (e.g., `172.22.0.0/16`). Depending on your setup, the network IP range used may change. Please, remember to use `ifconfig` inside the containers to know your current IP address, as these are assigned via DHCP)*

These containers have the software necessary installed to perform the activities of the laboratory. However, several of the products we are about to use are incompatible with containers and, therefore, they **must be installed into your Ubuntu VM**. In this case, the containers will function as "attacking clients" of the VM, so you can test that they work without needing more external elements. Which software is aimed to containers, and which is to be installed on the VMs is clearly indicated in each activity.

BLOCK 1: HOST NETWORKING PROTECTION

Host-based Packet-Filtering firewall: Managing *Firewalld* zones

Practical application: *You need to separate your network interfaces in zones with firewalld so you can assign them different configurations*

This activity will teach you how to use `firewalld` to perform typical network-related security operations that illustrate parts of the SNI (*Secure Network Infrastructure*) of the theory topics. `firewalld` is a more powerful product that the traditional `ufw` that comes with every *Ubuntu* distribution, is available both in *Debian*-based and in *RedHat*-based *Linux* distros, and you also use it in ASR. However, having both enabled at the same time causes problems, so first we must ensure that `ufw` is disabled: `sudo ufw disable`.

Once this is done, we need to install the appropriate software to use and manage `firewalld`, consisting in the own firewall service and its GUI (`firewall-config`). You can install both like this: `sudo apt install firewalld firewall-config`. After this is done, **it is important to reboot the VM.**

Once the VM has rebooted we will be able to work with `firewalld`. The first thing we must understand of this firewall is that it distributes the different network devices we have in the VM in **zones**. Zones are predefined sets of rules specifying what traffic should be allowed based on the level of trust on the networks your computer is connected to. You can assign network interfaces to a zone. Below are the zones provided by `firewalld` ordered according to the trust level of the zone from untrusted to trusted:

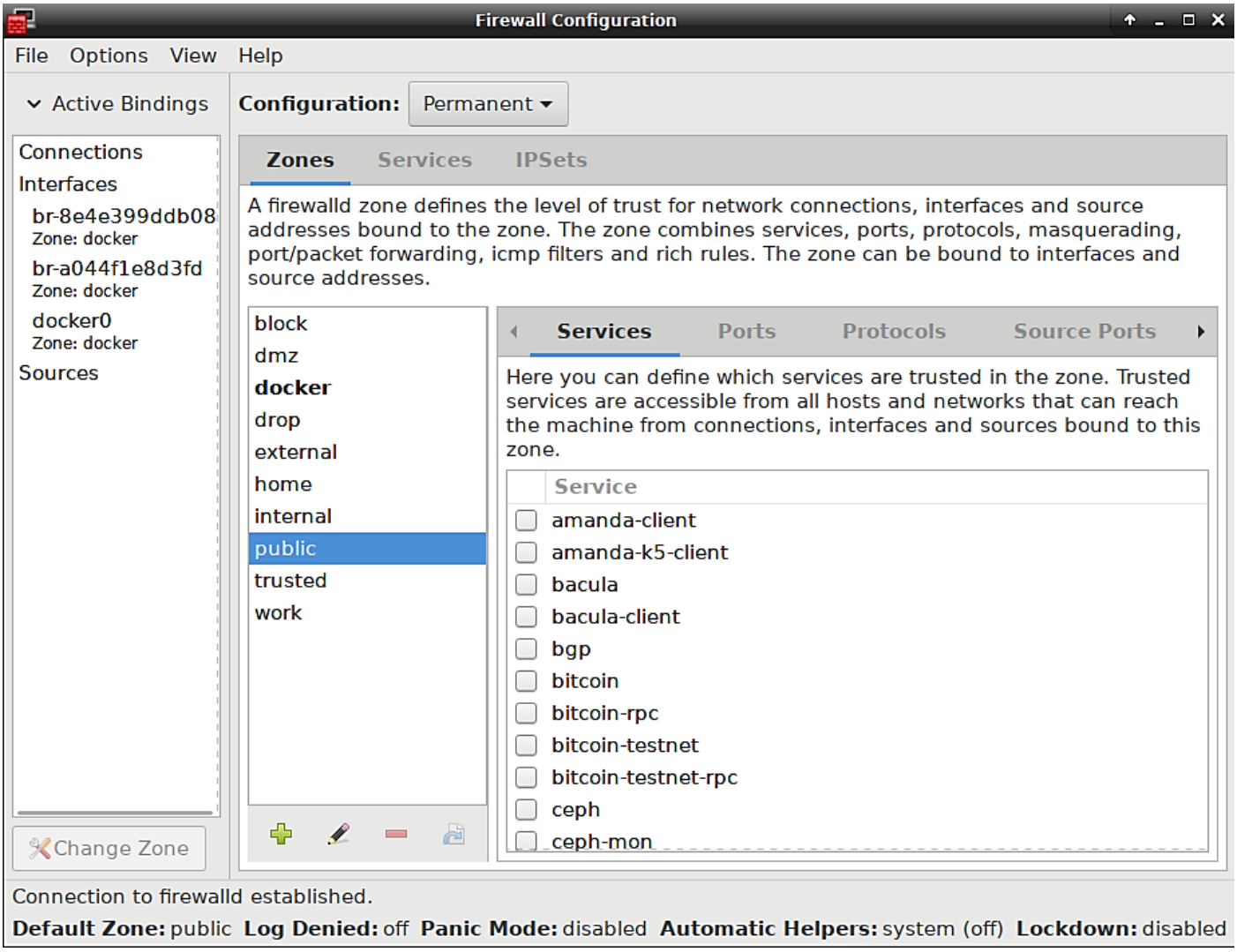
- **drop**: All incoming connections are dropped without any notification. Only outgoing connections are allowed.
- **block**: All incoming connections are rejected. Only outgoing connections are allowed.
- **public**: For use in untrusted public areas. You do not trust other computers on the network, but you can allow selected incoming connections.
- **external**: For use on external networks with NAT masquerading enabled when your system acts as a gateway or router. Only selected incoming connections are allowed.
- **internal**: For use on internal networks when your system acts as a gateway or router. Other systems on the network are trusted. Only selected incoming connections are allowed.
- **dmz**: Used for computers located in your demilitarized zone that will have limited access to the rest of your network. Only selected incoming connections are allowed.
- **work**: Used for work machines. Other computers on the network are generally trusted. Only selected incoming connections are allowed.
- **home**: Used for home machines. Other computers on the network are generally trusted. Only selected incoming connections are allowed.
- **trusted**: All network connections are accepted. Trust all the computers in the network.
- **docker**: A special zone that initially contains all the networks created by our *Docker* containers.

You do not have to understand all these zones, but you need to learn how to change interfaces between zones. The good part of using zones is that once you enable a service or port for a zone, **you enable it for all the interfaces belonging to that zone at the same time**!

We are going to use the *Firewall Config* GUI program to manage most of the firewall operations, but you can also do the same things via command line:

- https://computingforgeeks.com/install-and-use-firewalld-on-ubuntu/
- https://www.supportsages.com/everything-you-need-to-know-about-firewalld/

To do that, first run `sudo firewall-config` and change the **Configuration** *ComboBox* to *Permanent*, as shown in this image:



Unfortunately, most of you will find that the network cards `eth0` and `eth1` are not assigned to any zone and they do not appear in the GUI. We can fix this using the command line, making these network cards appear, and being able to manipulate them using the GUI from now on. Do the following operations to allow that:

- Add `eth0` (the card that allows the VM to connect to internet) to the "*public*" zone: `sudo firewall-cmd --zone=public --change-interface=eth0`
- Add `eth1` (the card that allows the VM to connect to your PC) to the "*work*" zone: `sudo firewall-cmd --zone=work --change-interface=eth1`

Once you do that, make sure that the `work` zone has the `ssh` service enabled and test that `ssh` is working from your PC

Expected results: This activity will be finished when you are able to assign zones to network cards in the firewall and activate and test that a service can be accessed in the "work" zone.

Host-based Packet-Filtering firewall: Blocking IPs and networks

Practical application: *You need to block individual IPs or entire networks from access to your machine*

One of the most common uses of a firewall is allowing/blocking access from certain IPs and networks to other IPs and networks. To practice how to do this, follow this procedure:

- Ensure that you can access from your PC to the VM via `ssh`
- Go to the "`work`" zone in the firewall GUI and search the "*Rich rules*" tab.
- **Create a new rich rule** that **drop** all connections to the `ssh` service from the IP `192.168.56.1` (normally the IP of your PC in the host-only network) to the IP `192.168.56.10` (the IP of your VM in the host-only network). Check all options that the rule creation has, and think about its possibilities, especially the *limit*, *log*, and *audit* options.
- Test that now you cannot connect via `ssh` anymore
- Modify the previous rule to block the entire `192.168.56.0/24` network
- Test that you still cannot connect via `ssh`
- Remove the rich rule and test that now you can connect again.

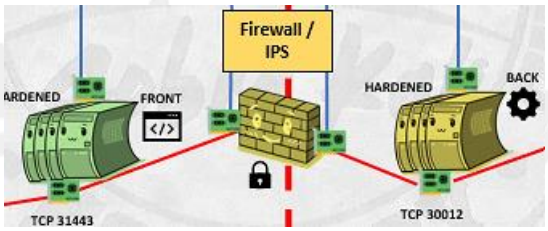
NOTE: Some of you could not install an `eth1` interface (especially in the lab class), so the "`work`" zone is not available. You may test the same feature (block a concrete IP from using a particular service) at home (most of you did not report problems with `eth1` at home) or using other zones. For example, in the next exercise you can temporally drop access to `ssh` from a machine in the `public` zone (*Firefox* client) or its entire network to the VM.

Expected results: This activity will be finished when you are able to block an IP or network to access a service in a firewall zone.

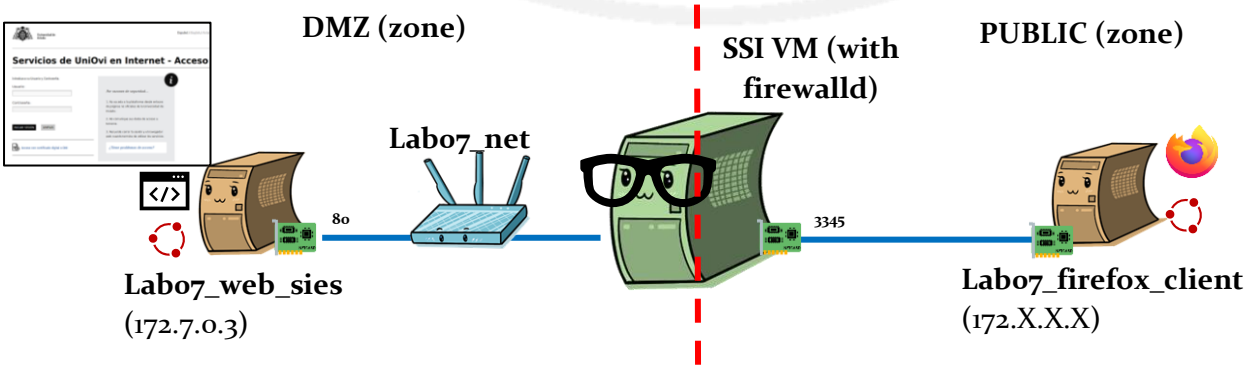
Port forwarding in *Firewalld*

Practical application: *You need to hide a service IP and port redirecting its access through a firewall*

The goal of this exercise is to emulate the following part of the SNI:



We have two machines situated in different network zones, and therefore unable to communicate directly. Between both zones there is a firewall that can connect to both, and we can instruct the firewall to redirect a connection made to one of its ports on one of the zones to another machine placed in a different zone: this is called **port forwarding**. The machine in one of the zones will not know who is the machine that serves their request in the other zone, and the serving machine will never have direct contact with their clients: the firewall isolated both. To emulate this, we are going to use the following elements of our **Lab 7 infrastructure** to emulate this behavior (see diagram):

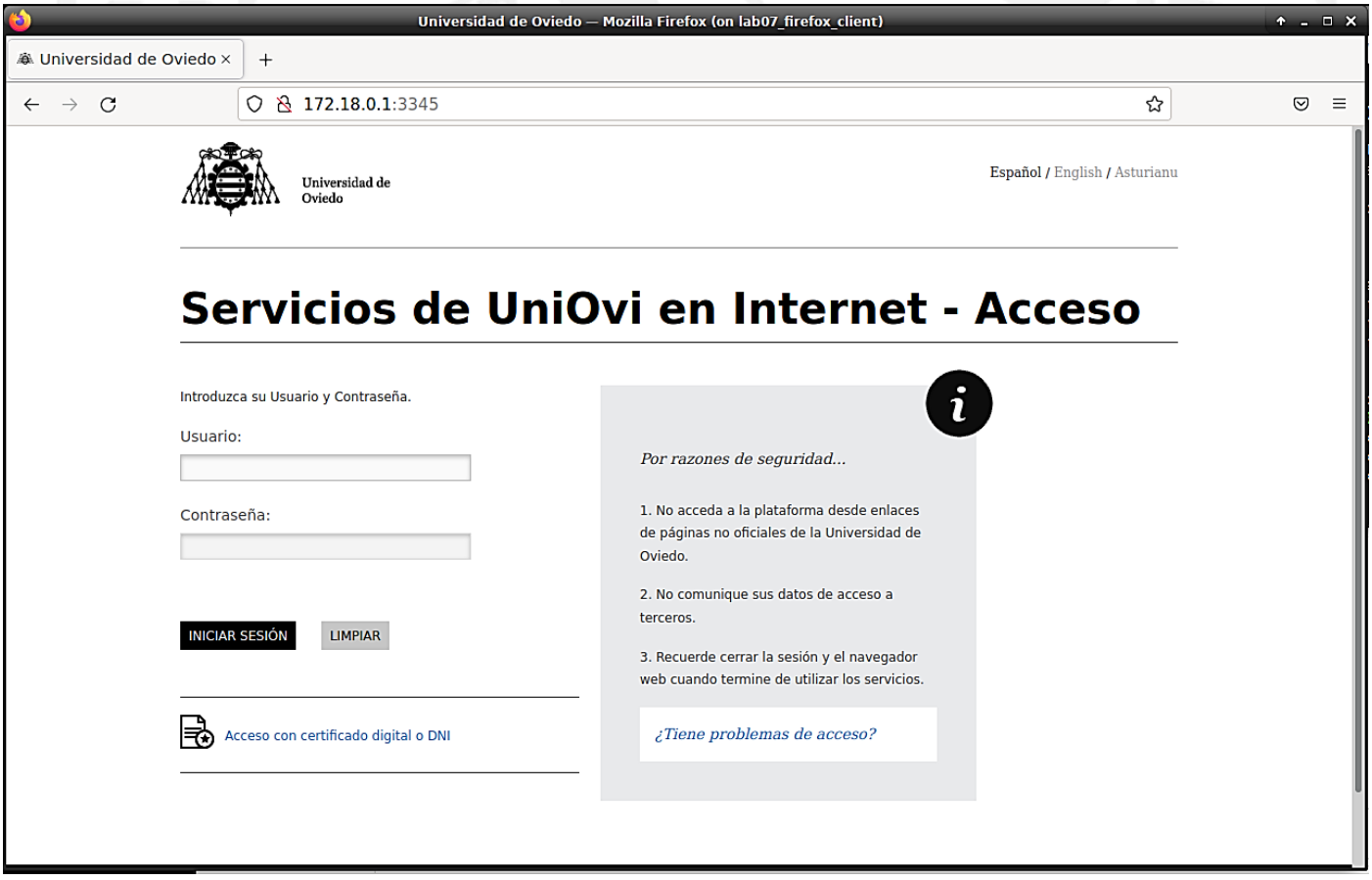


- Enter the `lab07_firefox_client` container and annotate its IP (`ifconfig`)
- Go to the VM and locate the network card that the *Firefox* container uses looking to the one that has a compatible IP with the one you annotated in the first step. Its name should be something like `br-5c8ea19b05fb`
- Using the firewall GUI, put this network card in the "*public*" zone.
- Now, locate the network card serving the `lab07_web_sies` container (the one that has an IP in the `172.7.0.0/16` network).
- Put this network card in the "*dmz*" zone.
- **Reload the firewall (*Options – Reload firewall*).**

Now you have set up two containers in different zones and the firewall (your VM) in between. To be able to communicate them using port forwarding, do the following:

- Go to the "*public*" zone in the GUI and search the "*Port forwarding*" tab (press the right arrow to see more tabs)
- Create a new port forward rule that redirect all connections to the port `3345` to the port `80` of the IP `172.7.0.3` (the `lab07_web_sies` container). If the firewall asks you to enable masquerading, answer *Yes*.
- **Reload the firewall (*Options – Reload firewall*).**

To test this, you can run *Firefox* in the `lab07_firefox_client` container bash command line. The first time will likely crash**, but if you restart it, it will work**. Now you have a container-isolated *Firefox* in the "public" zone. Try to browse to the VM IP in this zone (if the container IP is `172.22.0.2`, the VM IP in this network uses the same network, but ends in `1`) but to the port `3345` (Example: `172.22.0.1:3345`). If everything is right, you should see this webpage:



With this, you have effectively connected a machine in a network with a machine into another network through a firewall. Think about what this means regarding security (who can access the second machine, what services

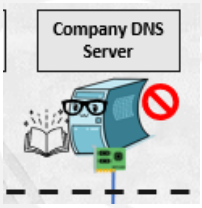
are being exposed…) and how could you replicate this to connect an application front-end with its backend if both are placed in different network zones.

Expected results: This activity will be finished when you are able to communicate a container in the public zone with a container in the DMZ zone so you can access a web front-end placed in the DMZ from the public zone using port forwarding.

Protecting outgoing connections: DNS filtering with *PiHole*

Practical application: *You need block connections to known malicious domains*

We strongly recommend you to create a snapshot of your VM before doing this exercise. The goal of this exercise is to show you the importance of controlling your own DNS server to filter potentially malicious outgoing traffic. This corresponds to this part of the theory *Secure Network Infrastructure* (SNI), but you can also do this at home 😊.



We will do this setting up a professional product, *PiHole* (https://pi-hole.net/), that allows you to have a DNS server that also filter any request it receives using a **blocklist** approach: any connection performed to an IP present in the blocklist will be aborted. *PiHole* allows installation on many systems, but for this lab we have pre-packaged it in a container. Go to the `pihole` folder of the lab files and execute the `run_pihole.sh` script (please, ensure that *Apache2* is not installed in the VM, and `sudo apt-get remove apache2` it if it is). Once it finishes, browse to the `127.0.0.1/admin` URL in your VM browser to see the *PiHole* welcome statistics screen. The admin password is `test123...`

We recommend you to browse to any ad-ridden web page (such as www.elmundo.es) and leave that web open. Now that *PiHole* is working, to start filtering connections we only must change the DNS of our `eth0` network card (the Internet-facing one) to `127.0.0.1` (*PiHole* runs locally). Do not worry, you will not lose Internet connection, *PiHole* knows what to do 😊. The following image shows the changes you need to do for this:

```
 GNU nano 2.9.3              /etc/netplan/01-netcfg.yaml              Modified

1 network:
2   version: 2
3   ethernets:
4     eth0:
5       dhcp4: true
6       nameservers:
7                 addresses: [127.0.0.1]
```

Reload now the previous website. Do you see any difference? What will happen now to any device that uses this DNS server?

Expected results: This activity will be finished when you are able to set up the *PiHole* DNS filtering server to test how your browsing changes once you use its features.

PiHole outbound connections advanced filtering: getting rid of lots of malicious URLs

Practical application: *You need block connections to even more known malicious domains*

Although *PiHole* filters a great number of malicious websites by default, we can give it more power using the malicious URLs collected by *The Block List Project* (https://blocklistproject.github.io/Lists/). Follow these instructions to incorporate this IP list to *PiHole*:

- Copy the link of the most complete list (*Everything*) in the *PiHole* format (https://blocklistproject.github.io/Lists/everything.txt).
- Add the URL to your *PiHole's* block lists (*Login - Group Management -Adlists -Paste list URL in the "Address" field, add comment - Click "Add"*)
- Update *Gravity* (*Tools-Update Gravity-Click "Update")* to incorporate the new URLs to the blocklist. The process takes a while, **but please wait and do not browse outside this page!**

Browse again with the *PiHole* statistics webpage open and see how the much bigger blocklists take effect now.

Expected results: This activity will be finished when you are able to improve the PiHole blocklists to block connections to more types of malicious websites.

BLOCK 2. CONNECTION PROTECTION

Fail2Ban

Practical application: *You need block connections to machines that repeatedly try to unsuccessfully access your SSH service*

This is an application that processes system logs to search signs of automatic attacks. When these are detected, according to the rules it defines, `fail2ban` adds a new rule to the firewall blocking the identified attack attempt. This rule blocks the attacker's IP either temporarily or permanently, depending on the configuration. You can also be notified via email that an attack is underway (by installing `sendmail`, but we will not use this in this lab). `fail2ban` is typically used **to protect** `ssh` but has several predefined rules by default (named **jails) that also work with other services, like *Apache* or *Nginx* (web servers).** Any service using logs can be technically protected by `fail2ban` provided you have the appropriate rules. To do that, you can create your own rules, but this exceeds the purpose of this laboratory. This program does not work inside containers, so you must **install it in your VM**.

Protecting SSH using Fail2Ban

Using `fail2ban` requires the following steps:

- Install it (the service automatically starts once installed): `sudo apt install fail2ban`
- Ensure that the services to be protected via `fail2ban` (SSH, HTTP, …) are not blocked by the firewall (nothing to protect then! 😊).
- Modify `fail2ban` configuration (`/etc/fail2ban/fail2ban.conf` file). It is **HIGHLY** recommended to copy this file to `fail2ban.local` and edit this new file instead of the original `.conf` one. In this file you can configure things like log details (`loglevel`), where to write log info (`logtarget`), etc. We will use the default configuration in this laboratory.
- Modify the `/etc/fail2ban/jail.local` file to enable or disable jails. To do that, consider the following:
  - If the `jail.local` file does not exist, create it copying `jail.conf`
  - **At the beginning, only the `sshd` jail is active by default**. The rest must be activated manually by adding `enabled = true` after each jail name you want to activate (the one between `[]`).
  - There are multiple other commands and parameters (like IPs to ignore), but we are not using them in this laboratory as we aim for a simple (but effective) `fail2ban` deployment.
  - When a jail is activated, the `bantime`, `findtime` and `maxretry` parameters define how an IP is disallowed. A negative `bantime` means a **permanent block**. **A host is blocked by a jail** if it activates its conditions successfully more than `maxretry` times during `findtime` seconds.
  - Available jails are defined in a section of this file. As we said, `sshd` is enabled by default, while the others must be activated manually. Of course, this is only recommended if we really have the service running.

```
[sshd]

# To use more aggressive sshd modes set filter parameter "mode" in jail.local:
# normal (default), ddos, extra or aggressive (combines all).
# See "tests/files/logs/sshd" or "filter.d/sshd.conf" for usage example and det$
#mode   = normal
port    = ssh
logpath = %(sshd_log)s
backend = %(sshd_backend)s


[dropbear]

port     = ssh
logpath  = %(dropbear_log)s
backend  = %(dropbear_backend)s


[selinux-ssh]

port     = ssh
logpath  = %(auditd_log)s


#
# HTTP servers
#

[apache-auth]

port     = http,https
logpath  = %(apache_error_log)s


[apache-badbots]
# Ban hosts which agent identifies spammer robots crawling the web
# for email addresses. The mail outputs are buffered.
port     = http,https
```

- Each jail behavior is defined in a different file in the `/etc/fail2ban/filter.d` directory. Each of these files contains **the regular expressions that**, when matching with a log entry, fires the rule. They also define operation modes and other parameters. It is not necessary to examine them in this laboratory, this is just for your information 😊.
- For this lab we will **enable the aggressive mode for `sshd` to ensure the maximum SSH protection mode**.

Checking Fail2Ban Status

`fail2ban-client` can be used to manage `fail2ban` through the following options:

- `start`: starts `fail2ban`
- `reload`: Reloads `fail2ban` configuration
- `reload JAIL`: Replace the current configuration of the specified jail for the one placed in the file you pass
- `stop`: Stops `fail2ban`
- `status`: show the current `fail2ban` status and active jails (including jail names)
- `status JAIL`: show the status of the specified jail. It is a particularly important option as it also shows how many times it has been activated and the currently banned IPs.

Every time you activate new jails in the previous file **you need to reload** `fail2ban` and check how many jails are active.

Testing Fail2Ban

Testing `fail2ban` with SSH is quite easy: you only have to do several login attempts with incorrect passwords inside the time you have configured. It is recommended to do that from one of the containers in the **Lab 7 infrastructure**. Once you notice that you have been banned, you can use the previous commands to check that the IP appears as banned.

To unban the IP, you can wait the specified time (if it is not infinite! 😊) or run: `fail2ban-client set <jail name> unbanip <ip>`

Expected results: This activity will be finished when you are able to protect `ssh` using the aggressive protection mode of `fail2ban`. You are also able to check that the protection works and undo the IP blocks created by this program.

Honeypots: *PortSentry*

Practical application: *You need block connections to attackers trying to scan you*

Nmap port scans will be the subject of the next laboratory, and one of the most common operations in any pentesting activity. In this lab we will see how to stop them using this tool. *Portsentry* can detect port scans and block these attempts, so the scanner machine obtains no information.

There are multiple ways to use `portsentry`, but we are using one of the most interesting ones: to combine it with the firewall we deployed in a previous activity to emulate the behavior of a *Honeypot*. To do this, we will **leave some ports open on purpose in the firewall**, even if there is no service listening to them. Scan attempts over these "fake open" ports will be detected and the tool can proceed to ban the scanning machine. `portsentry` cannot listen to ports that have actual services listening to them, as they "occupy" the ports. To do that, follow these steps:

- Install the *PortSentry* service in your *Ubuntu* VM (this is not compatible with containers): `sudo apt install portsentry`
- Check that it is running: `sudo service portsentry status`
- Understand the two **working modes** of the tool (values of the `TCP_MODE` and `UDP_MODE` parameters in the `/etc/default/portsentry` file)
    - **Basic mode** (values: `"tcp"`, `"udp"`): It listens to a static list of predefined ports in the configuration file. We will **not** use this mode as the presence of the tool can be easily detected.
    - **Advanced Stealth mode** (values: `"atcp"`, `"audp"`): Uses a raw socket to detect scan attempts so the tool cannot be detected: the ports appear to have the "normal" service expected there, but `portsentry` is listening and detecting suspicious activity.
- Understand the **possible answers** to a scan attempt (values of `BLOCK_UDP` and `BLOCK_TCP` in the `/etc/portsentry/portsentry.conf` file)
    - Just **log** suspicious traffic in **/var/log/syslog** (value: `"0"`) **(default behavior)**
    - **Block** the offending client (value: `"1"`)
    - **Run a program** as an answer (value: `"2"`). This gives us a lot of flexibility and enables "wild" answers to scan attempts 😄, but we will not use it here.

Log and block scan attempts

The first thing to do to enable `portsentry` is to enable the following services in the `docker` firewall zone: SSH, FTP, Telnet. This way, we have the "bait" to lure scan attempts to our server.

Once this is done, open the `/etc/default/portsentry` file and put `TCP_MODE` to `"atcp"` and `UDP_MODE` to `"audp"` to enable the *advanced stealth* mode. Restart `portsentry` and perform a simple *Nmap* scan from any of the **Lab 7 infrastructure** containers to the host machine. Once the scan finishes, check the contents of `/var/log/syslog` to see if the scan attempts have been detected.

Successful log of scan attempts is the condition to finish configuring `portsentry` usage as intended: Open `/etc/portsentry/portsentry.conf` and change the values of the `BLOCK_UDP` and `BLOCK_TCP` parameters to `"1"`. Restart `portsentry` and perform the same scan again.

Revert blocks

To revert a block, we need to:

- Remove the IP of the blocked machine from `/etc/hosts.deny`. Being in this file prevents any communication from this IP to our machine.

```
  GNU nano 2.9.3                    /etc/hosts.deny

# /etc/hosts.deny: list of hosts that are _not_ allowed to access the system.
#                  See the manual pages hosts_access(5) and hosts_options(5).
#
# Example:    ALL: some.host.name, .some.domain
#             ALL EXCEPT in.fingerd: other.host.name, .other.domain
#
# If you're going to protect the portmapper use the name "rpcbind" for the
# daemon name. See rpcbind(8) and rpc.mountd(8) for further information.
#
# The PARANOID wildcard matches any host whose name does not match its
# address.
#
# You may wish to enable this to ensure any programs that don't
# validate looked up hostnames still leave understandable logs. In past
# versions of Debian this has been the default.
# ALL: PARANOID

ALL: 192.168.8.100 : DENY
```

- Remove the rule of the firewall that rejects (`REJECT`) every traffic from the offending client: `sudo route del -host <IP of the offending client> reject`
- We can use `netstat -rn` to check that the rule has been removed.

Expected results: This activity will be finished when you are able to protect a host against `nmap` scan attempts with `portsentry` and understand the behavior of the blocking it performs, all without interfering with the machine firewall. You are also able to check that the protection works and undo the IP blocks created by this program.

BLOCK 3. EXTREMING SECURITY IN CONNECTIONS

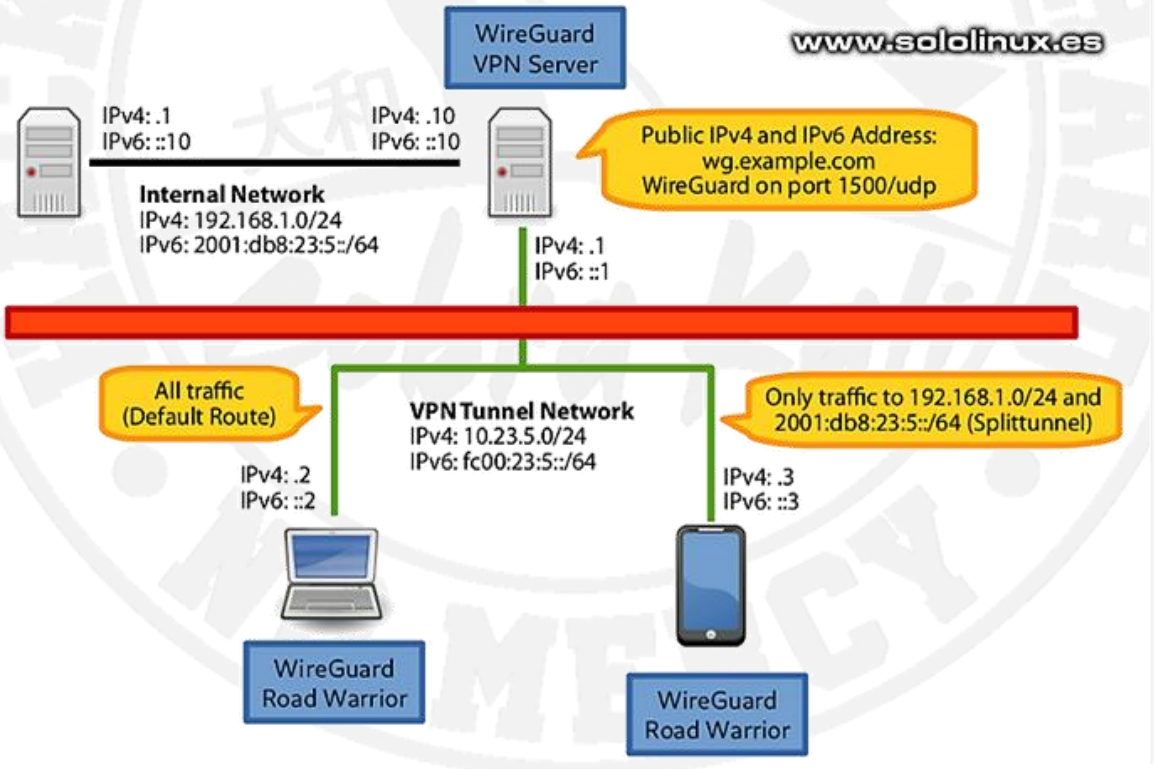
WireGuard VPN

Practical application: *You need to setup your own VPN server*

An VPN is a private network that **guarantees end-to-end private communication ("tunnel") between a client and a server even through public networks**. *WireGuard* is one of the best (and free) VPN implementations available that we can use to access any server with a public IP from any part of the world securely. If our server only accepts correctly configured VPN connections, it will be invisible for the rest of the world. This puts a client device inside an internal network we define, accessing all this internal network resources as if it were inside a company/home LAN.

This activity has been included because this is one of the suitable solutions for secure remote work. As it is a special activity that may get complicated if something fails, its explanation has been detailed to be easily followed. This will also guide you in case you want to create an infrastructure like this on a real setup. It is no secret that we also included this to help you in the future in case of "pandemic scenarios".

The following image represents how this setup works: our clients (a mobile and a laptop) can connect to the internal network and obtain an IP inside of it thanks to a special network interface in a VPN server. VPNs are possible thanks to **public key cryptography**, so this is also a practical application of the corresponding topics of the course. This means that each member of the VPN (the server and each client) will need to generate **a public/private key pair** to be able to use this.



VPN Server Setup

Our Ubuntu VM will function as a VPN Server for the purpose of this laboratory. We can install *WireGuard* like any other typical *Ubuntu* software: `sudo apt install wireguard`. *WireGuard* includes two tools, `wg` and `wg-quick` to configure and manage the *WireGuard* interfaces.

To generate the public and private keys in the server we need to run: `wg genkey | sudo tee /etc/wireguard/privatekey | wg pubkey | sudo tee /etc/wireguard/publickey`. The files will be generated in the `/etc/wireguard` directory.

Once generated, we need to configure the tunnel device that will route the VPN traffic. To do this, we create the following configuration file with a text editor for a tunnel device called `wg0` (you can use any name you want) (`sudo nano /etc/wireguard/wg0.conf`):

```
[Interface]
Address = 10.0.0.1/24
SaveConfig = true
ListenPort = 51820
PrivateKey = <SERVER_PRIVATE_KEY (the one that was just generated!)>
PostUp = iptables -A FORWARD -i %i -j ACCEPT; iptables -t nat -A POSTROUTING -o docker0 -j
MASQUERADE
PostDown = iptables -D FORWARD -i %i -j ACCEPT; iptables -t nat -D POSTROUTING -o docker0 -j
MASQUERADE
```

The settings in the interface section have the following meaning:

- **Address**: a comma-separated list of v4 or v6 IP addresses for the `wg0` interface. Use IPs from a range that is reserved for the private networks (`10.0.0.0/8`, `172.16.0.0/12`, or `192.168.0.0/16`).
- **ListenPort:** the port on which *WireGuard* will accept incoming connections. Remember this for later, as **it needs to be open in the server firewall**.
- **PrivateKey**: the private key generated by the `wg genkey` command (`sudo cat /etc/wireguard/privatekey`)
- **SaveConfig**: if `true`, the current state of the interface is saved to the configuration file when shutdown.
- **PostUp**: command or script that is executed before bringing the interface up. In this example, we are using `iptables` to enable masquerading. This will allow traffic to leave the server, giving the VPN clients access to the Internet as in a typical real setup (not ours, as we are not using a real "public" interface). Additionally, as we will be using a container from the **Lab 7 infrastructure** as the client, we will use its interface name as the "public" interface after `-A POSTROUTING`. **Although in the image it appears as `docker0`, in your case is more probable that the name will be different (something like `br-7b3b22d34e28`). To know this, look for the interface in your VM (`ifconfig`) that has an IP in the same network than the `lab07_vpn_client` container you are using**. Please be sure to change it by a real public interface if you are using this in a real setup.
- **PostDown**: command or script which is executed before bringing the interface down. The `iptables` rules will be removed once the interface is down.

The `wg0.conf` and `privatekey` files should not be readable to normal users. Use `chmod` to set the permissions to `600`: `sudo chmod 600 /etc/wireguard/{privatekey,wg0.conf}`. Once done, bring the `wg0` interface up using the attributes specified in the configuration file: `sudo wg-quick up wg0`. The command will produce an output like this:

```
[#] ip link add wg0 type wireguard
[#] wg setconf wg0 /dev/fd/63
[#] ip -4 address add 10.0.0.1/24 dev wg0
[#] ip link set mtu 1420 up dev wg0
[#] iptables -A FORWARD -i wg0 -j ACCEPT; iptables -t nat -A POSTROUTING -o docker0 -j
MASQUERADE
```

Run `sudo wg show wg0` to check the interface state and configuration:

```
interface: wg0
  public key: r3imyh3MCYggaZACmkx+CxlD6uAmICI8pe/PGq8+qCg=
  private key: (hidden)
  listening port: 51820
```

You can also run `sudo ip a show wg0` to verify the interface state:

```
4: wg0: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1420 qdisc noqueue state UNKNOWN group default qlen
1000
    link/none
    inet 10.0.0.1/24 scope global wg0
       valid_lft forever preferred_lft forever
```

Finally, to bring the *WireGuard* interface at boot time you can run the following command: `sudo systemctl enable wg-quick@wg0`

Configure Server Networking

This VPN requires NAT and this conflicts with one of the settings that we set up in a previous lab, so we must change it to correctly enable the VPN functionality. For NAT to work, we need to **enable IP forwarding**. Open the `/etc/sysctl.conf` file and modify this line to ensure its value is `1`: `net.ipv4.ip_forward=1`. Save the file and apply the change with `sudo sysctl -p`. Additionally, we need to open the UDP traffic on port `51820` (or the one we set up in the previous file) in the firewall.

VPN Client Setup

Linux clients also need the `wireguard` package installed. In our **Lab 7 infrastructure** there is a special **privileged** container (see the lab infrastructure description section) that was created precisely for this purpose and includes this package and other auxiliar ones to do the following operations, so you do not have to install them. The process for setting up a *Linux* client is pretty much the same as in the server. Start by generating the public and private keys: `wg genkey | sudo tee /etc/wireguard/privatekey | wg pubkey | sudo tee /etc/wireguard/publickey`. Then, create the file `wg0.conf` and add the following contents (`sudo nano /etc/wireguard/wg0.conf`):

```
[Interface]
PrivateKey = <CLIENT_PRIVATE_KEY (the one you just generated)>
Address = 10.0.0.2/24


[Peer]
PublicKey = <SERVER_PUBLIC_KEY (the one generated in the previous section)>
Endpoint = <SERVER_IP_ADDRESS (typically something like 172.17.0.1, the Docker host)>:51820
AllowedIPs = 0.0.0.0/0
```

The settings in the interface section have the same meaning as when setting up the server. We will just describe the following ones. If you need to configure additional clients, just repeat the same steps using a different private IP address:

- **Endpoint**: an IP or hostname of the peer you want to connect to followed by a colon, and then a port number on which the remote peer listens to. In our case this will be **the Ubuntu VM** address in the internal container network interface. Please pay attention to your client container network (172.X.X.X) to put the correct IP here.
- **AllowedIPs**: a comma-separated list of v4 or v6 IP addresses from which incoming traffic for the peer is allowed and to which outgoing traffic for this peer is directed. We use `0.0.0.0/0` because we are routing the traffic and want the server peer to send packets with any source IP.

Pairing Clients and Servers

The last step is to add the client public key and IP address to the server: `sudo wg set wg0 peer <CLIENT_PUBLIC_KEY> allowed-ips 10.0.0.2`. Make sure to change the `CLIENT_PUBLIC_KEY` with the public key you generated on the client machine and adjust the client IP address if it is different. Once done, go back to the client machine to bring up the tunneling interface.

Test VPN connection

On the *Linux* container client run the following command to bring up the interface: `sudo wg-quick up wg0`. Now you should be connected to the *Ubuntu* server, and the traffic from your client machine should be routed through it. You can check the connection with: `sudo wg`

```
root@18c398d86a51:/etc/wireguard# sudo wg-quick up wg0
[#] ip link add wg0 type wireguard
[#] wg setconf wg0 /dev/fd/63
[#] ip -4 address add 10.0.0.2/24 dev wg0
[#] ip link set mtu 1420 up dev wg0
[#] wg set wg0 fwmark 51820
[#] ip -4 route add 0.0.0.0/0 dev wg0 table 51820
[#] ip -4 rule add not fwmark 51820 table 51820
[#] ip -4 rule add table main suppress_prefixlength 0
[#] sysctl -q net.ipv4.conf.all.src_valid_mark=1
[#] nft -f /dev/fd/63
root@18c398d86a51:/etc/wireguard# wg
interface: wg0
  public key: M3VmkQTiBKu8I5HMkYhpEVvH9byTz94ZdHLgYbHj0jk=
  private key: (hidden)
  listening port: 48953
  fwmark: 0xca6c

peer: E5PZb/t/UCRGaR0o2GJMJR0lhvd4VNUUAN2Txf1tzyE=
  endpoint: 172.17.0.1:51820
  allowed ips: 0.0.0.0/0
root@18c398d86a51:/etc/wireguard#
```

*(**NOTE**: If the last command in the image is `iptables restore` instead of `nft -f`, the process may give you an error. In this case, you can fix it by installing the `iptables` package in the client)*

If we examine the network interfaces in the container, the `wg0` will appear with the local IP of the client inside the VPN network.

```
wg0: flags=209<UP,POINTOPOINT,RUNNING,NOARP>  mtu 1420
        inet 10.0.0.2  netmask 255.255.255.0  destination 10.0.0.2
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000
 (UNSPEC)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@18c398d86a51:/etc/wireguard#
```

A final test can be done by pinging the server from the client using the internal VPN network IP.

```
root@18c398d86a51:/etc/wireguard# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.774 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.172 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=2.01 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.294 ms
^C
```

Install the `apache2` package on the *Ubuntu* VM and check that you can access it from the IP of the server machine on the VPN network (`10.0.0.1`). Make a `nmap` scan from the container to this IP to see the results. Finally, try to access the `10.0.0.1:3345` with `curl` to see if you can access the website of the `lab07_web_sies` container of a previous exercise now. If all tests are successful, it means that the VPN is correctly configured.

To stop the tunneling bring down the `wg0` interface: `sudo wg-quick down wg0`. You can find details about how to use *Windows* clients here: https://linuxize.com/post/how-to-set-up-wireguard-vpn-on-ubuntu-18-04/

Expected results: This activity will be finished when you are able to successfully set up a *WireGuard* VPN connection between a special container of the Lab 7 infrastructure and the *Ubuntu* VM, passing all the proposed tests.

BADGES AND SELF-ASSESSMENT

NOTE: You have a version of this badge table in an editable format available in the *Virtual Campus*. You can use this file to easily create a document in the format you want and take extended notes of your activities to create a log of what you did. Remember that this material is key to keep track of your self-evaluation.

Badge Level	Unlocked when	Unlocked?
	You know how to allow or deny firewall services/ports	
	You know how to add network interfaces to firewall zones	
	You can allow or deny connections from single IPs or networks	
	You can answer this question: *What kind of attacks will a firewall request limit to a service potentially mitigate?*	
	You can answer this question: *What do you think it is the purpose of logging/auditing explicitly banned connections in the firewall?*	
	You can answer these questions: *Why do you think banning machines that perform scans is useful?* ***TIP****: Do you scan machines as part of your normal routine?*	
	You understand how to perform port forwarding and how to connect machines in different network segments separated by firewalls	
	You understand why DNS IP blocking is useful. You can answer this question: *What is the advantage of using PiHole to block IPs instead of a malicious IP blocking browser plugin?*	
	You can answer these questions: *Do you think that* `fail2ban` *replaces a firewall? or complements it?*	
	You can answer this question: *What kind of attacks did* `fail2ban` *prevent?*	
	You can answer this question: *Do you think that a temporal block is enough for preventing a DoS attack attempt?*	
	You can answer these questions: *Why do you think that leaving open ports on purpose can be a useful scan detection measure? Using logic, which ports would you leave open this way?*	
	You can answer these questions: *What happens when a machine is blocked by* `portsentry`*? Is the blocked machine able to contact the blocking one or not?*	
	You can answer these questions: *Does* `portsentry` *blocks affect scan speed. If the answer is yes, do you think that this can be also used as a security measure?*	
	You can answer this question: *How many things you think you can do to protect* `ssh` *connections from a network point of view?* (think about combining techniques we saw here and previous labs)	
	You can protect `ssh` attack attempts (and similar attacks to other log-enabled services) with `fail2ban` predefined jails.	
	You can enable a "port scan trap" in a machine without conflicting the firewall normal behavior. You can answer this question: *If you can collect the blocked IPs, what would you create with them?*	

	You can deploy a *Wireguard* VPN connection between a container and your host VM.	
	If we have web servers in a different internal network of our company, and we successfully deploy a VPN connection like the one we saw in this laboratory, you can answer this question: *which firewall feature do you think it will be able to provide me access to these web servers easily?*	
	**Protection against the dark networks**: Your knowledge about network security allows you to deploy a fortified server able to resist most common network attack attempts against typical services and as a final user	