University of sciences and technology
Houari-Boumediène

Cloud Computing Security Using Virtual Machines

Subject: ***Cloud Security***

Final Project Report

Realized by:

AIBECHE Yasmine

BENTOUMI Lyna Racha

KHOUAS Ihsene Noor

SADAOUI Sara Rahma

Table of contents:

I. Introduction:

The goal of this project was to simulate a secure private cloud environment using virtual machines (VMs) and open-source tools, in order to learn and demonstrate:

- How to secure a Linux-based server offering web and SSH services
- How to encrypt and safely transfer data
- How to simulate real-world network attacks (MITM, packet sniffing)
- How to detect security vulnerabilities using OpenVAS / GVM

Virtual Environment Setup

We used **VMware** to configure 3 virtual machines:

- **Ubuntu Server**: the secured cloud server
- **Ubuntu Desktop**: the user/client machine
- **Kali Linux**: the attacker machine

This lab environment allowed us to test real attacks and security mechanisms safely and efficiently, without risking any production system.

II. Network Configuration – Bridged Mode:

What is "bridged" mode and why did we choose it?

- In **Bridged mode**, each VM is **connected directly to the host's physical network interface**, and receives an **IP from the local router. In our case, we worked together on-site using a mobile data connection shared from a phone.**
- The VM is "seen" as a full device on the LAN, just like a smartphone or another PC

This is **essential** for realistic scenarios like:

- ARP spoofing
- Packet sniffing between machines
- SSH and HTTP(S) connectivity
- Running vulnerability scans over LAN

In contrast, **NAT mode** hides the VM behind the host and does not allow direct access from the LAN.

Manual IP Configuration (Static)

Instead of using DHCP, we **manually assigned static IP addresses** to each VM to maintain full control over their network identity and ensure consistent addressing across reboots.

This was done by editing the network interface configuration file:

Configuration file: `sudo nano /etc/network/interfaces`

Example static configuration for Ubuntu Server:

```
auto eth0/ens33 #network interface
iface eth0 inet static
    address 172.20.10.4
    netmask 255.255.255.0
    gateway 172.20.10.1
```

- **auto eth0:** tells the system to bring the interface up at boot
- **inet static**: specifies that we are assigning a static IPv4 address
- **address:** the fixed IP we want to assign to the VM
- **netmask:** subnet mask (commonly 255.255.255.0 for /24 networks)
- **gateway**: IP of the router or box (used for Internet access)

Each VM was given a different static IP, in the same subnet (e.g., 172.20.10.4, 172.20.10.13, etc.), to allow proper communication.

Applying the configuration

After editing the file, we applied the changes using:

```
sudo systemctl restart networking
```

Connectivity Verification

To make sure everything worked, we ran the following commands:**`ip a`**

To confirm that the assigned static IP appears on the correct interface (**`eth0`**), and:

```
ping 172.20.10.4      # From client to server
ping 172.20.10.13     # From server to client
```

These tests confirmed:

- All machines are in the **same subnet**
- **Bi-directional communication** is working
- Network setup is ready for SSH, HTTP/HTTPS, and MITM simulations

III. SECURING THE CLOUD SERVER (UBUNTU SERVER):

Web Server – SSH – SSL – Firewall (UFW)

The goal of this phase was to configure a **secure cloud service** on our Ubuntu Server. This included:

- Deploying a web server (Apache)
- Activating HTTPS with a self-signed SSL certificate
- Enabling remote access via SSH (with keys, not passwords)
- Restricting unauthorized access via a firewall (UFW)

All services were configured to **respect modern security standards**.

1. Installing Required Services

We first installed the essential packages for running and securing the server:

```
sudo apt update

sudo apt install apache2 openssh-server ufw openssl -y
```

- **apache2 :**Web server that serves HTTP/HTTPS content
- **openssh-server :**Secure remote shell access (SSH)
- **ufw:**Simplified firewall for access control
- **openssl:**Toolset for creating cryptographic certificates and performing encryption

2. Firewall Configuration with UFW

We used **UFW (Uncomplicated Firewall)** to define which services are allowed to communicate with the server.

```
sudo ufw allow 22      # Allow SSH

sudo ufw allow 80      # Allow HTTP (non-secure web)

sudo ufw allow 443     # Allow HTTPS (secure web)

sudo ufw enable        # Activate the firewall
```

By default, UFW blocks all incoming connections unless explicitly allowed.

Why use a firewall?

To reduce the attack surface by only exposing the **necessary ports**. All others (e.g., FTP, Telnet, etc.) remain blocked to prevent unauthorized access or scanning.

3. Verifying Apache Installation

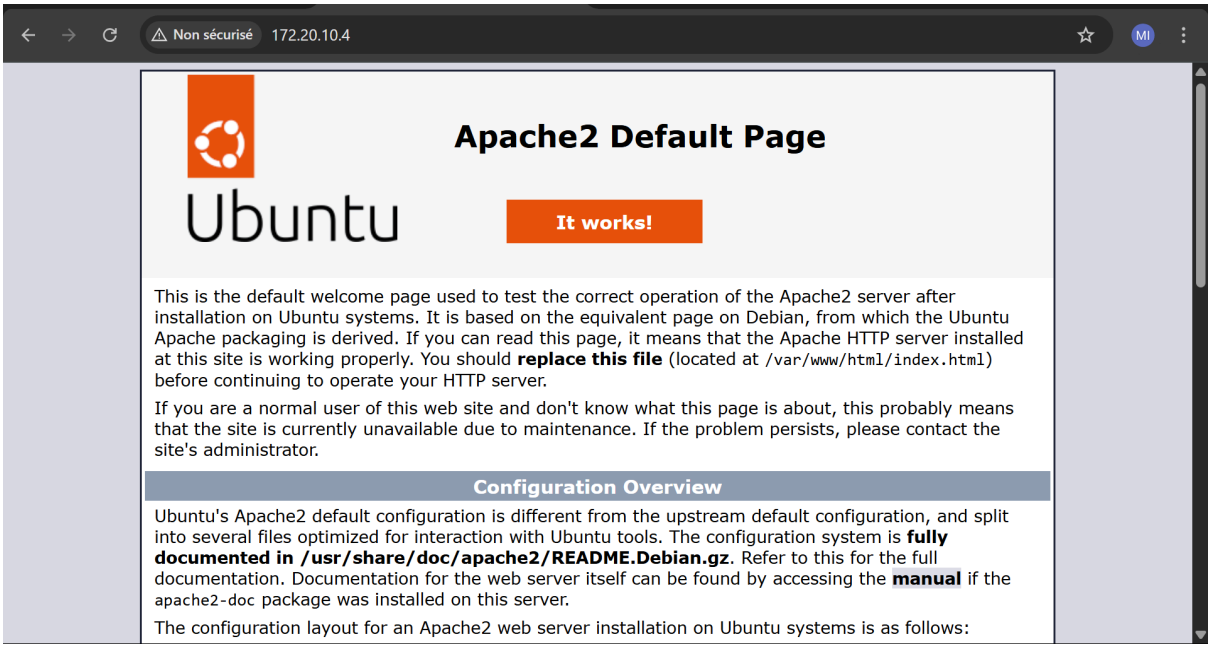
Once Apache was installed and the firewall rules applied, we checked that it was working:

```
sudo systemctl status apache2
```

We then opened a browser on the client and accessed:

```
http://172.20.10.4
```

The default Apache welcome page was displayed, confirming that the HTTP service was running properly.



IV. ADDING HTTPS WITH A SELF-SIGNED SSL CERTIFICATE:

Why use HTTPS?

HTTP is **not encrypted**, which means any data (login forms, messages, credentials) sent between client and server can be **eavesdropped** or **manipulated**.

HTTPS (HTTP over SSL/TLS) encrypts the traffic, preventing **MITM attacks** and **packet sniffing**.

Since we didn't purchase a certificate from a Certificate Authority (CA), we created our **own certificate** using `openssl`.

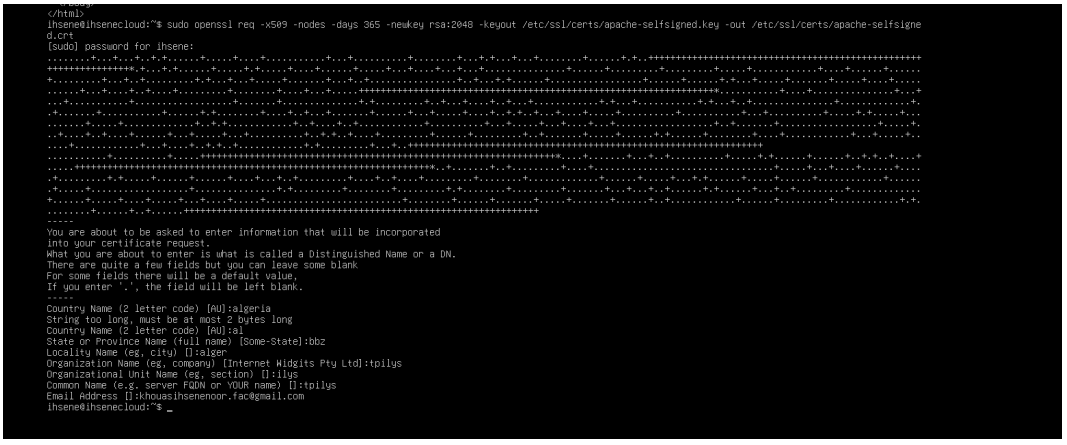
1. Creating the SSL Certificate (self-signed)

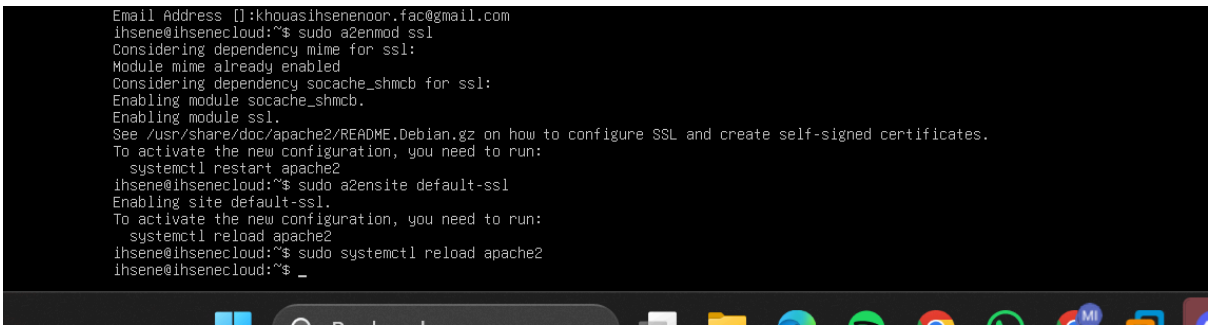
We generated both the **private key** and the **certificate** in one command:

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048
\

-keyout /etc/ssl/private/apache-selfsigned.key \

-out /etc/ssl/certs/apache-selfsigned.crt
```

- **-x509:**Generate a certificate (not a request)
- **-nodes**:Do not encrypt the private key with a passphrase
- **-days 365**:Validity of the certificate
- **-newkey rsa**:2048:Generate a new 2048-bit RSA private key
- **-keyout**:Path to save the private key
- **-out:**Path to save the actual certificate

During generation, we entered some basic identity info (country, organization, etc.).
The result: Apache now had a working certificate and private key pair.





2.Enabling HTTPS in Apache

We activated Apache's SSL module and the default SSL site:

```
sudo a2enmod ssl
```

```
sudo a2ensite default-ssl
```

```
sudo systemctl reload apache2
```

What these commands do:

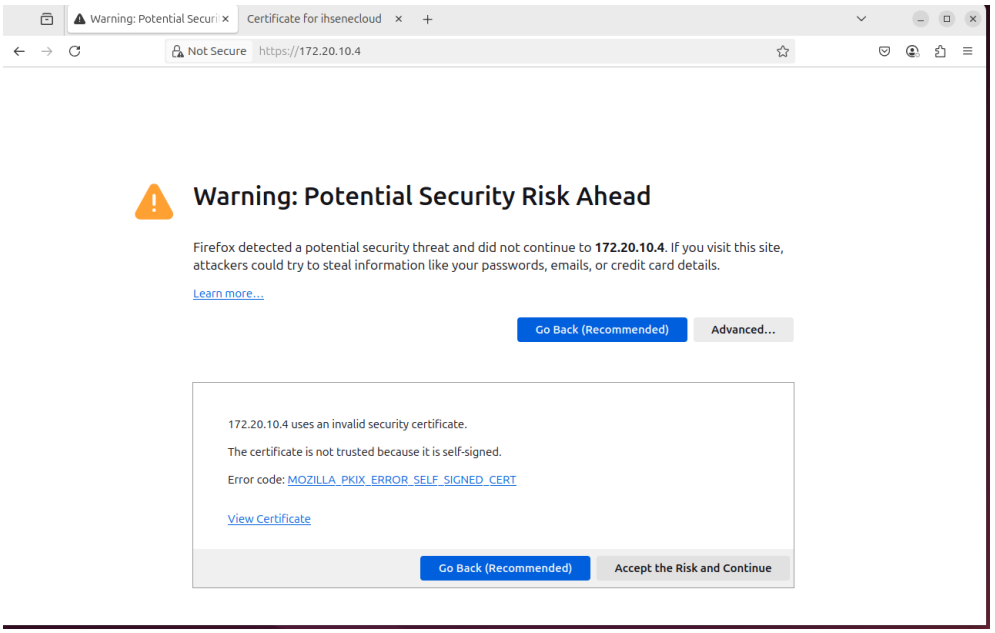
- **`a2enmod ssl`:** loads the SSL module in Apache
- **`a2ensite default-ssl`:** enables the SSL-enabled virtual host
- **`reload apache2`:** restarts the service to apply changes

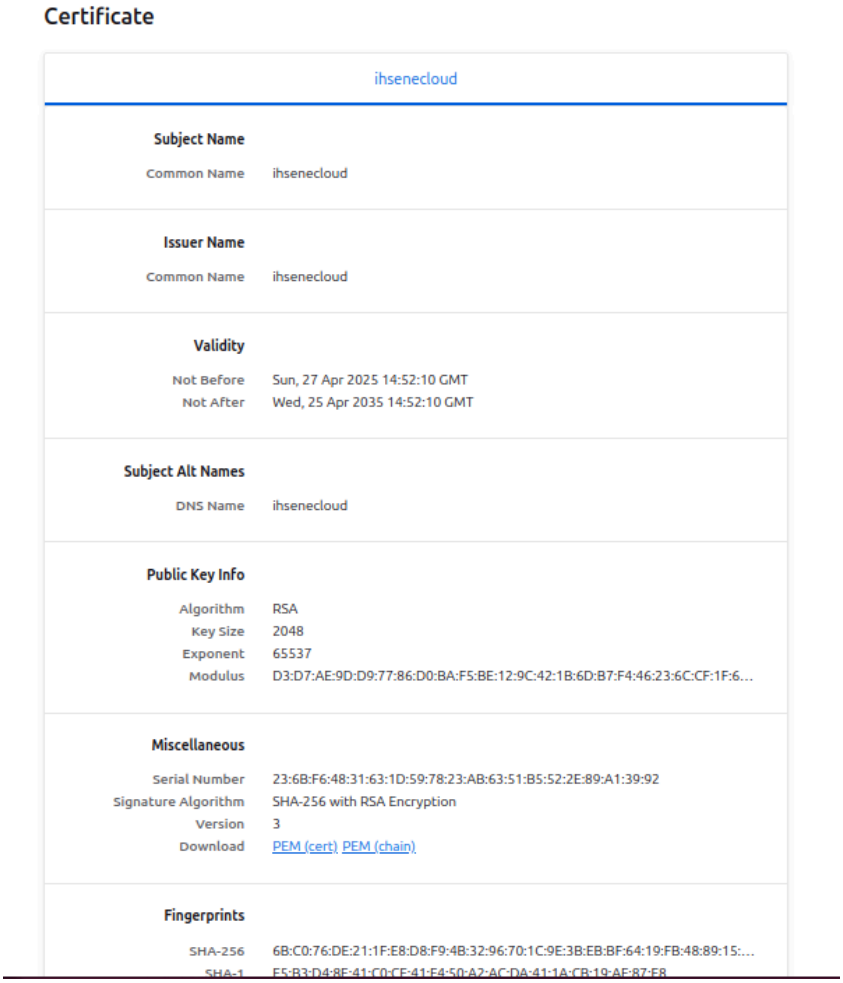
3. Verifying HTTPS from the Client

From the client machine (Ubuntu Desktop), we visited:

```
https://172.20.10.4
```

The browser warned about a **self-signed certificate**, which is expected since the certificate is not trusted by a CA. After accepting the warning, the secure version of the Apache page was loaded.





V. CONFIGURING SECURE REMOTE ACCESS VIA SSH:

Why not use the root account?

Using the root account directly for remote access is:

- Risky (full access if credentials are stolen)
- Difficult to audit
- A common target for brute-force attacks

Instead, we create a dedicated user with **limited privileges** and **sudo access**, which:

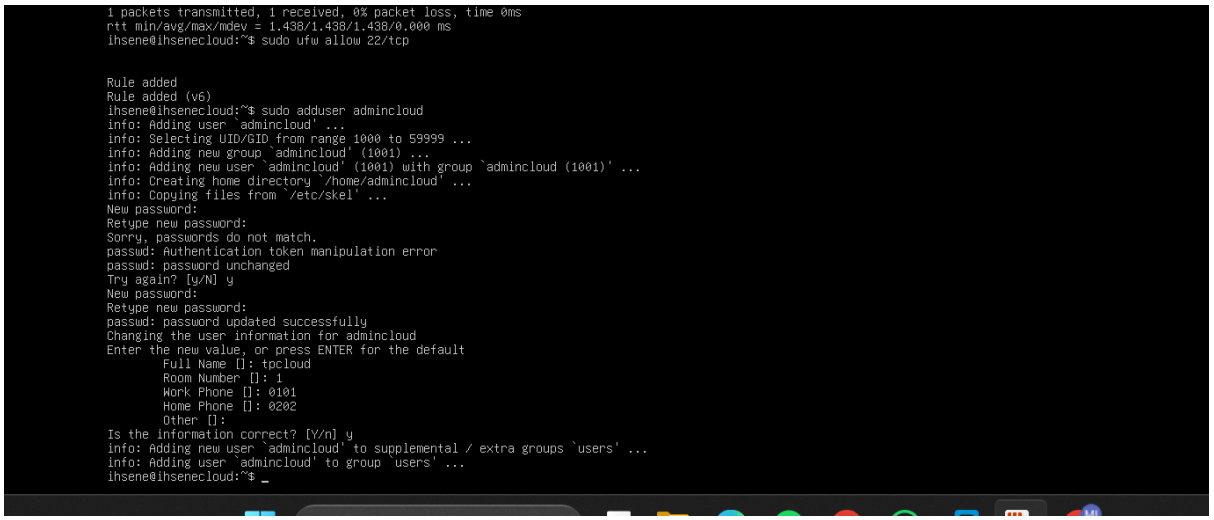
- Increases security
- Allows controlled administrative actions
- Enables clear tracking of system activity

1. Creating a Non-Root User for SSH Access

On the Ubuntu Server: **`sudo adduser admincloud`**

This command prompts for:

- A password for the new user
- Optional personal details (name, room, etc.)



Then we add the user to the **`sudo`** group: **`sudo usermod -aG sudo admincloud`**

- **`adduser`:** creates a new user along with their home directory **(`/home/admincloud`)**
- **`usermod -aG sudo`**: adds the user to the `sudo` group, giving them administrative rights (when using `sudo`)

2. Setting Up SSH Key Authentication

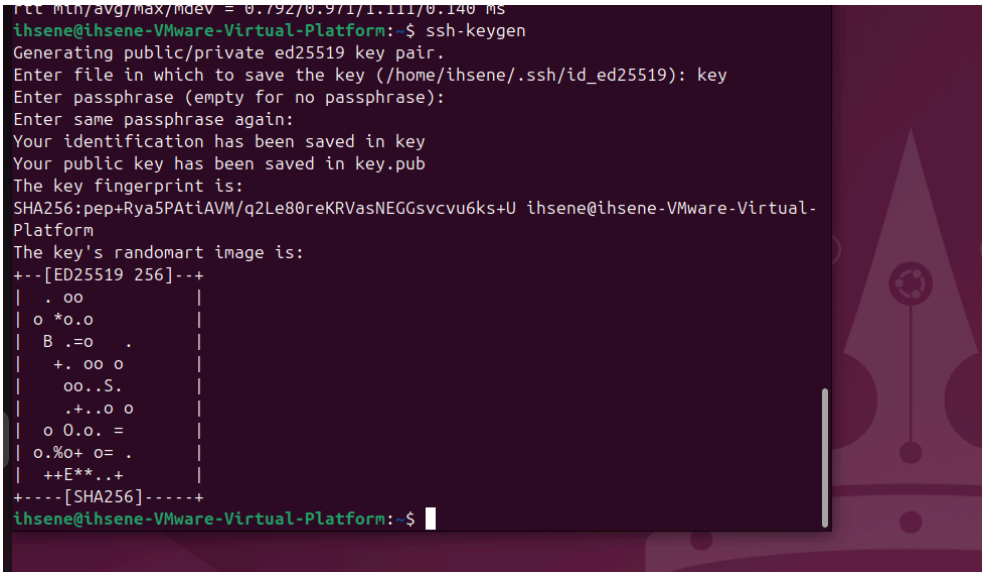
Using **SSH key pairs** is far more secure than using passwords:

- Passwords can be guessed or intercepted
- SSH keys are long, randomly generated cryptographic keys
- Even if traffic is intercepted, **keys cannot be reversed like passwords**

On the Ubuntu Desktop (client): `ssh-keygen`

This generates:

- A **private key** (stored in **`~/.ssh/id_rsa`**)
- A **public key** (stored in **`~/.ssh/id_rsa.pub`**)



We then copy the **public key** to the server:

`ssh-copy-id admincloud@172.20.10.4`

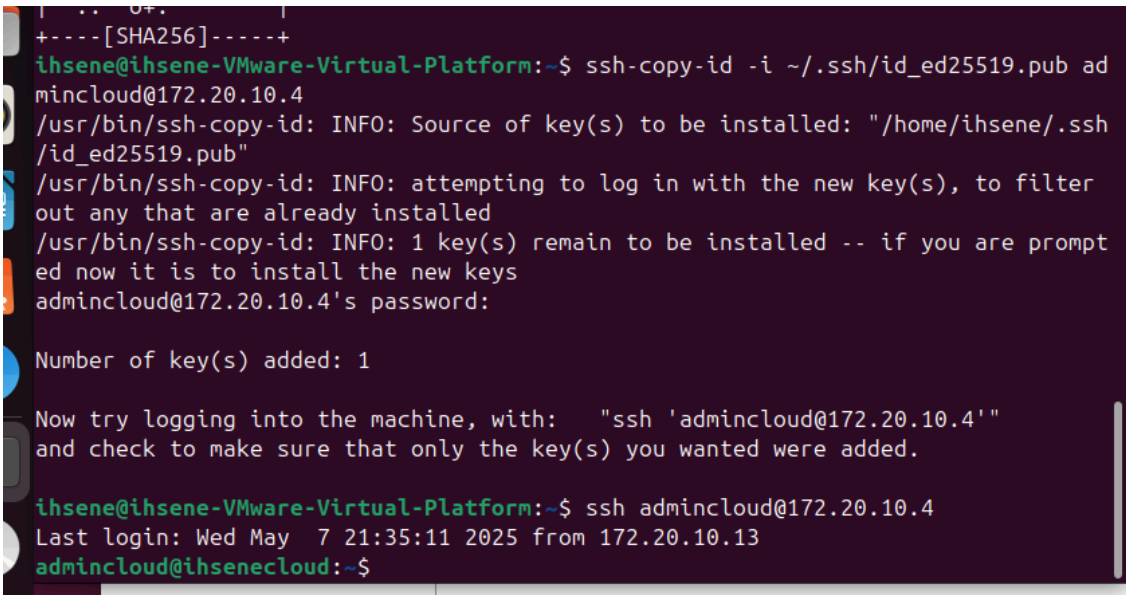
This securely appends the public key to:

`/home/admincloud/.ssh/authorized_keys`

From now on, the client can connect using the private key without typing a password.

3. Testing the Connection

Back on the client:**`ssh admincloud@172.20.10.4`**



Result: **direct access to the server** without entering a password.

4. Disabling Password Authentication (SSH Hardening)

To fully harden the server, we disable password-based logins entirely:

```
sudo nano /etc/ssh/sshd_config
```

Ensure the following line is **uncommented and set to "no"**:

```
PasswordAuthentication no
```

Then reload the SSH service: `sudo systemctl restart ssh`

This prevents any user — even root — from logging in via SSH using a password.

Why is this important?

- Brute-force attacks against SSH are common
- If only key-based login is allowed, even thousands of guesses per second won't work unless the attacker has the private key

VI. SECURE FILE ENCRYPTION AND TRANSFER – OpenSSL + SCP:

The goal was to:

- **Encrypt a sensitive file** on the server
- **Transfer it securely** to the client
- **Decrypt it on the client side**
- Demonstrate the **confidentiality and integrity** of data transmitted over the network

We used:

- **OpenSSL**: for local encryption (AES-256-CBC)
- **SCP (Secure Copy Protocol)**: for transmission over SSH

1. Creating and Encrypting a File on the Server

On the Ubuntu **Server**, we started by creating a basic text file:

```
echo "secret data" > secret.txt
```

Then we used `openssl` to encrypt it with a symmetric cipher:

`openssl enc -aes-256-cbc -salt -in secret.txt -out secret.txt.enc`

- **`-enc:`** Launch OpenSSL's built-in encryption tool
- **`-aes-256-cbc`** : Use AES encryption with 256-bit key in CBC mode
- **`-salt:`** Random salt to prevent pattern recognition (adds randomness)
- **`-in`** : Specifies the input file (plaintext)
- **`-out:`** Specifies the output file (encrypted)

2. Transferring the Encrypted File to the Client (SCP)

We used **`scp`** to send the encrypted file from the **server to the client**, over a **secure SSH tunnel**.

From the **Ubuntu Desktop (client)**:

`scp admincloud@172.20.10.4:/home/admincloud/secret.txt.enc.`

- This command connects to the server using SSH
- Downloads the file **`secret.txt.enc`** to the current local directory
- All traffic is encrypted via SSH

Why SCP instead of regular copy or FTP?

- SCP is based on SSH, so it's inherently secure (encryption, authentication, integrity)
- FTP or plain **`cp`** (copy) over unprotected channels exposes files to interception

3. Decrypting the File on the Client Side

Back on the client, we decrypt the file:

`openssl enc -aes-256-cbc -d -in secret.txt.enc -out decrypted_secret.txt`

You will be asked for the **same passphrase** used during encryption.

If the passphrase is correct, the file will be restored in plaintext.

- Create and protect confidential files
- Move them securely across the network
- Reconstruct them safely on the receiving end

VII. SIMULATING A MAN-IN-THE-MIDDLE (MITM) ATTACK:

Using Kali Linux, arpspoof, and Wireshark

The objective was to:

- **Intercept communication between the client and the server**
- Perform a **Man-in-the-Middle attack (MITM)** using **ARP spoofing**
- Use **Wireshark** to analyze what data is visible in intercepted packets
- Compare **HTTP** vs **HTTPS** traffic visibility

1. Understanding the MITM Concept

In a local network (LAN), devices communicate using **MAC addresses**.
The **ARP protocol (Address Resolution Protocol)** maps IP addresses to MAC addresses. It is **trust-based and unprotected**, which makes it vulnerable.

In ARP spoofing:

- The attacker (Kali) sends fake ARP replies to both the **client** and the **server**
- Both machines believe that the attacker is the other machine
- This allows the attacker to **intercept, forward, and possibly alter traffic**

This creates a **transparent MITM** setup, where the attacker can observe and capture data **in real time**.

2. Attack Setup – Kali Linux

a. Enable IP Forwarding

This step allows Kali to act as a **router** by forwarding packets between client and server.

```
sudo sysctl -w net.ipv4.ip_forward=1
```

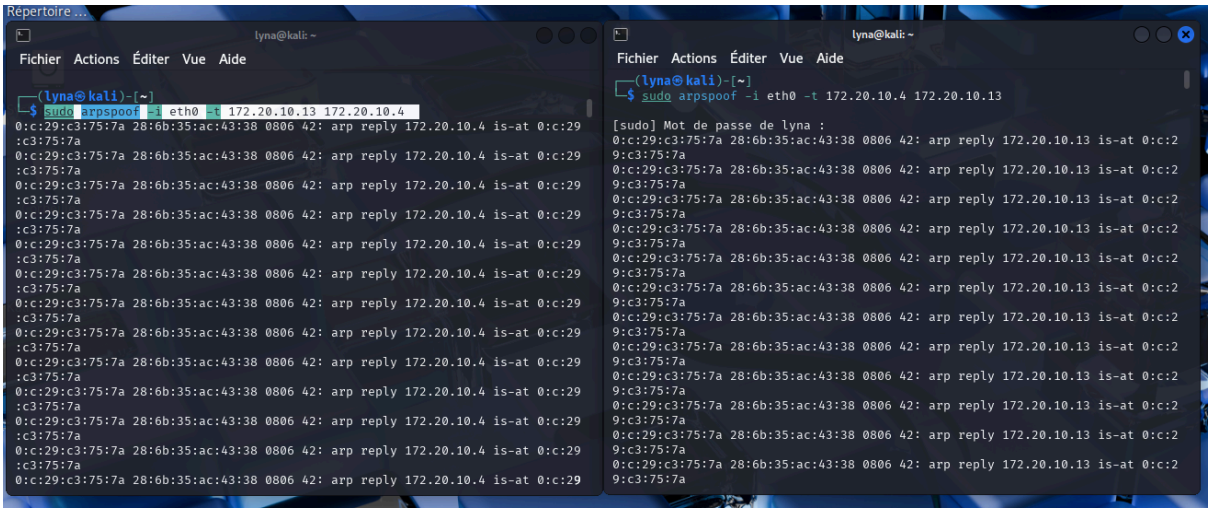
This changes a kernel parameter temporarily.

b. Launch ARP Spoofing (two terminals)

We used two terminals to poison both directions (client → server and server → client):

```
# Terminal 1

sudo arpspoof -i eth0 -t 172.20.10.13 172.20.10.4

# Terminal 2

sudo arpspoof -i eth0 -t 172.20.10.4 172.20.10.13
```

- **`-i eth0`**: interface to attack on (check with `ip a`)
- **`-t`:** specifies the target of the spoofing



- The second IP is the one we impersonate

So:

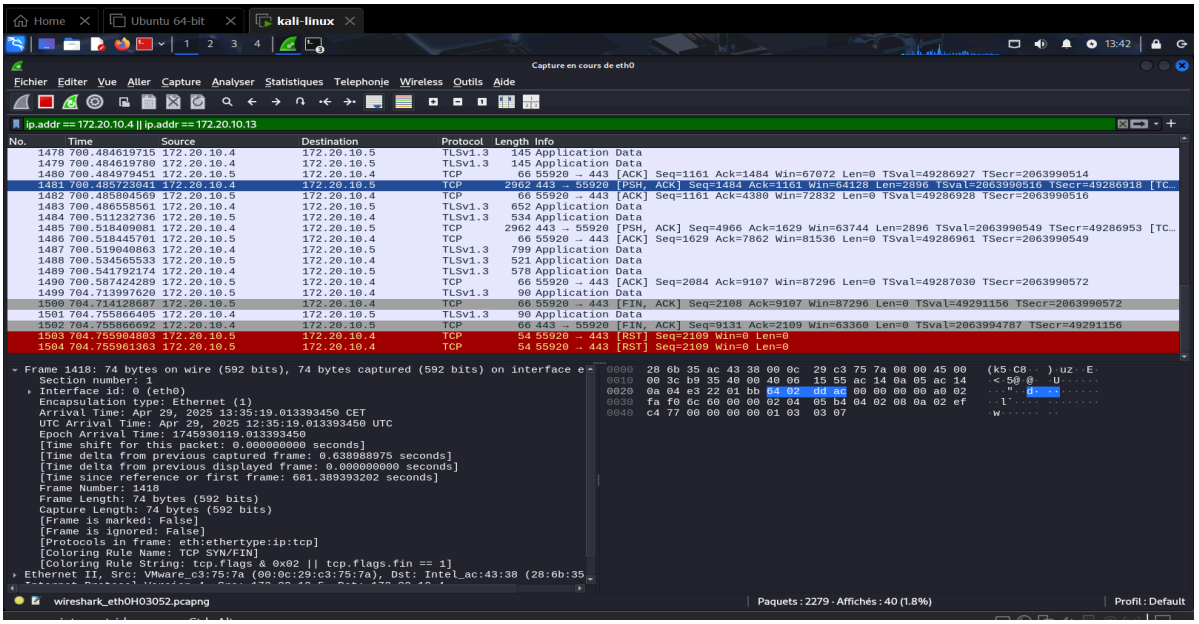
- Terminal 1 tells the **client (13)** that Kali is the **server (4)**
- Terminal 2 tells the **server (4)** that Kali is the **client (13)**

Now **all traffic flows through Kali**, unknowingly to both endpoints.

3. Intercepting and Analyzing Traffic with Wireshark

a. Start Wireshark on Kali and apply this filter:

This shows only packets between the **client and server**.



b. Test 1: Visiting a page over HTTP

On the client:**`http://172.20.10.4`**

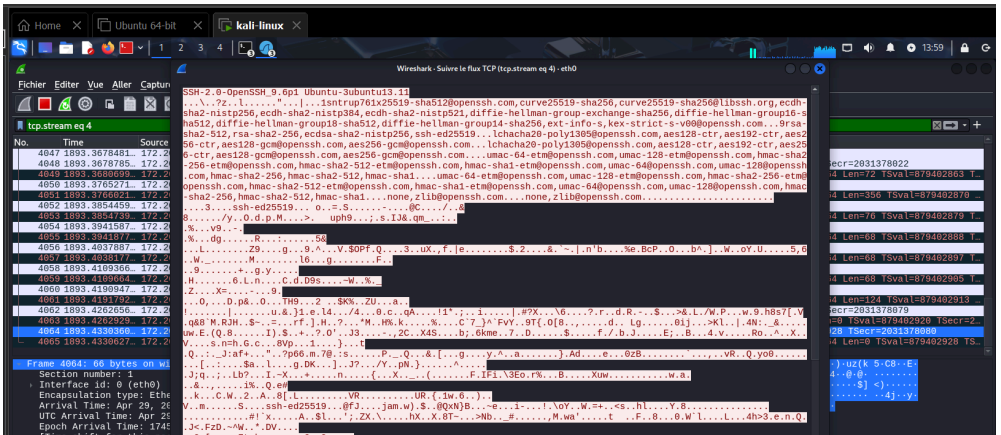
Result in Wireshark:

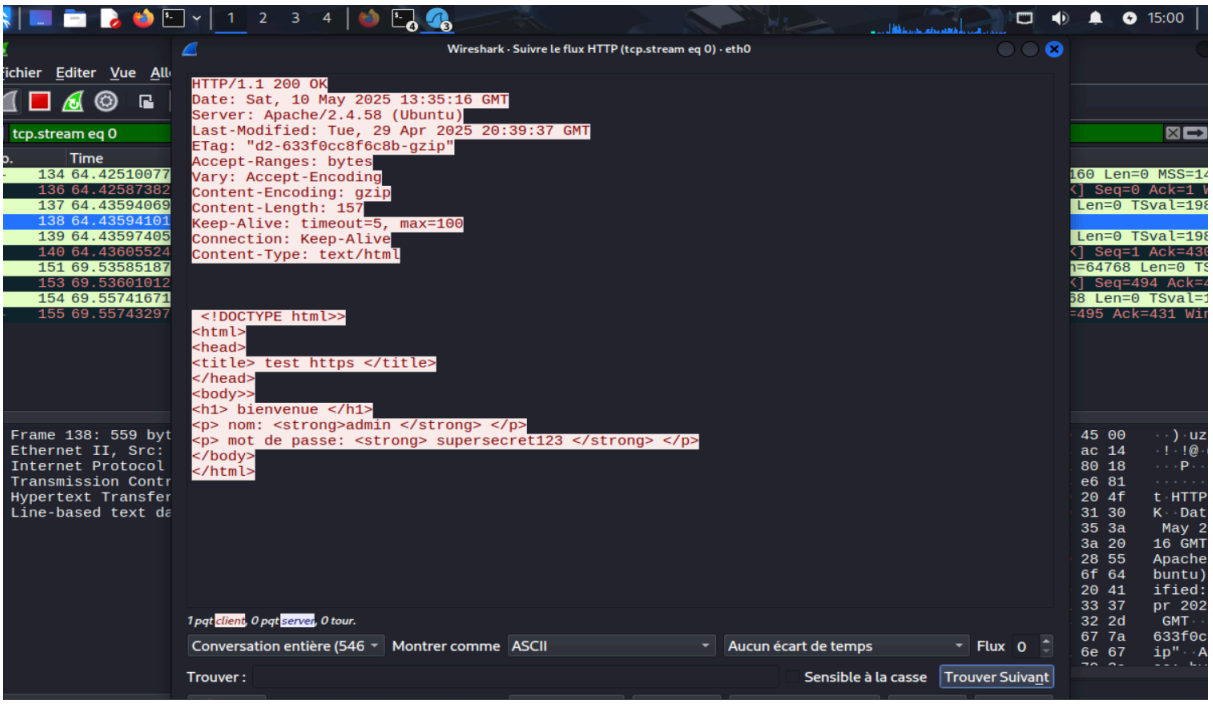
- Visible **GET** requests
- URLs, headers, and **entire HTML page content**
- If there's a login form, credentials are sent in **cleartext**

This proves that HTTP is completely vulnerable to MITM attacks.

c. Test 2: Visiting the same page over HTTPS

On the client:**`https://172.20.10.4`**





Result in Wireshark:

- **The actual content is encrypted and unreadable**
- No credentials or messages are visible

This demonstrates the effectiveness of HTTPS encryption.

VIII. VULNERABILITY SCANNING WITH OPENVAS (GVM):

Greenbone Vulnerability Management – Setup, Troubleshooting, and Results

The goal was to use **OpenVAS** (Open Vulnerability Assessment System), an open-source security scanner included in **Greenbone Vulnerability Management (GVM)**, to:

- Scan the Ubuntu server
- Detect exposed ports, outdated services, and known CVEs
- Generate vulnerability reports

1. Installing OpenVAS and GVM on Kali Linux

On the Kali machine, we installed OpenVAS and initialized its components:

```
sudo apt update
```

`sudo apt install openvas -y`

Then we initialized the environment using:

`sudo gvm-setup`

This command:

- Downloads the **vulnerability feeds** (NVTs, CVEs, SCAP data)
- Creates the system user **`_gvm`**
- Sets up the scanner daemon **(`ospd-openvas`)**
- Configures the GVM daemon **(`gvmd`)**
- Generates the default GSA (Greenbone Security Assistant) web interface

To start GVM: **`sudo gvm-start`**

Access the web interface on: **`https://127.0.0.1:9392`**

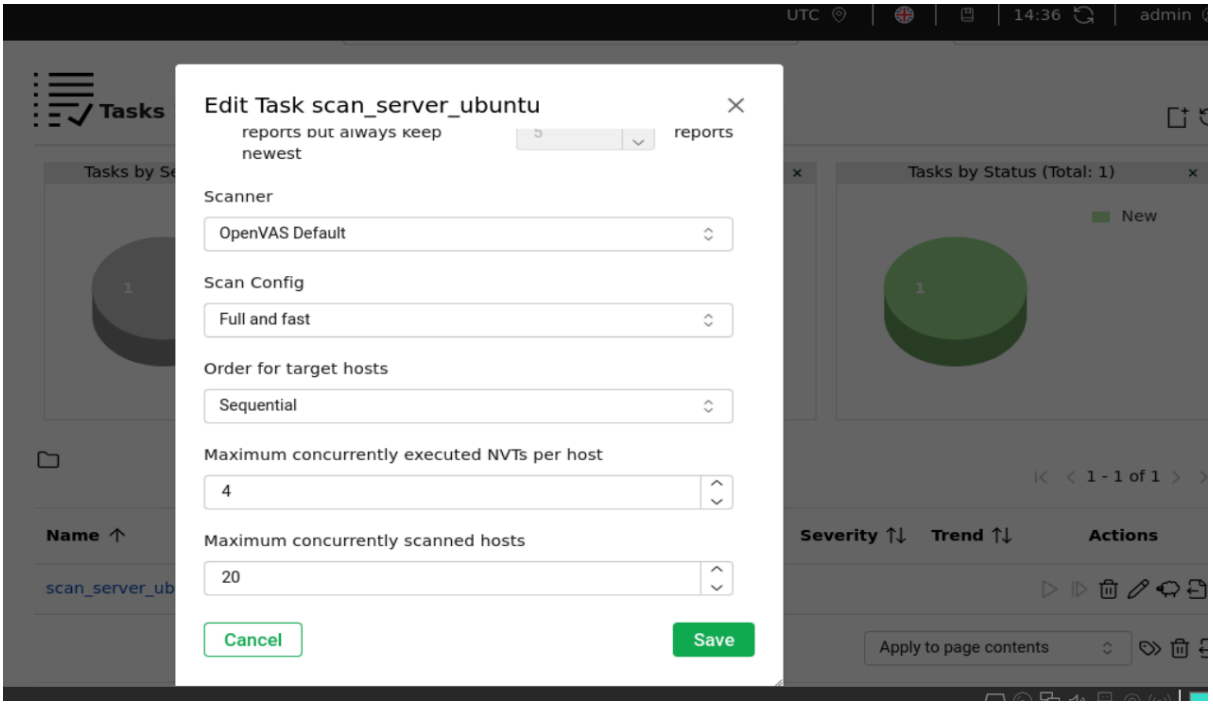
In the final phase, we attempted to launch a vulnerability scan after creating a scan task. This was done on two different machines:

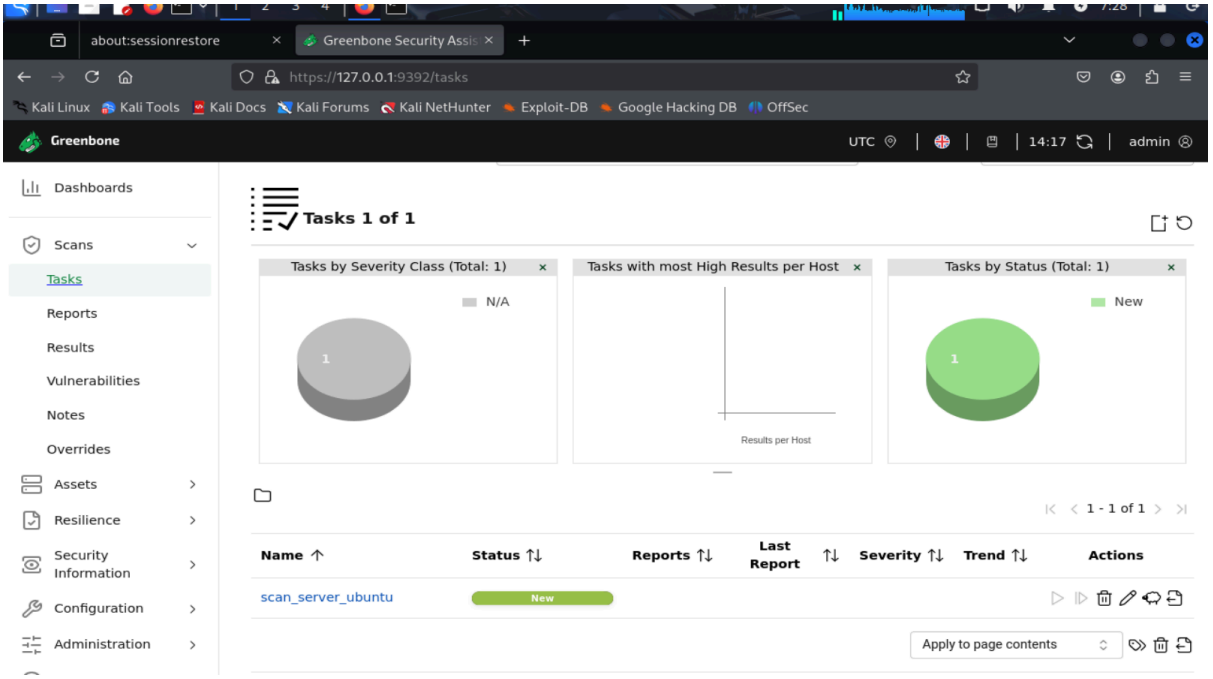
On the first machine, we successfully created the target, scan config, and used the OpenVAS Default scanner without any errors — all components were properly detected. However, the scan could not be launched because the "Start" button remained disabled. Upon investigation, we discovered that the vulnerability feed could not be downloaded due to insufficient disk space in the virtual machine and the host one, which blocked the update process required to activate the scanner.

On the second machine, we faced a different situation: the scan task could not be initiated due to persistent configuration issues.

Despite verifying the OpenVAS scanner host address, socket port (9391), and feed synchronization, the GVMD manager failed to connect correctly, causing the scan to remain stuck in the "New" state.

These issues prevented us from completing the vulnerability assessment, as illustrated in the final screenshot.





IX. FINAL CONCLUSION:

This project allowed us to set up a complete cloud security environment using virtual machines. We manually configured a bridged network with static IPs, secured the Ubuntu server with HTTPS, SSH key authentication, and a firewall, and ensured secure file transmission with OpenSSL and SCP. We also simulated a Man-in-the-Middle attack using ARP spoofing and verified

with Wireshark that encrypted traffic (HTTPS, SSH) remains protected, while unencrypted traffic (HTTP) is fully visible.

Despite this final limitation, the project was a success in demonstrating practical skills in network security, encryption, attack detection, and system hardening.