Project Report

22 Augst 2025

Ayush Kumar Thakur

Mentor:- Vasudev Jha

INFOTACT SOLUTION (SaaS)

Network Intrusion Detection System (NIDS) Rule Creation and Testing Lab Report.

"A Comprehensive Analysis of Challenges in Network Intrusion Detection Systems"

(Duration 3 Month)

Cybersecurity Internship (Remote)
Mentor: Vasudev Jha
Organization: Infotact Solution

Ayush Kumar Thakur
Group :- G7
14-october-2024

Abstract

This project focuses on the design, configuration, and evaluation of a Network Intrusion Detection System (NIDS) using **Snort**, one of the most widely used open-source intrusion detection tools. The primary goal was to build and test a system capable of monitoring live network traffic and identifying potential security threats in real-time. To achieve this, a set of **custom detection rules** was created to recognize various types of malicious activities, such as **reconnaissance scans (Nmap), brute-force login attempts (Hydra), and simulated command-and-control (C2) malware traffic**.

Through systematic testing, the project demonstrates how Snort can effectively detect these attacks as they occur, highlighting its role in **strengthening network defenses** and **reducing the mean time to detect (MTTD) cyber threats**. By bridging theoretical concepts of intrusion detection with practical implementation, this project showcases the importance of NIDS in modern cybersecurity environments and provides valuable insights into how customizable detection mechanisms can help secure organizational networks against evolving attack vectors.

1. Introduction

With the increasing number of cyber threats and attacks targeting organizational networks, intrusion detection has become one of the most critical aspects of modern cybersecurity. A **Network Intrusion Detection System (NIDS)** plays a vital role in monitoring network traffic, identifying suspicious activities, and alerting administrators before an attack can cause significant damage.

This project focuses on implementing a NIDS using **Snort**, an open-source and widely adopted intrusion detection tool, in a controlled virtualized environment. The setup consists of an **Ubuntu Server VM** acting as the target machine and a **Kali Linux VM** simulating an attacker. Snort was configured with a series of **custom detection rules** to identify different malicious behaviors such as **ICMP pings, Nmap reconnaissance scans, brute-force login attempts, and simulated malware Command-and-Control (C2) traffic**.

Each test was carefully executed step by step, and the alerts generated by Snort were recorded as proof-of-concept. The results not only validate Snort's ability to detect these attack patterns in real time but also demonstrate how customizable rules can enhance its effectiveness in safeguarding network infrastructure. This hands-on approach bridges theoretical cybersecurity concepts with practical application, providing a clear understanding of how intrusion detection systems contribute to proactive network defense.

Let's Start 👉 👇

2. Lab Environment Setup

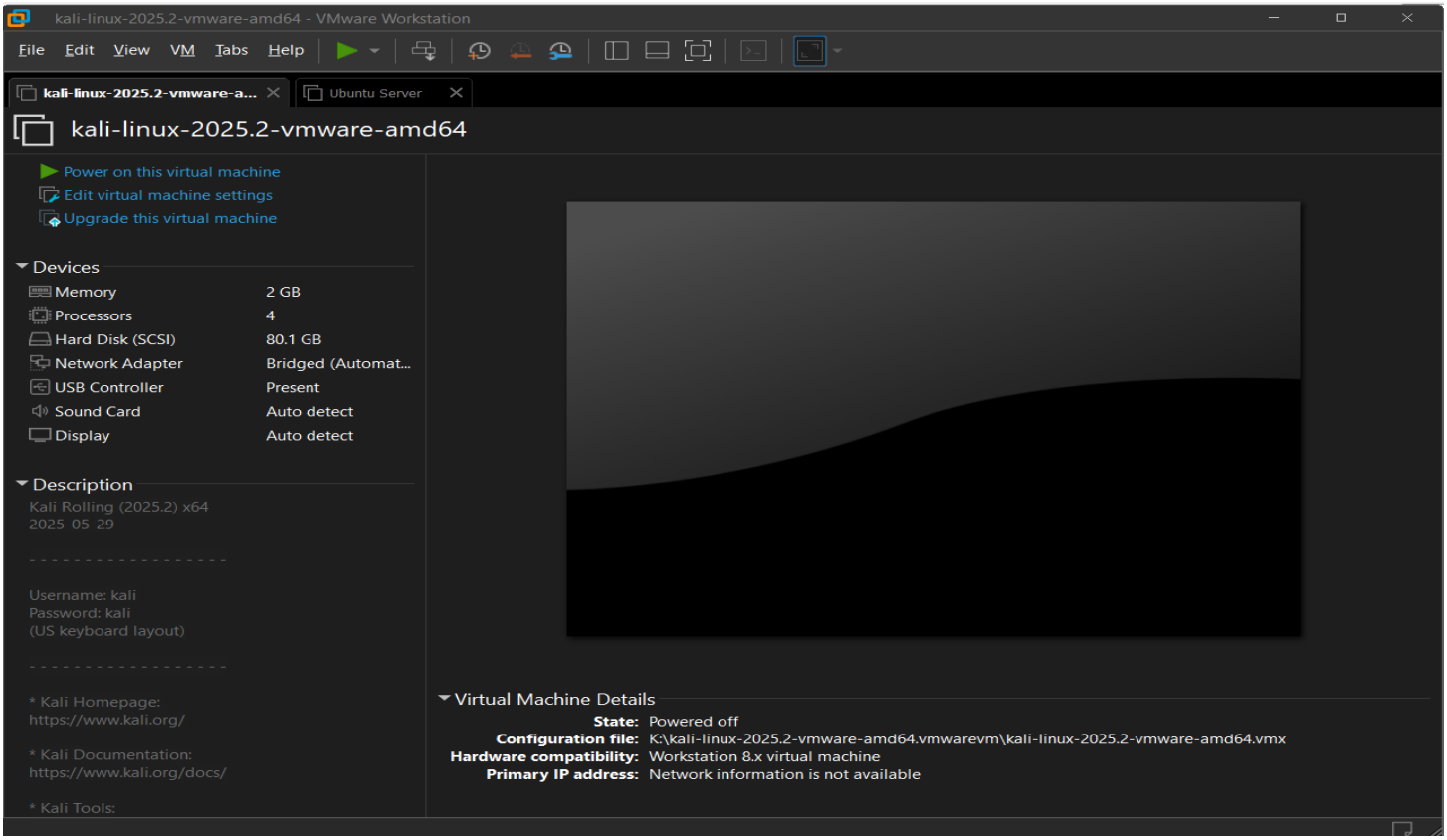
To build a controlled and realistic cybersecurity testing environment, I set up a small virtual lab consisting of two main virtual machines and a few essential tools.

1. **Virtual Machines**
    - **Ubuntu Server:** This machine serves as the primary target system and has **Snort** installed. Snort acts as our Intrusion Detection System (IDS), monitoring network traffic and helping us identify suspicious activities.
    - **Kali Linux:** This machine functions as the **attacker's workstation**. It comes preloaded with a wide range of penetration testing tools, making it ideal for simulating real-world attacks against the Ubuntu server.
2. **Virtualization Platform**
   Both machines are hosted on **VirtualBox/VMware**, running in **Bridged Networking Mode**. This configuration ensures that each VM behaves as if it were a separate device connected to the same local network, allowing them to communicate seamlessly with each other just like physical machines would.
3. **Tools Utilized**
    - **Snort:** Used for real-time traffic monitoring and intrusion detection.
    - **Nmap:** A powerful network scanner for discovering open ports and services on the target machine.
    - **Hydra:** A password-cracking tool used to perform brute-force attacks, particularly against SSH services.
    - **Wireshark:** A packet analyzer that helps capture and inspect network traffic, providing deeper insight into what is happening on the wire.
4. **Services Configured**
    - **OpenSSH Server:** Installed on the Ubuntu server to enable secure remote login. This service will be the main target for brute-force attack simulations, allowing us to test detection and response capabilities using Snort.

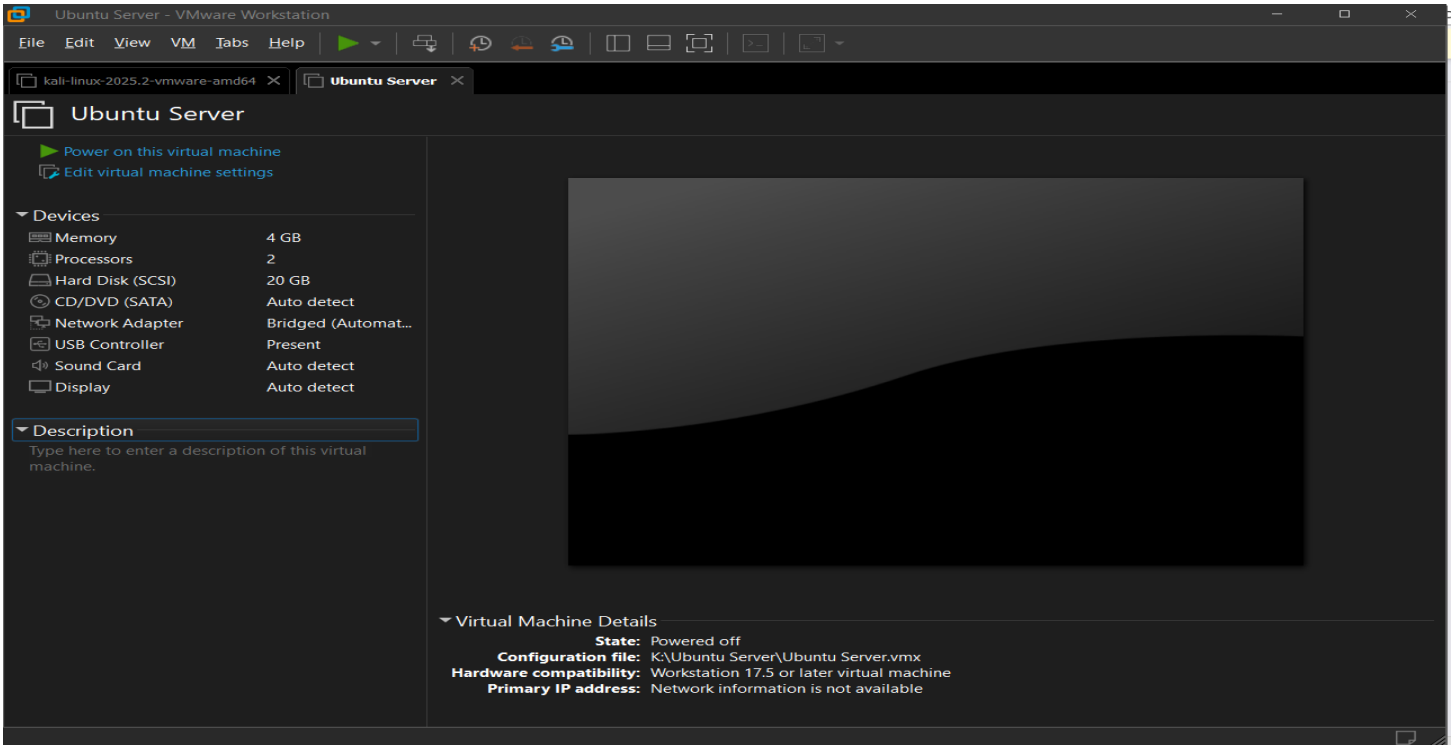
Implementation:-

- **Virtual Machines:** Ubuntu Server (Snort installed), Kali Linux (Attacker machine)
- **Virtualization:** VMware in Bridged Mode

Kali Linux (Attacker machine):-



Ubuntu Server (Snort installed):-



Week 1: Snort Setup and Initial Verification

During the first week of the lab, I focused on setting up **Snort** on the Ubuntu Server and verifying its basic functionality. The goal was to ensure that Snort could properly capture and log normal network traffic before moving on to more advanced attack simulations.

Step 1: Installing Snort on Ubuntu Server (Target Machine)

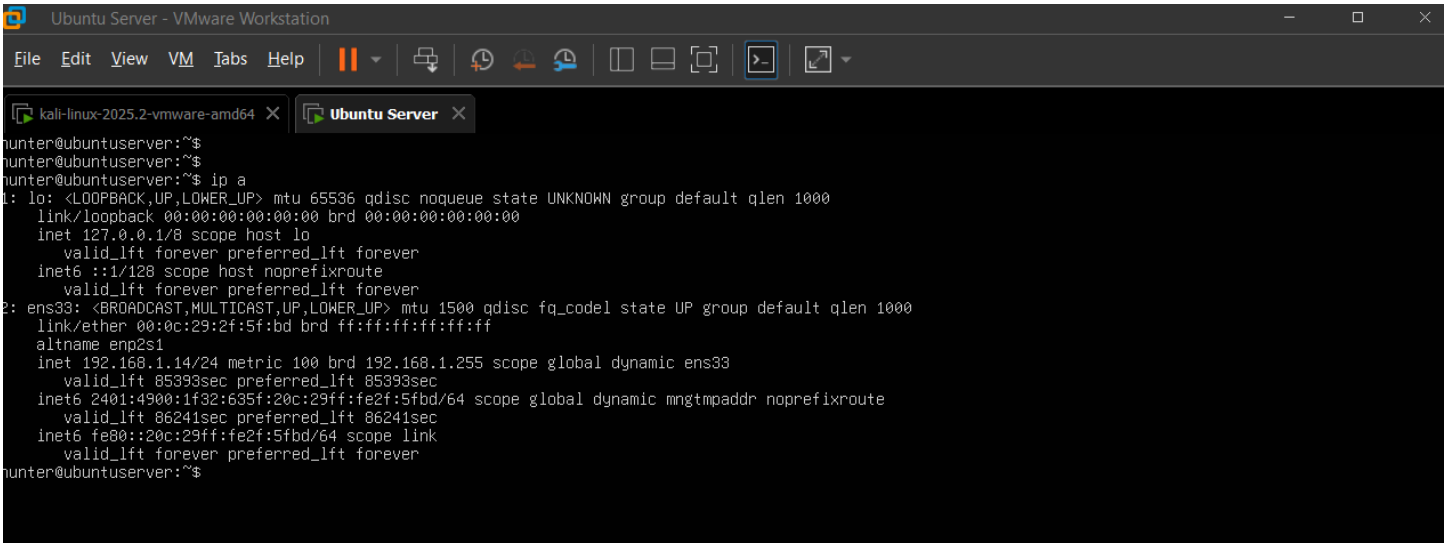
On the Ubuntu Server, I first updated the system packages and then installed Snort using the following command:

```
sudo apt update && sudo apt install -y snort
```

After installation, I verified the **network configuration** of the Ubuntu machine to confirm its IP address:

```
ip a
```

[Screenshot of Snort installation and IP address output]



Step 2: Installing Tools on Kali Linux (Attacker Machine)

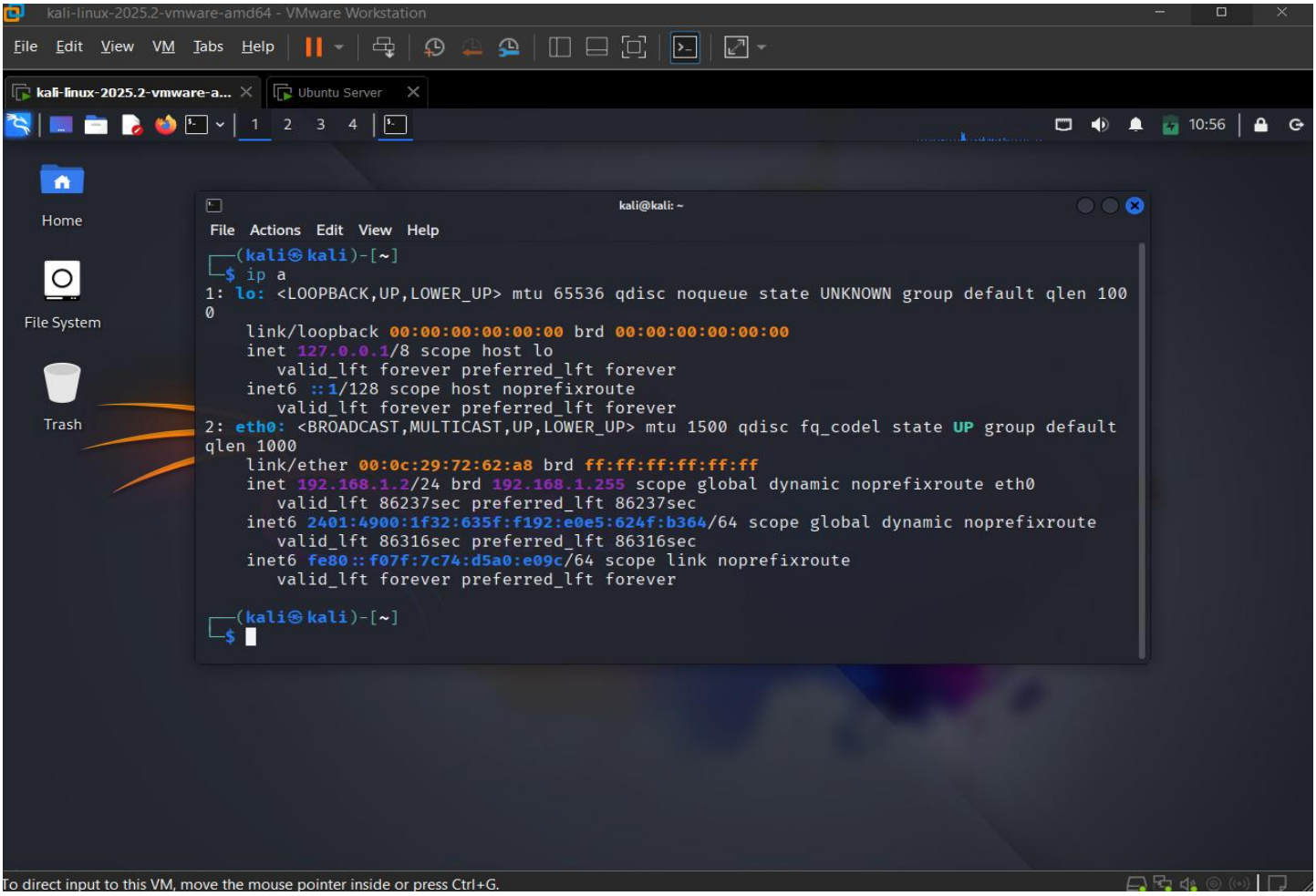
On the Kali Linux machine, I installed the essential tools required for testing, including **Nmap, Hydra, and Curl**. These tools will be used later for scanning, brute-force attacks, and generating traffic toward the Ubuntu server.

```
sudo apt update && sudo apt install -y nmap hydra curl
```

I then checked the network configuration of the Kali machine as well:

```
ip a
```

[Screenshot of tool installation and IP address output]



2. ICMP Detection Test

The next step in the lab was to verify Snort's ability to detect simple ICMP echo requests (commonly known as pings). This test helped confirm that Snort was not only monitoring traffic but also capable of generating alerts based on custom detection rules.

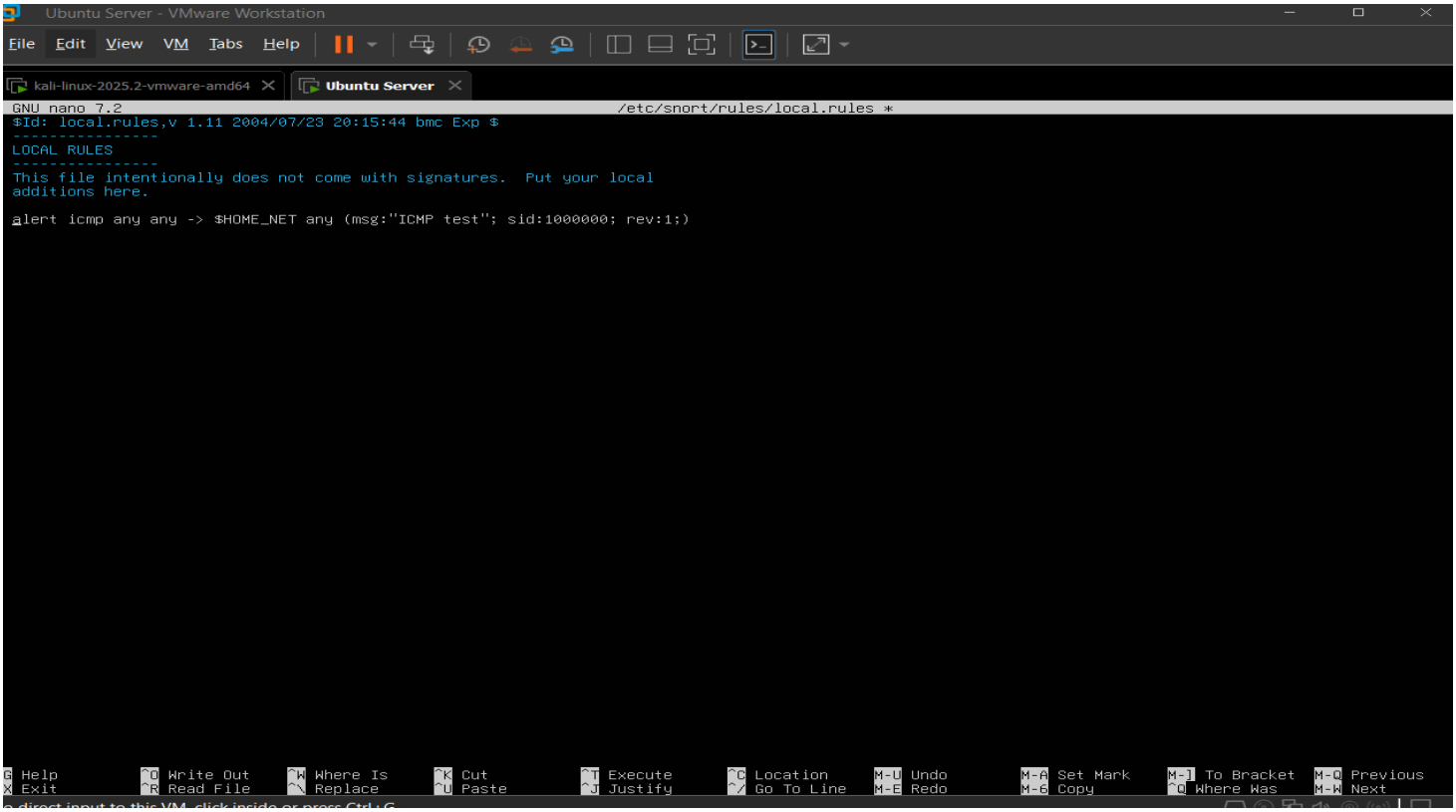
Step 1: Configuring the Snort Rule

A custom rule was added to Snort's configuration to specifically detect ICMP traffic. The rule is as follows:

```
alert icmp any any -> $HOME_NET any (msg:"ICMP test"; sid:1000000; rev:1;)
```

- **alert icmp** → Tells Snort to trigger on ICMP packets.
- **any any** → Matches traffic from any source IP and port.
- **$HOME_NET any** → Targets any destination IP within the defined home network.
- **msg:"ICMP test"** → Message displayed when the rule is triggered.
- **sid:1000000** → Unique Snort ID for the rule.
- **rev:1** → Revision number for rule tracking.

[Screenshot]



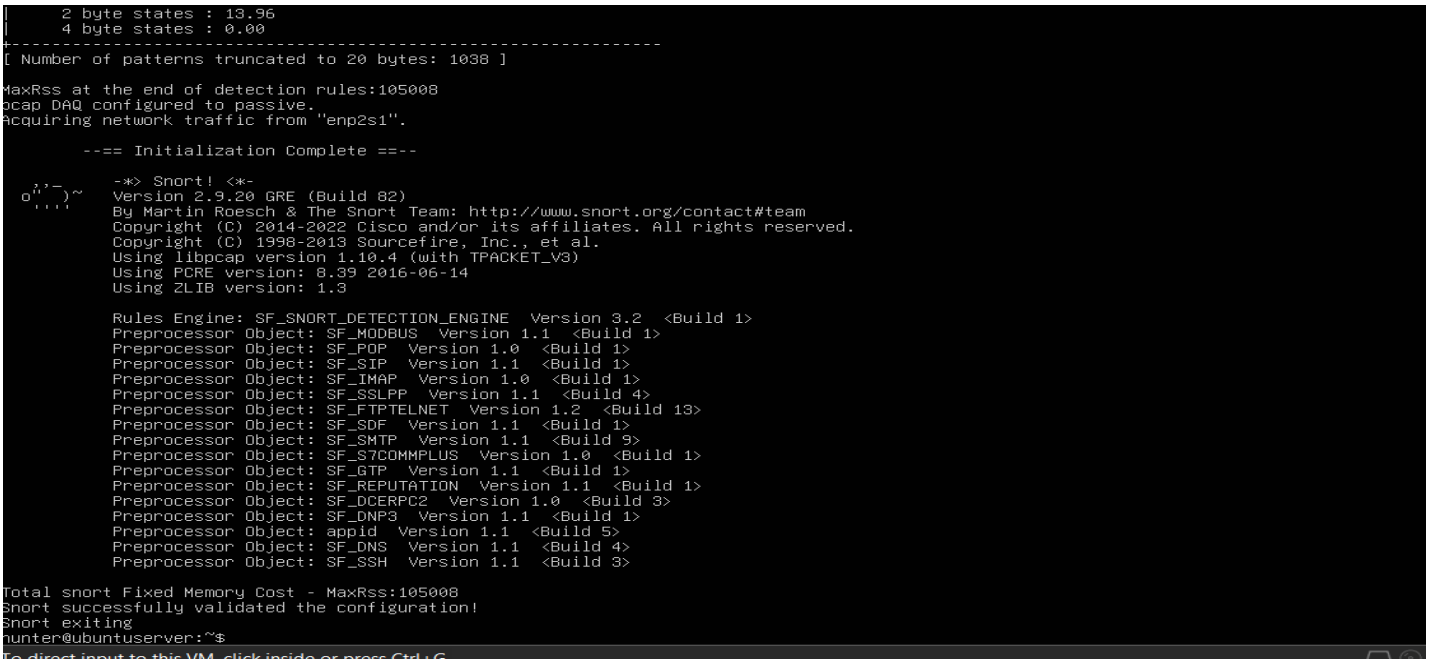
Step 3: Verifying Snort Baseline Functionality

To confirm that Snort was operational, I generated **normal traffic** between the two machines:

- **Ping Test:** Sending ICMP echo requests from Kali to Ubuntu.
- **HTTP Request:** Using `curl` from Kali to request a web page from Ubuntu.

Snort successfully captured these activities, proving that it was actively monitoring the traffic. This step was crucial for establishing a **baseline of normal network behavior** before moving forward with attack simulations.

[Screenshot]



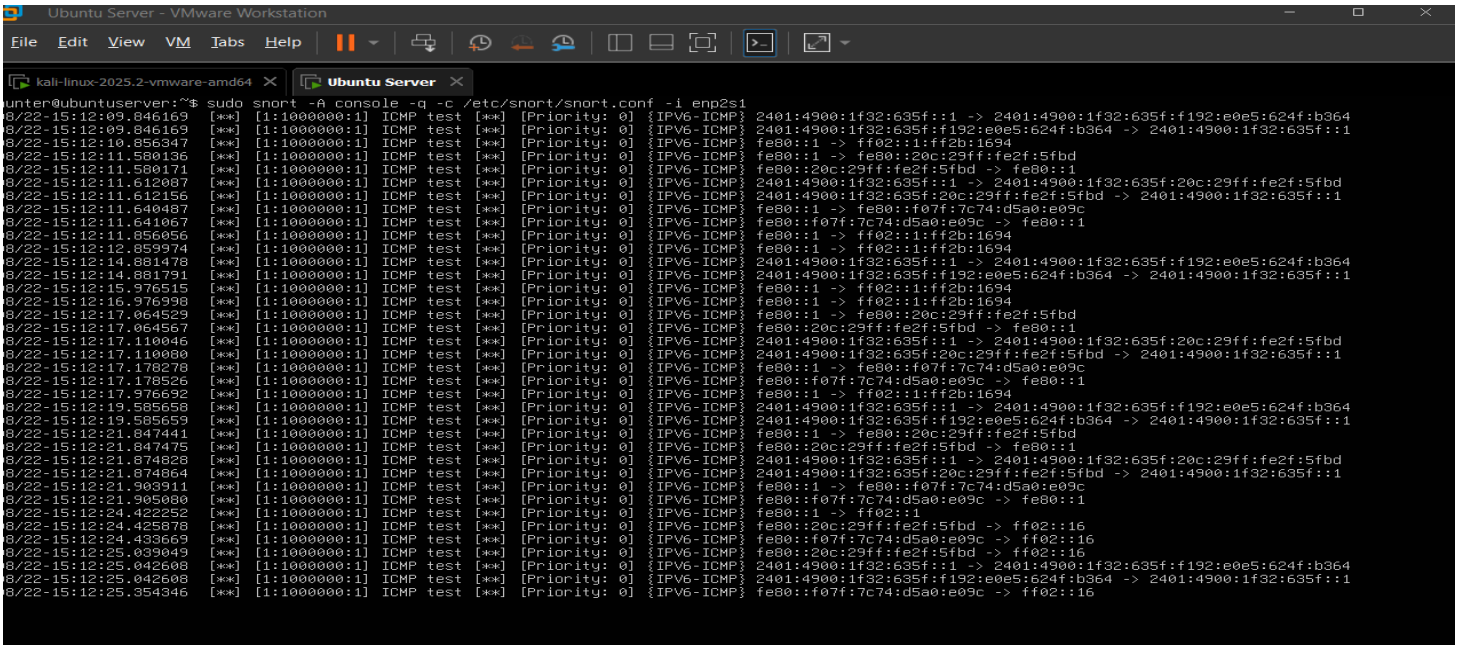
Step 2: Running Snort on the Target (Ubuntu Server)

Snort was started in IDS mode to listen on the network interface (`enp0s3`) and generate alerts when matching traffic was detected:

```
sudo snort -A console -q -c /etc/snort/snort.conf -i enp2s1
```

- **-A console** → Displays alerts in the console.
- **-q** → Quiet mode (suppresses extra output).
- **-c** → Specifies the Snort configuration file.
- **-i enp0s3** → Defines the network interface to listen on.

[Screenshot: Snort running and waiting for traffic]

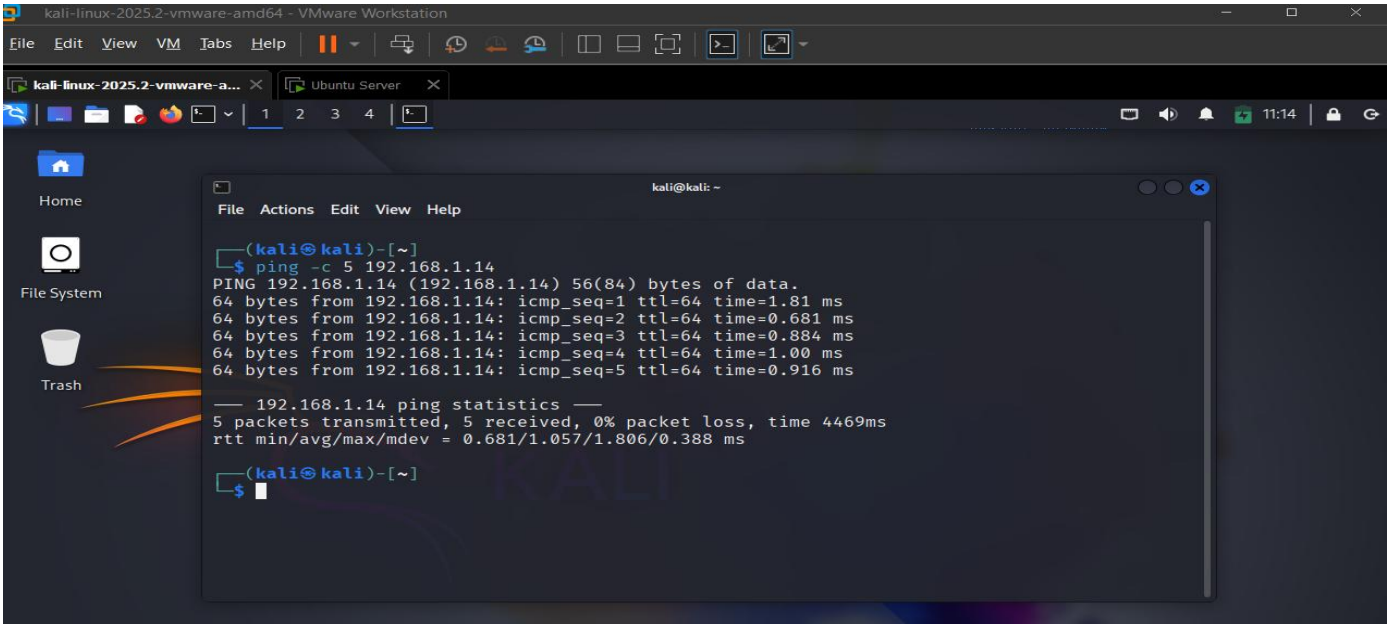


Step 3: Generating ICMP Traffic from the Attacker (Kali Linux)

From the Kali Linux machine, a series of **five ICMP echo requests** were sent to the Ubuntu server:

```
ping -c 5 192.168.1.14/24
```

[Screenshot: Ping results from Kali to Ubuntu]

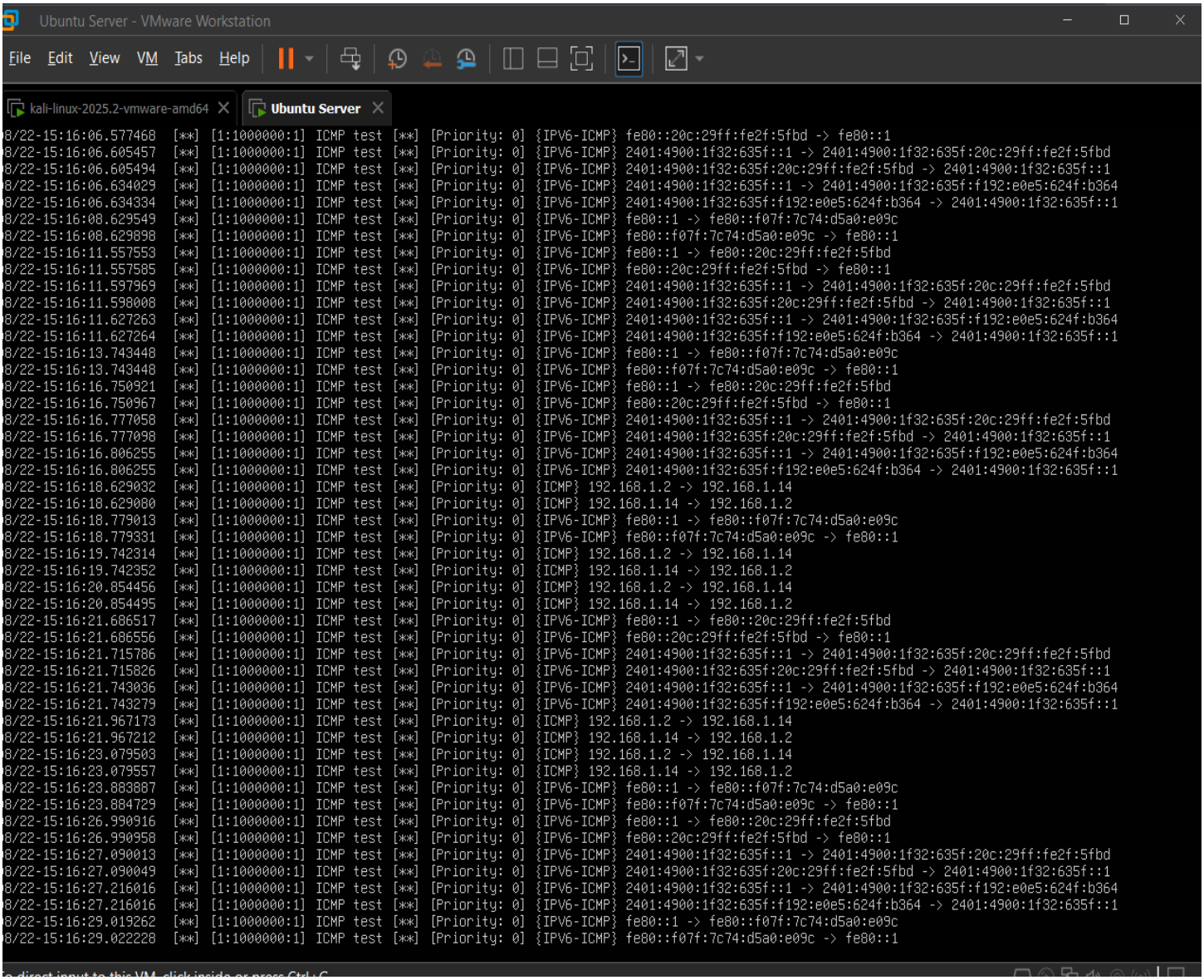


Step 4: Observing the Snort Alerts

On the Ubuntu server, Snort successfully detected the ping traffic and generated alerts in real-time. The alert messages included the custom message defined in the rule:

```
[**] [1:1000000:1] ICMP test [**]
```

📸 *[Screenshot: Snort console displaying ICMP alerts]*



✅ **Expected Result:** Snort generated alerts with the message **"ICMP test"** each time the ping request was detected.

✅ **Outcome:** This confirmed that Snort's custom rule engine was working correctly and capable of identifying ICMP echo requests on the network.

✅ **Outcome:** Snort was installed and confirmed to be functional. Both the Ubuntu Server (target) and Kali Linux (attacker) machines were configured properly and could communicate within the lab environment.

Week 2: Detecting Nmap Scans

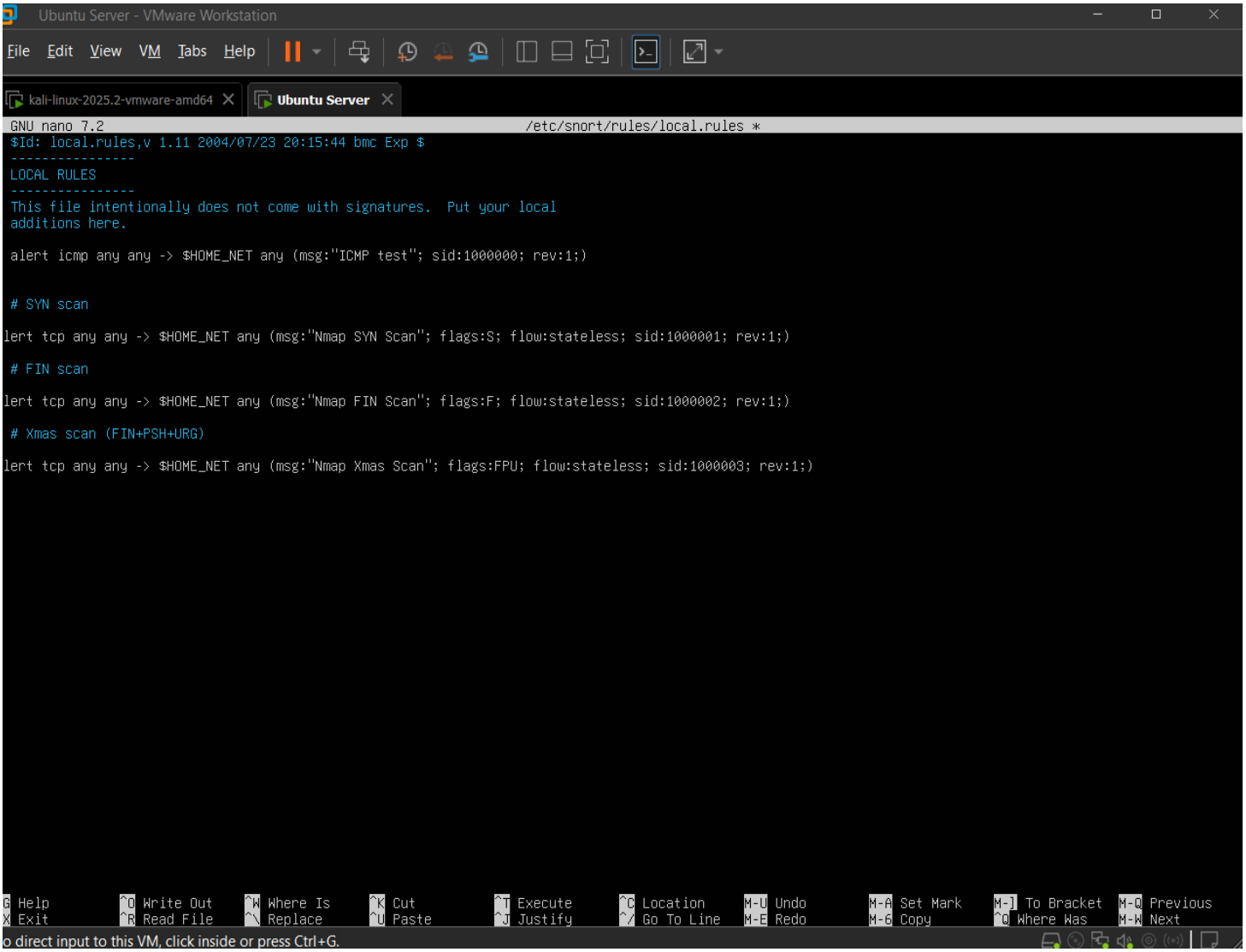
In the second week, the focus was on detecting **reconnaissance activities** performed with Nmap. Since port scanning is one of the most common techniques attackers use to gather information about a target system, Snort was configured with custom rules to detect various types of **TCP scans**. Specifically, the rules were designed to identify **SYN scans, FIN scans, and Xmas scans**.

Step 1: Writing Custom Snort Rules

The following rules were added to detect different Nmap scan techniques:

```
# Detects SYN Scan
alert tcp any any -> $HOME_NET any (msg:"Nmap SYN Scan"; flags:S; flow:stateless;
sid:1000001; rev:1;)

# Detects FIN Scan
alert tcp any any -> $HOME_NET any (msg:"Nmap FIN Scan"; flags:F; flow:stateless;
sid:1000002; rev:1;)

# Detects Xmas Scan
alert tcp any any -> $HOME_NET any (msg:"Nmap Xmas Scan"; flags:FPU; flow:stateless;
sid:1000003; rev:1;)
```

[Screenshot]



🔹 **Explanation of key components:**

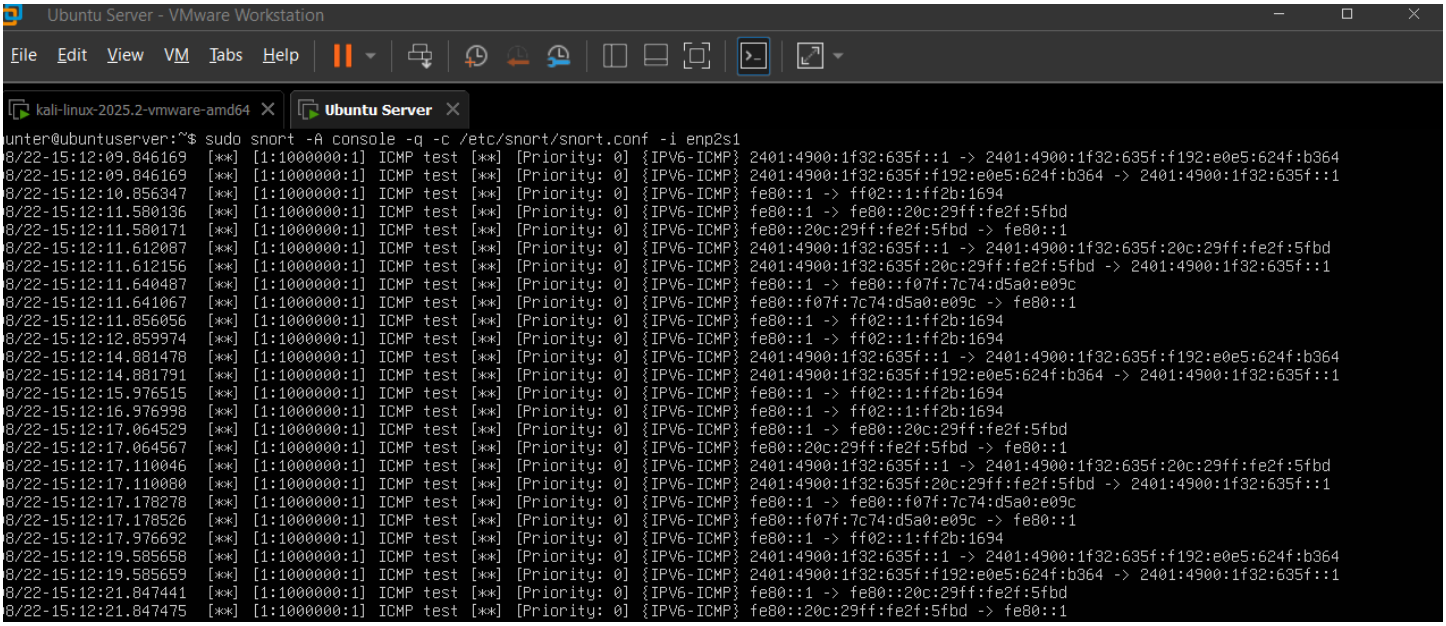
- **flags:S** → Matches TCP packets with only the SYN flag set (common in SYN scans).
- **flags:F** → Matches packets with only the FIN flag set (used in stealth FIN scans).
- **flags:FPU** → Matches packets with FIN, PSH, and URG flags set (used in Xmas scans).
- **flow:stateless** → Ensures detection works even without tracking full TCP sessions.
- **sid** → Unique Snort ID for each rule.

Step 2: Running Snort on the Target (Ubuntu Server)

On the Ubuntu server, Snort was started in IDS mode to monitor the **enp2s1** network interface:

```
sudo snort -A console -q -c /etc/snort/snort.conf -i enp2s1
```

📸 *[Screenshot: Snort running and waiting for traffic]*



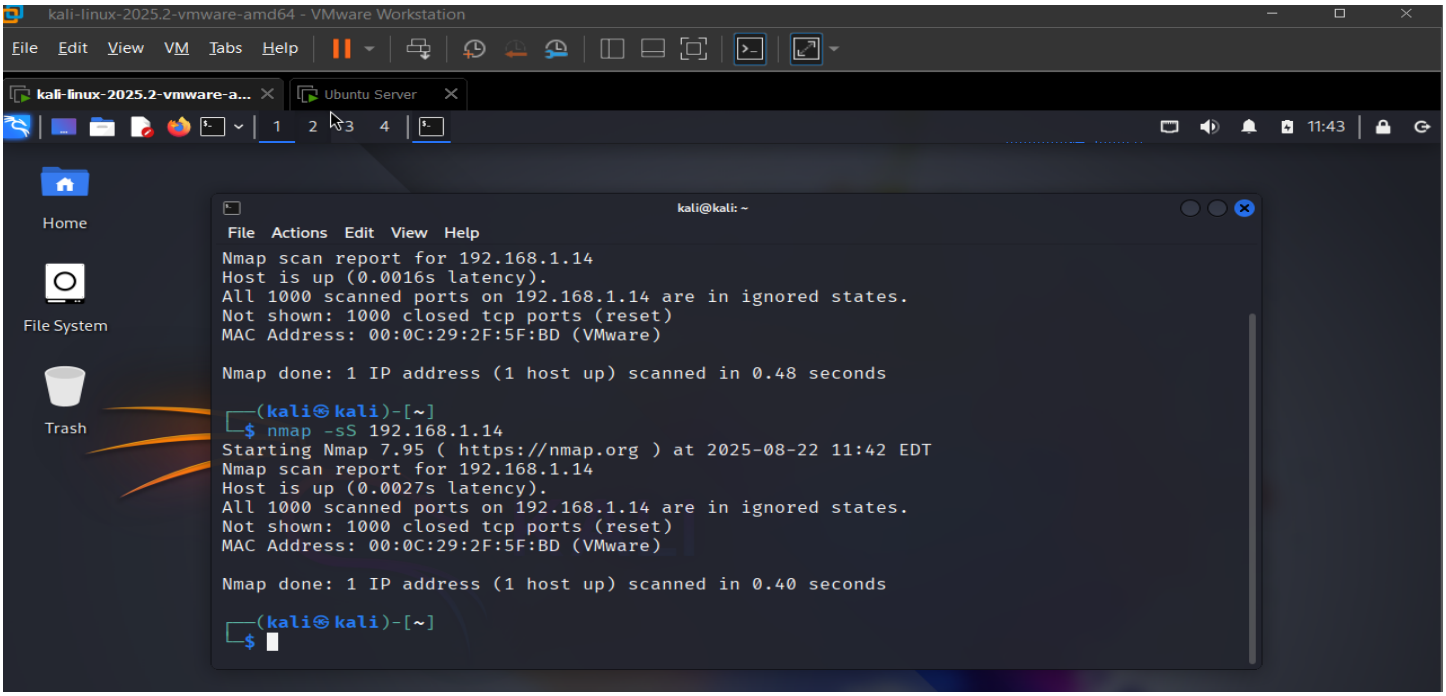
Step 3: Launching Nmap Scans from the Attacker (Kali Linux)

On the Kali machine, different scan types were executed against the Ubuntu target:

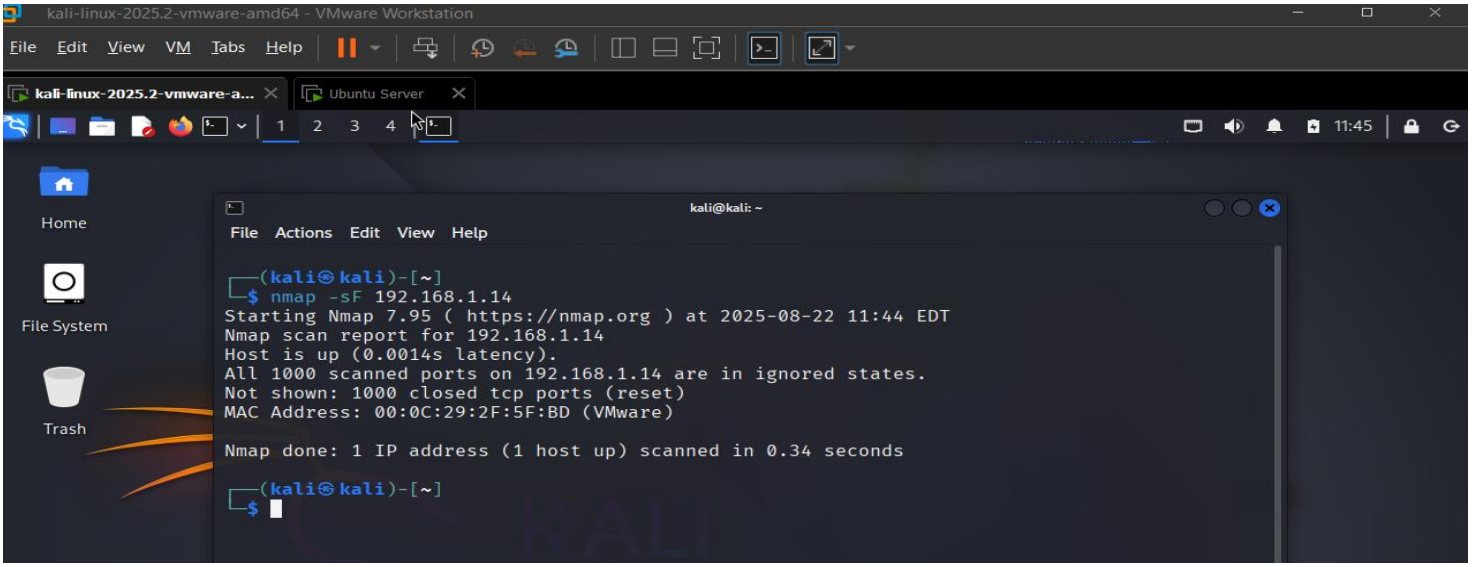
```
# SYN Scan
nmap -sS 192.168.1.14

# FIN Scan
nmap -sF 192.168.1.14

# Xmas Scan
nmap -sX 192.168.1.14
```

📸 *[Screenshot: Nmap scans results from Kali]*

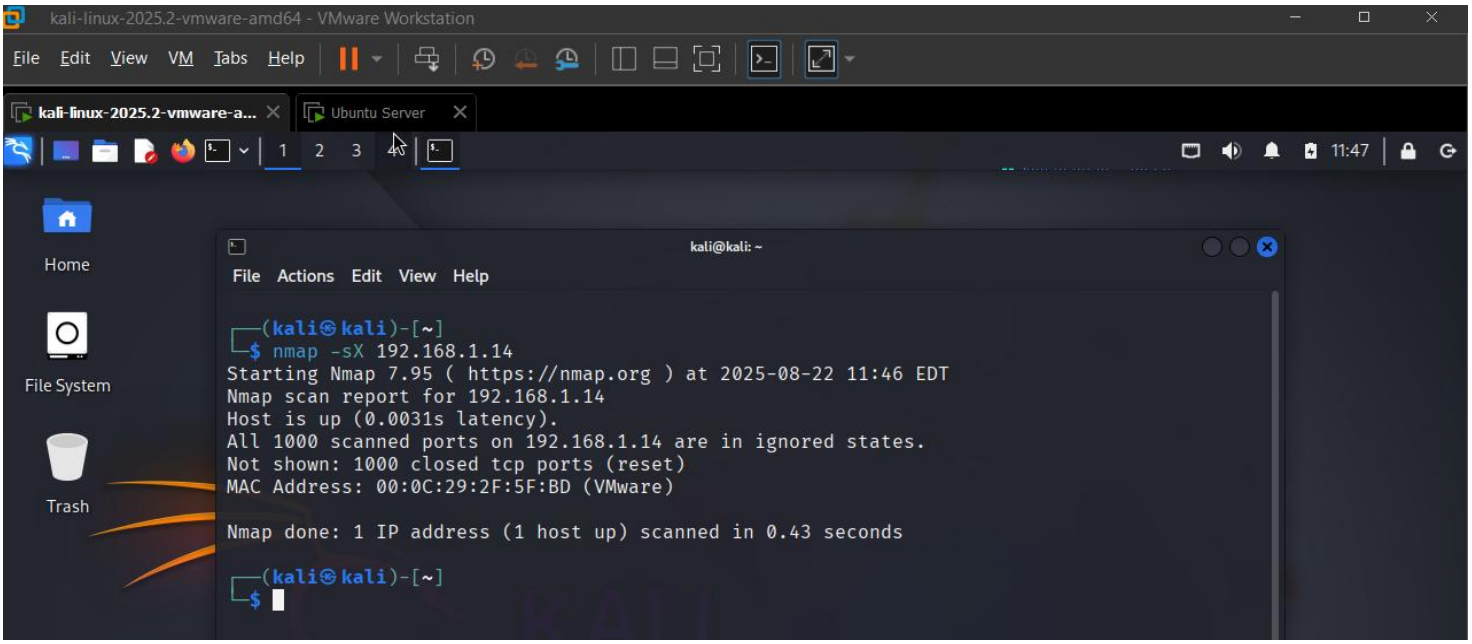
SYN Scan



FIN Scan



Xmas Scan



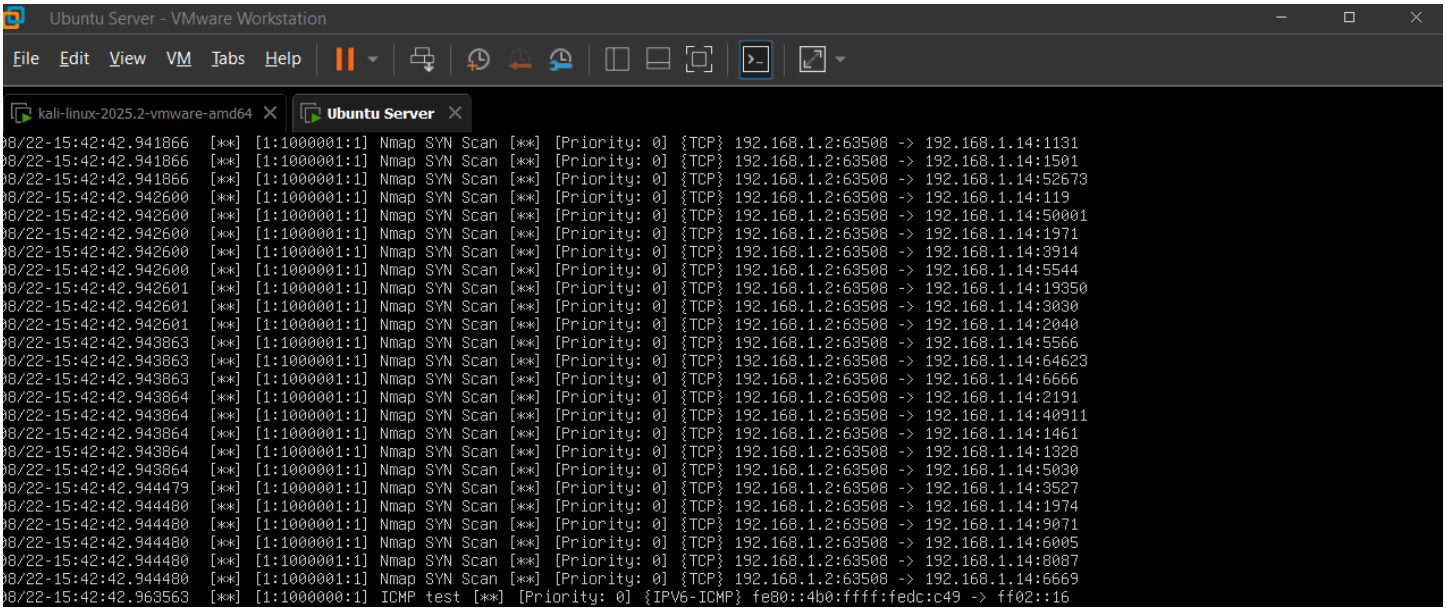
Step 4: Observing Snort Alerts

While the scans were running, Snort generated alerts in the console. Each alert displayed the **custom message** based on the rule triggered:

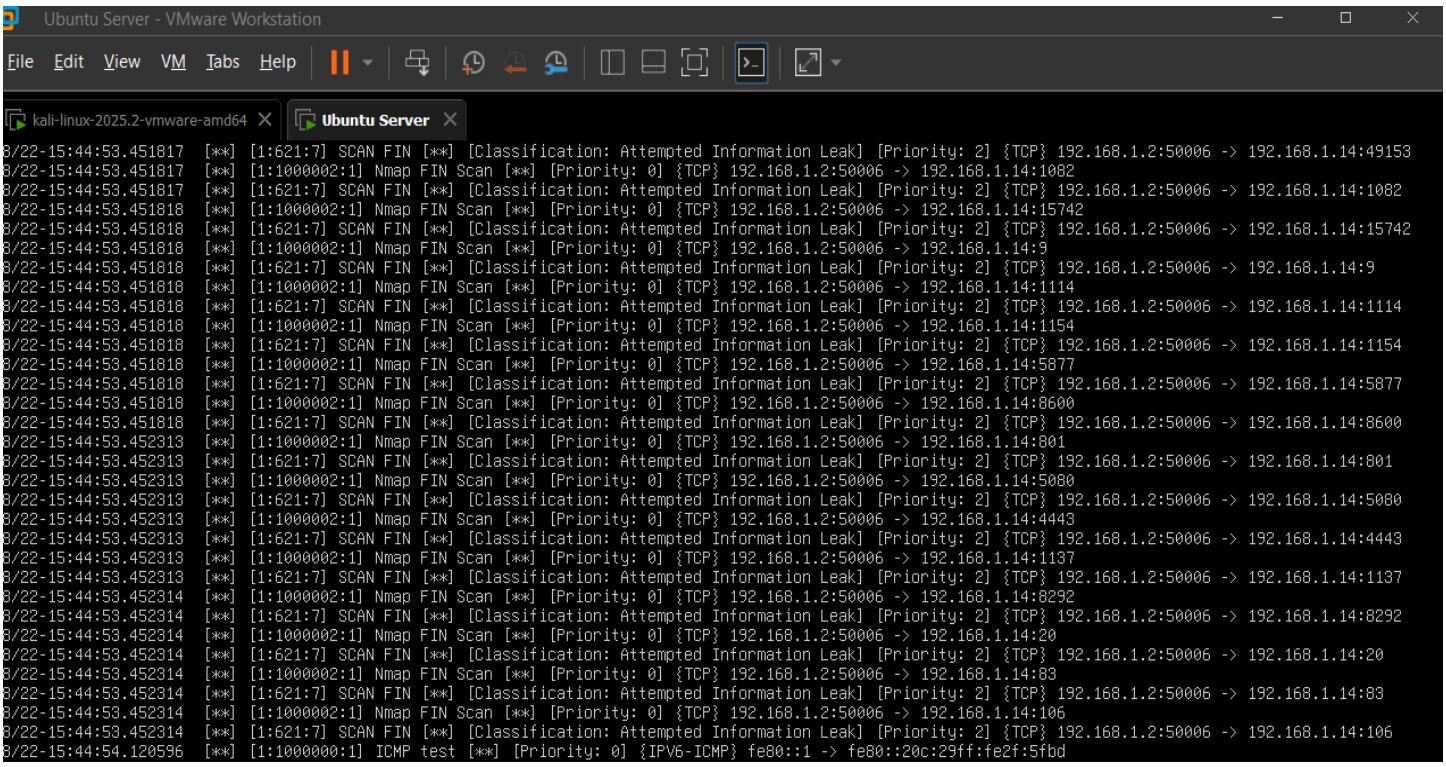
```
[**] [1:1000001:1] Nmap SYN Scan [**]
[**] [1:1000002:1] Nmap FIN Scan [**]
[**] [1:1000003:1] Nmap Xmas Scan [**]
```

[Screenshot: Snort console showing SYN, FIN, and Xmas scan alerts]

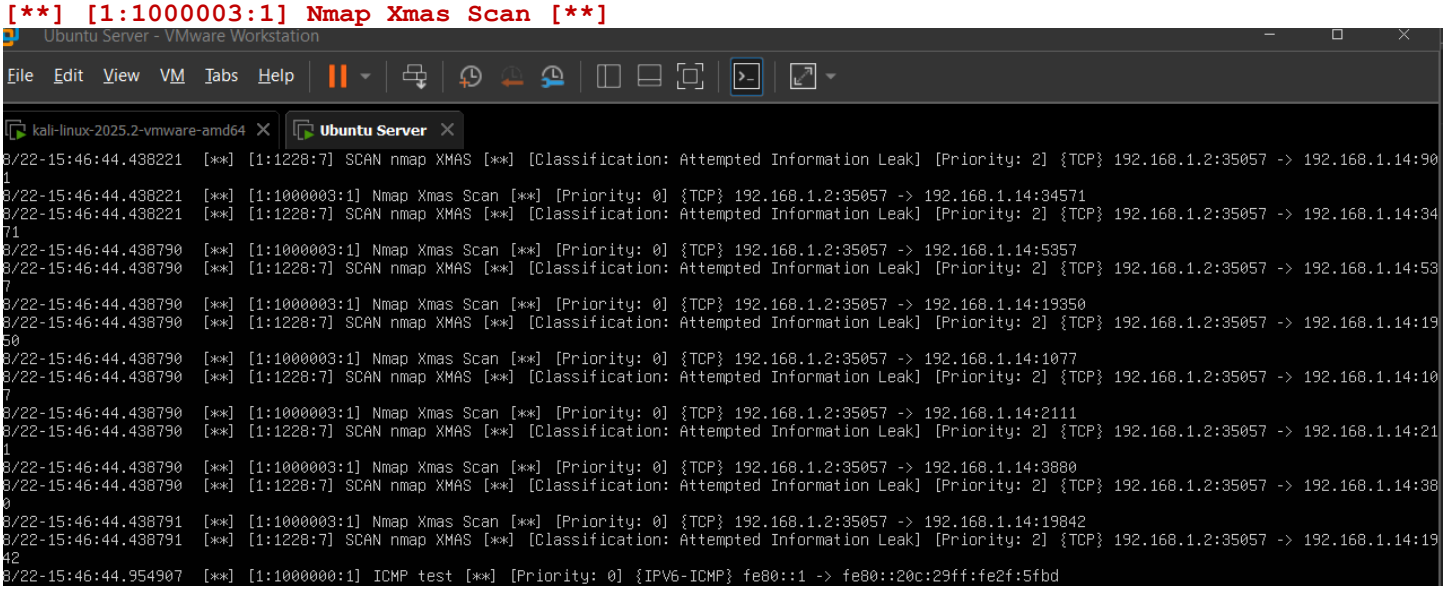
[**] [1:1000001:1] Nmap SYN Scan [**]



[**] [1:1000002:1] Nmap FIN Scan [**]



```
[**] [1:1000003:1] Nmap Xmas Scan [**]
```



🎯 Outcome

This test confirmed that Snort was able to detect **different scanning techniques** used by Nmap. The results demonstrated that custom rules could effectively identify reconnaissance attempts, an essential step in preventing potential intrusions.

Week 3: Detecting SSH Brute-Force Attempts with Snort

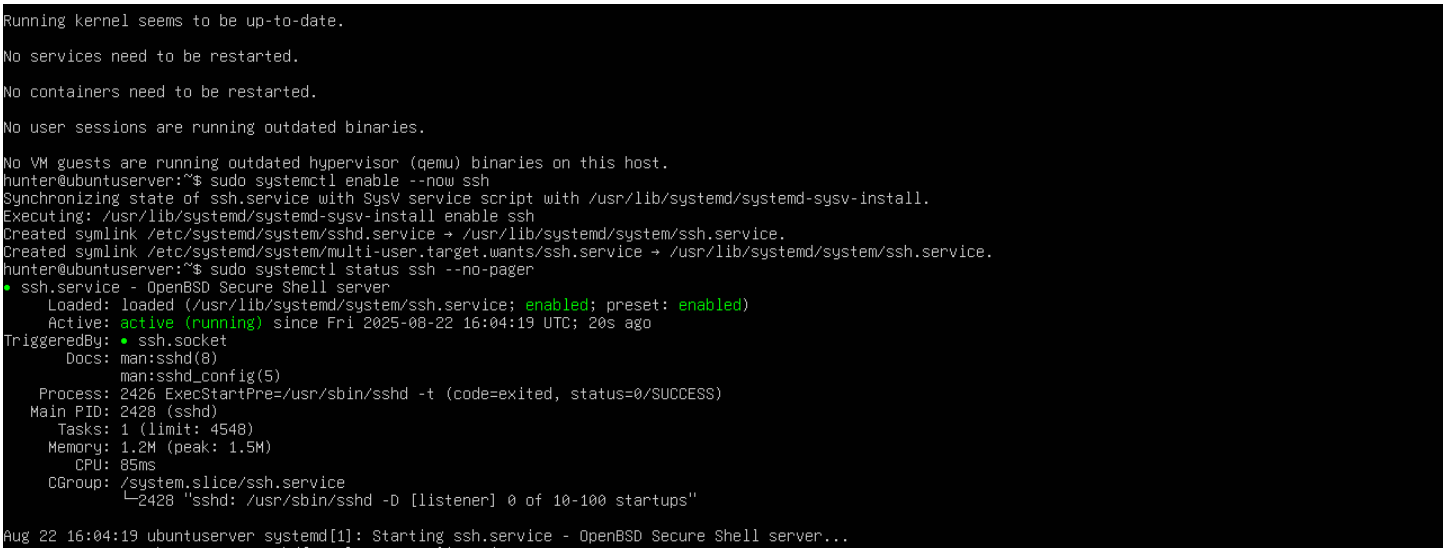
This week's focus was on identifying brute-force login attempts against an SSH service using **Snort**. To accomplish this, I set up an OpenSSH server on the target machine, wrote a custom Snort rule to monitor repeated login attempts, and then simulated an actual brute-force attack using the tool **Hydra**.

1. Environment Setup

- **Target Machine (Ubuntu):**
  - Installed and enabled the OpenSSH server:

```
sudo apt install -y openssh-server
sudo systemctl enable --now ssh
```

Verified the SSH service was running and listening on port **22**.



2. Snort Rule for SSH Brute-Force Detection

A custom Snort rule was written to detect multiple login attempts from the same source within a short time frame. The idea was to raise an alert if more than five SSH connection attempts occurred within 60 seconds.

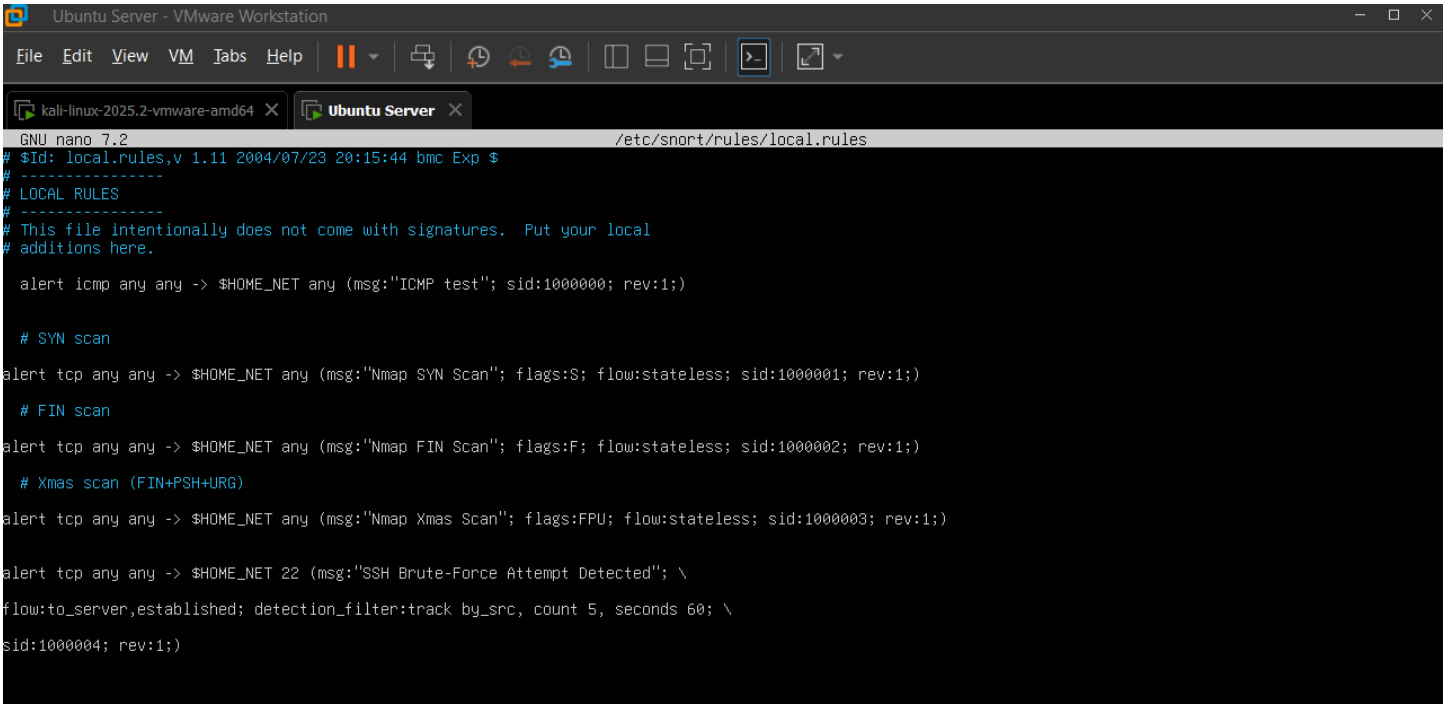
Rule:

```
alert tcp any any -> $HOME_NET 22 (msg:"SSH Brute-Force Attempt Detected";
flow:to_server,established;
detection_filter:track by_src, count 5, seconds 60;
sid:1000004; rev:1;)
```

Explanation of key parameters:

- **flow:to_server,established** → Ensures Snort only monitors established TCP sessions going to the server.
- **detection_filter** → Triggers an alert if 5 attempts are observed within 60 seconds from the same source IP.
- **sid:1000004** → Unique Snort rule ID.
- **msg** → Human-readable alert message.

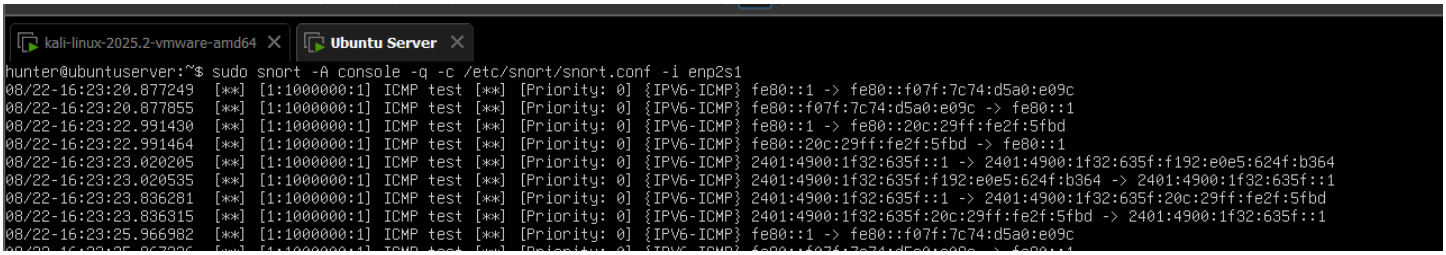
[Screenshot]



Snort was started with the following command to monitor traffic on the target machine:

```
sudo snort -A console -q -c /etc/snort/snort.conf -i enp2s1
```

[Screenshot]



3. Attacking Machine (Kali Linux):

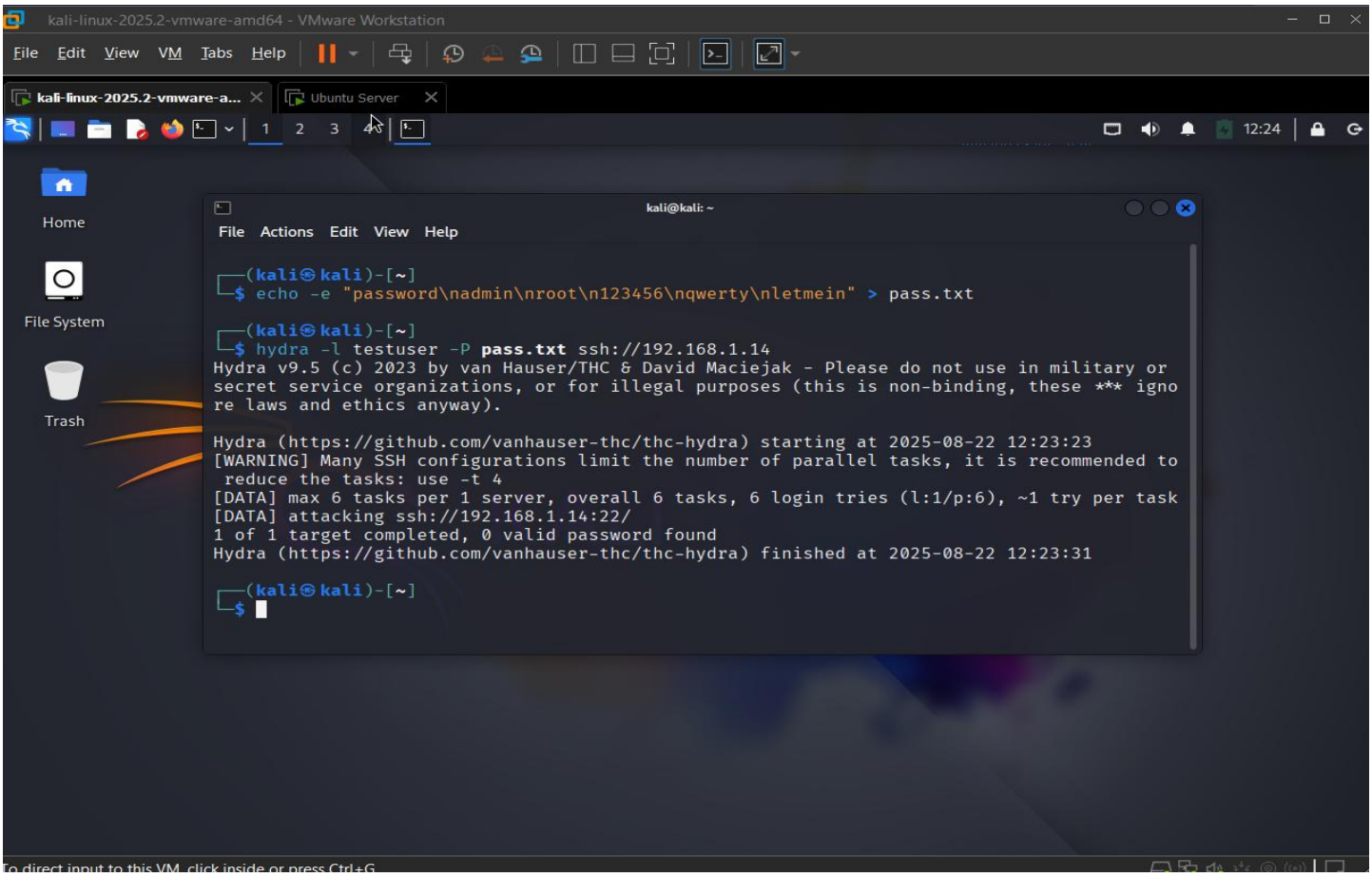
- Created a simple password list containing common weak credentials:

```
echo -e "password\nadmin\nroot\n123456\nqwerty\nletmein" > pass.txt
```

- Used **Hydra** to simulate a brute-force attack against the target:

```
hydra -l testuser -P pass.txt ssh://192.168.1.14
```

[Screenshot]



4. Attack Simulation

Using Hydra, the attacker repeatedly attempted to authenticate into the SSH server with the username **testuser** and a list of common passwords from **pass.txt**. Each attempt generated traffic towards the SSH service, which Snort analyzed in real-time.

5. Observing SSH Brute-Force Alerts

Whenever multiple SSH login attempts were detected within the defined time window (5 attempts in 60 seconds), Snort generated the alert:

```
SSH Brute-Force Attempt Detected
```

This confirmed that Snort was successfully configured to recognize brute-force activity targeting SSH.

[Screenshot: Snort console showing "SSH Brute-Force Attempt Dtected" alert]

```
08/22-16:23:26.913612  [**] [1:1000001:1] Nmap SYN Scan [**] [Priority: 0] {TCP} 192.168.1.2:56980 -> 192.168.1.14:22
08/22-16:23:26.915262  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56962 -> 192.168.1.14:22
08/22-16:23:26.915262  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56980 -> 192.168.1.14:22
08/22-16:23:26.915262  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56978 -> 192.168.1.14:22
08/22-16:23:26.966918  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56952 -> 192.168.1.14:22
08/22-16:23:26.984926  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56956 -> 192.168.1.14:22
08/22-16:23:26.984705  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56962 -> 192.168.1.14:22
08/22-16:23:26.993268  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56978 -> 192.168.1.14:22
08/22-16:23:26.989668  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56968 -> 192.168.1.14:22
08/22-16:23:26.995405  [**] [1:1000004:1] SSH Brute-Force Attempt Detected [**] [Priority: 0] {TCP} 192.168.1.2:56980 -> 192.168.1.14:22
08/22-16:23:27.605347  [**] [1:1000000:1] ICMP test [**] [Priority: 0] {IPV6-ICMP} fe80::1 -> fe80::20c:29ff:fe2f:5fbd
08/22-16:23:27.605382  [**] [1:1000000:1] ICMP test [**] [Priority: 0] {IPV6-ICMP} fe80::20c:29ff:fe2f:5fbd -> fe80::1
08/22-16:23:27.642974  [**] [1:1000000:1] ICMP test [**] [Priority: 0] {IPV6-ICMP} 2401:4900:1f32:635f::1 -> 2401:4900:1f32:635f:f192:e0e5:624f:b364
```

6. FTP Brute-Force Detection

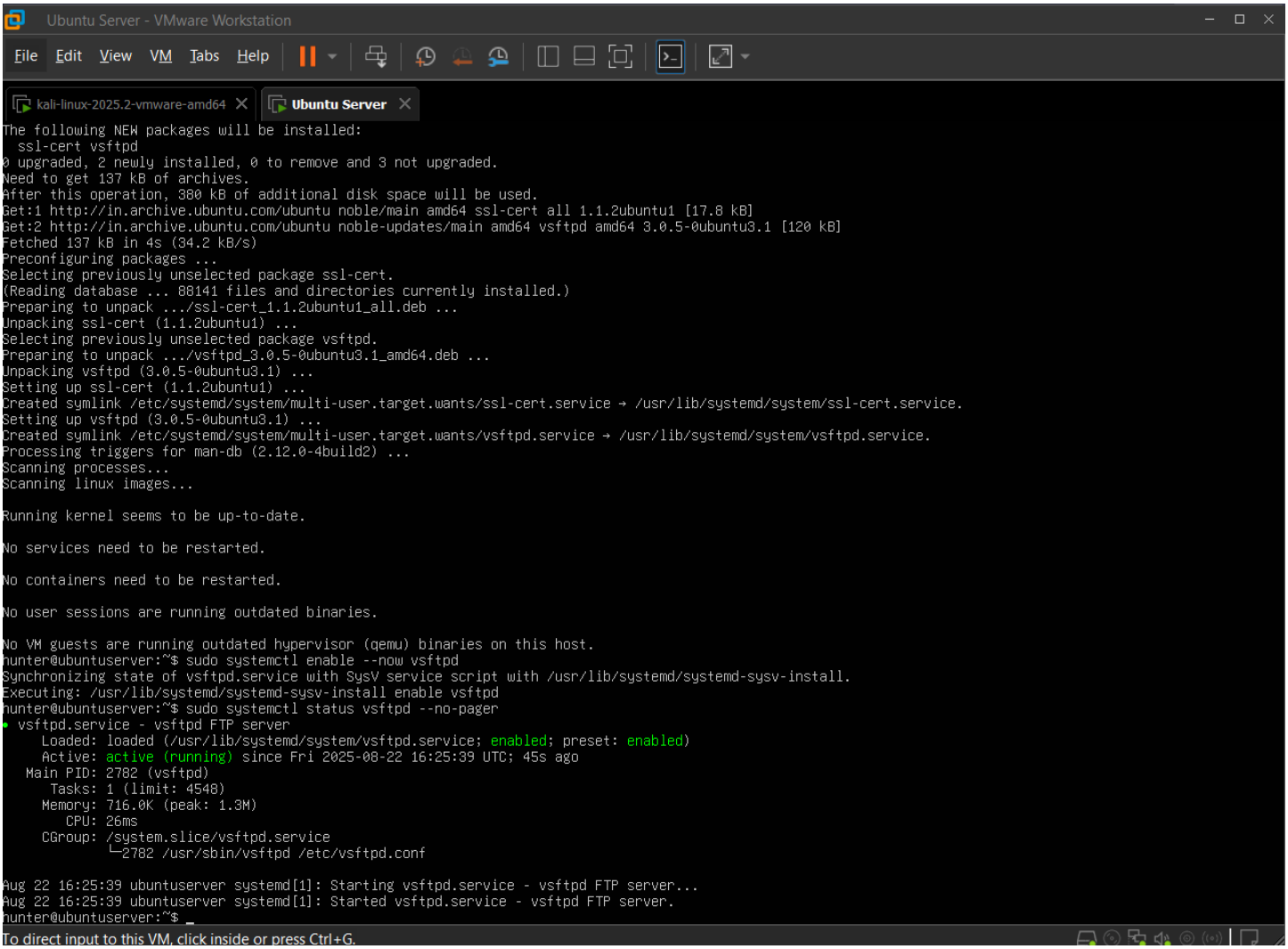
In this step, Snort was configured to detect brute-force login attempts against the **FTP service (port 21)**. Just like in the SSH brute-force lab, the idea was to raise an alert if multiple login attempts were observed within a short period of time.

1. Environment Setup

- **Target Machine (Ubuntu):**
  - Install and enable the **vsftpd** FTP server:

```
sudo apt update
sudo apt install -y vsftpd
sudo systemctl enable --now vsftpd
```

[Screenshot: vsftpd installation and running service on Ubuntu]

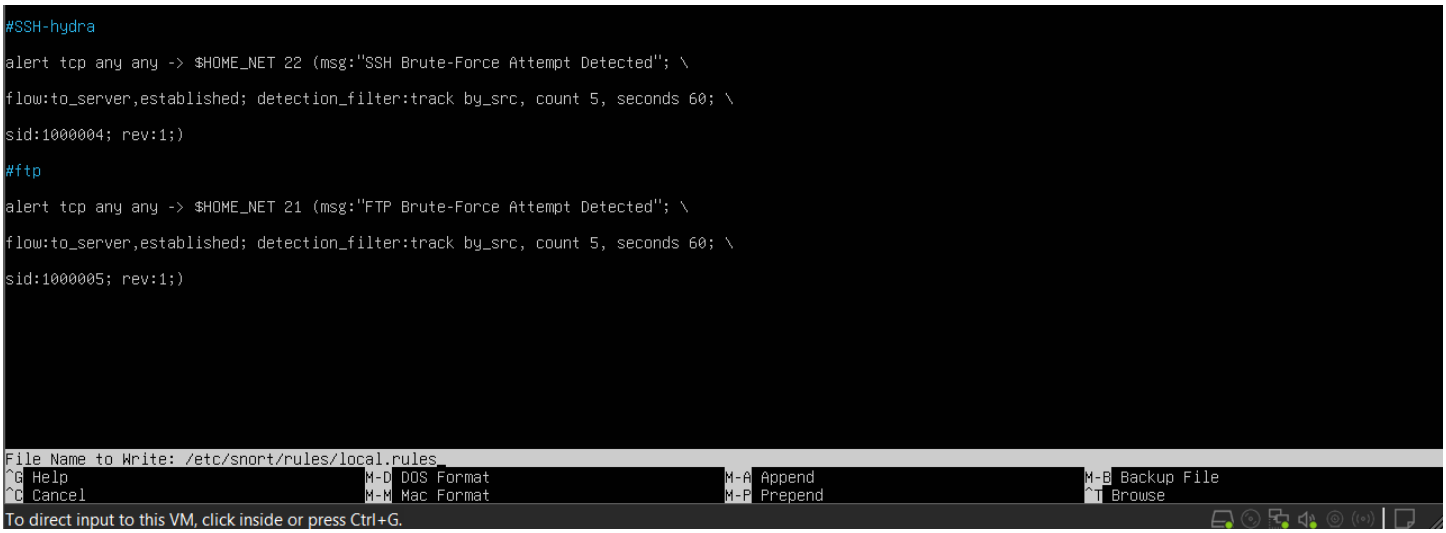


Snort Rule

```
alert tcp any any -> $HOME_NET 21 (msg:"FTP Brute-Force Attempt Detected";
flow:to_server,established; detection_filter:track by_src, count 5, seconds 60;
sid:1000005; rev:1;)
```

- **Port 21** → Targets FTP service.
- **flow:to_server,established** → Ensures rule only checks active connections going to the FTP server.
- **detection_filter** → Triggers if 5 attempts occur from the same source within 60 seconds.
- **msg** → Human-readable alert.

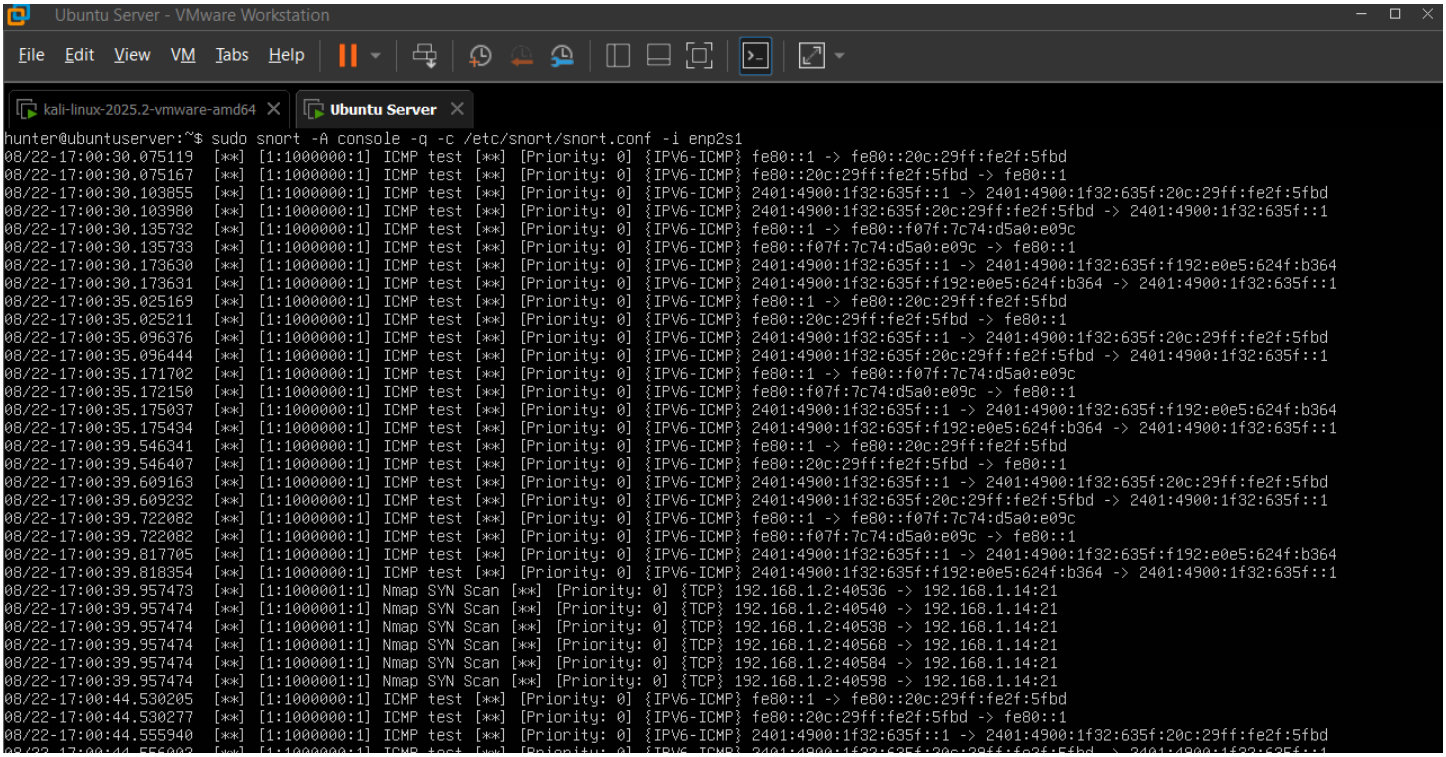
[Screenshot: FTP brute-force rule added in `local.rules`]



Started Snort in console alert mode to monitor FTP traffic:

```
sudo snort -A console -q -c /etc/snort/snort.conf -i enp2s1
```

[Screenshot: Snort running on Ubuntu target]



Attacker (Kali)

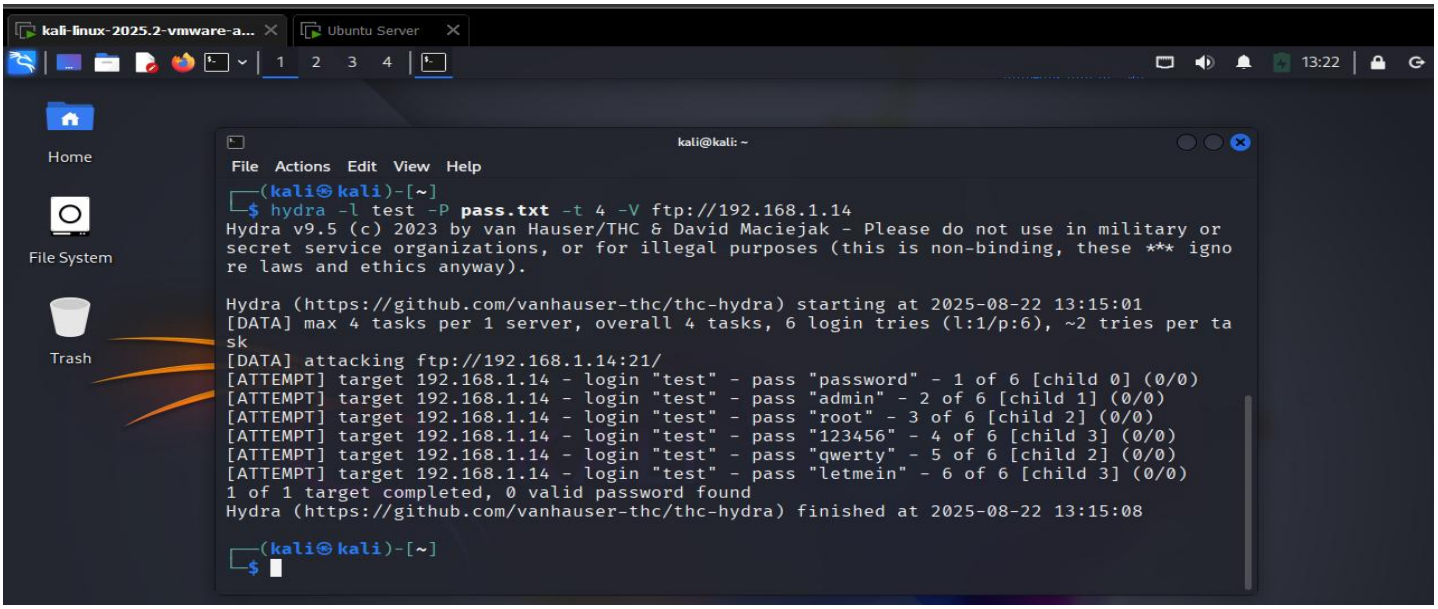
1. Created a simple password list:

   ```
   echo -e "password\nadmin\nroot\n123456\nqwerty\nletmein" > pass.txt
   ```

2. Launched brute-force attack against FTP service:

   ```
   hydra -l testuser -P pass.txt ftp://192.168.1.14
   ```

[Screenshot: Hydra brute-forcing FTP login attempts]

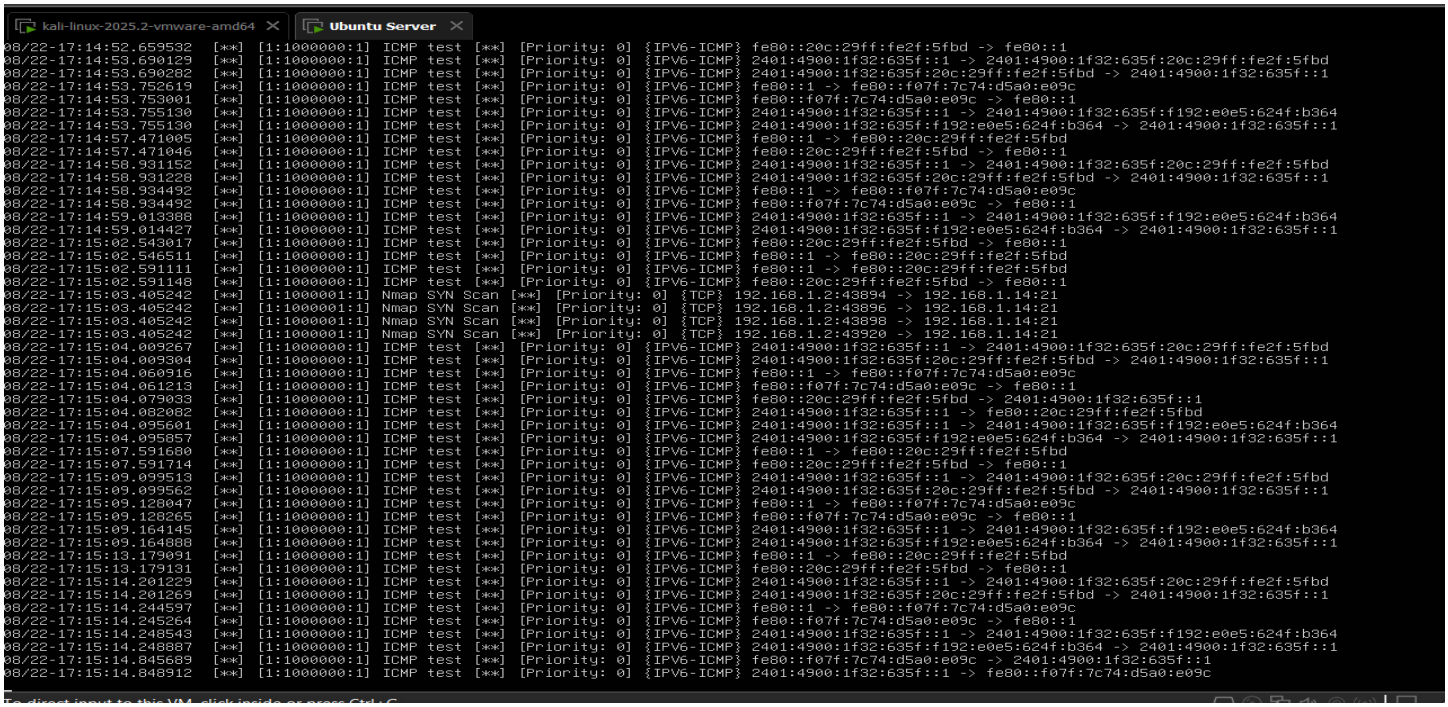


Expected Result

When Hydra made repeated login attempts, Snort detected the multiple failed connections and triggered the custom rule. The console displayed:

```
FTP Brute-Force Attempt Detected
```

[Screenshot: Snort console showing FTP brute-force alert]



👉 This confirmed that Snort successfully detected brute-force activity on FTP, similar to the SSH case, but this time monitoring **port 21** with the **vsftpd** service.

Week 4: Detecting Malware C2 Traffic

In this lab, Snort was configured to detect simulated **malware command-and-control (C2) beaconing traffic**. To simulate real-world beaconing, a script was used that periodically sent HTTP requests with a malicious domain in the Host header. Custom Snort rules were written to detect both the malicious domain name and the beacon URI.

1. Malware C2 Beacon Detection

Snort Rules

Two custom rules were created:

```
alert tcp any any -> $HOME_NET 80 (msg:"Malware C2 HTTP Host detected";
flow:to_server,established; content:"Host|3a 20|malicious-c2-server.com"; http_header;
sid:1000006; rev:1;)

alert tcp any any -> $HOME_NET 80 (msg:"Malware C2 Beacon URI detected";
flow:to_server,established; content:"/ping"; http_uri;
sid:1000007; rev:1;)
```

- **Rule 1 (Host header detection):** Looks for traffic with `Host: malicious-c2-server.com`.
- **Rule 2 (Beacon URI detection):** Triggers when `/ping` appears in the HTTP URI.

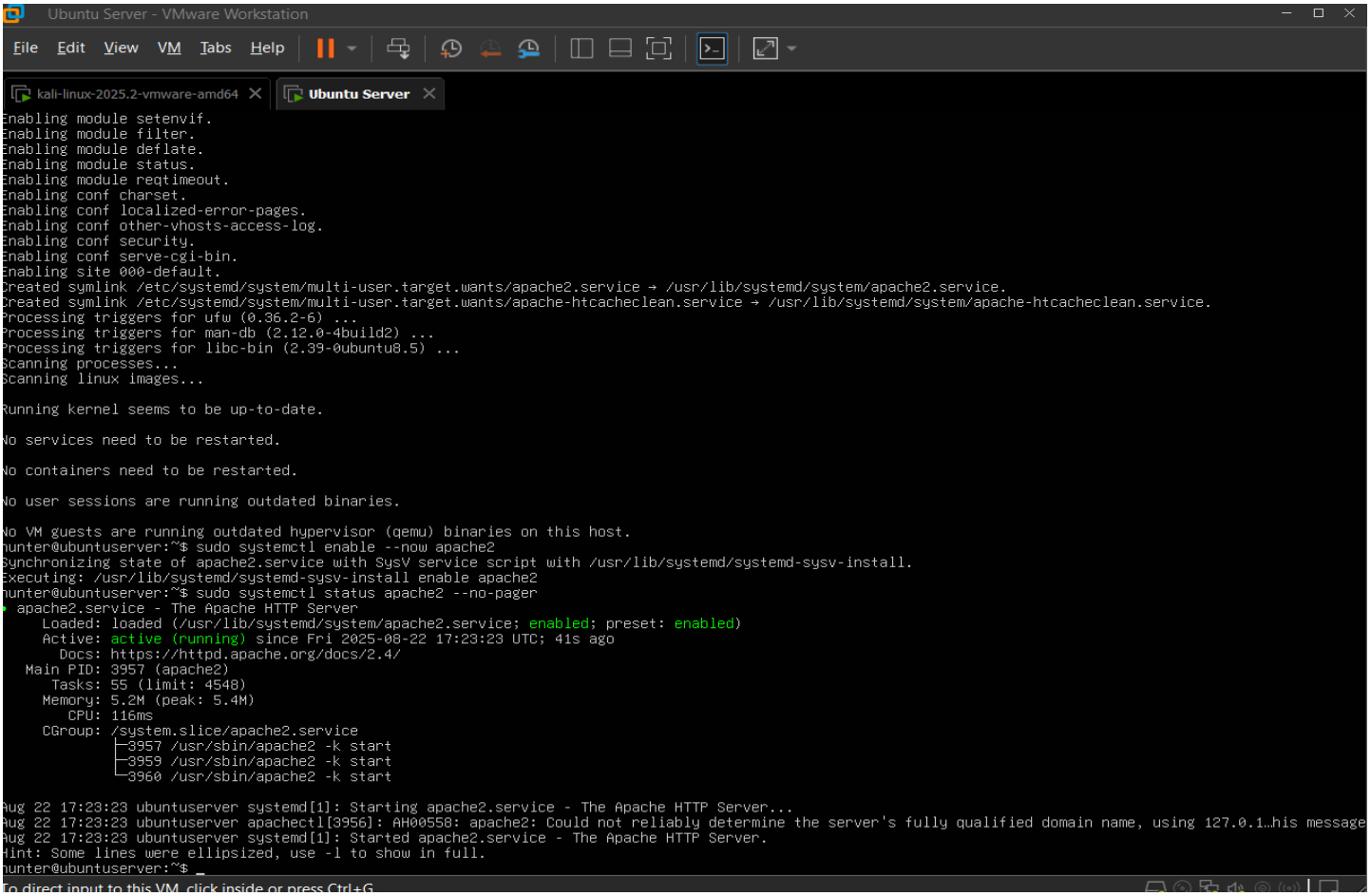
📸 *[Screenshot: local.rules file showing both C2 detection rules]*

Target Machine (Ubuntu)

1. Installed and enabled Apache web server to act as fake C2 server:

   ```
   sudo apt install -y apache2
   sudo systemctl enable --now apache2
   ```

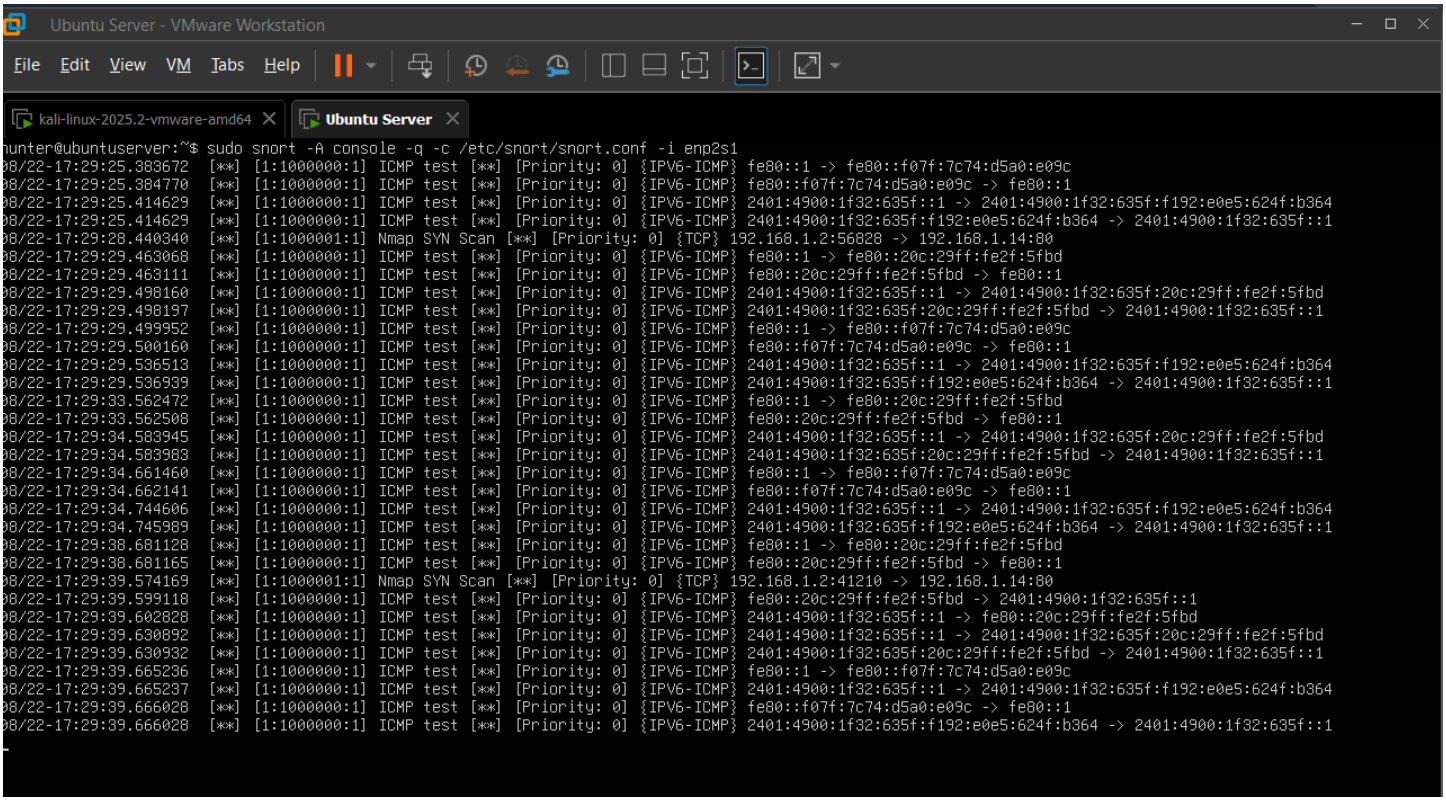
📸 *[Screenshot: Apache installed and running on Ubuntu]*



2. Started Snort in console mode:

```
sudo snort -A console -q -c /etc/snort/snort.conf -i enp2s1
```

[Screenshot: Snort running on Ubuntu target]



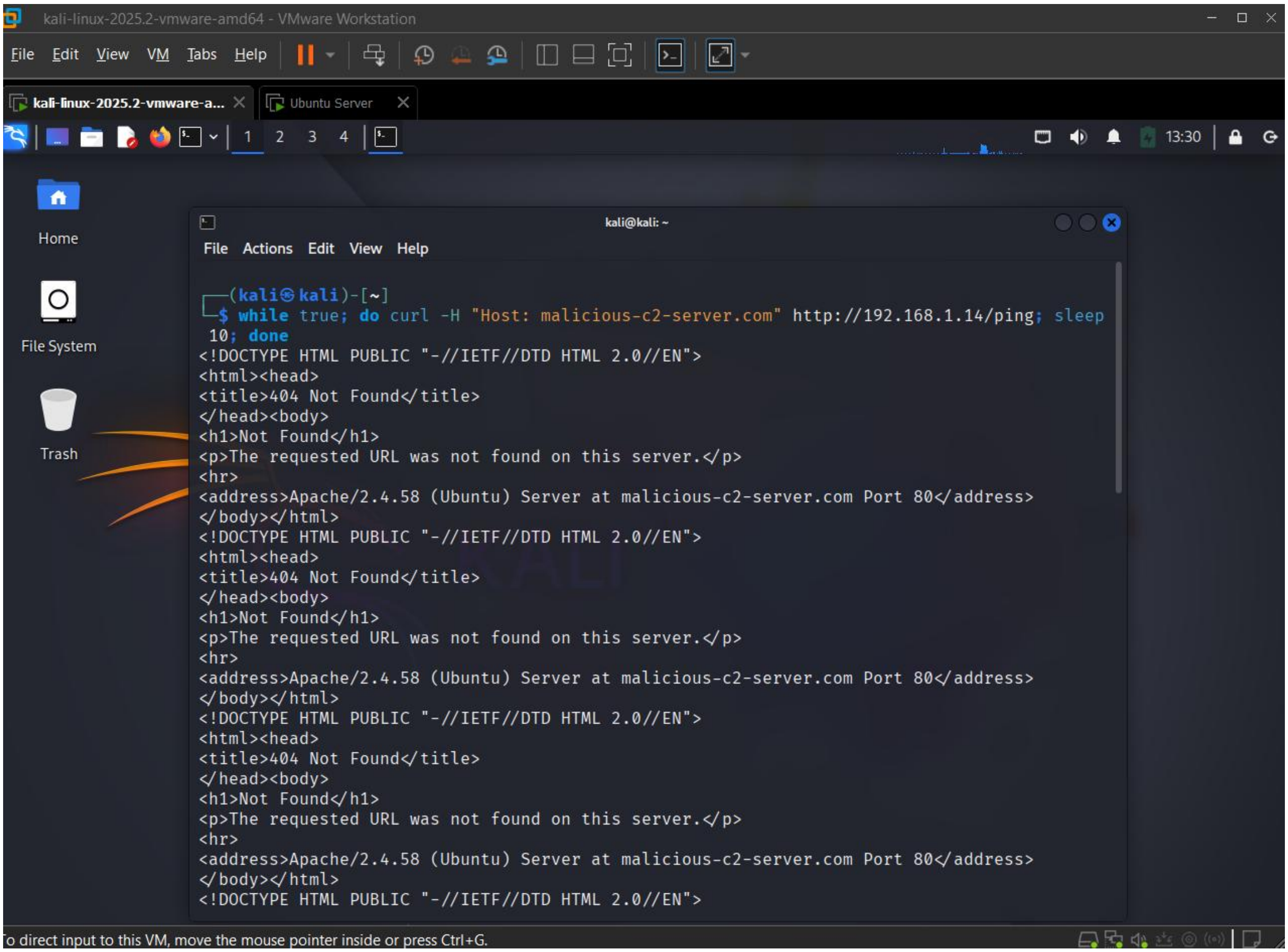
Attacker Machine (Kali)

Simulated beaconing traffic by sending repeated curl requests every 10 seconds:

```
while true; do curl -H "Host: malicious-c2-server.com" http://192.168.1.14/ping; sleep
10; done
```

- `-H "Host: malicious-c2-server.com"` → Adds fake malicious domain to HTTP header.
- `/ping` → Mimics periodic beacon URI.
- `sleep 10` → Waits 10 seconds before repeating (simulates beaconing interval).

📷 *[Screenshot: Kali terminal showing repeated curl requests to /ping with Host header]*

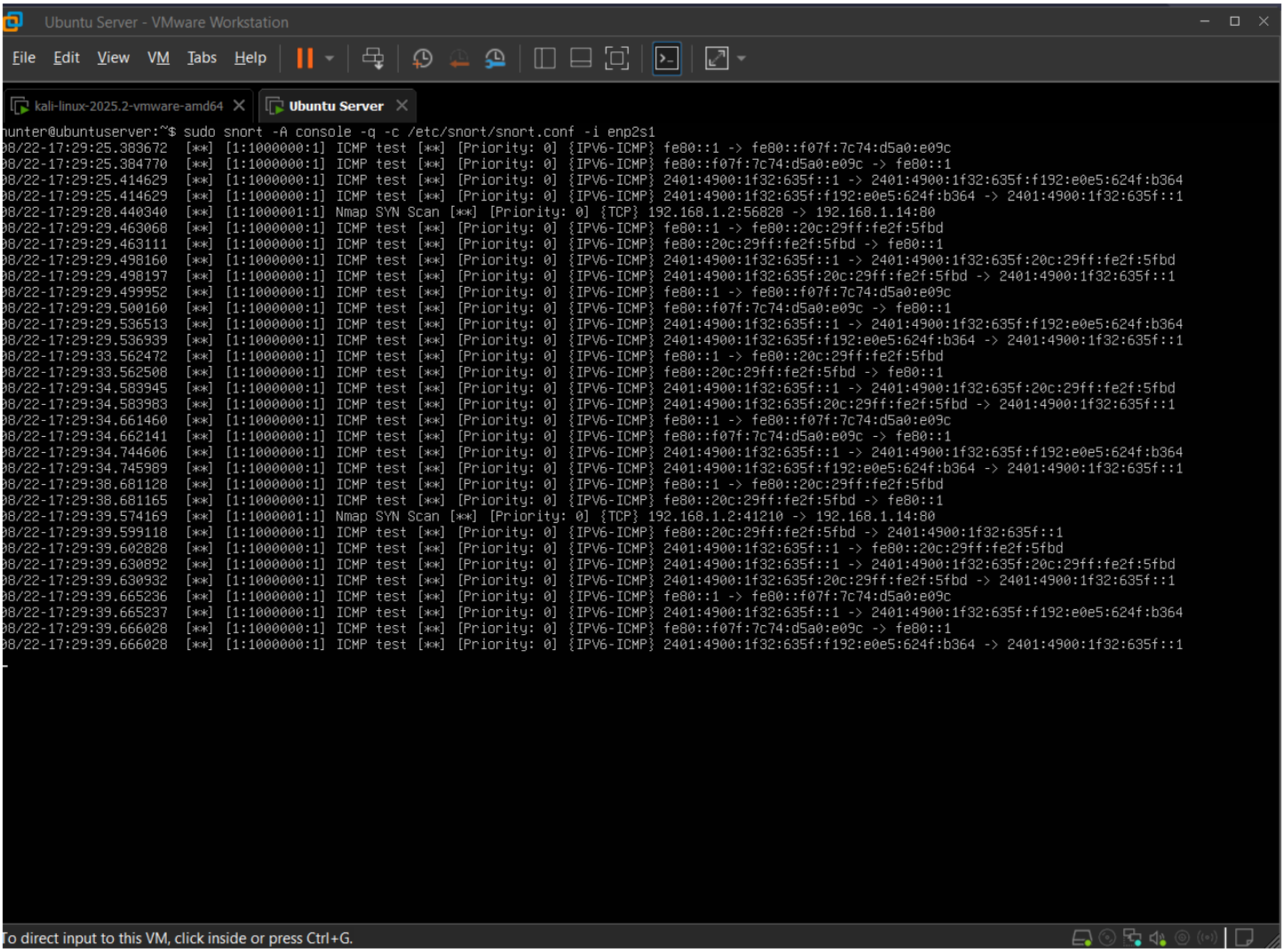


Expected Result

When beaconing traffic is sent:

- Rule 1 triggers → **“Malware C2 HTTP Host detected”**
- Rule 2 triggers → **“Malware C2 Beacon URI detected”**

📷 *[Screenshot: Snort console showing both C2 detection alerts]*



8. Logging and PCAP Capture

For evidence collection, alerts and packet captures were logged.

Commands (Ubuntu Target):

- **Save alerts in fast mode:**

```
sudo snort -A fast -q -c /etc/snort/snort.conf -i enp0s3 -l /var/log/snort
```

View a lerts live:

```
tail -f /var/log/snort/alert
```

Capture packets with tcpdump:

```
sudo tcpdump -i enp0s3 -w /tmp/nids_test.pcap
```

Expected Result

- Alerts written to `/var/log/snort/alert`.
- PCAP file `/tmp/nids_test.pcap` created with full traffic capture.

Conclusion:

This project successfully demonstrated the deployment of a **Network Intrusion Detection System (NIDS)** using **Snort**. Custom rules were created and tested to detect various types of malicious activity, including:

- **ICMP ping sweeps** and **Nmap scans** (network reconnaissance)
- **SSH and FTP brute-force attempts** (credential attacks)
- **Simulated malware C2 beaconing traffic** (HTTP Host header and beacon URI detection)

In every scenario, Snort effectively detected the attack in real-time and generated alerts. The collected **screenshots, logs, and packet captures** provide strong proof-of-concept for the effectiveness of the detection system.

Overall, this project showed how Snort can be configured with custom signatures to improve the **visibility and security posture** of a network environment.

References

- *IMPLEMENTATION-DOCUMENT.pdf* – Internship Guide
- *PROJECT.pdf* – Enterprise-Level Cybersecurity Projects
- *TRAINING.pdf* – Internship Training Plan
- **Snort Documentation** – https://www.snort.org/

Project github link :- *https://github.com/aayushthakur001/Infotact-prj/tree/main/NIDS-PRG-1*