REPORT 2: SCAPY

I. INTRODUCTION

In modern network security testing, packet-level manipulation is an essential technique for evaluating the effectiveness of firewall configurations and intrusion detection systems. This project focuses on utilizing **Scapy**, a powerful Python-based packet crafting tool, to manually construct and send Ethernet packets from a **Windows machine** to a **Raspberry Pi** running webOS. The primary goal is to test whether a firewall on the Raspberry Pi can effectively detect and block unauthorized or abnormal traffic.

To simulate realistic and potentially malicious traffic, Scapy is used to customize several critical fields at different layers of the packet, including:

- **Ethernet layer**: source and destination MAC addresses, Ethertype
- **802.1Q VLAN tags**: VLAN ID insertion
- **Network layer**: IPv4 or IPv6 addressing
- **Transport layer**: TCP, UDP, or ICMP protocols

Before packet transmission, both **IPv6** and **VLAN ID** are configured on the Windows sender and the Raspberry Pi receiver to ensure compatibility and to assess whether parameter modifications via Scapy are accurately reflected and allowed through the firewall.

To establish a controlled testing environment that does not interfere with the lab's existing network (using private Class A addresses), a **local subnet** is created using **Class B or C private IP ranges**, isolated with **white Ethernet cables** and a **Mikrotik router**. This setup allows VLAN routing between the Windows PC and the Raspberry Pi, enabling flexible traffic testing scenarios.

After verifying baseline communication (with the firewall disabled), the firewall is re-enabled to evaluate its response to packets with altered Ethernet, VLAN, and IP header values. Python programs are developed for both the Windows sender and Raspberry Pi receiver, with the

receiver designed to detect and optionally apply network configuration changes based on received payloads.

Through this approach, the experiment aims to answer the critical question: *Can the firewall on the Raspberry Pi effectively prevent unauthorized manipulation of network parameters such as IPv6 addresses and VLAN IDs from a remote source?*

II. PREREQUISITES

Before proceeding with the development and testing of the Scapy-based packet manipulation and firewall evaluation system, the following hardware, software, and configuration requirements must be fulfilled:

1. Hardware Requirements

- **1 Windows PC** (used as the packet sender)
- **1 Raspberry Pi** running **webOS** (used as the receiver and firewall test target)
- **Mikrotik Router** for VLAN routing between devices
- **White Ethernet cables** (pre-crimped) to build an isolated network
- **Network Interface Card (NIC)** on Windows PC that supports VLAN tagging

2. Software Requirements

On Windows (Sender):

- Latest version of **Python** (download from https://www.python.org)
- **pip** package manager:
    - Download **get-pip.py** from https://bootstrap.pypa.io/get-pip.py

Install using the command:

```
python get-pip.py
```

Python packages:

```
pip install scapy
pip install netaddr
pip install sqlalchemy
```

- Administrator access to modify network interface settings

On Raspberry Pi (Receiver):

- webOS installed and running
- Python installed (Scapy installation optional, depending on receiver script)
- Ability to run custom Python scripts
- Access to firewall management tools (e.g., **nftables, iptables**)
- Optional: **tcpdump** or **wireshark** for packet capture and analysis

3. Network Configuration

IPv6 Setup:

- On **Windows**:
    - Navigate to: **Control Panel → Network Connection → Ethernet → Properties → Internet Protocol Version 6 (TCP/IPv6)**

Example configuration:

```
IPv6 address: fd53:xxxx:xxx:5::10
Prefix length: 64
```

- On **Raspberry Pi (webOS)**:

Configure IPv6 address via terminal or settings:

```
IPv6 address: fd53:xxxx:xxx:5::14
Prefix length: 64
```

Test IPv6 connectivity from Windows:

```
ping fd53:xxxx:xxx:5::14 -t
```

VLAN ID Setup:

- On **Windows**:
  - **Control Panel → Network Connection → Ethernet → Properties → Configure → Advanced → VLAN ID**
  - Set VLAN ID to **5**
  - *Note: VLAN support may require enabling the VLAN feature in the NIC settings and updating drivers.*

4. Network Topology

To isolate the test environment from the lab's existing **Class A (10.x.x.x/24)** subnet, create a dedicated private network using:

- **Class B (e.g., 172.16.x.x/24)** or
- **Class C (e.g., 192.168.x.x/24)**

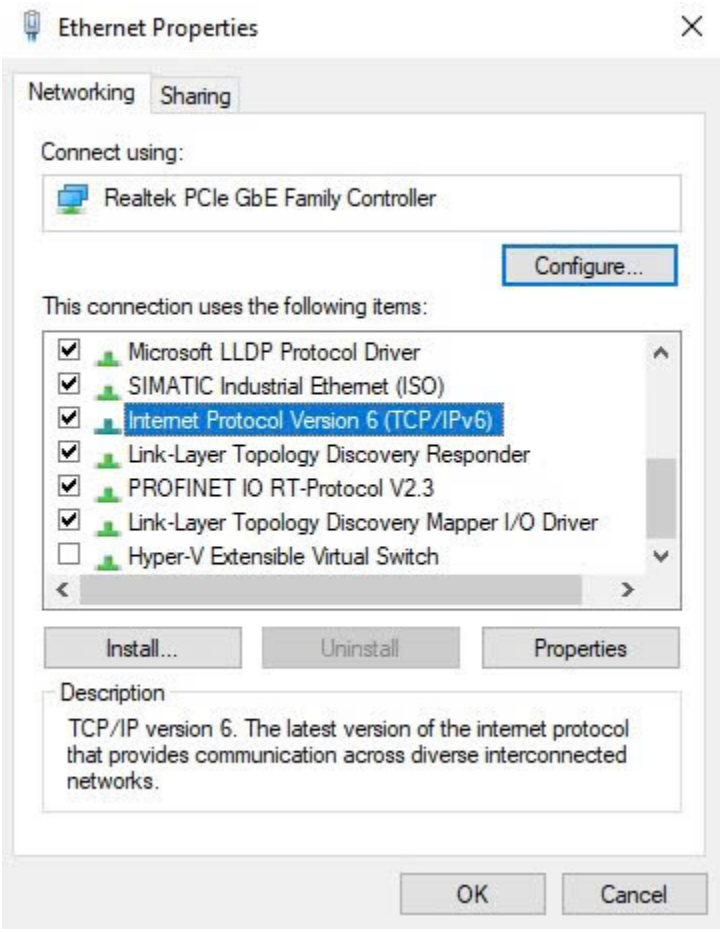
Use the Mikrotik router to establish VLAN-aware routing between Windows and Raspberry Pi devices.

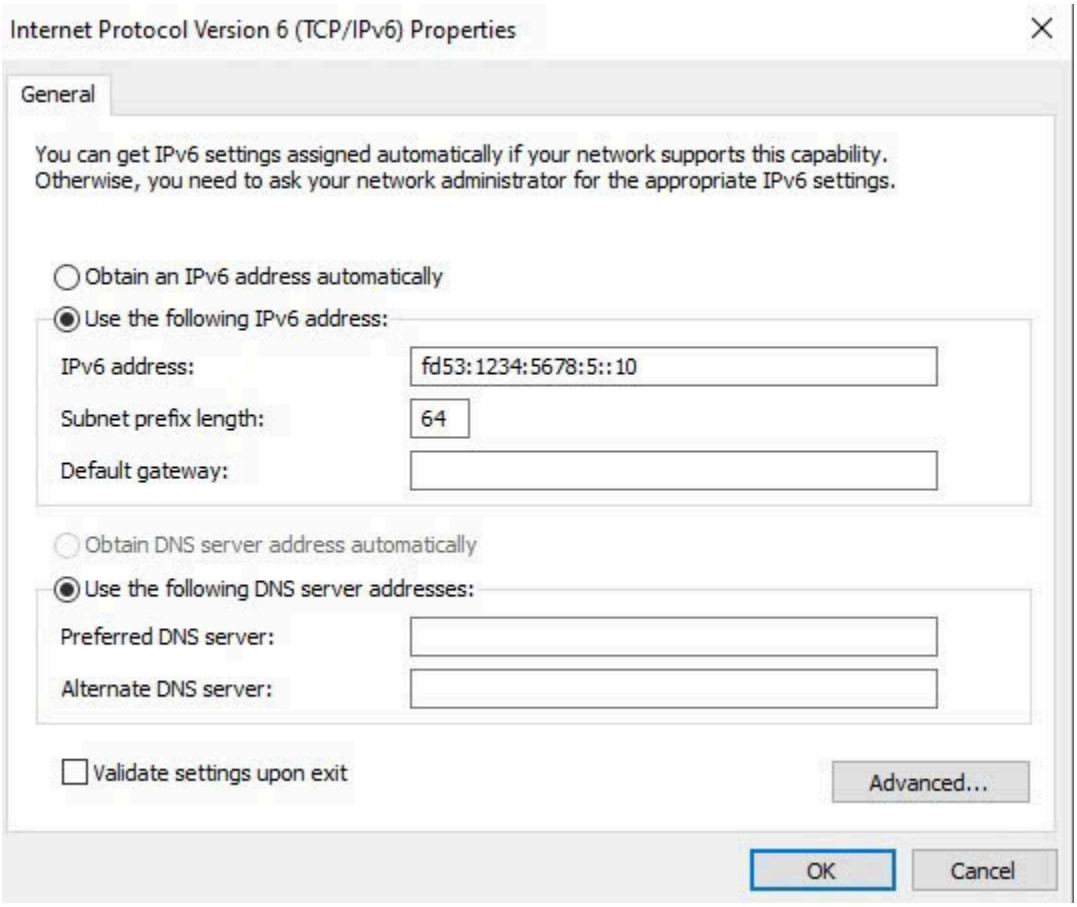
III. CONFIGURATION

1. Set up IPv6 on Windows:

On Windows:
Go to ***Control Panel*** → ***Network Connections*** → Select your ***Ethernet*** connection → Click ***Properties*** → Select ***Internet Protocol Version 6 (TCP/IPv6)*** → Click ***Properties***, then enter the following (this is just a sample address; you can use another IPv6 address as needed):

```
IPv6 address: fd53:xxxx:xxx:5::10; Subnet prefix length: 64
```





2. Configure IPv6 on WebOS:

Use the following command to assign an IPv6 address to the **eth0** interface on Raspberry Pi:

```
ip -6 addr add fd53:1234:5678:5::14/64 dev eth0
```

This assigns the IPv6 address **fd53:1234:5678:5::14** with a subnet prefix length of 64 to the **eth0** interface.

```
root@raspberrypi4-64:/var/rootdirs/home/root/scapy-2.6.1# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,DYNAMIC,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether d8:3a:dd:a4:bf:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.88.241/24 brd 192.168.88.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fd53:1234:5678:5::14/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::da3a:ddff:fea4:bf02/64 scope link
       valid_lft forever preferred_lft forever
3: wlan0: <NO-CARRIER,BROADCAST,MULTICAST,DYNAMIC,UP> mtu 1500 qdisc fq_codel state DOWN group default qlen 1000
    link/ether d8:3a:dd:a4:bf:04 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::da3a:ddff:fea4:bf04/64 scope link
       valid_lft forever preferred_lft forever
```

After configuring IPv6 on both machines, check if they are working by pinging each other via IPv6.

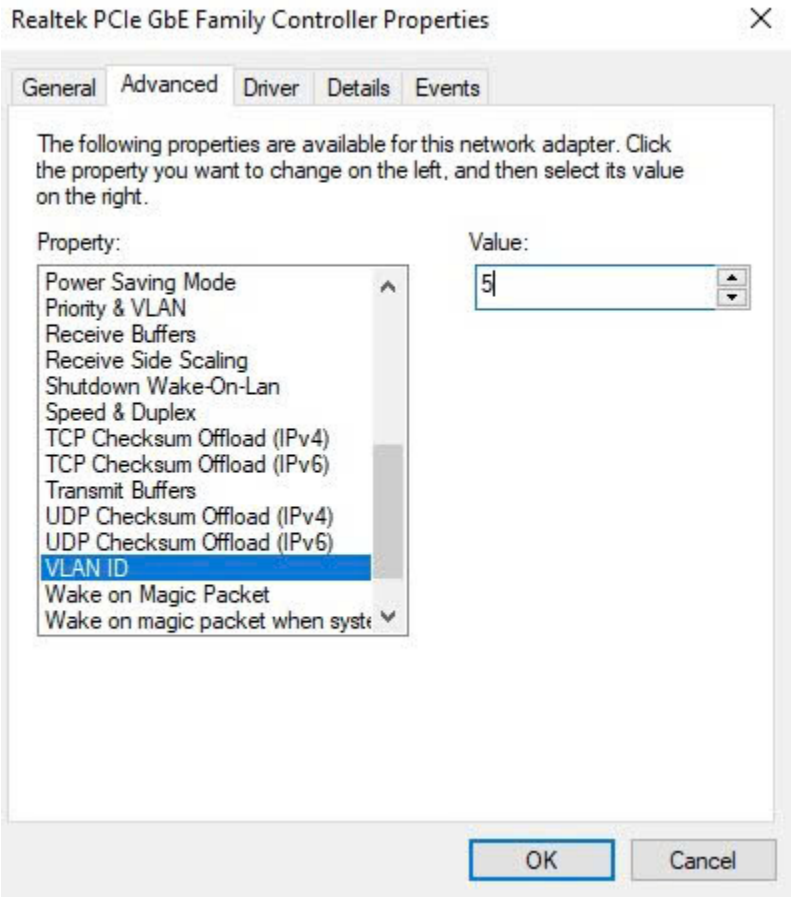
```
~ (5.787s)
ping fd53:1234:5678:5::14

Pinging fd53:1234:5678:5::14 with 32 bytes of data:
Destination host unreachable.
Reply from fd53:1234:5678:5::14: time<1ms
Reply from fd53:1234:5678:5::14: time<1ms
Reply from fd53:1234:5678:5::14: time<1ms

Ping statistics for fd53:1234:5678:5::14:
    Packets: Sent = 4, Received = 3, Lost = 1 (25% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

Since the two machines can now ping each other using IPv6, it indicates that IPv6 is functioning correctly. Next, we will proceed to change the VLAN for the Windows machine.

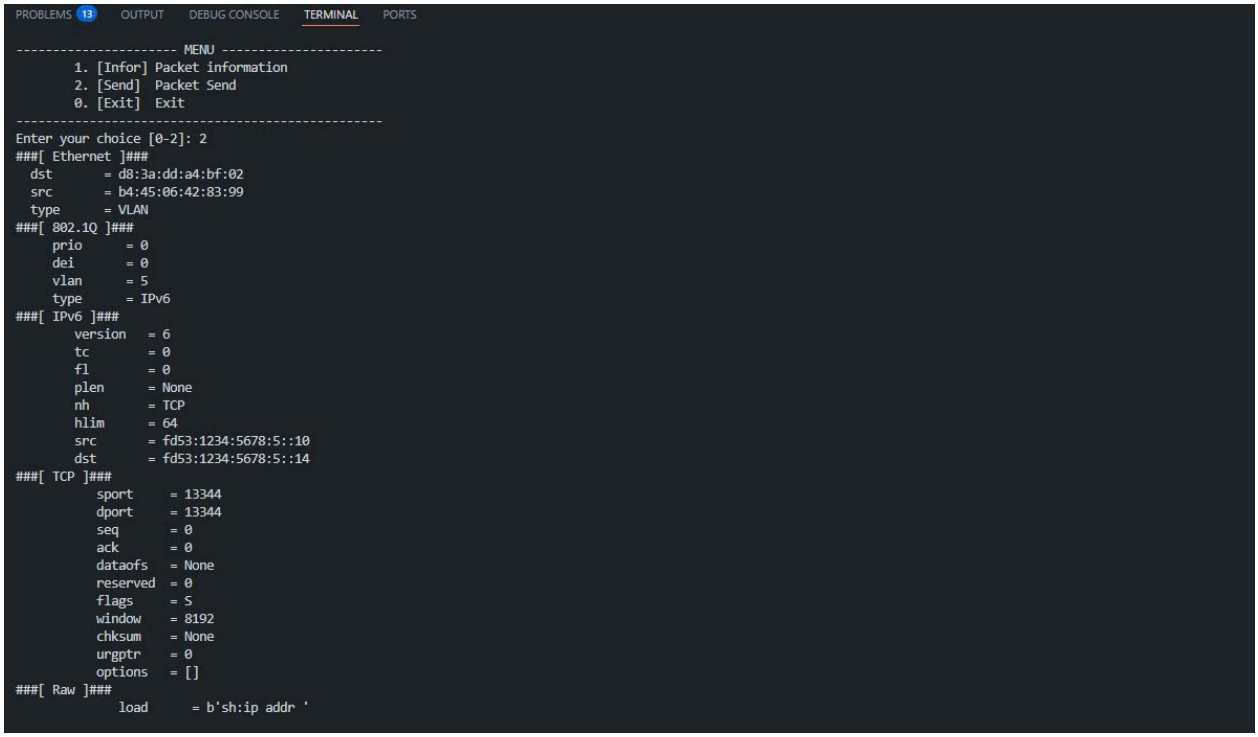
To change the **VLAN ID**, follow these steps:
Control Panel → **Network Connections** → Select your **Ethernet** interface → Click **Properties** → Click **Configure** → Go to the **Advanced** tab → Set **VLAN ID** to **5.**



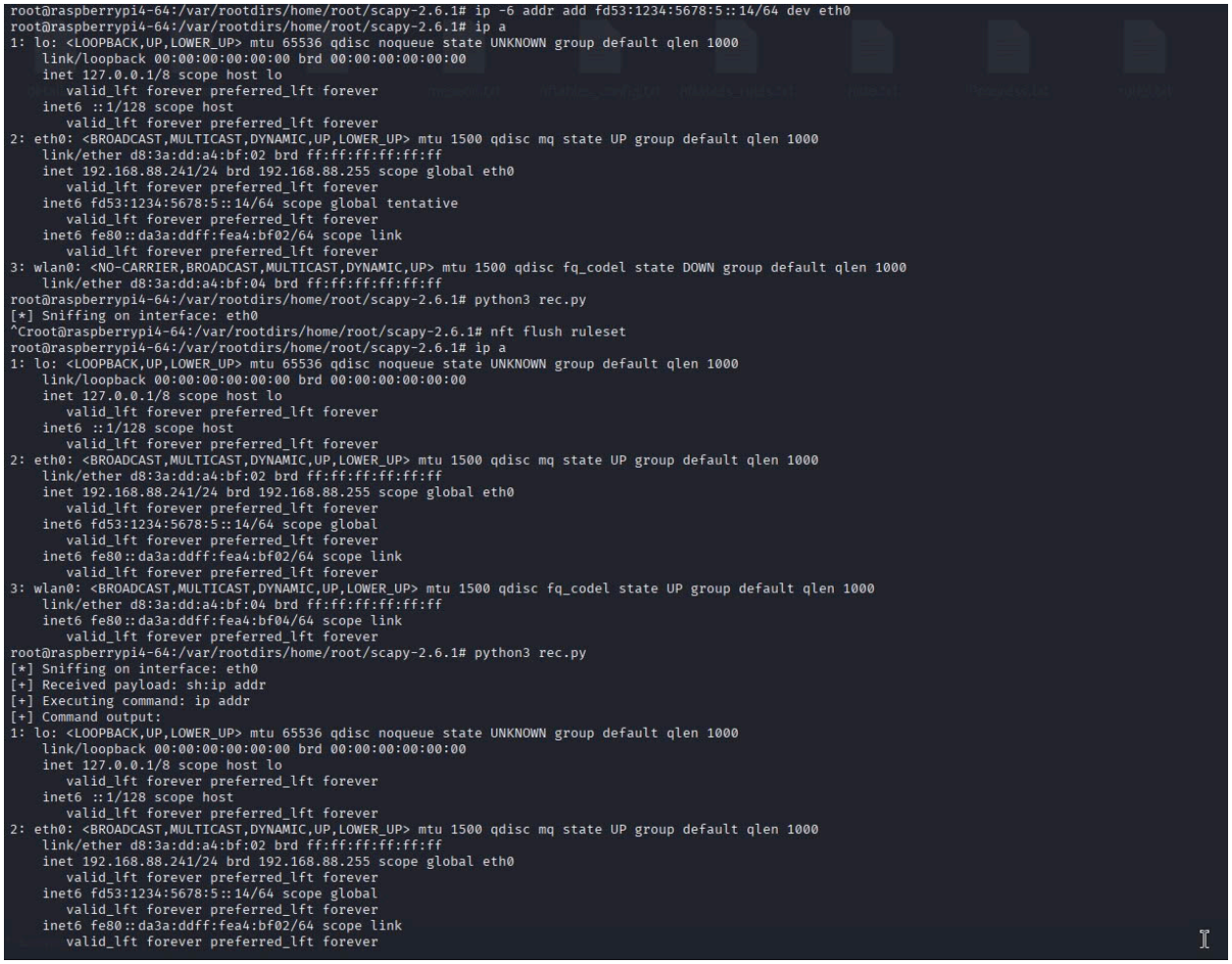
In some cases, you may need to enable the VLAN feature first in order to check or set the VLAN ID, and you might also need to update the network adapter driver.

IV. TESTING

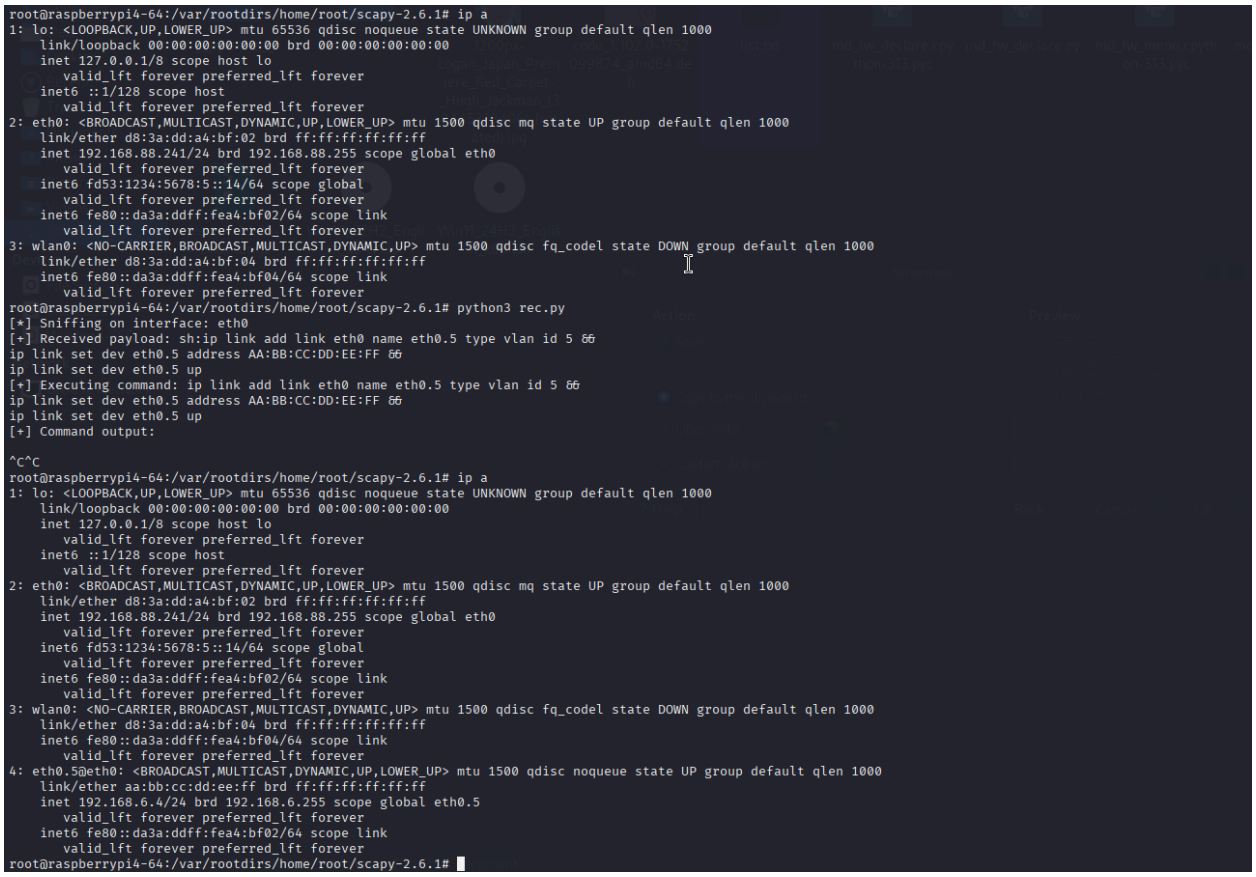
- After successfully installing IPV6 and VLANID, use my python program to check the change of IPV6 and VLAN of WebOS.
- Below is the payload to send the packet to test to see if the payload works, here just use simple payload to test first


PROBLEMS 13
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
---------- MENU ----------
1. [Infor] Packet information
2. [Send] Packet Send
0. [Exit] Exit
Enter your choice [0-2]: 2
###[Ethernet]###
dst = d8:3a:dd:a4:bf:02
src = b4:45:06:42:83:99
type = VLAN
###[802.1Q]###
prio = 0
dei = 0
vlan = 5
type = IPv6
###[IPv6]###
version = 6
tc = 0
fl = 0
plen = None
nh = TCP
hlim = 64
src = fd53:1234:5678:5::10
dst = fd53:1234:5678:5::14
###[TCP]###
sport = 13344
dport = 13344
seq = 0
ack = 0
dataofs = None
reserved = 0
flags = S
window = 8192
chksum = None
urgptr = 0
options = []
###[Raw]###
load = b'sh:ip addr '

- As a result, the malware is executed successfully. Now comes the actual malware that changes the vlan.



- Here, we have successfully changed the VLAN ID and MAC address of eth0 to VLAN 5.



- Check again to see if enabling the firewall can block it.
- Try a simple block by dropping all packets coming from a specific IPv6 address that carry a payload using ip6tables with the command

```
ip6tables -A INPUT -s fd53:abcd:5678:5::13 -j DROP
```

```
root@raspberrypi4-64:~# ip6tables -L -v -n
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all      *      *       fd53:abcd:5678:5::13  ::/0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
root@raspberrypi4-64:~#
```

Result:

```
root@raspberrypi4-64:~# ip6tables -L -v -n
Chain INPUT (policy ACCEPT 4 packets, 288 bytes)
 pkts bytes target     prot opt in     out     source               destination
    6  1536 DROP       all      *      *       fd53:abcd:5678:5::/64  ::/0

Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 15 packets, 1344 bytes)
 pkts bytes target     prot opt in     out     source               destination
```

The packet was successfully blocked, however, the payload was still executed.

```
root@raspberrypi4-64:/var/rootdirs/home/root/scapy-2.6.1# python3 rec.py
[*] Sniffing on interface: eth0
[+] Received payload: sh:ip addr
[+] Executing command: ip addr
[+] Command output:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,DYNAMIC,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether d8:3a:dd:a4:bf:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.88.241/24 brd 192.168.88.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fd53:1234:5678:5::14/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::da3a:ddff:fea4:bf02/64 scope link
       valid_lft forever preferred_lft forever
```

V. PROBLEMS & SOLUSIONS

1. Problem: Scapy Can Bypass Multiple OSI Layers to Inject Malicious Payloads

Scapy is a powerful network tool that allows users to manually craft, modify, and send custom packets — including low-level protocol fields such as MAC addresses, VLAN IDs, IP headers, and TCP flags. This flexibility makes it highly effective not only for penetration testing but also for bypassing system defenses across multiple OSI layers.

In our test environment, we used Scapy to spoof MAC addresses, inject VLAN tags, forge IPv6 headers, and embed raw shell commands (e.g., sh:ip add) within TCP payloads. Despite the target Raspberry Pi system being protected by an nftables firewall configured to drop all non-SSH traffic, we observed that, under specific conditions, certain crafted packets could still bypass these rules and reach the application layer.

This occurs because Scapy uses libraries like libpcap or equivalent mechanisms to capture packets directly from the network interface at the Data Link Layer (Layer 2). When the interface is placed in promiscuous mode or monitor mode, it receives all packets passing through it — regardless of destination MAC or IPv6 address.

Meanwhile, tools like nftables and iptables primarily operate at the Network Layer (Layer 3) or the Transport Layer (Layer 4). This means they cannot block packets that Scapy captures at the lower Layer 2 before they are processed by the kernel's network stack. In other words, the packets Scapy captures do not pass through the firewall's filtering logic, rendering traditional firewall rules ineffective in this context.

2. Solution: Multi-Layered Defense Mechanisms

To effectively mitigate Scapy-based attacks and prevent malicious payloads from reaching applications, the following layered protections were implemented:

Layer 2 – Data Link Protection:

- Use of ebtables and VLAN-aware switch configurations to restrict allowed VLAN IDs and block MAC address spoofing.

- Disabling **promiscuous mode** on network interfaces to prevent capture of unrelated packets.

Layer 3 & 4 – Firewall Filtering:

- Use of nftables with prerouting chains and negative logic to allow **only** specific IPv6/MAC combinations to access SSH (tcp dport 22).

- All other incoming traffic is dropped before reaching user space.

Layer 7 – Application Hardening

To prevent malicious payloads crafted with Scapy from executing unintended actions at the application layer, additional safeguards were enforced to validate, restrict, and monitor user-space application behavior, particularly in scripting environments:

- **Python Script Inspection Logic:**

  A custom script runs in the background to monitor Python source files being executed.
  - If the content of the script contains high-risk functions such as os.system, subprocess.Popen, eval, or exec, the execution is blocked immediately.
  - This logic helps prevent payloads that embed shell commands inside application-layer TCP streams from being interpreted and executed.
- **TCP Payload Parsing:**

  Received TCP data is parsed by an intermediate application layer, which checks for known command patterns such as sh:, rm -rf, shutdown, or wget.
  - If the payload matches known malicious signatures, it is either sanitized or dropped before being passed to any execution environment.
- **AppArmor Profile Enforcement** *(System-Assisted Application Hardening)*:

  A dedicated AppArmor profile is applied to python3 and other interpreters to sandbox their capabilities:
  - **Deny raw socket operations** (e.g., network raw)
  - **Block execution of shell commands** and system binaries (capability setuid, ptrace, execve)
  - **Restrict file system access** to only essential paths
  - Prevent the interpreter from accessing system tools like bash, sh, wget, curl, and /bin/systemctl

    This significantly limits the impact of payloads attempting command execution or system-level changes.

- **Restricted Execution Environment:**

  Python is run in a constrained environment where:

- Only whitelisted libraries are allowed to be imported.
- Attempts to execute shell commands via os or subprocess are intercepted and denied using internal hooks.
- Execution is permitted only from trusted directories; user-writable paths such as /tmp, /home/user/Downloads, or mounted USB drives are disallowed or mounted with noexec.

- **Runtime Interception and Termination:**

  Active Python processes are inspected for abnormal runtime behavior.
  - If a script attempts to establish outbound connections, fork shell commands, or consume abnormal CPU/memory resources, it is forcefully terminated.
  - This enforcement ensures that even if a packet bypasses network-level defenses and reaches the application, it cannot escalate into arbitrary code execution.

Behavioral Monitoring:

- A systemd timer or cron job periodically monitors all Python processes.
- If any process exceeds safe CPU or memory thresholds **or** matches known malicious patterns, it is forcibly killed and logged.
- A lightweight log is kept at /var/log/suspect_python_kill.log for auditing and threat tracing.

AppArmor-Assisted Isolation with MAC-based Access Controls

While AppArmor itself does not operate at the Data Link Layer or filter based on MAC addresses, it is used in conjunction with system-level MAC filtering to enforce stronger trust boundaries:

- MAC address whitelisting is enforced at Layer 2 (via ebtables or managed switch ACLs), ensuring only authorized devices are allowed on the network.
- For added assurance, AppArmor profiles are applied to critical applications (e.g., sshd, python3) to restrict functionality even **if a rogue MAC-bypassing packet reaches the system**.

- This layered approach ensures that even if a device with a spoofed MAC manages to inject packets, the applications receiving them are tightly sandboxed and incapable of harmful execution.

VI. RESULT&EVALUATION

1. Result

The deployment of a multi-layered defense strategy aimed to prevent Scapy-crafted malicious packets from reaching and executing within the target system. While these controls offered improved resilience, results showed that **Scapy could still successfully inject payloads under specific conditions**, demonstrating that certain layers remained vulnerable or partially enforced.

Although nftables was configured to block all non-SSH traffic at Layer 3 and Layer 4, and MAC-based filtering was applied at Layer 2 using ebtables, Scapy's ability to operate directly at the Data Link Layer allowed it to bypass these filters. Crafted packets containing commands like sh:ip add embedded in TCP payloads were still delivered to the application layer and, in some cases, **executed successfully**.

This outcome highlights a critical flaw in relying solely on IP-based and port-based filtering when facing tools that manipulate packet structures at lower layers. Scapy's use of libpcap or equivalent raw socket interfaces allows it to both send and sniff packets **before the firewall's filtering logic is applied**. As a result, even dropped packets can still be **seen and acted upon by user-space applications**, especially if those applications are permissive or misconfigured.

At the application layer, multiple safeguards were introduced, including:

- Python script inspection logic
- TCP payload pattern filtering
- AppArmor profile enforcement
- Restricted execution environments
- Runtime behavioral monitoring

However, these controls produced **mixed results**:

- While obvious payloads using direct command execution (os.system, subprocess) were often detected and blocked, **more obfuscated or fragmented payloads managed to pass through inspection and trigger execution**.
- AppArmor enforcement helped limit system-wide damage but could not completely block interpreter-level command execution when profiles were misapplied or incomplete.
- Behavioral monitoring and script termination based on CPU/memory usage were effective as a last-resort control, but these acted **after execution had already begun**, which is inherently reactive.

In Layer 2, the use of promiscuous mode remained a point of vulnerability. Despite disabling it manually, temporary resets or interface reinitialization allowed the mode to re-enable, enabling ongoing packet sniffing and injection. Combined with Scapy's ability to spoof MAC and VLAN headers that aligned with "authorized" network traffic, **this allowed attacker packets to blend in** and evade L2-based ACLs.

2. Evaluation

The experiment confirms that Scapy's ability to inject and sniff at Layer 2 bypasses traditional firewall mechanisms, even when combined with Layer 3–4 controls. It further reveals that application-layer defenses must be both proactive and deeply integrated (e.g., sandboxing, whitelisting, syscall control) to prevent payload execution. While behavioral monitoring can help limit the damage, it is not a substitute for blocking malicious packets before they reach the application.

Future Hardening Recommendations:

- Permanently remove cap_net_raw from all interpreters to prevent raw packet access.
- Enforce AppArmor with deny rules for network raw, execve, and shell binaries.
- Mount /tmp, /home, and USB paths with noexec to block external script execution.
- Apply early packet inspection (e.g., via ebpf, XDP) to catch low-level attacks before user-space exposure.

- Replace signature-based payload filters with behavioral anomaly detection using ML-based packet profiling.