LIVING LONG DOING PENTESTS

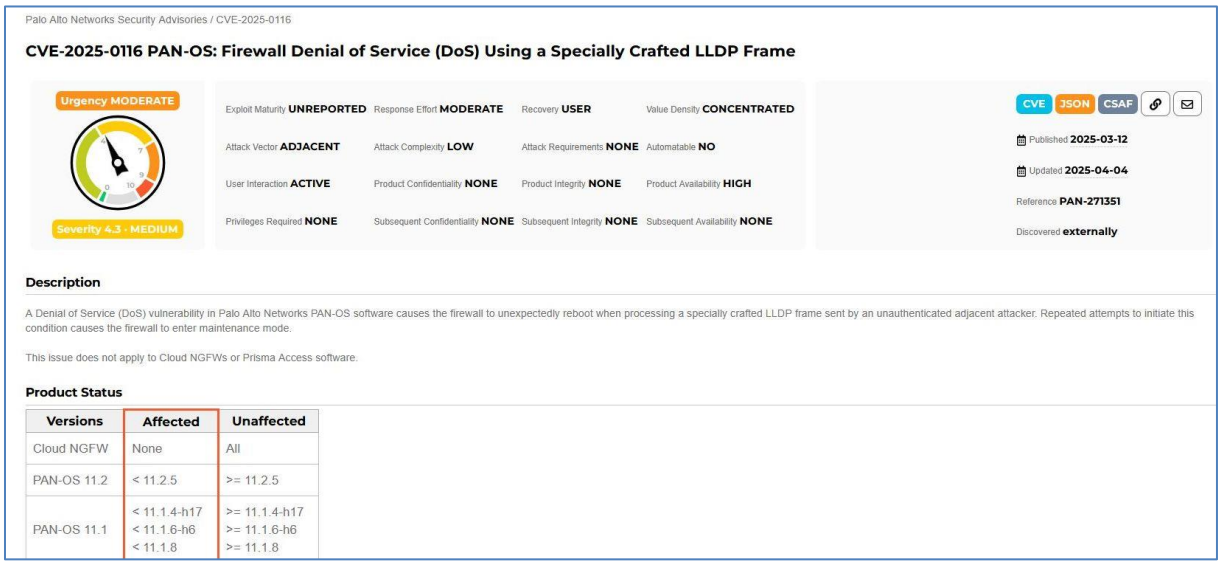
Playing with LLDP

Basic usage of LLDP for infrastructure pentests purposes

Contents

1. Intro

Some time ago, I was searching online for information about vulnerabilities in popular networking devices. One of the **links** I came across concerned the **CVE-2025-0116** vulnerability related to the handling of the LLDP (Link Layer Discovery Protocol) by Palo Alto devices.



Palo Alto Networks Security Advisories / CVE-2025-0116

CVE-2025-0116 PAN-OS: Firewall Denial of Service (DoS) Using a Specially Crafted LLDP Frame

Urgency MODERATE

Severity 4.3 · MEDIUM

Exploit Maturity **UNREPORTED** Response Effort **MODERATE** Recovery **USER** Value Density **CONCENTRATED**

Attack Vector **ADJACENT** Attack Complexity **LOW** Attack Requirements **NONE** Automatable **NO**

User Interaction **ACTIVE** Product Confidentiality **NONE** Product Integrity **NONE** Product Availability **HIGH**

Privileges Required **NONE** Subsequent Confidentiality **NONE** Subsequent Integrity **NONE** Subsequent Availability **NONE**

CVE JSON CSAF

Published **2025-03-12**

Updated **2025-04-04**

Reference **PAN-271351**

Discovered **externally**

Description

A Denial of Service (DoS) vulnerability in Palo Alto Networks PAN-OS software causes the firewall to unexpectedly reboot when processing a specially crafted LLDP frame sent by an unauthenticated adjacent attacker. Repeated attempts to initiate this condition causes the firewall to enter maintenance mode.

This issue does not apply to Cloud NGFWs or Prisma Access software.

Product Status

Versions	Affected	Unaffected
Cloud NGFW	None	All
PAN-OS 11.2	< 11.2.5	>= 11.2.5
PAN-OS 11.1	< 11.1.4-h17 < 11.1.6-h6 < 11.1.8	>= 11.1.4-h17 >= 11.1.6-h6 >= 11.1.8

Intrigued by the description, I decided to check how it looks on my own device in a home lab environment. I chose to continue the tests in the setup I mentioned in one of my recent posts about a vulnerability in the CLI-handling application found in Palo Alto.

Below are some notes I gathered on the topic...

2. Protocol description

The basic information I was able to gather about the LLDP protocol includes:

> "The LLDP (Link Layer Discovery Protocol) is a standardized network protocol operating at the data link layer (Layer 2 of the OSI model), which allows network devices to exchange information about their configuration and parameters. It enables automatic discovery of neighboring devices such as switches, routers, or access points, and the collection of data about their MAC addresses, IP addresses, names, ports, and network capabilities. LLDP is based on transmitting TLV (Type-Length-Value) frames that contain various types of information in a standardized format. It is widely used in network management, making it easier to map topology and configure devices. The protocol is vendor-neutral, which distinguishes it from similar solutions like Cisco CDP. It operates in multicast mode, sending information periodically or when a configuration change occurs. LLDP is defined in the IEEE 802.1AB standard and is supported by most modern network devices."

At this stage, I decided to explore the potential use and application of the LLDP protocol in the context of Palo Alto devices. A brief outline of the concept is provided below:

> LLDP support in Palo Alto Networks devices is available on firewalls, enabling the discovery of neighboring devices and their parameters at Layer 2 of the OSI model. (...) LLDP must be enabled globally as well as on selected interfaces, and an LLDP profile must be configured to define the mode (transmit/receive) and optional TLVs. The firewalls support only one MAC address for LLDP frames (01-80-C2-00-00-0E), and the information is stored in the MIB database accessible via SNMP. LLDP facilitates network management, especially in virtual wire topologies where firewalls are invisible to tools like ping or traceroute. (...) It is also possible to configure the transmission interval (default: 30 seconds) and hold time (default: 120 seconds).

As I continued researching the topic, I came across the following information:

> The MAC address 01-80-C2-00-00-0E is a standard multicast address used by the LLDP protocol, as defined in the IEEE 802.1AB specification. (...) All LLDP-compliant devices, including Palo Alto firewalls, use this address to send and receive LLDP frames in multicast mode. In the case of Palo Alto, this means that the firewalls do not use any other MAC addresses for LLDP, which is in line with the universal protocol specification.

I found all of this information interesting and useful as a foundation for broader research on the protocol in my home lab.

3. Environment Setup

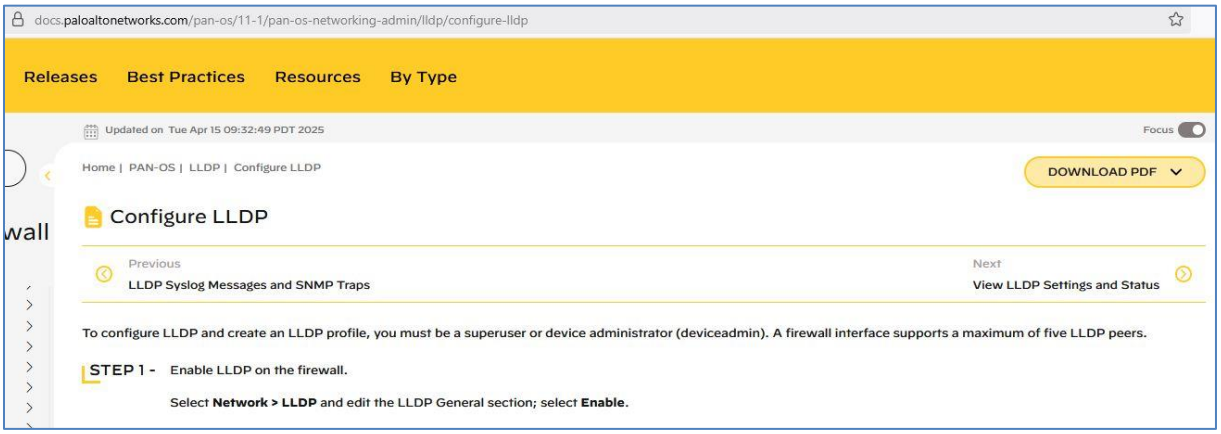
While reading about the LLDP protocol and its potential use in penetration testing, I installed and configured two virtual machines: Kali and Palo Alto (similar to the setup described earlier in this post).

Unlike in previous tests, this time I used VMware to install both machines.

In the meantime—while researching how to install and configure a suitable testing environment—I learned that in LLDP-related tests involving virtualization (e.g., Palo Alto), devices may not "see" each other properly. To avoid this issue, both network interfaces (on the Kali and Palo Alto machines) should be set to Bridge mode. This allows the devices to "communicate" via LLDP and enables us, in later steps, to sniff traffic and easily debug or interact using our custom LLDP fuzzing client.

To summarize:
On the latest version of Kali Linux, I installed Scapy to facilitate working on potential tools (for this purpose, I planned to use Python 3).

As for the installation and configuration of the Palo Alto device:
I used version 11.1.4-h7 (as described in a previous post) with "default settings." The changes I made specifically for conducting the LLDP-related tests are presented below:



First and foremost, I followed the settings available through the WebGUI and enabled LLDP support as shown in the table below:

```
configure
set network lldp enable yes
commit
```

"Checking whether Wireshark could detect any LLDP communication—I unfortunately didn’t see any results, so I decided to read up on configuring LLDP support via the CLI.

Following this lead, I found additional settings which I applied on the Palo Alto device to complete the configuration and actually enable LLDP:"

```
configure
set network lldp enable yes
set zone trust network layer3
set network interface ethernet ethernet1/1 layer3 ip 192.168.56.1/24
set network interface ethernet ethernet1/1 zone trust
set network interface ethernet ethernet1/1 lldp enable yes
set network interface ethernet ethernet1/1 management-profile allow-ping
commit
show interface ethernet1/1
```

At this stage, we should already have the environment ready for further testing.

4. Purpose and Description of the Study

Taking into account the information gathered above, my goal was not to replicate the attack described in the CVE. This time, I aimed for different results, so I divided the work into several stages:

- creating a simple "LLDP client" — a program designed to send LLDP packets via a network interface (e.g., eth0) on Kali Linux
- creating a simple "LLDP sniffer" — a program intended to listen to the network to detect information transmitted using LLDP
- creating a simple "LLDP fuzzer" — with which I could prepare attack scenarios against devices using this protocol for communication

Equipped with Python 3, Scapy, Wireshark, and tcpdump, I decided to start working using Kali Linux (as the "pentester's station") installed earlier.

My goal was to develop several simple "tools" (scripts) that could potentially be used in future penetration tests of network infrastructure.

Verification of potential vulnerabilities (e.g., through collecting and analyzing data from tested network devices, logs, memory dumps) is beyond the scope of this document and is therefore left to the reader.

5. Basic „LLDP client”

An example of a simple client using LLDP is shown below:

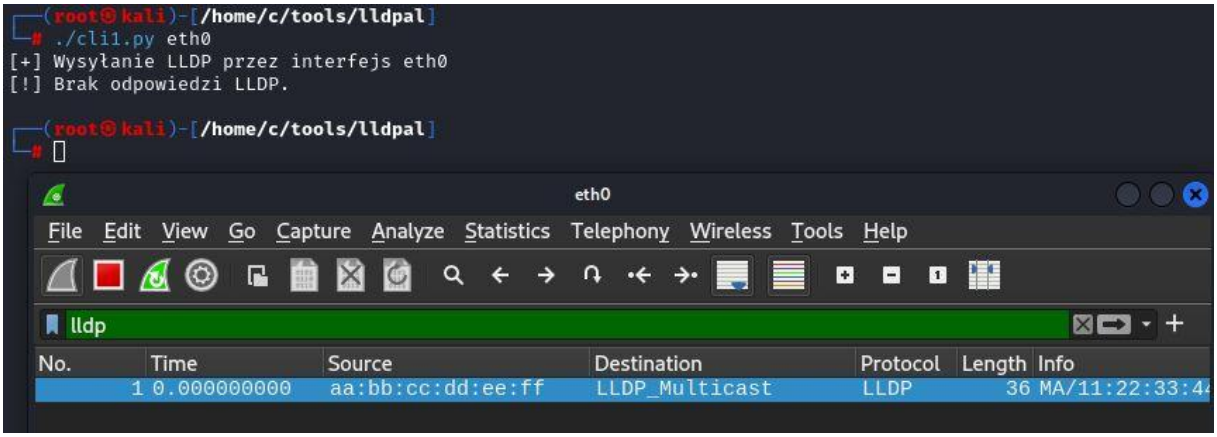
```
#!/usr/bin/env python3
import sys
import socket
import struct
import time


def make_tlv(tlv_type, tlv_value):
    length = len(tlv_value)
    tlv_header = ((tlv_type & 0x7f) << 9) | (length & 0x1ff)
    return struct.pack('!H', tlv_header) + tlv_value

def build_lldp_packet():
    # TLV 1: Chassis ID, subtype 4 (MAC address)
    chassis_tlv = make_tlv(1, b'\x04' + b'\x11\x22\x33\x44\x55\x66')

    # TLV 2: Port ID, subtype 5 (interface name)
    port_tlv = make_tlv(2, b'\x05' + b'eth0')

    # TLV 3: TTL, 2 bajty (seconds)
    ttl_tlv = make_tlv(3, struct.pack('!H', 120))

    # TLV 0: End of LLDPDU
    end_tlv = struct.pack('!H', 0)

    return chassis_tlv + port_tlv + ttl_tlv + end_tlv


def send_lldp(interface):
    raw_socket = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
    raw_socket.bind((interface, 0))

    dst_mac = b'\x01\x80\xc2\x00\x00\x0e'  # LLDP multicast address
    src_mac = b'\xaa\xbb\xcc\xdd\xee\xff'  # arbitrary source MAC

    ethertype = b'\x88\xcc'

    lldp_payload = build_lldp_packet()
    ethernet_frame = dst_mac + src_mac + ethertype + lldp_payload

    print(f"[+] Wysyłanie LLDP przez interfejs {interface}")
    raw_socket.send(ethernet_frame)

    # Odbieranie odpowiedzi (może być pusto!)
    raw_socket.settimeout(5)
    try:
        while True:
            pkt = raw_socket.recv(1500)
            if pkt[12:14] == b'\x88\xcc':
                print("[*] Otrzymano LLDP:")
                print(pkt.hex())
    except socket.timeout:
        print("[!] Brak odpowiedzi LLDP.")
    finally:
        raw_socket.close()

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"Użycie: sudo python3 {sys.argv[0]} <interfejs>")
```

```
    sys.exit(1)

send_lldp(sys.argv[1])
```

The assumptions for the code operation are:

- to construct a simple LLDP packet
- to "broadcast" the created LLDP packet within the tested network.

During this test, I used Wireshark to confirm the client's operation:



The code created in this way enabled me, in the next steps, to build a simple fuzzer utilizing the LLDP protocol.

6. Basic „LLDP Sniffer”

An example of a simple sniffer detecting messages using LLDP is shown below:

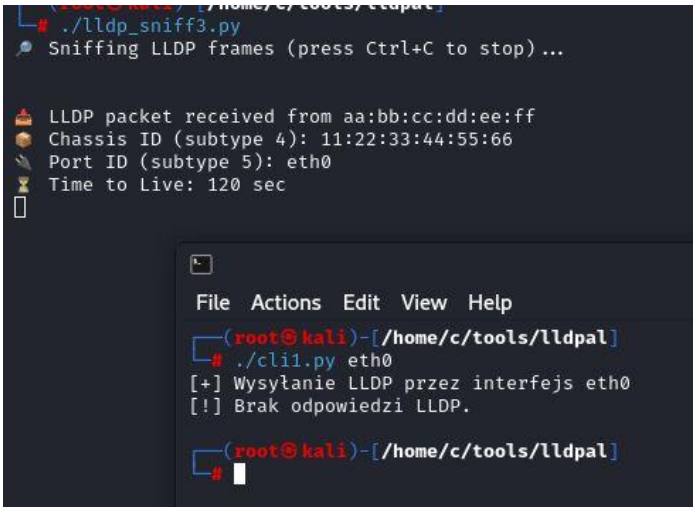
```
#!/usr/bin/env python3

from scapy.all import sniff, Ether, Raw
import struct

LLDP_MULTICAST_MAC = "01:80:c2:00:00:0e"
LLDP_ETHERTYPE = 0x88cc

TLV_TYPE_MAP = {
   0: "End of LLDPDU",
   1: "Chassis ID",
   2: "Port ID",
   3: "Time to Live",
   4: "Port Description",
   5: "System Name",
   6: "System Description",
   7: "System Capabilities",
   8: "Management Address"
}

CAPABILITY_NAMES = [
   (0, "Other"),
   (1, "Repeater"),
   (2, "Bridge"),
   (3, "WLAN Access Point"),
   (4, "Router"),
   (5, "Telephone"),
   (6, "DOCSIS Cable Device"),
   (7, "Station Only"),
   (8, "C-VLAN Component"),
   (9, "S-VLAN Component"),
   (10, "Two-port MAC Relay")
]

def decode_capabilities(bits):
   return " + ".join(name for bit, name in CAPABILITY_NAMES if bits & (1 << bit)) or "None"

def parse_lldpdu(payload):
   idx = 0
   while idx + 2 <= len(payload):
      tlv_header = struct.unpack("!H", payload[idx:idx+2])[0]
      tlv_type = (tlv_header >> 9) & 0x7F
      tlv_len = tlv_header & 0x1FF
      idx += 2

      if tlv_len == 0 or idx + tlv_len > len(payload):
         break

      value = payload[idx:idx+tlv_len]
      idx += tlv_len

      desc = TLV_TYPE_MAP.get(tlv_type, f"Unknown TLV ({tlv_type})")

      if tlv_type == 1:  # Chassis ID
         subtype = value[0]
         data = value[1:]
         print(f"  Chassis ID (subtype {subtype}): {data.hex(':')}")
      elif tlv_type == 2:  # Port ID
         subtype = value[0]
```

```
            data = value[1:]
            print(f"  Port ID (subtype {subtype}): {data.decode(errors='ignore')}")
        elif tlv_type == 3:  # TTL
            ttl = struct.unpack("!H", value)[0]
            print(f"  Time to Live: {ttl} sec")
        elif tlv_type == 5:  # System Name
            print(f"  System Name: {value.decode(errors='ignore')}")
        elif tlv_type == 6:  # System Description
            print(f"  System Description: {value.decode(errors='ignore')}")
        elif tlv_type == 7:  # System Capabilities
            if len(value) >= 4:
                sys_caps, enabled_caps = struct.unpack("!HH", value[:4])
                print(f"  System Capabilities: {decode_capabilities(sys_caps)} (0x{sys_caps:04x})")
                print(f"  Enabled Capabilities: {decode_capabilities(enabled_caps)} (0x{enabled_caps:04x})")
            else:
                print("  System Capabilities: malformed")
        elif tlv_type == 0:
            print("  End of LLDPDU")
            break
        else:
            print(f"  {desc}: {value.hex()}")

def handle_packet(pkt):
    if Ether in pkt and pkt.type == LLDP_ETHERTYPE:
        eth = pkt[Ether]
        print(f"\n LLDP packet received from {eth.src}")
        if Raw in pkt:
            payload = pkt[Raw].load
            parse_lldpdu(payload)

print(" Sniffing LLDP frames (press Ctrl+C to stop)...\n")
#sniff(filter="ether proto 0x88cc", prn=handle_packet, store=0)
sniff(filter="ether proto 0x88cc", prn=handle_packet, store=0, iface="eth0")
```

The assumptions for the code operation are:

- detecting communication using the LLDP protocol in the tested network
- detecting network devices.

An example of the sniffer's operation is shown in the screenshot below:



At the beginning, we focus on simply detecting LLDP communication in the network where the pentest is being conducted. In subsequent stages of updating this simple sniffer's code, we can modify the way packets are analyzed to try to extract more data potentially interesting from the perspective of our study.

My interest was whether LLDP packets would be visible to the sniffer and if I could prepare the script code in a way that would facilitate identifying Palo Alto machines.

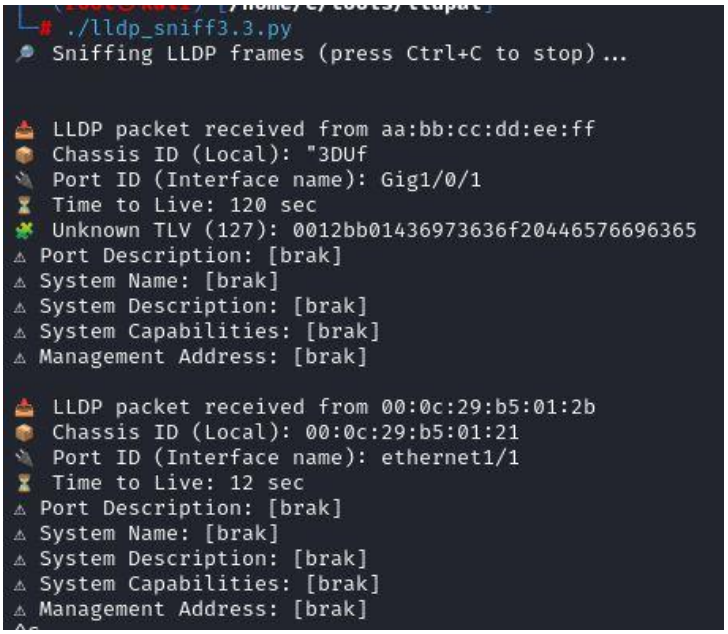
While modifying the code, I decided to add more information to the displayed results upon detecting LLDP communication. The updated code is presented in the box below:

```
#!/usr/bin/env python3
from scapy.all import sniff, Ether, Raw
import struct

LLDP_MULTICAST_MAC = "01:80:c2:00:00:0e"
LLDP_ETHERTYPE = 0x88cc

TLV_TYPE_MAP = {
   0: "End of LLDPDU",
   1: "Chassis ID",
   2: "Port ID",
   3: "Time to Live",
   4: "Port Description",
   5: "System Name",
   6: "System Description",
   7: "System Capabilities",
   8: "Management Address"
}

CAPABILITY_NAMES = [
   (0, "Other"),
   (1, "Repeater"),
   (2, "Bridge"),
   (3, "WLAN Access Point"),
   (4, "Router"),
   (5, "Telephone"),
   (6, "DOCSIS Cable Device"),
   (7, "Station Only"),
   (8, "C-VLAN Component"),
   (9, "S-VLAN Component"),
   (10, "Two-port MAC Relay")
```

```
]

def decode_capabilities(bits):
    return " + ".join(name for bit, name in CAPABILITY_NAMES if bits & (1 << bit)) or "None"

def parse_lldpdu(payload):
    idx = 0
    seen = set()

    while idx + 2 <= len(payload):
        tlv_header = struct.unpack("!H", payload[idx:idx+2])[0]
        tlv_type = (tlv_header >> 9) & 0x7F
        tlv_len = tlv_header & 0x1FF
        idx += 2

        if tlv_len == 0 or idx + tlv_len > len(payload):
            break

        value = payload[idx:idx+tlv_len]
        idx += tlv_len

        desc = TLV_TYPE_MAP.get(tlv_type, f"Unknown TLV ({tlv_type})")
        seen.add(tlv_type)

        if tlv_type == 1:
            subtype = value[0]
            data = value[1:]
            subtype_str = {
                1: "Chassis MAC",
                2: "Interface name",
                4: "Local"
            }.get(subtype, f"Subtype {subtype}")
            try:
                output_str = data.decode("utf-8")
                if not output_str.strip():
                    raise ValueError
            except:
                output_str = data.hex(":")
            print(f"□ Chassis ID ({subtype_str}): {output_str}")

        elif tlv_type == 2:
            subtype = value[0]
            data = value[1:]
            subtype_str = {
                3: "MAC address",
                5: "Interface name",
                7: "Local"
            }.get(subtype, f"Subtype {subtype}")
            output = data.hex(":") if subtype == 3 else data.decode(errors='ignore')
            print(f"□ Port ID ({subtype_str}): {output}")

        elif tlv_type == 3:
            ttl = struct.unpack("!H", value)[0]
            print(f"□ Time to Live: {ttl} sec")

        elif tlv_type == 4:
            print(f"□ Port Description: {value.decode(errors='ignore')}")

        elif tlv_type == 5:
            print(f"□ System Name: {value.decode(errors='ignore')} ← peer hostname")

        elif tlv_type == 6:
            print(f"□ System Description: {value.decode(errors='ignore')} ← peer OS or platform")
```

```
        elif tlv_type == 7:
            if len(value) >= 4:
                sys_caps, enabled_caps = struct.unpack("!HH", value[:4])
                print(f"  System Capabilities: {decode_capabilities(sys_caps)} (0x{sys_caps:04x})")
                print(f"  Enabled Capabilities: {decode_capabilities(enabled_caps)} (0x{enabled_caps:04x})")
            else:
                print("  System Capabilities: malformed")

        elif tlv_type == 8:
            mgmt_addr_len = value[0]
            mgmt_addr = value[1:1+mgmt_addr_len]
            addr_str = ".".join(str(b) for b in mgmt_addr) if mgmt_addr_len == 4 else mgmt_addr.hex()
            print(f"  Management Address: {addr_str}")

        elif tlv_type == 0:
            print("  End of LLDPDU")

        else:
            print(f"  {desc}: {value.hex()}")

    #  Wypisz placeholdery dla brakujących TLV
    for required in [1, 2, 3, 4, 5, 6, 7, 8]:
        if required not in seen:
            label = TLV_TYPE_MAP.get(required, f"TLV {required}")
            print(f"  {label}: [brak]")

def handle_packet(pkt):
    if Ether in pkt and pkt.type == LLDP_ETHERTYPE:
        eth = pkt[Ether]
        print(f"\n LLDP packet received from {eth.src}")
        if Raw in pkt:
            payload = pkt[Raw].load
            parse_lldpdu(payload)

print(" Sniffing LLDP frames (press Ctrl+C to stop)...\n")
sniff(filter="ether proto 0x88cc", prn=handle_packet, store=0, iface="eth0")
```

An example of the operation and results is shown below:



Moving forward...

7. Basic „LLDP Fuzzer”

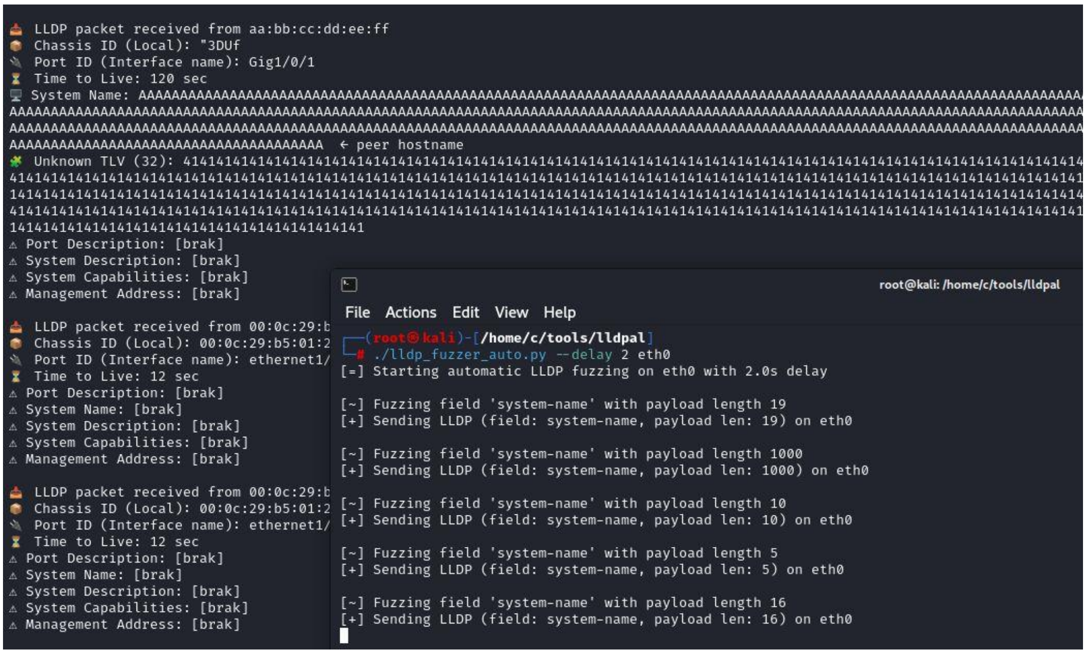
The goal I set here was to create a simple LLDP fuzzer. However, considering how the protocol works, I decided that the fuzzer would focus on sending rather than sending and receiving data. We are not “attacking” a typical client-server application here; instead, we politely wait for (possible) messages *from the server* — in our case, for what a given device XYZ propagates in the network we are pentesting.

An example of a simple fuzzer utilizing LLDP is shown below:

```
#!/usr/bin/env python3
import sys
import socket
import struct
import time
import argparse

def make_tlv(tlv_type, tlv_value):
    length = len(tlv_value)
    tlv_header = ((tlv_type & 0x7f) << 9) | (length & 0x1ff)
    return struct.pack('!H', tlv_header) + tlv_value

def build_lldp_packet(fuzz_field=None, fuzz_payload=b''):
    chassis = b'\x04' + b'\x11\x22\x33\x44\x55\x66'
    chassis_tlv = make_tlv(1, chassis)

    port = b'\x05' + b'Gig1/0/1'
    port_tlv = make_tlv(2, port)

    ttl_tlv = make_tlv(3, struct.pack('!H', 120))

    extra_tlv = b''
    if fuzz_field == "system-name":
        extra_tlv = make_tlv(5, fuzz_payload)
    elif fuzz_field == "system-description":
        extra_tlv = make_tlv(6, fuzz_payload)
    elif fuzz_field == "custom":
        oui = b'\x00\x12\xbb'  # Cisco OUI
        subtype = b'\x01'
        custom_value = oui + subtype + fuzz_payload
        extra_tlv = make_tlv(127, custom_value)

    end_tlv = struct.pack('!H', 0)
    return chassis_tlv + port_tlv + ttl_tlv + extra_tlv + end_tlv

def send_lldp(interface, fuzz_field, fuzz_payload):
    raw_socket = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
    raw_socket.bind((interface, 0))

    dst_mac = b'\x01\x80\xc2\x00\x00\x0e'
    src_mac = b'\xaa\xbb\xcc\xdd\xee\xff'
    ethertype = b'\x88\xcc'

    lldp_payload = build_lldp_packet(fuzz_field, fuzz_payload)
    ethernet_frame = dst_mac + src_mac + ethertype + lldp_payload

    print(f"[+] Sending LLDP (field: {fuzz_field}, payload len: {len(fuzz_payload)}) on {interface}")
    raw_socket.send(ethernet_frame)
    raw_socket.close()
```

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="LLDP Auto Fuzzer")
    parser.add_argument("interface", help="Network interface (e.g. eth0)")
    parser.add_argument("--delay", type=float, default=2.0, help="Delay between fuzz sends in seconds (default 2s)")
    args = parser.parse_args()

    fields = ["system-name", "system-description", "custom"]
    payloads = [
        b"https://phish.local",
        b"A" * 1000,
        b"TestDevice",
        b"\x00\xff\xfe\x01\x02",  # binary garbage example
        b"NormalPayload123"
    ]

    print(f"[=] Starting automatic LLDP fuzzing on {args.interface} with {args.delay}s delay")

    try:
        for field in fields:
            for payload in payloads:
                print(f"\n[~] Fuzzing field '{field}' with payload length {len(payload)}")
                send_lldp(args.interface, field, payload)
                time.sleep(args.delay)
    except KeyboardInterrupt:
        print("\n[!] Interrupted by user. Exiting.")
        sys.exit(0)
```

Sample results after running this fuzzer (along with visible results obtained using the sniffer whose code was presented in the previous section of this document):



The code operation at this stage can be described as:

- modifying specific fields (e.g., System Name)
- attempting to send the modified packet.

8. Enhancing the fuzzer's functionality

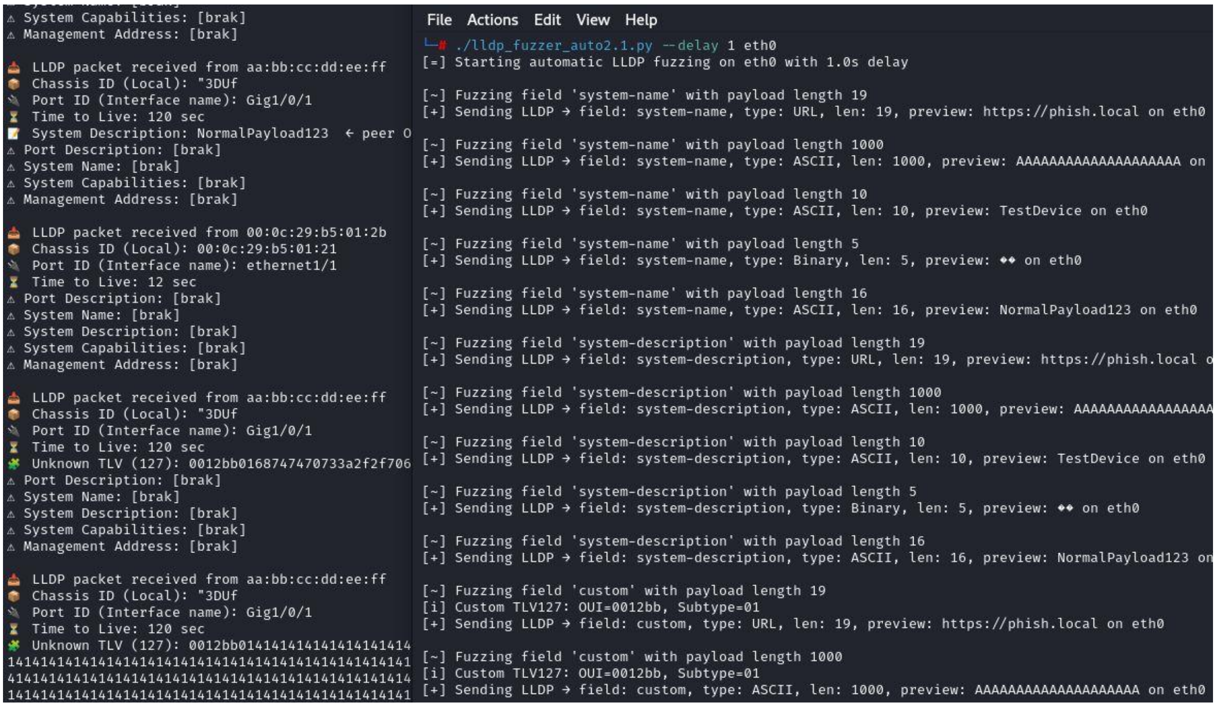
Considering the potential of the LLDP protocol, my next steps were aimed at increasing the modification of additional script elements.
The code below adds the `classify_payload()` function, the ability to modify OUI and subtype, and support for the `--once` parameter to send only one packet.

The updated fuzzer code is presented below:

```
#!/usr/bin/env python3
import sys
import socket
import struct
import time
import argparse

def make_tlv(tlv_type, tlv_value):
    length = len(tlv_value)
    tlv_header = ((tlv_type & 0x7f) << 9) | (length & 0x1ff)
    return struct.pack('!H', tlv_header) + tlv_value

def build_lldp_packet(fuzz_field=None, fuzz_payload=b''):
    chassis = b'\x04' + b'\x11\x22\x33\x44\x55\x66'
    chassis_tlv = make_tlv(1, chassis)

    port = b'\x05' + b'Gig1/0/1'
    port_tlv = make_tlv(2, port)

    ttl_tlv = make_tlv(3, struct.pack('!H', 120))

    extra_tlv = b''
    if fuzz_field == "system-name":
        extra_tlv = make_tlv(5, fuzz_payload)
    elif fuzz_field == "system-description":
        extra_tlv = make_tlv(6, fuzz_payload)
    elif fuzz_field == "custom":
        oui = b'\x00\x12\xbb'  # Cisco OUI
        subtype = b'\x01'
        print(f"[i] Custom TLV127: OUI={oui.hex()}, Subtype={subtype.hex()}")
        custom_value = oui + subtype + fuzz_payload
        extra_tlv = make_tlv(127, custom_value)

    end_tlv = struct.pack('!H', 0)
    return chassis_tlv + port_tlv + ttl_tlv + extra_tlv + end_tlv

def classify_payload(payload):
    if payload.startswith(b"http"):
        return "URL"
    elif all(32 <= b <= 126 for b in payload):  # Printable ASCII
        return "ASCII"
    elif len(payload) > 200:
        return "Long ASCII"
    else:
        return "Binary"

def send_lldp(interface, fuzz_field, fuzz_payload):
    raw_socket = socket.socket(socket.AF_PACKET, socket.SOCK_RAW)
    raw_socket.bind((interface, 0))

    dst_mac = b'\x01\x80\xc2\x00\x00\x0e'
```

```
    src_mac = b'\xaa\xbb\xcc\xdd\xee\xff'
    ethertype = b'\x88\xcc'

    lldp_payload = build_lldp_packet(fuzz_field, fuzz_payload)
    ethernet_frame = dst_mac + src_mac + ethertype + lldp_payload

    ptype = classify_payload(fuzz_payload)
    summary = fuzz_payload[:20].decode('ascii', 'replace')
    print(f"[+] Sending LLDP → field: {fuzz_field}, type: {ptype}, len: {len(fuzz_payload)}, preview: {summary} on {interface}")

    raw_socket.send(ethernet_frame)
    raw_socket.close()

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="LLDP Auto Fuzzer")
    parser.add_argument("interface", help="Network interface (e.g. eth0)")
    parser.add_argument("--delay", type=float, default=2.0, help="Delay between fuzz sends in seconds (default 2s)")
    parser.add_argument("--once", action="store_true", help="Send each payload only once and exit")
    args = parser.parse_args()

    fields = ["system-name", "system-description", "custom"]
    payloads = [
        b"https://phish.local",
        b"A" * 1000,
        b"TestDevice",
        b"\x00\xff\xfe\x01\x02",  # binary garbage example
        b"NormalPayload123"
    ]

    print(f"[=] Starting automatic LLDP fuzzing on {args.interface} with {args.delay}s delay")

    try:
        for field in fields:
            for payload in payloads:
                print(f"\n[~] Fuzzing field '{field}' with payload length {len(payload)}")
                send_lldp(args.interface, field, payload)
                if args.once:
                    sys.exit(0)
                time.sleep(args.delay)
    except KeyboardInterrupt:
        print("\n[!] Interrupted by user. Exiting.")
        sys.exit(0)
```

An example of the code in action is shown in the screenshot below:



9. Enhancing the sniffer's functionality

My main goal for the LLDP sniffer was to identify as many different network devices as possible that we might encounter during penetration tests in the Client's network.
After some minor modifications, the sniffer code is presented in the table below:

```
#!/usr/bin/env python3
from scapy.all import sniff, Ether, Raw
import struct

LLDP_MULTICAST_MAC = "01:80:c2:00:00:0e"
LLDP_ETHERTYPE = 0x88cc

TLV_TYPE_MAP = {
    0: "End of LLDPDU",
    1: "Chassis ID",
    2: "Port ID",
    3: "Time to Live",
    4: "Port Description",
    5: "System Name",
    6: "System Description",
    7: "System Capabilities",
    8: "Management Address"
}

CAPABILITY_NAMES = [
    (0, "Other"),
    (1, "Repeater"),
    (2, "Bridge"),
    (3, "WLAN Access Point"),
```

```
    (4, "Router"),
    (5, "Telephone"),
    (6, "DOCSIS Cable Device"),
    (7, "Station Only"),
    (8, "C-VLAN Component"),
    (9, "S-VLAN Component"),
    (10, "Two-port MAC Relay")
]

device_signatures = {
    'cisco': {
        'system-name': ['Cisco', 'Catalyst', 'IOS'],
        'system-description': ['Cisco IOS Software', 'Cisco Systems'],
        'oui': ['00:0c:29', '00:1b:54', '00:1e:49'],
    },
    'juniper': {
        'system-name': ['Juniper'],
        'system-description': ['Juniper Networks'],
        'oui': ['00:05:85', '00:1c:73'],
    },
    'paloalto': {
        'system-name': ['PA-', 'PaloAlto'],
        'system-description': ['Palo Alto Networks'],
        'oui': ['00:0c:29'],
    }
}

def decode_capabilities(bits):
    return " + ".join(name for bit, name in CAPABILITY_NAMES if bits & (1 << bit)) or "None"

def match_vendor(system_name, system_description, chassis_mac):
    system_name = system_name.lower() if system_name else ""
    system_description = system_description.lower() if system_description else ""
    oui = chassis_mac.lower()[0:8] if chassis_mac else ""

    for vendor, patterns in device_signatures.items():
        # Check system-name
        if any(pat.lower() in system_name for pat in patterns.get('system-name', [])):
            return vendor
        # Check system-description
        if any(pat.lower() in system_description for pat in patterns.get('system-description', [])):
            return vendor
        # Check OUI
        if any(oui.startswith(oui_pat) for oui_pat in patterns.get('oui', [])):
            return vendor
    return None

def parse_lldpdu(payload):
    idx = 0
    seen = set()

    chassis_mac = None
    system_name = None
    system_description = None

    while idx + 2 <= len(payload):
        tlv_header = struct.unpack("!H", payload[idx:idx+2])[0]
```

```
        tlv_type = (tlv_header >> 9) & 0x7F
        tlv_len = tlv_header & 0x1FF
        idx += 2

        if tlv_len == 0 or idx + tlv_len > len(payload):
            break

        value = payload[idx:idx+tlv_len]
        idx += tlv_len

        desc = TLV_TYPE_MAP.get(tlv_type, f"Unknown TLV ({tlv_type})")
        seen.add(tlv_type)

        if tlv_type == 1:
            subtype = value[0]
            data = value[1:]
            subtype_str = {
                1: "Chassis MAC",
                2: "Interface name",
                4: "Local"
            }.get(subtype, f"Subtype {subtype}")
            try:
                output_str = data.decode("utf-8")
                if not output_str.strip():
                    raise ValueError
            except:
                output_str = data.hex(":")
            print(f"  Chassis ID ({subtype_str}): {output_str}")
            if subtype == 1:
                chassis_mac = output_str

        elif tlv_type == 2:
            subtype = value[0]
            data = value[1:]
            subtype_str = {
                3: "MAC address",
                5: "Interface name",
                7: "Local"
            }.get(subtype, f"Subtype {subtype}")
            output = data.hex(":") if subtype == 3 else data.decode(errors='ignore')
            print(f"  Port ID ({subtype_str}): {output}")

        elif tlv_type == 3:
            ttl = struct.unpack("!H", value)[0]
            print(f"  Time to Live: {ttl} sec")

        elif tlv_type == 4:
            print(f"  Port Description: {value.decode(errors='ignore')}")

        elif tlv_type == 5:
            decoded = value.decode(errors='ignore')
            print(f"  System Name: {decoded}  ← peer hostname")
            system_name = decoded

        elif tlv_type == 6:
            decoded = value.decode(errors='ignore')
            print(f"  System Description: {decoded}  ← peer OS or platform")
```

```
            system_description = decoded

        elif tlv_type == 7:
            if len(value) >= 4:
                sys_caps, enabled_caps = struct.unpack("!HH", value[:4])
                print(f"□ System Capabilities: {decode_capabilities(sys_caps)} (0x{sys_caps:04x})")
                print(f"□ Enabled Capabilities: {decode_capabilities(enabled_caps)} (0x{enabled_caps:04x})")
            else:
                print("□ System Capabilities: malformed")

        elif tlv_type == 8:
            mgmt_addr_len = value[0]
            mgmt_addr = value[1:1+mgmt_addr_len]
            addr_str = ".".join(str(b) for b in mgmt_addr) if mgmt_addr_len == 4 else mgmt_addr.hex()
            print(f"□ Management Address: {addr_str}")

        elif tlv_type == 0:
            print("□ End of LLDPDU")

        else:
            print(f"□ {desc}: {value.hex()}")

    # □ Wypisz placeholdery dla brakujących TLV
    for required in [1, 2, 3, 4, 5, 6, 7, 8]:
        if required not in seen:
            label = TLV_TYPE_MAP.get(required, f"TLV {required}")
            print(f"□ {label}: [brak]")

    vendor = match_vendor(system_name, system_description, chassis_mac)
    if vendor:
        print(f"□ Detected vendor: {vendor.upper()}")

def handle_packet(pkt):
    if Ether in pkt and pkt.type == LLDP_ETHERTYPE:
        eth = pkt[Ether]
        print(f"\n□ LLDP packet received from {eth.src}")
        if Raw in pkt:
            payload = pkt[Raw].load
            parse_lldpdu(payload)

print("□ Sniffing LLDP frames (press Ctrl+C to stop)...\n")
sniff(filter="ether proto 0x88cc", prn=handle_packet, store=0, iface="eth0")
```

An example of the sniffer's operation is shown in the screenshot below:

To enhance the detection of other devices, I modified the code — the final version is presented in the table below:

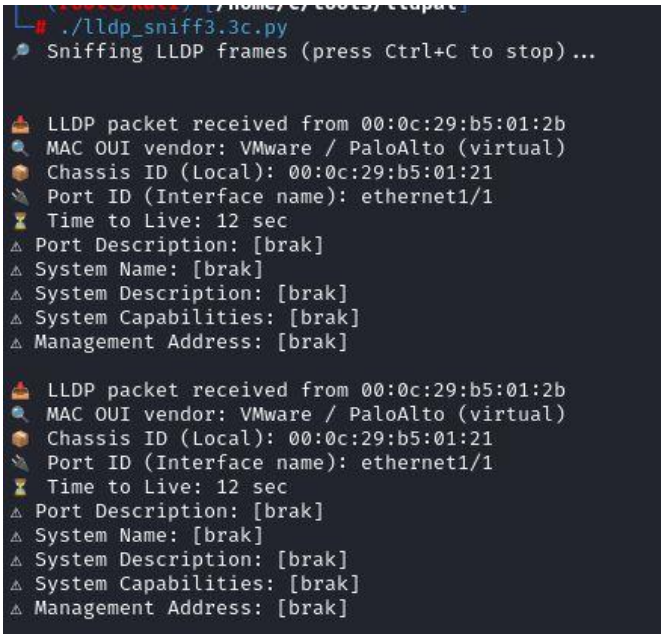
```
#!/usr/bin/env python3
from scapy.all import sniff, Ether, Raw
import struct

LLDP_MULTICAST_MAC = "01:80:c2:00:00:0e"
LLDP_ETHERTYPE = 0x88cc

TLV_TYPE_MAP = {
   0: "End of LLDPDU",
   1: "Chassis ID",
   2: "Port ID",
   3: "Time to Live",
   4: "Port Description",
   5: "System Name",
   6: "System Description",
   7: "System Capabilities",
   8: "Management Address"
}

CAPABILITY_NAMES = [
   (0, "Other"),
   (1, "Repeater"),
   (2, "Bridge"),
   (3, "WLAN Access Point"),
   (4, "Router"),
   (5, "Telephone"),
   (6, "DOCSIS Cable Device"),
   (7, "Station Only"),
   (8, "C-VLAN Component"),
   (9, "S-VLAN Component"),
   (10, "Two-port MAC Relay")
]

device_signatures = {
   'cisco': {
      'system-name': ['Cisco', 'Catalyst', 'IOS'],
      'system-description': ['Cisco IOS Software', 'Cisco Systems'],
      'oui': ['00:0c:29', '00:1b:54', '00:1e:49'],
   },
   'juniper': {
      'system-name': ['Juniper'],
      'system-description': ['Juniper Networks'],
      'oui': ['00:05:85', '00:1c:73'],
   },
   'paloalto': {
      'system-name': ['PA-', 'PaloAlto'],
      'system-description': ['Palo Alto Networks'],
      'oui': ['00:0c:29'],
   }
}

# Mapa OUI → Vendor
oui_map = {
   "00:0c:29": "VMware / PaloAlto (virtual)",
   "00:1b:54": "Cisco",
   "00:1e:49": "Cisco",
   "00:05:85": "Juniper",
   "00:1c:73": "Juniper",
   "00:1a:8c": "HPE",
```

```
    "dc:9f:db": "Ubiquiti",
    "00:50:56": "VMware",
    "b0:6e:bf": "Aruba",
}

def decode_capabilities(bits):
    return " + ".join(name for bit, name in CAPABILITY_NAMES if bits & (1 << bit)) or "None"

def match_vendor(system_name, system_description, chassis_mac):
    system_name = system_name.lower() if system_name else ""
    system_description = system_description.lower() if system_description else ""
    oui = chassis_mac.lower()[0:8] if chassis_mac else ""

    for vendor, patterns in device_signatures.items():
        if any(pat.lower() in system_name for pat in patterns.get('system-name', [])):
            return vendor
        if any(pat.lower() in system_description for pat in patterns.get('system-description', [])):
            return vendor
        if any(oui.startswith(oui_pat) for oui_pat in patterns.get('oui', [])):
            return vendor
    return None

def parse_lldpdu(payload):
    idx = 0
    seen = set()

    chassis_mac = None
    system_name = None
    system_description = None

    while idx + 2 <= len(payload):
        tlv_header = struct.unpack("!H", payload[idx:idx+2])[0]
        tlv_type = (tlv_header >> 9) & 0x7F
        tlv_len = tlv_header & 0x1FF
        idx += 2

        if tlv_len == 0 or idx + tlv_len > len(payload):
            break

        value = payload[idx:idx+tlv_len]
        idx += tlv_len

        desc = TLV_TYPE_MAP.get(tlv_type, f"Unknown TLV ({tlv_type})")
        seen.add(tlv_type)

        if tlv_type == 1:
            subtype = value[0]
            data = value[1:]
            subtype_str = {
                1: "Chassis MAC",
                2: "Interface name",
                4: "Local"
            }.get(subtype, f"Subtype {subtype}")
            try:
                output_str = data.decode("utf-8")
                if not output_str.strip():
                    raise ValueError
            except:
                output_str = data.hex(":")
            print(f"🔹 Chassis ID ({subtype_str}): {output_str}")
            if subtype == 1:
                chassis_mac = output_str
```

```
        elif tlv_type == 2:
            subtype = value[0]
            data = value[1:]
            subtype_str = {
                3: "MAC address",
                5: "Interface name",
                7: "Local"
            }.get(subtype, f"Subtype {subtype}")
            output = data.hex(":") if subtype == 3 else data.decode(errors='ignore')
            print(f"□ Port ID ({subtype_str}): {output}")

        elif tlv_type == 3:
            ttl = struct.unpack("!H", value)[0]
            print(f"□ Time to Live: {ttl} sec")

        elif tlv_type == 4:
            print(f"□ Port Description: {value.decode(errors='ignore')}")

        elif tlv_type == 5:
            decoded = value.decode(errors='ignore')
            print(f"□ System Name: {decoded} ← peer hostname")
            system_name = decoded

        elif tlv_type == 6:
            decoded = value.decode(errors='ignore')
            print(f"□ System Description: {decoded} ← peer OS or platform")
            system_description = decoded

        elif tlv_type == 7:
            if len(value) >= 4:
                sys_caps, enabled_caps = struct.unpack("!HH", value[:4])
                print(f"□ System Capabilities: {decode_capabilities(sys_caps)} (0x{sys_caps:04x})")
                print(f"□ Enabled Capabilities: {decode_capabilities(enabled_caps)} (0x{enabled_caps:04x})")
            else:
                print("□ System Capabilities: malformed")

        elif tlv_type == 8:
            mgmt_addr_len = value[0]
            mgmt_addr = value[1:1+mgmt_addr_len]
            addr_str = ".".join(str(b) for b in mgmt_addr) if mgmt_addr_len == 4 else mgmt_addr.hex()
            print(f"□ Management Address: {addr_str}")

        elif tlv_type == 0:
            print("□ End of LLDPDU")

        else:
            print(f"□ {desc}: {value.hex()}")

    for required in [1, 2, 3, 4, 5, 6, 7, 8]:
        if required not in seen:
            label = TLV_TYPE_MAP.get(required, f"TLV {required}")
            print(f"□ {label}: [brak]")

    vendor = match_vendor(system_name, system_description, chassis_mac)
    if vendor:
        print(f"□ Detected vendor: {vendor.upper()}")

def handle_packet(pkt):
    if Ether in pkt and pkt.type == LLDP_ETHERTYPE:
        eth = pkt[Ether]
        print(f"\n□ LLDP packet received from {eth.src}")
        oui_prefix = eth.src.lower()[0:8]
        vendor_from_oui = oui_map.get(oui_prefix)
```

```
    if vendor_from_oui:
        print(f"🔍 MAC OUI vendor: {vendor_from_oui}")
    if Raw in pkt:
        payload = pkt[Raw].load
        parse_lldpdu(payload)

print("🔎 Sniffing LLDP frames (press Ctrl+C to stop)...\n")
sniff(filter="ether proto 0x88cc", prn=handle_packet, store=0, iface="eth0")
```

The operation of the updated version is shown in the screenshot below:



10. Outro

Here are a few links that were helpful to me while creating this document:

- https://security.paloaltonetworks.com/CVE-2025-0116
- https://nvd.nist.gov/vuln/detail/cve-2024-20294
- https://www.armis.com/research/cdpwn
- https://docs.paloaltonetworks.com/pan-os/11-1/pan-os-networking-admin/lldp/configure-lldp
- https://www.watchguard.com/help/docs/help-center/en-US/Content/en-US/Wi-Fi-Cloud/discover/monitor/monitor_switch.html
- https://code610.blogspot.com

I encourage you to continue exploring on your own. ;)

Cheers