PYTHON ARP Monitor

Contents

Time required: 60 minutes

What is a MAC Address?

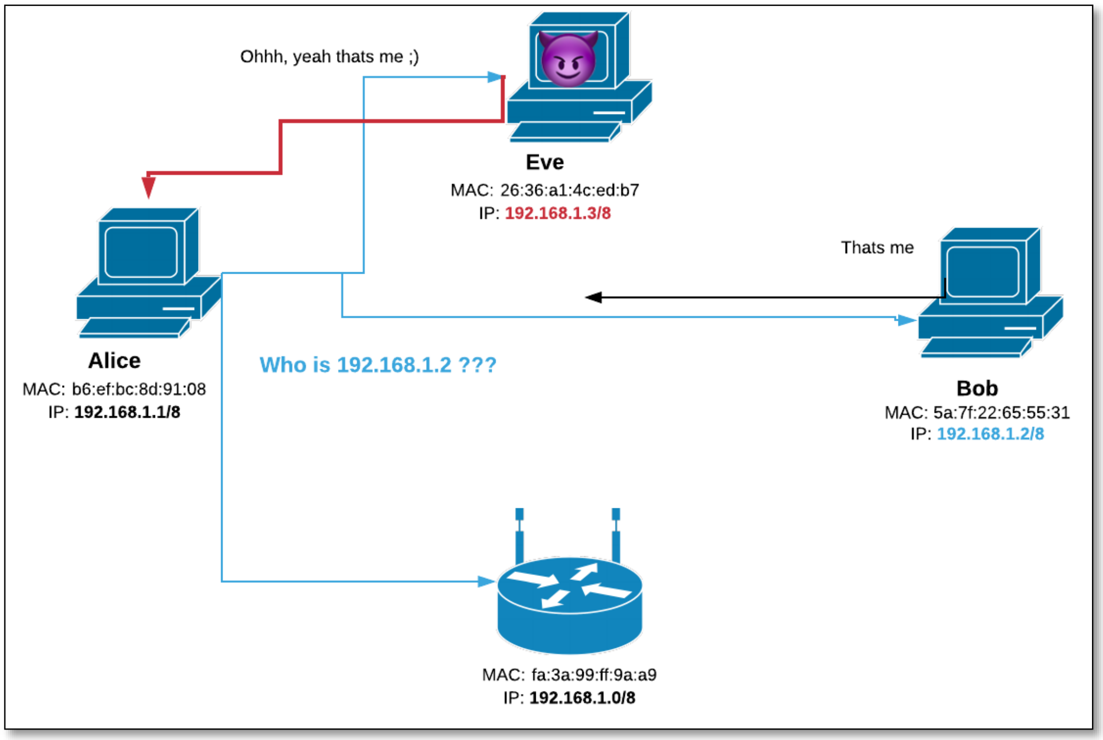
MAC addresses (Media Access Control or physical addresses) are unique identifiers assigned to network devices at the hardware level. They are used to uniquely identify devices within a local network. MAC addresses are assigned by manufacturers and are usually hardcoded into the network interface card (NIC) or adapter of a device. These addresses are composed of 12 hexadecimal characters (0-9 and A-F) grouped in pairs and are typically displayed in a format like XX:XX:XX:XX:XX:XX.

What is ARP?

Address Resolution Protocol (ARP) is a protocol used in computer networking to map an IP address (logical address) to a MAC address (physical address) within a local network.

When a device wants to communicate with another device on the same network, it needs to know the MAC address of the destination device. ARP helps in resolving this by maintaining a table (ARP table or cache) that maps IP addresses to corresponding MAC addresses. If a device wants to communicate with an IP address on its local network and does not have the MAC address in its ARP cache, it will broadcast an ARP request asking, "Who has this IP address?" The device with the corresponding IP address will respond with its MAC address, and this information is then stored in the sender's ARP cache for future reference, facilitating direct communication between devices on the same network.

Purpose: ARP helps devices on a local network find the MAC address associated with an IP address. Since data transmission within a local network relies on MAC addresses, ARP facilitates communication between devices by resolving IP addresses to MAC addresses.



1. **How it Works:** When a device wants to communicate with another device in the same network and knows the IP address but not the MAC address, it sends out an ARP request broadcast packet containing the target IP address. This broadcast is received by all devices on the local network.

2. **Response:** The device that has the IP address specified in the ARP request responds directly to the requesting device with its MAC address. This response is unicast and not broadcast, providing the requesting device with the necessary MAC address mapping.

3. **ARP Cache**: To optimize performance, devices maintain an ARP cache that stores recently resolved IP-to-MAC address mappings. This cache helps avoid repeated ARP requests for devices already communicated with recently.

ARP is crucial for local network communication by facilitating the translation between IP addresses, which are used for logical addressing, and MAC addresses, which are used for physical addressing within a network segment.

Install nmap

We will be using the Python scapy library. It requires something to interact with the packets.

Install nmap if you haven't already. Nmap.org

Tutorial 1: ARP Monitor In Windows

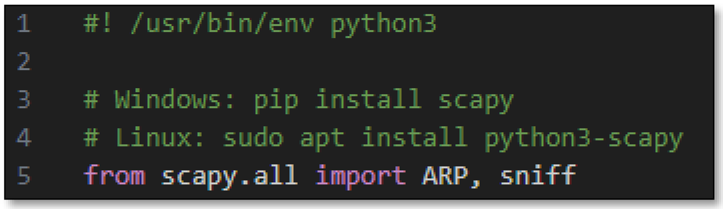
Let's create a Python program to monitor ARP packets called **arp_monitor.py**

We will run this program in Windows and Linux.

Start by creating the program in Windows with VSCode.

Install the scapy library.

```
# Windows
pip install scapy
```



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

# Windows: pip install scapy
# Linux: sudo apt install python3-scapy
from scapy.all import ARP, sniff
```

The Python Scapy library is a powerful packet manipulation tool that facilitates network programming, interaction, and analysis at a low level. Its purpose is to construct, capture, manipulate, and decode network packets to perform various tasks like network discovery, packet crafting, sniffing, and scanning.

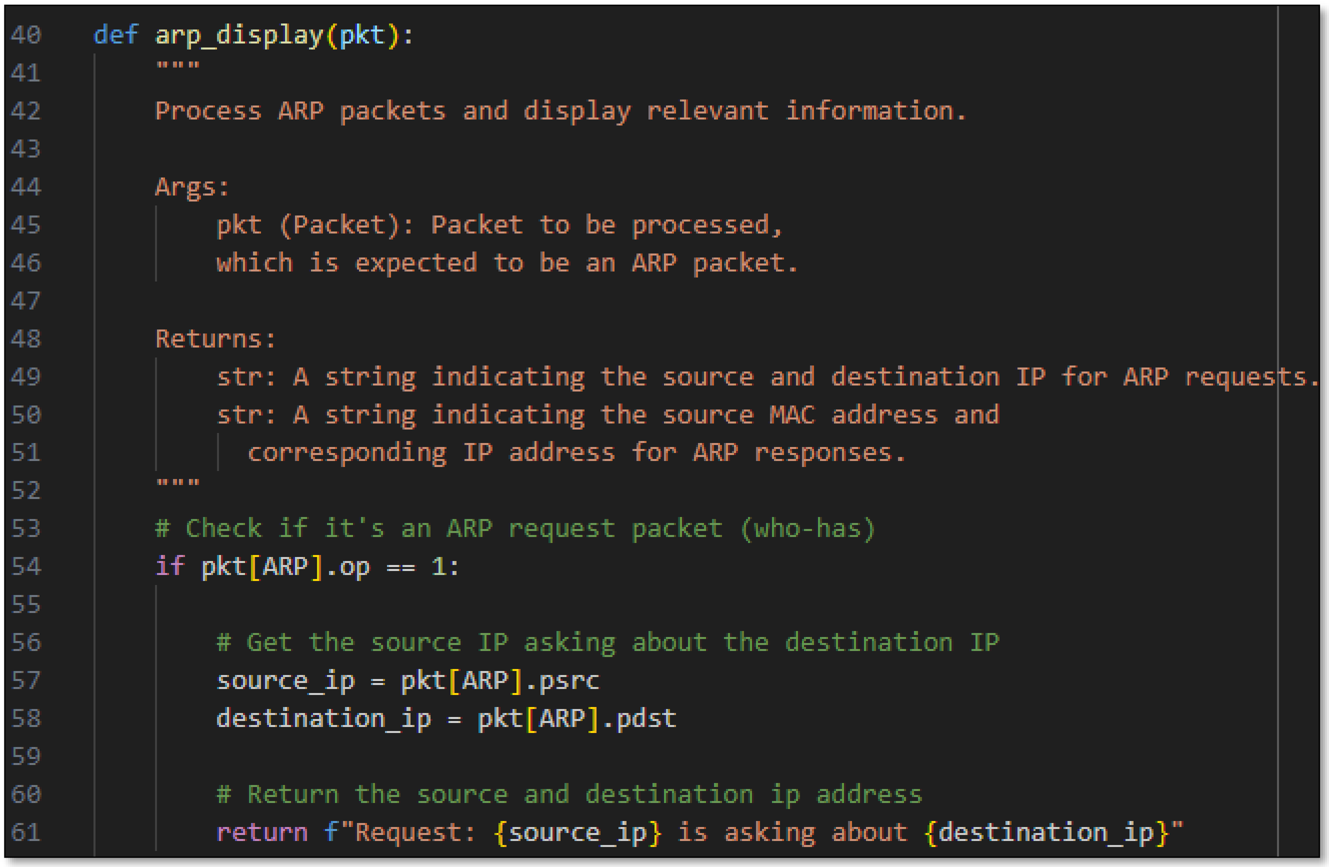
This Python script uses the **scapy** library to perform an ARP (Address Resolution Protocol) sniffing operation. Here's a breakdown of the code:

#!/usr/bin/env python3

This is called a shebang line. It's used in Unix-like operating systems to indicate to the system that the script should be executed using the Python interpreter located at \`/usr/bin/env python3\`. Make sure this is the very top line.

from scapy.all import ARP, sniff

This line imports specific classes (\`ARP\` and \`sniff\`) from the \`scapy\` library. \`ARP\` is used for crafting ARP packets, while \`sniff\` is a function used for packet sniffing and processing.

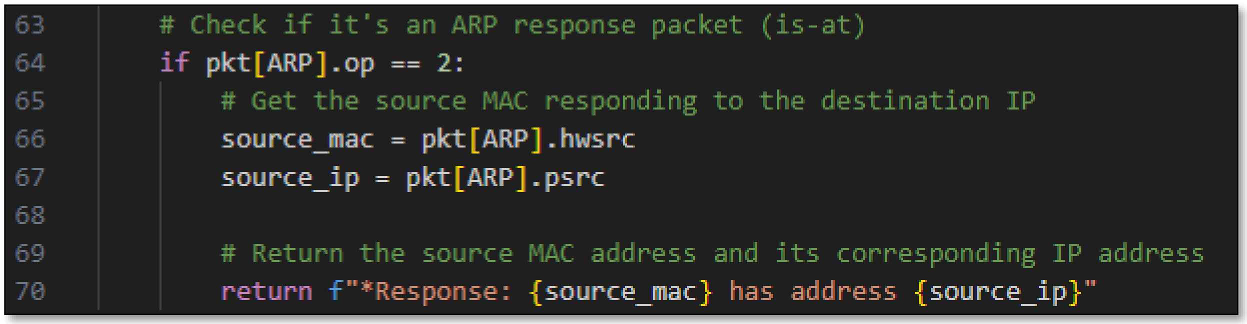


```python
def arp_display(pkt):
    """
    Process ARP packets and display relevant information.

    Args:
        pkt (Packet): Packet to be processed,
        which is expected to be an ARP packet.

    Returns:
        str: A string indicating the source and destination IP for ARP requests.
        str: A string indicating the source MAC address and
          corresponding IP address for ARP responses.
    """
    # Check if it's an ARP request packet (who-has)
    if pkt[ARP].op == 1:

        # Get the source IP asking about the destination IP
        source_ip = pkt[ARP].psrc
        destination_ip = pkt[ARP].pdst

        # Return the source and destination ip address
        return f"Request: {source_ip} is asking about {destination_ip}"
```

def arp_display(pkt) - Defines a function named `arp_display` that takes a packet (`pkt`) as an argument. This function is called for each packet captured by the `sniff` function.

if pkt[ARP].op == 1: - Checks if the packet's ARP operation code (`op`) is equal to 1, which indicates an ARP request (`who-has`).

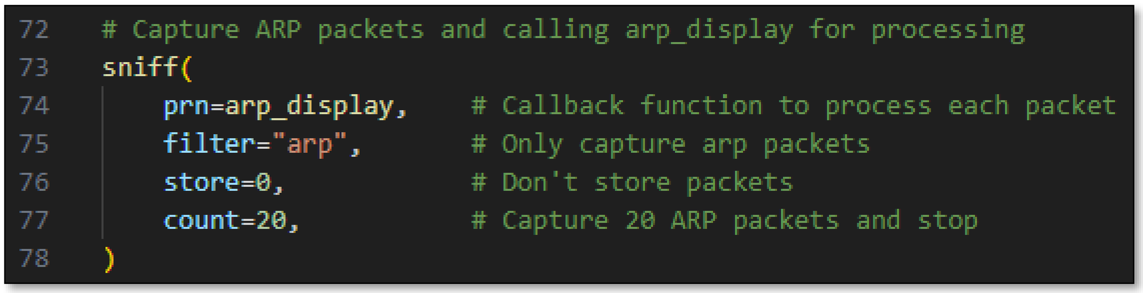
**return f"Request: {pkt[ARP].psrc} is asking about {pkt[ARP].pdst}"`** :- If it's an ARP request, this line generates a string indicating the source IP address (`pkt[ARP].psrc`) asking about the destination IP address (`pkt[ARP].pdst]`).



```python
    # Check if it's an ARP response packet (is-at)
    if pkt[ARP].op == 2:
        # Get the source MAC responding to the destination IP
        source_mac = pkt[ARP].hwsrc
        source_ip = pkt[ARP].psrc

        # Return the source MAC address and its corresponding IP address
        return f"*Response: {source_mac} has address {source_ip}"
```

if pkt[ARP].op == 2: - Checks if the packet's ARP operation code is equal to 2, indicating an ARP response (`is-at`).

return f"*Response: {pkt[ARP].hwsrc} has address {pkt[ARP].psrc}" - If it's an ARP response, this line generates a string indicating the source hardware (MAC) address (`pkt[ARP].hwsrc`) and its corresponding IP address (`pkt[ARP].psrc`).



```python
# Capture ARP packets and calling arp_display for processing
sniff(
    prn=arp_display,    # Callback function to process each packet
    filter="arp",       # Only capture arp packets
    store=0,            # Don't store packets
    count=20,           # Capture 20 ARP packets and stop
)
```

This code is what executes the program. It must be aligned to the left side.

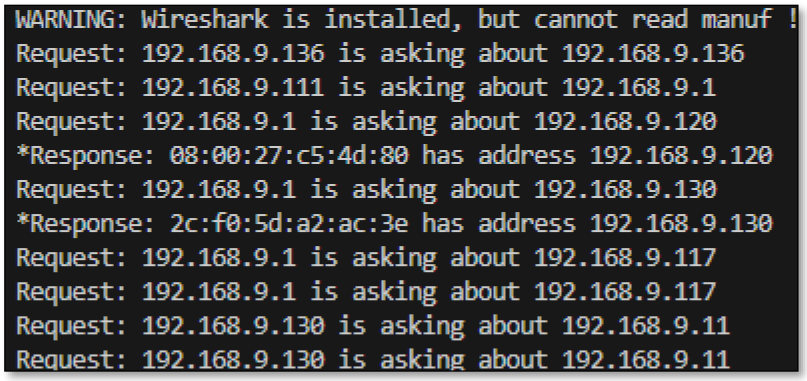
sniff(prn=arp_display, filter="arp", store=0, count=20) - Initiates the packet sniffing process. It captures ARP packets (`filter="arp"`) and processes them using the `arp_display` function (`prn=arp_display`). The `count=20` parameter limits the sniffing to the first 10 packets, and `store=0` specifies that captured packets should not be stored in memory.

Example run:

The program will capture 20 packets. Go to a web site. That should give you some ARP requests and responses.

Please complete this lab on Windows and Linux.

Windows response. If you have Wireshark installed in Windows, you may see this error. Don’t worry about it. The program still works just fine.



```
WARNING: Wireshark is installed, but cannot read manuf !
Request: 192.168.9.136 is asking about 192.168.9.136
Request: 192.168.9.111 is asking about 192.168.9.1
Request: 192.168.9.1 is asking about 192.168.9.120
*Response: 08:00:27:c5:4d:80 has address 192.168.9.120
Request: 192.168.9.1 is asking about 192.168.9.130
*Response: 2c:f0:5d:a2:ac:3e has address 192.168.9.130
Request: 192.168.9.1 is asking about 192.168.9.117
Request: 192.168.9.1 is asking about 192.168.9.117
Request: 192.168.9.130 is asking about 192.168.9.11
Request: 192.168.9.130 is asking about 192.168.9.11
```

Tutorial 2: ARP Monitor in Kali Linux

1. Change your Kali Linux VM to the Bridged Adapter. This will allow you to monitor ARP packets on your local network.

2. Update Kali Linux.

```
sudo apt update
sudo apt upgrade
```

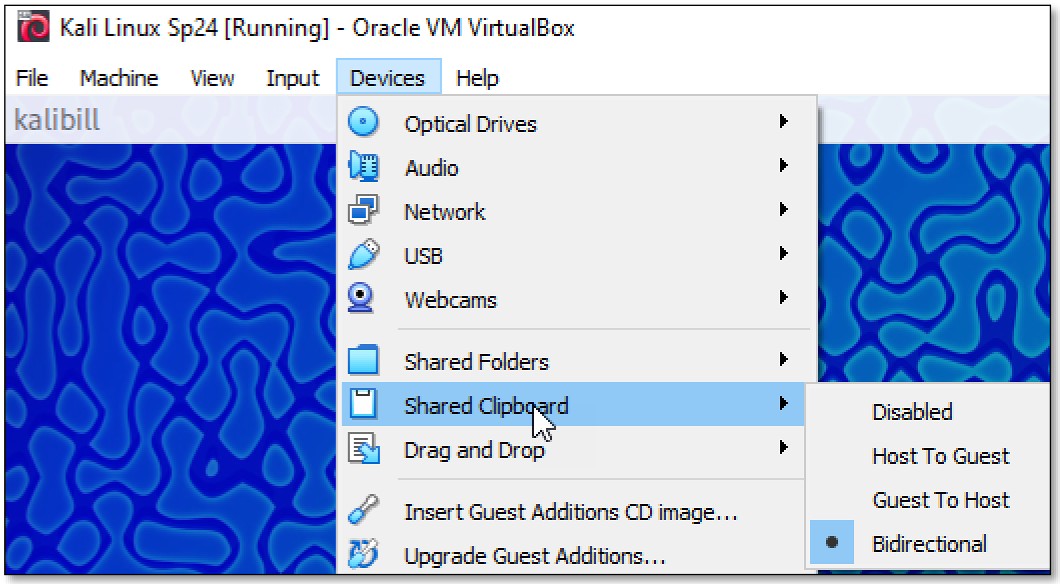
Geany is a simple code editor that can be used Linux.

3. Install Geany.

```
sudo apt install geany
```

To copy and paste code from your local computer to the Kali Linux VM, we want to enable the clipboard in VirtualBox

4. In the Kali Linux VM at the top of the screen → Devices → Shared Clipboard → Bidirectional.



5. Open a terminal in Kali Linux → Create a Code folder.

```
mkdir Code
```

6. Whenever you want to go to the Code folder

```
cd Code
# Use the ls command to see what is in the folder.
ls
```

7. Start Geany and create the arp_monitor.py file.

```
# If arp_monitor.py does not exist, this command will create it.
# If arp_monitor.py does exist, this command will open it.
geany arp_monitor.py
```

8. On your local computer → Copy the arp_monitor.py code.

9. Go to Kali Linux to Geany. Paste the code. (CTRL-V)

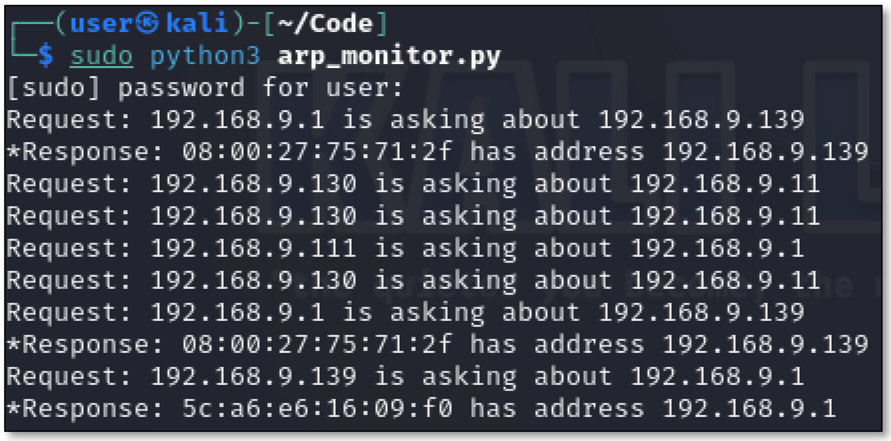
10. Save the file.

11. Open another terminal.

12. The new terminal automatically starts in your home folder. cd to the Code folder.

```
# Change directory to your Code folder.
cd Code
# Run arp_monitor.py
sudo python3 arp_monitor.py
```

Kali Linux response.



```
┌──(user㉿kali)-[~/Code]
└─$ sudo python3 arp_monitor.py
[sudo] password for user:
Request: 192.168.9.1 is asking about 192.168.9.139
*Response: 08:00:27:75:71:2f has address 192.168.9.139
Request: 192.168.9.130 is asking about 192.168.9.11
Request: 192.168.9.130 is asking about 192.168.9.11
Request: 192.168.9.111 is asking about 192.168.9.1
Request: 192.168.9.130 is asking about 192.168.9.11
Request: 192.168.9.1 is asking about 192.168.9.139
*Response: 08:00:27:75:71:2f has address 192.168.9.139
Request: 192.168.9.139 is asking about 192.168.9.1
*Response: 5c:a6:e6:16:09:f0 has address 192.168.9.1
```

Assignment Submission

1. Attach the program files.
2. Attach screenshots showing the successful operation of the program.
3. Submit in Blackboard.