Lab - Packet Crafting with Scapy

Objectives

In this lab, you will use Scapy, a Python-based packet manipulation tool, to craft custom packets. These custom packets will be used to perform reconnaissance on a target system.

- Part 1: Investigate the Scapy Tool.
- Part 2: Use Scapy to Sniff Network Traffic.
- Part 3: Create and Send an ICMP Packet.
- Part 4: Create and Send TCP SYN Packets.

Background / Scenario

Penetration testers and ethical hackers often use specially-crafted packets to discover and/or exploit vulnerabilities in clients' infrastructure and systems. You have used Nmap to scan and analyze vulnerabilities in a computer attached to the local network. In this lab, you will perform further reconnaissance on the same computer using custom ICMP and TCP packets.

Required Resources

- Kali VM customized for Ethical Hacker course
- Internet Access

Instructions

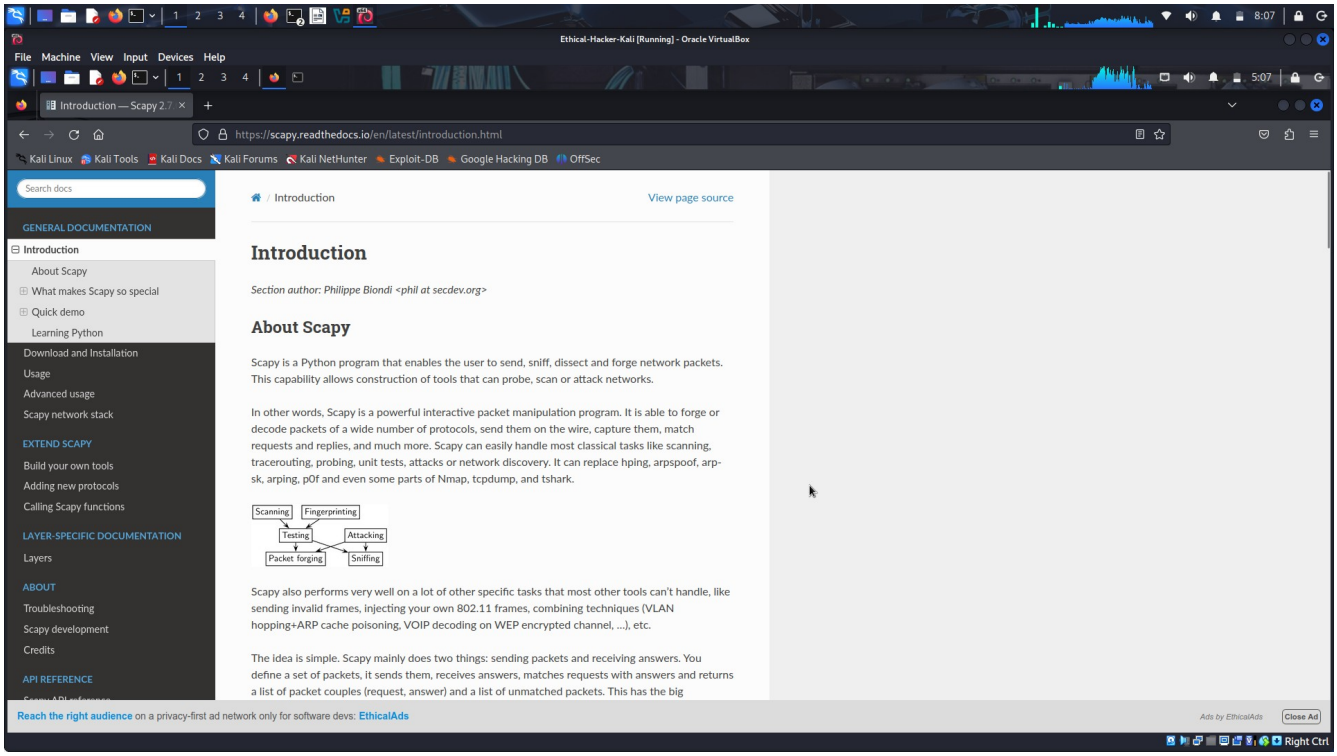
Part 1: Investigate the Scapy Tool

Scapy is a multi-purpose tool originally written by Philippe Biondi. In this part, you will load the Scapy tool and explore some of its capabilities. Tools like Scapy should only be used to scan or communicate with machines that you own or have written permission to access.

Step 1: Investigate Scapy documentation and resources.

Scapy can be run interactively from the Python shell or can be incorporated into Python programs by importing the python-scapy module.

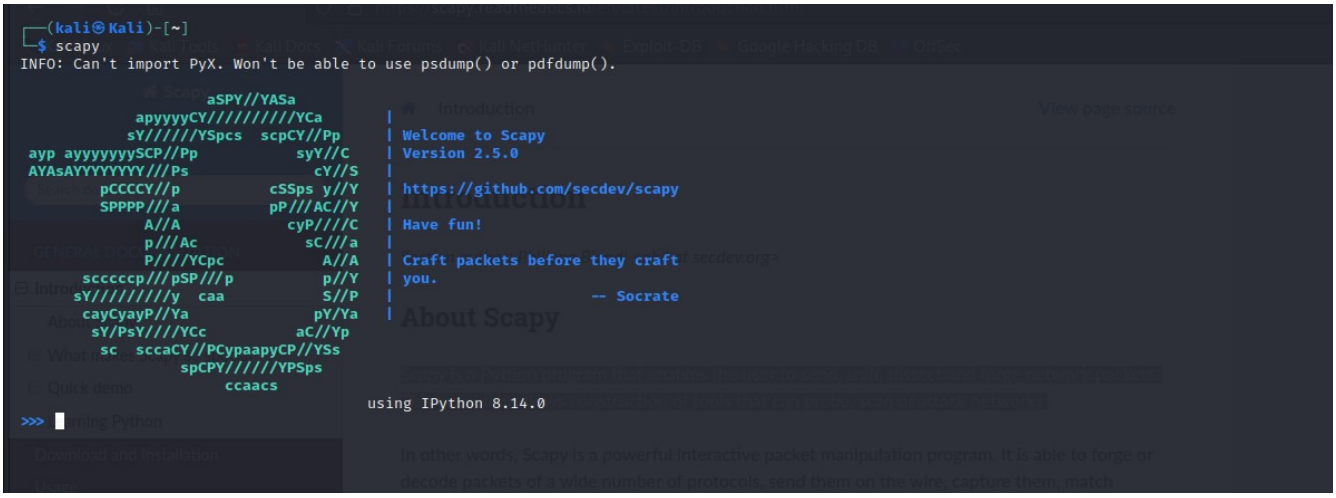
a. Start the Kali VM and login.
b. Open the Firefox browser and navigate to https://scapy.readthedocs.io/en/latest/introduction.html. Read the Introduction page written by the Scapy creator, Philippe Biondi.



How does the author describe the capabilities of Scapy in the first paragraph of the page? S**capy is a Python program that enables the user to send, sniff, dissect and forge network packets. This capability allows construction of tools that can probe, scan or attack networks.**

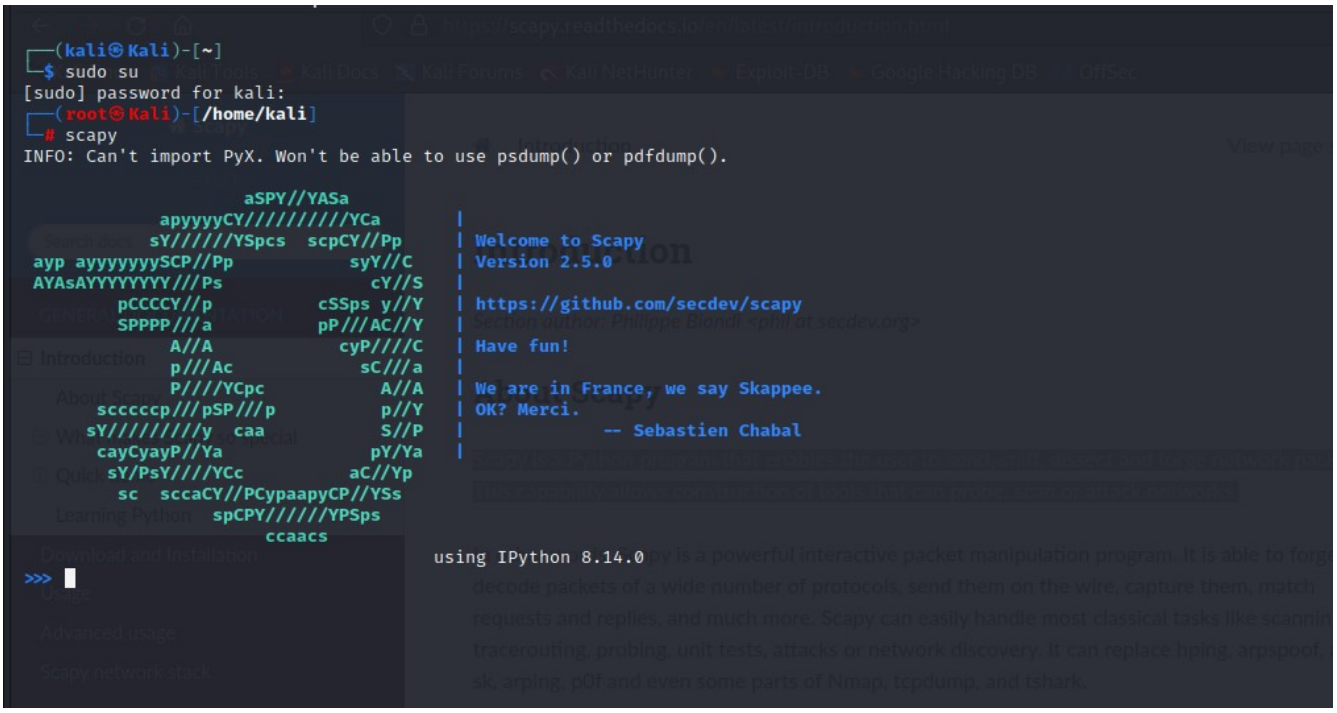
Step 2: Use Scapy interactive command mode.

Enter the **scapy** command in a terminal window to load the Python interpreter. By using this command, the interpreter runs pre-loaded with the Scapy classes and objects. You will enter Scapy commands interactively and receive the output. Scapy commands can also be embedded in a Python script.

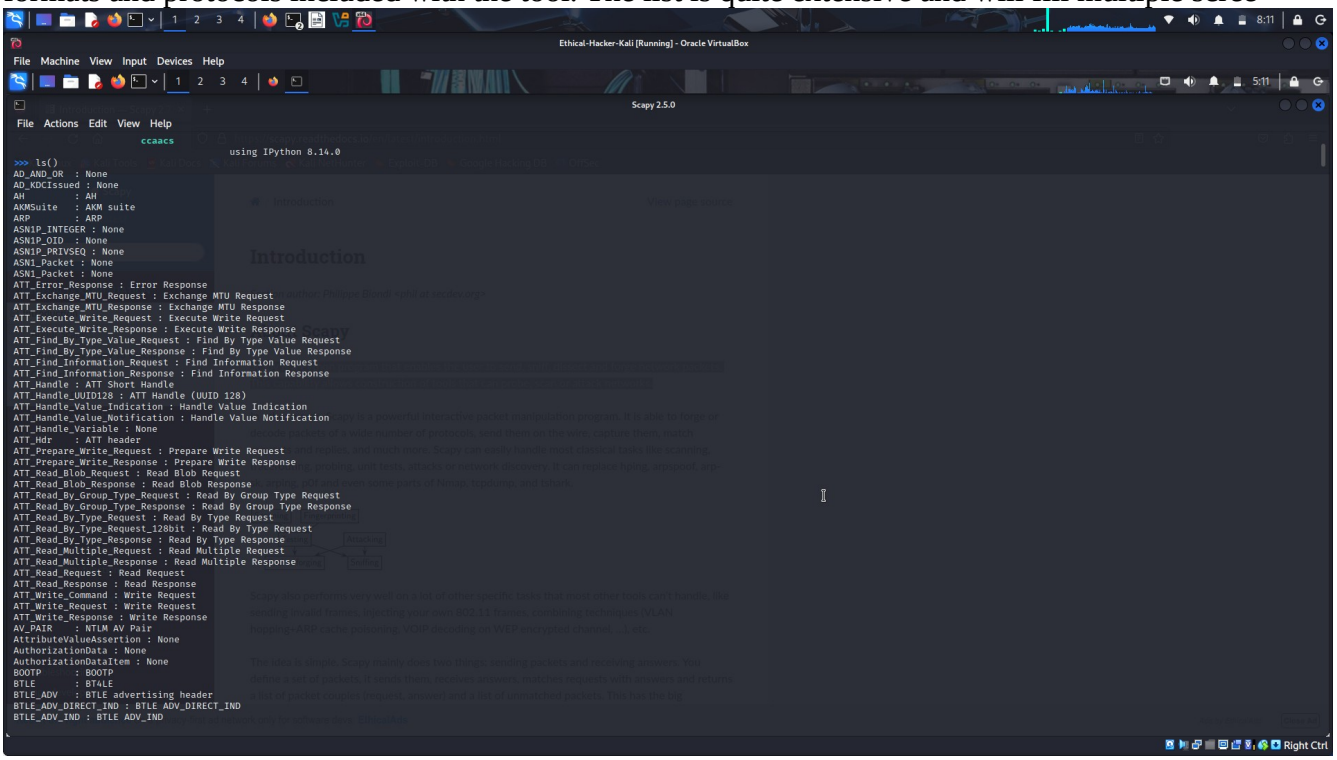


a. The commands to craft and send packets require root privileges to run. Use the **sudo su** command to obtain root privileges before starting Scapy. If prompted for a password, enter **kali**.

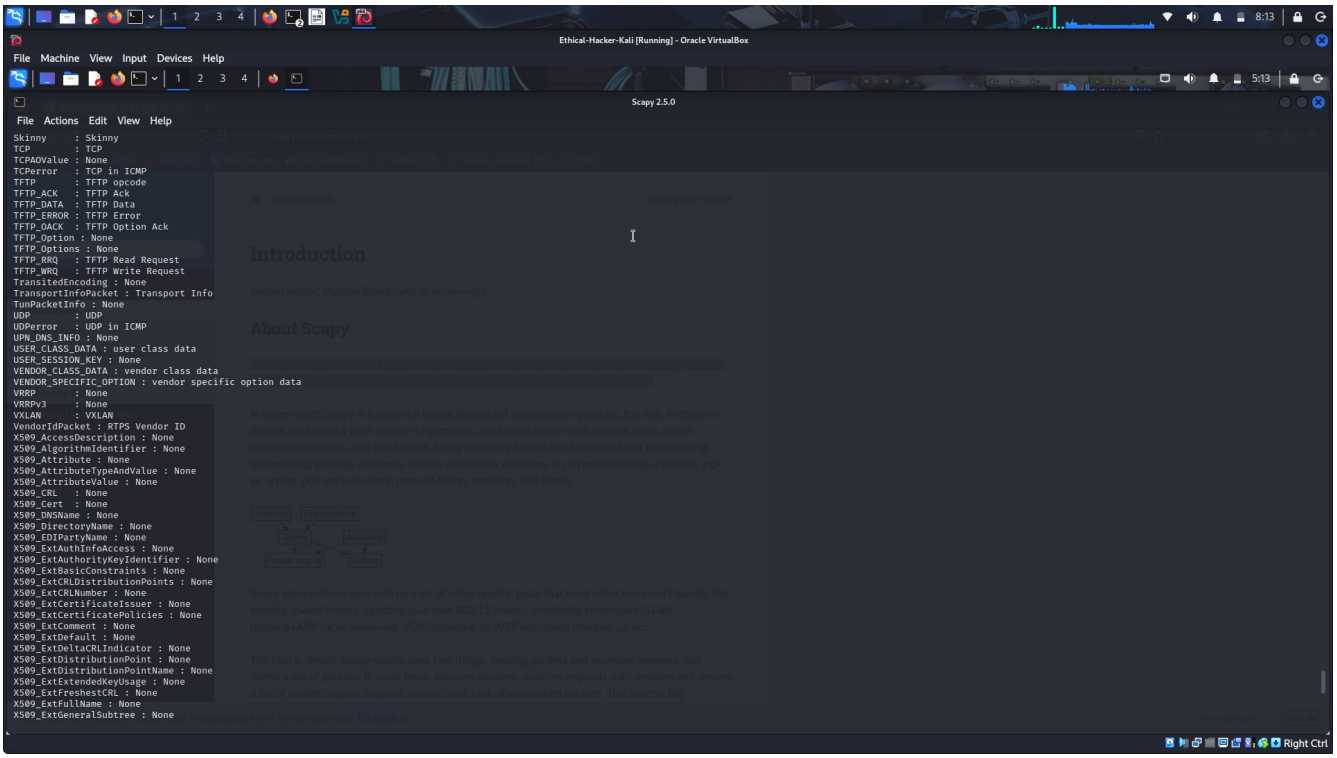
b. Load the Scapy tool using the **scapy** command. The interactive Python interpreter will load and present a screen image similar to that shown.



At the >>> prompt within the Scapy shell, enter the **ls()** function to list all of the available default formats and protocols included with the tool. The list is quite extensive and will fill multiple scree



TFTP is a protocol used to send and receive files on a LAN segment. It is commonly used to back up configuration files on networking devices. Scroll up to view the available TFTP packet formats. How many types of TFTP packet formats are listed? **9**



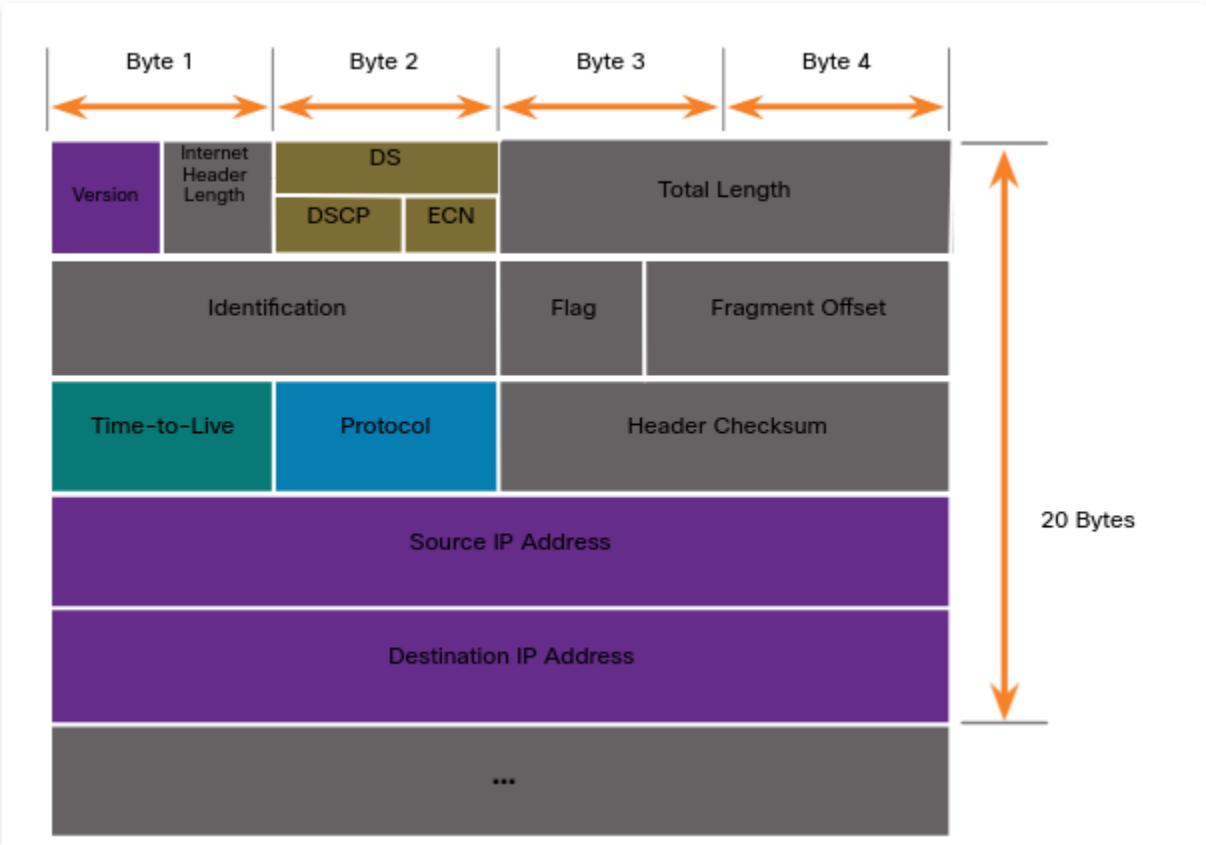
Step 3: Examine the fields in an IPv4 packet header.

a. It is important to understand the structure of an IP packet before creating and sending custom packets over the network. Each IP packet has an associated header that provides information about the structure of the packet. Review this information before continuing with the lab.

IPv4 Packet Header Fields

The binary values of each field identify various settings of the IP packet. Protocol header diagrams, which are read left to right, and top down, provide a visual to refer to when discussing protocol fields. The IP protocol header diagram in the figure identifies the fields of an IPv4 packet.

Fields in the IPv4 Packet Header

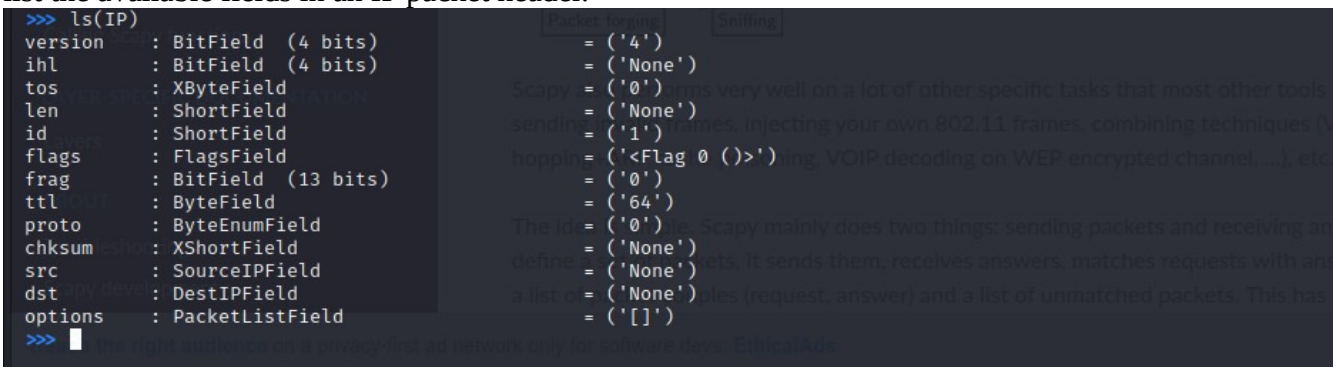


Significant fields in the IPv4 header include the following:

- **Version** - Contains a 4-bit binary value set to 0100 that identifies this as an IPv4 packet.
- **Differentiated Services or DiffServ (DS)** - Formerly called the type of service (ToS) field, the DS field is an 8-bit field used to determine the priority of each packet. The six most significant bits of the DiffServ field are the differentiated services code point (DSCP) bits and the last two bits are the explicit congestion notification (ECN) bits.
- **Time to Live (TTL)** - TTL contains an 8-bit binary value that is used to limit the lifetime of a packet. The source device of the IPv4 packet sets the initial TTL value. It is decreased by one each time the packet is processed by a router. If the TTL field decrements to zero, the router discards the packet and sends an Internet Control Message Protocol (ICMP) Time Exceeded message to the source IP address. Because the router decrements the TTL of each packet, the router must also recalculate the Header Checksum.
- **Protocol** - This field is used to identify the next level protocol. This 8-bit binary value indicates the data payload type that the packet is carrying, which enables the network layer to pass the data to the appropriate upper-layer protocol. Common values include ICMP (1), TCP (6), and UDP (17).
- **Header Checksum** - This is used to detect corruption in the IPv4 header.
- **Source IPv4 Address** - This contains a 32-bit binary value that represents the source IPv4 address of the packet. The source IPv4 address is always a unicast address.

- **Destination IPv4 Address** - This contains a 32-bit binary value that represents the destination IPv4 address of the packet. The destination IPv4 address is a unicast, multicast, or broadcast address.

The **ls()** function can also be used to list details of the fields and options available in each protocol header. The syntax to use a function in Scapy is ***function_name(arguments).*** Use the **ls(IP)** function to list the available fields in an IP packet header.



Compare the fields in the IP detail on Scapy with the packet header described in Step 3a. Are there any differences between the two? **The source IPv4 address (SRC) field.**

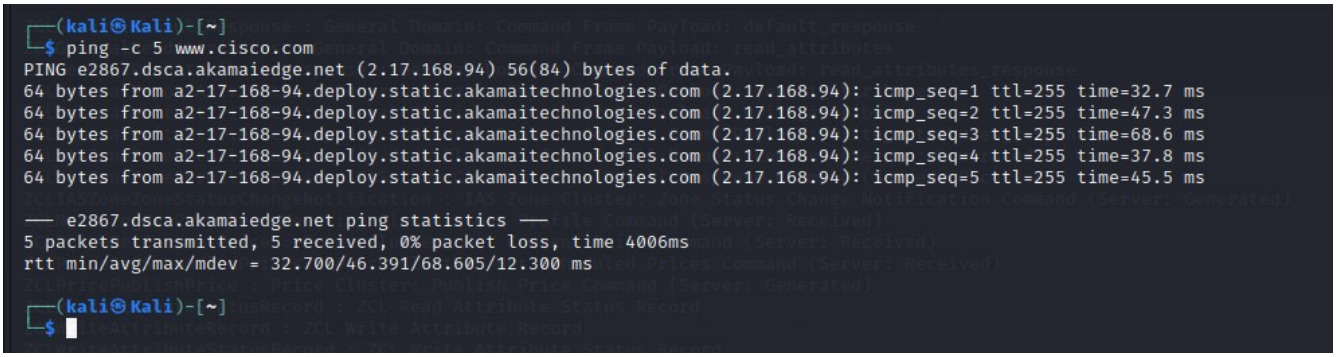
Part 2: Use Scapy to Sniff Network Traffic

Scapy can be used to capture and display network traffic, similar to a tcpdump or tshark packet collection.

Step 1: Use the sniff() function.

a. Use the **sniff()** function to collect traffic using the default eth0 interface of your VM. Start the capture with the **sniff()** function without specifying any arguments.

b. Open a second terminal window and **ping** an internet address, such as **www.cisco.com**. Remember to specify the count using the **-c** argument.



c. Return to the terminal window that is running the Scapy tool. Press **CTRL-C** to stop the capture. You should receive output similar to what is shown here:

```
>>> sniff()
^C<Sniffed: TCP:0 UDP:14 ICMP:10 Other:5>
>>> 
```

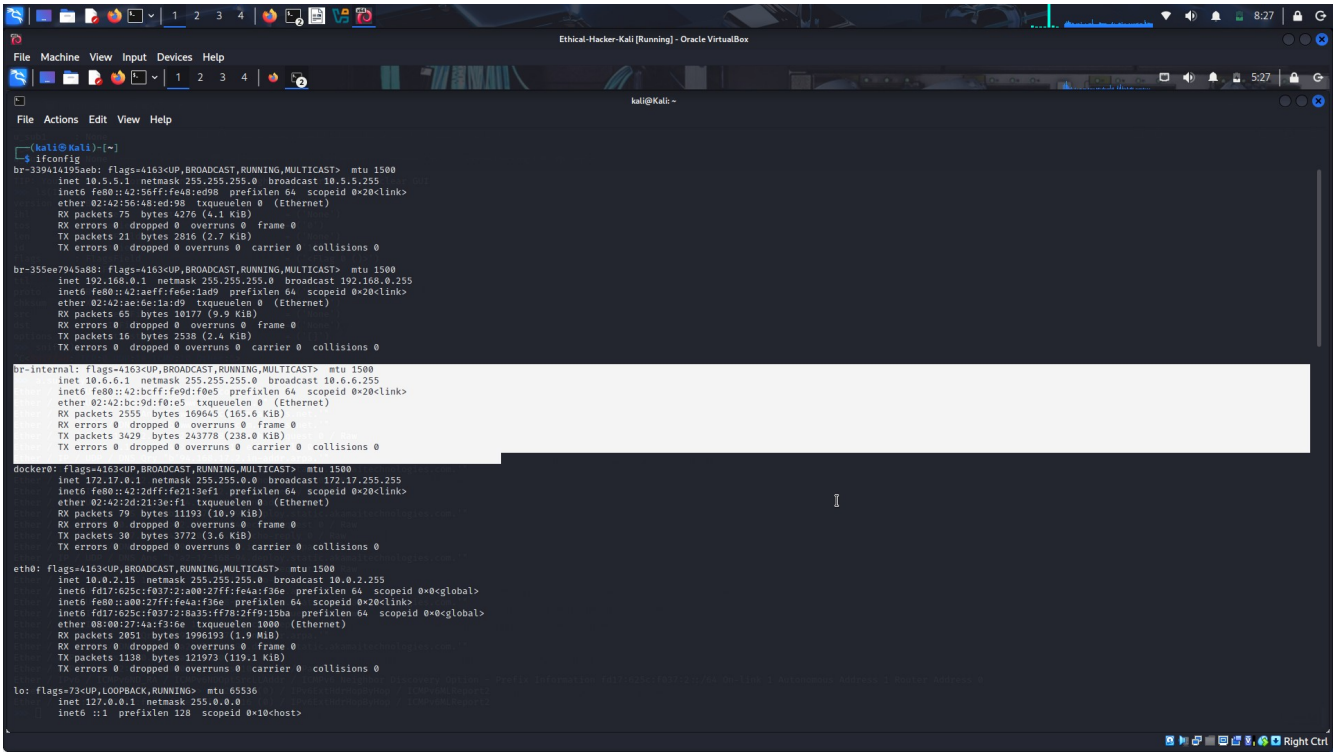
View the captured traffic using the **summary()** function. The **a=_** assigns the variable **a** to hold the output of the **sniff()** function. The underscore (_) in Python is used to temporarily hold the output of the last function executed. The output of this command can be extensive, depending on the applications running on the network.

```
>>> a=_
>>> a.summary()
Ether / IP / UDP / DNS Qry "b'www.cisco.com.'"
Ether / IP / UDP / DNS Qry "b'www.cisco.com.'"
Ether / IP / UDP / DNS Ans "b'www.cisco.com.akadns.net.'"
Ether / IP / UDP / DNS Ans "b'www.cisco.com.akadns.net.'"
Ether / IP / ICMP 10.0.2.15 > 2.17.168.94 echo-request 0 / Raw
Ether / IP / ICMP 2.17.168.94 > 10.0.2.15 echo-reply 0 / Raw
Ether / IP / UDP / DNS Qry "b'94.168.17.2.in-addr.arpa.'"
Ether / IP / UDP / DNS Ans "b'a2-17-168-94.deploy.static.akamaitechnologies.com.'"
Ether / IP / ICMP 10.0.2.15 > 2.17.168.94 echo-request 0 / Raw
Ether / IP / ICMP 2.17.168.94 > 10.0.2.15 echo-reply 0 / Raw
Ether / IP / UDP / DNS Qry "b'94.168.17.2.in-addr.arpa.'"
Ether / IP / UDP / DNS Ans "b'a2-17-168-94.deploy.static.akamaitechnologies.com.'"
Ether / IP / ICMP 10.0.2.15 > 2.17.168.94 echo-request 0 / Raw
Ether / IP / ICMP 2.17.168.94 > 10.0.2.15 echo-reply 0 / Raw
Ether / IP / UDP / DNS Qry "b'94.168.17.2.in-addr.arpa.'"
Ether / IP / UDP / DNS Ans "b'a2-17-168-94.deploy.static.akamaitechnologies.com.'"
Ether / IP / ICMP 10.0.2.15 > 2.17.168.94 echo-request 0 / Raw
Ether / IP / ICMP 2.17.168.94 > 10.0.2.15 echo-reply 0 / Raw
Ether / IP / UDP / DNS Qry "b'94.168.17.2.in-addr.arpa.'"
Ether / IP / UDP / DNS Ans "b'a2-17-168-94.deploy.static.akamaitechnologies.com.'"
Ether / IP / ICMP 10.0.2.15 > 2.17.168.94 echo-request 0 / Raw
Ether / IP / ICMP 2.17.168.94 > 10.0.2.15 echo-reply 0 / Raw
Ether / IP / UDP / DNS Qry "b'94.168.17.2.in-addr.arpa.'"
Ether / IP / UDP / DNS Ans "b'a2-17-168-94.deploy.static.akamaitechnologies.com.'"
Ether / ARP who has 10.0.2.2 says 10.0.2.15
Ether / ARP is at 52:55:0a:00:02:02 says 10.0.2.2 / Padding
Ether / IPv6 / ICMPv6ND_RA / ICMPv6NDOptSrcLLAddr / ICMPv6 Neighbor Discovery Option - Prefix Information fd17:625c:f037:2::/64 On-link 1 Autonomous Address 1 Router Address 0
Ether / fe80::a00:27ff:fe4a:f36e > ff02::16 (0) / IPv6ExtHdrHopByHop / ICMPv6MLReport2
Ether / fe80::a00:27ff:fe4a:f36e > ff02::16 (0) / IPv6ExtHdrHopByHop / ICMPv6MLReport2
>>> 
```

Step 2: Capture and save traffic on a specific interface.

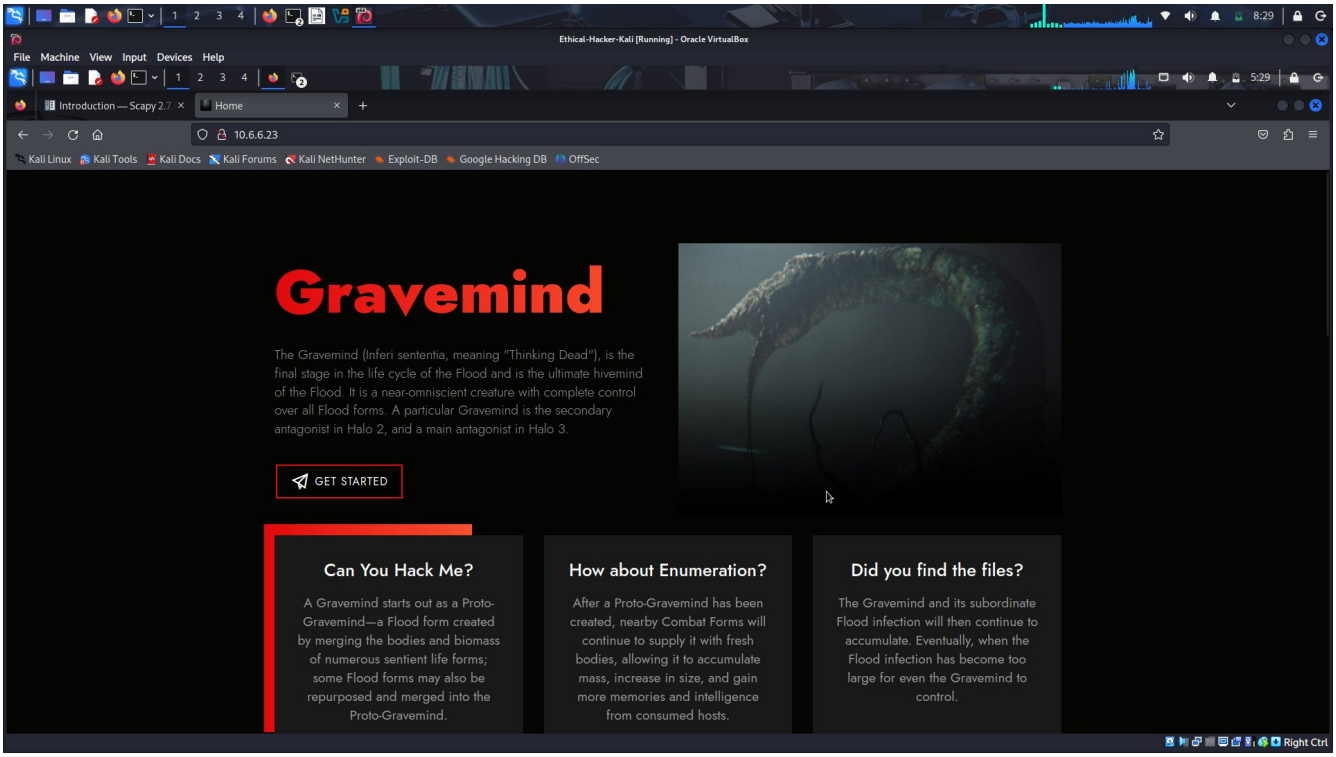
In this step, you will capture traffic to and from a device connected to a virtual network in your Kali Linux VM.

a. Open a new terminal window. Use the **ifconfig** command to determine the name of the interface that is assigned the IP address **10.6.6.1**. This is the default gateway address for one of the virtual networks running inside Kali. Note the name of interface.



b. Return to the terminal window that is running the Scapy tool. Use the syntax **sniff(iface="***interface name***")** to begin the capture on the **br-internal** virtual interface.

c. Open Firefox and navigate to the URL **http://10.6.6.23/**.



When the Gravemind home page opens, return to the terminal window that is running the Scapy tool. Press **CTRL-C**. You should receive output similar to:

```
>>> sniff(iface="br-internal")
^C<Sniffed: TCP:114 UDP:0 ICMP:0 Other:4>
>>> 
```

View the captured traffic as you did in Step 1d.

```
>>> a=_
>>> a.summary()
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http S
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 SA
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http S
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 SA
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 A / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 A / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35492 PA / Raw
Ether / IP / TCP 10.6.6.1:35492 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http A
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
Ether / IP / TCP 10.6.6.1:35508 > 10.6.6.23:http PA / Raw
Ether / IP / TCP 10.6.6.23:http > 10.6.6.1:35508 PA / Raw
```

Step 3: Examine the collected packets.

In this step, you will filter the collected traffic to include only ICMP traffic, limit the number of packets being collected, and view the individual packet details.

a. Use interface ID associated with 10.6.6.1 (br-internal) to capture ten ICMP packets sent and received on the internal virtual network. The syntax is **sniff(iface="***interface name***", filter = "***protocol***", count =** *integer***)**.

```
>>> sniff(iface="br-internal", filter="icmp", count=10)
... : 
```

Open a second terminal window and ping the host at 10.6.6.23.

```
┌──(kali㉿Kali)-[~]
└─$ ping -c 10 10.6.6.23
PING 10.6.6.23 (10.6.6.23) 56(84) bytes of data.
64 bytes from 10.6.6.23: icmp_seq=1 ttl=64 time=0.050 ms
64 bytes from 10.6.6.23: icmp_seq=2 ttl=64 time=0.040 ms
64 bytes from 10.6.6.23: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.6.6.23: icmp_seq=4 ttl=64 time=0.051 ms
64 bytes from 10.6.6.23: icmp_seq=5 ttl=64 time=0.038 ms
64 bytes from 10.6.6.23: icmp_seq=6 ttl=64 time=0.205 ms
64 bytes from 10.6.6.23: icmp_seq=7 ttl=64 time=0.038 ms
64 bytes from 10.6.6.23: icmp_seq=8 ttl=64 time=0.036 ms
64 bytes from 10.6.6.23: icmp_seq=9 ttl=64 time=0.049 ms
64 bytes from 10.6.6.23: icmp_seq=10 ttl=64 time=0.034 ms

--- 10.6.6.23 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9217ms
rtt min/avg/max/mdev = 0.034/0.062/0.205/0.049 ms

┌──(kali㉿Kali)-[~]
└─$ 
```

Return to the terminal window running the Scapy tool. The capture automatically stopped when 10 packets were sent or received. View the captured traffic with line numbers using the **nsummary()** function.

```
>>> sniff(iface="br-internal", filter="icmp", count=10)
... :
<Sniffed: TCP:0 UDP:0 ICMP:10 Other:0>
>>>
```

The summary should only contain 10 lines because the capture count is equal to 10.

```
>>> a=_
>>> a.summary()
Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
>>>
```

To view details about a specific packet in the series, refer to the blue line number of the packet. Do not include the leading zeros. COMMAND: **a[2]** The detail output shows the layers of information about the protocol data units (PDUs) that make up the packet. The protocol layer names appear in red in the output. Examine the source (src) and destination (dst) addresses as well as the raw data (load=) portion of the collected packet. Why are there two sets of source and destination fields? **The first set are the hexadecimal data link layer MAC addresses of source and Ethernet adapters. The second set are the network layer IP source and destination addresses of the packet.**

```
>>> a[2]
<Ether  dst=02:42:0a:06:06:17 src=02:42:bc:9d:f0:e5 type=IPv4 |<IP  version=4 ihl=5 tos=0x0 len=84 id=17924 flags=DF frag=0 ttl=64 proto=icmp chksum=0xd481 src=10.6.6.1 dst=10.6.6.23 |<ICM
5d4d seq=0x2 unused='' |<Raw  load='\\xe8\\xcfLi\x00\x00\x00\x00\\xf4\x03\t\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567' |>
>>>
```

Use the **wrpcap()** function to save the captured data to a pcap file that can be opened by Wireshark and other applications. The syntax is **wrpcap("***filename.pcap*", *variable name***),** in this example the variable that you stored the output is "**a**".

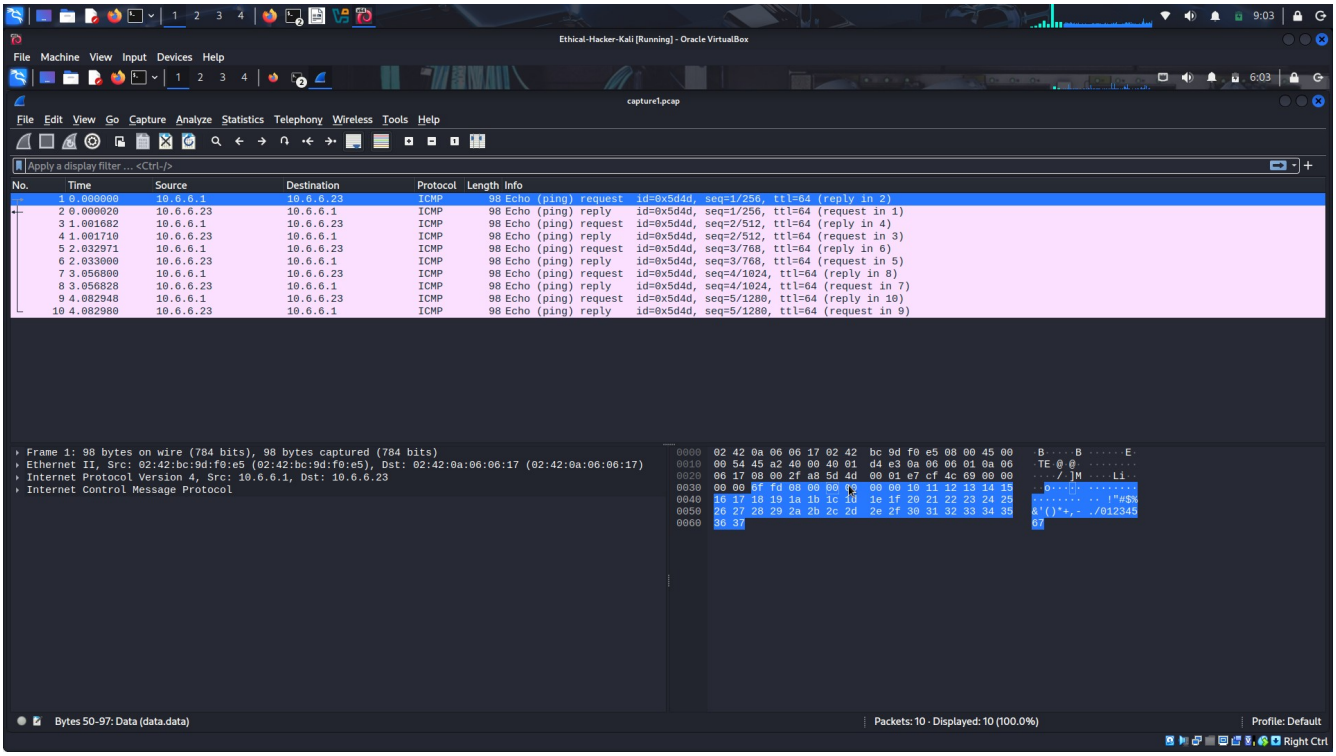
```
>>> wrpcap("capture1.pcap", a)
... :
>>>
```

The .pcap file will be written to the default user directory. Use a different terminal window to verify the location of the **capture1.pcap** file using the Linux **ls** command.

```
┌──(kali㉿Kali)-[~]
└─$ ls
Desktop  Documents  Downloads  Music  OTHER  Pictures  Public  Templates  Videos  capture1.pcap  sfa_cert.html

┌──(kali㉿Kali)-[~]
└─$
```

Open the capture in Wireshark to view the file contents. Command: **wireshark capture1.pcap**



Part 3: Create and Send an ICMP Packet.

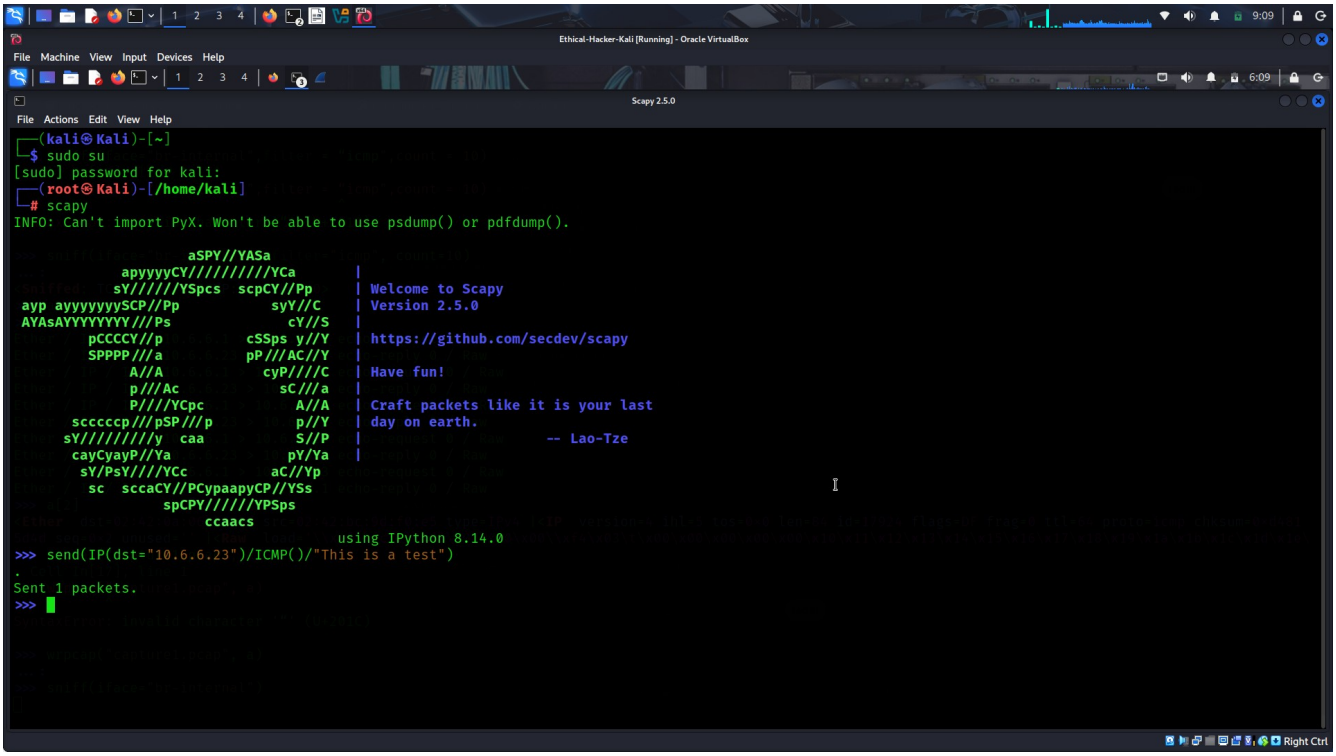
ICMP is a protocol designed to send control messages between network devices for various purposes. There are many types of ICMP packets, with echo-request and echo-reply the most familiar to IT technicians. To see a list of the message types that can be sent and received using ICMP, navigate to https://www.iana.org/assignments/icmp-parameters/icmp-parameters.xhtml.

Step 1: Use interactive mode to create and send a custom ICMP packet.

a. In a Scapy terminal window, enter the command to sniff traffic from the interface connected to the 10.6.6.0/24 network.

```
>>> sniff(iface="br-internal")
```

b. Open another terminal window, enter **sudo su** to perform packet crafting as root. Start a second instance of Scapy. Enter the **send**() function to send a packet to 10.6.6.23 with a modified ICMP payload.



Return to the first terminal window and press **CTRL-C**. You should receive a response similar to this:

```
>>> sniff(iface="br-internal")
^C<Sniffed: TCP:0 UDP:0 ICMP:2 Other:4>
>>>
```

Enter the summary command to display the summary with packet numbers. What type of packets are shown in the summary output? **Echo-request, Echo-reply, ARP, ARP reply**

```
>>> a=_
>>> a.summary()
Ether / ARP who has 10.6.6.23 says 10.6.6.1
Ether / ARP is at 02:42:0a:06:06:17 says 10.6.6.23
Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
Ether / ARP who has 10.6.6.1 says 10.6.6.23
Ether / ARP is at 02:42:bc:9d:f0:e5 says 10.6.6.1
>>> a.nsummary()
0000 Ether / ARP who has 10.6.6.23 says 10.6.6.1
0001 Ether / ARP is at 02:42:0a:06:06:17 says 10.6.6.23
0002 Ether / IP / ICMP 10.6.6.1 > 10.6.6.23 echo-request 0 / Raw
0003 Ether / IP / ICMP 10.6.6.23 > 10.6.6.1 echo-reply 0 / Raw
0004 Ether / ARP who has 10.6.6.1 says 10.6.6.23
0005 Ether / ARP is at 02:42:bc:9d:f0:e5 says 10.6.6.1
>>>
```

Step 2: View and compare the ICMP packet contents.

Use the packet numbers to view the individual ICMP Echo-request and Echo-reply packets. Compare those packets to the ones that you examined in Part 2, Step 3d.

```
>>> a[5]
<Ether  dst=02:42:0a:06:06:17 src=02:42:bc:9d:f0:e5 type=ARP |<ARP  hwtype=Ethern
et (10Mb) ptype=IPv4 hwlen=6 plen=4 op=is-at hwsrc=02:42:bc:9d:f0:e5 psrc=10.6.6.
1 hwdst=02:42:0a:06:06:17 pdst=10.6.6.23 |>>
>>>
```

What is different between the original ICMP packet conversation and the custom ICMP packet conversation? **The raw data portion has changed to "This is a test"**

Part 4: Create and Send a TCP SYN Packet.

In this part, you will use Scapy to determine if port 445, a Microsoft Windows drive share port, is open on the target system at 10.6.6.23.

Step 1: Start the packet capture on the internal interface.

a. In the original Scapy terminal window, begin a packet capture on the internal interface attached to the 10.6.6.0/24 network. Use the interface name that you obtained previously.

```
>>> sniff(iface="br-internal")
```

b. Navigate to the second terminal window. Create and send a TCP SYN packet using the command shown.

```
>>> send(IP(dst="10.6.6.23")/TCP(dport=445, flags="S"))
.
Sent 1 packets.
>>>
```

This command sent an IP packet to the host with IP address 10.6.6.23. The packet is addressed to TCP port 445 and has the S (SYN) flag set.

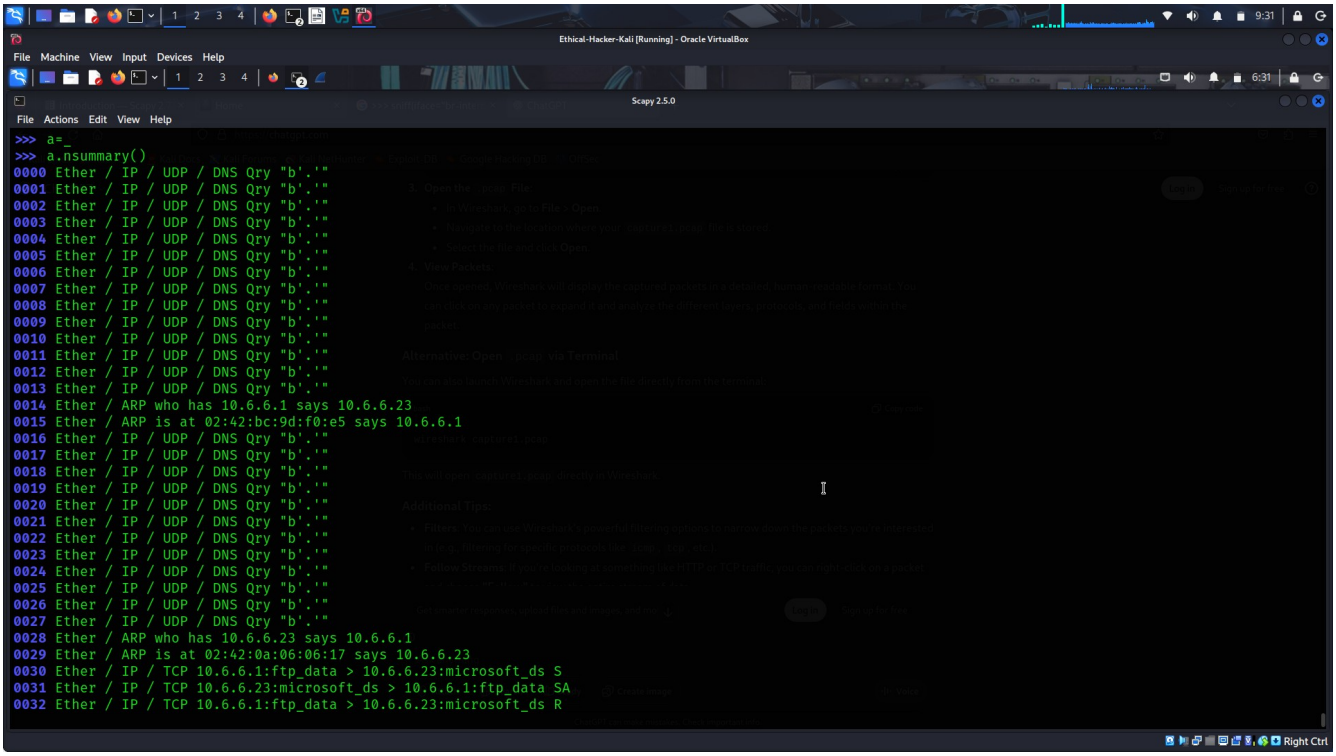
Close the terminal window.

Step 2: Review the captured packets.

a. In the original Scapy terminal window, stop the packet capture by pressing **CTRL-C**. The output should be similar to that shown.

```
>>> sniff(iface="br-internal")
^C<Sniffed: TCP:3 UDP:26 ICMP:0 Other:6>
>>>
```

View the captured TCP packets using the **nsummary()** function.



Display the detail of the TCP packet that was returned from the target computer at 10.6.6.23.

What does the SA flag indicate in the packet returned from 10.6.6.23? **The SA flag stands for SYN-ACK. It means that the port 445 is open on the target computer, because it acknowledged the SYN packet.**

How can crafting various TCP SYN packets be used to perform passive reconnaissance on a target host.? **sending SYN packets and receiving a SYN-ACK in response indicates that the service is operational, and the port is in listening mode. Crafting packets for different TCP ports will indicate which ports are active.**

How could creating an ICMP echo-request packet with a spoofed source address create a denial of service attack on against a target host? **Sending thousands of packets to different hosts with same spoofed source address will cause all of the echo-reply packets to be sent to the target host. This will result in a distributed denial of service attack.**