

Sniffing and Spoofing

Dr. Ruba Al Omari

EECS 4482 – Network Security and Forensics

Reference

- The lecture notes in this course are based on the textbook: Internet Security: A Hands-on Approach (3rd ed., 2022), by Wenliang Du, ISBN: 978-17330039-6-4.
- Unless otherwise specified, all the figures, tables, code examples, and content are from the above-listed textbook.

Topics

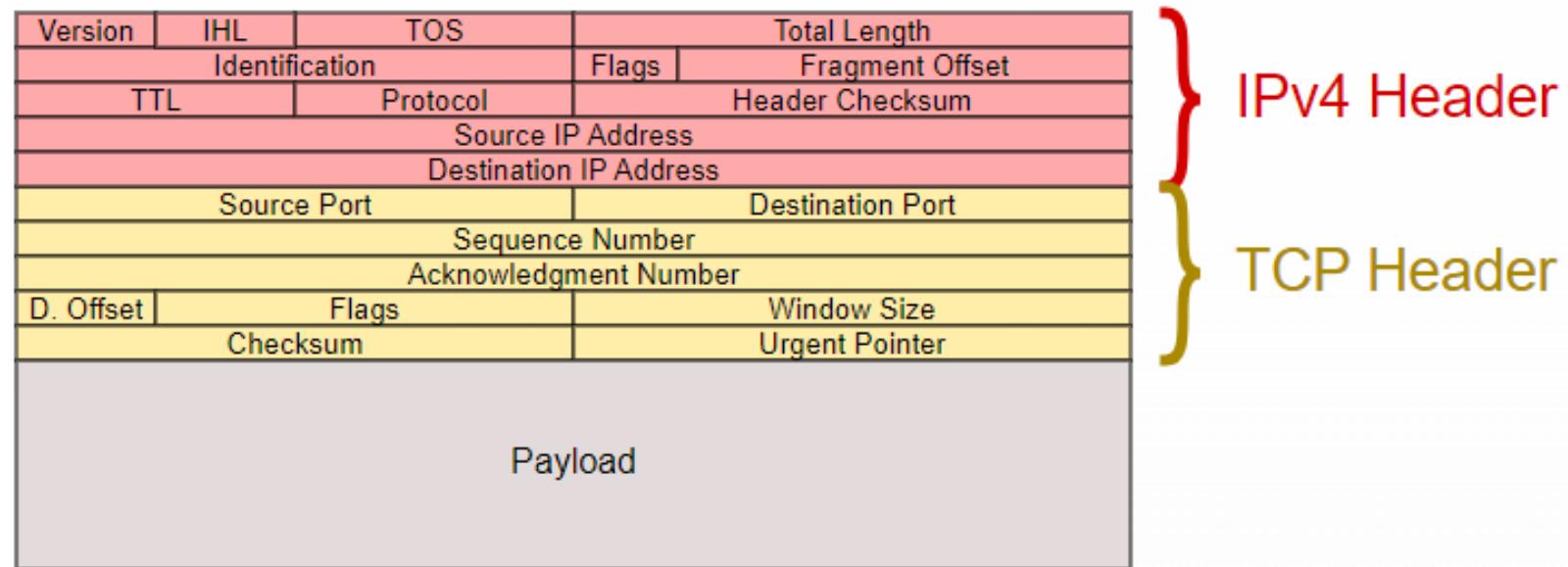
- Ports and Sockets
- Sending/Receiving packets
- Sniffing and Spoofing Packets
- Tools of the Trade

Topics

- Ports and Sockets
- Sending/Receiving packets
- Sniffing and Spoofing Packets
- Tools of the Trade

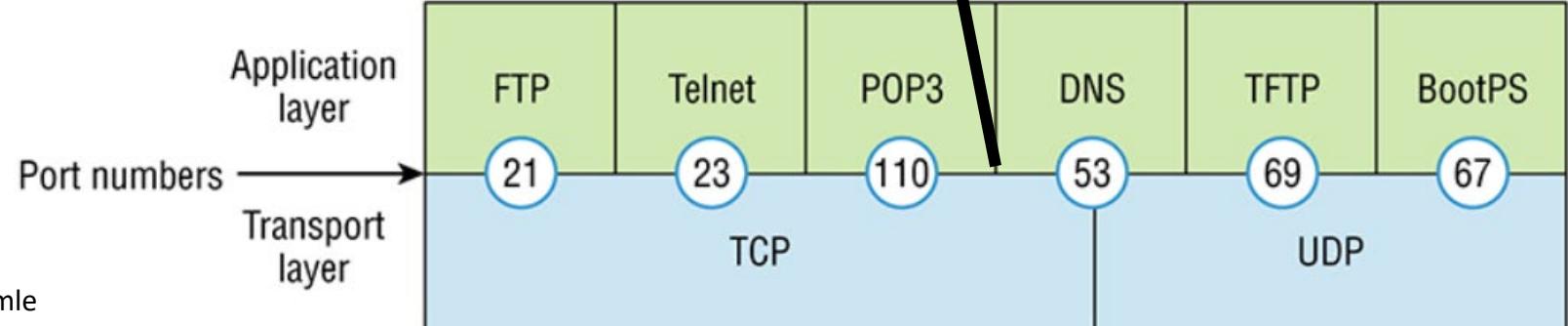
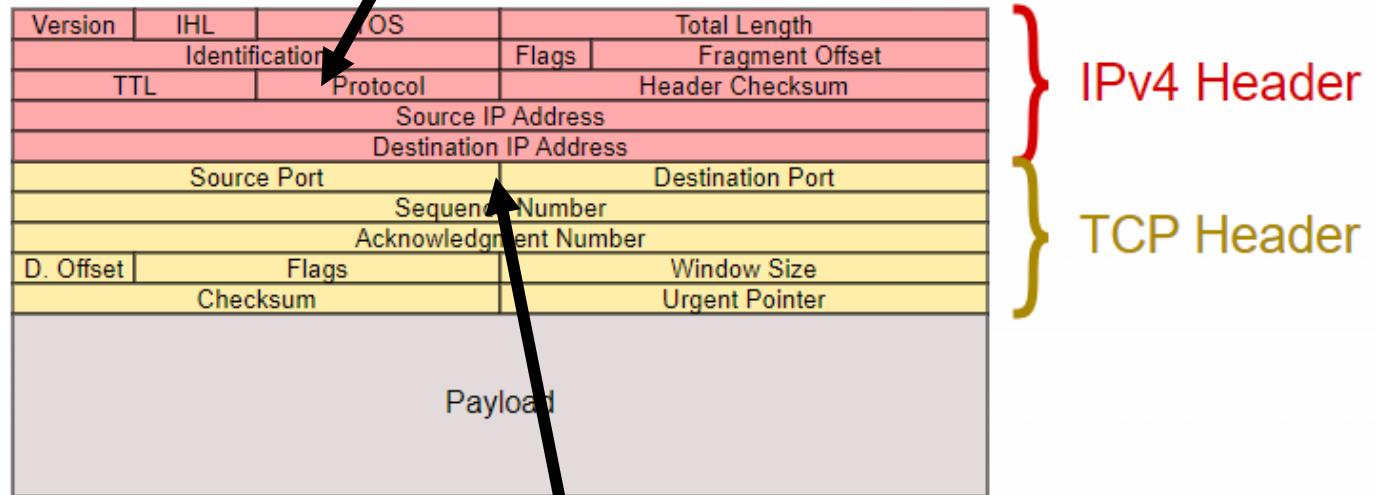
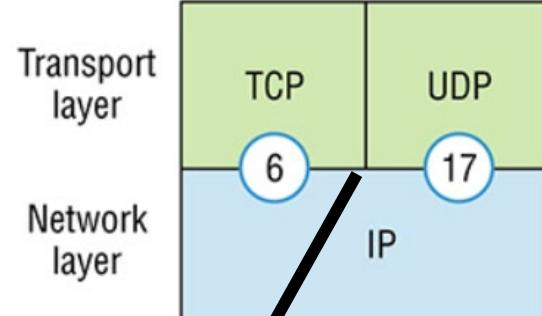
Ports and Sockets

- When a connection is established via the **Transport** layer, it is done using **ports**.
- Port number size is 16-bits $\Rightarrow 2^{16} = 65,536$ available port numbers.
 - Numbered from 0 through 65,535.
- Ports allow a **single IP address** to be able to support **multiple simultaneous communications**, each using a different port number.



Ports Types

- Ports are divided into:
 - Well-Known Ports.
 - Registered Ports.
 - Dynamic/Private Ports.



Well-Known Ports

- Range **0 – 1023**: Assigned by IANA to widely used and well-known applications: ftp (20, 21), ssh (22), telnet (23), smtp (25), DNS (53), http (80), https (443).

Well-Known Ports (0-1023)

- Port 20: FTP Data Transfer (TCP)
- Port 21: FTP Control (Command) (TCP)
- Port 22: SSH (Secure Shell) (TCP)
- Port 23: Telnet (TCP)
- Port 25: SMTP (Simple Mail Transfer Protocol) (TCP)
- Port 53: DNS (Domain Name System) (TCP/UDP)
- Port 67: DHCP Server (Dynamic Host Configuration Protocol) (UDP)
- Port 68: DHCP Client (Dynamic Host Configuration Protocol) (UDP)
- Port 69: TFTP (Trivial File Transfer Protocol) (UDP)
- Port 80: HTTP (HyperText Transfer Protocol) (TCP)
- Port 110: POP3 (Post Office Protocol v3) (TCP)
- Port 119: NNTP (Network News Transfer Protocol) (TCP)
- Port 123: NTP (Network Time Protocol) (UDP)
- Port 135: Microsoft RPC (TCP/UDP)
- Port 137: NetBIOS Name Service (UDP)
- Port 138: NetBIOS Datagram Service (UDP)

- Port 139: NetBIOS Session Service (TCP)
- Port 143: IMAP (Internet Message Access Protocol) (TCP)
- Port 161: SNMP (Simple Network Management Protocol) (UDP)
- Port 162: SNMP Trap (UDP)
- Port 179: BGP (Border Gateway Protocol) (TCP)
- Port 194: IRC (Internet Relay Chat) (TCP)
- Port 389: LDAP (Lightweight Directory Access Protocol) (TCP/UDP)
- Port 443: HTTPS (HTTP Secure) (TCP)
- Port 445: Microsoft-DS (Microsoft Directory Services) (TCP)
- Port 465: SMTPS (Simple Mail Transfer Protocol Secure) (TCP)
- Port 514: Syslog (UDP)
- Port 520: RIP (Routing Information Protocol) (UDP)
- Port 587: SMTP (Mail Submission) (TCP)
- Port 631: IPP (Internet Printing Protocol) (TCP/UDP)
- Port 993: IMAPS (Internet Message Access Protocol over SSL) (TCP)
- Port 995: POP3S (Post Office Protocol 3 over SSL) (TCP)

Registered Ports

- Range **1024 – 49151**: Can be registered with IANA for use by specific applications and services but are less known: OpenVPN (1194), Microsoft SQL server (1433), Docker (2375-2377).

Registered Ports (1024-49151)

- Port 1433: Microsoft SQL Server (TCP)
- Port 1434: Microsoft SQL Monitor (UDP)
- Port 1521: Oracle Database (TCP)
- Port 1723: PPTP (Point-to-Point Tunneling Protocol) (TCP)
- Port 2049: NFS (Network File System) (TCP/UDP)
- Port 2082: cPanel default (TCP)
- Port 2083: cPanel secure (TCP)
- Port 2483: Oracle Database (TCP)
- Port 2484: Oracle Database (TCP)
- Port 3306: MySQL (TCP)

- Port 3389: RDP (Remote Desktop Protocol) (TCP)
- Port 3690: Subversion (TCP)
- Port 4444: Metasploit Framework (TCP)
- Port 5432: PostgreSQL (TCP)
- Port 5900: VNC (Virtual Network Computing) (TCP)
- Port 5984: CouchDB (TCP)
- Port 6379: Redis (TCP)
- Port 6667: IRC (Internet Relay Chat) (TCP)
- Port 8000: Common alternative HTTP (TCP)
- Port 8080: HTTP Proxy (TCP)
- Port 8443: HTTPS (HTTP Secure) alternative (TCP)

Dynamic/Private Ports

- Range **49152 – 65535**: Open for use without restriction.

Ephemeral ports.
Temporarily &
Random

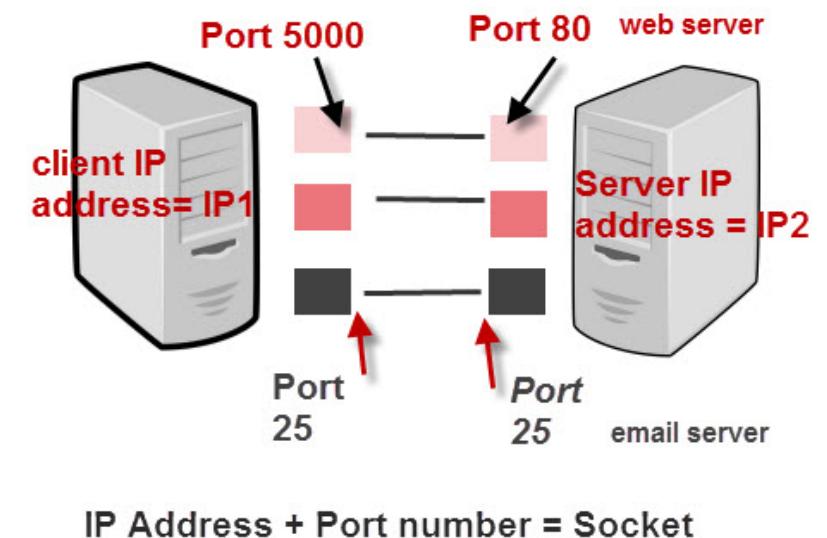
Subset of
dynamic
ports

Aspect	Dynamic Ports	Private Ports
Range	49152 - 65535	49152 - 65535
Assignment	Assigned dynamically by OS for outbound connections.	Can be used by any application, often manually assigned.
Usage	Primarily for temporary client-side use.	Can be used by both clients and servers for custom services.
Nature	Temporary, reused after session ends.	Can be permanent or temporary, flexible use.

e.g., an admin assigning a port 50000 to a webserver instead of 80.

Sockets

- The combination of an IP address and a port number (e.g., 10.1.1.1:8080) is known as a **socket**.
- A network socket is an endpoint for a data communications channel.
- At any given time, a typical workstation has **dozens** of open sockets, each representing an existing data communications channel.
 - Servers can have **thousands** or even tens of thousands of them.



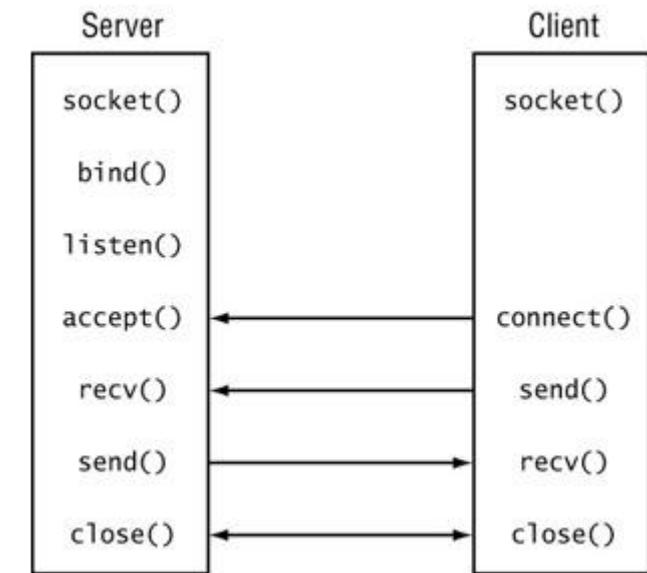
Types of Sockets

- A socket is a layer 4 (transport) construct:
 - **Stream Sockets (TCP):** Use TCP to provide a reliable, connection-oriented communication channel.
 - Recall: TCP ensures that data is delivered in order, without duplicates, and with error checking.
 - **Datagram Sockets (UDP):** Use UDP to provide a connectionless communication channel.
 - Recall: UDP is faster but does not guarantee delivery, order, or error checking.

TCP/IP Socket Lifecycle

■ Server Side:

- 1. Socket Creation:** The server creates a socket using the **socket()** function.
- 2. Binding:** The server binds the socket to a specific IP address and port number using the **bind()** function.
- 3. Listening:** The server listens for incoming connections using the **listen()** function.
- 4. Accepting Connections:** The server accepts incoming connections using the **accept()** function, creating a new socket for each connection.

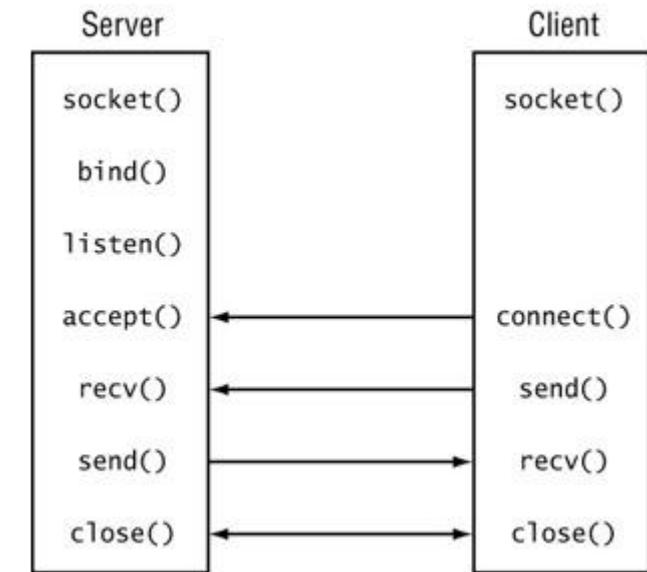


Server-client communication using TCP/IP

TCP/IP Socket Lifecycle

■ Client Side:

- 1. Socket Creation:** The client creates a socket using the **socket()** function.
- 2. Connecting:** The client connects to the server using the **connect()** function, specifying the server's IP address and port number.



Server-client communication using TCP/IP

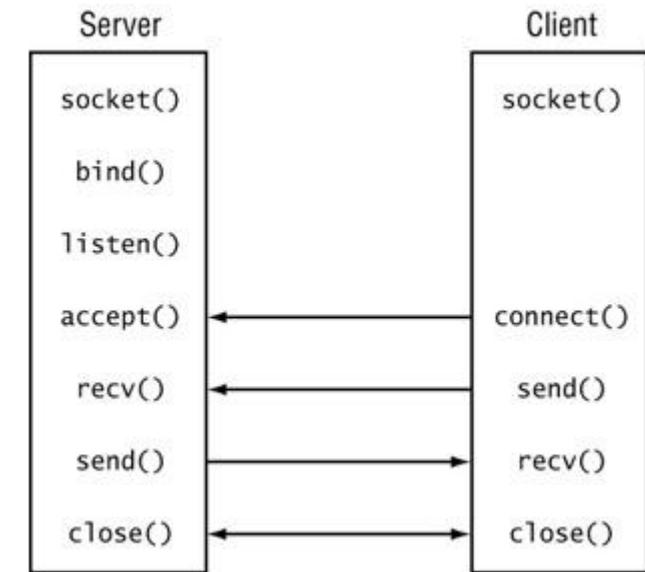
TCP/IP Socket Lifecycle

■ Data Transmission:

- **Sending Data:** Both the server and the client can send data using the **send()** function.
- **Receiving Data:** Both the server and the client can receive data using the **recv()** function.

■ Termination:

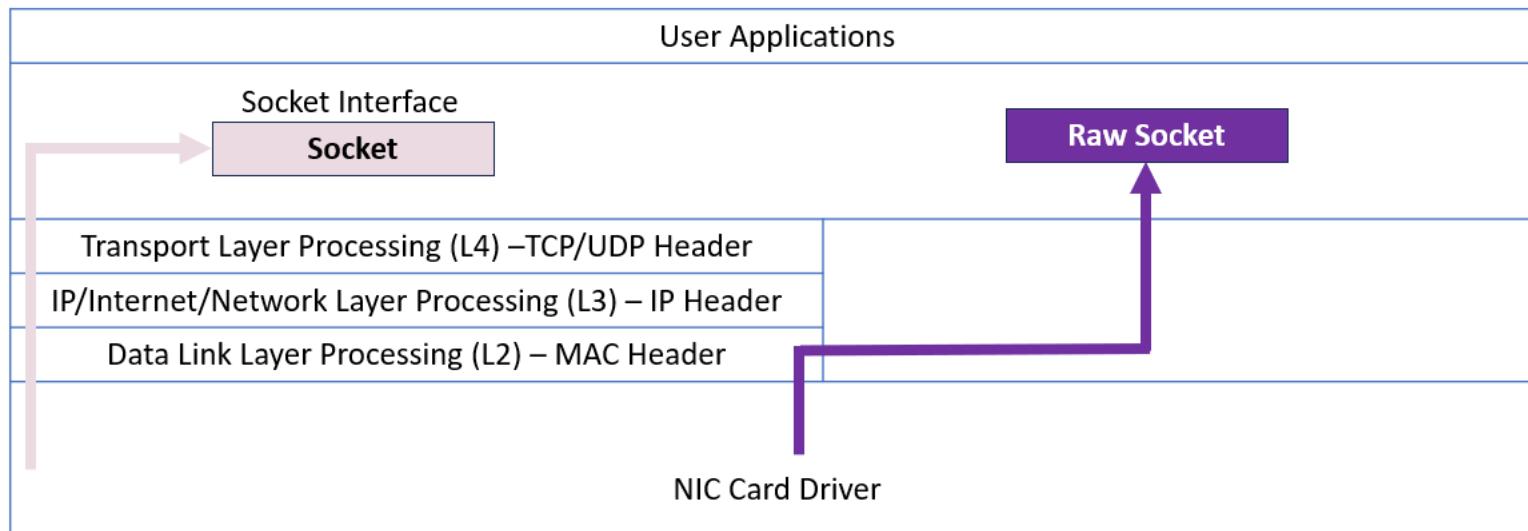
- **Closing the Socket:** Once the communication is complete, both the server and the client close their sockets using the **close()** function.



Server-client communication using TCP/IP

Raw Sockets

- TCP/IP **raw sockets** are used to receive raw packets → packets received at the Ethernet layer will directly pass to the raw socket.
- A raw socket bypasses the normal TCP/IP processing and sends the packets to the specific user application.



Topics

- Ports and Sockets
- **Sending/Receiving packets**
- Sniffing and Spoofing Packets
- Tools of the Trade

Sending Packets

- Packet sending tools:

- Use netcat:

```
$ nc <ip> <port> #send out TCP packet  
$ nc -u <ip> <port> #send out UDP packet
```

- Send to bash's pseudo device /dev/tcp and /dev/udp:

```
$ echo "data" > /dev/tcp/<ip>/<port> #send out TCP packet to <ip>:<port>  
$ echo "data" > /dev/udp/<ip>/<port> #send out UDP packet to <ip>:<port>
```

- Other tools examples:

```
$ ping <ip> #send out ICMP packet  
$ telnet <ip> #send out TCP packet
```

Sending Packets - UDP Client Example

The image shows a Kali Linux desktop environment with several windows open:

- Terminal Window (Left):** Shows the command `nc -l -u -p 9090` being run, followed by the message "Hello, World".
- Code Editor Window (Top Right):** Shows a Python script named `udp_client.py` containing the following code:

```
1 #!/usr/bin/python3
2
3 import socket
4 IP = "127.0.0.1"
5 PORT = 9090
6 data = b'Hello, World'
7
8 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
9 sock.sendto(data, (IP, PORT))
10
```
- Terminal Window (Bottom Left):** Shows the command `sudo python udp_client.py` being run.
- Status Bar (Bottom):** Displays the quote "the quieter you become, the more you are able to hear".

udp_client.py

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

```
import socket
IP = "127.0.0.1"
PORT = 9090
data = b'Hello, World'

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(data, (IP, PORT))
```

.send: for TCP.

.socket(): Creates a new socket object.

.AF_INET: Specifies the address family for the socket, which in this case is IPv4.

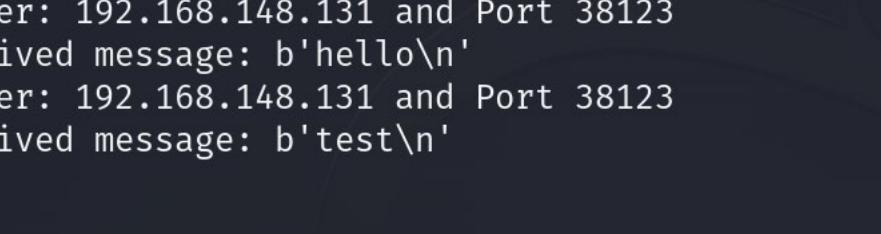
- **.AF_INET6**

.SOCK_DGRAM: Specifies that the socket type is UDP (Datagram).

- **.SOCK_STREAM** (TCP).

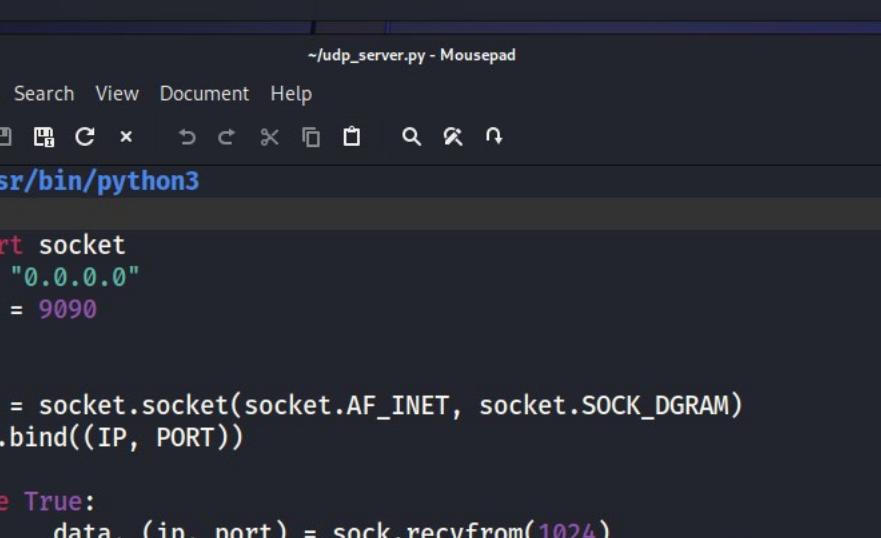
Receiving Packets - UDP Server Example

```
seed@VM: ~
alomari-seed@VM:192.168.148.131:~$ nc -u 192.168.148.128 9090
hello
test
```



```
kali㉿kali: ~
File Actions Edit View Help
Sender: 192.168.148.131 and Port 38123
Received message: b'hello\n'
Sender: 192.168.148.131 and Port 38123
Received message: b'test\n'

[ ]
```



```
~/udp_server.py - Mousepad
File Edit Search View Document Help
File New Open Save Close Find Replace Undo Redo Cut Copy Paste Select All Delete Select All
1 #!/usr/bin/python3
2
3 import socket
4 IP = "0.0.0.0"
5 PORT = 9090
6
7
8 sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
9 sock.bind((IP, PORT))
10
11 while True:
12     data, (ip, port) = sock.recvfrom(1024)
13     print("Sender: {} and Port {}".format(ip, port))
14     print("Received message: {}".format(data))
15
```

udp_server.py

.recvfrom(1024): receive data from udp socket, 1024 is buffer size.
▪ **.accept**: for TCP.

```
#!/usr/bin/python3

import socket
IP = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP, PORT))

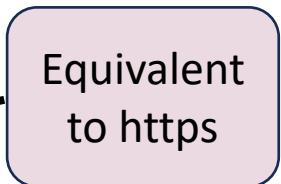
while True:
    data, (ip, port) = sock.recvfrom(1024)
    print("Sender: {} and Port {}".format(ip, port))
    print("Received message: {}".format(data))
```

.bind((IP, PORT)): Bind the socket to an address and port.
▪ **.bind('::', 8080)** binds to all available IPv6 interfaces.

TCP/IP – Multilayer Protocol

- Example: When communicating between a web server and a web browser, HTTP is encapsulated in TCP, which in turn is encapsulated in IP, which in turn is encapsulated in Ethernet:

- [Ethernet [IP [TCP [HTTP [Payload]]]]]]



- ## ■ Add TLS encryption:

- [Ethernet [IP [TCP [TLS [HTTP [Payload]]]]]]]

- Encapsulate further with Network layer encryption:

- [Ethernet [IPsec [IP [TCP [TLS [HTTP [Payload]]]]]]]]



TCP/IP – Multilayer Protocol

- Not always benign! Examples:
- With **HTTPTunnel** tool, the standard payload is replaced with an alternative protocol.
 - [Ethernet [IP [TCP [HTTP [FTP [Payload]]]]]]]
- With **ptunnel**, ICMP is transformed into a tunnel protocol to support TCP communications.
 - [Ethernet [IP [ICMP [TCP [HTTP [Payload]]]]]]]]

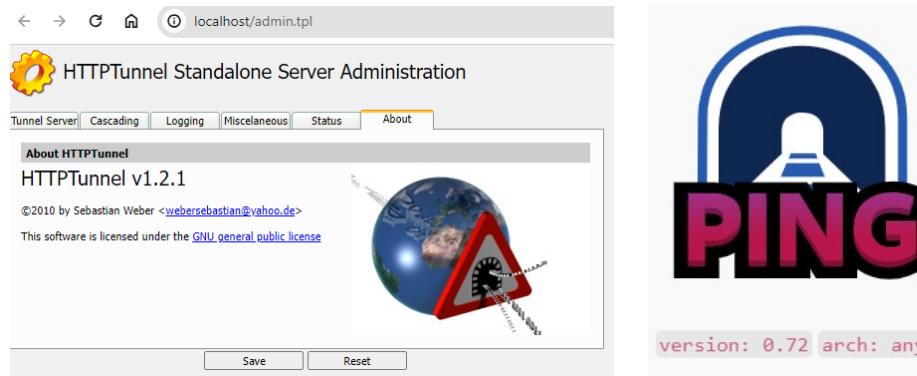


Image Source: <http://phrack.org/issues/49/6.html#article>
<https://sourceforge.net/projects/http-tunnel/>

```
kali@kali: ~
File Actions Edit View Help
romari@kali:[~]ptunnel -h
ptunnel v 0.72.
Usage: ptunnel -p <addr> -lp <port> -da <dest_addr> -dp <dest_port> [-m max_tunnels] [-v verbosity] [-f logfile]
      ptunnel [-m max_threads] [-v verbosity] [-c <device>]
-p: Set address of peer running packet forwarder. This causes
    ptunnel to operate in forwarding mode - the absence of this
    option causes ptunnel to operate in proxy mode.
-lp: Set TCP listening port (only used when operating in forward mode)
-da: Set remote proxy destination address if client
    Restrict to only this destination address if server
-dp: Set remote proxy destination port if client
    Restrict to only this destination port if server
-m: Set maximum number of concurrent tunnels
-v: Verbosity level (-1 to 4, where -1 is no output, and 4 is all output)
-c: Enable libpcap on the given device.
-f: Specify a file to log to, rather than printing to standard out.
-s: Client only. Enables continuous output of statistics (packet loss, etc.)
-daemon: Run in background, the PID will be written in the file supplied as argument
-syslog: Output debug to syslog instead of standard out.
```

Topics

- Ports and Sockets
- Sending/Receiving packets
- Sniffing and Spoofing Packets
- Tools of the Trade

Sniffing and Spoofing Attacks

- Two of the common attacks on networks are sniffing and spoofing attacks.
- The basis for many attacks on the Internet, such as the DNS cache poisoning and TCP session hijacking attacks.

Sniffing & Spoofing Attacks

- Sniffing Attacks: Attackers can eavesdrop on a physical network, wired or wireless, and capture the packets transmitted over the network.
- Spoofing Attacks: When some critical information in the packet is forged.

Spoofing Packets Using Raw Sockets

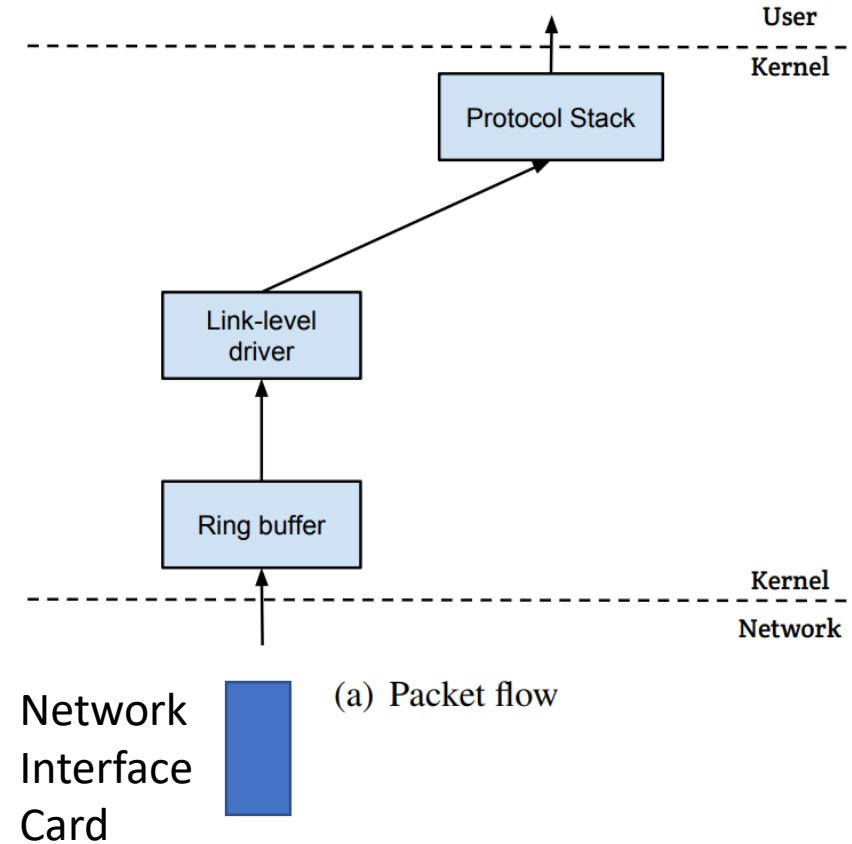
- Two main steps in packet spoofing:
 - Constructing the packet.
 - Sending the packet out.

Sniffing and Then Spoofing

- In many situations, we need to capture packets first, and then spoof a response based on the captured packets.
- Procedure (using UDP as an example):
 - Use an API to capture the packets of interest.
 - Make a copy from the captured packet.
 - Replace the UDP data field with a new message and swap the source and destination fields.
 - Send out the spoofed reply.

How Packets Are Received

- Each NIC (Network Interface Card) has a MAC address.
- Every NIC on the network will hear all the frames on the wire.
- NIC checks the destination address for every packet, if the address matches the card's MAC address, it is further copied into a **buffer** in the kernel.
- The frames that are not destined to a given NIC are **discarded** → impossible to sniff network traffic.
- **Question:** How do we get a copy of the received packets for our sniffer program?



Promiscuous Mode

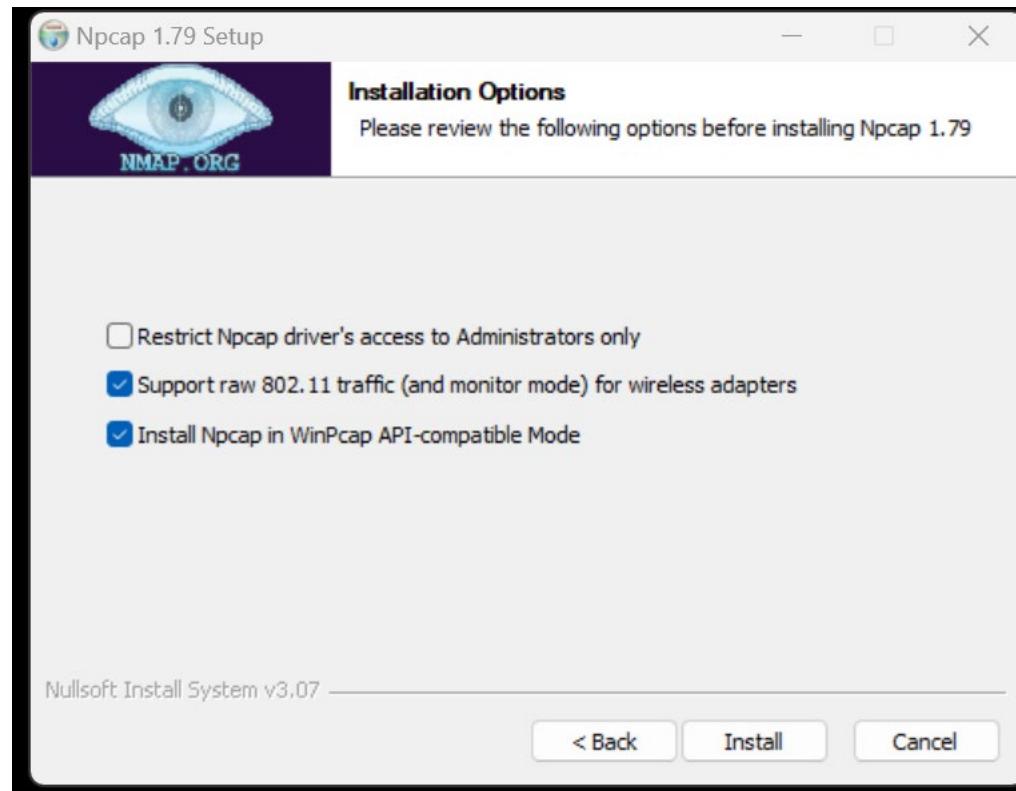
- Allows the capture of all network packets passing through a network interface, regardless of whether they are intended for the capturing machine or not.

```
romari@kali-[~]sudo ifconfig eth0 promisc
romari@kali-[~]ifconfig eth0
eth0: flags=4419<UP,BROADCAST,RUNNING,PROMISC,MULTICAST
      mtu 1500
      inet 192.168.148.128 netmask 255.255.255.0
            broadcast 192.168.148.255
            inet6 fe80::6d29:aa3f:ac41:35b prefixlen 64
            scopeid 0x20<link>
            ether 00:0c:29:98:fb:db txqueuelen 1000 (Ethernet)
                  RX packets 10013 bytes 3172013 (3.0 MiB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 8127 bytes 1014001 (990.2 KiB)
                  TX errors 0 dropped 0 overruns 0 carrier 0
                  collisions 0

romari@kali-[~]
```

Promiscuous Mode

- In Wi-Fi, it is called Monitor Mode.



Promiscuous Mode

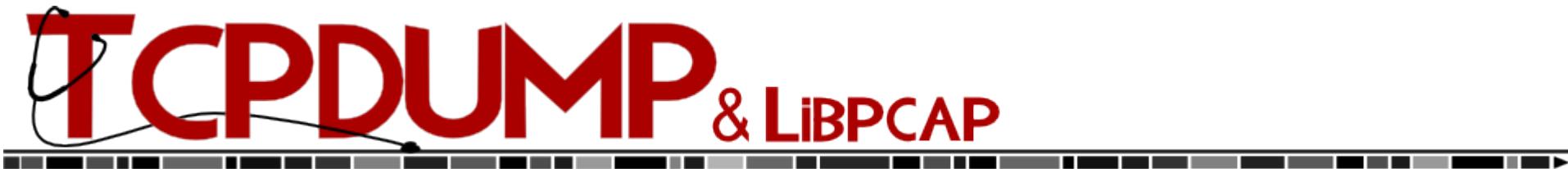
- When operating in **promiscuous mode**, a NIC passes **every frame** received from the network to the kernel.
- If a sniffer program is registered with the kernel, it will be able to see all the packets and make a copy for analysis.
- **Question:** Does each program need to write its own API to sniff packets?

PCAP: Packet Capture API

- PCAP is a fundamental tool in network analysis (originally coming from tcpdump), providing a standardized API for capturing and analyzing network packets:
 - Linux: libpcap
 - Windows: WinPcap, Npcap
- Written in C:
 - Other languages implement wrappers.
- Basis for many tools: Wireshark, tcpdump, Scapy, McAfee, Nmap, Snort,...

PCAP: Packet Capture API

- **1987:** tcpdump was created, leveraging the initial concepts of packet capturing.
- **1994:** The first official release of **libpcap**, providing a portable and standardized API for packet capture.
- **Late 1990s to Early 2000s:** Development of **WinPcap** for Windows, expanding the reach of pcap libraries to non-UNIX systems.
- **2006:** Wireshark (originally named Ethereal) became one of the most popular network protocol analyzers, heavily relying on pcap libraries.



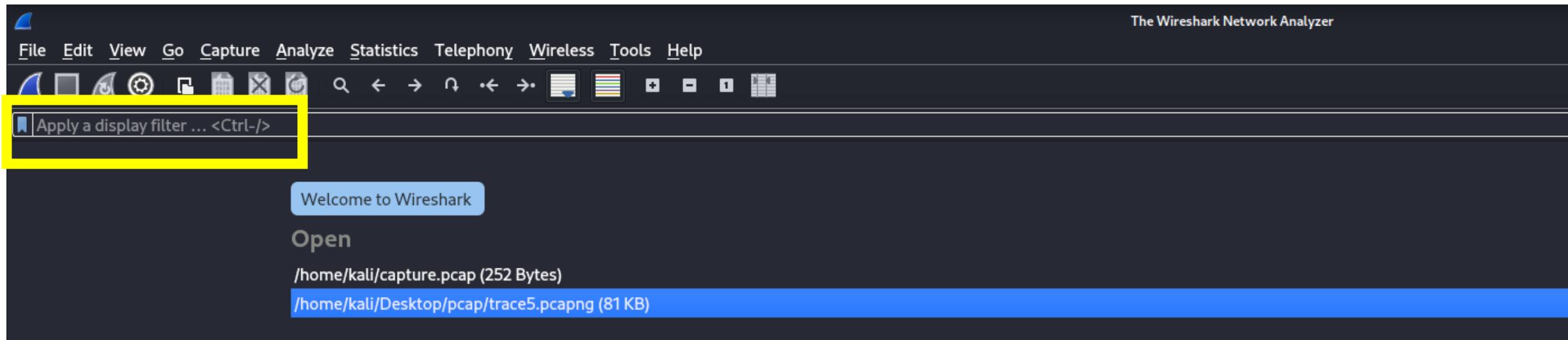
Npcap

- WinPcap was eventually superseded by **Npcap**, a modern packet capture and network monitoring library for Windows, developed by the Nmap Project.
- Npcap offers better performance and support for Windows 7 through 11 and beyond.



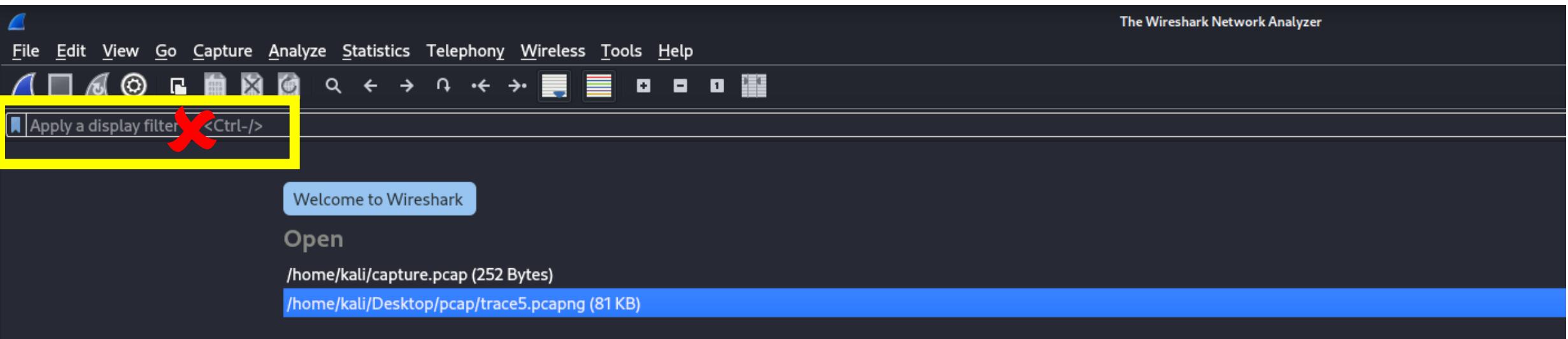
Filtering Out Unwanted Packets

- When sniffing network traffic, it is common that sniffers are only interested in **certain types of packets**, such as TCP packets or DNS query packets.



Filtering Out Unwanted Packets

- Filtering after capturing is inefficient:
 1. **Processing Overhead:** The sniffer program must process every captured packet, even if it's irrelevant → Consumes CPU cycles and memory.
 2. **Delivery Time:** The kernel delivers all packets to the sniffer application → takes time, and delay becomes significant when there are many unwanted packets.



Berkeley Packet Filter (BPF)

- To address these inefficiencies, the Berkeley Packet Filter (**BPF**) was developed.
- BPF allows a user-program to attach a filter to the socket, which tells the kernel to discard unwanted packets.
- The filter is written in a human-readable format using Boolean operators.

Capture traffic to and from IP host 192.168.1.1:

```
ip host 192.168.1.1
```

Capture traffic from IP host 192.168.1.1:

```
ip src host 192.168.1.1
```

Capture traffic to IP host 192.168.1.1:

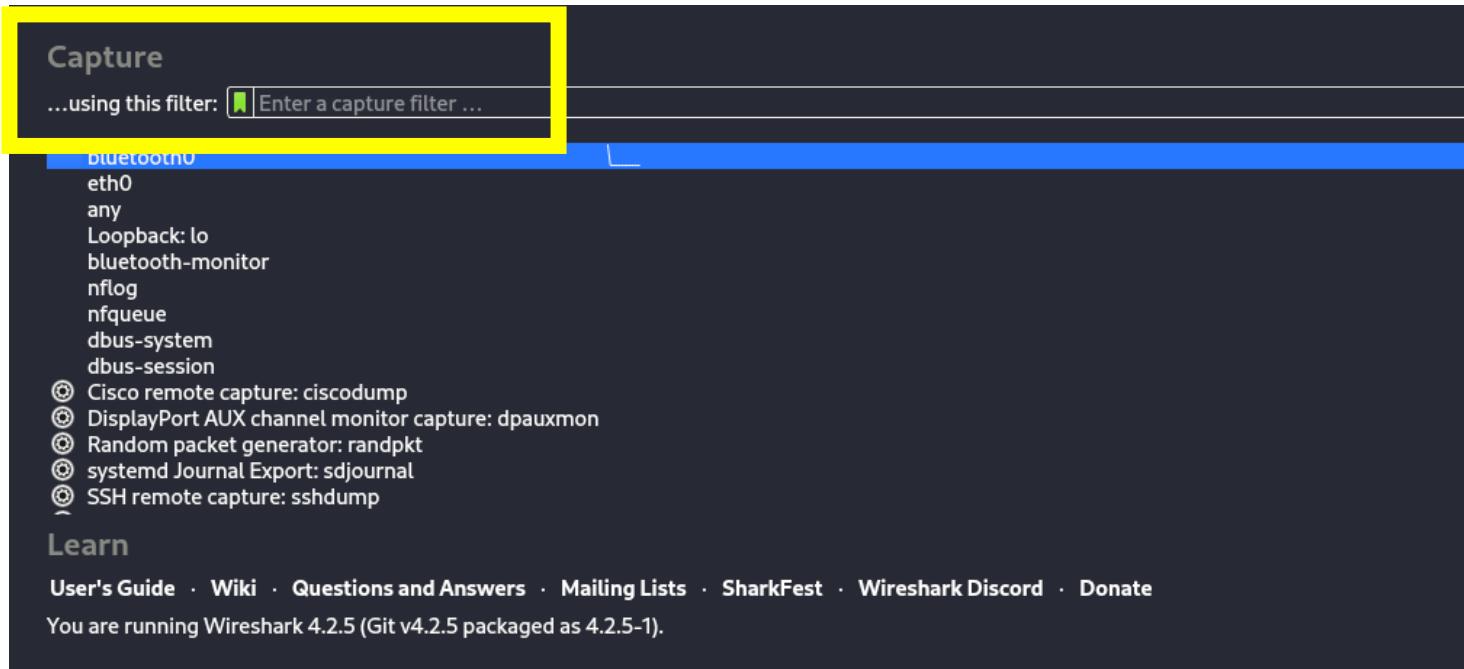
```
ip dst host 192.168.1.1
```

Capture traffic for a particular IP protocol:

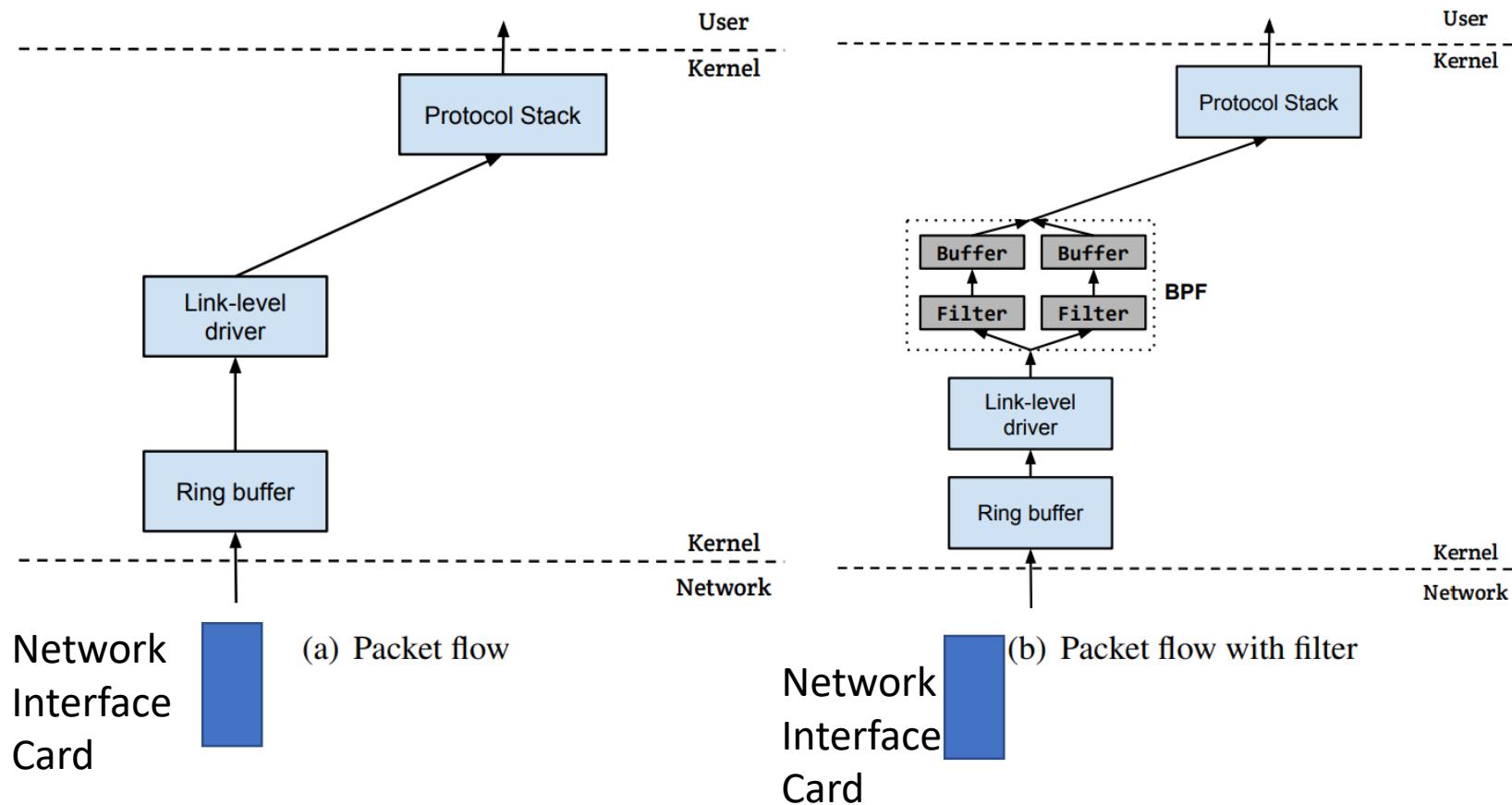
```
ip proto protocol
```

Berkeley Packet Filter (BPF)

- **Filter Attachment:** A user-space program (e.g., wireshark) attaches a BPF filter to a network socket → Specifies which packets to capture and which to discard.
- **Early Filtering:** The kernel applies the filter to incoming packets at the lowest level (before delivering them to the sniffer application).
 - Unwanted packets are discarded immediately, reducing processing overhead.



Packet Flow With/Without Filters



Topics

- Ports and Sockets
- Sending/Receiving packets
- Sniffing and Spoofing Packets
- Tools of the Trade

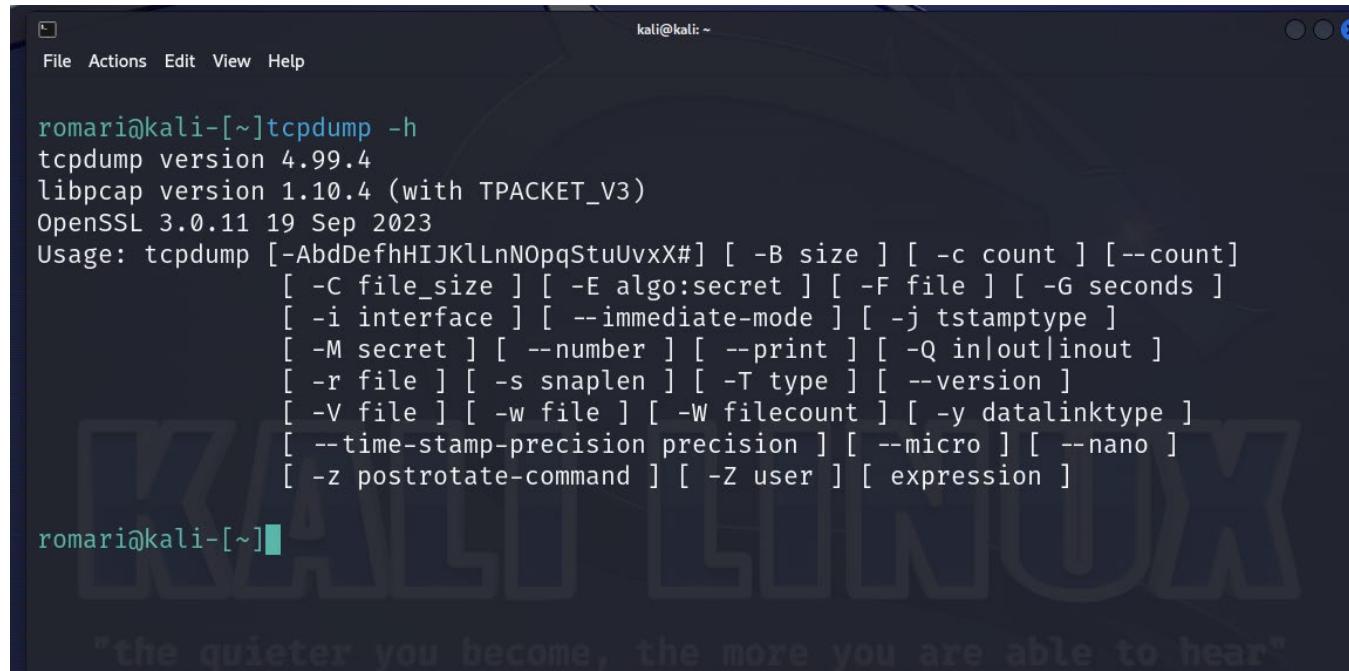
Tools of the Trade

- **Tcpdump**: Quick and efficient for basic packet capture.
- **Scapy**: Python-based, customizable, and scriptable.
- **Wireshark**: Comprehensive GUI tool for detailed analysis.
 - **tshark**: Terminal-based Wireshark.

tcpdump

tcpdump

- Command line based packet sniffer.
 - Capture on an interface, or read **pcap** file.
- Ideal for quick capturing, integrating into scripts, and is lightweight.
- Simple with basic capabilities.



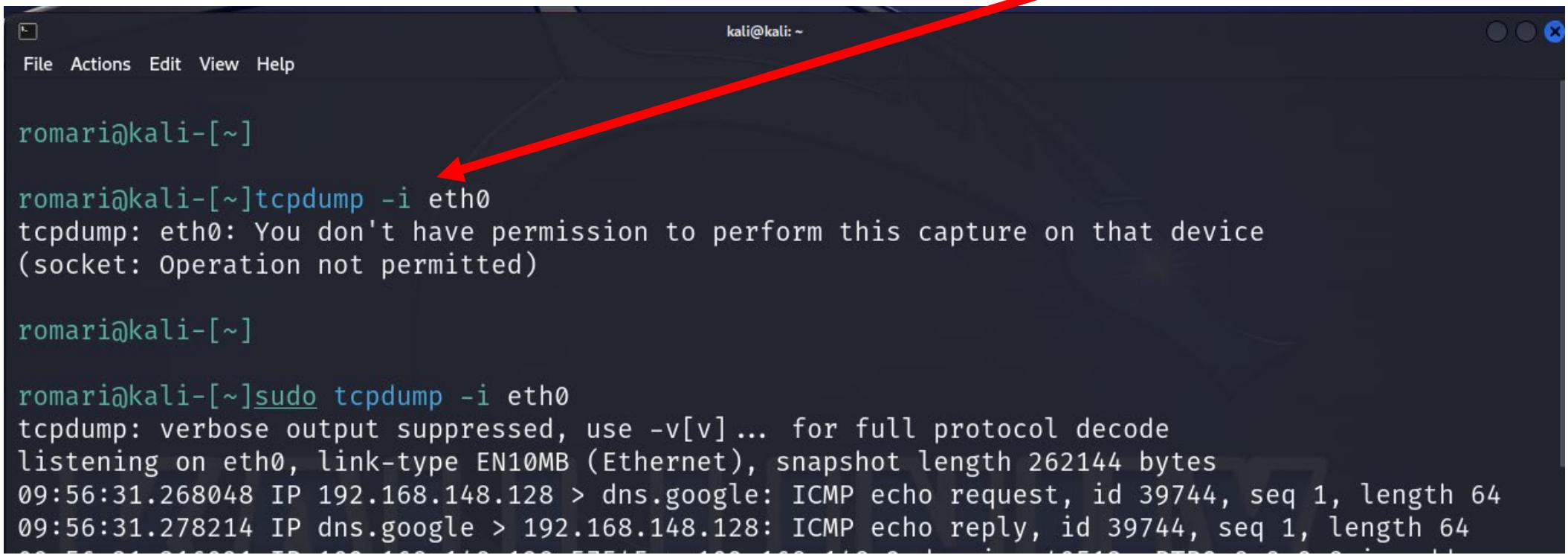
A screenshot of a terminal window titled "kali@kali: ~". The window shows the output of the command "tcpdump -h". The output includes the version information for tcpdump (4.99.4), libpcap (1.10.4), and OpenSSL (3.0.11 19 Sep 2023). It also displays the usage syntax for the "tcpdump" command, which includes various options for capturing network traffic. The terminal has a dark background with light-colored text. In the bottom right corner, there is a watermark for "KALI LINUX" with the tagline "the quieter you become, the more you are able to hear".

```
romari@kali:[~]tcpdump -h
tcpdump version 4.99.4
libpcap version 1.10.4 (with TPACKET_V3)
OpenSSL 3.0.11 19 Sep 2023
Usage: tcpdump [-AbdDefhHIJKLnNOpqStuUvxX#] [ -B size ] [ -c count ] [--count]
               [ -C file_size ] [ -E algo:secret ] [ -F file ] [ -G seconds ]
               [ -i interface ] [ --immediate-mode ] [ -j tstamptype ]
               [ -M secret ] [ --number ] [ --print ] [ -Q in|out|inout ]
               [ -r file ] [ -s snaplen ] [ -T type ] [ --version ]
               [ -V file ] [ -w file ] [ -W filecount ] [ -y datalinktype ]
               [ --time-stamp-precision precision ] [ --micro ] [ --nano ]
               [ -z postrotate-command ] [ -Z user ] [ expression ]
```

tcpdump

- tcpdump -i eth0

listen on
interface eth0



A screenshot of a terminal window titled "kali@kali: ~". The terminal shows the following session:

```
romari@kali:[~]
romari@kali-[~]tcpdump -i eth0
tcpdump: eth0: You don't have permission to perform this capture on that device
(socket: Operation not permitted)

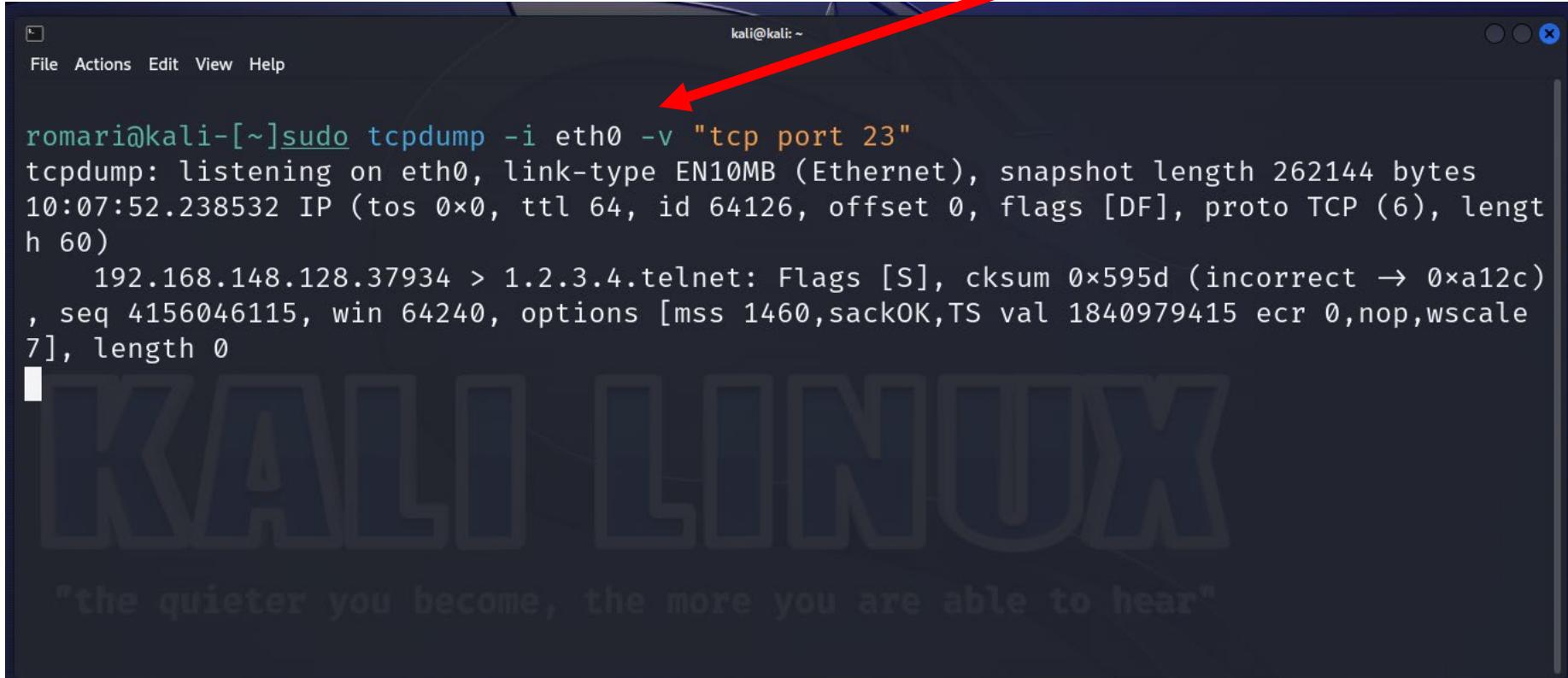
romari@kali:[~]
romari@kali-[~]sudo tcpdump -i eth0
tcpdump: verbose output suppressed, use -v[v] ... for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
09:56:31.268048 IP 192.168.148.128 > dns.google: ICMP echo request, id 39744, seq 1, length 64
09:56:31.278214 IP dns.google > 192.168.148.128: ICMP echo reply, id 39744, seq 1, length 64
```

An arrow points from the text "listen on interface eth0" in the top right corner to the command "tcpdump -i eth0" in the terminal session.

tcpdump

- `tcpdump -i eth0 -v "tcp port 23"`

-v produce more verbose output.

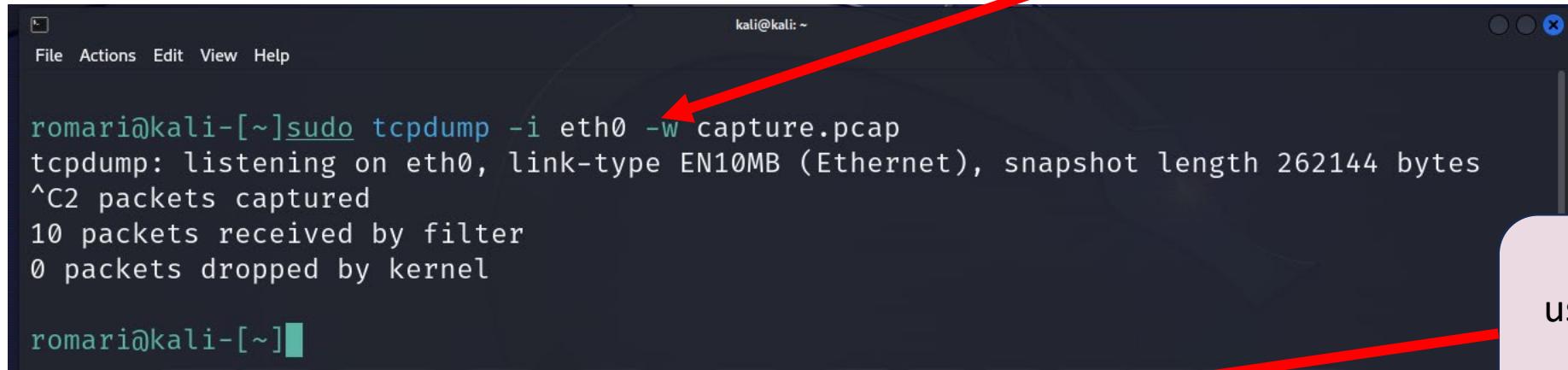


```
romari@kali:[~]sudo tcpdump -i eth0 -v "tcp port 23"
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
10:07:52.238532 IP (tos 0x0, ttl 64, id 64126, offset 0, flags [DF], proto TCP (6), length 60)
    192.168.148.128.37934 > 1.2.3.4.telnet: Flags [S], cksum 0x595d (incorrect → 0xa12c)
        , seq 4156046115, win 64240, options [mss 1460,sackOK,TS val 1840979415 ecr 0,nop,wscale 7], length 0
```

tcpdump

- tcpdump -i eth0 -w capture.pcap

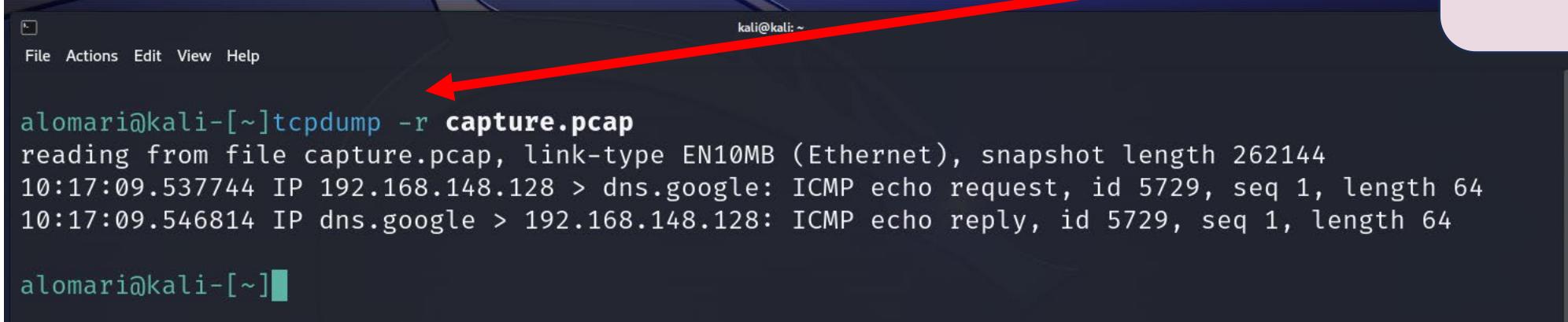
-w capture packets and save them to a pcap file



```
romari@kali:[~]sudo tcpdump -i eth0 -w capture.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C2 packets captured
10 packets received by filter
0 packets dropped by kernel

romari@kali:[~]
```

use -r to read the file.



```
alomari@kali:[~]tcpdump -r capture.pcap
reading from file capture.pcap, link-type EN10MB (Ethernet), snapshot length 262144
10:17:09.537744 IP 192.168.148.128 > dns.google: ICMP echo request, id 5729, seq 1, length 64
10:17:09.546814 IP dns.google > 192.168.148.128: ICMP echo reply, id 5729, seq 1, length 64

alomari@kali:[~]
```

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

capture.pcap

Apply a display filter ... <Ctrl-/>

No.	Time	Source	Destination	Protocol	Length	Stream index	Info
1	0.000...	192.168.1...	8.8.8.8	ICMP	98		Echo (ping) request id=0x1661, seq=1/256, ttl=64 (reply in 2)
2	0.009...	8.8.8.8	192.168.1...	ICMP	98		Echo (ping) reply id=0x1661, seq=1/256, ttl=128 (request in 1)

```

Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
Ethernet II, Src: VMware_98:fb:db (00:0c:29:98:fb:db), Dst: VMware_ff:e5:03 (00:50:56:ff:e5:03)
Internet Protocol Version 4, Src: 192.168.148.128, Dst: 8.8.8.8
Internet Control Message Protocol
0000  00 50 56 ff e5 03 00 0
0010  00 54 b4 11 40 00 40 0
0020  08 08 08 00 20 03 16 6
0030  00 00 3a 34 08 00 00 0
0040  16 17 18 19 1a 1b 1c 1
0050  26 27 28 29 2a 2b 2c 2
0060  36 37

```

Packets: 2 - Displayed: 2 (100.0%)

Profile: Default

Scapy

What is Scapy?

- Scapy is a powerful interactive packet manipulation library written in **Python**, capable of:
 - Packet parsing.
 - Packet sending and receiving.
 - Packet sniffing.
 - Packet Spoofing.
 - Create custom protocols.
- Quick Tutorial
<https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial>



A REPL and a Library

- A REPL and a Library:

- Scapy provides a REPL (**R**ead-**E**val-**P**rint **L**oop) interface for interactive exploration.
- It can also be used as a library to build custom network tools.

- Cross-Platform Support:

- Scapy runs natively on Linux, macOS, most Unix systems, and Windows (with Npcap).

```
git clone --depth 1 https://github.com/secdev/scapy
cd scapy
sudo ./run_scapy

          aSPY//YASa
          apyyyyCY/////////YCa
          sY//////YSpcs  scpCY//Pp | Welcome to Scapy
          ayp ayyyyyyySCP//Pp      syY//C | Version 4ad7977
          AYAsAYYYYYYYY//Ps      cY//S |
          pCCCCY//p           cSSps y//Y | https://github.com/secdev/scapy
          SPPPP//a           pP///AC//Y |
          A//A               cyP///C | Have fun!
          p///Ac            sC///a |
          P///YCpc          A//A | To craft a packet, you have to be a
          scccccP///pSP///p    p//Y | packet, and learn how to swim in
          sY/////////y caa     S//P | the wires and in the waves.
          cayCyayP//Ya       pY/Ya | -- Jean-Claude Van Damme
          sY/PsY///YCc       aC//Yp |
          sc  sccaCY//PCypaapyCP//YSs
                           spCPY//////YPSpS
                           ccaacs
```

Installation

- Comes pre-installed in Kali.
 - And on SEED VM.

- Installation:

```
sudo pip3 install scapy
```

- Import Scapy Module in Python Program:

```
from scapy.all import *
```

```
>>> (Ether()/IP()).show()
###[ Ethernet ]###
dst      = None (resolved on build)
src      = 00:00:00:00:00:00
type     = IPv4
###[ IP ]###
version  = 4
ihl      = None
tos      = 0x0
len      = None
id       = 1
flags    =
frag     = 0
ttl      = 64
proto   = hopopt
chksum  = None
src      = 127.0.0.1
dst      = 127.0.0.1
\options \
>>> 
```

Packet Creation and Manipulation /1

- **Ether()**: Creates an Ethernet layer packet.

- Example: `pkt = Ether()`

- **IP()**: Creates an IP layer packet.

- Example: `pkt = IP(dst="8.8.8.8")`

- **TCP()**: Creates a TCP layer packet.

- Example: `pkt = TCP(dport=80)`

Packet Creation and Manipulation /2

- **UDP():** Creates a UDP layer packet.
 - Example: `pkt = UDP(dport=53)`
- **ICMP():** Creates an ICMP layer packet.
 - Example: `pkt = ICMP()`
- **DNS():** Creates a DNS layer packet.
 - Example: `pkt = DNS()`

Packet Sending /1

- **send()**: Sends packets at layer 3 (IP layer).
 - Example: `send(IP(dst="8.8.8.8")/ICMP())`
- **sendp()**: Sends packets at layer 2 (Ethernet layer).
 - Example: `sendp(Ether()/IP(dst="8.8.8.8")/ICMP())`
- **sr()**: Sends packets and receives answers at layer 3.
 - Example: `ans, unans = sr(IP(dst="8.8.8.8")/ICMP())`

Packet Sending /2

- **srp()**: Sends packets and receives answers at layer 2.
 - Example: `ans, unans = srp(Ether()/ARP(pdst="192.168.1.1/24"))`
- **sr1()**: Sends packets and receives only the first answer.
 - Example: `ans = sr1(IP(dst="8.8.8.8")/ICMP())`
- **srp1()**: Sends packets and receives only the first answer at layer 2.
 - Example: `ans = srp1(Ether()/ARP(pdst="192.168.1.1"))`

Packet Sniffing

- **sniff()**: Captures packets from the network.
 - Example: `pkts = sniff(count=10)`

Packet Display and Analysis

- **show()**: Displays a packet with detailed information.
 - Example: `pkt.show()`
- **hexdump()**: Displays a packet in hexadecimal and ASCII format.
 - Example: `hexdump(pkt)`
- **ls()**: Lists the fields of a layer or protocol.
 - Example: `ls(IP)`
- **lsc()**: Lists all available commands in Scapy.
 - Example: `lsc()`

Network Scanning and Discovery

- **arping()**: Sends ARP requests to determine which hosts are up in a local network.
 - Example: `arping("192.168.1.1/24")`
- **traceroute()**: Performs a traceroute to a destination.
 - Example: `traceroute("8.8.8.8")`
- **traceroute6()**: Performs an IPv6 traceroute to a destination.
 - Example: `traceroute6("google.com")`

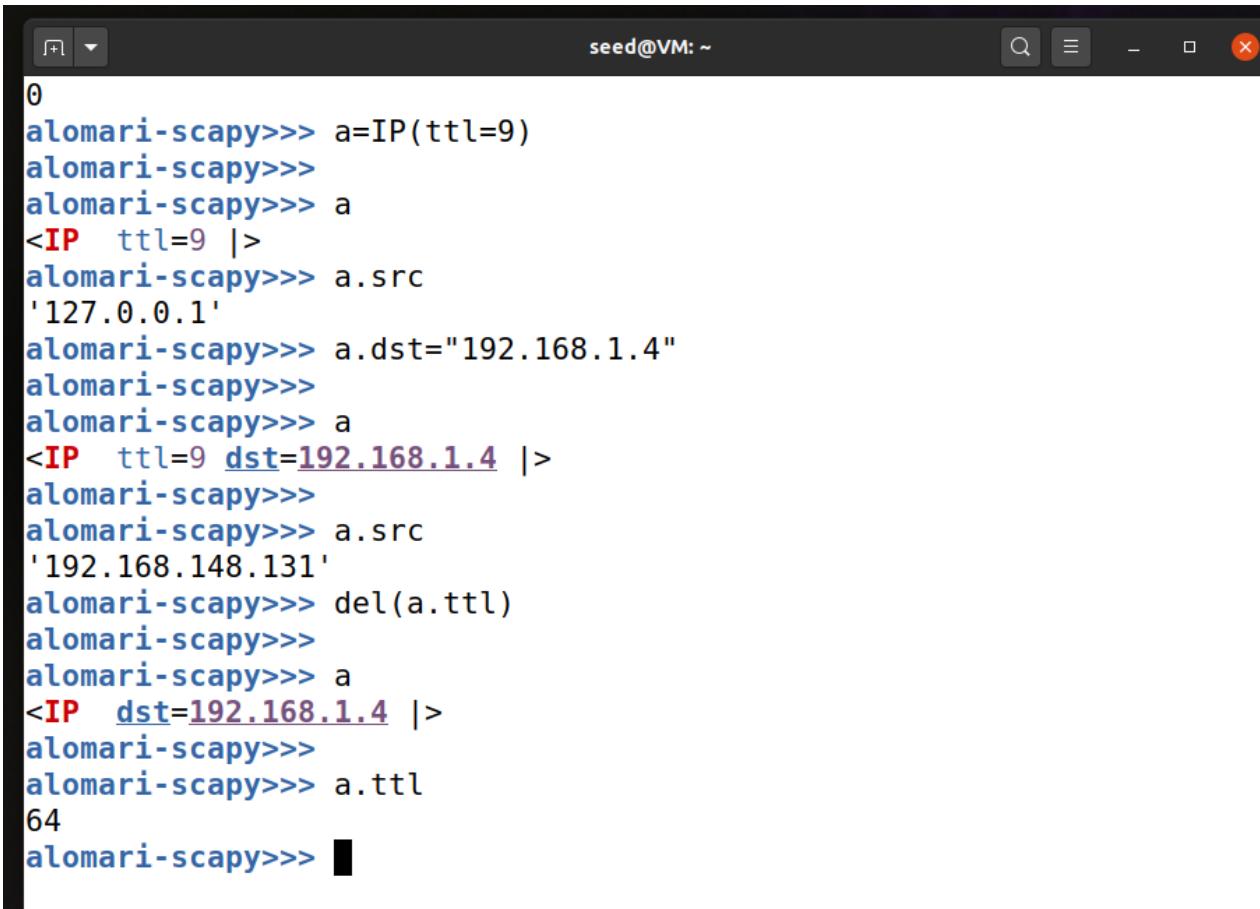
ARP Manipulation

- **ARP():** Creates an ARP packet.
 - Example: `pkt = ARP(pdst="192.168.1.1")`

- **arpcache poison():** Poisons the ARP cache of a target.
 - Example: `arpcache poison("192.168.1.1", "192.168.1.100")`

First Steps

- Let's build a packet and play with it.

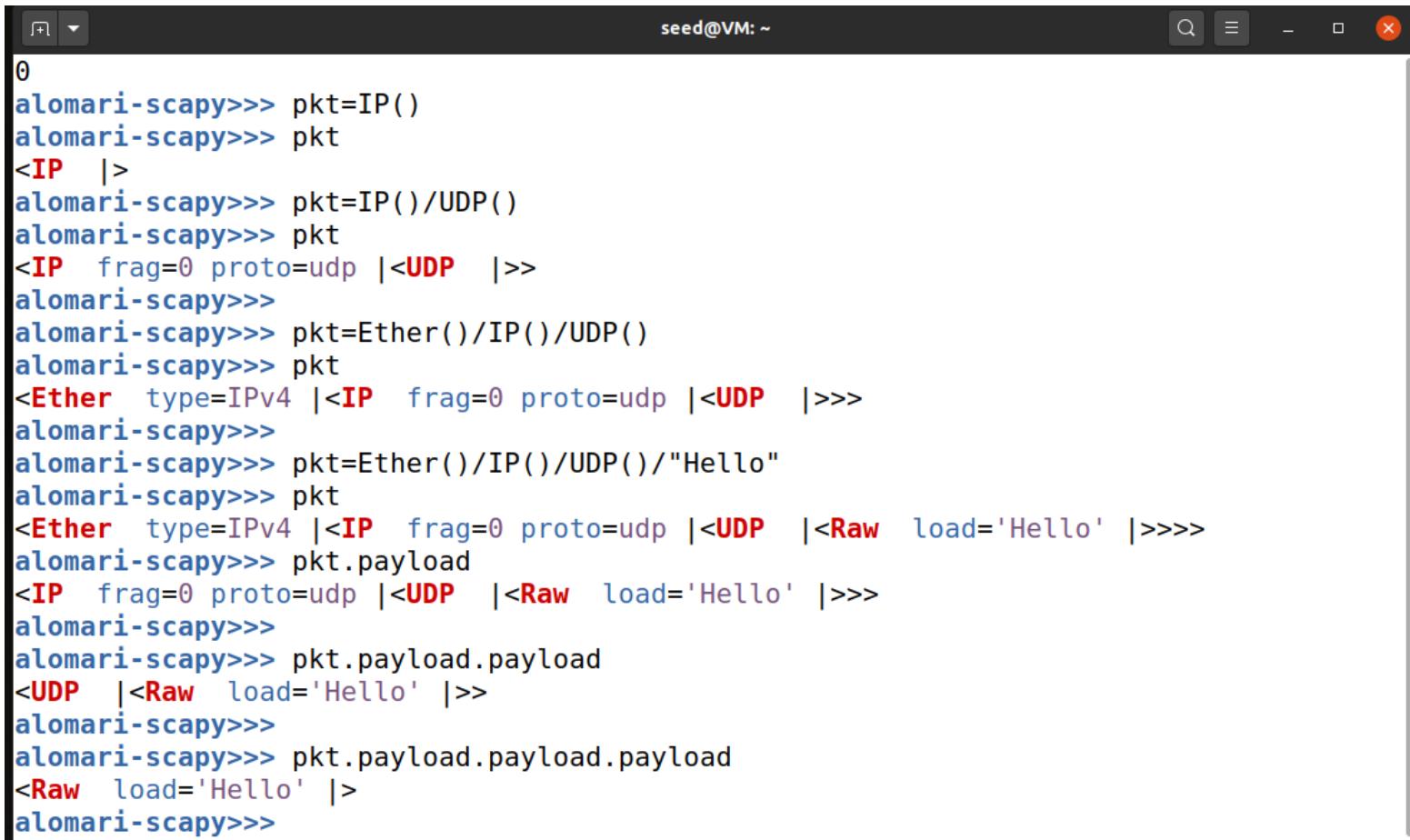


A screenshot of a terminal window titled "seed@VM: ~". The window contains a Scapy interactive session. The user has built an IP packet with source '127.0.0.1' and destination '192.168.1.4'. They then modified the source to '192.168.148.131' and deleted the TTL field. Finally, they set the TTL back to 64.

```
0
alomari-scapy>>> a=IP(ttl=9)
alomari-scapy>>>
alomari-scapy>>> a
<IP ttl=9 |>
alomari-scapy>>> a.src
'127.0.0.1'
alomari-scapy>>> a.dst="192.168.1.4"
alomari-scapy>>>
alomari-scapy>>> a
<IP ttl=9 dst=192.168.1.4 |>
alomari-scapy>>>
alomari-scapy>>> a.src
'192.168.148.131'
alomari-scapy>>> del(a.ttl)
alomari-scapy>>>
alomari-scapy>>> a
<IP dst=192.168.1.4 |>
alomari-scapy>>>
alomari-scapy>>> a.ttl
64
alomari-scapy>>>
```

Stacking Layers

- Packets are objects, and the `/` operator is used to stack packets.



The screenshot shows a terminal window titled "seed@VM: ~" displaying Scapy code. The code demonstrates the creation of a multi-layered packet by stacking various protocol layers using the division operator (`/`). The layers include Ether, IP, UDP, and Raw, with the final payload being the string "Hello".

```
0
alomari-scapy>>> pkt=IP()
alomari-scapy>>> pkt
<IP |>
alomari-scapy>>> pkt=IP()/UDP()
alomari-scapy>>> pkt
<IP frag=0 proto=udp |<UDP |>>
alomari-scapy>>>
alomari-scapy>>> pkt=Ether()/IP()/UDP()
alomari-scapy>>> pkt
<Ether type=IPv4 |<IP frag=0 proto=udp |<UDP |>>>
alomari-scapy>>>
alomari-scapy>>> pkt=Ether()/IP()/UDP()/"Hello"
alomari-scapy>>> pkt
<Ether type=IPv4 |<IP frag=0 proto=udp |<UDP |<Raw load='Hello' |>>>
alomari-scapy>>> pkt.payload
<IP frag=0 proto=udp |<UDP |<Raw load='Hello' |>>>
alomari-scapy>>>
alomari-scapy>>> pkt.payload.payload
<UDP |<Raw load='Hello' |>>
alomari-scapy>>>
alomari-scapy>>> pkt.payload.payload.payload
<Raw load='Hello' |>
alomari-scapy>>>
```

Stacking Layers

- Each packet can be built or dissected.

```
seed@VM: ~
0
alomari-scapy> raw(IP())
b'E\x00\x00\x14\x00\x01\x00|\x00@\x00|\xe7\x7f\x00\x00\x01\x7f\x00\x00\x01'
alomari-scapy>
alomari-scapy> IP(_)
<IP version=4 ihl=5 tos=0x0 len=20 id=1 flags= frag=0 ttl=64 proto=hopopt checksum=0x7ce7
src=127.0.0.1 dst=127.0.0.1 >
alomari-scapy>
alomari-scapy> pkt=Ether()/IP(dst="www.google.ca")/TCP()/"GET /index.html HTTP/1.0 \n\n"
alomari-scapy>
alomari-scapy> hexdump(pkt)
0000  00 50 56 FF E5 03 00 0C 29 07 CF 1E 08 00 45 00  .PV.....)....E.
0010  00 43 00 01 00 00 40 06 6D 4A C0 A8 94 83 8E FB  .C....@ mJ.....
0020  29 43 00 14 00 50 00 00 00 00 00 00 00 50 02  )C...P.....P.
0030  20 00 B0 47 00 00 47 45 54 20 2F 69 6E 64 65 78  ..G..GET /index
0040  2E 68 74 6D 6C 20 48 54 54 50 2F 31 2E 30 20 0A  .html HTTP/1.0 .
0050  0A
alomari-scapy> b=raw(pkt)
alomari-scapy> b
b'\x00PV\xff\xe5\x03\x00\x0c)\x07\xcf\x1e\x08\x00E\x00\x00C\x00\x01\x00\x00@\x06mJ\xc0\x
8\x94\x83\x8e\xfb)C\x00\x14\x00P\x00\x00\x00\x00\x00\x00P\x02 \x00\xb0G\x00\x00GE
T /index.html HTTP/1.0 \n\n'
alomari-scapy> c=Ether(b)
alomari-scapy> c
<Ether dst=00:50:56:ff:e5:03 src=00:0c:29:07:cf:1e type=IPv4 |<IP version=4 ihl=5 tos=0
x0 len=67 id=1 flags= frag=0 ttl=64 proto=tcp checksum=0x6d4a src=192.168.148.131 dst=142.2
51.41.67 |<TCP sport=ftp_data dport=http seq=0 ack=0 dataofs=5 reserved=0 flags=S window
=8192 checksum=0xb047 urgptr=0 |<Raw load='GET /index.html HTTP/1.0 \n\n' |>>>
alomari-scapy>
```

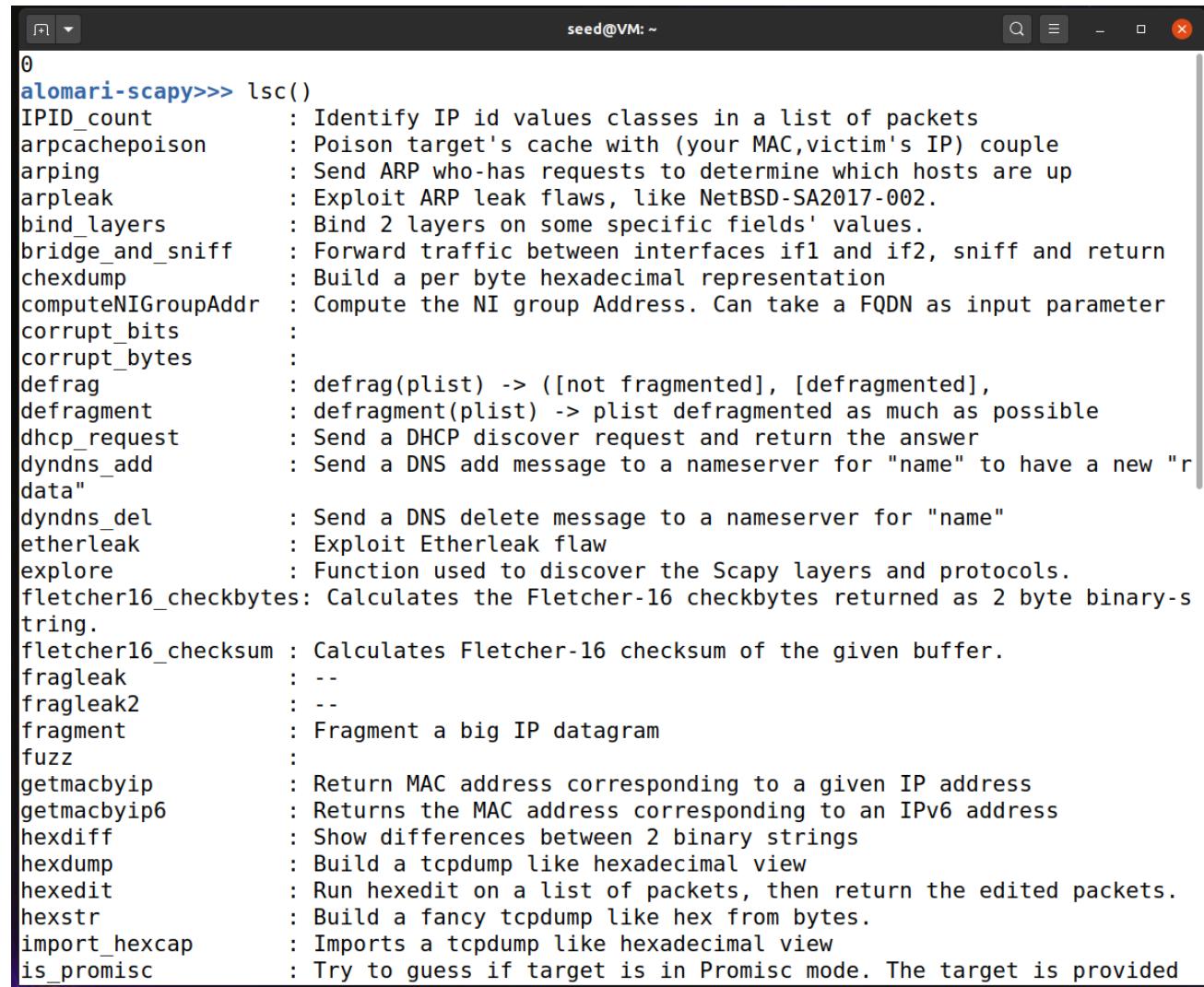
in python `_` is
the latest result

rdpcap() & hide_defaults()

```
seed@VM: ~
0
alomari-scapy>>> a=rdpcap("capture.pcap")
alomari-scapy>>>
alomari-scapy>>> a
<capture.pcap: TCP:0 UDP:0 ICMP:2 Other:0>
alomari-scapy>>>
```

```
seed@VM: ~
alomari-scapy>>> c
<Ether dst=00:50:56:ff:e5:03 src=00:0c:29:07:cf:1e type=IPv4 |<IP version=4 ihl=5 tos=0x0 len=67 id=1 flags= frag=0 ttl=64 proto=tcp checksum=0xb89f src=192.168.148.131 dst=104.18.4.215 |<TCP sport=ftp_data dport=http seq=0 ack=0 dataofs=5 reserved=0 flags=S window=8192 checksum=0xfb9c urgptr=0 |<Raw load='GET /index.html HTTP/1.0 \n\n' |>>>
alomari-scapy>>> c.hide_defaults()
alomari-scapy>>> c
<Ether dst=00:50:56:ff:e5:03 src=00:0c:29:07:cf:1e type=IPv4 |<IP ihl=5 len=67 frag=0 proto=tcp checksum=0xb89f src=192.168.148.131 dst=104.18.4.215 |<TCP dataofs=5 checksum=0xfb9c |<Raw load='GET /index.html HTTP/1.0 \n\n' |>>>
alomari-scapy>>>
```

lsc() function lists available commands



```
0
alomari-scapy>>> lsc()
IPID_count          : Identify IP id values classes in a list of packets
arpcachepoison      : Poison target's cache with (your MAC,victim's IP) couple
arping               : Send ARP who-has requests to determine which hosts are up
arpleak              : Exploit ARP leak flaws, like NetBSD-SA2017-002.
bind_layers          : Bind 2 layers on some specific fields' values.
bridge_and_sniff     : Forward traffic between interfaces if1 and if2, sniff and return
chxdump              : Build a per byte hexadecimal representation
computeNIGroupAddr   : Compute the NI group Address. Can take a FQDN as input parameter
corrupt_bits          :
corrupt_bytes         :
defrag               : defrag(plist) -> ([not fragmented], [defragmented],
defragment           : defragment(plist) -> plist defragmented as much as possible
dhcp_request          : Send a DHCP discover request and return the answer
dyndns_add            : Send a DNS add message to a nameserver for "name" to have a new "r
data"
dyndns_del            : Send a DNS delete message to a nameserver for "name"
etherleak             : Exploit Etherleak flaw
explore               : Function used to discover the Scapy layers and protocols.
fletcher16_checkbytes: Calculates the Fletcher-16 checkbytes returned as 2 byte binary-s
tring.
fletcher16_checksum   : Calculates Fletcher-16 checksum of the given buffer.
fragleak              : --
fragleak2             : --
fragment              : Fragment a big IP datagram
fuzz                  :
getmacbyip            : Return MAC address corresponding to a given IP address
getmacbyip6           : Returns the MAC address corresponding to an IPv6 address
hexdiff               : Show differences between 2 binary strings
hexdump               : Build a tcpdump like hexadecimal view
hexedit               : Run hexedit on a list of packets, then return the edited packets.
hexstr                : Build a fancy tcpdump like hex from bytes.
import_hexcap          : Imports a tcpdump like hexadecimal view
is_promisc            : Try to guess if target is in Promisc mode. The target is provided
```

ls() function lists fields of a protocol

```
0
alomari-scapy>>> ls(IP, verbose=True)
version      : BitField  (4 bits)          = (4)
ihl         : BitField  (4 bits)          = (None)
tos         : XByteField                 = (0)
len         : ShortField                = (None)
id          : ShortField                = (1)
flags        : FlagsField   (3 bits)       = (<Flag 0 ()>)
                  MF, DF, evil
frag        : BitField   (13 bits)       = (0)
ttl          : ByteField                = (64)
proto        : ByteEnumField             = (0)
chksum       : XShortField              = (None)
src          : SourceIPField            = (None)
dst          : DestIPField               = (None)
options      : PacketListField           = ([])

alomari-scapy>>> █
```

Packet Manipulation

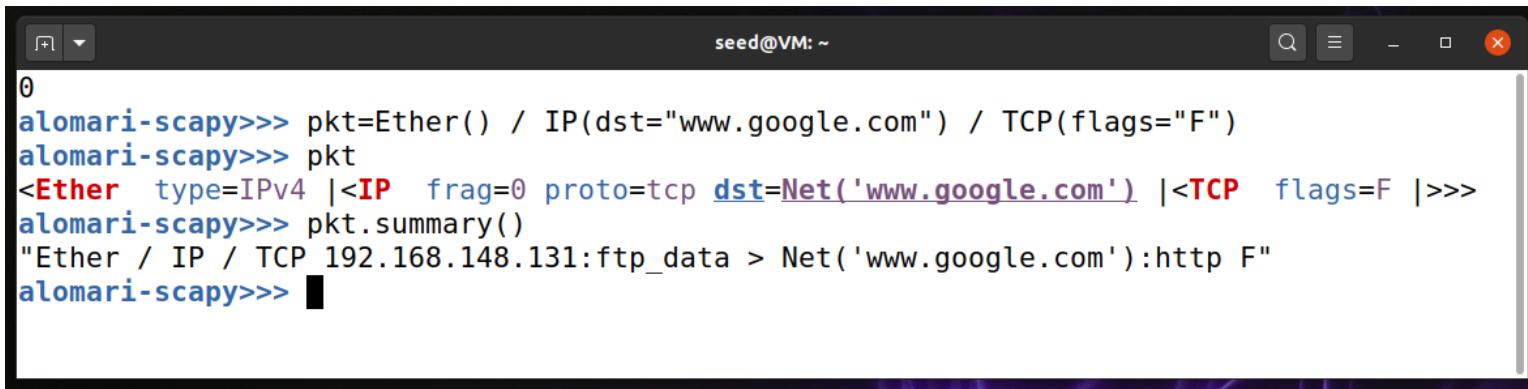
- You can set any field.

```
0
alomari-scapy>>> pkt=IP()/TCP()
alomari-scapy>>> pkt
<IP frag=0 proto=tcp |<TCP |>
alomari-scapy>>> pkt=IP(proto=55)/TCP()
alomari-scapy>>>
alomari-scapy>>> pkt
<IP frag=0 proto=55 |<TCP |>
alomari-scapy>>> ls(pkt)
version      : BitField (4 bits)          = 4          (4)
ihl         : BitField (4 bits)          = None      (None)
tos         : XByteField                = 0          (0)
len         : ShortField               = None      (None)
id          : ShortField               = 1          (1)
flags        : FlagsField (3 bits)       = <Flag 0 ()> (<Flag 0 ()>)
frag        : BitField (13 bits)        = 0          (0)
ttl          : ByteField                = 64         (64)
proto        : ByteEnumField           = 55         (0)
chksum       : XShortField             = None      (None)
src          : SourceIPField           = '127.0.0.1' (None)
dst          : DestIPField              = '127.0.0.1' (None)
options      : PacketListField         = []         ([])

sport        : ShortEnumField          = 20         (20)
dport        : ShortEnumField          = 80         (80)
seq          : IntField                 = 0          (0)
ack          : IntField                 = 0          (0)
dataofs     : BitField (4 bits)        = None      (None)
reserved    : BitField (3 bits)        = 0          (0)
flags        : FlagsField (9 bits)       = <Flag 2 (S)> (<Flag 2 (S)>)
window       : ShortField              = 8192      (8192)
checksum    : XShortField             = None      (None)
urgptr      : ShortField              = 0          (0)
options      : TCPOptionsField        = []         (b' ')
alomari-scapy>>>
```

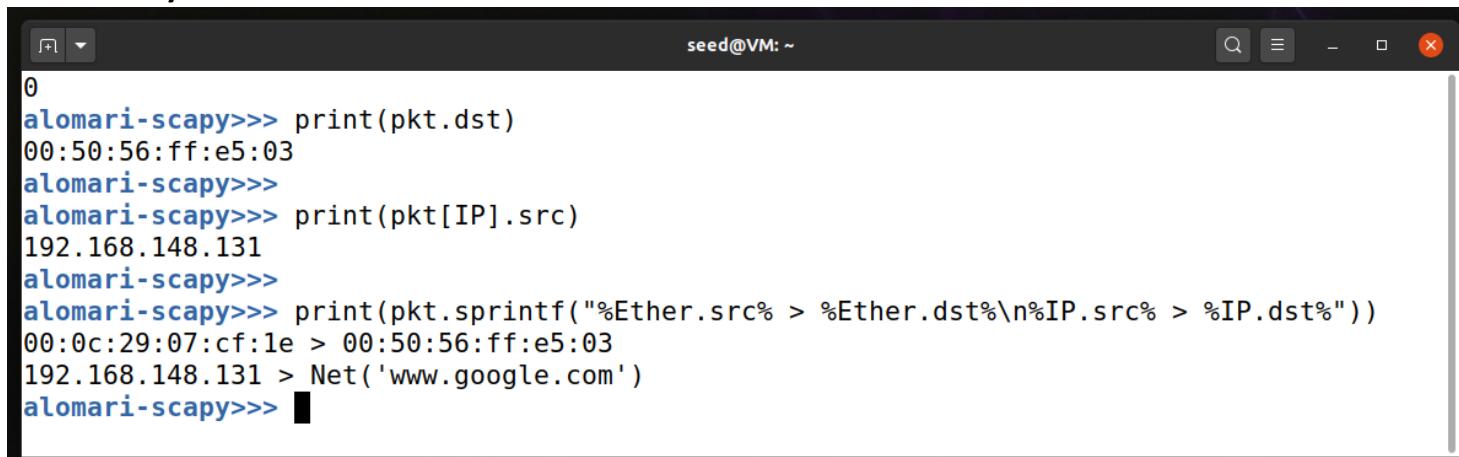
Packet Manipulation

- Scapy selects the correct source IPv4 address, MAC address,....



```
seed@VM: ~
0
alomari-scapy>>> pkt=Ether() / IP(dst="www.google.com") / TCP(flags="F")
alomari-scapy>>> pkt
<Ether type=IPv4 |<IP frag=0 proto=tcp dst=Net('www.google.com') |<TCP flags=F |>>>
alomari-scapy>>> pkt.summary()
"Ether / IP / TCP 192.168.148.131:ftp_data > Net('www.google.com'):http F"
alomari-scapy>>> █
```

- All fields can be easily accessed



```
seed@VM: ~
0
alomari-scapy>>> print(pkt.dst)
00:50:56:ff:e5:03
alomari-scapy>>>
alomari-scapy>>> print(pkt[IP].src)
192.168.148.131
alomari-scapy>>>
alomari-scapy>>> print(pkt.sprintf("%Ether.src% > %Ether.dst%\n%IP.src% > %IP.dst%"))
00:0c:29:07:cf:1e > 00:50:56:ff:e5:03
192.168.148.131 > Net('www.google.com')
alomari-scapy>>> █
```

Generating Sets of Packets

- A field can store many value.

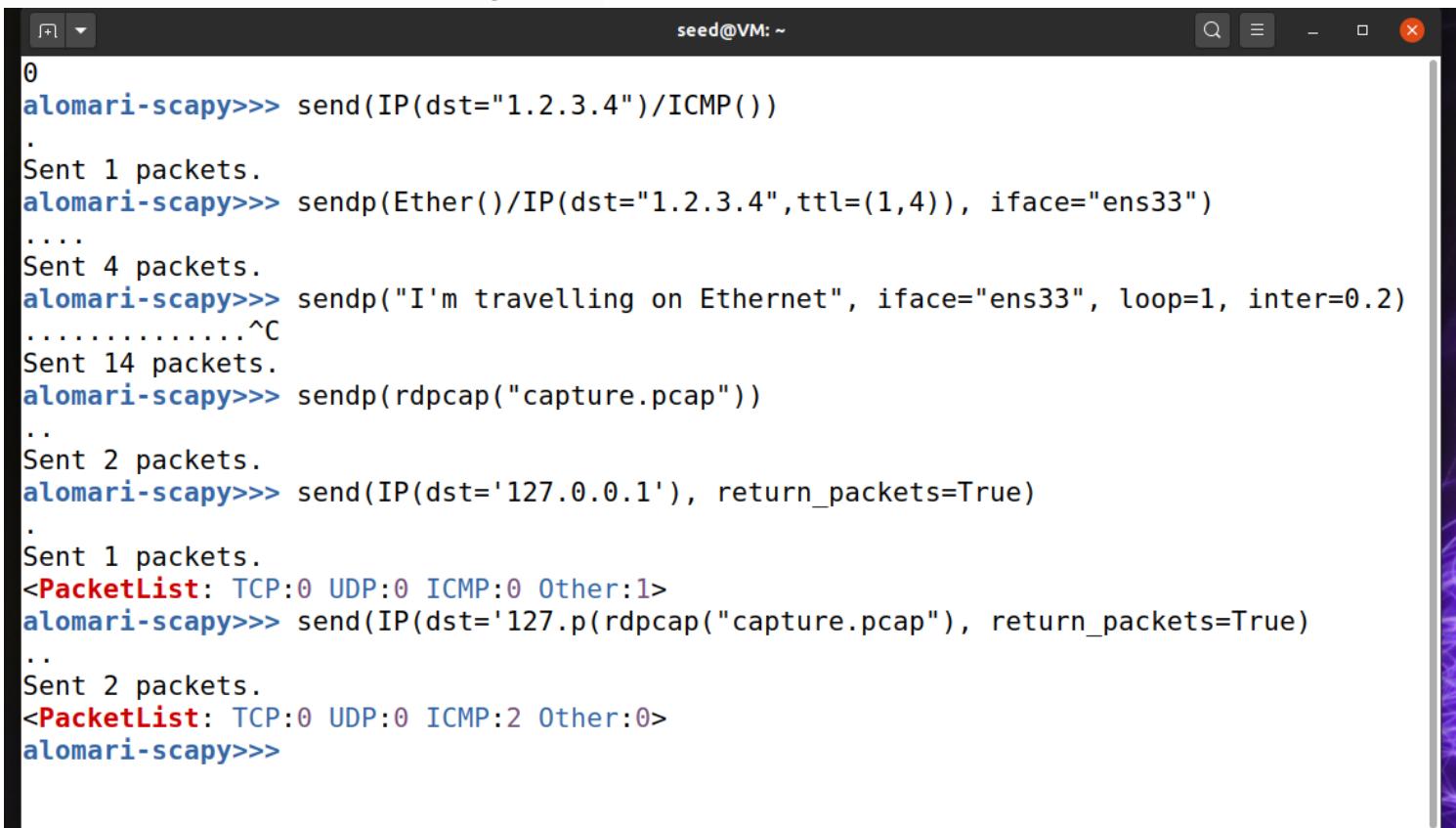


The screenshot shows a terminal window titled "seed@VM: ~" running the scapy interactive mode. The user has defined three sets of IP addresses and then generated a list of combined TCP and HTTP packets for each set. The code and output are as follows:

```
alomari-scapy>>> a=IP(dst="www.google.ca/30")
alomari-scapy>>>
alomari-scapy>>> a
<IP dst=Net('www.google.ca/30') |>
alomari-scapy>>>
alomari-scapy>>> [p for p in a]
[<IP dst=142.251.41.64 |>, <IP dst=142.251.41.65 |>, <IP dst=142.251.41.66 |>, <IP dst=142.251.41.67 |>]
alomari-scapy>>>
alomari-scapy>>> b=IP(ttl=[1,2,(5,9)])
alomari-scapy>>> b
<IP ttl=[1, 2, (5, 9)] |>
alomari-scapy>>>
alomari-scapy>>> [p for p in b]
[<IP ttl=1 |>, <IP ttl=2 |>, <IP ttl=5 |>, <IP ttl=6 |>, <IP ttl=7 |>, <IP ttl=8 |>, <IP ttl=9 |>]
alomari-scapy>>>
alomari-scapy>>> c=TCP(dport=[80,443])
alomari-scapy>>>
alomari-scapy>>> [p for p in a/c]
[<IP frag=0 proto=tcp dst=142.251.41.64 |<TCP dport=http |>, <IP frag=0 proto=tcp dst=142.251.41.64 |<TCP dport=https |>, <IP frag=0 proto=tcp dst=142.251.41.65 |<TCP dport=http |>, <IP frag=0 proto=tcp dst=142.251.41.65 |<TCP dport=https |>, <IP frag=0 proto=tcp dst=142.251.41.66 |<TCP dport=http |>, <IP frag=0 proto=tcp dst=142.251.41.66 |<TCP dport=https |>, <IP frag=0 proto=tcp dst=142.251.41.67 |<TCP dport=http |>, <IP frag=0 proto=tcp dst=142.251.41.67 |<TCP dport=https |>]
alomari-scapy>>>
```

Sending Packets

- `send()` function sends packets at layer 3 → it will handle routing and layer 2 for you.
- The `sendp()` function works at layer 2.

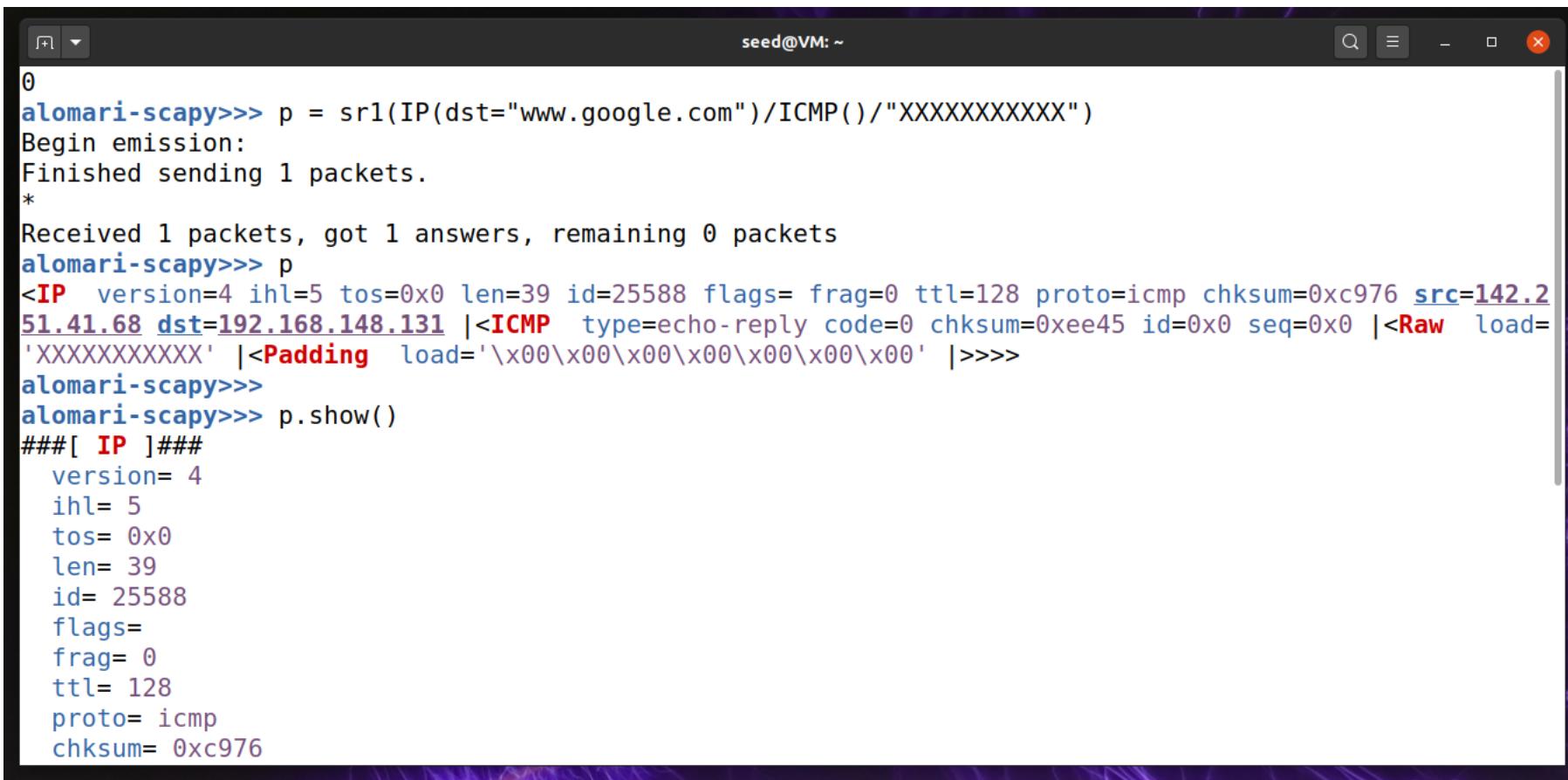


The screenshot shows a terminal window titled "seed@VM: ~" running the Scapy interactive session. The session demonstrates various packet sending functions:

```
0
alomari-scapy>>> send(IP(dst="1.2.3.4")/ICMP())
.
Sent 1 packets.
alomari-scapy>>> sendp(Ether()/IP(dst="1.2.3.4", ttl=(1,4)), iface="ens33")
....
Sent 4 packets.
alomari-scapy>>> sendp("I'm travelling on Ethernet", iface="ens33", loop=1, inter=0.2)
.....^C
Sent 14 packets.
alomari-scapy>>> sendp(rdpcap("capture.pcap"))
..
Sent 2 packets.
alomari-scapy>>> send(IP(dst='127.0.0.1'), return_packets=True)
.
Sent 1 packets.
<PacketList: TCP:0 UDP:0 ICMP:0 Other:1>
alomari-scapy>>> send(IP(dst='127.0.0.1'), rdpcap("capture.pcap"), return_packets=True)
..
Sent 2 packets.
<PacketList: TCP:0 UDP:0 ICMP:2 Other:0>
alomari-scapy>>>
```

Send and Receive Packets (sr)

■ .show()



```
0
alomari-scapy>>> p = sr1(IP(dst="www.google.com")/ICMP()/"XXXXXXXXXX")
Begin emission:
Finished sending 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
alomari-scapy>>> p
<IP version=4 ihl=5 tos=0x0 len=39 id=25588 flags= frag=0 ttl=128 proto=icmp checksum=0xc976 src=142.2
51.41.68 dst=192.168.148.131 |<ICMP type=echo-reply code=0 checksum=0xee45 id=0x0 seq=0x0 |<Raw load=
'XXXXXXXXXX' |<Padding load='\x00\x00\x00\x00\x00\x00\x00' |>>>
alomari-scapy>>>
alomari-scapy>>> p.show()
###[ IP ]###
    version= 4
    ihl= 5
    tos= 0x0
    len= 39
    id= 25588
    flags=
    frag= 0
    ttl= 128
    proto= icmp
    checksum= 0xc976
```

Send and Receive Packets (sr)

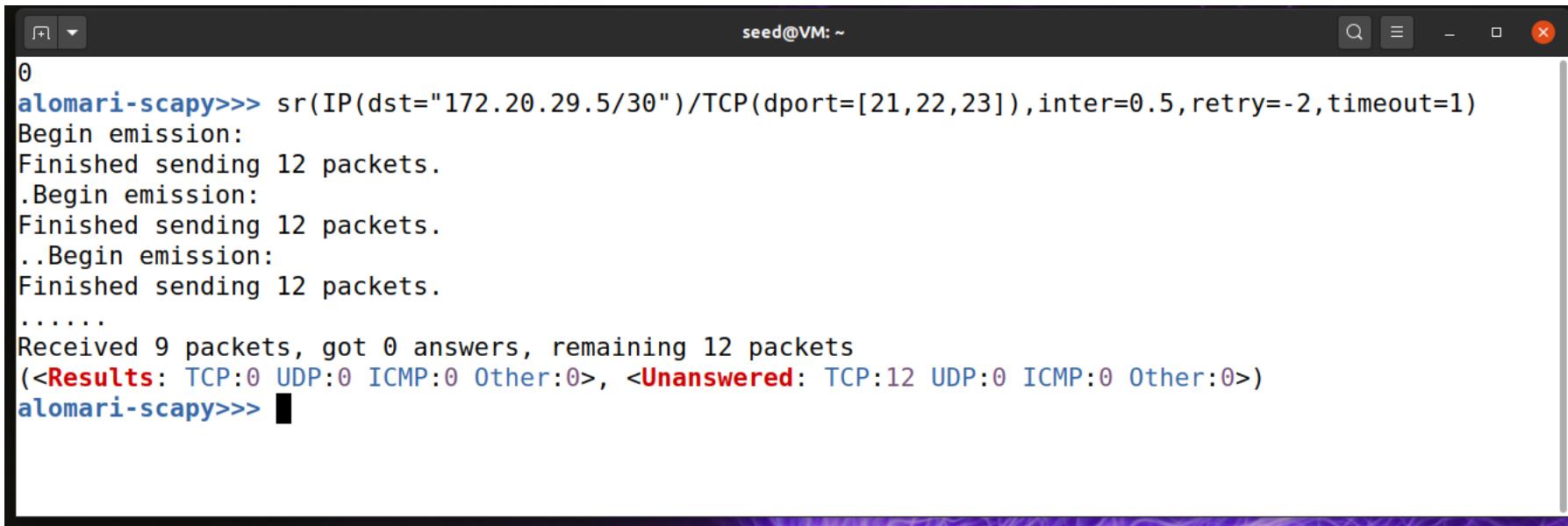
- `sr` returns two elements/lists: 1- (packet sent, answer), and 2- unanswered packets.

```
seed@VM: ~
0
alomari-scapy>>> sr(IP(dst="8.8.8.8")/TCP(dport=[21,22,23]))
Begin emission:
Finished sending 3 packets.
^C
Received 0 packets, got 0 answers, remaining 3 packets
(<Results: TCP:0 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:3 UDP:0 ICMP:0 Other:0>)
alomari-scapy>>> unans.summary()
IP / TCP 192.168.148.131:ftp_data > 192.168.8.1:ftp S
IP / TCP 192.168.148.131:ftp_data > 192.168.8.1:ssh S
IP / TCP 192.168.148.131:ftp_data > 192.168.8.1:telnet S
alomari-scapy>>>
alomari-scapy>>> ans.summary()
alomari-scapy>>>
```

```
seed@VM: ~
0
alomari-scapy>>> sr(IP()/UDP()/DNS(rd=1,qd=DNSQR(qname="www.google.com")))
Begin emission:
Finished sending 1 packets.
.
^C
Received 1 packets, got 0 answers, remaining 1 packets
(<Results: TCP:0 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0 UDP:1 ICMP:0 Other:0>)
alomari-scapy>>> ans.summary()
alomari-scapy>>> unans.summary()
IP / TCP 192.168.148.131:ftp_data > 192.168.8.1:ftp S
IP / TCP 192.168.148.131:ftp_data > 192.168.8.1:ssh S
IP / TCP 192.168.148.131:ftp_data > 192.168.8.1:telnet S
alomari-scapy>>> ans, unans =
alomari-scapy>>> unans.summary()
IP / UDP / DNS Qry "b'www.google.com'"
alomari-scapy>>> ans.summary()
alomari-scapy>>>
```

Send and Receive Packets (sr)

- If there is a limited rate of answers, you can specify a time interval (in seconds) to wait between two packets with the inter parameter.



The screenshot shows a terminal window titled "seed@VM: ~" running the scapy interactive session. The user has entered the command:

```
0
alomari-scapy>>> sr(IP(dst="172.20.29.5/30")/TCP(dport=[21,22,23]),inter=0.5,retry=-2,timeout=1)
```

The session output indicates three rounds of packet transmission:

- First round: "Begin emission", "Finished sending 12 packets".
- Second round: ".Begin emission", "Finished sending 12 packets".
- Third round: "..Begin emission", "Finished sending 12 packets".

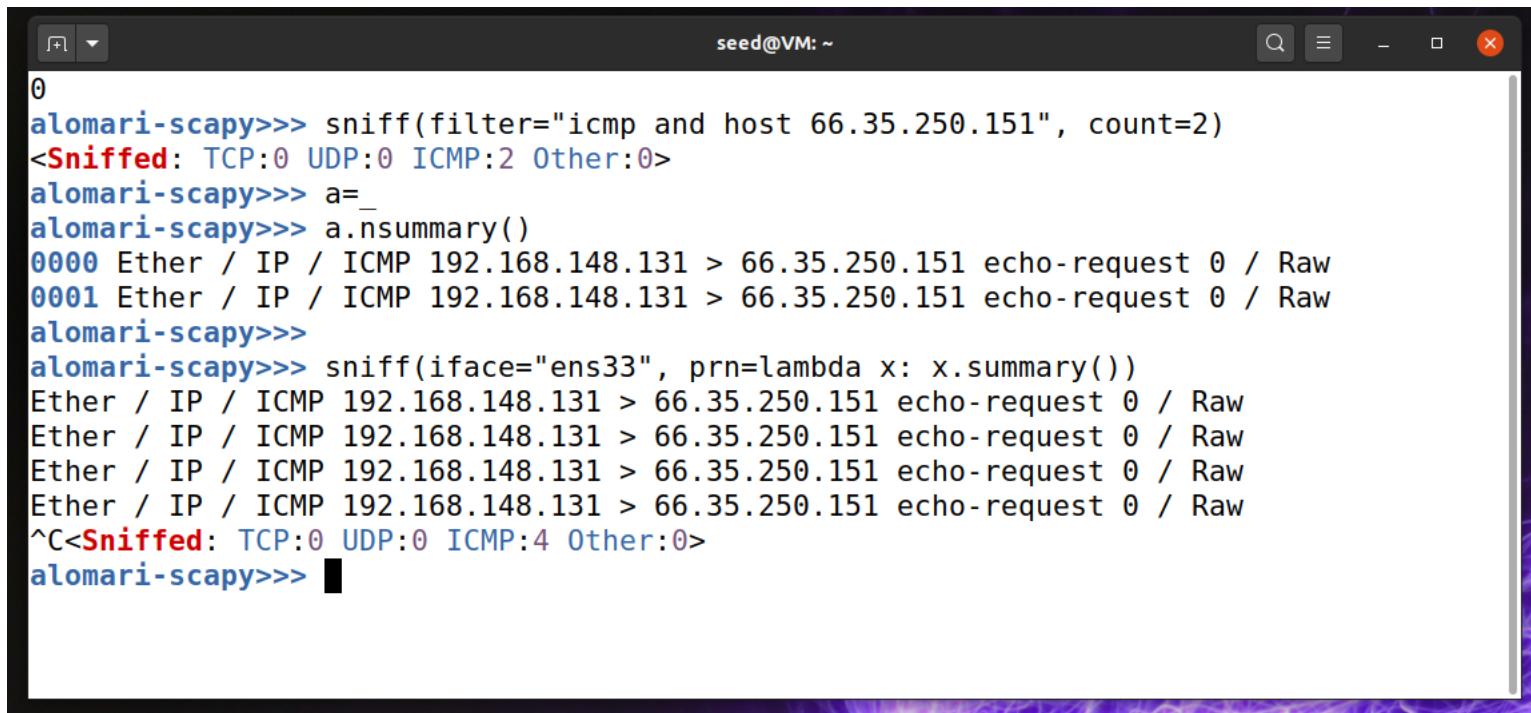
After the third round, the status is:

- "Received 9 packets, got 0 answers, remaining 12 packets"
- Statistics: "(<Results: TCP:0 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:12 UDP:0 ICMP:0 Other:0>)"

The prompt "alomari-scapy>>>" is shown again at the bottom.

Sniffing

- We can easily capture traffic using `sniff()` .



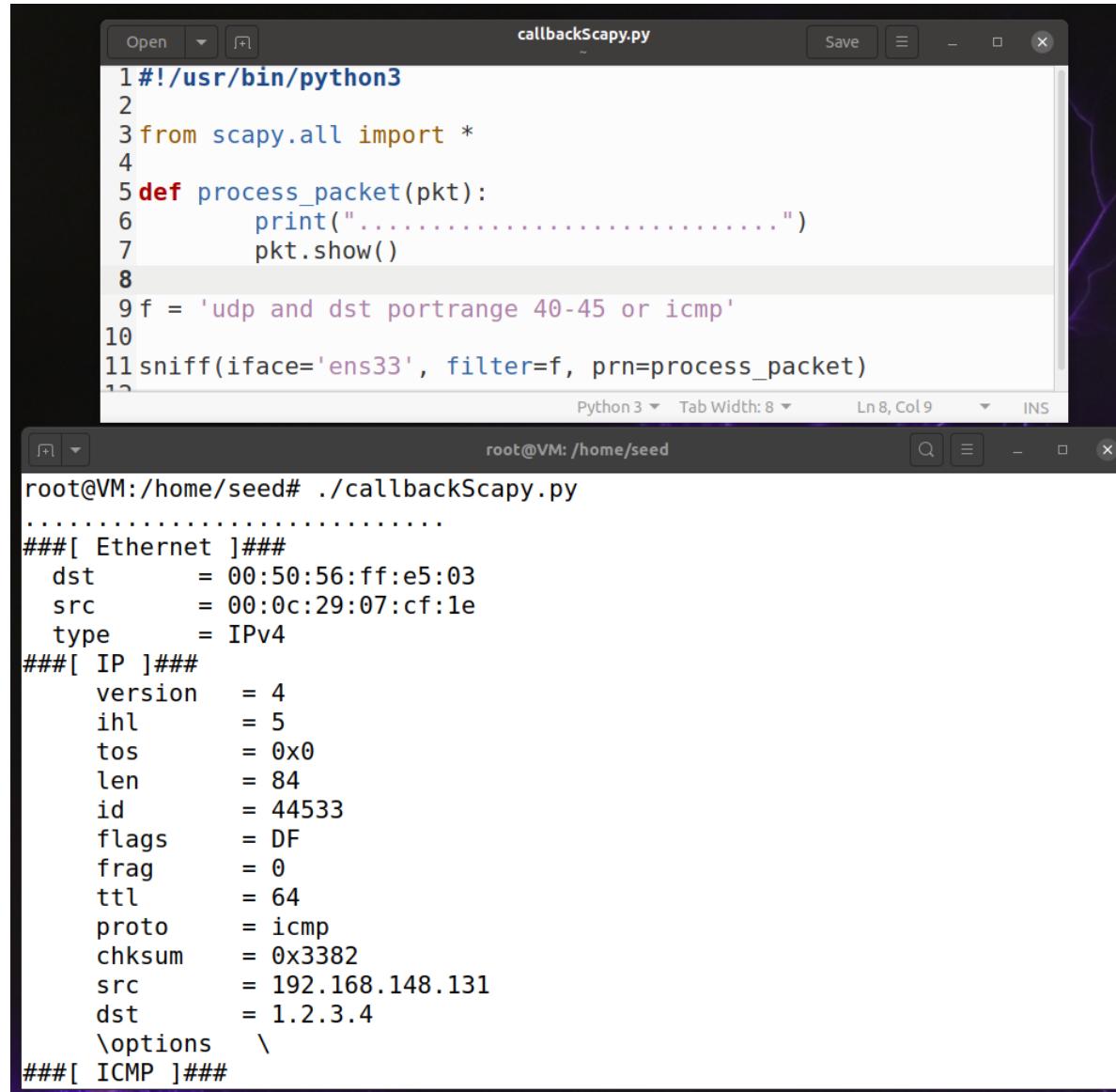
The screenshot shows a terminal window titled "seed@VM: ~" running the scapy Python module. The user has run the command `sniff(filter="icmp and host 66.35.250.151", count=2)`, which captures two ICMP echo-request packets from the host 192.168.148.131 to 66.35.250.151. The user then runs `a.nsummary()` to display the summary of these two captured packets. Subsequently, the user runs `sniff(iface="ens33", prn=lambda x: x.summary())` to capture packets on the interface ens33. This results in four more ICMP echo-request packets being captured and displayed. Finally, the user presses Ctrl-C to stop the sniffing process.

```
0
alomari-scapy>>> sniff(filter="icmp and host 66.35.250.151", count=2)
<Sniffed: TCP:0 UDP:0 ICMP:2 Other:0>
alomari-scapy>>> a=
alomari-scapy>>> a.nsummary()
0000 Ether / IP / ICMP 192.168.148.131 > 66.35.250.151 echo-request 0 / Raw
0001 Ether / IP / ICMP 192.168.148.131 > 66.35.250.151 echo-request 0 / Raw
alomari-scapy>>>
alomari-scapy>>> sniff(iface="ens33", prn=lambda x: x.summary())
Ether / IP / ICMP 192.168.148.131 > 66.35.250.151 echo-request 0 / Raw
Ether / IP / ICMP 192.168.148.131 > 66.35.250.151 echo-request 0 / Raw
Ether / IP / ICMP 192.168.148.131 > 66.35.250.151 echo-request 0 / Raw
Ether / IP / ICMP 192.168.148.131 > 66.35.250.151 echo-request 0 / Raw
^C<Sniffed: TCP:0 UDP:0 ICMP:4 Other:0>
alomari-scapy>>> █
```

Command	Effect
raw(pkt)	assemble the packet
hexdump(pkt)	have a hexadecimal dump
ls(pkt)	have the list of fields values
pkt.summary()	for a one-line summary
pkt.show()	for a developed view of the packet
pkt.show2()	same as show but on the assembled packet (checksum is calculated)
pkt.sprintf()	fills a format string with fields values of the packet
pkt.decode_payload_as()	changes the way the payload is decoded
pkt.psdump()	draws a PostScript diagram with explained dissection
pkt.pdfdump()	draws a PDF with explained dissection
pkt.command()	return a Scapy command that can generate the packet
pkt.json()	return a JSON string representing the packet

Using Scapy in Python – Sniff Packets

Invoke a Callback function



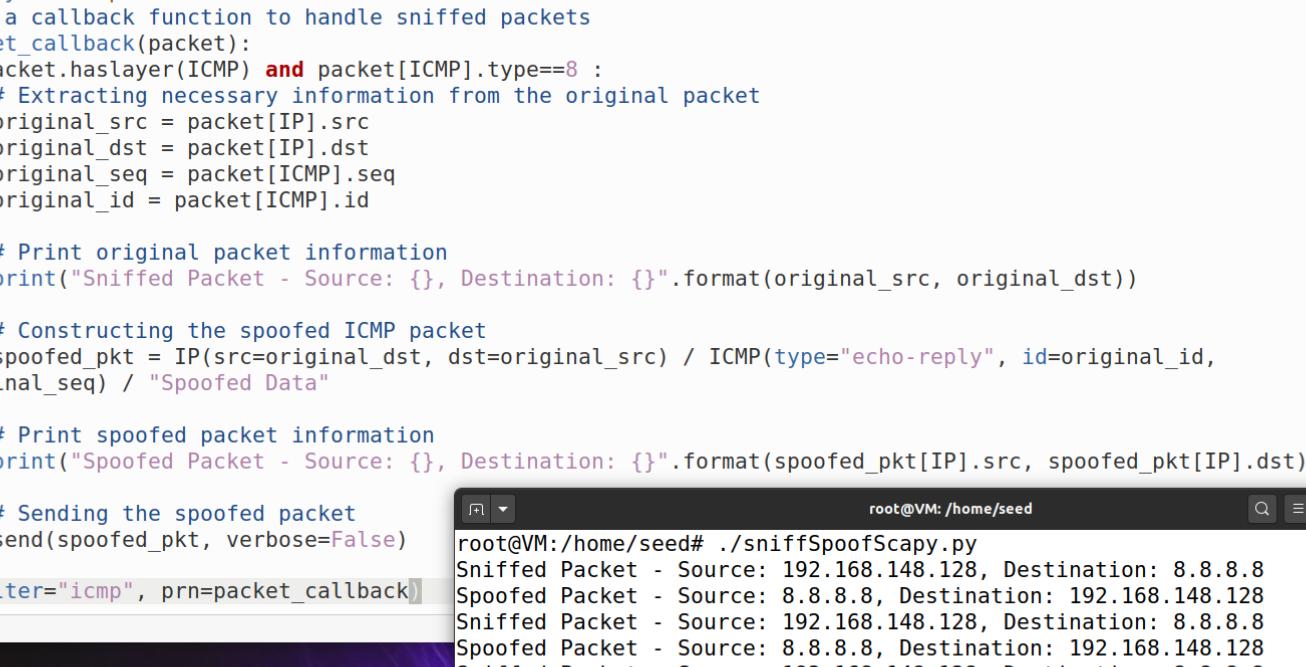
The image shows a terminal window titled "root@VM: /home/seed" running a Python script named "callbackScapy.py". The script uses the Scapy library to sniff network traffic on interface "ens33" with a specific filter. It defines a callback function "process_packet" that prints a dotted separator and shows the packet details. The terminal output shows the details of an ICMP packet captured by the script.

```
#!/usr/bin/python3
from scapy.all import *
def process_packet(pkt):
    print(".....")
    pkt.show()
f = 'udp and dst portrange 40-45 or icmp'
sniff(iface='ens33', filter=f, prn=process_packet)
```

```
root@VM:/home/seed# ./callbackScapy.py
.....
###[ Ethernet ]###
dst      = 00:50:56:ff:e5:03
src      = 00:0c:29:07:cf:1e
type     = IPv4
###[ IP ]###
version  = 4
ihl      = 5
tos      = 0x0
len      = 84
id       = 44533
flags    = DF
frag     = 0
ttl      = 64
proto    = icmp
chksum   = 0x3382
src      = 192.168.148.131
dst      = 1.2.3.4
\options  \
###[ ICMP ]###
```

1- Sniff Then Spoof – 8.8.8.8

- Scapy automatically handles the MAC address resolution when constructing and sending packets.
 - **Question:** Why are we getting duplicate replies?



```
1#!/usr/bin/python3
2from scapy.all import *
3# Define a callback function to handle sniffed packets
4def packet_callback(packet):
5    if packet.haslayer(ICMP) and packet[ICMP].type==8 :
6        # Extracting necessary information from the original packet
7        original_src = packet[IP].src
8        original_dst = packet[IP].dst
9        original_seq = packet[ICMP].seq
10       original_id = packet[ICMP].id
11
12       # Print original packet information
13       print("Sniffed Packet - Source: {}, Destination: {}".format(original_src, original_dst))
14
15       # Constructing the spoofed ICMP packet
16       spoofed_pkt = IP(src=original_dst, dst=original_src) / ICMP(type="echo-reply", id=original_id,
 seq=original_seq) / "Spoofed Data"
17
18       # Print spoofed packet information
19       print("Spoofed Packet - Source: {}, Destination: {}".format(spoofed_pkt[IP].src, spoofed_pkt[IP].dst))
20
21       # Sending the spoofed packet
22       send(spoofed_pkt, verbose=False)
23
24sniff(filter="icmp", prn=packet_callback)
```

root@VM:/home/seed# ./sniffSpoofScapy.py

Sniffed Packet - Source: 192.168.148.128, Destination: 8.8.8.8

Spoofed Packet - Source: 8.8.8.8, Destination: 192.168.148.128

Sniffed Packet - Source: 192.168.148.128, Destination: 8.8.8.8

Spoofed Packet - Source: 8.8.8.8, Destination: 192.168.148.128

Sniffed Packet - Source: 192.168.148.128, Destination: 8.8.8.8

Spoofed Packet - Source: 8.8.8.8, Destination: 192.168.148.128

Sniffed Packet - Source: 192.168.148.128, Destination: 8.8.8.8

Spoofed Packet - Source: 8.8.8.8, Destination: 192.168.148.128

```
File Actions Edit View Help
kali㉿kali ~
192.168.148.128 romari@kali:[~]ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=128 time=18.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=128 time=10.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=128 time=9.25 ms

— 8.8.8.8 ping statistics —
3 packets transmitted, 3 received, 0% packet loss, time 2006ms
rtt min/avg/max/mdev = 9.245/12.875/18.576/4.081 ms

192.168.148.128 romari@kali:[~]ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=128 time=9.39 ms
20 bytes from 8.8.8.8: icmp_seq=1 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=2 ttl=128 time=10.2 ms
20 bytes from 8.8.8.8: icmp_seq=2 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=3 ttl=128 time=7.86 ms
20 bytes from 8.8.8.8: icmp_seq=3 ttl=64 (truncated)
64 bytes from 8.8.8.8: icmp_seq=4 ttl=128 time=8.51 ms
20 bytes from 8.8.8.8: icmp_seq=4 ttl=64 (truncated)

— 8.8.8.8 ping statistics —
4 packets transmitted, 4 received, +4 duplicates, 0% packet loss, time 3009ms
rtt min/avg/max/mdev = 7.858/4.493/10.195/4.537 ms

192.168.148.128 romari@kali:[~]
```

2- Sniff Then Spoof – 1.2.3.4

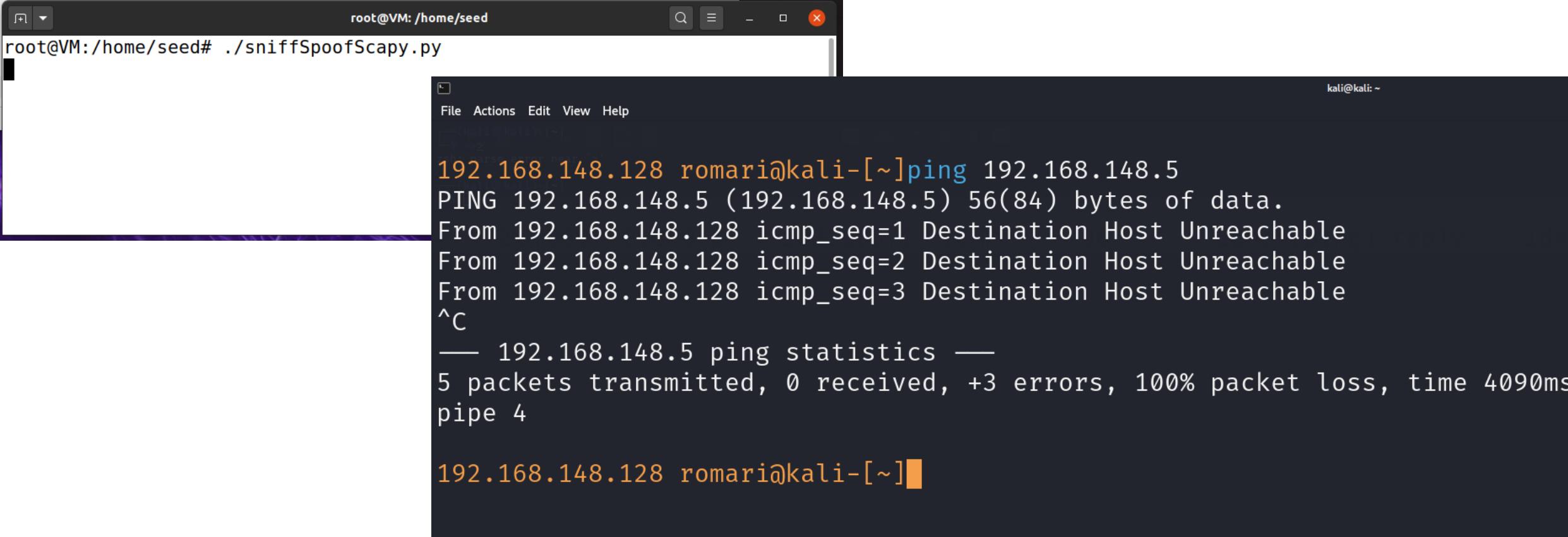
- **Question:** How are we getting replies when 1.2.3.4 doesn't exist?

The image shows two terminal windows side-by-side. The left window, titled 'root@VM: /home/seed', displays the output of the command `./sniffSpoofScapy.py`. It shows several 'Sniffed Packet' and 'Spoofed Packet' entries, indicating traffic being monitored and modified. The right window, titled 'File Actions Edit View Help' and showing the path `/home/romari/Desktop/`, shows a ping session to the IP address 1.2.3.4. The user sends five ICMP echo requests (ping) to 1.2.3.4, which is shown as the source in the traffic capture. The responses are truncated ICMP echo replies from 1.2.3.4. After the fifth packet, the user presses `^C` to stop the ping. The statistics show 5 packets transmitted, 5 received, and 0% packet loss.

```
root@VM:/home/seed# ./sniffSpoofScapy.py
Sniffed Packet - Source: 192.168.148.128, Destination: 1.2.3.4
Spoofed Packet - Source: 1.2.3.4, Destination: 192.168.148.128
Sniffed Packet - Source: 192.168.148.128, Destination: 1.2.3.4
Spoofed Packet - Source: 1.2.3.4, Destination: 192.168.148.128
Sniffed Packet - Source: 192.168.148.128, Destination: 1.2.3.4
Spoofed Packet - Source: 1.2.3.4, Destination: 192.168.148.128
Sniffed Packet - Source: 192.168.148.128, Destination: 1.2.3.4
Spoofed Packet - Source: 1.2.3.4, Destination: 192.168.148.128
192.168.148.128 romari@kali-[~]ping 1.2.3.4
PING 1.2.3.4 (1.2.3.4) 56(84) bytes of data.
20 bytes from 1.2.3.4: icmp_seq=1 ttl=64 (truncated)
20 bytes from 1.2.3.4: icmp_seq=2 ttl=64 (truncated)
20 bytes from 1.2.3.4: icmp_seq=3 ttl=64 (truncated)
20 bytes from 1.2.3.4: icmp_seq=4 ttl=64 (truncated)
20 bytes from 1.2.3.4: icmp_seq=5 ttl=64 (truncated)
^C
— 1.2.3.4 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4007ms
rtt min/avg/max/mdev = 9223372036854775.807/0.000/0.000/0.000 ms
192.168.148.128 romari@kali-[~]
```

3- Sniff Then Spoof – Host on local network that doesn't exist – e.g., 192.168.148.5

- **Question:** Why is our sniff then spoof program not producing any output?



The screenshot shows a terminal window with two panes. The top pane is a root shell on a VM, and the bottom pane is a user shell on a Kali Linux host.

In the root shell (root@VM: /home/seed#), the command ./sniffSpoofScapy.py is run, which triggers a ping to 192.168.148.5.

In the user shell (kali@kali: ~), the ping command is shown failing:

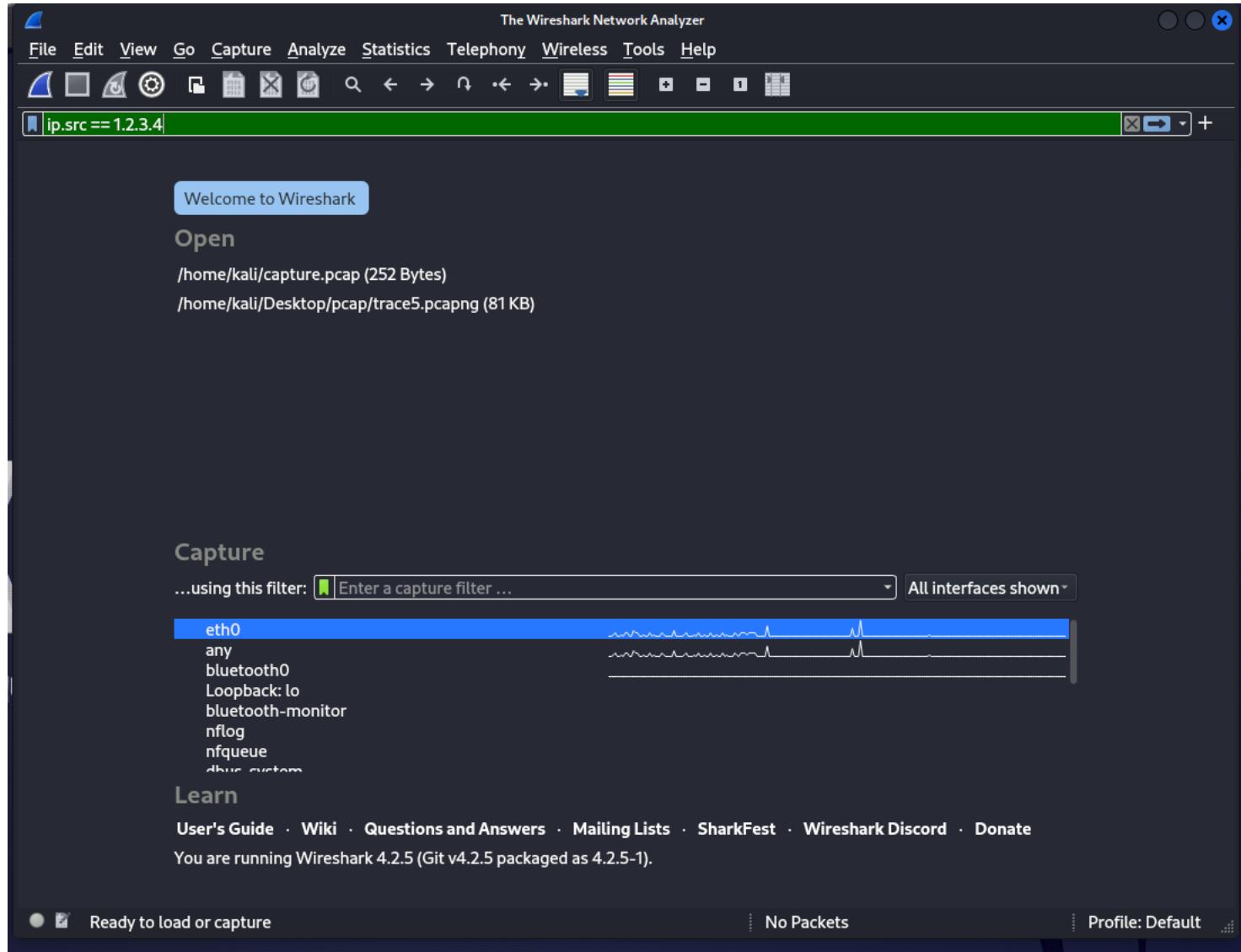
```
192.168.148.128 romariokali-[~]ping 192.168.148.5
PING 192.168.148.5 (192.168.148.5) 56(84) bytes of data.
From 192.168.148.128 icmp_seq=1 Destination Host Unreachable
From 192.168.148.128 icmp_seq=2 Destination Host Unreachable
From 192.168.148.128 icmp_seq=3 Destination Host Unreachable
^C
— 192.168.148.5 ping statistics —
5 packets transmitted, 0 received, +3 errors, 100% packet loss, time 4090ms
pipe 4
```

The user shell prompt 192.168.148.128 romariokali-[~] is visible at the bottom.

Wireshark

- Wireshark provides a graphical user interface (GUI) for capturing, viewing, and analyzing packets.
- It decodes data payloads if encryption keys are available and offers advanced filtering options.

Wireshark



The screenshot shows a Wireshark capture of two ICMP echo requests and replies. The first packet (Frame 1) is an ICMP echo request from 192.168.1.1 to 8.8.8.8. The second packet is an ICMP echo reply from 8.8.8.8 back to 192.168.1.1. Both packets have a length of 98 bytes and a stream index of 98. The Info column provides detailed information about each packet's type and sequence.

No.	Time	Source	Destination	Protocol	Length	Stream index	Info
1	10.000...	192.168.1...	8.8.8.8	ICMP	98	98	Echo (ping) request id=0x1661, seq=1/256, ttl=64 (reply in 2)
2	20.009...	8.8.8.8	192.168.1...	ICMP	98	98	Echo (ping) reply id=0x1661, seq=1/256, ttl=128 (request in 1)

Frame details for the selected ICMP echo request (Frame 1):

- Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
- Ethernet II, Src: VMware_98:fb:db (00:0c:29:98:fb:db), Dst: VMware_ff:e5:03 (00:50:56:ff:e5:03)
- Internet Protocol Version 4, Src: 192.168.148.128, Dst: 8.8.8.8
- Internet Control Message Protocol

Type: 8 (Echo (ping) request)
Code: 0
Checksum: 0x2003 [correct]
[Checksum Status: Good]
Identifier (BE): 5729 (0x1661)
Identifier (LE): 24854 (0x6116)
Sequence Number (BE): 1 (0x0001)
Sequence Number (LE): 256 (0x0100)

Hex dump of the selected ICMP echo request:

0000	00	50	56	ff	e5	03	00	00
0010	00	54	b4	11	40	00	40	01
0020	08	08	08	00	20	03	16	61
0030	00	00	3a	34	08	00	00	00
0040	16	17	18	19	1a	1b	1c	1d
0050	26	27	28	29	2a	2b	2c	2d
0060	36	37						

Packets: 2 - Displayed: 2 (100.0%)

Profile: Default

Using Wireshark for Hacking or Forensics

- Perform Passive Discovery
- Detect Unsecured Applications
- Detect Suspicious Protocols and Applications
- Reassemble and Export Objects
- Decrypt Traffic with an RSA Key

Wireshark Filter Format

- General format:
 1. **Protocol Field:** Specifies which protocol or protocol field to filter on.
 2. **Comparison Operator:** Compares the protocol field to a specified value.
 3. **Logical Operator:** Combines multiple expressions.

Protocol Fields

- Common fields include:
 - **ip.addr** (any IP address)
 - **ip.src** (source IP address)
 - **ip.dst** (destination IP address)
 - **tcp.port** (any TCP port)
 - **tcp.srcport** (source TCP port)
 - **tcp.dstport** (destination TCP port)
 - **http** (HTTP protocol)

Comparison and Logical Operators

Comparison Operators

- `==`: Equal to
- `!=`: Not equal to
- `>`: Greater than
- `<`: Less than
- `>=`: Greater than or equal to
- `<=`: Less than or equal to
- `matches`: Matches a regular expression

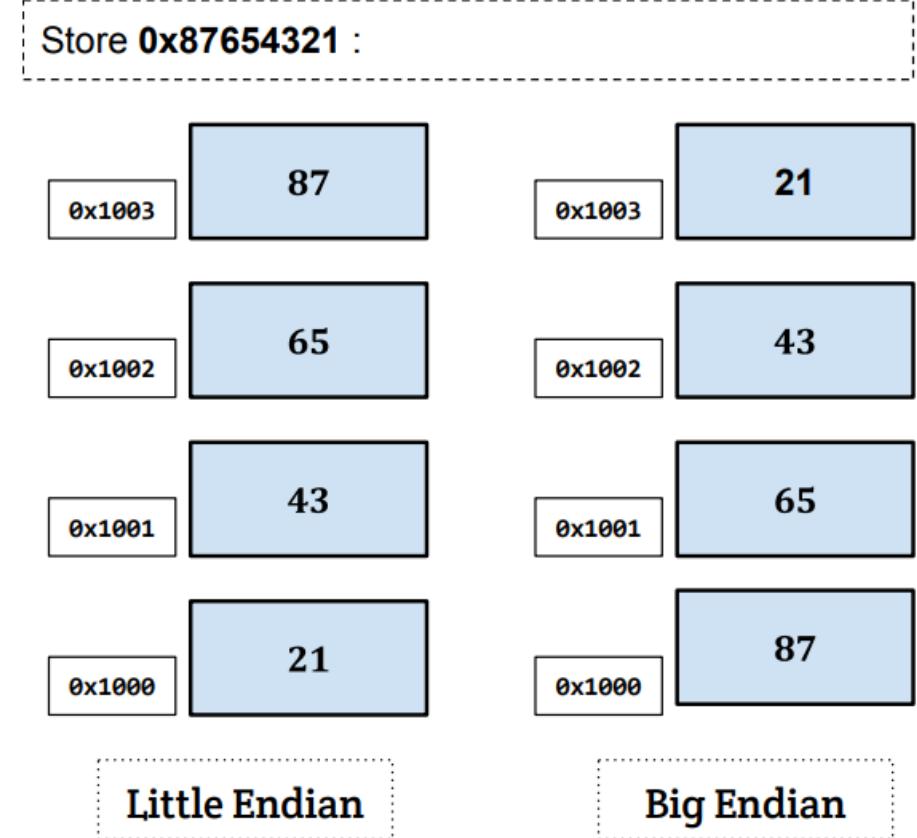
Logical Operators

- `&&`: Logical AND
- `||`: Logical OR
- `!`: Logical NOT

BASIC: Filter out 10.1.1.1 traffic from view	<code>!ip.addr==10.1.1.1</code>
BASIC: TCP traffic to or from port 443	<code>tcp.port==443</code>
BASIC: ICMP or ICMPv6	<code>icmp icmpv6</code>
BASIC: TLS Handshake Packets (including Client Hello, Server Hello)	<code>tls.record.content_type == 22</code>
BASIC: TLS traffic - all [Wireshark v3 supports <code>ssl</code> and <code>tls</code> filters, not just <code>ssl</code>]	<code>tls</code>
TCP: SYN packets	<code>tcp.flags.syn==1 && tcp.flags.ack==0</code>
TCP: SYN/ACK packets (example of a bitwise filter)	<code>tcp.flags & 0x12</code>
TCP: SYN packet with non-zero ACK# field	<code>tcp.flags.syn==1 && tcp.flags.ack==0 && tcp.ack==0</code>
TCP: To/from port 443 or 4430 through 4434	<code>tcp.port in {443 4430..4434}</code>
TCP: Connection refusal or ACK scan	<code>tcp.flags.reset==1 && tcp.flags.ack==1 && tcp.seq==1 && tcp.ack==1</code>
TCP: Data in Urgent Pointer field (rarely seen for legitimate purposes)	<code>tcp.urgent_pointer>0</code>
TLS: Client Hello [Wireshark v3 supports <code>ssl</code> and <code>tls</code> filters, not just <code>ssl</code>]	<code>tls.handshake.type == 1</code>
TLS: Server Hello [Wireshark v3 supports <code>ssl</code> and <code>tls</code> filters, not just <code>ssl</code>]	<code>tls.handshake.type == 2</code>
TLS: TLS Encrypted Alert (followed by FIN, it's probably a connection close)	<code>tls.record.content_type == 21</code>
TLS: Target server contains "badsite" in server name	<code>tls.handshake.extensions_server_name contains "badsite"</code>
DNS: DNS PoinTeR (PTR) query/response	<code>dns.qry.type == 12</code>
DNS: Unusually large DNS answer count (check answers - C&C server list?)	<code>dns.count.answers>10</code>
DNS: Non port-53 traffic to DNS server (x.x.x.x) – TCP and UDP	<code>ip.dst==x.x.x.x && !udp.port==53 && !tcp.port==53</code>
HTTP: HTTP PUT and POST messages	<code>http.request.method in {PUT POST}</code>
HTTP: Requested HTTP objects with.exe/.zip/.jar file name extensions (PERL regex)	<code>http.request.uri matches "\.(exe zip jar)\$"</code>
HTTP: Content type "application" from server	<code>http.content_type contains "application"</code>
HTTP: Redirections (all response codes starting with 3)	<code>http.response.code>299 && http.response.code<400</code>
HTTP: GET command not running over TCP port 80	<code>frame contains "GET" && !tcp.port==80</code>
FTP Unusually long FTP USER name (USER command, space and 0d0a take 7 bytes)	<code>ftp.request.command=="USER" && tcp.len>50</code>
P2P: Peer-to-peer traffic among hosts on 172.16 subnet, but not subnet broadcasts	<code>ip.src==172.16.0.0/16 && ip.dst==172.16.0.0/16 && !ip.dst==172.16.255.255</code>
IRC: Frame contains the JOIN command (upper or lower case) (PERL regex)	<code>frame matches "join #"</code>
Nmap: "Nmap" Identified in HTTP User Agent field (case sensitive)	<code>http.user_agent contains "Nmap"</code>
Nessus: Frame offset 100-199 contain "nessus" in lower case	<code>frame[100-199] contains "nessus"</code>
Nessus: Frame offset 100-199 contains "nessus" in upper or lower case (PERL regex)	<code>frame[100-199] matches "nessus"</code>
MISC: Unexpected applications on the network (see reverse)	<code>tftp irc bittorrent</code>

Endianness

- Endianness: a term that refers to the order in which a given multi-byte data item is stored in memory.
 - **LittleEndian**: store the most significant byte of data at the highest address
 - **BigEndian**: store the most significant byte of data at the lowest address



Endianness In Network Communication

- Computers with different byte orders will “misunderstand” each other.
 - Solution: agree upon a common order for communication
 - This is called “network order”, which is the same as big endian order
- All computers need to convert data between “host order” and “network order”.

Macro	Description
<code>htons ()</code>	Convert unsigned short integer from host order to network order.
<code>htonl ()</code>	Convert unsigned integer from host order to network order.
<code>ntohs ()</code>	Convert unsigned short integer from network order to host order.
<code>ntohl ()</code>	Convert unsigned integer from network order to host order.

Sniffing and Spoofing Attacks - Mitigation

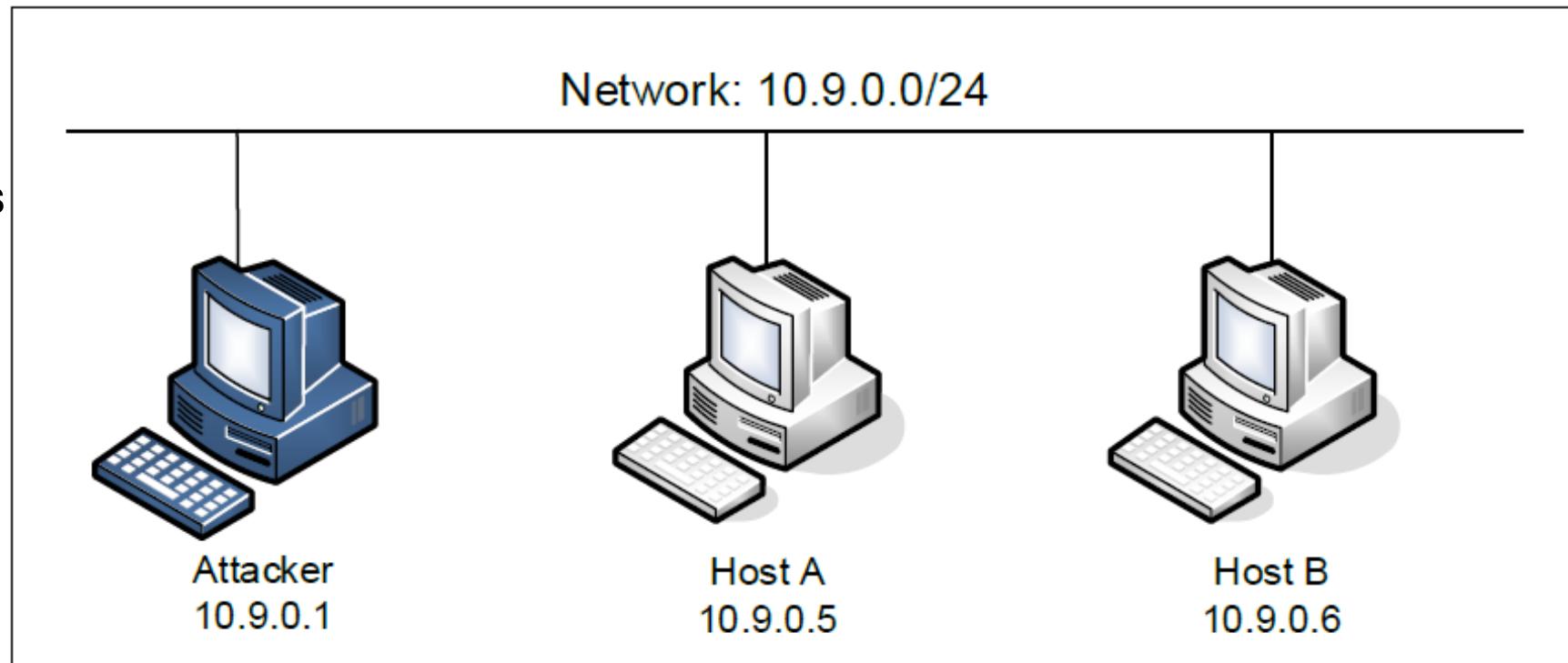
- Any protocol that uses cleartext communication is susceptible to this kind of attack.
- The only requirement for this attack to succeed is to have access to a system between the two communicating systems.
- The mitigation lies in adding an encryption layer on top of any network protocol.

Topics

- Ports and Sockets
- Sending/Receiving packets
- Sniffing and Spoofing Packets
- Tools of the Trade

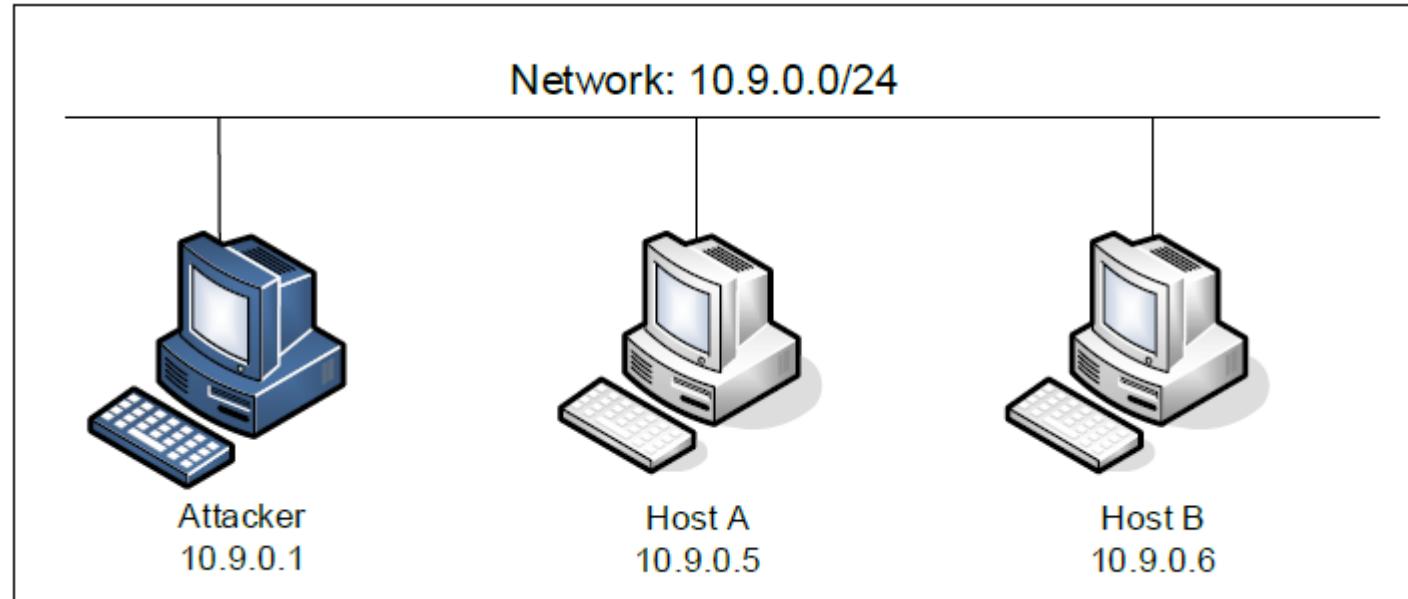
Lab Setup

- Sniffing Packets
- Spoofing ICMP Packets
- Traceroute
- Sniffing and-then
Spoofing



Lab Setup – Part 1 - Using Scapy to Sniff and Spoof Packets

- Task 1.1: Sniffing Packets
- Task 1.2: Spoofing ICMP Packets
- Task 1.3: Traceroute
- Task 1.4: Sniffing and then Spoofing



Lab Setup – Part 2 - Using Wireshark for Forensics

- Task 2.1: Perform Passive Discovery
- Task 2.2: Detect Unsecured Applications
- Task 2.3: Detect Suspicious Protocols and Applications
- Task 2.4: Reassemble and Export Objects
- Task 2.5: Decrypt Traffic with an RSA Key