LASSONDE SCHOOL OF ENGINEERING | YORK U

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

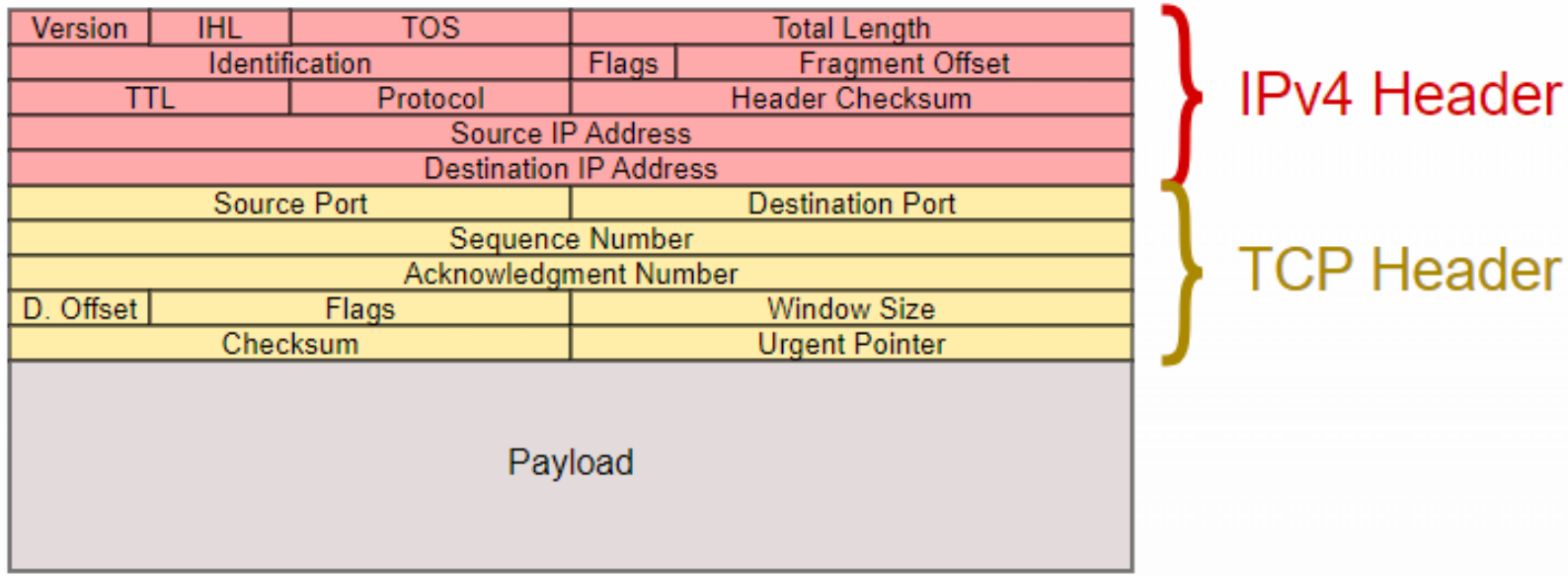
- Ports and Sockets
- Sending/Receiving packets
- Sniffing and Spoofing Packets
- Tools of the Trade

Topics

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

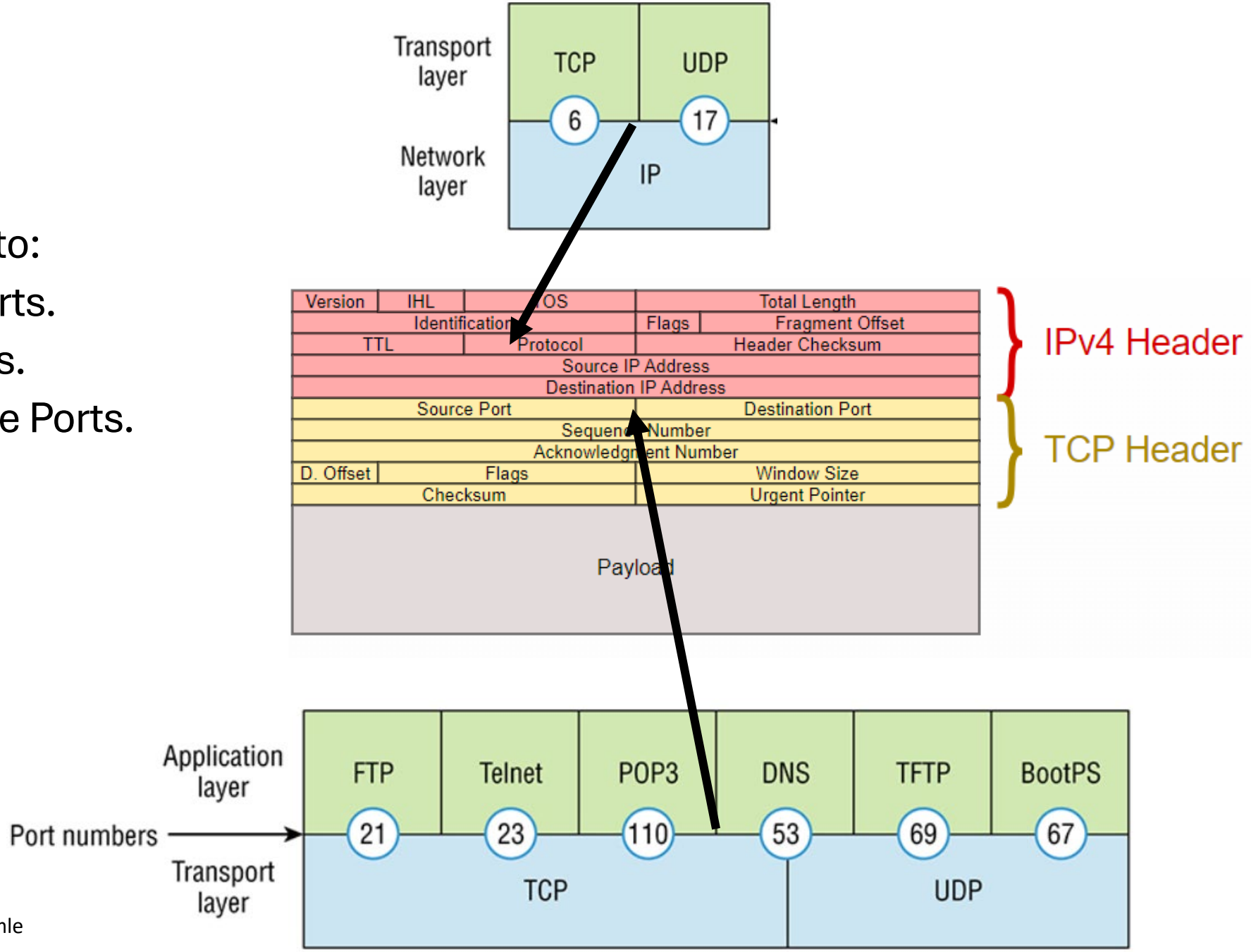
Ports and Sockets

- When a connection is established via the **Transport** layer, it is done using **ports**.
- Port number size is 16-bits ⇨ 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.



Source: CCNA Certification Study Guide by Todd Lammle

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

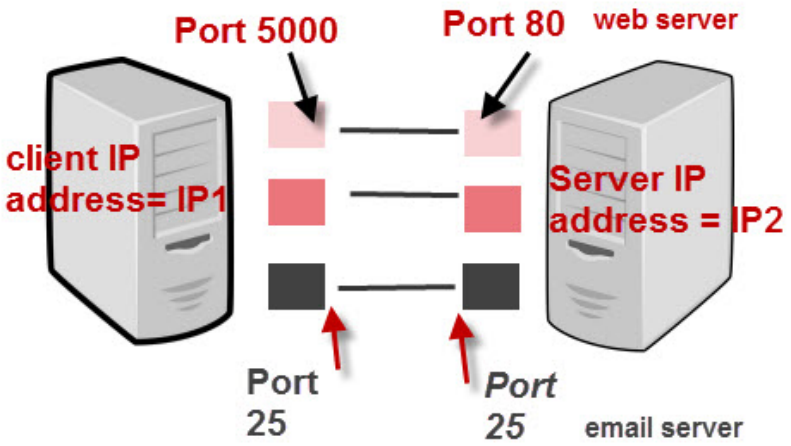
Subset of dynamic ports

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.



IP Address + Port number = Socket

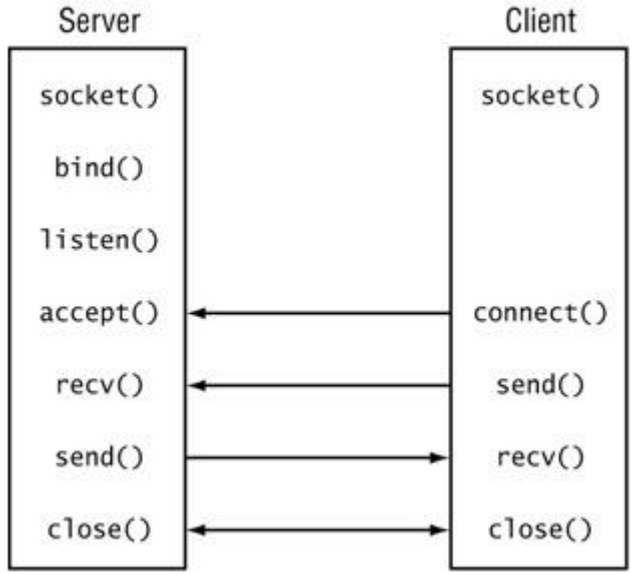
Image Source: http://www.steves-internet-guide.com/tcpip-ports-sockets/

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.

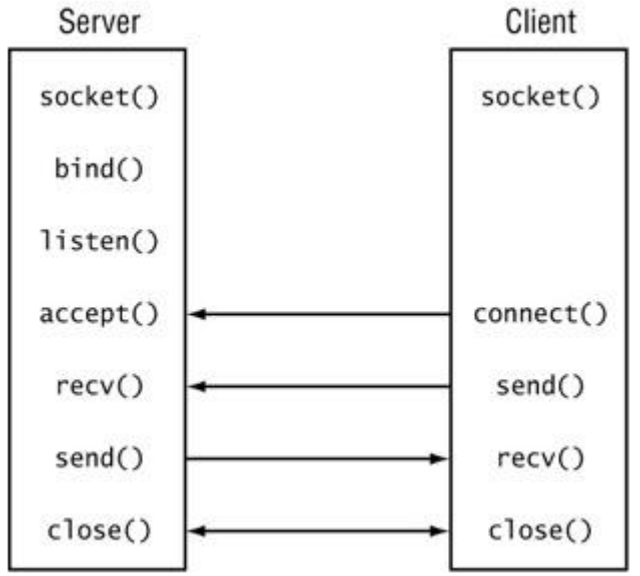


Server-client communication using TCP/IP

Image Source: https://www.engineersgarage.com/communication-between-server-clients-through-socket-programming-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.



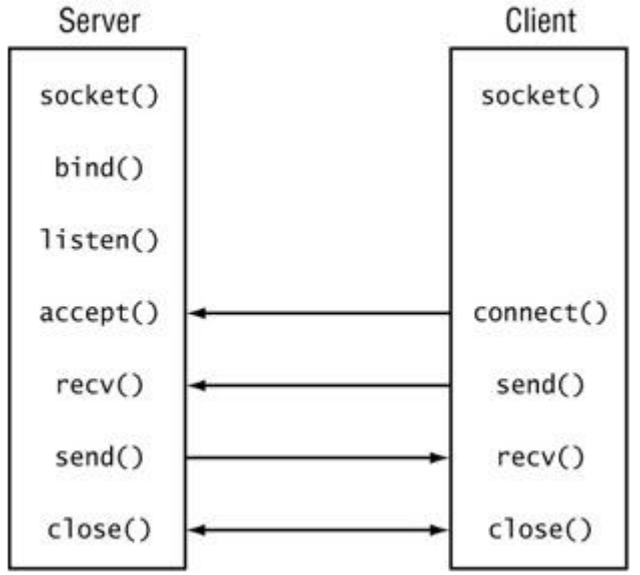
Server-client communication using TCP/IP

Image Source: https://www.engineersgarage.com/communication-between-server-clients-through-socket-programming-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.

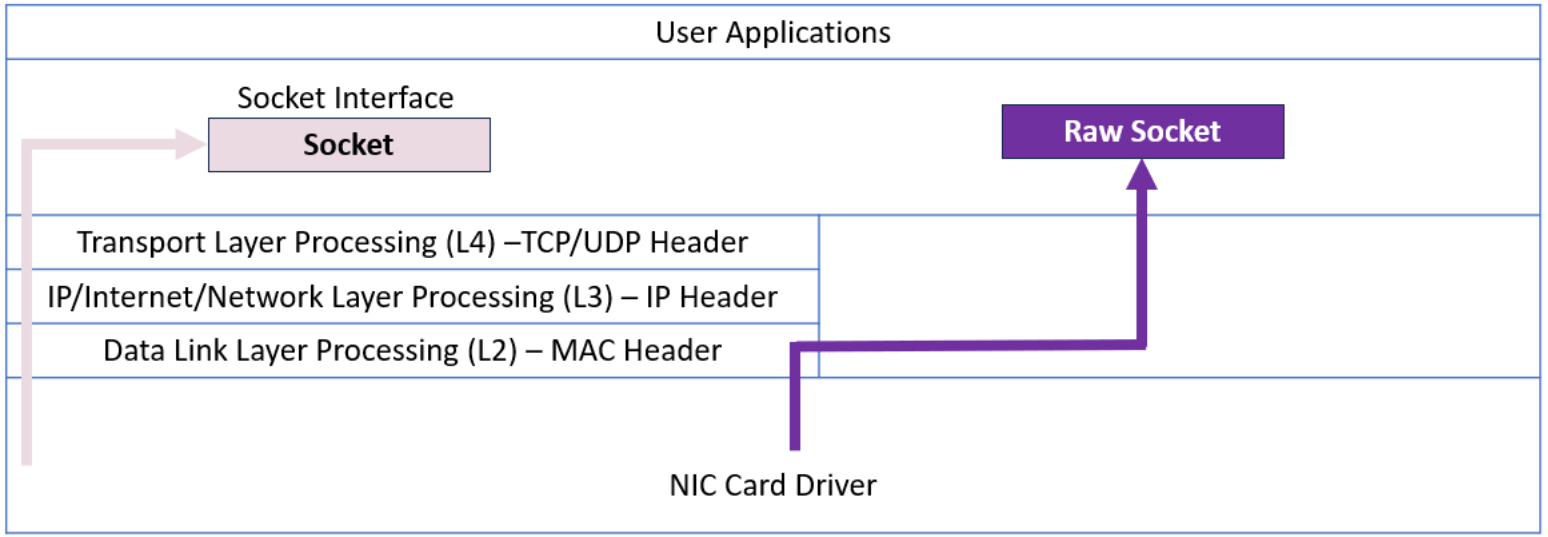


Server-client communication using TCP/IP

Image Source: https://www.engineersgarage.com/communication-between-server-clients-through-socket-programming-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
```

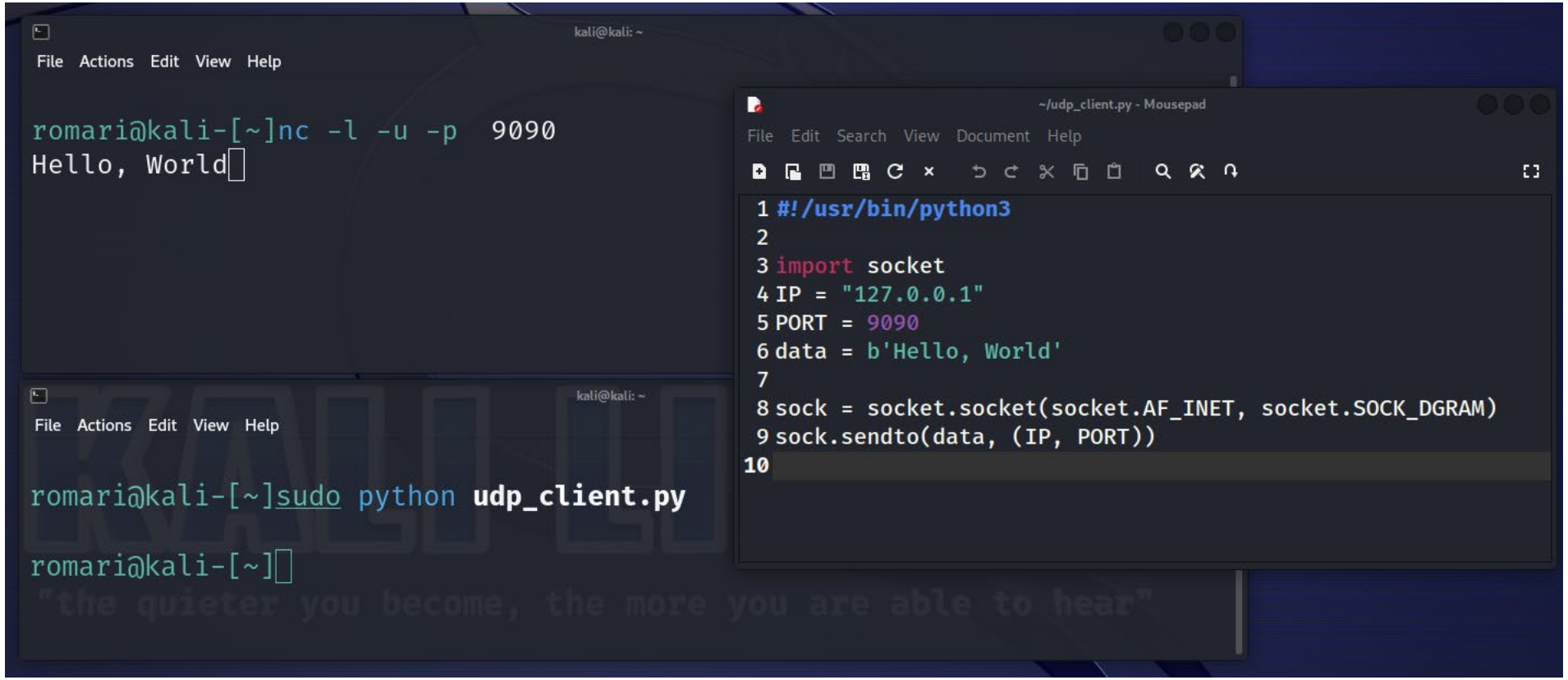
  - Send to bash's pseudo device /dev/tcp and /dev/udp:

```
$ 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



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))
```

.socket(): Creates a new socket object.

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

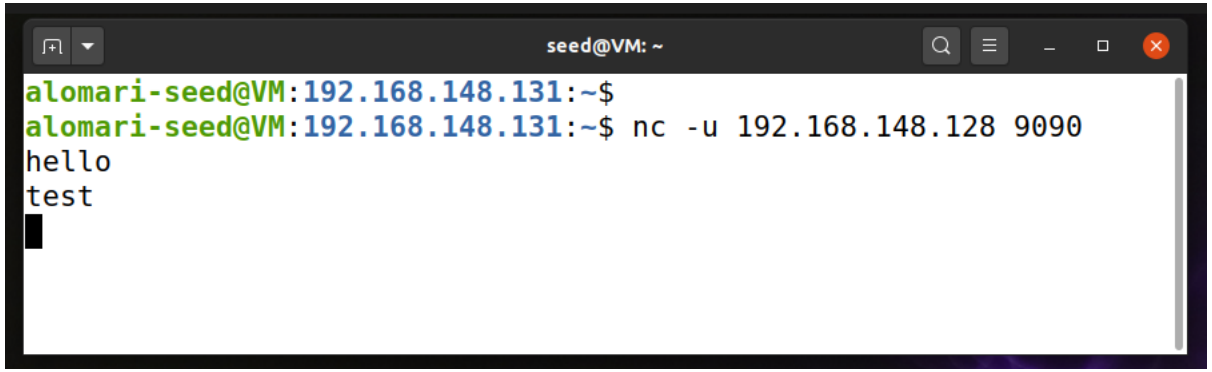
- **.AF_INET6**

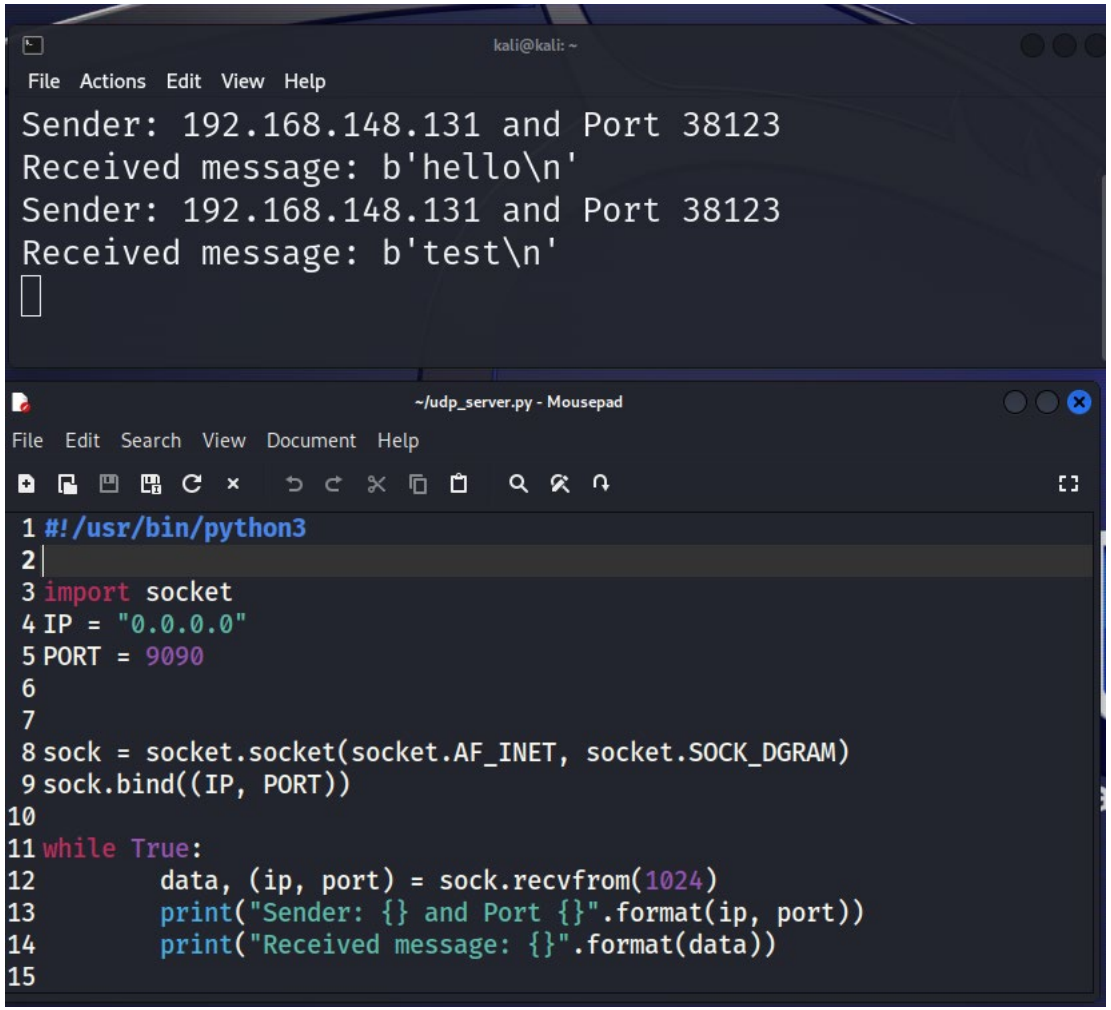
.send: for TCP.

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

- **.SOCK_STREAM** (TCP).

Receiving Packets - UDP Server Example





udp_server.py

.bind((IP, PORT)): Bind the socket to an address and port.

- **.bind(('::', 8080))** binds to all available IPv6 interfaces.

.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))
```

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]]]]]

Equivalent to https

- Add TLS encryption:
  - [Ethernet [IP [TCP [TLS [HTTP [Payload]]]]]]

- Encapsulate further with Network layer encryption:
  - [Ethernet [IPsec [IP [TCP [TLS [HTTP [Payload]]]]]]]

VPN

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]]]]]]

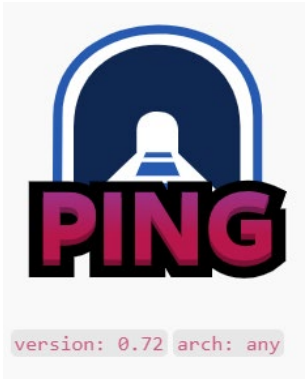
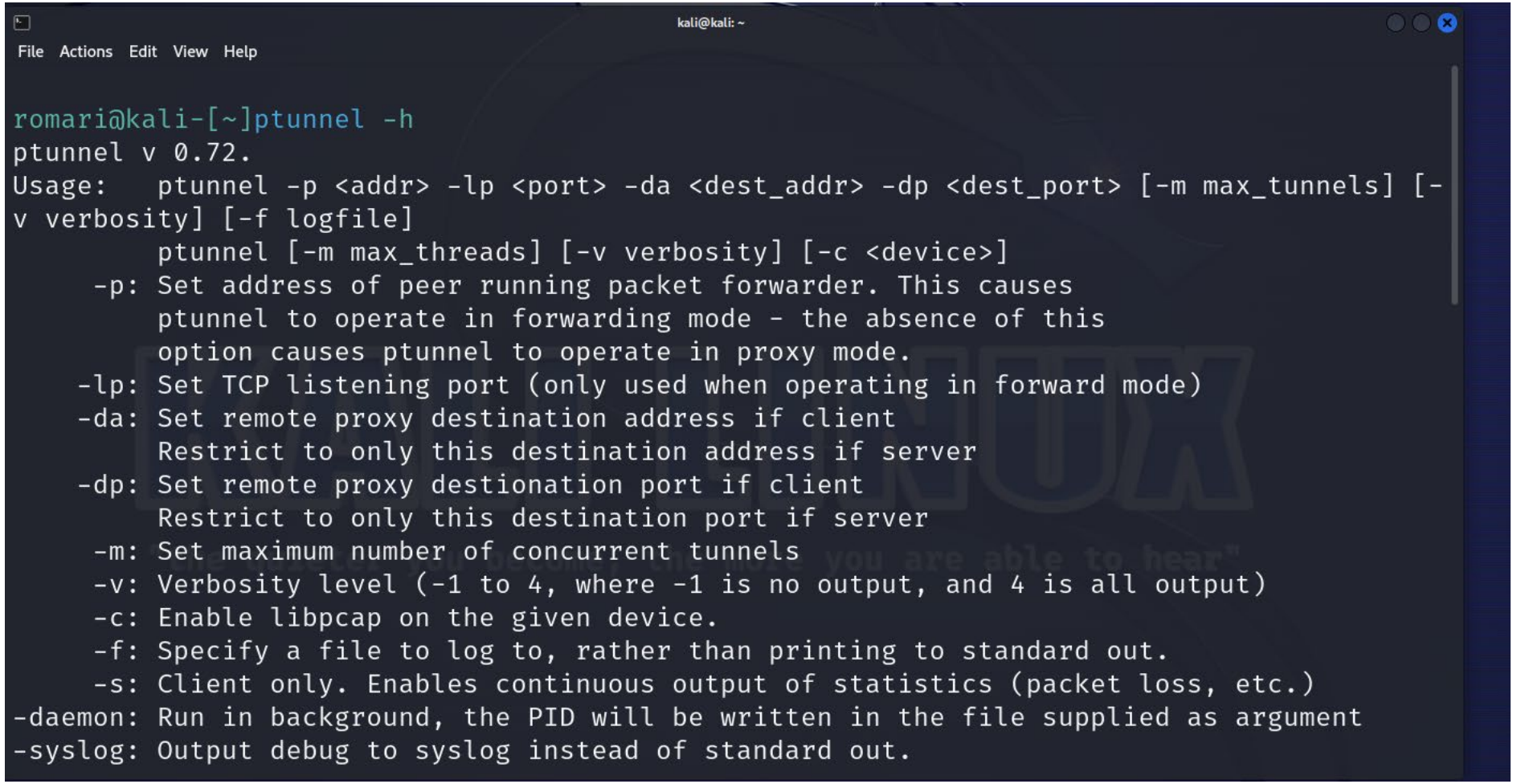


Image Source: http://phrack.org/issues/49/6.html#article
https://sourceforge.net/projects/http-tunnel/


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 destionation 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.

Leonardo.ai

Spoofing Packets Using Raw Sockets

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

Leonardo.ai

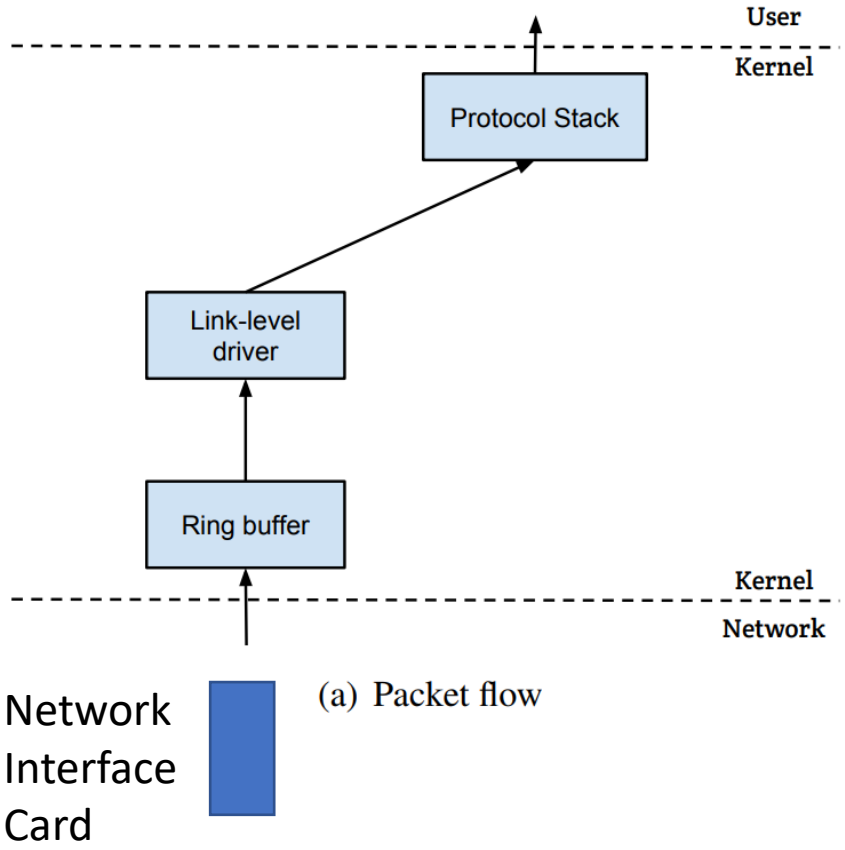
Sniffing and Then Spoofing

- 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.

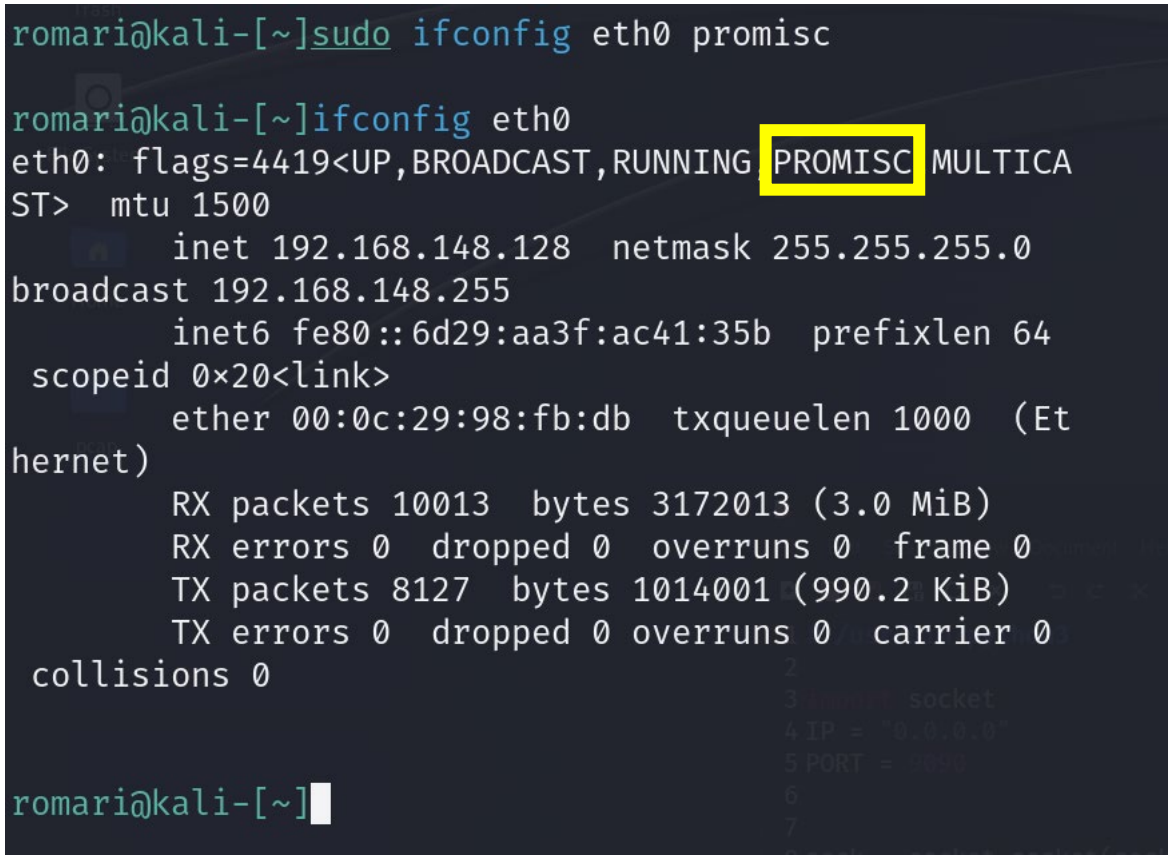
How Packets Are Received

- 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?



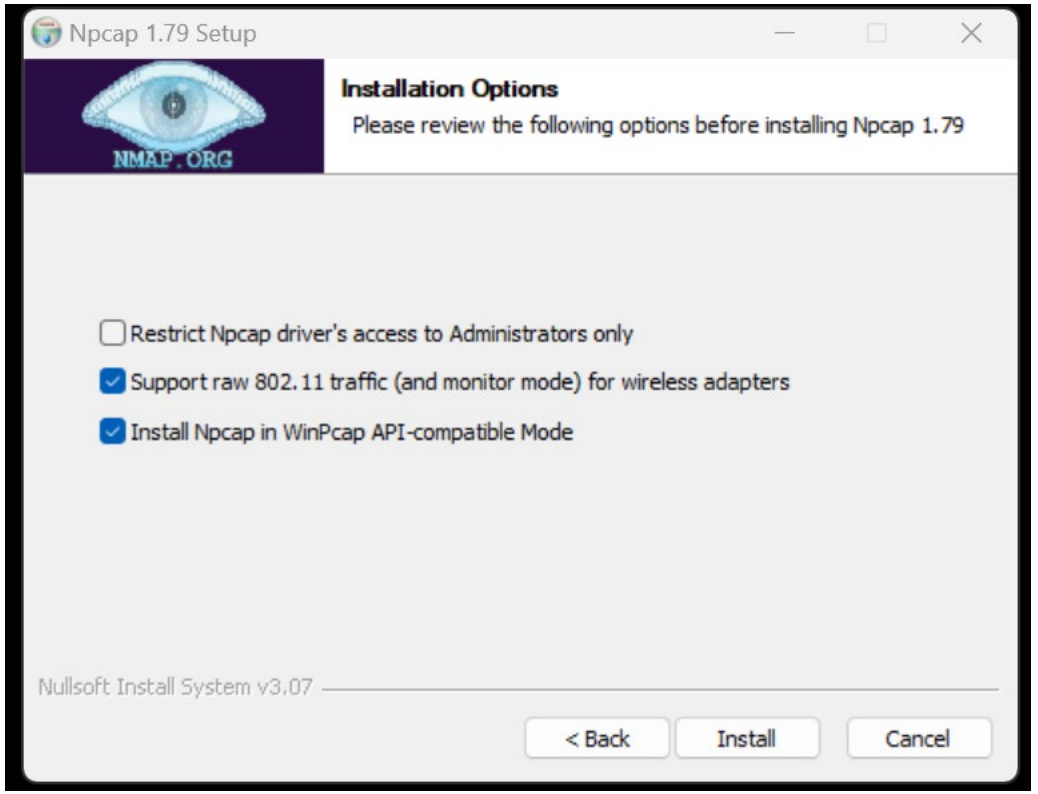
Promiscuous Mode

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



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,…

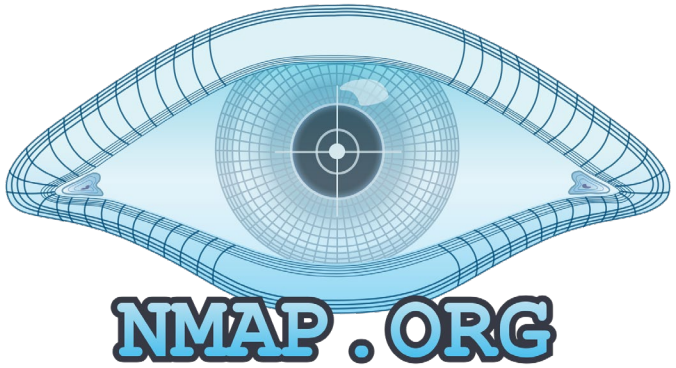
PCAP: Packet Capture API

- **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

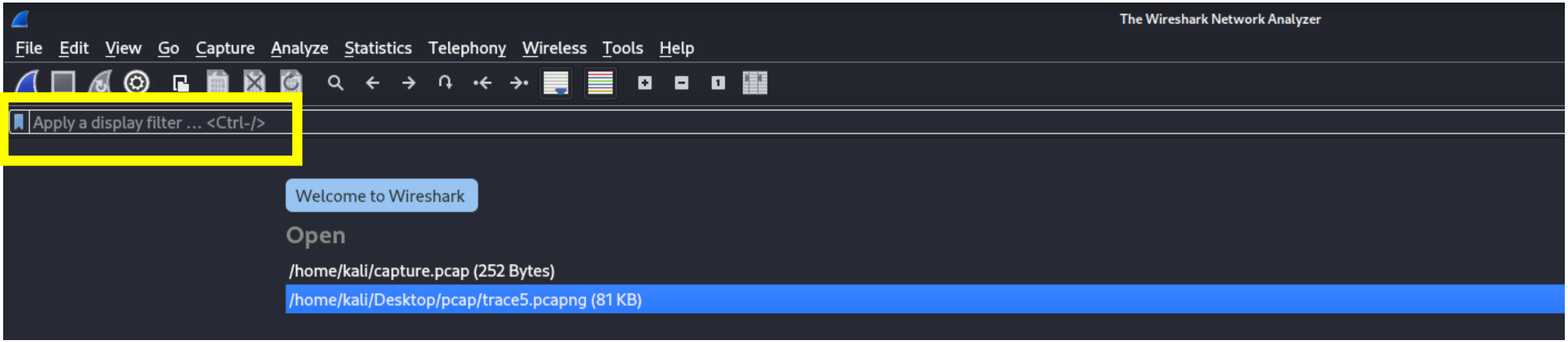
- WinPcap was eventually superseded by **Npcap**, a modern packet capture and network monitoring library for Windows, developed by the Nmap Project.

- Npcap offers better performance and support for Windows 7 through 11 and beyond.



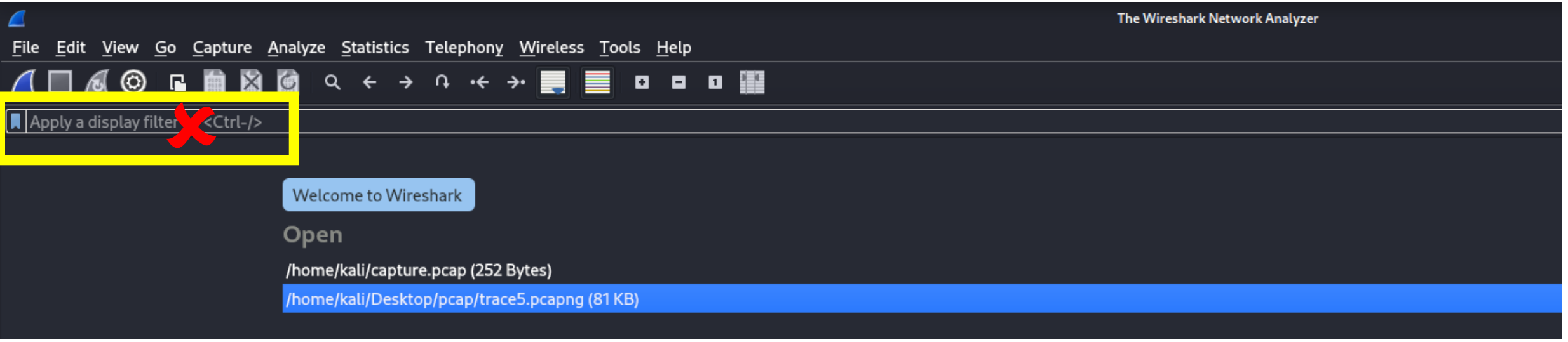
Filtering Out Unwanted Packets

- 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
```

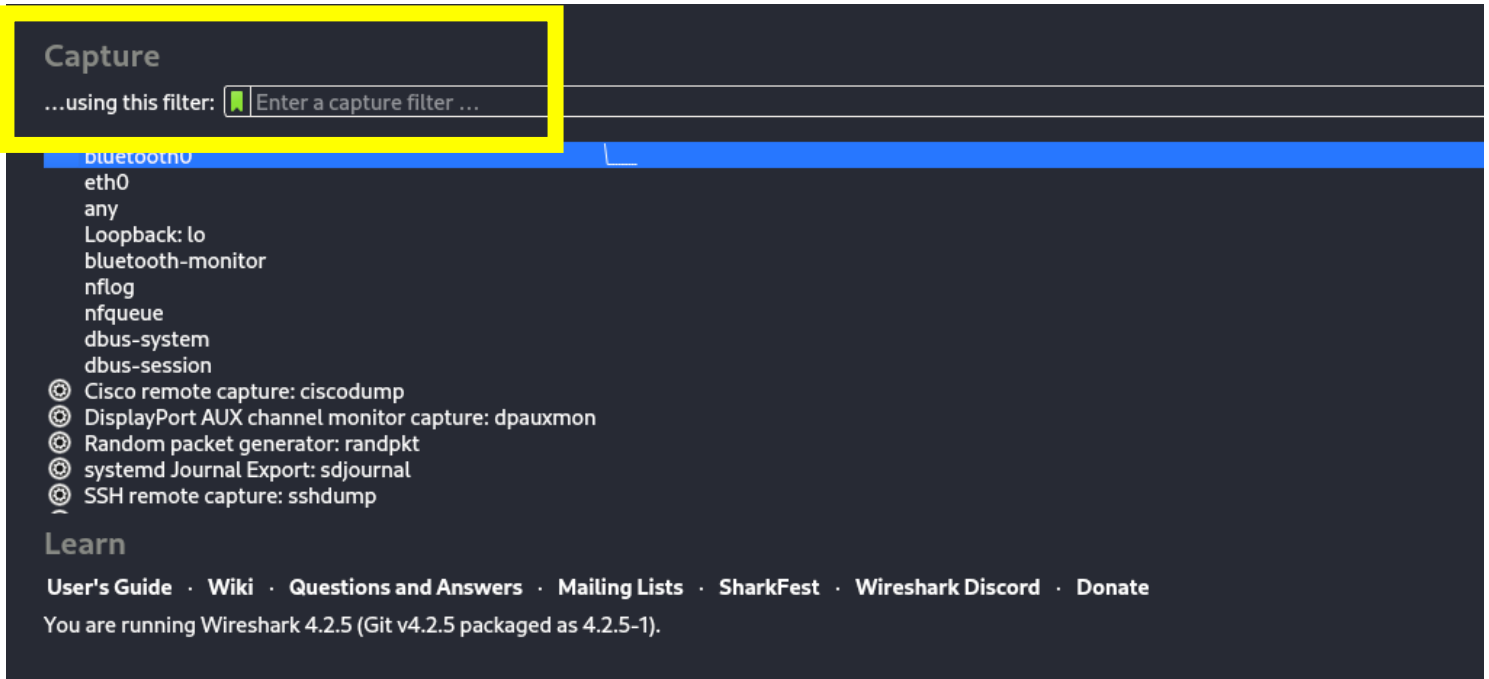
Capture traffic for a particular IP protocol:

```
ip proto protocol
```

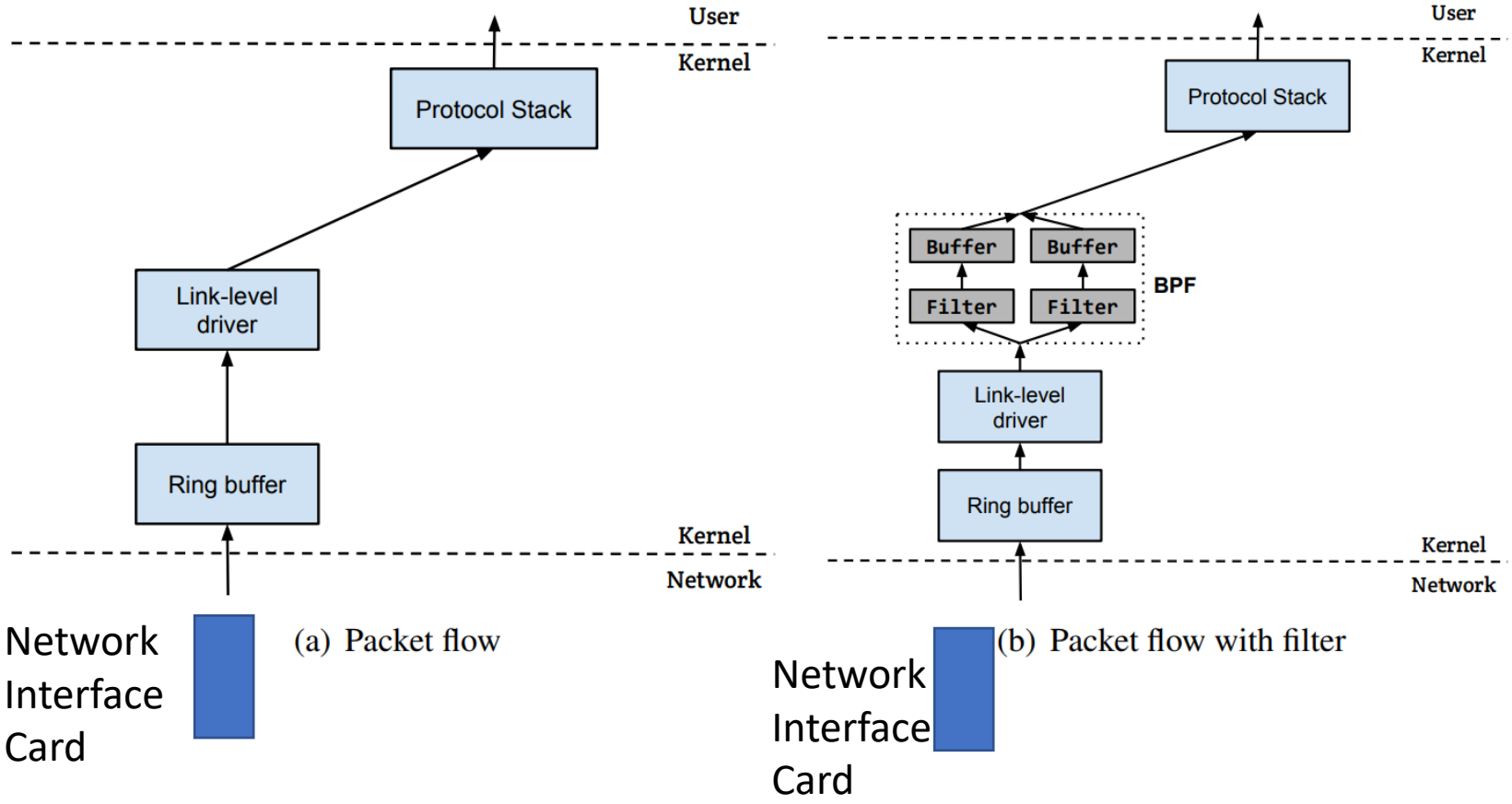
Image Source: https://www.ibm.com/docs/en/qsip/7.4?topic=queries-berkeley-packet-filters

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

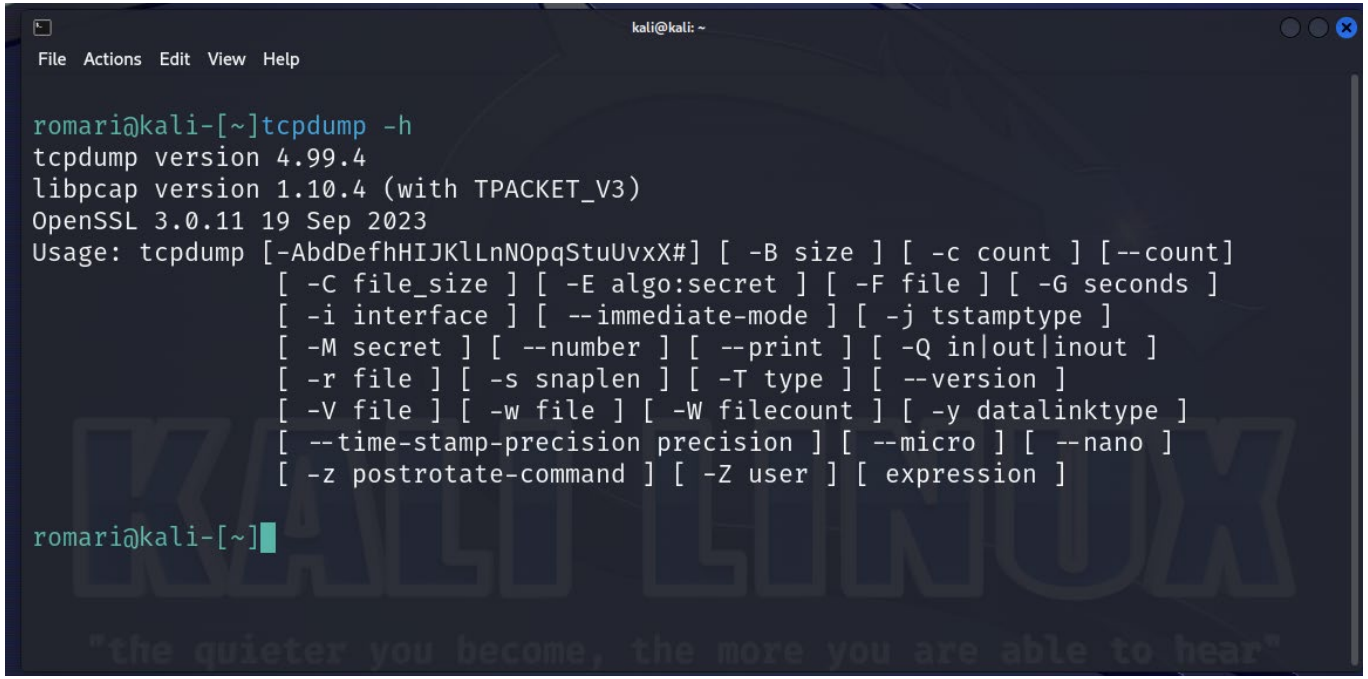
Tools of the Trade

- **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

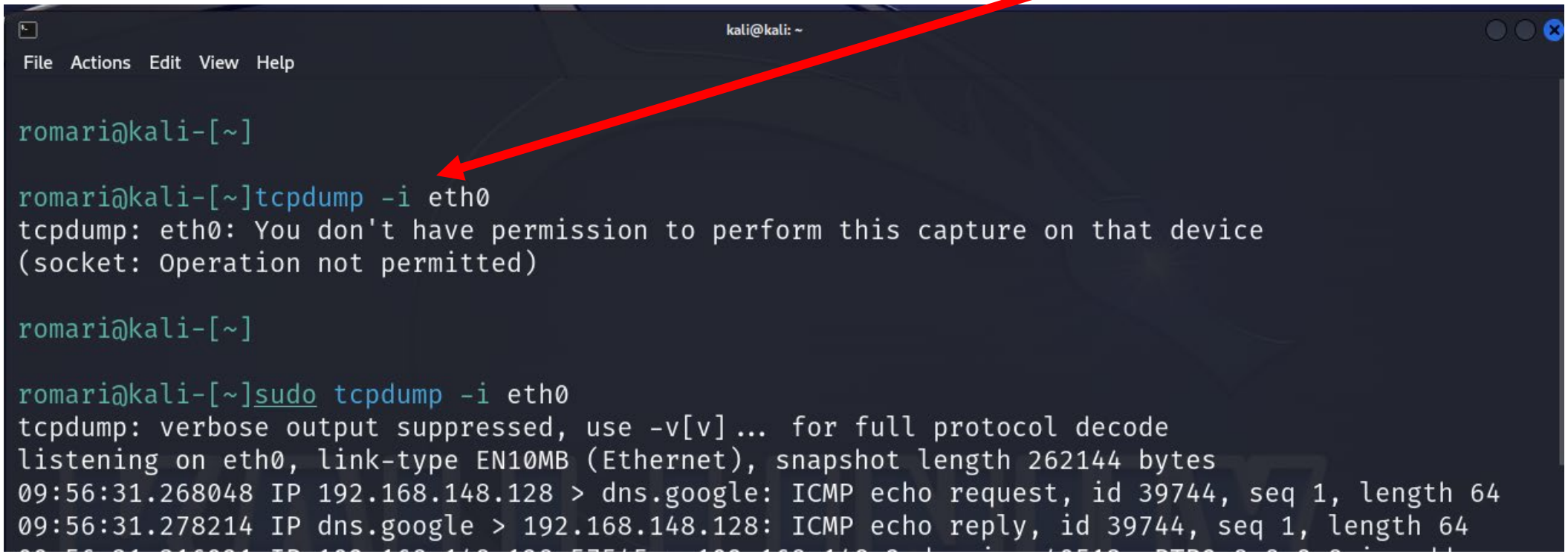
- Command line based packet sniffer.
  - Capture on an interface, or read **`pcap`** file.
- Ideal for quick capturing, integrating into scripts, and is lightweight.
- Simple with basic capabilities.



tcpdump

- `tcpdump -i eth0`

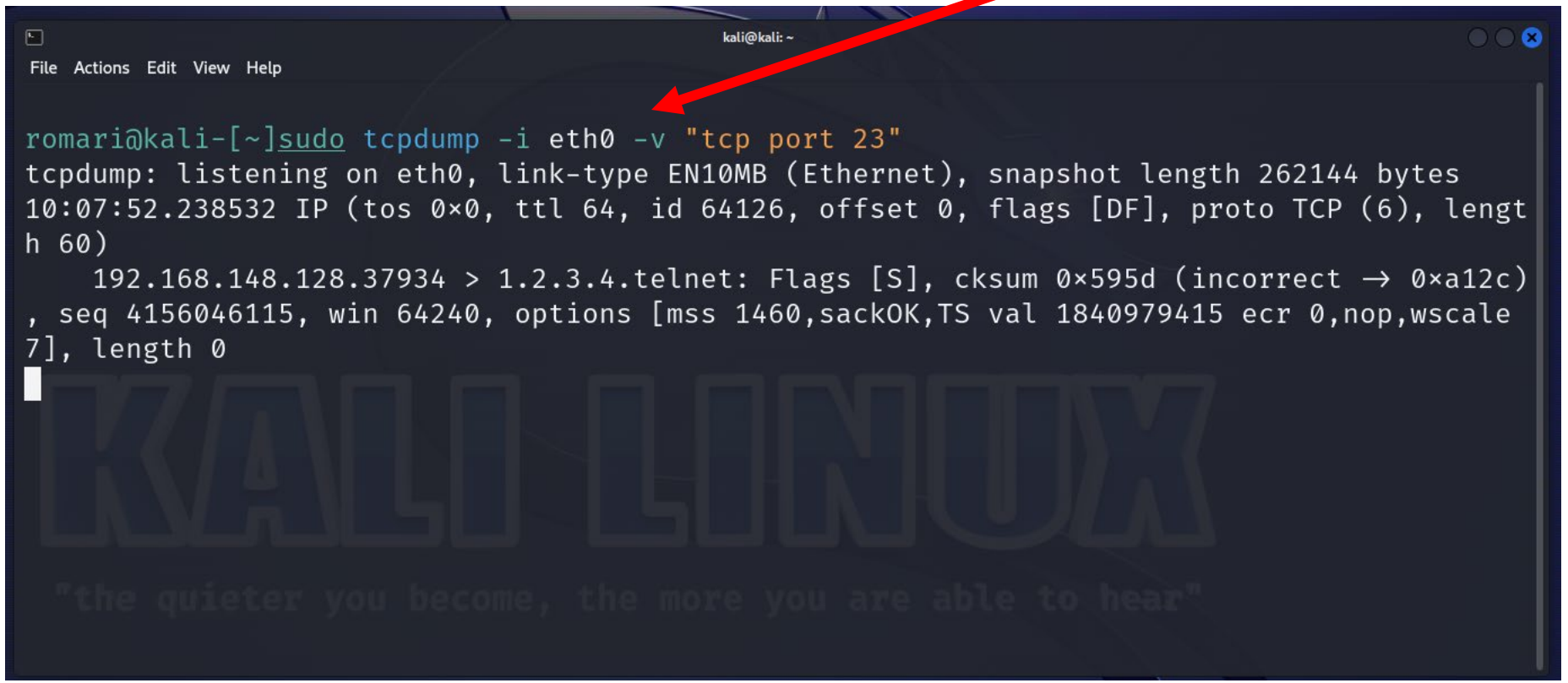
listen on interface eth0



tcpdump

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

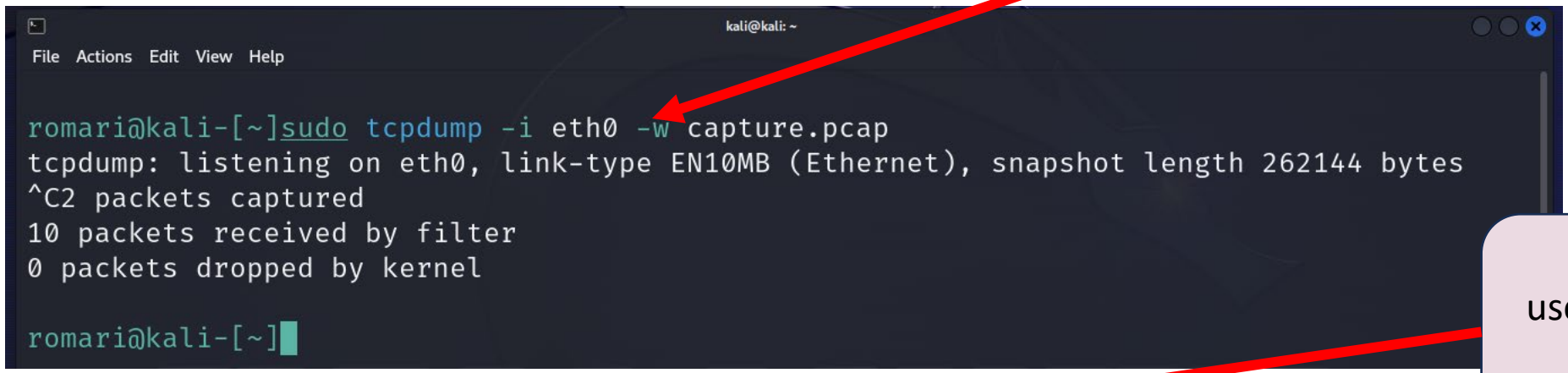
-v produce more verbose output.



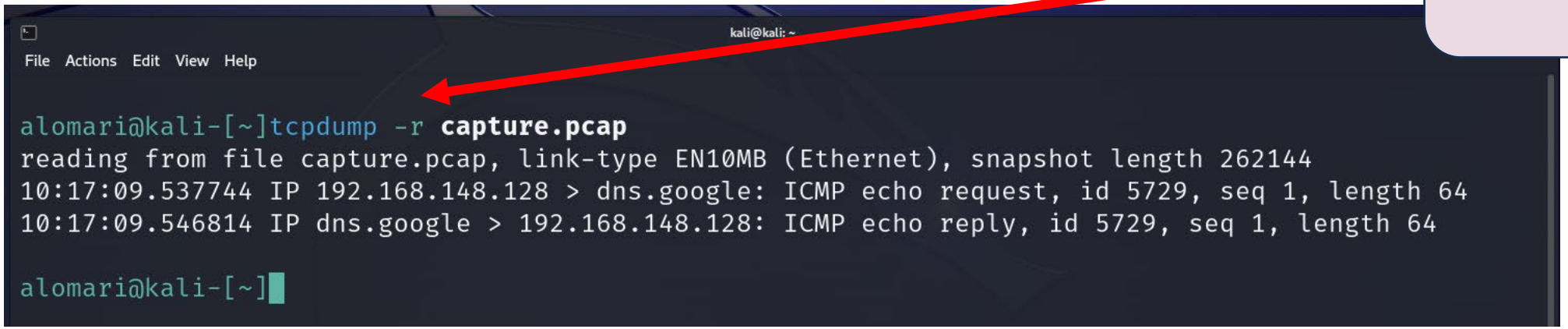
tcpdump

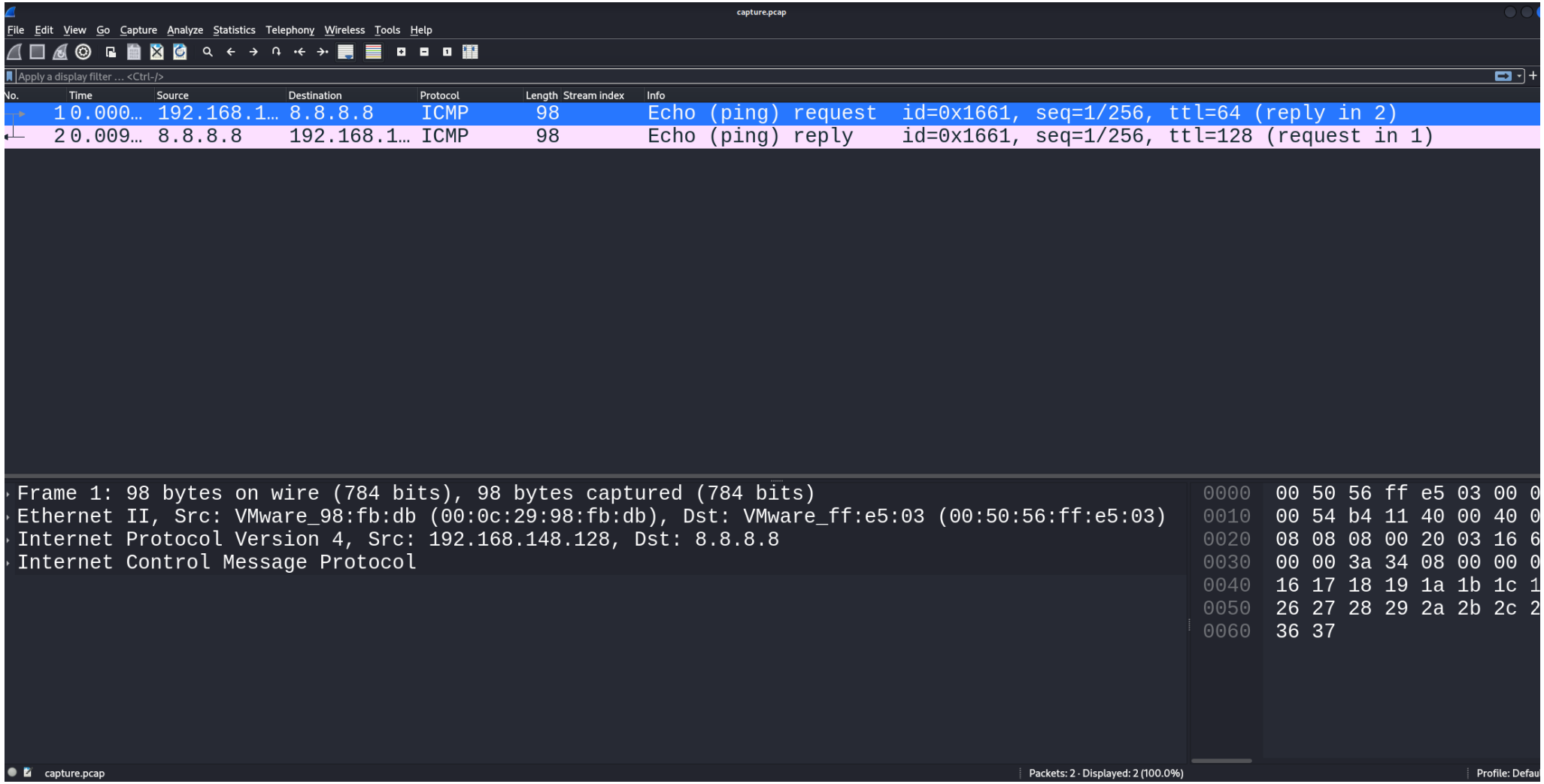
- `tcpdump -i eth0 -w capture.pcap`

-w capture packets and save them to a pcap file



use **-r** to read the file.




capture.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
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
capture.pcap
Packets: 2 · Displayed: 2 (100.0%)
Profile: Defau

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

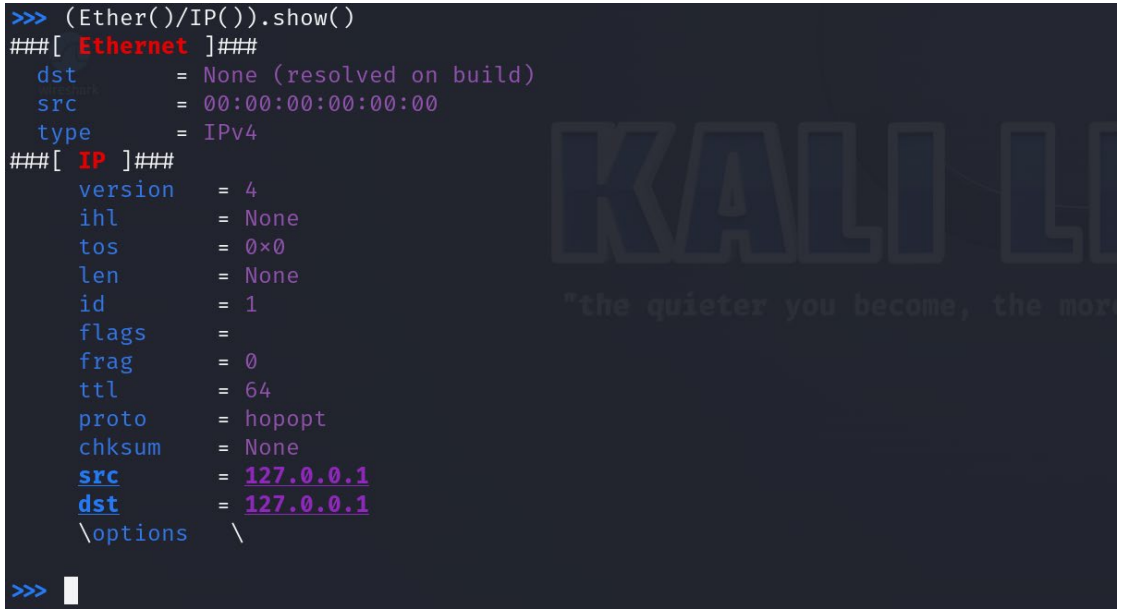
- Comes pre-installed in Kali.
  - And on SEED VM.

- Installation:

```
sudo pip3 install scapy
```

- Import Scapy Module in Python Program:

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



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")

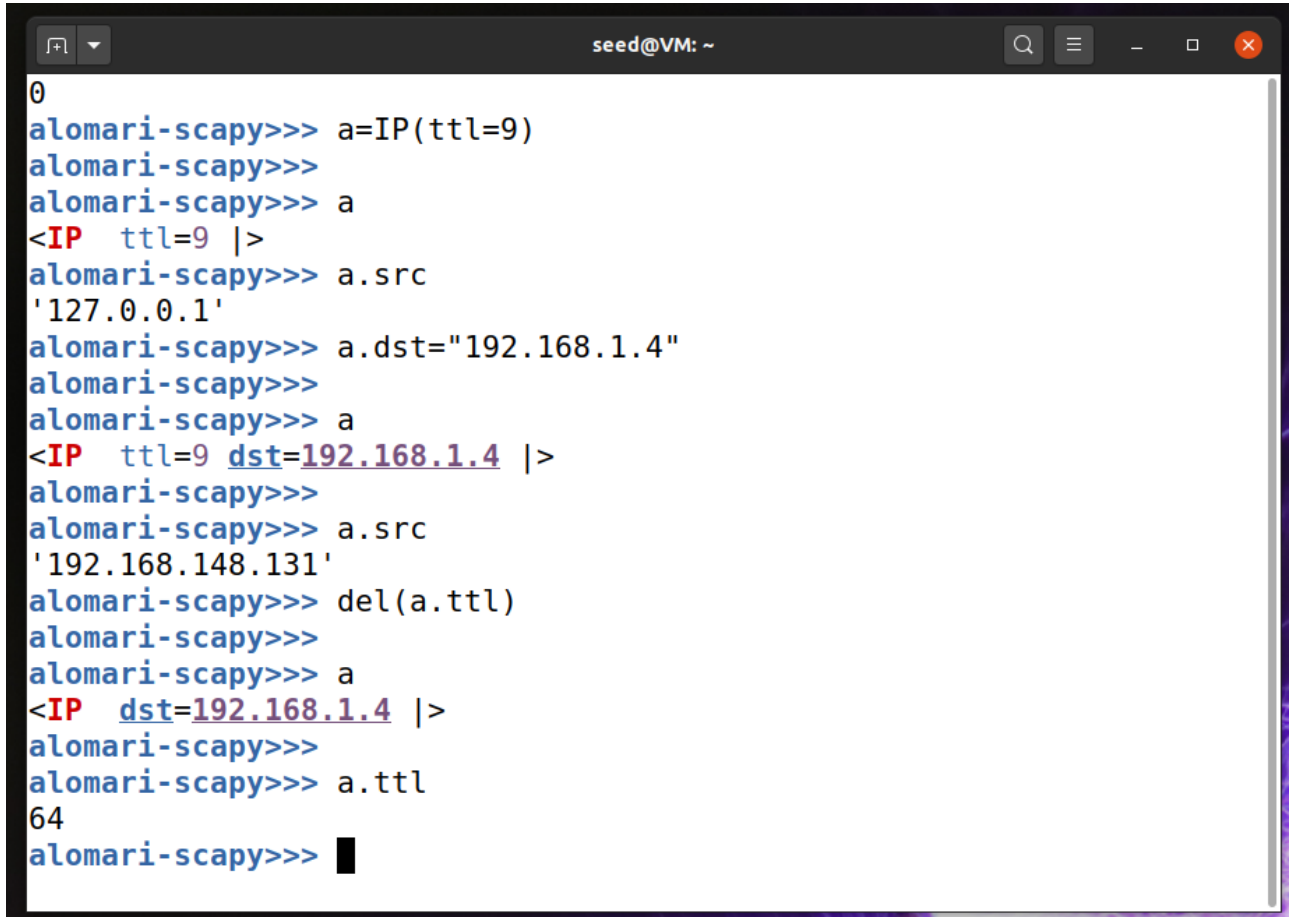
ARP Manipulation

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

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

First Steps

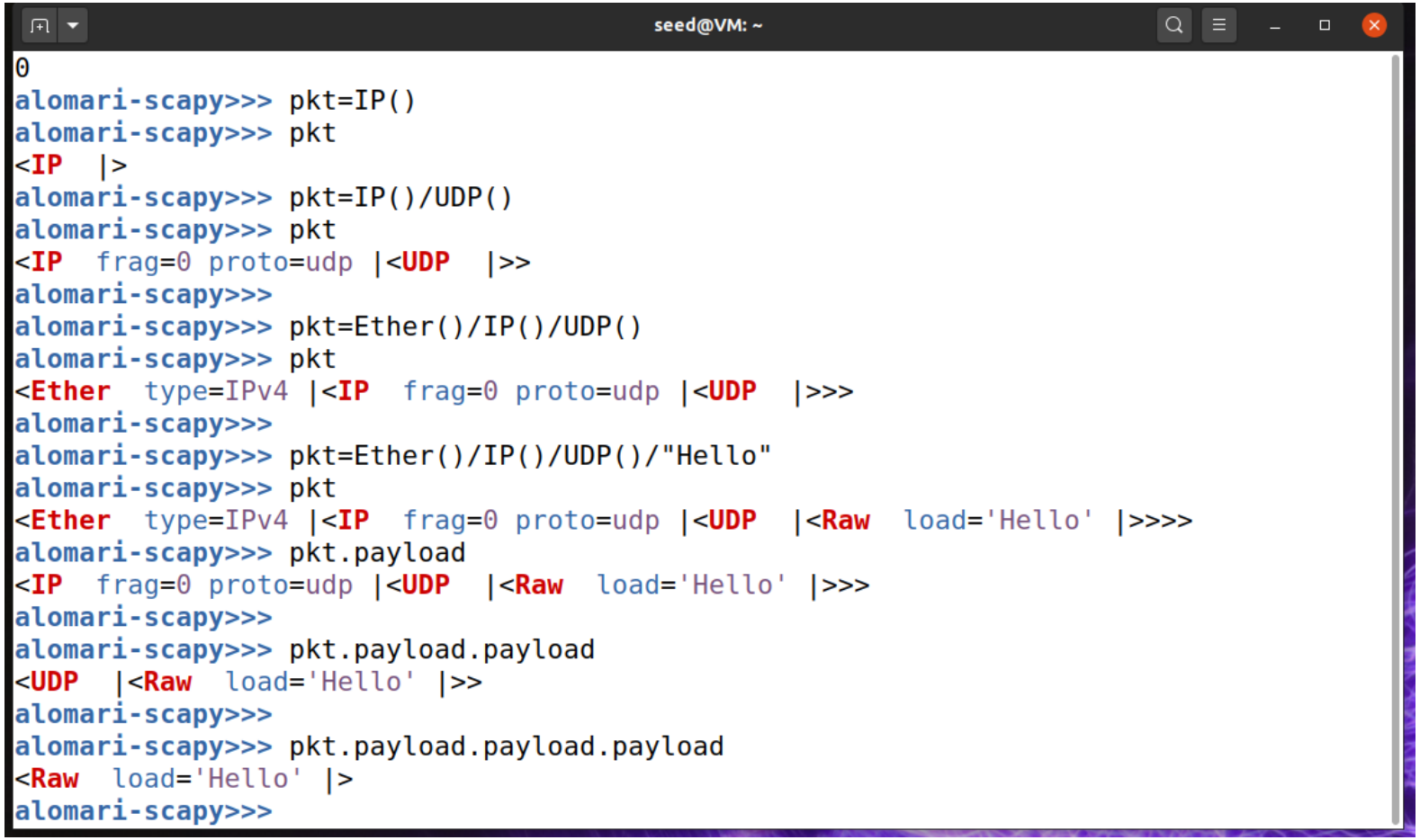
- Let's build a packet and play with it.



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Stacking Layers

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

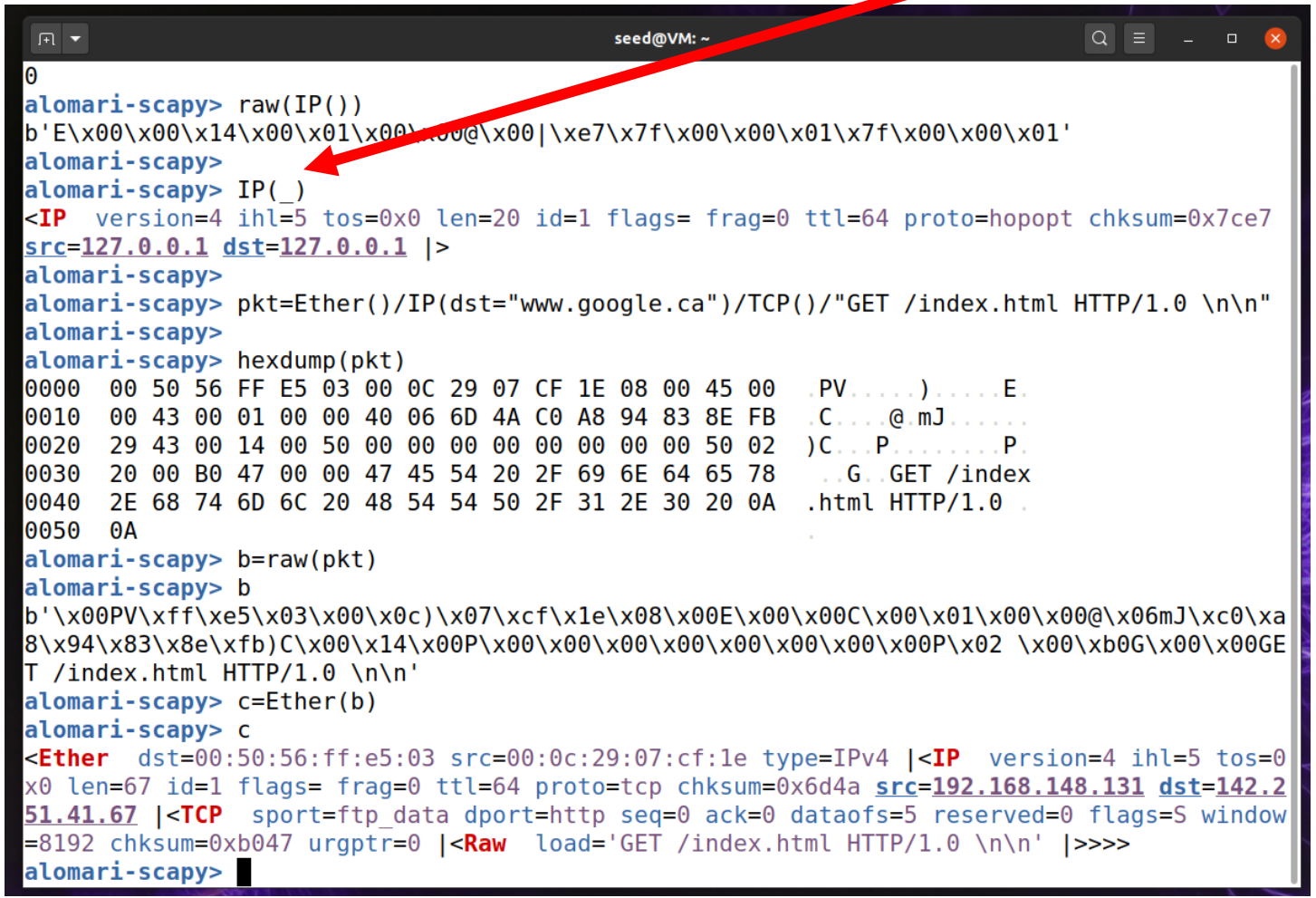


https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

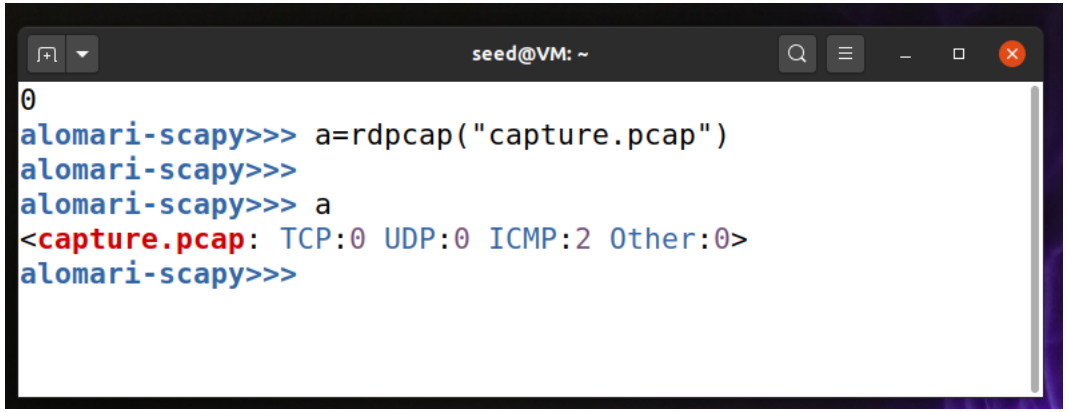
Stacking Layers

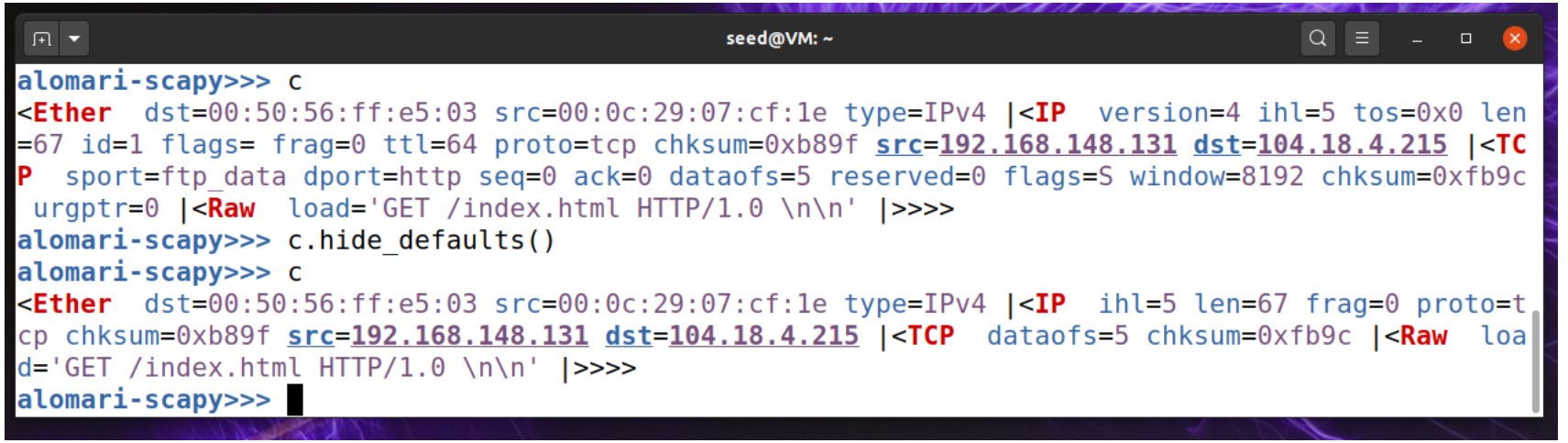
- Each packet can be built or dissected.

in python _ is the latest result



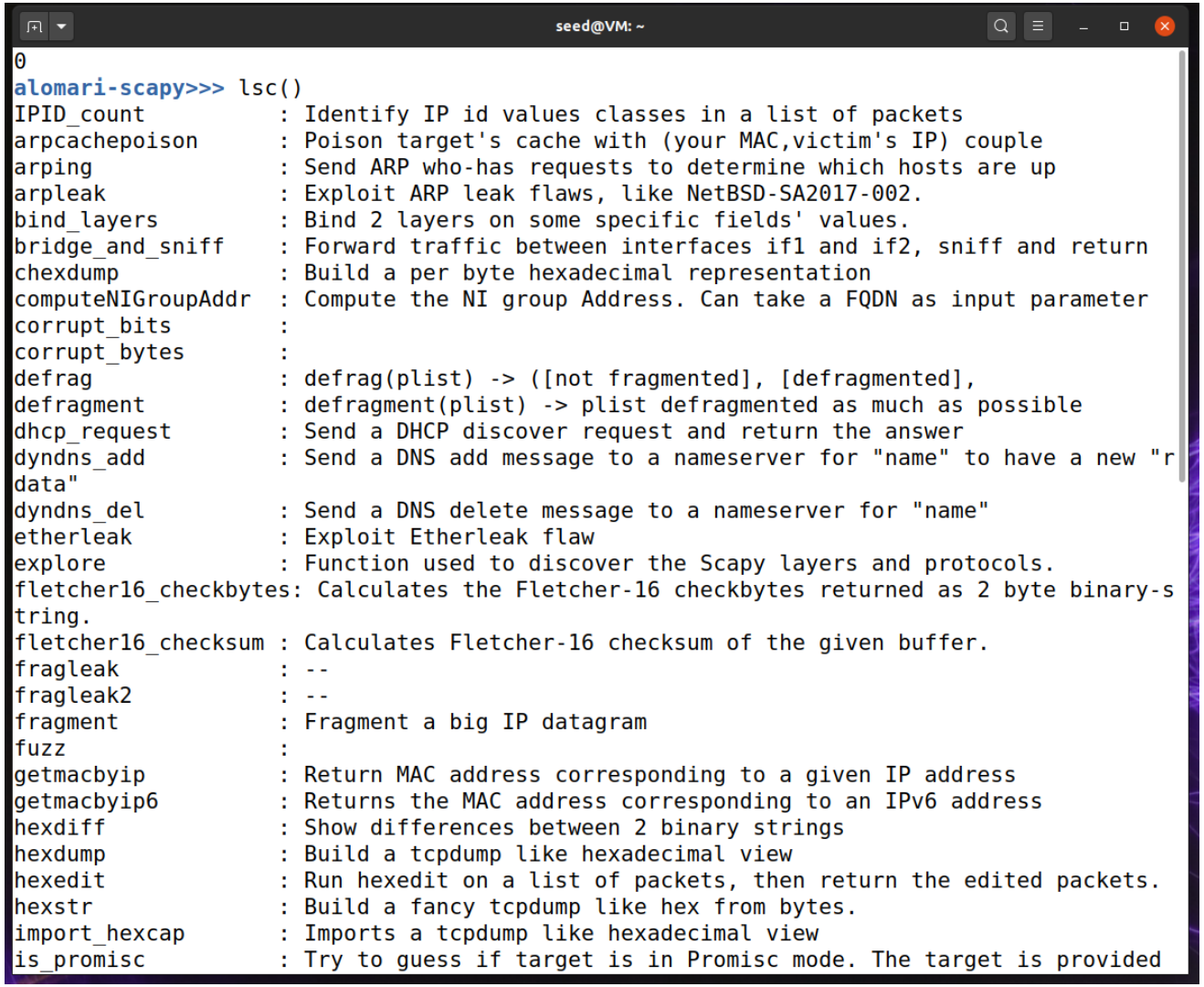
rdpcap() & hide_defaults()





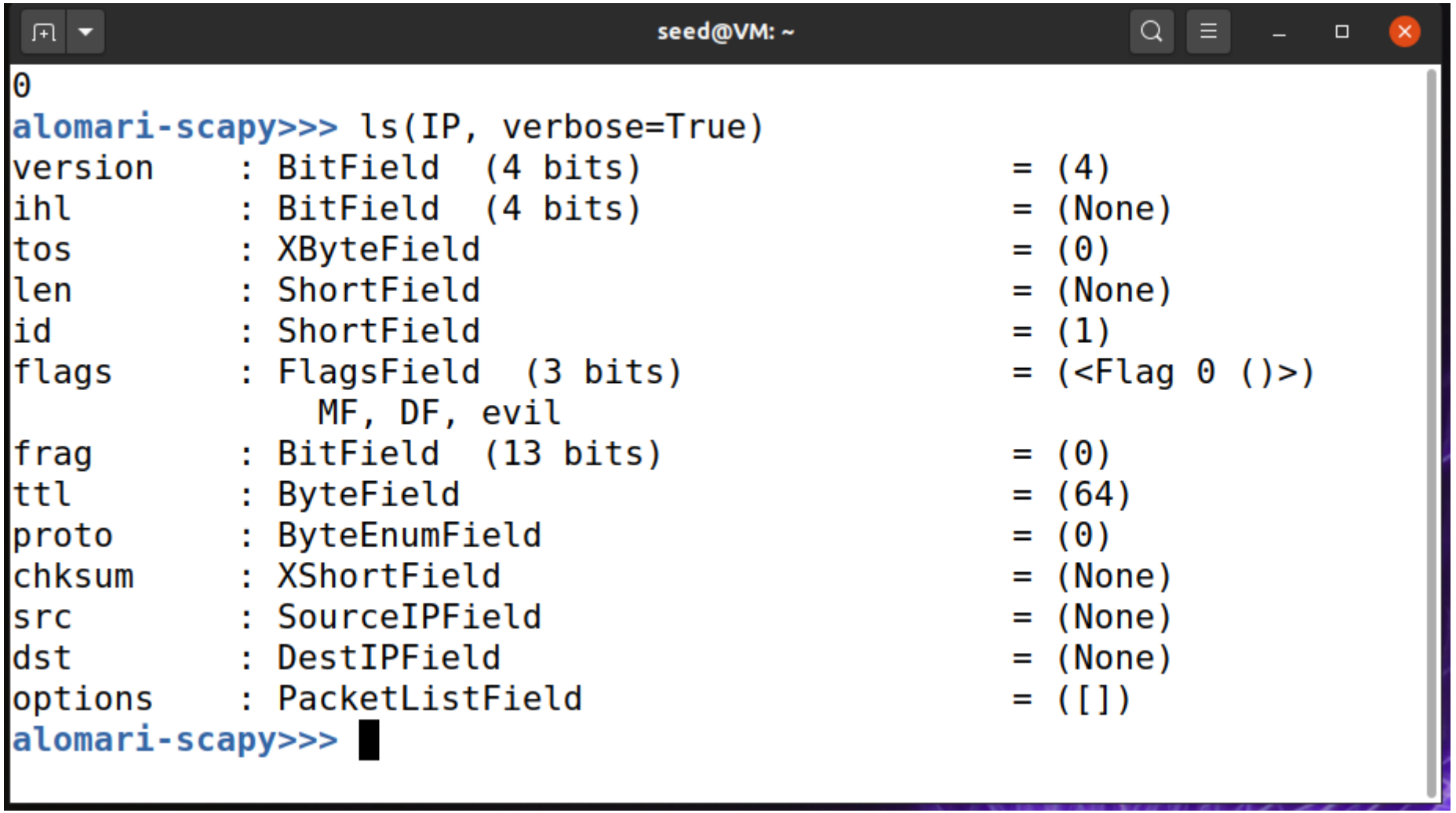
https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

lsc() function lists available commands



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

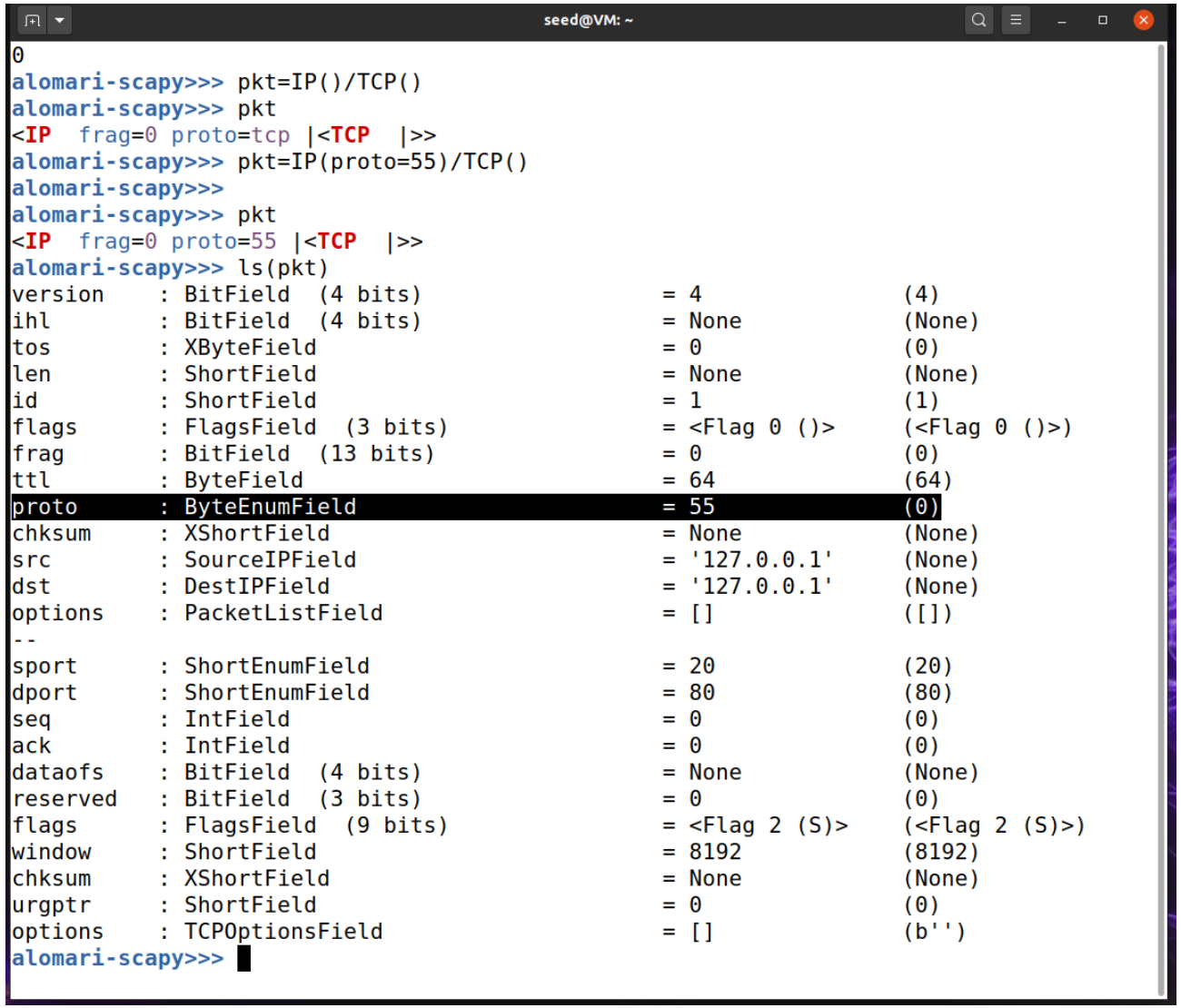
ls() function lists fields of a protocol



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Packet Manipulation

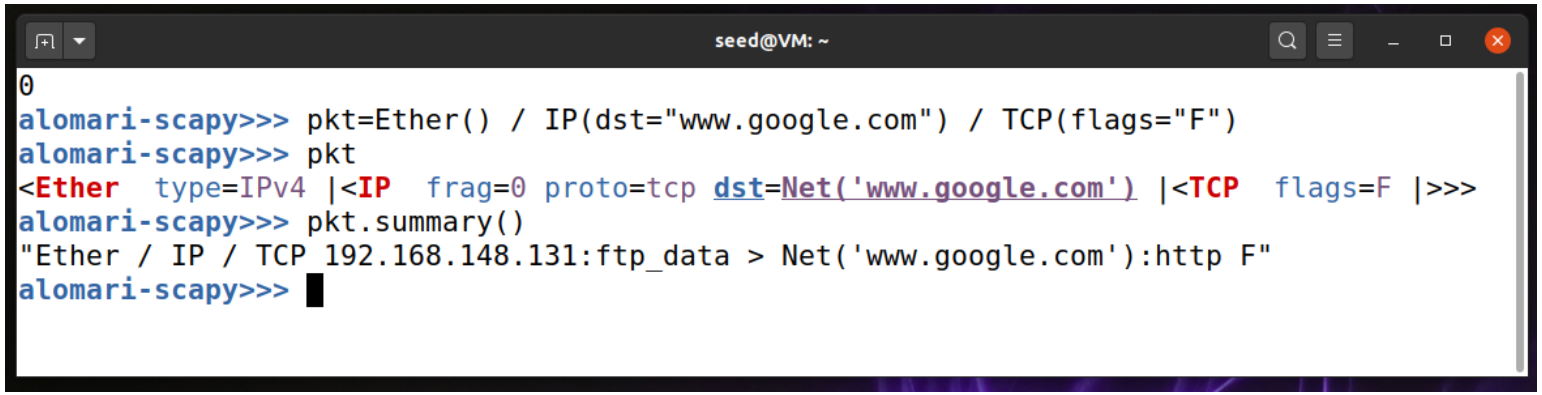
- You can set any field.



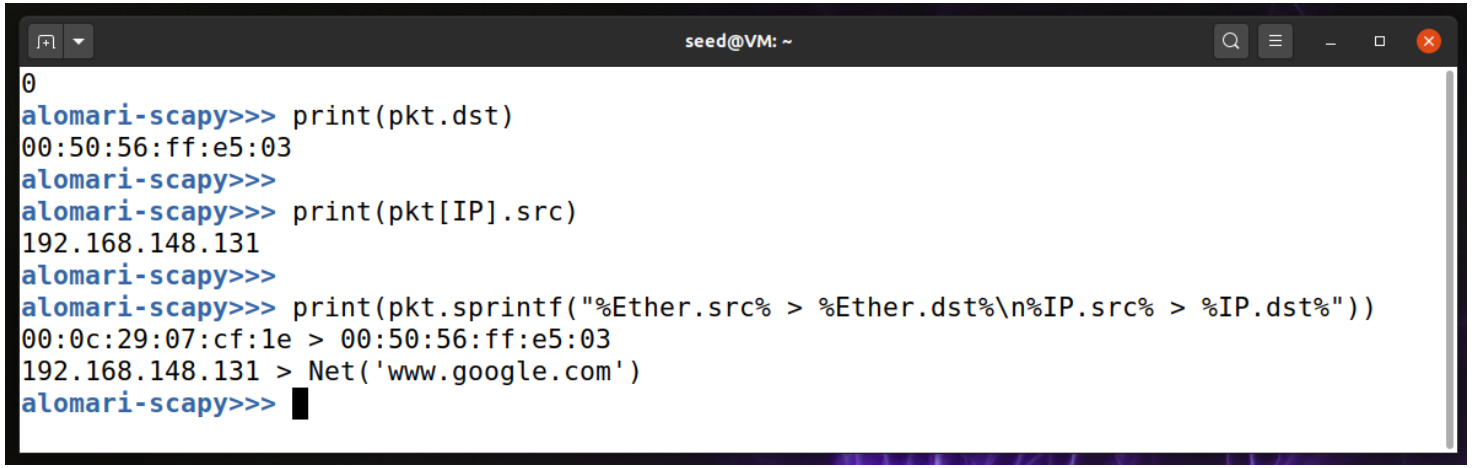
https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Packet Manipulation

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



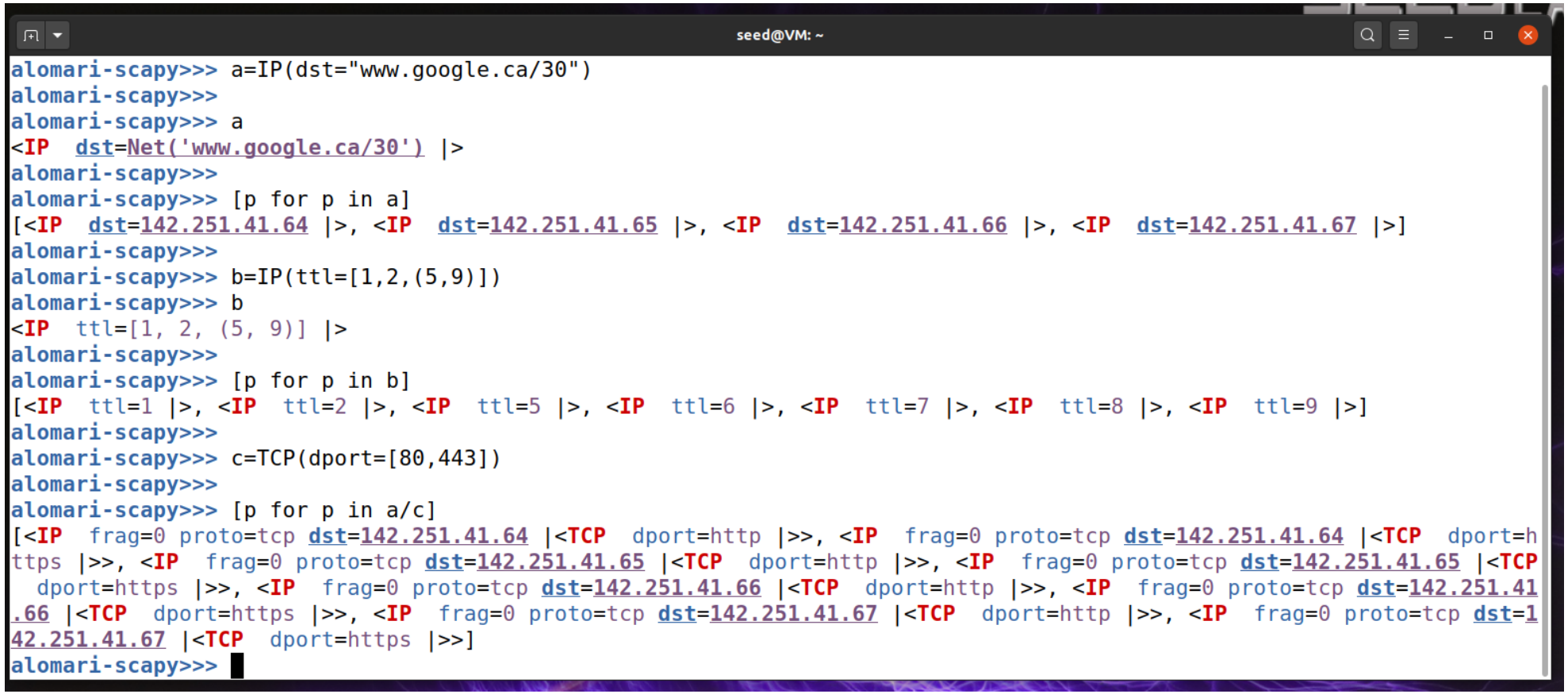
- All fields can be easily accessed



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Generating Sets of Packets

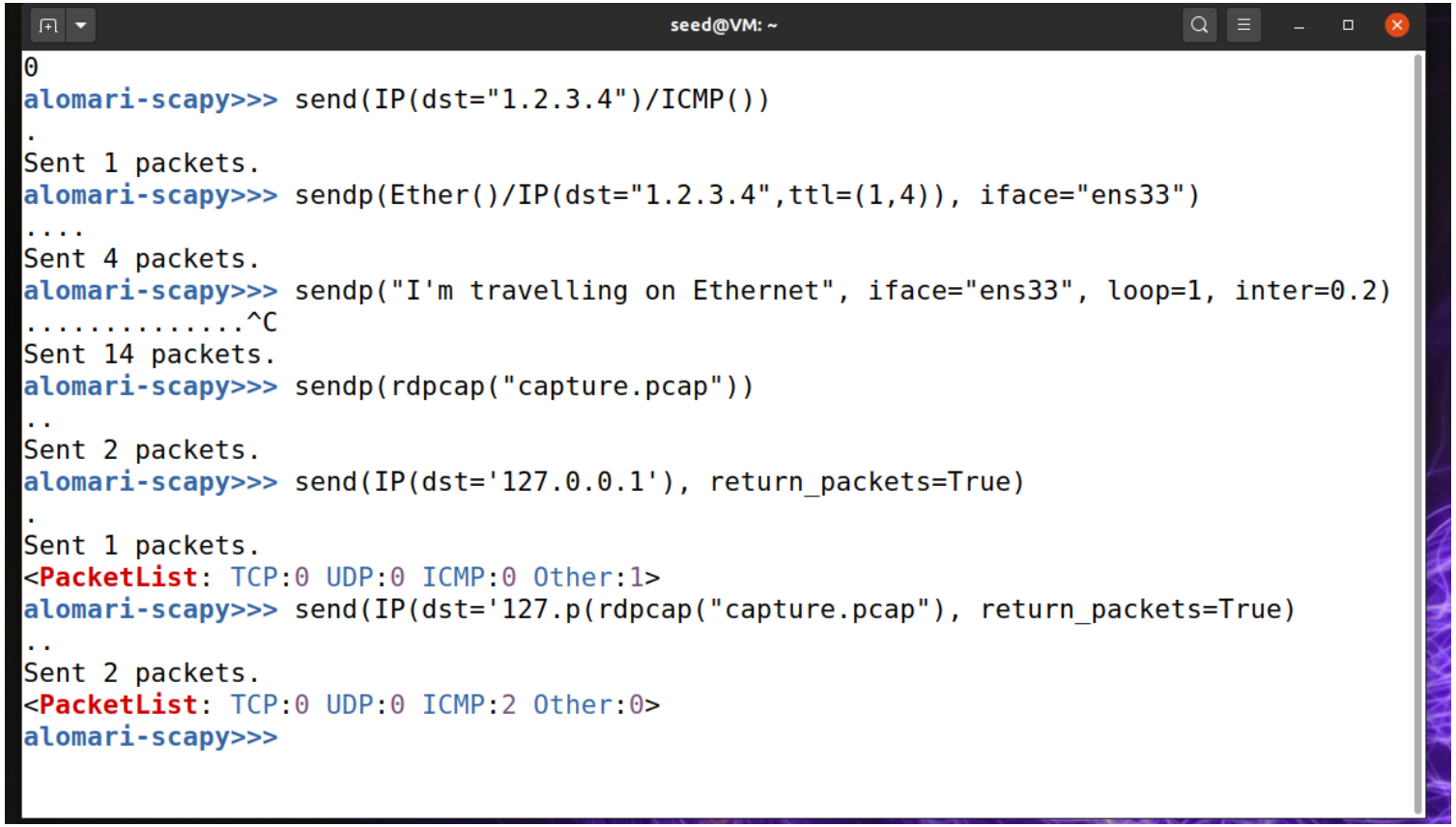
- A field can store many value.



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

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.



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Send and Receive Packets (sr)

- **`.show()`**

https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Send and Receive Packets (sr)

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

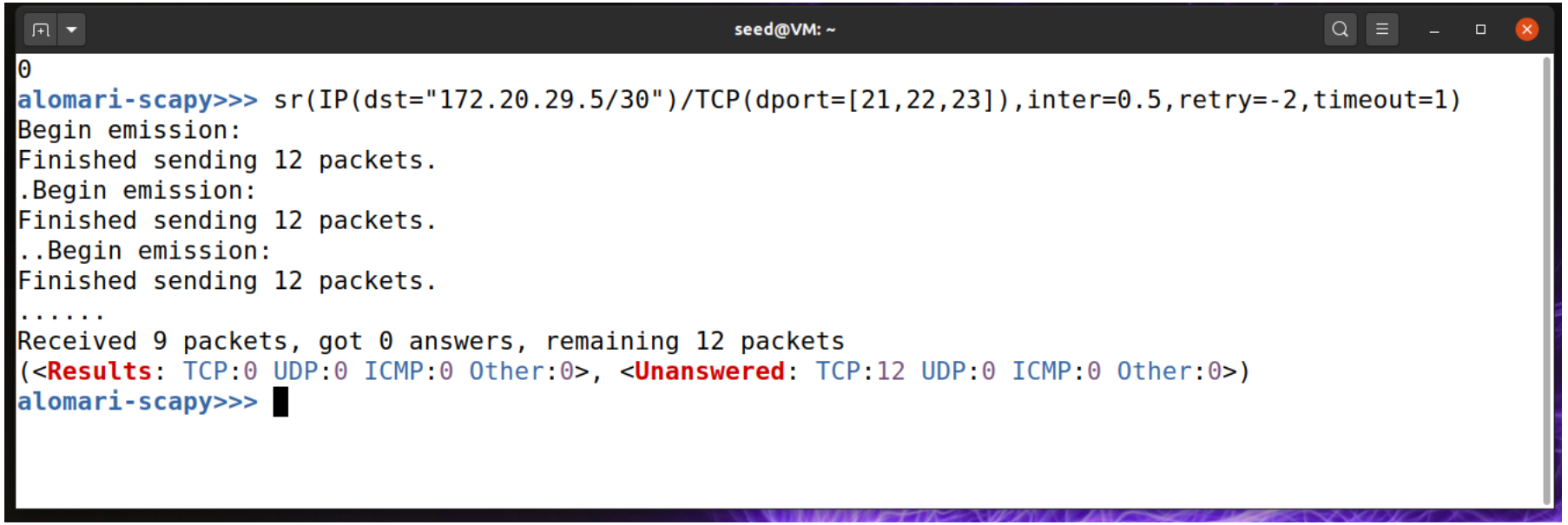
```
seed@VM: ~
0
alomari-scapy>>> sr(IP(dst="8.8.8.8")/TCP(dport=[21,22,23]))
Begin emission:
Finished sending 3 packets.
^C
Received 0 packets, got 0 answers, remaining 3 packets
(<Results: TCP:0 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:3 UDP:0 ICMP:0 Other:0>)
alomari-scapy>>> unans.summary()
IP / TCP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

https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

Send and Receive Packets (sr)

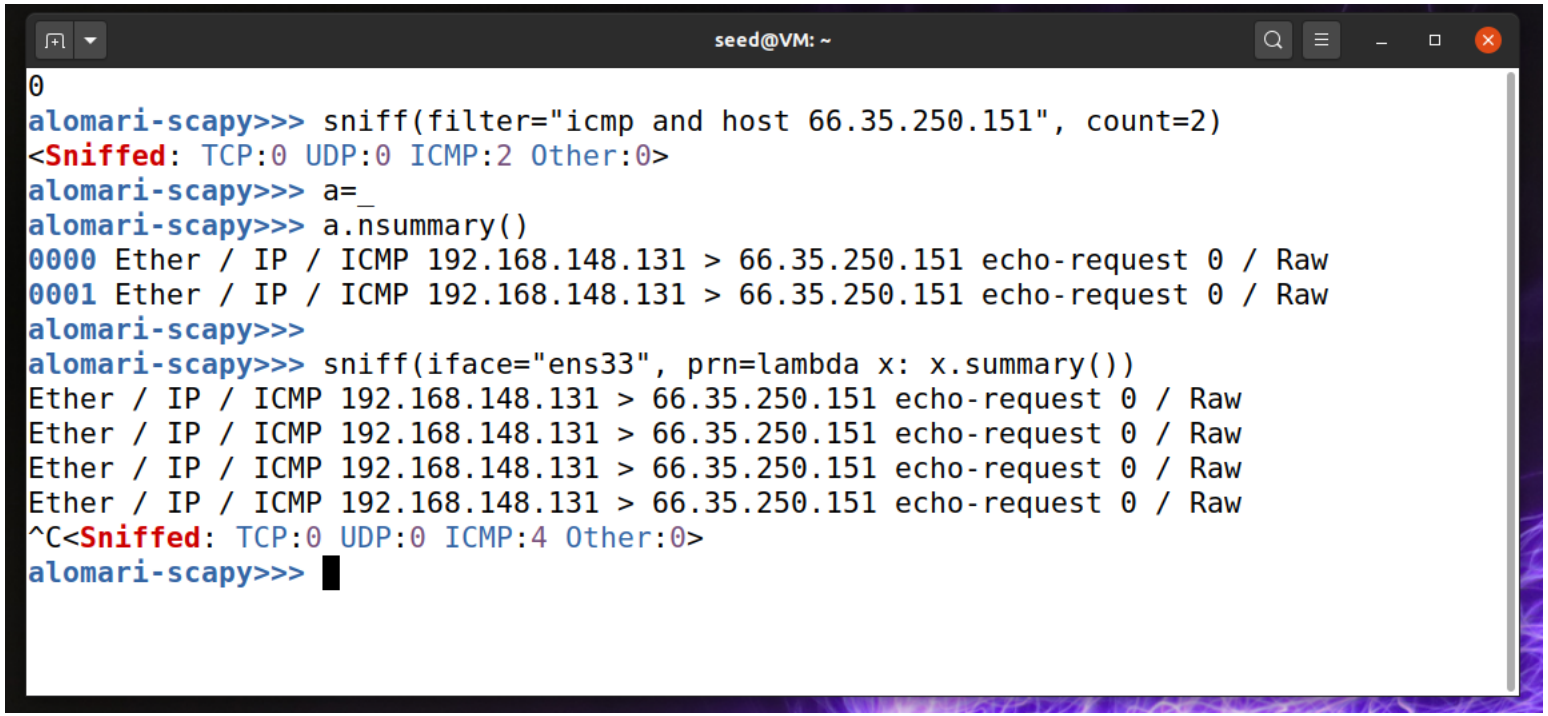
- If there is a limited rate of answers, you can specify a time interval (in seconds) to wait between two packets with the inter parameter.



https://scapy.readthedocs.io/en/latest/usage.html#interactive-tutorial

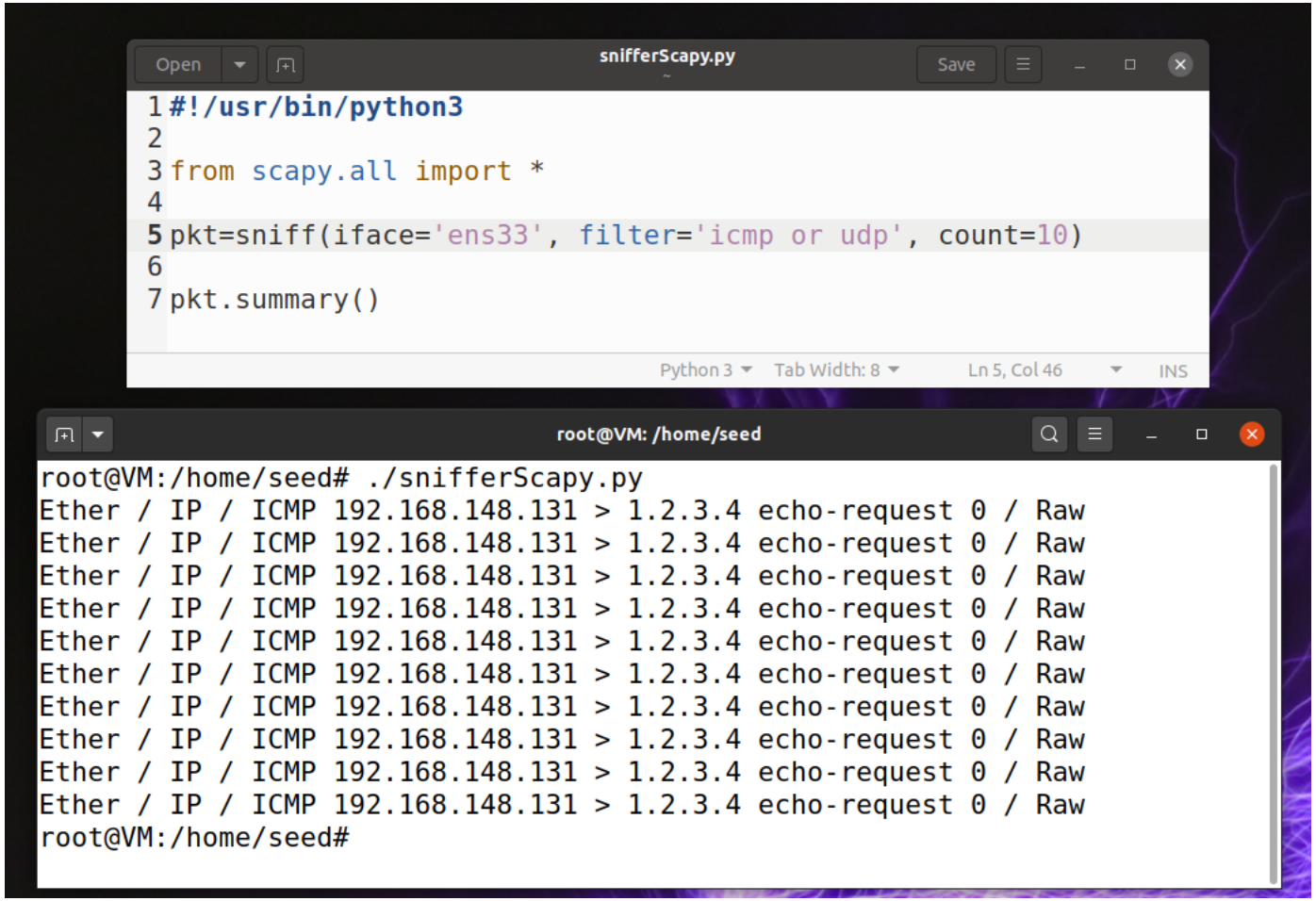
Sniffing

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

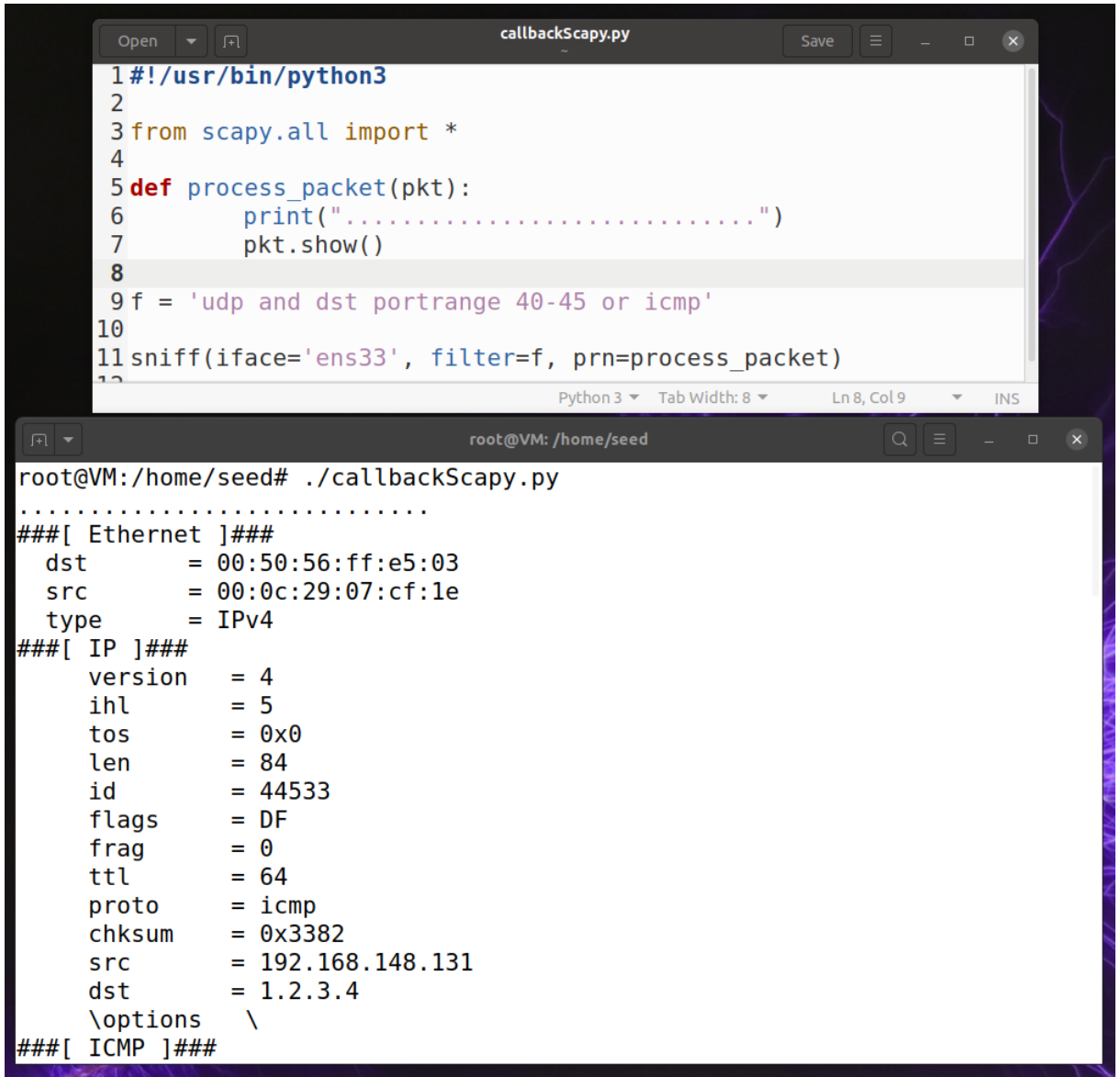


Command	Effect
raw(pkt)	assemble the packet
hexdump(pkt)	have a hexadecimal dump
ls(pkt)	have the list of fields values
pkt.summary()	for a one-line summary
pkt.show()	for a developed view of the packet
pkt.show2()	same as show but on the assembled packet (checksum is calculated
pkt.sprintf()	fills a 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



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?

sniffSpoofScapy.py

```
#!/usr/bin/python3
from scapy.all import *
# Define a callback function to handle sniffed packets
def packet_callback(packet):
    if packet.haslayer(ICMP) and packet[ICMP].type==8 :
        # Extracting necessary information from the original packet
        original_src = packet[IP].src
        original_dst = packet[IP].dst
        original_seq = packet[ICMP].seq
        original_id = packet[ICMP].id

        # Print original packet information
        print("Sniffed Packet - Source: {}, Destination: {}".format(original_src, original_dst))

        # Constructing the spoofed ICMP packet
        spoofed_pkt = IP(src=original_dst, dst=original_src) / ICMP(type="echo-reply", id=original_id, seq=original_seq) / "Spoofed Data"

        # Print spoofed packet information
        print("Spoofed Packet - Source: {}, Destination: {}".format(spoofed_pkt[IP].src, spoofed_pkt[IP].dst))

        # Sending the spoofed packet
        send(spoofed_pkt, verbose=False)

sniff(filter="icmp", prn=packet_callback)
```

root@VM: /home/seed

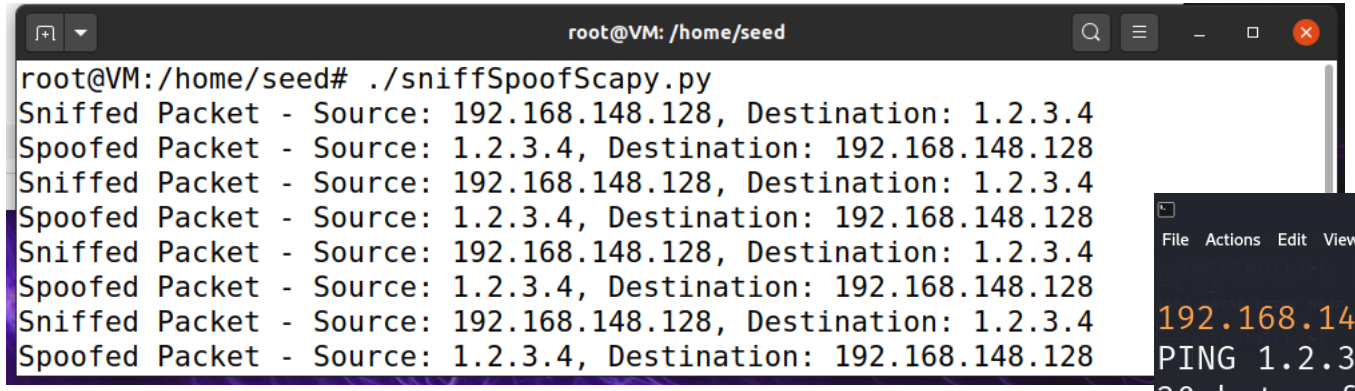
```
root@VM:/home/seed# ./sniffSpoofScapy.py
Sniffed Packet - Source: 192.168.148.128, Destination: 8.8.8.8
Spoofed Packet - Source: 8.8.8.8, Destination: 192.168.148.128
Sniffed Packet - Source: 192.168.148.128, Destination: 8.8.8.8
Spoofed Packet - Source: 8.8.8.8, Destination: 192.168.148.128
Sniffed 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
```

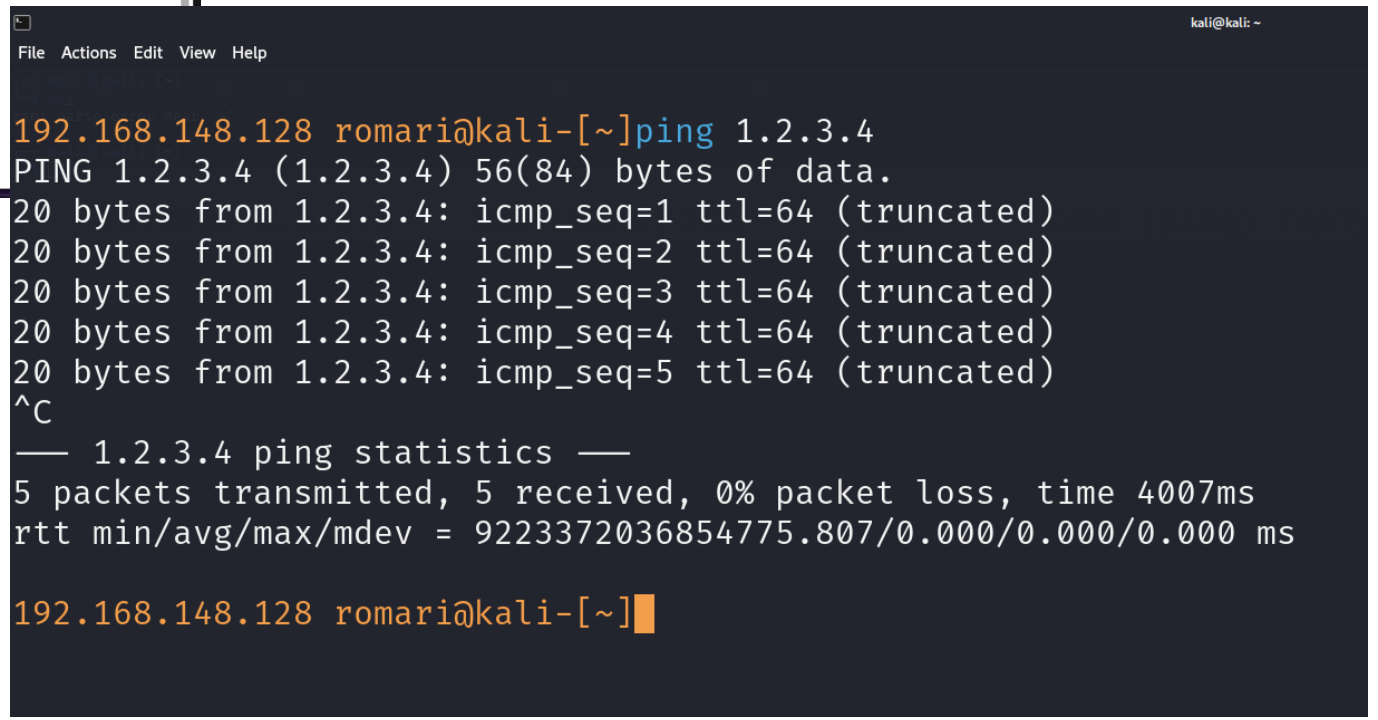
kali@kali: ~

```
File  Actions  Edit  View  Help

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)
^C
── 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

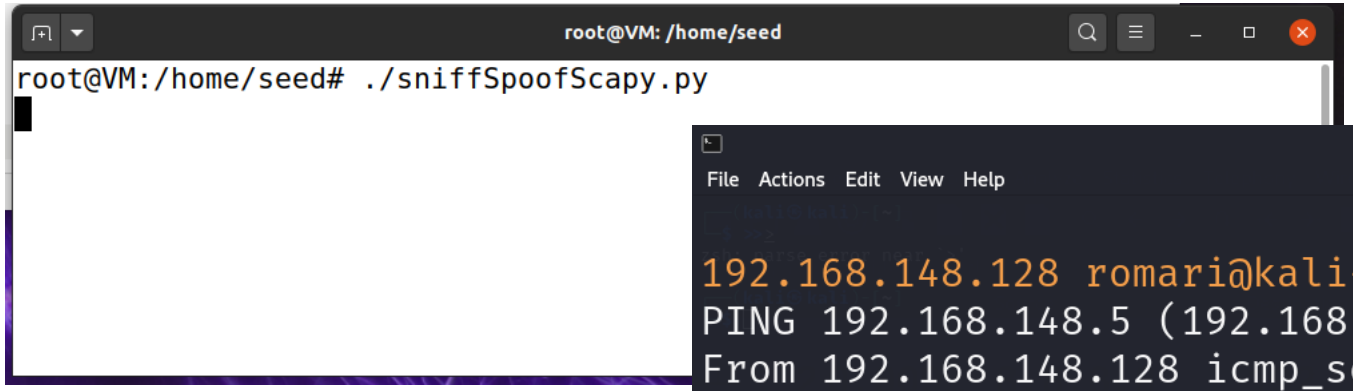
- **Question:** How are we getting replies when 1.2.3.4 doesn't exist?

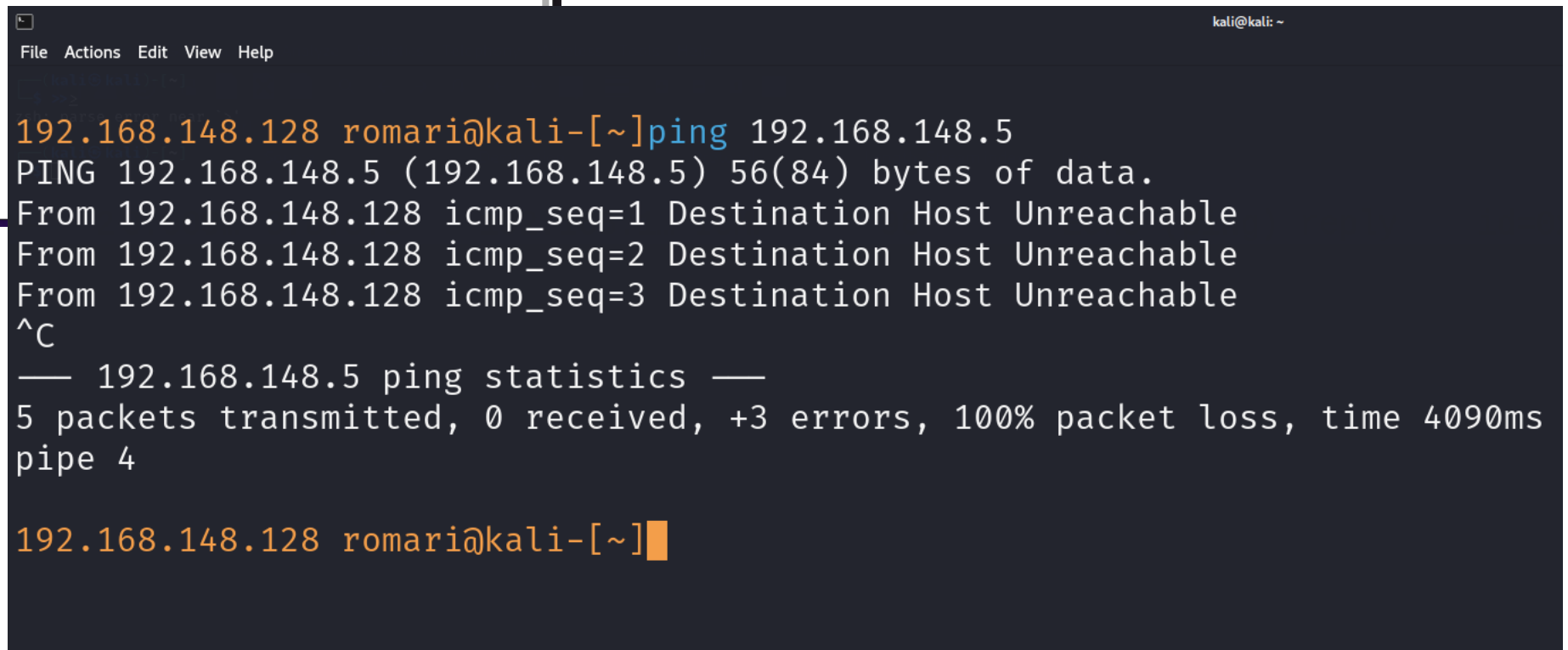




3- Sniff Then Spoof – Host on local network that doesn't exist – e.g., 192.168.148.5

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



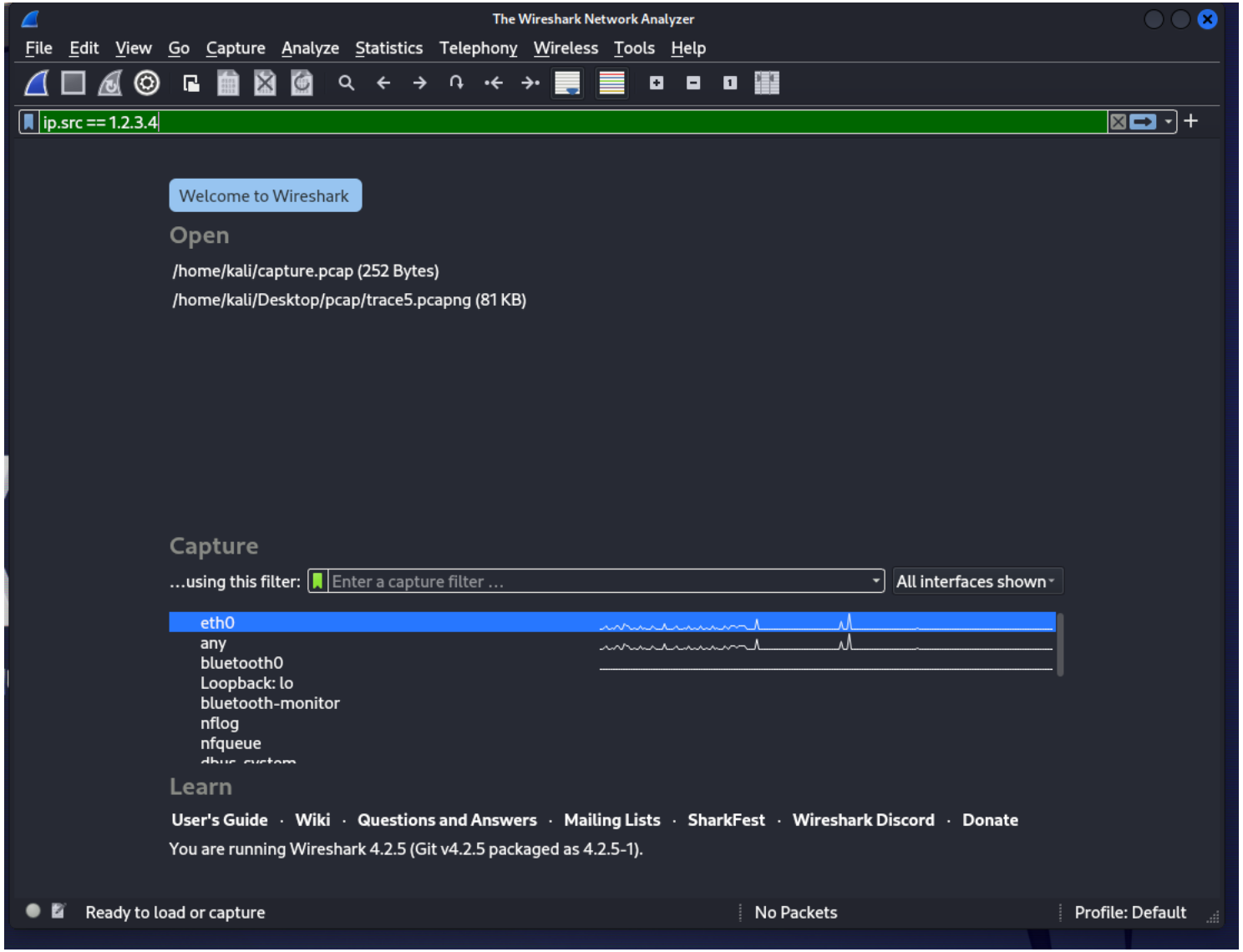


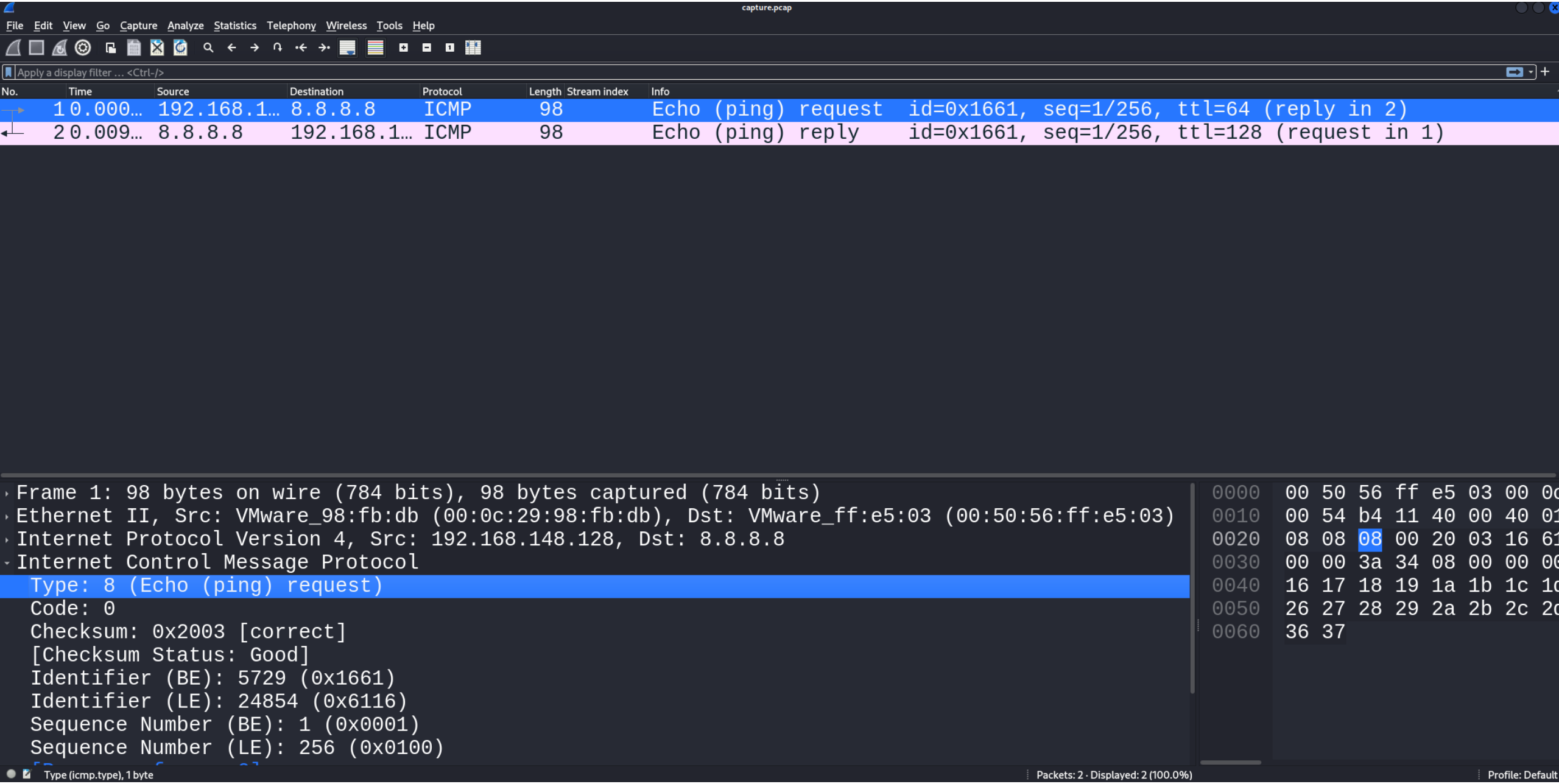
Wireshark

- 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 Wireshark Network Analyzer
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ip.src == 1.2.3.4
Welcome to Wireshark
Open
/home/kali/capture.pcap (252 Bytes)
/home/kali/Desktop/pcap/trace5.pcapng (81 KB)
Capture
...using this filter:
Enter a capture filter ...
All interfaces shown
eth0
any
bluetooth0
Loopback: lo
bluetooth-monitor
nflog
nfqueue
Learn
User's Guide · Wiki · Questions and Answers · Mailing Lists · SharkFest · Wireshark Discord · Donate
You are running Wireshark 4.2.5 (Git v4.2.5 packaged as 4.2.5-1).
Ready to load or capture
No Packets
Profile: Default


capture.pcap
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
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
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)
0000 00 50 56 ff e5 03 00 0c
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
Type (icmp.type), 1 byte
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

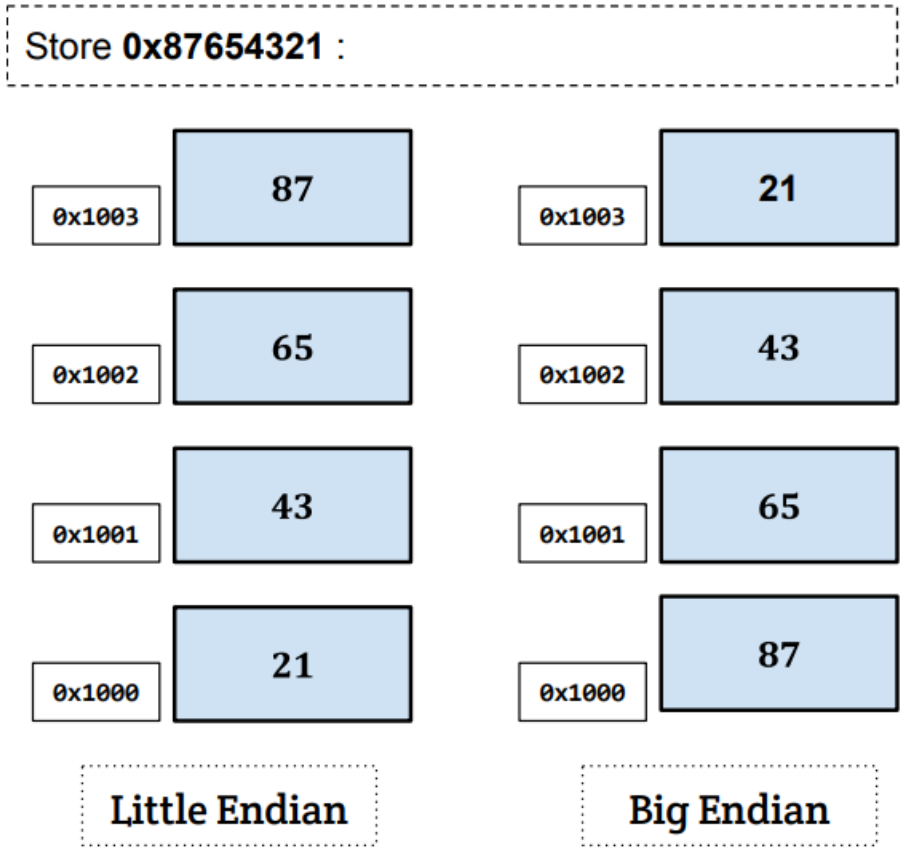
- &&: Logical AND
- ||: Logical OR
- !: Logical NOT

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

Copyright 2019 Chappell University (www.chappellU.com)

Endianness

- Endianness: a term that refers to the order in which a given multi-byte data item is stored in memory.
  - **Little Endian**: store the most significant byte of data at the highest address
  - **Big Endian**: 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
`htons()`	Convert unsigned short integer from host order to network order.
`htonl()`	Convert unsigned integer from host order to network order.
`ntohs()`	Convert unsigned short integer from network order to host order.
`ntohl()`	Convert unsigned integer from network order to host order.

Sniffing and Spoofing Attacks - Mitigation

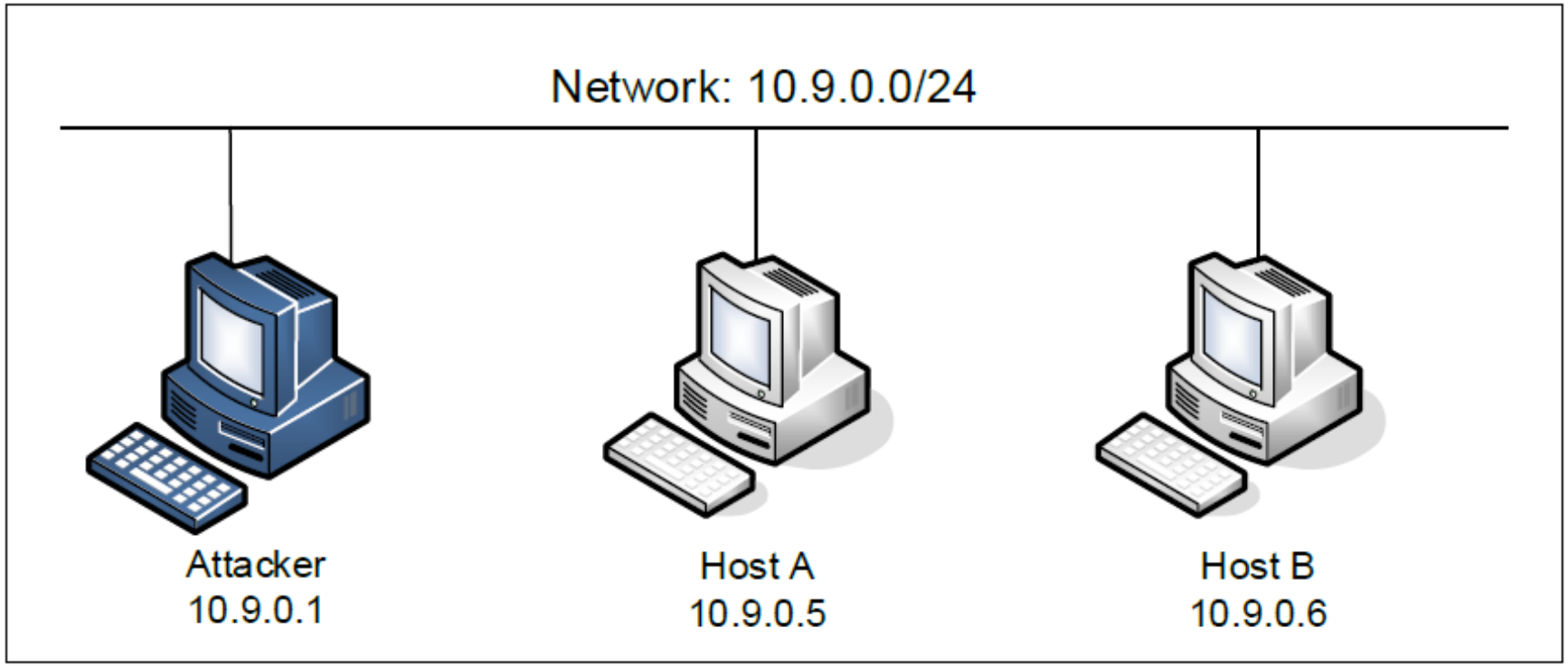
- Any protocol that uses cleartext communication is susceptible to this kind of attack.
- The only requirement for this attack to succeed is to have access to a system between the two communicating systems.
- The mitigation lies in adding an encryption layer on top of any network protocol.

Topics

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

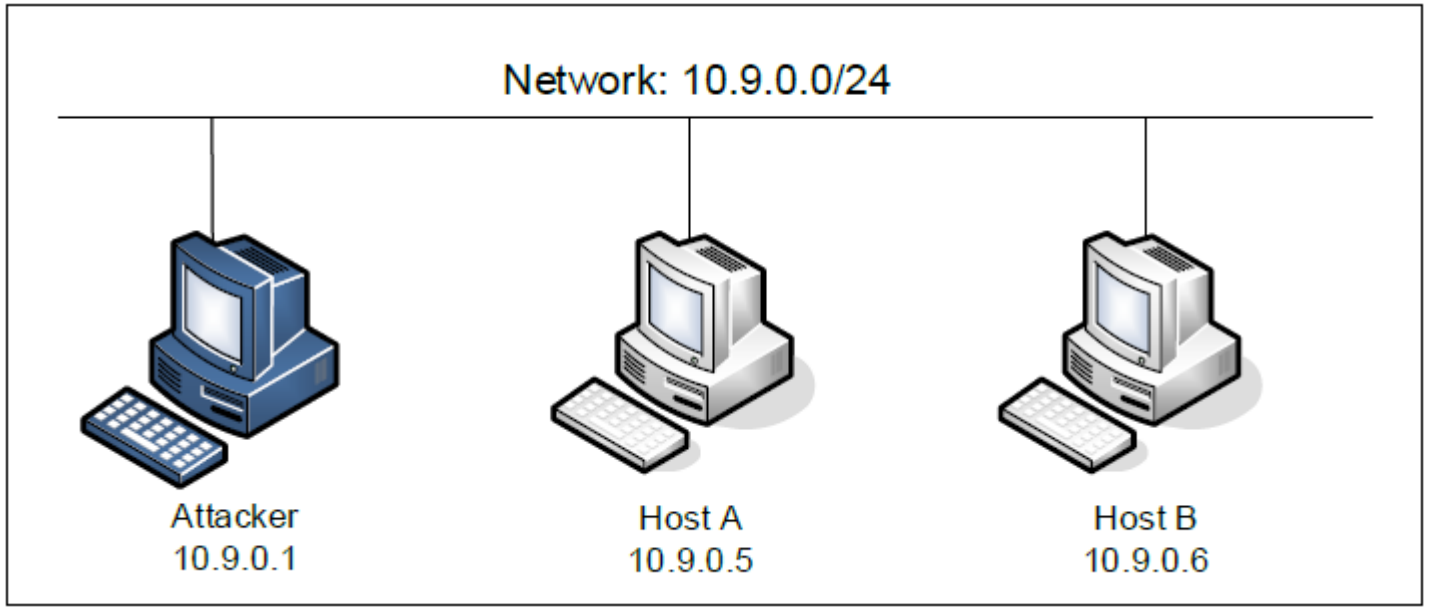
Lab Setup

- 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