CSCI 476: Computer Security

Network Security: Packet Sniffing and Spoofing

Reese Pearsall

Spring 2023

Announcement

Lab 5 (XSS) Due Sunday
3/26 @ 11:59 PM

XSS Countermeasures

Filtering → Remove any ability for a user to enter something that might look like a script

Encoding → HTML encode specific characters; e.g

`<script>blah</script>` → `&lt;blah&gt;`

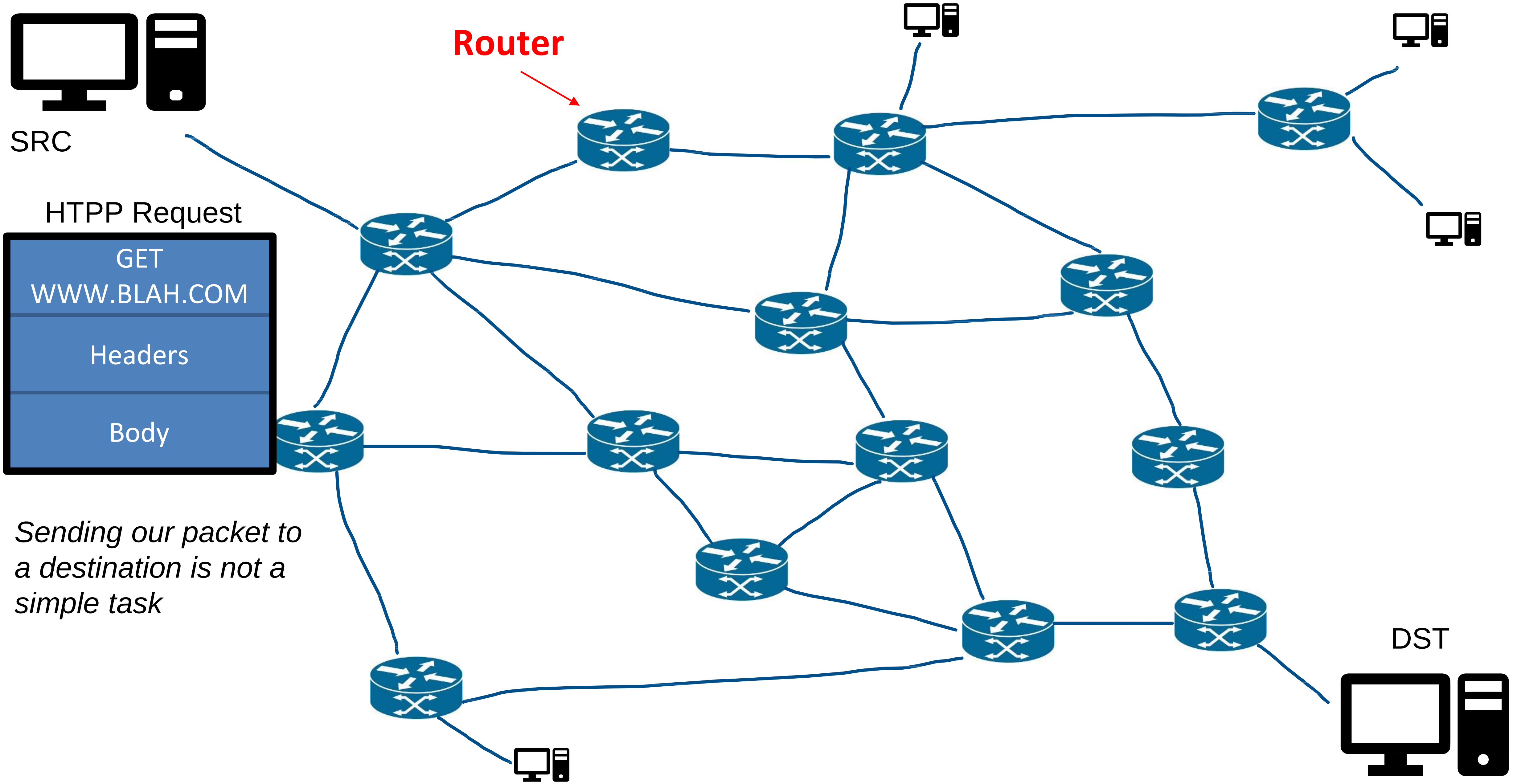
Content-Security-Policy (CSP)- The better countermeasure for XSS/Clickjacking attacks

- ❑ Clearly delineate code vs data via HTTP header values set by a server
- ❑ Restricts resources, such as scripts, that a page can load

CSP RULES

- `default-src ‘self’` → Only allows javascript code from current domain
- `script-src https://trusted-website.com` → only allows javascript code from trusted domain

Same **O**rigin **P**olicy, **C**ross **O**rigin **R**esource **S**haring policies



HTPP Request

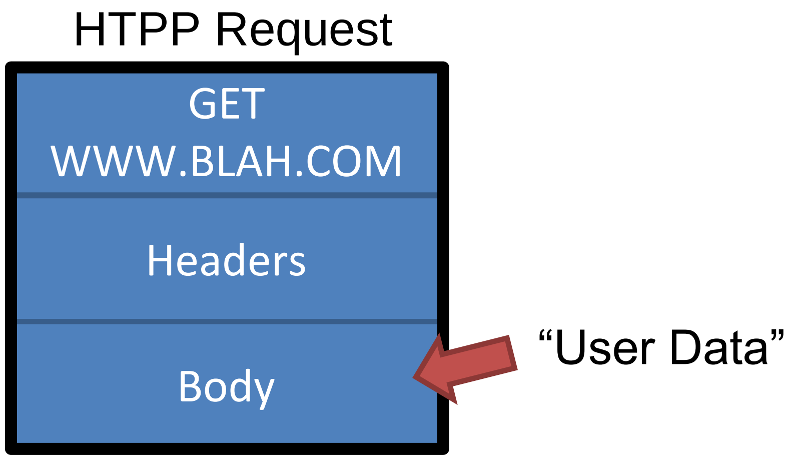
GET WWW.BLAH.COM

Headers

Body

Sending our packet to a destination is not a simple task

There is a lot of stuff that gets added onto our data being send



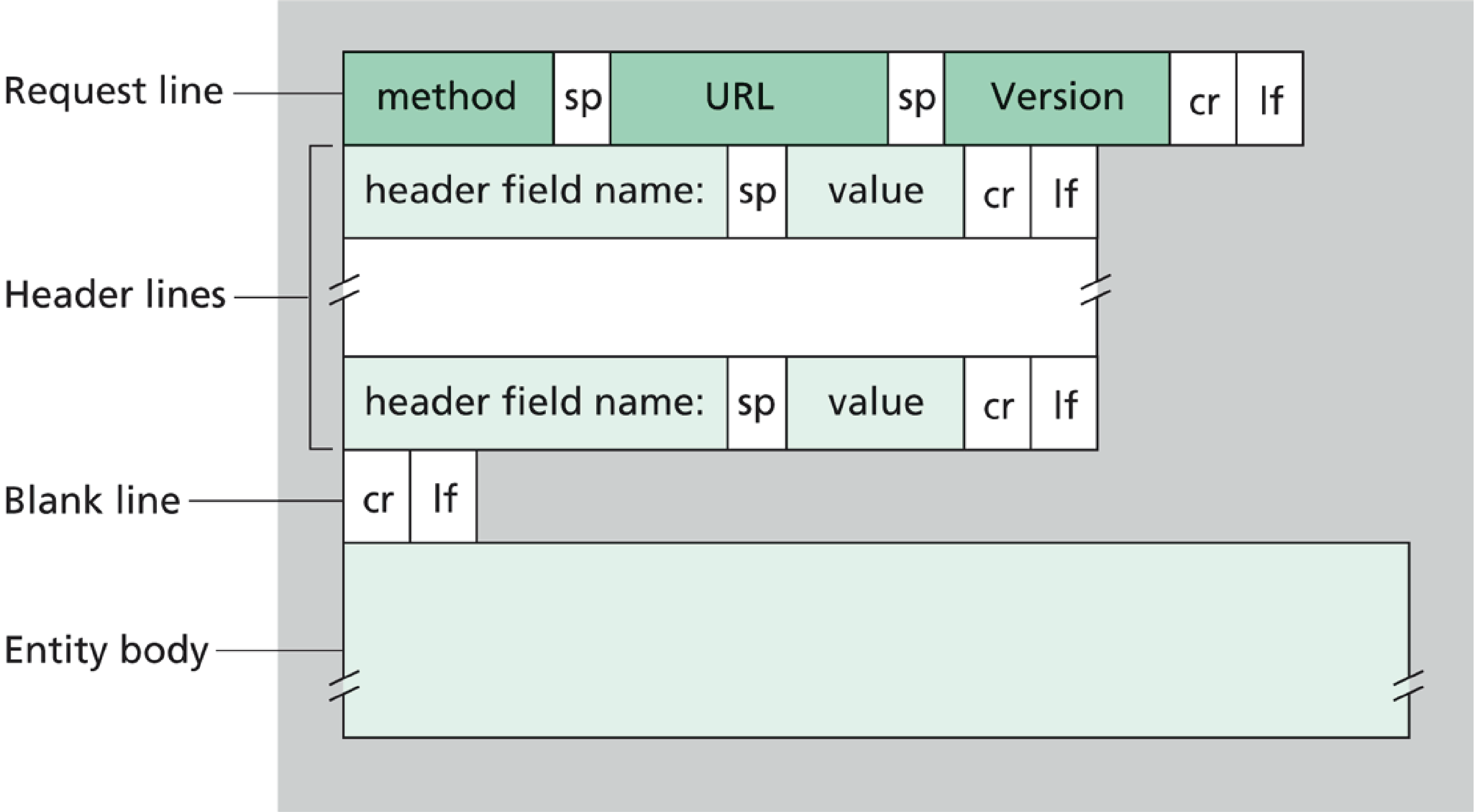
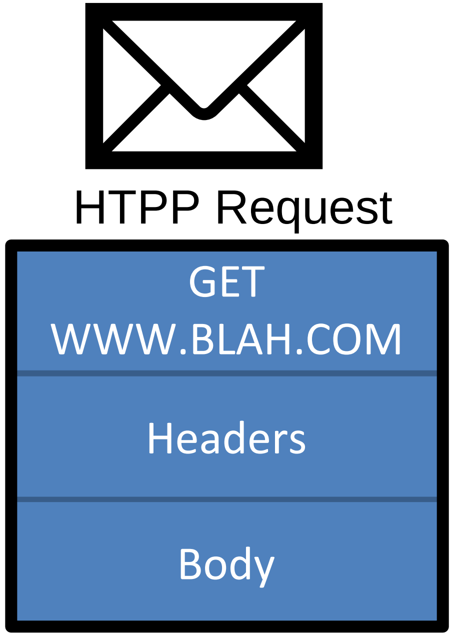


Figure 2.8 ♦ General format of a request message

There are a few pieces of information a packet needs in order to arrive to its destination



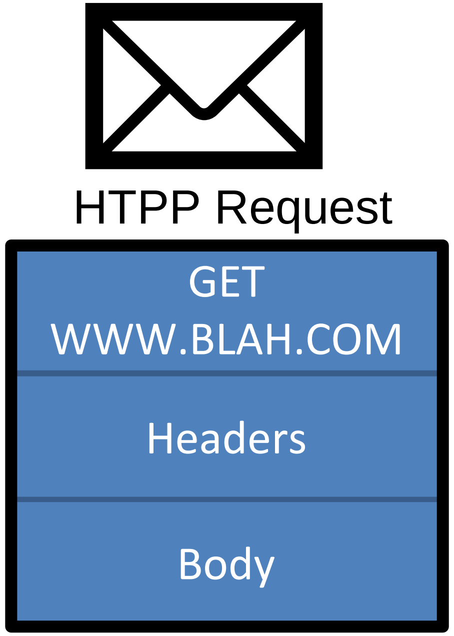
HTPP Request

GET WWW.BLAH.COM

Headers

Body

There are a few pieces of information a packet needs in order to arrive to its destination

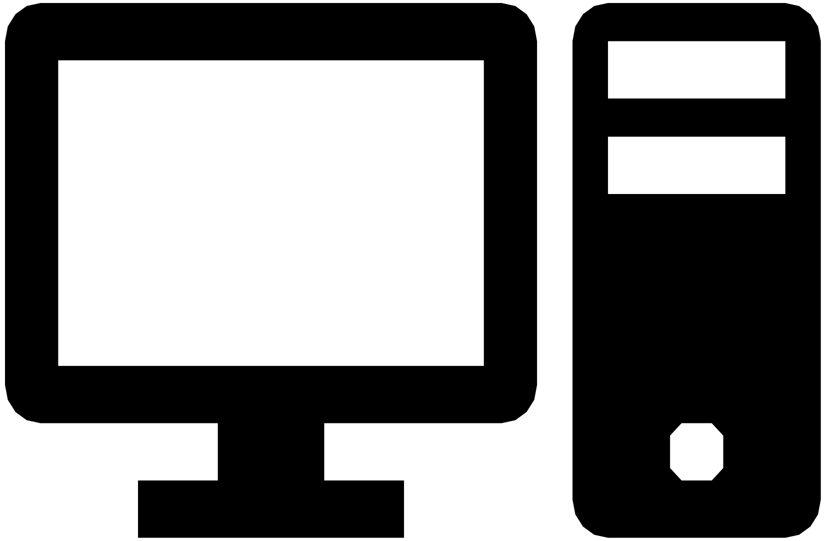


HTPP Request

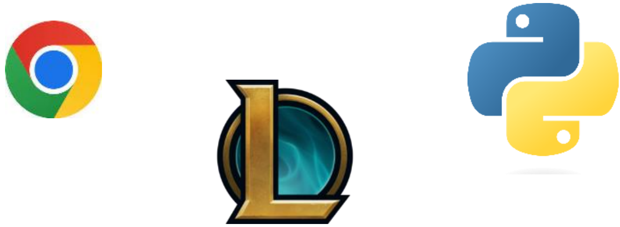
GET WWW.BLAH.COM

Headers

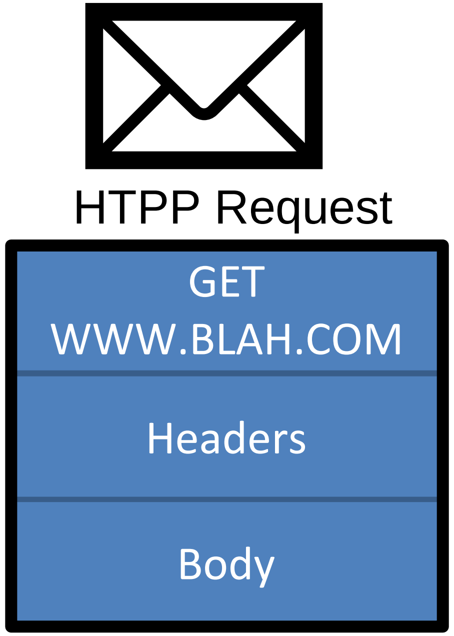
Body



A packet arriving to a machine needs to know which **process**/application to go to



There are a few pieces of information a packet needs in order to arrive to its destination

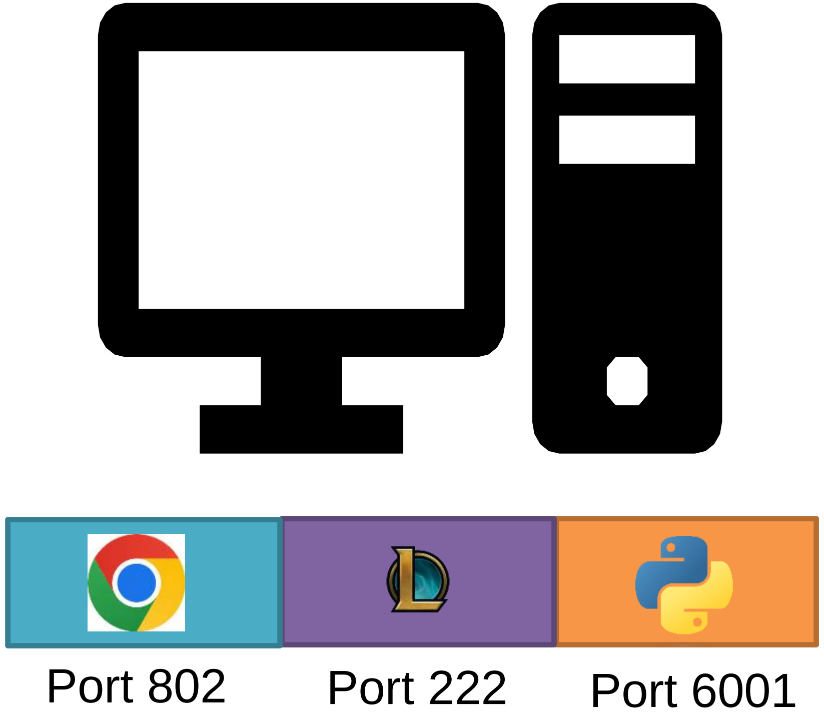


HTPP Request

GET WWW.BLAH.COM

Headers

Body



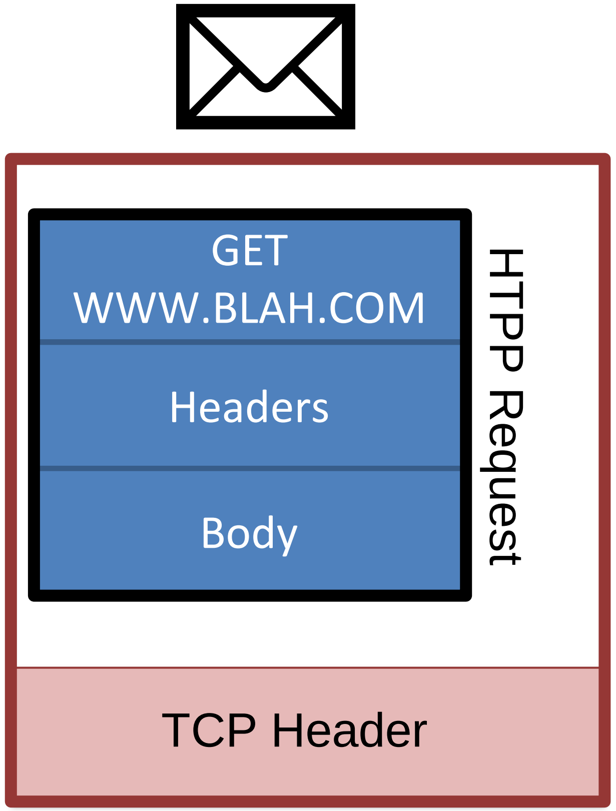
Port 802

Port 222

Port 6001

Each application is bound to a **port**, so each packet will need to know what port they need to go to

TCP is a transport-layer protocol that ensures data gets delivered, and controls how the two endpoints communicate with each other



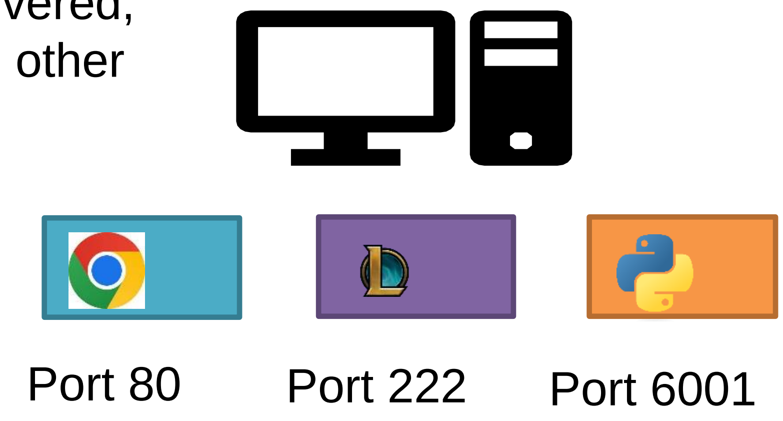
GET WWW.BLAH.COM

Headers

Body

HTPP Request

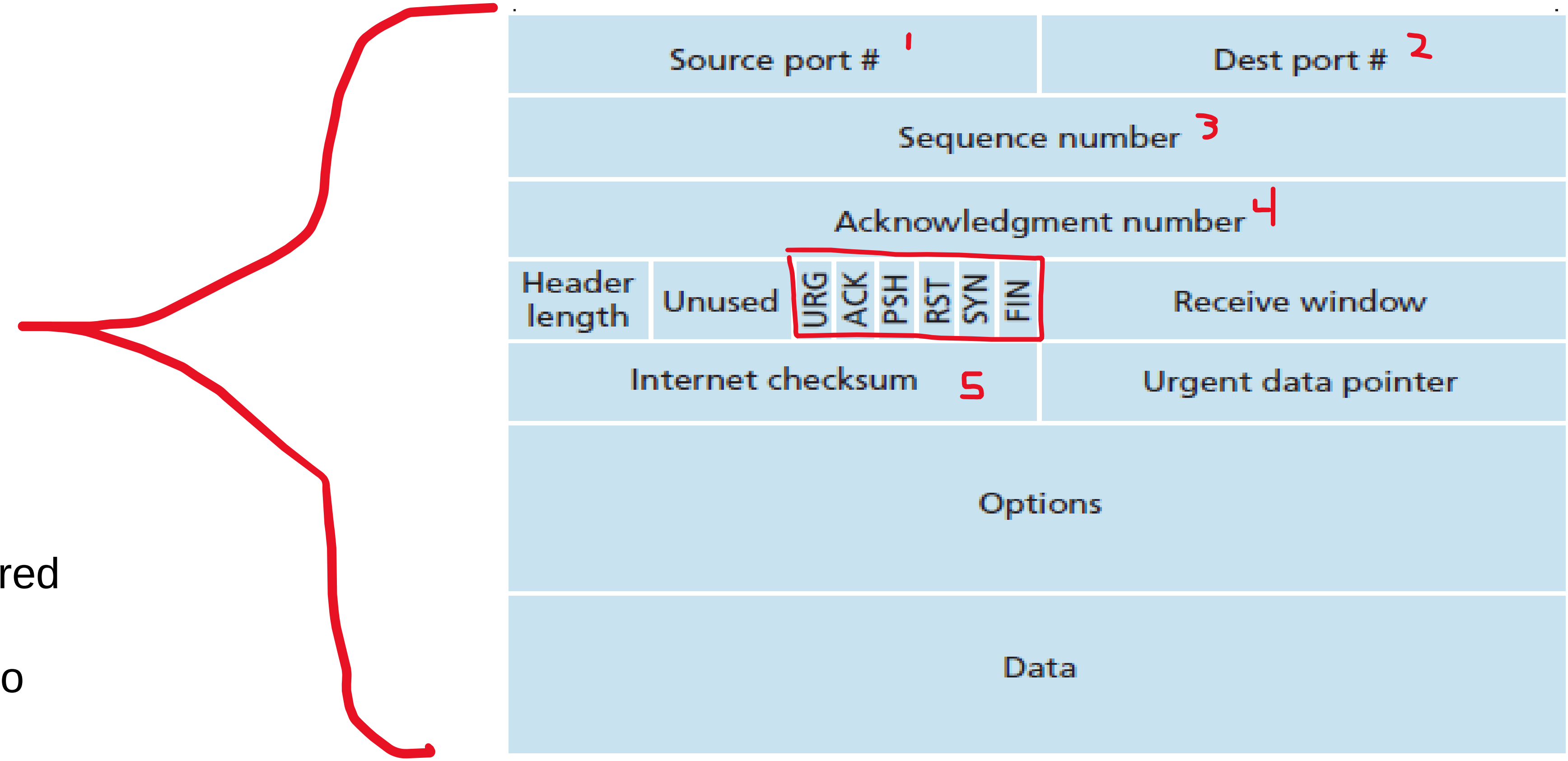
TCP Header



Port 80

Port 222

Port 6001



Source port # 1

Dest port # 2

Sequence number 3

Acknowledgment number 4

Header length

Unused

URG

ACK

PSH

RST

SYN

FIN

Receive window

Internet checksum 5

Urgent data pointer

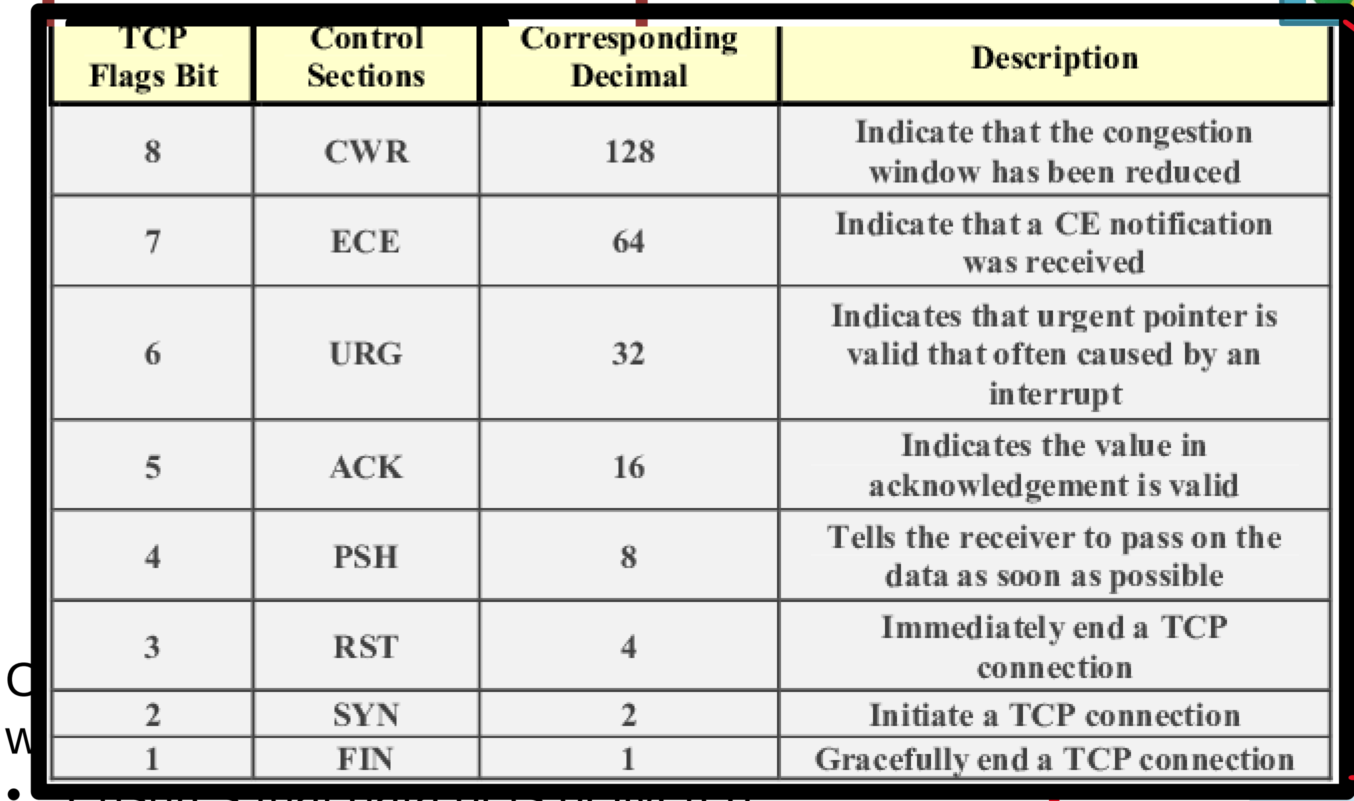
Options

Data

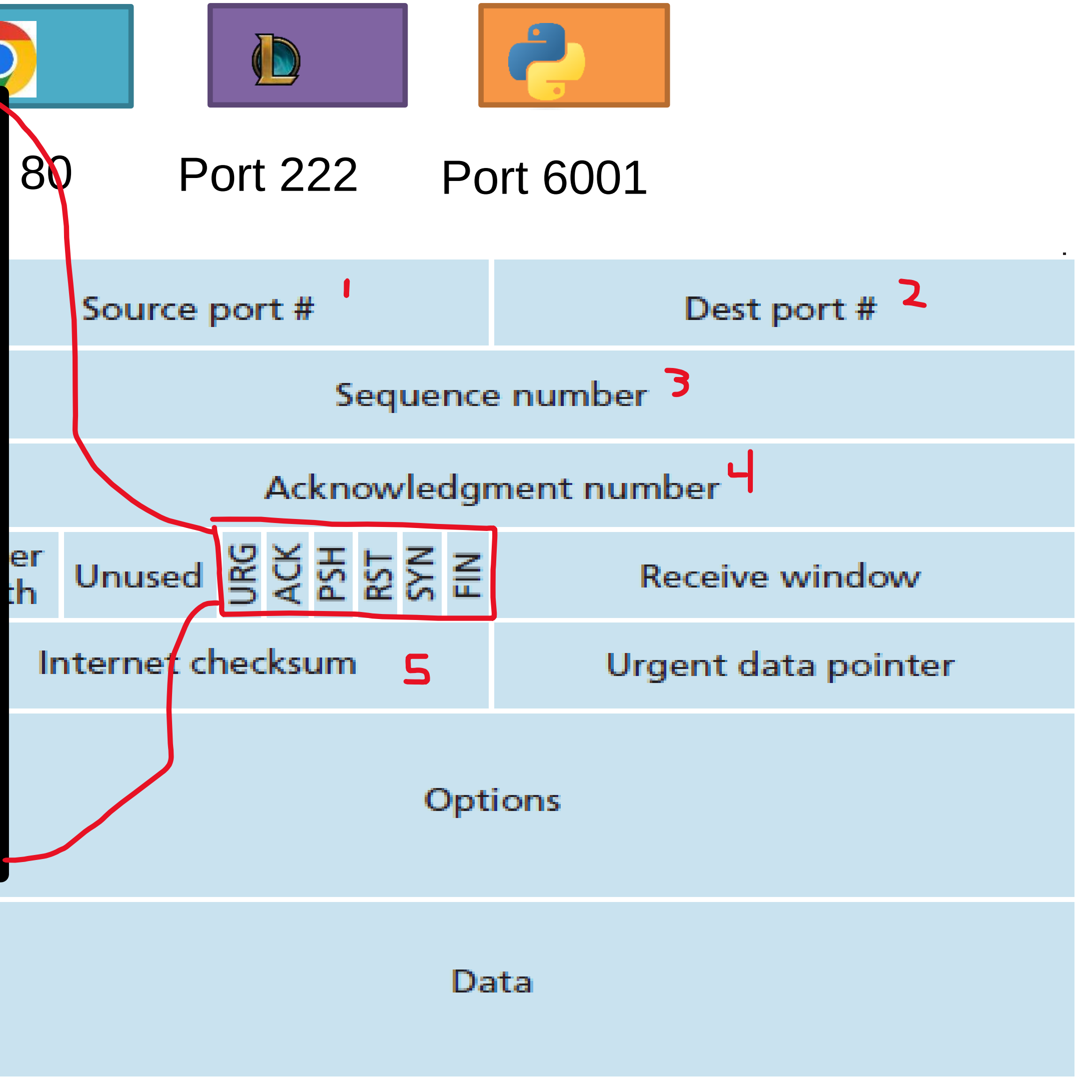
Our packet of information gets wrapped in a **TCP Header**

- Ensures that data gets delivered reliably (Seq/Ack #s)
- Ensures data gets delivered to the correct process (Port #s)

TCP is a transport-layer protocol that ensures data gets delivered, and controls how the two endpoints communicate with each other



TCP Flags Bit	Control Sections	Corresponding Decimal	Description
8	CWR	128	Indicate that the congestion window has been reduced
7	ECE	64	Indicate that a CE notification was received
6	URG	32	Indicates that urgent pointer is valid that often caused by an interrupt
5	ACK	16	Indicates the value in acknowledgement is valid
4	PSH	8	Tells the receiver to pass on the data as soon as possible
3	RST	4	Immediately end a TCP connection
2	SYN	2	Initiate a TCP connection
1	FIN	1	Gracefully end a TCP connection

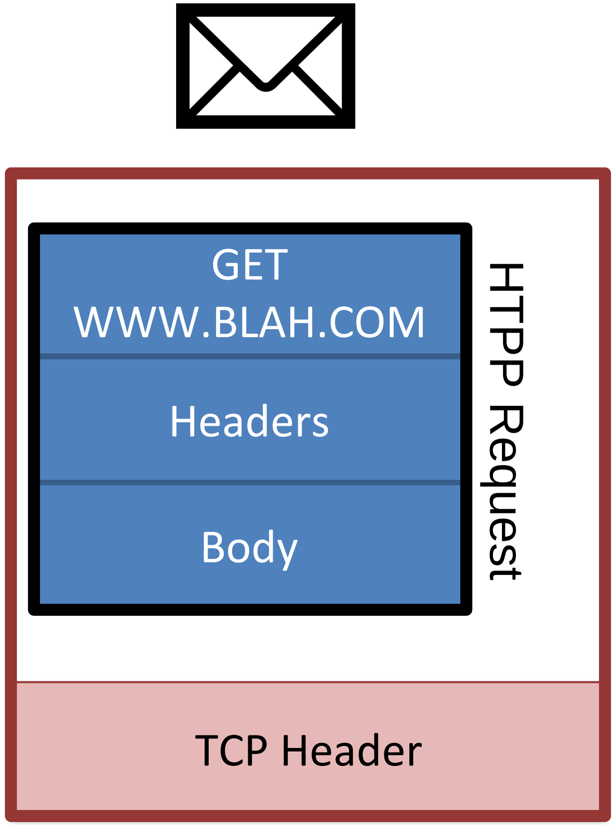


C
w

- Ensures that data gets delivered reliably (Seq/Ack #s)
- Ensures data gets delivered to the correct process (Port #s)

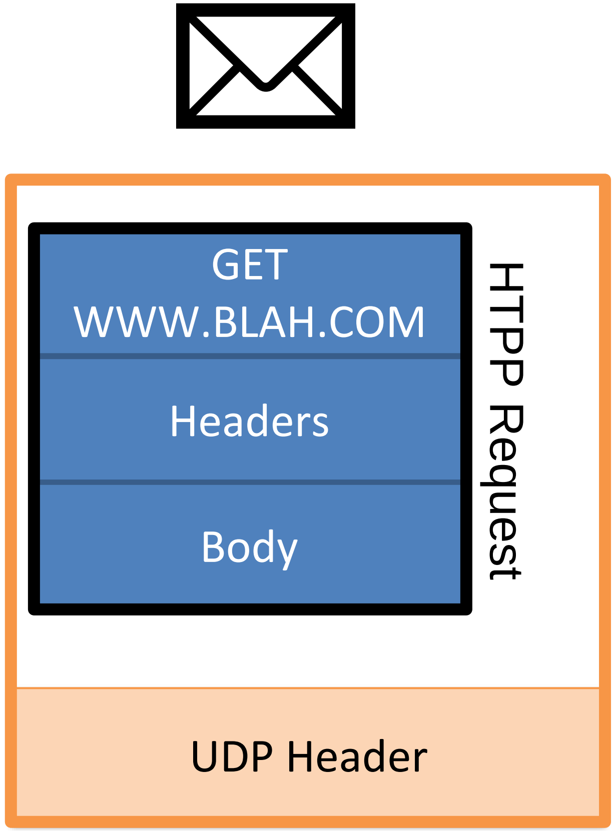
Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)



Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)

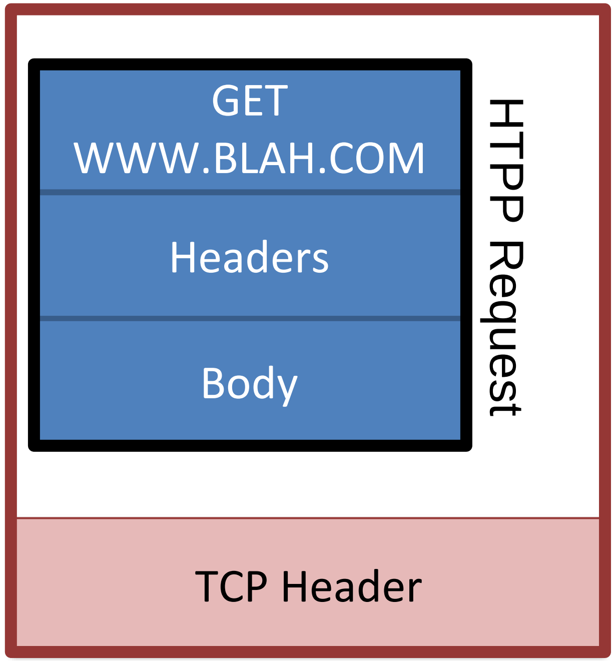


Applications will either user **TCP** or **UDP** to send their data. UDP adds on port #s just like TCP, but does not ensure reliable delivery

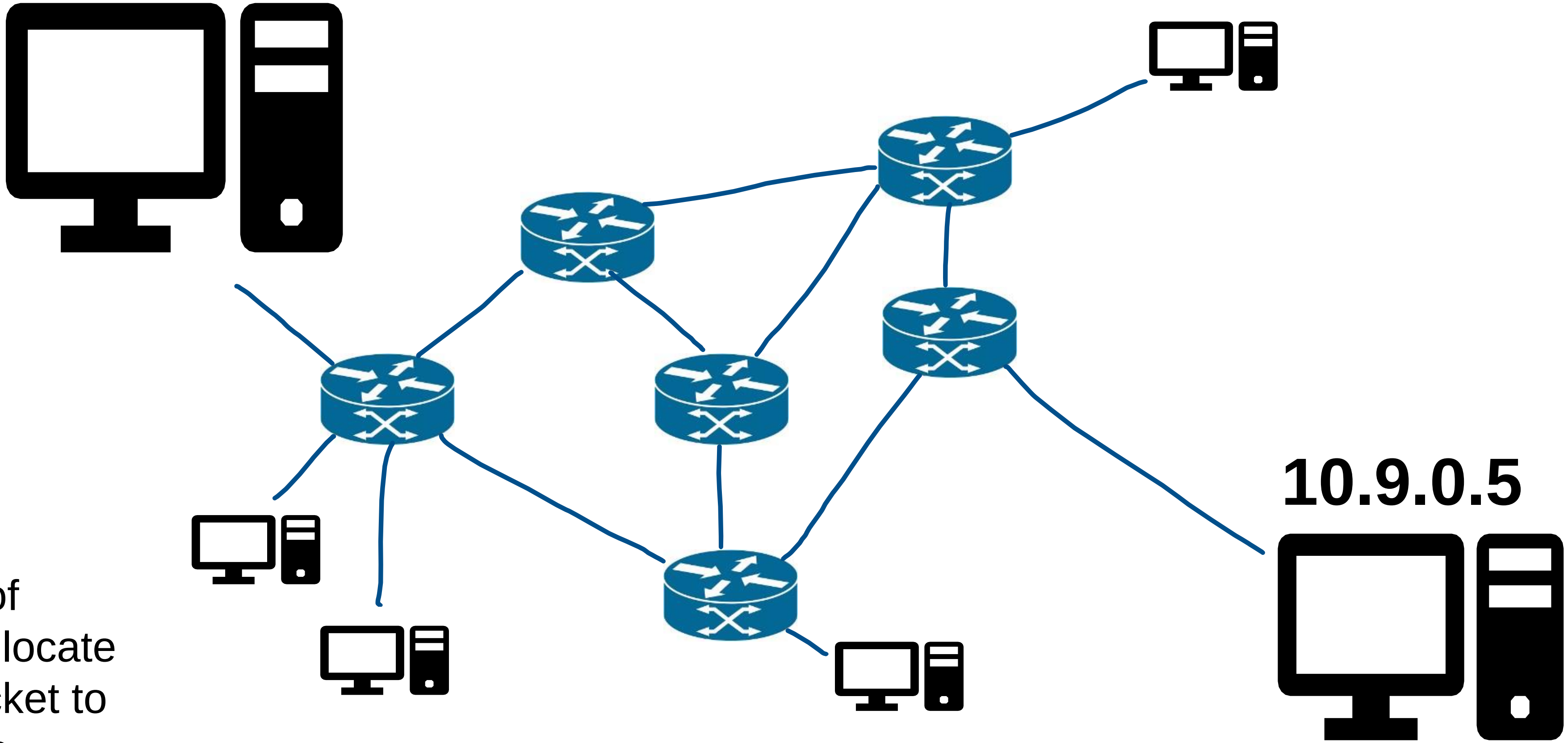
HTTP/HTTPS uses TCP, DNS protocol uses UDP

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)



We also need to know which device to send to → IP Address

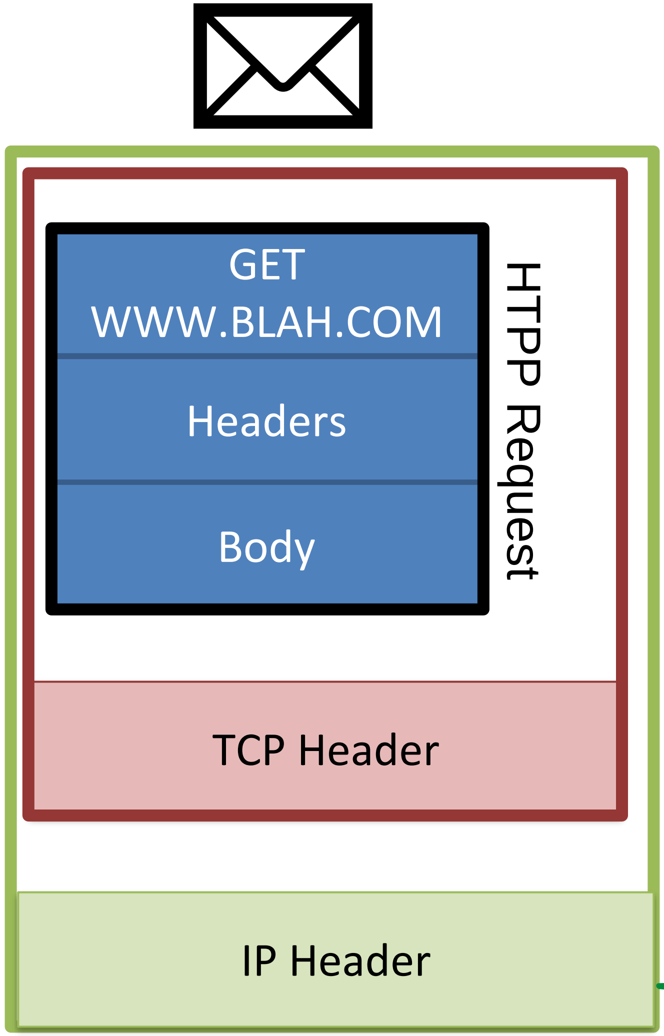


Think of the internet as a bunch of islands. The IP address helps us locate the correct island to send the packet to (Routers look at the IP address to determine where to forward the packet to)

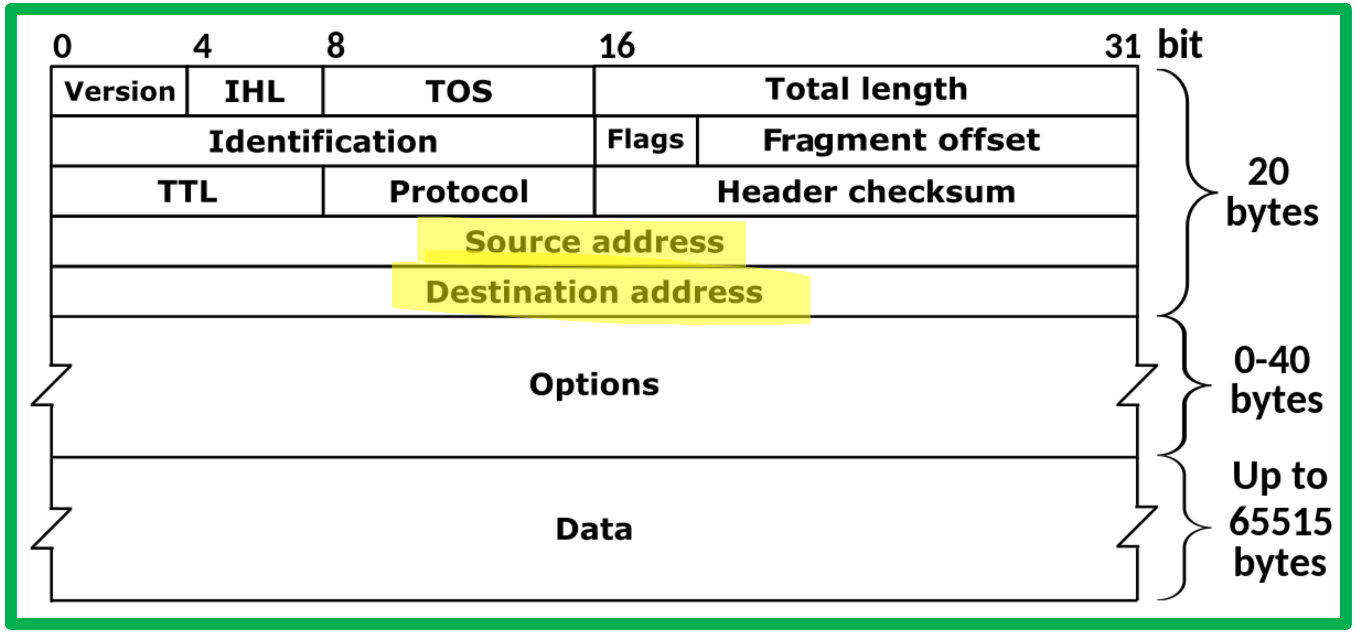
(IP address is more important for **locating** a machine rather than **identifying** a machine)

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)



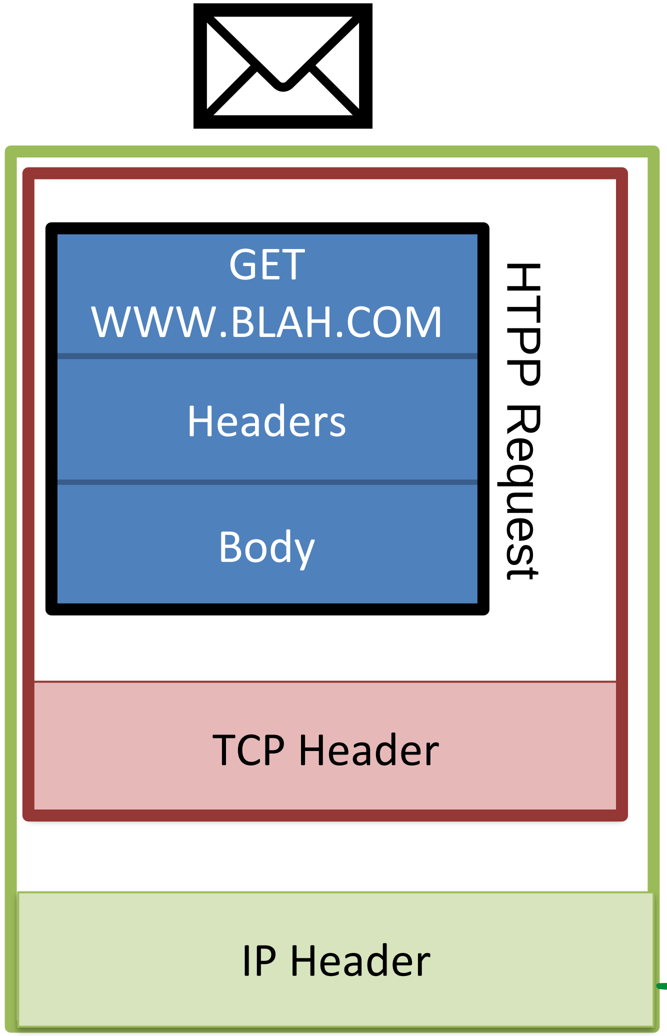
To add an IP address to our packet, we add another header, called the **IP Header**



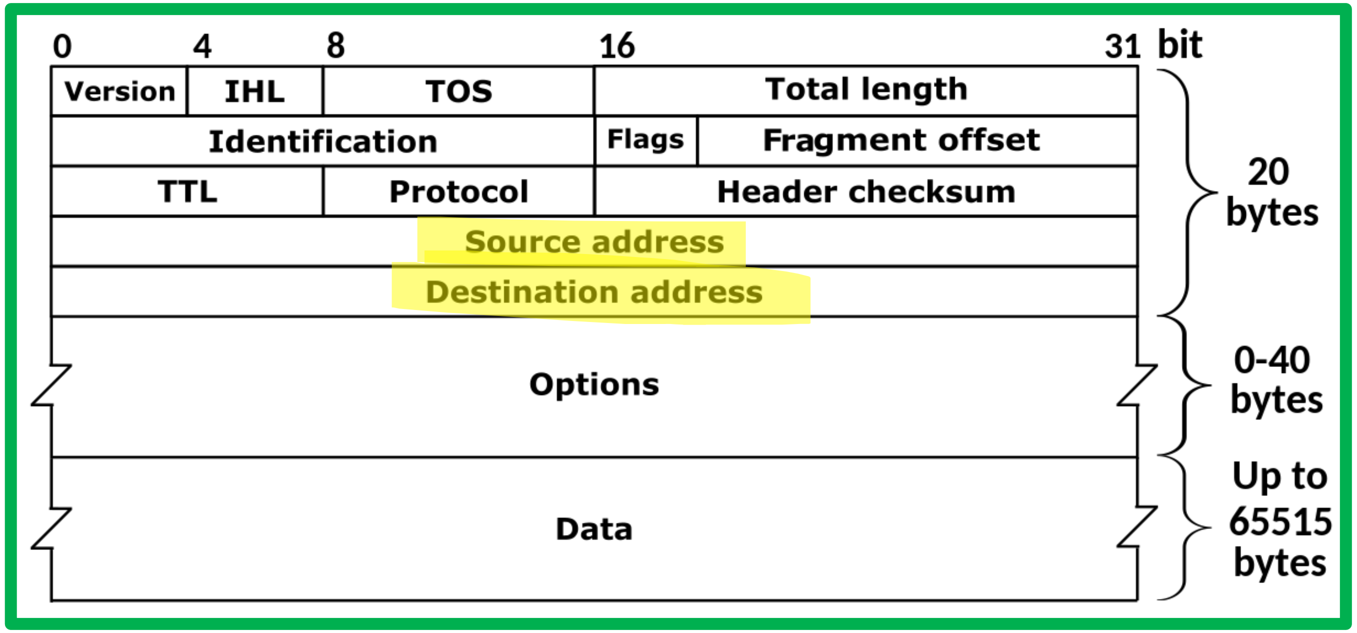
There are two types of IP addresses: IPv4 (32 bits) and IPv6 (128 bits), we use IPv4 in this class ☺

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)



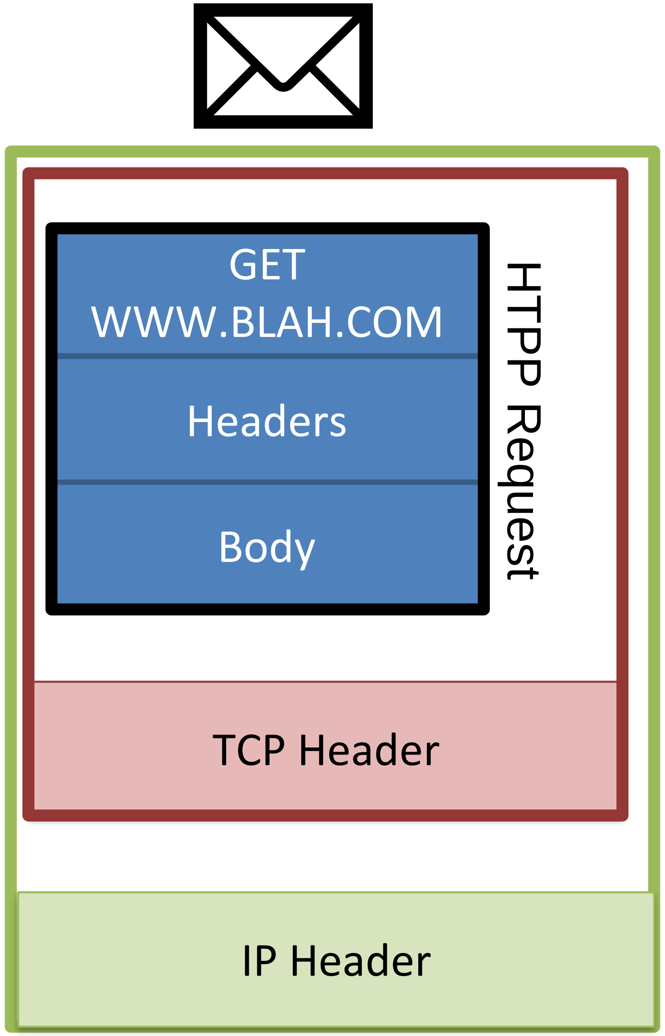
To add an IP address to our packet, we add another header, called the **IP Header**



IP Addresses are dynamic (generally), can be public/private, and can sometimes be shared between multiple devices

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)



GET WWW.BLAH.COM

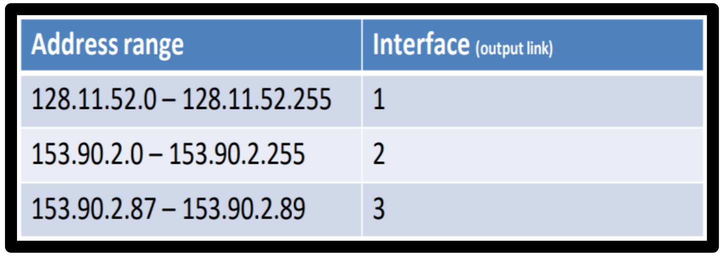
Headers

Body

HTPP Request

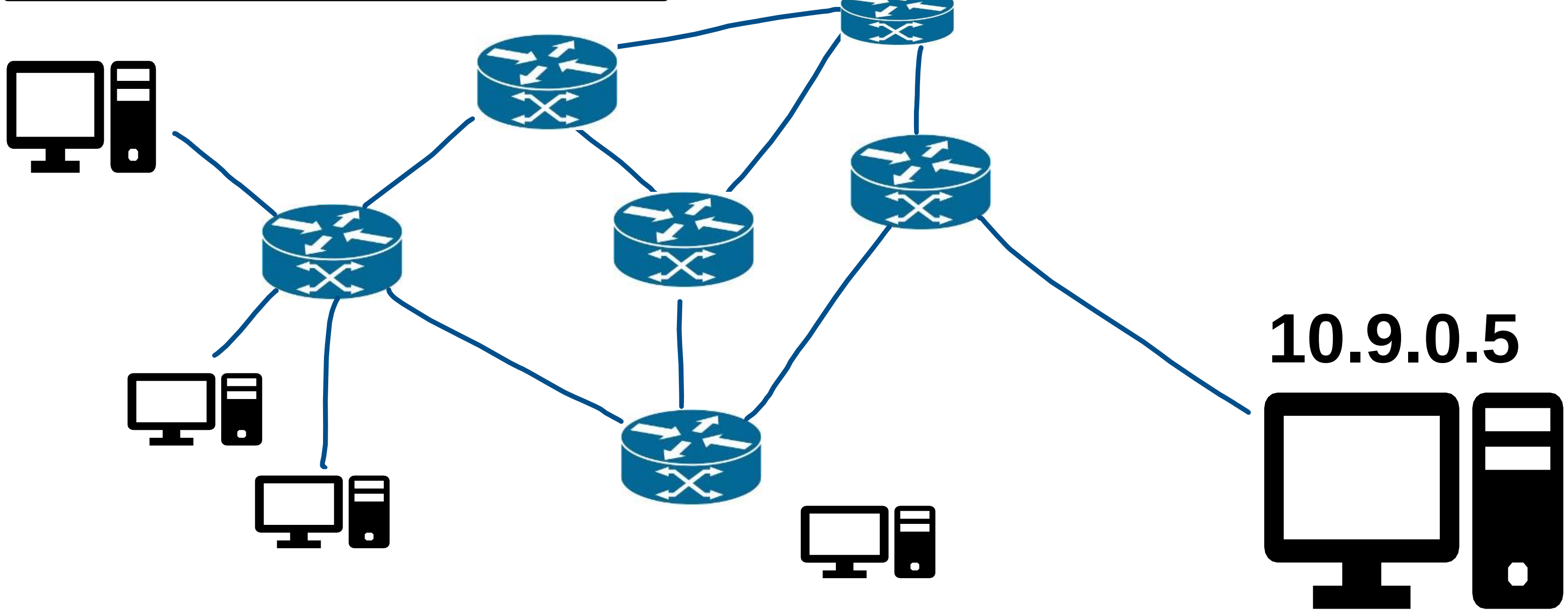
TCP Header

IP Header



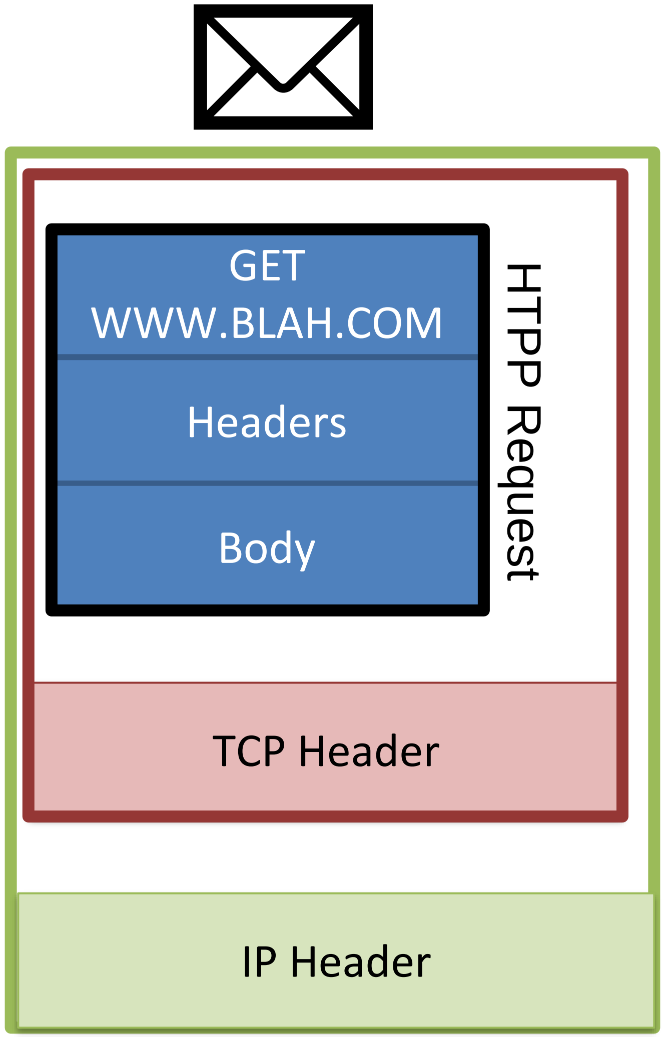
Address range	Interface (output link)
128.11.52.0 – 128.11.52.255	1
153.90.2.0 – 153.90.2.255	2
153.90.2.87 – 153.90.2.89	3

Routers maintain a table that will forward a packet to the next router based on the packet's destination IP address*

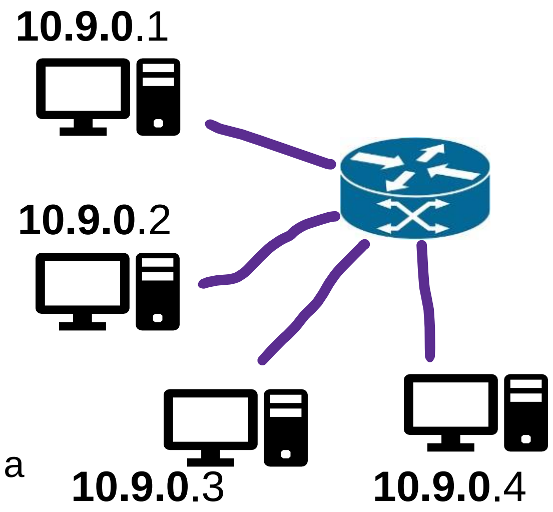


Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)



A packet may arrive to a network, but there is likely many devices under one network

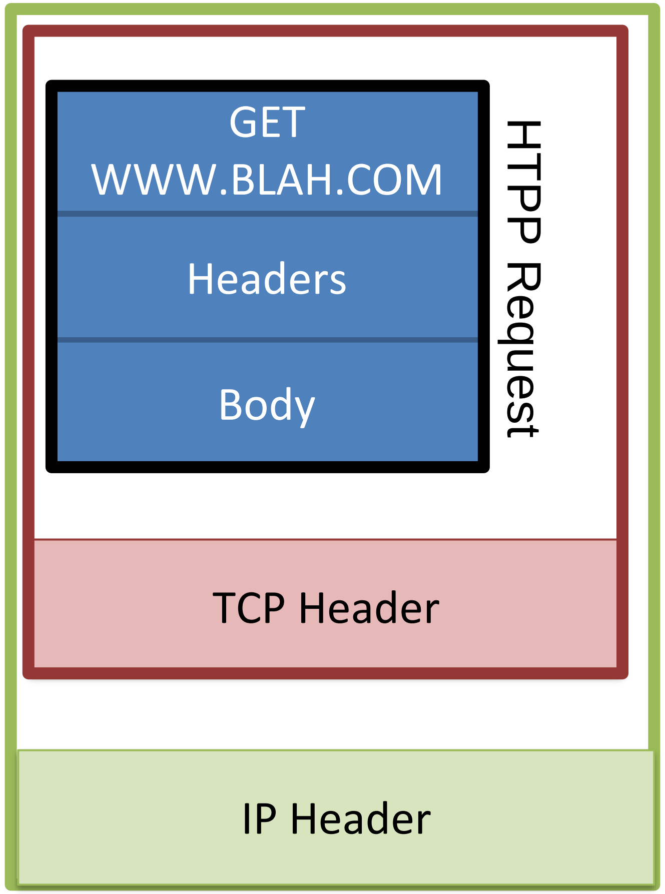


We now need a unique identifier to find the destination device on this local network

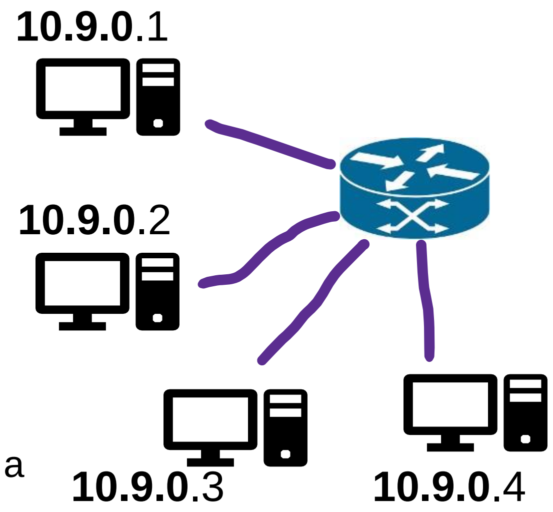
Devices in a **subnet** share a common prefix for their IP addresses

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)



A packet may arrive to a network, but there is likely many devices under one network

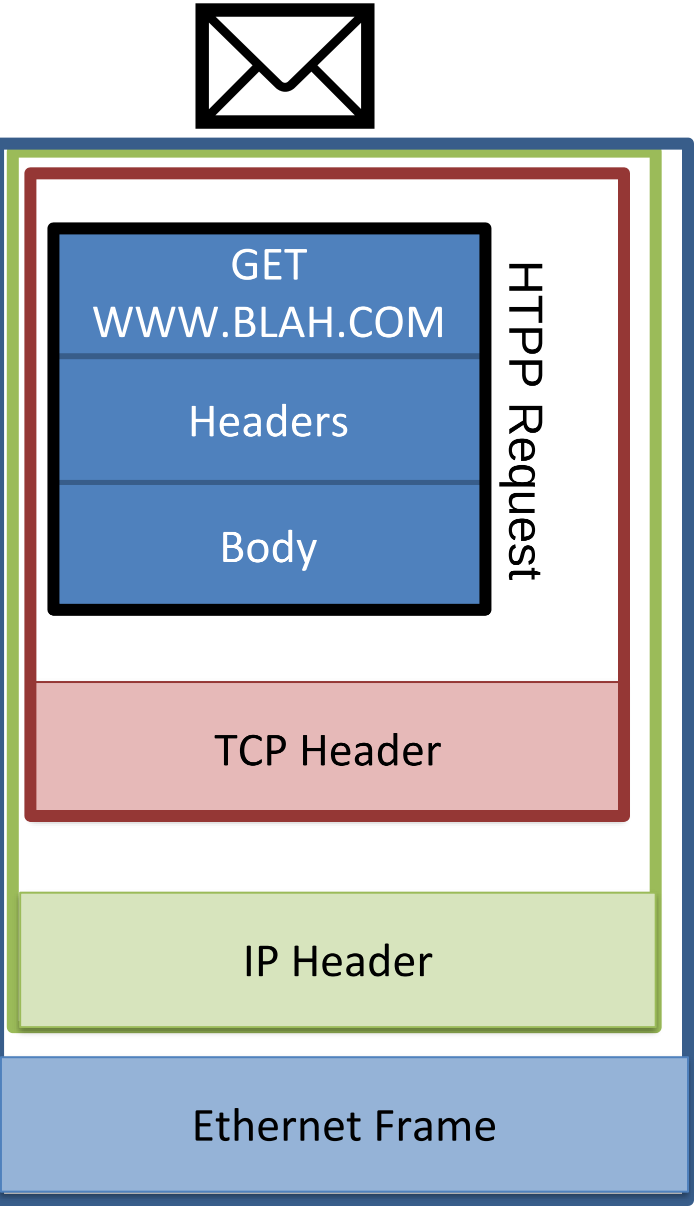


We now need a unique identifier to find the destination device on this local network

A **MAC address** is a unique, hard-coded value given to each device connected to a network

(Additionally, there might be times computers communicate without IP address)

Devices in a **subnet** share a common prefix for their IP addresses



GET WWW.BLAH.COM

Headers

Body

HTPP Request

TCP Header

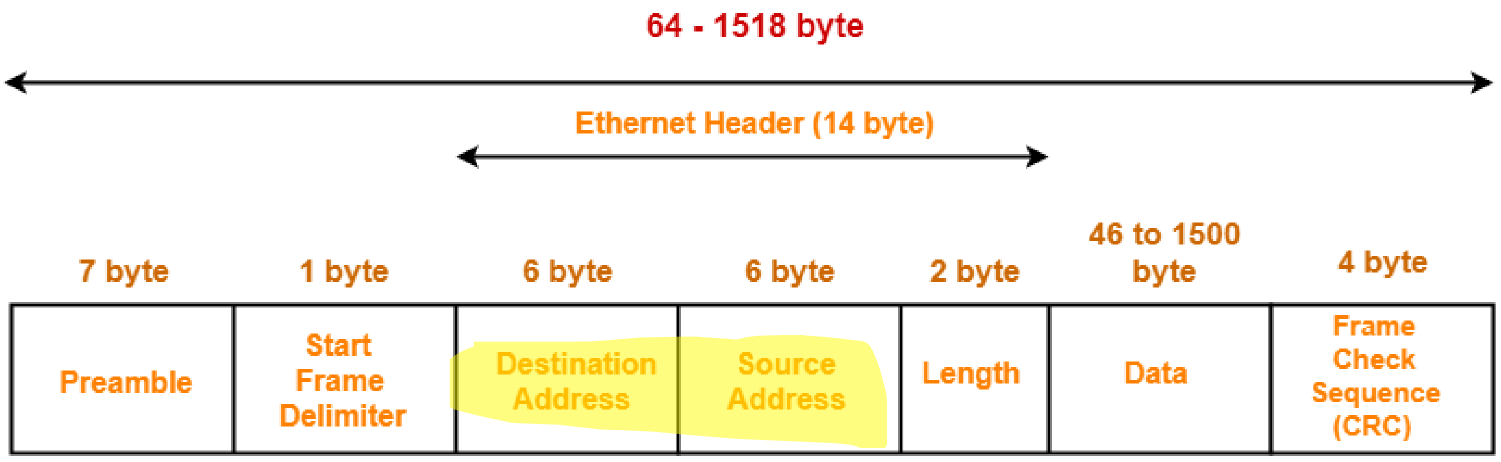
IP Header

Ethernet Frame

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)
- A unique identifier for our destination (MAC Address/Frame)

To add the MAC Address to our packet, we wrap our packet in an **ethernet frame** (usually)



64 - 1518 byte

Ethernet Header (14 byte)

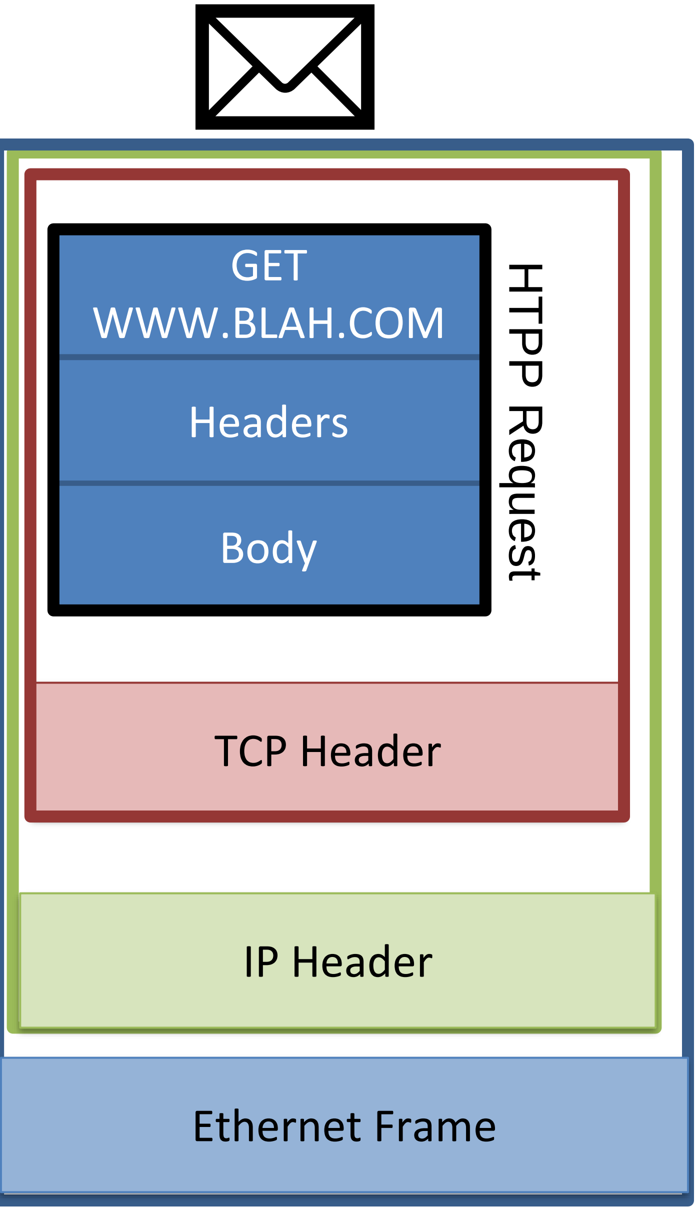
7 byte	1 byte	6 byte	6 byte	2 byte	46 to 1500 byte	4 byte
Preamble	Start Frame Delimiter	Destination Address	Source Address	Length	Data	Frame Check Sequence (CRC)

IEEE 802.3 Ethernet Frame Format

(We have protocols that can map IP Address → Mac Address)

Our packet currently has

- Some application-level message (HTTP Request)
- Port number of that application process (TCP header)
- Mechanism to ensure our packet arrives correctly (TCP Header)
- A way to locate the computer (IP address/IP Header)
- A unique identifier for our destination (MAC Address/Frame)



GET WWW.BLAH.COM

Headers

Body

HTPP Request

TCP Header

IP Header

Ethernet Frame

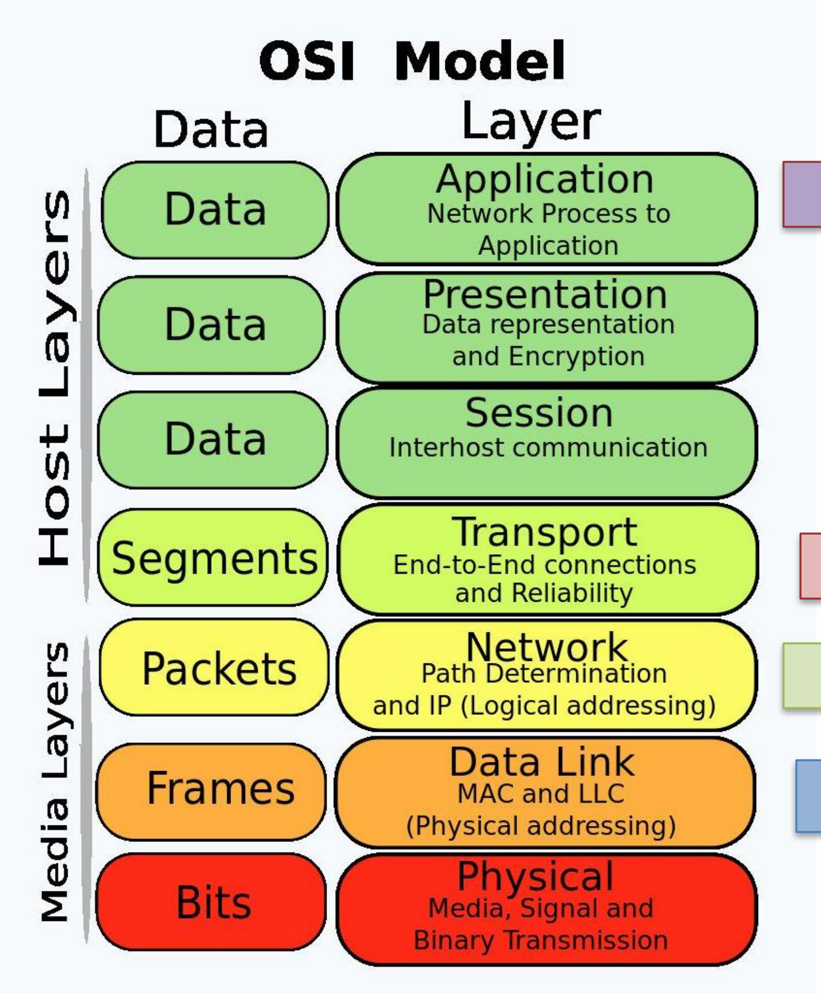
Our final packet!

Our initially packet gets encapsulated multiple times, sort of like a nesting doll!

(Jump scare warning for CSCI 466 people)

The Journey of a packet

Packets are **encapsulated** in various protocol layers; each has a **header** and **payload**


OSI Model
Data
Layer
Host Layers
Data
Application
Network Process to Application
Data
Presentation
Data representation and Encryption
Data
Session
Interhost communication
Segments
Transport
End-to-End connections and Reliability
Media Layers
Packets
Network
Path Determination and IP (Logical addressing)
Frames
Data Link
MAC and LLC (Physical addressing)
Bits
Physical
Media, Signal and Binary Transmission

HTTP Request

(Encryption, Authentication, Management, HTTPS)

TCP Header

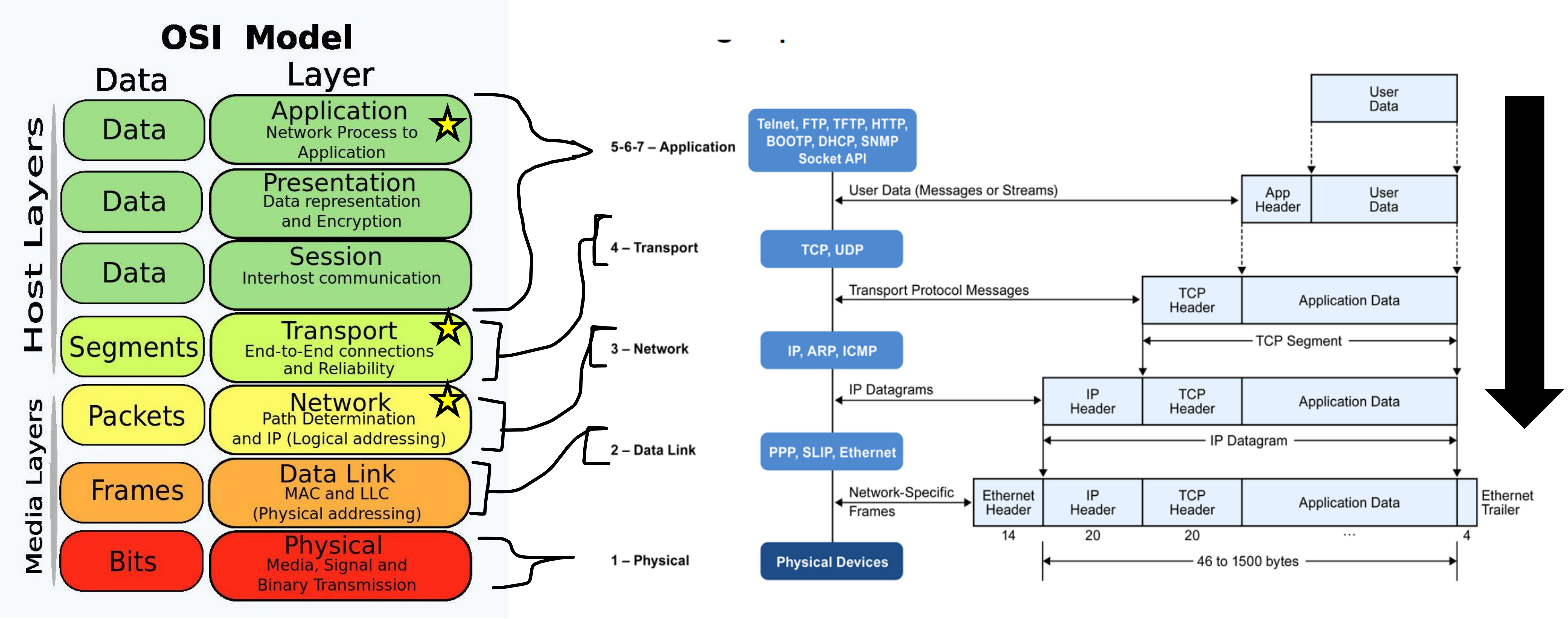
IP Header

Ethernet Frame

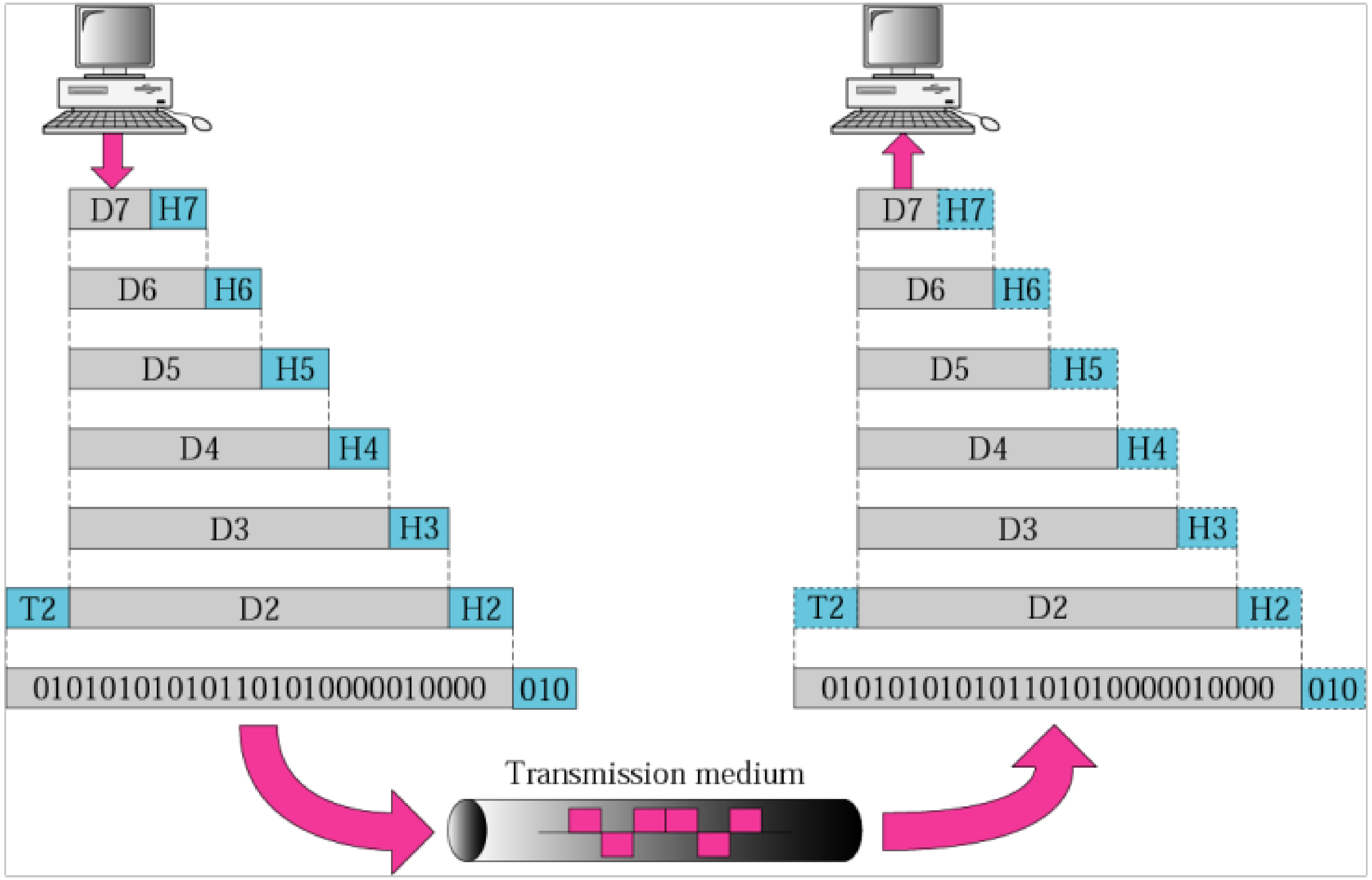
The **OSI Model** is a very popular internet stack model that describes the layers of the internet, and the different responsibilities of each layer

The Journey of a packet

Packets are **encapsulated** in various protocol layers; each has a **header** and **payload**



Our focus in the next few weeks will be on the transport layer (**TCP/UDP**), network layer (**IP**), and application layer

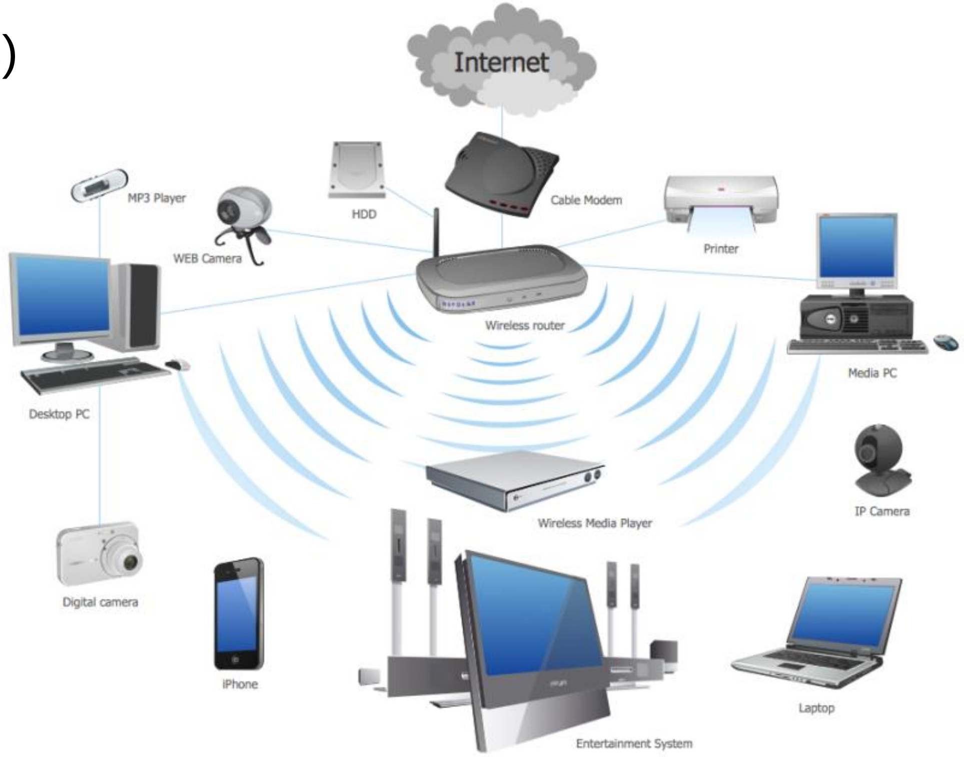


WIFI, Fiber optic, Copper Wire, Birds

*** Many devices are sharing this medium*

Devices connect to a network via a **Network Interface Card** (NIC)

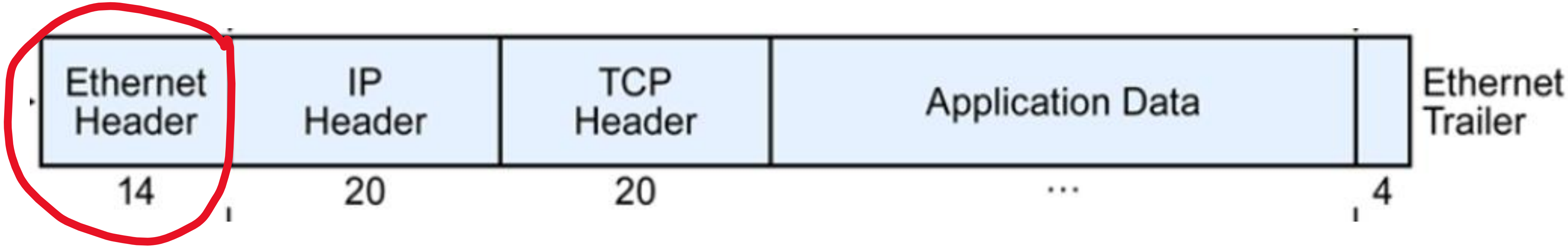
25-6B-78-1D-A0-57



Each NIC as a **Medium Access Control** (MAC) address

Every NIC “hears” all the frames “on the wire” (or “in the air”)

NIC checks destination (dst) address of the packet’s link layer header



Ethernet Header	IP Header	TCP Header	Application Data	Ethernet Trailer
14	20	20	...	4

Accept packets that match the NIC’s MAC address, “**drop**” other packets

25-6B-78-1D-A0-57 ✓ **41-91-31-CB-F0-BD** ✗

How do we get *all* the network traffic?

Promiscuous Mode

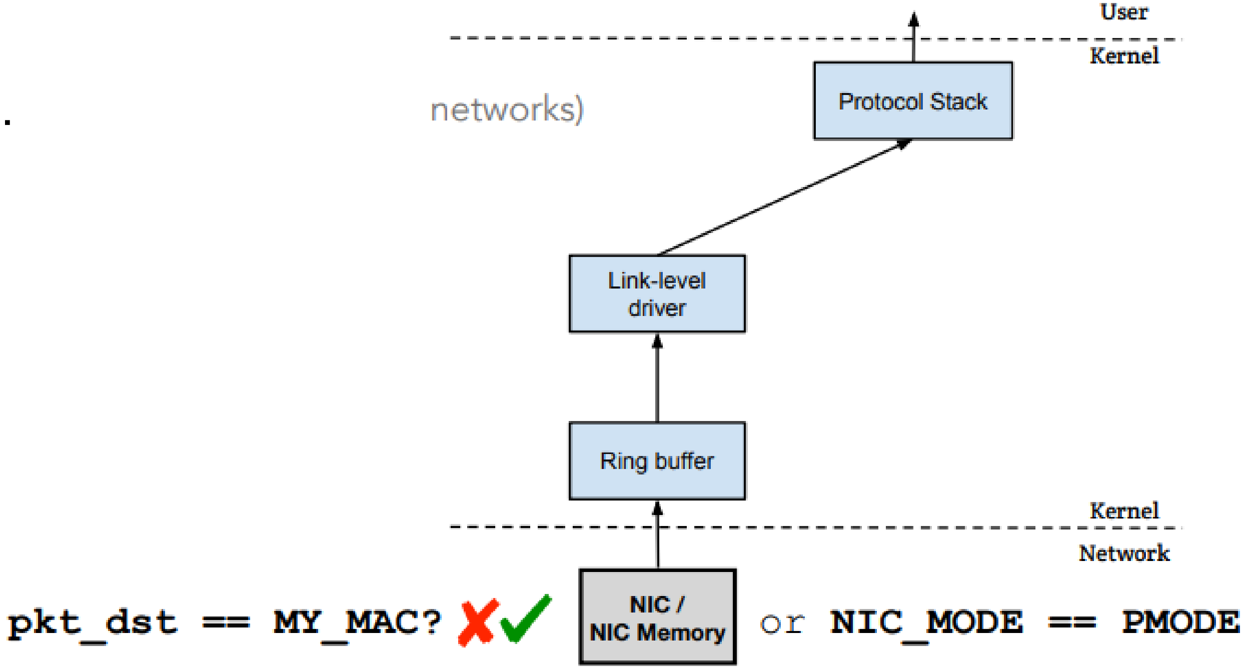
- Frames that are not destined to a given NIC are normally discarded
- When operating in promiscuous mode, the NIC passes every frame received from the network to the kernel
- If a **sniffer** program is registered with the kernel, it will be able to see all the packets

There are **tons** of packets. We don’t need all of them…

The interesting ones are **TCP**, **UDP**, **DNS**, ~~HTTPS~~

Lets start “sniffing” for packets!

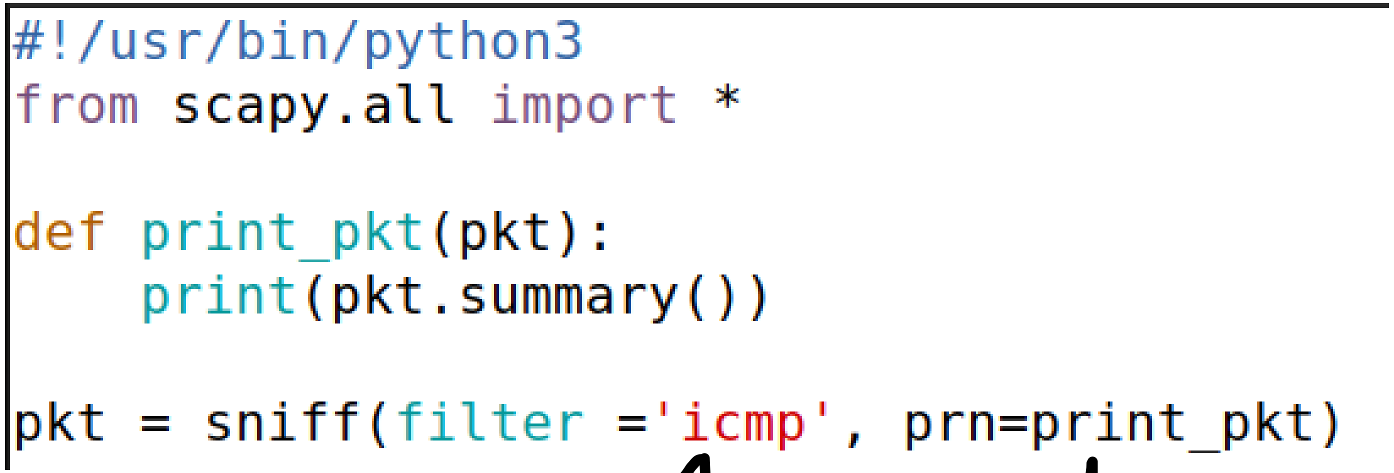
We can write a python program that will sniff packets for us!



Packet Sniffing (Python)

scapy is a python module designed for packet sniffing and spoofing

sniffer.py



```
#!/usr/bin/python3
from scapy.all import *

def print_pkt(pkt):
    print(pkt.summary())

pkt = sniff(filter ='icmp', prn=print_pkt)
```

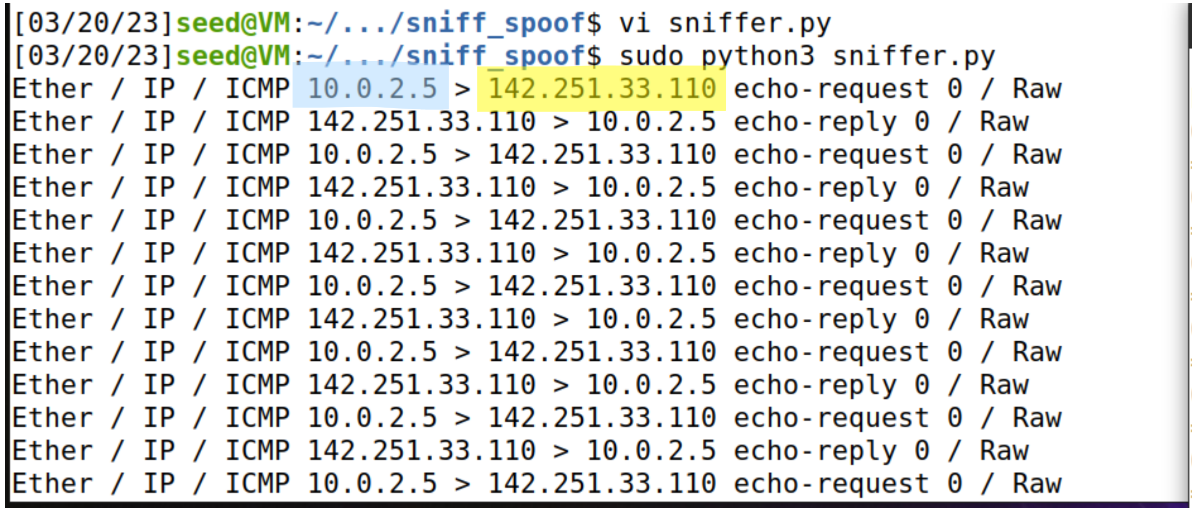
Sniff only `icmp` packets

When a matching packet is caught, run the `print_pkt` function

Scapy uses **Berkeley Packet Filter** (**BPF**) syntax to filter packets

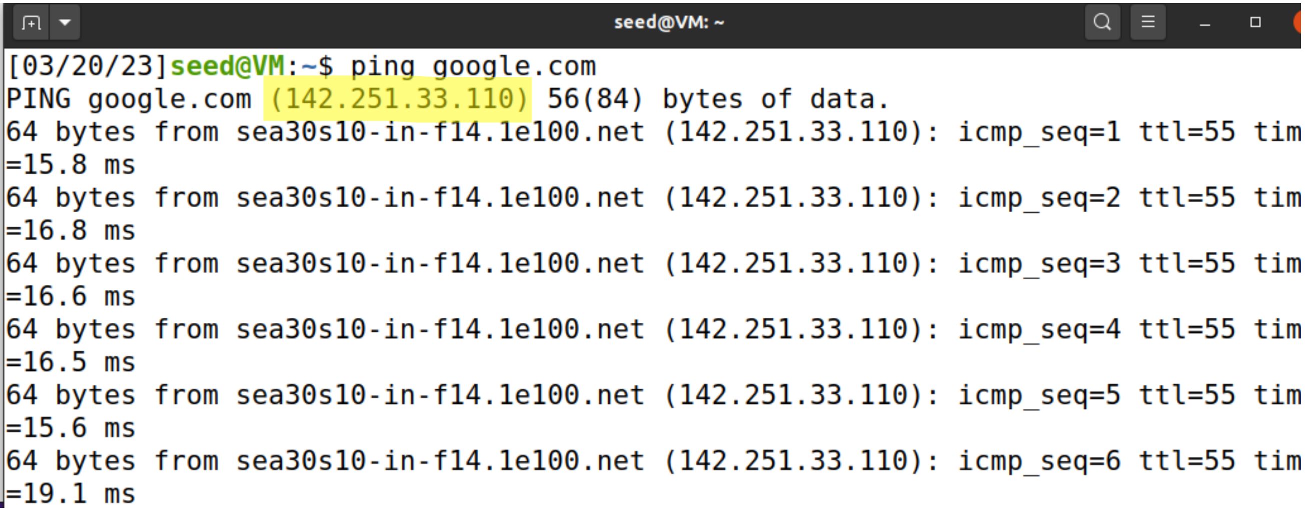
Packet Sniffing (Python)

1. Start the sniffer program



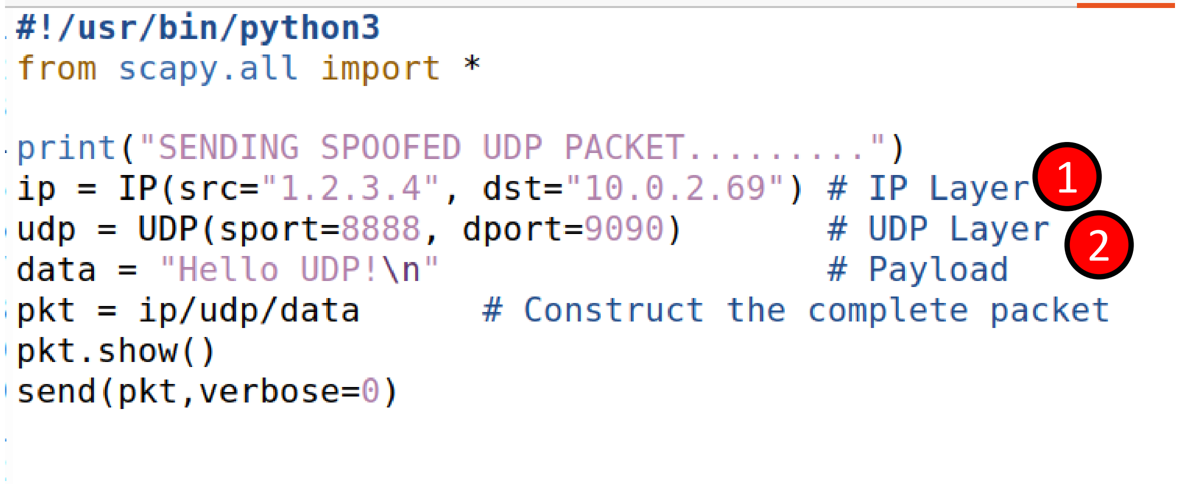
```
[03/20/23]seed@VM:~/.../sniff_spoof$ vi sniffer.py
[03/20/23]seed@VM:~/.../sniff_spoof$ sudo python3 sniffer.py
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
Ether / IP / ICMP 142.251.33.110 > 10.0.2.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
Ether / IP / ICMP 142.251.33.110 > 10.0.2.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
Ether / IP / ICMP 142.251.33.110 > 10.0.2.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
Ether / IP / ICMP 142.251.33.110 > 10.0.2.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
Ether / IP / ICMP 142.251.33.110 > 10.0.2.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
Ether / IP / ICMP 142.251.33.110 > 10.0.2.5 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.5 > 142.251.33.110 echo-request 0 / Raw
```

2. In another terminal, start generating ICMP packets



```
[03/20/23]seed@VM:~$ ping google.com
PING google.com (142.251.33.110) 56(84) bytes of data.
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=1 ttl=55 tim
=15.8 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=2 ttl=55 tim
=16.8 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=3 ttl=55 tim
=16.6 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=4 ttl=55 tim
=16.5 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=5 ttl=55 tim
=15.6 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=6 ttl=55 tim
=19.1 ms
```

We can see all the packets being sent in the `ping` request



```
#!/usr/bin/python3
from scapy.all import *

print("SENDING SPOOFED UDP PACKET.........")
ip = IP(src="1.2.3.4", dst="10.0.2.69") # IP Layer
udp = UDP(sport=8888, dport=9090)       # UDP Layer
data = "Hello UDP!\n"                   # Payload
pkt = ip/udp/data      # Construct the complete packet
pkt.show()
send(pkt,verbose=0)
```

1 We can set the packets source IP and destination IP

```
Souce ip: 1.2.3.4 (bogus)
Destination IP: 10.0.2.69 (also bogus)
```

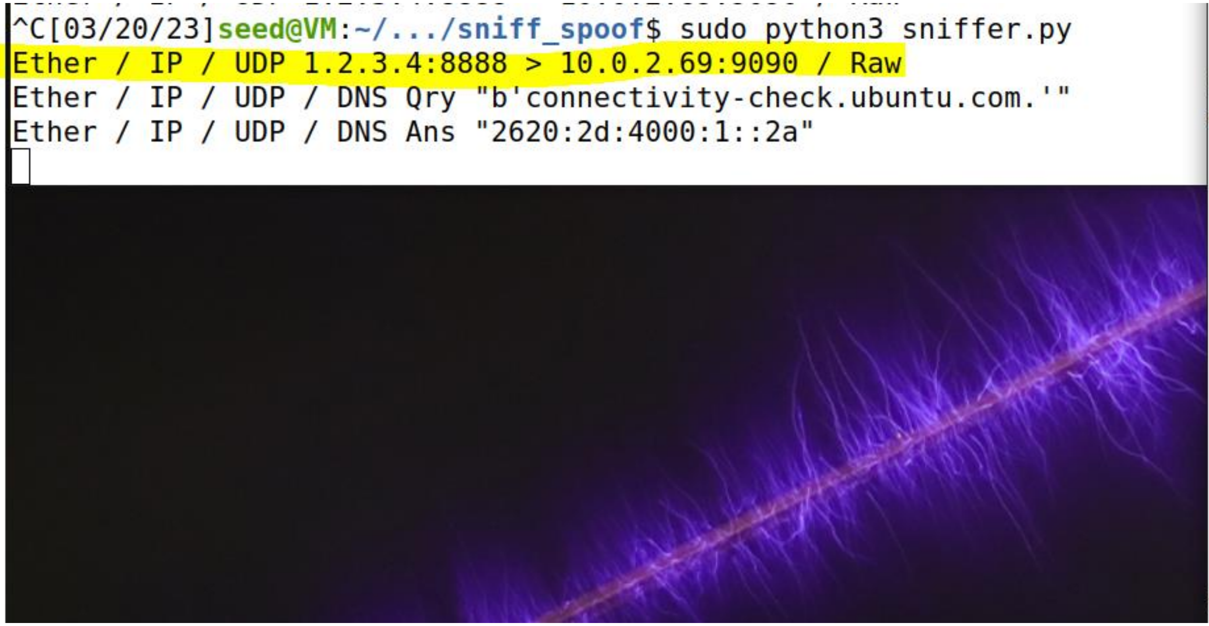
2 We can set the packets source port and destination port (udp)

```
Source port: 8888 (bogus)
Destination port: 9090 (also bogus)
```

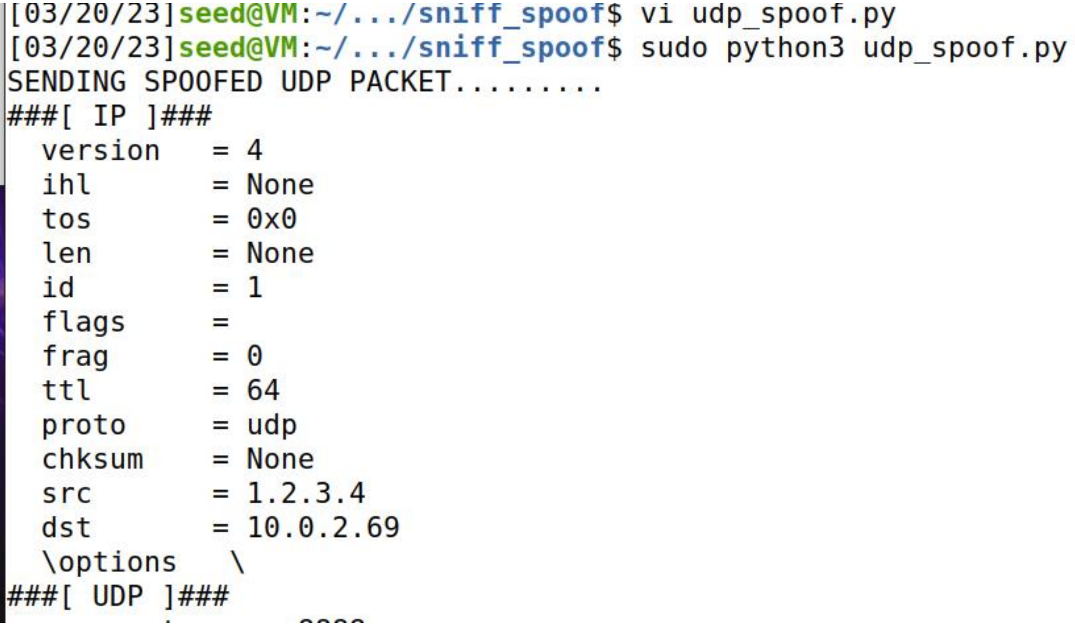
We can write a program that will craft and send out packets that we create

We can modify

- Src/dst IP address
- Port #s
- TCP Header information



```
^C[03/20/23]seed@VM:~/.../sniff_spoof$ sudo python3 sniffer.py
Ether / IP / UDP 1.2.3.4:8888 > 10.0.2.69:9090 / Raw
Ether / IP / UDP / DNS Qry "b'connectivity-check.ubuntu.com.'"
Ether / IP / UDP / DNS Ans "2620:2d:4000:1::2a"
```

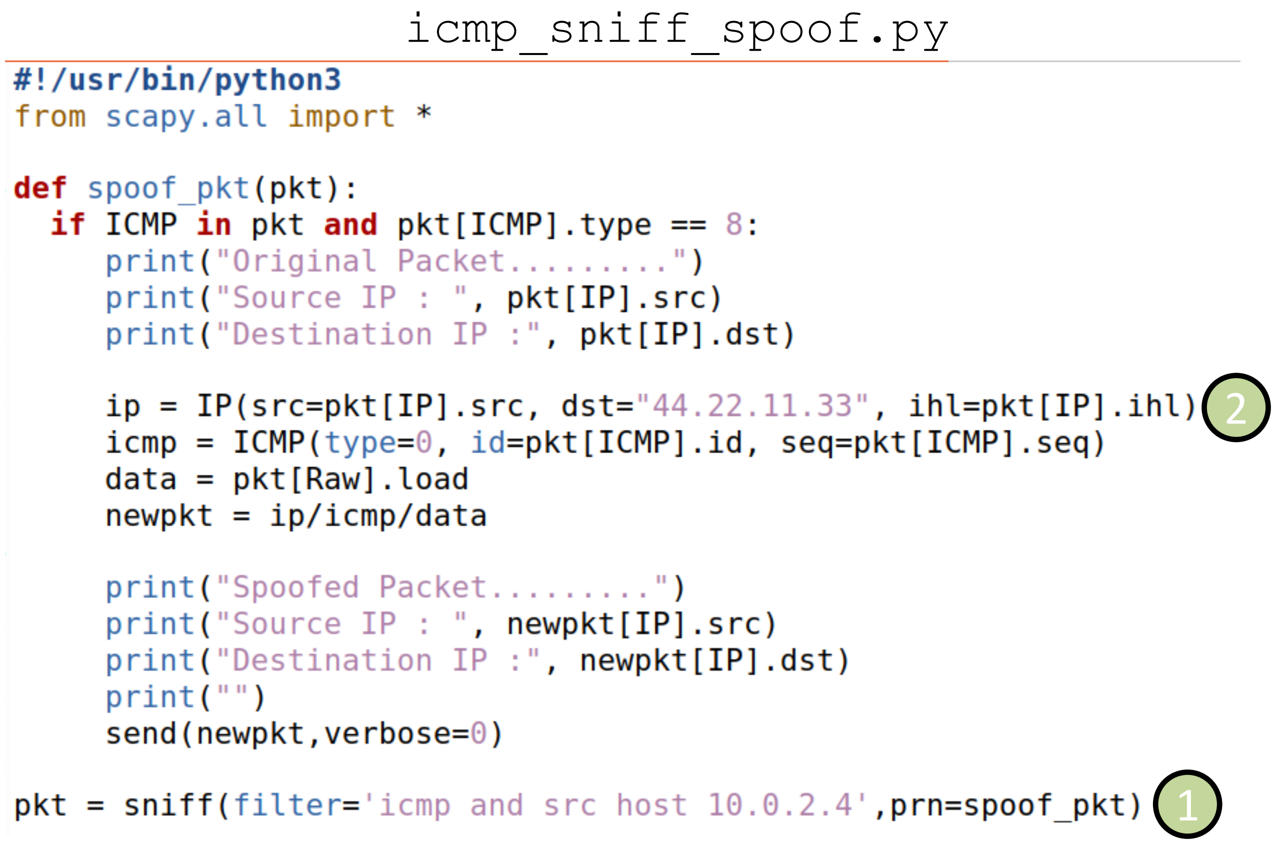


```
[03/20/23]seed@VM:~/.../sniff_spoof$ vi udp_spoof.py
[03/20/23]seed@VM:~/.../sniff_spoof$ sudo python3 udp_spoof.py
SENDING SPOOFED UDP PACKET.........
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = None
  src       = 1.2.3.4
  dst       = 10.0.2.69
  \options   \
###[ UDP ]###
```

1 Sniff/listen for ICMP packets coming from `10.0.2.4`

2 When we intercept an ICMP packet, extract the packets source IP, and then create a spoofed packet

- `44.22.11.33` will receive a packet from `10.0.2.4`



icmp_sniff_spoof.py

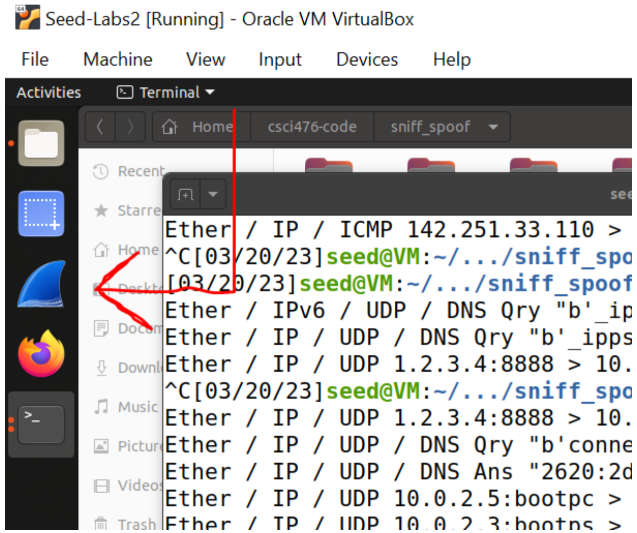
```
#!/usr/bin/python3
from scapy.all import *

def spoof_pkt(pkt):
  if ICMP in pkt and pkt[ICMP].type == 8:
     print("Original Packet.........")
     print("Source IP : ", pkt[IP].src)
     print("Destination IP :", pkt[IP].dst)

     ip = IP(src=pkt[IP].src, dst="44.22.11.33", ihl=pkt[IP].ihl)
     icmp = ICMP(type=0, id=pkt[ICMP].id, seq=pkt[ICMP].seq)
     data = pkt[Raw].load
     newpkt = ip/icmp/data

     print("Spoofed Packet.........")
     print("Source IP : ", newpkt[IP].src)
     print("Destination IP :", newpkt[IP].dst)
     print("")
     send(newpkt,verbose=0)

pkt = sniff(filter='icmp and src host 10.0.2.4',prn=spoof_pkt)
```

We can sniff for packets, and then spoof packets using the sniffed information!

Wireshark is a very popular network analysis tool that allows you to analyze and view network traffic

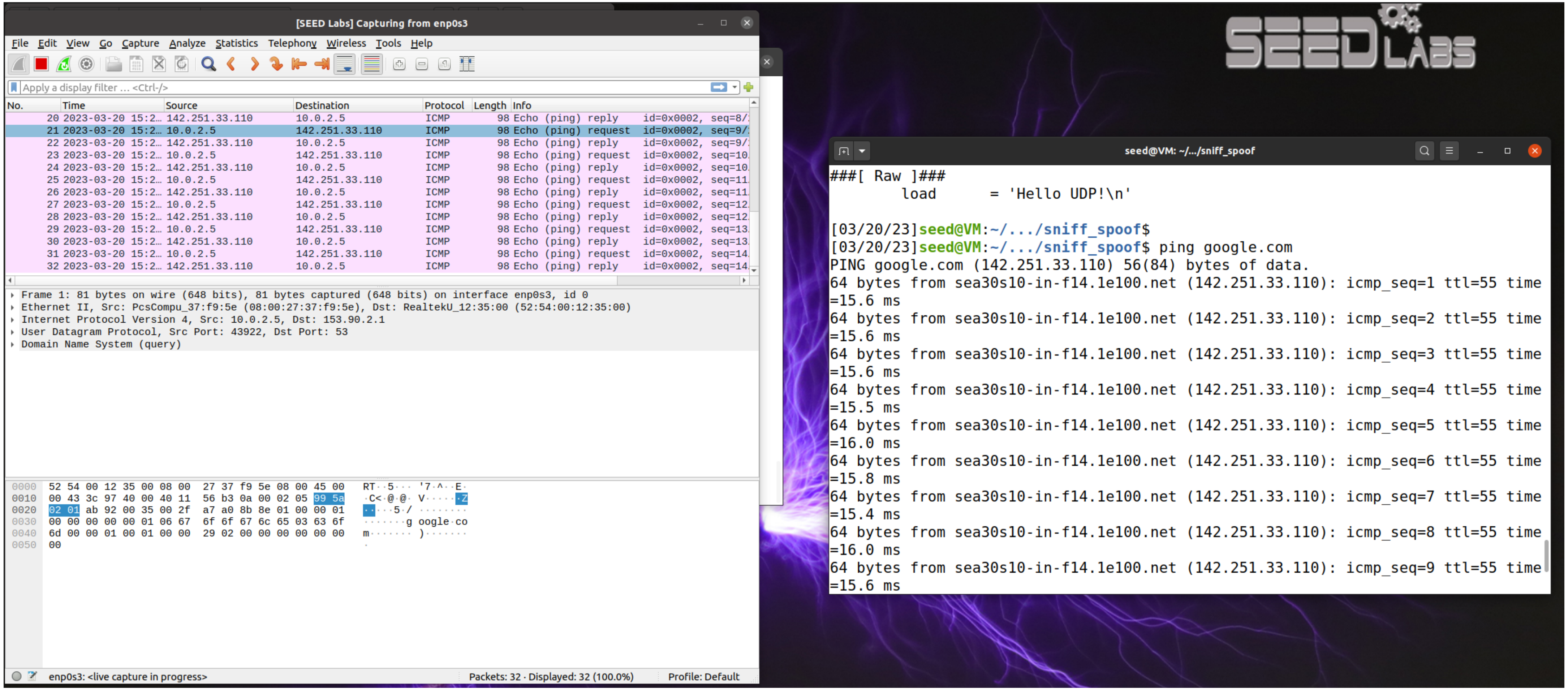
We will use Wireshark to sniff packets instead of Python ☺

And it's installed on your VM!!



```
Ether / IP / ICMP 142.251.33.110 >
^C[03/20/23]seed@VM:~/.../sniff_spo
[03/20/23]seed@VM:~/.../sniff_spoof
Ether / IPv6 / UDP / DNS Qry "b'_ip
Ether / IP / UDP / DNS Qry "b'_ipps
Ether / IP / UDP 1.2.3.4:8888 > 10.
^C[03/20/23]seed@VM:~/.../sniff_spo
Ether / IP / UDP 1.2.3.4:8888 > 10.
Ether / IP / UDP / DNS Qry "b'conne
Ether / IP / UDP / DNS Ans "2620:2d
Ether / IP / UDP 10.0.2.5:bootpc >
Ether / IP / UDP 10.0.2.3:bootps >
```

Sniffing packets using Wireshark



[SEED Labs] Capturing from enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>
SEED LABS
seed@VM: ~/.../sniff_spoof

No.	Time	Source	Destination	Protocol	Length	Info
20	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=8/
21	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=9/
22	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=9/
23	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=10
24	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=10
25	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=11
26	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=11
27	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=12
28	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=12
29	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=13
30	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=13
31	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=14
32	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=14

```
› Frame 1: 81 bytes on wire (648 bits), 81 bytes captured (648 bits) on interface enp0s3, id 0
› Ethernet II, Src: PcsCompu_37:f9:5e (08:00:27:37:f9:5e), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
› Internet Protocol Version 4, Src: 10.0.2.5, Dst: 153.90.2.1
› User Datagram Protocol, Src Port: 43922, Dst Port: 53
› Domain Name System (query)
```

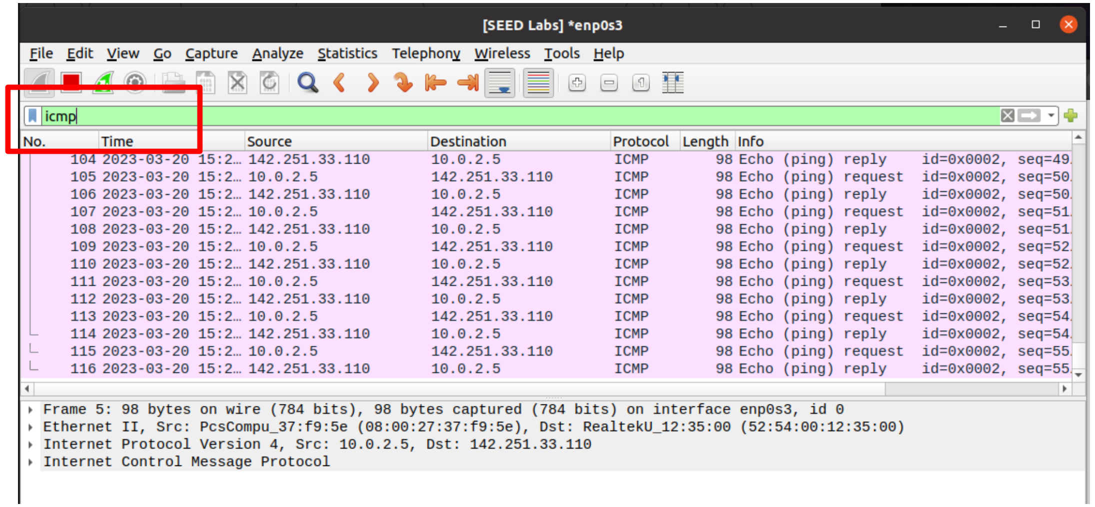
```
0000  52 54 00 12 35 00 08 00  27 37 f9 5e 08 00 45 00   RT··5··· '7·^··E·
0010  00 43 3c 97 40 00 40 11  56 b3 0a 00 02 05 99 5a   ·C<·@·@· V······Z
0020  02 01 ab 92 00 35 00 2f  a7 a0 8b 8e 01 00 00 01   ·····5·/ ········
0030  00 00 00 00 00 01 06 67  6f 6f 67 6c 65 03 63 6f   ·······g oogle·co
0040  6d 00 00 01 00 01 00 00  29 02 00 00 00 00 00 00   m······· )·······
0050  00                                                 ·
```

enp0s3: <live capture in progress>   Packets: 32 · Displayed: 32 (100.0%)   Profile: Default

```
###[ Raw ]###
        load      = 'Hello UDP!\n'

[03/20/23]seed@VM:~/.../sniff_spoof$
[03/20/23]seed@VM:~/.../sniff_spoof$ ping google.com
PING google.com (142.251.33.110) 56(84) bytes of data.
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=1 ttl=55 time
=15.6 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=2 ttl=55 time
=15.6 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=3 ttl=55 time
=15.6 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=4 ttl=55 time
=15.5 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=5 ttl=55 time
=16.0 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=6 ttl=55 time
=15.8 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=7 ttl=55 time
=15.4 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=8 ttl=55 time
=16.0 ms
64 bytes from sea30s10-in-f14.1e100.net (142.251.33.110): icmp_seq=9 ttl=55 time
=15.6 ms
```

Sniffing packets using Wireshark

We can apply filters in Wireshark to sniff for certain packers

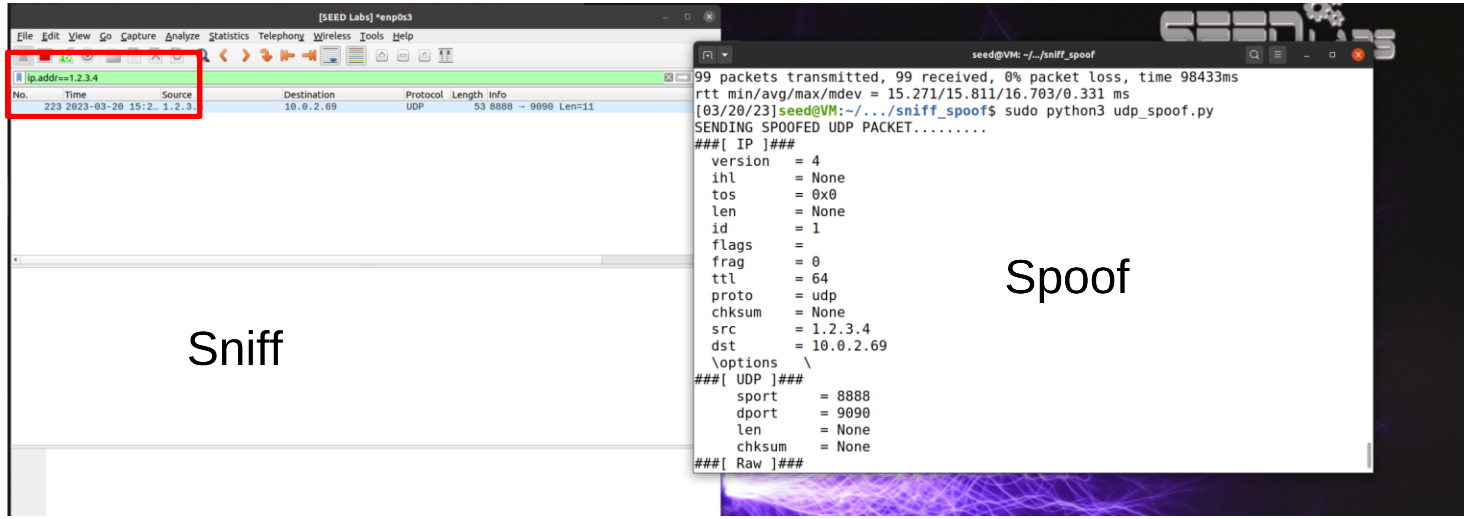

[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

icmp

No.	Time	Source	Destination	Protocol	Length	Info
104	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=49
105	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=50
106	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=50
107	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=51
108	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=51
109	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=52
110	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=52
111	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=53
112	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=53
113	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=54
114	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=54
115	2023-03-20 15:2…	10.0.2.5	142.251.33.110	ICMP	98	Echo (ping) request id=0x0002, seq=55
116	2023-03-20 15:2…	142.251.33.110	10.0.2.5	ICMP	98	Echo (ping) reply id=0x0002, seq=55

```
› Frame 5: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface enp0s3, id 0
› Ethernet II, Src: PcsCompu_37:f9:5e (08:00:27:37:f9:5e), Dst: RealtekU_12:35:00 (52:54:00:12:35:00)
› Internet Protocol Version 4, Src: 10.0.2.5, Dst: 142.251.33.110
› Internet Control Message Protocol
```

Show only ICMP packets


[SEED Labs] *enp0s3
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
seed@VM: ~/.../sniff_spoof

ip.addr==1.2.3.4

No.	Time	Source	Destination	Protocol	Length	Info
223	2023-03-20 15:2…	1.2.3.	10.0.2.69	UDP	53	8888 → 9090 Len=11

```
99 packets transmitted, 99 received, 0% packet loss, time 98433ms
rtt min/avg/max/mdev = 15.271/15.811/16.703/0.331 ms
[03/20/23]seed@VM:~/.../sniff_spoof$ sudo python3 udp_spoof.py
SENDING SPOOFED UDP PACKET.........
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = udp
  chksum    = None
  src       = 1.2.3.4
  dst       = 10.0.2.69
  \options   \
###[ UDP ]###
     sport     = 8888
     dport     = 9090
     len       = None
     chksum    = None
###[ Raw ]###
```

Sniff

Spoof

Show packets going to/coming from a certain IP address