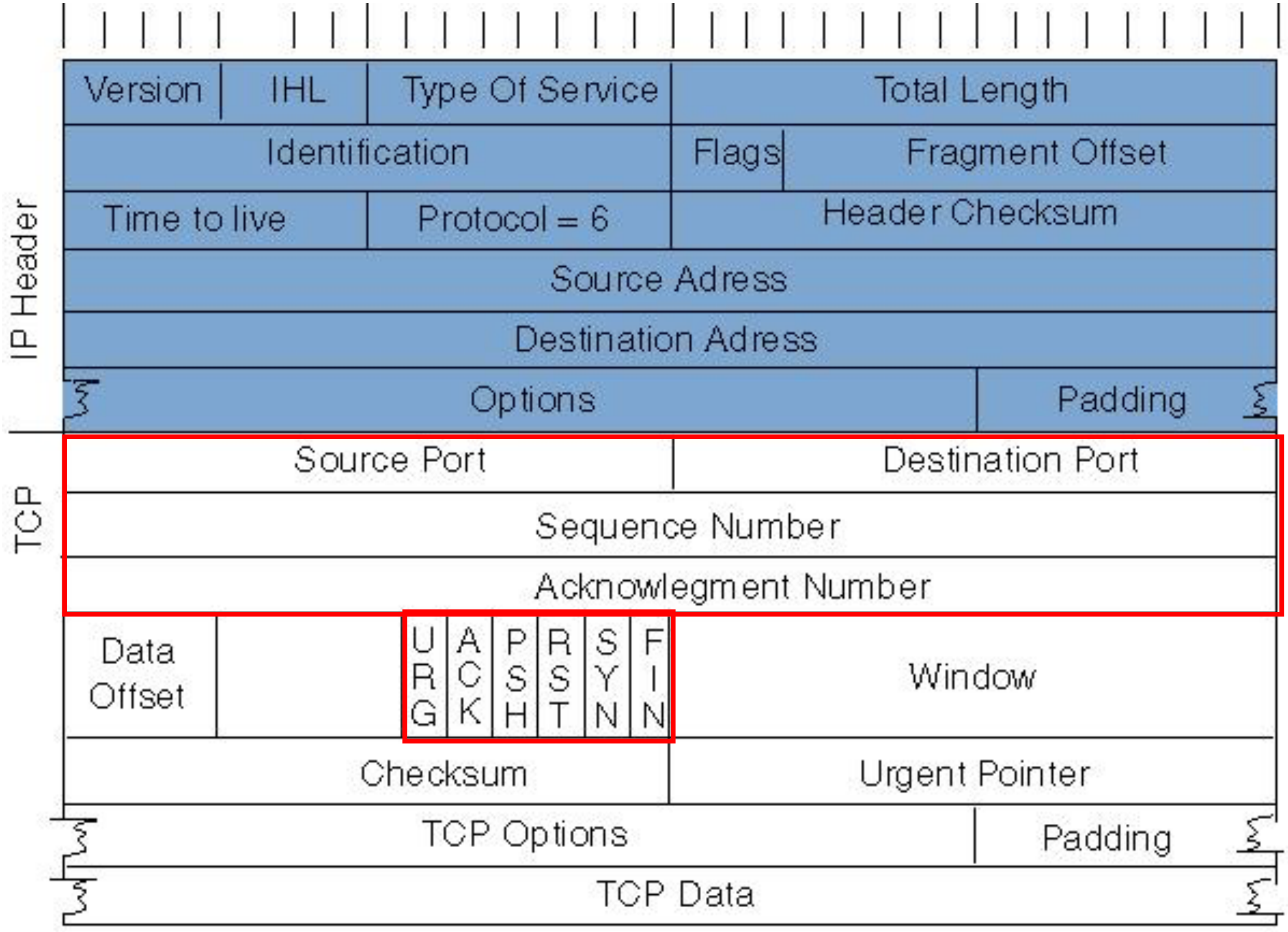
OSI Transport layer

Transmission Control Protocol (TCP)

- IP can only be used to send datagrams
  - ”chunks” or “streams” of information
  - From **sender IP** to **destination IP**
- TCP builds on top of IP the notion of “state”
  - Systems that communicate using the TCP protocol engage in a **stateful** communication
- IP → delivers the data
- TCP → manages the data **segments**
  - Checksums
  - Re-delivery of unreceived packets
  - Re-delivery of corrupt packets

TCP/IP header



IP Header

Version | IHL | Type Of Service | Total Length

Identification | Flags | Fragment Offset

Time to live | Protocol = 6 | Header Checksum

Source Adress

Destination Adress

Options | Padding

TCP

Source Port | Destination Port

Sequence Number

Acknowlegment Number

Data Offset | URG | ACK | PSH | RST | SYN | FIN | Window

Checksum | Urgent Pointer

TCP Options | Padding

TCP Data

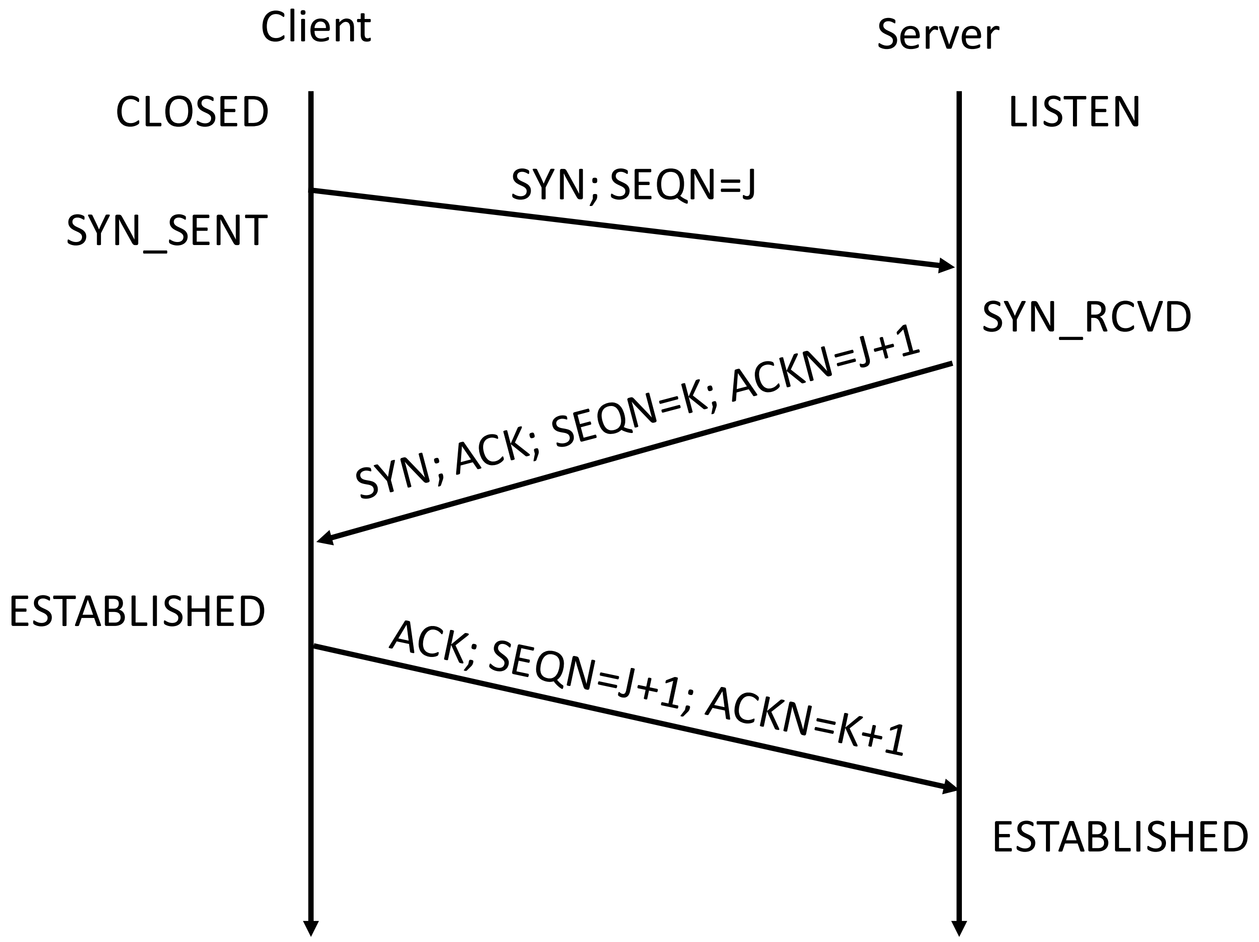
TCP basics (details → RFC 793)

- TCP is based on IP
- Server and client that participate in a TCP connection open a ”socket”
  - SOURCEIP:SOURCEPORT
  - DESTIP:DESTPORT
- A connection between a client and a server is identified by the tuple
  - <SOURCEIP:SOURCEPORT, DESTIP:DESTPORT >
- All TCP packets are directed toward a **port**
  - Common dest ports:
    - **SSH port 22**
    - **HTTP port 80**
    - **HTTPS port 443**
    - **FTP port 21**
  - Client usually generates source port randomly
  - LISTEN → service listening on port (open)
  - CLOSE → no service listening on port (closed)

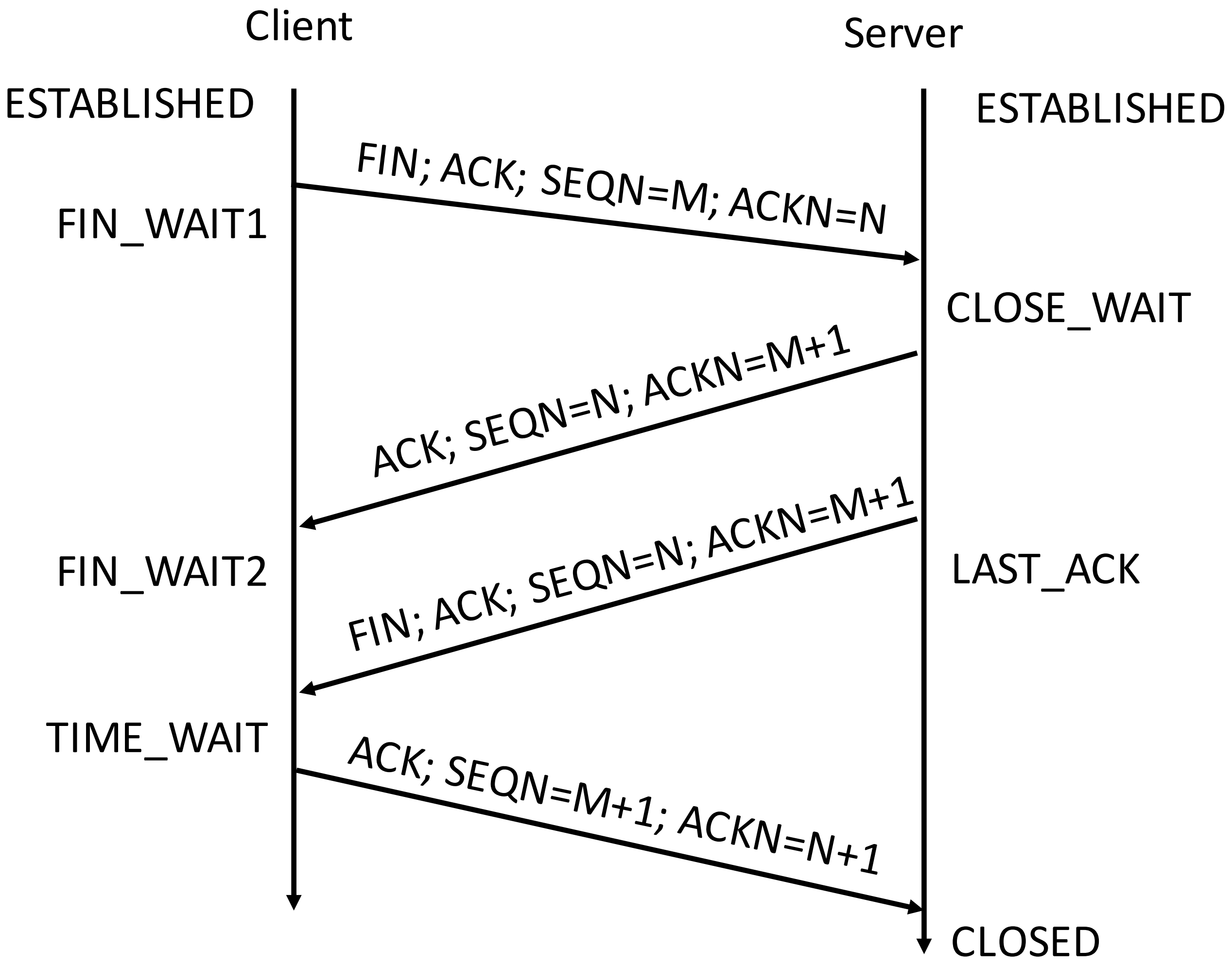
TCP, a few details

- **SYN**: initialize the TCP session → should be set to 1 only for first datagram by client and server
- **ACK**: acknowledge the reception of the segment
  - Associated with an ACK number
- **FIN**: signals intention to close the connection (end of data)
- **RST**: connection is dropped (reset)
- **Sequence number:** 32 bit number generated by each end
  - communication start (SYN=1)
    - Client_seq = J / Server_seq = K
  - During communication
    - SEQN = “this is packet x”
- **Acknowledgement number:** 32 bits
  - ACKN = “expecting x+1”

TCP 3-way handshake (SYN)



TCP 4-way handshake (FIN)



Keeping track of TCP connections

- The server receives a SYN request → SYN_RCVD
- Must keep track of this in order to establish a connection → ESTABLISHED
- Both ends set up a "**Transmission Control Block**" (TCB) to keep track of connection
  - Special data structure that stores information about connection
    - Sockets, seq numbers, pointers to buffer in memory
- → Allocate memory buffer to store data that will arrive
- TCB structure is freed from memory when connection reaches status CLOSED

Some TCP specifics

- A packet with RST flag up does not receive an answer
- CLOSED state
  - ANY packet with no RST receives a RST
- LISTEN state
  - A packet with SYN flag up and no ACK opens a TCP session. Answer is SYN+ACK
  - A packet with only ACK receives a RST
  - Drop with no answer otherwise
- An unsolicited SYN+ACK gets a RST regardless of listening state

SYN Denial of service attack

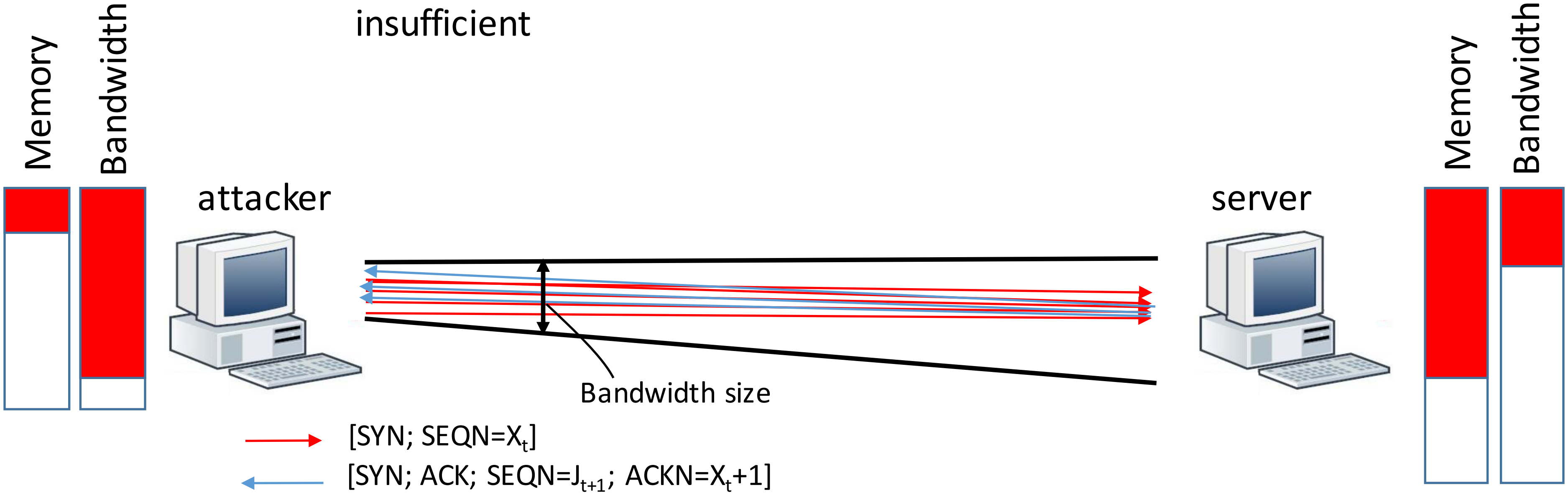
- When the server receives SYN J, it answers back with SYN K, ACK J+1
- Server opens new session in separate thread / allocates resources
  - Transmission control block allocation
- Server then waits for ACK K+1 from client
  - How long to wait before sending RST back?
    - Maximum Segment Lifetime (MSL) → set by default to 2 minutes
- Same mechanism sender side
  - Attacker controls the system, so it may bypass it

SYN Flood DoS, naïve solution

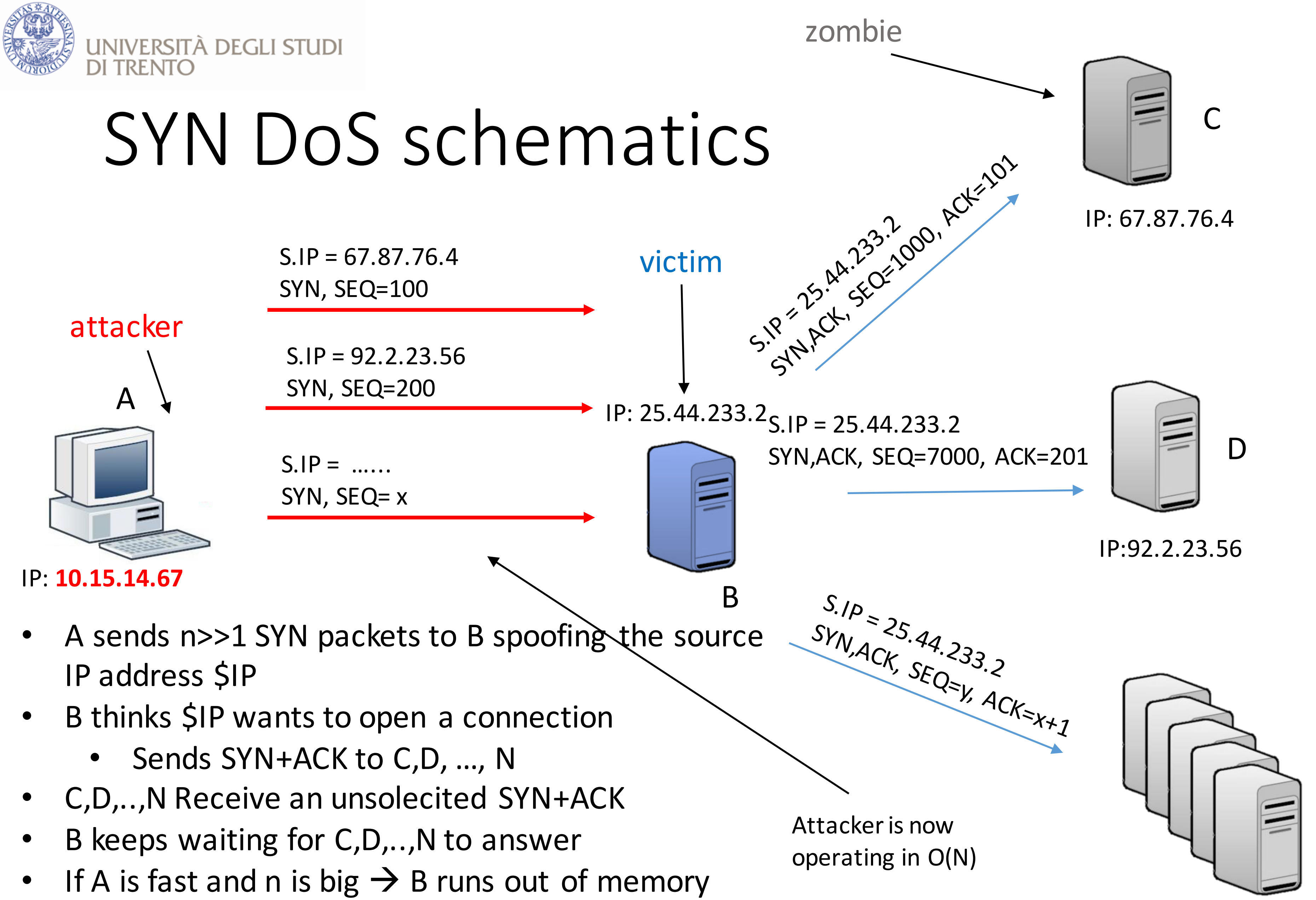
Server typically has more bandwidth available than single client

Client can drop all SYN ACKs (e.g. with a firewall) to not exhaust its own memory, but throughput necessarily slows down by O(2N)

- for each SYN, get a SYN ACK → bandwidth quickly decays
- Must exhaust server's memory before throughput becomes insufficient



SYN DoS schematics



- A sends n>>1 SYN packets to B spoofing the source IP address $IP
- B thinks $IP wants to open a connection
  - Sends SYN+ACK to C,D, …, N
- C,D,..,N Receive an unsolecited SYN+ACK
- B keeps waiting for C,D,..,N to answer
- If A is fast and n is big → B runs out of memory

Denial of service limitations

- In theory this attack should not work. Why?
  - B should receive a RST by each zombie → would free TCB → no DoS
    - Attacker can choose destination IPs that do not reply
    - Firewalls may simply drop the packet with no RST
    - Some IPs may actually not be in use
      - In theory this will generate an ICMP packet (host not reachable) and close the connection.
      - RFC 1122: A Destination Unreachable message that is received MUST be reported to the transport layer.
- SYN packets must arrive at very high rates
- Other more sophisticated techniques exist
  - Distributed Denial of Service (nowadays more common)
  - Coremelt DoS
  - We'll see these

DoS Mitigation (pointers)

- Load balancing → distribute traffic loads evenly
- Rate limiter → deny traffic above a certain rate of SYN/sec
- Proof of work → require source to solve a crypto puzzle before allocating resources to connection
  - Requires protocol support

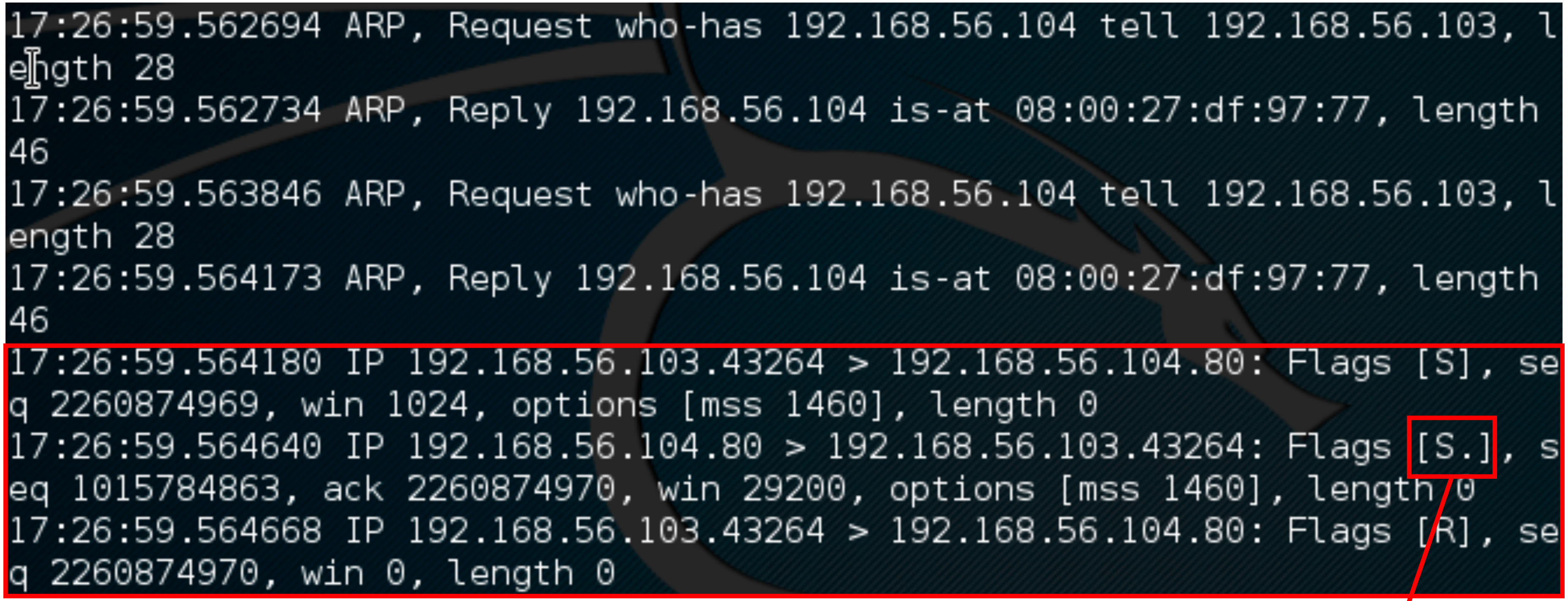
Network scans

- It's possible to exploit specifications of a network protocol (TCP, UDP,..) to learn something about a system or a network
- Some examples:
  - Build a list of services running on a remote system
  - Infer a network's structure
  - Build a list of zombie IPs that do not send RST back
- Several types of scans
- Several popular tools to do one
  - nmap

SYN Scan

- Attacker forges TCP packets
  - SYN=1
- Useful to measure whether remote system accepts incoming connections on port=x
  - Typically this corresponds to a specific service
  - SYN ACK from port 22 → SSH is likely listening
  - SYN ACK from port 80 → HTTP server is likely listening
  - RST → port x is closed on remote system
- Half-open SYN scan
  - After server's SYN ACK reply, attacker sends RST
  - 3-way handshake is never finished

Example of Half-open connection



```
17:26:59.562694 ARP, Request who-has 192.168.56.104 tell 192.168.56.103, l
ength 28
17:26:59.562734 ARP, Reply 192.168.56.104 is-at 08:00:27:df:97:77, length
46
17:26:59.563846 ARP, Request who-has 192.168.56.104 tell 192.168.56.103, l
ength 28
17:26:59.564173 ARP, Reply 192.168.56.104 is-at 08:00:27:df:97:77, length
46
17:26:59.564180 IP 192.168.56.103.43264 > 192.168.56.104.80: Flags [S], se
q 2260874969, win 1024, options [mss 1460], length 0
17:26:59.564640 IP 192.168.56.104.80 > 192.168.56.103.43264: Flags [S.], s
eq 1015784863, ack 2260874970, win 29200, options [mss 1460], length 0
17:26:59.564668 IP 192.168.56.103.43264 > 192.168.56.104.80: Flags [R], se
q 2260874970, win 0, length 0
```

From http://www.tcpdump.org/tcpdump_man.html
Flags are some combination of S (SYN), F (FIN), P (PUSH), R (RST), U (URG), W (ECN CWR), E (ECN-Echo) or `.' (ACK), or `none' if no flags are set.

Host fingerprinting

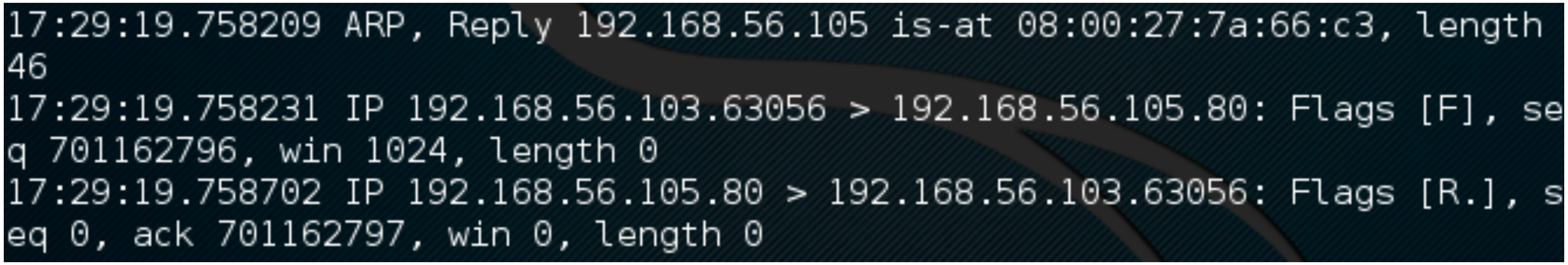
- RFC 793 is the reference document for TCP stack implementation
- However, not all specifications are always implemented as stated
- Different operating systems have their own independent implementation
  - It's possible to infer which operating system is on the other side on the basis of the received answers
  - Technique is called **fingerprinting**

FIN/Xmas/Null scan

- An example of scan that allows for some level of fingerprinting
  - FIN → flag FIN = 1
  - Null → all flags = 0
  - Xmas → FIN, URG, PSH = 1
- From RFC
  - Port is OPEN → DROP, no answer
  - Port is CLOSED → DROP, RST
- For example, Windows XP, HP/UX
  - Always reply RST

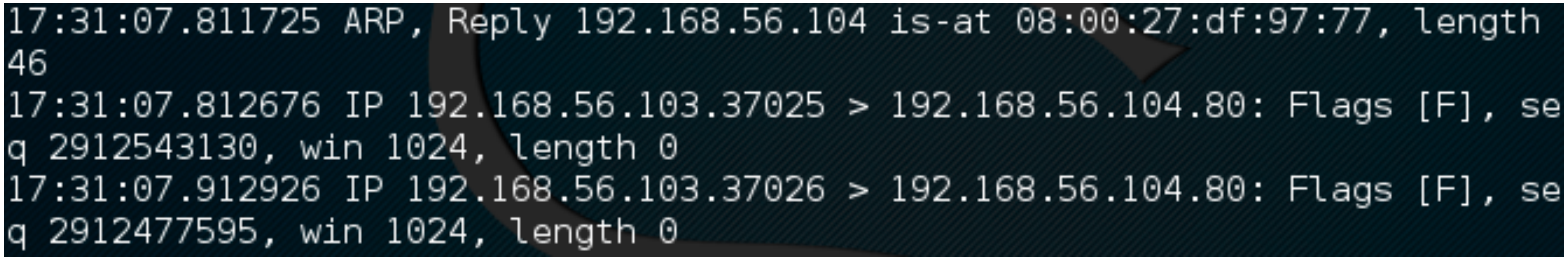
Different hosts, different answers

Windows XP 64bit sp0 (192.168.54.105)



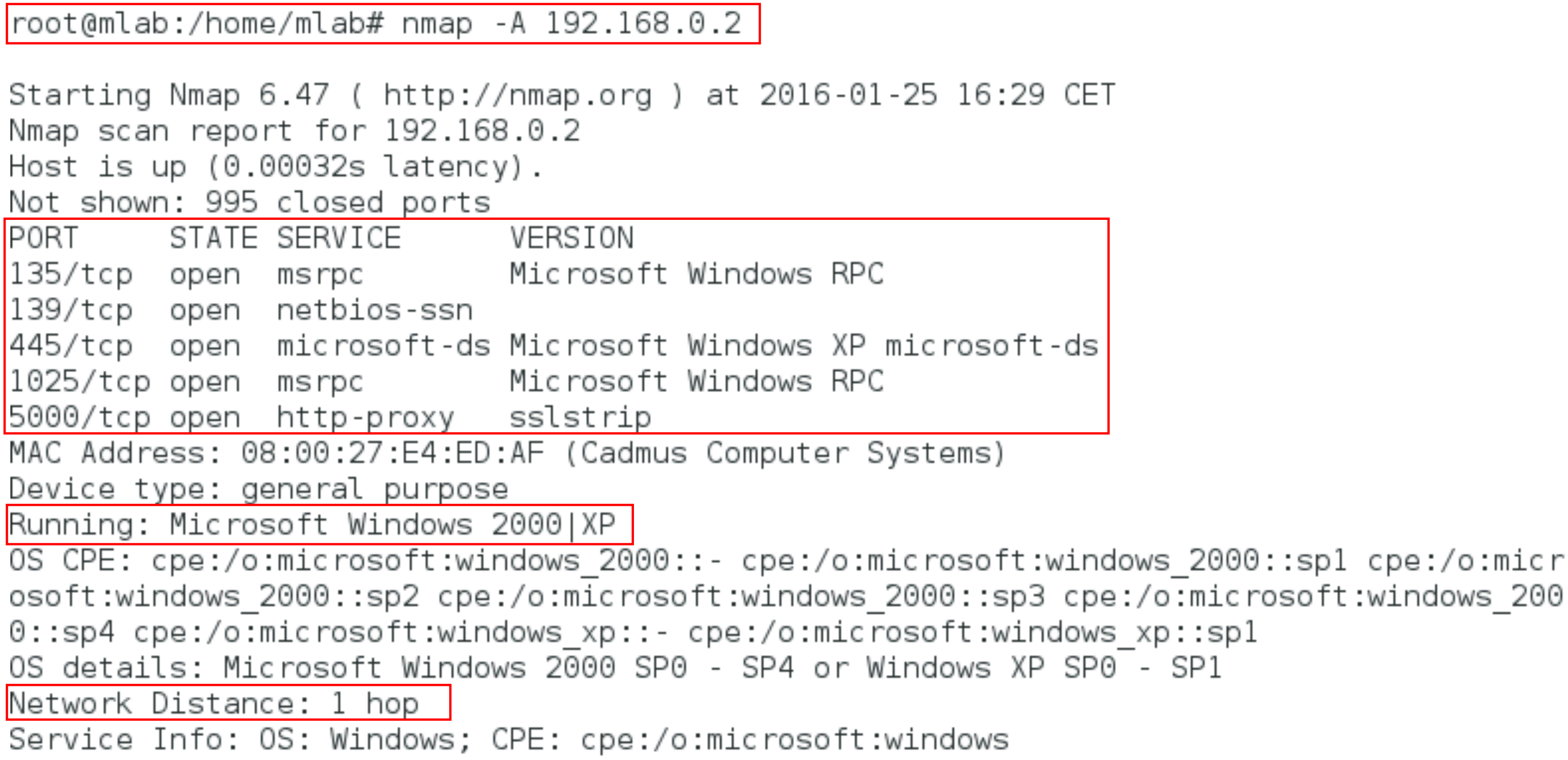
```
17:29:19.758209 ARP, Reply 192.168.56.105 is-at 08:00:27:7a:66:c3, length
46
17:29:19.758231 IP 192.168.56.103.63056 > 192.168.56.105.80: Flags [F], se
q 701162796, win 1024, length 0
17:29:19.758702 IP 192.168.56.105.80 > 192.168.56.103.63056: Flags [R.], s
eq 0, ack 701162797, win 0, length 0
```

Debian Linux 3.16.04-amd64 (192.168.54.104)



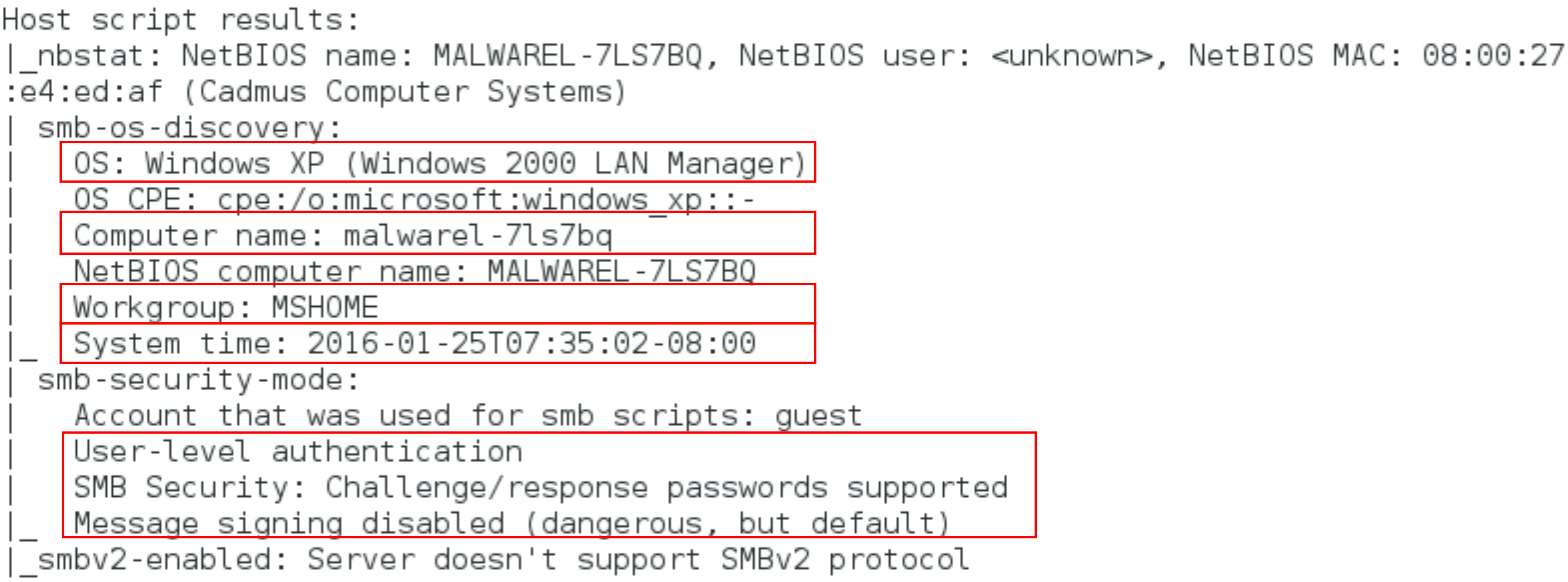
```
17:31:07.811725 ARP, Reply 192.168.56.104 is-at 08:00:27:df:97:77, length
46
17:31:07.812676 IP 192.168.56.103.37025 > 192.168.56.104.80: Flags [F], se
q 2912543130, win 1024, length 0
17:31:07.912926 IP 192.168.56.103.37026 > 192.168.56.104.80: Flags [F], se
q 2912477595, win 1024, length 0
```

Fingerprinting - An example



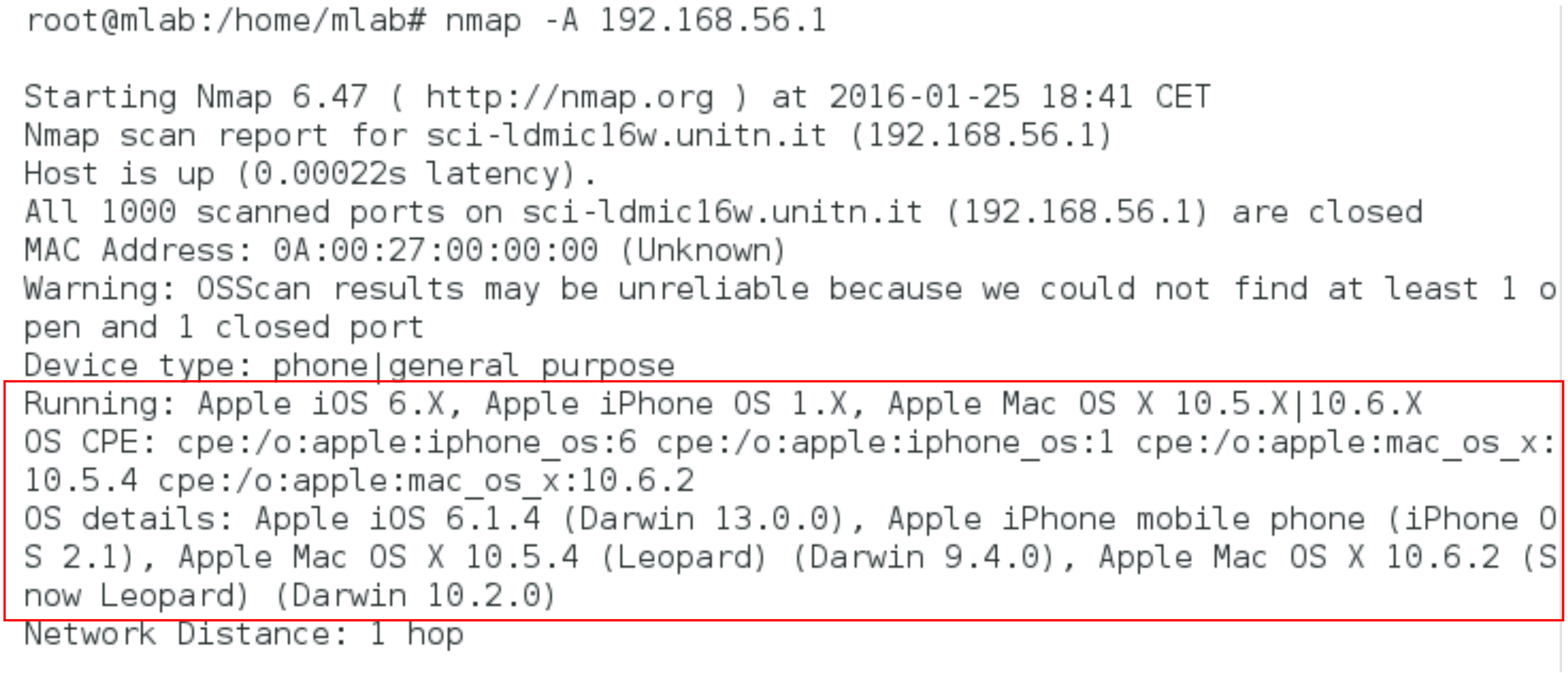
```
root@mlab:/home/mlab# nmap -A 192.168.0.2

Starting Nmap 6.47 ( http://nmap.org ) at 2016-01-25 16:29 CET
Nmap scan report for 192.168.0.2
Host is up (0.00032s latency).
Not shown: 995 closed ports
PORT     STATE SERVICE      VERSION
135/tcp  open  msrpc        Microsoft Windows RPC
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds Microsoft Windows XP microsoft-ds
1025/tcp open  msrpc        Microsoft Windows RPC
5000/tcp open  http-proxy   sslstrip
MAC Address: 08:00:27:E4:ED:AF (Cadmus Computer Systems)
Device type: general purpose
Running: Microsoft Windows 2000|XP
OS CPE: cpe:/o:microsoft:windows_2000::- cpe:/o:microsoft:windows_2000::sp1 cpe:/o:micr
osoft:windows_2000::sp2 cpe:/o:microsoft:windows_2000::sp3 cpe:/o:microsoft:windows_200
0::sp4 cpe:/o:microsoft:windows_xp::- cpe:/o:microsoft:windows_xp::sp1
OS details: Microsoft Windows 2000 SP0 - SP4 or Windows XP SP0 - SP1
Network Distance: 1 hop
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows
```

And it’s not finished..



```
Host script results:
|_nbstat: NetBIOS name: MALWAREL-7LS7BQ, NetBIOS user: <unknown>, NetBIOS MAC: 08:00:27
:e4:ed:af (Cadmus Computer Systems)
| smb-os-discovery:
|   OS: Windows XP (Windows 2000 LAN Manager)
|   OS CPE: cpe:/o:microsoft:windows_xp::-
|   Computer name: malwarel-7ls7bq
|   NetBIOS computer name: MALWAREL-7LS7BQ
|   Workgroup: MSHOME
|_  System time: 2016-01-25T07:35:02-08:00
| smb-security-mode:
|   Account that was used for smb scripts: guest
|   User-level authentication
|   SMB Security: Challenge/response passwords supported
|_  Message signing disabled (dangerous, but default)
|_smbv2-enabled: Server doesn't support SMBv2 protocol
```

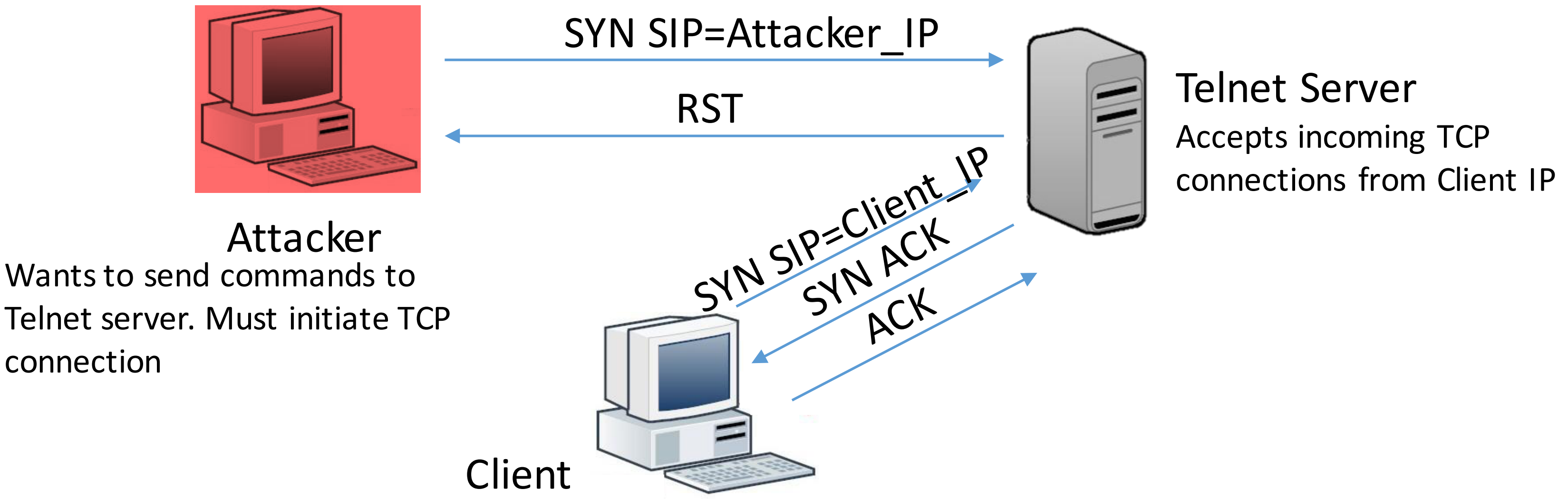
Not only XP



```
root@mlab:/home/mlab# nmap -A 192.168.56.1

Starting Nmap 6.47 ( http://nmap.org ) at 2016-01-25 18:41 CET
Nmap scan report for sci-ldmic16w.unitn.it (192.168.56.1)
Host is up (0.00022s latency).
All 1000 scanned ports on sci-ldmic16w.unitn.it (192.168.56.1) are closed
MAC Address: 0A:00:27:00:00:00 (Unknown)
Warning: OSScan results may be unreliable because we could not find at least 1 o
pen and 1 closed port
Device type: phone|general purpose
Running: Apple iOS 6.X, Apple iPhone OS 1.X, Apple Mac OS X 10.5.X|10.6.X
OS CPE: cpe:/o:apple:iphone_os:6 cpe:/o:apple:iphone_os:1 cpe:/o:apple:mac_os_x:
10.5.4 cpe:/o:apple:mac_os_x:10.6.2
OS details: Apple iOS 6.1.4 (Darwin 13.0.0), Apple iPhone mobile phone (iPhone O
S 2.1), Apple Mac OS X 10.5.4 (Leopard) (Darwin 9.4.0), Apple Mac OS X 10.6.2 (S
now Leopard) (Darwin 10.2.0)
Network Distance: 1 hop
```

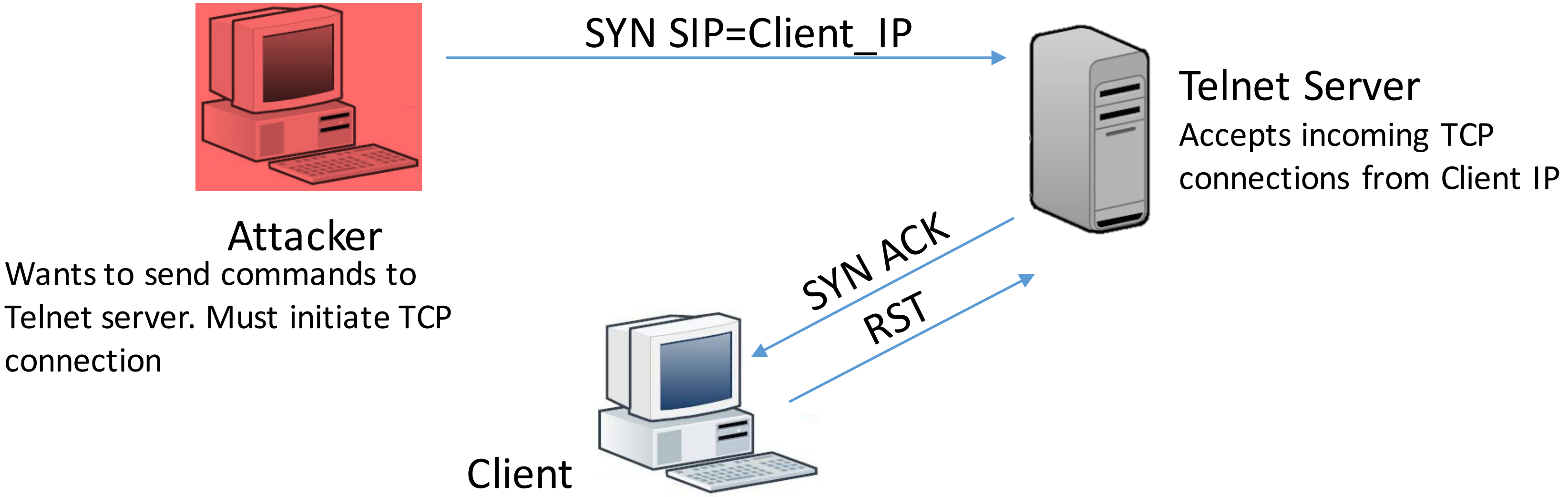
More advanced attacks – TCP Session Hijacking

- Goal → the attacker wants to send commands to a server they have no access to
  - client is authorized (e.g. simple IP address authentication)
  - the server must think that the attacker is the client
  - but the attacker does not sit in between client and server..


SYN SIP=Attacker_IP
RST
Telnet Server
Accepts incoming TCP connections from Client IP
Attacker
Wants to send commands to Telnet server. Must initiate TCP connection
SYN SIP=Client_IP
SYN ACK
ACK
Client

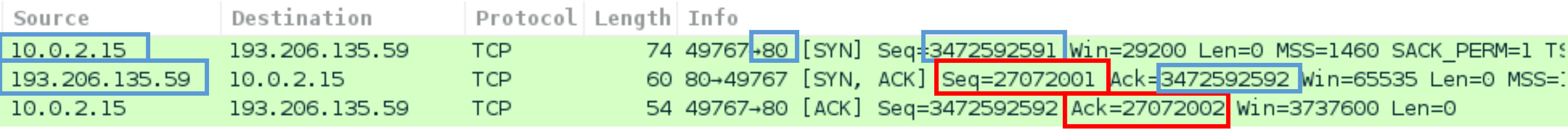
More advanced attacks – TCP Session Hijacking

- Goal → the attacker wants to send commands to a server they have no access to
  - client is authorized (e.g. simple IP address authentication)
  - the server must think that the attacker is the client
  - but the attacker does not sit in between client and server..


SYN SIP=Client_IP
Telnet Server
Accepts incoming TCP connections from Client IP
Attacker
Wants to send commands to Telnet server. Must initiate TCP connection
SYN ACK
RST
Client

How can the attacker circumvent this?

- By pretending he is the client!
- A TCP segment between a client and a server is identified and validated by
  - Client IP → known (public)
  - Destination IP → known (public)
  - Port → known (public – if not standard, scan)
  - Client SEQ number → known (attacker generates it)
  - Server SEQ number → unknown (randomly generated by server and sent to $CLIENT_IP)



Source	Destination	Protocol	Length	Info
10.0.2.15	193.206.135.59	TCP	74	49767→80 [SYN] Seq=3472592591 Win=29200 Len=0 MSS=1460 SACK_PERM=1 T
193.206.135.59	10.0.2.15	TCP	60	80→49767 [SYN, ACK] Seq=27072001 Ack=3472592592 Win=65535 Len=0 MSS=
10.0.2.15	193.206.135.59	TCP	54	49767→80 [ACK] Seq=3472592592 Ack=27072002 Win=3737600 Len=0

Sequence number prediction

From RFC 793:

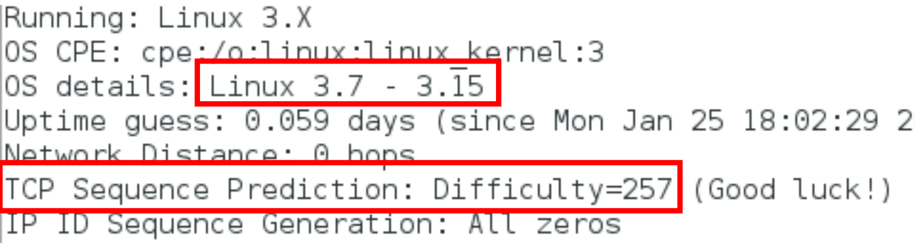
- *When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. The generator is bound to a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds.*

- Original BSD Unix implementation:
  - *Increment by n units / second*
  - *Increment by n/2 units per new TCP connection*

- Nowadays implementations are (closer to) a random number generator

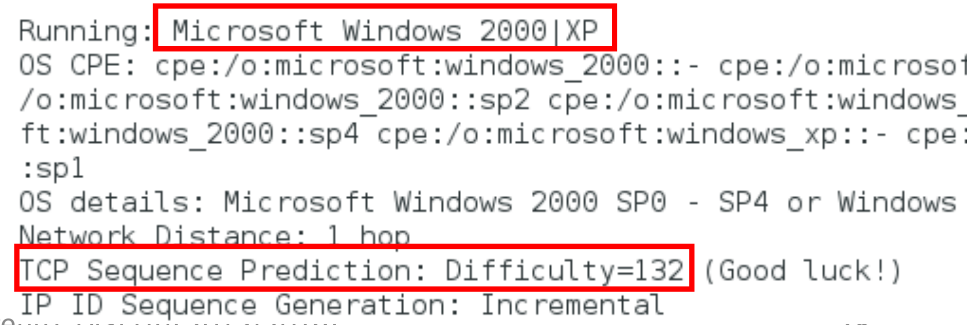
Mitnick attack

- In order to impersonate the client, the attacker only needs to correctly guess the server's SEQ number
  - $1/2^{32}$ chances of getting it right
    - Assuming perfect implementation of server's random number generator
  - In realty this may be much simpler
    - "TCP Sequence prediction"



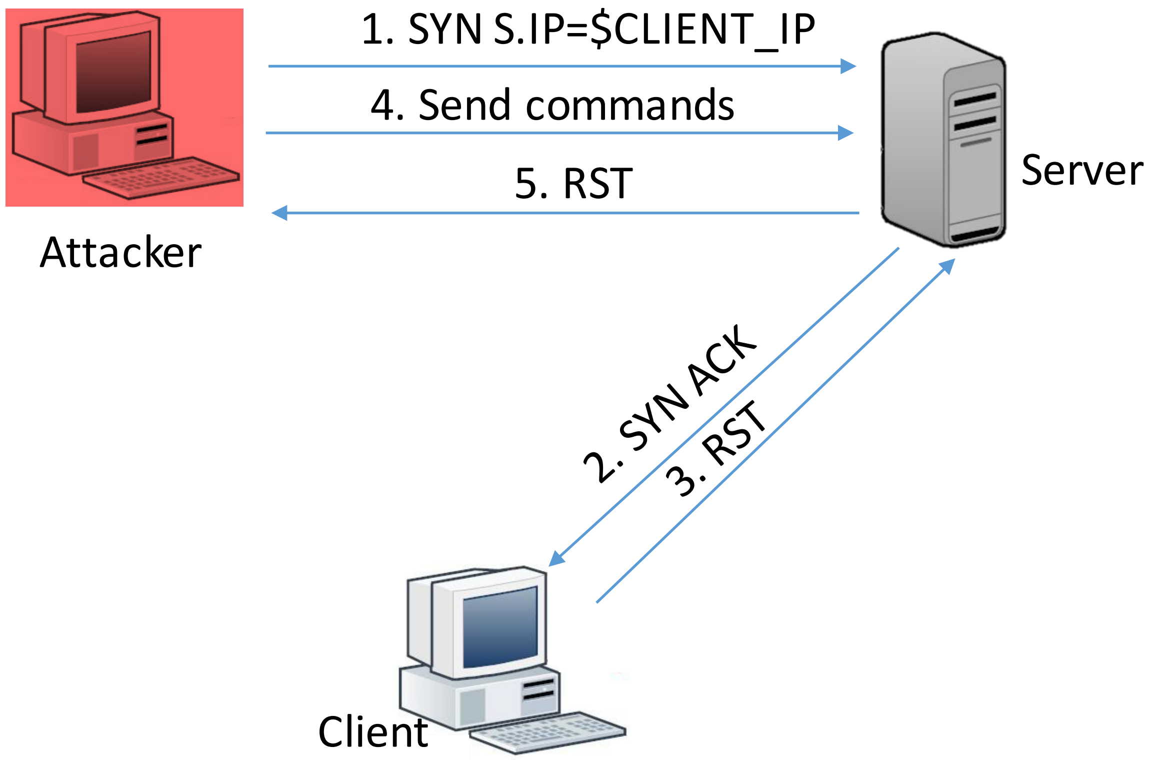
```
Running: Linux 3.X
OS CPE: cpe:/o:linux:linux_kernel:3
OS details: Linux 3.7 - 3.15
Uptime guess: 0.059 days (since Mon Jan 25 18:02:29 2016)
Network Distance: 0 hops
TCP Sequence Prediction: Difficulty=257 (Good luck!)
IP ID Sequence Generation: All zeros
```

Windows '95
Difficulty=1

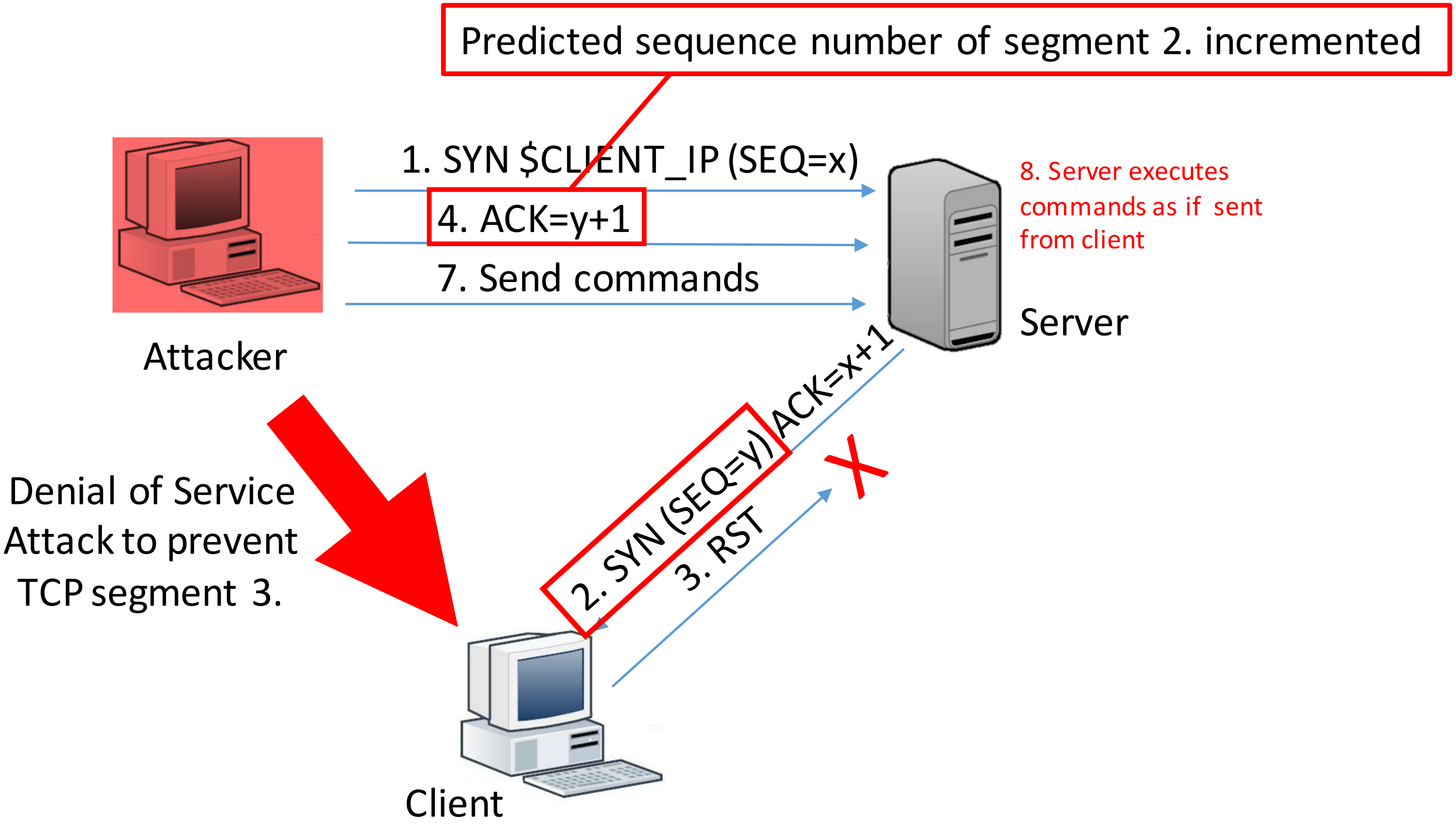


```
Running: Microsoft Windows 2000|XP
OS CPE: cpe:/o:microsoft:windows_2000::- cpe:/o:microsoi
/o:microsoft:windows_2000::sp2 cpe:/o:microsoft:windows_
ft:windows_2000::sp4 cpe:/o:microsoft:windows_xp::- cpe:
:sp1
OS details: Microsoft Windows 2000 SP0 - SP4 or Windows
Network Distance: 1 hop
TCP Sequence Prediction: Difficulty=132 (Good luck!)
IP ID Sequence Generation: Incremental
```

Mitnick attack – the problem

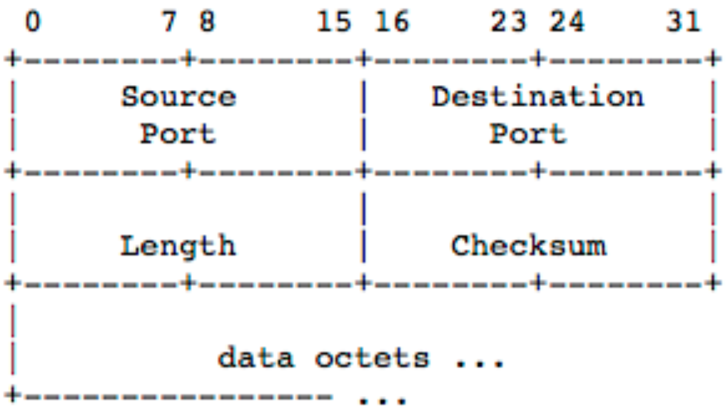


Mitnick attack - the solution


Predicted sequence number of segment 2. incremented
1. SYN $CLIENT_IP (SEQ=x)
4. ACK=y+1
7. Send commands
8. Server executes commands as if sent from client
Attacker
Server
Denial of Service Attack to prevent TCP segment 3.
2. SYN (SEQ=y) ACK=x+1
3. RST
X
Client

User Datagram Protocol

- Differently from TCP, UDP is a *stateless* protocol
- Fast delivery of data
  - Data integrity can be controlled at application level
  - Relies on reliability of underlying network link
  - Does not guarantee delivery (no acknowledgment mechanism)



```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

UDP usage

- UDP is used by some of the most important infrastructures of the Internet
  - DNS servers → to resolve internet domains
  - NFS (Network File System) → distributed FS
  - SNMP (Simple Network Management Protocol) → management of IP devices on a network
  - DHCP (Dynamic Host Configuration Protocol) → assign IP addresses to network devices
  - Most real-time applications (real-time transactions, DBs, etc..)

UDP scans

- Interesting as many core services are running over UDP and listening to UDP ports
- Can be used to discover (likely) open ports on the network
  - CLOSED → ICMP port unreachable
  - OPEN → no answer
- Prone to errors
  - ICMP packet can be filtered or dropped
    - Firewalls/routers
  - Possible to configure a ”stealth” system that does not reply to UDP requests to CLOSED ports