Lab: Exploring TCP with Scapy

In this lab you will explore TCP (RFC 9293) by crafting raw packets with Scapy. You will perform a 3-way handshake, negotiate a TCP option, send data, observe how the receiver handles out-of-order segments, and interact with two simple application protocols that run over TCP.

Prerequisites

- Python 3 with Scapy installed in the virtual environment
- Root privileges (`sudo`) — Scapy requires raw socket access
- `iptables` and `ip6tables` — the scripts manage these automatically
- `tshark` and Wireshark for optional packet capture and analysis

> **Note**
>
> All scripts in `tcp/` must be run with `sudo`. Each script installs a narrow `iptables` / `ip6tables` rule that drops the RST packets the kernel would otherwise send in response to server traffic (the kernel has no state for connections owned by Scapy). The rule is removed automatically when the script exits.

Test server

All exercises use the server `test.computer-networking.info`. Pass its address directly on the command line — do not use the hostname, as the scripts expect a numeric IP address.

Address family	Address
IPv4	`51.68.231.111`
IPv6	`2001:41d0:305:2100::4857`

The following TCP ports are available on the server:

Port	Purpose
8080	HTTP — used to test the handshake (any open port would do)
9	Discard service (RFC 863)
7	Port 7 service (RFC 862)
13	Port 13 service (RFC 867)

The TCP 3-way Handshake

TCP is connection-oriented: two endpoints must agree on a connection before any data can flow. The connection setup uses a **3-way handshake** defined in RFC 9293.

Complete the handshake script

Open `tcp/scapy-tcp-handshake-student.py` and fill in the six `FIXME` markers.

The first three concern the SYN segment:

- **FIXME 1** — the Initial Sequence Number (ISN)
- **FIXME 2** — the TCP flags for a SYN segment

- **FIXME 3** — the TCP options list; read RFC 2018 Section 2 to understand the SACK-Permitted option and what value it carries

The last three concern the ACK that completes the handshake:

- **FIXME 4** — the sequence number for the ACK
- **FIXME 5** — the acknowledgment number for the ACK
- **FIXME 6** — the TCP flags for the ACK

Once all FIXMEs are filled in, run the script:

```
sudo python3 tcp/scapy-tcp-handshake-student.py \
    --host 51.68.231.111 --port 8080
```

Then try with the IPv6 address:

```
sudo python3 tcp/scapy-tcp-handshake-student.py \
    --host 2001:41d0:305:2100::4857 --port 8080
```

The script prints the full header of each of the three segments via `show2()`, which displays all fields with computed values (including the checksum). Examine the output and answer:

- What ISN did you choose? What ISN did the server choose?
- What is the relationship between the SYN segment's sequence number and the ACK number in the SYN-ACK?
- What is the relationship between the server's ISN and the ACK number in your final ACK?
- Does the server include the SAckOK option in its SYN-ACK?

> **Hint**
>
> In Scapy, TCP options are represented as a list of `(name, value)` tuples. The `SAckOK` option has no data, so its value is an empty byte string.

Capture and analyze the handshake

Start a packet capture before running the script:

```
sudo tshark -i any -p -w tcp-handshake.pcap \
    -f "tcp port 8080 and host 51.68.231.111"
```

Run the script in a second terminal, then stop `tshark` with `Ctrl-C` and open the trace:

```
wireshark tcp-handshake.pcap
```

Useful Wireshark display filters

Filter	Description
`tcp`	All TCP segments
`tcp.flags.syn == 1 and tcp.flags.ack == 0`	SYN only (connection request)

`tcp.flags.syn == 1 and tcp.flags.ack == 1`	SYN-ACK (connection reply)
`tcp.flags.fin == 1`	FIN (connection teardown)
`tcp.flags.push == 1`	Segments with the PSH flag set
`tcp.options.sack_perm == 1`	Segments carrying the SACK-Permitted option
`tcp.analysis.out_of_order`	Out-of-order segments
`tcp.analysis.duplicate_ack`	Duplicate ACKs

In the trace, locate the three segments of the handshake and answer:

- What source port did the script use?
- How many bytes of payload does the SYN carry?
- Can you see the SAckOK option in the SYN and SYN-ACK? Under which field does Wireshark display it?
- Use `Statistics` → `Flow Graph` to visualise the handshake as a time-sequence diagram.

Sending Data to a Discard Server

The discard service (RFC 863) accepts TCP connections, receives all data sent by the client, silently discards it, and acknowledges every segment.

Complete the data transfer script

Open `tcp/scapy-tcp-data-student.py` and fill in the three `FIXME` markers:

- **FIXME 1** — the TCP flags for the data segment; two flags should be set: one to carry a valid acknowledgment number, one to push the data to the server application immediately
- **FIXME 2** — the payload; in Scapy a payload is appended as a `Raw` layer: `ip / TCP(...) / Raw(load=<bytes>)`
- **FIXME 3** — the TCP flags for the FIN segment; two flags should be set

Run the script:

```
sudo python3 tcp/scapy-tcp-data-student.py \
    --host 51.68.231.111 --message "Hello, discard!"
```

Try with the IPv6 address:

```
sudo python3 tcp/scapy-tcp-data-student.py \
    --host 2001:41d0:305:2100::4857 --message "Hello, discard!"
```

Examine the output and answer:

- What sequence number does the data segment carry?
- What ACK number does the server send back after receiving the data? How does it relate to the sequence number and the length of your message?
- What sequence number does the FIN carry? Why does the FIN consume a sequence number even though it carries no data?
- How many segments does the full exchange use? List them with their flags.

Capture and analyze the data transfer

```
sudo tshark -i any -p -w tcp-data.pcap \
    -f "tcp port 9 and host 51.68.231.111"
```

Run the script, stop `tshark`, and open the trace in Wireshark. Use the filters from the previous section to identify each segment.

- Can you see the PSH flag on the data segment?
- What is the `Len` field (payload length) in Wireshark for the data segment?
- Verify that the server's ACK number equals your sequence number plus the number of bytes in your message.

Out-of-Order Segments

TCP guarantees that data is delivered to the application **in order**, regardless of the order in which segments arrive. When a receiver gets a segment whose sequence number is higher than expected, it buffers it and sends a **duplicate ACK** for the last in-order byte it has received. Once the missing segment arrives, the receiver can deliver all the buffered data and send a **cumulative ACK**.

Complete the out-of-order script

Open `tcp/scapy-tcp-ooo-student.py`. The script splits the message into two halves and sends the second half before the first.

Fill in the two `FIXME` markers:

- **FIXME 1** — the sequence number for the second half; it must reflect where this half sits in the overall byte stream
- **FIXME 2** — the sequence number for the first half

Run the script with a message that is easy to split:

```
sudo python3 tcp/scapy-tcp-ooo-student.py \
    --host 51.68.231.111 --message "Hello, out-of-order!"
```

Examine the output and answer:

- What ACK number does the server send after receiving the second half? Is it a duplicate ACK or a new ACK?
- What ACK number does the server send after receiving the first half?
- How do the two ACK numbers differ? What does this tell you about how the server handles out-of-order delivery?

Capture and analyze

```
sudo tshark -i any -p -w tcp-ooo.pcap \
    -f "tcp port 9 and host 51.68.231.111"
```

Open the trace in Wireshark and use the filters `tcp.analysis.out_of_order` and `tcp.analysis.duplicate_ack`.

- Does Wireshark flag the second-half segment as out-of-order?
- Can you identify the duplicate ACK in the trace?

- Use `Statistics → Flow Graph` to see the exchange as a time-sequence diagram. How does it differ from the in-order case?

Port 7

Connect to port 7 on the server, send a message, and examine what the server sends back. Read RFC 862 to understand what this service is supposed to do before running the script.

Complete the port-7 script

Open `tcp/scapy-tcp-port7-student.py` and fill in the two `FIXME` markers:

- **FIXME 1** — the acknowledgment number when ACKing the server's response; the server sent `len(server_data)` bytes starting at sequence number `server_seq`
- **FIXME 2** — write code to verify that the server's response satisfies the specification in RFC 862; print a message indicating pass or fail

Run the script:

```
sudo python3 tcp/scapy-tcp-port7-student.py \
    --host 51.68.231.111 --message "Hello"
```

Try several messages of different lengths:

```
sudo python3 tcp/scapy-tcp-port7-student.py \
    --host 51.68.231.111 --message "ABCDEFGHIJ"

sudo python3 tcp/scapy-tcp-port7-student.py \
    --host 2001:41d0:305:2100::4857 --message "Hello IPv6"
```

Examine the output and answer:

- What does the server send back? Does it match what RFC 862 specifies?
- Does the server's response have the same length as your message?
- What happens if you send an empty message (`--message ""`)?
- What ACK number did you compute for FIXME 1? Show the calculation.

Capture and analyze

```
sudo tshark -i any -p -w tcp-port7.pcap \
    -f "tcp port 7 and host 51.68.231.111"
```

In Wireshark, apply the filter `tcp.port == 7` and examine the full exchange.

- How many segments carry payload? Which direction does each one go?
- Compare the sequence and acknowledgment numbers on the server's data segment with those on your data segment.
- Use `Follow → TCP Stream` (right-click any segment) to see the complete byte stream in both directions.

Port 13

Connect to port 13 on the server. Unlike the previous exercises, you do not need to send anything: the server responds immediately after the connection is established and then closes it. Read RFC 867 before running the script.

Complete the port-13 script

Open `tcp/scapy-tcp-rfc867-student.py` and fill in the two `FIXME` markers:

- **FIXME 1** — the acknowledgment number for the server's data and FIN; remember that a FIN consumes one sequence number, just like a data byte
- **FIXME 2** — decode `server_data` as an ASCII string and display it; then check whether the received string is consistent with RFC 867

Run the script:

```
sudo python3 tcp/scapy-tcp-rfc867-student.py \
    --host 51.68.231.111

sudo python3 tcp/scapy-tcp-rfc867-student.py \
    --host 2001:41d0:305:2100::4857
```

Examine the output and answer:

- How many bytes did the server send?
- What does the string contain? Is its format specified by RFC 867?
- Does the server initiate the close, or does the client?
- Compare the received time to the current UTC time printed by the script. Are they consistent?
- Run the script twice in quick succession. Does the server's response change between the two runs?

Capture and analyze

```
sudo tshark -i any -p -w tcp-port13.pcap \
    -f "tcp port 13 and host 51.68.231.111"
```

In Wireshark, examine the full exchange.

- How many segments does the server send before the FIN?
- Which side sends the first FIN?
- Compare the total number of segments in this exchange with the exchange on port 9. Why is the port-13 exchange longer?

TCP Under Packet Loss: HTTP/1.1 vs HTTP/2

TCP is designed to be reliable: lost segments are retransmitted. But retransmissions take time, and every loss event typically reduces the congestion window — TCP sends less data until it recovers. When downloading many small objects, these effects compound differently depending on whether each object uses its own TCP connection (HTTP/1.1) or shares one connection with other streams (HTTP/2).

The test server exposes the same files on three ports that emulate different levels of packet loss:

Port	Loss rate	Purpose
8080	0 %	No loss (baseline)
8081	10 %	Moderate loss
8082	20 %	Noticeable loss
8083	30 %	Significant loss

The server hosts files of various sizes at `/files/`:

Path	Size
`/files/1KB`	1 KB
`/files/5KB`	5 KB
`/files/10KB`	10 KB
`/files/100KB`	100 KB
`/files/1000KB`	1 MB

Baseline: no loss

First establish a baseline without any packet loss. Download four files with `curl` (HTTP/1.1) and with `nghttp` (HTTP/2) using port 8080:

```
curl -o /dev/null -w "%{time_total}s\n" \
    http://test.computer-networking.info:8080/files/1KB \
    http://test.computer-networking.info:8080/files/5KB \
    http://test.computer-networking.info:8080/files/100KB \
    http://test.computer-networking.info:8080/files/1000KB
```

```
nghttp -n \
    http://test.computer-networking.info:8080/files/1KB \
    http://test.computer-networking.info:8080/files/5KB \
    http://test.computer-networking.info:8080/files/100KB \
    http://test.computer-networking.info:8080/files/1000KB
```

Note the total download times. These will serve as your reference.

HTTP/1.1 under loss

Repeat the `curl` downloads on each lossy port and record the total time:

```
curl -o /dev/null -w "%{time_total}s\n" \
    http://test.computer-networking.info:8081/files/1KB \
    http://test.computer-networking.info:8081/files/5KB \
    http://test.computer-networking.info:8081/files/100KB \
    http://test.computer-networking.info:8081/files/1000KB
```

Repeat for ports 8082 and 8083.

While running, capture the traffic:

```
sudo tshark -i any -p -w http11-loss10.pcap \
    -f "tcp port 8081"
```

Stop `tshark` after `curl` finishes, then open the trace:

```
wireshark http11-loss10.pcap
```

Useful Wireshark display filters

Filter	Description
`tcp.analysis.retransmission`	Retransmitted segments

`tcp.analysis.duplicate_ack`	Duplicate ACKs (signal of a lost segment)
`tcp.analysis.lost_segment`	Gaps detected by Wireshark
`tcp.window_size_value`	Advertised receive window
`tcp.flags.fin == 1`	FIN segments (connection teardown)

In the trace, answer:

- Can you see retransmissions? Which files triggered them?
- How does the number of retransmissions change between 10 %, 20 %, and 30 % loss?
- When a segment is lost, what happens to the TCP window size in the segments that follow?
- How many separate TCP connections does `curl` open to download the four files?
- Compare the total download time at each loss rate with the baseline. How does loss affect HTTP/1.1 transfers?

HTTP/2 under loss

Repeat with `nghttp` on the same ports:

```
nghttp -n \
    http://test.computer-networking.info:8081/files/1KB \
    http://test.computer-networking.info:8081/files/5KB \
    http://test.computer-networking.info:8081/files/100KB \
    http://test.computer-networking.info:8081/files/1000KB
```

Repeat for ports 8082 and 8083.

Capture the traffic:

```
sudo tshark -i any -p -w http2-loss10.pcap \
    -f "tcp port 8081"
```

Open the trace and add the filter `http2` in addition to the TCP filters from the previous section.

Additional Wireshark filters for HTTP/2

Filter	Description
`http2`	All HTTP/2 frames
`http2.streamid`	HTTP/2 stream identifier field
`http2.type == 0`	DATA frames only
`http2.type == 1`	HEADERS frames only
`http2.flags.end_stream == 1`	Last frame of a stream

In the trace, answer:

- How many TCP connections does `nghttp` open?
- Can you see HTTP/2 stream IDs in the frames? How many streams carry data simultaneously?
- When a segment is retransmitted, which streams are affected?
- Does a loss event on the single TCP connection stall all streams or just one?
- Compare the total download time of `nghttp` with `curl` at each loss rate.

> Hint
>
> Use `Statistics → Flow Graph` in Wireshark to visualise the exchange as a time-sequence diagram. Select *TCP flow* to see sequence numbers, or *HTTP/2 stream* if your Wireshark version supports it.

Comparing small and large objects

Focus on the smallest file (1 KB) and the largest (1000 KB) separately. Run the following at 30 % loss and capture each experiment:

```
curl -o /dev/null -w "%{time_total}s\n" \
    http://test.computer-networking.info:8083/files/1KB
```

```
curl -o /dev/null -w "%{time_total}s\n" \
    http://test.computer-networking.info:8083/files/1000KB
```

```
nghttp -n \
    http://test.computer-networking.info:8083/files/1KB \
    http://test.computer-networking.info:8083/files/5KB \
    http://test.computer-networking.info:8083/files/100KB \
    http://test.computer-networking.info:8083/files/1000KB
```

Answer:

- At 30 % loss, how long does it take `curl` to fetch the 1 KB file? How many TCP segments does the exchange require?
- How does that compare with fetching 1000 KB over the same lossy connection?
- When `nghttp` fetches all four files together over one connection, does the 1 KB file finish before or after the 1000 KB file?
- What does this tell you about the benefit of HTTP/2 multiplexing for small objects when there are packet losses?