Analysing HTTP/1.1, HTTP/2 and HTTP/3+QUIC Packet Traces

In this lab you will analyse packet traces captured on localhost between a test client and a local server that supports HTTP/1.1, HTTP/2 and HTTP/3+QUIC.

Pre-collected traces are provided in the lab files.

Plain HTTP/1.1 (no TLS)

Corresponding trace : http1-plain-localhost.pcapng

Open the trace in Wireshark. You can use the filters in the table below to locate specific packets packets.

Wireshark filters — plain HTTP/1.1

Filter	Description
`tcp.flags.syn == 1`	SYN packets (TCP handshake, options visible here)
`http.request`	The GET request sent by the client
`http.response`	The server response

Questions:

1. Look at the packets exchanged. In practice, how many round trips are necessary to carry out the request ?
2. How many bytes of HTML content does the response carry? How many additional bytes were required by HTTP headers, TCP, and IPv6 for this exchange? Use *Statistics* > *Capture File Properties* for the total, and `tcp.len > 0` to isolate data-carrying segments.

HTTP/1.1 with TLS 1.3

Corresponding trace : quic/http1-tls-localhost.pcapng

We embedded the TLS secrets in the capture to allow wireshark to decrypt the packets.

Wireshark filters — HTTP/1.1 + TLS 1.3

Filter	Description
`tls.handshake.type == 1`	ClientHello
`tls.handshake.type == 2`	ServerHello
`http.request`	Decrypted GET request
`http.response`	Decrypted server response

Questions:

1. How many bytes does the ClientHello occupy (TCP payload)? How many bytes does the ServerHello occupy?
2. The ClientHello contains an SNI extension. What is the role of SNI in TLS? What value is sent here, and why? See RFC 6066 Section 3.
3. How many bytes in total were exchanged to transfer the same HTML file as in the previous section? How does this compare with the plain HTTP/1.1 exchange?

HTTP/2 with TLS 1.3

Corresponding trace : quic/http2-tls-localhost.pcapng

As with the previous trace, TLS session keys are embedded; Wireshark decrypts all records automatically.

Wireshark filters — HTTP/2 + TLS 1.3

Filter	Description
`http2`	All HTTP/2 frames
`http2.type == 4`	SETTINGS frames
`http2.type == 1`	HEADERS frames (request and response)
`http2.type == 0`	DATA frames (response body)
`http2.streamid == 1`	All frames belonging to stream 1

Questions:

1. How many bytes in total were exchanged to transfer the same HTML file? How does this compare with the HTTP/1.1+TLS exchange?
2. How many HTTP/2 streams are used? What is the purpose of stream 0? What is carried on stream 1? See RFC 9113 Section 5.1.

HTTP/3 over QUIC

Corresponding trace : quic/http3-localhost.pcapng

TLS session keys are embedded; Wireshark decodes QUIC and HTTP/3 frames automatically.

Wireshark filters — HTTP/3 + QUIC

Filter	Description
`quic.long.packet_type == 0`	Initial packets (start of handshake)
`quic.long.packet_type == 2`	Handshake packets
`quic`	All QUIC packets
`http3`	HTTP/3 frames (decoded after decryption)
`quic.ack`	ACK frames

Questions:

1. Analyze the handshake and compare it with a TCP+TLS handshake. Examine the client transport parameters carried in the Initial packet. List each parameter and explain its role. See RFC 9000 Section 18.2.
2. How many bytes in total were exchanged? How does this compare with the HTTP/2+TLS exchange?
3. What connection identifiers do the client and the server use throughout the connection? How do they differ in length, and why does QUIC use connection identifiers instead of the IP-address/port pair? See RFC 9000 Section 5.1.
4. How many QUIC streams are opened in this connection? For each stream, identify its direction and its role in HTTP/3. See RFC 9114 Sections 6.2 and 7.2.

QUIC 0-RTT

Corresponding trace : quic/http3-0rtt-localhost.pcapng

This trace contains **two successive QUIC connections** to the same server. The first connection performs a full handshake; the second reuses the session ticket issued at the end of the first handshake to send the HTTP/3 request before the handshake completes (0-RTT early data).

Wireshark filters — QUIC 0-RTT

Filter	Description
`quic.long.packet_type == 0`	Initial packets (identify start of each connection)
`quic.long.packet_type == 1`	0-RTT packets (early data sent before handshake completes)
`tls.handshake.type == 4`	NewSessionTicket (issued at end of first connection)
`tls.handshake.extensions_pre_shared_key`	PSK extension in the second ClientHello

Questions:

1. How many bytes in total are exchanged in the trace? Does 0-RTT reduce the total number of bytes exchanged compared with a regular QUIC connection? Explain why.
2. Compare the ClientHello carried in the Initial packet of the first connection with the ClientHello of the second connection. Which TLS extensions are present in the second ClientHello but absent from the first? What do they carry? See RFC 8446 Section 4.2.11 and RFC 9001 Section 4.6.1.
3. In the second connection, locate the 0-RTT packet (`quic.long.packet_type == 1`). What application data does it carry? At what point in the handshake is it sent relative to the ClientHello?
4. 0-RTT data can be replayed by an attacker who captures the packet. What does RFC 9001 Section 8.1 say about how servers should handle this risk?

Going Further — Capturing Your Own Traces

The traces in this lab were collected on a loopback interface. Real-world deployments differ: server certificates, negotiated cipher suites, connection IDs, and transport parameters all vary. Capturing traces against production servers (`www.google.com`, `www.cloudflare.com`) exposes these differences and lets you see how major CDNs tune their QUIC stacks.

To decrypt TLS and QUIC traffic you need the session keys. The `SSLKEYLOGFILE` environment variable, supported by curl, Firefox, and Chromium, instructs the TLS library to write each session's key material to a file. The capture tool records the encrypted packets; `editcap --inject-secrets` combines both into a single self-contained `.pcapng` that Wireshark decrypts automatically.

Capturing with a browser

Before launching the browser, set `SSLKEYLOGFILE` in its environment:

- On **Linux**:

```
export SSLKEYLOGFILE=~/tls-keys.txt
google-chrome &     # or firefox
```

- On **macOS**:

```
export SSLKEYLOGFILE=~/tls-keys.txt
open -a "Google Chrome"   # or: open -a Firefox
```

Chromium-based browsers honour the variable unconditionally. Firefox requires it to be set *before* the first window opens.

Capture scripts

Two helper scripts are provided, one per OS. Both capture packets while you browse, then embed the key file into the pcapng output.

scripts/capture-browser-linux.sh

scripts/capture-browser-mac.sh

Usage:

```
# Linux
sudo bash scripts/capture-browser-linux.sh www.google.com

# macOS
sudo bash scripts/capture-browser-mac.sh www.google.com
```

Each script:

1. Resolves the target hostname and starts a `tcpdump` capture filtered to that host (requires tcpdump ≥ 4.99 for reliable pcapng output).
2. Prints the `SSLKEYLOGFILE` path to set in the browser environment.
3. Waits for you to press Enter, then stops the capture.
4. Runs `editcap --inject-secrets tls,<keyfile> <pcap> <pcapng>` to embed the keys and produce the final `.pcapng` file.

Open the resulting file in Wireshark; TLS and QUIC records will be decrypted automatically.