Project 03 – Wireshark TCP Analysis: File Upload, Segmentation & RTT Measurement

Author
Oluwaseun Joseph

Project Context
This report is part of my cybersecurity GitHub portfolio and demonstrates hands-on packet-level analysis of TCP using Wireshark. The investigation documents real network behavior using captured evidence and aligns with skills expected in SOC, Blue Team, and Network Security roles.

1. Project Overview

In this project, I analyzed how the Transmission Control Protocol (TCP) ensures reliable data delivery by capturing and inspecting a real file upload using Wireshark. I uploaded a text file (*Alice's Adventures in Wonderland*, approximately 150 KB) from my local machine to the remote server gaia.cs.umass.edu and captured all packets exchanged during the transfer.
By examining the captured trace, I observed how TCP establishes connections, segments application data, manages flow control, and delivers data reliably without retransmissions. I also measured round-trip time (RTT) using real packet timestamps to evaluate performance.

2. Environment & Tools

- **Operating System:** Windows 10
- **Packet Analysis Tool:** Wireshark
- **Protocols Analyzed:** TCP, HTTP
- **Remote Server:** gaia.cs.umass.edu
- **File Uploaded:** alice.txt (~150 KB)
- **Network Type:** Wi-Fi

3. Evidence Collection & Methodology

3.1 Downloading the File

I launched a web browser and accessed the following URL to retrieve the ASCII version of *Alice's Adventures in Wonderland*:
http://gaia.cs.umass.edu/wireshark-labs/alice.txt
I saved the file locally on my system for use during the upload process.

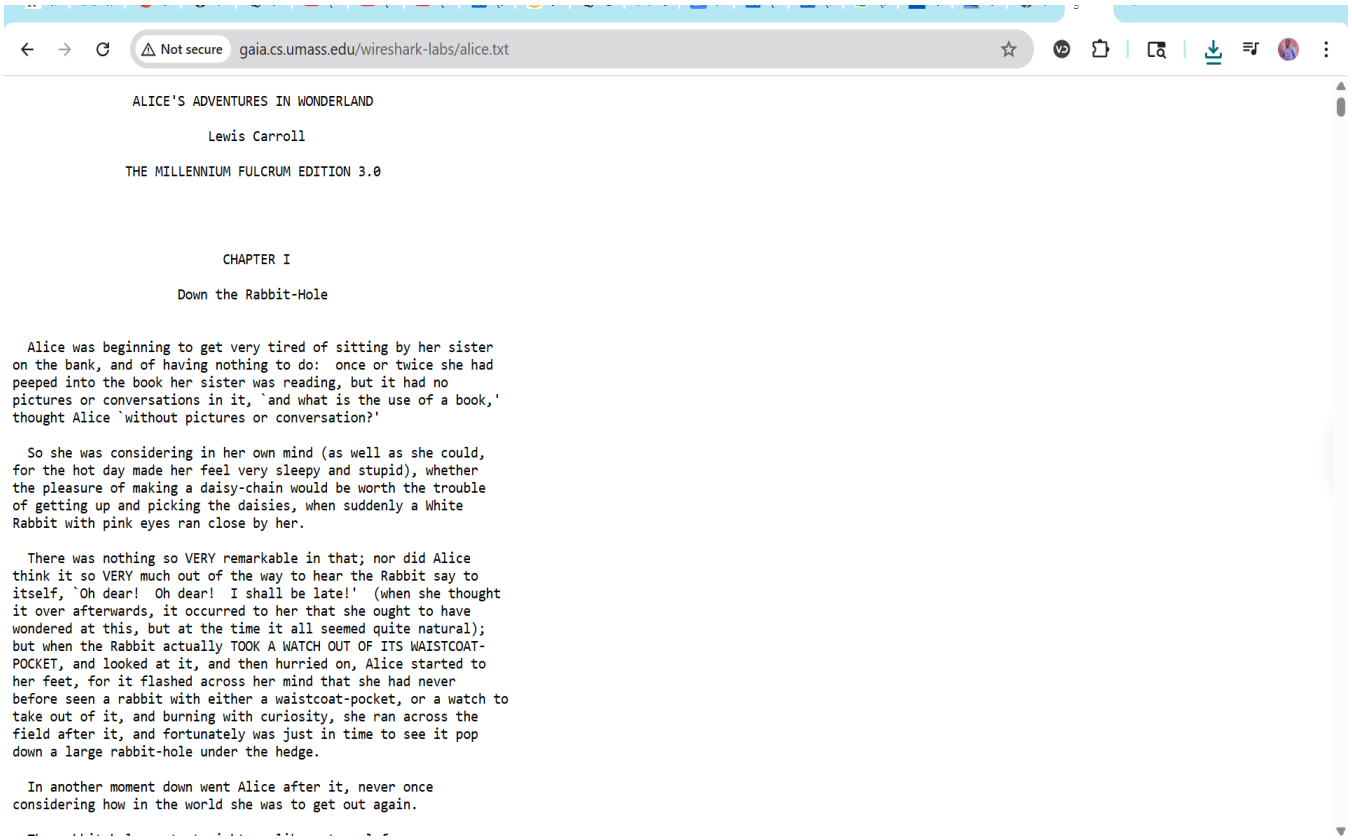


Figure 1 – Browser accessing alice.txt

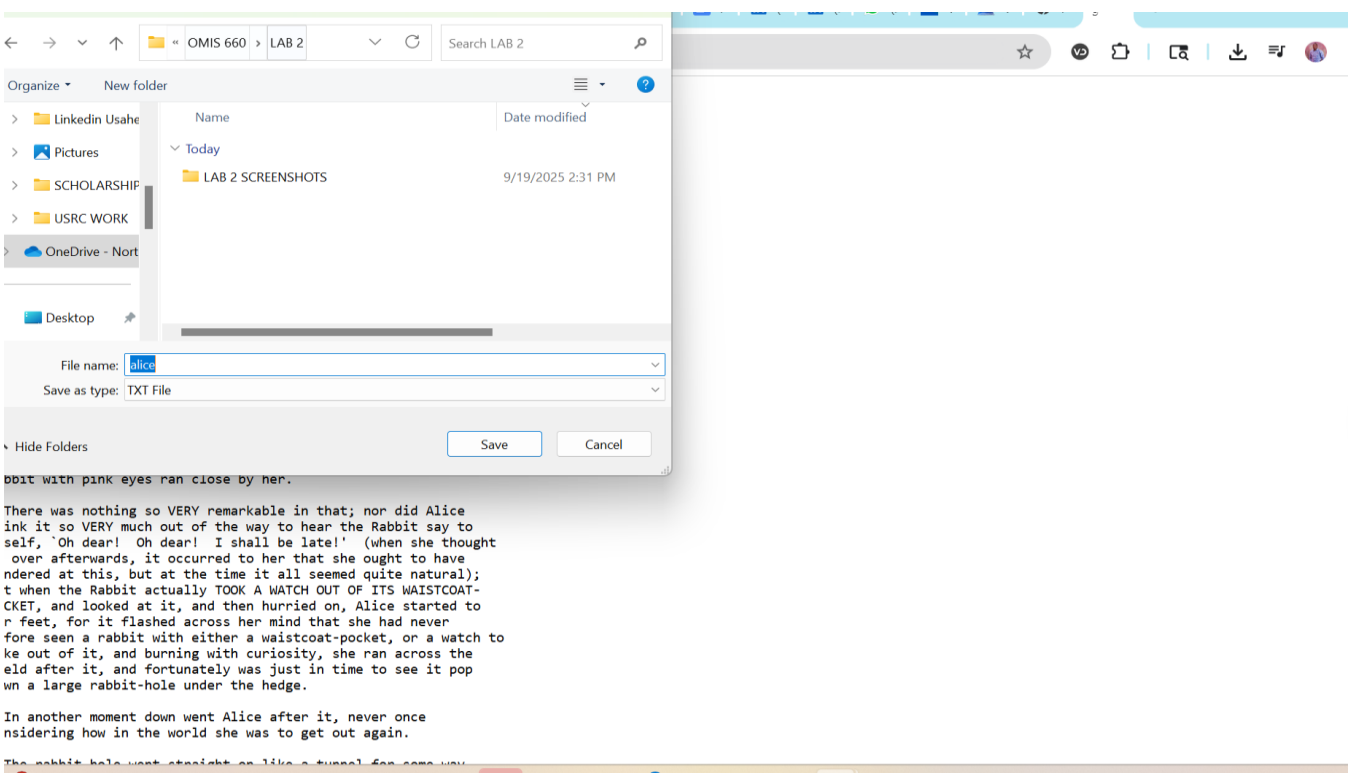


Figure 2 – Saving alice.txt locally

3.2 Navigating to the TCP Upload Page

Next, I navigated to the TCP upload page provided by the Wireshark lab:
http://gaia.cs.umass.edu/wireshark-labs/TCP-wireshark-file1.html
I verified that the upload interface was correctly loaded and ready for use before proceeding.

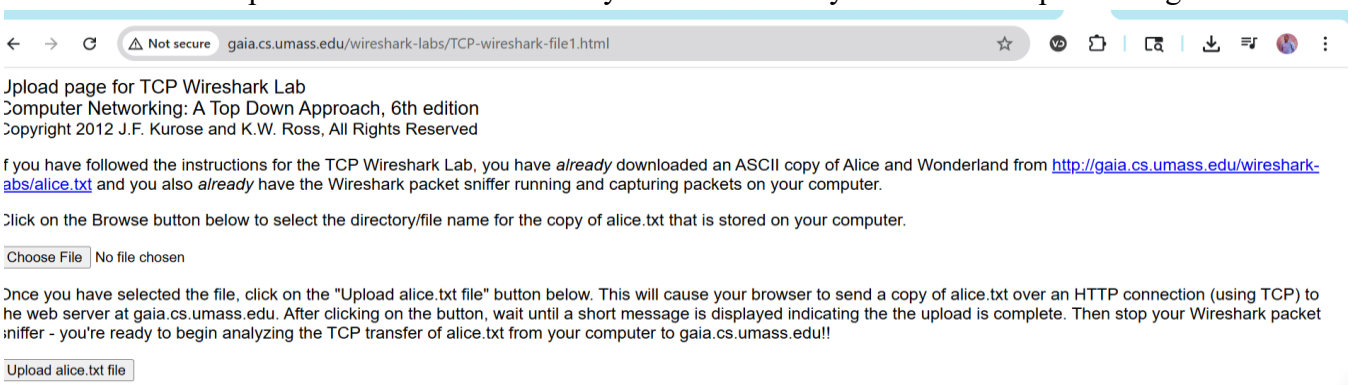


Figure 3 – TCP upload page on gaia.cs.umass.edu

3.3 Selecting the File for Upload

Using the **Browse** button on the upload page, I selected the previously saved alice.txt file. At this stage, I did not yet initiate the upload.

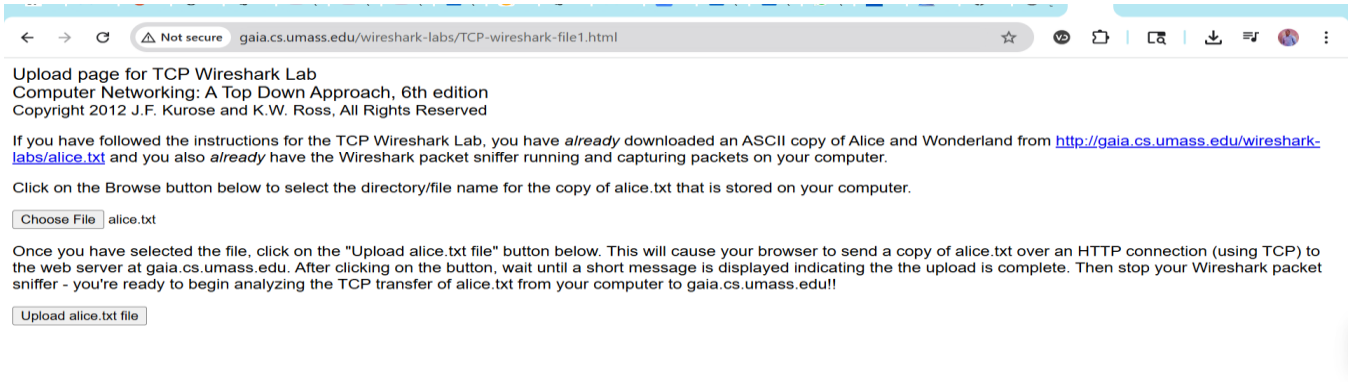


Figure 4 – Selecting the alice.txt file

3.4 Starting Wireshark Capture

Before uploading the file, I launched Wireshark and started packet capture on the active network interface to ensure that all TCP packets related to the transfer were recorded.

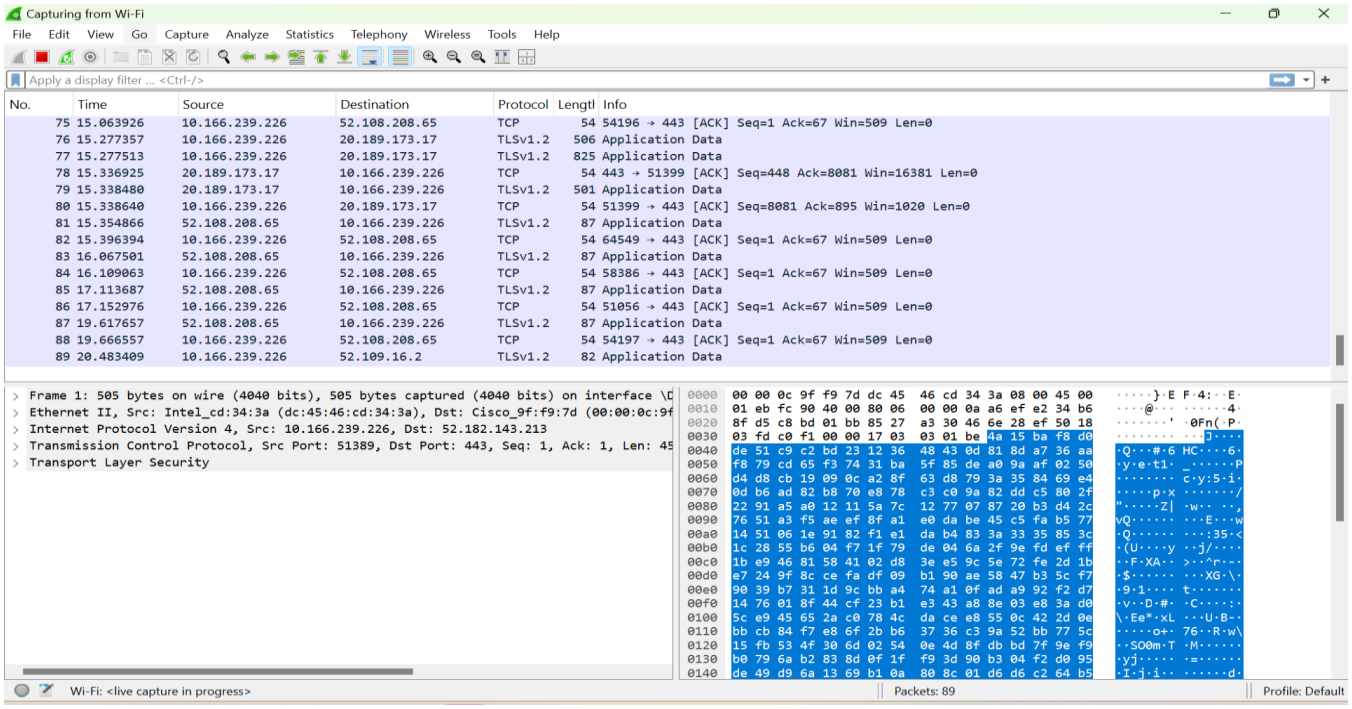
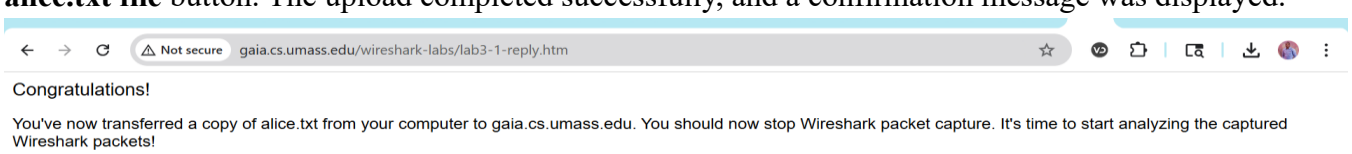


Figure 5 – Wireshark packet capture started

3.5 Uploading the File

After confirming that packet capture was running, I returned to the browser and clicked the **Upload alice.txt file** button. The upload completed successfully, and a confirmation message was displayed.



Congratulations!

You've now transferred a copy of alice.txt from your computer to gaia.cs.umass.edu. You should now stop Wireshark packet capture. It's time to start analyzing the captured Wireshark packets!

Figure 6 – File upload confirmation message

4. Packet Analysis

4.1 Inspecting the HTTP POST Request

After stopping the capture, I examined the packet trace to locate the **HTTP POST** request responsible for uploading the file. I expanded the HTTP details in Wireshark to inspect the application-layer content.

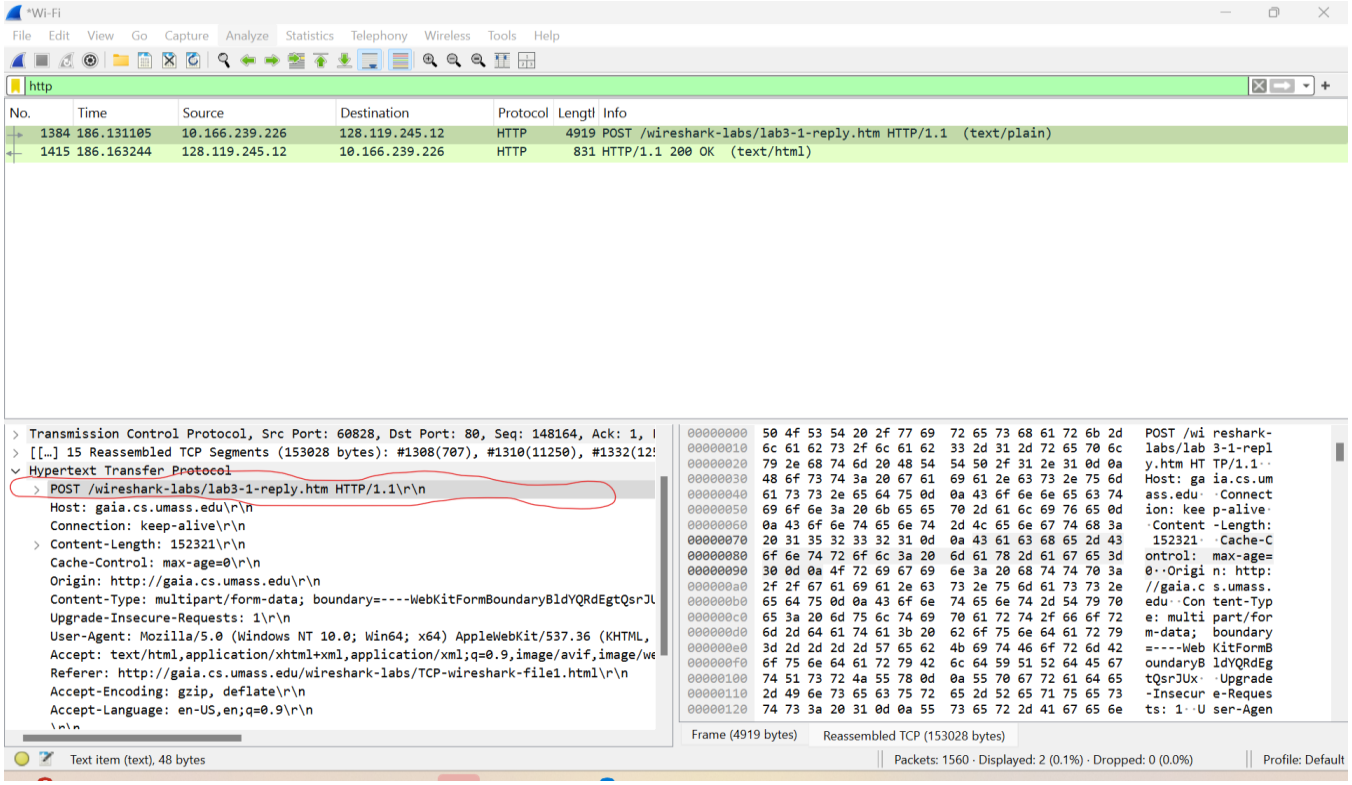


Figure 7 – Expanded HTTP POST request

I observed that the POST payload contained the full contents of alice.txt (over 150 KB). Since this exceeded the size of a single TCP segment, the data was segmented across multiple TCP packets.

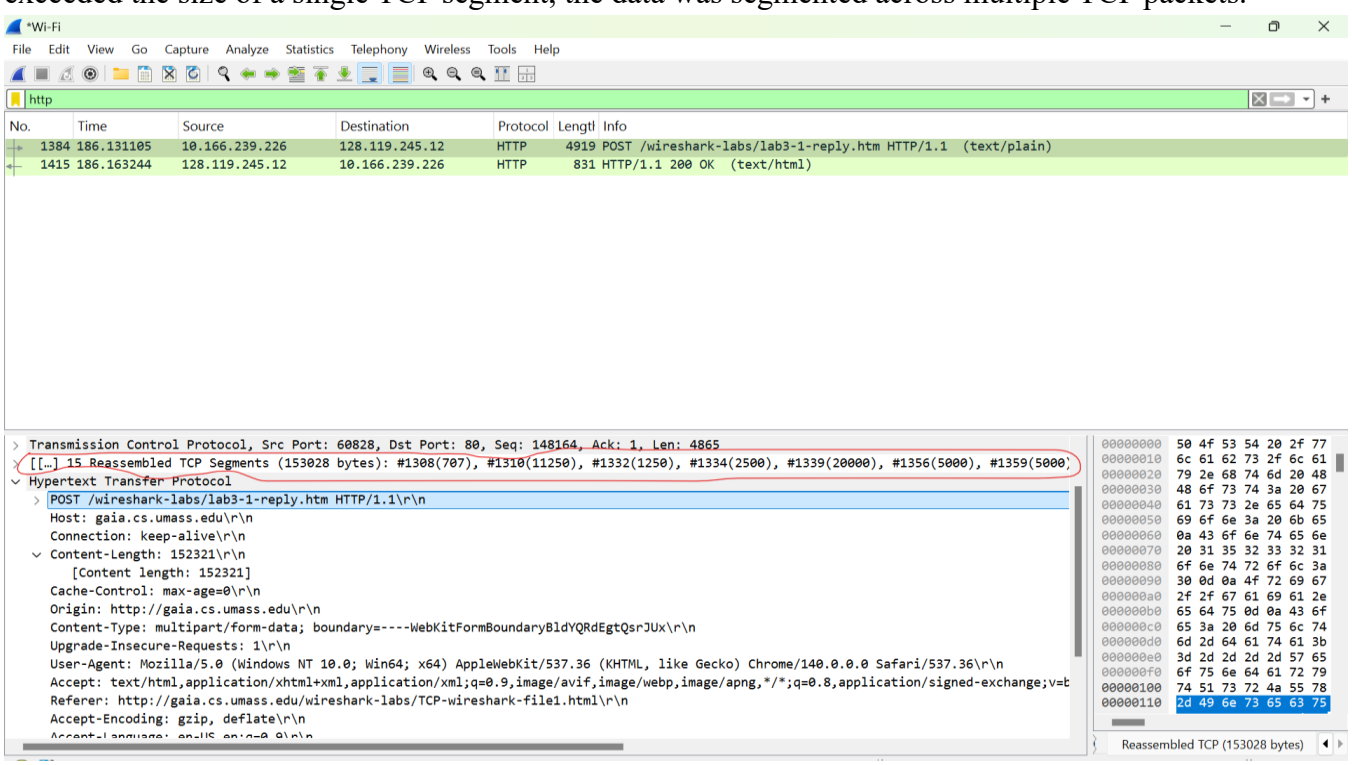


Figure 8 – TCP segment containing the beginning of the POST payload

4.2 Observing the TCP Three-Way Handshake

To focus on TCP traffic, I applied a display filter (tcp) in Wireshark. From the filtered packets, I identified the **SYN**, **SYN/ACK**, and **ACK** segments that established the TCP connection.

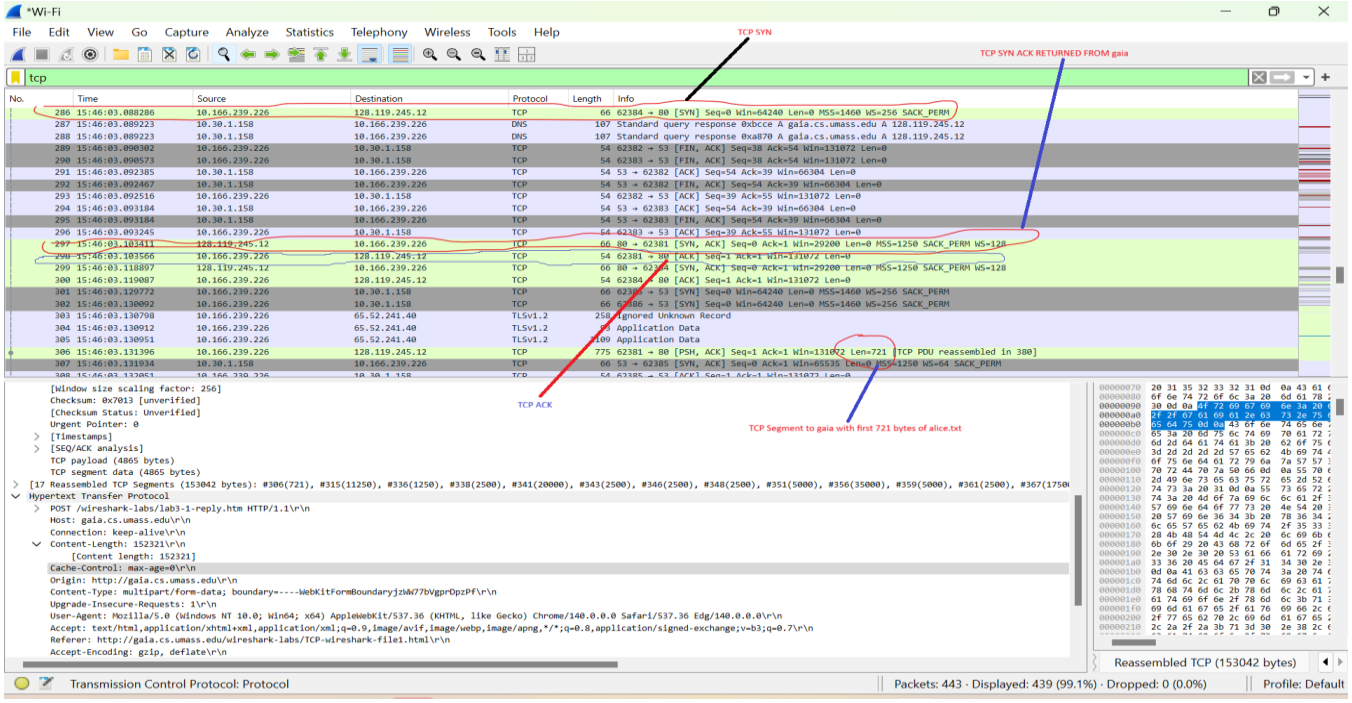


Figure 9 – TCP three-way handshake (SYN, SYN/ACK, ACK)

5. Findings & Evidence

5.1 Client and Server Identification

By inspecting the TCP header fields in the HTTP POST packet, I identified the client's IP address and source port.

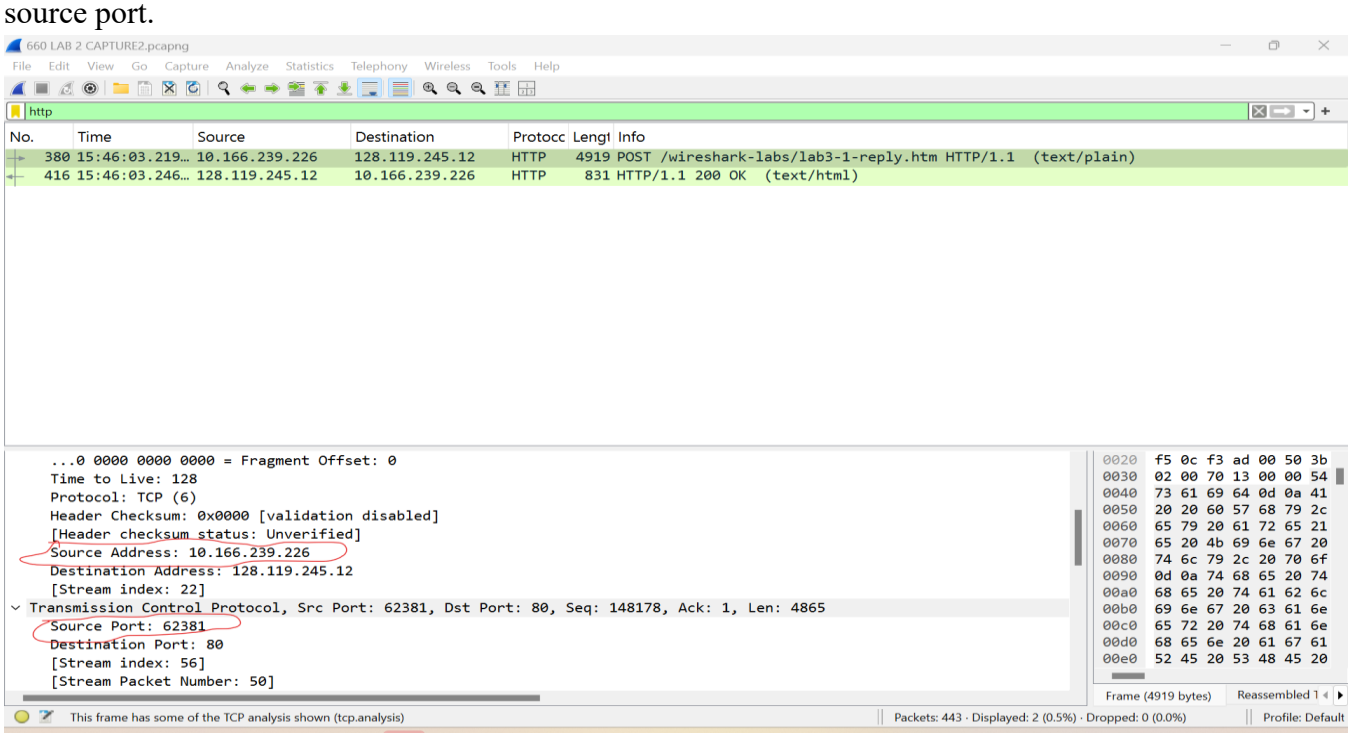


Figure 10 – Client IP address and source TCP port

- **Client IP Address:** 10.166.239.226
- **Source TCP Port:** 62381

I also identified the server's IP address and destination port.

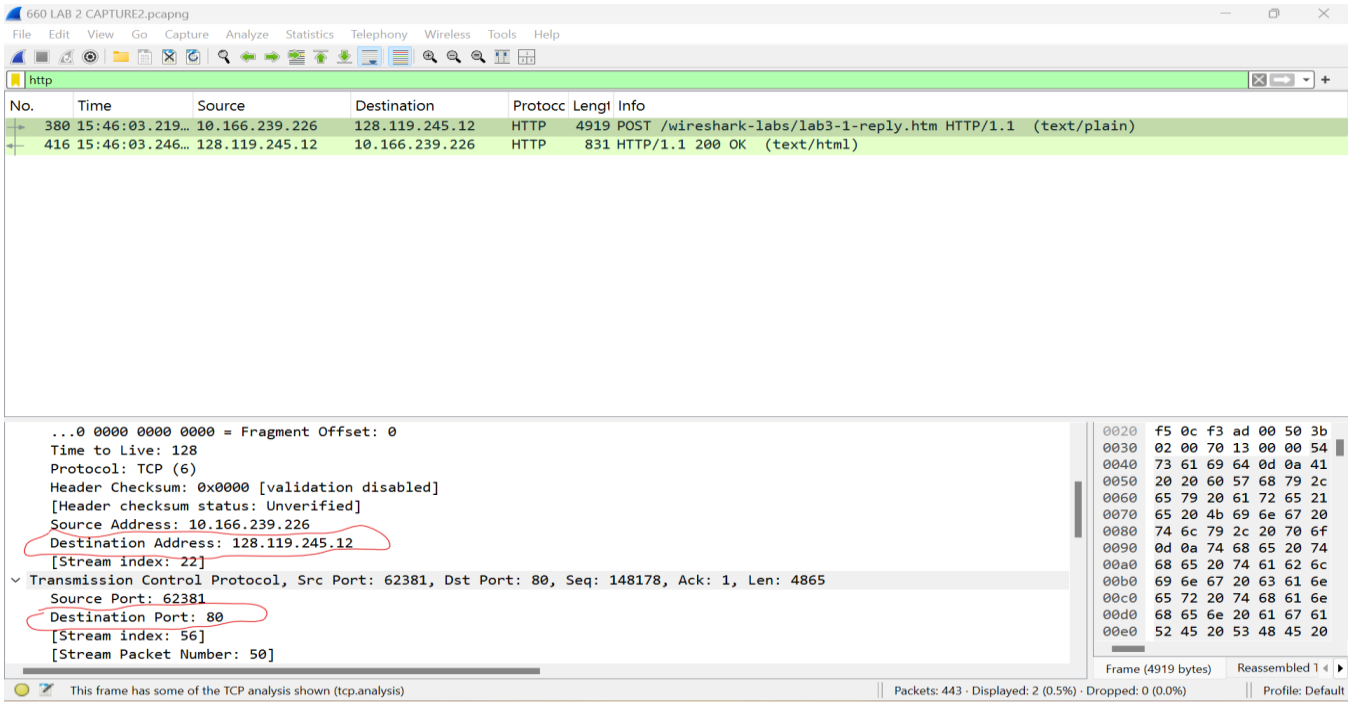


Figure 11 – Server IP address and destination TCP port

- **Server IP Address:** 128.119.245.12
- **Destination TCP Port:** 80

5.2 TCP Control Segments

I located the initial TCP SYN segment used to initiate the connection and examined its sequence number and control flags.

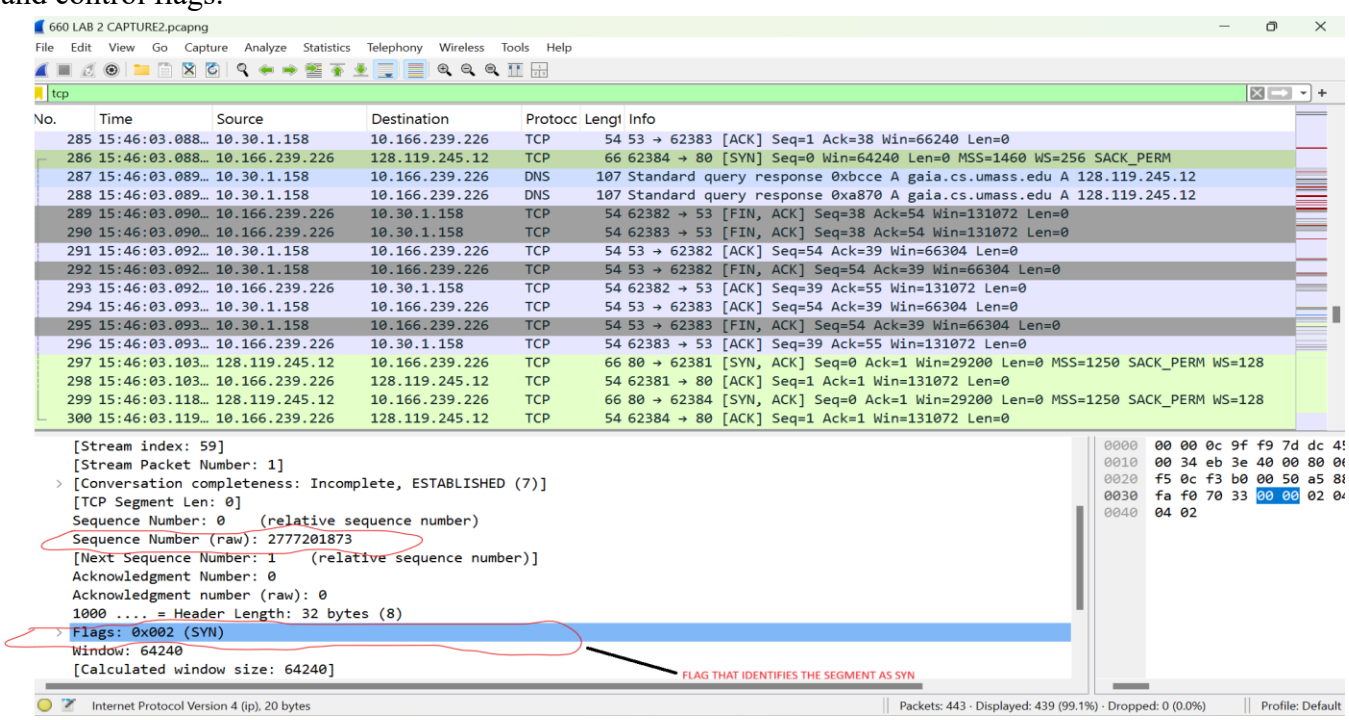


Figure 12 – Initial TCP SYN segment

I then examined the server's response, which contained both the SYN and ACK flags.

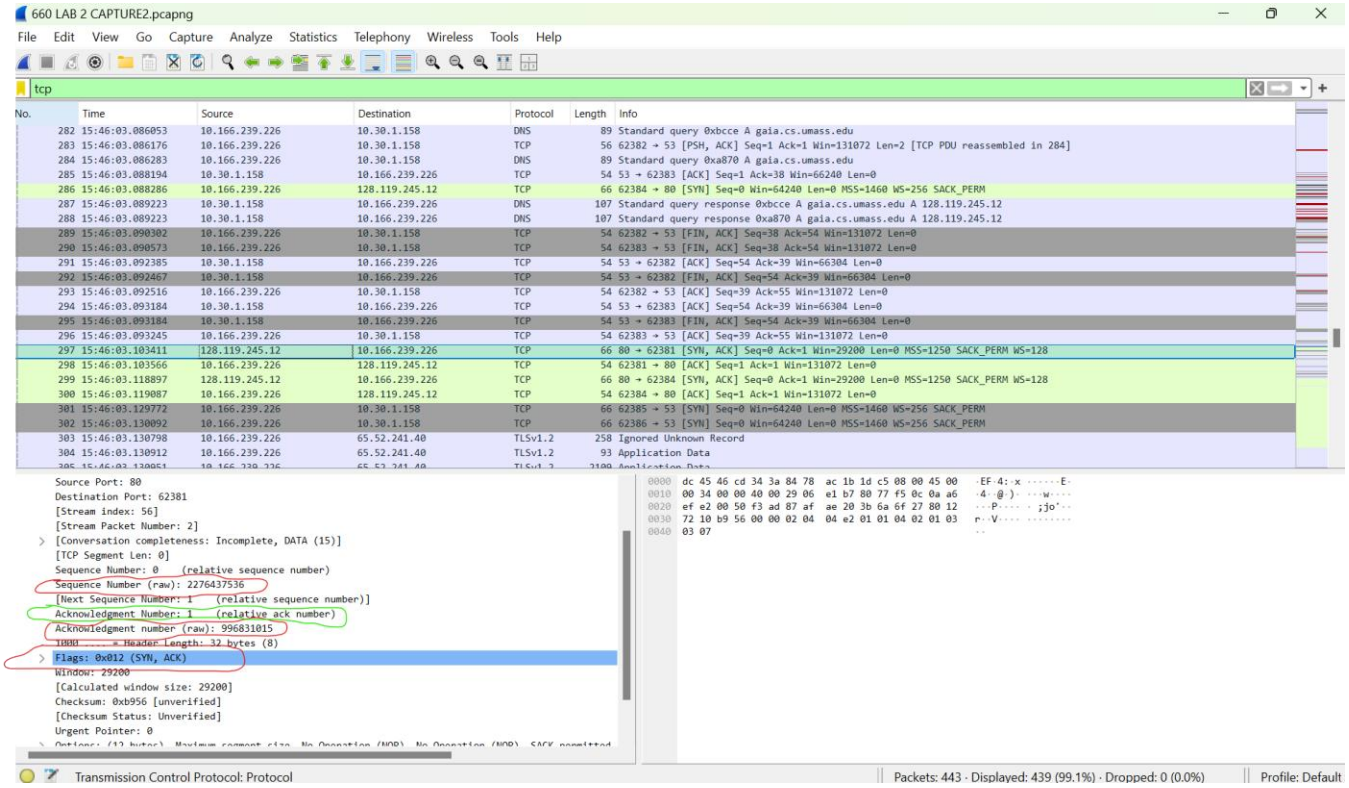


Figure 13 – SYN/ACK segment from the server

5.3 TCP Segment Carrying the HTTP POST Header

I identified the TCP segment that carried the HTTP POST header and the beginning of the file data.

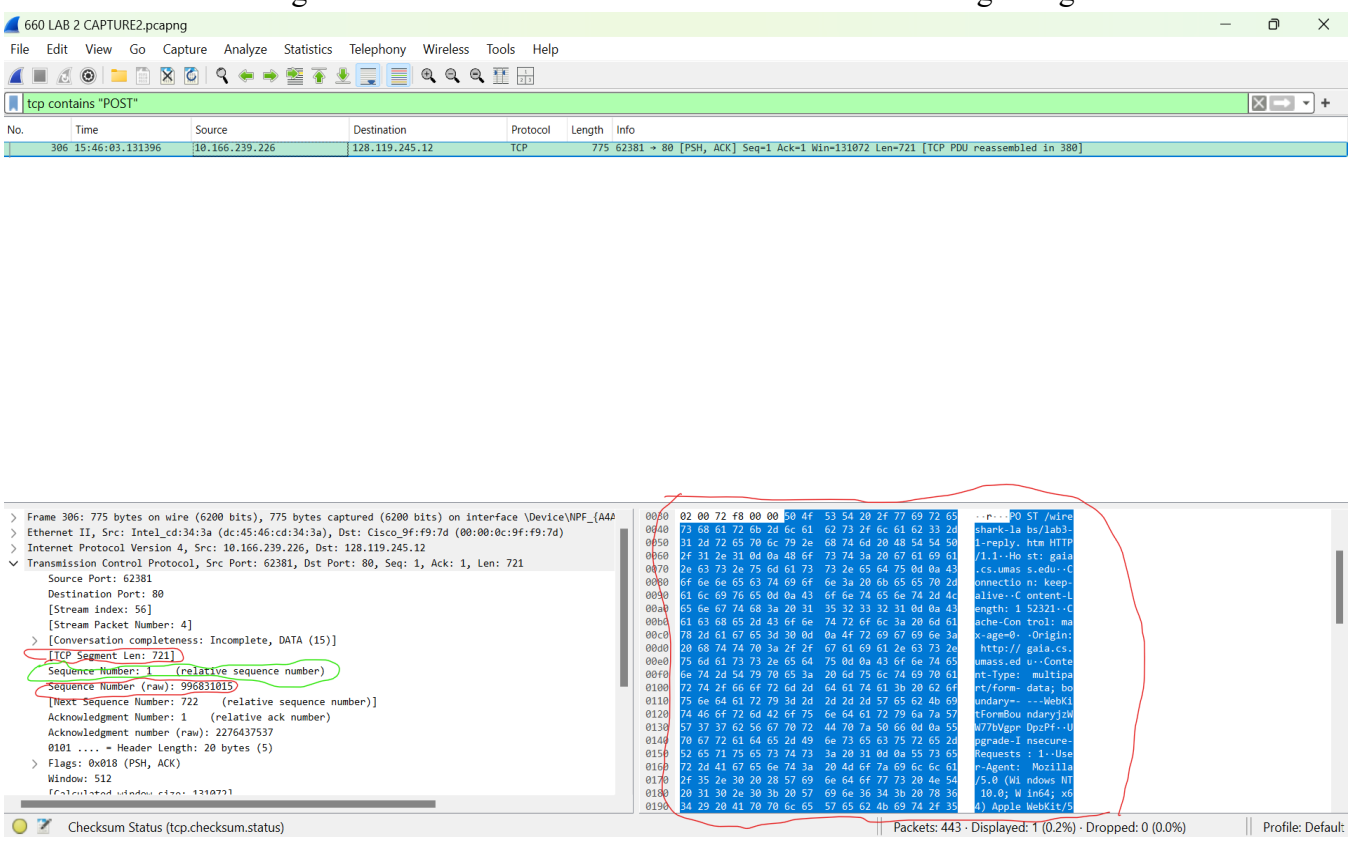


Figure 14 – TCP segment carrying the HTTP POST header

- **Payload Size:** 721 bytes
- **Observation:** The entire file did not fit in one segment and was transmitted across multiple TCP segments.

5.4 Round-Trip Time (RTT) Measurement

To measure RTT, I compared the timestamp of a data-carrying segment with the timestamp of its corresponding ACK.

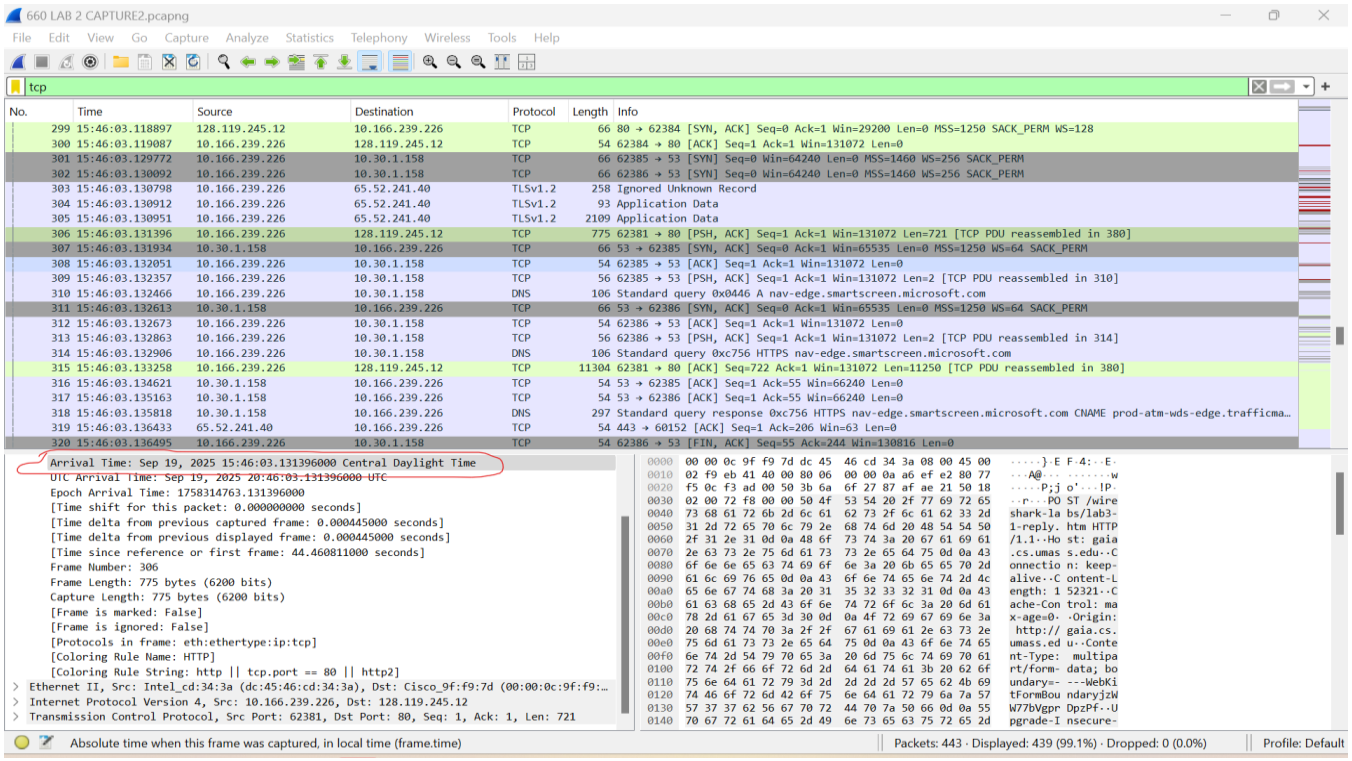


Figure 15 – Time the POST segment was sent

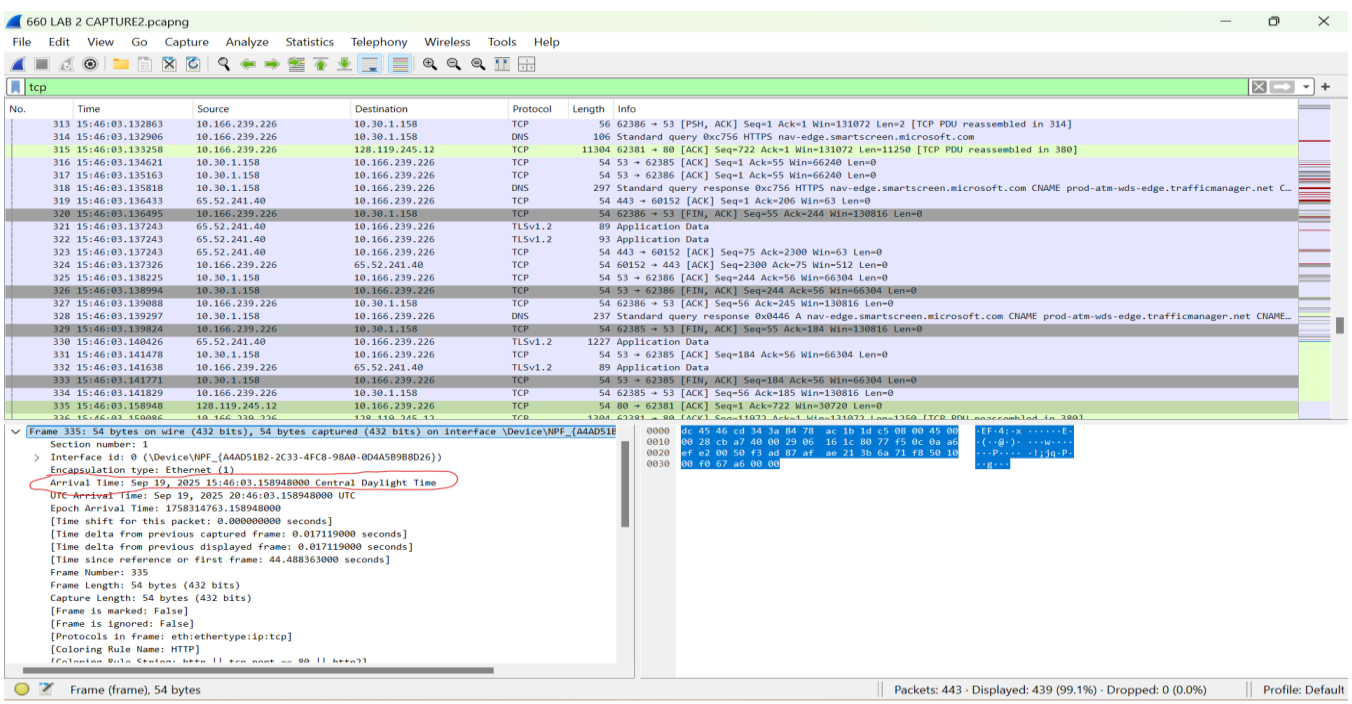


Figure 16 – ACK received for the POST segment

RTT = ACK_time − Data_send_time.

Do the subtraction digit-by-digit:

- Send time = 15:46:03.131396000
- ACK time = 15:46:03.158948000

0.158948000 − 0.131396000 = 0.027552000 seconds = 27.552 ms

RTT ≈ 27.552 ms

- **Measured RTT:** ≈ 27.55 ms

I repeated the process for the next data-carrying TCP segment.

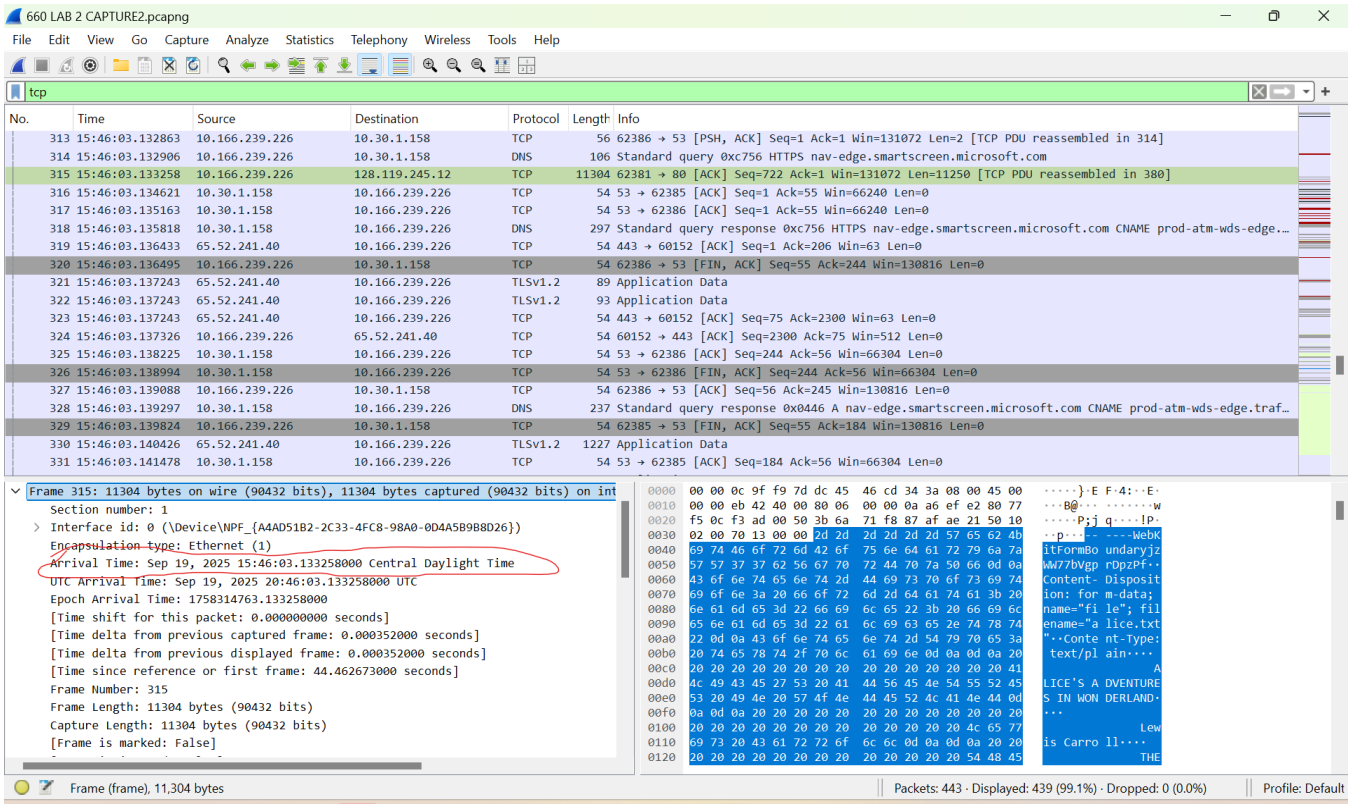


Figure 17 – Second data segment sent

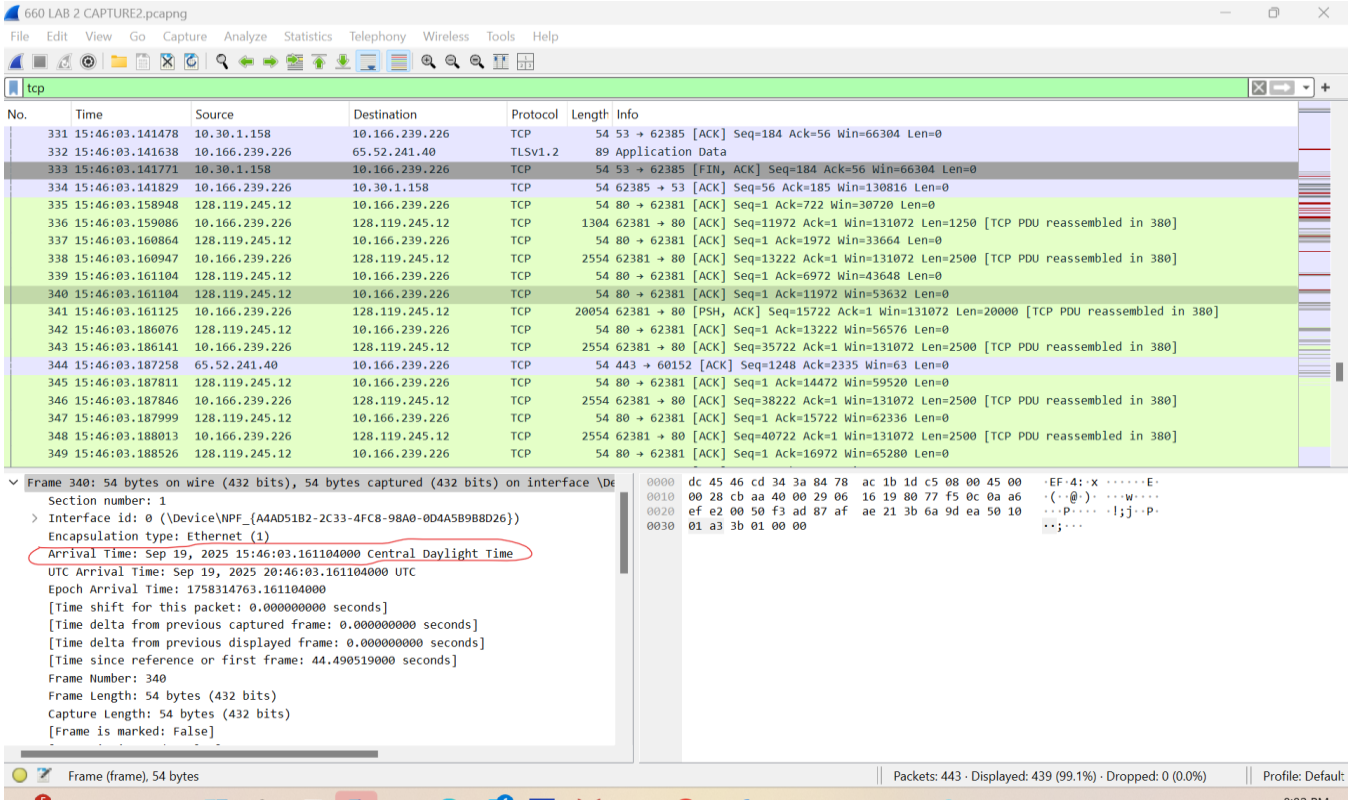


Figure 18 – ACK received for second data segment

Therefore, **RTT** = time (ACK) − time (segment sent).
= 15:46:03.161104 − 15:46:03.133258 = 0.027846 s = 27.846 ms
Answer (second segment): **RTT ≈ 27.85 ms**
Measured RTT: ≈ 27.85 ms

5.5 TCP Segment Length Analysis

I examined the first four data-carrying TCP segments and recorded their total lengths (header + payload). I derived the total size of each data-carrying TCP segment by examining the frame summary fields in Wireshark, specifically the "bytes on wire" and "bytes captured" values. These fields represent the complete packet size transmitted on the network and include both the TCP/IP headers and the TCP payload.

Using this approach, I analyzed the first four data-carrying TCP segments and recorded their total lengths.

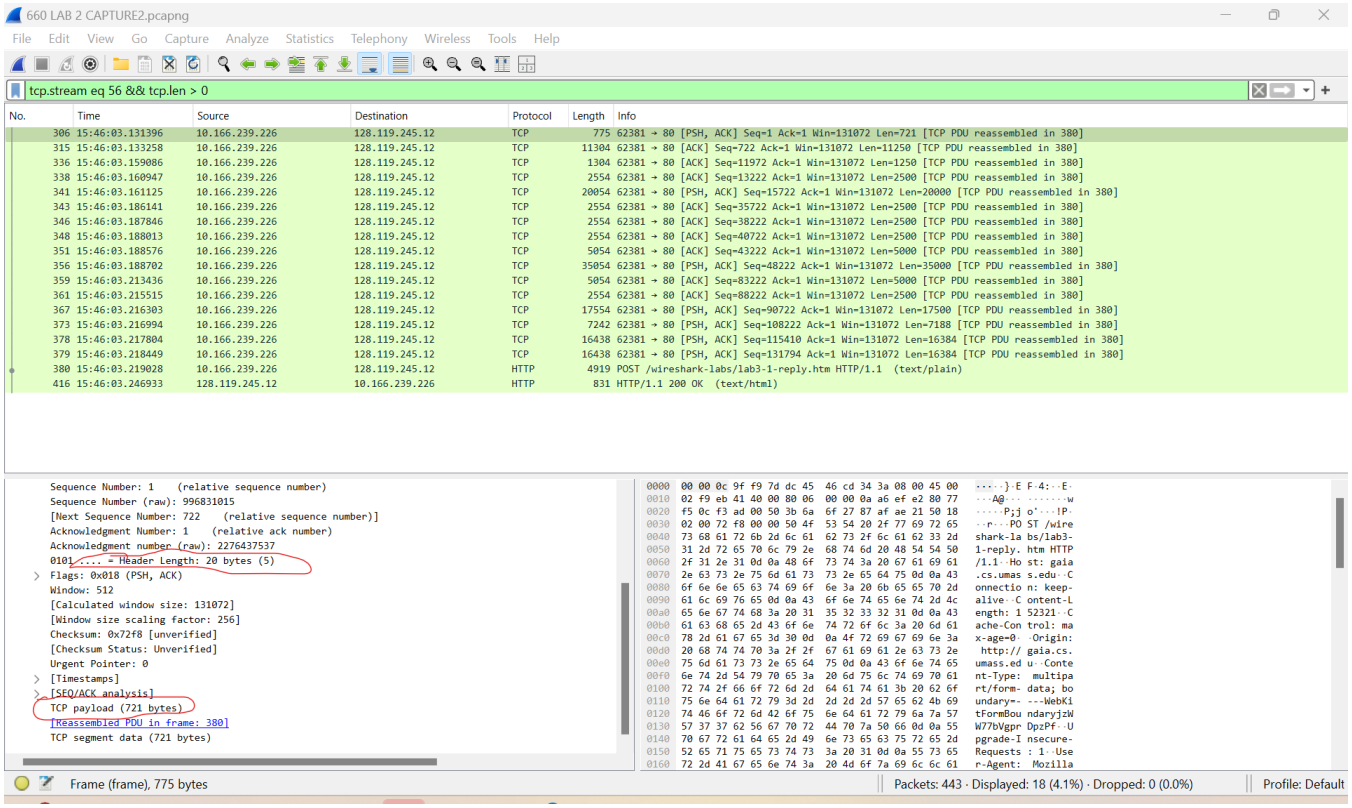


Figure 19a – Segment 1 total length

Segment 1 (Frame 306)

From the attached Figure 19a above it can be deduced that:

- Header Length: 20 bytes
- TCP Payload: 721 bytes
- Total Length (Header + Payload): 741 bytes

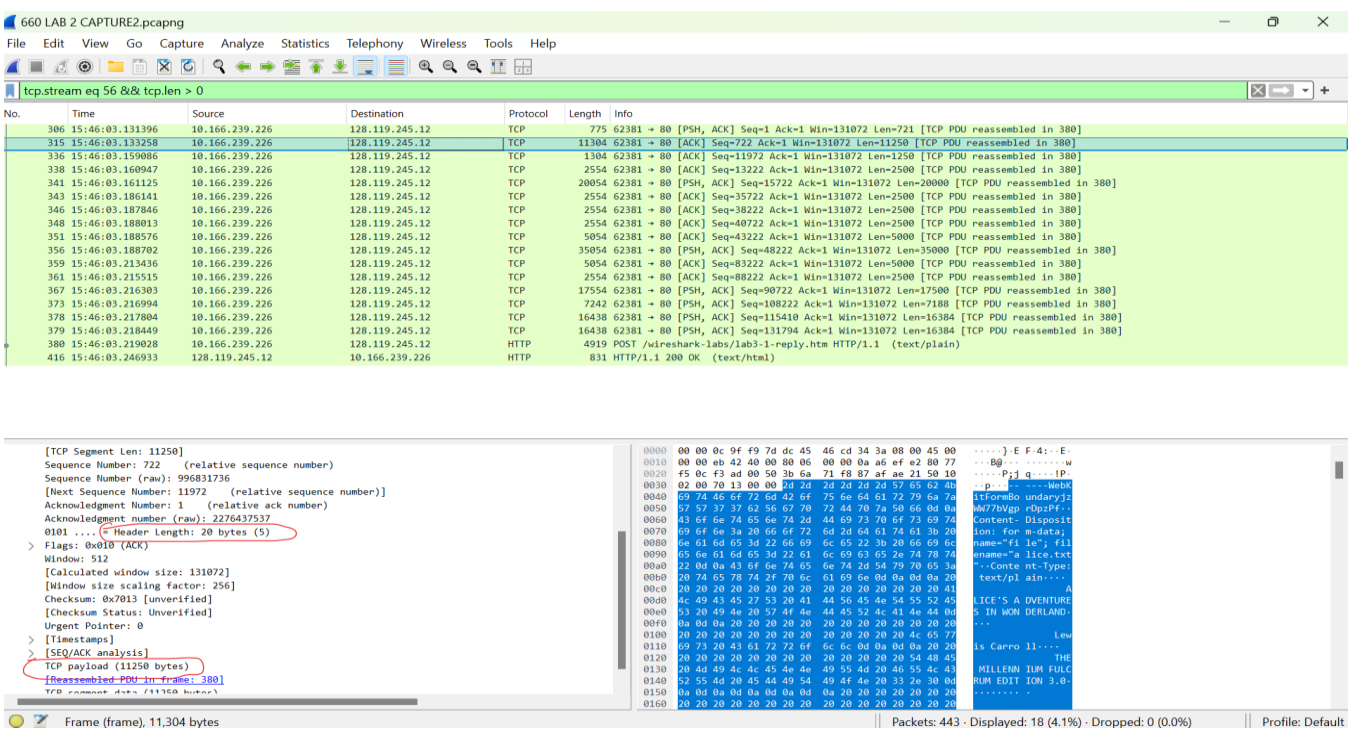


Figure 19b – Segment 2 total length

Segment 2 (Frame 315)

From the attached Figure 19b above, it can be deduced that:

- Header Length: 20 bytes
- TCP Payload: 11,250 bytes
- Total Length (Header + Payload): 11,270 bytes

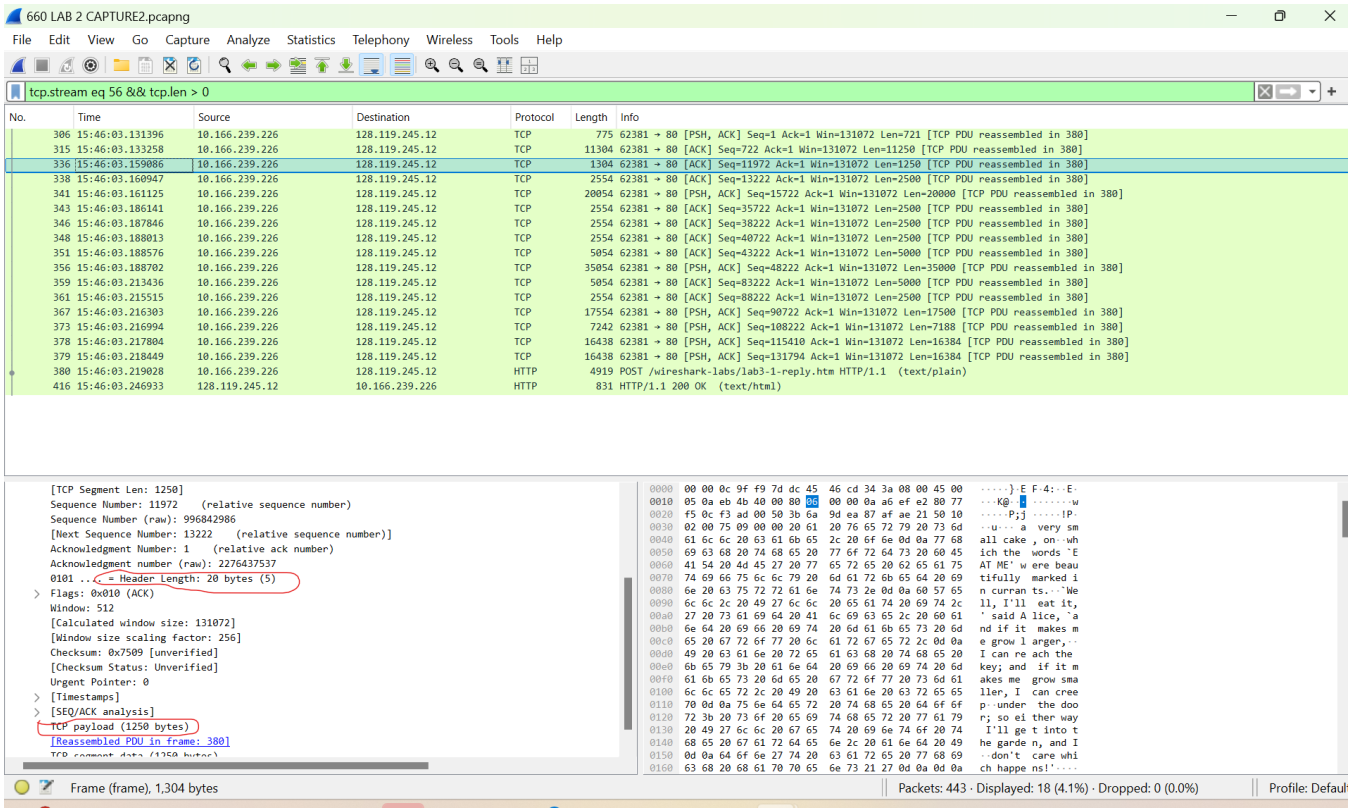


Figure 19c – Segment 3 total length

Segment 3 (Frame 336)

From the attached Figure 19c below it can be deduced that:

- Header Length: 20 bytes
- TCP Payload: 1,250 bytes
- Total Length (Header + Payload): 1,270 bytes

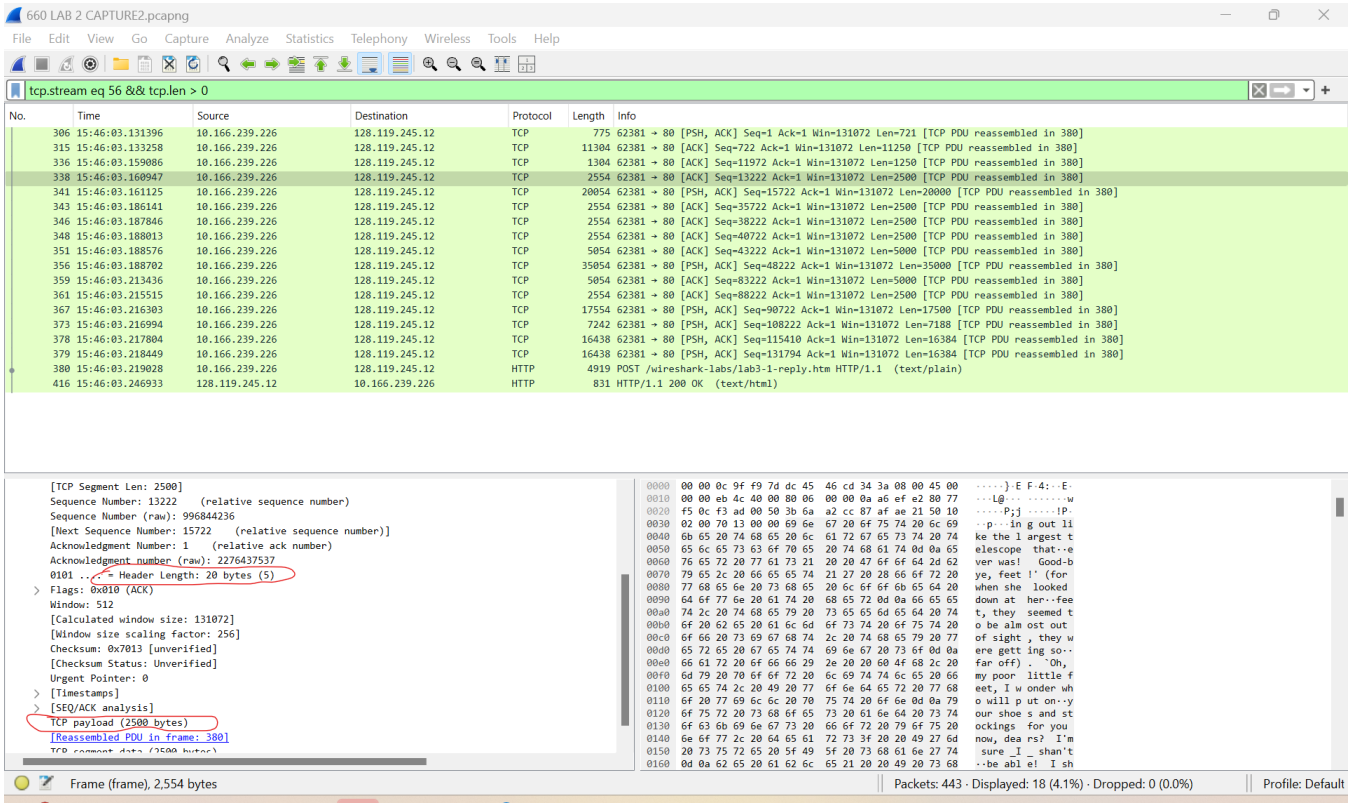


Figure 19d – Segment 4 total length

Segment 4 (Frame 338)

From the attached Figure 19d above it can be deduced that:

- Header Length: 20 bytes
- TCP Payload: 2,500 bytes
- Total Length (Header + Payload): 2,520 bytes

5.6 Flow Control (Receiver-Advertised Window)

I inspected the TCP acknowledgment packets to analyze the receiver-advertised window size.

From the first four ACKs in the trace as shown in Figure 20a to 20d, the receiver (gaia.cs.umass.edu) advertises the following buffer sizes:

Data Segment	ACK Frame	Calculated Window (bytes)
1	335	30,720
2	337	33,664
3	339	43,648
4	340	53,632

Screenshots of the TCP header fields for each ACK frame are attached below:

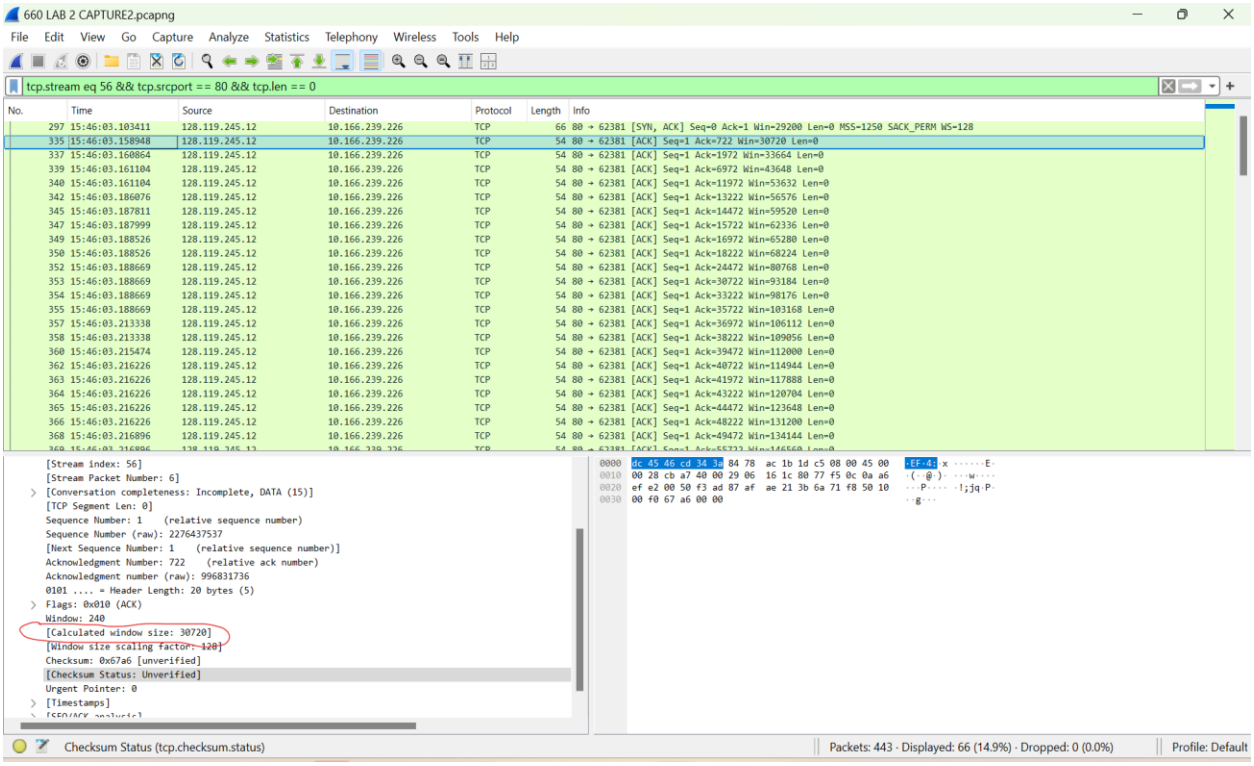


Figure 20a: Frame 335 — Calculated Window = 30,720 bytes

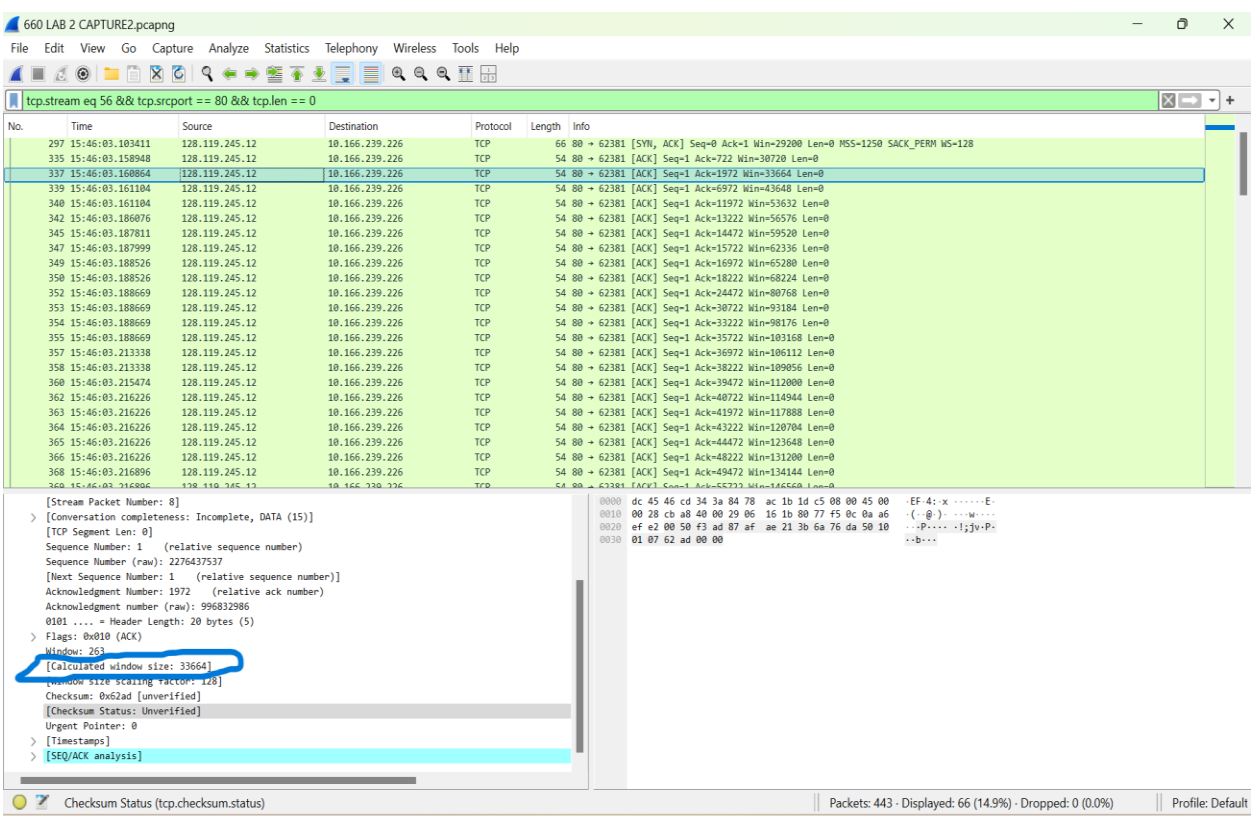


Figure 20b: Frame 337 — Calculated Window = 33,664 bytes

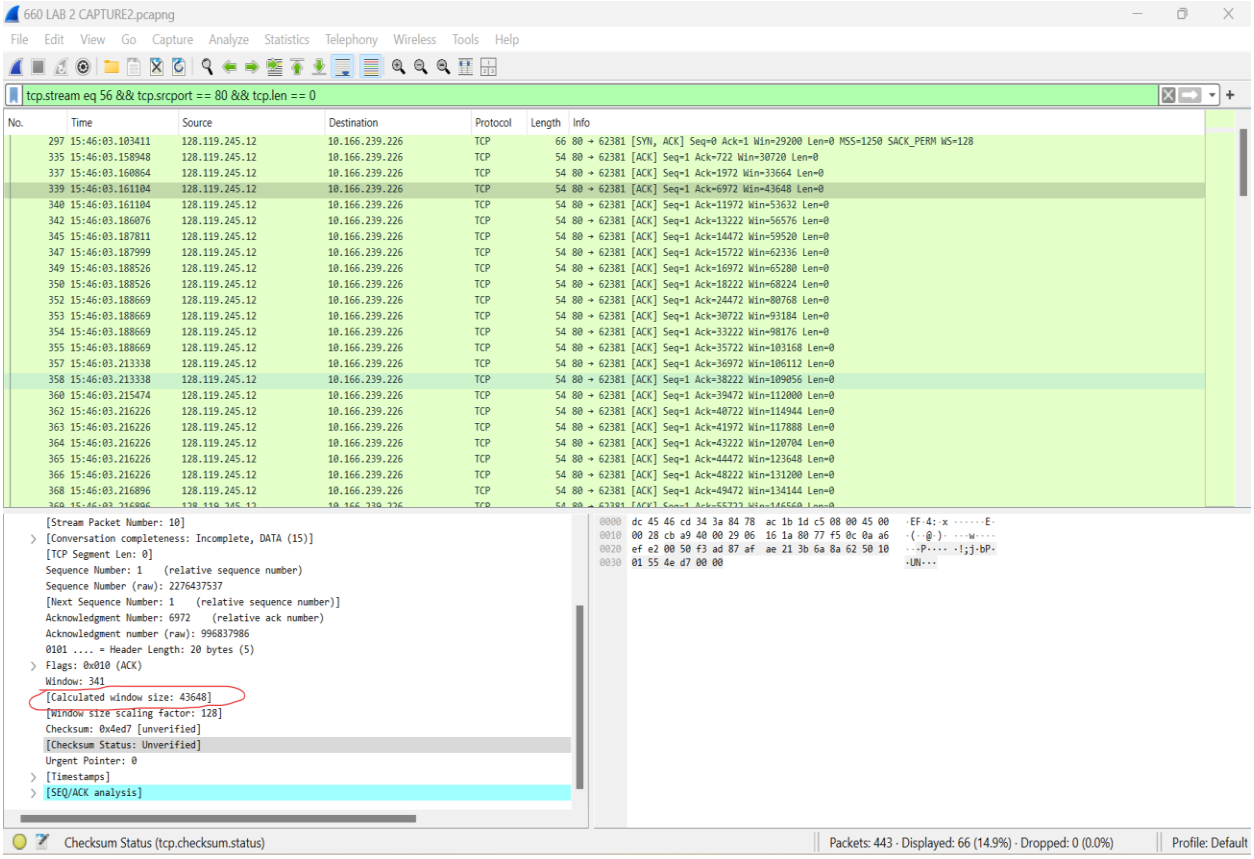


Figure 20c: Frame 339 — Calculated Window = 43,648 bytes

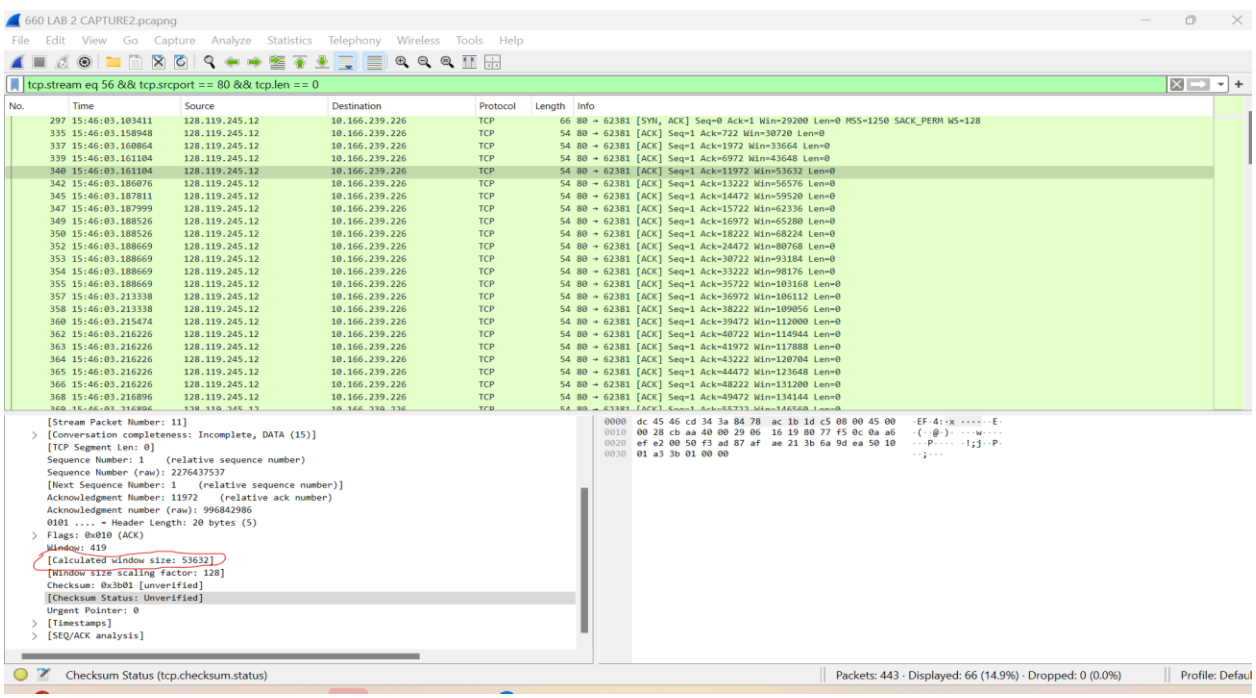


Figure 20d: Frame 340 — Calculated Window = 53,632 bytes

Minimum advertised buffer = 30,720 bytes (Frame 335) as seen above.
Since none of the advertised window sizes equal zero, **the sender is never throttled** for these first four data-carrying segments.

5.7 Retransmission Analysis

To check for retransmissions, I applied Wireshark's retransmission filter.

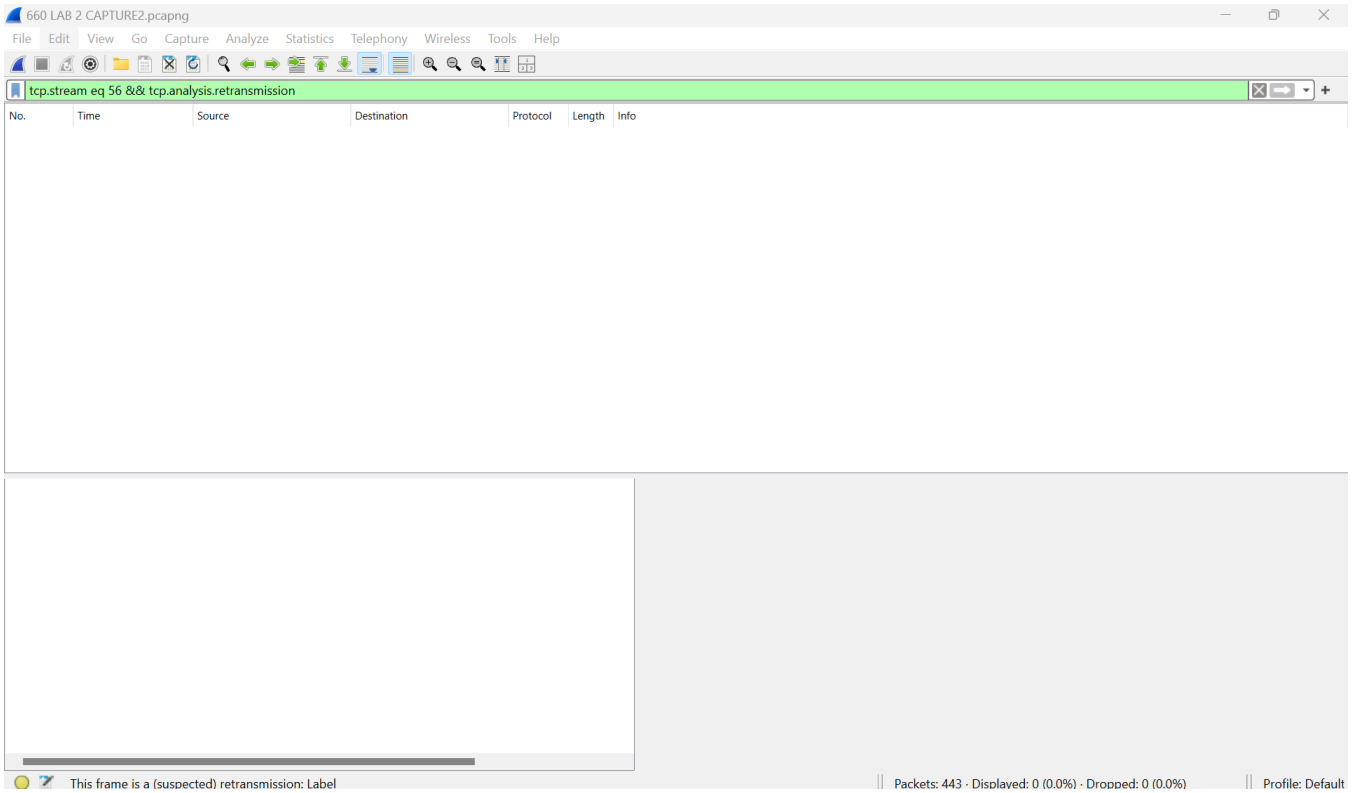


Figure 21 – Retransmission filter applied (tcp.analysis.retransmission)

No retransmitted TCP segments were observed during the file transfer.

6. Conclusion

In this project, I performed a packet-level analysis of a real TCP file upload using Wireshark. The captured evidence demonstrated how TCP establishes connections using a three-way handshake, segments large application data, manages flow control using advertised window sizes, and ensures reliable delivery without retransmissions.

By measuring RTT and inspecting TCP headers, I validated key transport-layer concepts using real network traffic. This project reflects practical skills directly applicable to **SOC analysis, network security monitoring, and blue-team investigations**.