Neil Kumar

Task 1: Hosting a Local Web Server

For this task, I set up a local web server using Python's built-in HTTP server module. From my terminal, I ran the command python3 -m http.server 8000, which created a server that listens on port 8000 and serves files from the current directory. This method was convenient since it doesn't require any extra software or configuration. Once the server was running, I confirmed it was working by visiting http://localhost:8080 in my browser. The browser displayed a directory listing of my local files, confirming that the server was successfully serving HTTP traffic.

Figure 1: Python server running

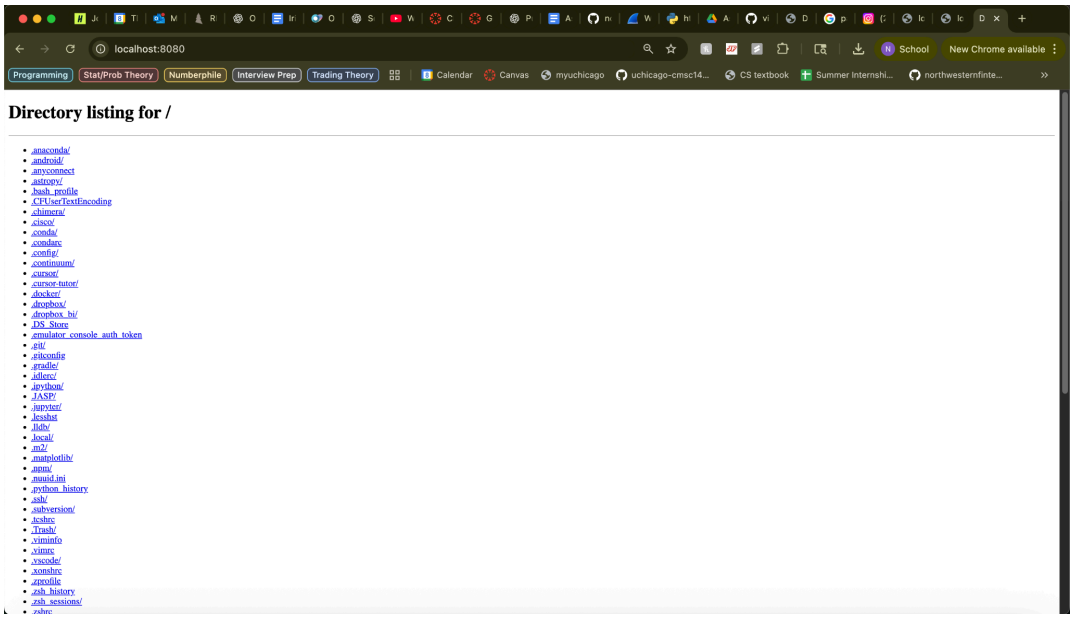

Figure 2: Directory Listing in Browser

Neil Kumar

This setup demonstrates how basic HTTP servers can make local resources available to clients. However, at this point, all communication between the client and server is unencrypted.

Task 2: Why HTTP Is Not Secure

HTTP traffic is transmitted in plaintext, meaning anyone with access to the same network can intercept and read it. There is no encryption or authentication to protect data from eavesdropping or tampering. To show this, I used Wireshark to capture packets between my browser and the local HTTP server.

I opened Wireshark and selected my Wi-Fi interface. Then I applied the filter tcp.port == 8080 to isolate traffic going through port 8080. While capturing, I refreshed the browser page being served by the local HTTP server.

The capture showed packets containing clear-text HTTP requests and responses. I could easily identify “GET /index.html HTTP/1.1” and the server’s “HTTP/1.0 200 OK” response. The bottom panel of Wireshark even displayed the full HTML body of the page, confirming that the transmitted data was human-readable.

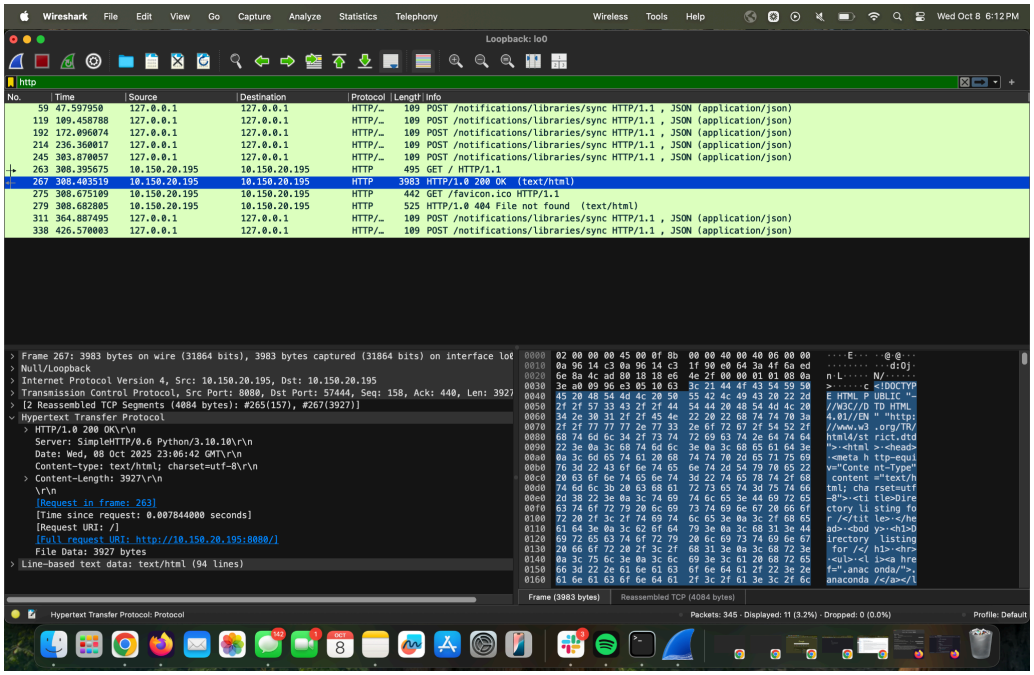

Figure 3: Wireshark showing HTTP 200 OK with plaintext HTML]

This demonstrates that HTTP lacks confidentiality. Anyone intercepting this traffic could read or modify the transmitted data, which is why HTTP is unsuitable for transmitting sensitive information like passwords or credit card details.

Task 3: Creating a Self-Signed Certificate and Upgrading to HTTPS

Neil Kumar

To secure communications, I generated a self-signed SSL certificate using OpenSSL. In my terminal, I ran the command: openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes.

This created a private key (key.pem) and a public certificate (cert.pem) valid for one year. During setup, I entered identifying information such as country, state, organization, and “localhost” as the common name since this is a local server.

Why can't you obtain an SSL certificate for your local web server from a certificate authority?

Certificate authorities only issue certificates for verified domain names that exist on the public internet. Since localhost and private IP addresses are not publicly registered, a CA cannot validate ownership, so a self-signed certificate must be used instead.

Next, I modified my Python server to use HTTPS by wrapping it with SSL. I used the Python **ssl** module to bind my certificate and key to the server and hosted it on port 8443. I then visited https://localhost:8443 in my browser. The browser displayed a warning because the certificate was self-signed, but I proceeded to access the site successfully.

I repeated the Wireshark capture using the display filter tcp.port == 8443 while visiting the HTTPS site. The captured packets showed the TLS handshake process, including the “Client Hello” and “Server Hello” messages. These packets contained information about encryption methods and keys but did not reveal any actual content.

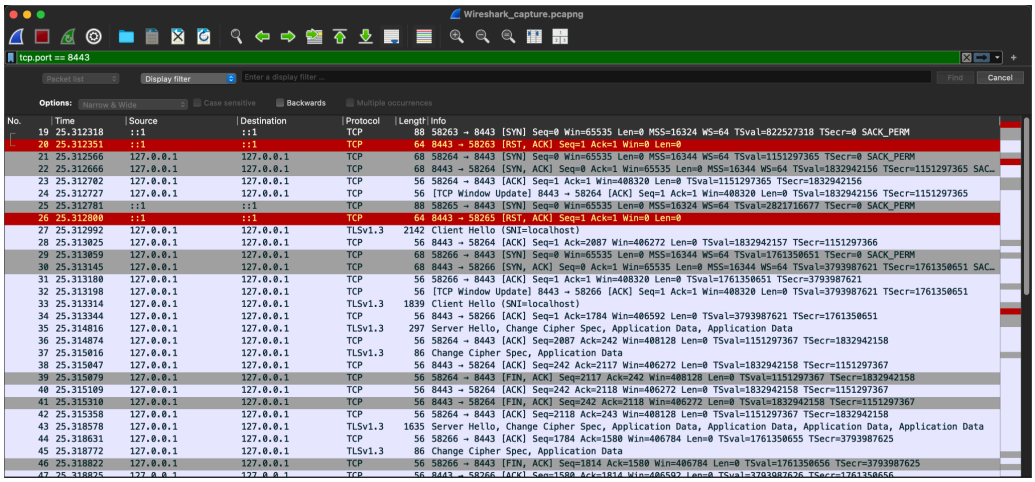

Figure 5: Relevant TLS handshake

Following the handshake, all subsequent packets appeared as “Application Data.” The data field in these packets was completely unreadable, confirming that the payload was encrypted.

Neil Kumar

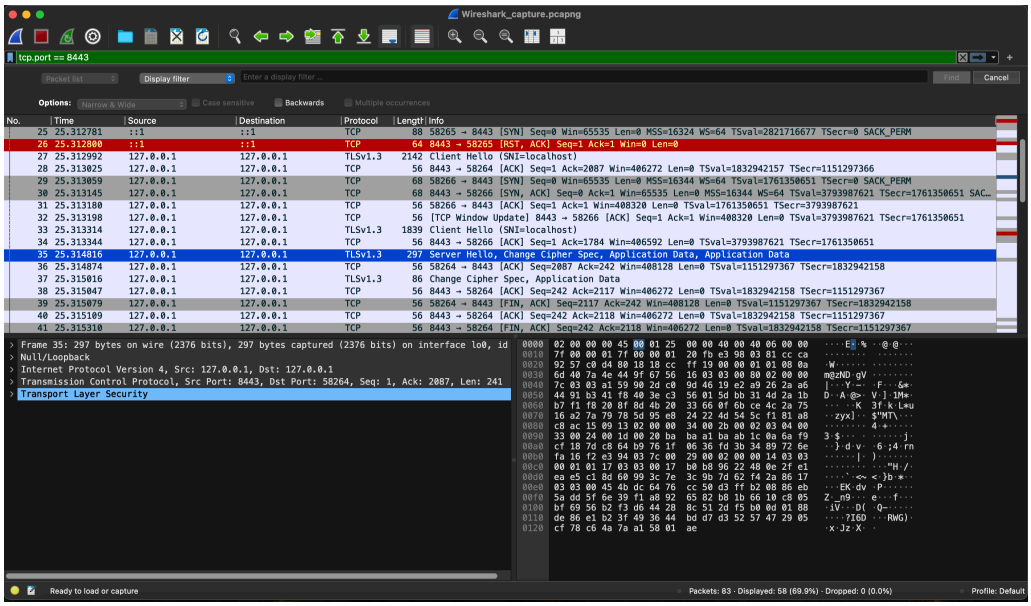

Figure 6: Encrypted TLS Application Data

The difference between this trace and the earlier HTTP one is clear. In the HTTP trace, the contents were visible in plaintext, while in the HTTPS trace, everything after the handshake was encrypted. This encryption ensures that even if someone captures the traffic, they cannot read or alter it.

In summary, upgrading the server to HTTPS transforms readable network traffic into secure, encrypted communication. It prevents eavesdropping, preserves data integrity, and authenticates the server's identity through digital certificates. This demonstrates the importance of HTTPS as the foundation of secure web communication.