CMSC 30350 Assignment 1

Chris Low

October 2025

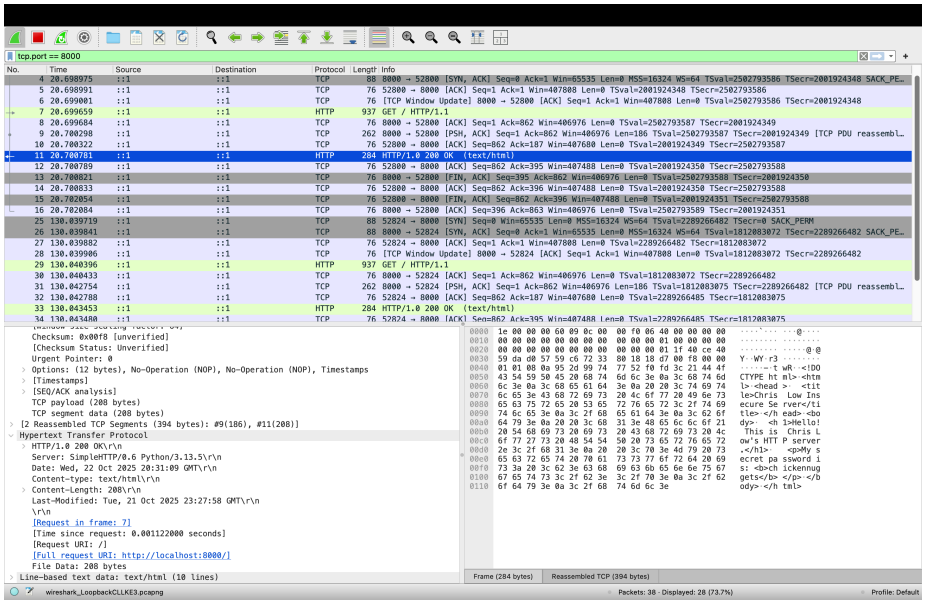

Figure 1: At the bottom right corner, you see all the information in my website, including the secret password - "chickennuggets".

Task 1 & 2: Hosting a Local HTTP Server and Understanding Insecurity of HTTP

Setting up the HTTP Server

For this part, I used Python's built-in `http.server` class to host a simple local website on port 8000. The page included a line of text and, by mistake, also revealed a "secret password." The HTML file looked like this:

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Chris Low Insecure Server</title>
</head>
<body>
  <h1>Hello! This is Chris Low's HTTP server.</h1>
  <p>My secret password is: <b>chickennuggets</b></p>
</body>
</html>
```

I ran the following command in my terminal:

```
python3 -m http.server 8000
```

Then I opened `http://localhost:8000` in my browser to access the page.

Capturing HTTP Traffic Using Wireshark

To analyze the network traffic, I used Wireshark:

1. I selected the **Loopback** interface (`lo0` on macOS/Linux).
2. I started packet capture by double-clicking the interface.
3. I filtered the traffic using the display filter: `tcp.port == 8000`.
4. While Wireshark was capturing, I refreshed `http://localhost:8000` in my browser.
5. I stopped the capture and found the packet that said `GET / HTTP/1.1`.
6. I expanded the "Hypertext Transfer Protocol" section to see the plain text request.
7. I then found the `HTTP/1.0 200 OK` response and expanded it to see the full HTML of my webpage.

The HTML source, including the line with the "secret password" (shown in the screenshot above, you can see "chickennuggets"), appeared clearly in plain text inside the Wireshark packet details. This confirmed that HTTP traffic is not encrypted.

Why HTTP Is Not Secure

HTTP does not provide encryption. All information sent between the browser and the server is visible to anyone who can access the network. If an attacker is on the same network (for example, the same Wi-Fi), they can capture and read these packets with tools like Wireshark.

They can see, for example, the exact URL being visited, the text and images of the page, and any information entered into forms (in my case, the password `chickennuggets`).

This shows us the core weakness of HTTP: it does not protect the confidentiality or integrity of data in transit.

A packet trace named `http-trace.pcapng` has been included as part of the submission. It contains the captured packets showing the unencrypted HTTP request and response.

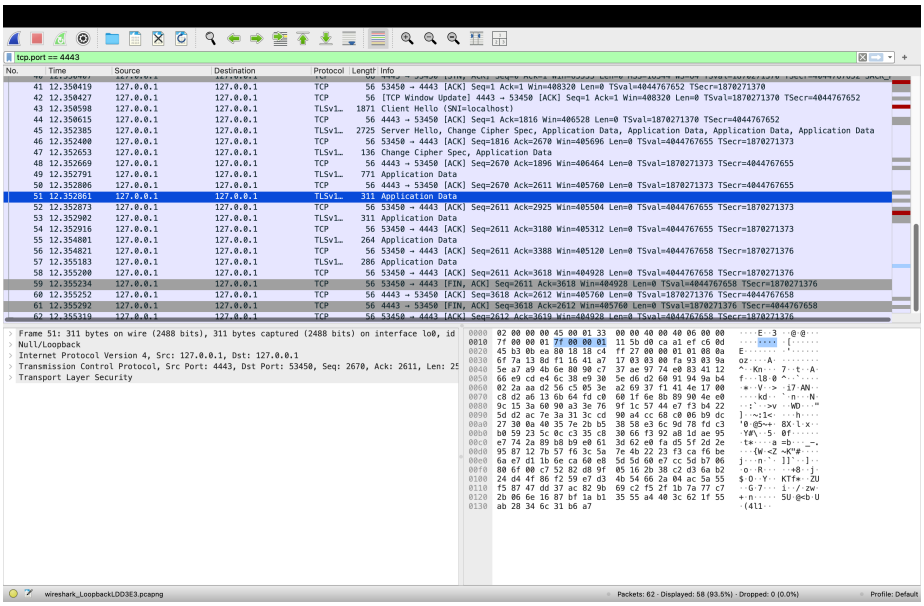

Figure 2: At the bottom right corner, we now see that our password (or any h1) is encrypted.

Task 3: Enabling HTTPS with a Self-Signed Certificate

(a) Why I Cannot Get a Real SSL Certificate for localhost

A Certificate Authority (CA) is a trusted third party that gives out SSL certificates to prove that a website is who it says it is. When a CA issues a certificate for `google.com`, it checks the public DNS records and confirms that the person requesting the certificate really controls that domain.

`localhost` (or `127.0.0.1`) is not a public domain. It is a special address that every computer uses to refer to itself. There is no single "owner" of `localhost`, so no CA can verify that I control it. Because of this, CAs will never issue a trusted certificate for `localhost`.

For testing, I must create a self-signed certificate. This means I act as my own CA just for my machine. Browsers will not automatically trust this certificate, but the traffic is still encrypted.

(b) Creating a Self-Signed Certificate

I generated the key and certificate using `openssl` in my terminal:

```
openssl req -x509 -newkey rsa:2048 -nodes -keyout key.pem -out cert.pem -days 365
```

Explanation:

- `req -x509`: Self-signed certificate.
- `-newkey rsa:2048`: New 2048-bit RSA key pair.
- `-nodes`: Do not encrypt the private key.
- `-keyout key.pem`: Save the private key as `key.pem`.
- `-out cert.pem`: Save the certificate as `cert.pem`.
- `-days 365`: Valid for one year.

When prompted, I set the "Common Name (CN)" to `localhost`, since that's the domain I am serving on.

This produced two files:

- `key.pem` — my private key
- `cert.pem` — my public certificate

(c) Launching the HTTPS Server

I then modified my Python web server code to use TLS.

```
import http.server
import ssl

server_address = ('localhost', 4443)
key_file = "key.pem"
cert_file = "cert.pem"

httpd = http.server.HTTPServer(server_address, http.server.SimpleHTTPRequestHandler)

context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile=cert_file, keyfile=key_file)

# Wrap the socket using the context
httpd.socket = context.wrap_socket(httpd.socket, server_side=True)

print(f"Starting HTTPS server on https://{server_address[0]}:{server_address[1]} ...")
httpd.serve_forever()
```

I committed and pushed this server code to GitHub as part of my assignment.

When I visited `https://localhost:4443` in Google Chrome, I saw the message: "Your connection is not private."

This happens because my certificate is self-signed and not issued by a trusted CA. I bypassed or trusted the warning locally so the browser could continue and make a full HTTPS connection.

(d) Capturing HTTPS Traffic

Next, I captured the traffic using Wireshark:

1. Select the **Loopback** interface (`lo0`) in Wireshark.
2. Start capturing.
3. Use the display filter: `tcp.port == 4443`
4. Visit `https://localhost:4443` in the browser.

Unlike my HTTP trace, this time there were **no plain HTTP packets**. Wireshark showed packets labeled `TLSv1.3`, including Client Hello, Server Hello, Change Cipher Spec, and Encrypted Application Data.

When I clicked on an `Application Data` packet (shown in Figure 2), the content was unreadable — just binary encrypted data. I could no longer see my HTML tags or the "chickennuggets" password.

This shows that TLS encryption is working. The client and server performed a handshake to agree on encryption keys, and then encrypted all traffic. Even if someone intercepted the packets, they would not be able to read the content.

The file `https-trace.pcapng` contains my TLS capture.