

## Documentation & Responses

### Task 1 (Hosting a Local Web Server)

In setting up the HTTP web server I used Python's HTTP server class. This was done in the following simple manner:

1. '\$ mkdir website'
2. '\$ cd website'
3. '\$ code index.html'

The file 'index.html' was then populated with the following information to add some text to the HTTP server. [Note: I consulted Claude Sonnet 4.5 for formatting guidance on an HTML file.]

```
<!DOCTYPE html>
<html>
  <head>
    <title>Local Server</title>
  </head>
  <body>
    <h1>Hello! This is my web server.</h1>
    <p> This is running on HTTP.</p>
  </body>
</html>
```

4. Then, in the 'website' folder, I executed the command '\$ python3 -m http.server 8000'.
5. I then entered the URL 'http://localhost:8000' into my web browser and verified my HTTP page:

**Hello! This is my web server.**

This is running on HTTP.

## Task 2 (Insecurity of HTTP)

Having set up the HTTP server and using Wireshark, I then began to capture HTTP traffic. Considering that the server created in the prior task is a local web server, I captured traffic on the Loopback interface (designated as ‘Loopback: lo0’ in Wireshark). With packet traffic now being visible, I paired it down to HTTP traffic using the filter bar in Wireshark.

Next, I refreshed the page ‘http://localhost:8000’ a few times to generate packet traffic that could be intercepted.

8282	48046.479528	127.0.0.1	127.0.0.1	TCP	65	51645 → 8000 [WIN] Seq=8 Win=65535 Len=0 MSS=16344 WS=64 TSval=2937848136 TSecr=0 SACK_PERM
8283	48046.479608	127.0.0.1	127.0.0.1	TCP	66	8000 → 51645 [WIN, ACK] Seq=8 Ack=65535 Len=0 MSS=16344 WS=64 TSval=4129583818 TSecr=2937848136 SACK_PERM
8284	48046.479718	127.0.0.1	127.0.0.1	TCP	56	51645 → 8000 [ACK] Seq=3 Ack=1 W=488328 Len=0 TSval=2937848136 TSecr=4129583818
8285	48046.479727	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 8000 → 51645 [ACK] Seq=1 Ack=1 W=488328 Len=0 TSval=4129583818 TSecr=2937848136
8286	48046.480030	127.0.0.1	127.0.0.1	HTTP	55	GET / HTTP/1.1
8287	48046.479821	127.0.0.1	127.0.0.1	TCP	56	8000 → 51645 [ACK] Seq=1 Ack=453 Win=487872 Len=0 TSval=4129583818 TSecr=2937848136
8288	48046.489586	127.0.0.1	127.0.0.1	TCP	243	8000 → 51645 [PSH, ACK] Seq=1 Ack=453 Win=487872 Len=187 TSval=4129583828 TSecr=2937848136 [TCP PDU reassembled in 8298]
8289	48046.489542	127.0.0.1	127.0.0.1	TCP	56	51645 → 8000 [ACK] Seq=453 Ack=188 Win=488192 Len=0 TSval=2937848146 TSecr=4129583828
8290	48046.489758	127.0.0.1	127.0.0.1	HTTP	256	HTTP/1.0 200 OK (text/html)
8291	48046.489778	127.0.0.1	127.0.0.1	TCP	56	51645 → 8000 [ACK] Seq=453 Ack=398 Win=488888 Len=0 TSval=2937848146 TSecr=4129583828
8292	48046.489864	127.0.0.1	127.0.0.1	TCP	56	8000 → 51645 [PSH, ACK] Seq=298 Ack=453 W=487872 Len=0 TSval=4129583828 TSecr=2937848146
8293	48046.489867	127.0.0.1	127.0.0.1	TCP	56	51645 → 8000 [ACK] Seq=453 Ack=393 W=488888 Len=0 TSval=2937848146 TSecr=4129583828
8294	48046.489923	127.0.0.1	127.0.0.1	TCP	56	51645 → 8000 [FIN, ACK] Seq=453 Ack=393 W=488888 Len=0 TSval=2937848146 TSecr=4129583828
8295	48046.489963	127.0.0.1	127.0.0.1	TCP	56	8000 → 51645 [ACK] Seq=393 Ack=454 W=487872 Len=0 TSval=4129583828 TSecr=2937848146

Above, Wireshark displays a number of transmitted packets that essentially confirm the HTTP server is up and running as intended. Indeed, there are multiple GET requests for the server and others with a code of ‘200’, indicating that the server has been fetched successfully. Following the TCP stream for one of the GET / HTTP/1.1 requests we see the following:

```
GET / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:144.0) Gecko/20100101 Firefox/144.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br, zstd
Connection: Keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Priority: u=0, i

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.10.10
Date: Fri, 24 Oct 2025 14:45:47 GMT
Content-type: text/html
Content-Length: 202
Last-Modified: Fri, 24 Oct 2025 14:33:27 GMT

<!DOCTYPE html>
<html>
  <head>
    <title>Local Server</title>
  </head>
  <body>
    <h1>Hello! This is my web server.</h1>
    <p> This is running on HTTP.</p>
  </body>
</html>
```

This is troubling because it is basically confirmation that HTTP is unencrypted. All information exchanged between the client and server is visible in plaintext here. Both the HTTP request headers (indicating what the browser sent) and the HTTP response (the content of the page) are completely visible.

**Response:** HTTP is fundamentally insecure because any eavesdropper can ‘sniff’ traffic and read any credentials or content on the page. Whether this consists of URLs, body content, or headers, everything is readable with HTTP to someone capturing packets on the same network. Importantly, this could also include any sensitive data (such as passwords or personal information). This is because all information for the server is in plaintext and is completely unencrypted. As a result, it is just as visible to me as it would be to someone using Wireshark on the same network.

### Task 3 (SSL Certificate and HTTPS Upgrade)

**Response:** It is not possible to obtain an SSL certificate for my local web server from a certificate authority because these are reserved for public domain names. Since I am running a local server (localhost) that is not a public domain and cannot conclusively be tied to me, a CA will not grant a certificate. Instead, I would need a domain name that I both verifiably control and is publicly accessible.

Nonetheless, I can generate a self-signed certificate from my machine. This is done by running the following command:

```
‘openssl req -x509 -newkey rsa:4096 -nodes -out cert.pem -keyout key.pem -days 365’
```

I then filled in the requisite information (fields such as location and organization name), which generated the files cert.pem (with my certificate) and key.pem (with my private key).

Next, I created a short Python file to serve my HTTPS server, with the following contents:

```
import http.server
import ssl

server_address = ('localhost', 8443)
httpd = http.server.HTTPServer(server_address,
http.server.SimpleHTTPRequestHandler)

httpd.socket = ssl.wrap_socket(httpd.socket,
```

```
server_side=True,
certfile='cert.pem',
keyfile='key.pem',
ssl_version=ssl.PROTOCOL_TLS)
```

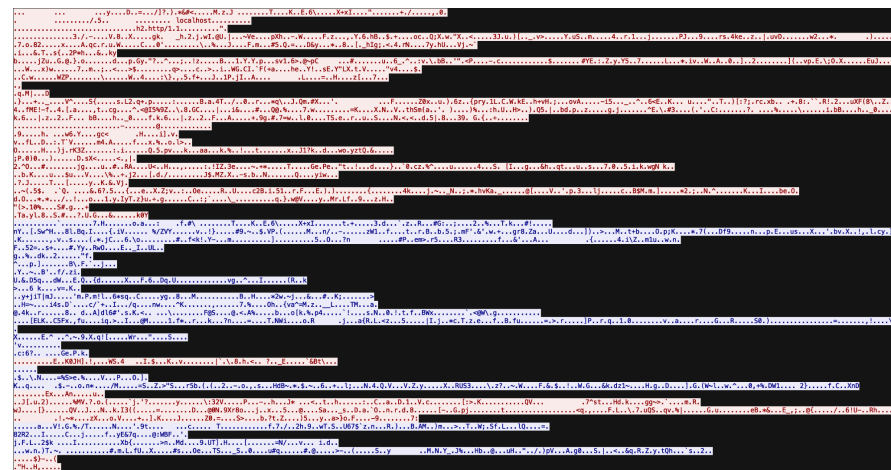
```
httpd.serve_forever()
```

Upon running ‘\$ python3 -m https\_server.py’, I entered the URL ‘https://localhost:8443’ and verified my HTTPS page.

Then, I started a capture and refreshed the page ‘https://localhost:8443’ to generate packet traffic.

1	0.000000	127.0.0.1	127.0.0.1	TCP	68	51951 → 8443 [SYN] Seq=0 Win=0 Len=0 MSS=16344 WS=64 TSval=3846794852 TSecr=0 SACK_PERM
2	0.000220	127.0.0.1	127.0.0.1	TCP	68	8443 → 51951 [SYN, ACK] Seq=0 Ack=1 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852 SACK_PERM
3	0.000450	127.0.0.1	127.0.0.1	TCP	56	51951 → 8443 [ACK] Seq=1 Ack=1 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
4	0.000280	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 8443 → 51951 [ACK] Seq=1 Ack=1 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
5	0.001343	127.0.0.1	127.0.0.1	TLSv1	2549	Client Hello (Certificate Request)
6	0.001815	127.0.0.1	127.0.0.1	TLSv1	1541	Server Hello, Change Cipher Spec, Application Data
7	0.002074	127.0.0.1	127.0.0.1	TLSv1	1385	Server Hello, Change Cipher Spec, Application Data
8	0.002092	127.0.0.1	127.0.0.1	TLSv1	56	51951 → 8443 [ACK] Seq=2538 Ack=138 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
9	0.002535	127.0.0.1	127.0.0.1	TLSv1	136	Change Cipher Spec, Application Data
10	0.002568	127.0.0.1	127.0.0.1	TLSv1	56	8443 → 51951 [ACK] Seq=1330 Ack=2538 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
11	0.002772	127.0.0.1	127.0.0.1	TLSv1	311	Application Data
12	0.002790	127.0.0.1	127.0.0.1	TCP	56	51951 → 8443 [ACK] Seq=2538 Ack=138 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
13	0.002808	127.0.0.1	127.0.0.1	TLSv1	565	Application Data
14	0.002836	127.0.0.1	127.0.0.1	TCP	56	8443 → 51951 [ACK] Seq=1585 Ack=2839 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
15	0.002894	127.0.0.1	127.0.0.1	TLSv1	265	Application Data
16	0.003222	127.0.0.1	127.0.0.1	TCP	56	51951 → 8443 [ACK] Seq=3039 Ack=1794 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
17	0.003593	127.0.0.1	127.0.0.1	TLSv1	280	Application Data
18	0.004511	127.0.0.1	127.0.0.1	TCP	56	51951 → 8443 [ACK] Seq=3039 Ack=2819 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
19	0.005784	127.0.0.1	127.0.0.1	TCP	56	8443 → 51951 [FIN, ACK] Seq=3039 Ack=2819 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
20	0.005759	127.0.0.1	127.0.0.1	TCP	56	51951 → 8443 [ACK] Seq=3039 Ack=2819 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
21	0.005773	127.0.0.1	127.0.0.1	TLSv1	40	Application Data
22	0.005785	127.0.0.1	127.0.0.1	TCP	56	51951 → 8443 [FIN, ACK] Seq=3039 Ack=2819 Win=0 Len=0 MSS=16344 WS=64 TSval=2848474355 TSecr=3846794852
23	0.005851	127.0.0.1	127.0.0.1	TCP	44	8443 → 51951 [RST] Seq=2819 Win=0 Len=0

As with the second task, Wireshark displays a number of transmitted packets that essentially confirm the HTTPS server is up and running as intended. Following the TCP stream for one of the Client Hello requests we see the following:



This signifies that the encryption (specifically, TLS encryption) is working as intended because all of the information exchanged between the client and server is gibberish to an eavesdropper. Although someone can tell that a connection has

occurred, the content of it is completely scrambled and any private information is unintelligible.