

Public Key Infrastructure

1. Host a Local Web Server

To begin the assignment, I hosted a local web server on my computer using Python's built-in HTTP server module. I chose this method because it is lightweight, requires no additional software, and allows me to easily serve static HTML files for local testing. For full transparency, I used Chat GPT to help with my code at times (mostly debugging), see code pastes throughout submission.

Steps Taken:

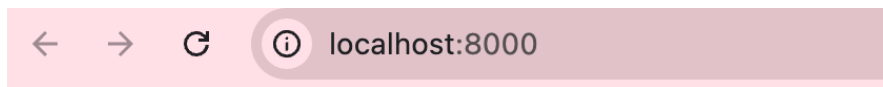
1. Verified Python installation
2. Created a project directory and HTML file
 - a. I created a new directory for this lab and added a simple web page named index.html

```
mkdir -p ~/pki-lab && cd ~/pki-lab
cat > index.html <<'EOF'
<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>PKI Lab</title></head>
<body>
  <h1>Hello, HTTP!</h1>
  <p>This is the insecure version.</p>
</body>
</html>
EOF
```

3. Started the local web server
 - a. I used Python's HTTP module to serve the files in this directory

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

4. Tested in the browser
 - a. I opened a web browser and navigated to <http://localhost:8000>
 - b. The page displayed the message "Hello, HTTP! This is the insecure version."
Confirming the local server was working correctly.



Hello, HTTP!

This is the insecure version.

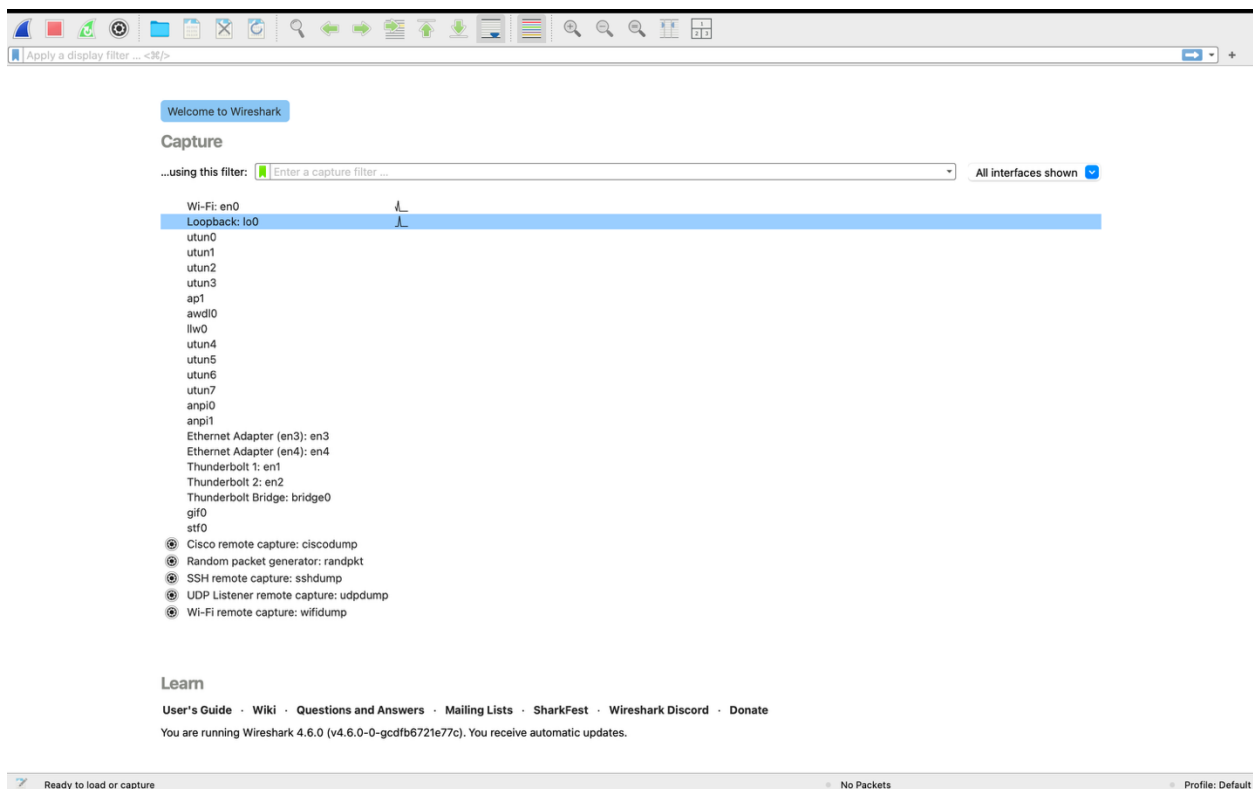
2. Identify why HTTP is not secure.

How an eavesdropper can "sniff" web traffic between a client and HTTP server: The critical flaw in HTTP is that it sends all information in plaintext, without any encryption or integrity protection. Because of this, anyone connected to the same network can use a packet-sniffing tool like Wireshark to intercept and read the traffic as it travels between the client and the server. An eavesdropper can see exactly what is being communicated — including which resources are being fetched (such as / or /favicon.ico), the full contents of web pages, and even details like cookies or login information.

In other words, HTTP offers no confidentiality or safeguard against interception or tampering. An attacker could easily observe or modify the data before it reaches its destination. I demonstrate this below using a Wireshark capture of my local web server.

Using Wireshark:

Downloaded Wireshark and installed ChmodBPF so that I had permission to capture local network traffic.



Next, I opened Wireshark and selected the Loopback (lo0) interface, which records traffic sent between programs running on my own machine (like my web browser and local web server).

Then, with Wireshark running, I went to my browser and refreshed my local site at <http://localhost:8000> several times. This generated packets that Wireshark captured in real time.

The screenshot shows the Wireshark interface with a packet list on the left and packet details on the right. The packet list shows a series of packets, including TCP and HTTP traffic. The packet details pane shows the structure of the selected packet, including the Ethernet II, Internet Protocol Version 4, User Datagram Protocol, and Hypertext Transfer Protocol layers.

No.	Time	Source	Destination	Protocol	Length	Info
18	5.770112	:::1	:::1	TCP	76	56495 → 8000 [FIN, ACK] Seq=780 Ack=365 Win=407424 Len=0 TSval=4222813725 TSecr=2833800831
19	5.770141	:::1	:::1	TCP	76	8000 → 56495 [ACK] Seq=365 Ack=781 Win=406976 Len=0 TSval=2833800831 TSecr=4222813725
20	5.814882	:::1	:::1	HTTP	775	GET /favicon.ico HTTP/1.1
21	5.814924	:::1	:::1	TCP	76	8000 → 56495 [ACK] Seq=1 Ack=700 Win=407040 Len=0 TSval=3015778752 TSecr=1291702977
22	5.816151	:::1	:::1	TCP	261	8000 → 56495 [PSH, ACK] Seq=1 Ack=700 Win=407040 Len=185 TSval=3015778753 TSecr=1291702977 [TCP PDU rea...
23	5.816177	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=700 Ack=186 Win=407552 Len=0 TSval=1291702978 TSecr=3015778753
24	5.816193	:::1	:::1	HTTP	545	HTTP/1.0 404 File not found (text/html)
25	5.816201	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=700 Ack=655 Win=407104 Len=0 TSval=1291702978 TSecr=3015778753
26	5.816229	:::1	:::1	TCP	76	8000 → 56495 [FIN, ACK] Seq=655 Ack=700 Win=407040 Len=0 TSval=3015778753 TSecr=1291702978
27	5.816245	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=700 Ack=656 Win=407104 Len=0 TSval=1291702978 TSecr=3015778753
28	5.816366	:::1	:::1	TCP	76	56495 → 8000 [FIN, ACK] Seq=700 Ack=656 Win=407104 Len=0 TSval=1291702978 TSecr=3015778753
29	5.816395	:::1	:::1	TCP	76	8000 → 56495 [ACK] Seq=656 Ack=701 Win=407040 Len=0 TSval=3015778753 TSecr=1291702978
30	8.221215	10.150.68.238	239.255.255.250	SSDP	203	M-SEARCH * HTTP/1.1
31	9.222608	10.150.68.238	239.255.255.250	SSDP	203	M-SEARCH * HTTP/1.1
32	10.224756	10.150.68.238	239.255.255.250	SSDP	203	M-SEARCH * HTTP/1.1
33	11.224808	10.150.68.238	239.255.255.250	SSDP	203	M-SEARCH * HTTP/1.1
34	21.429408	10.150.68.238	224.0.0.251	MDNS	97	Standard query 0x0000 PTR _spotify-connect_tcp.local, "QM" question
35	21.429568	fe80::c40c:51ff:fe...	ff02::fb	MDNS	97	Standard query 0x0000 PTR _spotify-connect_tcp.local, "QM" question
36	21.429672	fe80::c40c:51ff:fe...	ff02::fb	MDNS	97	Standard query 0x0000 PTR _spotify-connect_tcp.local, "QM" question
37	21.429780	fe80::c40c:51ff:fe...	ff02::fb	MDNS	97	Standard query 0x0000 PTR _spotify-connect_tcp.local, "QM" question
38	21.507683	10.150.68.238	239.255.255.250	SSDP	157	M-SEARCH * HTTP/1.1

After stopping the capture (by pressing the red square), I applied the filter:

The screenshot shows the Wireshark interface with the filter 'tcp.port == 8000' applied. The packet list shows only the packets related to the HTTP server, including the GET /favicon.ico request and the 200 OK response.

No.	Time	Source
18	5.770112	:::1

This displayed only the packets related to my HTTP server. The list clearly showed GET / HTTP/1.1 requests from the client and HTTP/1.0 200 OK responses from the server, proving that the browser and server were exchanging data in plaintext.

The screenshot shows the Wireshark interface with the packet list and packet details. The packet list shows the GET /favicon.ico request and the 200 OK response. The packet details pane shows the structure of the selected packet, including the Ethernet II, Internet Protocol Version 4, Transmission Control Protocol, and Hypertext Transfer Protocol layers.

No.	Time	Source	Destination	Protocol	Length	Info
11	5.754175	:::1	:::1	TCP	76	8000 → 56495 [ACK] Seq=1 Ack=780 Win=406976 Len=0 TSval=2833800816 TSecr=4222813710
12	5.769189	:::1	:::1	TCP	262	8000 → 56495 [PSH, ACK] Seq=1 Ack=780 Win=406976 Len=186 TSval=2833800831 TSecr=4222813710 [TCP PDU...
13	5.769225	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=780 Ack=187 Win=407552 Len=0 TSval=4222813725 TSecr=2833800831
14	5.769391	:::1	:::1	HTTP	253	HTTP/1.0 200 OK (text/html)
15	5.769409	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=780 Ack=365 Win=406976 Len=0 TSval=2833800816 TSecr=4222813710
16	5.769463	:::1	:::1	TCP	76	8000 → 56495 [FIN, ACK] Seq=364 Win=406976 Len=0 TSval=2833800831 TSecr=4222813725
17	5.769480	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=780 Ack=365 Win=406976 Len=0 TSval=2833800831 TSecr=4222813725
18	5.770112	:::1	:::1	TCP	76	56495 → 8000 [FIN, ACK] Seq=780 Ack=365 Win=406976 Len=0 TSval=4222813725 TSecr=2833800831
19	5.770141	:::1	:::1	TCP	76	8000 → 56495 [ACK] Seq=365 Ack=781 Win=406976 Len=0 TSval=2833800831 TSecr=4222813725
20	5.814882	:::1	:::1	HTTP	775	GET /favicon.ico HTTP/1.1
21	5.814924	:::1	:::1	TCP	76	8000 → 56495 [ACK] Seq=1 Ack=700 Win=407040 Len=0 TSval=3015778752 TSecr=1291702977
22	5.816151	:::1	:::1	TCP	261	8000 → 56495 [PSH, ACK] Seq=1 Ack=700 Win=407040 Len=185 TSval=3015778753 TSecr=1291702977 [TCP PDU...
23	5.816177	:::1	:::1	TCP	76	56495 → 8000 [ACK] Seq=700 Ack=186 Win=407552 Len=0 TSval=1291702978 TSecr=3015778753

By right-clicking on one of the packets and selecting Follow → TCP Stream, I was able to see the entire HTTP conversation between the client and server in readable text. This included both the request and the full HTML response of my page.

```

GET / HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
sec-ch-ua: "Google Chrome";v="141", "Not?A_Brand";v="8", "Chromium";v="141"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/141.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: cross-site
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.9
Cookie: ajs_anonymous_id=a2364d4e-abfe-41e6-b513-8a918d28ccae

HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.9.16
Date: Thu, 23 Oct 2025 22:45:14 GMT
Content-type: text/html
Content-Length: 177
Last-Modified: Thu, 23 Oct 2025 22:27:55 GMT

<!doctype html>
<html lang="en">
<head><meta charset="utf-8"><title>PKI Lab</title></head>
<body>
<h1>Hello, HTTP!</h1>
<p>This is the insecure version.</p>
</body>
</html>

```

This demonstrates why HTTP is not secure: the communication between the client and the server is completely unencrypted.

Anyone on the same network (or with access to the data path) could use a packet-sniffing tool like Wireshark to intercept the traffic and see:

- Which resources are being requested (e.g., /, /favicon.ico)
- The exact contents of each page in plain text
- Potentially sensitive data such as cookies or form fields

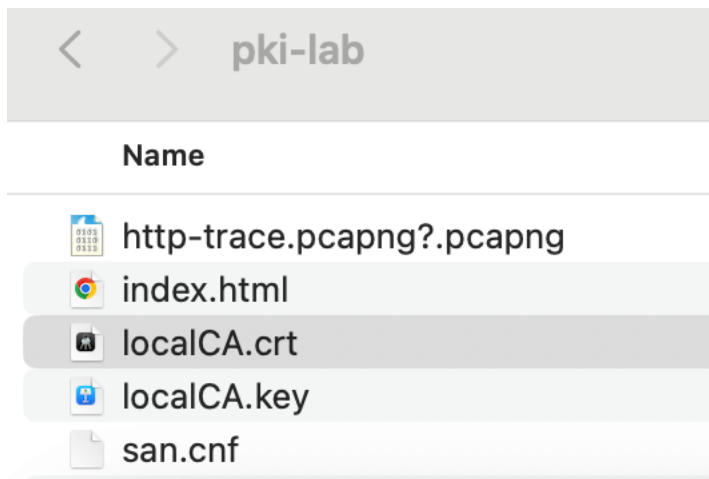
In other words, HTTP offers no confidentiality or protection from interception. The packets clearly show both the resource paths and the web page content in human-readable form. This makes it easy for an eavesdropper to understand exactly what is being communicated between the client and the server.

This part of the experiment successfully shows that HTTP traffic can be captured and read without any decryption. The next step will be to switch to HTTPS and observe how encryption protects the data from being read in Wireshark.

Task 3 - Create a self-signed certificate and upgrade your web server to HTTPS.

Why can't you obtain an SSL certificate for your local web server from a certificate authority? Public Certificate Authorities (CAs) can only issue SSL certificates for public domain names that can be verified through DNS or HTTP challenges. A local web server like localhost isn't publicly accessible or associated with a real domain, so ownership can't be verified by a CA. Because of that, it's impossible to get a trusted public certificate for localhost. The only way to secure a local environment is by creating a self-signed certificate or using a local CA that I manually trust on my system.

I created a small OpenSSL config with Subject Alternative Names (SANs) for localhost, 127.0.0.1, and ::1. Then I generated a local CA (root key + root cert) that I control. This CA will be used to sign my server certificate.



“Creating OpenSSL Config” code block:

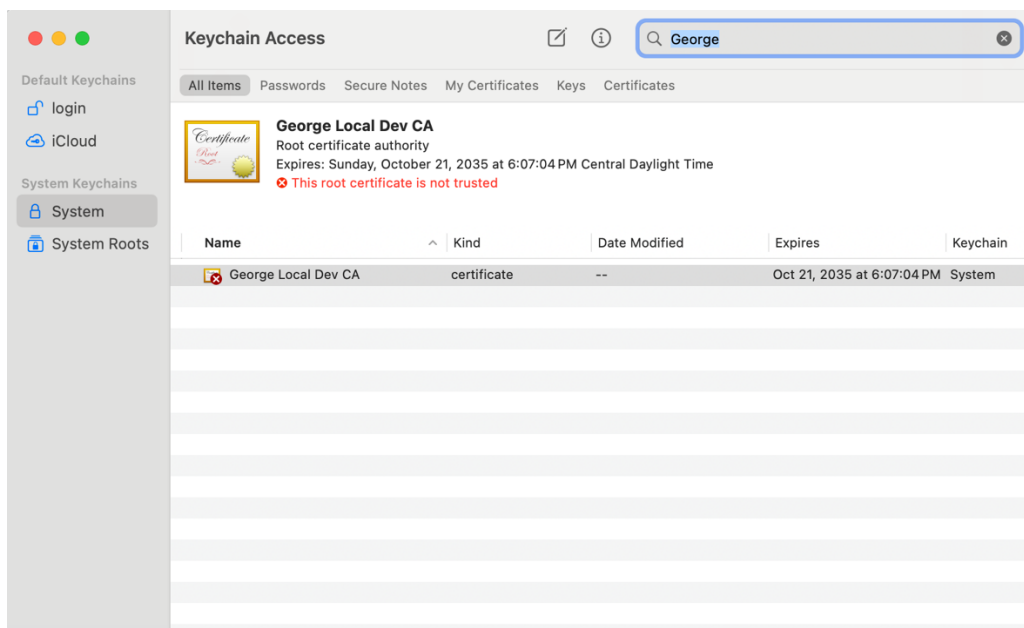
```
cat > san.cnf <<'EOF'
[ req ]
default_bits    = 2048
prompt         = no
default_md      = sha256
req_extensions  = req_ext
distinguished_name = dn
[ dn ]
C = US
ST = Illinois
```

```

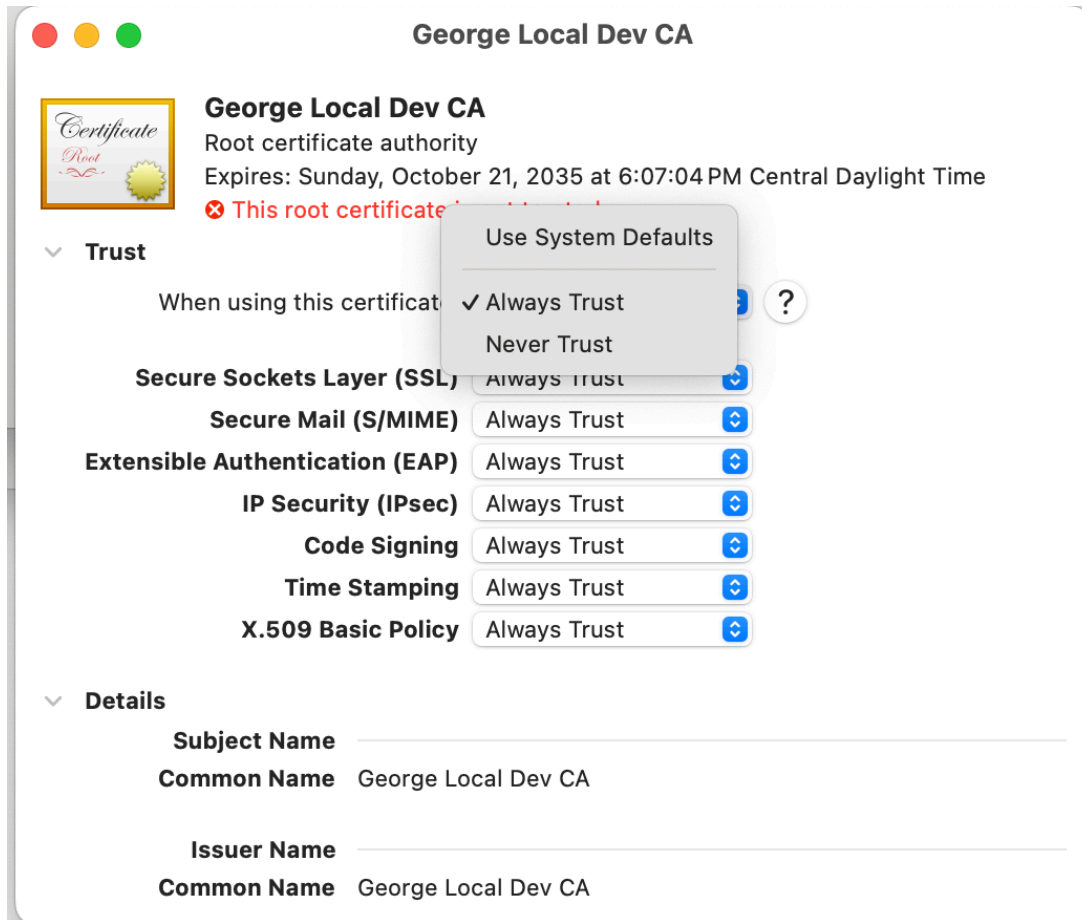
L = Chicago
O = Local Dev
CN = localhost
[ req_ext ]
subjectAltName = @alt_names
[ alt_names ]
DNS.1 = localhost
IP.1 = 127.0.0.1
IP.2 = ::1
[ v3_ext ]
authorityKeyIdentifier=keyid,issuer
basicConstraints=CA:FALSE
keyUsage = digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names
EOF
Creating local CA
openssl genrsa -out localCA.key 4096
openssl req -x509 -new -nodes -key localCA.key -sha256 -days 3650 \
-subj "/CN=George Local Dev CA" -out localCA.crt

```

Adding the CA certificate to the System keychain (so all users trust it).



Explicitly trusting the local CA so browsers accept leaf certs it signs. Trusting the CA in the System keychain makes the trust apply at the OS level. Browsers inherit this, so leaf certificates signed by this CA will be accepted without warning.



Using the trusted CA, I generated a server key and a CSR for localhost, then signed it to produce server.crt. I restarted the site using a small Python HTTPS server that loads server.crt and server.key.

```
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr -config san.cnf
openssl x509 -req -in server.csr -CA localCA.crt -CAkey localCA.key \
  -CAcreateserial -out server.crt -days 825 -sha256 \
  -extfile san.cnf -extensions v3_ext
```

```
Script:
cat > https_server.py <<'PY'
import http.server, ssl, socketserver
PORT = 8443
Handler = http.server.SimpleHTTPRequestHandler
```

with `socketserver.TCPServer(("0.0.0.0", PORT), Handler)` as `httpd`:

```
ctx = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
ctx.load_cert_chain(certfile="server.crt", keyfile="server.key")
httpd.socket = ctx.wrap_socket(httpd.socket, server_side=True)
print(f"Serving HTTPS on https://localhost:{PORT}")
httpd.serve_forever()
```

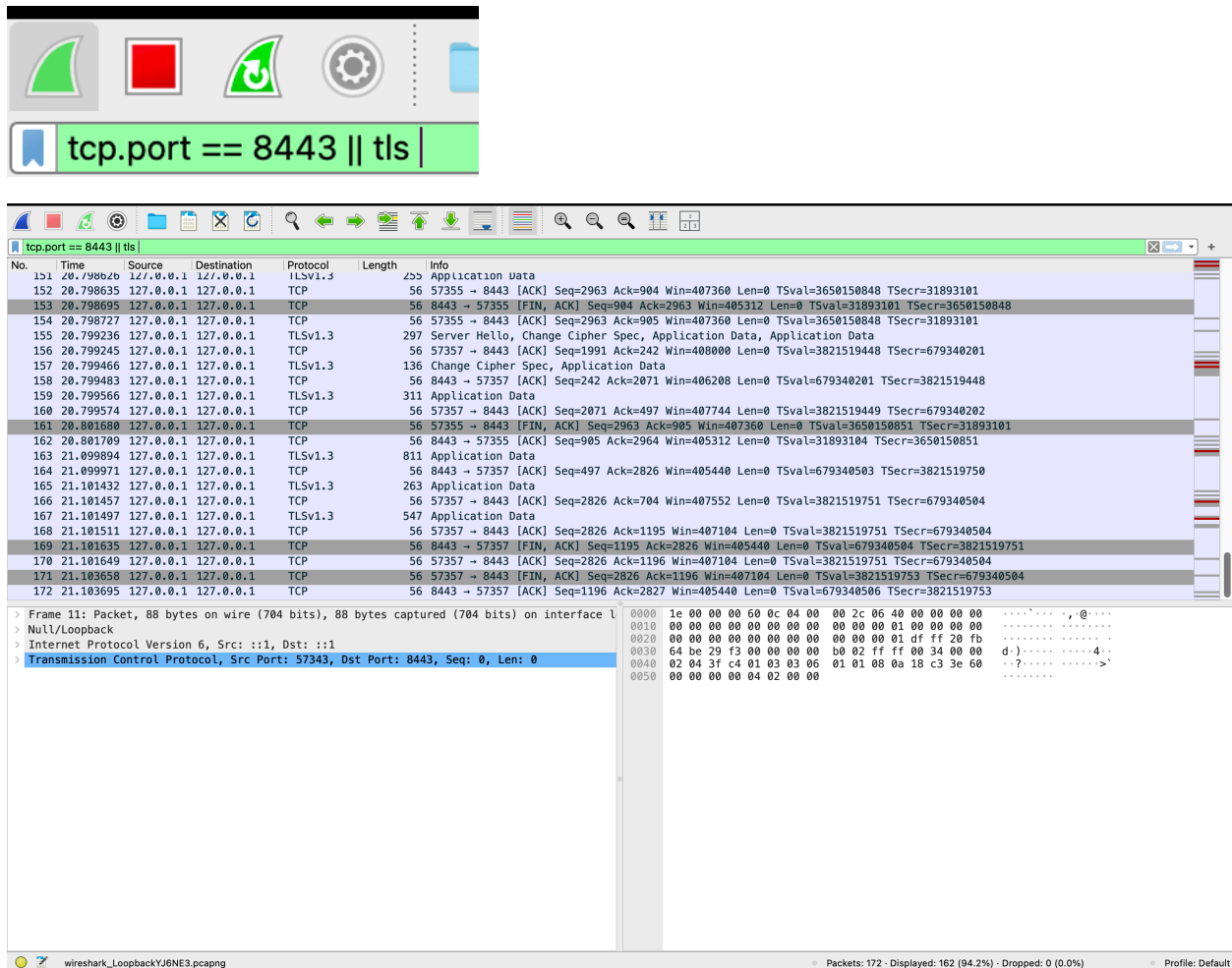
PY

`python3 https_server.py`

After starting the server, I visited <https://localhost:8443> to confirm the page loads over TLS. To verify encryption on the wire, I captured traffic on the loopback (lo0) interface while loading `https://localhost:8443`. I filtered with

`tcp.port == 8443 || tls`.

The trace shows the TLS handshake (ClientHello, ServerHello, Certificate) followed by TLS Application Data packets.



The image shows a Wireshark packet capture of a TLS handshake and application data over localhost:8443. The filter is `tcp.port == 8443 || tls`. The packet list shows the following details:

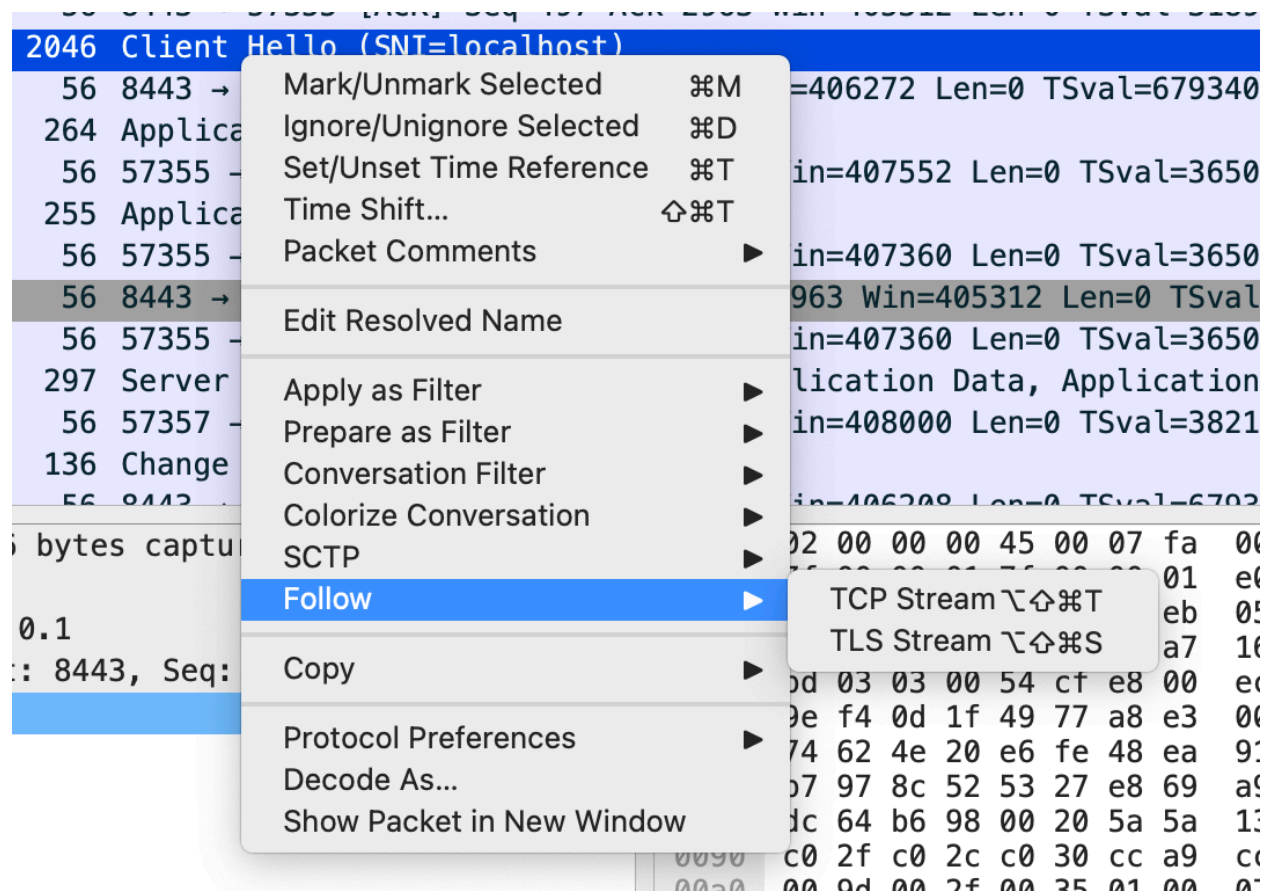
No.	Time	Source	Destination	Protocol	Length	Info
151	20.798626	127.0.0.1	127.0.0.1	TCP	56	57355 → 8443 [ACK] Seq=2963 Ack=904 Win=407360 Len=0 TSval=3650150848 TSecr=31893101
152	20.798635	127.0.0.1	127.0.0.1	TCP	56	8443 → 57355 [FIN, ACK] Seq=904 Ack=2963 Win=405312 Len=0 TSval=31893101 TSecr=3650150848
153	20.798695	127.0.0.1	127.0.0.1	TCP	56	57355 → 8443 [ACK] Seq=2963 Ack=905 Win=407360 Len=0 TSval=3650150848 TSecr=31893101
154	20.798727	127.0.0.1	127.0.0.1	TCP	56	57355 → 8443 [ACK] Seq=2963 Ack=905 Win=407360 Len=0 TSval=3650150848 TSecr=31893101
155	20.799236	127.0.0.1	127.0.0.1	TLSv1.3	297	Server Hello, Change Cipher Spec, Application Data, Application Data
156	20.799245	127.0.0.1	127.0.0.1	TCP	56	57357 → 8443 [ACK] Seq=1991 Ack=242 Win=408000 Len=0 TSval=3821519448 TSecr=679340201
157	20.799466	127.0.0.1	127.0.0.1	TLSv1.3	136	Change Cipher Spec, Application Data
158	20.799483	127.0.0.1	127.0.0.1	TCP	56	8443 → 57357 [ACK] Seq=242 Ack=2071 Win=406208 Len=0 TSval=679340201 TSecr=3821519448
159	20.799566	127.0.0.1	127.0.0.1	TLSv1.3	311	Application Data
160	20.799574	127.0.0.1	127.0.0.1	TCP	56	57357 → 8443 [ACK] Seq=2071 Ack=497 Win=407744 Len=0 TSval=3821519449 TSecr=679340202
161	20.801680	127.0.0.1	127.0.0.1	TCP	56	57355 → 8443 [FIN, ACK] Seq=2963 Ack=905 Win=407360 Len=0 TSval=3650150851 TSecr=31893101
162	20.801709	127.0.0.1	127.0.0.1	TCP	56	8443 → 57355 [ACK] Seq=905 Ack=2964 Win=405312 Len=0 TSval=31893104 TSecr=3650150851
163	21.099894	127.0.0.1	127.0.0.1	TLSv1.3	811	Application Data
164	21.099971	127.0.0.1	127.0.0.1	TCP	56	8443 → 57357 [ACK] Seq=497 Ack=2826 Win=405440 Len=0 TSval=679340503 TSecr=3821519750
165	21.101432	127.0.0.1	127.0.0.1	TLSv1.3	263	Application Data
166	21.101457	127.0.0.1	127.0.0.1	TCP	56	57357 → 8443 [ACK] Seq=2826 Ack=704 Win=407552 Len=0 TSval=3821519751 TSecr=679340504
167	21.101497	127.0.0.1	127.0.0.1	TCP	547	Application Data
168	21.101511	127.0.0.1	127.0.0.1	TCP	56	57357 → 8443 [ACK] Seq=2826 Ack=1195 Win=407104 Len=0 TSval=3821519751 TSecr=679340504
169	21.101635	127.0.0.1	127.0.0.1	TCP	56	8443 → 57357 [FIN, ACK] Seq=1195 Ack=2826 Win=405440 Len=0 TSval=679340504 TSecr=3821519751
170	21.101649	127.0.0.1	127.0.0.1	TCP	56	57357 → 8443 [ACK] Seq=2826 Ack=1196 Win=407104 Len=0 TSval=3821519751 TSecr=679340504
171	21.103658	127.0.0.1	127.0.0.1	TCP	56	57357 → 8443 [FIN, ACK] Seq=2826 Ack=1196 Win=407104 Len=0 TSval=3821519753 TSecr=679340504
172	21.103695	127.0.0.1	127.0.0.1	TCP	56	8443 → 57357 [ACK] Seq=1196 Ack=2827 Win=405440 Len=0 TSval=679340506 TSecr=3821519753

The packet details pane shows the following information for the selected packet (Frame 11):

- Packet: 88 bytes on wire (704 bits), 88 bytes captured (704 bits) on interface lo0
- Null/Loopback
- Internet Protocol Version 6, Src: ::1, Dst: ::1
- Transmission Control Protocol, Src Port: 57343, Dst Port: 8443, Seq: 0, Len: 0

The packet bytes pane shows the raw data of the packet, which is a TLS handshake packet.

For contrast with HTTP, I also followed a TLS stream. Unlike HTTP, the stream contents are unintelligible binary, which indicates successful encryption.



Following a TLS stream shows encrypted bytes instead of plaintext HTML.

So... Why unencrypted HTTP is not secure and how HTTPS addresses some of the vulnerabilities of HTTP:

HTTP is insecure because all data is sent in plain text, meaning anyone on the same network can intercept and read requests, responses, and even sensitive information like cookies or login details. HTTPS solves this by using TLS encryption, which secures the connection between client and server: preventing eavesdropping, tampering, and impersonation. In short, HTTPS provides confidentiality, integrity, and authentication, making communication private and trustworthy.