

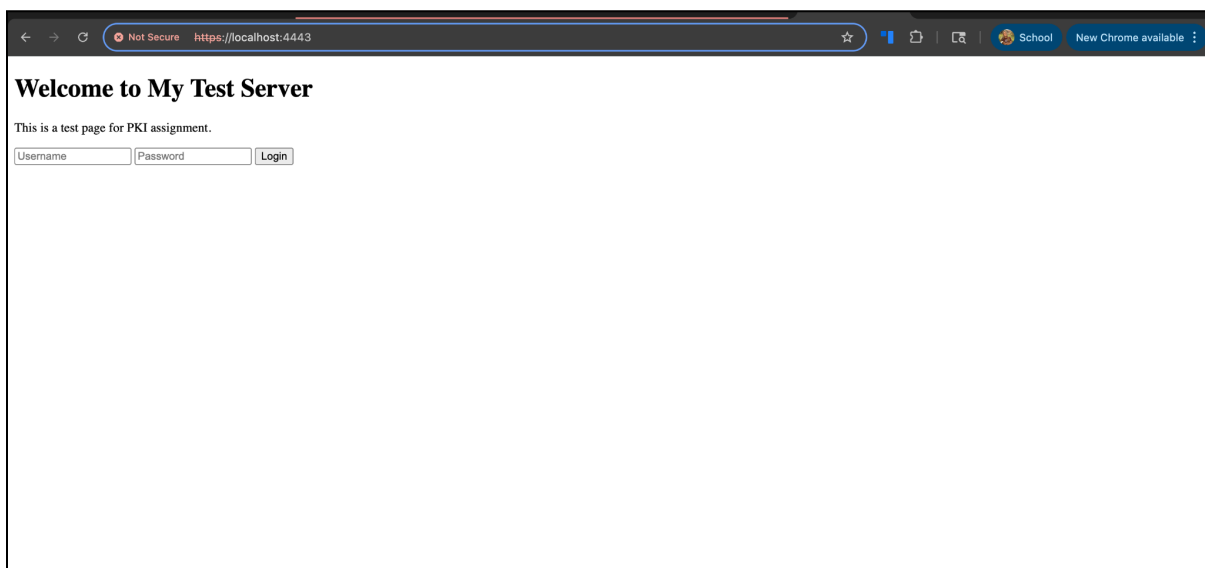
PKI Lab Assignment Write Up

Security, Privacy, and Consumer Protection

Grace Harrell

Task 1: Hosting a Local Web Server

Using ClaudeAI as a guide, I used Python's HTTP server module to host a local web server. I first created a directory with an index.html file containing a test login form that Claude created. I then started the server from the terminal at: <http://localhost:8000> for HTTP and <https://localhost:4443> for HTTPS (Screenshot below).



Task 2: Why HTTP is Not Secure

HTTP is insecure because all data is transmitted in plaintext without any encryption. This creates critical vulnerabilities with confidentiality, authentication, and integrity. That is, all communication is readable in the packet traces, the server identities are not verified, and data can be sneakily modified without detection.

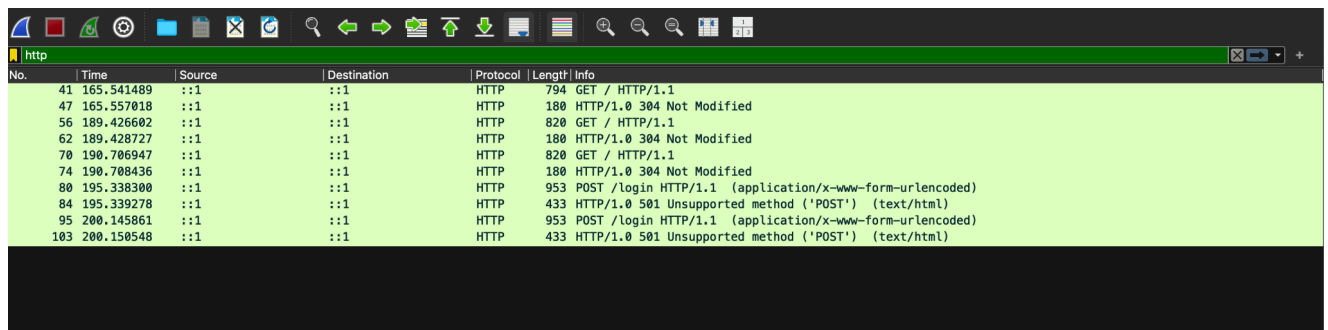
An eavesdropper can "sniff" HTTP traffic through several methods after gaining network access, which can be acquired by being on the same WiFi network as the victim, compromising routers, or gaining access to network infrastructure. After gaining network access, the eavesdropper can use packet capture tools like we did with Wireshark to capture network packets as they travel across the network. The adversary can then piece together the TCP streams to reveal full conversations between client and server. We also discussed man-in-the-middle attacks, where adversaries position themselves between the client and server to actively intercept traffic without detection. In these types of attacks,

the attacker acts as a relay. They receive requests from the client, forward them to the server, and then send the responses back to the client, all while reading and potentially modifying the data in transit. Because HTTP provides no mechanism to verify the identity of either party or detect tampering, both the client and server remain unaware that their communication is being intercepted and potentially altered.

The critical vulnerability of HTTP is that the attacker doesn't need to decrypt anything because HTTP transmits everything in plaintext, making all data immediately readable upon capture.

Wireshark Capture Analysis:

To demonstrate the vulnerability of no encryption, I captured HTTP traffic using Wireshark on the Loopback interface and filtered for the http protocol. I then visited `http://localhost:8000` from my browser and submitted a login form with username 'gkharrell' and password 'gkharrell.'

A screenshot of the Wireshark network protocol analyzer. The top toolbar shows various icons for file operations, editing, and analysis. Below the toolbar, a green filter bar displays 'http'. The main packet list pane shows a table of captured packets. The table has columns for 'No.', 'Time', 'Source', 'Destination', 'Protocol', 'Length', and 'Info'. The packets listed are: #41 (GET / HTTP/1.1), #47 (HTTP/1.0 304 Not Modified), #56 (GET / HTTP/1.1), #62 (HTTP/1.0 304 Not Modified), #70 (GET / HTTP/1.1), #74 (HTTP/1.0 304 Not Modified), #80 (POST /login HTTP/1.1), #84 (HTTP/1.0 501 Unsupported method ('POST')), #95 (POST /login HTTP/1.1), and #103 (HTTP/1.0 501 Unsupported method ('POST')). The source and destination for all packets are ::1. The packet details pane is currently empty.

No.	Time	Source	Destination	Protocol	Length	Info
41	165.541489	::1	::1	HTTP	794	GET / HTTP/1.1
47	165.557018	::1	::1	HTTP	180	HTTP/1.0 304 Not Modified
56	189.426602	::1	::1	HTTP	820	GET / HTTP/1.1
62	189.428727	::1	::1	HTTP	180	HTTP/1.0 304 Not Modified
70	190.706947	::1	::1	HTTP	820	GET / HTTP/1.1
74	190.708436	::1	::1	HTTP	180	HTTP/1.0 304 Not Modified
80	195.338300	::1	::1	HTTP	953	POST /login HTTP/1.1 (application/x-www-form-urlencoded)
84	195.339278	::1	::1	HTTP	433	HTTP/1.0 501 Unsupported method ('POST') (text/html)
95	200.145861	::1	::1	HTTP	953	POST /login HTTP/1.1 (application/x-www-form-urlencoded)
103	200.150548	::1	::1	HTTP	433	HTTP/1.0 501 Unsupported method ('POST') (text/html)

As shown in the screenshot above, the packet list clearly displays HTTP requests including GET requests for fetching the webpage (packets #41, 56, 70) and POST requests for form submissions (packets #80, 95). An eavesdropper can immediately identify which resources are being requested. Then, using Wireshark's "Follow TCP Stream" feature on the POST request, the complete conversation becomes visible in plaintext.

The screenshot below reveals the severity of HTTP's lack of encryption. This demonstrates that an attacker can see not only which resources are being fetched (the requested URLs and paths), but also the complete contents of those resources, like the full HTML responses, form data, and any sensitive information transmitted. Additionally, the screenshot also shows the server's HTML response in plaintext (in blue), including the complete HTML document structure. Every detail of the communication is completely exposed to anyone capturing the network traffic.

```
POST /login HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Content-Length: 207
Cache-Control: max-age=0
sec-ch-ua: "Not;A Brand";v="99", "Google Chrome";v="139", "Chromium";v="139"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "macOS"
Origin: http://localhost:8080
Content-Type: application/x-www-form-urlencoded
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/139.0.8.0 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=0.3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8080/
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en-US,en;q=0.5

username=gkharrell&password=gkharrell
HTTP/1.0 501 Unsupported method ('POST')
Server: SimpleHTTP/0.6 Python/3.12.2
Date: Wed, 22 Oct 2025 21:57:41 GMT
Connection: close
Content-Type: text/html; charset=utf-8
Content-Length: 207

<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error response</title>
</head>
<body>
<div>Error response</div>
<p>Error code: 501</p>
<p>Message: Unsupported method ('POST').</p>
<p>Error code explanation: 501 - Server does not support this operation.</p>
</body>
</html>
```

← Username and Password

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

Certificate Authorities will not issue SSL/TLS certificates for localhost for several critical reasons. CAs must verify that the applicant owns or controls a domain name through methods like HTTP-based challenges, DNS-based challenges, or email verification to domain administrators. However, "localhost" is a standard hostname used by every computer in the world, meaning there is no unique ownership to verify. CAs also require that domains be publicly accessible from the internet for verification purposes, but local servers on localhost or 127.0.0.1 are only accessible from the local machine and cannot be reached by CA verification servers. Additionally, if a CA issued a certificate for "localhost," it would be valid on every computer globally, creating a massive security vulnerability that could be exploited for phishing attacks or man-in-the-middle attacks on any system. SSL/TLS certificates exist to establish identity and trust, but "localhost" doesn't represent a unique entity, so there's no specific identity for the CA to authenticate. For local development and testing, self-signed certificates are the only option, though they provide encryption but lack the authentication and trust benefits of CA-signed certificates.

Generating SSL Certificate and Configuring HTTPS Server

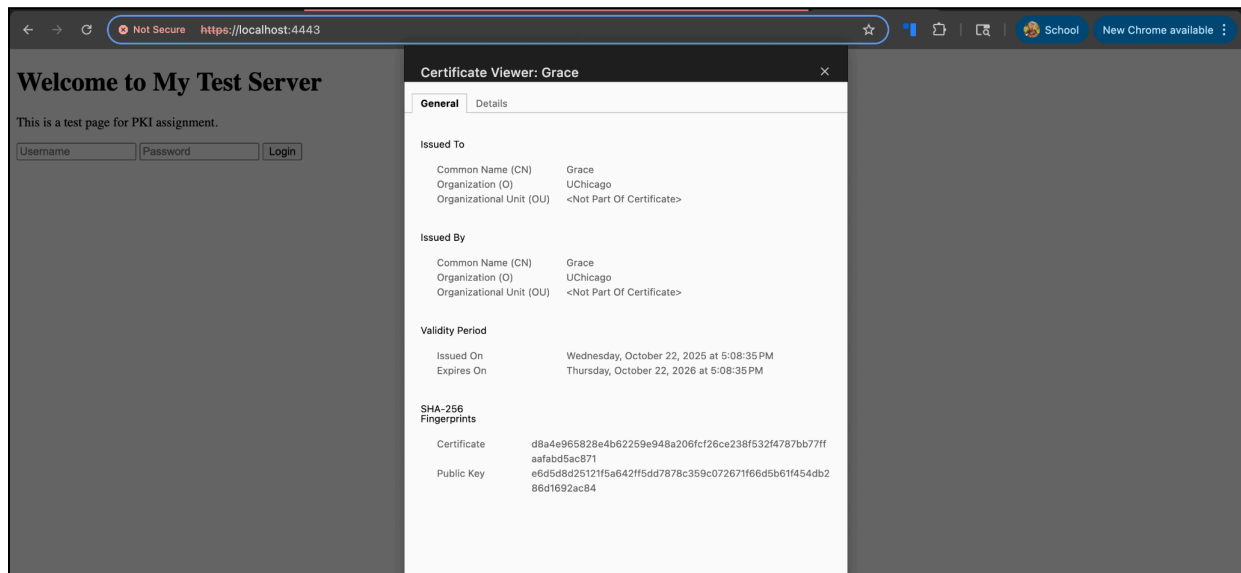
I generated a self-signed certificate using OpenSSL with the following command from ClaudeAI:

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

This created an RSA private key (key.pem) and a self-signed certificate (cert.pem). During generation, I set the Common Name (CN) to localhost, which is critical for the browser to properly recognize the certificate. I then created an HTTPS server using Python's http.server module with

SSL/TLS support. The server uses `ssl.SSLContext` to load the certificate and key, wraps the HTTP socket with SSL, and runs on port 4443. I started the server with `python3 https_server.py`, making it accessible at <https://localhost:4443>.

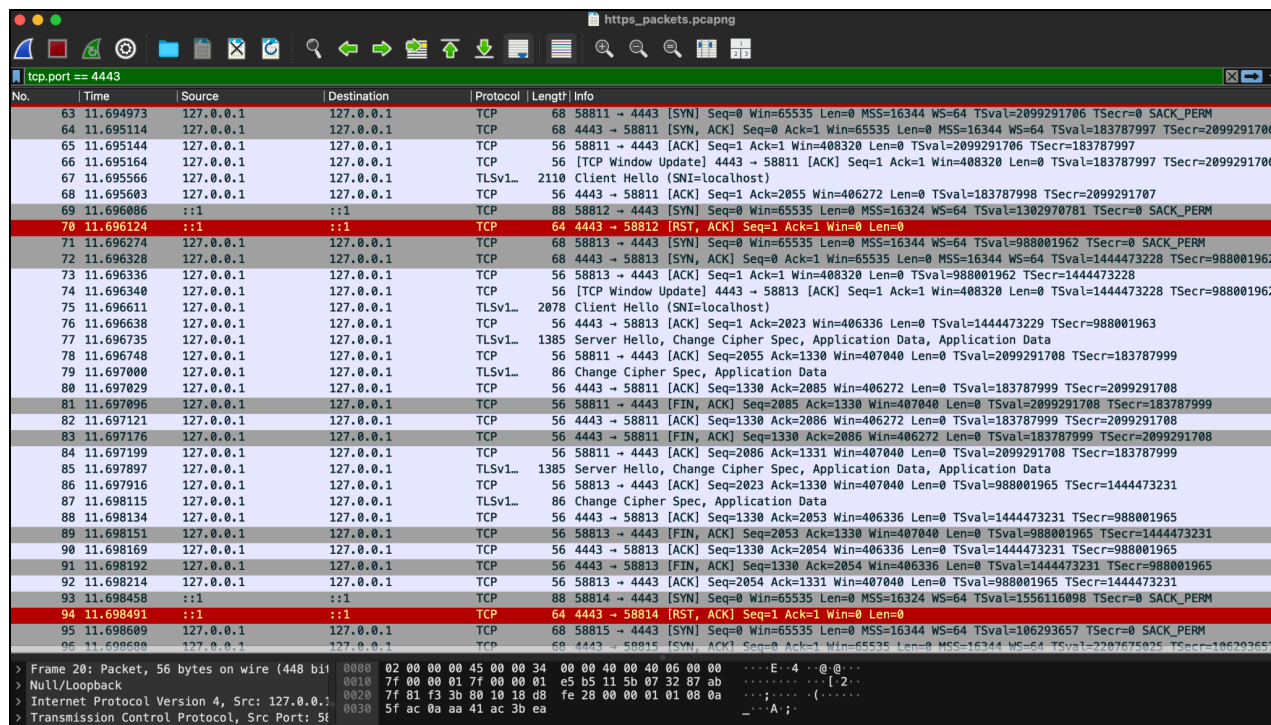
When accessing the HTTPS server, the browser displayed a security warning ("Your connection is not private") because the certificate is self-signed and not from a trusted CA. The browser cannot verify authenticity through a chain of trust. However, it's important to note that despite the warning, the connection is encrypted. The warning concerns trust and authentication, not encryption itself.



The screenshot above displays the details of the self-signed certificate I generated for the HTTPS server. The certificate shows that both the "Issued By" and "Issued To" fields contain the same information (Grace, UChicago), which is characteristic of self-signed certificates because I signed it rather than having it signed by a trusted Certificate Authority. This certificate enables encrypted communication between the browser and server, but because it's self-signed and not issued by a trusted CA in the browser's root certificate store, the browser still displays security warnings about the connection not being trusted.

To capture the HTTPS traffic, I started Wireshark again and applied the filter: `'tcp.port == 4443'`. I then visited <https://localhost:4443> multiple times and attempted to use "Follow TCP Stream"

to view the communication.

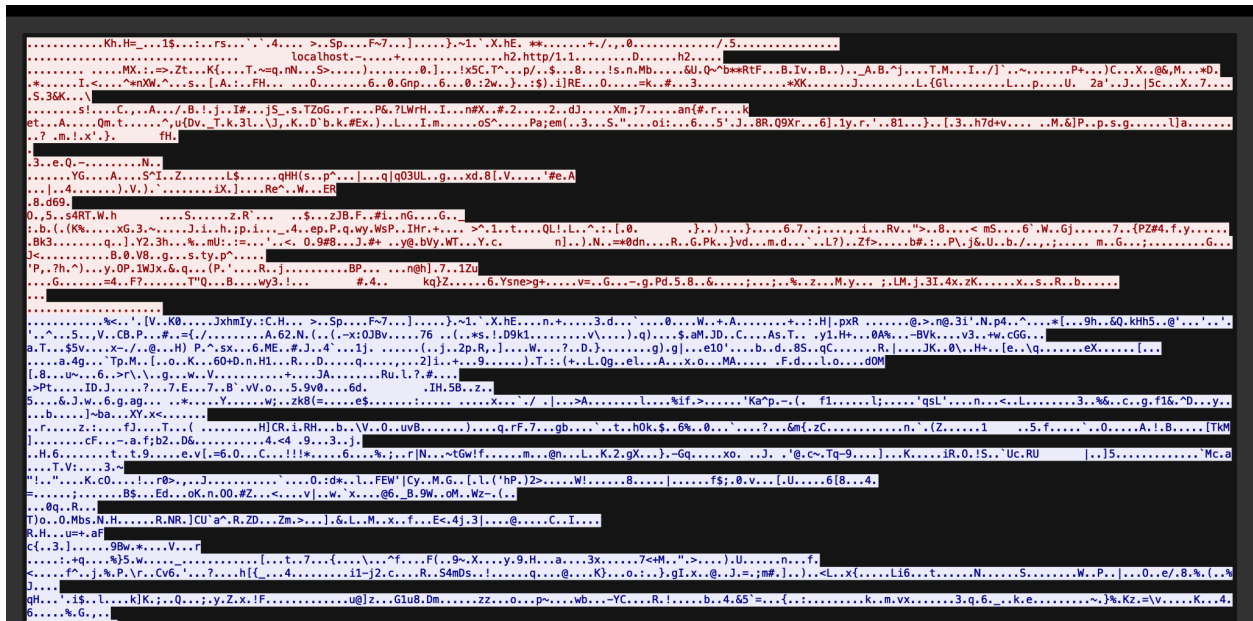


No.	Time	Source	Destination	Protocol	Length	Info
63	11.694973	127.0.0.1	127.0.0.1	TCP	68	58811 → 4443 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=2099291706 TSecr=0 SACK_PERM
64	11.695114	127.0.0.1	127.0.0.1	TCP	68	4443 → 58811 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=183787997 TSecr=2099291706
65	11.695144	127.0.0.1	127.0.0.1	TCP	56	58811 → 4443 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=2099291706 TSecr=183787997
66	11.695164	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 4443 → 58811 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=183787997 TSecr=2099291706
67	11.695566	127.0.0.1	127.0.0.1	TLSv1.3	2110	Client Hello (SNI=localhost)
68	11.695603	127.0.0.1	127.0.0.1	TCP	56	4443 → 58811 [ACK] Seq=1 Ack=2055 Win=406272 Len=0 TSval=183787998 TSecr=2099291707
69	11.696086	:::1	:::1	TCP	88	58812 → 4443 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=1302970781 TSecr=0 SACK_PERM
70	11.696124	:::1	:::1	TCP	64	4443 → 58812 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
71	11.696274	127.0.0.1	127.0.0.1	TCP	68	58813 → 4443 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=988001962 TSecr=0 SACK_PERM
72	11.696328	127.0.0.1	127.0.0.1	TCP	68	4443 → 58813 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=1444473228 TSecr=988001962
73	11.696336	127.0.0.1	127.0.0.1	TCP	56	58813 → 4443 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=988001962 TSecr=1444473228
74	11.696340	127.0.0.1	127.0.0.1	TCP	56	[TCP Window Update] 4443 → 58813 [ACK] Seq=1 Ack=1 Win=408320 Len=0 TSval=1444473228 TSecr=988001963
75	11.696611	127.0.0.1	127.0.0.1	TLSv1.3	2878	Client Hello (SNI=localhost)
76	11.696638	127.0.0.1	127.0.0.1	TCP	56	4443 → 58813 [ACK] Seq=1 Ack=2023 Win=406336 Len=0 TSval=1444473229 TSecr=988001963
77	11.696735	127.0.0.1	127.0.0.1	TLSv1.3	1385	Server Hello, Change Cipher Spec, Application Data, Application Data
78	11.696748	127.0.0.1	127.0.0.1	TCP	56	58811 → 4443 [ACK] Seq=2055 Ack=1330 Win=407040 Len=0 TSval=2099291708 TSecr=183787999
79	11.697000	127.0.0.1	127.0.0.1	TLSv1.3	86	Change Cipher Spec, Application Data
80	11.697029	127.0.0.1	127.0.0.1	TCP	56	4443 → 58811 [ACK] Seq=1330 Ack=2085 Win=406272 Len=0 TSval=183787999 TSecr=2099291708
81	11.697096	127.0.0.1	127.0.0.1	TCP	56	58811 → 4443 [FIN, ACK] Seq=2085 Ack=1330 Win=407040 Len=0 TSval=2099291708 TSecr=183787999
82	11.697121	127.0.0.1	127.0.0.1	TCP	56	4443 → 58811 [ACK] Seq=1330 Ack=2086 Win=406272 Len=0 TSval=183787999 TSecr=2099291708
83	11.697176	127.0.0.1	127.0.0.1	TCP	56	4443 → 58811 [FIN, ACK] Seq=1330 Ack=2086 Win=406272 Len=0 TSval=183787999 TSecr=2099291708
84	11.697199	127.0.0.1	127.0.0.1	TCP	56	58811 → 4443 [ACK] Seq=2086 Ack=1331 Win=407040 Len=0 TSval=2099291708 TSecr=183787999
85	11.697897	127.0.0.1	127.0.0.1	TLSv1.3	1385	Server Hello, Change Cipher Spec, Application Data, Application Data
86	11.697916	127.0.0.1	127.0.0.1	TCP	56	58813 → 4443 [ACK] Seq=2023 Ack=1330 Win=407040 Len=0 TSval=988001965 TSecr=1444473231
87	11.698115	127.0.0.1	127.0.0.1	TLSv1.3	86	Change Cipher Spec, Application Data
88	11.698134	127.0.0.1	127.0.0.1	TCP	56	4443 → 58813 [ACK] Seq=1330 Ack=2053 Win=406336 Len=0 TSval=1444473231 TSecr=988001965
89	11.698151	127.0.0.1	127.0.0.1	TCP	56	58813 → 4443 [FIN, ACK] Seq=2053 Ack=1330 Win=407040 Len=0 TSval=988001965 TSecr=1444473231
90	11.698169	127.0.0.1	127.0.0.1	TCP	56	4443 → 58813 [ACK] Seq=1330 Ack=2054 Win=406336 Len=0 TSval=1444473231 TSecr=988001965
91	11.698192	127.0.0.1	127.0.0.1	TCP	56	4443 → 58813 [FIN, ACK] Seq=1330 Ack=2054 Win=406336 Len=0 TSval=1444473231 TSecr=988001965
92	11.698214	127.0.0.1	127.0.0.1	TCP	56	58813 → 4443 [ACK] Seq=2054 Ack=1331 Win=407040 Len=0 TSval=988001965 TSecr=1444473231
93	11.698458	:::1	:::1	TCP	88	58814 → 4443 [SYN] Seq=0 Win=65535 Len=0 MSS=16324 WS=64 TSval=1556116098 TSecr=0 SACK_PERM
94	11.698491	:::1	:::1	TCP	64	4443 → 58814 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
95	11.698609	127.0.0.1	127.0.0.1	TCP	68	58815 → 4443 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=106293657 TSecr=0 SACK_PERM
96	11.698608	127.0.0.1	127.0.0.1	TCP	68	4443 → 58815 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2207675025 TSecr=106293657

> Frame 20: Packet, 56 bytes on wire (448 bits) captured on eth0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 58811, Dst Port: 4443

The screenshot shows the HTTPS/TLS packet capture. Unlike the HTTP capture, the packets are labeled with TLS protocol information. The visible TLS handshake includes Client Hello packets (packet #9, where the client announces supported TLS versions and cipher suites) and Server Hello packets (packets #19, 27, where the server selects TLS version 1.3 and cipher suite). In TLSv1.3, the Certificate Exchange and Key Exchange are encrypted within the "Application Data" portions of the Server Hello packets, making them not separately visible in the packet list. After the handshake completes, all subsequent packets are labeled "Application Data" and contain encrypted content.

The screenshot below demonstrates the stark difference between HTTP and HTTPS. When using "Follow TCP Stream" on the HTTPS packets, instead of readable text, we see the encrypted gibberish above. The HTTP request, response headers, HTML content, and any form data that would have been plaintext in HTTP are now completely encrypted and unreadable. An eavesdropper with Wireshark can see that a connection is occurring and observe the TLS handshake, but they cannot read any of the actual application data.



Difference Between HTTP and HTTPS Traffic

With HTTP, the "Follow TCP Stream" feature revealed the complete plaintext conversation, including the POST request with the clear username and password, all HTTP headers, and the full HTML response. With HTTPS/TLS, the same feature shows only encrypted data that is completely unreadable. While an eavesdropper can see metadata with HTTPS, they cannot see the HTTP request methods, paths, headers, form data, passwords, response content, session tokens, or any application data. This demonstrates that HTTPS provides strong confidentiality through encryption, making it computationally infeasible for attackers to decrypt the traffic without the session keys. This is why HTTPS is essential for modern web security and why HTTP should never be used for transmitting sensitive information.