Thomas Szafir

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 (acknowledging as per Syllabus) 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.
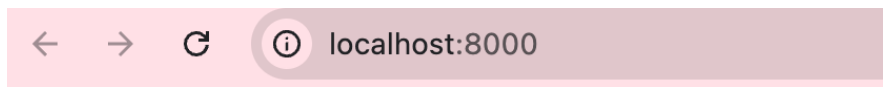
Hello, HTTP!

This is the insecure version.

2. Identify why HTTP is not secure.

<mark>How an eavesdropper can "sniff" web traffic between a client and HTTP server:</mark> 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.
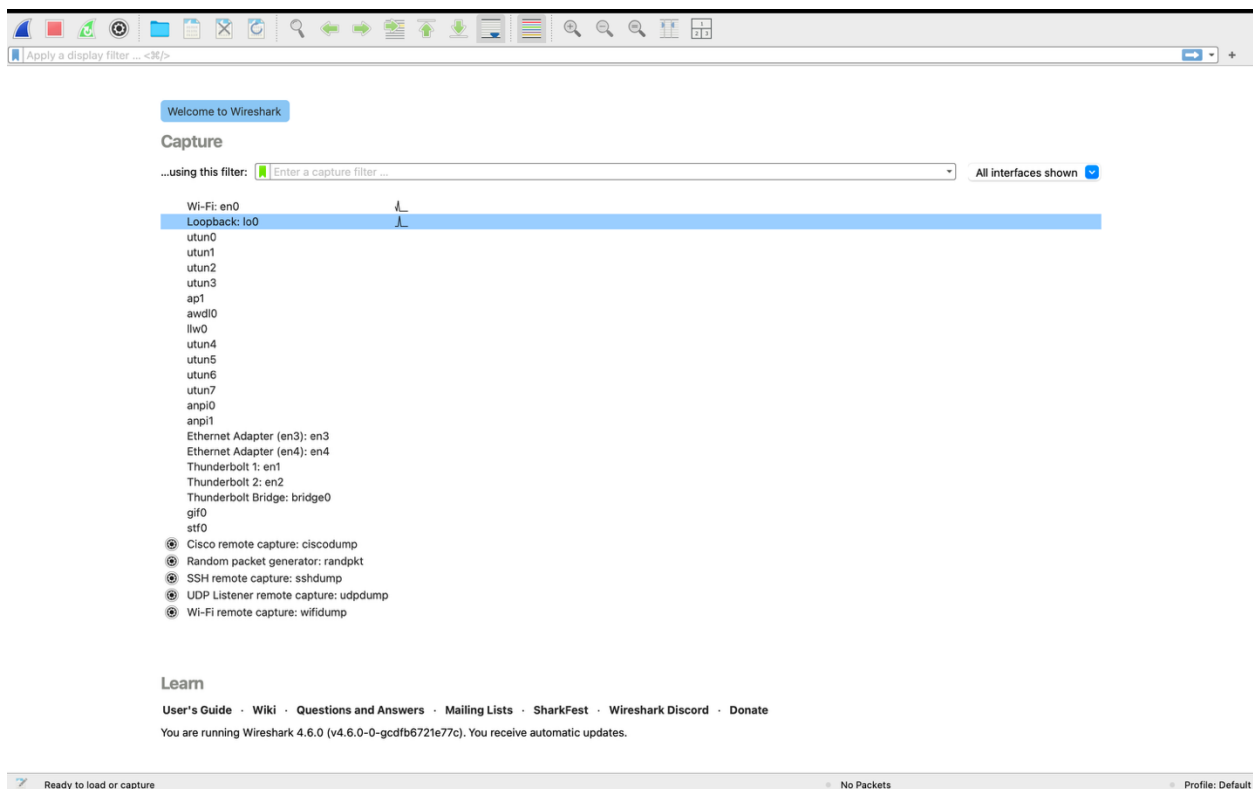
Next, I opened Wireshark and selected the Loopback (lo0) interface, which records traffic sent between programs running on my own machine (like my web browser and local web server).

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.

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

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.

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>
```

1 client pkt, 2 server pkts, 1 turn.

Entire conversation (1142 bytes) ⬦   Show as   ASCII ⬦   No delta times ⬦   Stream 0 ⬦

Find: [   ]   ☐ Case sensitive   Find Next

Help   Filter Out This Stream   Print   Save as...   Back   Close

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.

<mark>Why can't you obtain an SSL certificate for your local web server from a certificate authority?</mark> 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.

Generating an SSL certificate for my web server:

I created a small OpenSSL config with Subject Alternative Names 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.
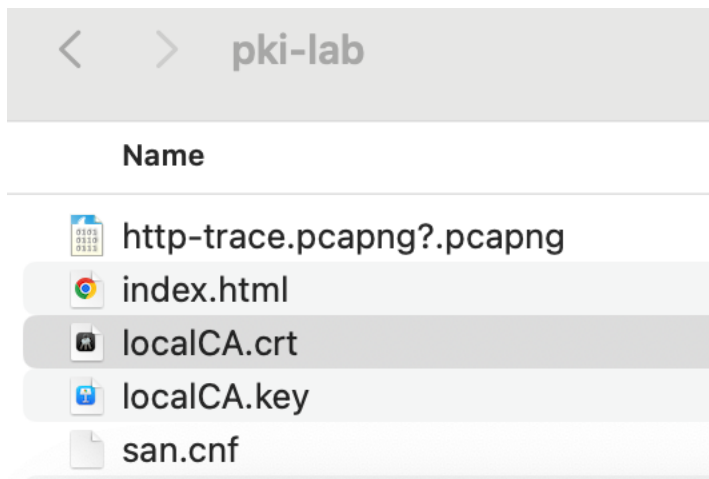
"Creating OpenSSL Config" code block:

```
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).
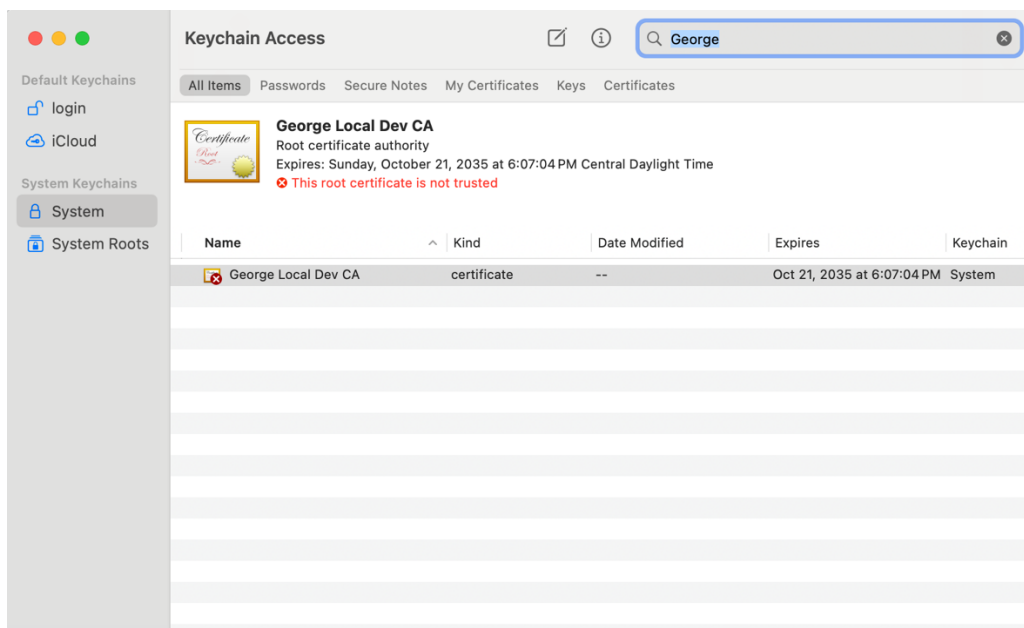
Explicitly trusting the local CA so browsers accept leaf certs it signs. Trusting the CA in the System keychain makes the trust apply at the OS level. Browsers pick this up automatically, so certificates signed by my local CA load without any security warnings.
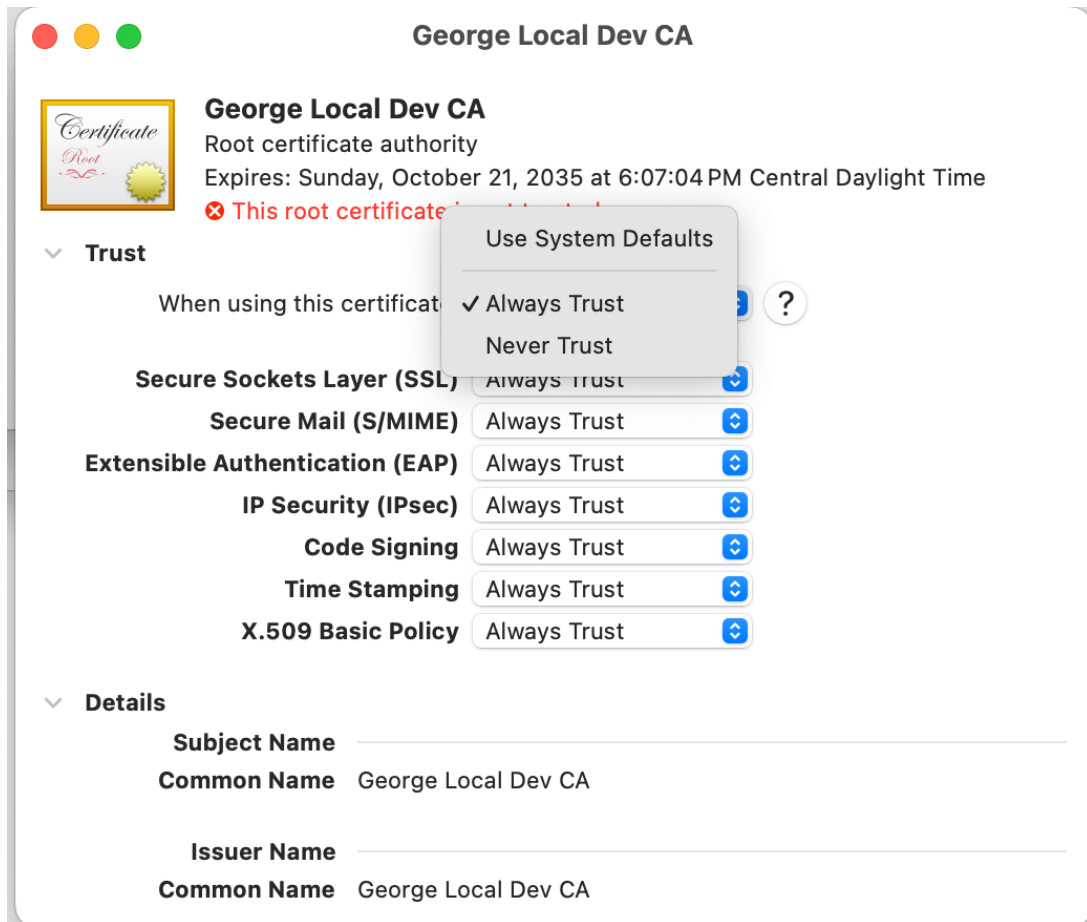
Using the trusted CA, I generated a server key and a CSR for localhost, then signed it to produce server.crt. I restarted the site using a small Python HTTPS server that loads server.crt and server.key.

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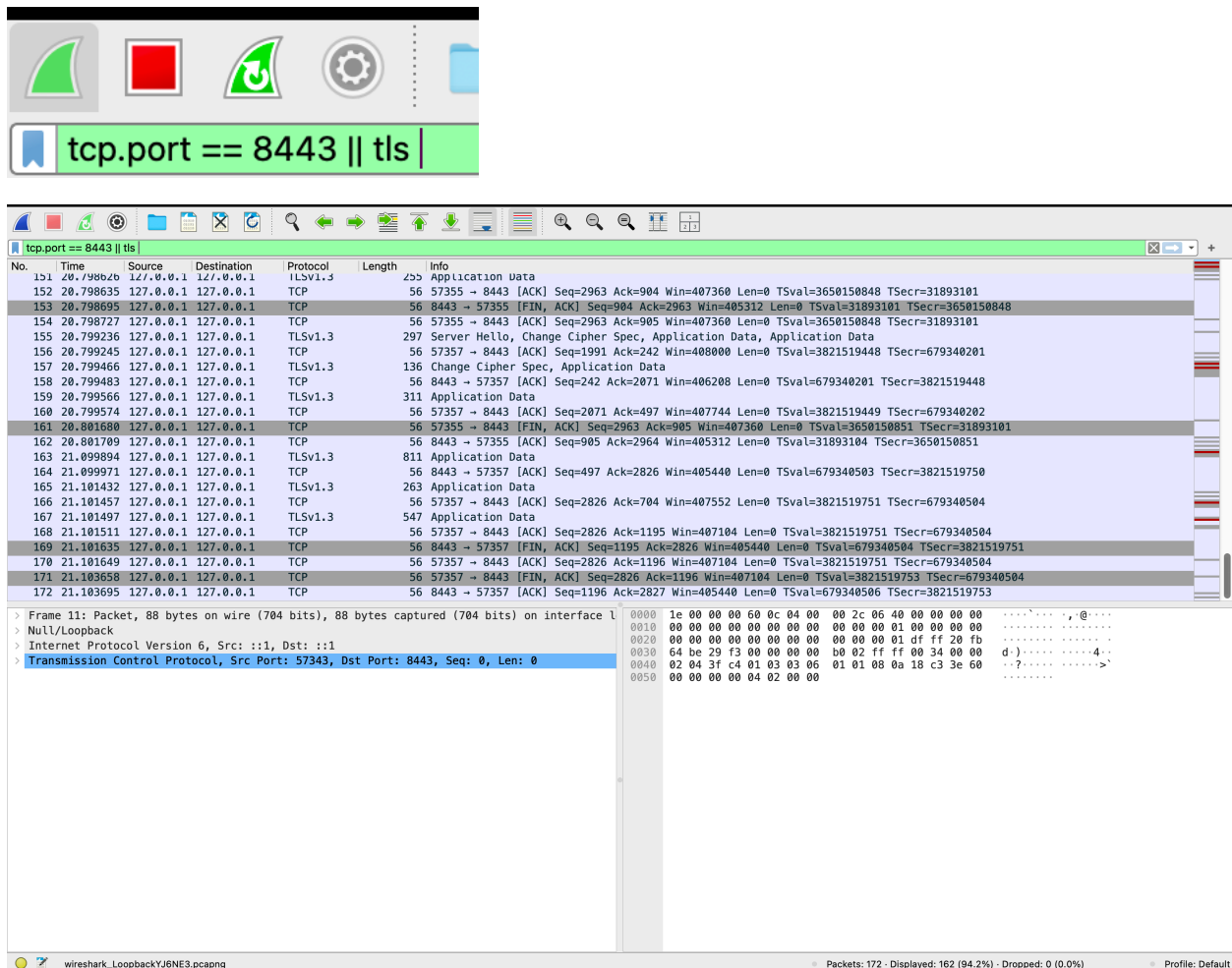
```
ctx = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
ctx.load_cert_chain(certfile="server.crt", keyfile="server.key")
httpd.socket = ctx.wrap_socket(httpd.socket, server_side=True)
print(f"Serving HTTPS on https://localhost:{PORT}")
httpd.serve_forever()
```

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.

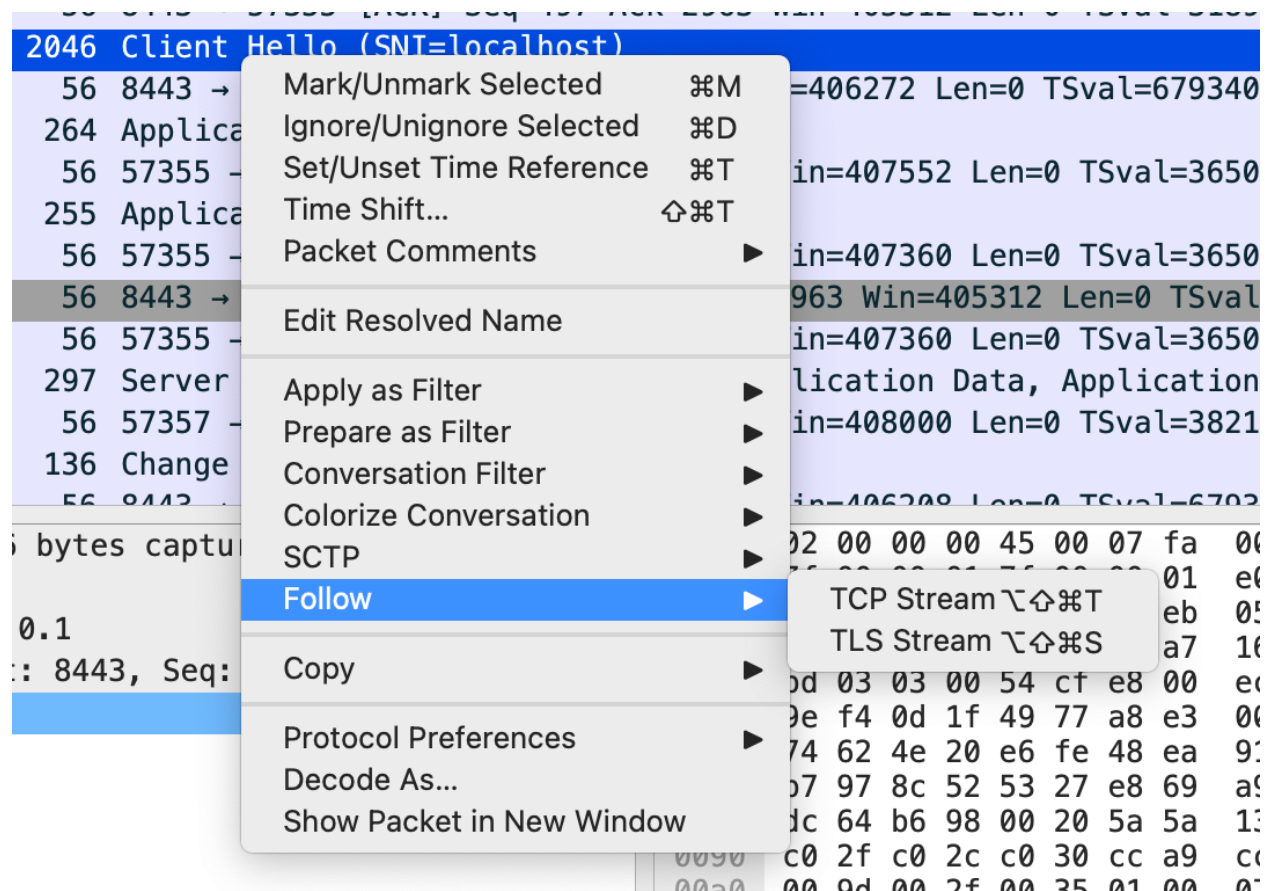The trace shows the TLS handshake (ClientHello, ServerHello, Certificate) followed by TLS Application Data packets.

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.

```
...........T........h.p..
.Iw...Lf..G.tbN ..H........X...RS'.i.Xm...lu.d... ZZ........+./.,.0............../.5...Tjj...
...
**.........
..................h2.http/1.1D.....h2.
......... .2...{S.).q..W.ze.P..f..v.\Z]SG.....K...F..;.H.d..F....i....#)...x#...T..D'..il......kd.|D.V.B.n.....Ef.I.r.J.^...i.....P..7..zNR.......z.-#.$..q.5....K...*.9.....gFA...
........................ localhost.+.........-...........3....**......mZC.x..q......)A.h.H-h#x.5.).....2.).o....$.lx.......QWx(......{.#X..?..`.1'.      ....`a.4....|IV..
.y.|...
.....AQ...&*..^..w.4Uh`.../..C....\..1.y....
.l*.k.......u......,.Q.(.IxC......`.(...j8.7.RT V
fa...    b...qe........qL<0^.......".*..^k ..w..b.\.V.l...(\=p..%<......I=.@...S*..'.:..A..J..fpIgT..\;..8.+5Ve#yyE...R...|..._..Z.0....$r*...#T&wN      `g?(.....l....C.b.Q....g..f
.........np...\.n..k5.........Z\.*u9.....k8D..Y...D\.:...s..+.,.{.....1....
.
q.G..A..~.5.r....sz`w..0.u.D.....(....49I.$i...b...P...n...t...x....(...\........9
#.!E..U..%d...[..m...@.T.}...X..c.}4......).Qe..|~(....eS  4.V...W...w...3.D./......@.'.^...%[N5A...A[.....2....."....E.r....A+.X......t.].-c.n.g..\.j.!e.5.^......j......y.
..L.S.1`.|@..   K ..~  .{.    ....3:.{.`y...R.;.lr..d.u.UX..P....z......6....f.st.^.6.d..z..'E..3.0....C.'#..e.v
..>.;.F..Q.Wh....aM..K.@|a..Ma.cEg..._.@:.~......bLi.;.m-..h@...7.......{.....i...T.Cz.g....Q.s.@.(kY..MY.....0.{..XdB.c..u._.<.S..P.U..T;..AI....v.....Q.}$Z.}.0..+....t.+K..y..
DS.J..f.%...Ibas..].2&.f.g2._T
.iX..-.vE.U..6..5..k>........c.s..x8cD'.6g.Y%fe'...6Vt..........   .......%y!.=H&R.Q.V%O.Ed.*.....<.m...(.$..i.....'.`.c.*.o..-f..gy......M-}f.8X).O\%..[.............o...j.
F...@... Y....R5+...#..G3.h..~;>i./.m.@.#........................)......?.....*f..O...8.Q.G.}..QZ...C.....a..T"........>..j.;...u.Vr.z{..Qw.C.9{
.....j
...n...-...R"..N.en#....3..f....2..(.....=>_u...6f;]//..~..+.80|%.`0.<.>81.p..=44..B..    ......q...s.q.r....S[..x..2..........Ng+.10.d.tt-.`..hC[-
.;W[ZP........f...........w..nq.?
.........|..)...^&.a.h..
o...5P.t..7..0.`4.Q ..H........X...RS'.i.Xm...lu.d.....4.+.....3.$... ...gN2......L.Vc..R......#%..bg>.)..................2..K.miv...Z8....<'....ESmf?.'....?.   h.C{...
.T....}..<.h..d".p.l..v....k.......H.(..'W..|:.;
...........E.]$.{..s.....o...2;.=&.].UB...7|D (J.j.7+.M`..s..>...f....,!.q".I.{.
.......Y.....'[....7.H4k..MKr.a}.j[s.&f....XP
.iQ.|....6.f..=^..Yp{..A..lv..Z'u)......VB......y.h)F.V.....s..5*z..@3E..Ru(%')P.2...Vo.$.....}.MV.........!.}..O.M..%.#.j.i..1..G+7..w..az.f..b4....x,$._.,.mT....2....[...=.....\..
n..q..|.]l.S[...z.3..e..;.
.....{....gZX."...XX.l.).....u;x..>L..y.v..'Qi....&3......%.....5....B..4h.\q....A.
u...Mp...Nc ..
M{.....7S..V....I..Z$i.h|sJ..~k.b)...~.e.........%.1..CH."...V.s{.(.Z..s.8.x67u.P..........E......m  C.M...-..A.@..R....o&.. ..~?..{V..._..P....P%.wW\.H<.g..F.M.7..q.<...:...g";eq.
.}.x<......7.;...l..a[.&.....Q.]........i.R.....s..'h..96H  u...K.`...dJl...6f...*.....s{..E.M$......?b=...      30g..0....`.'...3Ic.;..0.[..U.G..k......KF.6t
..?..B..V...].:a.J.0....F...uG.7.YFz.(8o.........^..0..5^ao.w..?6.m+..\......F.....4a...8..^..#..nP....K.........<...V...LTG....5..`\.;.I.|.......e^zk.EYR.{...................y"4..
:..T.B...F. +z].;.#..
..J....8.E..y...>...'....t.JV.+H..jt.`......?.\...xG]J.u.A
O.>...._..B..u..D....{..fn.Vy  ./..M....
..HK..|.....
.....<.$.xS...M.B..F...Hi^...`f4..........R...........ei.$......f..s......t....D.....Z....YrMd...e.y".o.r..k.[.\..-\I..K.b...W....L.r....K!...%+\...g...N...'.D.\|:.`.ed.#.j......
.8.
..!.......9........H.............:....I...
...c..v.p.
.......Y.....B.ZF..?.a..:..\.L...........>..S+.B.."....
```

3 client pkts, 4 server pkts, 5 turns.

| Entire conversation (4019 bytes) ⌄ | Show as | ASCII ⌄ | No delta times ⌄ | Stream 13 ⌄ |

Find: [   ] ☐ Case sensitive [Find Next]

[Help] [Filter Out This Stream] [Print] [Save as...] [Back]   [Close]

Difference between the contents of HTTP and HTTPS (TLS) traffic: The key difference between HTTP and HTTPS traffic is that HTTPS encrypts all communication using the TLS protocol. In the HTTP capture, the requests and responses are visible in plaintext, meaning anyone intercepting packets could see the full contents of the messages. In contrast, the HTTPS capture only shows the TLS handshake (Client Hello, Server Hello, Certificate, etc.) followed by "Application Data," which appears as random binary bytes. This indicates that the actual data is encrypted and unreadable to eavesdroppers. This encryption provides confidentiality, integrity, and authentication between the client and server.

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.