**Task 1: Host a local web server**

I set up a local web server using nginx running in a Docker container. First, I installed Docker Desktop on my laptop. Once Docker was running, I pulled the official nginx image from Docker Hub. I then started the nginx container and configured it to map port 80 on my local machine to port 80 inside the container, making the web server accessible through my browser.

To verify everything was working, I opened my web browser and navigated to localhost. The nginx default welcome page appeared, showing "Welcome to nginx!" with some additional information about successful installation. At this point, I had a fully functional HTTP web server running locally.

**Task 2: Identify why HTTP is not secure**

HTTP is fundamentally insecure because all communication between the client and server is transmitted in plaintext. There is no encryption protecting the data as it travels across the network. This means that anyone who can intercept the network traffic can read everything being communicated. An eavesdropper can "sniff" web traffic between a client and HTTP server through capturing packets being passed through network devices. Once the packets are captured, they can be easily reassembled and read because the data is not encrypted.

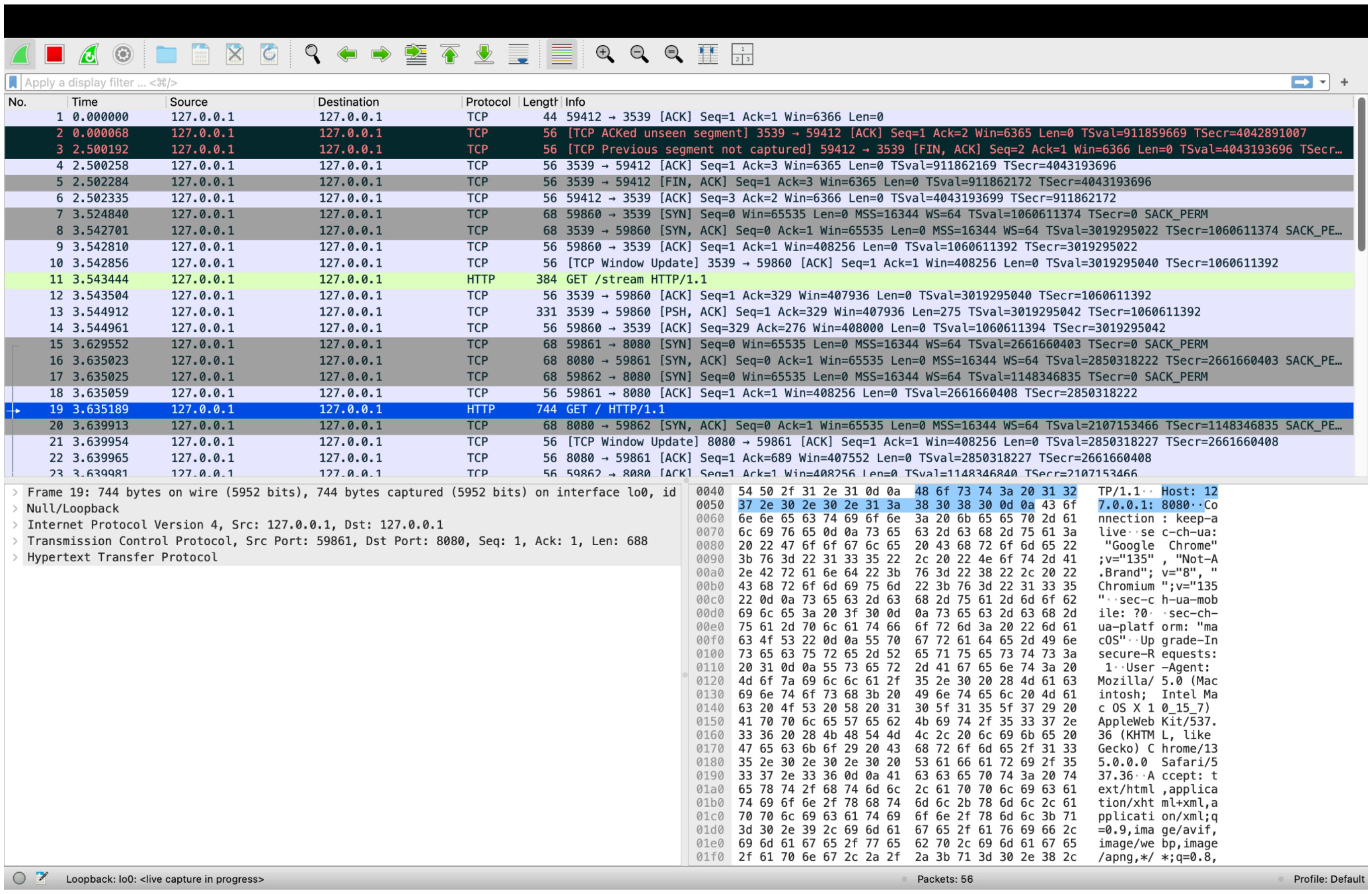
I then opened Wireshark on my laptop and started capturing traffic on my network interface. I configured it to monitor the localhost loopback interface since my web server and browser were running on the same machine. Then I opened my web browser and navigated to my local HTTP server at localhost.

After the page fully loaded, I stopped the Wireshark capture to analyze what had been recorded.

*What I Observed:* The Wireshark capture revealed everything about my HTTP communication in complete detail. I could see the TCP three-way handshake that established the connection between my browser and the nginx server. More importantly, I could see the actual HTTP traffic in plain, readable text. The HTTP GET request was completely visible. I could read the request line showing that my browser requested the root page, the Host header showing it was connecting to localhost, and the User-Agent string revealing details about my browser and operating system. Additional headers showed my browser's preferred languages and the types of content it could accept.

The server's response was equally transparent. I could see the HTTP response status showing 200 OK, the Server header revealing it was nginx with a specific version number, and headers describing the content type and length.

I took a screenshot showing the Wireshark interface with visible HTTP packets, the detailed view of individual packets showing headers and content, and the TCP stream view displaying the entire conversation in readable text format.



```
No.   Time      Source     Destination  Protocol Length Info
    1 0.000000  127.0.0.1  127.0.0.1    TCP       44 59412 → 3539 [ACK] Seq=1 Ack=1 Win=6366 Len=0
    2 0.000068  127.0.0.1  127.0.0.1    TCP       56 [TCP ACKed unseen segment] 3539 → 59412 [ACK] Seq=1 Ack=2 Win=6365 Len=0 TSval=911859669 TSecr=4042891007
    3 2.500192  127.0.0.1  127.0.0.1    TCP       56 [TCP Previous segment not captured] 59412 → 3539 [FIN, ACK] Seq=2 Ack=1 Win=6366 Len=0 TSval=4043193696 TSecr…
    4 2.500258  127.0.0.1  127.0.0.1    TCP       56 3539 → 59412 [ACK] Seq=1 Ack=3 Win=6365 Len=0 TSval=911862169 TSecr=4043193696
    5 2.502284  127.0.0.1  127.0.0.1    TCP       56 3539 → 59412 [FIN, ACK] Seq=1 Ack=3 Win=6365 Len=0 TSval=911862172 TSecr=4043193696
    6 2.502335  127.0.0.1  127.0.0.1    TCP       56 59412 → 3539 [ACK] Seq=3 Ack=2 Win=6366 Len=0 TSval=4043193699 TSecr=911862172
    7 3.524840  127.0.0.1  127.0.0.1    TCP       68 59860 → 3539 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1060611374 TSecr=0 SACK_PERM
    8 3.542701  127.0.0.1  127.0.0.1    TCP       68 3539 → 59860 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=3019295022 TSecr=1060611374 SACK_PE…
    9 3.542810  127.0.0.1  127.0.0.1    TCP       56 59860 → 3539 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1060611392 TSecr=3019295022
   10 3.542856  127.0.0.1  127.0.0.1    TCP       56 [TCP Window Update] 3539 → 59860 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=3019295040 TSecr=1060611392
   11 3.543444  127.0.0.1  127.0.0.1    HTTP     384 GET /stream HTTP/1.1
   12 3.543504  127.0.0.1  127.0.0.1    TCP       56 3539 → 59860 [ACK] Seq=1 Ack=329 Win=407936 Len=0 TSval=3019295040 TSecr=1060611392
   13 3.544912  127.0.0.1  127.0.0.1    TCP      331 3539 → 59860 [PSH, ACK] Seq=1 Ack=329 Win=407936 Len=275 TSval=3019295042 TSecr=1060611392
   14 3.544961  127.0.0.1  127.0.0.1    TCP       56 59860 → 3539 [ACK] Seq=329 Ack=276 Win=408000 Len=0 TSval=1060611394 TSecr=3019295042
   15 3.629552  127.0.0.1  127.0.0.1    TCP       68 59861 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=2661660403 TSecr=0 SACK_PERM
   16 3.635023  127.0.0.1  127.0.0.1    TCP       68 8080 → 59861 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2850318222 TSecr=2661660403 SACK_PE…
   17 3.635025  127.0.0.1  127.0.0.1    TCP       68 59862 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=64 TSval=1148346835 TSecr=0 SACK_PERM
   18 3.635059  127.0.0.1  127.0.0.1    TCP       56 59861 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=2661660408 TSecr=2850318222
   19 3.635189  127.0.0.1  127.0.0.1    HTTP     744 GET / HTTP/1.1
   20 3.639913  127.0.0.1  127.0.0.1    TCP       68 8080 → 59862 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=64 TSval=2107153466 TSecr=1148346835 SACK_PE…
   21 3.639954  127.0.0.1  127.0.0.1    TCP       56 [TCP Window Update] 8080 → 59861 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=2850318227 TSecr=2661660408
   22 3.639965  127.0.0.1  127.0.0.1    TCP       56 8080 → 59861 [ACK] Seq=1 Ack=689 Win=407552 Len=0 TSval=2850318227 TSecr=2661660408
   23 3.639981  127.0.0.1  127.0.0.1    TCP       56 59862 → 8080 [ACK] Seq=1 Ack=1 Win=408256 Len=0 TSval=1148346840 TSecr=2107153466
```

```
> Frame 19: 744 bytes on wire (5952 bits), 744 bytes captured (5952 bits) on interface lo0, id
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 59861, Dst Port: 8080, Seq: 1, Ack: 1, Len: 688
> Hypertext Transfer Protocol
```

```
0040  54 50 2f 31 2e 31 0d 0a  48 6f 73 74 3a 20 31 32   TP/1.1·· Host: 12
0050  37 2e 30 2e 30 2e 31 3a  38 30 38 30 0d 0a 43 6f   7.0.0.1: 8080··Co
0060  6e 6e 65 63 74 69 6f 6e  3a 20 6b 65 65 70 2d 61   nnection : keep-a
0070  6c 69 76 65 0d 0a 73 65  63 2d 63 68 2d 75 61 3a   live··se c-ch-ua:
0080  20 22 47 6f 6f 67 6c 65  20 43 68 72 6f 6d 65 22    "Google  Chrome"
0090  3b 76 3d 22 31 33 35 22  2c 20 22 4e 6f 74 2d 41   ;v="135" , "Not-A
00a0  2e 42 72 61 6e 64 22 3b  76 3d 22 38 22 2c 20 22   .Brand"; v="8", "
00b0  43 68 72 6f 6d 69 75 6d  22 3b 76 3d 22 31 33 35   Chromium ";v="135
00c0  22 0d 0a 73 65 63 2d 63  68 2d 75 61 2d 6d 6f 62   "··sec-c h-ua-mob
00d0  69 6c 65 3a 20 3f 30 0d  0a 73 65 63 2d 63 68 2d   ile: ?0· ·sec-ch-
00e0  75 61 2d 70 6c 61 74 66  6f 72 6d 3a 20 22 6d 61   ua-platf orm: "ma
00f0  63 4f 53 22 0d 0a 55 70  67 72 61 64 65 2d 49 6e   cOS"··Up grade-In
0100  73 65 63 75 72 65 2d 52  65 71 75 65 73 74 73 3a   secure-R equests:
0110  20 31 0d 0a 55 73 65 72  2d 41 67 65 6e 74 3a 20    1··User -Agent: 
0120  4d 6f 7a 69 6c 6c 61 2f  35 2e 30 20 28 4d 61 63   Mozilla/ 5.0 (Mac
0130  69 6e 74 6f 73 68 3b 20  49 6e 74 65 6c 20 4d 61   intosh;  Intel Ma
0140  63 20 4f 53 20 58 20 31  30 5f 31 35 5f 37 29 20   c OS X 1 0_15_7) 
0150  41 70 70 6c 65 57 65 62  4b 69 74 2f 35 33 37 2e   AppleWeb Kit/537.
0160  33 36 20 28 4b 48 54 4d  4c 2c 20 6c 69 6b 65 20   36 (KHTM L, like 
0170  47 65 63 6b 6f 29 20 43  68 72 6f 6d 65 2f 31 33   Gecko) C hrome/13
0180  35 2e 30 2e 30 2e 30 20  53 61 66 61 72 69 2f 35   5.0.0.0  Safari/5
0190  33 37 2e 33 36 0d 0a 41  63 63 65 70 74 3a 20 74   37.36··A ccept: t
01a0  65 78 74 2f 68 74 6d 6c  2c 61 70 70 6c 69 63 61   ext/html ,applica
01b0  74 69 6f 6e 2f 78 68 74  6d 6c 2b 78 6d 6c 2c 61   tion/xht ml+xml,a
01c0  70 70 6c 69 63 61 74 69  6f 6e 2f 78 6d 6c 3b 71   pplicati on/xml;q
01d0  3d 30 2e 39 2c 69 6d 61  67 65 2f 61 76 69 66 2c   =0.9,ima ge/avif,
01e0  69 6d 61 67 65 2f 77 65  62 70 2c 69 6d 61 67 65   image/we bp,image
01f0  2f 61 70 6e 67 2c 2a 2f  2a 3b 71 3d 30 2e 38 2c   /apng,*/ *;q=0.8,
```

Loopback: lo0: <live capture in progress>   Packets: 56   Profile: Default

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

A certificate authority cannot issue SSL certificates for local web servers for several reasons: 1) There’s no publicly verifiable domain as certificate authorities only issue certificates for publicly accessible domain names like google.com or amazon.com. They cannot issue certificates for

localhost because these addresses are not globally unique. 2) It’s impossible to validate the domain through DNS validation, HTTP validation, or email validation.
Therefore, I created a self-signed certificate and then manually configured my system to trust. I used mkcert, a tool designed for creating locally trusted development certificates. First, I installed mkcert using Homebrew. Then I ran the mkcert install command, which created a local Certificate Authority and added it to my system's trusted root certificate store. This step tells my operating system and browsers to trust certificates signed by this local CA.
Next, I generated the SSL certificate and private key for localhost, which created two files: the certificate containing the public key and the private key. I then created a custom nginx configuration file that specified port 443 for HTTPS, pointed to my certificate and private key files, and configured modern TLS protocols. After stopping my HTTP container, I started a new nginx container with the HTTPS configuration, mapping port 443 and mounting both my certificates and configuration file into the container. Testing the setup by navigating to https://localhost in my browser showed a padlock icon and no security warnings—the self-signed certificate was trusted because my system trusted the local CA that signed it.

*Comparing HTTP and HTTPS traffic:*
The HTTPS traffic capture began with a TLS handshake where my browser and server negotiated encryption settings: my browser sent a Client Hello listing supported TLS versions and cipher suites, the server responded with a Server Hello selecting TLS 1.3 and a cipher suite, sent my certificate, and both sides exchanged cryptographic keys to establish a shared secret. After the handshake, all packets showed only encrypted Application Data. While Wireshark captured the packets, it could only show encrypted gibberish. The actual content—URLs, headers, and page data—was protected and unreadable without the private key. The HTTP requests, headers, and HTML content that were clearly visible in the HTTP capture were now completely hidden by encryption. On the contrary, HTTP transmitted everything in plaintext—URLs, headers, and complete page content were all visible in Wireshark. Any network observer could read the entire communication.