

Hewitt Watkins

Assignment 1: Public Key Infrastructure

10/24/25

Sources: Class Notes, ChatGPT (code generation and paragraph proofreading)

Task 1: Host A Local Web Server

Task 1 asks us to create our own web server over http. In order to do this, I used Flask, which is a lightweight Python web framework. Part of what Flask does automatically is running its own HTTP server, which in my case I configured to run host="0.0.0.0" and port=8000. This setup allows me to access the Flask site on devices connected to my local network. Flask is really cool because it makes it easy to handle routing for different pages on the site, such as a login page or register an account page. For my site, having used Flask in the past, I used ChatGPT to help me setup a simple webpage in which a user could register a login, login, and if successful, see a "home page". I wanted this interface because I specifically wanted to see if private information like passwords were protected over http (spoiler: we know it's not) and https.

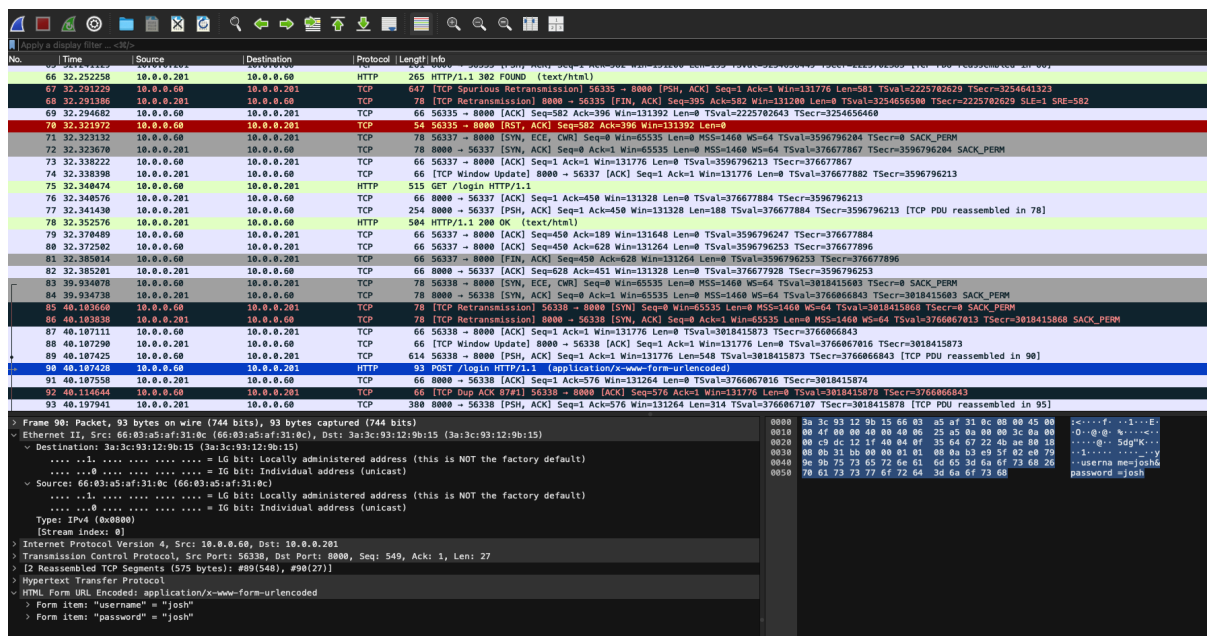
If you're interested in the specific setup of my small web application, it's in my repo. It consists of [app.py](#), index.html, and a templates directory with 4 html templates.

Note: ChatGPT also used the Werkzeug library for password hashing when storing the passwords – this is a bit overkill, but doesn't affect what http or https are doing, so I left it in.

The image below is the server / web application [app.py](#) running in my terminal:

```
(base) school@Hewitts-MacBook-Pro-3 assignment1 % python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8000
* Running on http://10.0.0.201:8000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 113-684-673
127.0.0.1 - - [21/Oct/2025 22:09:36] "GET / HTTP/1.1" 302 -
127.0.0.1 - - [21/Oct/2025 22:09:36] "GET /login HTTP/1.1" 200 -
127.0.0.1 - - [21/Oct/2025 22:09:36] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [21/Oct/2025 22:09:38] "GET /login HTTP/1.1" 200 -
127.0.0.1 - - [21/Oct/2025 22:09:39] "GET /register HTTP/1.1" 200 -
127.0.0.1 - - [21/Oct/2025 22:09:51] "POST /register HTTP/1.1" 302 -
127.0.0.1 - - [21/Oct/2025 22:09:51] "GET /login HTTP/1.1" 200 -
127.0.0.1 - - [21/Oct/2025 22:09:58] "POST /login HTTP/1.1" 302 -
127.0.0.1 - - [21/Oct/2025 22:09:58] "GET /dashboard HTTP/1.1" 200 -
127.0.0.1 - - [21/Oct/2025 22:10:07] "GET /logout HTTP/1.1" 302 -
127.0.0.1 - - [21/Oct/2025 22:10:07] "GET /login HTTP/1.1" 200 -
10.0.0.60 - - [21/Oct/2025 22:12:39] "GET / HTTP/1.1" 302 -
10.0.0.60 - - [21/Oct/2025 22:12:40] "GET /login HTTP/1.1" 200 -
10.0.0.60 - - [21/Oct/2025 22:12:40] "GET /favicon.ico HTTP/1.1" 404 -
10.0.0.60 - - [21/Oct/2025 22:12:57] "POST /login HTTP/1.1" 401 -
10.0.0.60 - - [21/Oct/2025 22:13:11] "POST /login HTTP/1.1" 302 -
```

Task 2: Identify Why HTTP Isn't Secure (see below)



The image shows a Wireshark packet capture of an HTTP POST request. The packet list on the left shows a POST request to /login. The packet details pane on the right shows the HTTP form items: username=josh and password=josh. The packet bytes pane on the right shows the raw data of the POST request.

```
> Form item: "username" = "josh"
> Form item: "password" = "josh"
```

As we learned in class, HTTP is not secure because it sends all data in plaintext, with no encryption at all. In turn, if I were to communicate with a server over http, any contents of the requests (URLs, headers, and any sensitive information) would be sent just as they are - in human readable form. This means that anyone on the same network as you could “sniff” your web traffic by using a tool like wireshark to capture your network traffic as a packet trace. By investigating that packet trace, they will find all of your network activity between you and the server you're communicating with in human readable form. For platforms with sensitive information, like your bank app, you probably wouldn't want someone to be able to “sniff” your username and password at login.

My Flask app demonstrates this very well. I had it running as described in step one, and then made sure my phone was on the same local network as my computer. I then went to the web application on my phone (<http://10.0.0.201:8000>) using safari, created a username and login, and logged in. All of this activity was recorded in http-trace.pcap (via wireshark using Wi-Fi: en0 and relevant display filters). Remember, wireshark was running on my laptop, independent of my phone or hosting the web application. In turn, it's acting like any potential third party, showing how anyone on my network could have recorded this packet trace. The image above is a packet showing a HTTP Post request to “/login”, which in plain human-readable text, shows my username and password.

In addition to fields like username and password, the image above demonstrates an eavesdropper can see which resources I (or any client) requested. This includes URLs and paths, what type of HTTP method is used (GET, POST), HTTP request headers, and the complete contents of the server's response (all HTML, JavaScript, CSS, etc.). In turn, this means an eavesdropper could essentially capture enough information to recreate your entire browsing session with a website.

Overall, unencrypted HTTP is bad. Your data is NOT safe.

Task 3: Identify Why HTTP Isn't Secure (see below)

The reason I can't obtain an SSL Certificate for my local web server from a certificate authority (CA) is because I do not have a publicly registered and verified domain name. The reason for a CA is to verify that domain names are being hosted/used by people who are who they say they are. For example, I want apple.com to be for Apple. My locally hosted web app, however, uses <http://127.0.0.1:8000>, which is not a publicly registered domain name, and there's no way to register it to me permanently (I could never own it). In turn the CA will not issue me a certificate for a local server because ownership cannot be proven and there is no public DNS ownership to validate.

In turn, because I can't, I generated my own local SSL (TLS) certificate (self-signed) and added it to my machine's list of locally trusted root certificates.

```
(base) school@Hewitts-MacBook-Pro-3 assignment1 % openssl req -x509 -newkey rsa:2048 -nodes \
-keyout key.pem -out cert.pem -days 365 \
-subj "/CN=localhost" \
-addext "subjectAltName=DNS:localhost,IP:127.0.0.1"

Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'key.pem'
```

To do this, I generated a self-signed TLS certificate and key using OpenSSL (seen in the image above). Key.pem is the private key and cert.pem is the self-signed certificate.

I then added the certificate to my local trusted root in keychain. After that I created a new app file, [app2.py](#), with updated code so that HTTPS/TLS was used. All that changed was after line 70, in main.

In the original [app.py](#) it was:

```
app.run(host="0.0.0.0", port=8000, debug=True)
```

And in the new [app2.py](#):

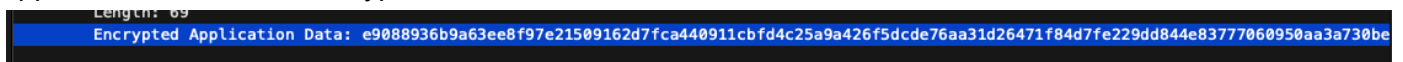
```
ssl_ctx = ("cert.pem", "key.pem")
app.run(host="0.0.0.0", port=8443, debug=True, ssl_context=ssl_ctx)
```

I then ran the [app2.py](#) application in terminal:

```
(base) school@Hewitts-MacBook-Pro-3 assignment1 % python3 app2.py
* Serving Flask app 'app2'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on https://127.0.0.1:8443
* Running on https://10.0.0.201:8443
```

Finally, I connected to the website again in my computer's browser and captured all relevant network activity over Wireshark (using Loopback: lo0 and relevant display filters like "tcp port 8443") (I used my computer this time because adding the certificate to my phone was a bit tricky). Like last time, I created a new username and password, and logged in with the registered account. However, as is discussed below, none of this can be seen in the packet trace.

Overall, our packet trace of HTTPS activity looks very different from our HTTP application activity. In our HTTPS trace, we can see the TLS handshake process. However, any following packet that contains any actual web traffic data is now labeled as "Application Data". That application data is now encrypted and in a non-human-readable form.



Now, any data that an eavesdropper might capture in these packets is unreadable and completely useless to them. They do not have the secret key needed to decrypt the encrypted data, and in turn will never know what's contained in these packets. Furthermore, it's not only details like form data that's encrypted. It's URL paths, cookies, HTML and more – it's all encrypted now!

Overall, next to no information is visible to a sniffer that is collecting network activity when using HTTPS. They can see IP addresses and port numbers (which are needed for routing), a TLS handshake message, and domain names. But that's it! Everything else is now complete jargon except for the server and client who are actively talking to one another.

Image of HTTPS packet trace:

