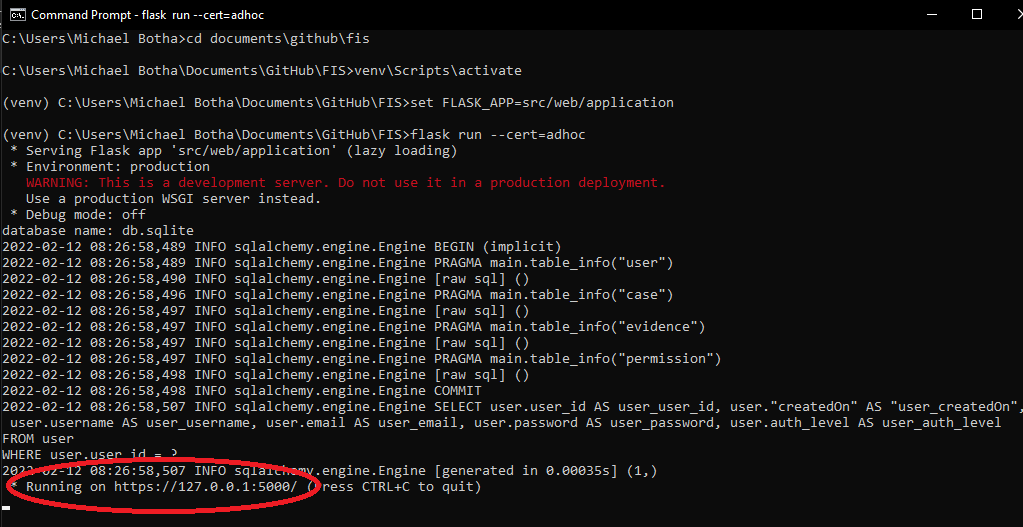
Code Execution and Testing Evidence

HTTPs vs HTTP

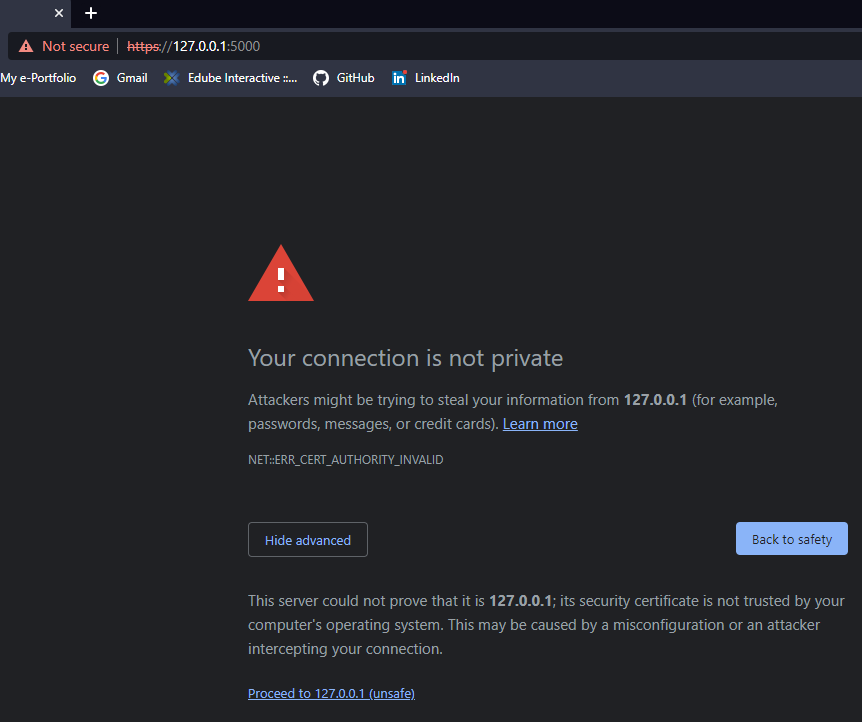
As conveyed in the “README” document, HTTP was used instead of HTTPs due to issues with regards to trusted certificates. One can run Flask servers in HTTPs mode using self-signed certificates as shown in the following figures.

In Figure 1 self-signing certificates are activates using the “--cert=adhoc” command option when running the Flask application.



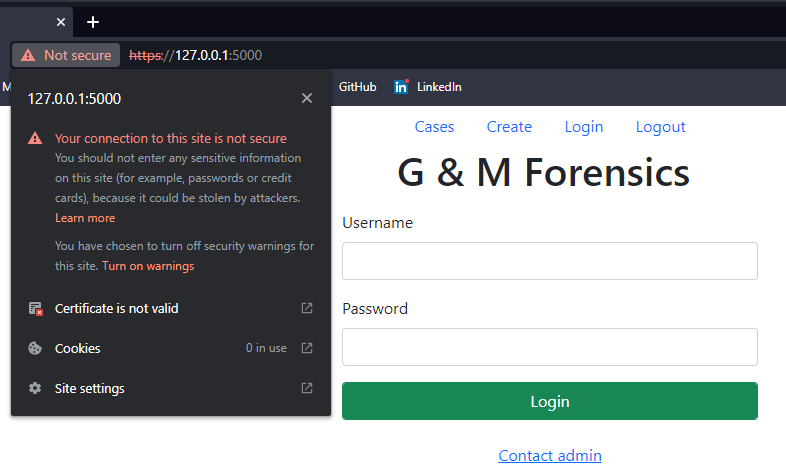
**Figure 1 – Executing the Flask Server with HTTPs**

In Figure 2 one can see how the browser recognises the self-signed certificate and flags it as a warning. However, the connection can still be granted by selecting “Advanced” and then “Proceed to 127.0.0.1 (unsafe).



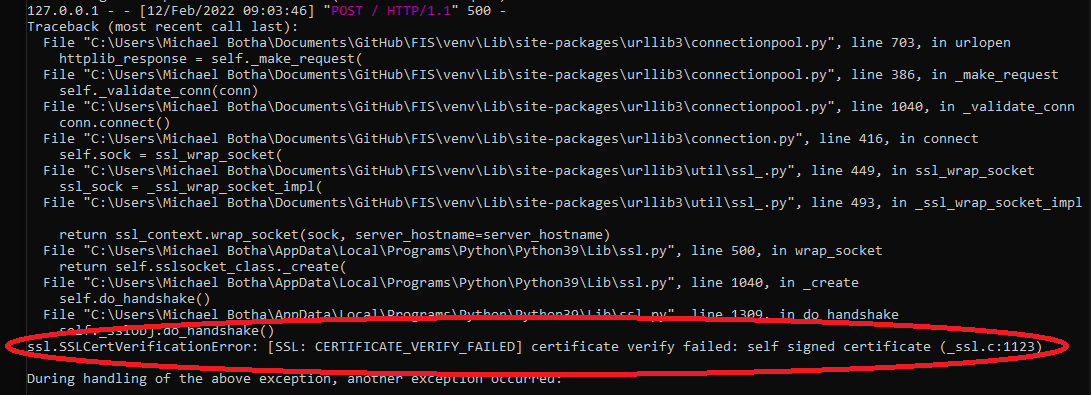
**Figure 2 – Invalid Certificate Warning**

In Figure 3 the sign in home page is seen but with the warning from the browser still present.



**Figure 3 – Overriding the Protection**

By removing the exception handling in the code that handles failed connection attempts with the microservice, and running the flask server in debug mode, one can observe in Figure 4 that the connection to the microservice is denied due to the self-signed certificate.



**Figure 4 – HTTPs Connection Refusal**

To overcome the challenge HTTP was used for this non-production environment. With Triple DES as the encryption algorithm for communication with the microservice.

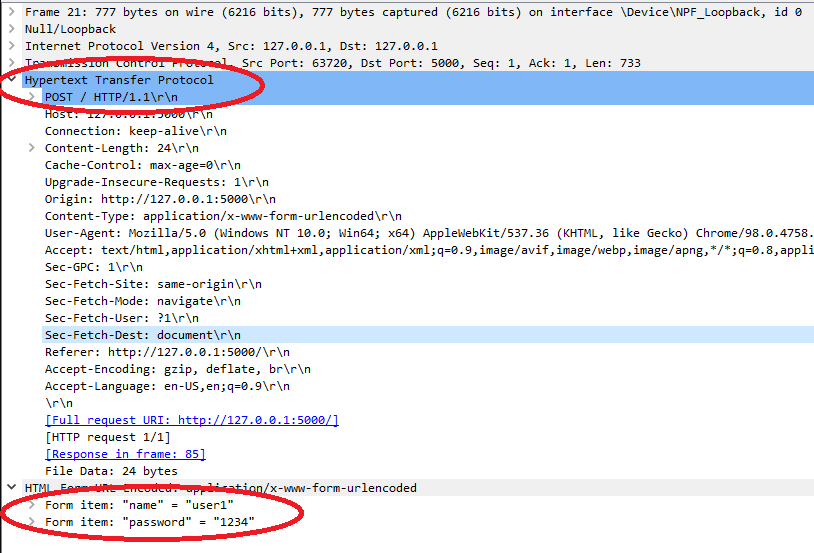
Authenticating

There are various conditions catered for during the authentication process, where the microservice is requested to perform verification using database records. The following section will present four conditions:

1. Correct authentication process.
2. Incorrect username.
3. Incorrect password.
4. SQL injection attack.

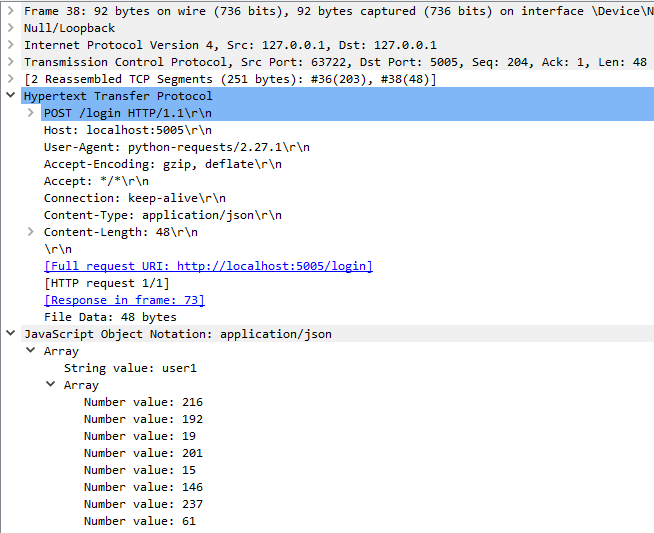
Log in credentials will be created by a system administrator, who has a direct-access permission to the SQL database. When a client enters their username and password and submits it to the web application/server, the credentials need to be verified to authenticate the user and provide the application with knowledge of the user’s authorisation level.

Firstly, the username and password will be sent to the web server as an HTTP POST message by the browser/client via and HTML form. The message is received by the application and processed under the section of code allocated to POST messages as seen in line 90 to 151 of the “application.py” file. One of the test results from performing this process can be seen in the Wireshark capture of Figure 5.

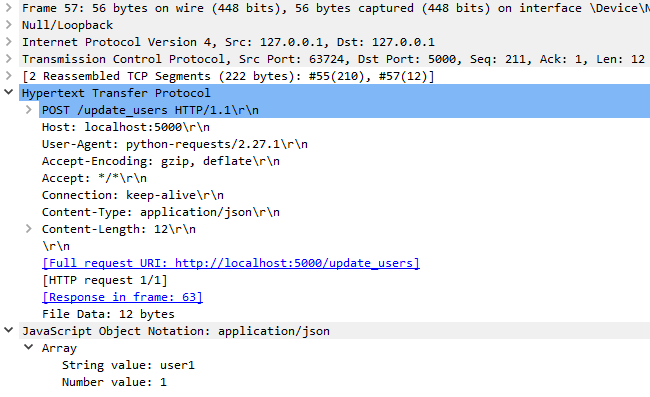


**Figure 5 – Wireshark Capture of Login Form Submission**

Once the form is submitted to the web application it encrypts the password, splits the bytecode string representing the password into a list of integers, and then creates a list with the username as the first element and the list of integers as the second. The list is converted into a JSON object and sent to the microservice’s API. Test results of this process are presented in Figure 6. Line 35 to 69 in the “Authenticate.py” module defines the API and the processing of the submitted JSON object. Once the microservice receives the JSON object, it decrypts the password and hashes it, thereafter, creates a user object by fetching the relevant user’s credentials, in the form of a username, hashed password, and authorisation level from the database. The fetched values are then compared with those submitted by the user and an appropriate response submitted to the web application as seen in Figure 7.

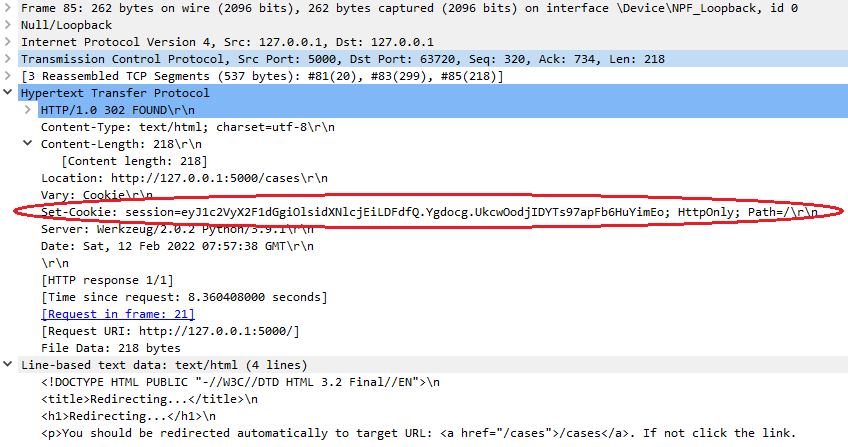


**Figure 6 – Wireshark Capture of JSON Object Sent to Microservice API**



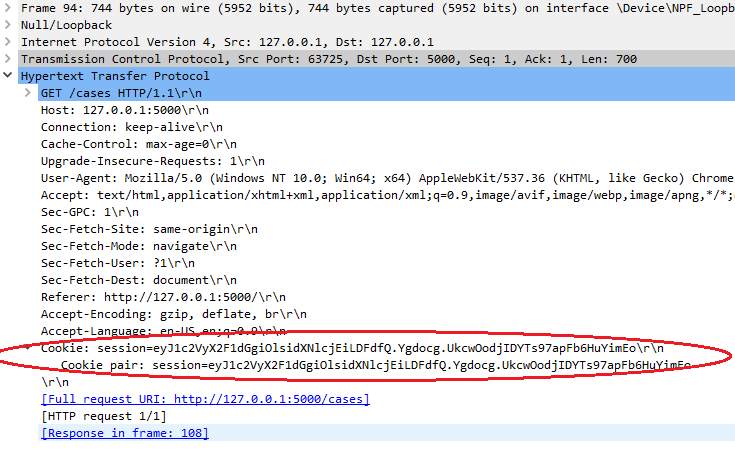
**Figure 7 – Wireshark Capture of JSON Object Returned to Web Application API**

The JSON response submitted to the web application’s appropriate API is stored in a global dictionary temporarily, for access by the separate process handling the submitted user credentials. The process was paused with a timer of 4 seconds prior to the API call to the microservice to enable the communication process to complete. In the case where the correct credentials were entered, the web application creates a Flask session, which contains a dictionary with string key of “user\_auth” and a value which is a list containing the username and authorisation level. The session details are sent to the user’s browser in a redirect message containing the “Set-Cookie” header field with a value set to a cryptographically signed session cookie as seen in the test result of Figure 8. The redirect message informs the browser to GET the “cases” page.



**Figure 8 – Wireshark Capture of Web Application Setting User’s Browser Session Cookie**

The browser application then requests the “cases” page, with the recently added cookie set to the relevant session value. This value is now used by the web application to ensure the user has been authenticated, and will be used to check the level of authorisation in future transaction requests, as can be seen in Figure 9.



**Figure 9 – Wireshark Capture of Browser using Cookie with Session Value**

Once the testing of the correct login process had been completed, it was necessary to ensure that incorrect authentication conditions had been sufficiently catered for. Therefore, a similar testing technique was used but now with more of a branch code testing mentality, in addition to the functionality and data flow test requirements already fulfilled (University of Essex Online, 2022).