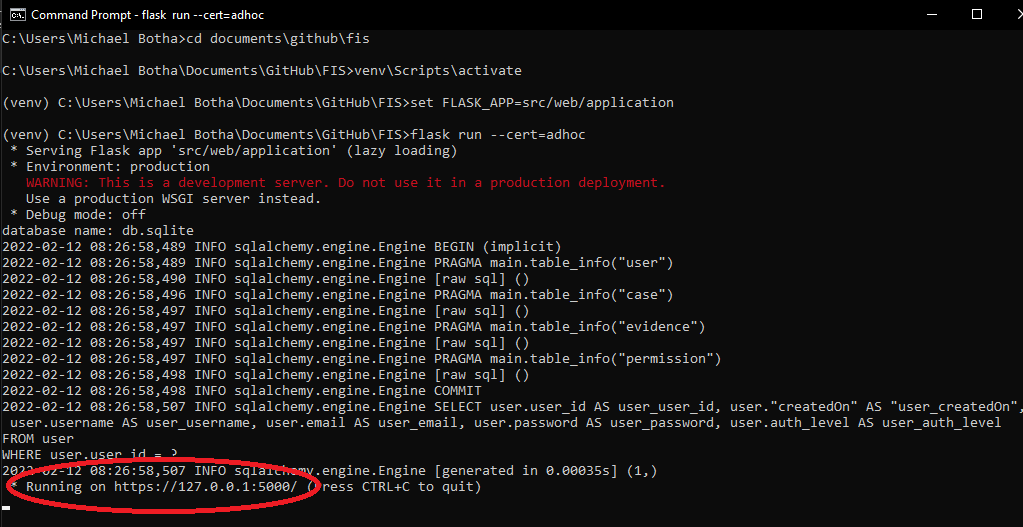
Code Execution and Testing Evidence

HTTPs vs HTTP

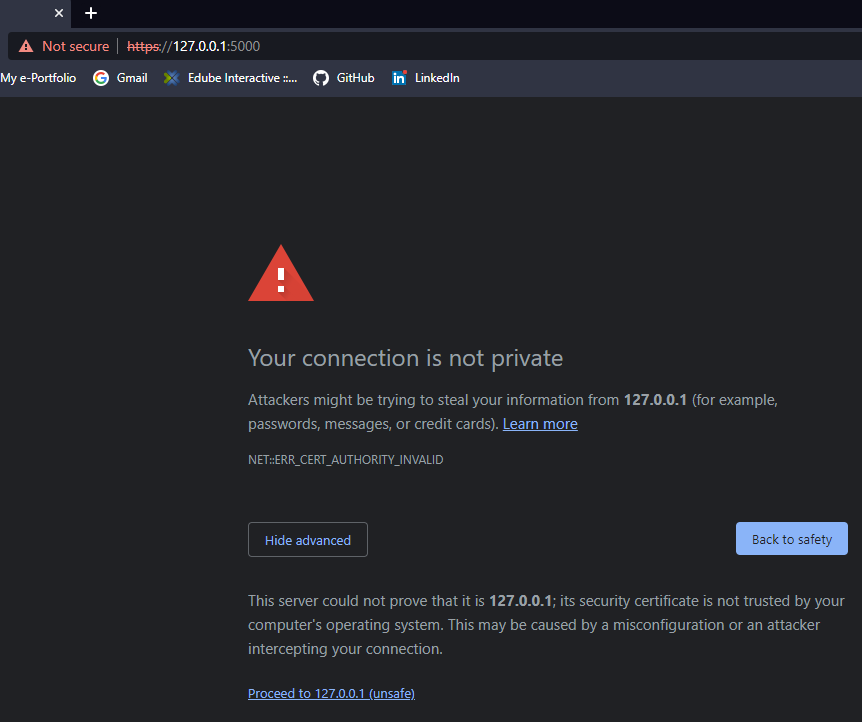
As conveyed in the “README” document, HTTP was used instead of HTTPs due to issues with regards to trusted certificates. One can run Flask servers in HTTPs mode using self-signed certificates as shown in the following figures.

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



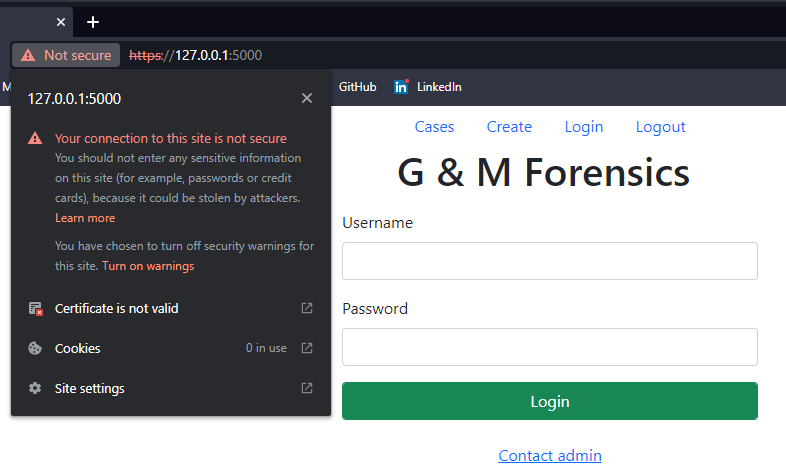
**Figure 1 – Executing the Flask Server with HTTPs**

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).



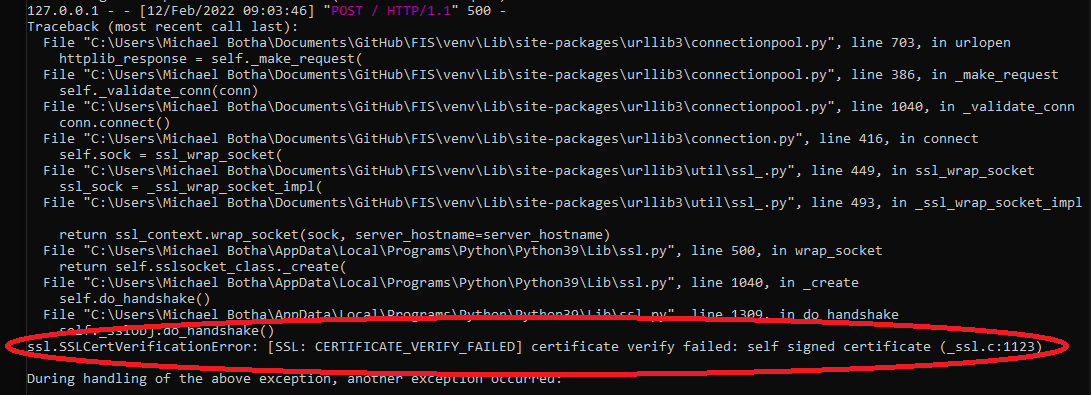
**Figure 2 – Invalid Certificate Warning**

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

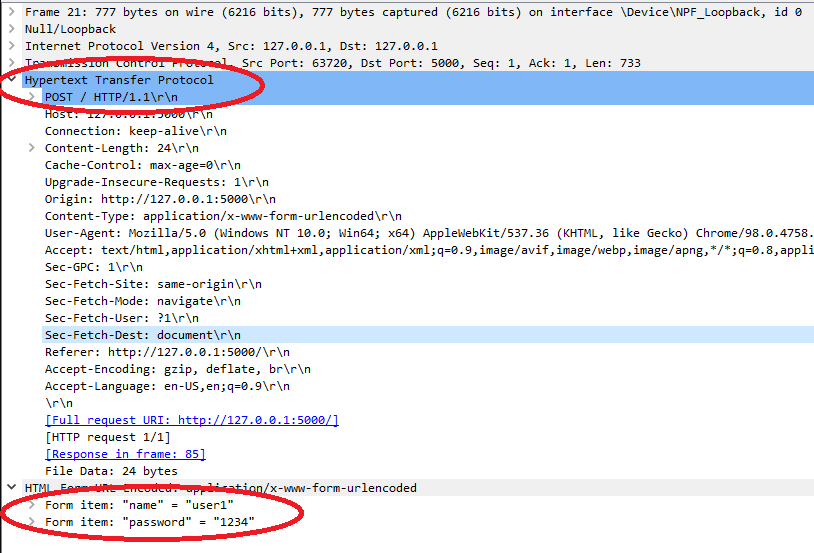
Authenticating

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 the testing of four conditions:

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

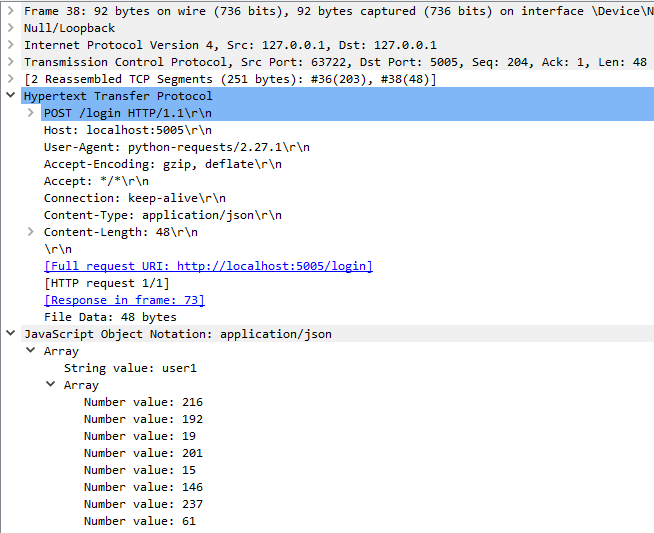
Log in credentials will be created by a system administrator, who has a direct-access permission to the SQL database. When a client enters their username and password and submits it to the web application/server, the credentials need to be verified to authenticate the user and provide the application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

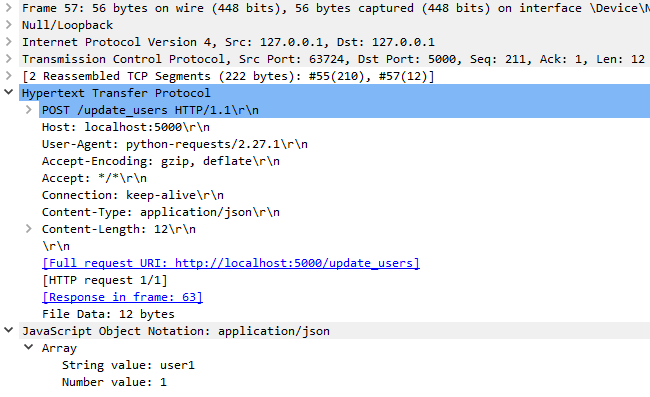


**Figure 5 – Wireshark Capture of Login Form Submission**

Once the form is submitted to the web application it encrypts the password, splits the bytecode string representing the encrypted 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 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.

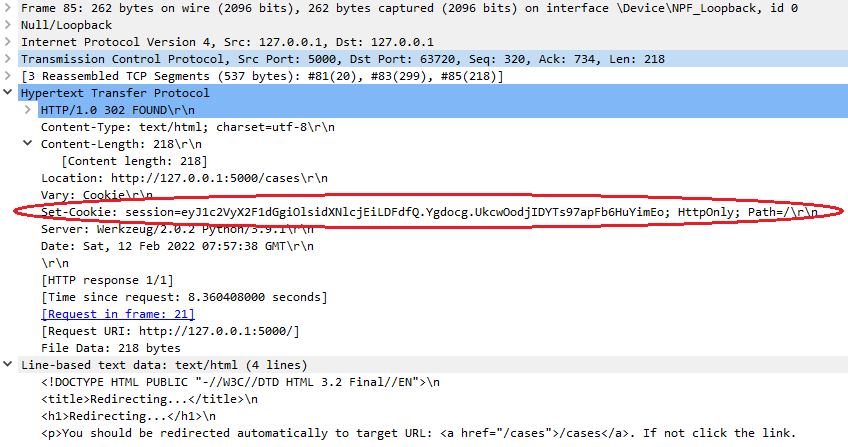


**Figure 6 – Wireshark Capture of JSON Object Sent to Microservice API**



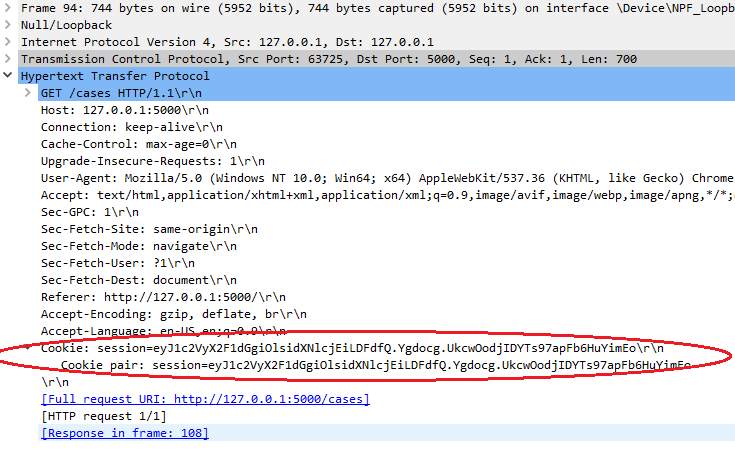
**Figure 7 – Wireshark Capture of JSON Object Returned to Web Application API**

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.

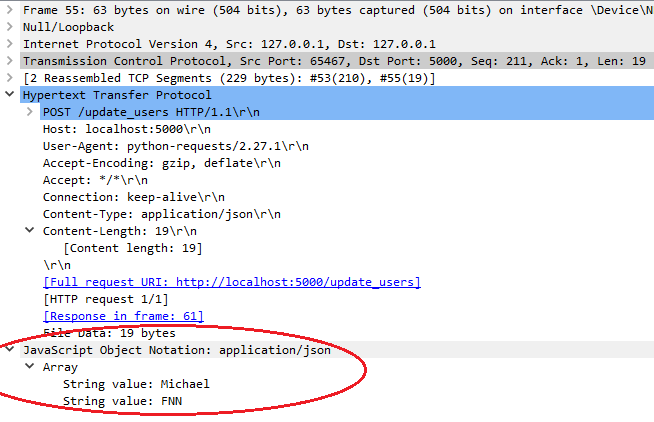


**Figure 9 – Wireshark Capture of Browser using Cookie with Session Value**

At this point it must be noted that whilst proving the correct functionality of the authentication process, the inherent concurrency of the Flask web framework has also been proved (Pallets, N.D). Flask runs thread locals which is confirmed when the login process is paused by 4 seconds in line 106 of “application.py”, whilst the process waits for the global dictionary to be updated after the appropriate API call to and response from the microservice (Pallets, N.D).

Once testing of the correct login process had been completed, it was necessary to ensure that incorrect authentication conditions as laid out in points 2 to 4 previously noted 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 testing techniques already used (University of Essex Online, 2021).

If a user were to submit a form containing a username not currently present in the relevant database, the JSON object returned to the main web application still contains the previously entered username, but now instead of the authorisation level being added, a flag of “FNN” representing Failed No Name is appended as seen in Figure 10. The main web application would then not create the session as previously performed (Figure 11) but send a message to the user as shown in Figure 12.

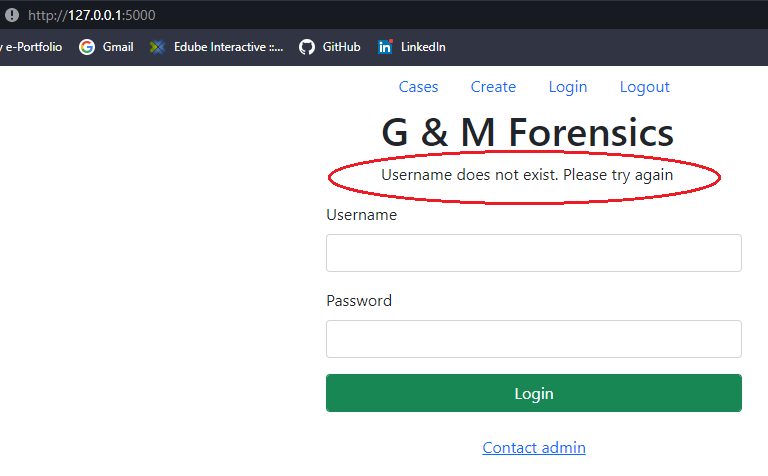


**Figure 10 – Wireshark Capture of the Returned “FNN” Flag from the Microservice**

Graphical user interface, text, application

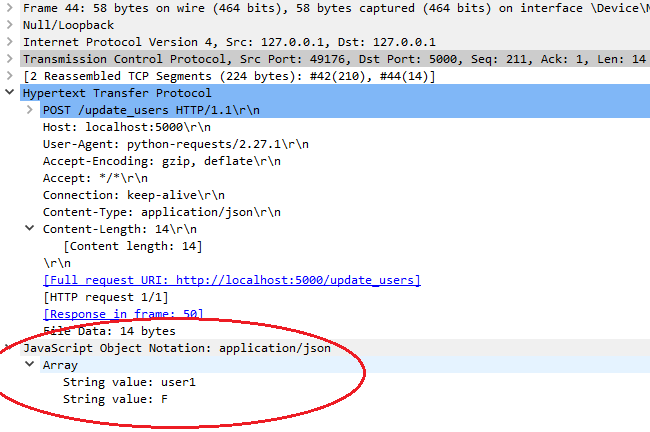
Description automatically generated

**Figure 11 – Web Application Response to the User Showing no Session Cookie**



**Figure 12 – Web Application Response to the User who entered an Incorrect Username**

Should the username exist but the password be incorrect the flag of “F” is returned to the main web application by the microservice (Figure 13), and the browser displays the message as seen in figure 14



**Figure 13 – Wireshark Capture of the Returned “F” Flag from the Microservice**

Graphical user interface, text, application

Description automatically generated

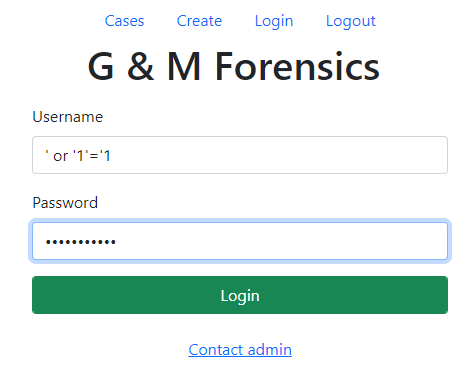
**Figure 14 – Web Application Response to the User who entered an Incorrect Password**

Injection attacks on web applications are common (OWASP, N.D). Therefore, it was essential that this was tested. Figure 15 presents popular SQL attacks that are used, and that was used to test the program (Bricks, N.D).

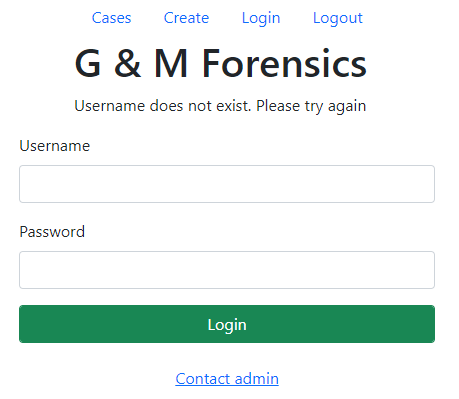


**Figure 15 – Popular SQL Injection Attacks used to test the Distributed Program (Bricks, N.D)**

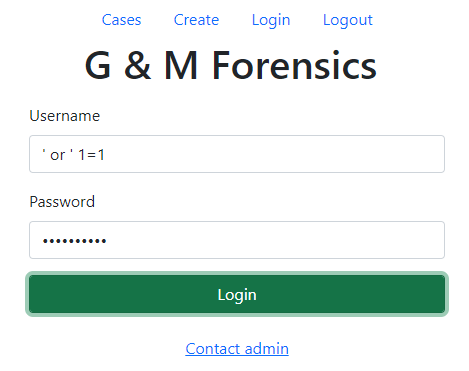
As a sample of running the test injections in Figure 15, Figures 17 and 19 depict the program’s response to row 5 and 6’s code.



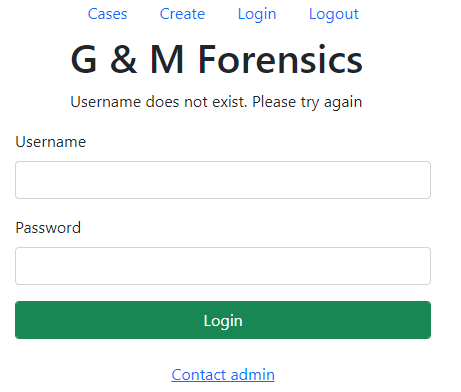
**Figure 16 – First SQL Injection Input into Program**



**Figure 17 – Program’s Response to First SQL Injection**



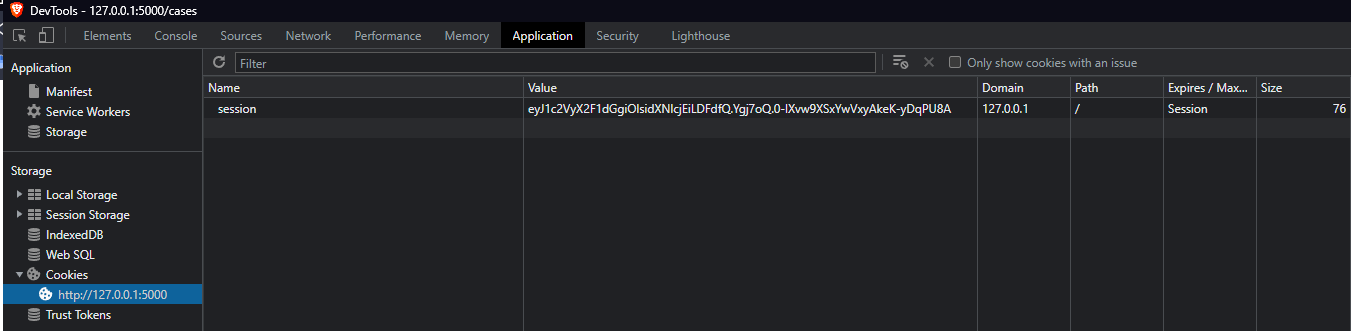
**Figure 18 – Second SQL Injection Input into Program**



**Figure 19 – Program’s Response to Second SQL Injection**

Multiple Sessions and Logging Out

Multiple sessions with the web server needed to be tested, where it was ensured that each user maintained a unique session ID in the form of the previously mentioned session cookie. This is important because if sessions are mixed a user or attacker could possibly attain a legitimate user’s session, and at a level of authorisation higher than the default. To test this scenario two separate browser applications were used, as the browser caches any session cookies, therefore overwriting any previously set ones when a new session is created. Each browser was checked to see a unique session cookie from the application was present. Using the developer tools resident in browsers Figures 20 and 21 were produced. Careful evaluation of the two figures will prove that separate session identifiers are present.



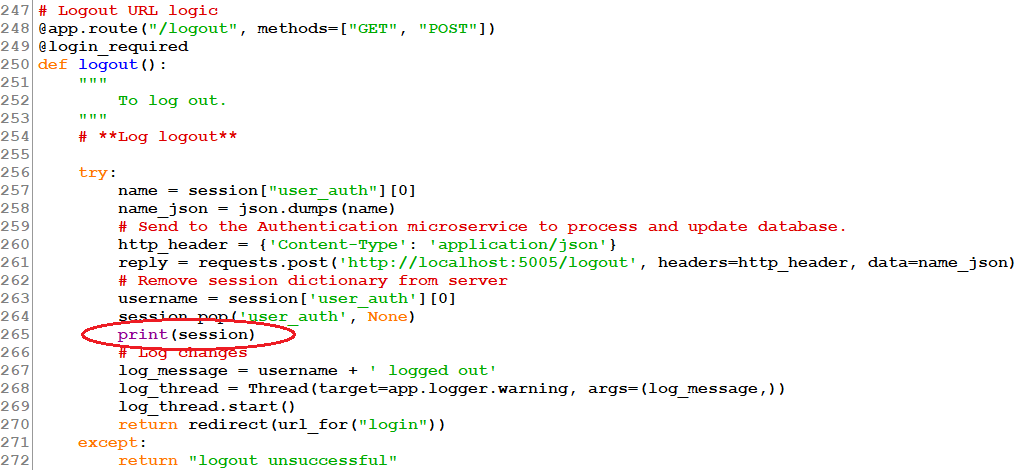
**Figure 20 – First Session’s Cookie**

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 21 – Second Session’s Cookie**

Further to the above, it was important to test that when a user logged out of the system that there session was in deed terminated on the server side. To test this a portion of code was added temporarily to the “/logout” route/URL portion of the application, where the session object was printed to the terminal once a session was terminated.



**Figure 22 – Code Added to Test Session Termination**

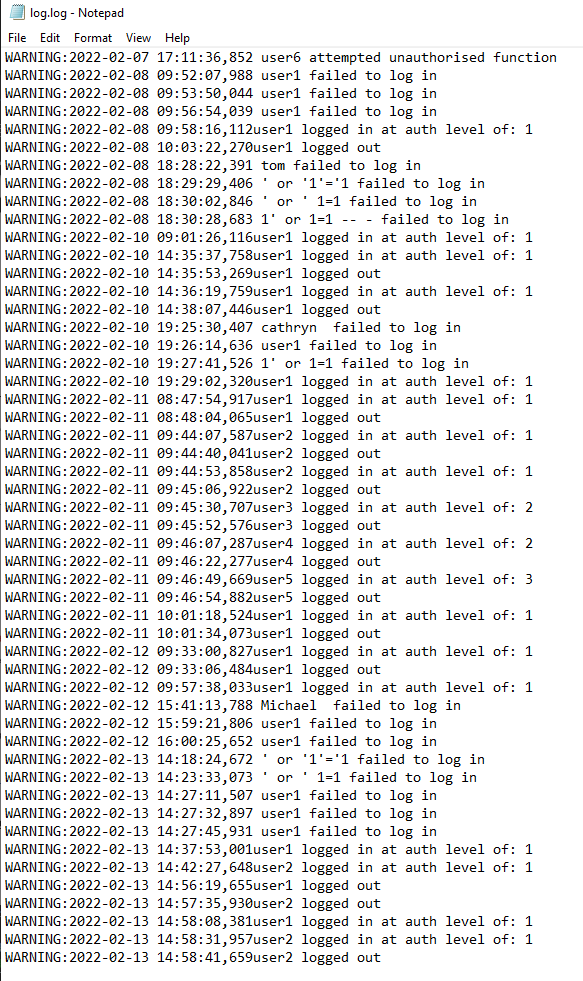
Text

Description automatically generated

**Figure 23 – Terminal Output Showing that the Session Object is Empty**

Logging

It was important to ensure that specific predefined warning conditions are logged, and logged to the right place with the correct meta data. This was tested by executing all the conditions that would produce a log to the local “log.log” file in the FIS directory, and then viewing the log file as seen in Figure 24.



**Figure 24 – Sample of log File**

To improve performance any log written to the log.log file was run in a separate thread. This could be tested in various complex and simple ways. One basic test was to put a 5 second delay on the logging thread, and see whether the main thread still responded to the initial condition raising the log within the normal expected time.

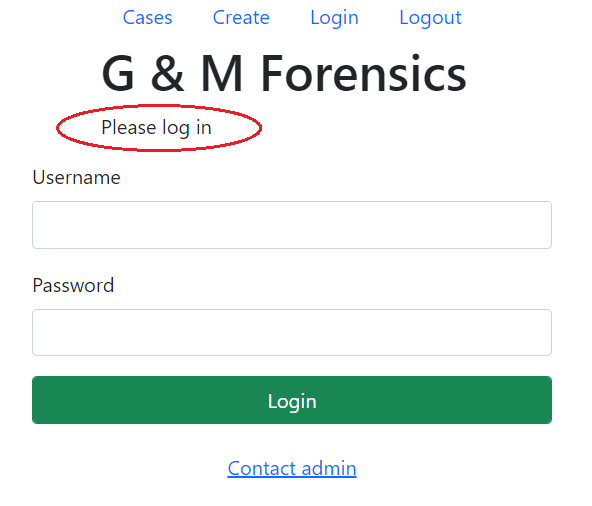
General Functionality and Authorisation

A user can perform various activities with the system in line with their level of authorisation. Without a complete security policy it was not possible to implement an extensive authorisation feature. However, four levels of authorisation were created as per Table 1.

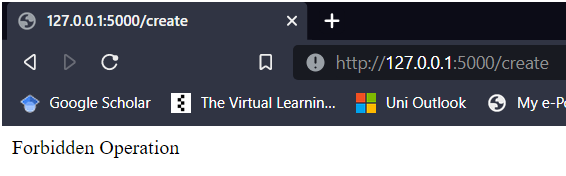
**Table 1 – Levels of Authorisation**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Auth. Level** | **Log in** | **View Case** | **Edit Case** | **Delete Case** | **Create Case** |
| Unauthenticated | Yes | No | No | No | No |
| 1 | N/A | Yes | Yes | Only one created by user | Yes |
| 2 | N/A | Yes | Yes | Only one created by user | No |
| 3 | N/A | Yes | Yes | Only one created by user | No |

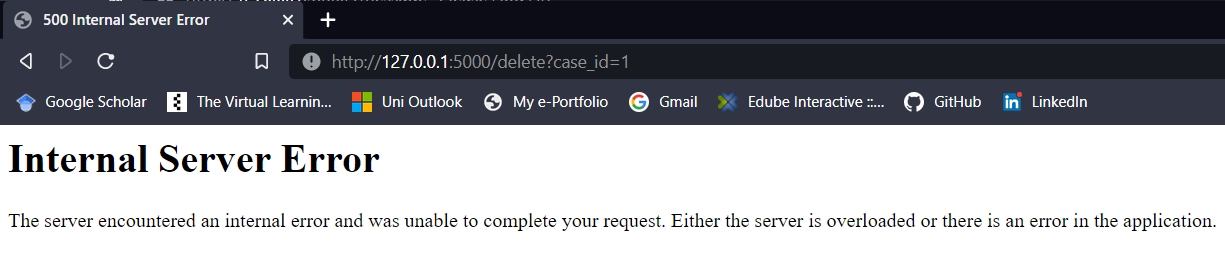
The various levels of authorisation were tested as per the following figures.



**Figure 25 – Message Displayed to Unauthorised User When Trying to Access any Options**



**Figure 25 – Message Displayed to User of Auth. 2 or 3 When Trying to Create a Case**



**Figure 26 – Message Displayed to any User Trying to Delete a Case they did not Create**

References

Bricks. (N.D) Login page #1. Available from: <https://sechow.com/bricks/docs/login-1.html> [Accessed 10 February 2022].

OWASP. (N.D) OWASP Top Ten. Available from: [https://owasp.org/www-project-top-ten/#](https://owasp.org/www-project-top-ten/) [Accessed 5 December 2021].

Pallets. (N.D) Design Decisions in Flask. Available from: <https://flask.palletsprojects.com/en/2.0.x/design/> [Accessed 17 January 2022].

University of Essex Online. (2021) Testing [Lecturecast]. SSDCS\_PCOM7E November 2021 Secure Software Development November 2021. University of Essex Online