



Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

H.Dip. Sci. – Computer Science

Enterprise Web Development – Security Assignment

Cathal Henchy – 20091405

3 May 2021

1. Threat Modelling using Misuse Cases

For this Section, I opted to use the web app that I developed for the ICT Skill I module in 2020. This web app is a Gym app, and it satisfies *most* of the basic requirements of this exercise, in that it has multiple different user types, and a variety of different functions (all of which are accessible based on user type). What it does not have is functions that are multi-step, so for the purpose of this exercise, I will introduce multi-step functions as requirements, that will in turn increase the amount of Use Cases. Initially, I created a very exhaustive model, that returned an extremely elaborate and excessively busy diagram. I therefore trimmed it back a bit to provide a simpler, less confusing result. However, I have included the original elaborate diagrams, just to display my understanding of the subject matter.

a. System Requirements:

The Software system under consideration here is the 'Sraithín Gym Services' web app. This is a web app that is used by members of the Gym, and by trainers who work at the Gym.

The web app provides the following functions for members:

- I. Input and storage of fitness assessments that can be tracked and compared with previously recorded fitness assessments so as to keep track of progress of gym work.
- II. Trend indicators based on weight change over time.
- III. Analysis of fitness assessments which return analytics, including BMI, BMI Category and Ideal Body Weight Status.
- IV. Access to Feedback on fitness assessments & analytics from the trainers.
- V. Access to other personal details (i.e., contact details, gender, height etc.).
- VI. The ability to edit personal details, and to delete fitness assessments & associated feedback.
- VII. Booking of a session with a specified Trainer on a specified day (this requirement is the additional – currently not provided for – multi-step function).

The web app provides the following functions for trainers:

- I. Access to the fitness assessments & analytics of all members.
- II. Provision of feedback on all assessments & analytics that is accessible to each corresponding member.
- III. The ability to delete membership of any member, including the deletion of all associated records (fitness assessments, personal details etc.).

b. Use Cases:

The benign Actors are as follows:

- I. Members, whose interactions with the system are outlined in Step a above.
- II. Trainers whose interactions with the system are outlined in Step a above.

The Use Cases are as follows:

- I. Members can register their details on the system.
- II. All Actors can log into the webapp by providing authentication credentials.
- III. Members can book a gym session. This Use Case includes requesting a specific Trainer and a specific day at the gym. Both 'request' Use Cases require approval from the chosen Trainer.
- IV. All Actors can cancel a pre-booked gym session.
- V. Members can edit their personal details.
- VI. Trainers can delete Members and their associated personal details from the system.
- VII. Members can upload their fitness assessments to the system, where they are stored.
- VIII. Members can delete any of their fitness assessments.
- IX. Trainers can access the fitness assessments of all members.
- X. Trainers can provide feedback on any fitness assessments, by leaving a comment, which in turn can be accessed by the corresponding member.

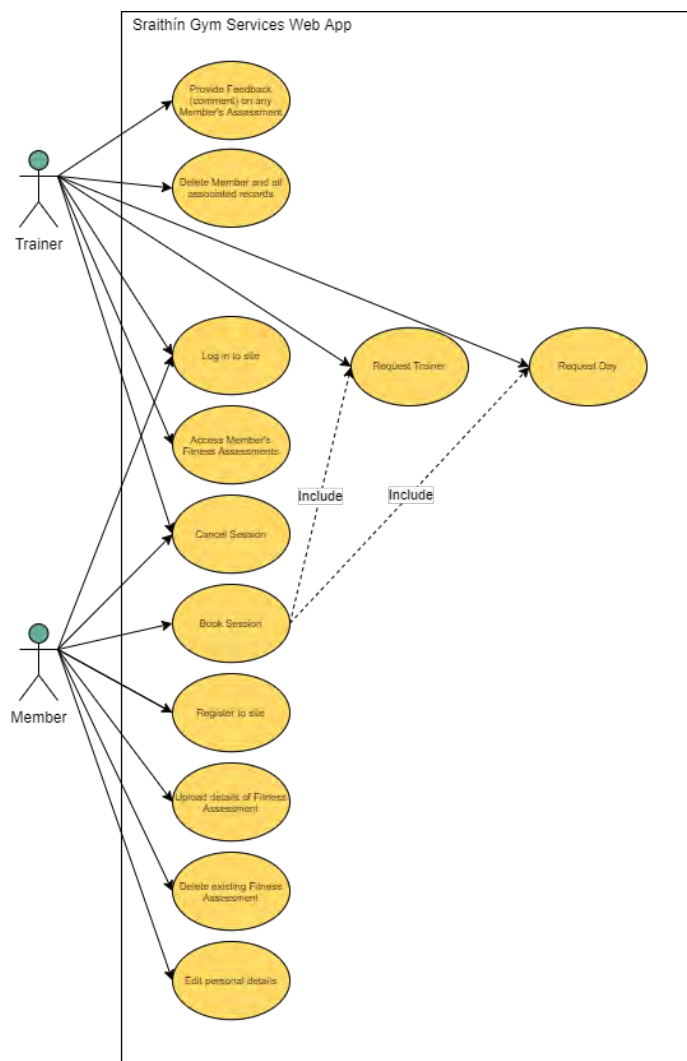


Figure 1: Use Cases

c. Misuse Cases:

The Misuse Actors are as follows:

- I. External Attackers.
- II. Internal Attackers, who are pre-registered Members. Once registered within the system, they have additional access to forms, and additional visibility of the system, languages and frameworks being used.

The Misuse Cases are as follows:

- I. External Attackers can flood the system by registering multiple members. This could lead to the system crashing.
- II. All Attackers can delete data contained within the system, through Injection in forms, such as when registering (External Attacker), or updating their personal details (Internal Attacker). They can also exploit weaknesses in the database, such as generically titled tables.
- III. All Attackers can input fake feedback to Members, which could lead to accusations of bullying etc. which could mean a loss of customers and hurt the reputation of the business.
- IV. All Attackers can reveal or steal Member's data (personal details, fitness assessments, feedback etc.), for example through Session Fixation Attack).
- V. All Attackers can access Member/Trainer authentication credentials, for example through cookie theft, or by exploiting old cryptographical technologies.
- VI. All Attackers can input malicious JavaScript into forms (as outlined in Part II above).

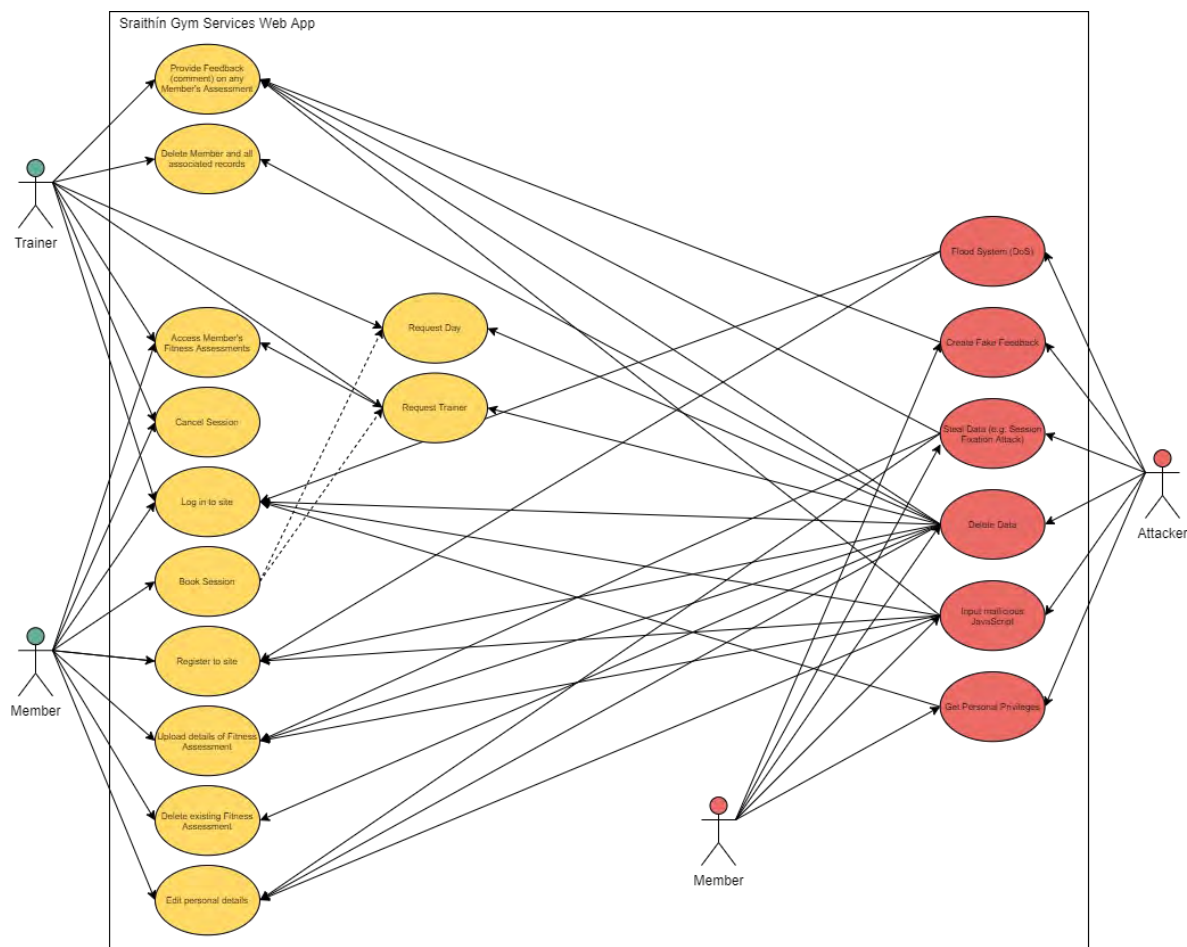


Figure 2: Misuse Cases

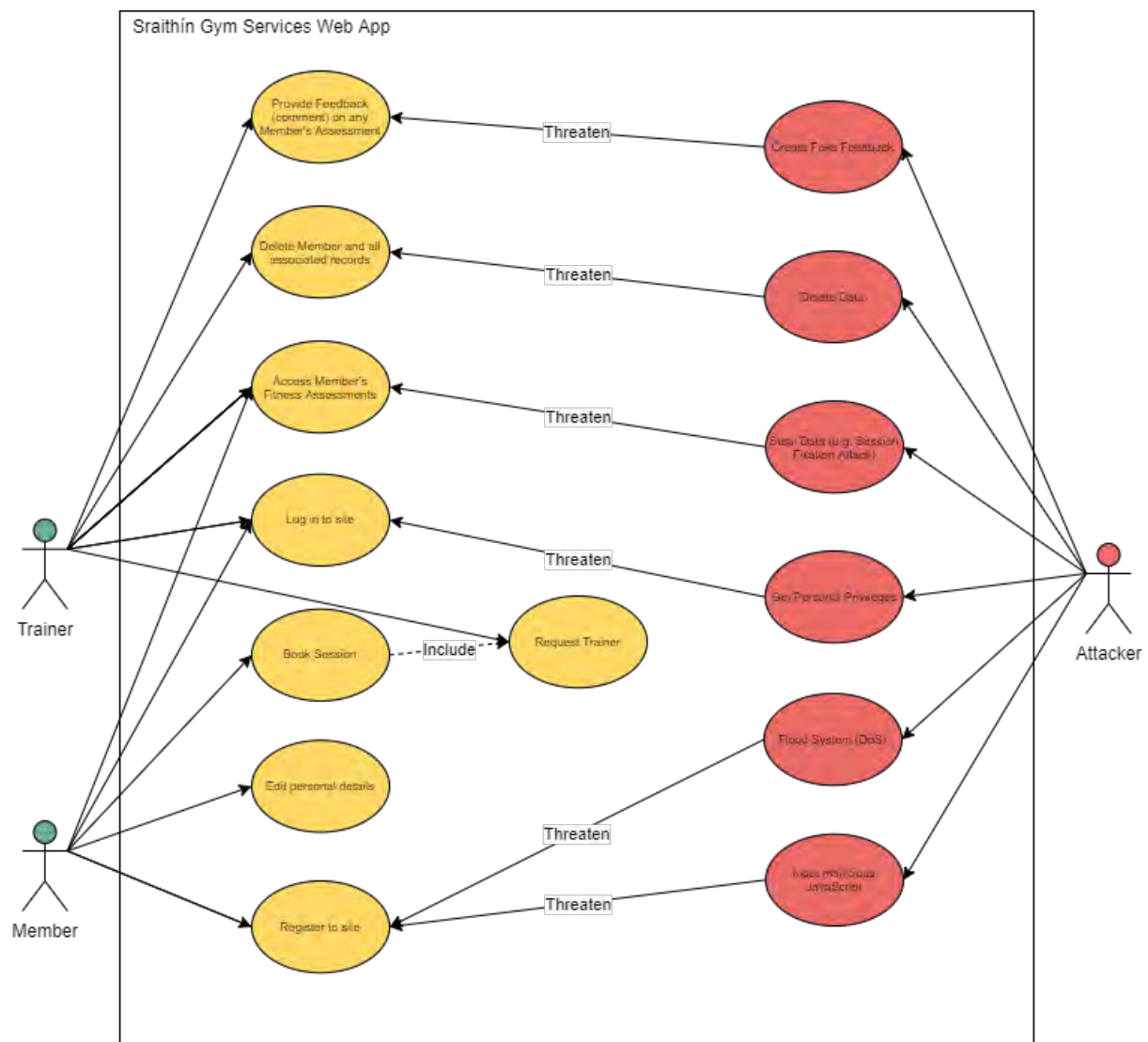


Figure 3: Misuse Cases (Concise Version)

d. Threat Mitigation:

The Mitigation Actor is as follows:

- I. Administrator, who moderates all feedback provided by Trainers. Despite there being no public access to the feedback (comment) functionality, this Actor is employed in case a Trainer's authentication is compromised.

Note: this seems highly inappropriate, as a Moderator/Administrator should only be required for publicly received comments, and other mitigations would be employed in the case of a Trainer's account being hacked. But as I've already included this in my diagram, I'm not going to change it at this stage.

The Threat Mitigations are as follows:

- I. Appropriate validation of all input data for all forms. This can prevent the system from being flooded (i.e., by preventing the input of duplicate data where appropriate). It can also prevent the input of malicious JavaScript - for example, where appropriate it can ensure that only numeric characters can be inputted into forms.
- II. Appropriate sanitisation can be used for all forms, which can also prevent the input of malicious JavaScript into forms, by filtering out quotation marks etc.
- III. Best practises can be employed on all databases, for example table names can be coded, therefore preventing Attackers from guessing their name and thus deleting or stealing data contained therein.
- IV. Good session management techniques, practises and technologies can be adopted, thus preventing cookie theft, which could lead to Attackers stealing authentication credentials, and thus entering the system. Examples of good session management include using http-only cookies, having logout buttons on every screen and keeping, maintaining & reviewing authentication logs.
- V. Prevent Broken Access Control. This would prevent Members from seeing the data belonging to other Members, or from obtaining the authorisation of trainers. This could be enforced by adopting role-based authentication mechanisms, whereby the Member would have no access to inappropriate data, as opposed to inappropriate data being hidden from the Member.

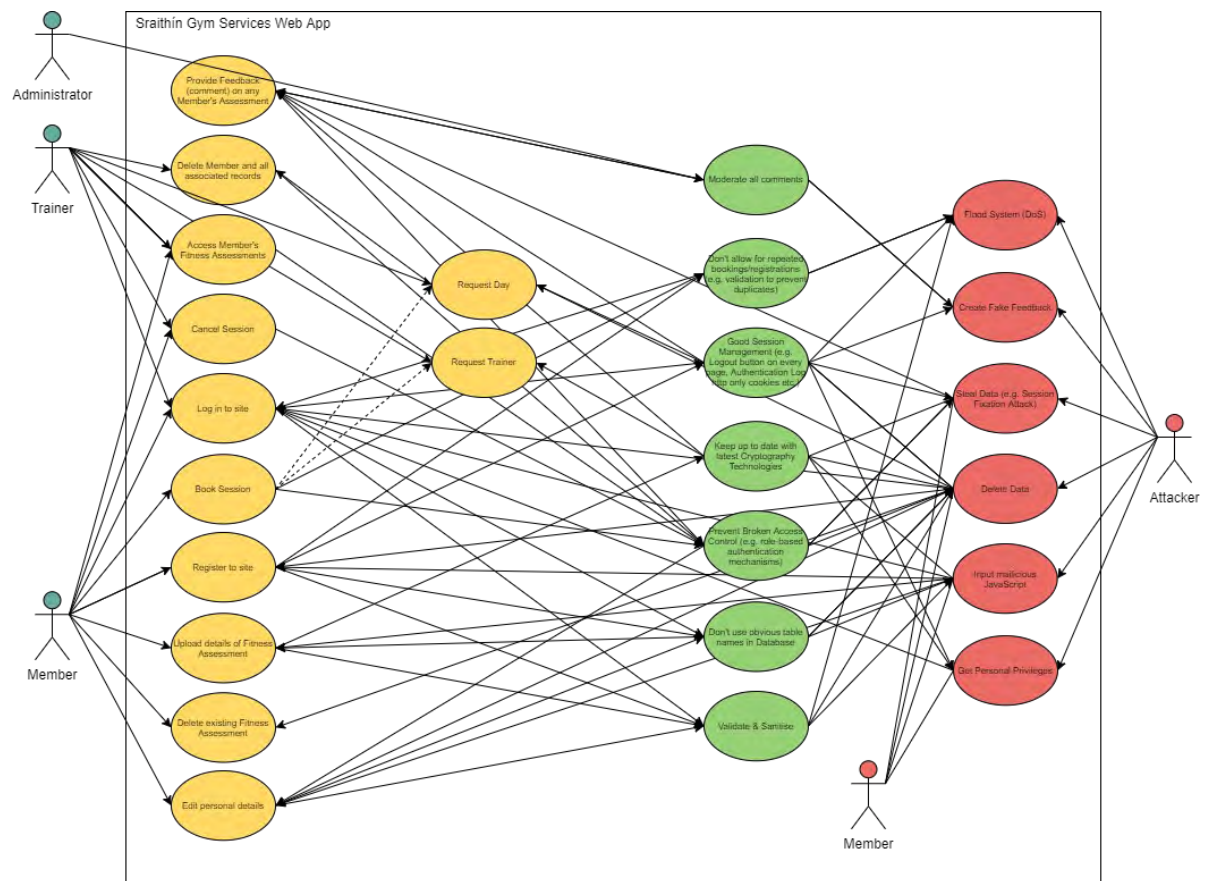


Figure 4: Mitigation of Threats

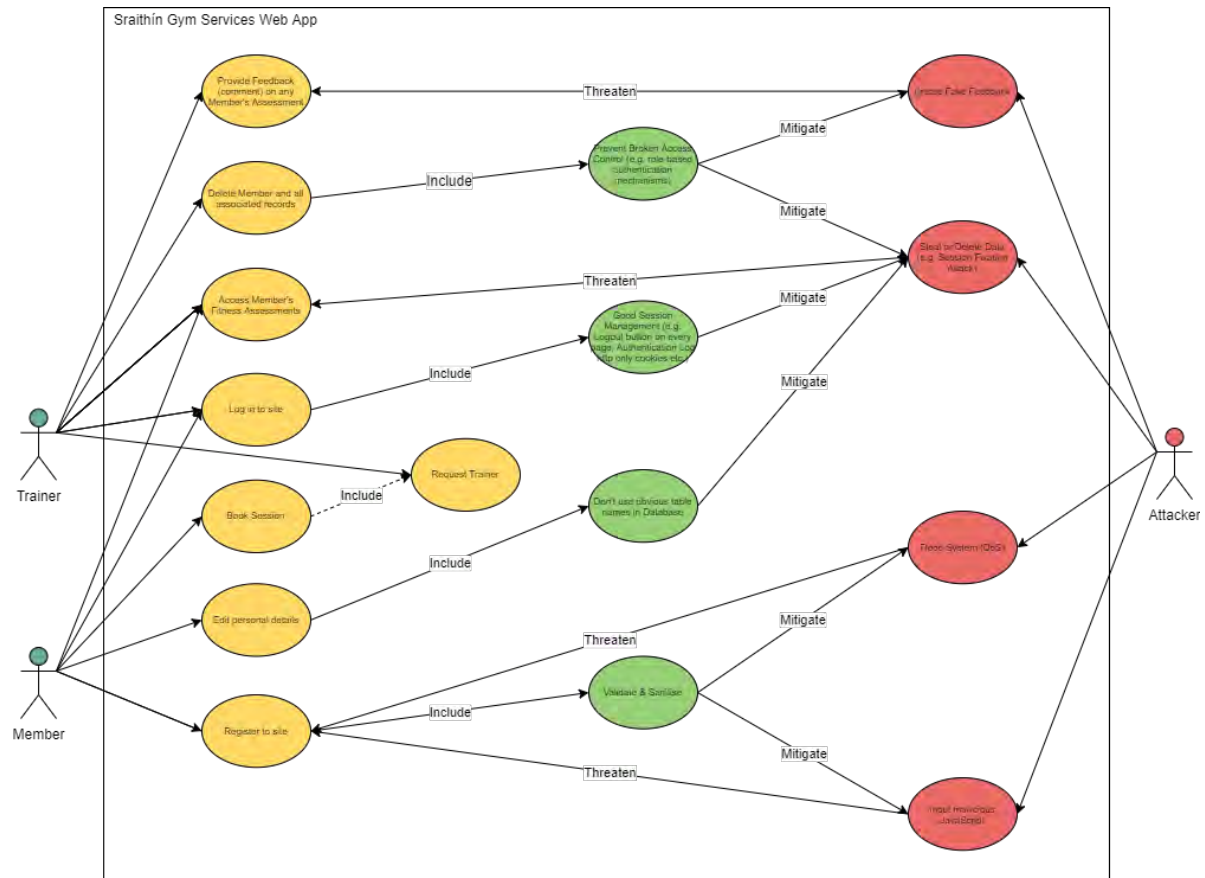


Figure 5: Mitigation of Threats (Concise Version)

2. Authentication and Encryption

a. Secure Web App with user authentication:

I wrote a simple web app (Secure Web App) to demonstrate the authentication process. This web app was created using Éamonn DeLeaster's Donation app (which is based on the hapi framework) from the EWD module as a starter template. It is intended to demonstrate how security authentication works when logging in or registering to a web application. To demonstrate secure registration of a new user with assignment of the appropriate privileges, it is connected to a mongodb database, and the db.js model has been configured to allow for persistence (i.e., not to overwrite the database with seeded data every time it is ran):

```
const dbData = await seeder.seed(data, options: { dropDatabase: false, dropCollections: false });
```

The password fields on both the 'login' and 'signup' pages are of type "password":

```
<span class="uk-form-icon" uk-icon="icon: lock"></span> <input class="uk-input uk-form-large" type="password" name="password">
```

b. Secure Web App Verification:

The authentication worked successfully for both registering new users, and for logging in (as both new users, or as existing users as seeded into the database).

The web app can be viewed and tested, by downloading from it git-hub repository here:

<https://github.com/cathalohinse/Secure-Web-App>

If the incorrect password is entered, the user is refused entry:

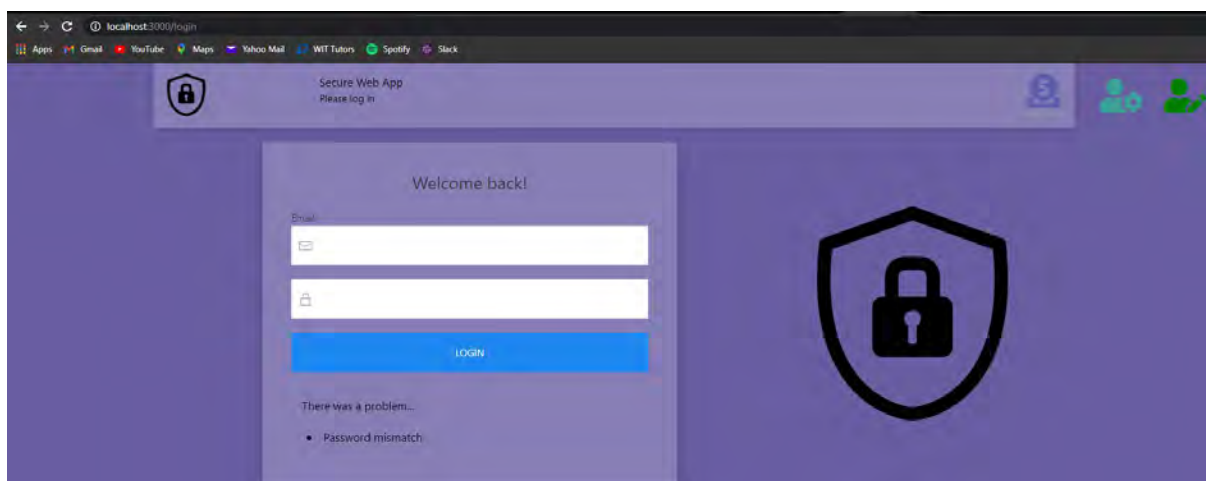


Figure 6: Authentication failed due to incorrect password being inputted

c. Request interception via Burp Suite proxy:

By using the Burp Suite (Community edition), I was able to intercept the http request made on my Secure Web App. This is a proxy server that any web browser can be configured to point any or all traffic to. As Chromium is already configured to do so, I chose this browser for this step. I opened the *Secure Web App* on Chromium, and then turned on the traffic interception function of the burp suite. This ensured that all http requests made on this browser were intercepted by the Burp Suite proxy server. I then logged on to the *Secure Web App* using existing credentials, and observed the http request interception on the Burp Suite application dashboard, which displayed the password being sent in the clear:

```
1 GET /login HTTP/1.1
2 Host: localhost:3000
3 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="90"
4 sec-ch-ua-mobile: ?0
5 Upgrade-Insecure-Requests: 1
6 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.85 Safari/537.36
7 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=0.9
8 Sec-Fetch-Site: same-origin
9 Sec-Fetch-Mode: navigate
10 Sec-Fetch-User: ?1
11 Sec-Fetch-Dest: document
12 Referer: http://localhost:3000/
13 Accept-Encoding: gzip, deflate
14 Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
15 Connection: close

1 POST /login HTTP/1.1
2 Host: localhost:3000
3 Content-Length: 38
4 Cache-Control: max-age=0
5 sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="90"
6 sec-ch-ua-mobile: ?0
7 Upgrade-Insecure-Requests: 1
8 Origin: http://localhost:3000
9 Content-Type: application/x-www-form-urlencoded
10 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.85 Safari/537.36
11 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=0.9
12 Sec-Fetch-Site: same-origin
13 Sec-Fetch-Mode: navigate
14 Sec-Fetch-User: ?1
15 Sec-Fetch-Dest: document
16 Referer: http://localhost:3000/login
17 Accept-Encoding: gzip, deflate
18 Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
19 Connection: close
20
21 email=DantheMan40@gmail.com&password=q
```

Figure 7: Burp Suite Proxy - password form request interception

d. TLS:

I set myself up as a CA (Certificate Authority) and issued a TLS Certificate to *Secure Web App* (which I preconfigured to that end). I created a private key, and I accessed the app on the browser. I had to acknowledge a warning message before I could access the app. This is because although the app had a certificate issued, the browser did not recognise the CA (i.e., the issuer of the certificate). This is obviously because I had just created the CA myself moments prior to this, thereby assigning the certificate myself.

```
Country Name (2 letter code) [AU]:IE
State or Province Name (full name) [Some-State]:Cork
Locality Name (eg, city) []:Cork
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dun Ora Certificate Authority Ltd.
Organizational Unit Name (eg, section) []:Issueing Dept.
Common Name (e.g. server FQDN or YOUR name) []:dunoraca.com
Email Address []:20091405@mail.wit.ie

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
compsys@compsys-virtualbox:~/EMD/security-assignment$ openssl x509 -req -trustout -signkey myca.key -sha256 -in myca.csr -out myca.crt
Signature ok
subject=C = IE, ST = Cork, L = Cork, O = Dun Ora Certificate Authority Ltd., OU = Issueing Dept., CN = dunoraca.com, emailAddress = 20091405@mail.wit.ie
Getting Private key
compsys@compsys-virtualbox:~/EMD/security-assignment$ openssl genrsa -out webserver.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
.....+++++
e is 65537 (0x010001)
compsys@compsys-virtualbox:~/EMD/security-assignment$ openssl req -new -key webserver.key -sha256 -out webserver.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IE
State or Province Name (full name) [Some-State]:Clare
Locality Name (eg, city) []:Clooney
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Norrie Henchys bar
Organizational Unit Name (eg, section) []:Marketing Dept.
Common Name (e.g. server FQDN or YOUR name) []:norriehenchy.ie
Email Address []:cathalohlne@gmail.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Figure 8: Configuration of CA & Web Server, and Creation of keys & certificate signing requests

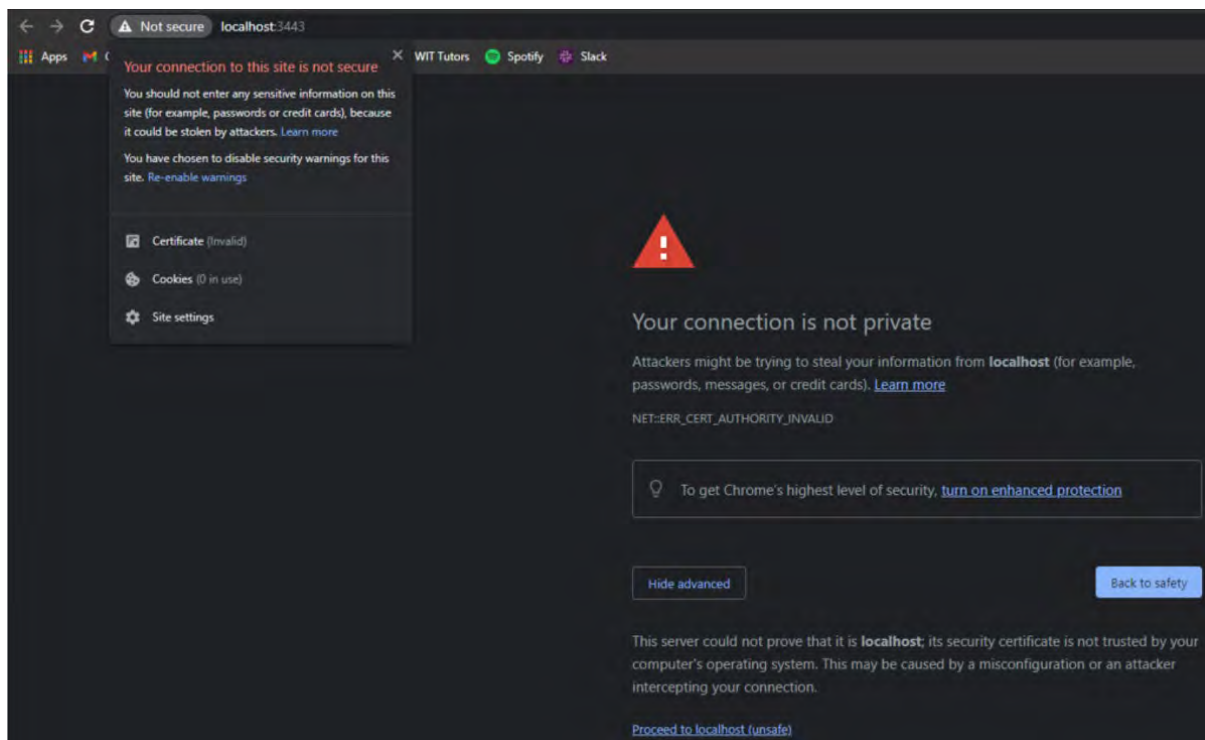


Figure 9: Browser Warning - No recognition of CA

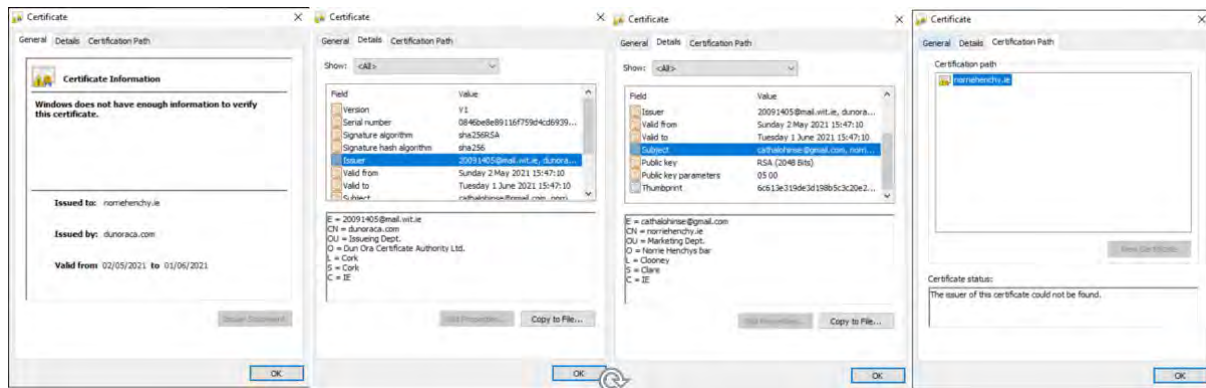


Figure 10: TLS Certificate

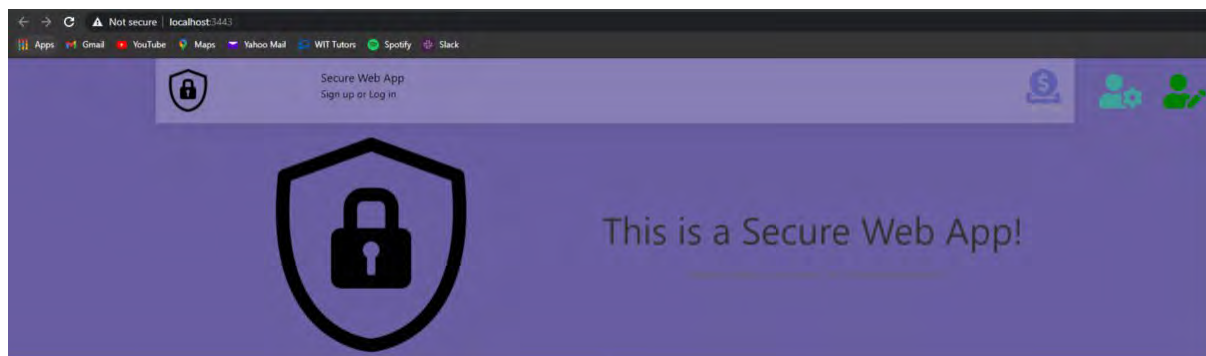


Figure 11: Browser warning persists in address bar

e. Hashing & Salting:

I installed the bcrypt module to *Secure Web App*. This enables the password hashing & salting functionality, which doesn't store any of the passwords in the database, but instead hashes them to encrypted values and stores these instead. In order to allow access to the app for existing data in the seeded database, I was obliged to re-configure the 'dropCollections' attribute in db.js:

```
const dbData = await seeder.seed(data, options: { dropDatabase: false, dropCollections: true });
```

Otherwise, this would only work for newly registered users.

Evidence of the application's acceptance of the non-hashed value of the passwords (despite storing the hashed value) is located in Figure 12, Figure 13 and Figure 14 below (Note the time-stamp, to verify authenticity of this effort):

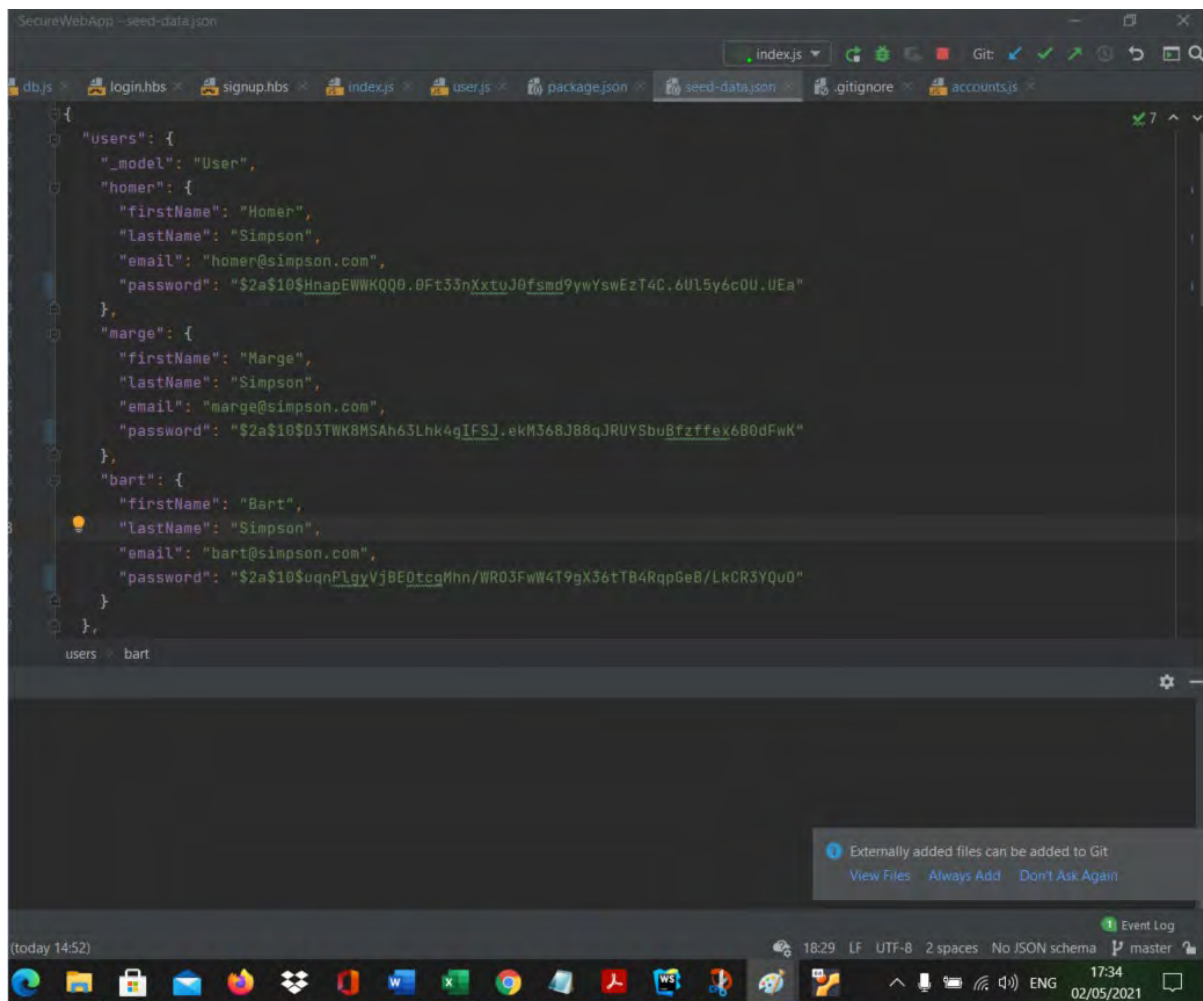


Figure 12: Database which contains only the hashed value of passwords

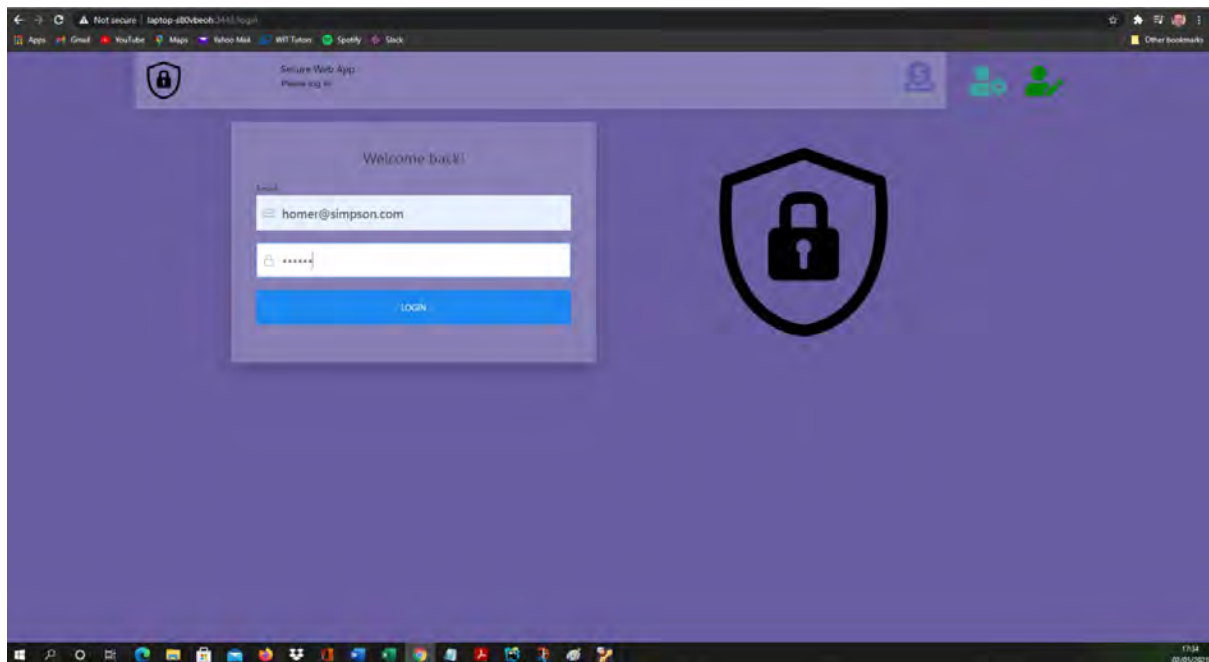


Figure 13: Logging into to app using password (not the hashed value)

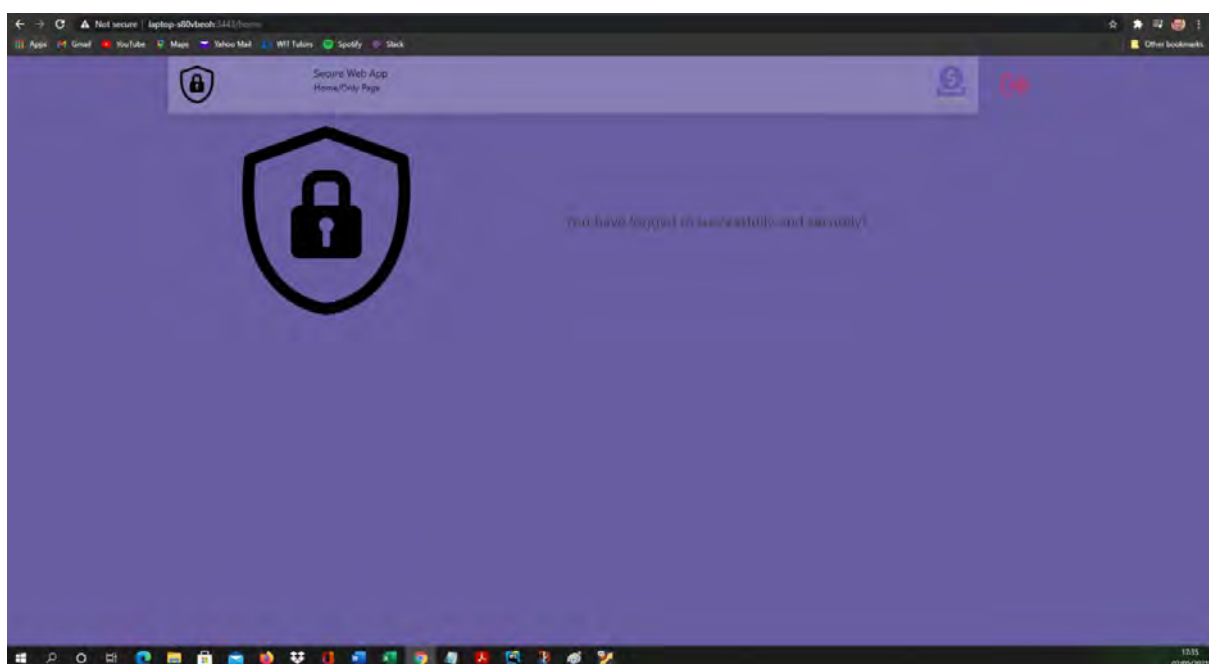


Figure 14: Password is accepted in this format

f. Another authentication/authorisation technique – OAuth:

OAuth is a standard that allows for secure communication between multiple server-side services. The services are given authorisation to communicate on behalf of the client, and thus have limited access to each other's resources.

In order to demonstrate this functionality (i.e. a client granting two separate servers authorisation to communicate), I created an application ('oauth-github') which is a js script that is configured to pull certain data from a git-hub account. Having granted *oauth-github* (which I actually wrote into *Secure Web App*, so it is essentially a part of this app) access to my git-hub account, I logged in to it via this app, and ran all configured commands, by visiting the various different routes that were programmed into it. This included logging out and logging back in again. (Note: logging out via the 'oauth-github' app is exactly just that – logging that app out of git-hub. I was still directly logged in to my account on the browser):



Figure 15: Initial creation of 'oauth-github' on Git-hub

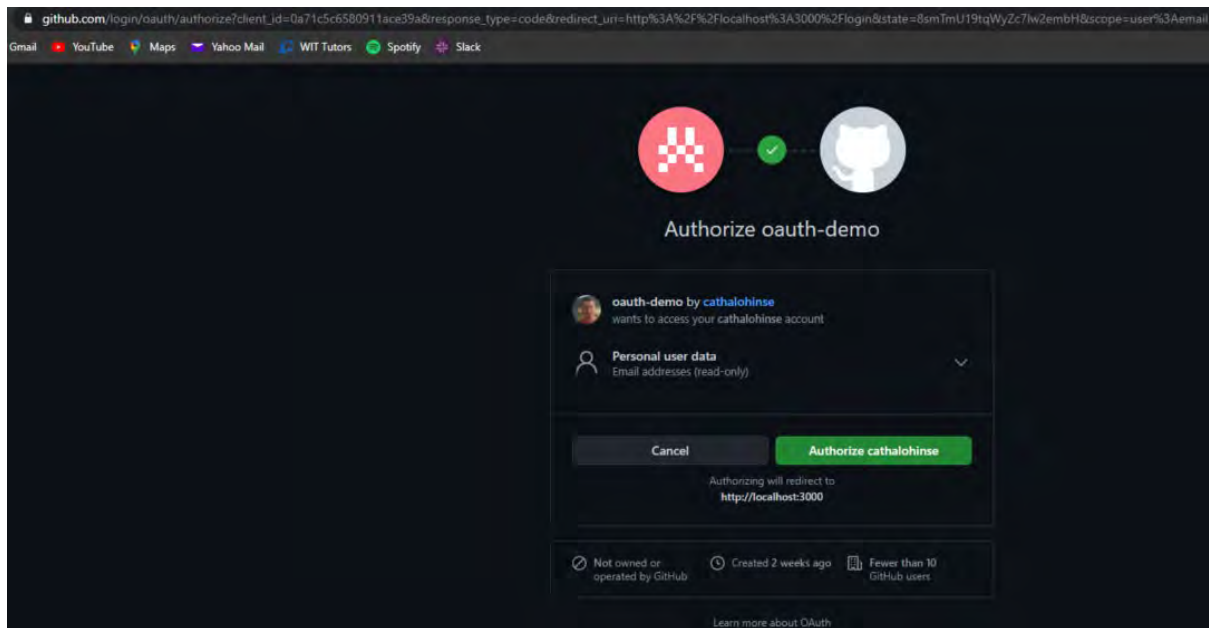


Figure 16: Logging into my Git-hub account via the 'oauth-github' app

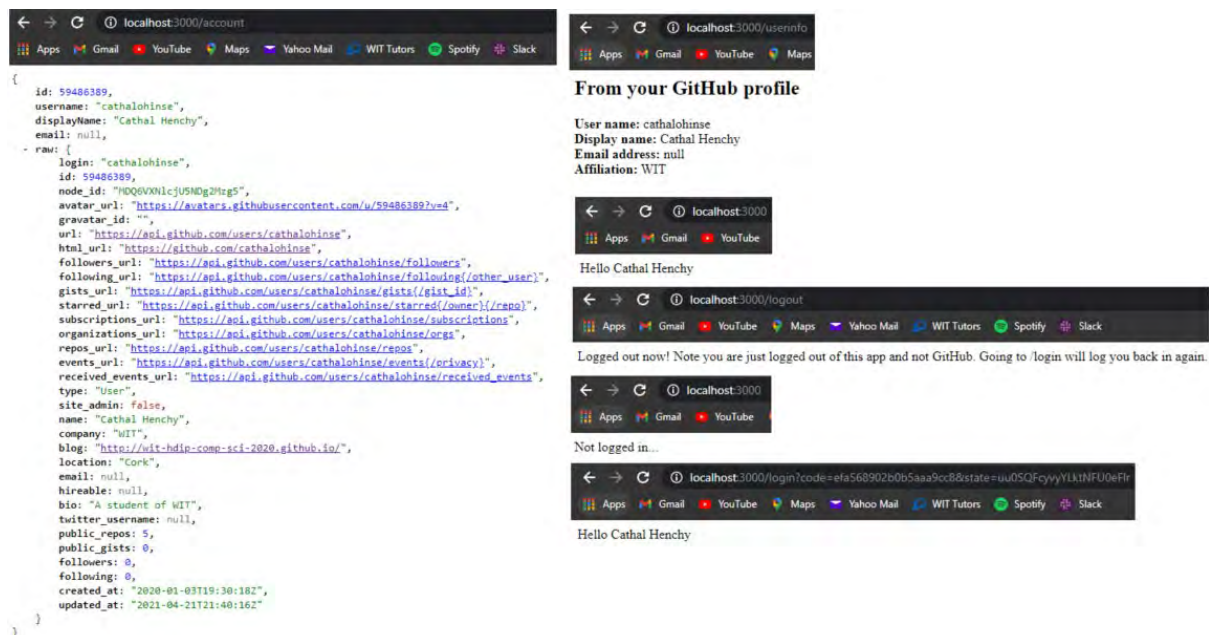


Figure 17: Series of commands being run on the 'oauth-github' app, including logging out, attempting to run a command while logged out, and then logging back in again

3. Architectural Diagram

a. Proxy brokered 'Meet-in-the-middle' attack:

In order to demonstrate how a proxy can carry out a 'meet-in-the-middle' attack, I used the *Secure Web App* as an example. I ran it on a browser that was configured to direct all traffic to a proxy server (the Chromium web browser and the Burp Suite proxy server, as per Part 2 Step c above), and I registered a new user. I ran the interception functionality so that all http requests would be intercepted by the server, so when the registration request was made, I had full visibility of the registration details. I then proceeded to change the password that had been registered on the client side. When I then attempted to log in using the client's originally requested password, I was denied entry, but when I tried the password that I switched this for on the Burp Suite dashboard, I was granted access. So, if this were a real-life attack, the user would have had their data stolen, and the hacker would then have had full access to the user's account (with the user losing all access). Of course, this isn't an ideal representation of a real life account being robbed, because it is a brand new account that the user hasn't even accessed yet, so would therefore be of very little use to the hacker. Nevertheless, this is how such an attack would be carried out.

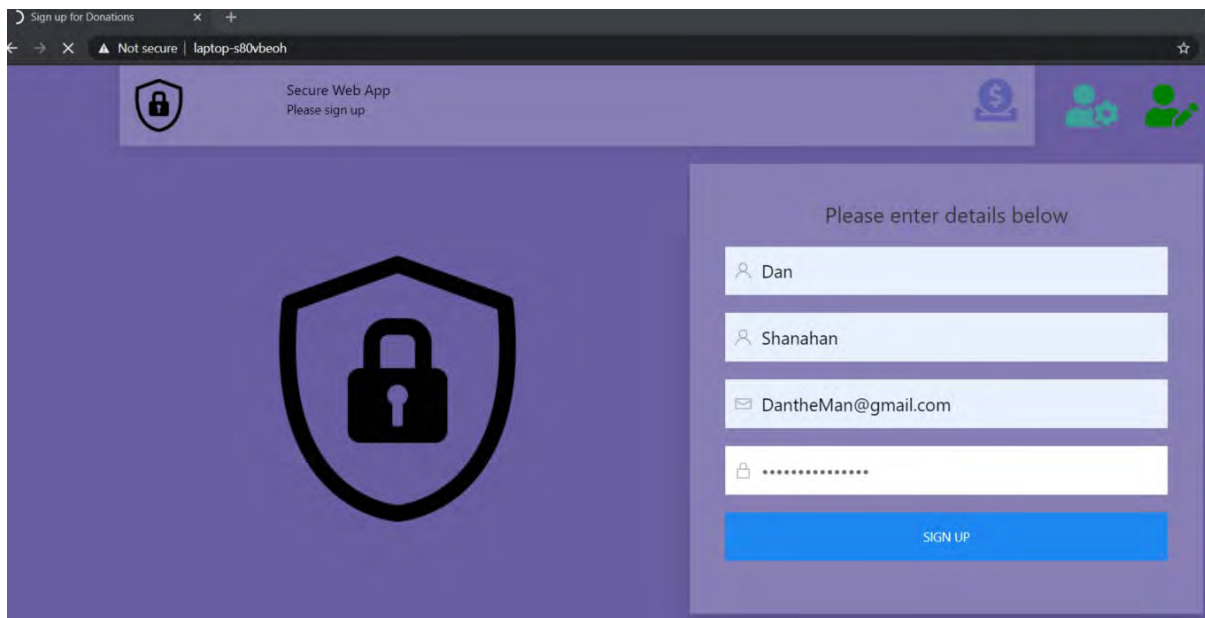


Figure 18: Initial user registration on client side

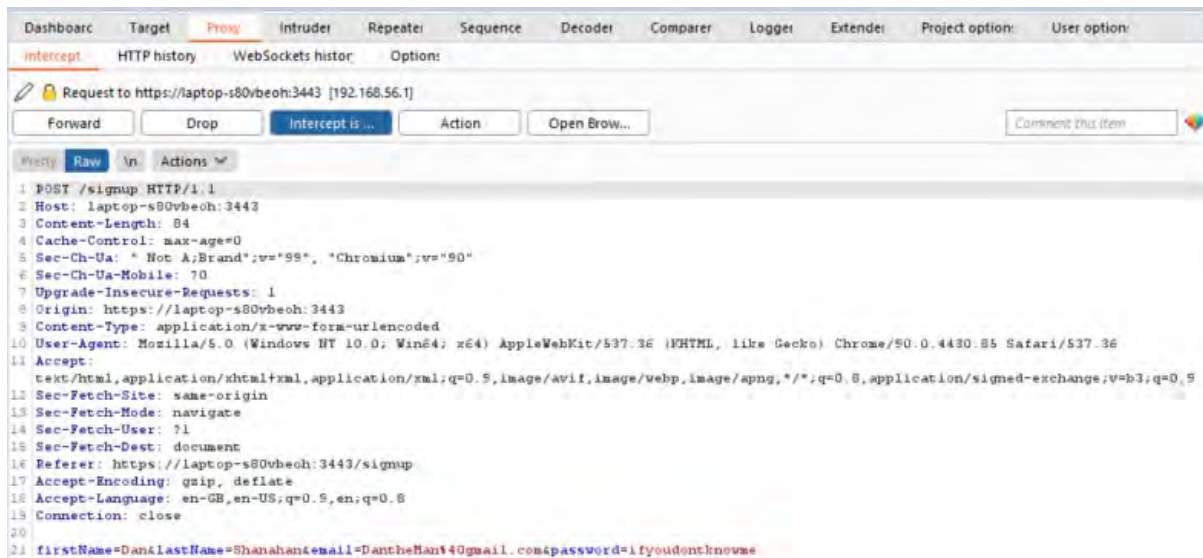


Figure 19: Proxy has intercepted this registration request

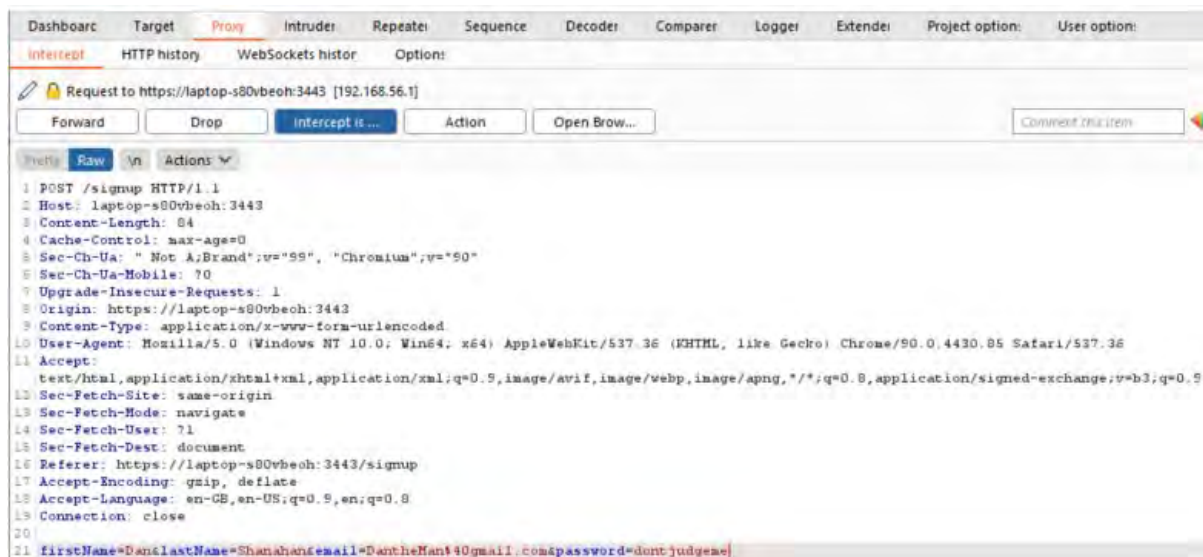


Figure 20: The proxy then modifies the password that was inputted by the user

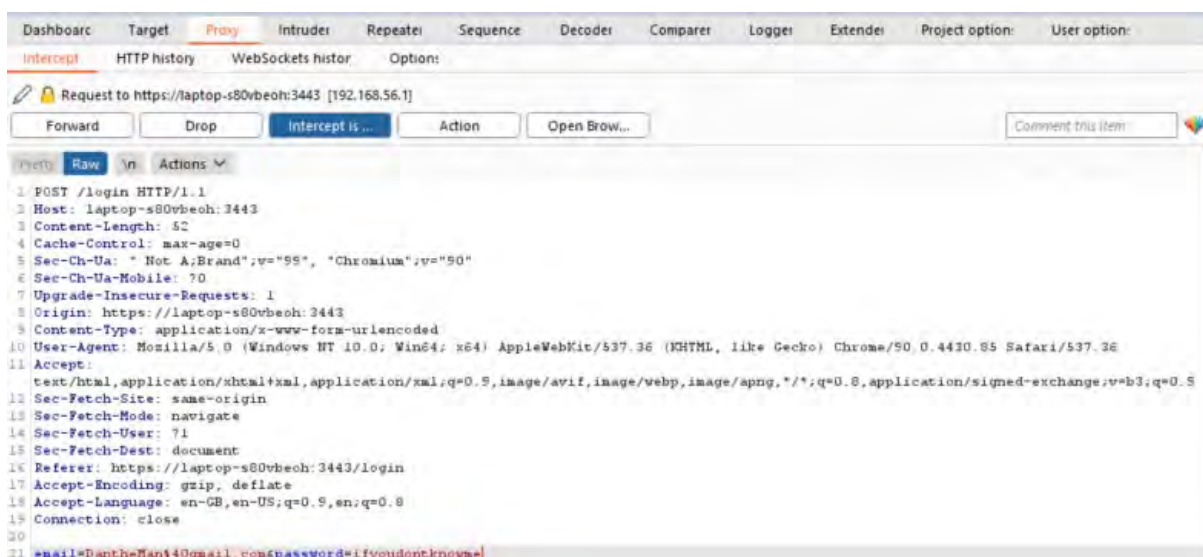


Figure 21: The user then attempts to log in using the password that they inputted at registration

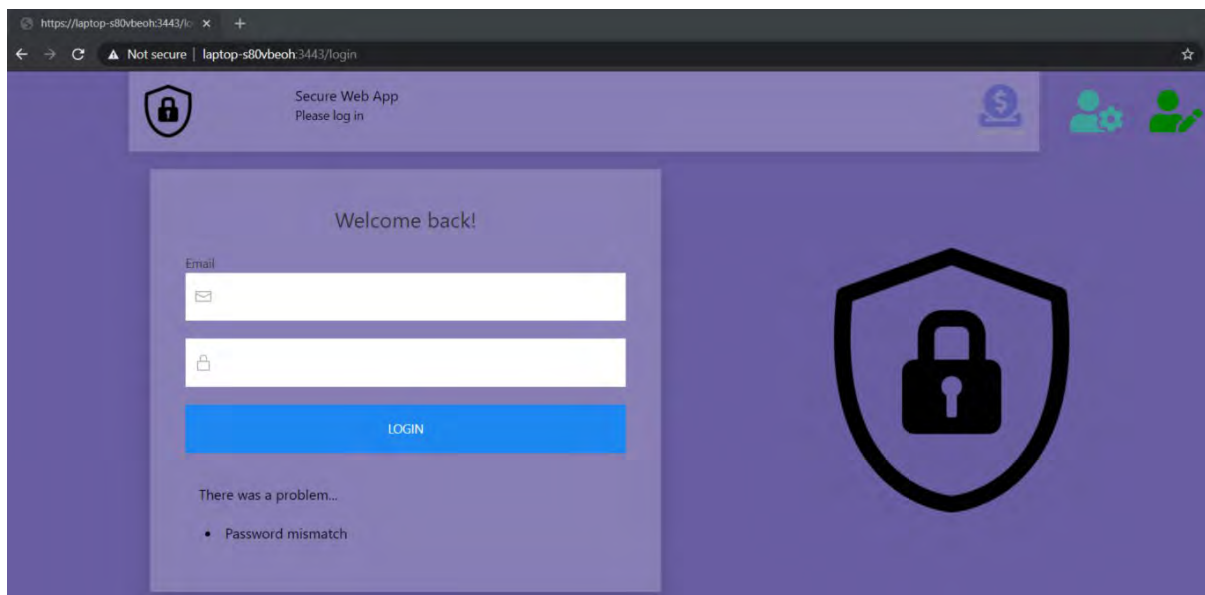


Figure 22: The user is denied access using this password

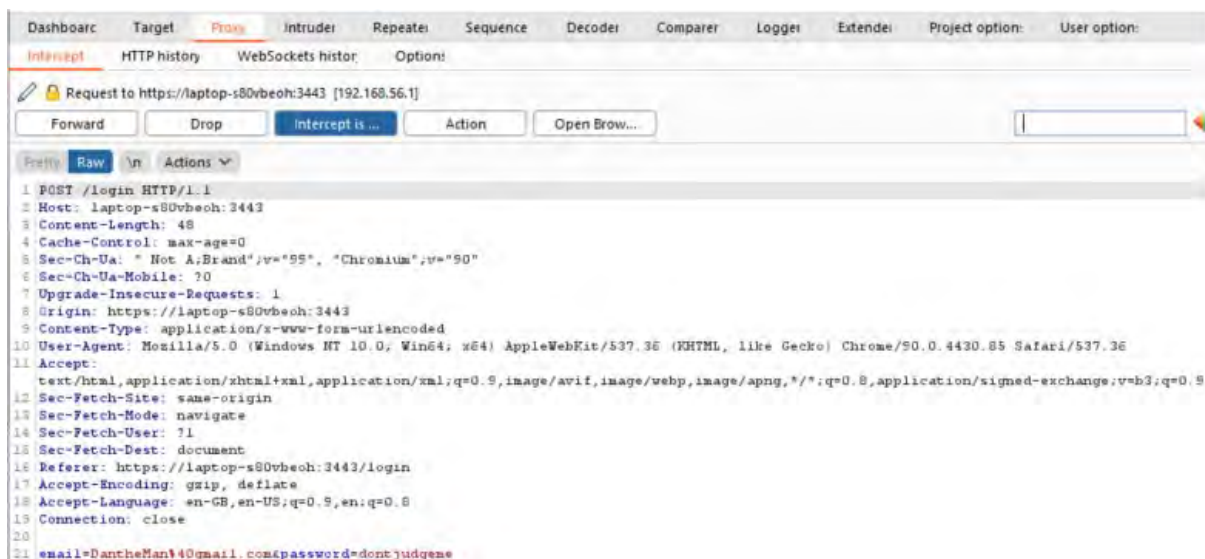


Figure 23: The user then attempts to log in using the password that the proxy inputted at the signup interception

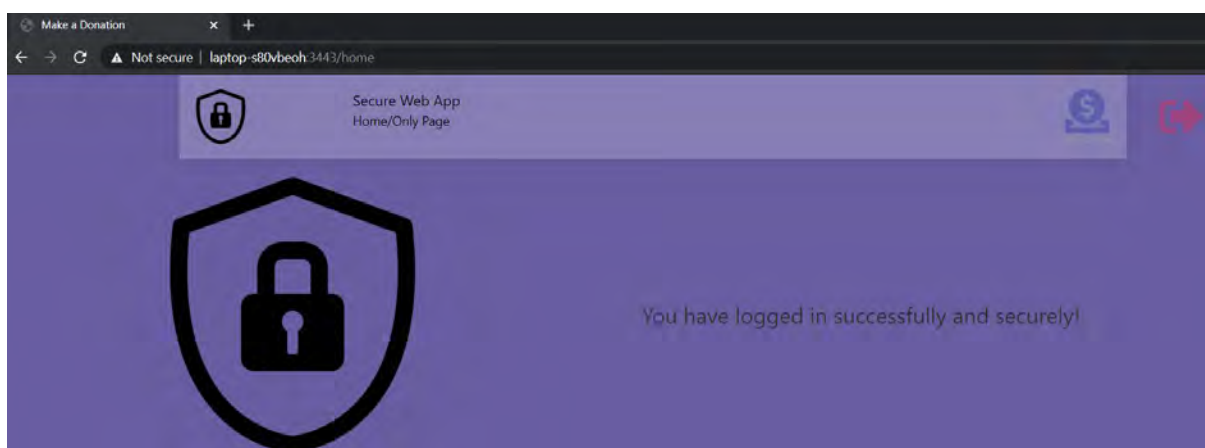


Figure 24: The user is granted access using the proxy modified password

b. Bypassing authentication via cookie theft:

To demonstrate how authentication can be bypassed (thus stealing one's identity) by stealing a cookie, I used the *aiteanna-speisiula* app, which was the app that I did for Assignment 1 of EWD. I quite simple took the cookie from a page of this app that contained sensitive information, I then used a proxy to intercept a request to visit the same page in the appropriate proxy oriented browser (Chromium and Burp Suite, as before), and modified the cookie on this request to match that which I took from the page with the sensitive information, and I was then given full access to that page and all data contained therein – no authentication required!

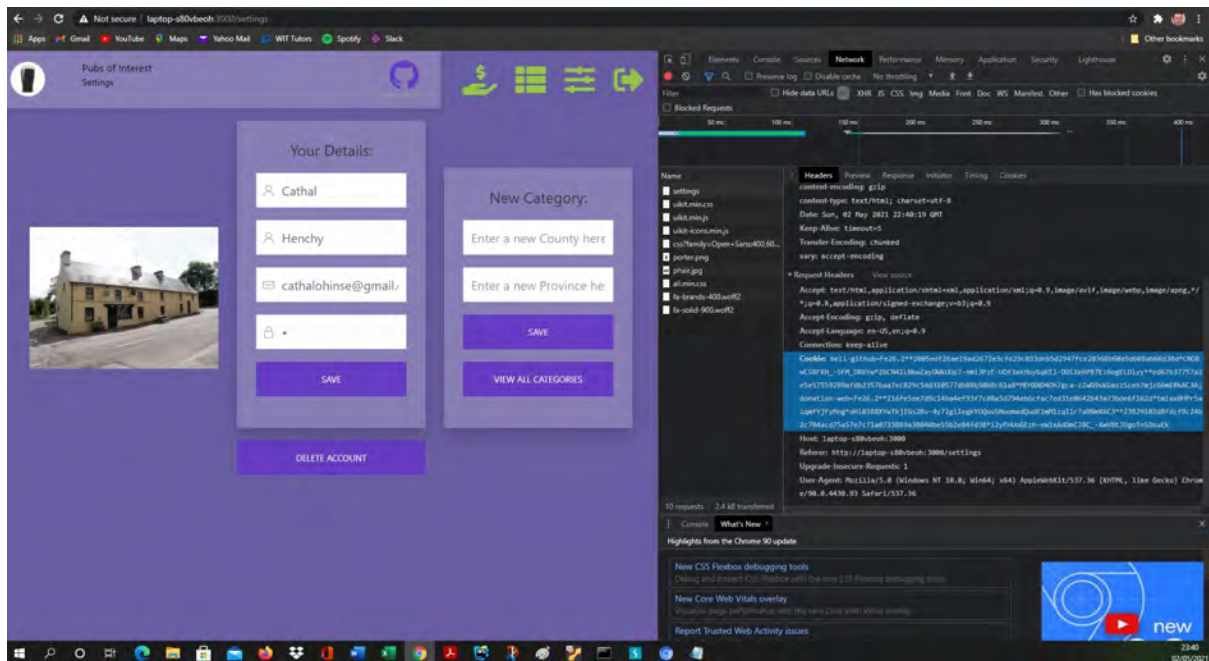


Figure 25: Copying the cookie from the logged in webpage on the app. Note all the sensitive information contained therein

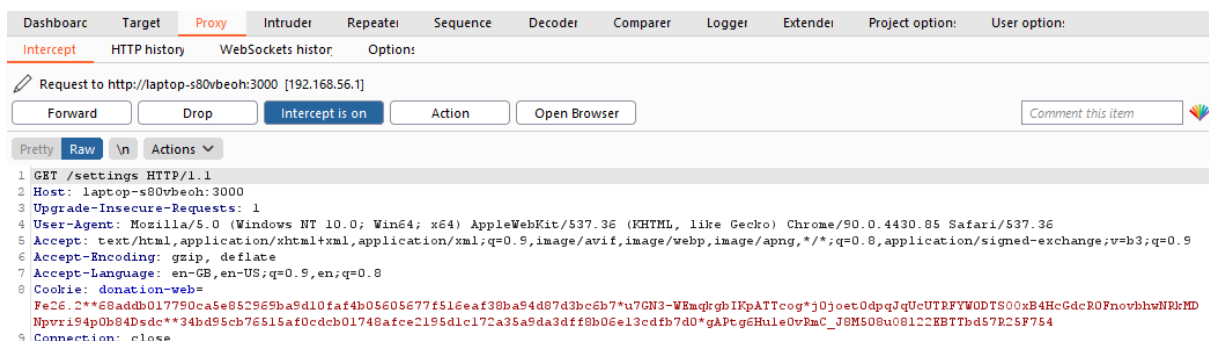


Figure 26: The proxy interception of the same page request in the Chromium browser

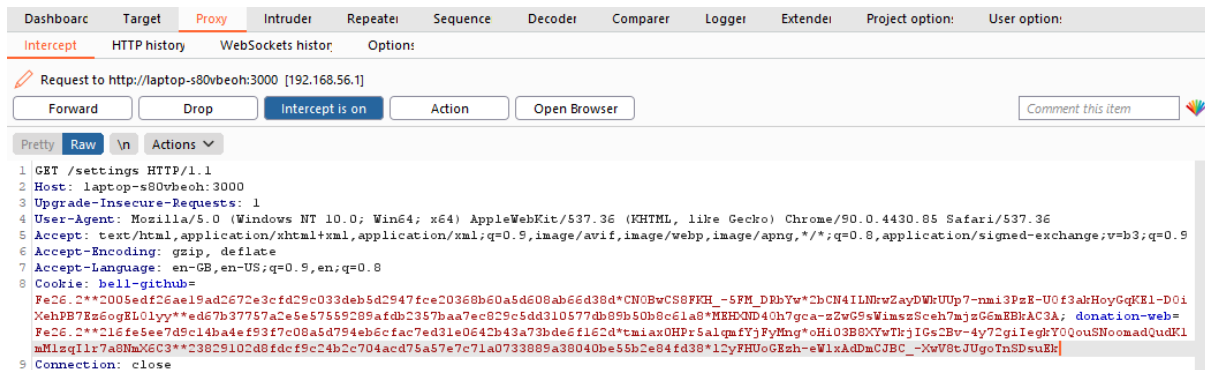


Figure 27: Modification of cookie in the page request

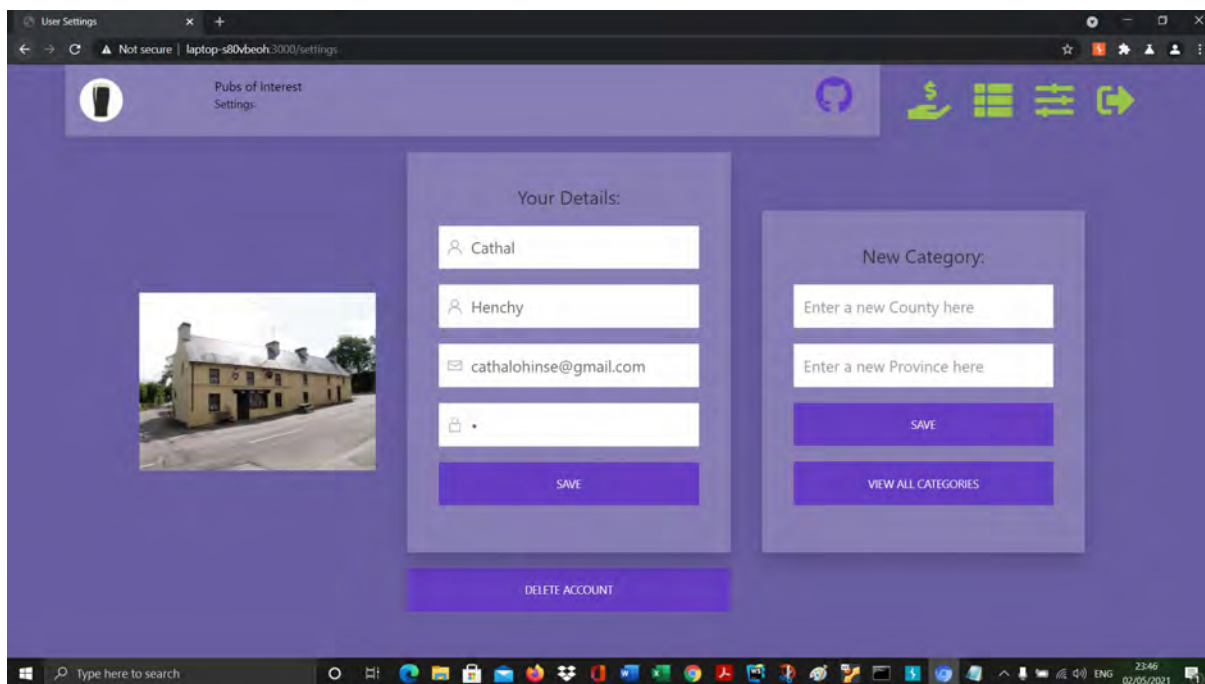


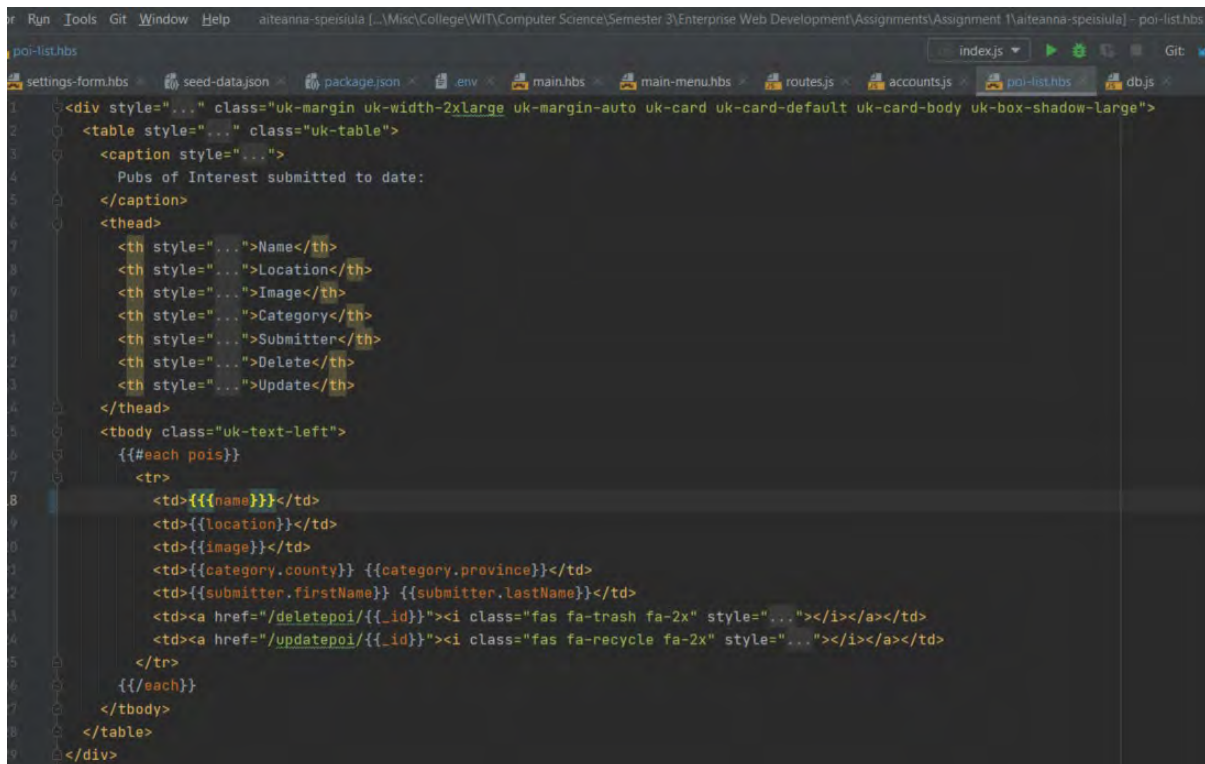
Figure 28: Full access granted - no authentication required!

c. XSS:

I demonstrated both XSS attack types using the *aiteanna-speisiula* web app:

i. Form Data

I first demonstrated the risk of XSS vulnerabilities through form attacks. I did this by intentionally making a web app vulnerable to such an attack (with the addition of additional braces on a *.hbs* view file). I was then able to inject javascript into the form field, and without the protections that had hitherto been in place, this javascript is executed every time any user visits this page of the app:



```
1 <div style="..." class="uk-margin uk-width-2xlarge uk-margin-auto uk-card uk-card-default uk-card-body uk-box-shadow-large">
2   <table style="..." class="uk-table">
3     <caption style="...">
4       Pubs of Interest submitted to date:
5     </caption>
6     <thead>
7       <th style="...">Name</th>
8       <th style="...">Location</th>
9       <th style="...">Image</th>
10      <th style="...">Category</th>
11      <th style="...">Submitter</th>
12      <th style="...">Delete</th>
13      <th style="...">Update</th>
14    </thead>
15    <tbody class="uk-text-left">
16      {{#each pois}}
17        <tr>
18          <td>{{name}}</td>
19          <td>{{location}}</td>
20          <td>{{image}}</td>
21          <td>{{category.county}} {{category.province}}</td>
22          <td>{{submitter.firstName}} {{submitter.lastName}}</td>
23          <td><a href="/deletepoi/{{_id}}"><i class="fas fa-trash fa-2x" style="..."></i></a></td>
24          <td><a href="/updatepoi/{{_id}}"><i class="fas fa-recycle fa-2x" style="..."></i></a></td>
25        </tr>
26      </tbody>
27    </table>
28  </div>
```

Figure 29: Reprogramming the app to introduce a vulnerability

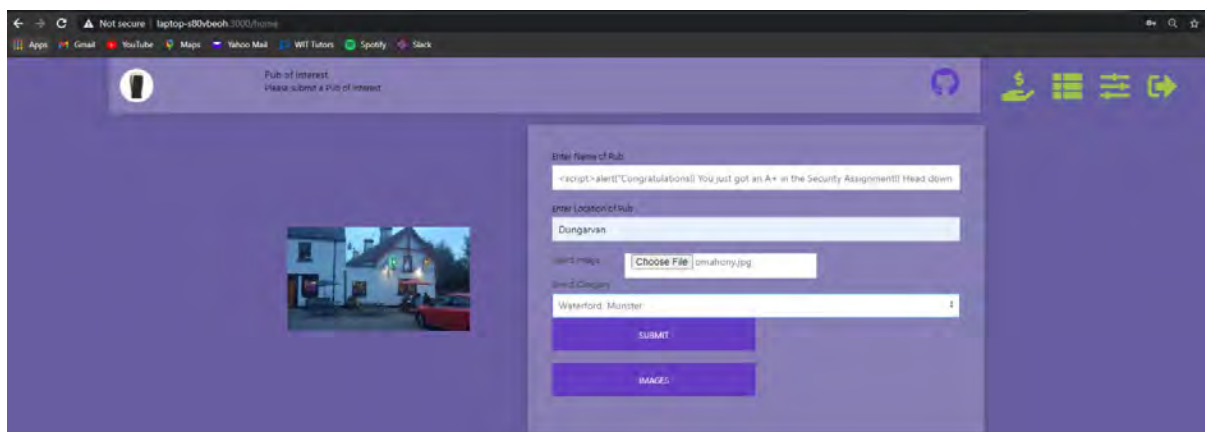


Figure 30: Inputting javascript into the form field

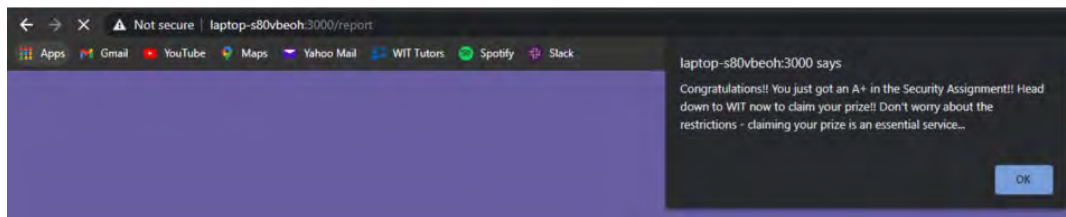


Figure 31: javascript is executed each time any user visits this page of the app. In this case a pop-up is triggered

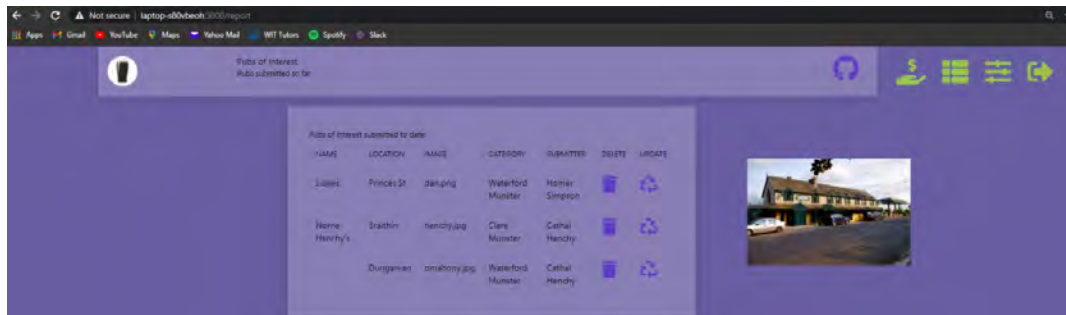


Figure 32: Because the input was executed as javascript, it is not actually saved as data

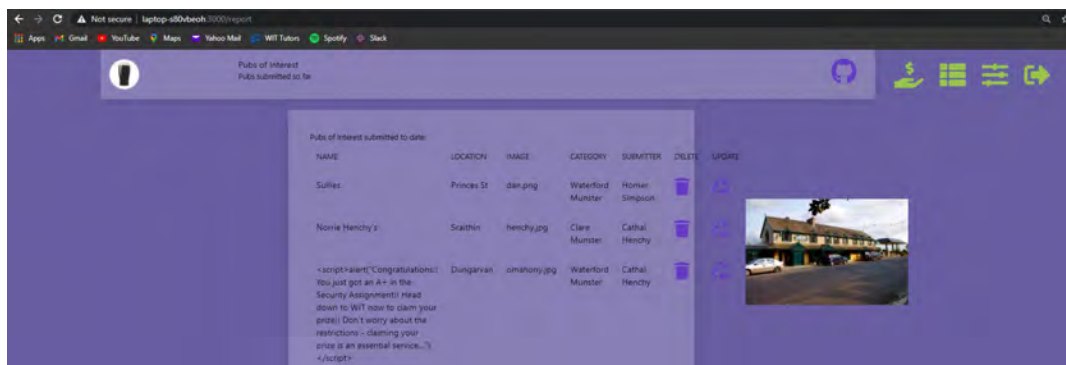


Figure 33: By reprogramming the app to remove the additional curly braces, the input is not recognised as javascript, but as a string, and it is saved as such

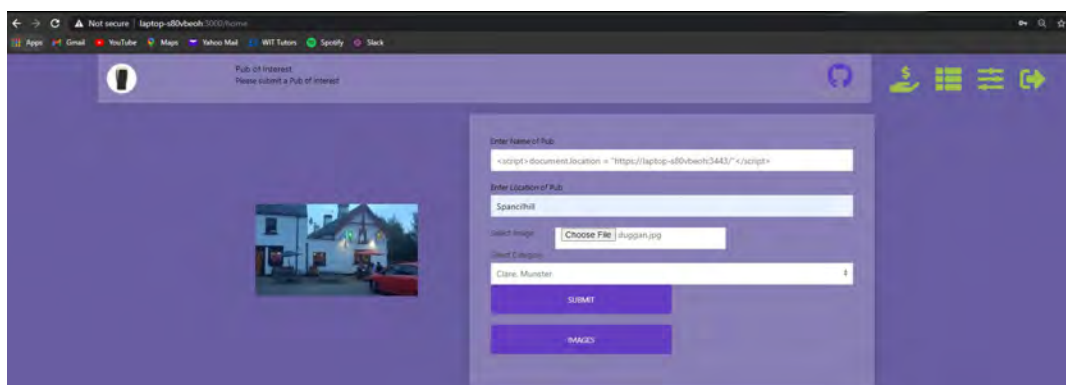


Figure 34: javascript can be injected to redirect the user to external website

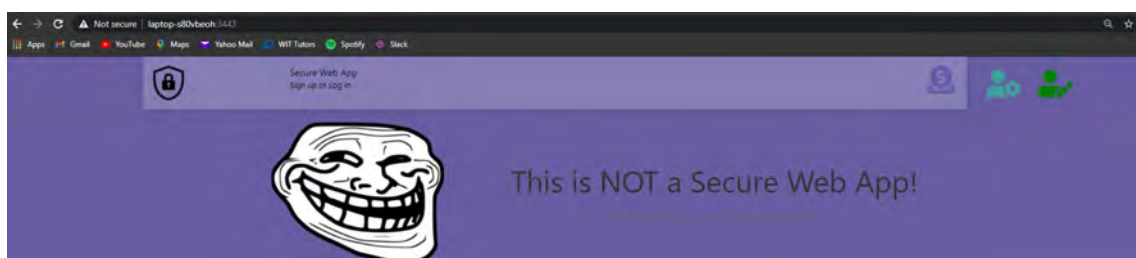
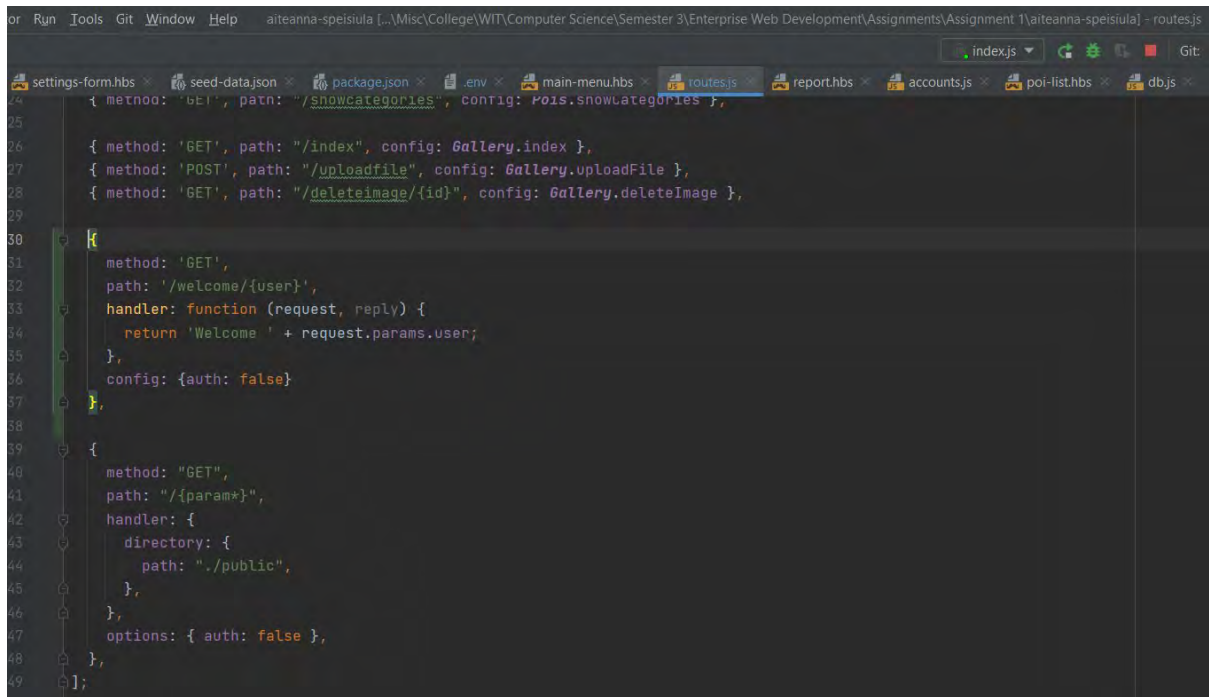


Figure 35: This time, by visiting this page the user has been re-directed to an external website

ii. URL

The use of a URL vulnerability in XSS is very easily demonstrated by introducing a parameterised route into the app. As any variable can be introduced here, it is possible to inject javascript (provided URL encoding is used appropriately, for example, html characters like / are required in html end tags but are interpreted in URL as directory delimiters. Therefore, URL encoding can mask them from the URL compiler by replacing them with '%2F') into the URL in the address bar of the browser, thus executing javascript on the website:



```
24 { method: 'GET', path: "/index", config: Pcis.showCategories },
25
26 { method: 'GET', path: "/index", config: Gallery.index },
27 { method: 'POST', path: "/uploadfile", config: Gallery.uploadFile },
28 { method: 'GET', path: "/deleteimage/{id}", config: Gallery.deleteImage },
29
30 {
31   method: 'GET',
32   path: '/welcome/{user}',
33   handler: function (request, reply) {
34     return 'Welcome ' + request.params.user;
35   },
36   config: { auth: false }
37 },
38
39 {
40   method: "GET",
41   path: "/{param*}",
42   handler: {
43     directory: {
44       path: "./public",
45     },
46   },
47   options: { auth: false },
48 },
49 ];
```

Figure 36: Reprogramming the app to introduce a vulnerability

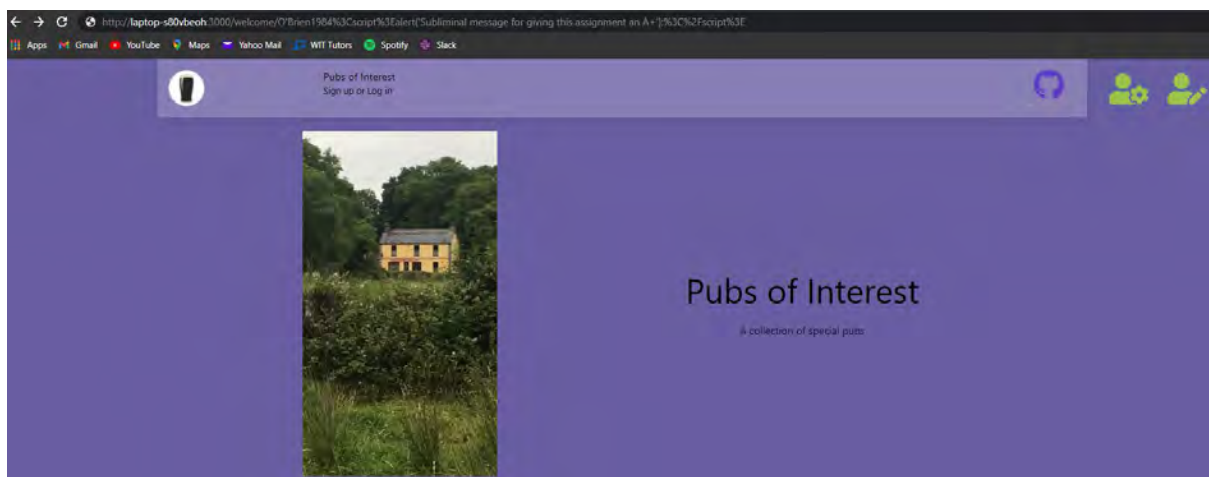


Figure 37: Inputting javascript into the URL

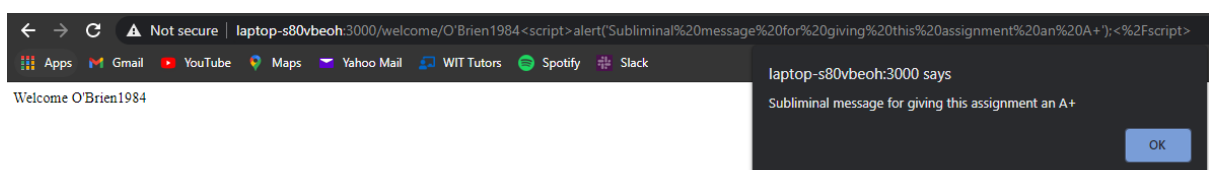


Figure 38: javascript is executed when this URL is entered

- d. Limitations of HTML form protections in security attacks:
HTML restrictions are merely client side restrictions that are confined to HTML only. They offer no real security, as they can very easily be side-stepped due to the fact that they do not look beyond HTML at all. I have demonstrated that here using a proxy server (Chromium used in tandem with Burp Suite as before) to side-step all HTML restrictions in a couple of examples (using the *aiteanna-speisiula* app):

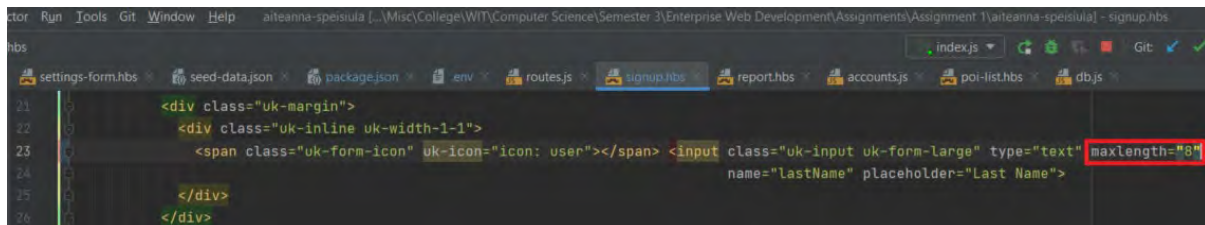


Figure 39: HTML restriction capping the string length of the form field input

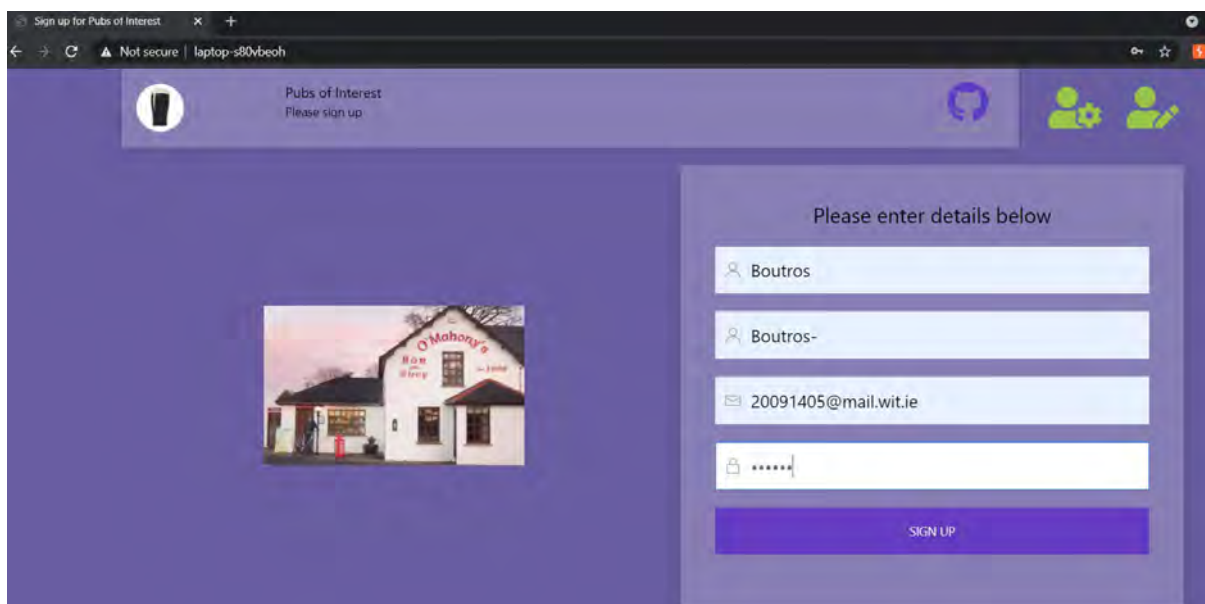


Figure 40: Attempt to input 'Boutros-Ghali' as a surname is unsuccessful due to HTML restriction

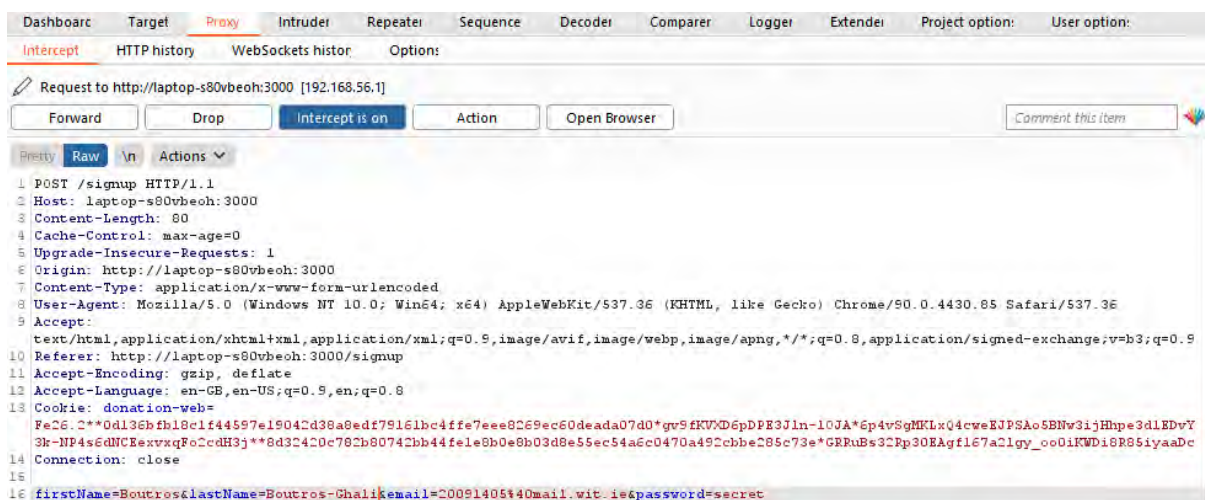


Figure 41: Data is modified to desired outcome by proxy interception

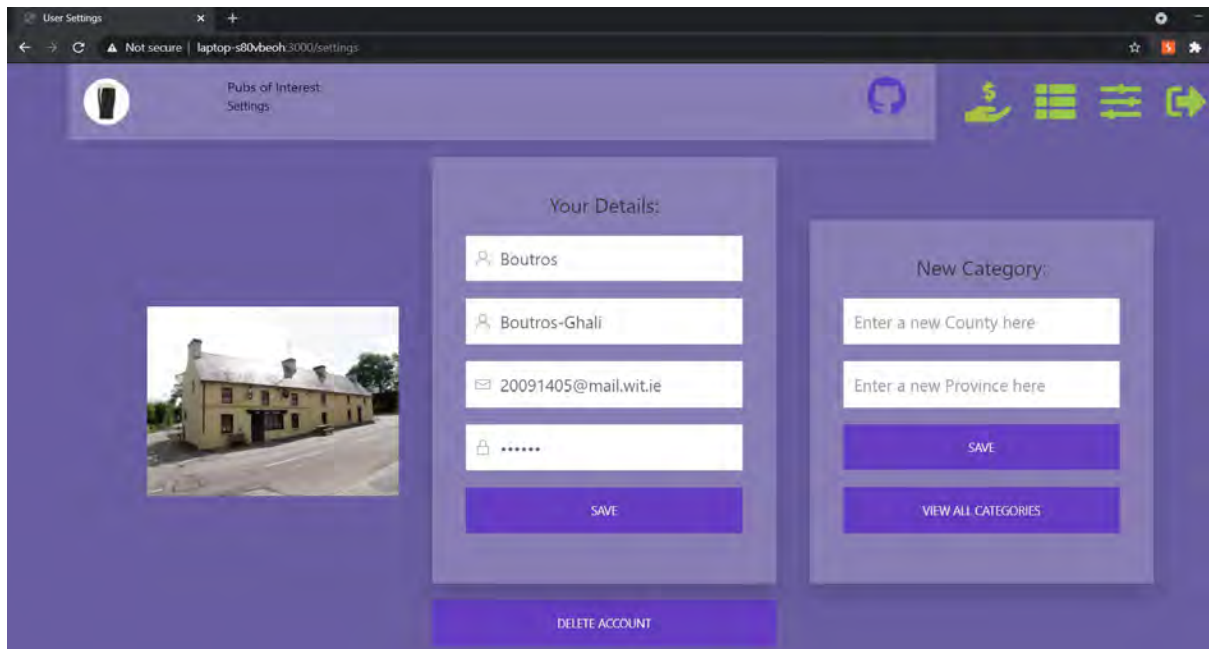


Figure 42: Despite the HTML restriction preventing the user from manually inputting a surname of desired length in the form field, it was easily manipulated using the proxy server

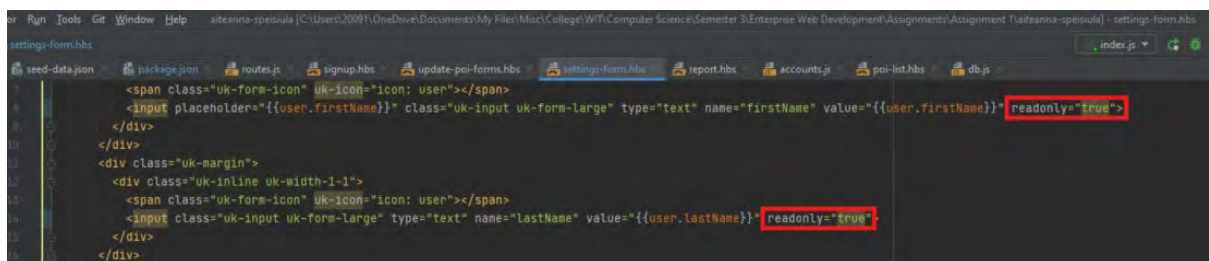


Figure 43: HTML restrictions preventing the content of the form fields from being edited

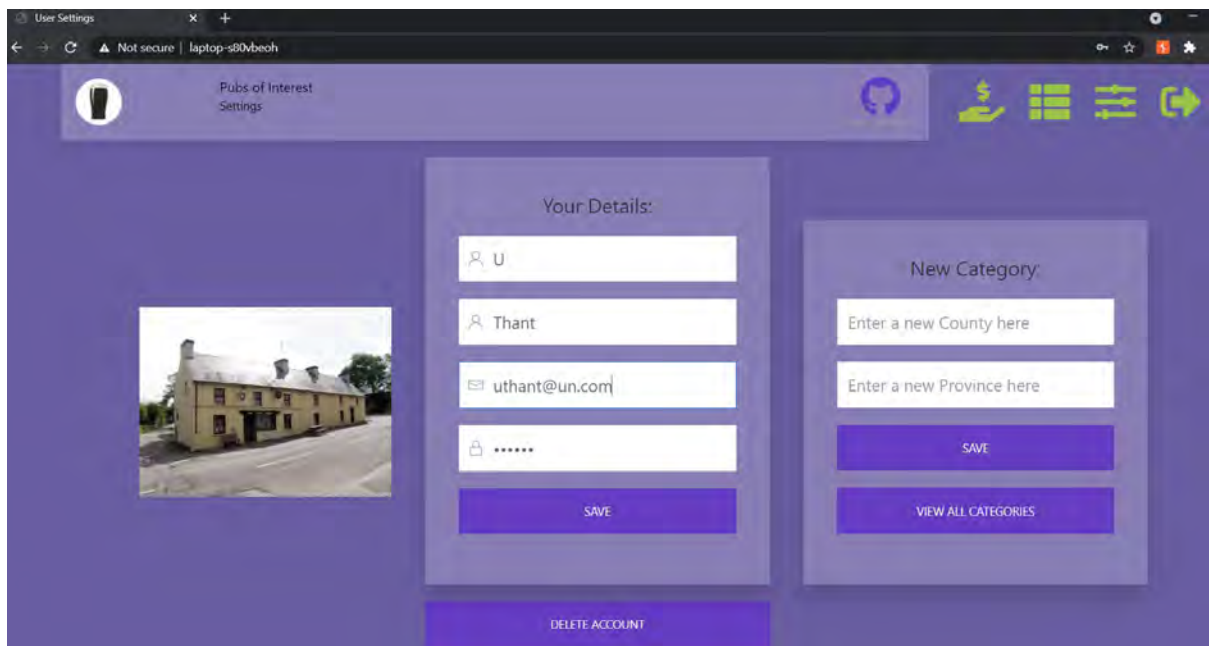


Figure 44: HTML restrictions applicable to the first two form fields have prevented them from being edited

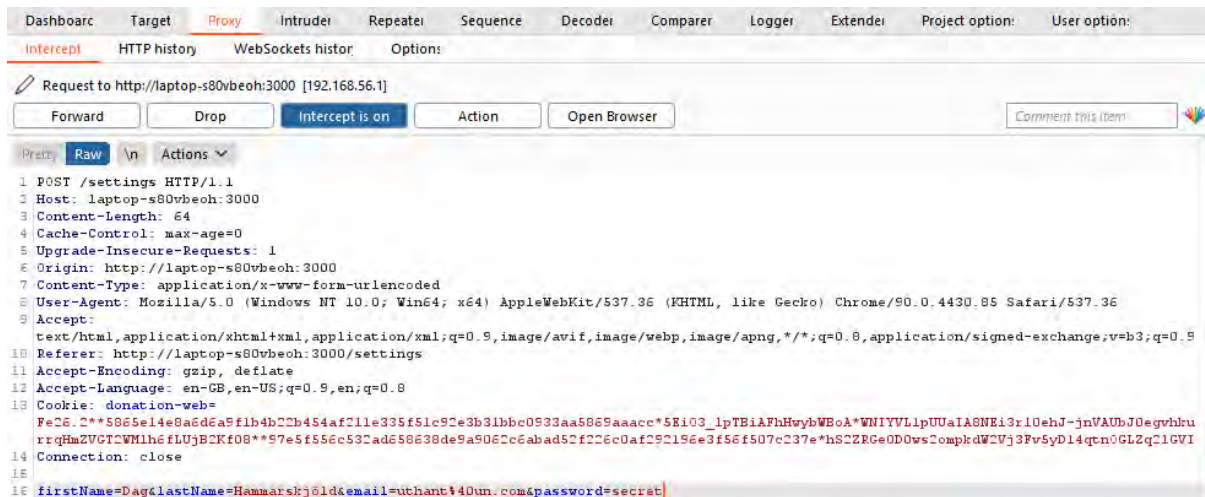


Figure 45: Data is modified to desired outcome by proxy interception

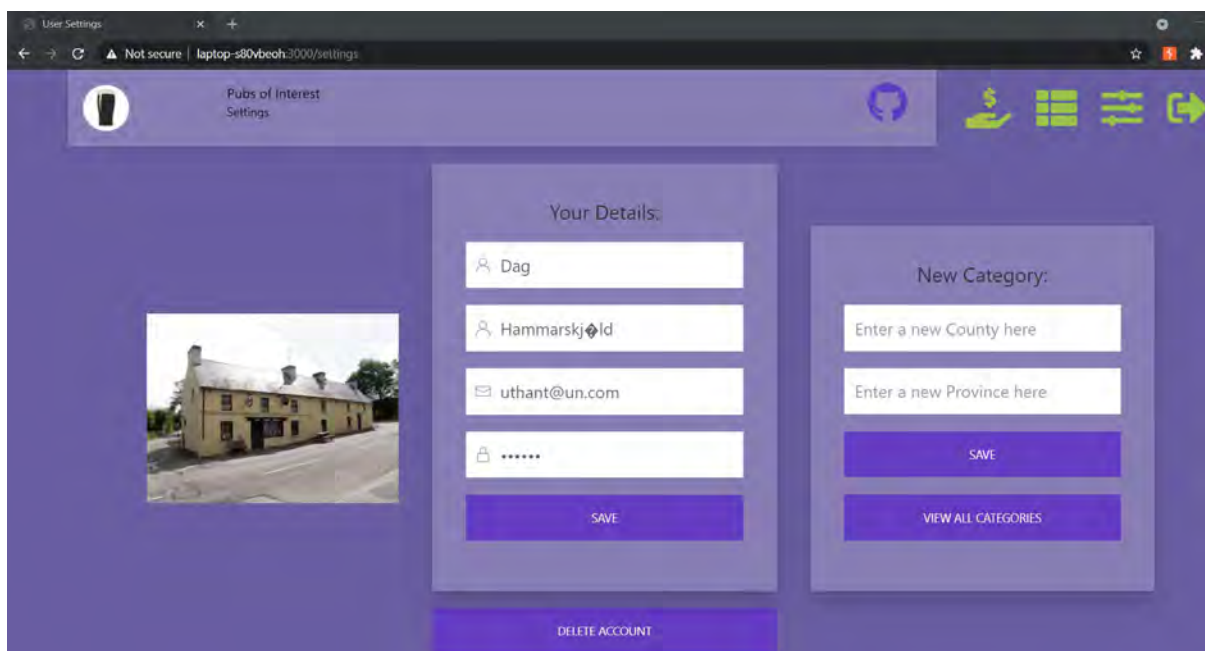


Figure 46: Despite the HTML restrictions preventing the user from the first two form fields, they were easily manipulated using the proxy server

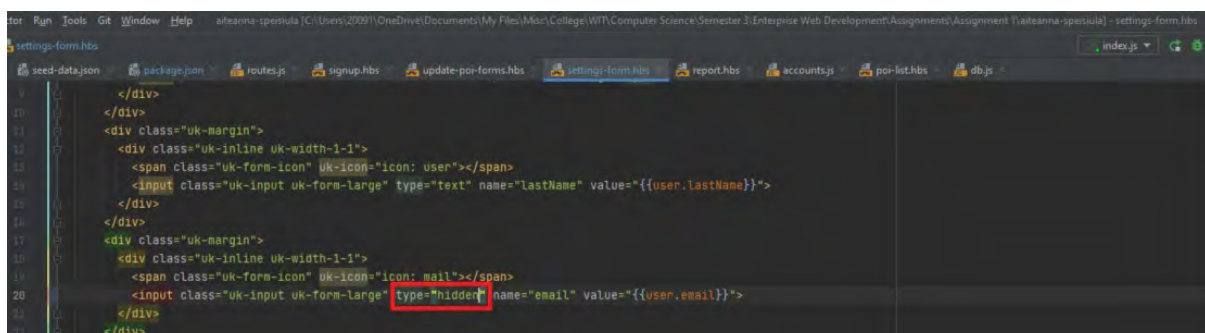


Figure 47: HTML restriction preventing the front-end visibility of an entire form field

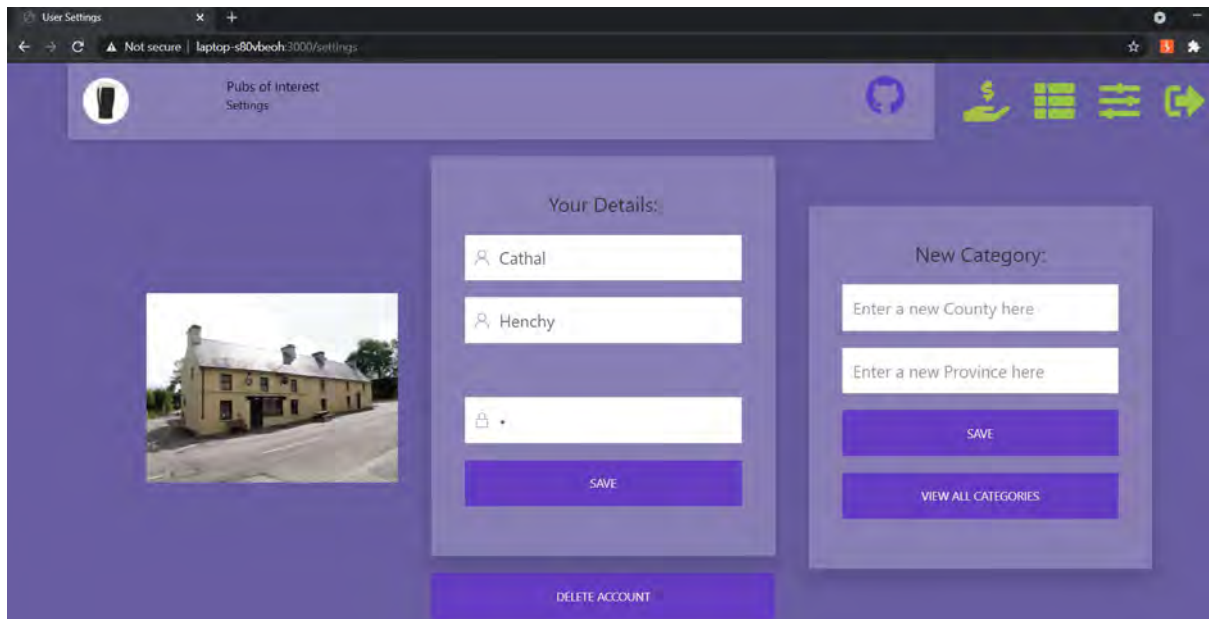


Figure 48: The HTML restriction has prevented the email form field from being visible on the UI

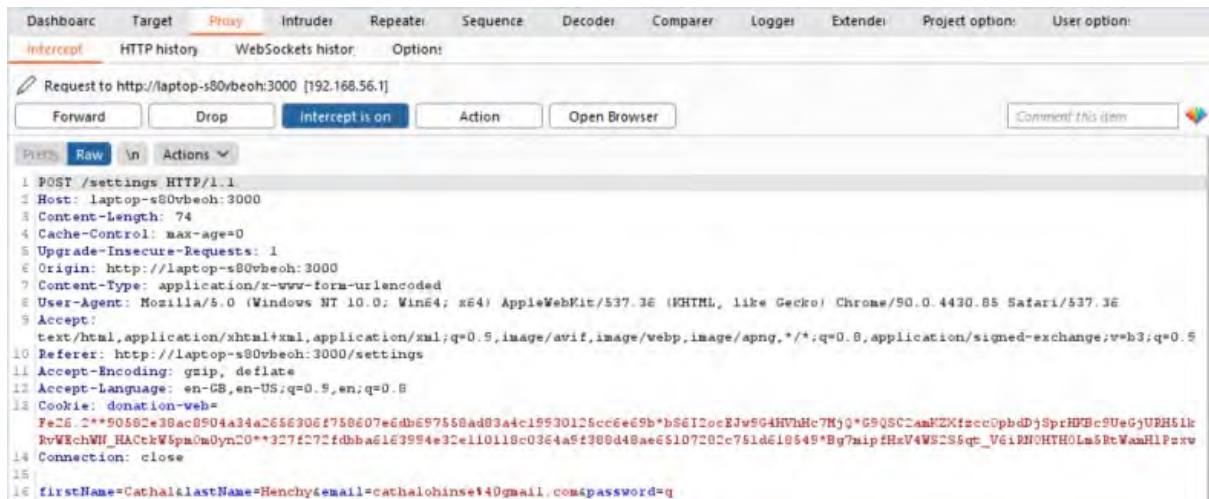


Figure 49: This hidden field is still visible on the proxy server however....

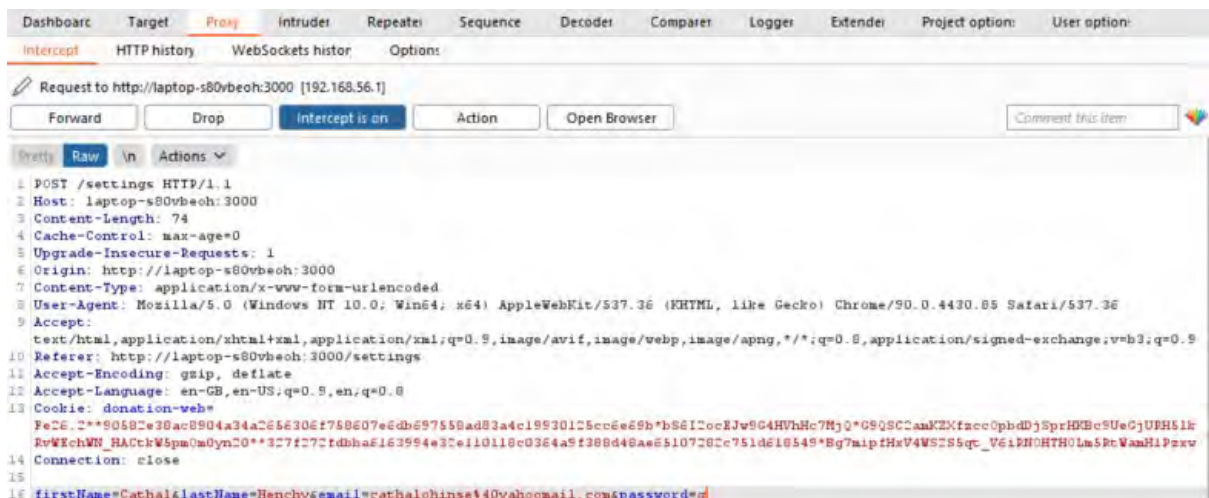


Figure 50: This can now be modified on the proxy server, with no visibility to the user

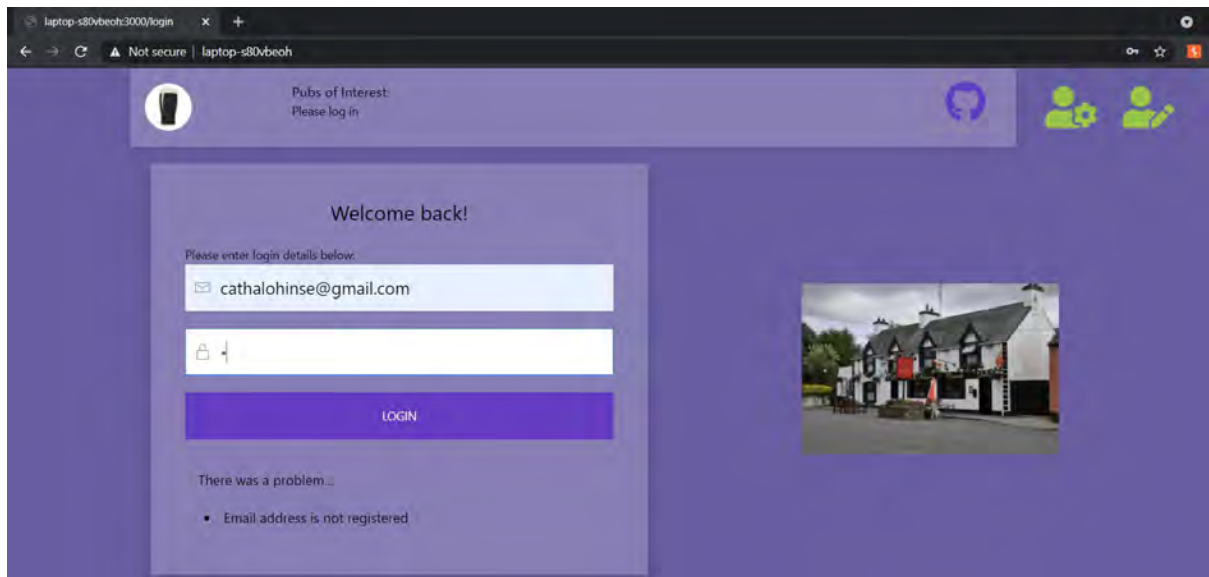


Figure 51: Because the user's email address has been changed on the proxy server, they no longer have access to the app using their old email address

e. Validation & Sanitisation:

I first edited my code to use filtering (validation) only (without using any sanitisation). This was achievable using the @hapi/joi plugin, which allowed me to write a set of rules (payload) into my form, all of which were required to be followed in order for any data input to be accepted:

```
signup: {  
  auth: false,  
  validate: {  
    payload: {  
      firstName: Joi.string()  
        .min(3)  
        .required(),  
      lastName: Joi.string()  
        .alphanum()  
        .required(),  
      email: Joi.string().email().required(),  
      password: Joi.string()  
        .min(6)  
        .required(),  
    },  
    options: {  
      abortEarly: false,  
    },  
    failAction: function (request, h, error) {  
      return h  
        .view("signup", {  
          title: "Sign up error",  
          errors: error.details,  
        })  
        .takeover()  
        .code(400);  
    },  
  },  
}
```

Figure 52: signup payload

The screenshot shows a web browser window with the address bar displaying 'laptop-s80vbeoh:3443/signup'. The browser's security warning bar indicates 'Not secure' and 'Secure Web App Please sign up'. The main content area features a large purple shield with a black padlock icon. To the right, there is a sign-up form titled 'Please enter details below' with four input fields: 'PJ', 'O'Connell', 'O'Callaghan's Mills', and a password field with a lock icon. A blue 'SIGN UP' button is at the bottom of the form.

Figure 53: Input using syntax that is in contravention of the payload

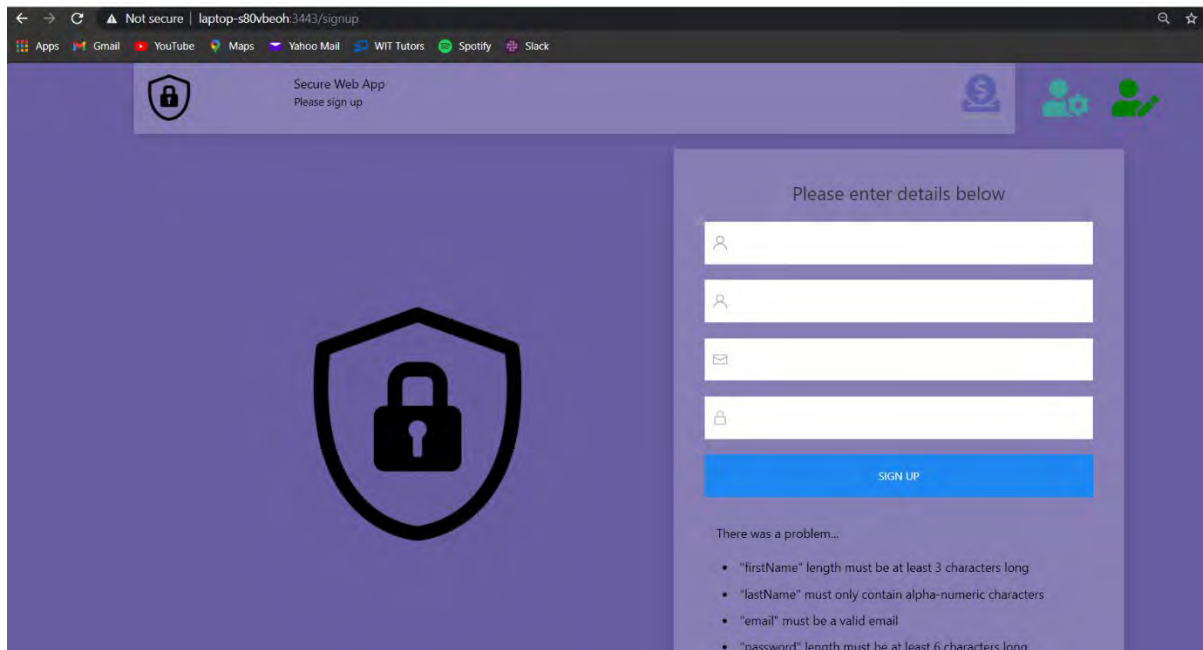


Figure 54: Rejection of all input ('abortEarly' is disabled)

This payload obviously isn't ideal in the real world, as PJ 'Fingers' O'Connell would like to use this wonderful app, but it demonstrates the concept of validation using a basic payload. I then enhanced the validation payload by incorporating regex (a regular expression) into the validation:

```
signup: {
  auth: false,
  validate: {
    payload: {
      firstName: Joi.string()
        // .min(3)
        // .required(),
        .regex(/^([A-Z][a-z]{2,})$/),
      lastName: Joi.string()
        .alphanum()
        .required(),
      email: Joi.string().email().required(),
      password: Joi.string()
        .min(6)
        .required(),
    },
    options: {
      abortEarly: false,
    },
    failAction: function (request, h, error) {
      return h
        .view("signup", {
          title: "Sign up error",
          errors: error.details,
        })
        .takeover()
        .code(400);
    },
  },
}
```

Figure 55: Regex that stipulates a minimum string length of 3, with the first letter being in uppercase

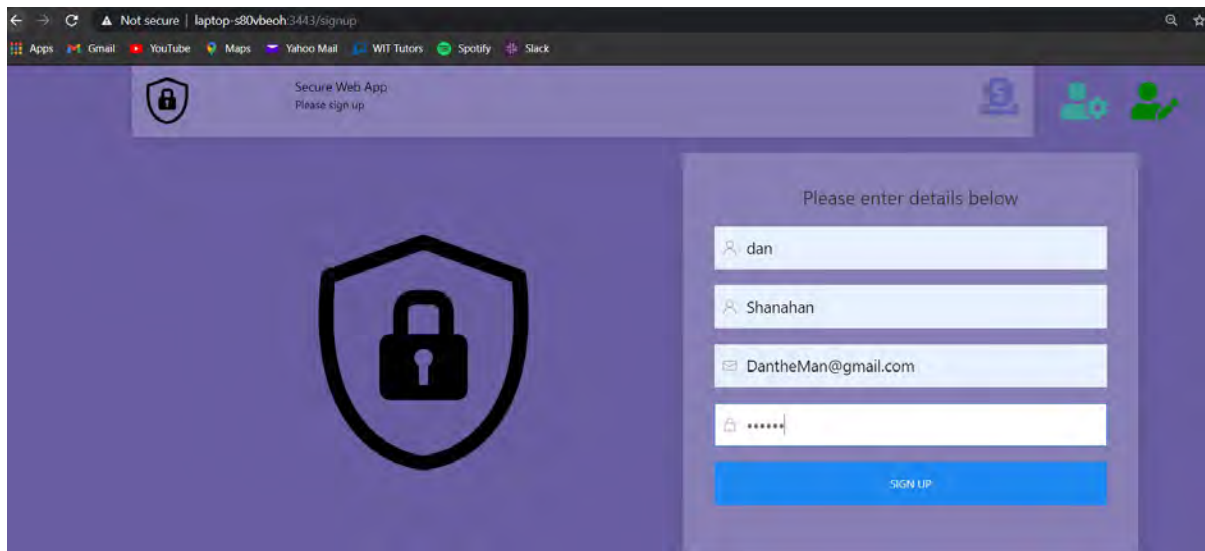


Figure 56: Input using syntax that is in contravention of the regex in the payload

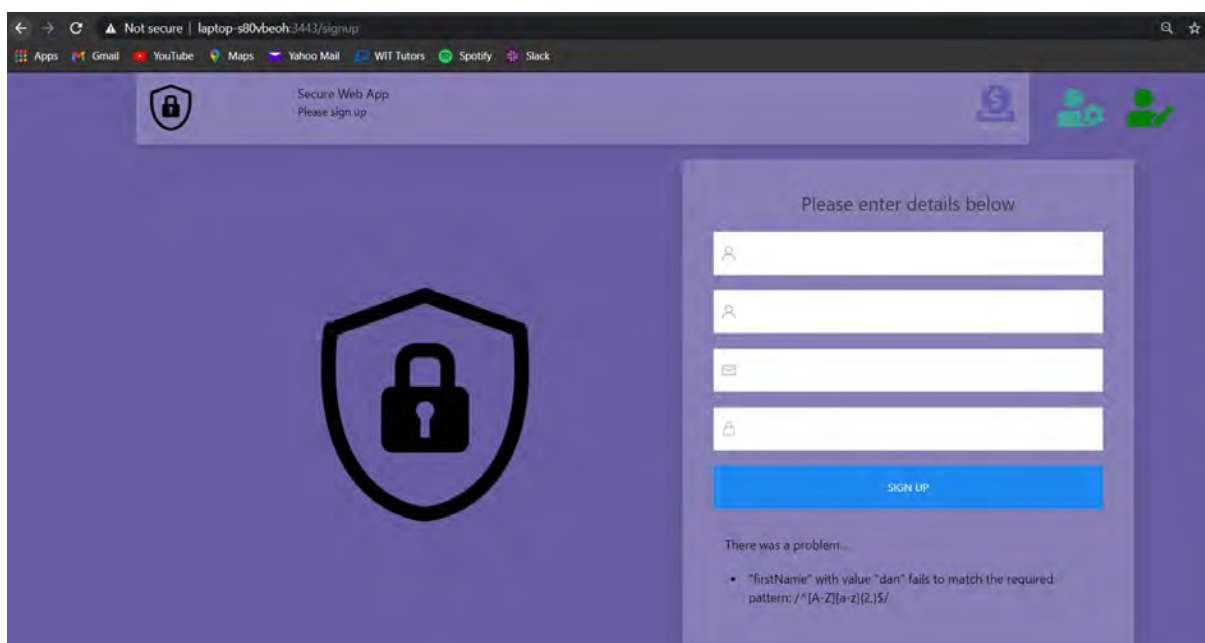


Figure 57: Rejection of input

I then spent an entire day trying to figure out how to execute sanitisation (i.e., editing or 'cleaning' the data (or 'dirty' data), by removing specific characters as per a pre-defined specification, and returning 'clean' data), but eventually gave up. I spent several hours examining and trying to make sense of several sanitisation plug-ins including *disinfect*, *sanitize*, *joi-plus* and many others. I installed a raft of them to my *aiteanna-speisiula* app but could not figure out how to get any of them to work. The documentation associated with all of them is excessively verbose and given that I don't have a background in computing, I was unable to make any sense whatsoever of any of it.

4. Conclusion

To summarise, all three applications that were examined here are quite secure. They have very few vulnerabilities in their code and are not reliant on HTML restrictions for secure protection and are thus unlikely to be overly susceptible to XSS or other form of attack. Also, given that there is password hashing & salting in place, any data theft would have a reduced impact.

One potential weakness is the lack of any digital certificate (TLS, SSL etc.) from a reputable CA. Although I issued it with one myself, a third-party CA would be preferable, as they would implement patches and updates to keep up to date with all relevant & required technological advancements.