6/26/2023

ESDE CA1 Report

Bee Design Website Vulnerability Report

Name: Tan Yong Kit, Eden

StudentID: P2243605

Class: DIT/FT/2A/06

Table of Contents

Executive Summary

This executive summary provides an overview of the vulnerabilities identified within the Bee Design website and outlines the recommended strategies for rectifying these vulnerabilities. Through analysis of the website there are critical security weaknesses that could potentially compromise the confidentiality, integrity, and availability of the website.

For this report, we will be addressing 6 vulnerabilities.

1. Cross-Site Scripting (XSS)
2. Injection (SQL Injection)
3. Broken Authentication
4. Broken Access Control
5. Sensitive Data Exposure
6. Insufficient Logging and Monitoring

1. Cross-Site Scripting

1.1 Definition

Cross-Site Scripting (XSS) is a type of web application vulnerability that occurs when an attacker injects malicious scripts into a trusted website, which are then executed by a victim's browser. It takes advantage of the trust a user has in a particular website to perform unauthorized actions or steal sensitive information.

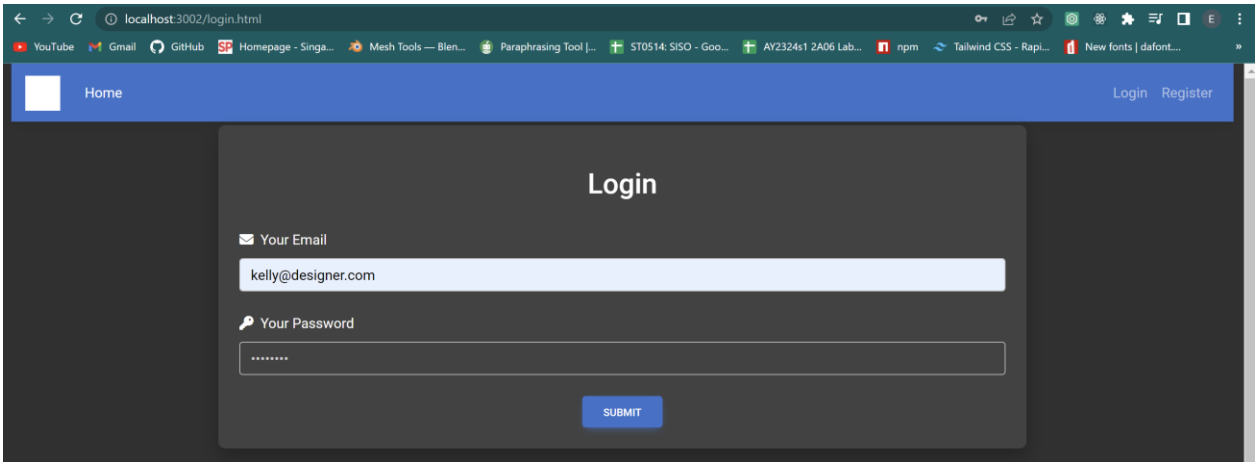
The vulnerability arises when a web application does not properly validate or sanitize user input before displaying it on a web page. This allows an attacker to insert malicious scripts, typically in the form of HTML or JavaScript, into the website's output. When a user visits the compromised page, the malicious script is executed in their browser, often without their knowledge.

There are 3 types of XXS attacks which are Stored, Reflected and DOM-Based.

1.2 How to exploit vulnerability

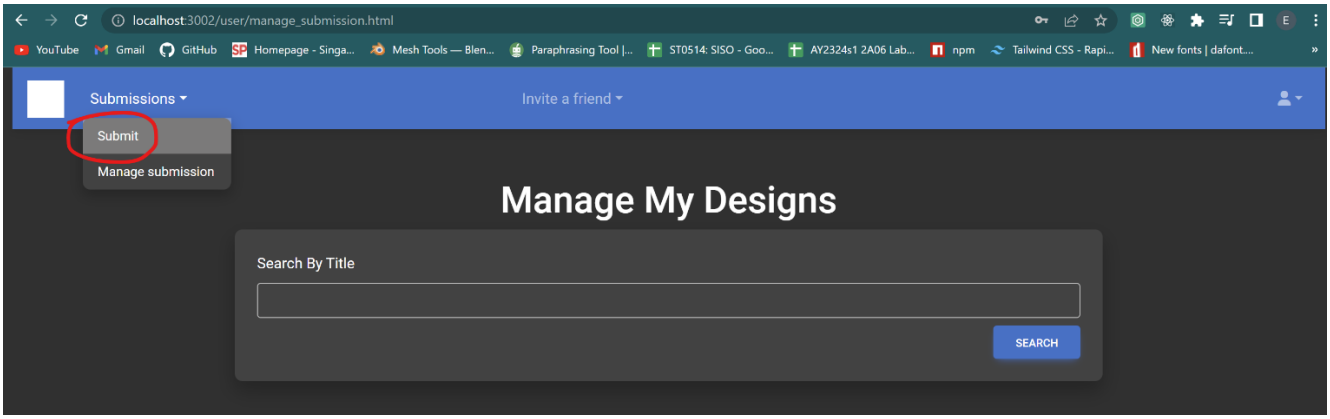
Step 1:

Login using Kelly's Email and Password into the Vulnerable Bee Design Website Login Page.



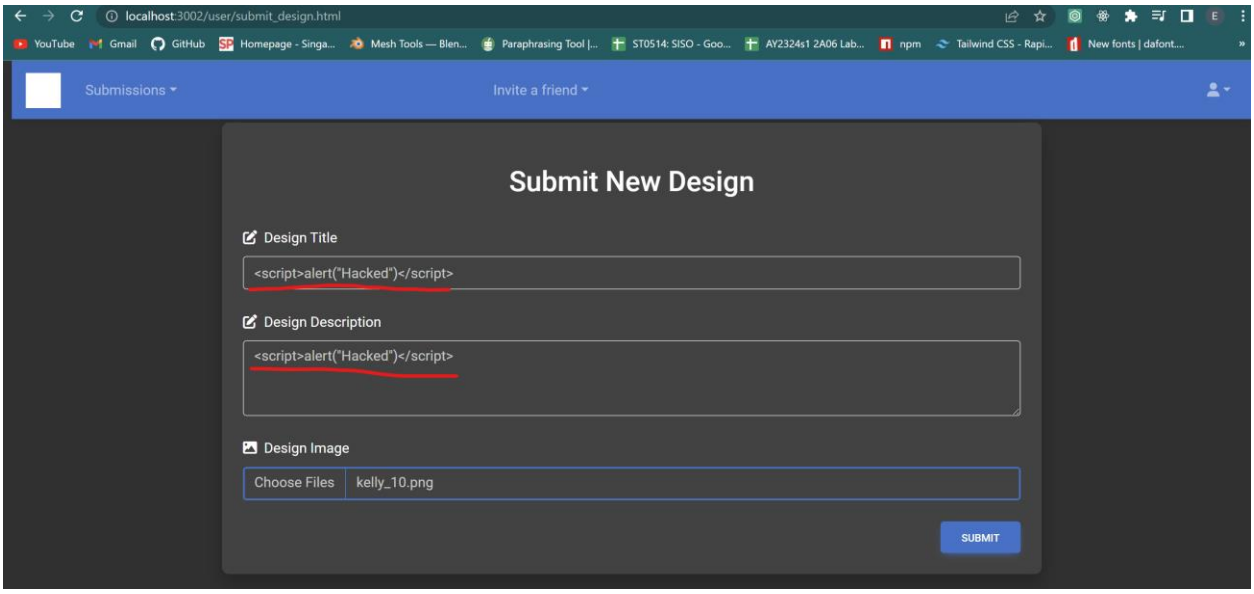
Step 2:

Go into the Submissions Tab at the top left corner of the website to submit a new design submission.



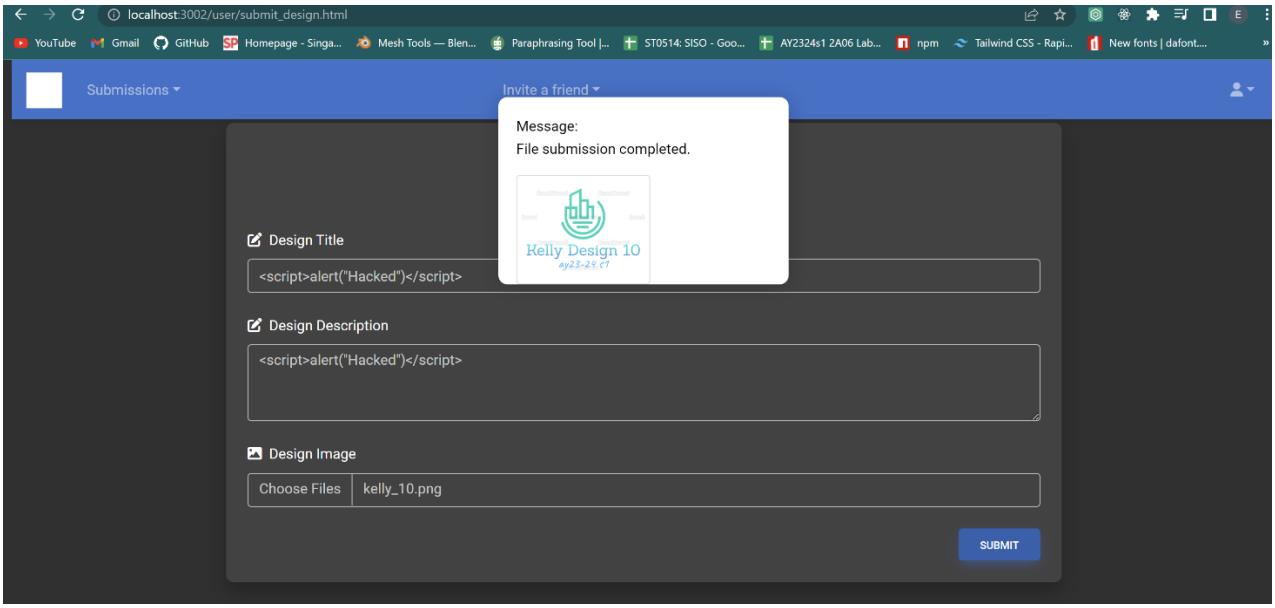
Step 3:

We will fill in the Design Title and Description with <script> tags to perform a Stored XSS Attack.



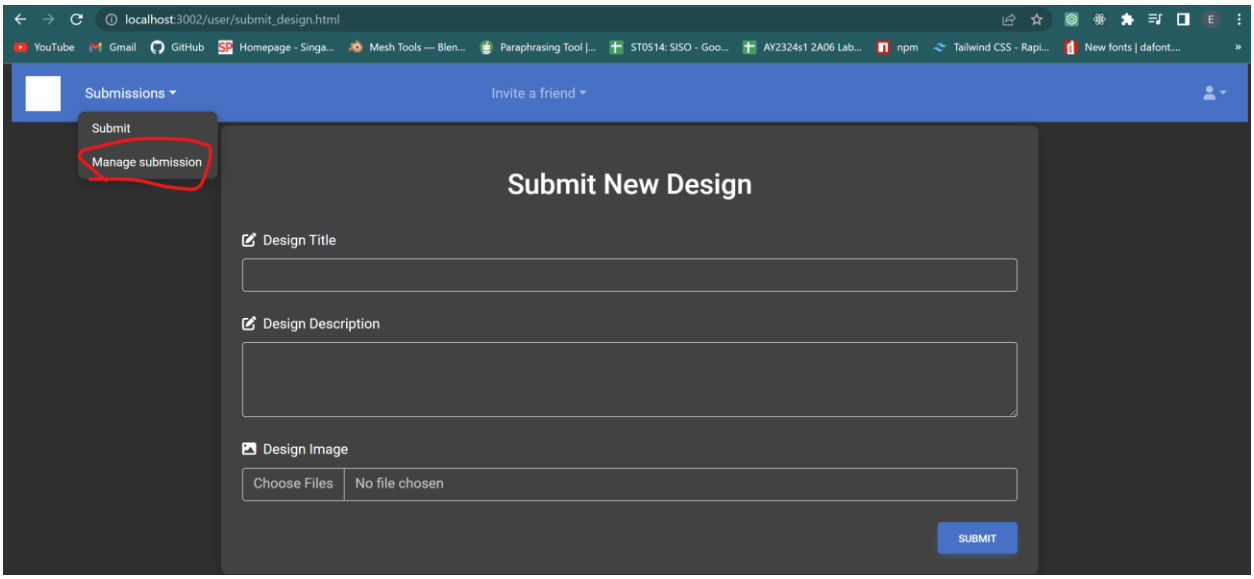
Step 4:

After submitting, the Design that we submitted successfully will be send and stored to the database.



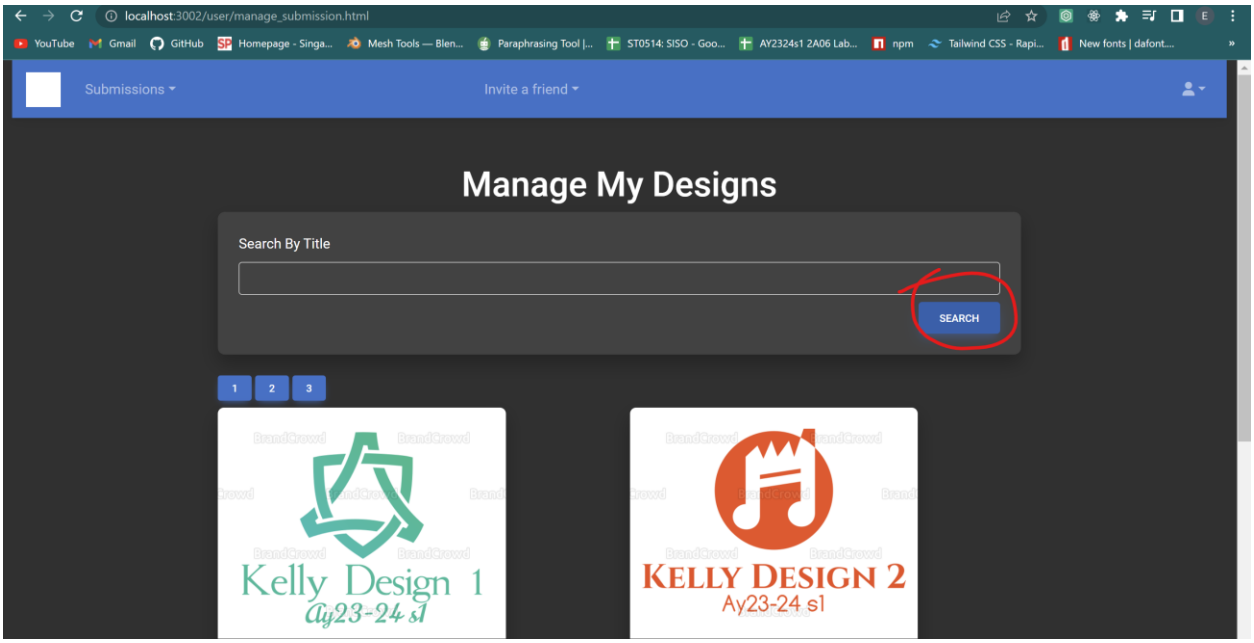
Step 5:

Next, we will go to Manage Submission to view the submission that we just submitted.



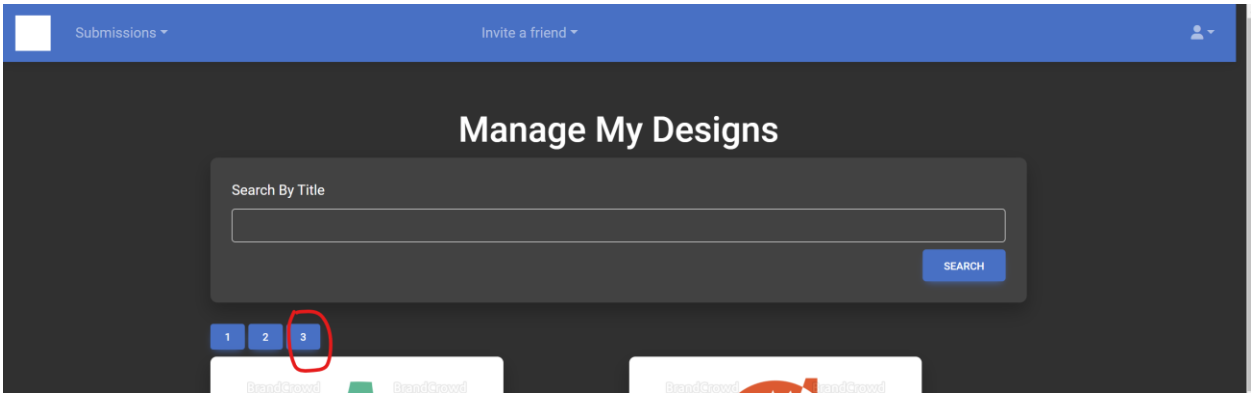
Step 6:

Click on the Search button to view the submissions of Kelly.



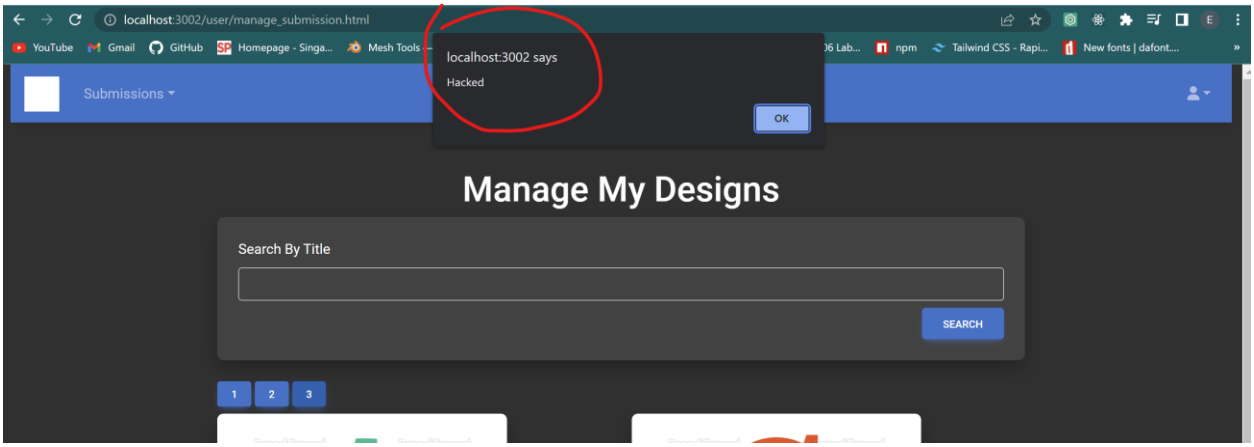
Step 7:

We will now go to view the recent submission that we submitted, click on the 3rd pagination to view.



Step 8:

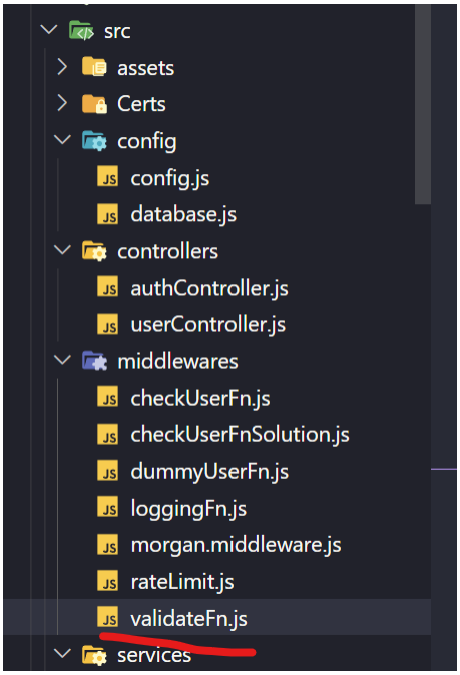
After clicking on pagination 3, you will be alerted with a Hacked message as shown below, which tells us that the Stored XSS Attack was successful.



1.3 How to rectify vulnerability

Step 1:

To prevent Stored XSS Attack, we will need to perform an Input Validation as well as an Output Sanitization. So, we will create a validateFn.js in the /backend/src/middlewares folder to validate the user input.



Step 2:

We will implement the following code in the validateFn.js file to validate the Design Title and Description before allowing the user to submit.

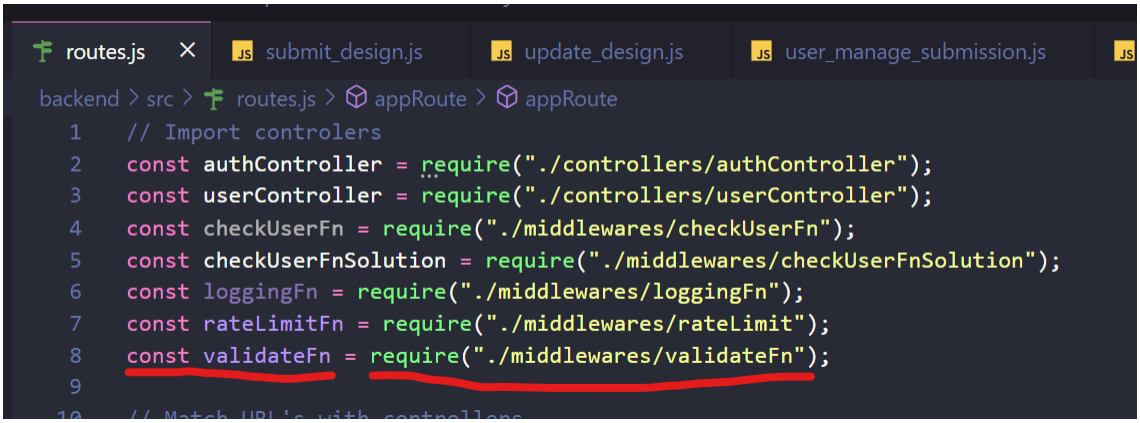
***Note declare the following code in validateFn.js first so that we can export it using module.exports to use this validateFn anywhere in the Backend Directory.**

```
const validateFn = {

        // The code below will be here

};

module.exports = validateFn;
```

```
73   validateDesignTitleAndDesc: function (req, res, next) {
74     var title = req.body.designTitle;
75     var description = req.body.designDescription;
76
77     // Description contains only alphabet, spaces, digit, single quote
78     const ReTitle = new RegExp(`^[\\w\\d\\s']+$`);
79     // Description contains only alphabet, spaces, digit, single/double quote or comma
80     const ReDescription = new RegExp(`^[\\w\\d\\s,'"]+$`);
81     if (ReTitle.test(title) && ReDescription.test(description)) {
82       next();
83     } else {
84       let message = "Unable to submit design, Title/Description is invalid.";
85       getIPAddress()
86         .then((ip) => {
87           logger.error({
88             method: req.method,
89             route: req.originalUrl,
90             ip: ip,
91             error: message,
92           });
93         })
94         .catch((error) => {
95           console.log(error);
96         });
97       res.status(400).send({
98         message: message,
99       });
100    }
101  },
```

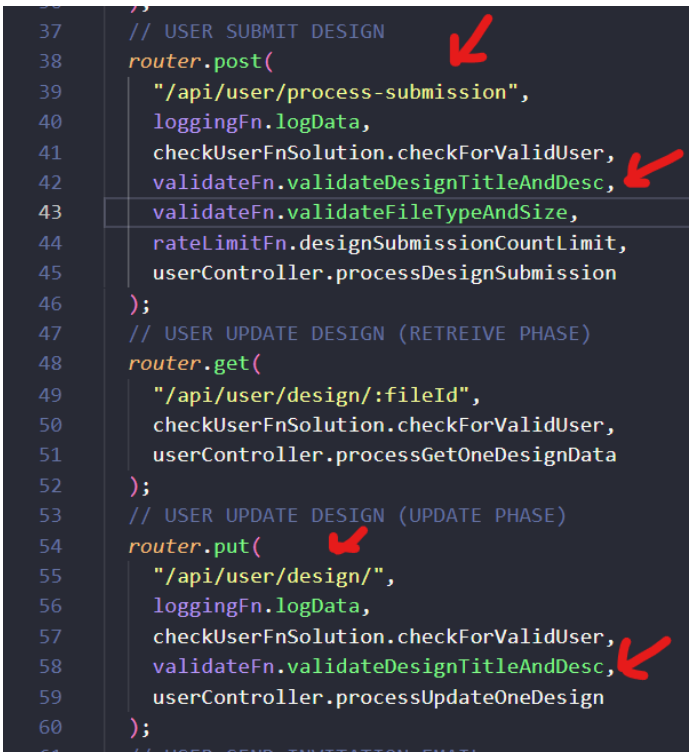
Step 3:

After implementing Input Validation, we will now use this middleware in the routes.js file under /backend/src. First, we will require the validateFn in the routes.js so let us bring it into routes.js.



Step 4:

We will now bring the middleware “validateFn.validateDesignTitleAndDesc” into the following endpoint which will help us validate the user input.



```
37  // USER SUBMIT DESIGN
38  router.post(
39    "/api/user/process-submission",
40    loggingFn.logData,
41    checkUserFnSolution.checkForValidUser,
42    validateFn.validateDesignTitleAndDesc,
43    validateFn.validateFileTypeAndSize,
44    rateLimitFn.designSubmissionCountLimit,
45    userController.processDesignSubmission
46  );
47  // USER UPDATE DESIGN (RETREIVE PHASE)
48  router.get(
49    "/api/user/design/:fileId",
50    checkUserFnSolution.checkForValidUser,
51    userController.processGetOneDesignData
52  );
53  // USER UPDATE DESIGN (UPDATE PHASE)
54  router.put(
55    "/api/user/design/",
56    loggingFn.logData,
57    checkUserFnSolution.checkForValidUser,
58    validateFn.validateDesignTitleAndDesc,
59    userController.processUpdateOneDesign
60  );
```

Step 5:

We will now go to the validateFn.js and import a library called validator first.

*Note Use “npm install validator” in your backend directory in the terminal if you have not installed in your Backend folder directory.

routes.js | submit_design.js | update_design.js | user_manage_submission.js | userController.js | validateFn.js | auth

backend > src > middlewares > validateFn.js > validateFn > sanitizeResult

```
1  var validator = require("validator");
2  var path = require("path");
3  var fs = require("fs");
4  var logger = require("../utils/logger");
5
6  const validateFn = {
7    validateRegister: function (req, res, next) {
8      var fullname = req.body.fullname;
```

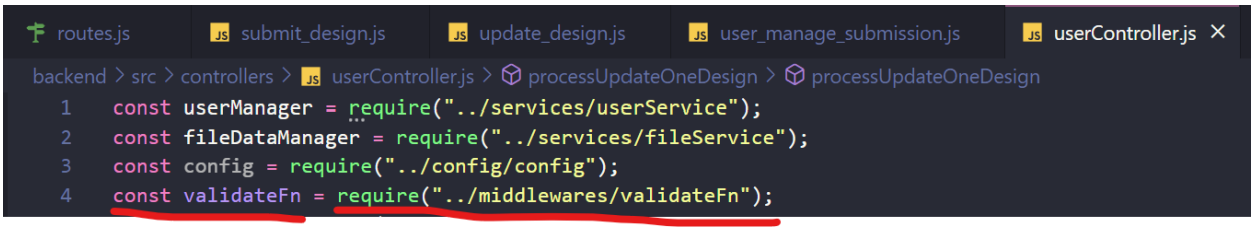
Step 6:

Next, add the following code which we will use to sanitize the results that we retrieve from the database.

```
253   sanitizeResult: function (result) {
254     if (result.filedata) {
255       for (file of result.filedata) {
256         file.design_title = validator.escape(file.design_title);
257         file.design_description = validator.escape(file.design_description);
258       }
259     } else if (result.userdata) {
260       if (result.userdata.length >= 1) {
261         for (user of result.userdata) {
262           user.fullname = validator.escape(user.fullname);
263           user.email = validator.escape(user.email);
264           user.role_name = validator.escape(user.role_name);
265         }
266       }
267     }
268     return result;
269   }, // end sanitizeResult
270 };
```

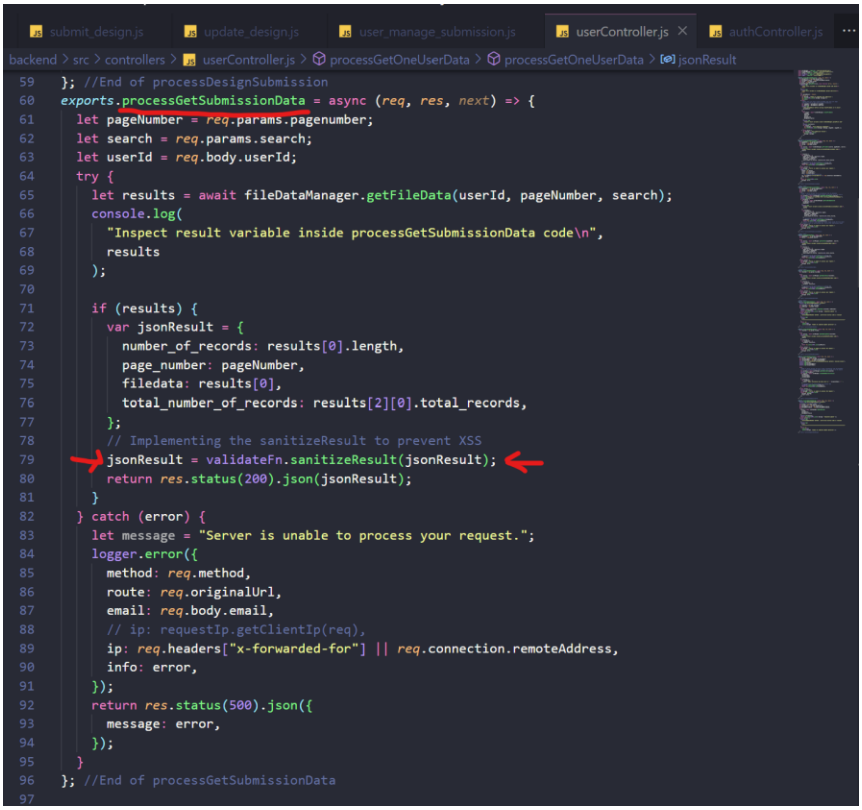
Step 7:

Let us bring in the validationFn.js file into the userController.js file for output sanitization located under /backend/src/controllers/userController.js .



Step 8:

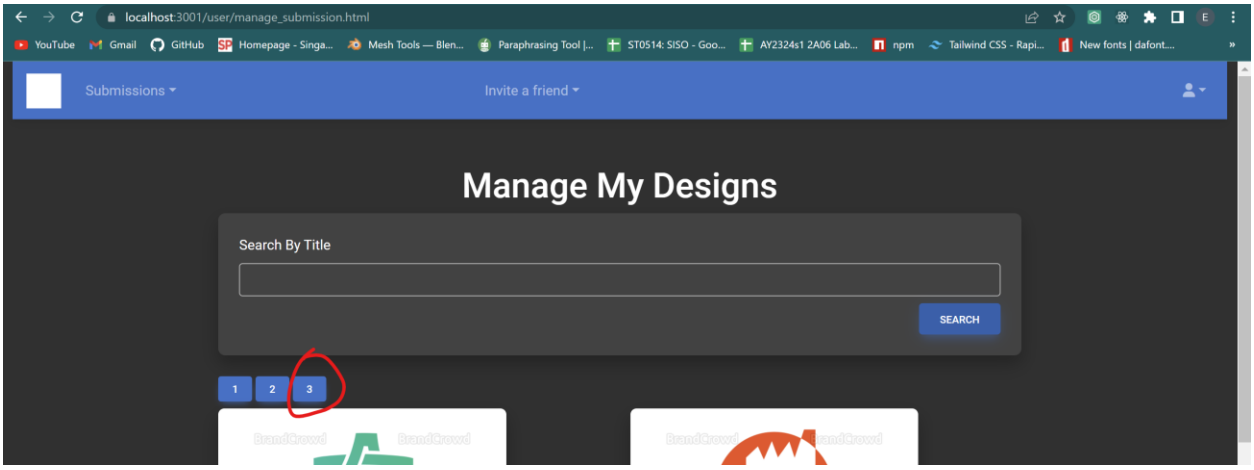
We will now use the Output Sanitization to prevent any existing script tags from working. Go to userController.js and look for the following "exports.processGetSubmissionData" to add the validateFn.sanitizeResult.



1.4 Evidence that the vulnerability has been rectified

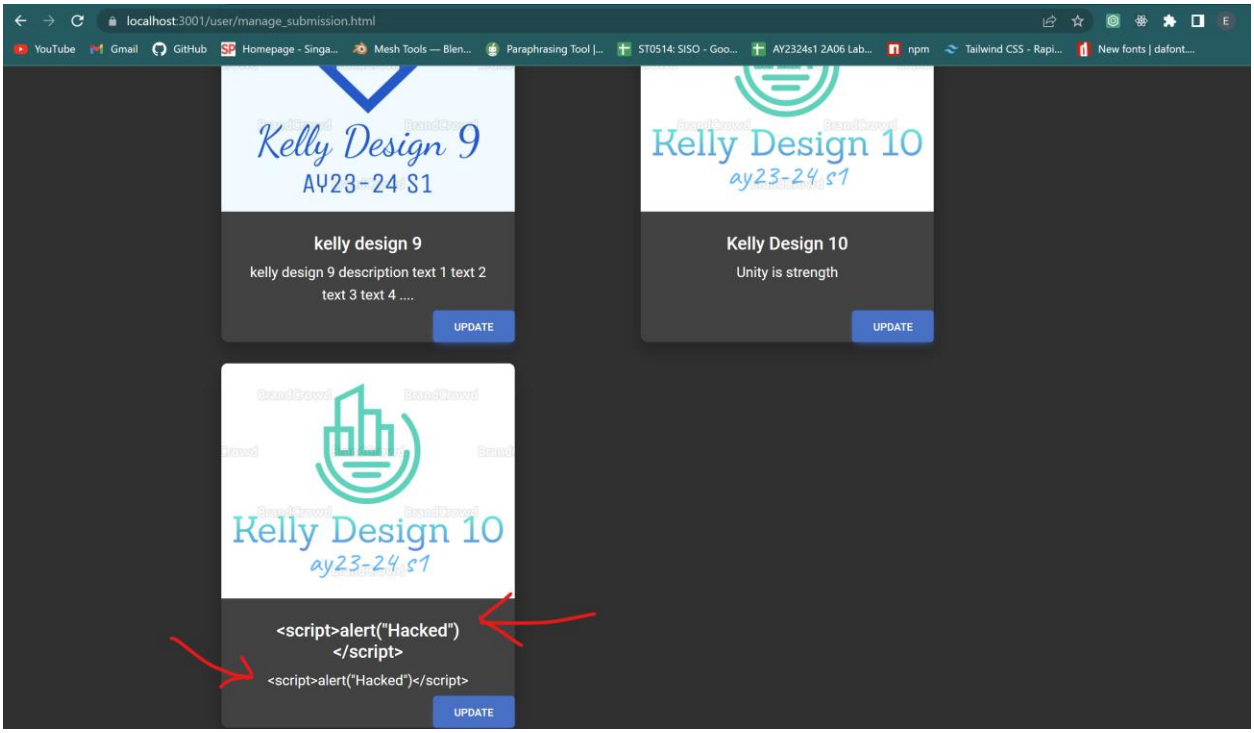
Step 1:

Go back to the Manage My Designs page to search for Kelly's submissions and click on the third pagination to see if the alert still shows.



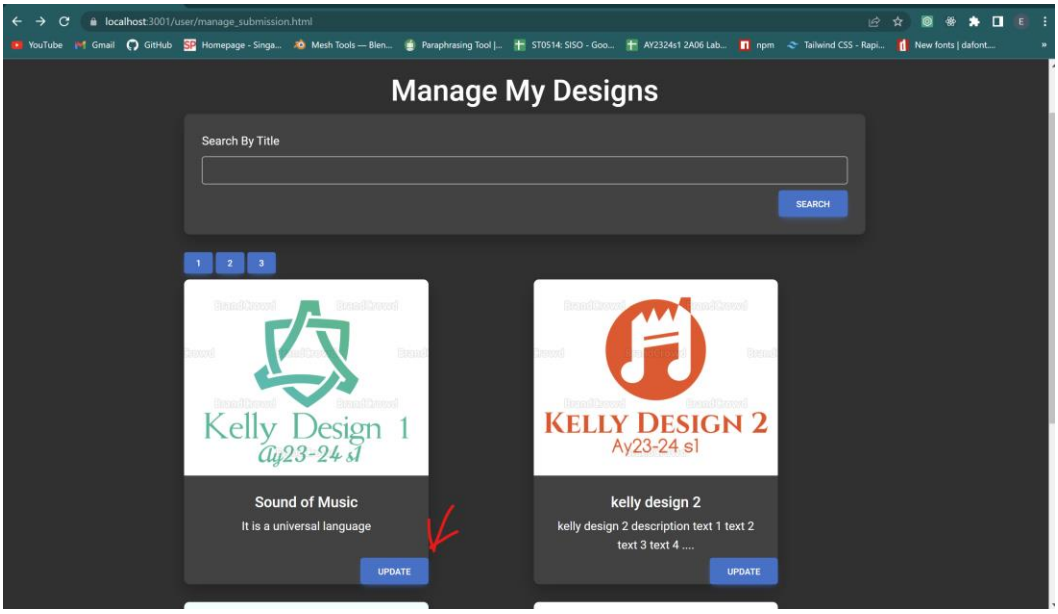
Step 2:

You should be able to see the escaped script tags in the title and description and no alert has popped up. This shows that we have successfully performed the output sanitization.



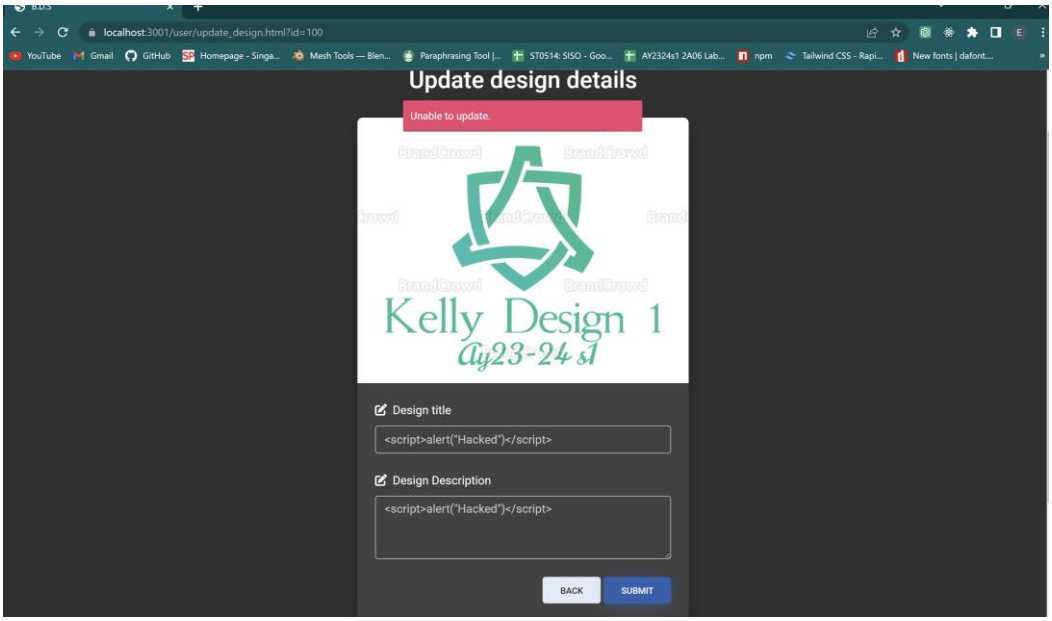
Step 3:

Lastly, let's try to update the design with script tags to see if our validation worked. Go to pagination 1 and click on the Update button.



Step 4:

Fill in the title and description with script tags to see if it will go through. Upon submitting you should see an Unable to update message which shows that our input validation is working.



2. Injection (SQL Injection)

2.1 Definition

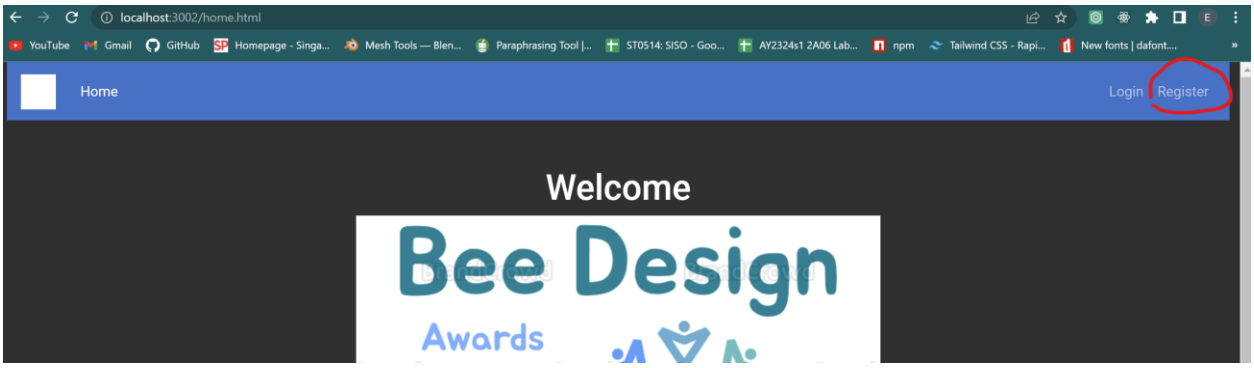
SQL Injection is a type of web application vulnerability that allows an attacker to manipulate a web application's database by injecting malicious SQL (Structured Query Language) code into user input fields or other vulnerable areas of the application. It occurs when the web application fails to properly validate or sanitize user input before incorporating it into SQL queries, allowing the attacker to execute arbitrary SQL commands.

The vulnerability arises when user-supplied data is concatenated directly into SQL statements without proper sanitization or parameterization. The attacker can exploit this vulnerability by submitting crafted input that includes SQL code as part of the user input. When the application processes this input, the malicious SQL code is executed by the database server, potentially leading to unauthorized access, data manipulation, or disclosure of sensitive information.

2.2 How to exploit vulnerability

Step 1:

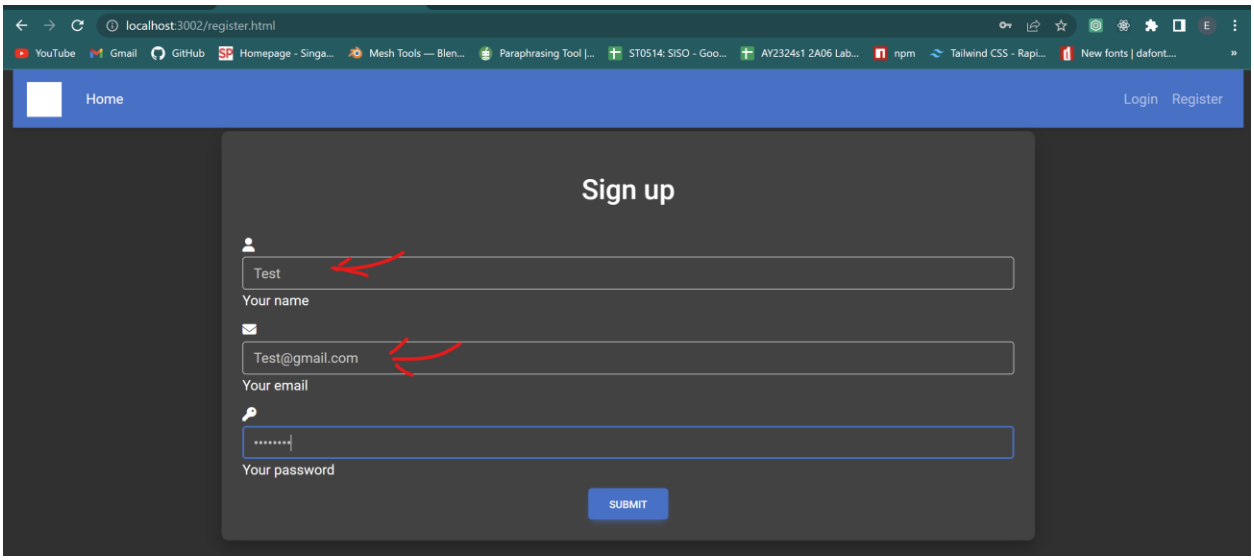
Go into the Register Page to Register as a new user by clicking on Register at the Navbar in the Welcome Page.



Step 2:

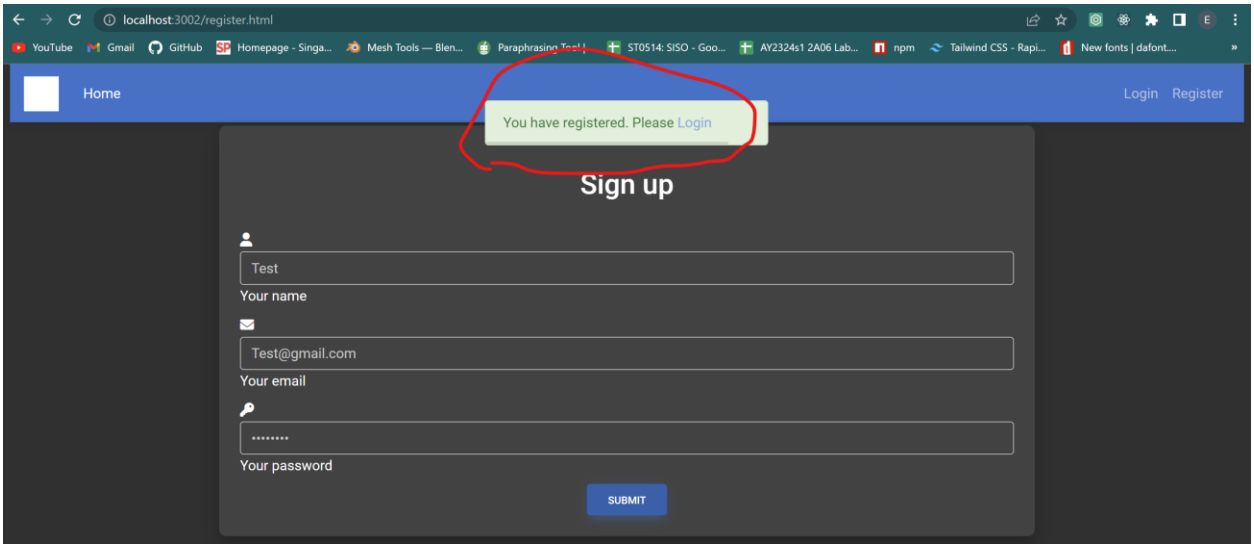
Fill in the necessary fields to register as a new user.

For this case - Name: Test, Email: Test@gmail.com, Password: password



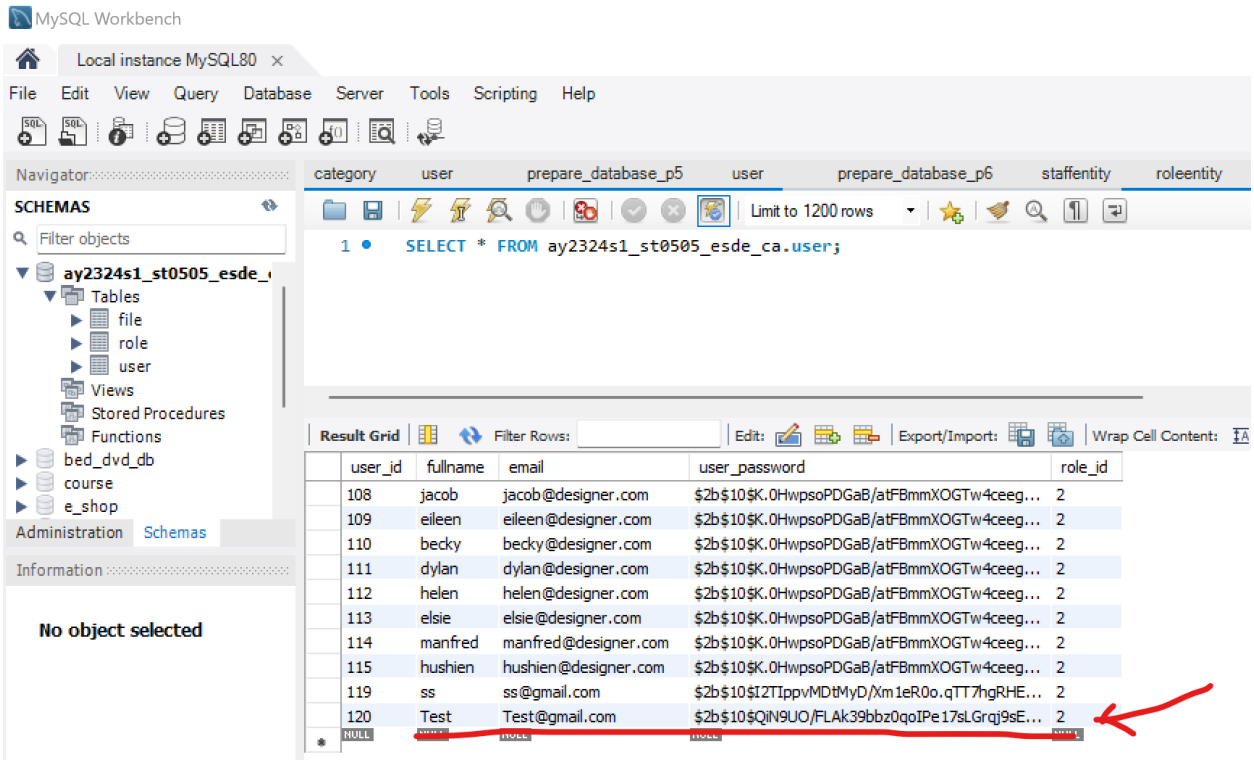
Step 3:

You should see that you have successfully registered.



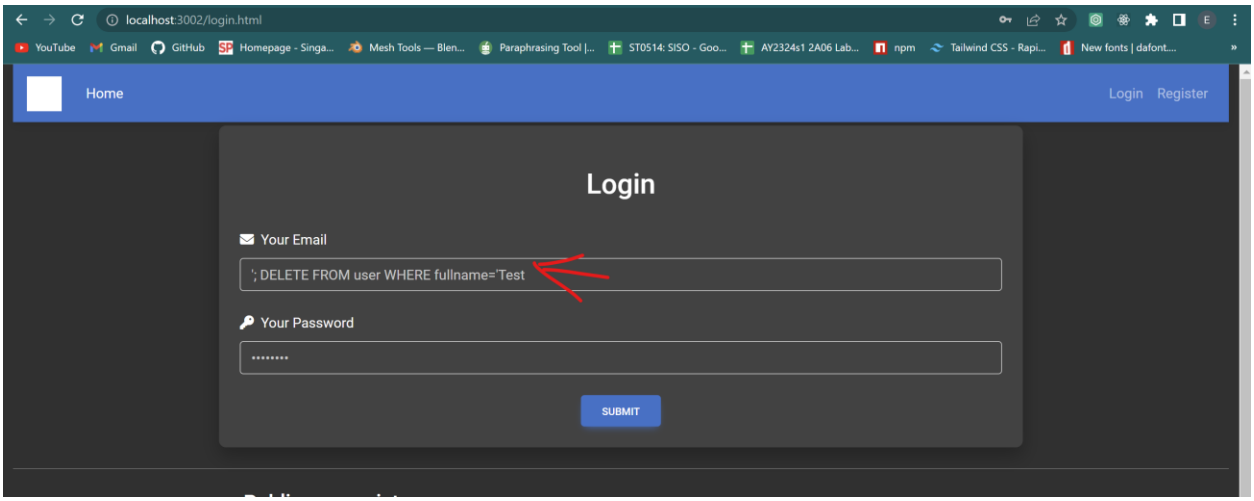
Step 4:

Go into MySQL Workbench and look for the user we just registered under the “user” table.



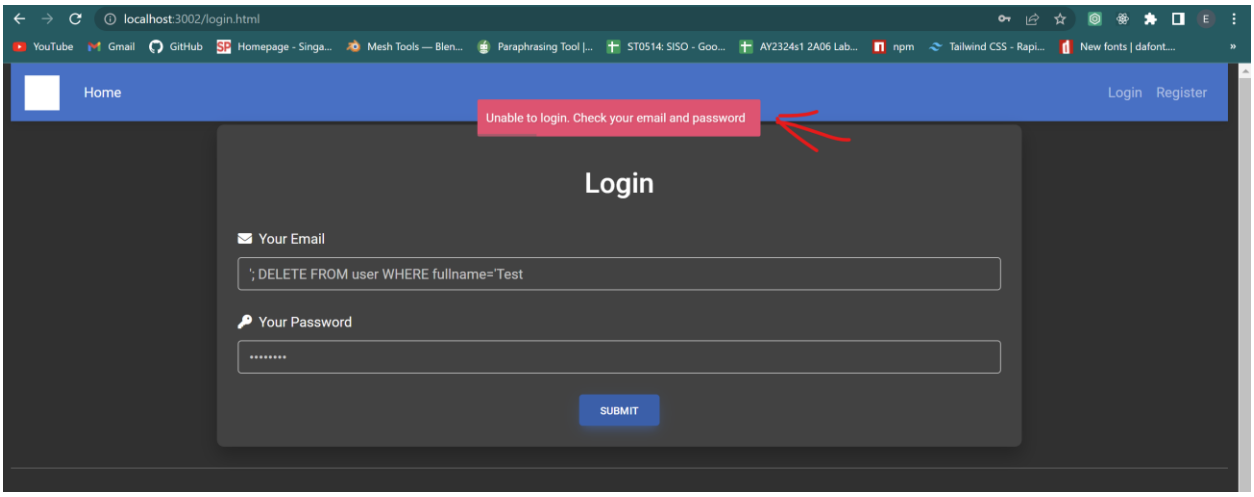
Step 5:

Now, go to the Vulnerable Bee Design Website Login Page we will add a SQL statement to try to delete the recent user that we just created.



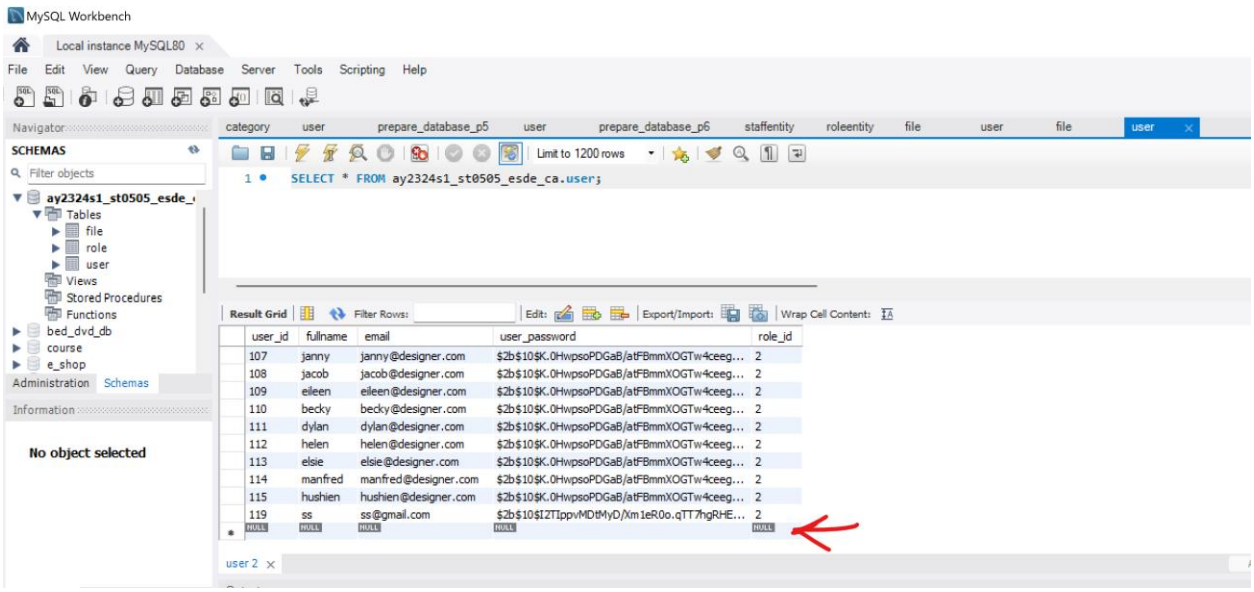
Step 6:

You should be able to see a message saying “Unable to Login. Check your email and password”.



Step 7:

Now, head over to MySQL Workbench to see if the user “Test” is still there. You will see that the user “Test” has been deleted. This shows that the Login page is susceptible to SQL Injection.

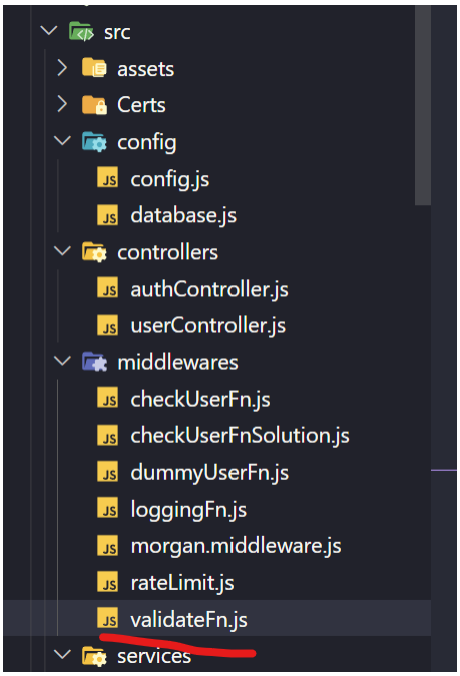


2.3 How to rectify vulnerability

Step 1:

To rectify this problem, we can do two different things, the first thing is input validation followed by using parameterized queries.

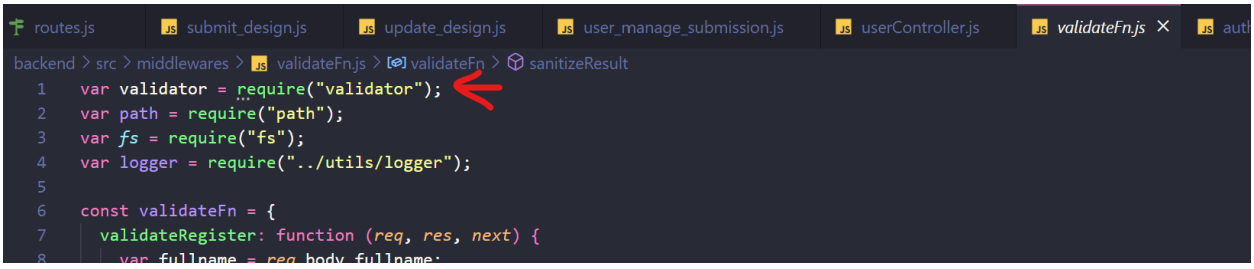
For Input Validation, head over to your VsCode and look for the middleware that we created from **the 1st Vulnerability** called validateFn.js.



Step 2:

Make sure that you have validator library being installed and imported into your validateFn.js file.

*Note Use “npm install validator” in your backend directory in the terminal if you have not installed in your Backend folder directory.



Step 3:

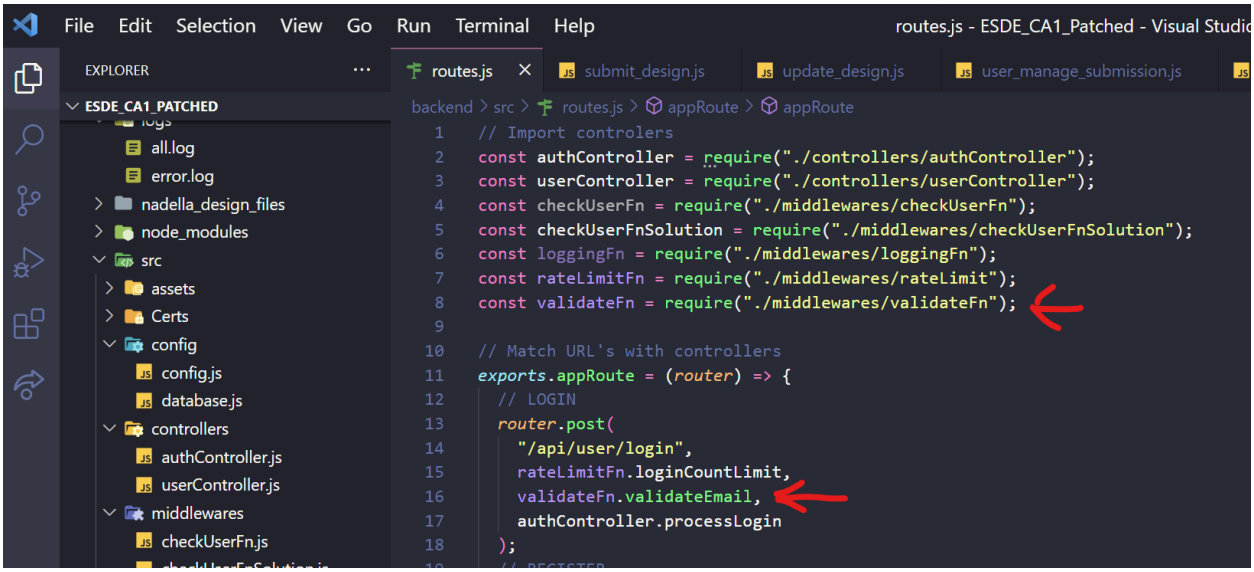
Add the following code into your validateFn.js file.

*Note: Ignore line 56-68 if you have not done the 6th Vulnerability Insufficient Logging and Monitoring.

```
50  validateEmail: function (req, res, next) {
51    var email = req.body.email;
52    if (validator.isEmail(email)) {
53      next();
54    } else {
55      let message = "Unable to Login, Please enter a valid Email";
56      getIPAddress()
57        .then((ip) => {
58          logger.error({
59            method: req.method,
60            route: req.originalUrl,
61            email: req.body.email,
62            ip: ip,
63            error: message,
64          });
65        })
66        .catch((error) => {
67          console.log(error);
68        });
69      res.status(400).send({ message: message });
70    }
71  },
```

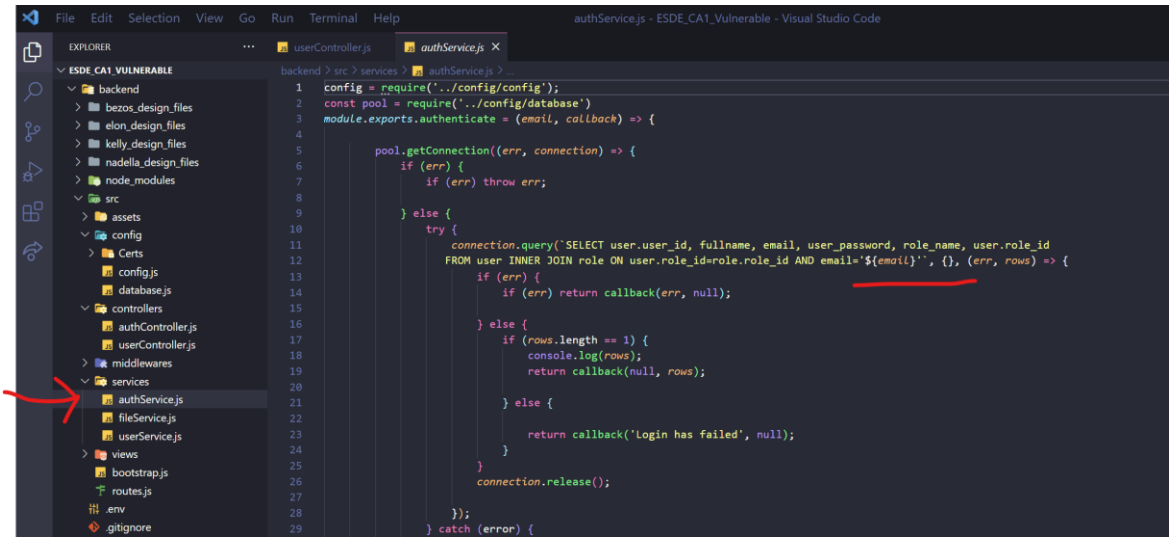
Step 4:

Next, head over to routes.js to add this middleware to the Login Endpoint. Do remember to import the validateFn.js file into the routes.js file.



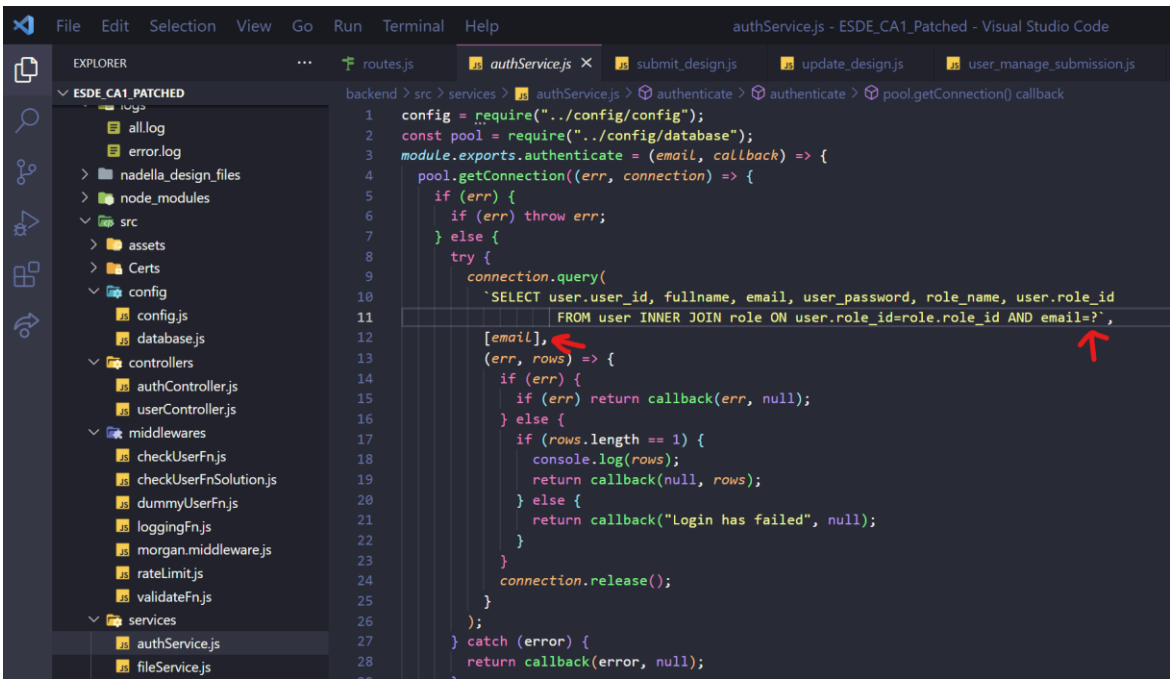
Step 5:

Now, let us go to authService.js to look at the SQL Statement used to check the Login credentials. Located at /backend/src/services/authService.js.



Step 6:

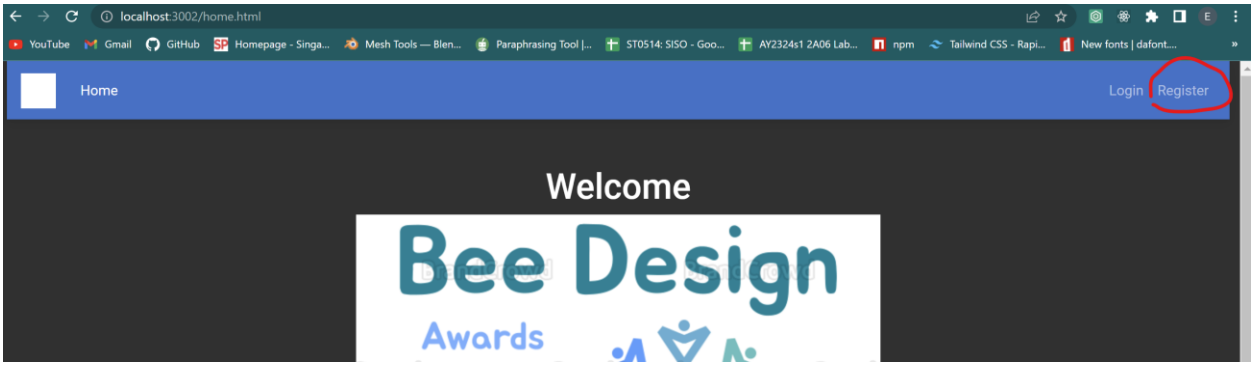
As you can see that the SQL Statement is not using Parameterized Query. So, let us change the following into the code shown below to use Parameterized Query.



2.4 Evidence that the vulnerability has been rectified

Step 1:

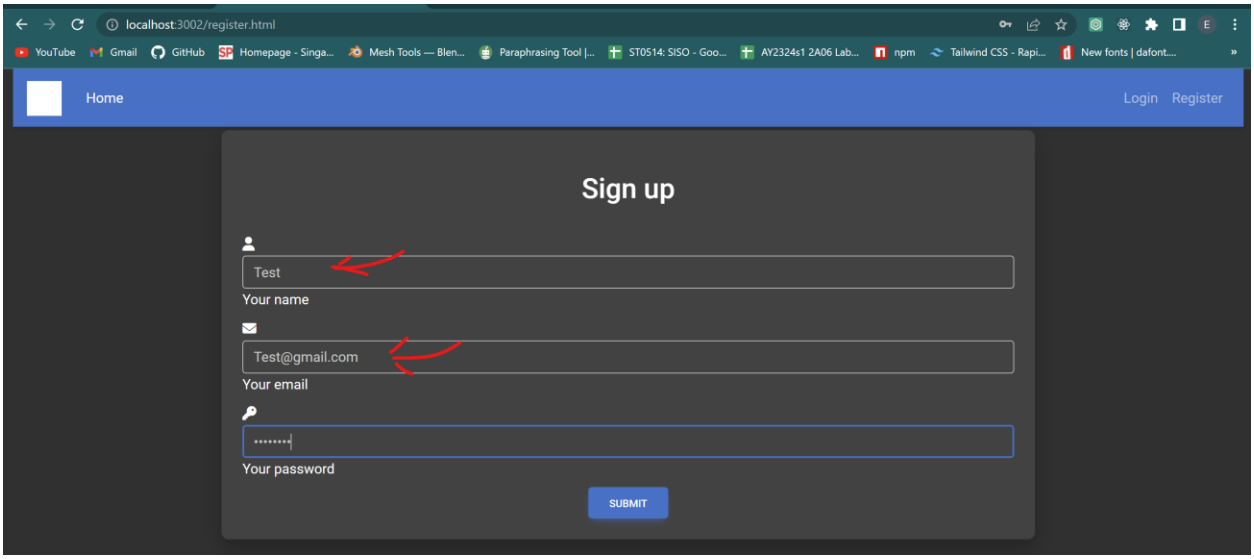
Go into the Register Page to Register as a new user by clicking on Register at the Navbar in the Welcome Page.



Step 2:

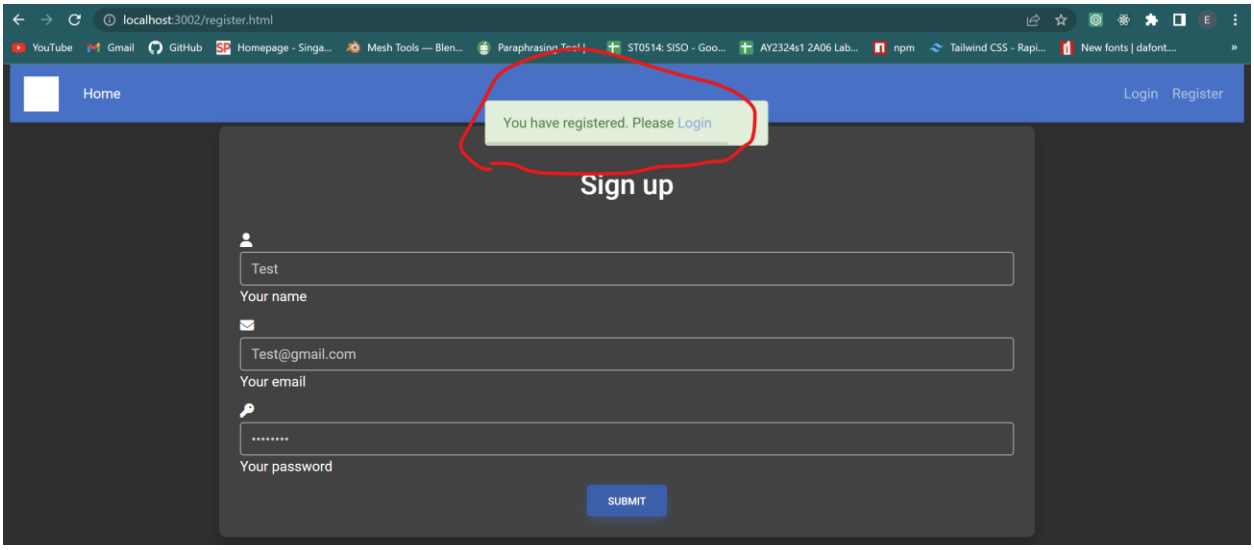
Fill in the necessary fields to register as a new user.

For this case - Name: Test, Email: Test@gmail.com, Password: password



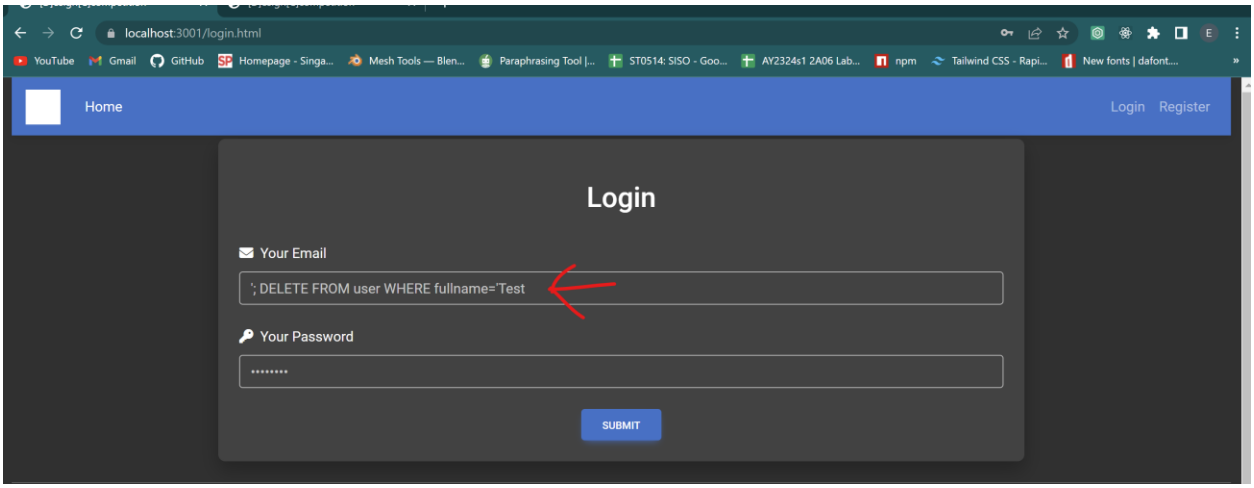
Step 3:

You should see that you have successfully registered.



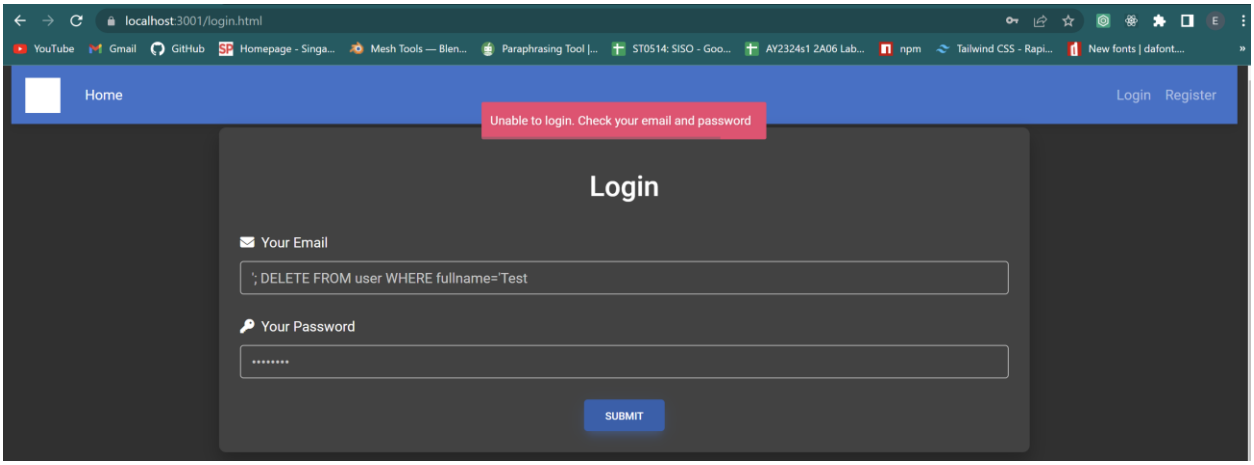
Step 4:

We will now see if the Input Validation and Parameterized Query has removed the SQL Injection Vulnerability.



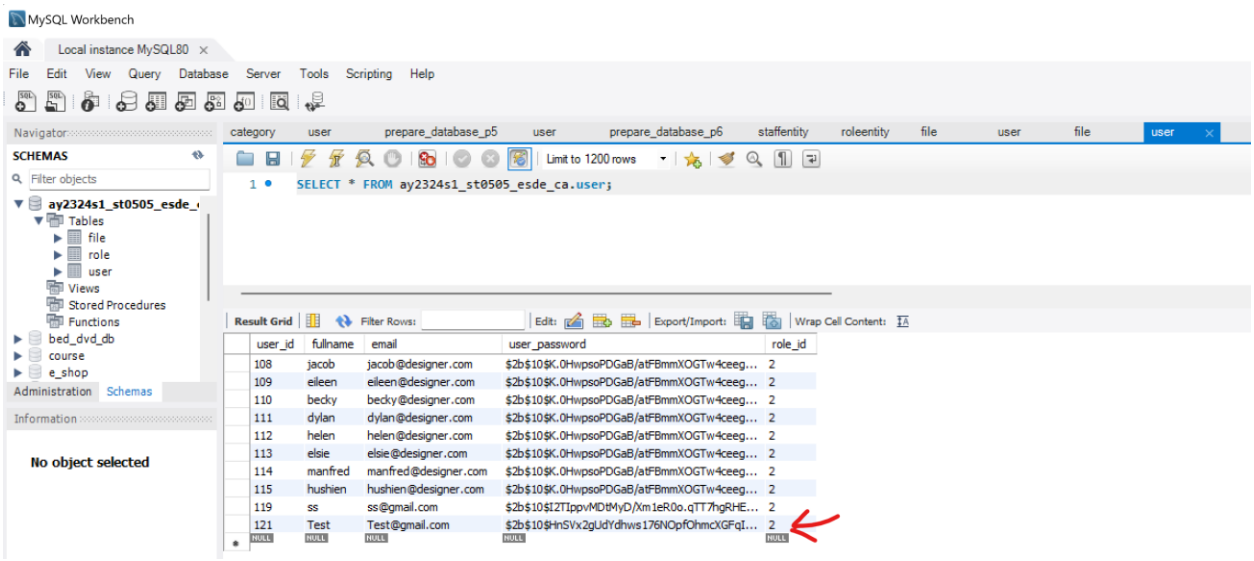
Step 5:

You should still see the same pop up showing invalid email and password.



Step 6:

Go over to see if the “Test” user is being deleted, you should be able to still see the “Test” user in your MySQL Workbench under the user table. This shows that we have successfully rectified the SQL Injection Vulnerability for the Login Page.



3. Broken Authentication

3.1 Definition

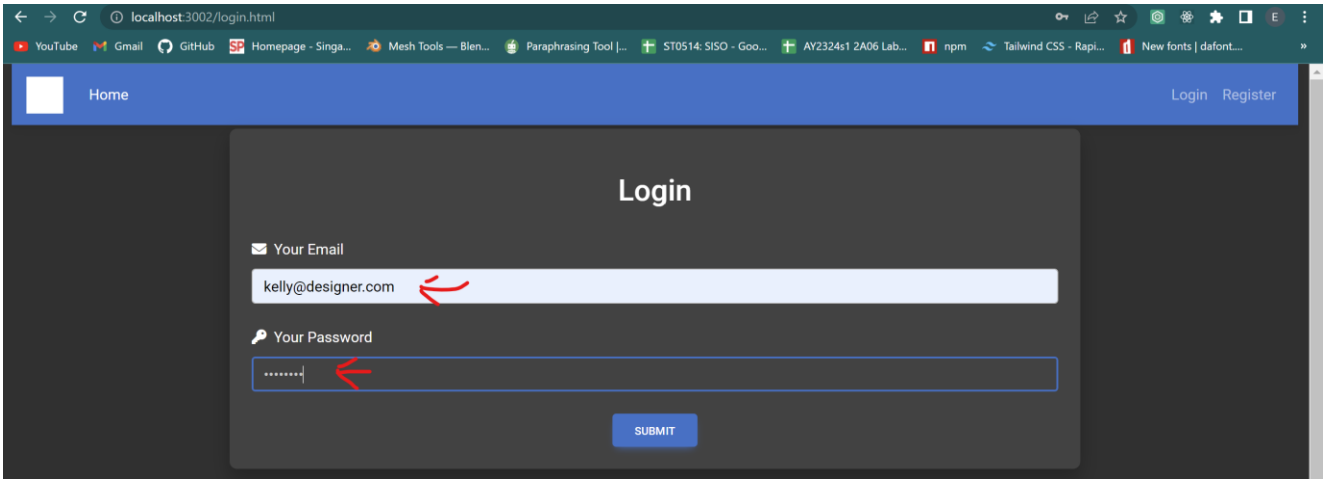
Broken Authentication is a security vulnerability that occurs when the authentication mechanisms of a web application are improperly implemented or configured, allowing attackers to bypass or compromise user authentication and gain unauthorized access to privileged resources or accounts.

When authentication mechanisms are broken or flawed, it can lead to various types of attacks like Credential Stuffing, Brute Force Attacks, Session Hijacking, Account Enumeration, and Password Reset Attacks.

3.2 How to exploit vulnerability

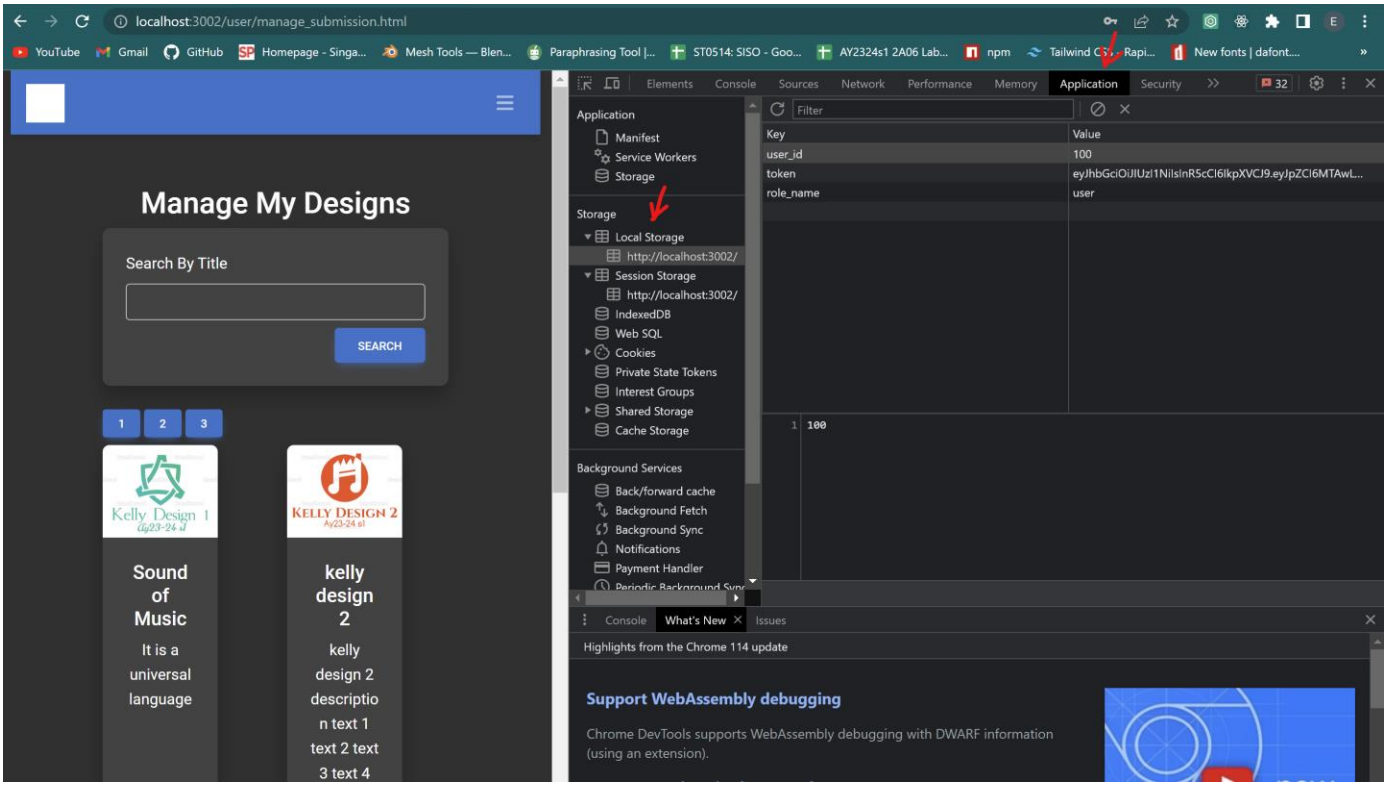
Step 1:

Login to the vulnerable website using Kelly's email and password.



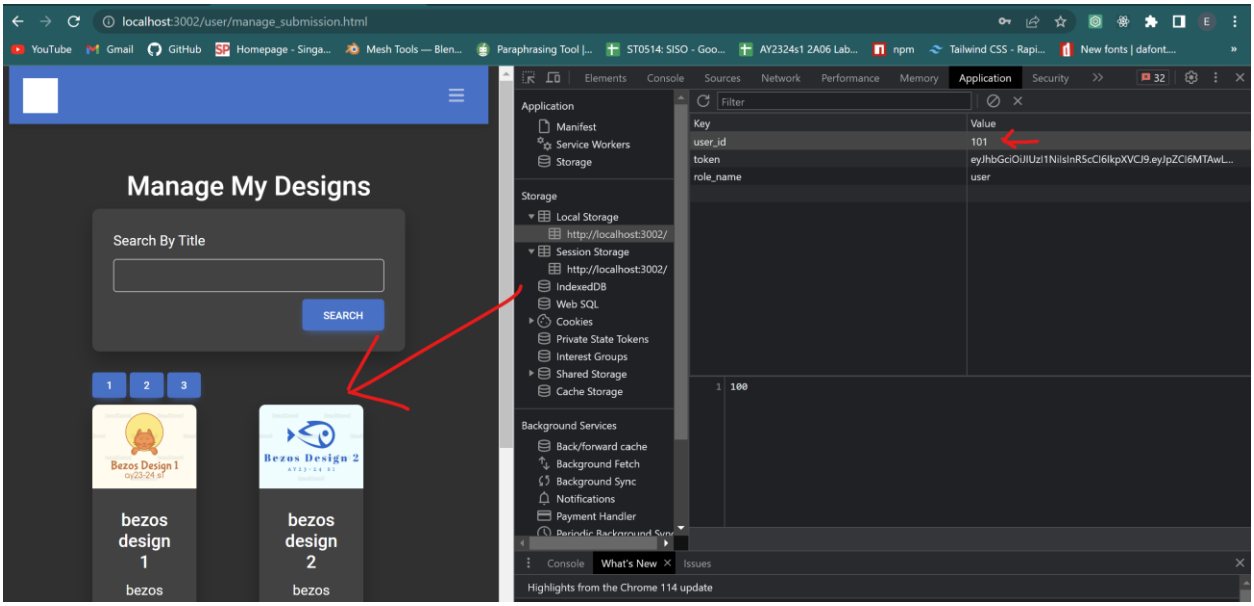
Step 2:

Right Click and open the inspect element as shown below and navigate to Application > Local Storage.



Step 3:

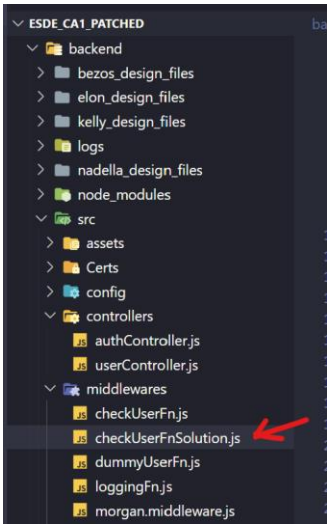
Let’s modify the user_id from 100 to 101 to see if we can view Bezos’s submissions. After modifying, click on search and you should be able to see bezos’s submissions.



3.3 How to rectify vulnerability

Step 1:

Let’s go over to the backend folder and look for checkuserFnSolution.js under /backend/src/middlewares/checkUserFnSolution.js.



Step 2:

Add the following code so that we will use the token to get the user id rather than retrieving from the storage where we can manipulate the value.

```
module.exports.checkForValidUser = (req, res, next) => {
  const authHeader = req.headers["authorization"];
  // check if authorization header is null, undefined or does not contain Bearer
  if (
    authHeader === null ||
    authHeader === undefined ||
    !authHeader.startsWith("Bearer ")
  ) {
    console.log("req.headers.authorization is undefined");
    return res
      .status(403)
      .send({ message: "Unauthorised! You are not logged in as user." });
  } else {
    // Retrieve the authorization header and parse out the
    // JWT using the split function
    console.log("retrieving authorization header");
    let token = authHeader.split(" ")[1];

    jwt.verify(token, config.JWTKey, (err, data) => {
      console.log("data extracted from token \n", data);
      if (err) {
        console.log(err);
        return res.status(403).send({ message: "Unauthorised Access" });
      } else {
        req.body.userId = data.id;
        next();
      }
    });
  }
};
```

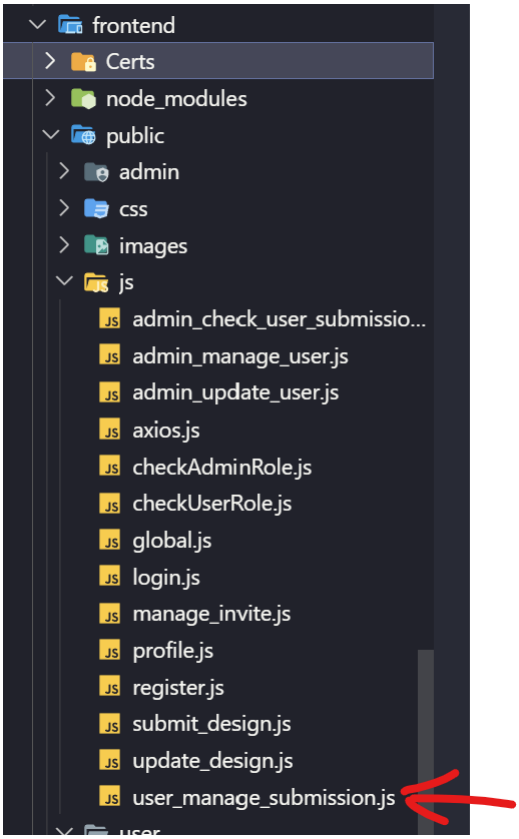
Step 3:

Head over to routes.js and we will implement this middleware to process-search-design endpoint. This is so that we can make sure that only the logged in user can view their own submission.

```
27  // USER SEARCH SUBMISSION
28  router.get(
29    "/api/user/process-search-design/:pagenumber/:search?",
30    checkUserFnSolution.checkForValidUser,
31    validateFn.validateUserSearchOwnSubmission,
32    userController.processGetSubmissionData
33  );
```

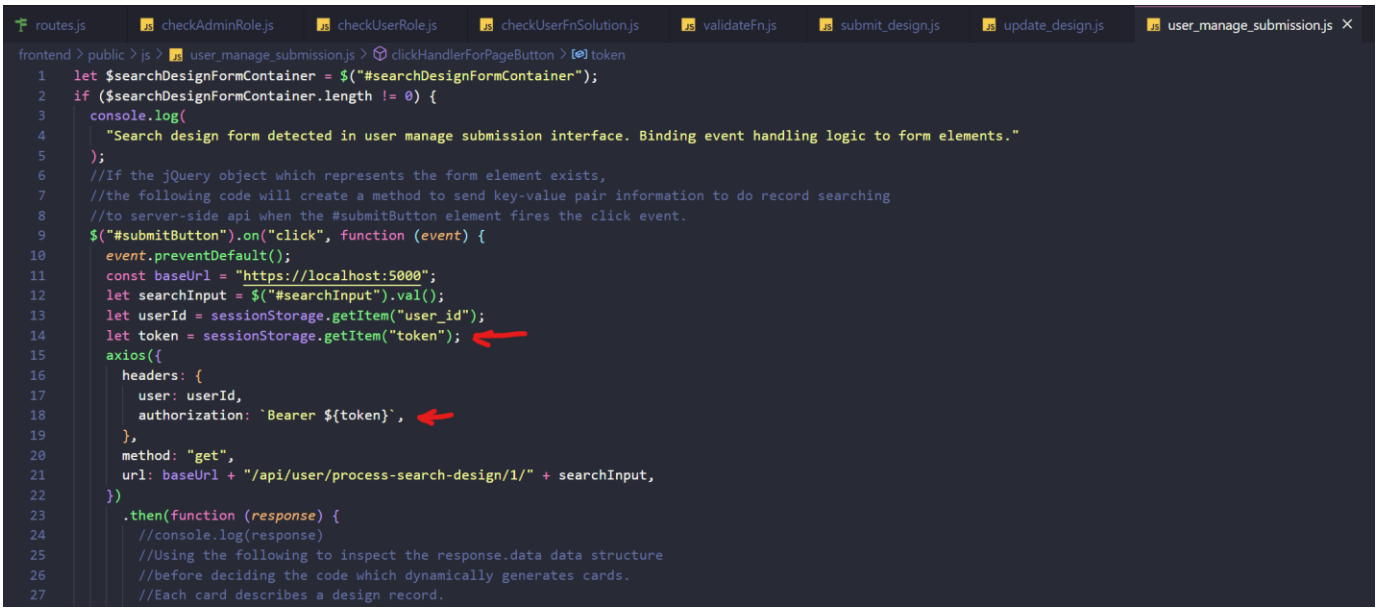
Step 4:

Go to the front end folder and look for user_manage_submission.js under frontend/public/js/user_manage_submission.js.

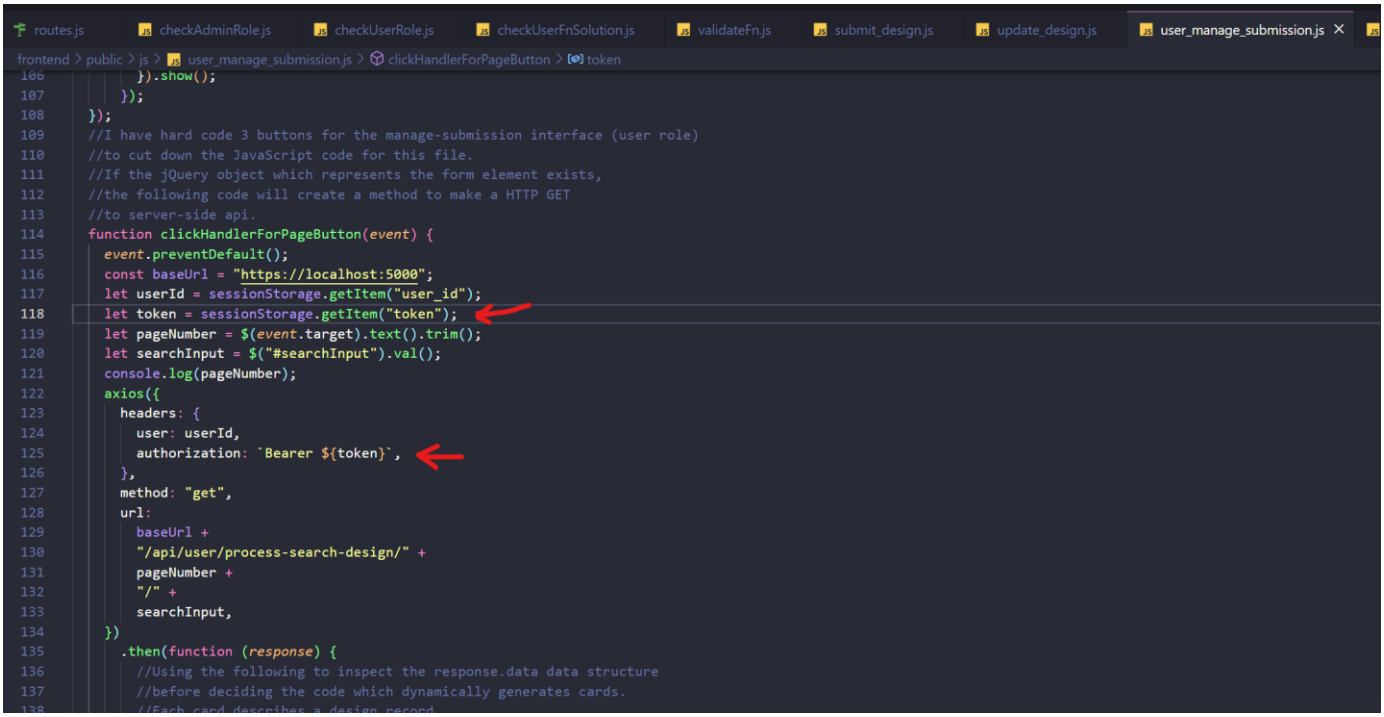


Step 5:

Add the following into the headers for us to retrieve and use the token from the storage from the frontend.



```
routes.js | checkAdminRole.js | checkUserRole.js | checkUserFnSolution.js | validateFn.js | submit_design.js | update_design.js | user_manage_submission.js
frontend > public > js > user_manage_submission.js > clickHandlerForPageButton > token
let $searchDesignFormContainer = $("#searchDesignFormContainer");
if ($searchDesignFormContainer.length != 0) {
  console.log(
    "Search design form detected in user manage submission interface. Binding event handling logic to form elements."
  );
  //If the jQuery object which represents the form element exists,
  //the following code will create a method to send key-value pair information to do record searching
  //to server-side api when the #submitButton element fires the click event.
  $("#submitButton").on("click", function (event) {
    event.preventDefault();
    const baseUrl = "https://localhost:5000";
    let searchInput = $("#searchInput").val();
    let userId = sessionStorage.getItem("user_id");
    let token = sessionStorage.getItem("token");
    axios({
      headers: {
        user: userId,
        authorization: `Bearer ${token}`,
      },
      method: "get",
      url: baseUrl + "/api/user/process-search-design/1/" + searchInput,
    })
      .then(function (response) {
        //console.log(response)
        //Using the following to inspect the response.data data structure
        //before deciding the code which dynamically generates cards.
        //Each card describes a design record.
```

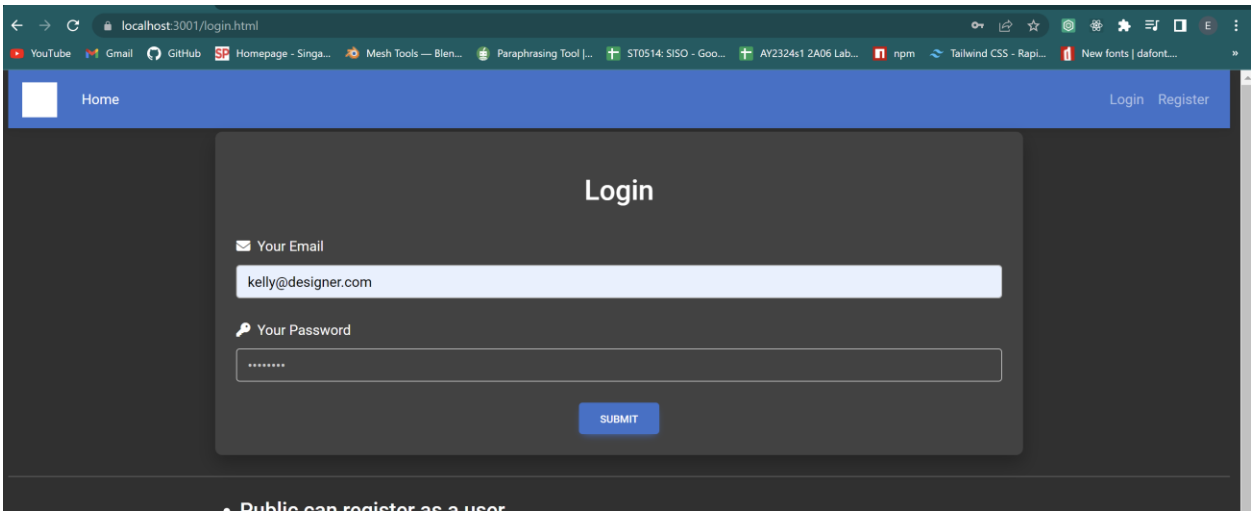


```
routes.js | checkAdminRole.js | checkUserRole.js | checkUserFnSolution.js | validateFn.js | submit_design.js | update_design.js | user_manage_submission.js
frontend > public > js > user_manage_submission.js > clickHandlerForPageButton > token
      }).show();
    });
  });
  //I have hard code 3 buttons for the manage-submission interface (user role)
  //to cut down the JavaScript code for this file.
  //If the jQuery object which represents the form element exists,
  //the following code will create a method to make a HTTP GET
  //to server-side api.
  function clickHandlerForPageButton(event) {
    event.preventDefault();
    const baseUrl = "https://localhost:5000";
    let userId = sessionStorage.getItem("user_id");
    let token = sessionStorage.getItem("token");
    let pageNumber = $(event.target).text().trim();
    let searchInput = $("#searchInput").val();
    console.log(pageNumber);
    axios({
      headers: {
        user: userId,
        authorization: `Bearer ${token}`,
      },
      method: "get",
      url:
        baseUrl +
        "/api/user/process-search-design/" +
        pageNumber +
        "/" +
        searchInput,
    })
      .then(function (response) {
        //Using the following to inspect the response.data data structure
        //before deciding the code which dynamically generates cards.
        //Each card describes a design record.
```

3.4 Evidence that the vulnerability has been rectified

Step 1:

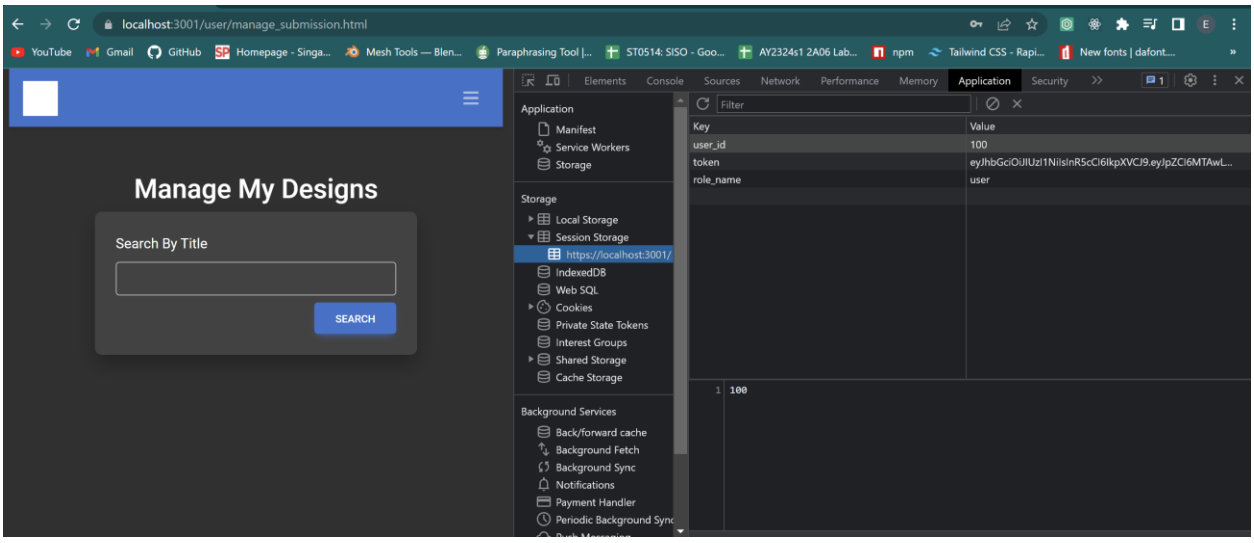
Login to the vulnerable website using Kelly's email and password.



Step 2:

Right Click and open the inspect element as shown below and navigate to Application > Local Storage.

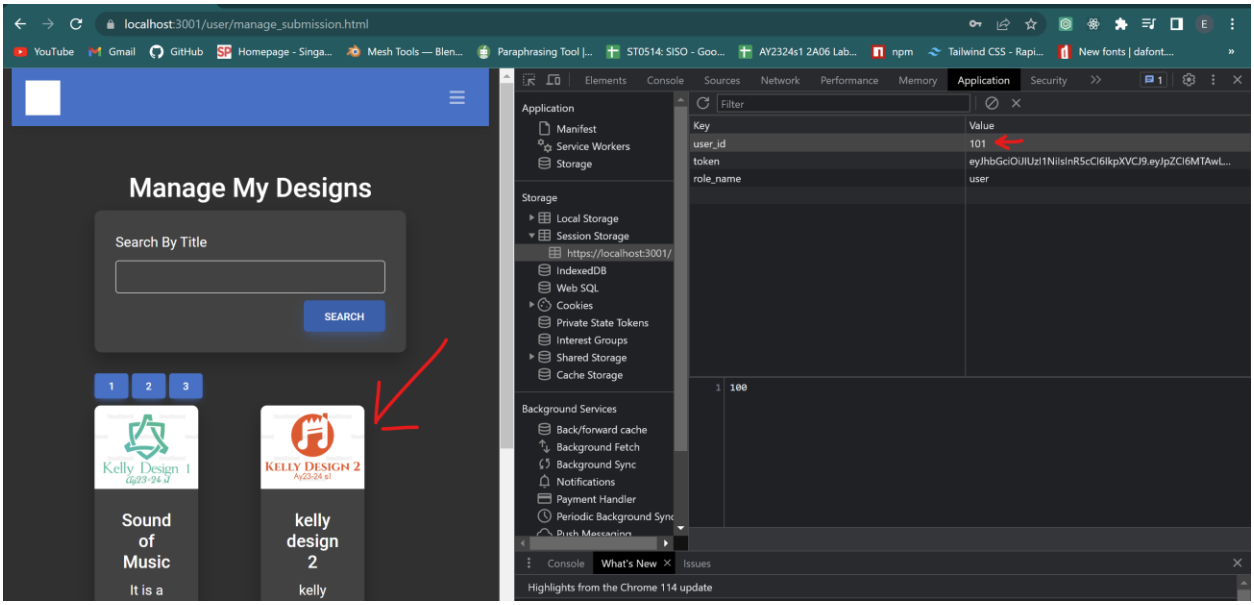
*Note: I used Session Storage for my patched code, but you can use local storage.



Step 3:

Modify the user_id to 101 to see if we can view bezos's submissions this time.

You will see that we are still retrieving Kelly's Submissions as we used the token for the user id and not the user_id from the storage. This shows that we have successfully rectified the Broken Authentication Vulnerability.



4. Broken Access Control

4.1 Definition

Broken Access Control is a web application vulnerability that occurs when access controls and authorization mechanisms are improperly implemented, allowing unauthorized users to access restricted resources or perform actions beyond their intended privileges. It involves the failure to properly enforce restrictions on what authenticated users can access or modify within the application.

When access controls are broken or flawed, it can lead to various security issues like Unauthorized Data Access, Horizontal Privilege Escalation, Vertical Privilege Escalation, Insecure Direct Object Reference, and Mass Assignment.

4.3.1 How to exploit vulnerability - About Role access control

Step 1:

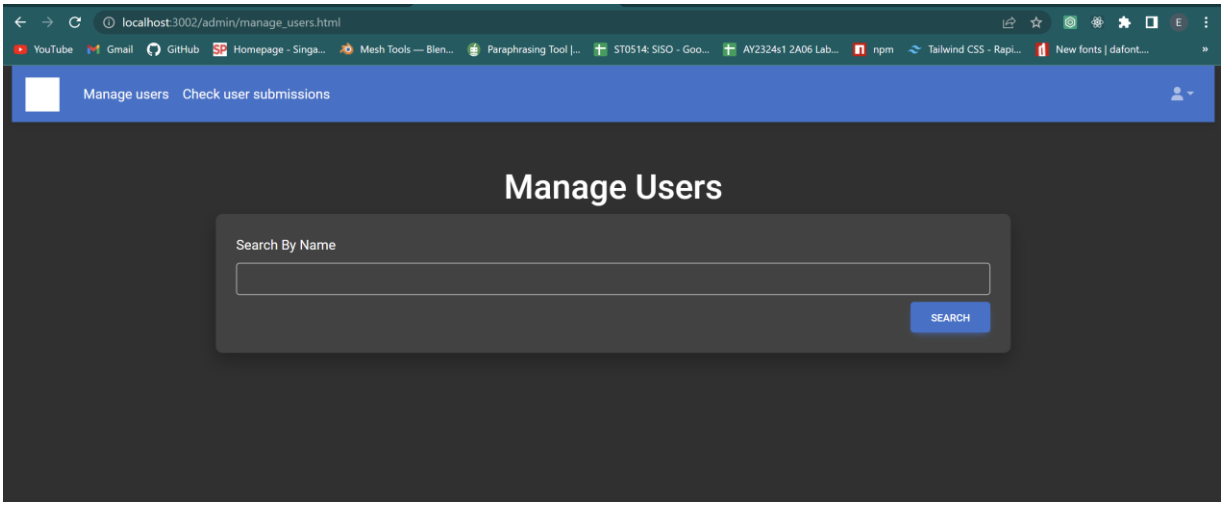
Change the search URL above to the following.

yourLocalhostPort/admin/manage_users.html

Step 2:

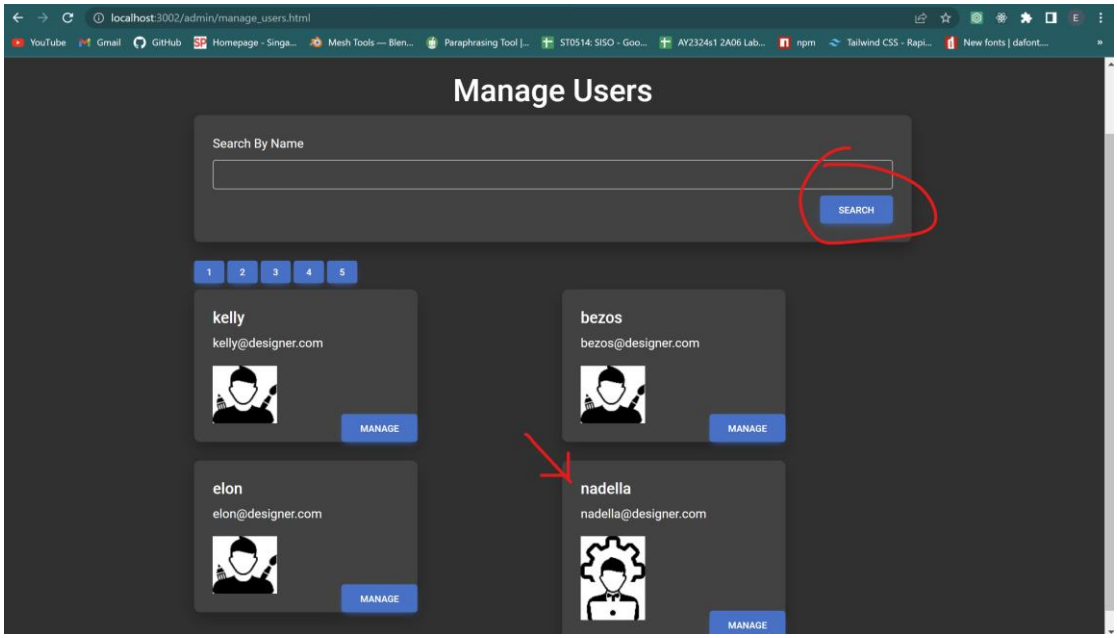
You should be brought to the admin page which we are not allowed to.

*Note: this is a 2nd vulnerability for Broken Access Control which is being rectified above under Bonus 7.2.0.



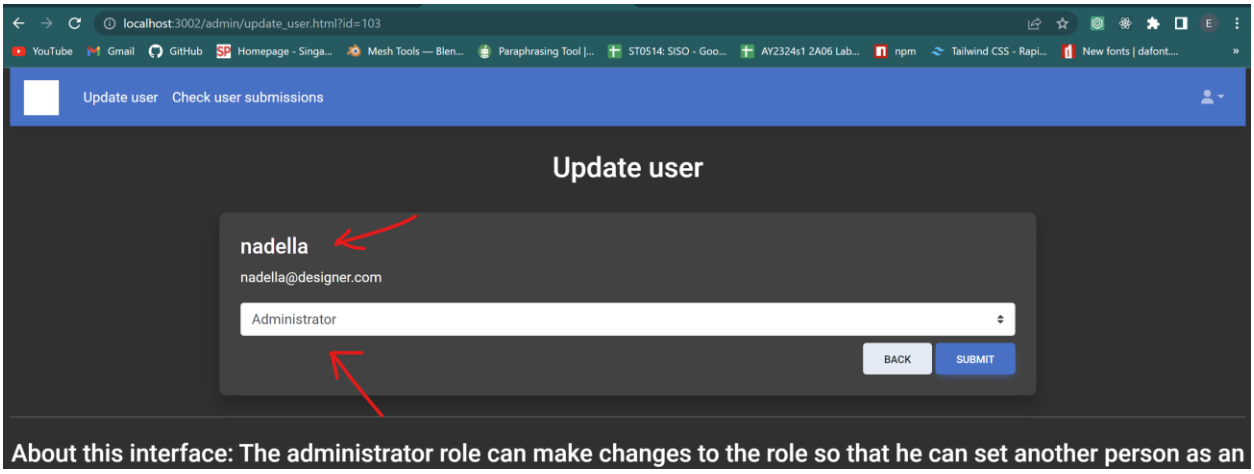
Step 3:

Click on the search button and look for a user called Nadella, click on Manage Button for Nadella to see her role.



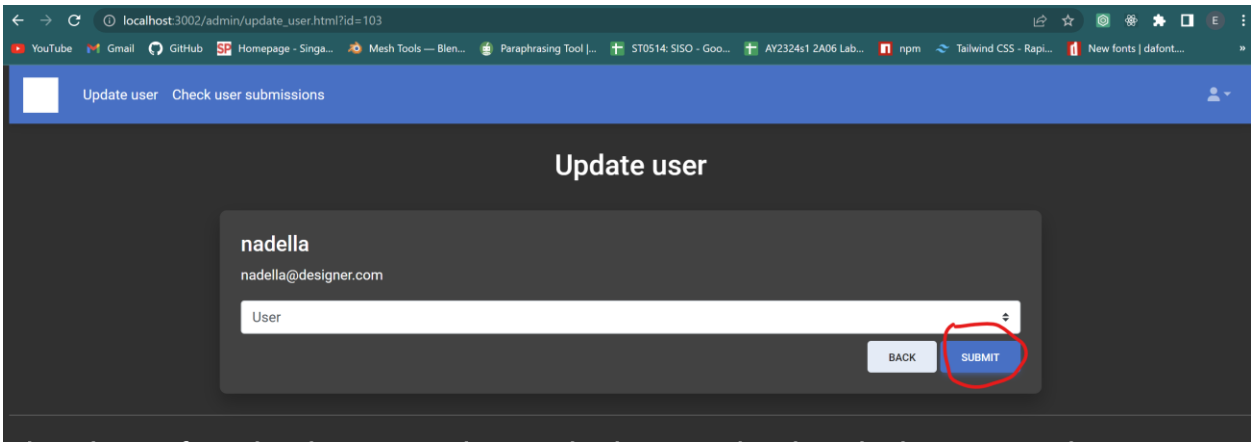
Step 4:

You should be able to see Nadella's role which is an administrator for my case.



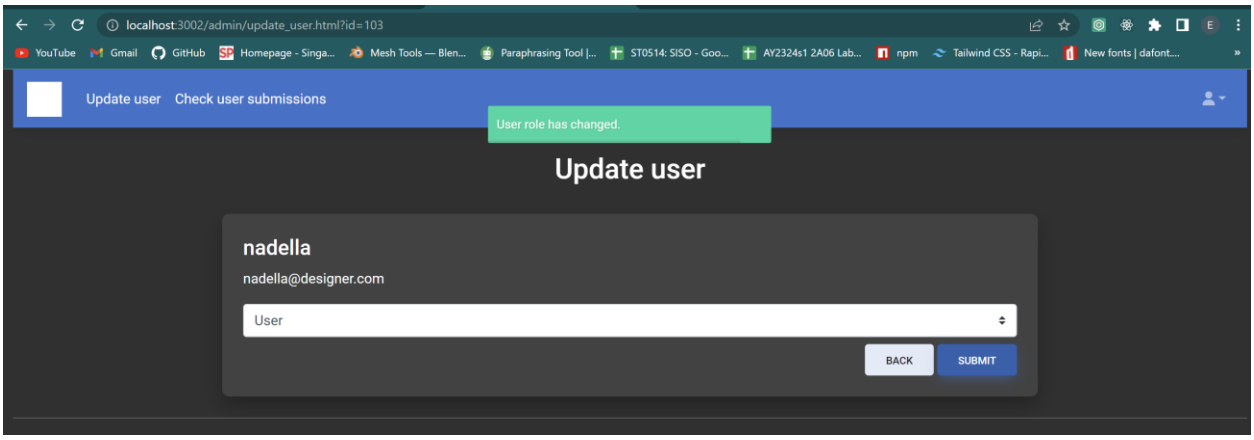
Step 5:

Let's change her role to User and submit.



Step 6:

You should see that you are able to change the role of Nadella even though we are not logged in.

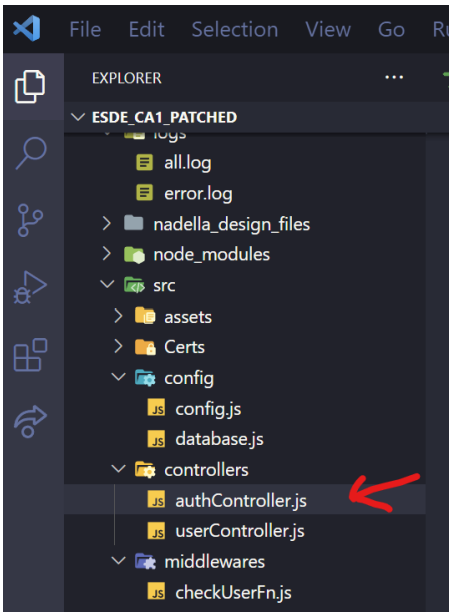


4.3.2 How to rectify vulnerability - About Role access control

Step 1:

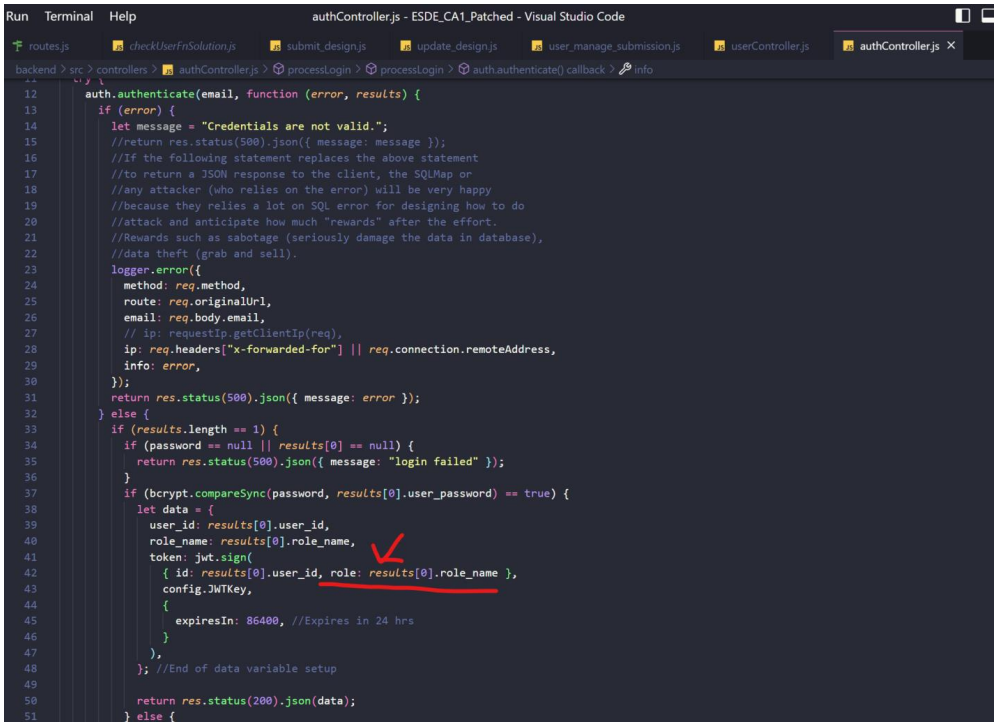
To rectify this vulnerability, let us first go to go to authController.js to do some changes to the cookie.

Under /src/controllers/authController.js



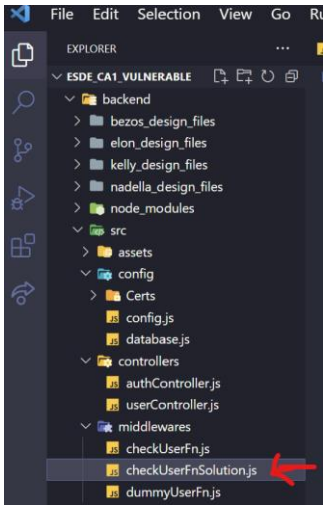
Step 2:

Apply the following changes to the exports.processLogin under /src/controllers/authController.js. This will allow us to check the role of the current user to see if they are valid to perform any admin actions from the token.



Step 3:

Head over to checkUserFnSolution.js to add a check admin role middleware. Under /src/middlewares/ checkUserFnSolution.js



Step 4:

Add the following code to create a middleware to check if the current logged in user is an admin or not by using the token.

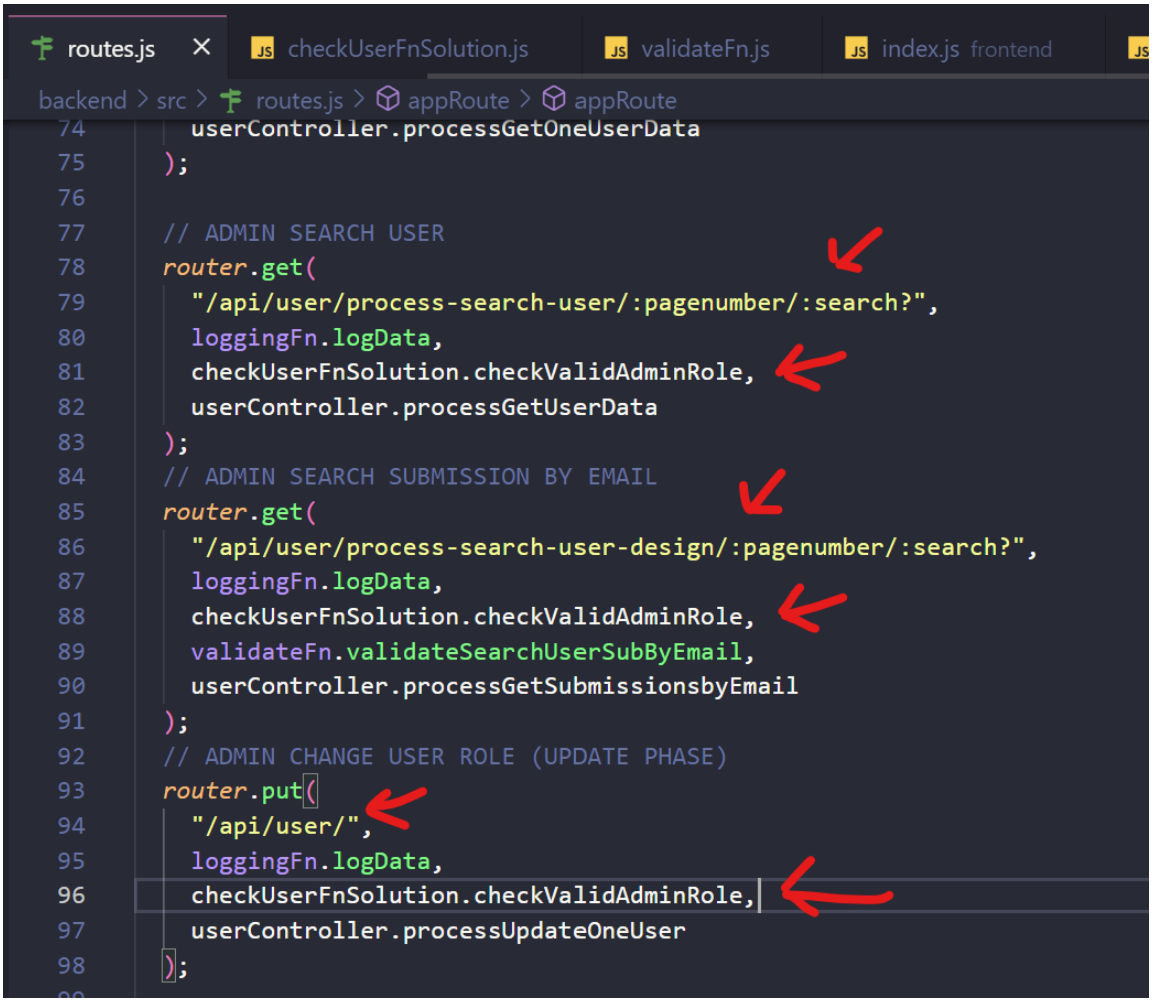
routes.js | checkUserFnSolution.js | submit_design.js | update_design.js | user_manage

backend > src > middlewares > checkUserFnSolution.js > checkValidAdminRole > checkValidAdminRole

```
70
71  module.exports.checkValidAdminRole = (req, res, next) => {
72    console.log("http header - admin ", req.headers["user"]);
73    console.log("authHeader : " + req.headers["authorization"]);
74    const authHeader = req.headers["authorization"];
75    if (
76      authHeader === null ||
77      authHeader === undefined ||
78      !authHeader.startsWith("Bearer ")
79    ) {
80      console.log(
81        "authHeader is null or undefined or does not start with Bearer"
82      );
83      return res.status(403).send({ message: "Unauthorized access" });
84    } else {
85      //Retrieve authorization header and parse out the JWT using the split function
86      console.log("Retrieving authorization header");
87      let token = authHeader.split(" ")[1];
88      console.log("Check for received token from frontend : \n");
89      console.log(token);
90
91      jwt.verify(token, config.JWTKey, (err, data) => {
92        console.log("data extracted from token \n", data);
93        if (err) {
94          console.log(err);
95          return res.status(403).send({ message: "Unauthorized access" });
96        } else {
97          console.log("data: " + data.role);
98          if (data.role == "admin") {
99            console.log("User is admin");
100           next();
101         } else {
102           console.log("User is not admin");
103           return res.status(403).send({
104             message: "Unauthorized access! You are not an admin.",
105           });
106         }
107       }
108     });
109   }
110 }; //End of checkValidAdminRole
111
```

Step 5:

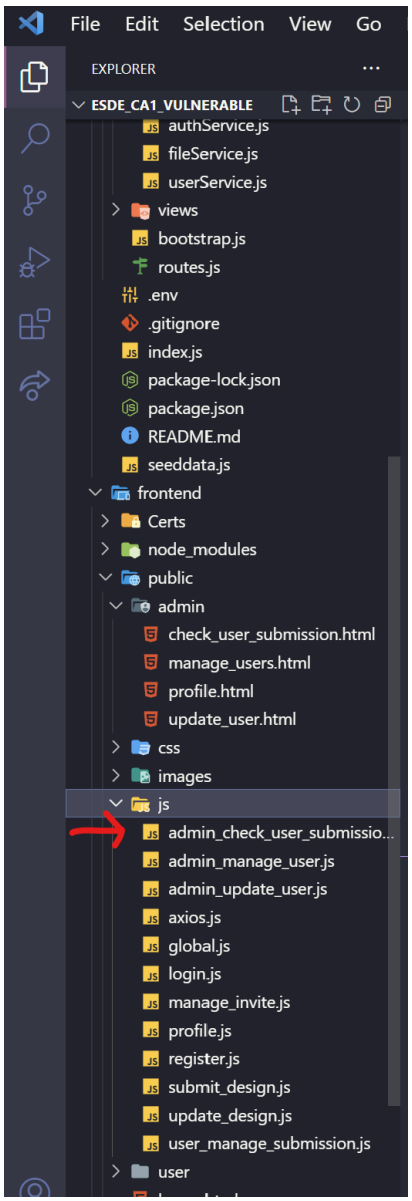
Next, head over to routes.js under /backend/src/routes.js where we will use the middleware we just created to the admin endpoints. Check that you have added the middleware to the correct endpoints with checkUserFnSolution.checkValidAdminRole.



Step 6:

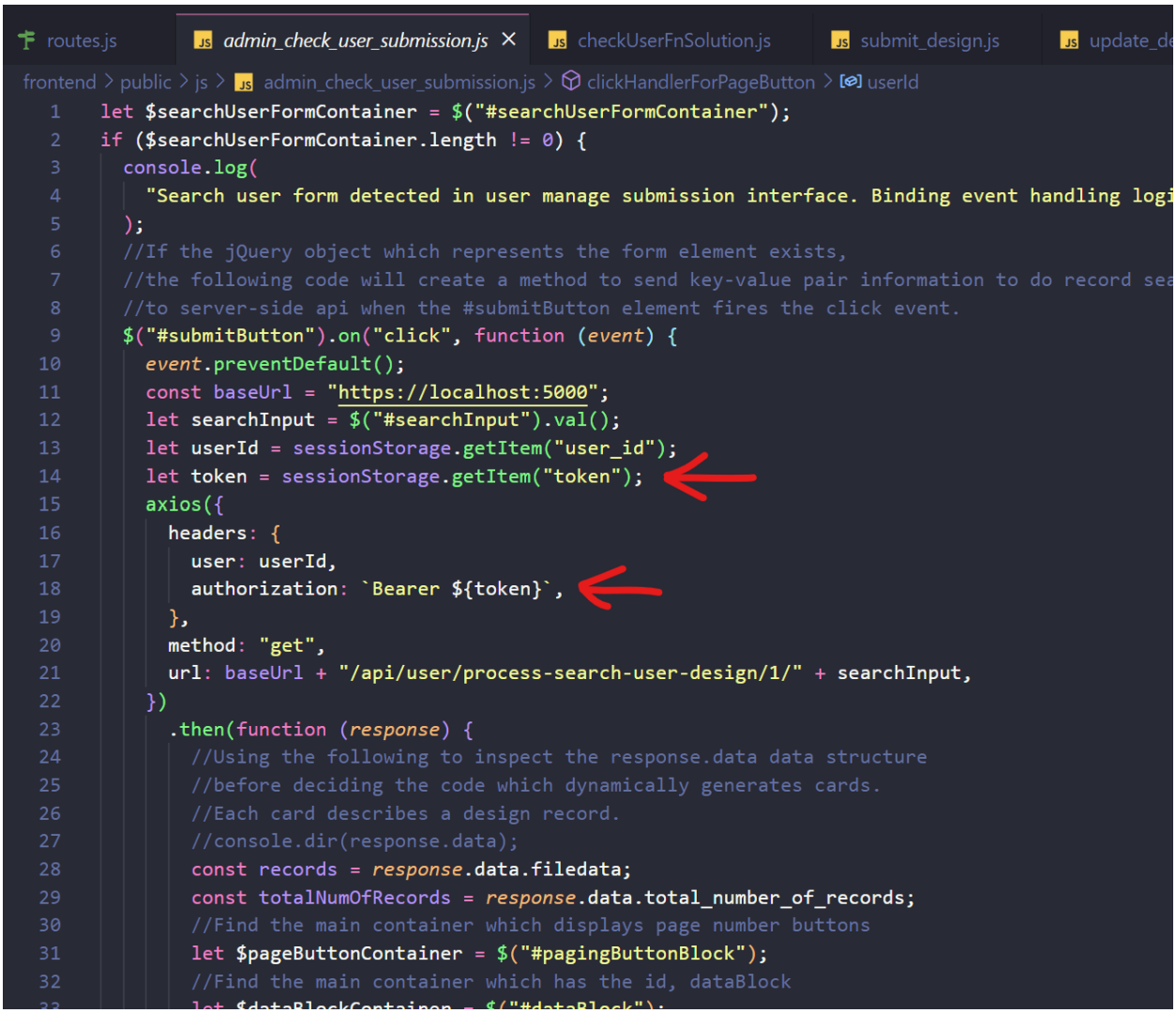
Next, go over to the Frontend Directory and look for admin_check_user_submission.js.

Under /frontend/public/js/admin_check_user_submission.js



Step 7:

We will now add a code to retrieve the token from the local storage and place it in the headers as shown below.

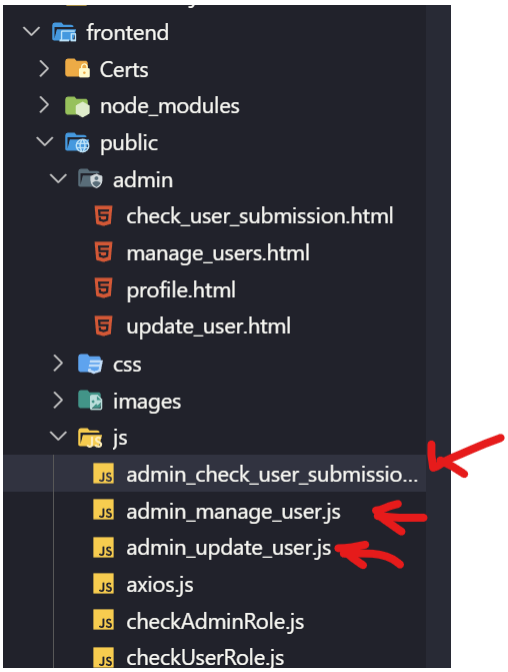


Step 8:

Continue doing the above changes for all the admin js files that uses axios like admin_check_user_submission, admin_manage_user, and admin_update_user.

*Note: There should be a total of six changes to the 3 files including the above changes.

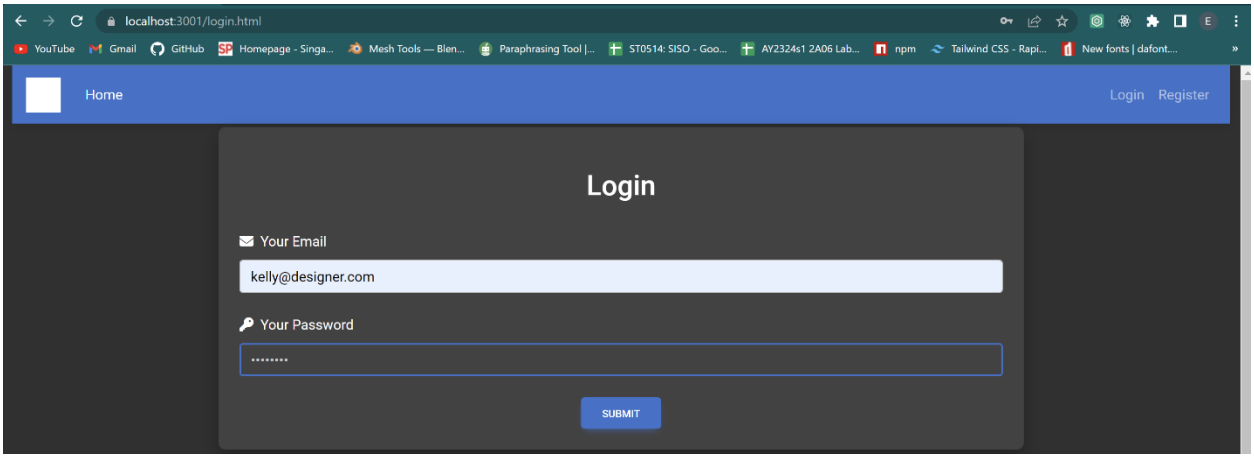
Under /frontend/public/js/.



4.4.2 Evidence that the vulnerability has been rectified - About Role access control

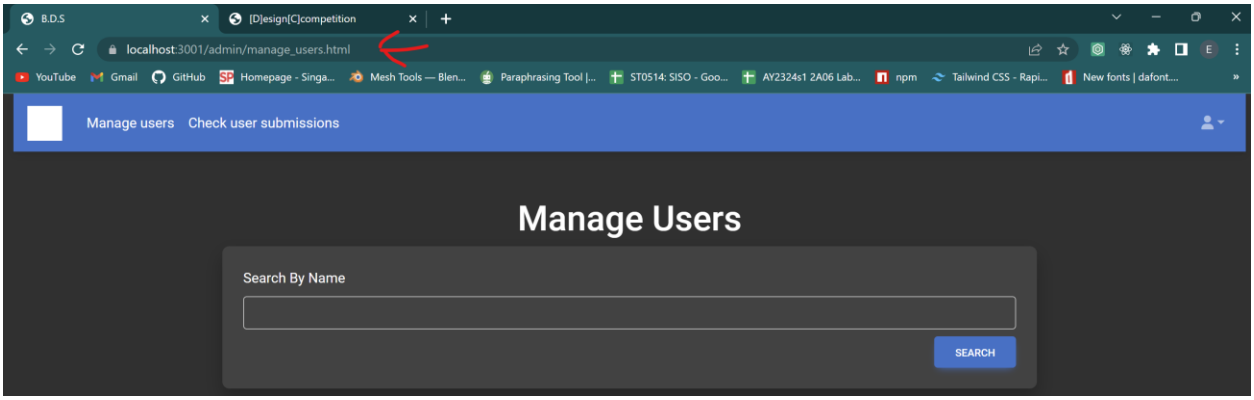
Step 1:

Login using Kelly's email and password in the Login Page.



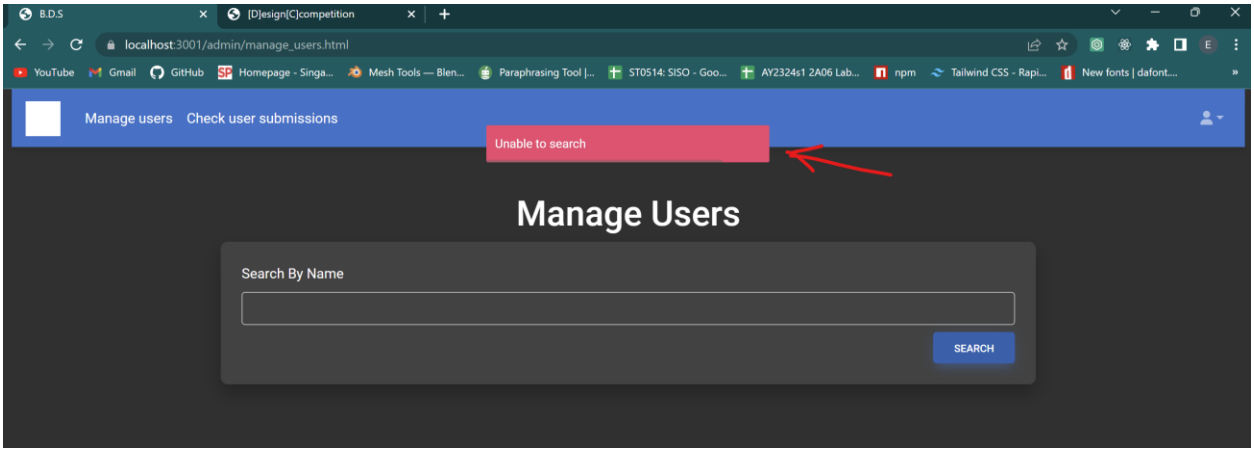
Step 2:

Change the above URL link to yourLocalhostPort/admin/manage_users.html.



Step 3:

Now, after the applying the checkValidAdminRole middleware to all the admin endpoints, you will immediately see that upon pressing the search button it will show an alert saying, “Unable to search”. This shows that we have successfully validated the role of the user and not allow non-admin users to do any admin right actions even though they can access the admin page.



5. Sensitive Data Exposure

5.1 Definition

Sensitive Data Exposure is a security vulnerability where confidential or sensitive information is exposed to unauthorized individuals or entities due to inadequate protection or improper handling of the data. It occurs when sensitive data, such as personal information, financial details, passwords, or intellectual property, is stored, transmitted, or processed without appropriate security measures in place.

This can occur through various means like Insecure Data Storage, Inadequate Transmission Security, Weak Authentication and Authorization, Poorly Configured Security Controls, and Vulnerabilities in Applications or Systems.

5.2 How to exploit vulnerability

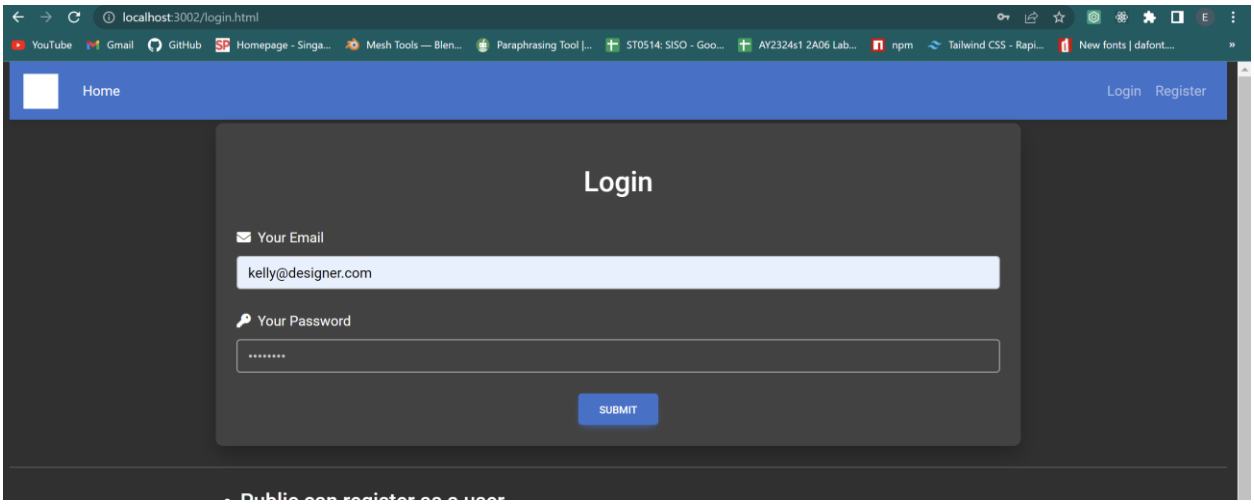
Step 1:

Install Wireshark as we will be using Wireshark for this vulnerability.

Link: https://www.wireshark.org/download.html

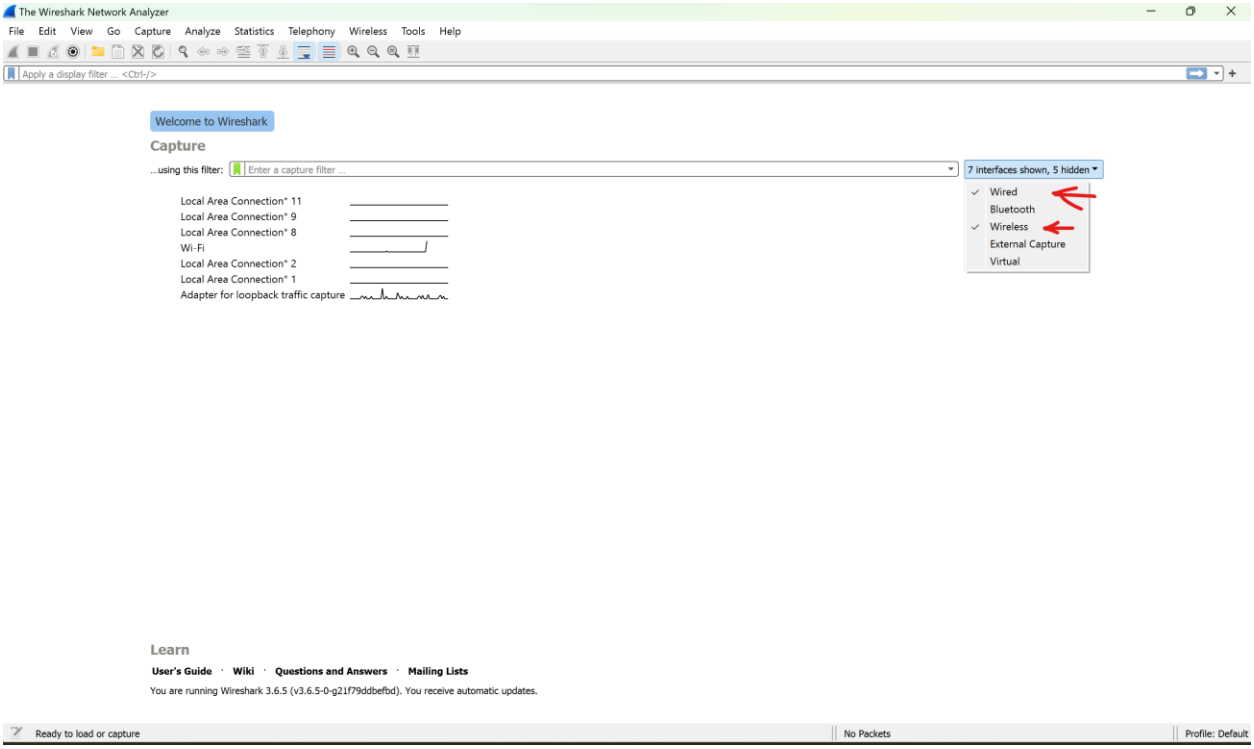
Step 2:

Head over to the vulnerable website fill in Kelly's Email and Password as shown below **but do not press submit**.



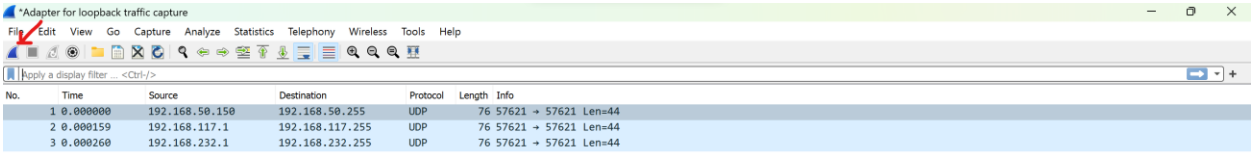
Step 3:

Head over to Wireshark where you will first see the image shown below and make sure that you have checked Wired and Wireless.



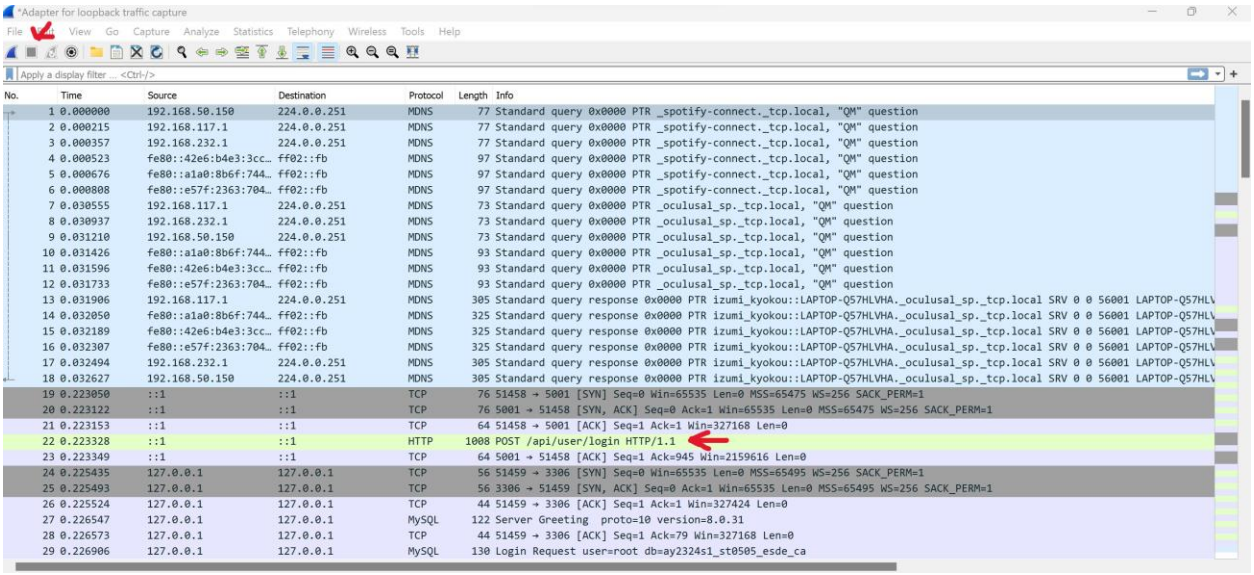
Step 4:

Next click on Adapter for loopback traffic capture and press the shark fin looking icon on the top left to begin capturing the traffic.



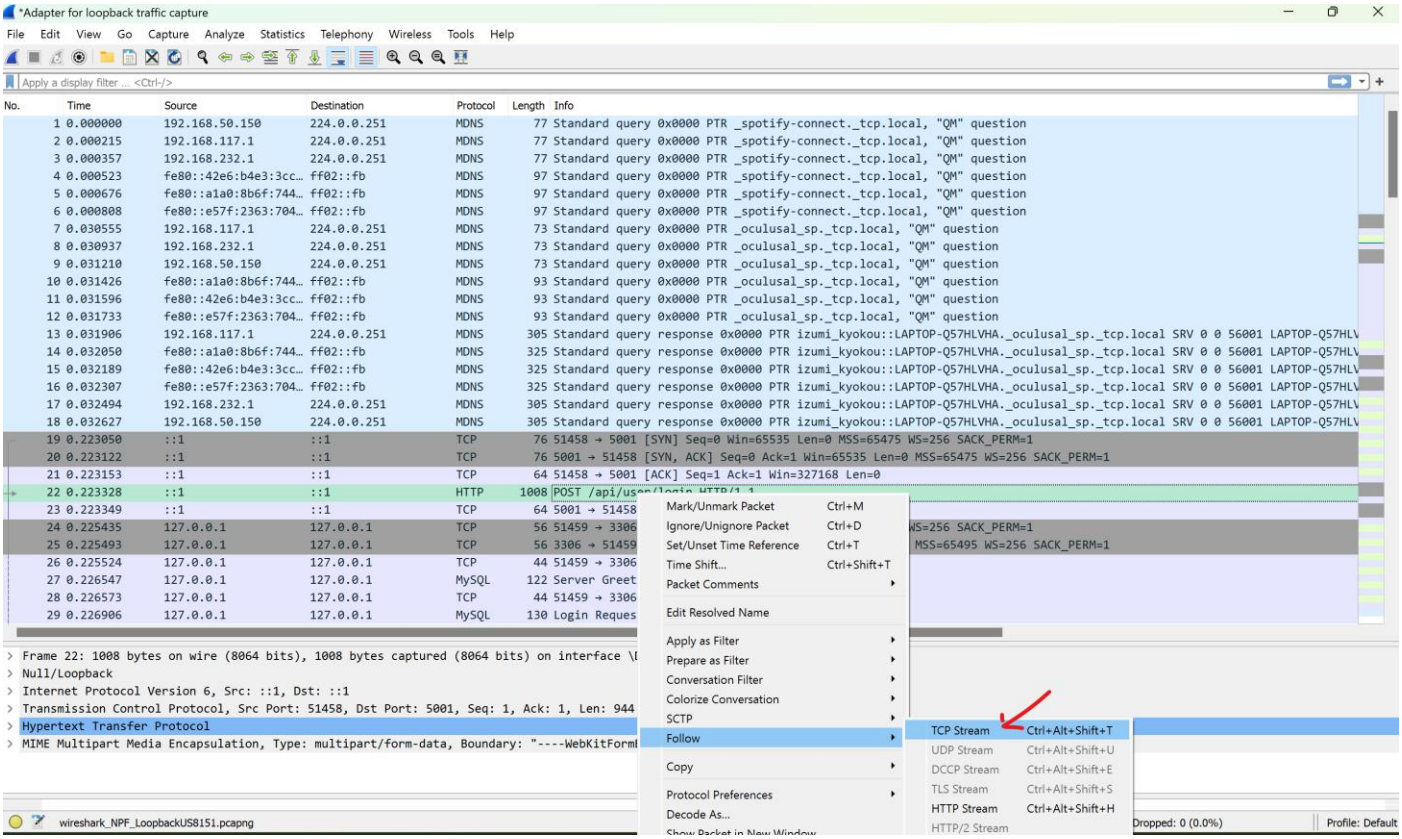
Step 5:

Next, go back to the vulnerable website and press submit to log in and go to Wireshark and press the stop button beside the shark fin looking button. You should see a lot of traffic being captured and one of it being the POST request for our login.



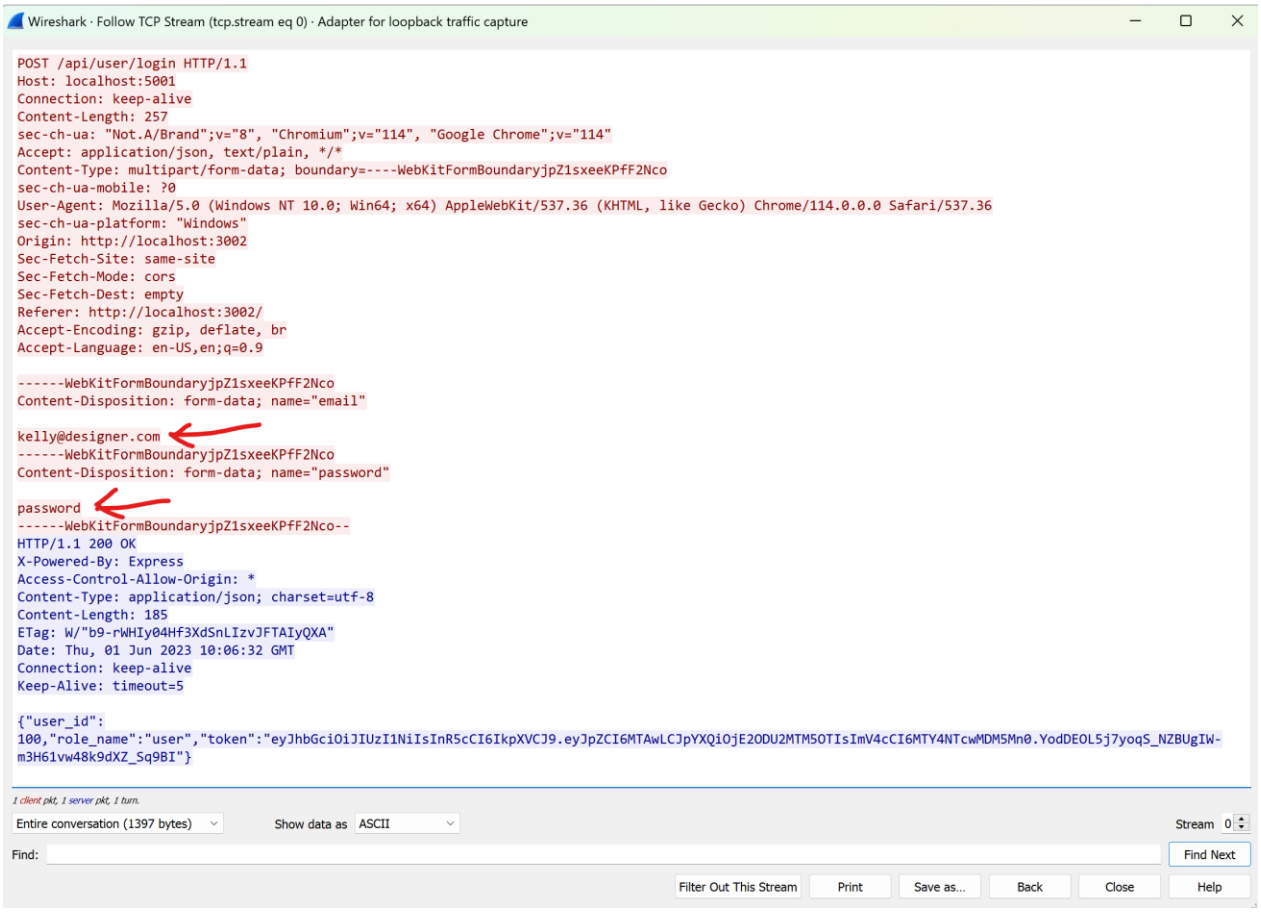
Step 6:

Next, right click on the POST request and select Follow > TCP Stream.



Step 7:

You should be able to see the login details in plain text as shown below.



5.3 How to rectify vulnerability

Step 1:

Let us install chocolatey so that we can install mkcert to make our site secure.

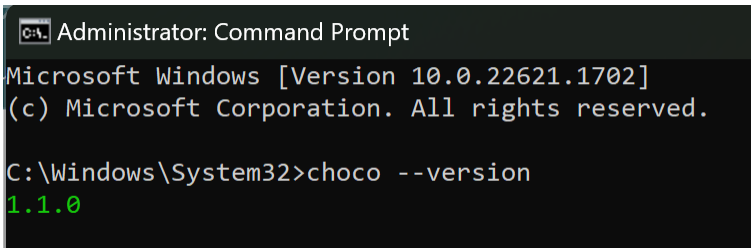
Type the following into your **Administrative Command Prompt.**

@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile -InputFormat None -ExecutionPolicy Bypass -Command " [System.Net.ServicePointManager]::SecurityProtocol = 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET "PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"

Reference: https://www.liquidweb.com/kb/how-to-install-chocolatey-on-windows/

Step 2:

Check that you have chocolatey installed by typing “choco –version”. For me, the version I am using is 1.1.0.



Step 3:

Next let us install mkcert to generate a self-signed certificate for us to use later.

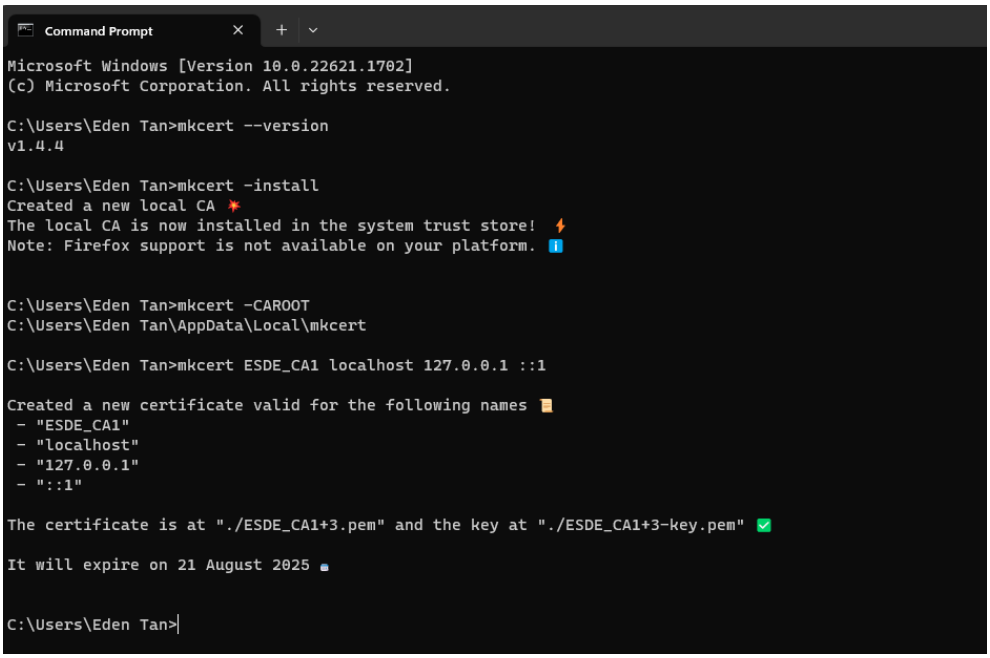
Enter the following **in order** into your command prompt.

1. choco install mkcert
2. mkcert -install
3. mkcert locahost

Your screen should look something like the picture below.

Reference: https://technixleo.com/create-locally-trusted-ssl-certificates-with-mkcert-on-windows/

*Note: The following image has extra command used to check for the install location of the mkcert like “mkcert -CAROOT”. Please use this command if you cannot find the directory of the installed mkcert.

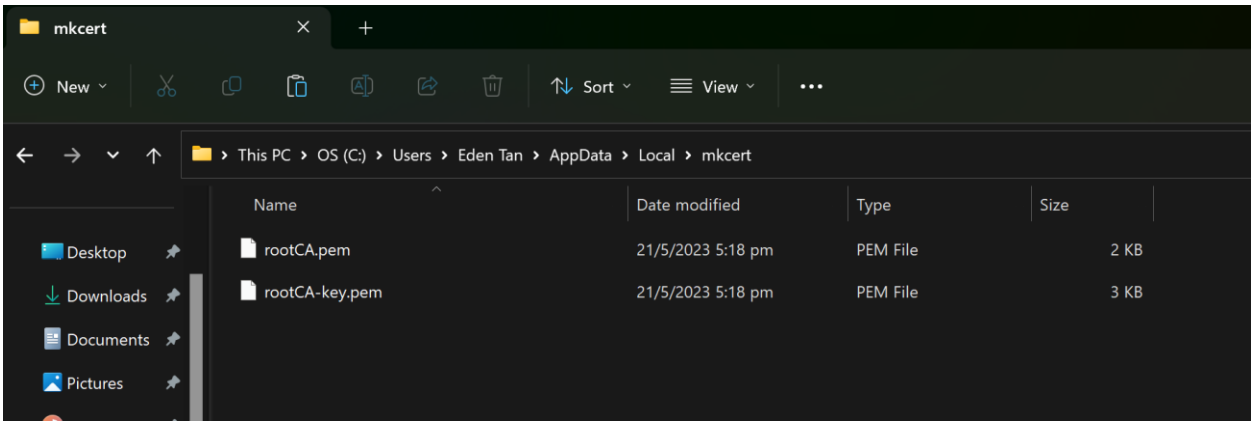


Step 4:

Let us go to the file path listed after typing the command mkcert -CAROOT.

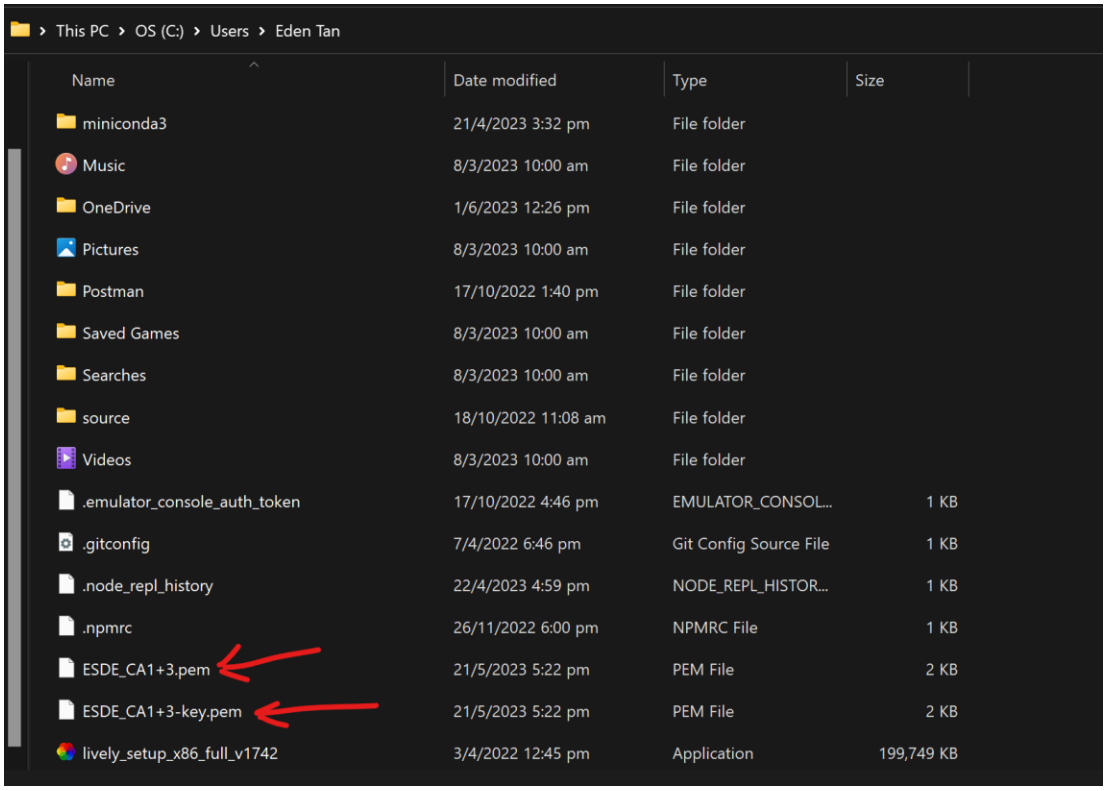
You should see a rootCA.pem file and a rootCA-key.pem file that we created from the command “mkcert -install”.

Remember the location of the rootCA.pem as we will be using it for our trusted certificate in google chrome for my case.



Step 5:

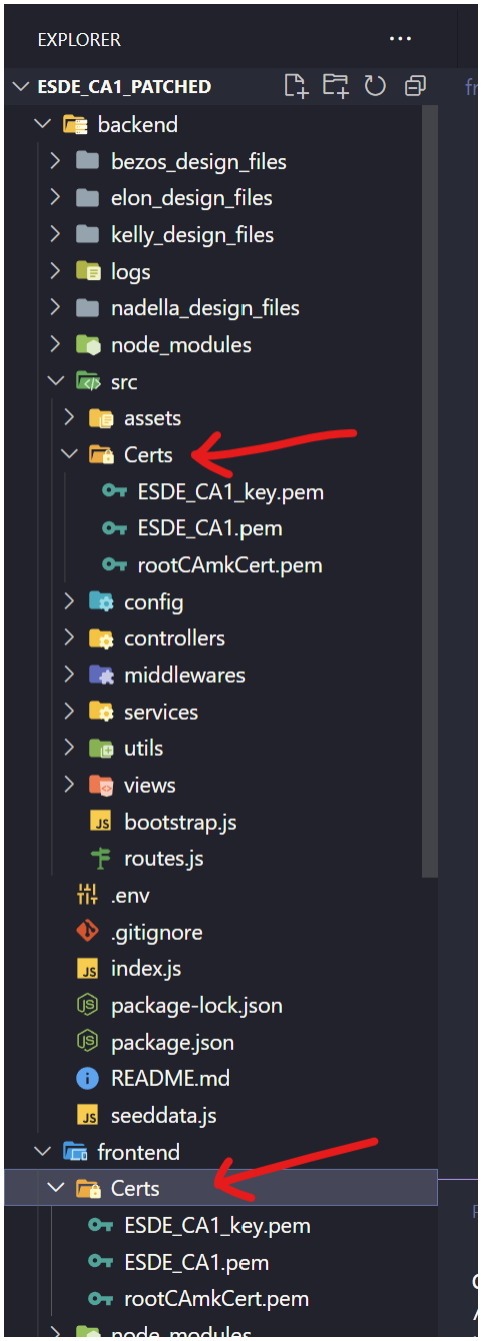
Next, we will go to the directory for my name in my case as shown below and we will see two .pem files that we can use to make our website secure which we got from the command “mkcert localhost”.



Step 6:

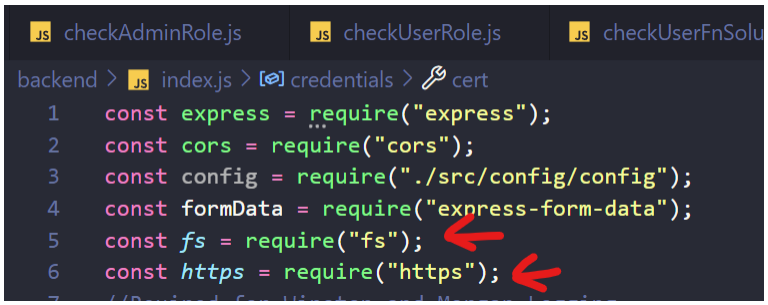
Copy the two .pem files and create a folder called Certs in both your backend and frontend folder and paste the .pem files inside.

*Note: I renamed the Cert and Key and added the rootCA.pem file here for convenience.



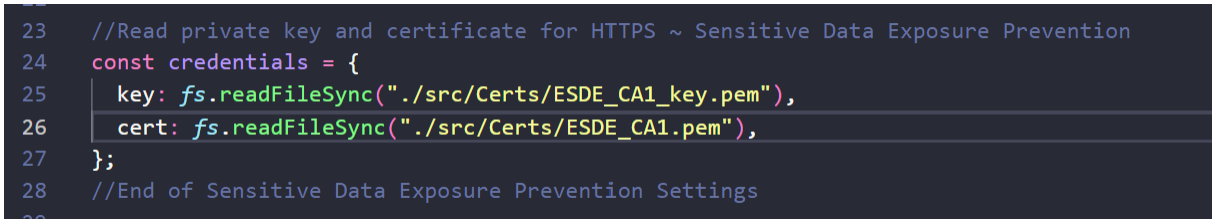
Step 7:

Next, go into index.js under /backend/src/index.js where we will use the certs we got. We will first use the built-in fs and https library from node.js.



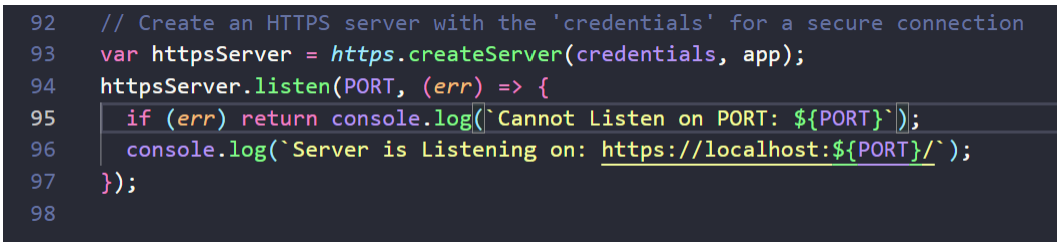
Step 8:

Next, let us add the following code to read the key and cert files.



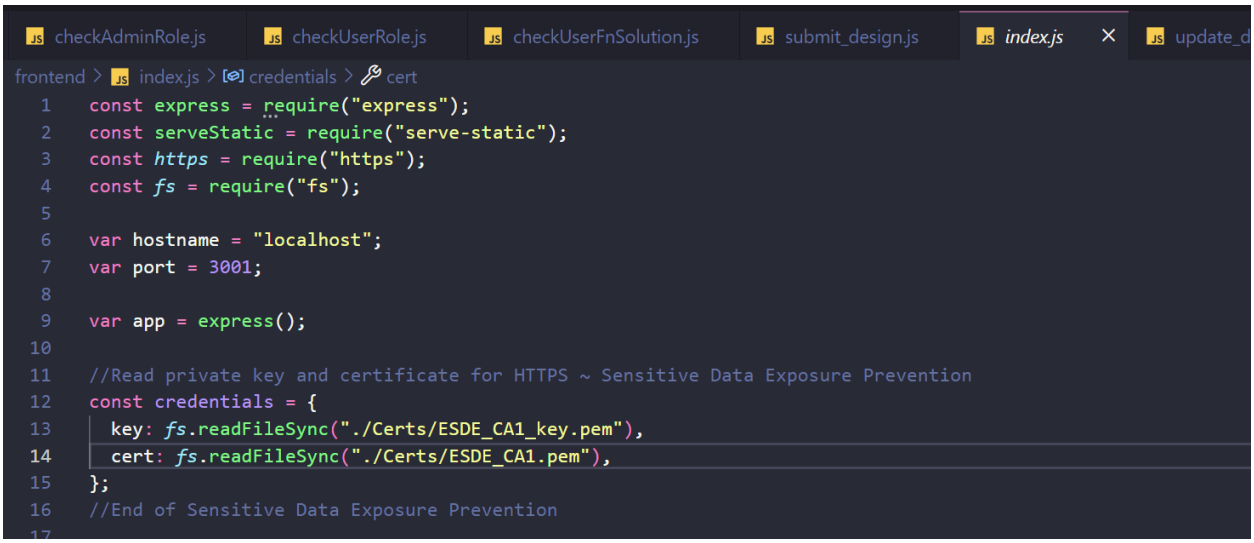
Step 9:

Next, we will add the following code at the bottom of the index.js file to create a https server for the backend.



Step 10:

Next, head over to the index.js file under /frontend/public/index.js. We will also use the built in https and fs library and create the credentials like above.



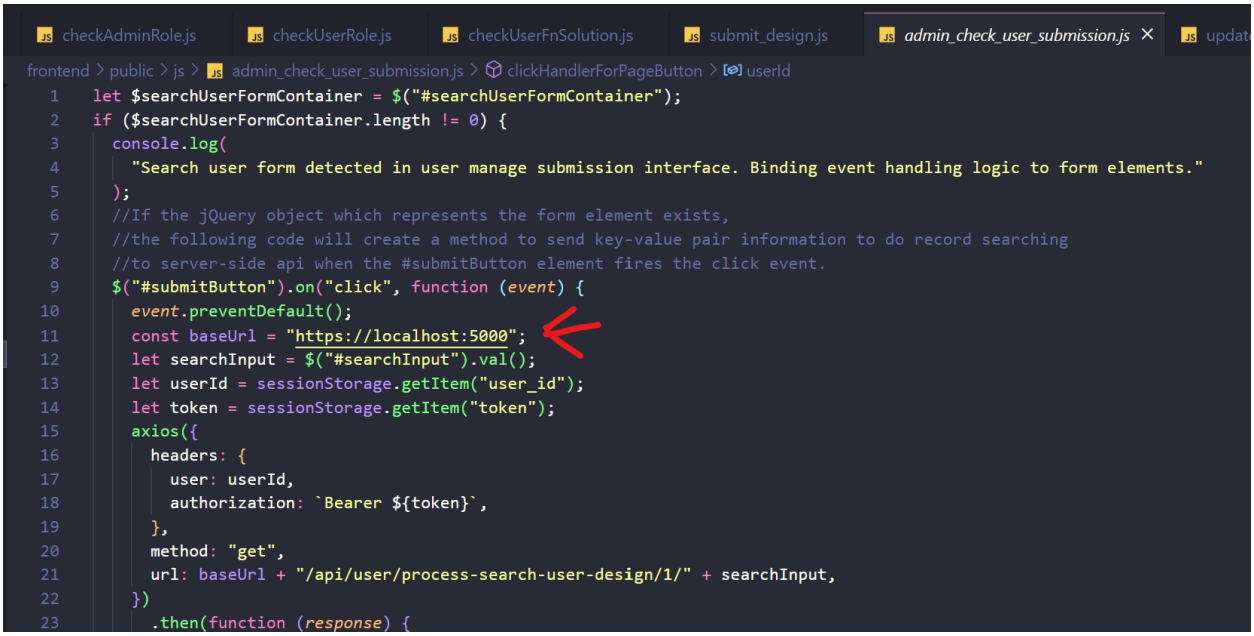
Step 11:

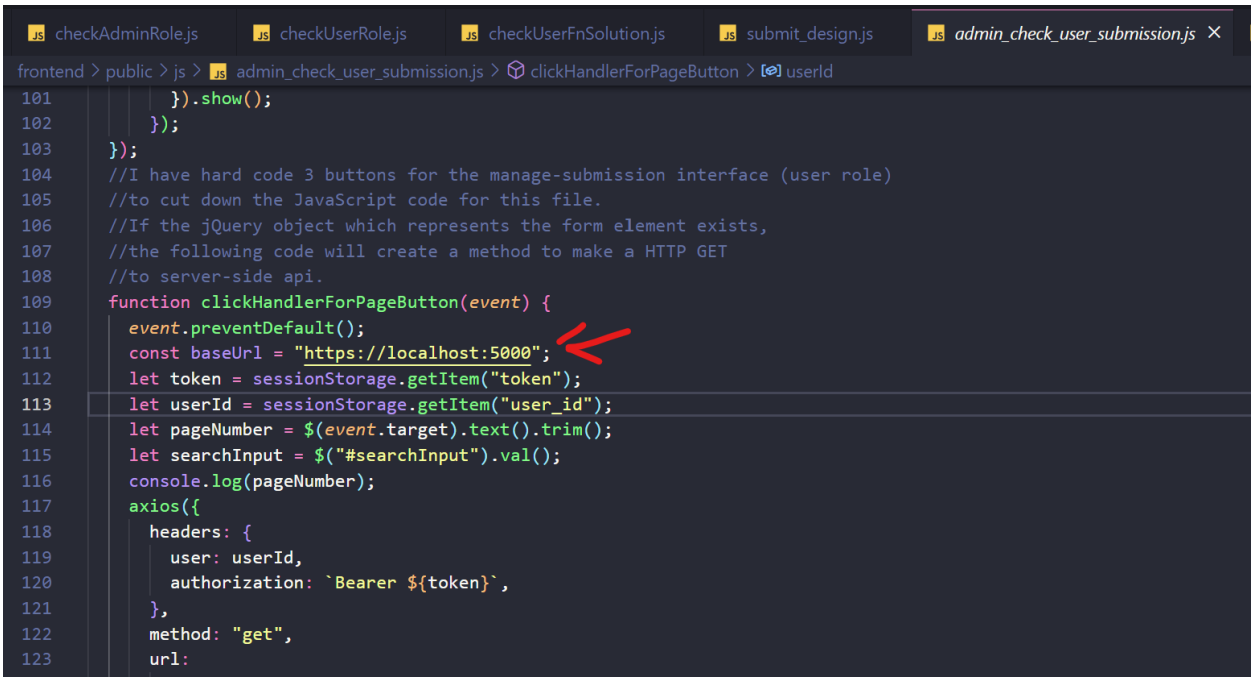
We will now add the following code at the bottom of the index.js file to secure the frontend.

Step 12:

We will now need to **change all request using http to https** in the js folder in the frontend folder. Located under /frontend/public/js/.

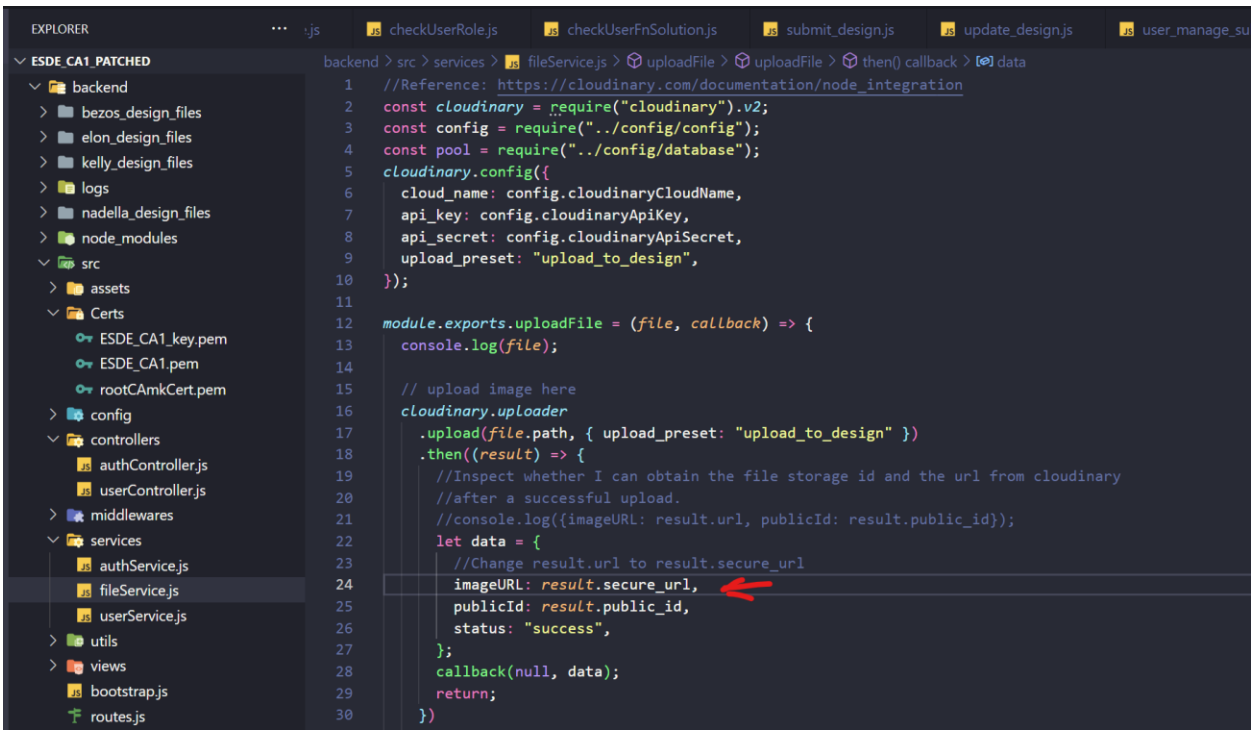
As shown below for admin_check_user_submission.js we will change all the request to https. Do the same for the other js files using http request under /frontend/public/js.





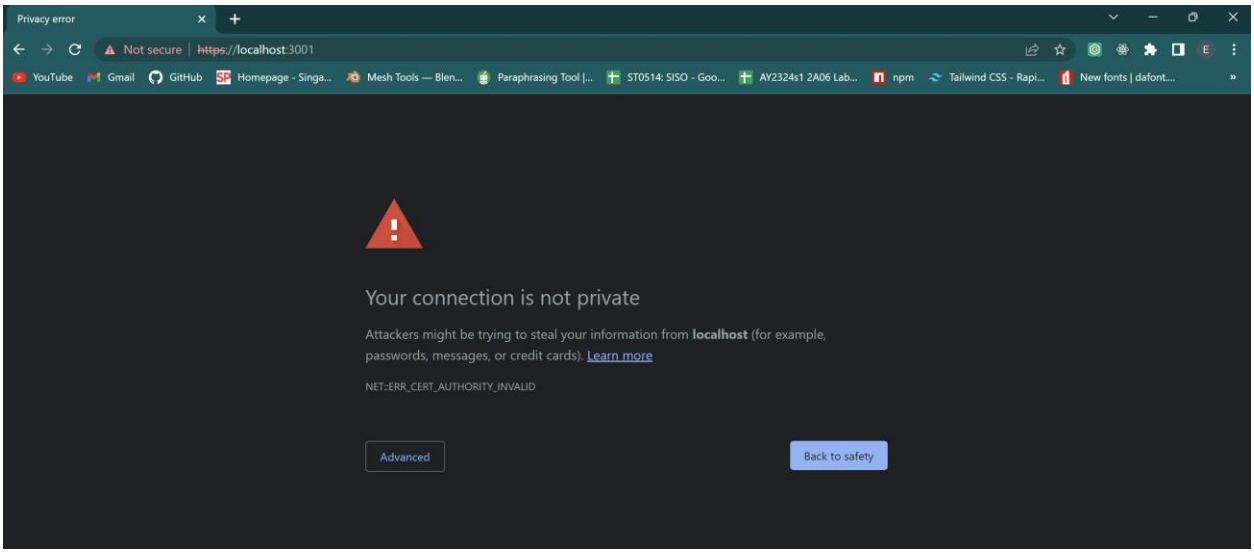
Step 13:

Let us go to fileService.js located under /backend/src/services/fileService.js and change the image link from result.url to result.secure_url to use https for our image upload too as shown below.



Step 14:

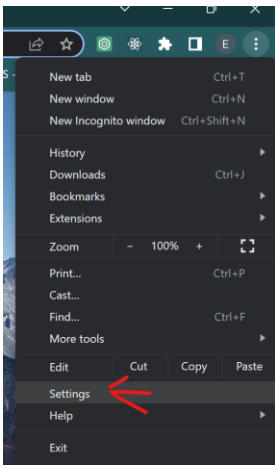
You will then see this problem when you type the https localhost link into the url.



Step 15:

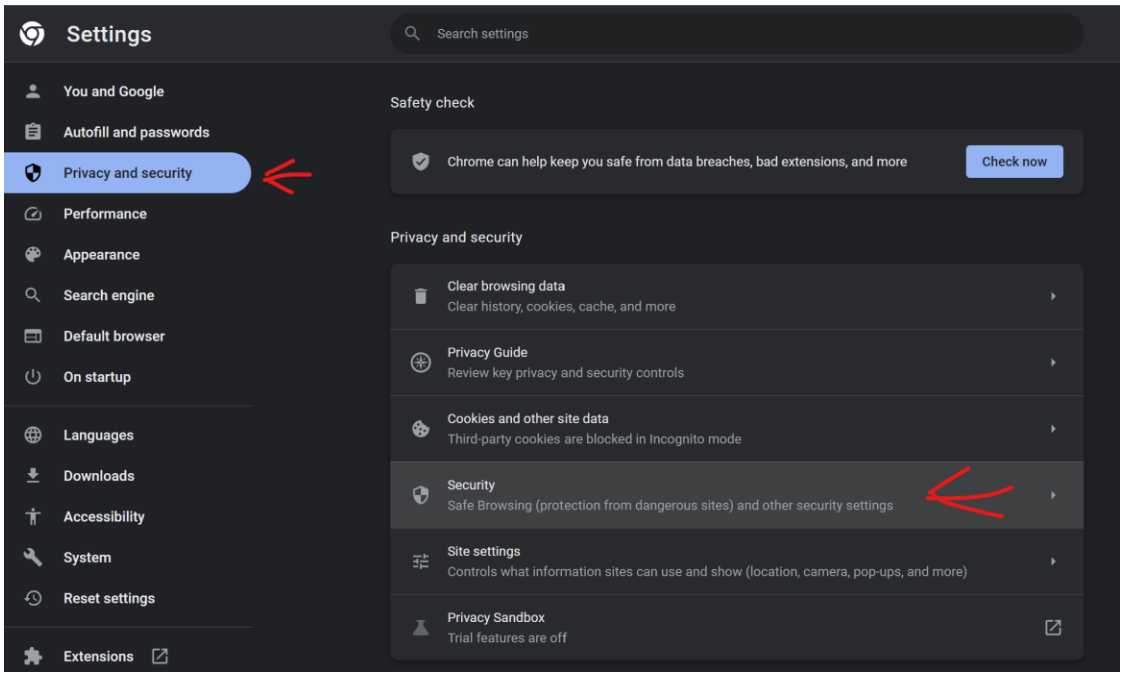
To solve this problem, we will import the rootCA.pem file into the trusted certs in google chrome.

First, go to settings after clicking on the 3 dot on the top right corner.



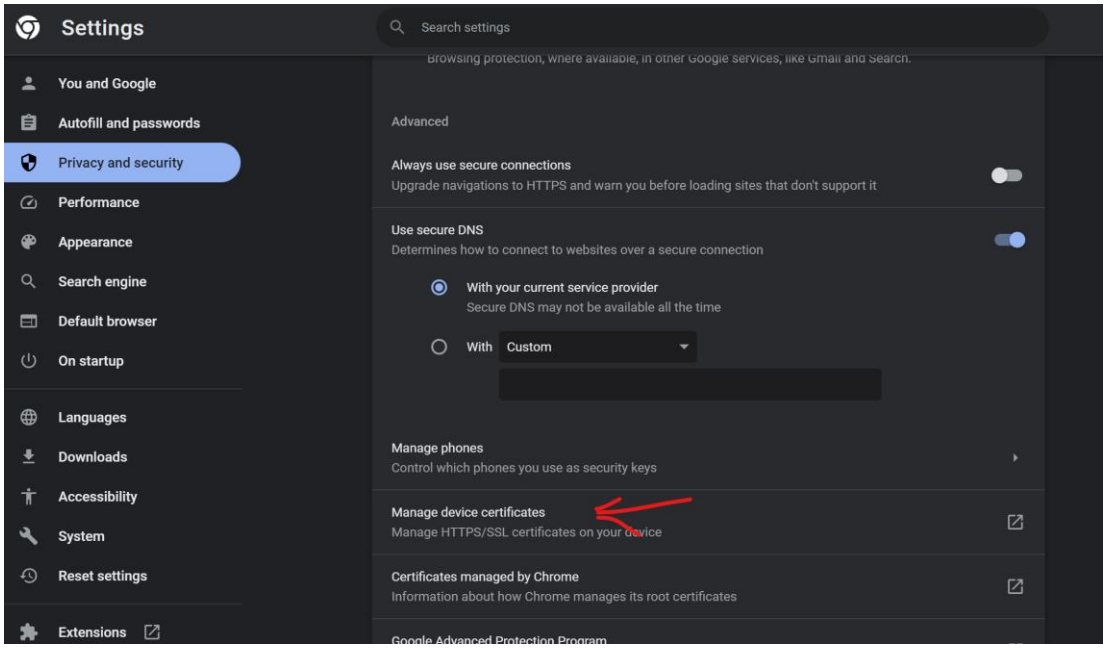
Step 16:

Go to Privacy and security and click on Security as shown below.



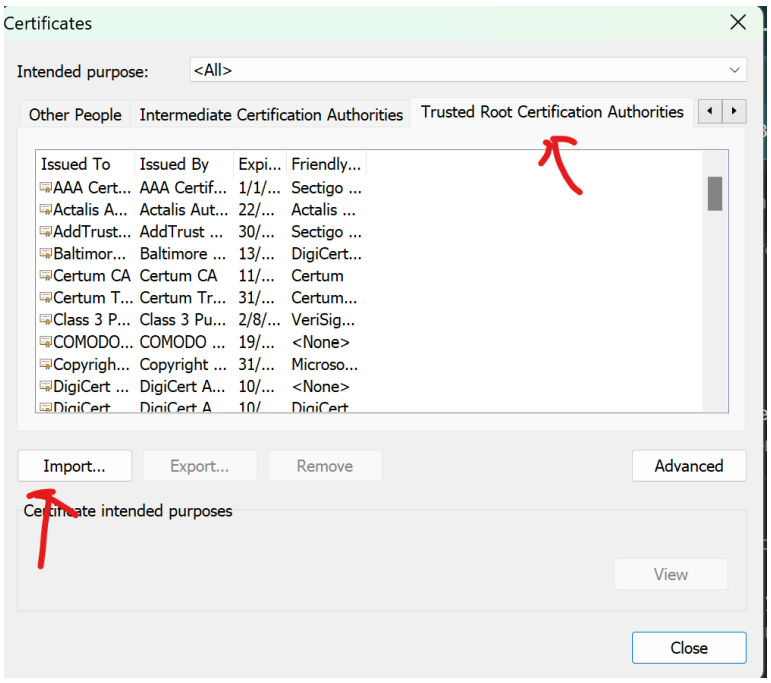
Step 17:

Scroll down and you will see a “Manage device certificates”. Click on it.



Step 18:

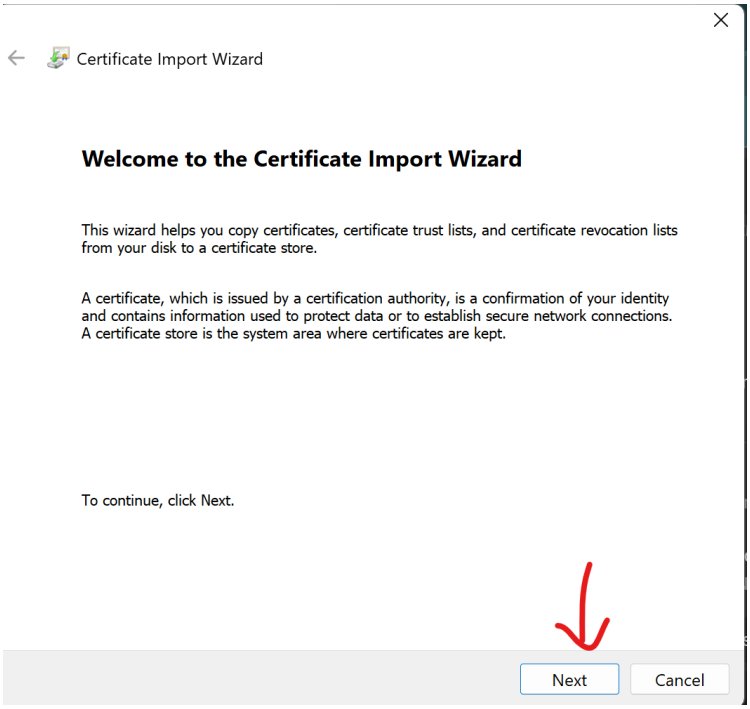
Next, you will see a pop-up go and click on Trusted Root Certification Authorities and import the rootCA.pem file that we got earlier.



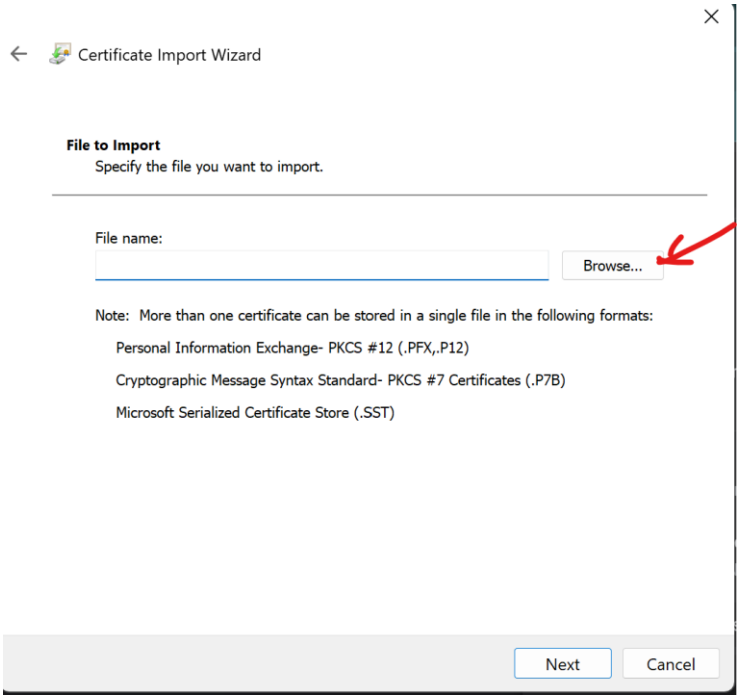
Step 19:

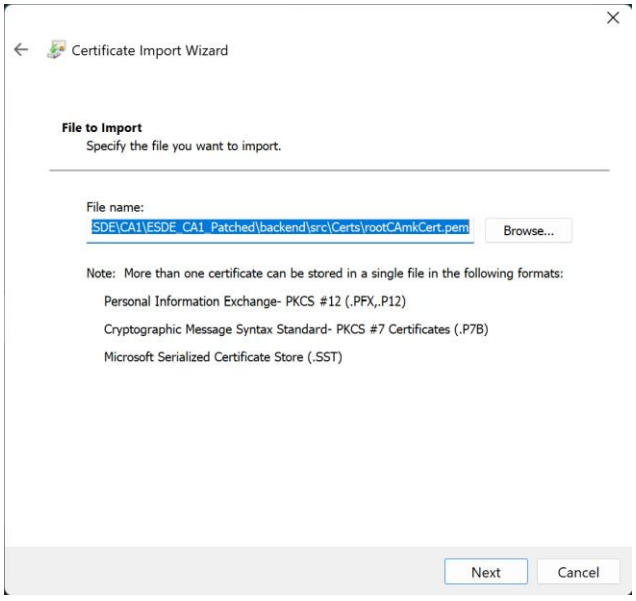
Follow the steps below to import the rootCA.pem file.

1. Click on next

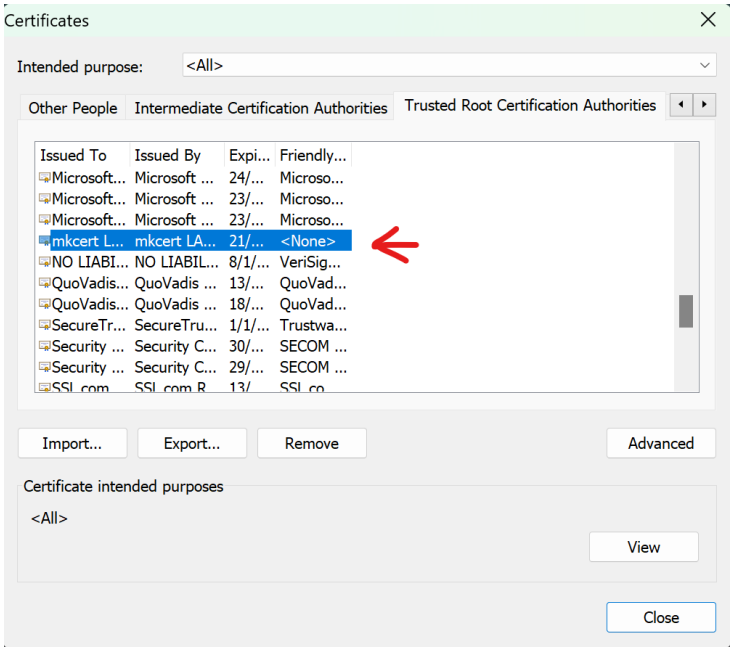


2. Click on Browse and look for the rootCA.pem file



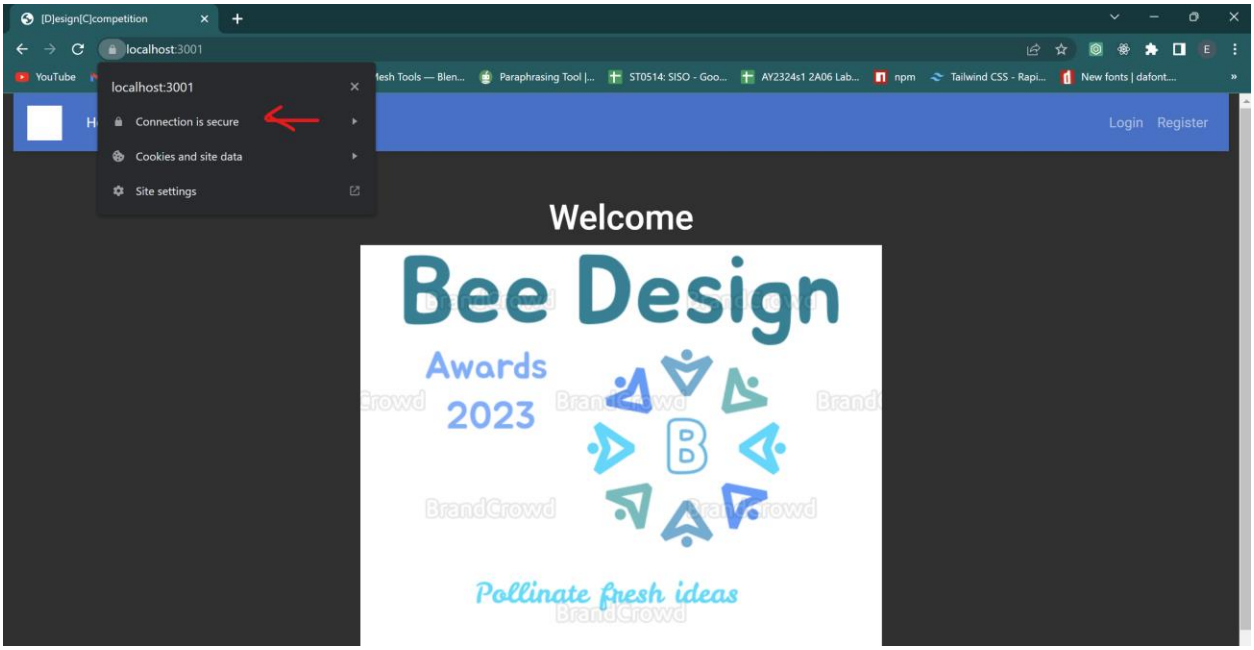


3. Click on Next, Yes and Finish and you have successfully imported the self-signed cert. You should see a mkcert under the Issued By column.



Step 20:

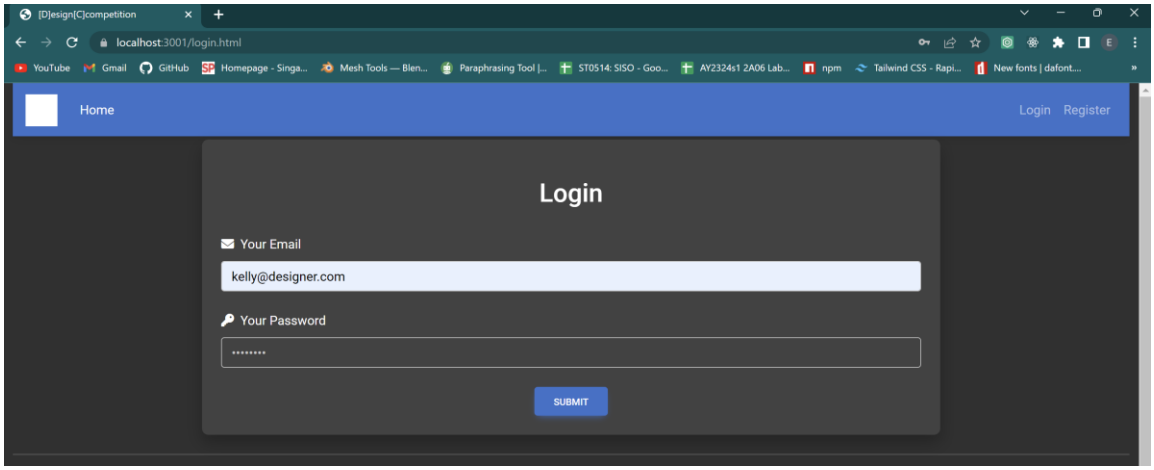
Now close the browser and reopen it and type the https localhost link again and the error should be removed. You should see a lock icon indicating that your site is now secure.



5.4 Evidence that the vulnerability has been rectified

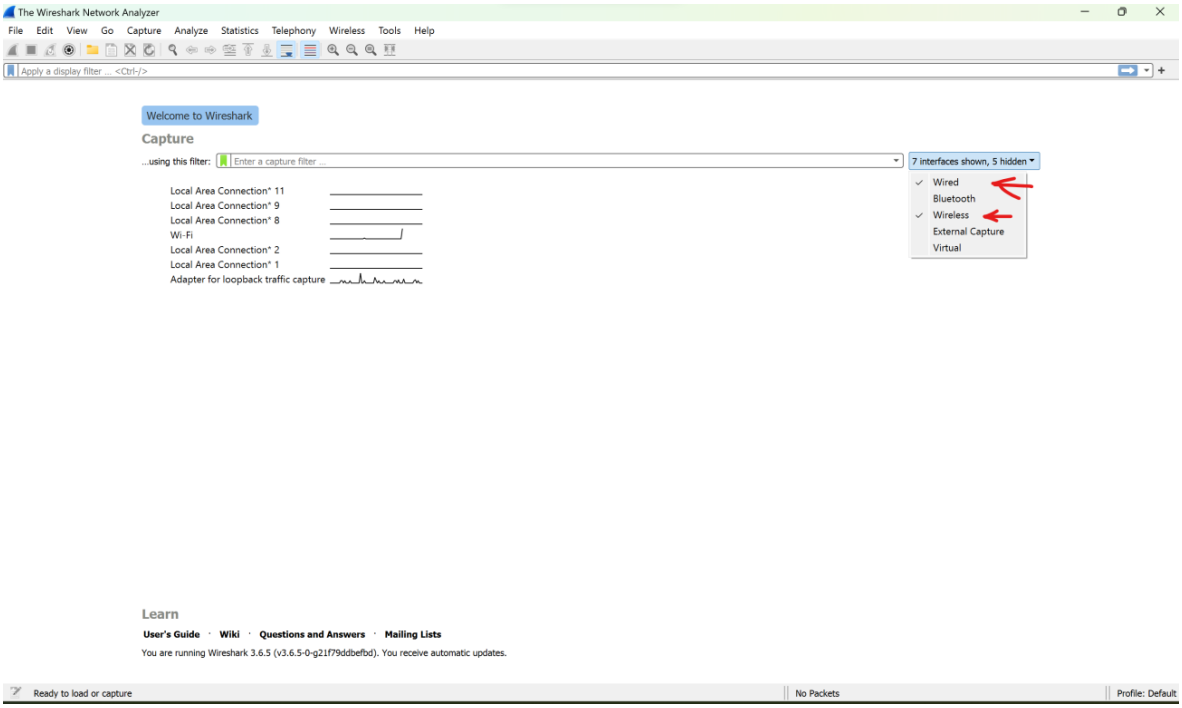
Step 1:

Now go into the login page and fill in Kelly's email and password **but do not press the submit button.**



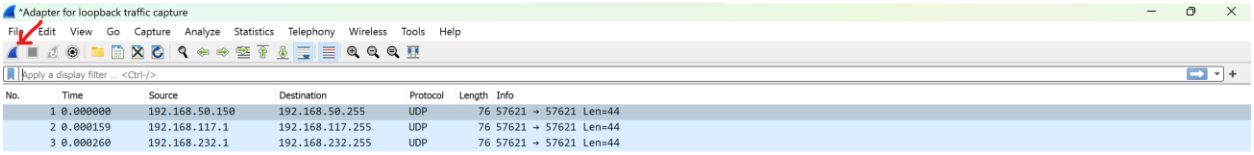
Step 2:

Head over to Wireshark and we will test if we are still capturing traffic showing plain text data and make sure that you have checked Wired and Wireless.



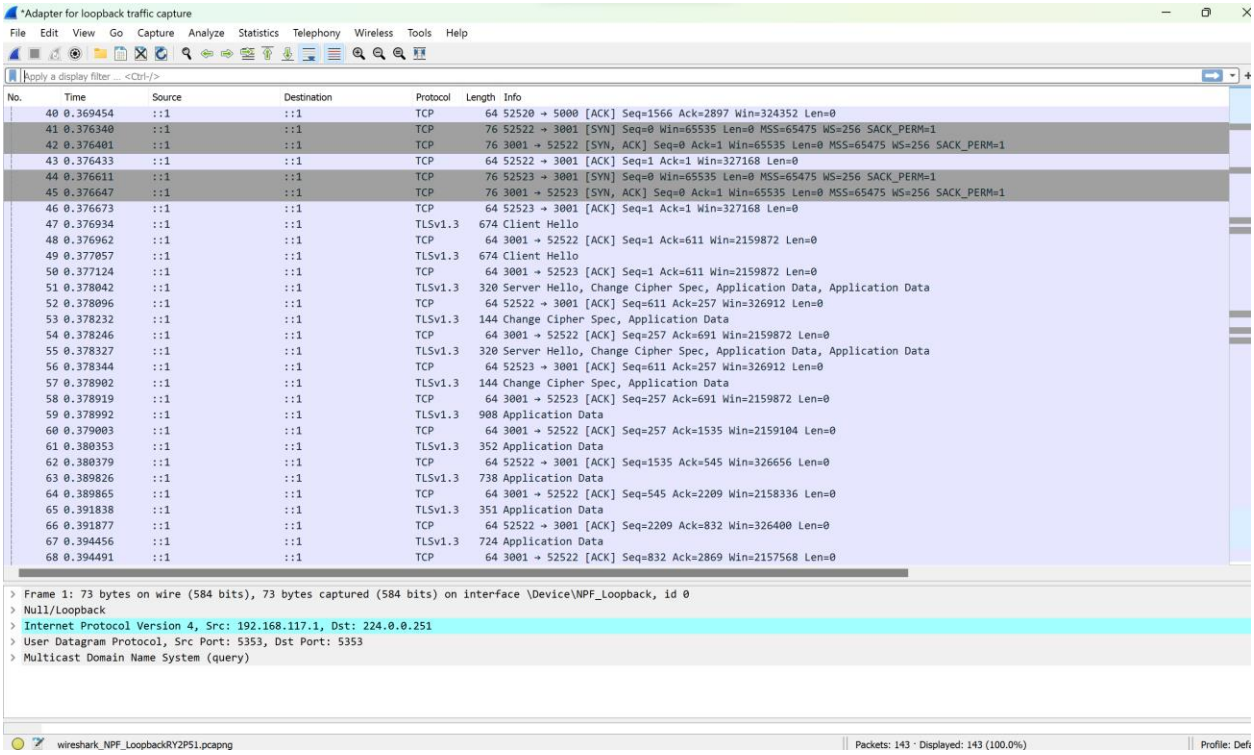
Step 3:

Next click on Adapter for loopback traffic capture and press the shark fin looking icon on the top left to begin capturing the traffic.



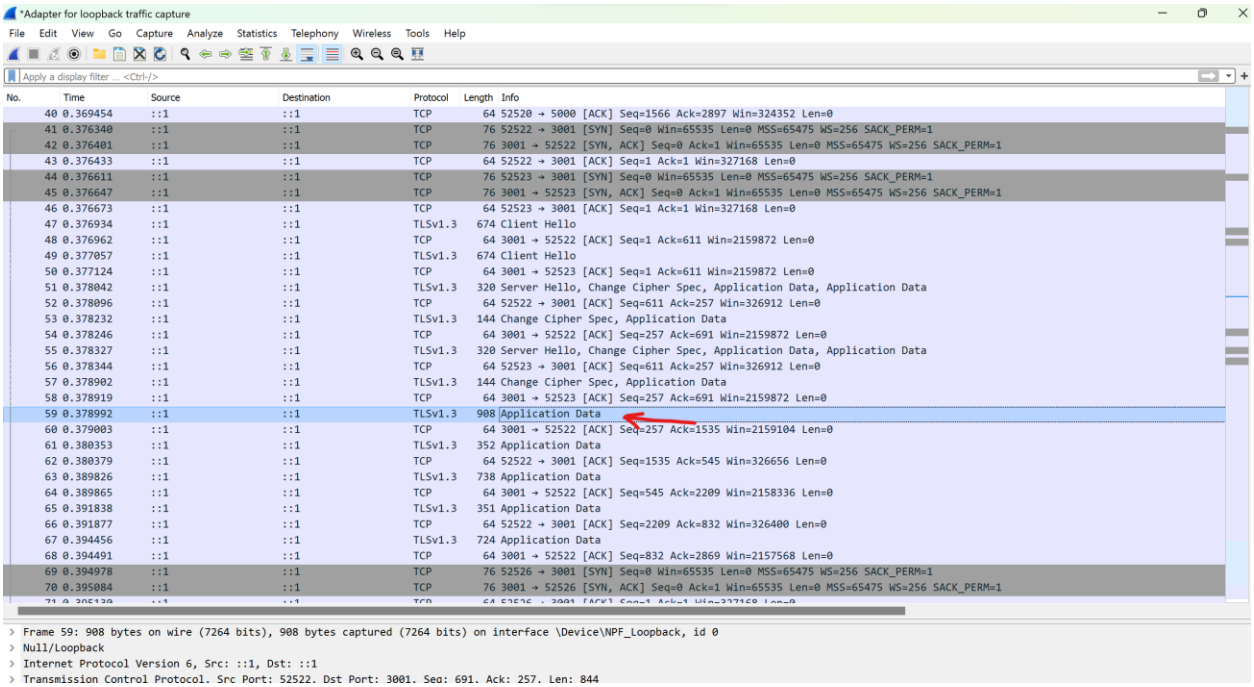
Step 4:

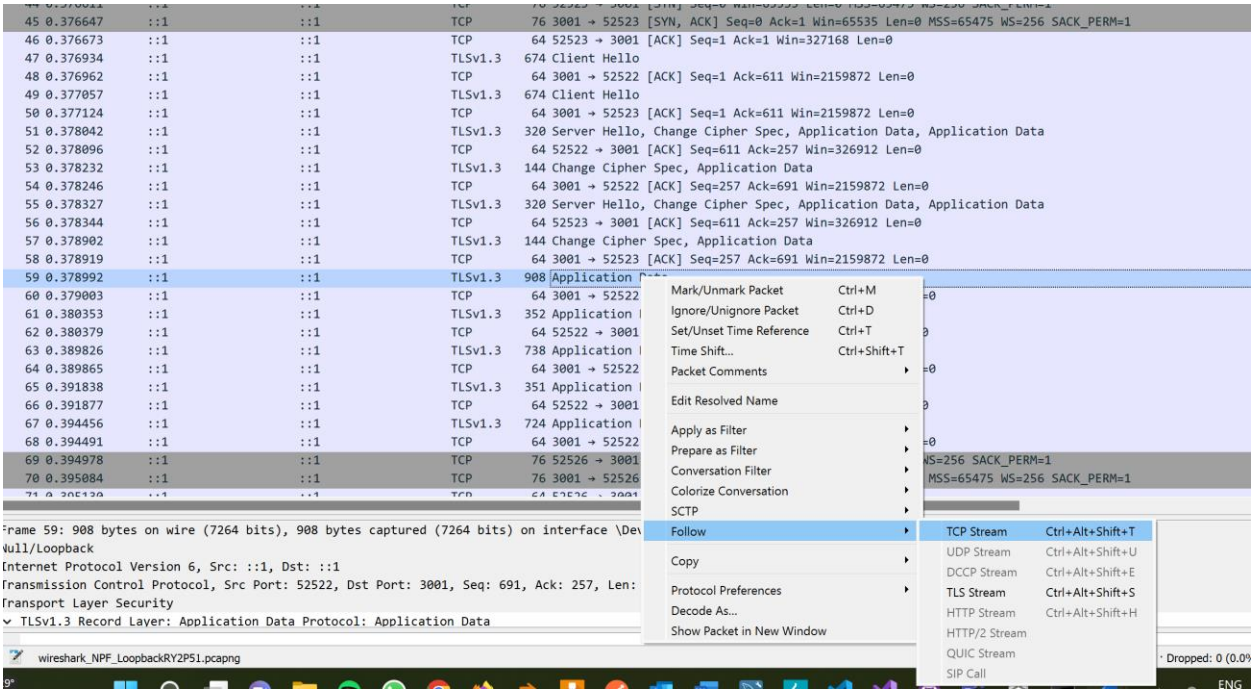
Click on the submit button to capture the traffic for our login request. And stop the Wireshark capture by clicking on the red square shaped button. You should not be able to see any request anymore compared to when our site is insecure.



Step 5:

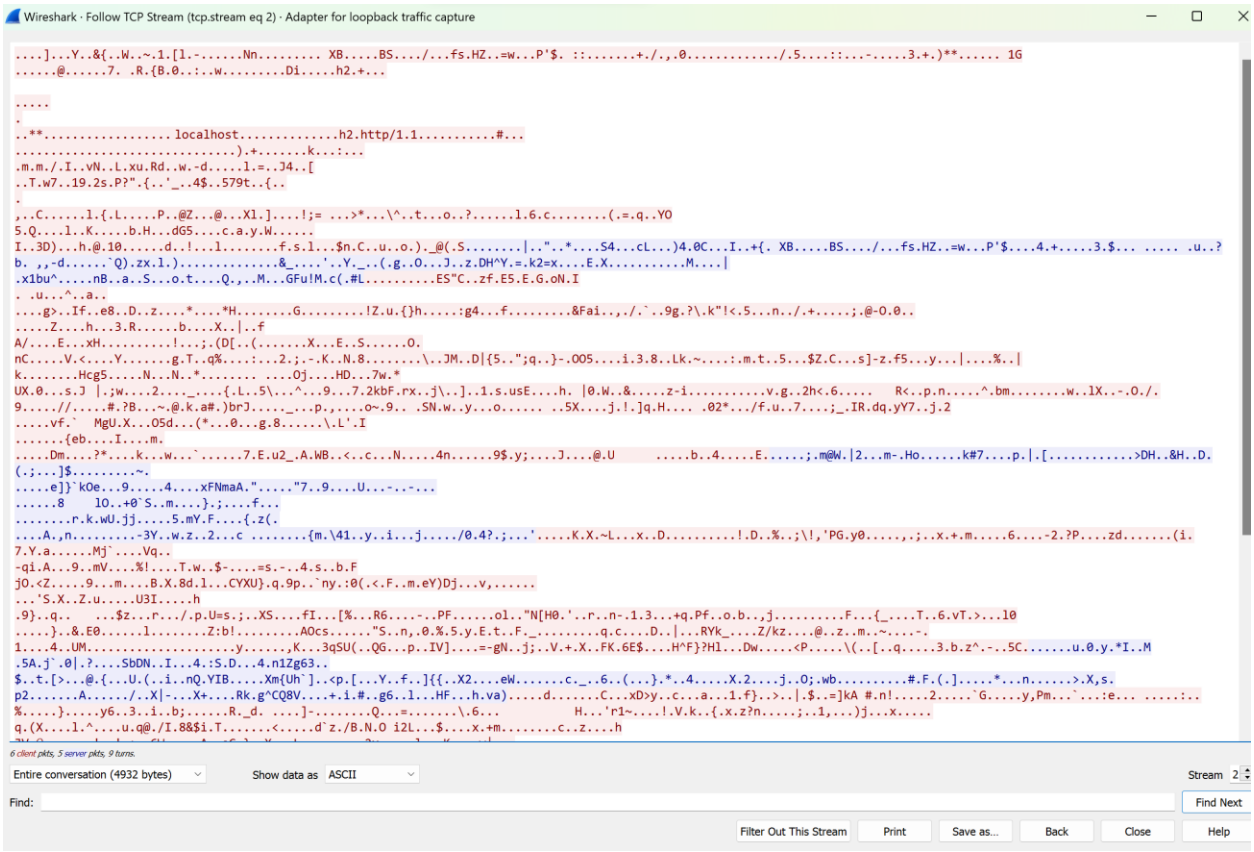
You will realize that our data is protected by a TLS protocol which helps us encrypt the data that is in transit as shown below. Click on an Application Data as shown below and right click to go into Follow > TCP Stream.





Step 6:

You should be able to see encrypted data as shown below. This shows that our cert has helped us to encrypt all the data in transit and we have successfully rectified this sensitive data exposure vulnerability.



6. Insufficient Logging & Monitoring

6.1 Definition

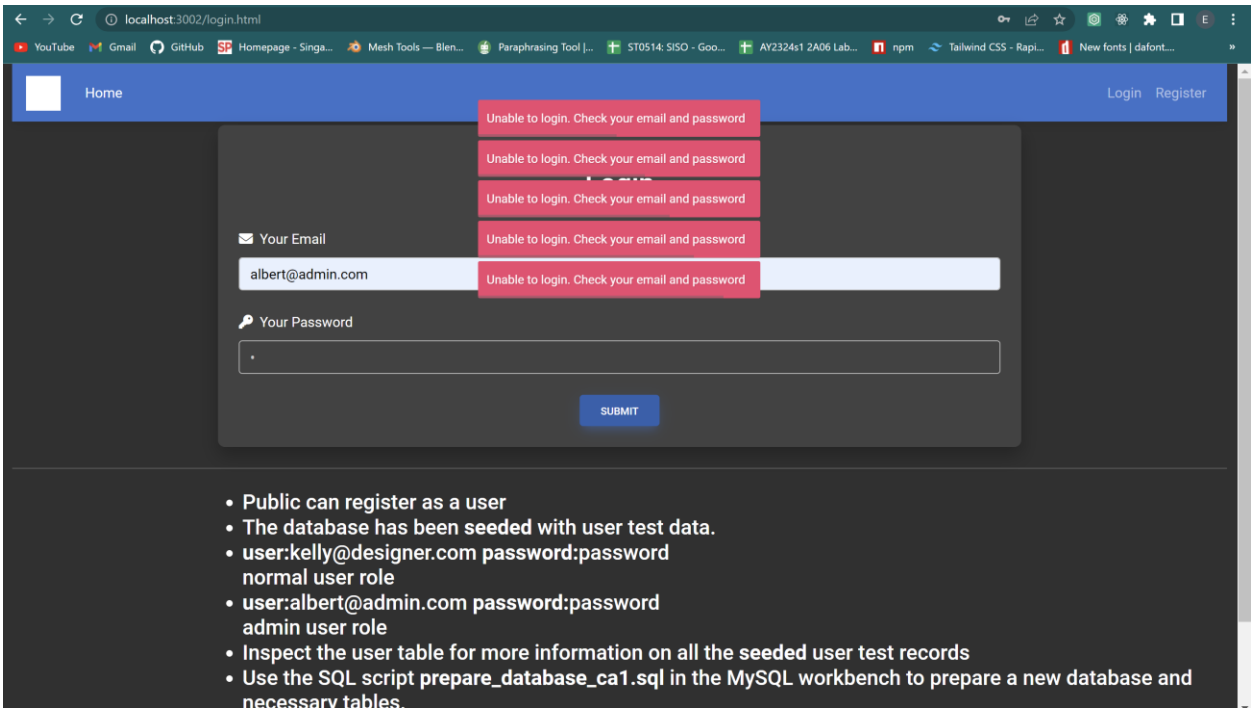
Insufficient Logging & Monitoring is a security weakness where an application or system lacks adequate mechanisms to generate, capture, retain, and analyze log events and security-related information. It occurs when an organization fails to implement robust logging and monitoring practices, which can hinder their ability to detect, investigate, and respond to security incidents effectively.

This can manifest in several ways like Lack of Event Logging, Inadequate Log Retention, Insensitive Log Data, and Lack of Real-Time Monitoring.

6.2 How to exploit vulnerability

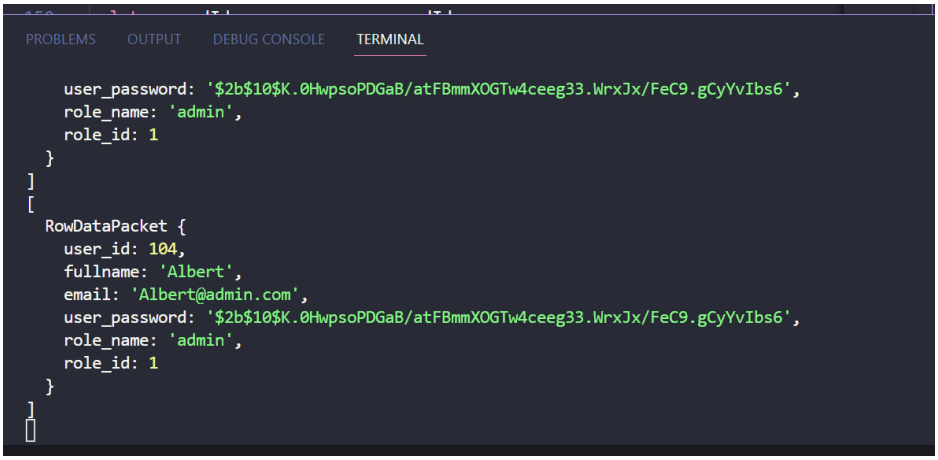
Step 1:

Go into the vulnerable website and try to login in with albert's email but for the password let's try to input incorrect ones to imitate an actual brute forcing scenario.



Step 2:

You will see that the only place that records the login process is only through the terminal as shown below. It is also not stating whether the login is successful or not and it also isn't being logged anywhere.



6.3 How to rectify vulnerability

Step 1:

To rectify the logging of the error etc. We will first import Winston and express-useragent as we will need to use this libraries to log the requests and errors.

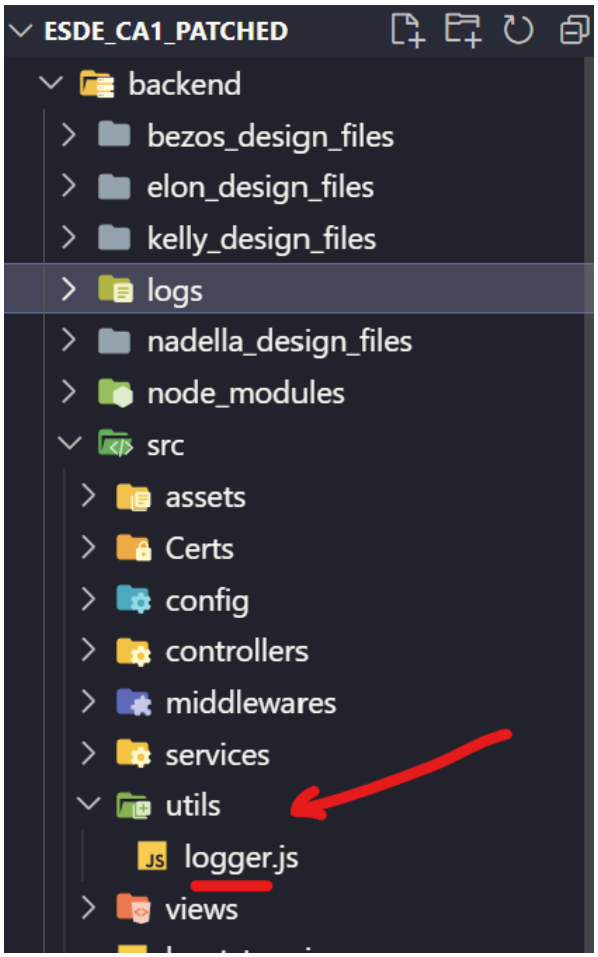
Copy the command and paste it into your backend terminal directory.

npm install winston express-useragent

Step 2:

Let’s create a utils folder and create a logger.js file to set up our winston logger.

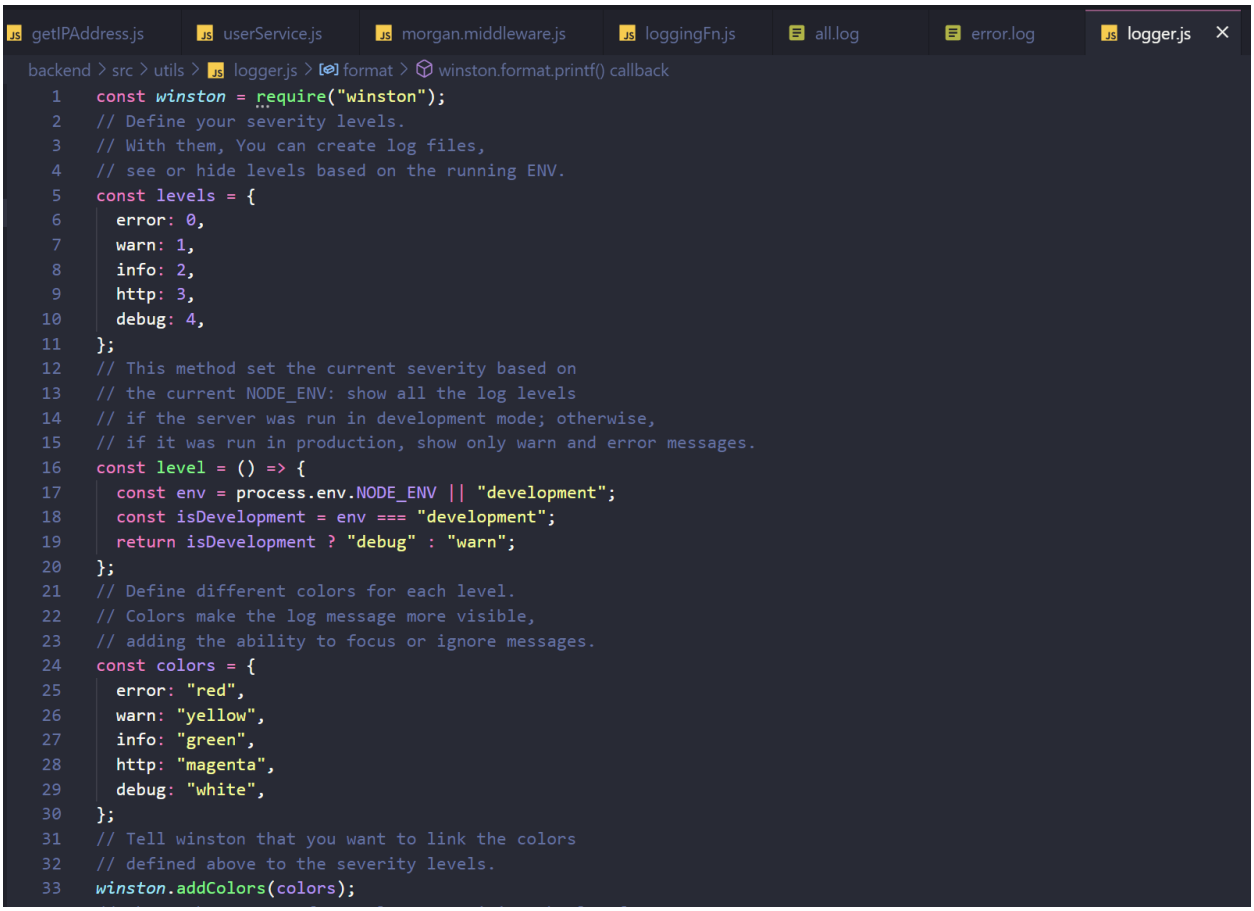
Under /backend/src/



Step 3:

Add the following codes to set up the winston logger.

Part 1 of logger.js:



Part 2 of logger.js:

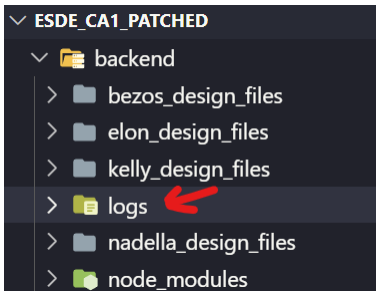
getIPAddress.js | userService.js | morgan.middleware.js | loggingFn.js | all.log | error.log | logger.js

backend > src > utils > logger.js > format > winston.format.printf() callback

```
// defined above to the severity levels.
winston.addColors(colors);
// Chose the aspect of your log customizing the log format.
const format = winston.format.combine(
  // Add the message timestamp with the preferred format
  winston.format.timestamp({ format: "DD-MM-YYYY HH:mm:ss:ms" }),
  // Tell Winston that the logs must be colored
  winston.format.colorize({ all: true }),
  // Define the format of the message showing the timestamp, the level and the message
  winston.format.printf(
    (info) => `${info.timestamp} ${info.level}: ${info.message}`
  )
);
// Define which transports the logger must use to print out messages.
// In this example, we are using two different transports
const transports = [
  // Allow the use the console to print the messages
  // new winston.transports.Console(),
  // Allow to print all the error level messages inside the error.log file
  new winston.transports.File({
    filename: "logs/error.log",
    level: "error",
  }),
  // Allow to print all the error message inside the all.log file
  // (also the error log that are also printed inside the error.log(
  new winston.transports.File({ filename: "logs/all.log" }),
];
// Create the logger instance that has to be exported
// and used to log messages.
const logger = winston.createLogger({
  level: level(),
  levels,
  format,
  transports,
});
module.exports = logger;
```

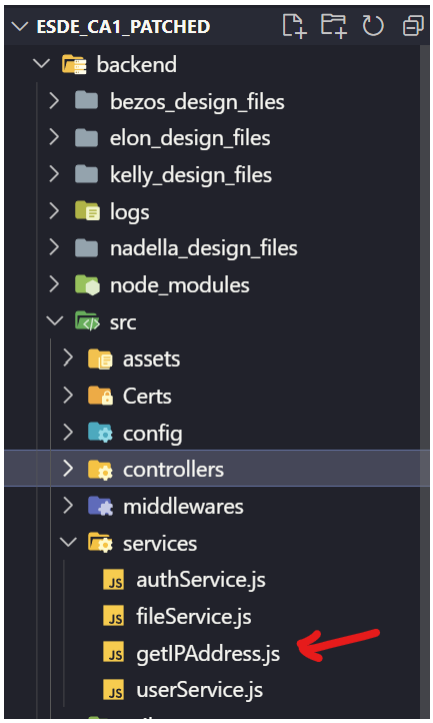
Step 4:

Create a logs folder under /backend/ we will use this to store our logs.



Step 5:

Next, create a file called getIPAddress.js the services folder under /backend/src/services.



Step 6:

In the file add the following code so that we will be able to get the IP Address of the user.

Step 7:

Next, head over to validateFn.js and import the following or anywhere where you would want to log any data.

Step 8:

We will create a validateEmail and add a log to it to capture the error in the login page if any hacker tries to brute force.

```
52  validateEmail: function (req, res, next) {
53    var email = req.body.email;
54    if (validator.isEmail(email)) {
55      next();
56    } else {
57      let message = "Unable to Login, Please enter a valid Email";
58      getIPAddress()
59        .then((ip) => {
60          logger.error({
61            userAgent: req.useragent.source,
62            method: req.method,
63            route: req.originalUrl,
64            email: req.body.email,
65            ip: ip,
66            error: message,
67            status: 400,
68          });
69        })
70        .catch((error) => {
71          console.log(error);
72        });
73      res.status(400).send({ message: message });
74    }
75  },
```

Step 9:

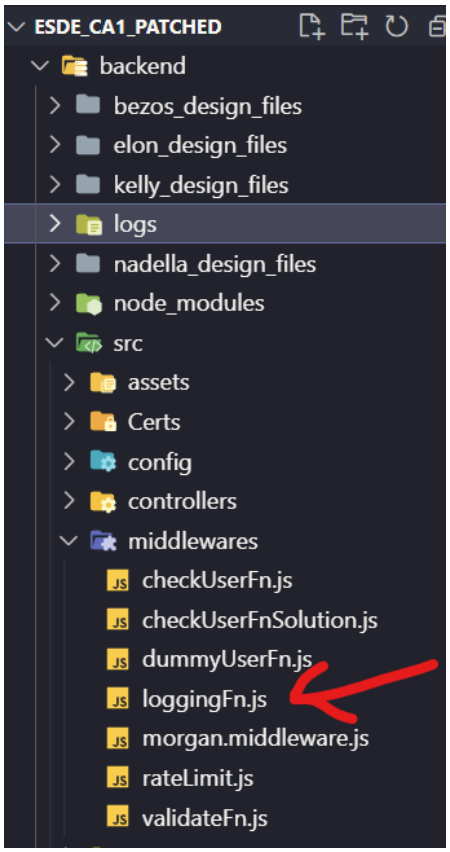
We will also add 1 logger to the bycrypt password at processLogin in the authController.js under /backend/src/controllers/authController.js.

We will add the logger to the else statement after checking if the brcrypt is not successful.

```
82          }
83          if (bcrypt.compareSync(password, results[0].user_password) == true) {
84            let data = {
85              user_id: results[0].user_id,
86              role_name: results[0].role_name,
87              token: jwt.sign(
88                { id: results[0].user_id, role: results[0].role_name },
89                config.JWTKey,
90                {
91                  expiresIn: 86400, //Expires in 24 hrs
92                }
93              ),
94            }; //End of data variable setup
95
96            return res.status(200).json(data);
97          } else {
98            getIPAddress()
99              .then((ip) => {
100               logger.error({
101                 userAgent: req.useragent.source,
102                 method: req.method,
103                 route: req.originalUrl,
104                 email: req.body.email,
105                 ip: ip,
106                 error: "Password does not match",
107                 status: 401,
108               });
109             })
110             .catch((error) => {
111               console.log(error);
112             });
113           return res.status(500).json({ message: error });
114         } //End of passowrd comparison with the retrieved decoded password.
```

Step 10:

We will now create 1 more file called logginfFn.js under /backend/src/middlewares where we will use it to log all requests in the routes.js.



Step 11:

Add the following code into the loggingFn.js file under /backend/src/middlewares.

Step 12:

Go to routes.js and we will import the logginFn.js inside and use it on all the endpoints with a red arrow pointing to it.

routes.js | checkUserFnSolution.js | validateFn.js | index.js frontend

backend > src > routes.js > ...

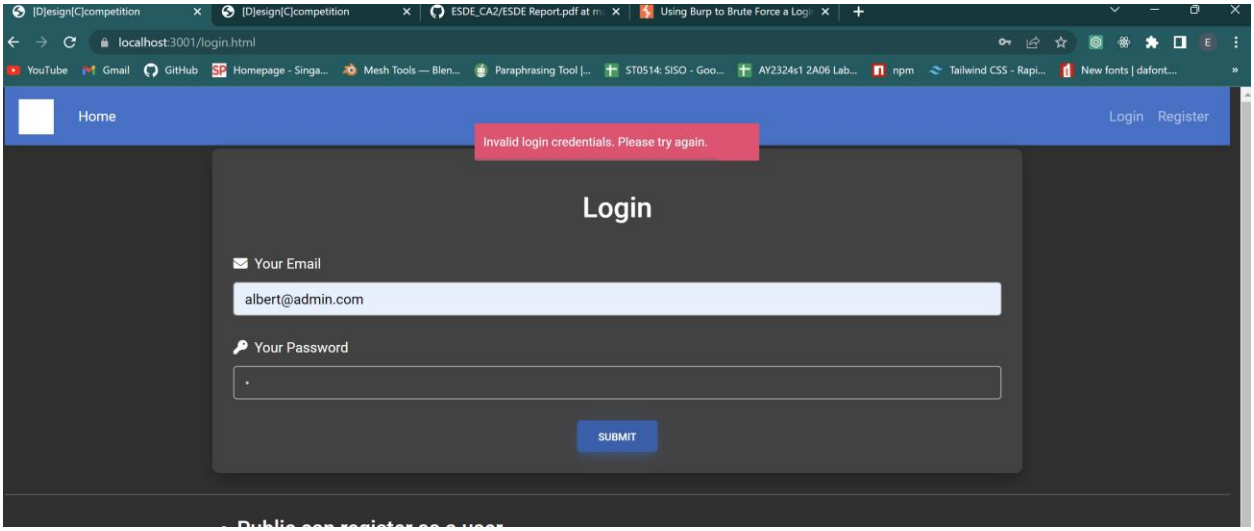
```
52    );
53    // USER UPDATE DESIGN (UPDATE PHASE)
54    router.put(
55      "/api/user/design/",
56      loggingFn.logData,
57      checkUserFnSolution.checkForValidUser,
58      validateFn.validateDesignTitleAndDesc,
59      userController.processUpdateOneDesign
60    );
61    // USER SEND INVITATION EMAIL
62    router.post(
63      "/api/user/processInvitation/",
64      loggingFn.logData,
65      checkUserFnSolution.checkForValidUser,
66      validateFn.validateEmailRecipient,
67      userController.processSendInvitation
68    );
69    // USER VIEW PROFILE (RETRIEVE PHASE)
70    router.get(
71      "/api/user/:recordId",
72      checkUserFnSolution.checkForValidUser,
73      userController.processGetOneUserData
74    );
75
76    // ADMIN SEARCH USER
77    router.get(
78      "/api/user/process-search-user/:pagenumber/:search?",
79      loggingFn.logData,
80      checkUserFnSolution.checkValidAdminRole,
81      userController.processGetUserData
82    );
83    // ADMIN SEARCH SUBMISSION BY EMAIL
84    router.get(
85      "/api/user/process-search-user-design/:pagenumber/:search?",
86      loggingFn.logData,
87      checkUserFnSolution.checkValidAdminRole,
88      validateFn.validateSearchUserSubByEmail,
89      userController.processGetSubmissionsbyEmail
90    );
91    // ADMIN CHANGE USER ROLE (UPDATE PHASE)
92    router.put(
93      "/api/user/",
94      loggingFn.logData,
95      checkUserFnSolution.checkValidAdminRole,
96      userController.processUpdateOneUser
97    );
```

6.4 Evidence that the vulnerability has been rectified

Step 1:

Now, let's try if our logs are all working. We will go into the patched website now and go to the login page and fill in albert's email and we will input an incorrect password and submit it. For my case, I should see a message saying "Invalid login credentials. Please try again".

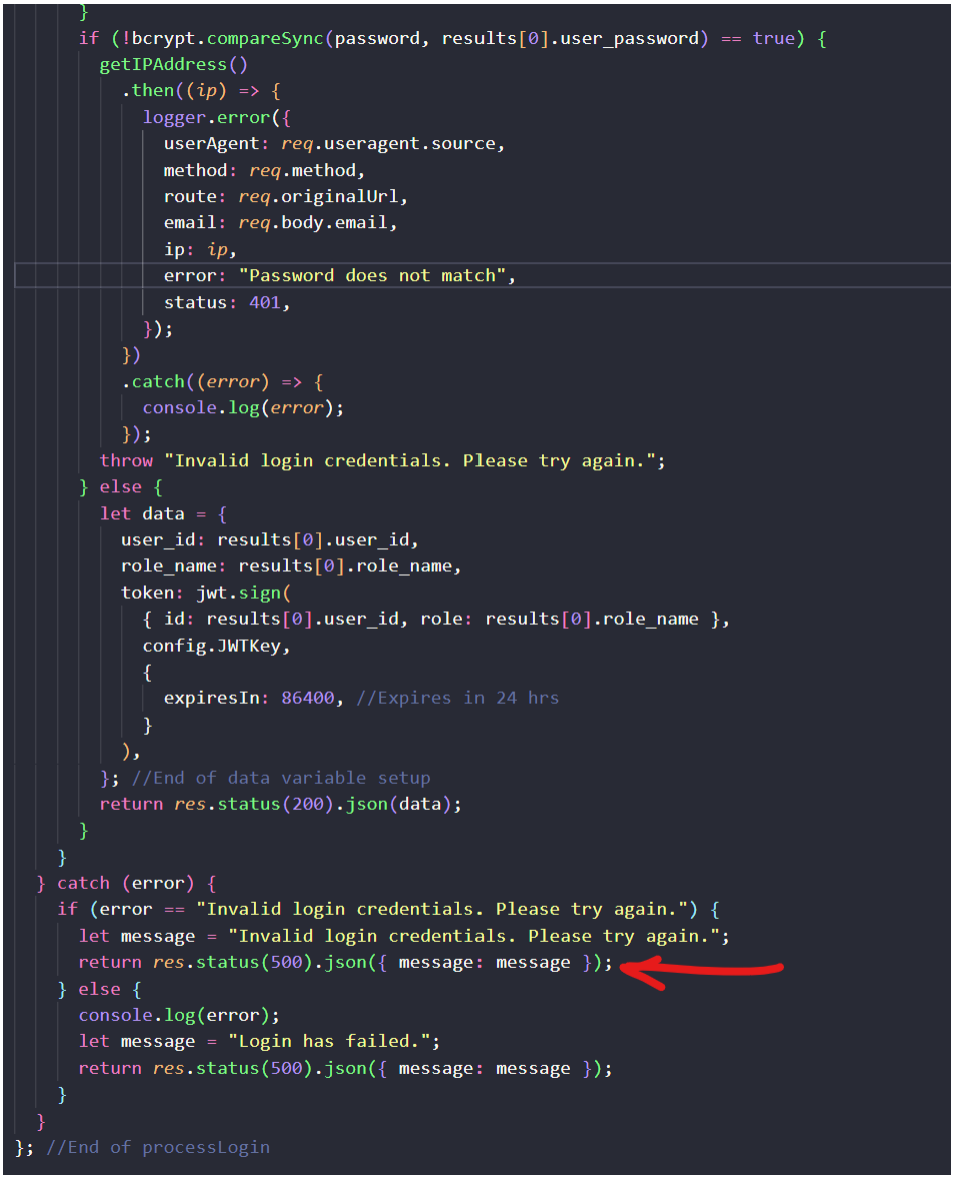
*Note: This depends on what message you return when the password does not match.



Continued Below…

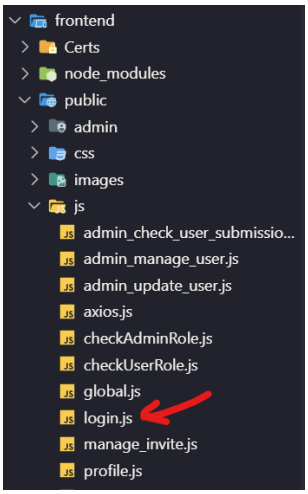
For my case, I added the error message in my authController.js processLogin under the catch error and return it.

This is found under /backend/src/authController.



```
      }
      if (!bcrypt.compareSync(password, results[0].user_password) == true) {
        getIPAddress()
          .then((ip) => {
            logger.error({
              userAgent: req.useragent.source,
              method: req.method,
              route: req.originalUrl,
              email: req.body.email,
              ip: ip,
              error: "Password does not match",
              status: 401,
            });
          })
          .catch((error) => {
            console.log(error);
          });
        throw "Invalid login credentials. Please try again.";
      } else {
        let data = {
          user_id: results[0].user_id,
          role_name: results[0].role_name,
          token: jwt.sign(
            { id: results[0].user_id, role: results[0].role_name },
            config.JWTKey,
            {
              expiresIn: 86400, //Expires in 24 hrs
            }
          ),
        }; //End of data variable setup
        return res.status(200).json(data);
      }
    }
  } catch (error) {
    if (error == "Invalid login credentials. Please try again.") {
      let message = "Invalid login credentials. Please try again.";
      return res.status(500).json({ message: message });
    } else {
      console.log(error);
      let message = "Login has failed.";
      return res.status(500).json({ message: message });
    }
  }
}; //End of processLogin
```

Head over to the frontend code and look for the file called login.js under /frontend/public/js/login.js.

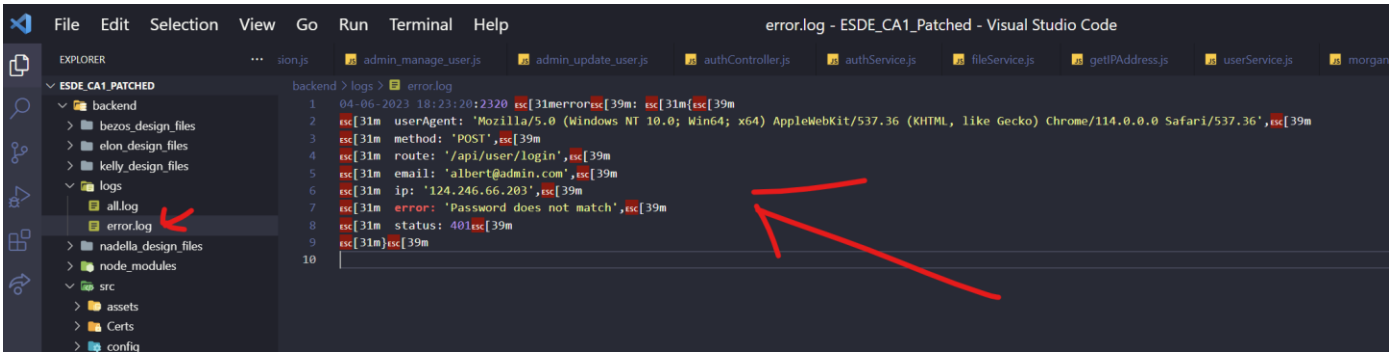


Change the static response "Unable to login. Check your email and password" to the following response message. This is so that we would be able to see our own custom error message that we return.

Step 2:

Head over to the logs folder under /backend and you should see 2 .log files, go to error.log and you should see the following error log as shown below.

You will be able to see the userAgent, method, route, email, IP address, error message and status. You will also be able to see the date and time where the following error occurred. This shows that we have successfully logged the error in the login page that would be sufficient for us.



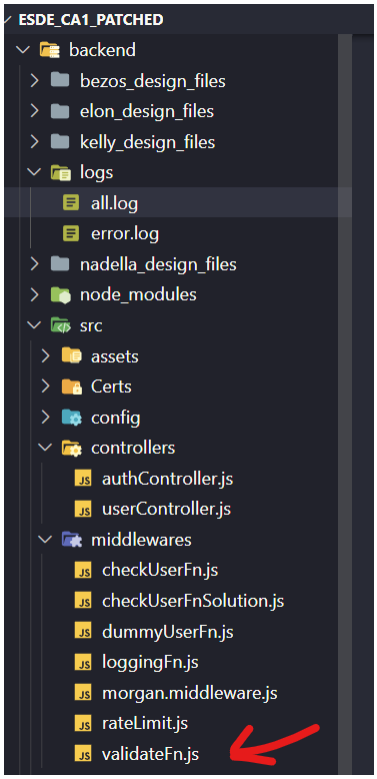
7. Bonus

7.0.1 Validating Invalid File Types

In the design submission page, there is a file submission available for users to submit their designs, however there are times where hackers may submit malicious files. So, let's add a validation middleware to the design submission endpoint to validate the file type and size before it is submitted to cloudinary.

Step 1:

Go into the validateFn.js file located under /backend/src/middlewares/validateFn.js



Step 2:

Create a new function called validateFileTypeAndSize and add the following code as shown below.

```
validateFileTypeAndSize: function (req, res, next) {
  var submittedFile = req.body.file;
  console.log("Validation for file type: " + submittedFile);

  var fileName = path.basename(submittedFile.path);
  console.log("Submitted file path: " + submittedFile.path);
  console.log("Path base name :" + path.basename(submittedFile.path));

  var fileExtension = fileName.split(".");
  fileExtension = fileExtension[fileExtension.length - 1];

  var validFileSize = false;

  //Get file size in bytes
  var fileStat = fs.statSync(submittedFile.path);
  var fileSizeInBytes = fileStat.size;
  console.log(`File size in bytes: ${fileSizeInBytes / 1000000}MB`);

  //Check file size
  if (fileSizeInBytes <= 1000000) {
    validFileSize = true;
  } else {
    console.log("File size exceeded 1MB");
  }

  var validFileType = false;

  switch (fileExtension.toLowerCase()) {
    case "jpg":
      validFileType = true;
      break;
    case "jpeg":
      validFileType = true;
      break;
    case "png":
      validFileType = true;
      break;
    case "gif":
      validFileType = true;
      break;
  }

  if (validFileSize == false || validFileType == false) {
    console.log("Invalid file type/size received");
    fs.unlink(submittedFile.path, function (err) {
      if (err) return console.log(err);
      console.log("Deleted file successfully");
    });
```

Continued Below…

```
    if (validFileSize == false || validFileType == false) {
      console.log("Invalid file type/size received");
      fs.unlink(submittedFile.path, function (err) {
        if (err) return console.log(err);
        console.log("Deleted file successfully");
      });

      if (!validFileSize) {
        let message = "Invalid file size. File size must be less than 1MB.";
        getIPAddress()
          .then((ip) => {
            logger.error({
              userAgent: req.useragent.source,
              method: req.method,
              route: req.originalUrl,
              ip: ip,
              userId: req.body.userId,
              error: message,
              status: 400,
            });
          })
          .catch((error) => {
            console.log(error);
          });

        return res.status(400).send({ message: message });
      } else if (!validFileType) {
        let message = "Invalid file type. Only jpg/jpeg/png/gif are allowed.";
        getIPAddress()
          .then((ip) => {
            logger.error({
              userAgent: req.useragent.source,
              method: req.method,
              route: req.originalUrl,
              ip: ip,
              userId: req.body.userId,
              error: message,
              status: 400,
            });
          })
          .catch((error) => {
            console.log(error);
          });

        return res.status(400).send({ message: message });
      }
    } else {
      console.log("Validation Passed");
      next();
    }
  },
```

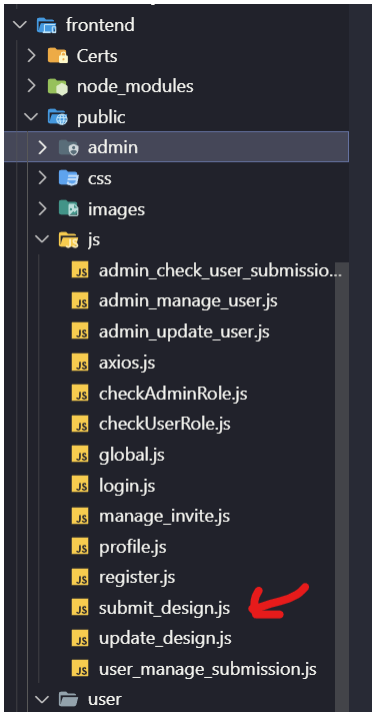
Step 3:

Let’s add the function we created in routes.js under /backend/src/routes.js

```
// USER SUBMIT DESIGN
router.post(
  "/api/user/process-submission",
  loggingFn.logData,
  checkUserFnSolution.checkForValidUser,
  validateFn.validateDesignTitleAndDesc,
  validateFn.validateFileTypeAndSize,
  rateLimitFn.designSubmissionCountLimit,
  userController.processDesignSubmission
);
```

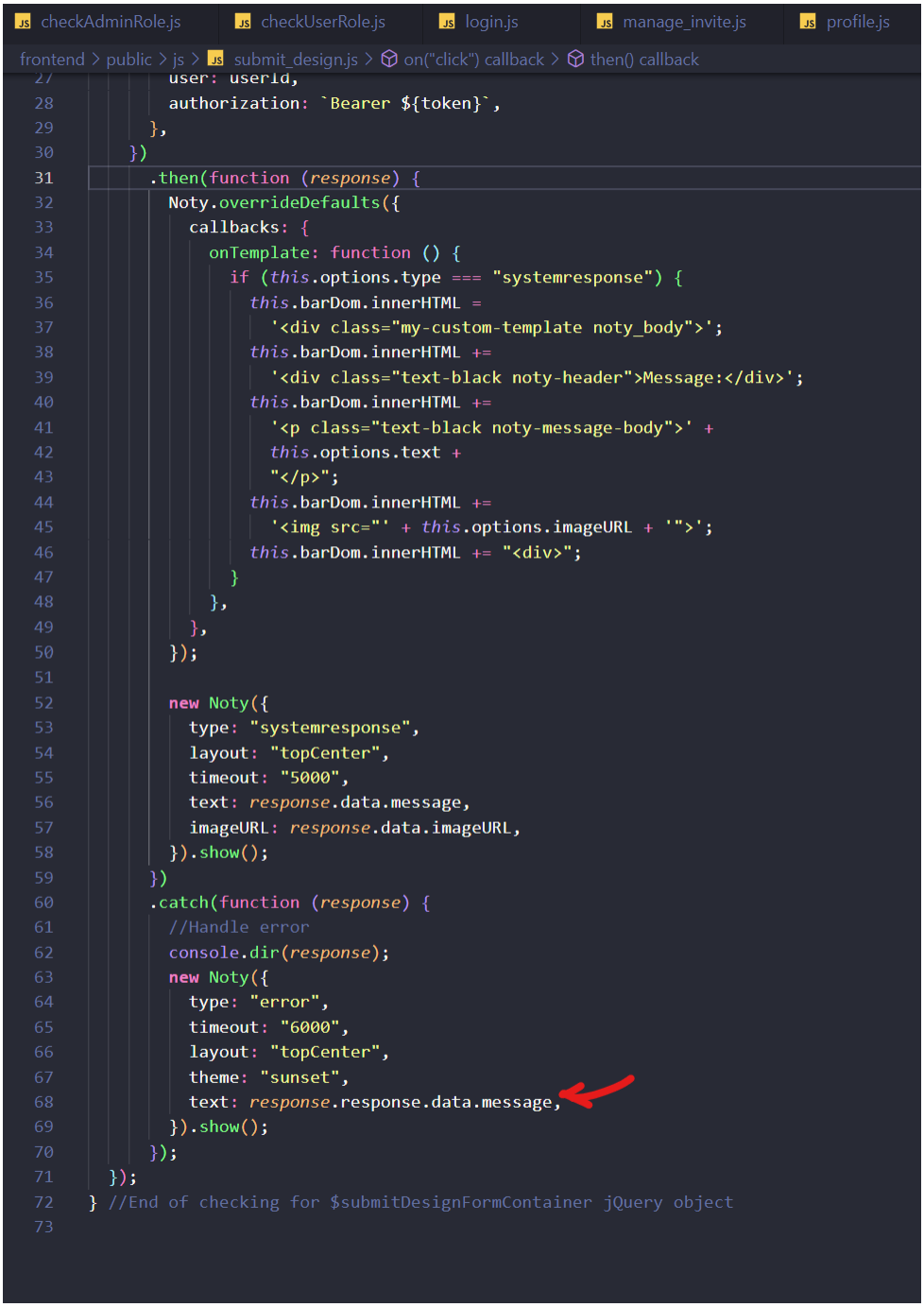
Step 4:

Let’s go to the submit_design.js file under /frontend/public/js/submit_design.js



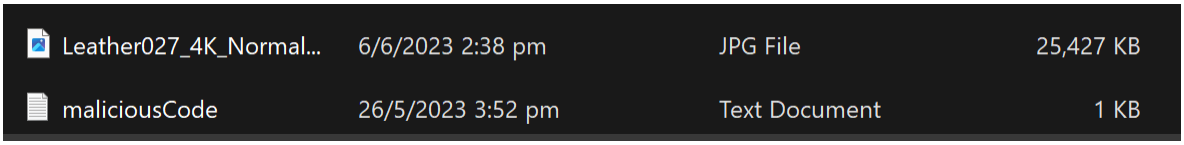
Step 5:

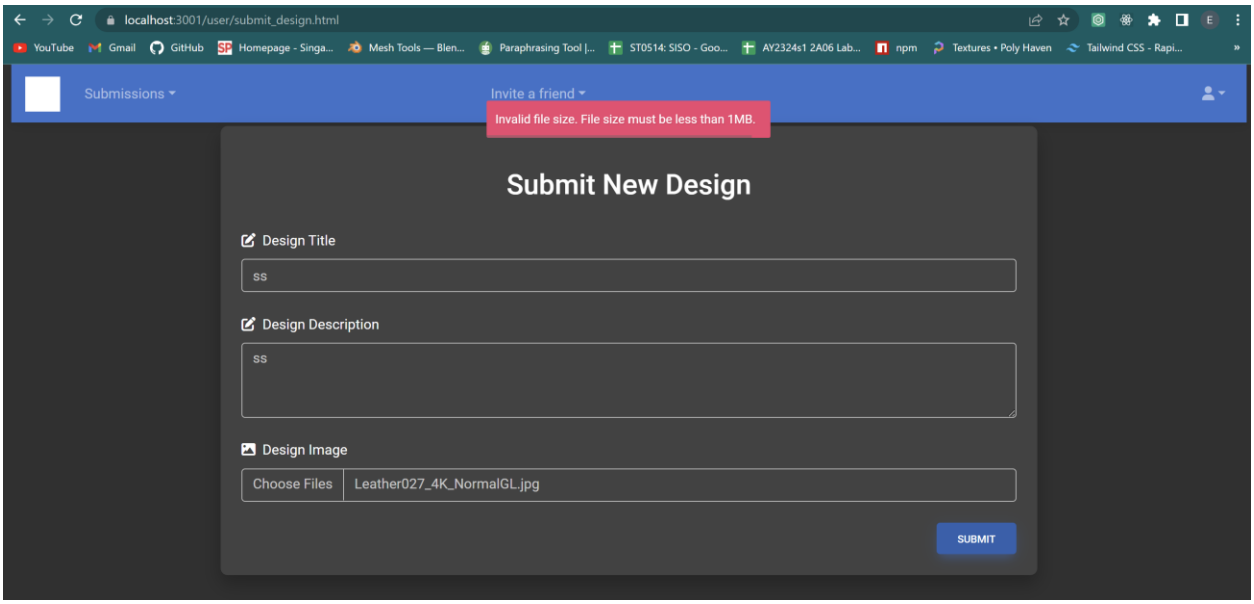
Change the following static error message to the following so that we will be able to display the respective error messages when the user submits a file type or size that is not valid.

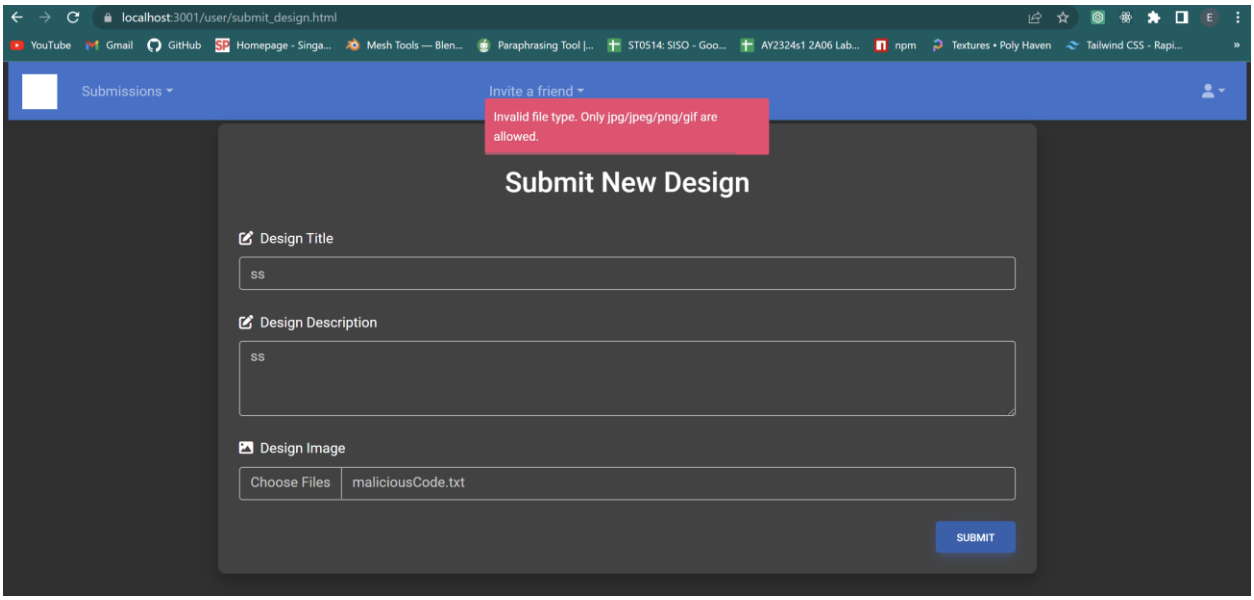


Step 6:

Let’s test if we can submit file types and file sizes before trying to submit to cloudinary.

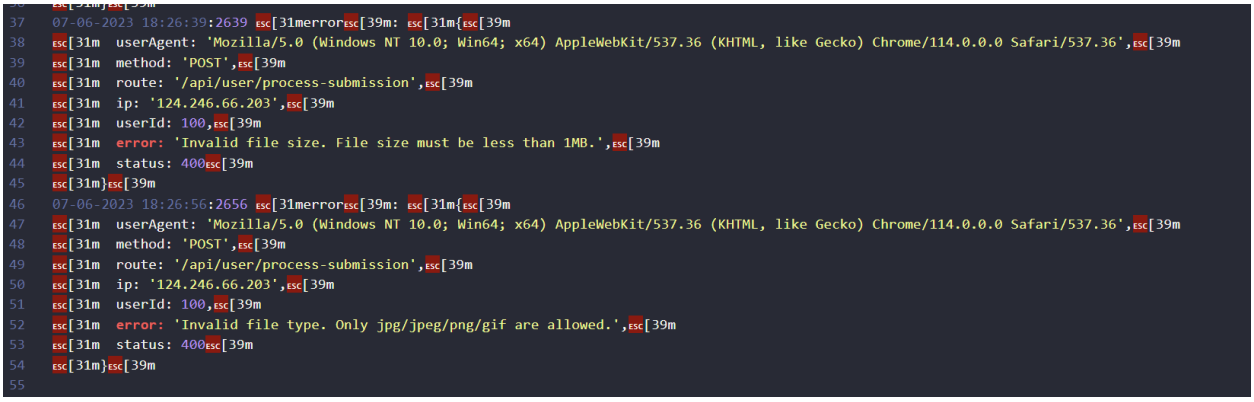






Step 7:

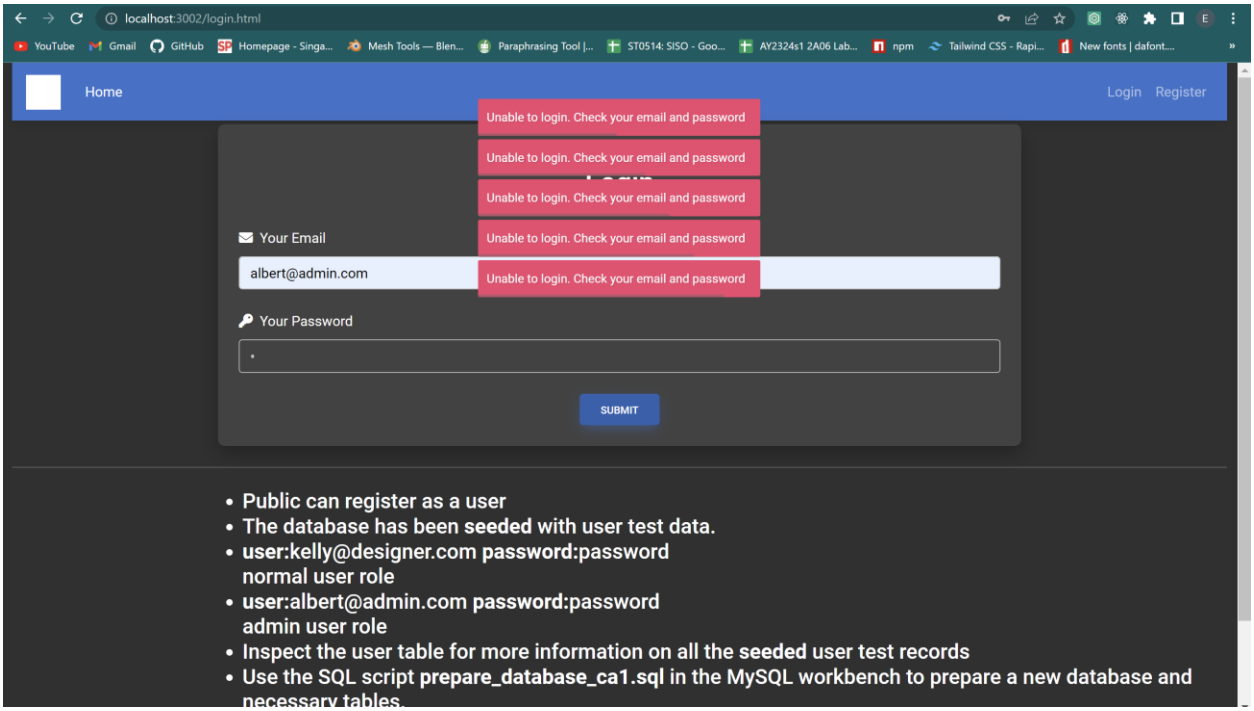
You will also see in your terminal that the errors have been successfully logged as well to have sufficient logging to track any suspicious activities. This shows that we have successfully validated the file type and size before sending it to cloudinary.



7.1.1 How to exploit vulnerability – Limiting brute forcing attacks

Step 1:

Go into the vulnerable website and try to login in with albert's email but for the password let's try to input incorrect ones to imitate an actual brute forcing scenario.

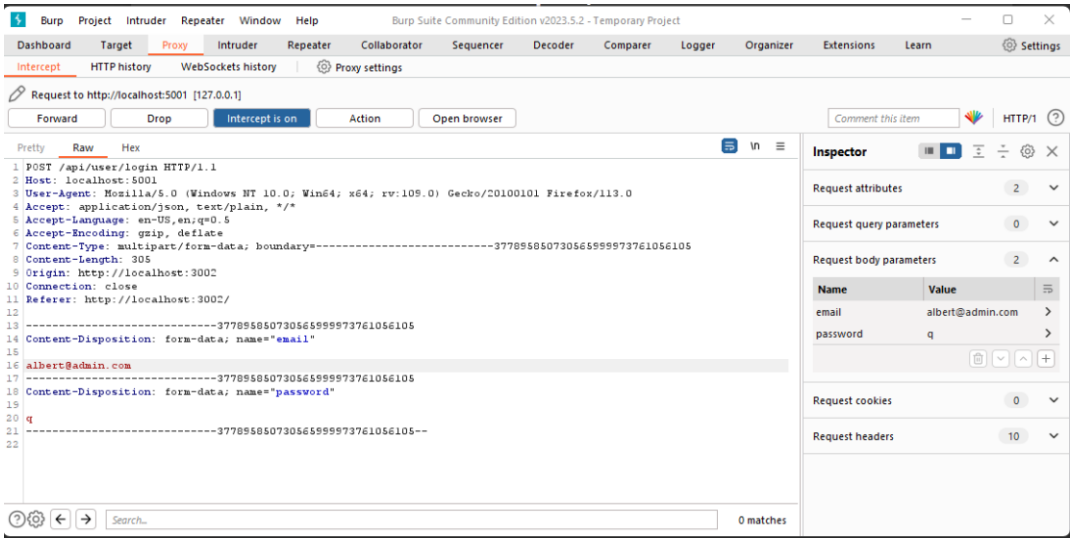


Step 2:

On the other hand, there will be a chance where the server will crash due to overwhelming requests.

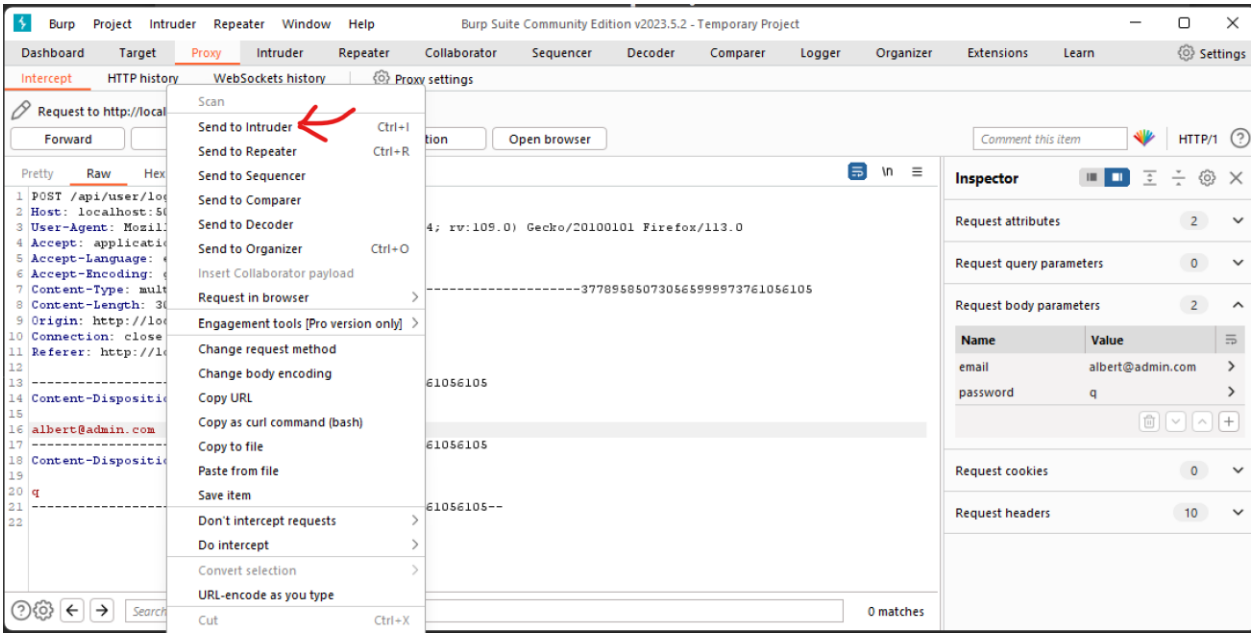
An example would be using burp suite to brute force the password of albert by sending a huge number of password combinations to try to guess albert's password.

Open your Firefox browser and intercept it using burp suite and stop at the login intercept as shown.



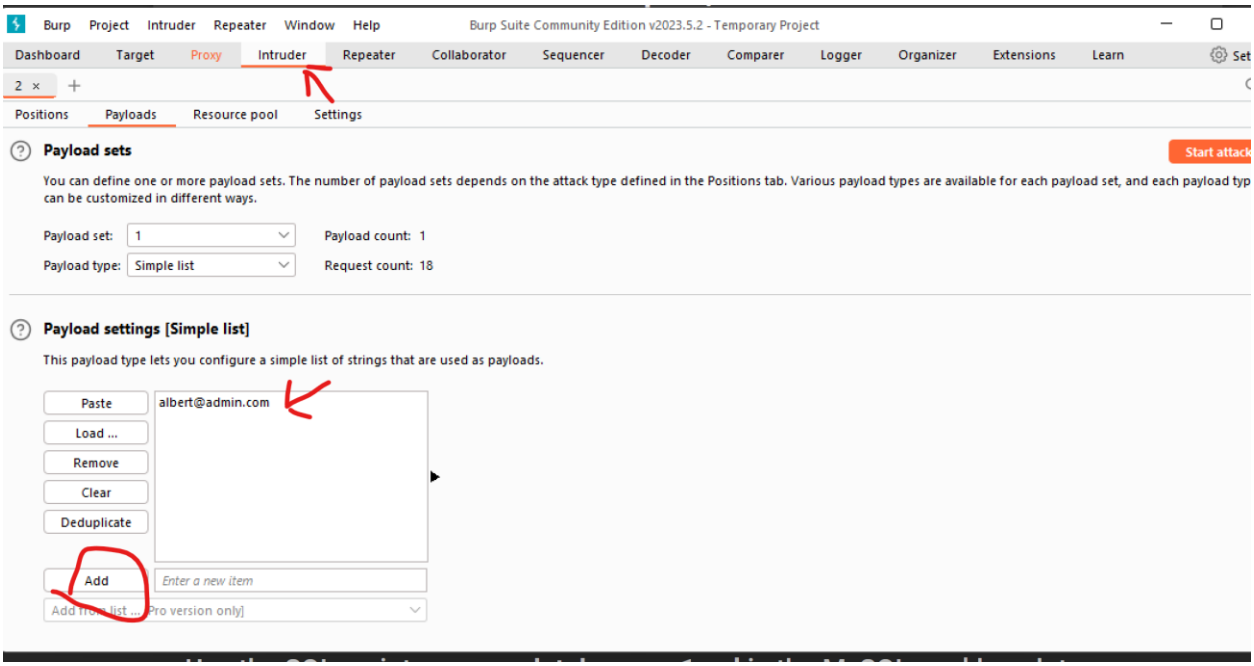
Step 3:

Right click and select Send to Intruder to set up the brute forcing.



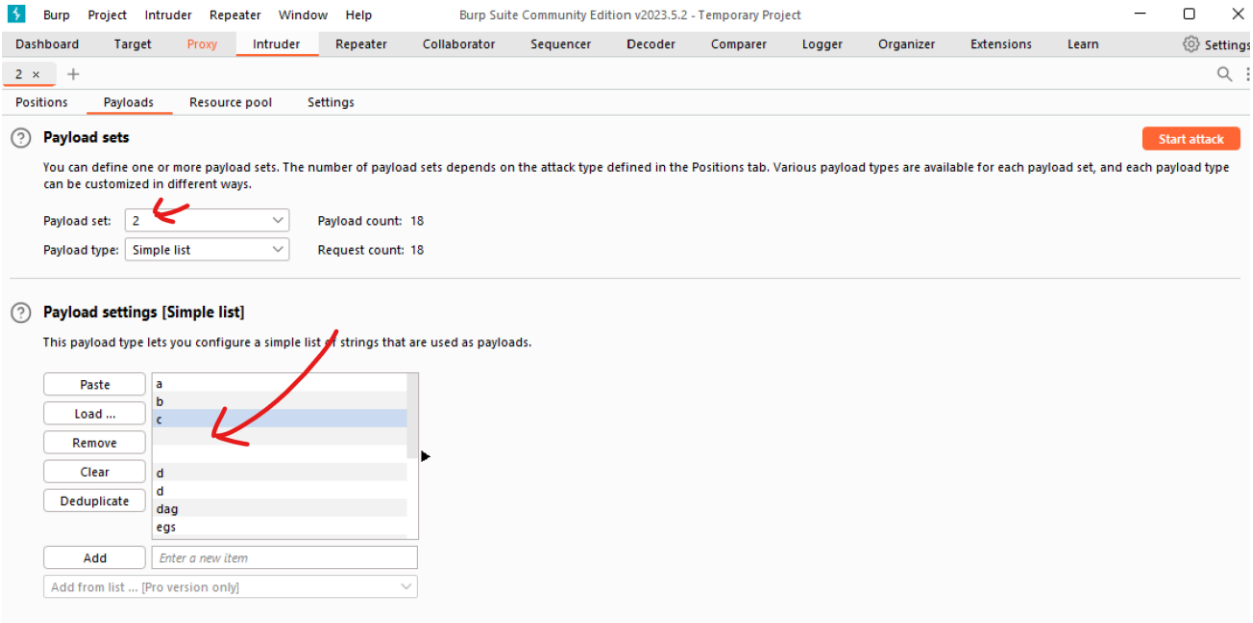
Step 4:

Click on Intruder at the top and add albert@admin.com into eh payload settings.



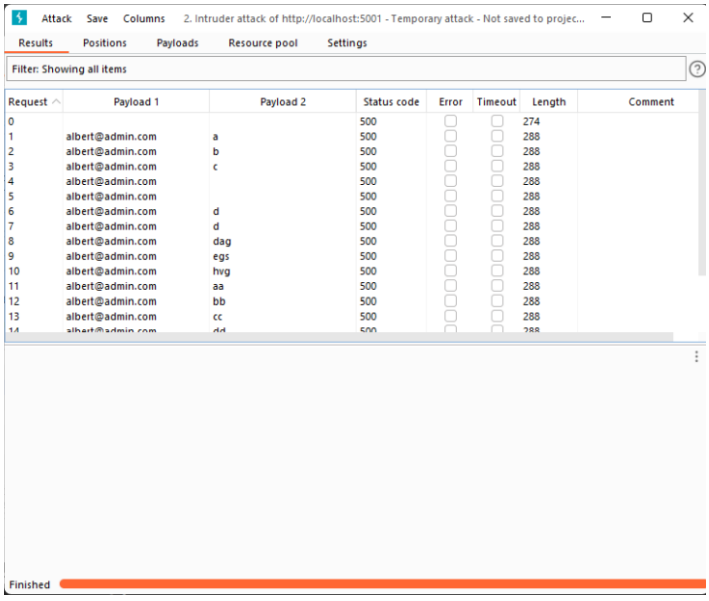
Step 5:

Select 2 in the payload set and fill in the passwords that we will try to use for brute forcing.



Step 6:

After adding the passwords click on start attack and you should see something as shown below where they will brute force the login page using the passwords that you provided.



7.1.2 How to rectify vulnerability - Limiting brute forcing attacks

Step 1:

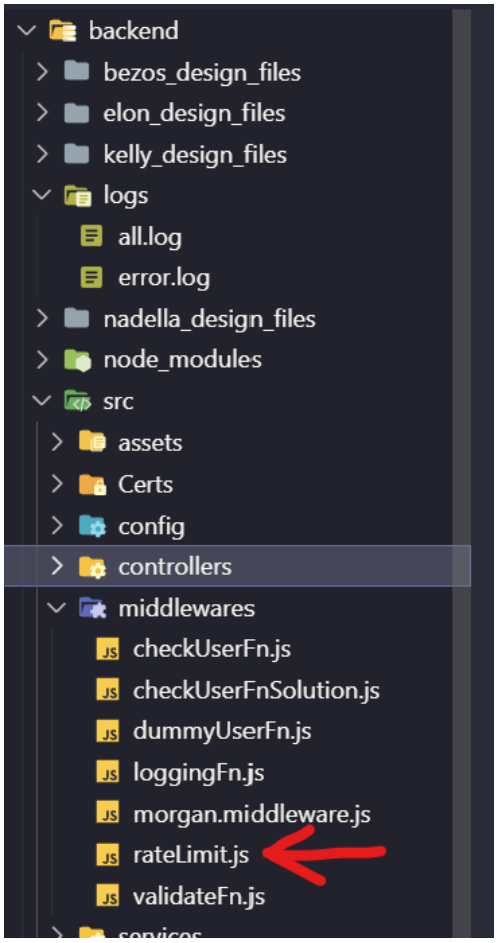
Let install a library called express-rate-limit which will help us to limit the number of requests send by a user at a given time to prevent brute forcing attacks.

Run the following command in your backend terminal:

npm install express-rate-limit

Step 2:

Next, we will create a new file called rateLimit.js under /backend/src/middlewares for us to limit the number of tries allowed for the login.

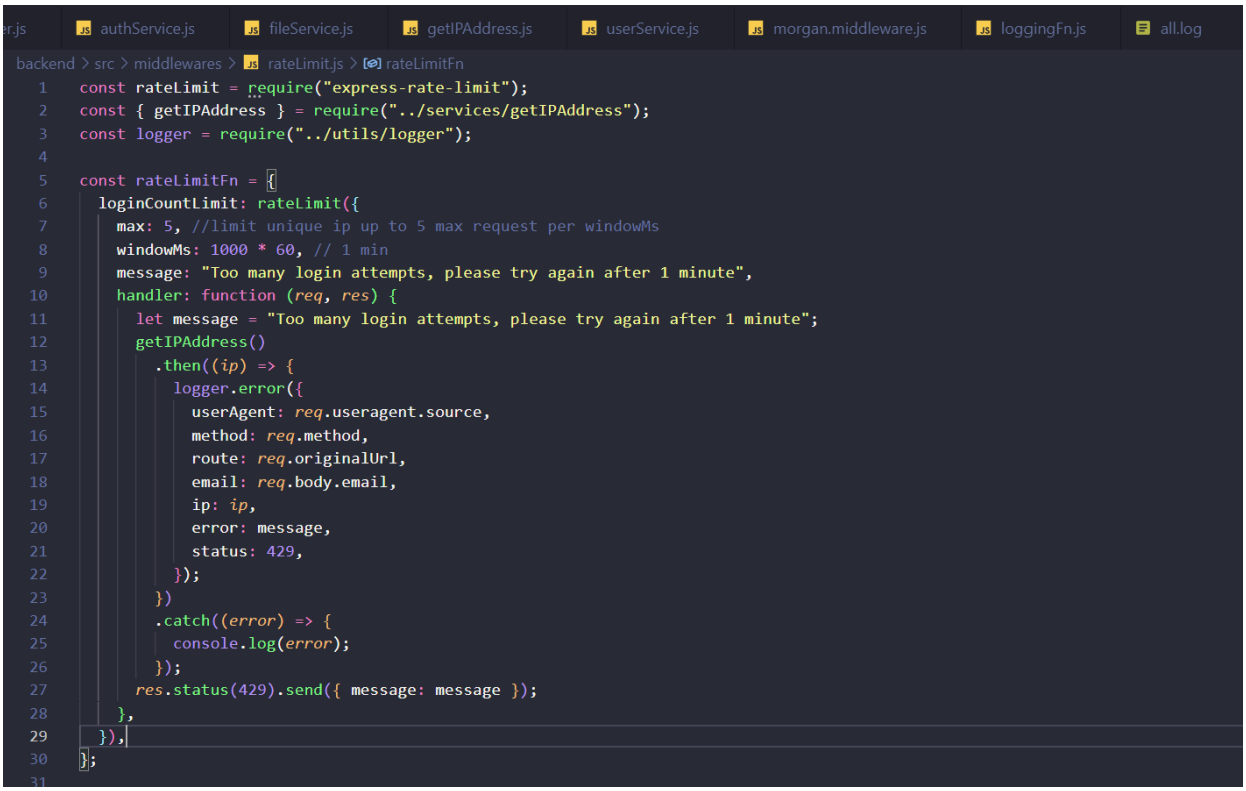


Step 3:

Add the following code into the rateLimit.js file under /backend/src/middlewares.

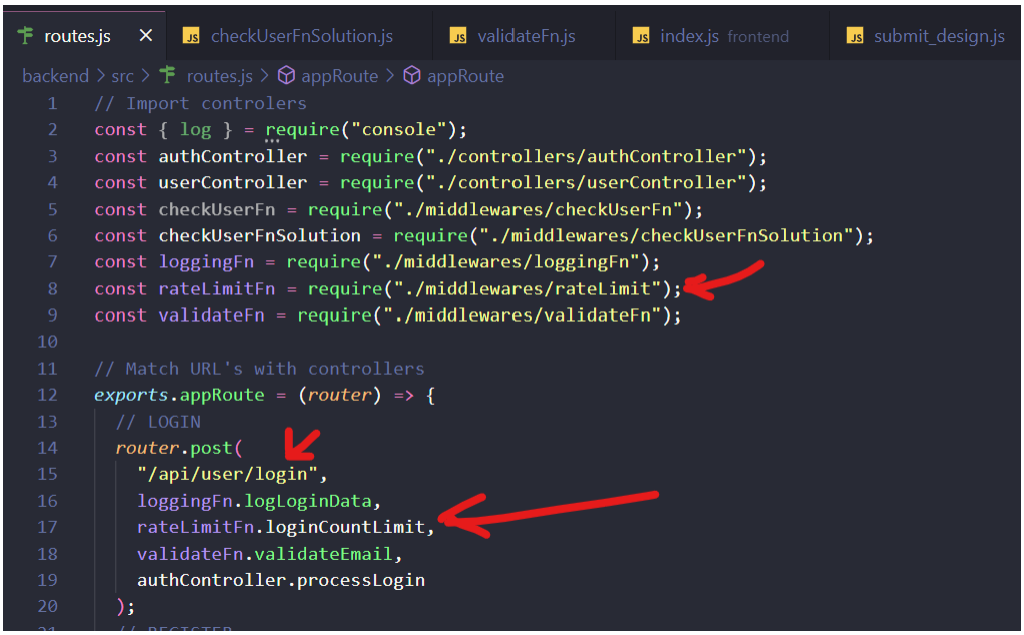
We will allow the user to login in a maximum of 5 times and if they are still not logged in they will need to wait for 1 minute before being able to attempt to login again. We will also add a log here to record if anyone has passed the 5 login tries.

Reference: https://blog.logrocket.com/rate-limiting-node-js/



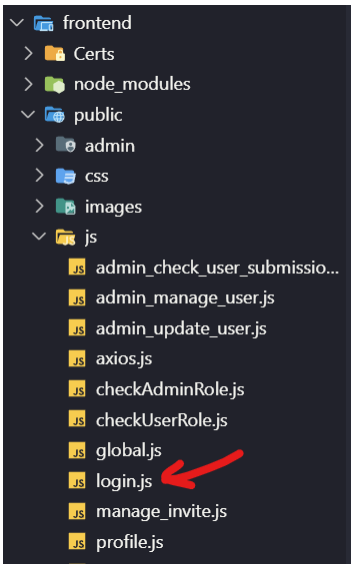
Step 4:

We will now add the rateLimit.js into routes.js under /backend/src/routes.js. Make sure to import the rateLimitFn into routes.js as shown below.



Step 5:

Head over to the frontend code and look for the file called login.js under /frontend/public/js/login.js.



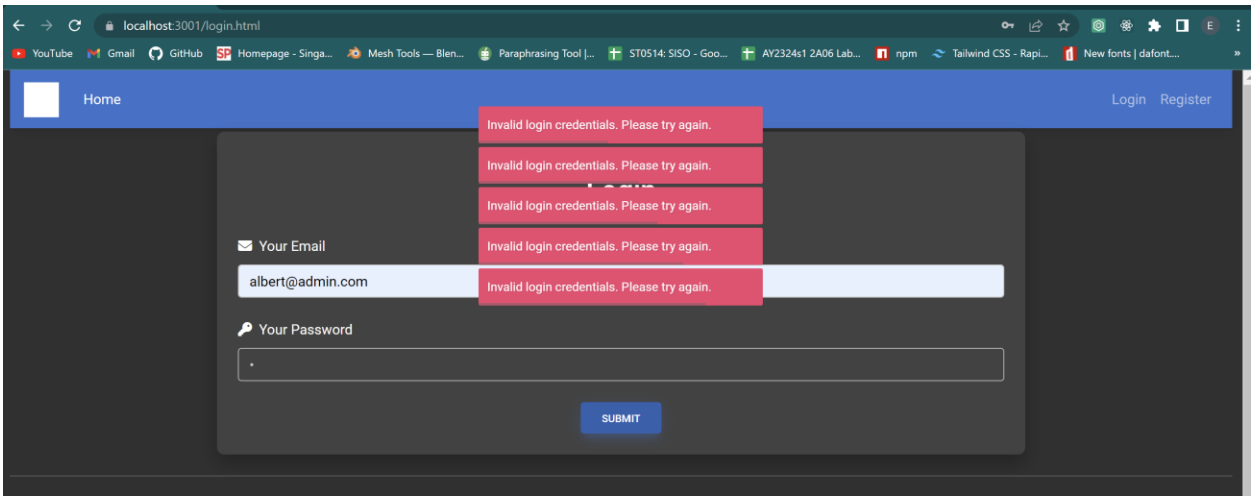
Step 6:

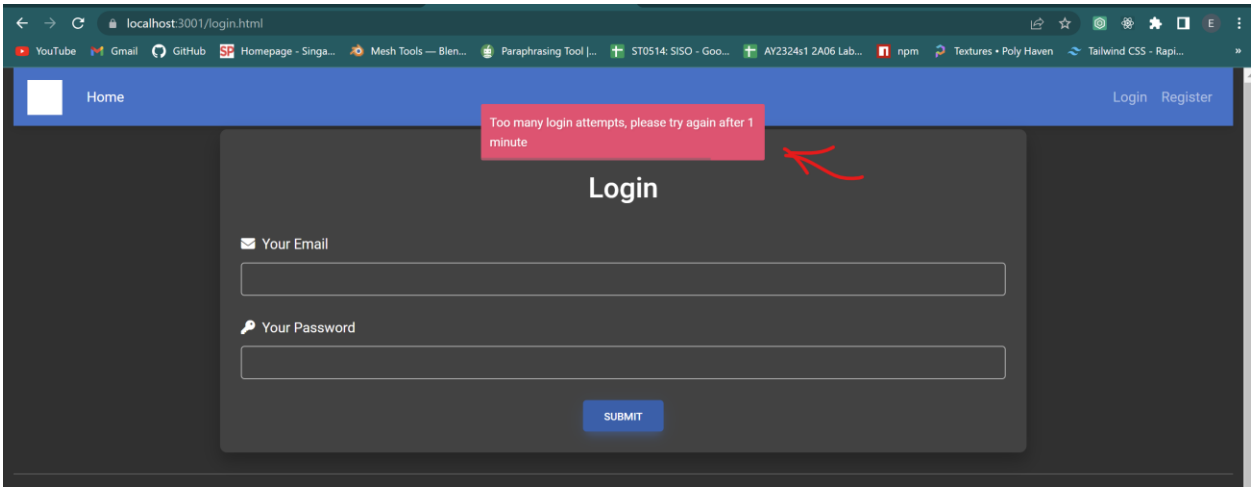
Change the static response "Unable to login. Check your email and password" to the following response message. This is so that we would be able to see the message when we get pass the 5 tries.

7.1.3 Evidence that the vulnerability has been rectified - Limiting brute forcing attacks

Step 1:

Now we will see if our rateLimit is working let's clear the error.log and try to login by clicking it 6 times at one go using an incorrect password. You should see a different alert message after submitting more than 6 times in a row stating that you have done too many requests at once.

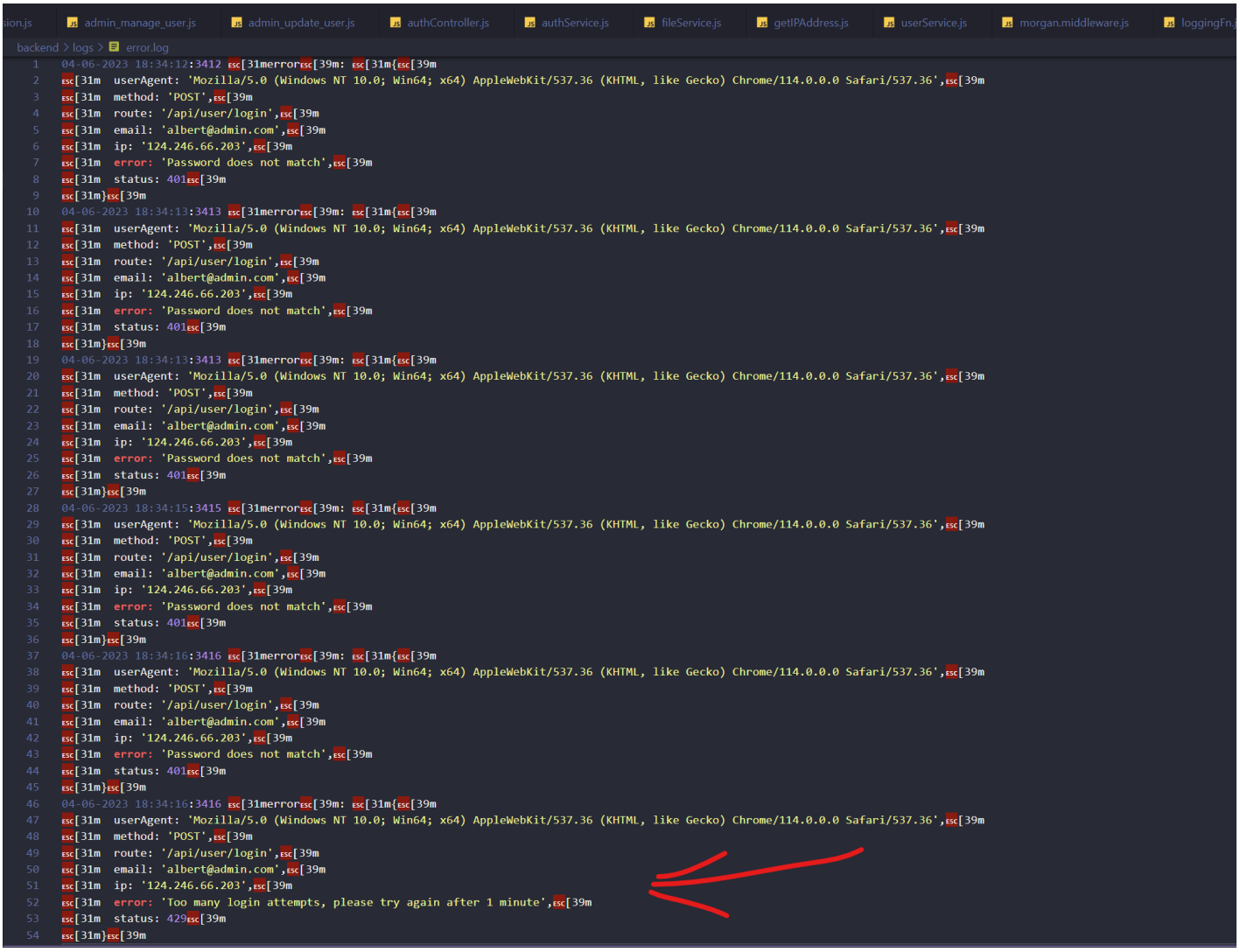




Step 2:

Head over to the error.log file. This time you should see that there are 5 login error attempts and the 6th being the rateLimit Function that we created, stopping anymore further requests including logging it as shown below after reaching the maximum of 5 tries for the number of allowed requests per window per user.

This shows that we have successfully limited the login count to prevent any brute forcing.

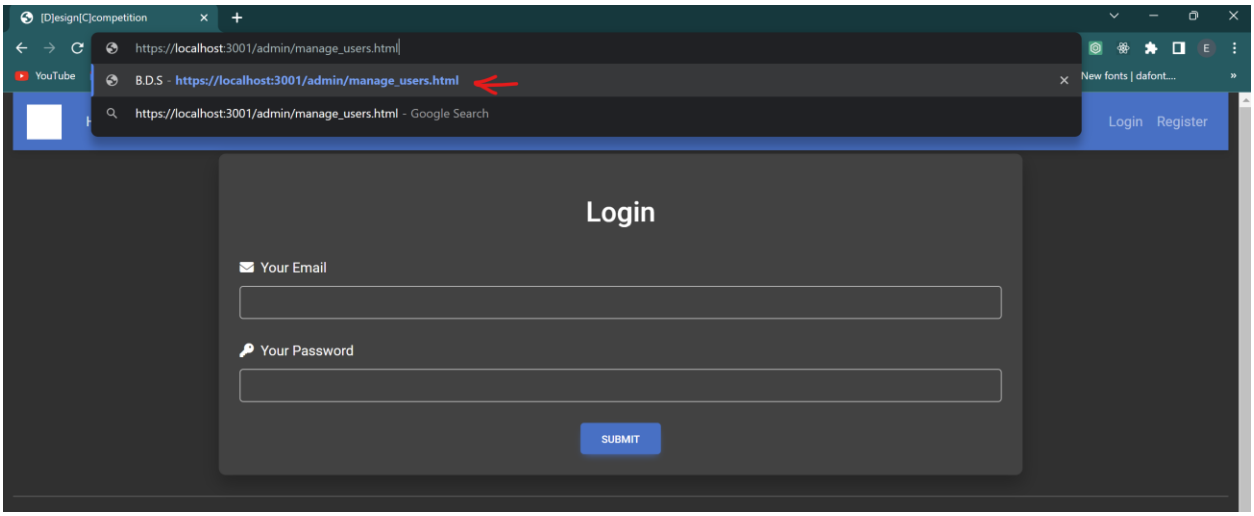


7.2.1 How to exploit vulnerability - About URL Link access control (Broken Access Control)

Step 1:

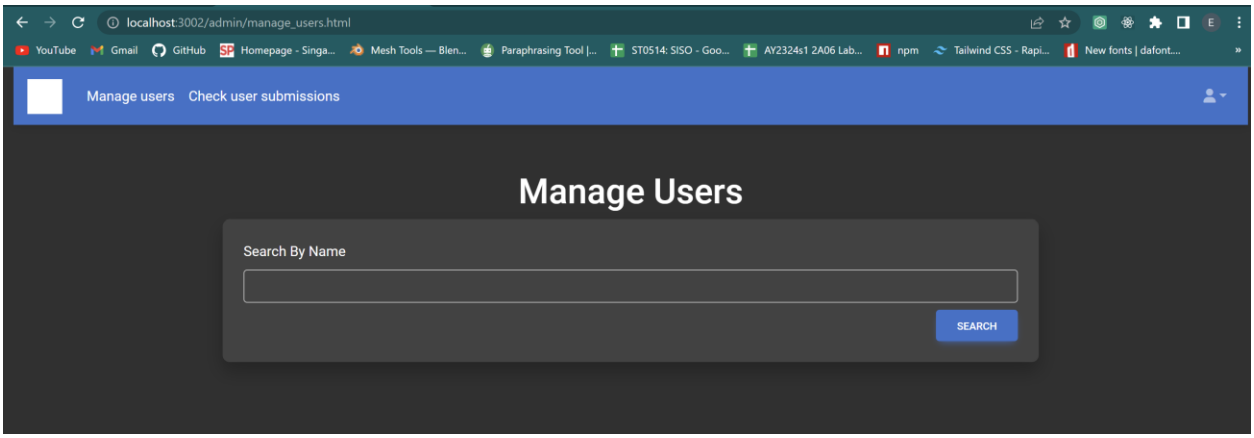
Paste the following URL link to see if we can access the admin page without logging in.

Link: yourLocalhostPort/admin/manage_users.html



Step 2:

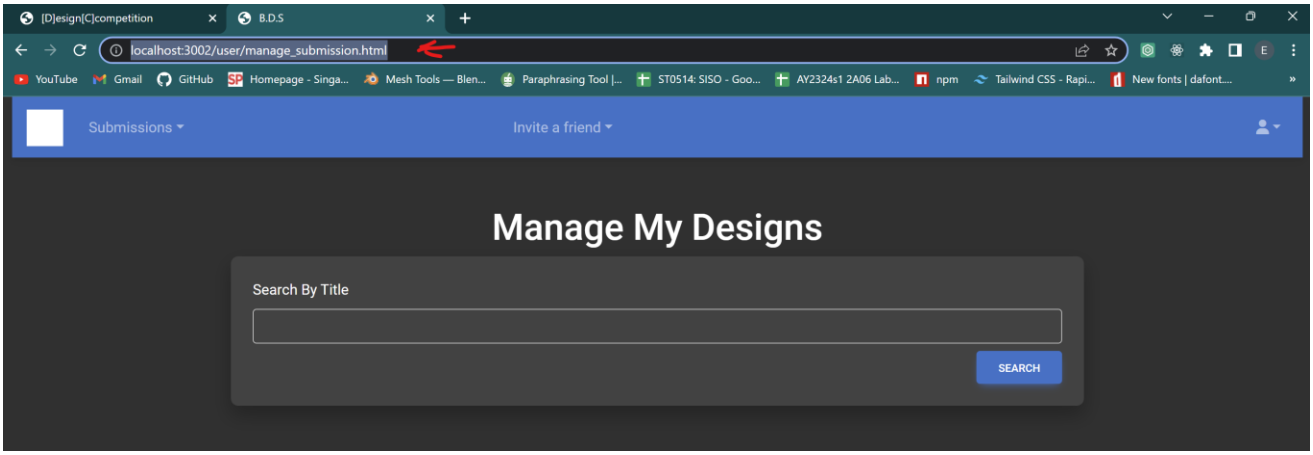
You will see that you would be able to access the admin page without logging in as an admin.



Step 3:

Similarly, we can also access the user page as well when we type the following link.

Link: yourLocalhostPort/user/manage_submission.html

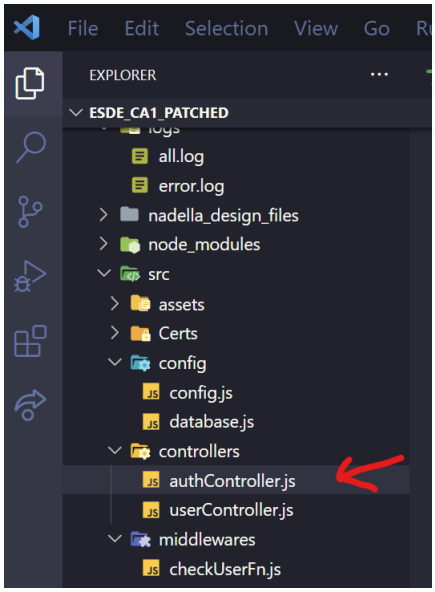


7.2.2 How to rectify vulnerability - About URL Link access

Step 1:

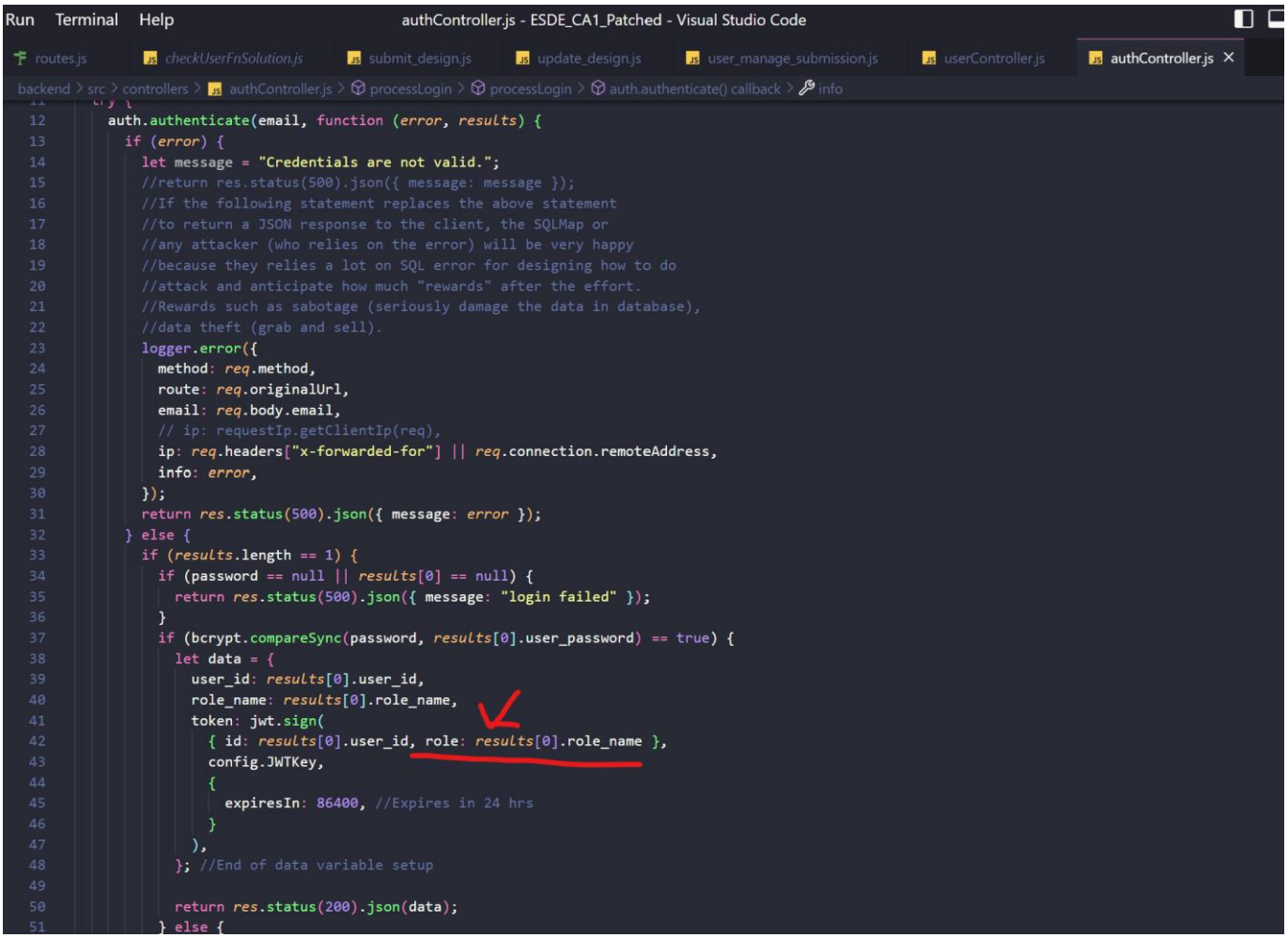
To rectify this vulnerability, let us first go to go to authController.js to do some changes to the token.

Under /backend/src/controllers/authController.js



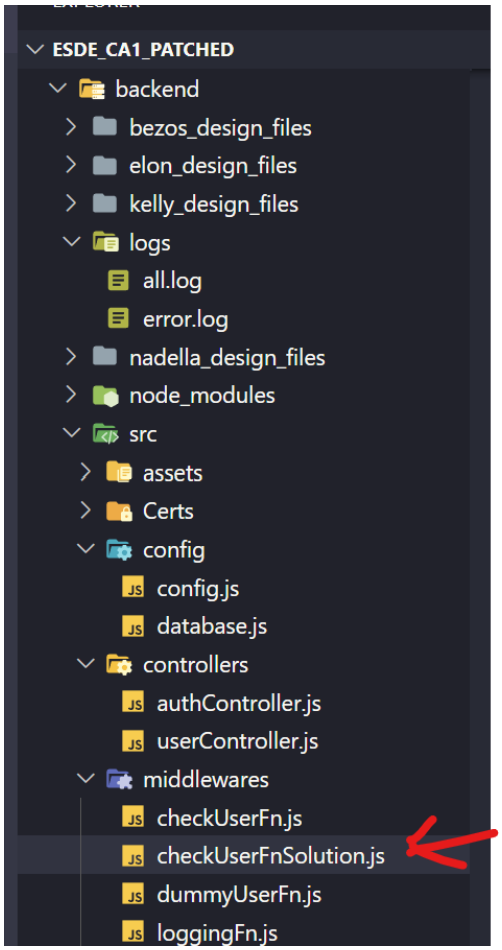
Step 2:

Apply the following changes to the exports.processLogin. This will allow us to check the role of the current logged in user through the token.



Step 3:

Next, let us go into the checkUserFnSolution.js file located at /backend/src/middlewares/checkUserFnSolution.js



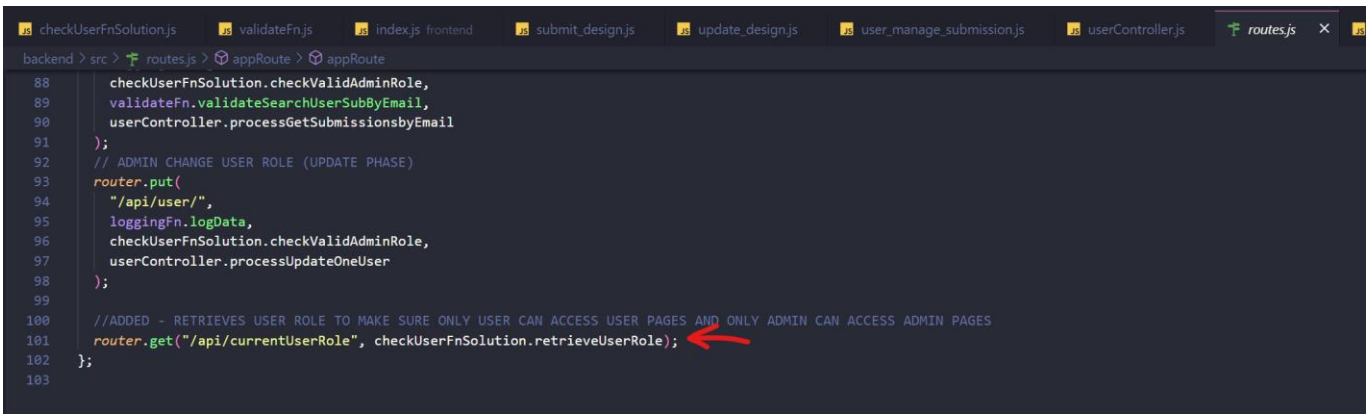
Step 4:

Add the following code as shown below located at /backend/src/middlewares/checkUserFnSolution.js for us to retrieve the role from the token and verify if the token has a user or admin role or no token at all in the front end.

*Note: You can ignore lines 131-138 for now as this will be used only for the 6th vulnerability

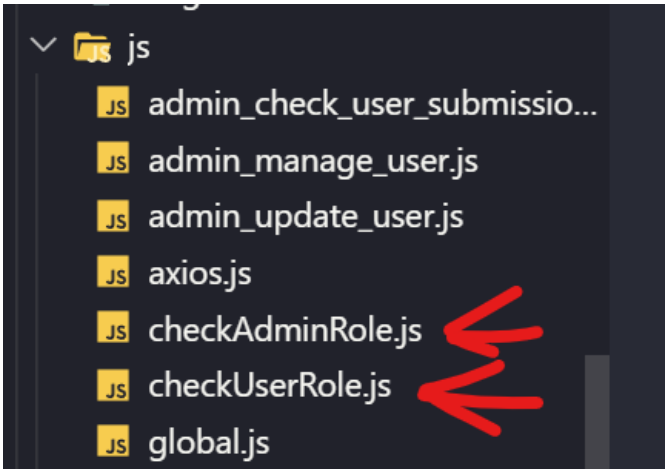
Step 5:

Next, head over to the routes.js file under /backend/src/routes.js to add the code that we previously added in Step 4 into a new endpoint created called currentUserRole.



Step 6:

After adding the endpoint in routes.js we will now go to the front-end code to add some files that will help to check the authentication of the user. Go to /frontend/public/js folder and add two js files as shown below.



Step 7:

Lets' go into the checkAdminRole.js file first located under /frontend/public/js/checkAdminRole.js and add the following code.

We will use the endpoint that we created earlier to check if there is a token to begin with if there isn't, we will direct them back to the login page. Also, they will be able to go to the admin page if they have a valid token but will be redirected back to the user page if a non-admin user tries to access the admin page.

routes.js | checkAdminRole.js | checkUserRole.js | validateFn.js | che

frontend > public > js > checkAdminRole.js > ...

```
function checkUserRole() {
  const baseUrl = "https://localhost:5000";

  let token = sessionStorage.getItem("token");
  let userId = sessionStorage.getItem("user_id");
  axios({
    headers: {
      user: userId,
      authorization: `Bearer ${token}`,
    },
    method: "get",
    url: baseUrl + "/api/currentUserRole/",
  })
    .then(function (response) {
      //retrieve the role of the user
      if (response.data.role == "user") {
        window.location.assign(
          "https://localhost:3001/user/manage_submission.html"
        );
      } else if (response.data.role != "admin") {
        sessionStorage.removeItem("token");
        sessionStorage.removeItem("user_id");
        sessionStorage.removeItem("role_name");
        window.location.assign("https://localhost:3001/login.html");
      }
    })
    .catch(function (response) {
      //Handle error
      console.dir(response);
      sessionStorage.removeItem("token");
      sessionStorage.removeItem("user_id");
      sessionStorage.removeItem("role_name");
      window.location.assign("https://localhost:3001/login.html");
    });
}
checkUserRole();

```

Step 8:

Let us go to the checkUserRole.js file located under /frontend/public/js/checkUserRole.js and add the following code.

We will use the endpoint that we created earlier to check if there is a token to begin with if there is not we will direct them back to the login page. Also, they will be able to go to the user page if they have a valid token but will be redirected back to the admin page if an admin user tries to access the user page.

routes.js | checkAdminRole.js | checkUserRole.js | validateFn.js | checkUs

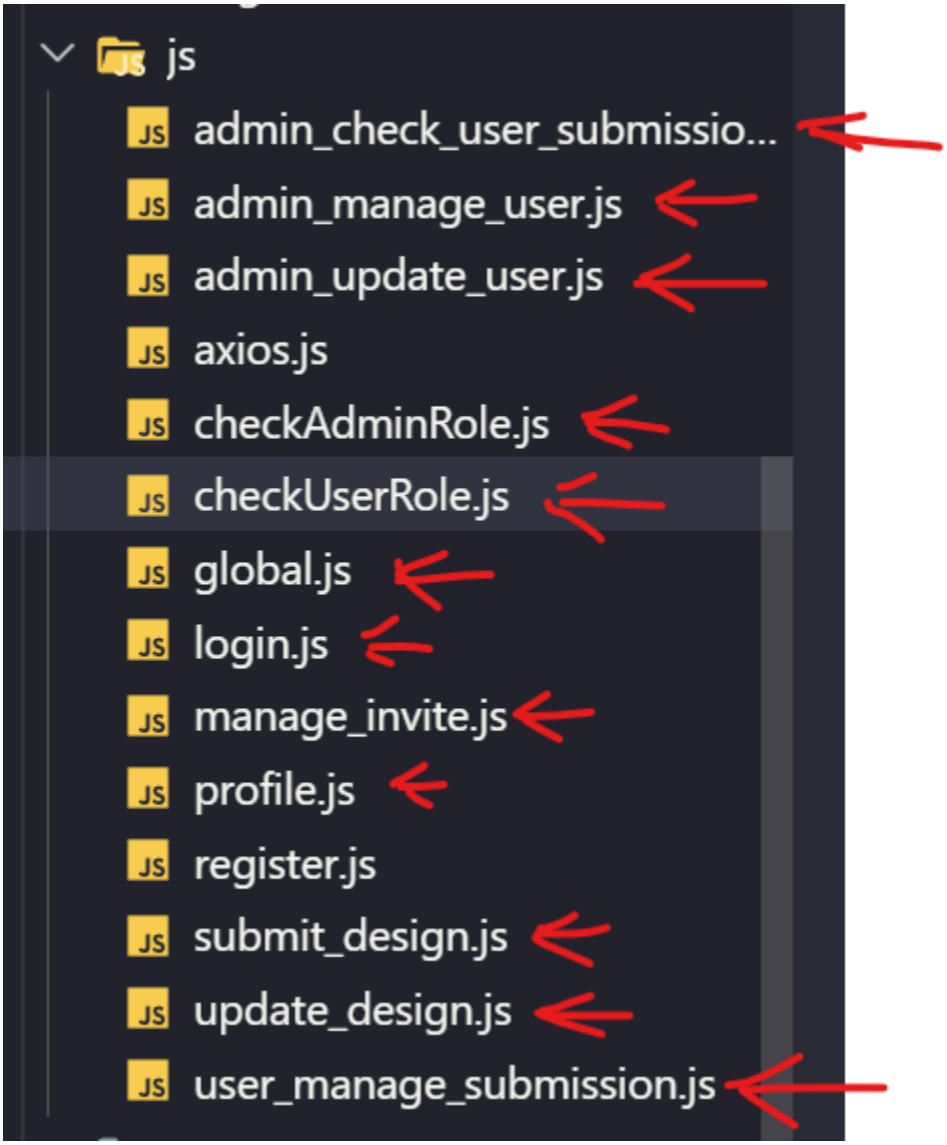
frontend > public > js > checkUserRole.js > ...

```
function checkUserRole() {
  const baseUrl = "https://localhost:5000";

  let token = sessionStorage.getItem("token");
  let userId = sessionStorage.getItem("user_id");
  axios({
    headers: {
      user: userId,
      authorization: `Bearer ${token}`,
    },
    method: "get",
    url: baseUrl + "/api/currentUserRole/",
  })
    .then(function (response) {
      //retrieve the role of the user
      if (response.data.role == "admin") {
        window.location.assign(
          "https://localhost:3001/admin/manage_users.html"
        );
      } else if (response.data.role != "user") {
        sessionStorage.removeItem("token");
        sessionStorage.removeItem("user_id");
        sessionStorage.removeItem("role_name");
        window.location.assign("https://localhost:3001/login.html");
      }
    })
    .catch(function (response) {
      //Handle error
      console.dir(response);
      sessionStorage.removeItem("token");
      sessionStorage.removeItem("user_id");
      sessionStorage.removeItem("role_name");
      window.location.assign("https://localhost:3001/login.html");
    });
}
checkUserRole();

```

Step 9:

You would realize that the code above uses **session storage** instead of local storage. This is so that when the user exits the page without logging out, the token will automatically be deleted to prevent any unwanted use of the token.

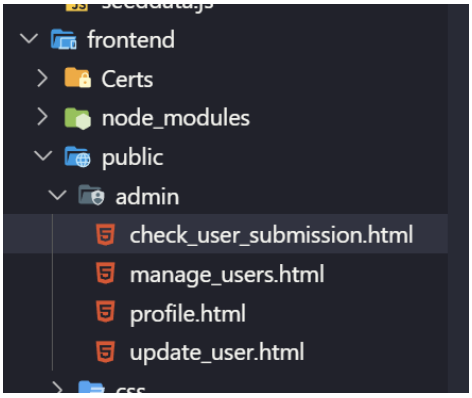
So, please do change all local storage to session storage for all the JavaScript files as shown in the bottom that has a red arrow pointing to it.



Step 10:

Next, let us go to the admin html pages and add a code to call the checkAdminRole.js file that we created just now for each of the respective admin role pages.

Under /frontend/public/admin/



Step 11:

Let us **add the following code shown below to the above 4 admin html pages in the head tag** to prevent anyone with no access from accessing the admin pages.

*Note: Do remember to add a defer to allow the page to fully load before calling the function.

routes.js | checkAdminRole.js | checkUserRole.js | checkUserFnSolution.js | submit_design.js | check_user_submission.html

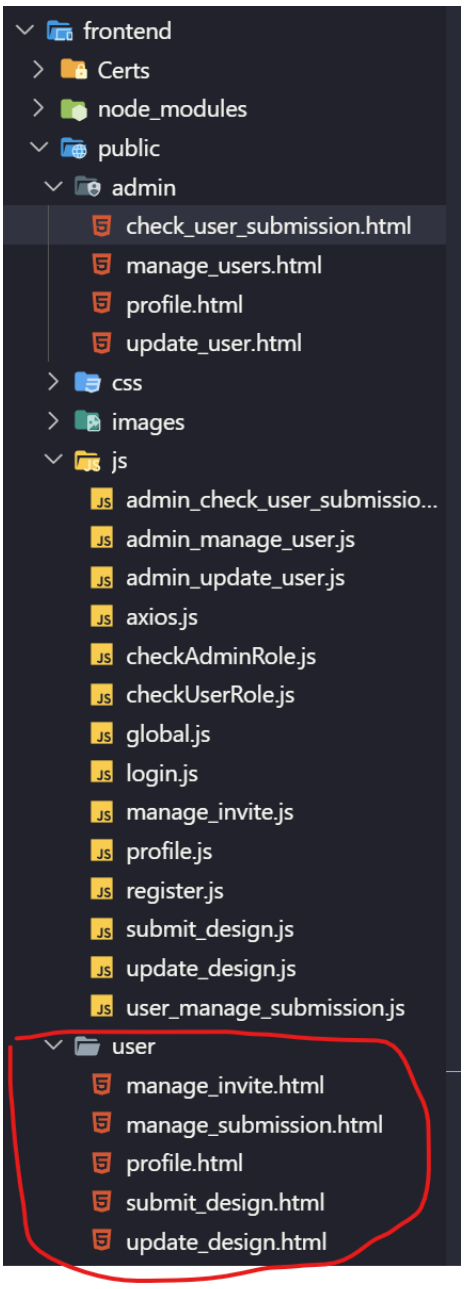
frontend > public > admin > check_user_submission.html > html > head

```
1   <!DOCTYPE html>
2   <html lang="en">
3
4   <head>
5       <title>B.D.S</title>
6       <!-- Font Awesome-->
7       <link rel="stylesheet" href="https://use.fontawesome.com/releases/v6.2.0/css/all.css">
8       <!-- Google Fonts-->
9       <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&amp;display=swap">
10      <!-- Bootstrap core CSS-->
11      <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.0/css/bootstrap.min.css" rel="stylesheet">
12      <!-- Material Design Bootstrap-->
13      <link href="https://cdnjs.cloudflare.com/ajax/libs/mdb-ui-kit/5.0.0/mdb.dark.min.css" rel="stylesheet"/>
14      <!-- Noty CSS-->
15      <link href="https://cdnjs.cloudflare.com/ajax/libs/noty/3.1.4/noty.css" rel="stylesheet">
16      <!-- Noty CSS Map-->
17      <link href="https://cdnjs.cloudflare.com/ajax/libs/noty/3.1.4/noty.css.map" rel="stylesheet">
18      <!-- Noty CSS Map-->
19      <link href="/css/site.css" rel="stylesheet">
20      <!-- Check if role is valid to prevent broken authentication -->
21      <script src="/js/checkAdminRole.js" defer></script>
22
23  </head>
```

Step 12:

Next, let us go to the user html pages and add a code to call the checkUserRole.js file that we created just now for each of the respective user role pages.

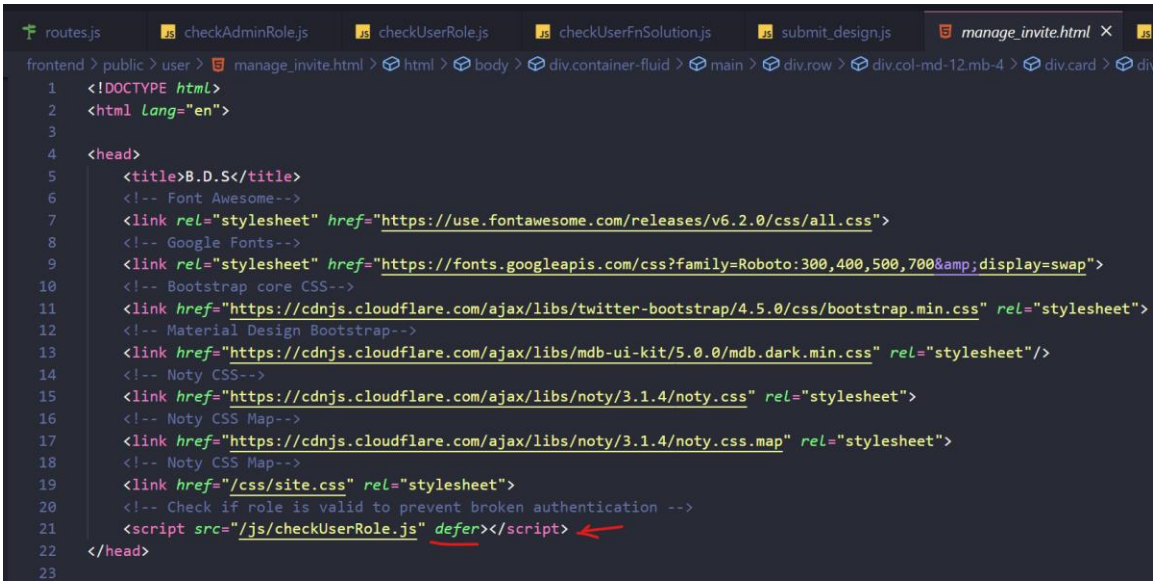
Under /frontend/public/user/



Step 13:

Let us **add the following code shown below to the above 5 user html pages in the head tag** to prevent anyone unauthenticated to access the user pages.

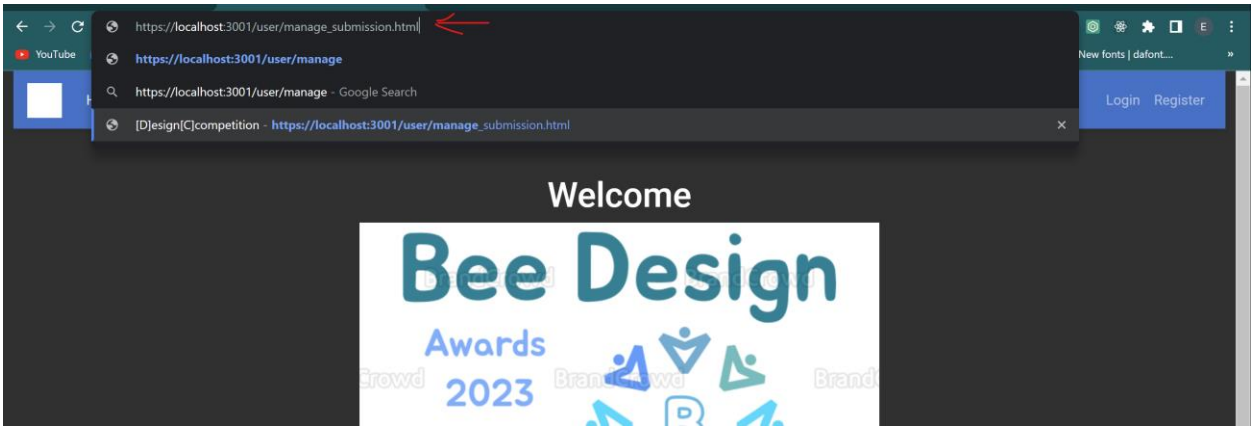
*Note: Do remember to add a defer to allow the page to fully load before calling the function.



7.2.3 Evidence that the vulnerability has been rectified - About URL Link access

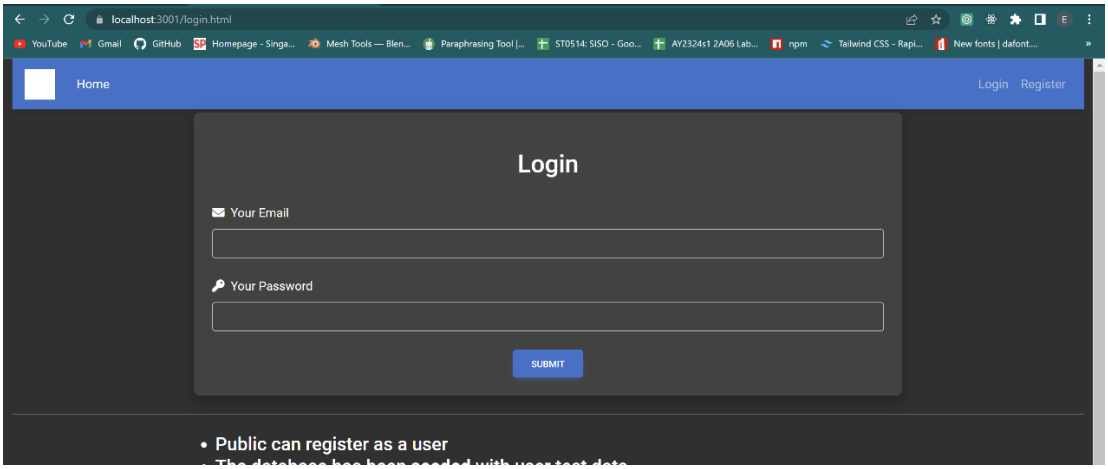
Step 1:

Now, let us check if the above changes have fixed the vulnerability. First, let us go into the webpage and paste the following into the URL link in the Welcome Page.



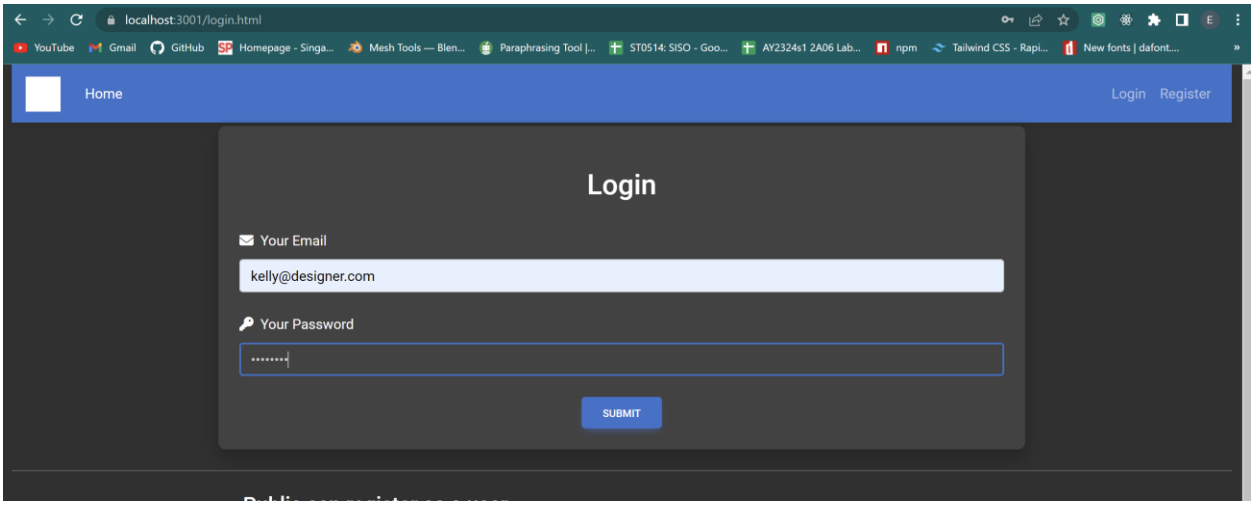
Step 2:

You should be able to see that the site will redirect you to the login page if we try to access the user page without logging in.



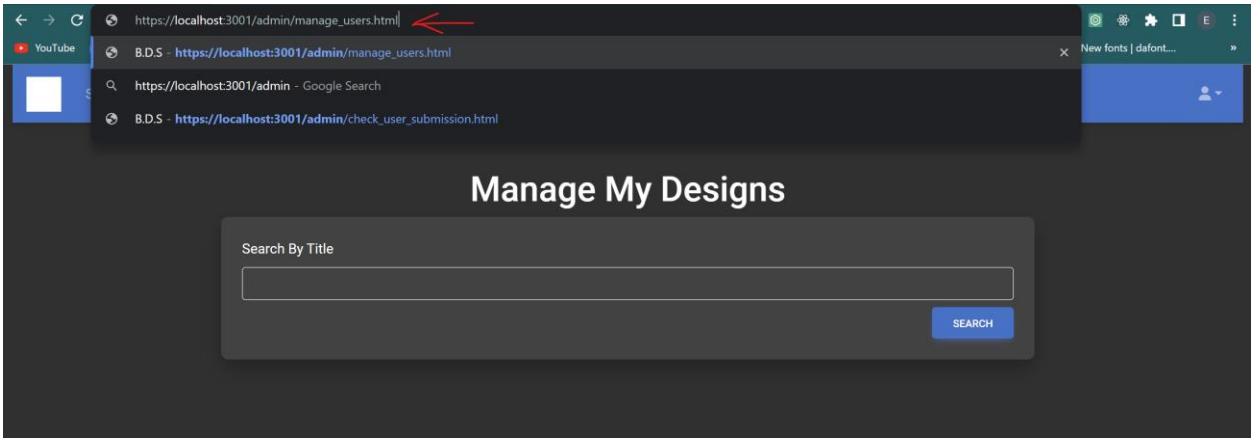
Step 3:

Let us now try to login using Kelly's Email and Password



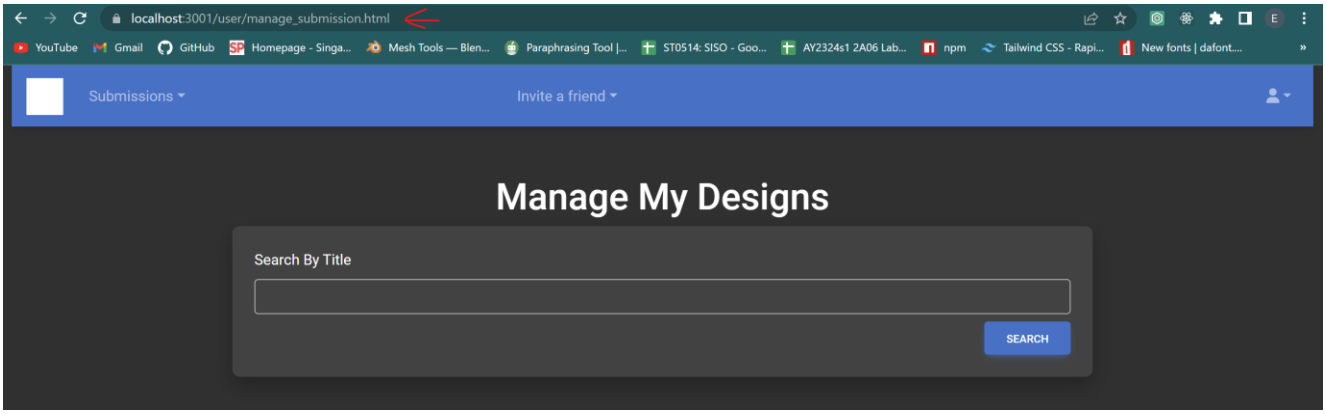
Step 4:

After logging in, let us enter the admin page URL to see if we can access the admin page even though our role is user.



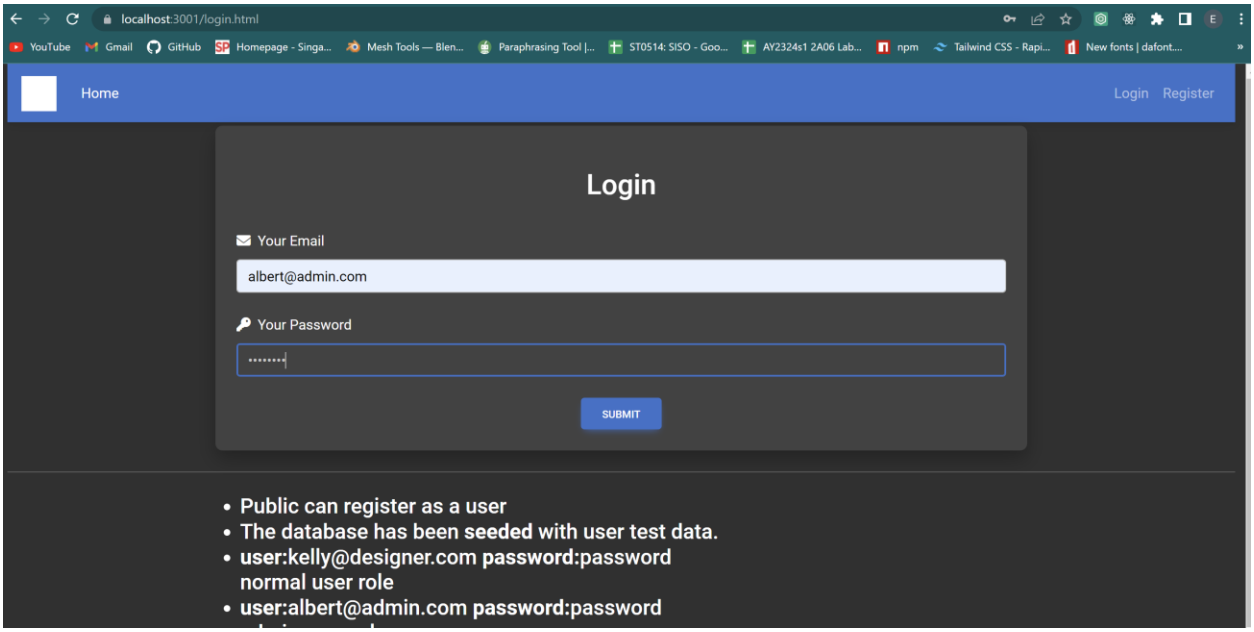
Step 5:

You should see that you are redirected back to the user page as Kelly is not an admin.



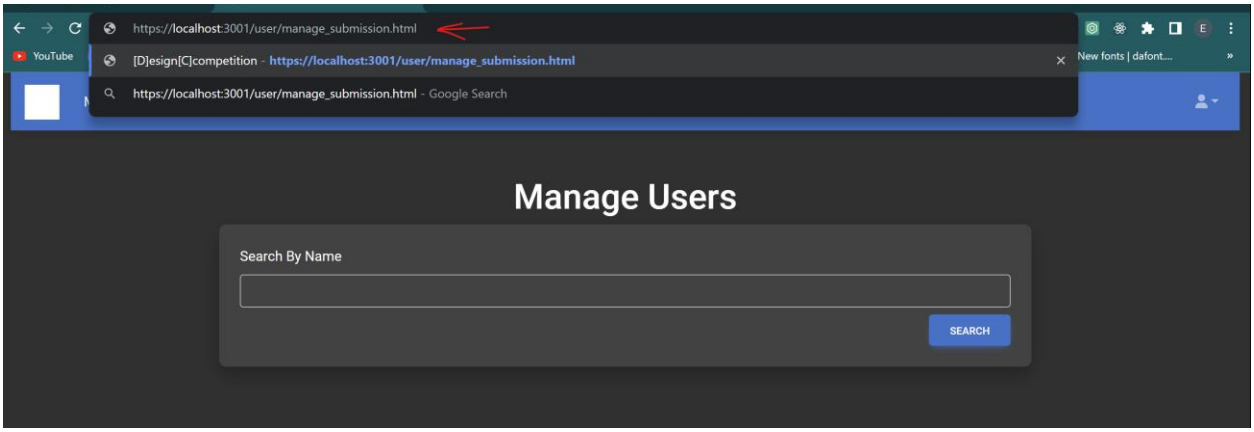
Step 6:

Next, let us try the same testing with an admin account this time. First, log out from Kelly's Account and Login using Albert's Account.



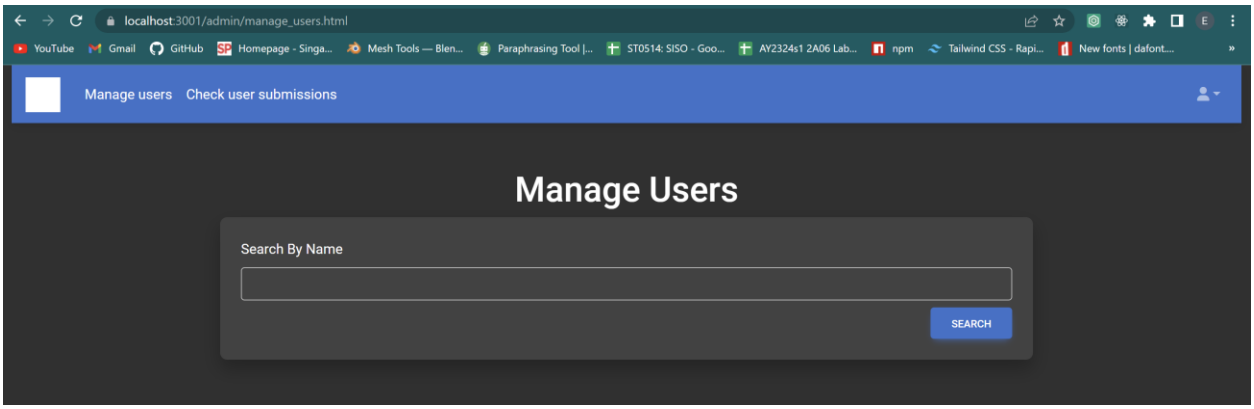
Step 7:

We will now change the above link to a user page URL and see if we can access the user page.



Step 8:

You should also be able to see that you have been redirected back to the admin page as albert is an admin role and not a user role.



7.3.0 Code Review – Preventing Future Vulnerabilities

Error handling and Nested Callbacks

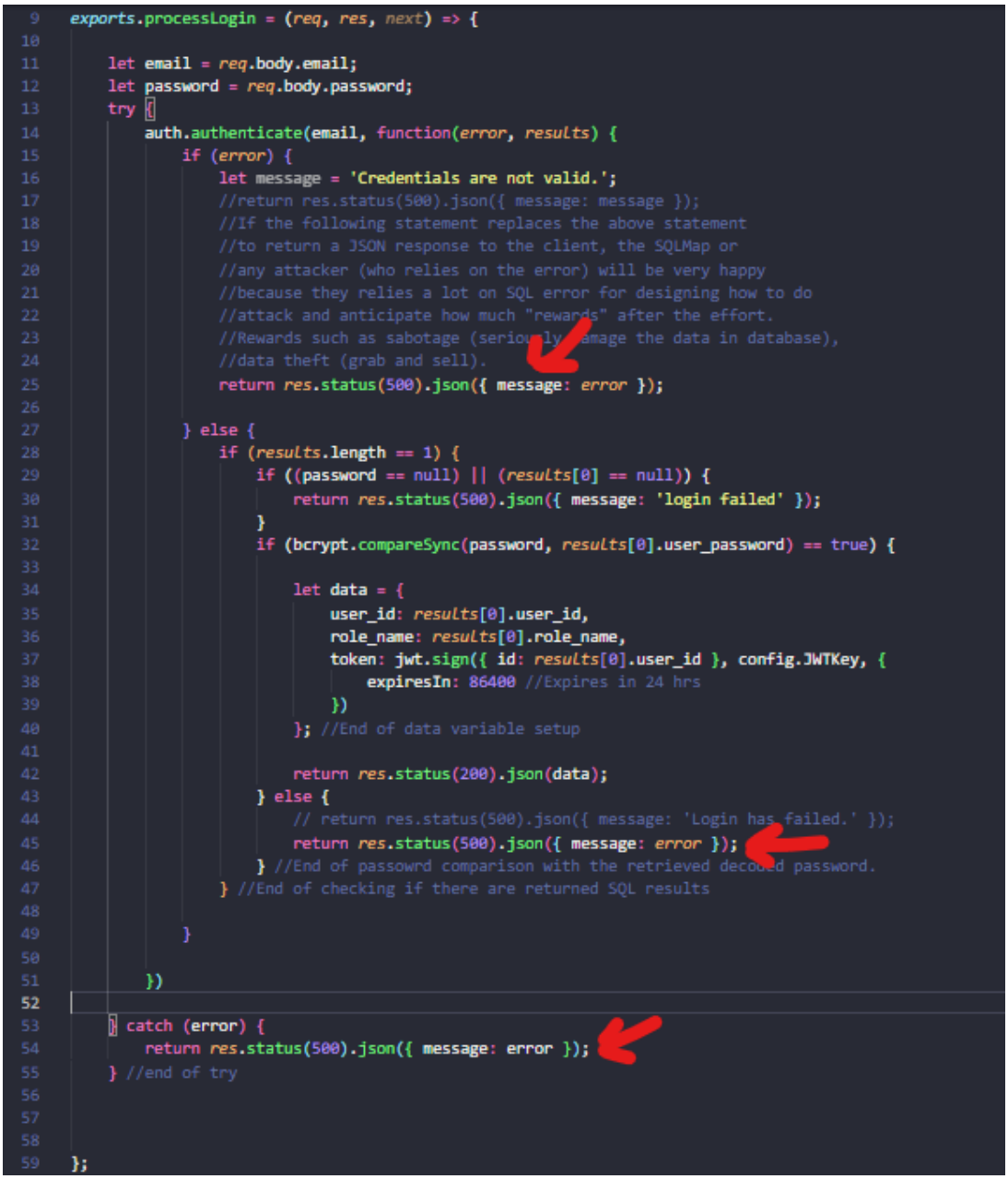
The application contains many different way to resolve the errors, where some endpoints handle the errors immediately.

A few examples are shown below:

Figure A:

```
61  exports.processRegister = (req, res, next) => {
62      console.log('processRegister running.');
63      let fullName = req.body.fullName;
64      let email = req.body.email;
65      let password = req.body.password;
66
67
68      bcrypt.hash(password, 10, async(err, hash) => {
69          if (err) {
70              console.log('Error on hashing password');
71              return res.status(500).json({ statusMessage: 'Unable to complete registration' });
72          } else {
73
74                  results = user.createUser(fullName, email, hash, function(results, error){
75                    if (results!=null){
76                      console.log(results);
77                      return res.status(200).json({ statusMessage: 'Completed registration.' });
78                    }
79                    if (error) {
80                      console.log('processRegister method : callback error block section is running.');
81                      console.log(error, '============================================================');
82                      return res.status(500).json({ statusMessage: 'Unable to complete registration' });
83                    }
84                    });//End of anonymous callback function
85
86
87          }
88      });
89
90
91  }; // End of processRegister
```

Figure B:



```
exports.processLogin = (req, res, next) => {

    let email = req.body.email;
    let password = req.body.password;
    try {
        auth.authenticate(email, function(error, results) {
            if (error) {
                let message = 'Credentials are not valid.';
                //return res.status(500).json({ message: message });
                //If the following statement replaces the above statement
                //to return a JSON response to the client, the SQLMap or
                //any attacker (who relies on the error) will be very happy
                //because they relies a lot on SQL error for designing how to do
                //attack and anticipate how much "rewards" after the effort.
                //Rewards such as sabotage (seriously damage the data in database),
                //data theft (grab and sell).
                return res.status(500).json({ message: error });

            } else {
                if (results.length == 1) {
                    if ((password == null) || (results[0] == null)) {
                        return res.status(500).json({ message: 'login failed' });
                    }
                    if (bcrypt.compareSync(password, results[0].user_password) == true) {

                        let data = {
                            user_id: results[0].user_id,
                            role_name: results[0].role_name,
                            token: jwt.sign({ id: results[0].user_id }, config.JWTKey, {
                                expiresIn: 86400 //Expires in 24 hrs
                            })
                        }; //End of data variable setup

                        return res.status(200).json(data);
                    } else {
                        // return res.status(500).json({ message: 'Login has failed.' });
                        return res.status(500).json({ message: error });
                    } //End of passowrd comparison with the retrieved decoded password.
                } //End of checking if there are returned SQL results

            }

        })

    } catch (error) {
        return res.status(500).json({ message: error });
    } //end of try


};
```

The examples above are what we usually call ”callback hell” where multiple callbacks are nested on top of each other. This can get even more unreadable when more features are added.

So, the bottom 2 images that rectifies processLogin and processRegister would be the way to go to avoid any confusion and callback hell by using try and catch for all authController and userController.js.

processLogin:

```
exports.processLogin = async (req, res, next) => {
  let email = req.body.email;
  let password = req.body.password;
  try {
    var results = await auth.authenticate(email);
    if (results.length == 1) {
      if (password == null || results[0] == null) {
        throw "Login has failed.";
      }
      if (!bcrypt.compareSync(password, results[0].user_password) == true)
        getIPAddress()
          .then((ip) => {
            logger.error({
              userAgent: req.useragent.source,
              method: req.method,
              route: req.originalUrl,
              email: req.body.email,
              ip: ip,
              error: "Password does not match",
              status: 401,
            });
          })
          .catch((error) => {
            console.log(error);
          });
        throw "Invalid login credentials. Please try again.";
      } else {
        let data = {
          user_id: results[0].user_id,
          role_name: results[0].role_name,
          token: jwt.sign(
            { id: results[0].user_id, role: results[0].role_name },
            config.JWTKey,
            {
              expiresIn: 86400, //Expires in 24 hrs
            }
          ),
        }; //End of data variable setup
        return res.status(200).json(data);
      }
    }
  } catch (error) {
    if (error == "Invalid login credentials. Please try again.") {
      let message = "Invalid login credentials. Please try again.";
      return res.status(500).json({ message: message });
    } else {
      console.log(error);
      let message = "Login has failed.";
      return res.status(500).json({ message: message });
    }
  }
}; //End of processLogin
```

processRegister:

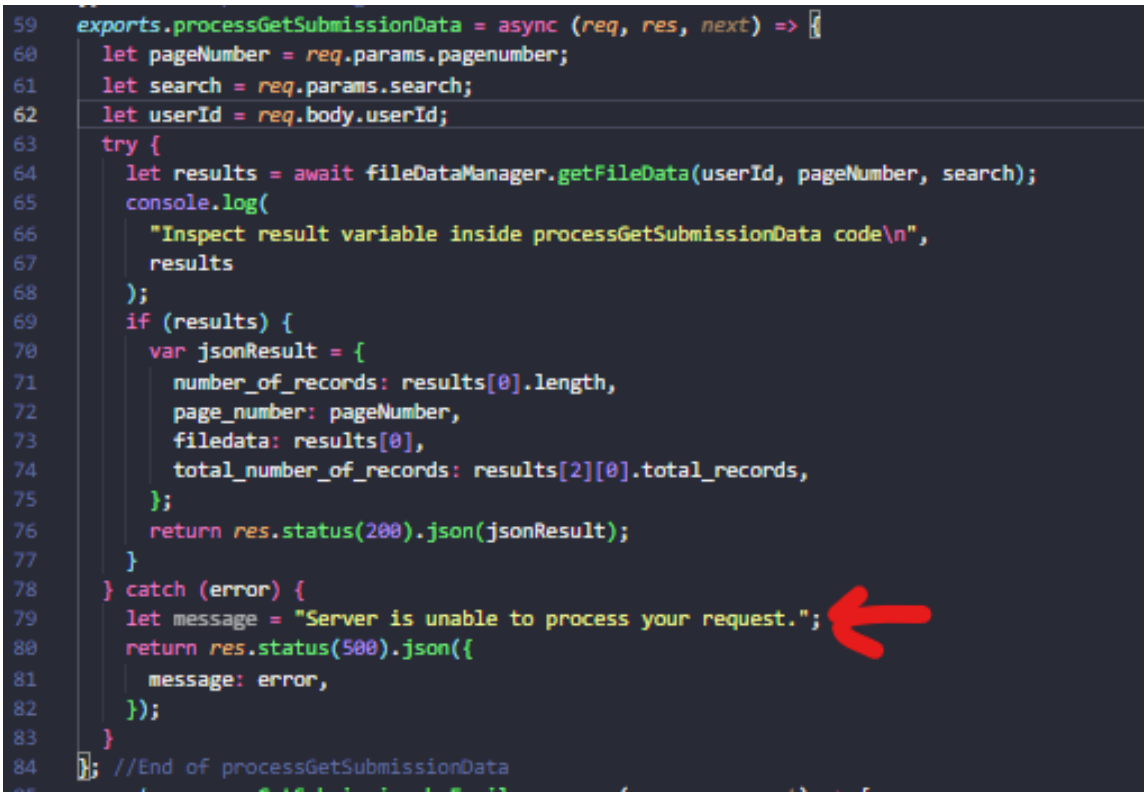
```
exports.processRegister = (req, res, next) => {
  console.log("processRegister running.");
  let fullName = req.body.fullName;
  let email = req.body.email;
  let password = req.body.password;

  bcrypt.hash(password, 10, async (error, hash) => {
    try {
      if (error) {
        console.log("Error on hashing password");
        throw "Error on hashing password.";
      } else {
        var results = await user.createUser(fullName, email, hash);
        if (results != null) {
          console.log(results);
          let message = "Completed registration.";
          return res.status(200).json({ message: message });
        }
      }
    } catch (error) {
      console.log(error);
      if (error.errno == 1062) {
        let message = "Email already exists.";
        return res.status(500).json({ message: message });
      } else {
        let message = "Unable to complete registration.";
        return res.status(500).json({ message: message });
      }
    }
  });
}; //End of processRegister

```

Standardizing the JSON Response Data

In the application, there are different types of ways used to send back a JSON Response.

In the example shown below, you can see that there is a declared variable message, but it was not used rather the message sends the error instead.

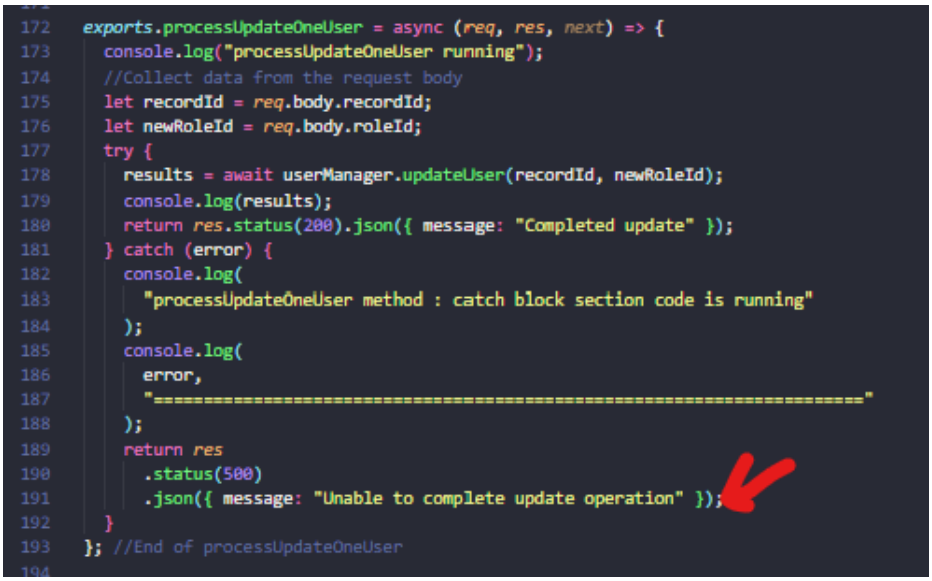
Figure C:



```
exports.processGetSubmissionData = async (req, res, next) => {
  let pageNumber = req.params.pagenumber;
  let search = req.params.search;
  let userId = req.body.userId;
  try {
    let results = await fileDataManager.getFileData(userId, pageNumber, search);
    console.log(
      "Inspect result variable inside processGetSubmissionData code\n",
      results
    );
    if (results) {
      var jsonResult = {
        number_of_records: results[0].length,
        page_number: pageNumber,
        filedata: results[0],
        total_number_of_records: results[2][0].total_records,
      };
      return res.status(200).json(jsonResult);
    }
  } catch (error) {
    let message = "Server is unable to process your request.";
    return res.status(500).json({
      message: error,
    });
  }
}; //End of processGetSubmissionData
```

On the other hand, there is also another JSON Response where it uses a string in the message instead of using a variable shown in Figure D.

Figure D:



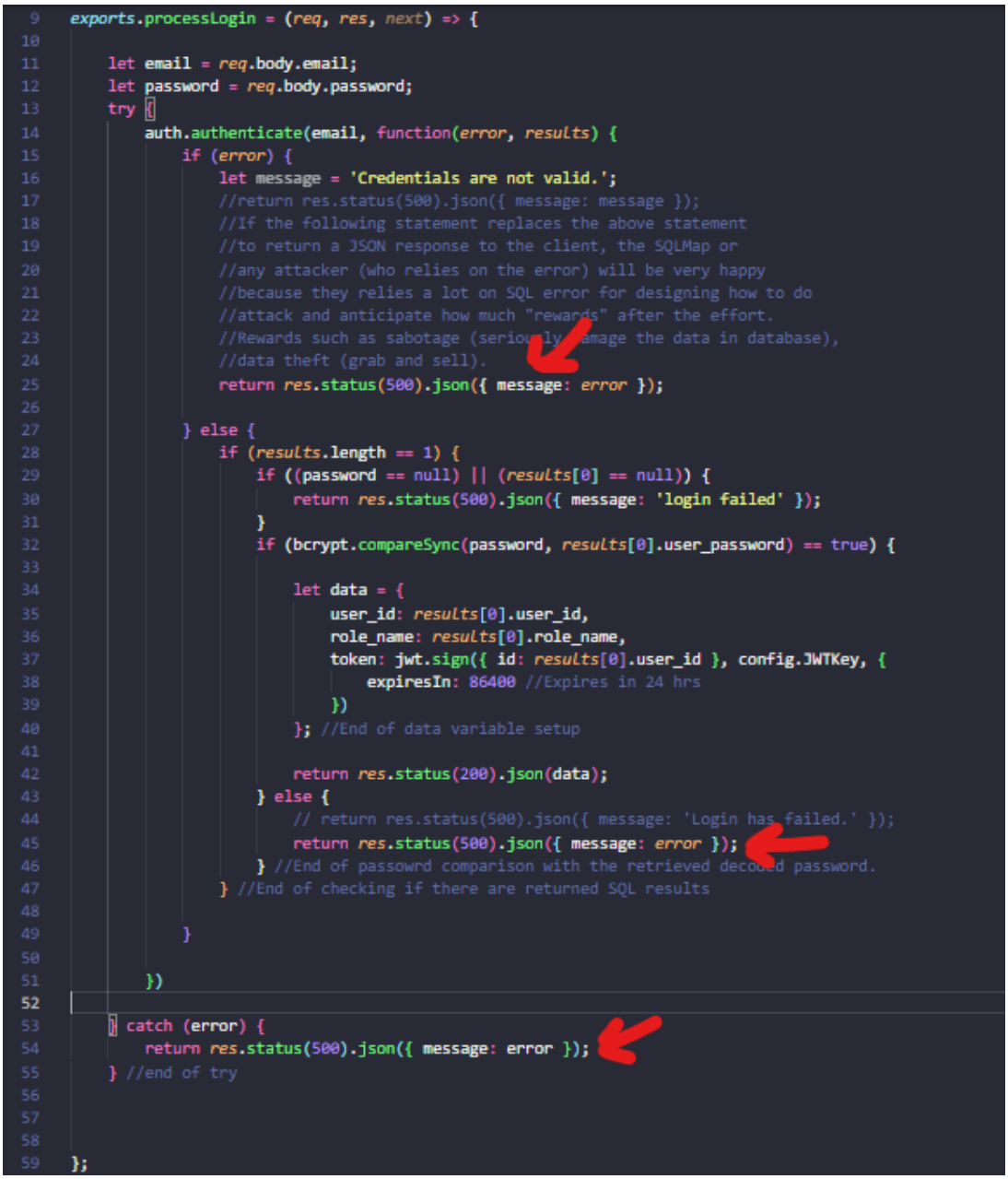
```
exports.processUpdateOneUser = async (req, res, next) => {
  console.log("processUpdateOneUser running");
  //Collect data from the request body
  let recordId = req.body.recordId;
  let newRoleId = req.body.roleId;
  try {
    results = await userManager.updateUser(recordId, newRoleId);
    console.log(results);
    return res.status(200).json({ message: "Completed update" });
  } catch (error) {
    console.log(
      "processUpdateOneUser method : catch block section code is running"
    );
    console.log(
      error,
      "================================================================="
    );
    return res
      .status(500)
      .json({ message: "Unable to complete update operation" });
  }
}; //End of processUpdateOneUser
```

On the other hand, in Figure E the JSON Response uses statusMesage while in Figure F it uses message instead of statusMessage which can cause some confusion for future developers as shown below.

Figure E:

```
exports.processRegister = (req, res, next) => {
    console.log('processRegister running.');
    let fullName = req.body.fullName;
    let email = req.body.email;
    let password = req.body.password;


    bcrypt.hash(password, 10, async(err, hash) => {
        if (err) {
            console.log('Error on hashing password');
            return res.status(500).json({ statusMessage: 'Unable to complete registration' });
        } else {

                results = user.createUser(fullName, email, hash, function(results, error){
                  if (results!=null){
                    console.log(results);
                    return res.status(200).json({ statusMessage: 'Completed registration.' });
                  }
                  if (error) {
                    console.log('processRegister method : callback error block section is running.');
                    console.log(error, '============================================================');
                    return res.status(500).json({ statusMessage: 'Unable to complete registration' });
                  }
                  });//End of anonymous callback function


        }
    });


}; // End of processRegister
```

Figure F:



```
exports.processLogin = (req, res, next) => {

    let email = req.body.email;
    let password = req.body.password;
    try {
        auth.authenticate(email, function(error, results) {
            if (error) {
                let message = 'Credentials are not valid.';
                //return res.status(500).json({ message: message });
                //If the following statement replaces the above statement
                //to return a JSON response to the client, the SQLMap or
                //any attacker (who relies on the error) will be very happy
                //because they relies a lot on SQL error for designing how to do
                //attack and anticipate how much "rewards" after the effort.
                //Rewards such as sabotage (seriously damage the data in database),
                //data theft (grab and sell).
                return res.status(500).json({ message: error });

            } else {
                if (results.length == 1) {
                    if ((password == null) || (results[0] == null)) {
                        return res.status(500).json({ message: 'login failed' });
                    }
                    if (bcrypt.compareSync(password, results[0].user_password) == true) {

                        let data = {
                            user_id: results[0].user_id,
                            role_name: results[0].role_name,
                            token: jwt.sign({ id: results[0].user_id }, config.JWTKey, {
                                expiresIn: 86400 //Expires in 24 hrs
                            })
                        }; //End of data variable setup

                        return res.status(200).json(data);
                    } else {
                        // return res.status(500).json({ message: 'Login has failed.' });
                        return res.status(500).json({ message: error });
                    } //End of passowrd comparison with the retrieved decoded password.
                } //End of checking if there are returned SQL results

            }

        })

    } catch (error) {
        return res.status(500).json({ message: error });
    } //end of try


};
```

So, to standardize the JSON Response, we will be using the following format for all the JSON Response in userController.js file as shown below.

```
} catch (error) {
  let message = "Server is unable to process your request.";
  return res.status(500).json({
    error: error,
    message: message,
  });
}
```

Summary of bonus parts including those not shown above and rectifications that are not shown in the above 6 vulnerabilities

1. Done Input Validation for all input fields like register new user, search inputs, email invitation, design file type submission etc.
2. Done Output Sanitization for User Data E.g., Full name, Email and Role name displayed in Admin Page and User Page.
3. To prevent easy password guessing against broken authentication I did a weak password check where the password must Contain at least 1 upper case, 1 lower case, 1 number, 1 special character with at least 7 characters.
4. Changed the cloudinary files in database to https instead of http by changing the *seeddata.js* file and the cloudinary upload image (uploadFile Function) in *fileService.js* by changing result.url to result.secure_url to remove the https warnings for future and current submissions.
5. Used Bearer Token for all and not using user Id and role for user access control and authentication.
6. Used Session-Storage instead of Local-Storage to delete token after user has closed the tab/browser instead of continuing to store it.
7. Used express-rate-limit to limit the number of attempts for register page, login page, email invitation page and design submission page to prevent brute forcing.

8. Created a check role from token to make sure that only user and admin can access their own sites without being able to access a web page that they don't have access to through the URL Link.
9. Used Try and Catch for all functions in authController.js and userController.js to handle all errors at the end to prevent callback hell.
10. Standardized fileService.js, authService.js and userService.js to use Promise and declare SQL statement before the Promise.
11. Used Standardized JSON Response for authController.js and userController.js.

References

- https://www.liquidweb.com/kb/how-to-install-chocolatey-on-windows/
- https://technixleo.com/create-locally-trusted-ssl-certificates-with-mkcert-on-windows/
- https://blog.logrocket.com/rate-limiting-node-js/