IT DEPARTMENT TEAM

Vulnerability Report

By Acosta Jan Jacob Domalanta (P2026271) and
Tan Yong Rui (P2004147)

Table of Contents

Executive Summary

In today’s world of advanced technology, competitions no longer use pen and paper for recording of contestant’s information and personal details, nor do they use postage for submission of works any longer. Nowadays, contestant’s information is stored digitally in a database, and works are uploaded onto the cloud for easy access.

However, the number of attackers have only been increasing alongside the advancement of technology. The IT Department Team has been tasked with performing vulnerability tests on the Bee Design Award Competition system and fixing those vulnerabilities to prevent theft of information and sabotaging of the competition. Further details of the vulnerabilities discovered can be found further down in the document.

1. Introduction

1.1 Overview

This report contains the findings that the IT Department Team found while conducting vulnerability tests on the Bee Design Award Competition system. The IT Department Team was called upon to review the code of the said system and perform vulnerability tests to find any vulnerabilities in the system.

1.2 Scope of Project

The IT Department Team conducted vulnerability tests Bee Design Award Competition system.

Below are the vulnerability categories that the IT Department Team have been asked to address:

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

1.3 Project Team Members

The IT Department Team consists of the following members:

Name	Role
Acosta Jan Jacob Domalanta	IT Department Team Member
Tan Yong Rui	IT Department Team Member

2. Key Findings

2.1 Introduction

This section highlights the issues that were identified while reviewing the system's codes and a rating from Low to High has been given for each key issue found. More detailed findings can be found in section 3 and we highly recommend reviewing each of the issues listed there and implement the solutions suggested where applicable.

2.2 Key Issues

The team has identified a number of vulnerabilities that may allow users to have elevated privileges and execute and access administrative functions. Some vulnerabilities even have the risk of allowing attackers to hang the system, or even access the database illegally to make unauthorised changes.

The following table outlines the vulnerabilities that are easily exploitable or contribute to a high level of risk:

Key Issue	Area of fault	Impact
The majority of the SQL statements found in the files were either using template literals (${}) or appending the value of the user's input to the SQL statement. This makes the system vulnerable to SQL injection, where malicious commands are keyed into the input field to view potentially sensitive information or even edit the database.	Design of the Application	High
Administrative functions such as viewing of users, viewing user's design submissions and editing of user roles do not require any form of authentication to run, which is a huge flaw and may lead to unfair advantages in the competition.	Authorization	High
Defence-in-depth is the theory of placing multiple layers of security controls (defence) throughout the application. The intent is that even if one security control fails or is exploited, it'll have the other security controls to fall back on. Currently, the application doesn't have many 'layered' defences. We plan to change that with our changes.	Design of the Application	Medium

3. Detailed Findings

3.1 SQL Injections

- SQL injections(SQLi) is an attack that convinces the server to run Structured Query Language (SQL) on the database that was not **intended**.
- Attackers achieve this by sending SQL code, instead of actual values, to inputs that usually need the user's input.
- A **layman** example is:

  SELECT user WHOSE name is X;
  Replace X with the user's first name.

  Normally, the user will put his name in X and the database will execute the query flawlessly. But what attackers can do is put code in X instead of a name.

  SELECT user WHOSE name is John AND DELETE ALL users;

  In the example above, the attacker put - John AND DELETE ALL users - in the field instead of his actual name. Clearly, this is dangerous as the attacker can execute code that he isn't supposed to have access to.

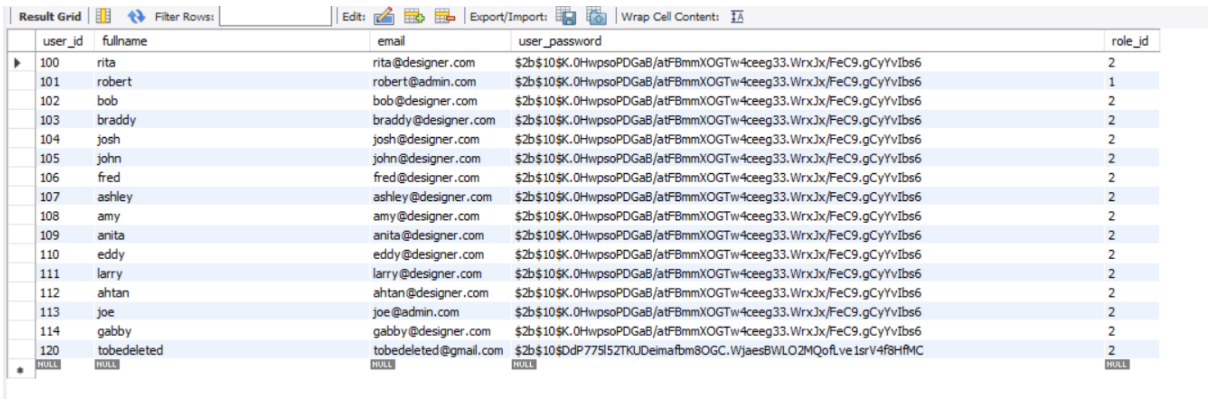
3.1.1 Login SQL Injection

- An attacker can execute SQL Injections in the email input field of the Login page.

  Impact: High
  The IT Department Team has decided on a rating of High as the Login page is susceptible to SQL Injection. If used right, it can be used to delete the entire database of users, files and roles, which would be a catastrophe.

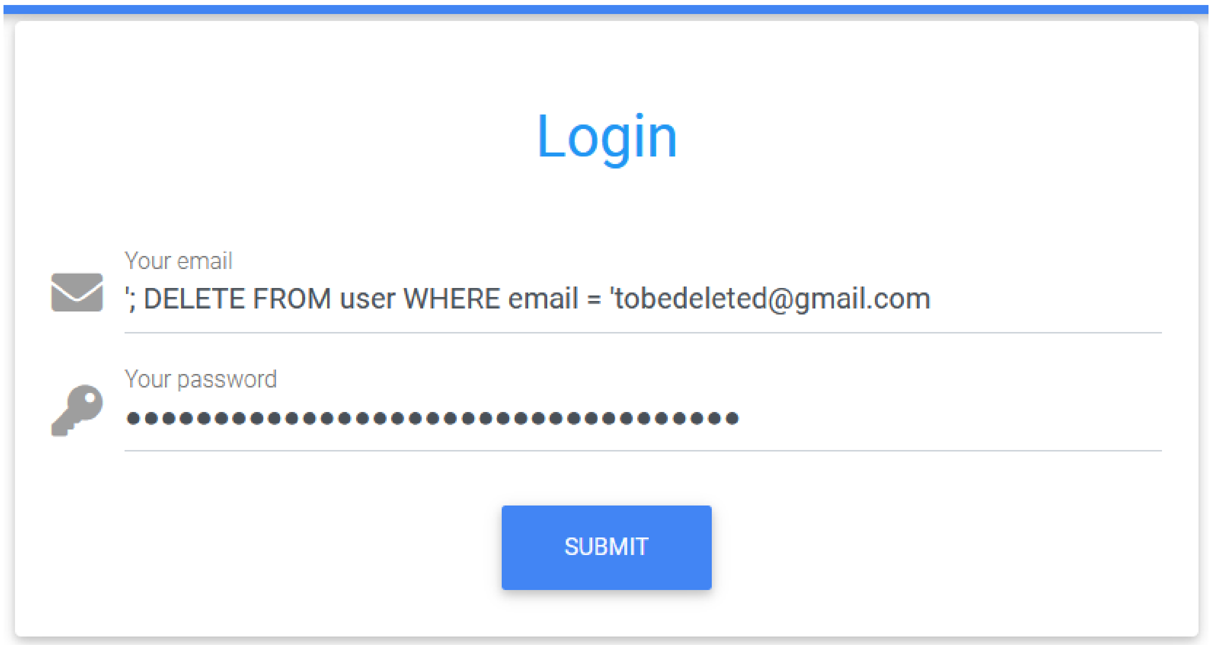
How to exploit flaw:

As seen in the image below, we have a test user named tobedeleted with the email “tobedeleted@gmail.com”.



user_id	fullname	email	user_password	role_id
100	rita	rita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
101	robert	robert@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	1
102	bob	bob@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
103	braddy	braddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
104	josh	josh@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
105	john	john@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
106	fred	fred@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
107	ashley	ashley@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
108	amy	amy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
109	anita	anita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
110	eddy	eddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
111	larry	larry@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
112	ahtan	ahtan@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
113	joe	joe@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
114	gabby	gabby@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
120	tobedeleted	tobedeleted@gmail.com	$2b$10$DdP775l52TKUDeimafbm8OGC.WjaesBWLO2MQofLve1srV4f8HfMC	2
NULL	NULL	NULL	NULL	NULL

Go to the login page and type as shown in the image. The password can be anything. Click on the Submit button.

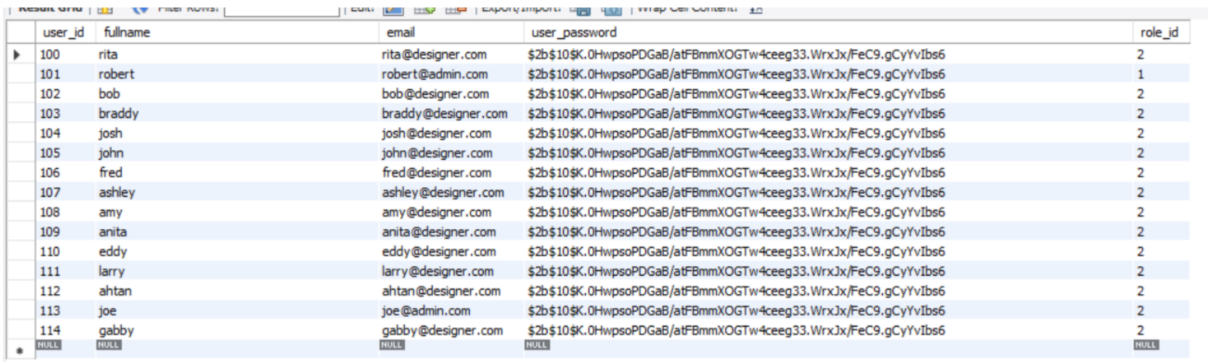


Login

Your email
'; DELETE FROM user WHERE email = 'tobedeleted@gmail.com

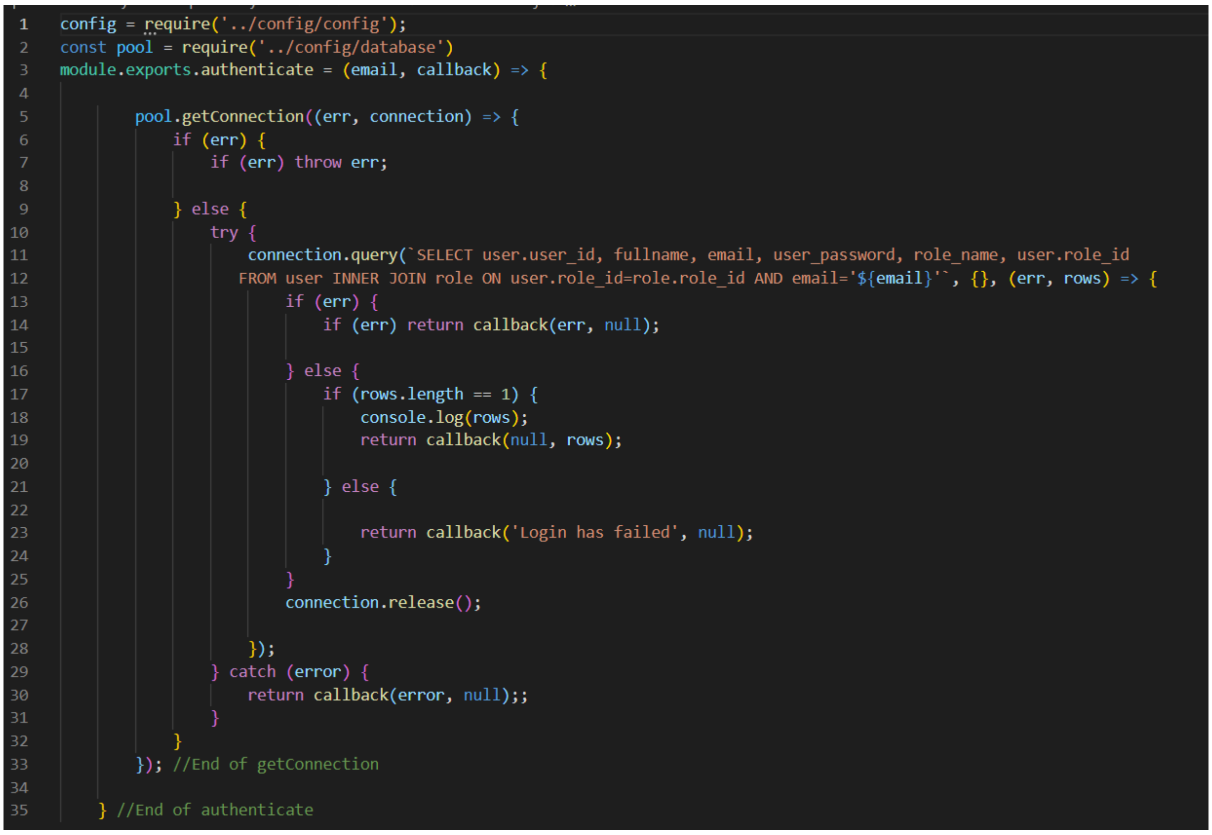
Your password
●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●●

The user with the email ‘tobedeleted@gmail.com’ is now gone.



user_id	fullname	email	user_password	role_id
100	rita	rita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
101	robert	robert@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	1
102	bob	bob@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
103	braddy	braddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
104	josh	josh@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
105	john	john@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
106	fred	fred@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
107	ashley	ashley@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
108	amy	amy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
109	anita	anita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
110	eddy	eddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
111	larry	larry@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
112	ahtan	ahtan@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
113	joe	joe@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
114	gabby	gabby@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
NULL	NULL	NULL	NULL	NULL

Where the flaw is found:



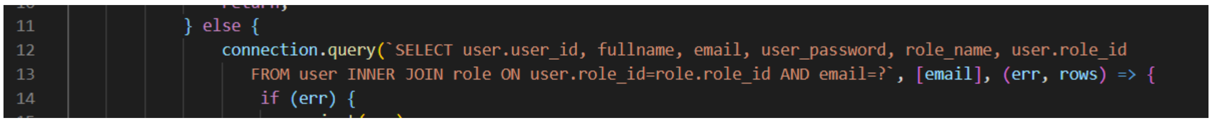
```
1   config = require('../config/config');
2   const pool = require('../config/database')
3   module.exports.authenticate = (email, callback) => {
4
5           pool.getConnection((err, connection) => {
6               if (err) {
7                   if (err) throw err;
8
9               } else {
10                  try {
11                      connection.query(`SELECT user.user_id, fullname, email, user_password, role_name, user.role_id
12                     FROM user INNER JOIN role ON user.role_id=role.role_id AND email='${email}'`, {}, (err, rows) => {
13                          if (err) {
14                              if (err) return callback(err, null);
15
16                          } else {
17                              if (rows.length == 1) {
18                                  console.log(rows);
19                                  return callback(null, rows);
20
21                              } else {
22
23                                  return callback('Login has failed', null);
24                              }
25                          }
26                          connection.release();
27
28                      });
29                  } catch (error) {
30                      return callback(error, null);;
31                  }
32              }
33          }); //End of getConnection
34
35      } //End of authenticate
```

Found in /src/services/authService.js in the original code.

The flaw is located on line 12, where the SQL statement isn't using Parameterized Queries (?) and instead is using a Template Literal (${email}). This makes the system susceptible to SQL Injection. Further explanation to parameterized queries will be explained in the recommended actions to be taken below.

Recommendations:

1. Use Parameterized Queries in place of Template Literals as shown below:

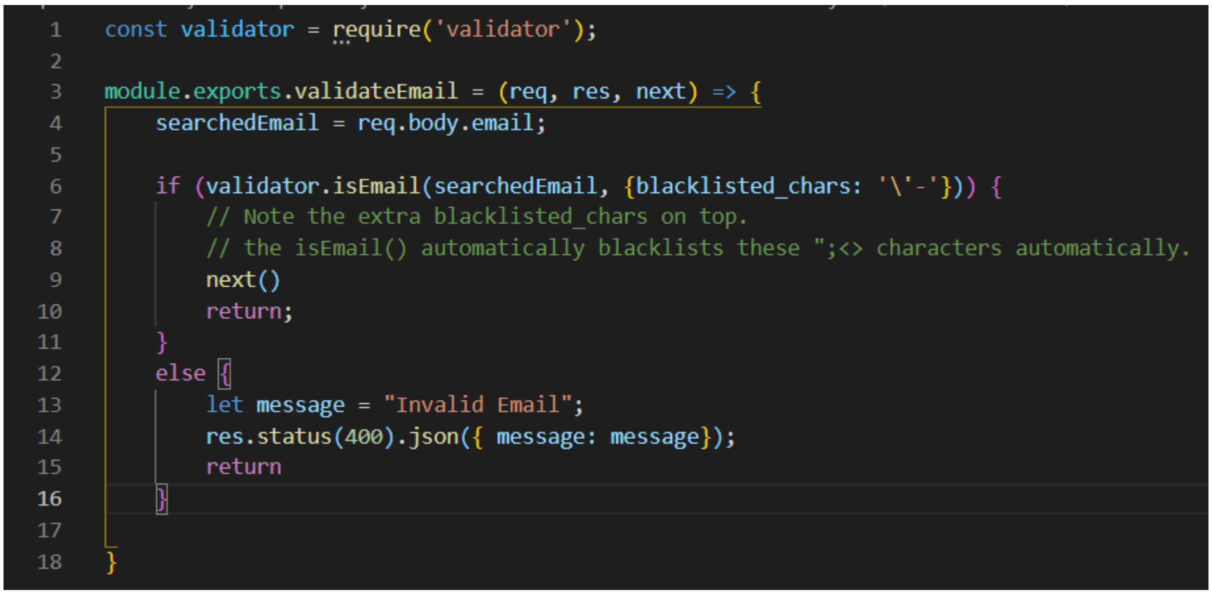


```
11              } else {
12                  connection.query(`SELECT user.user_id, fullname, email, user_password, role_name, user.role_id
13                     FROM user INNER JOIN role ON user.role_id=role.role_id AND email=?`, [email], (err, rows) => {
14                      if (err) {
```

Found in /src/services/authService.js in the edited code.

Parameterized Queries take in a value (in this case, the email from the input box on the login page) and will treat the value taken in as a value only. This disallows the email value obtained from the login page from altering the SQL Query we intend to do, which is to get a user's data based on his/her email.

2. Checking for illegal characters by making a middleware function to check for valid emails.

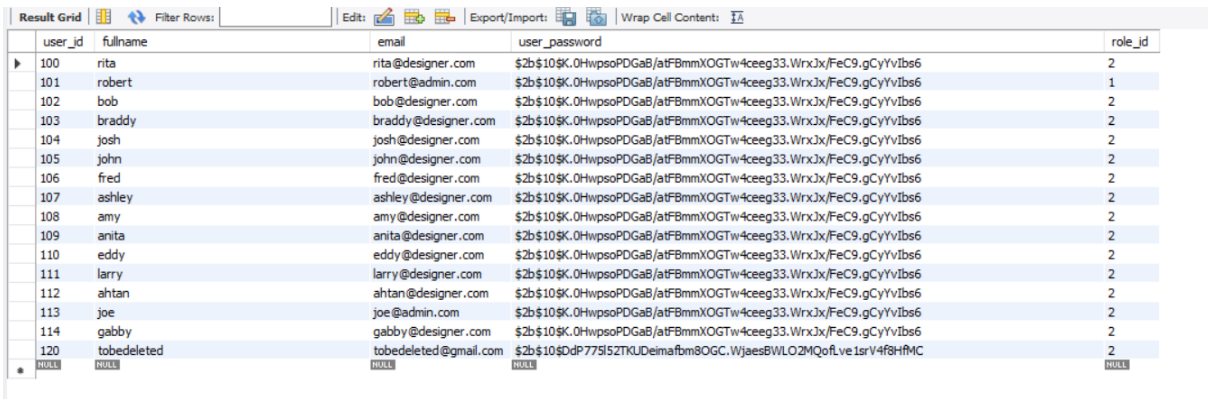


```javascript
const validator = require('validator');

module.exports.validateEmail = (req, res, next) => {
    searchedEmail = req.body.email;

    if (validator.isEmail(searchedEmail, {blacklisted_chars: '\'-'})) {
        // Note the extra blacklisted_chars on top.
        // the isEmail() automatically blacklists these ";<> characters automatically.
        next()
        return;
    }
    else {
        let message = "Invalid Email";
        res.status(400).json({ message: message});
        return
    }

}
```

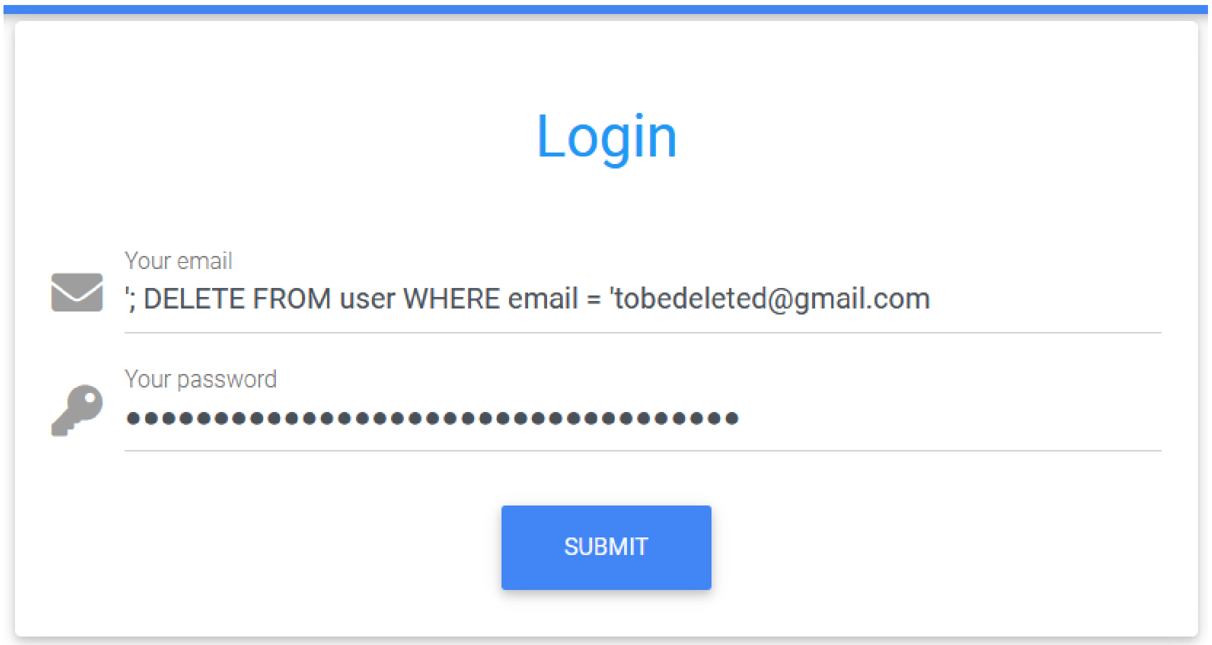
Found in /src/middlewares/checkValidEmail.js in the edited code.

This middleware function runs before running the SQL Query, and will only do so if the email is valid by running the input received from the login page. If the email is invalid, an error is returned and the SQL Query will not run. This is recommended as an added layer of protection against SQL Injection.

Now, if we try to delete the user with the email
“tobedeleted@gmail.com”...



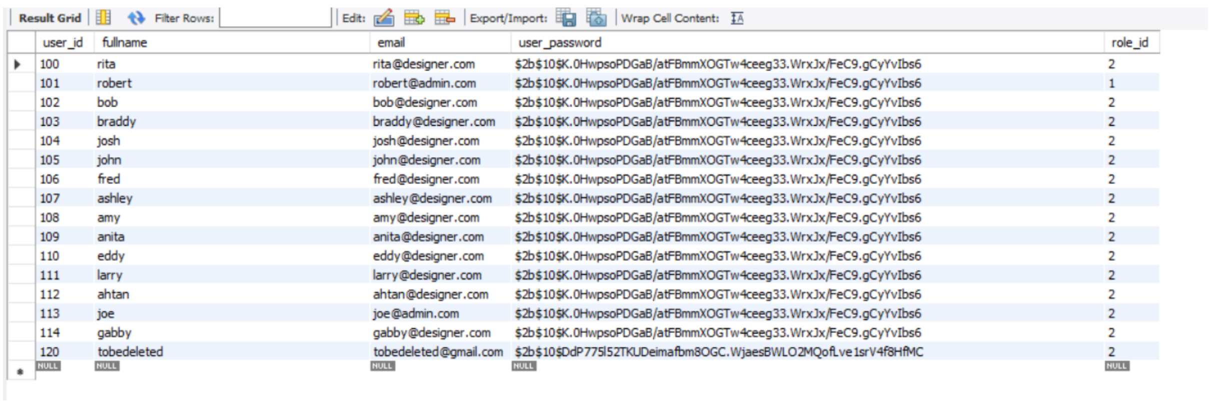
user_id	fullname	email	user_password	role_id
100	rita	rita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
101	robert	robert@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	1
102	bob	bob@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
103	braddy	braddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
104	josh	josh@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
105	john	john@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
106	fred	fred@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
107	ashley	ashley@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
108	amy	amy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
109	anita	anita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
110	eddy	eddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
111	larry	larry@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
112	ahtan	ahtan@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
113	joe	joe@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
114	gabby	gabby@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
120	tobedeleted	tobedeleted@gmail.com	$2b$10$DdP775l52TKUDeimafbm8OGC.WjaesBWLO2MQofLve1srV4f8HfMC	2
NULL	NULL	NULL	NULL	NULL



Login

Your email
'; DELETE FROM user WHERE email = 'tobedeleted@gmail.com

Your password
••••••••••••••••••••••••••••••••••



user_id	fullname	email	user_password	role_id
100	rita	rita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
101	robert	robert@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	1
102	bob	bob@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
103	braddy	braddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
104	josh	josh@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
105	john	john@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
106	fred	fred@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
107	ashley	ashley@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
108	amy	amy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
109	anita	anita@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
110	eddy	eddy@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
111	larry	larry@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
112	ahtan	ahtan@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
113	joe	joe@admin.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
114	gabby	gabby@designer.com	$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6	2
120	tobedeleted	tobedeleted@gmail.com	$2b$10$DdP775l52TKUDeimafbm8OGC.WjaesBWLO2MQofLve1srV4f8HfMC	2
NULL	NULL	NULL	NULL	NULL

The user cannot be deleted from the login page anymore.

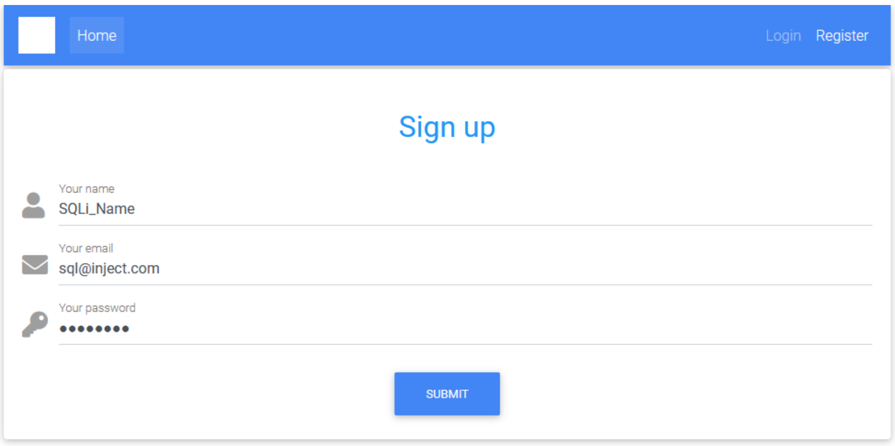
3.1.2 Check Submission SQL Injection

Impact: High

The IT Department Team has decided on a rating of High as the check submission page is susceptible to SQL Injection. If used right, it can be used to delete the entire database of users, files and roles, which would be a catastrophe.

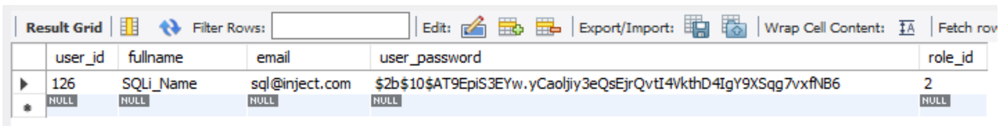
How to exploit flaw:

First, let's start by making a test user, SQLi_Name.



Your name: SQLi_Name
Your email: sql@inject.com
Your password: ••••••••

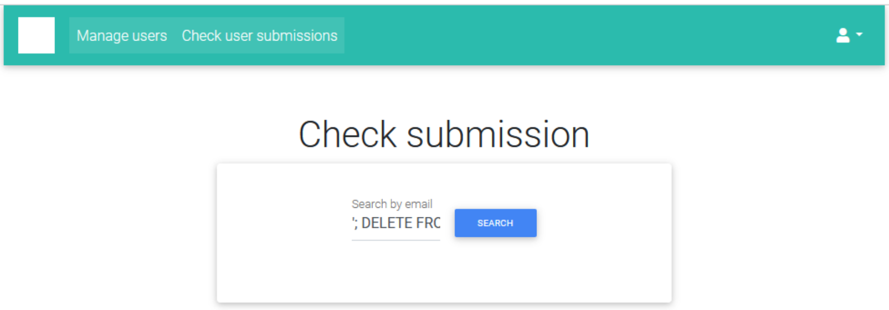
Below is the test user in the database.



user_id	fullname	email	user_password	role_id
126	SQLi_Name	sql@inject.com	$2b$10$AT9EpiS3EYw.yCaoljiy3eQsEjrQvtI4VkthD4IgY9XSqg7vxfNB6	2
NULL	NULL	NULL	NULL	NULL

Now let's go to the admin's check submissions page.
On this page, input the SQL code below.

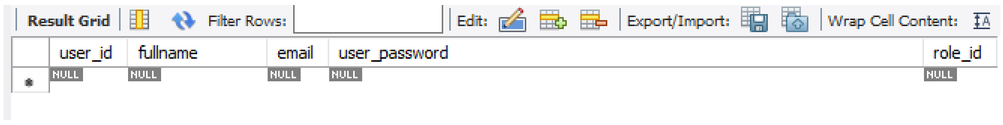
'; DELETE FROM user WHERE fullname = 'SQLi_Name



Search by email: '; DELETE FRC

(Continued Below)

After the SQL code is sent, the database row for the test user gets deleted in the database.

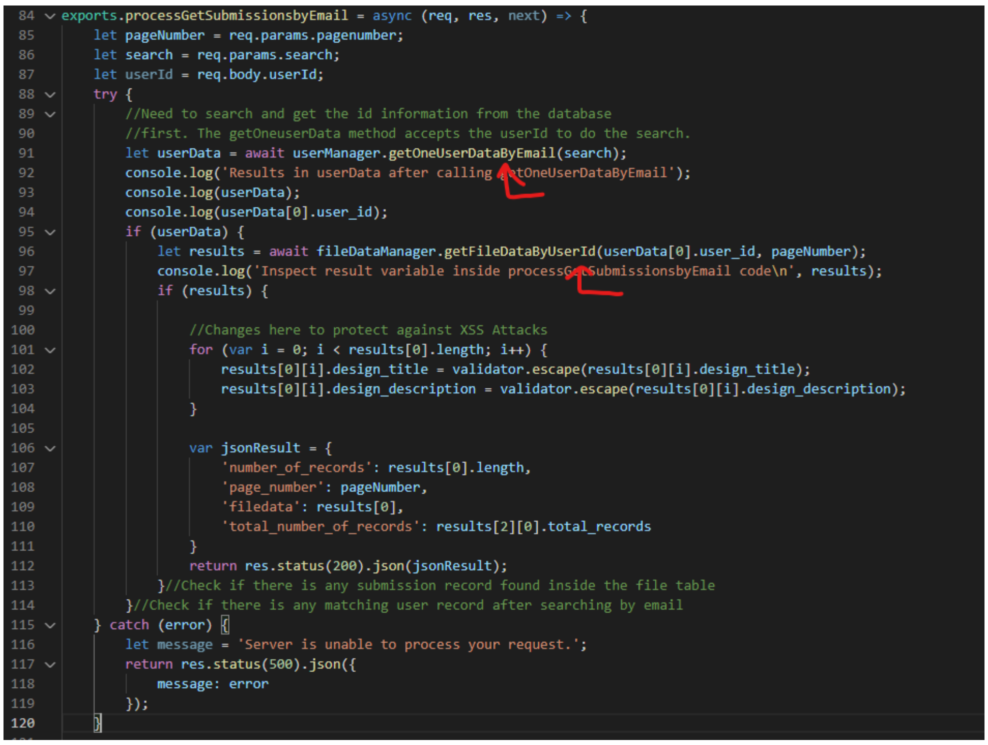


	user_id	fullname	email	user_password	role_id
*	NULL	NULL	NULL	NULL	NULL

As seen from above, admins who know how to use SQL, are now able to execute **anything** they want on the database.
From inserting new values to **deleting the whole database**, admins have unparalleled power to change anything.

Where flaw is found:

Presently, there aren't any checks to ensure that SQL code is intercepted. Or that only emails were inputted by the user.



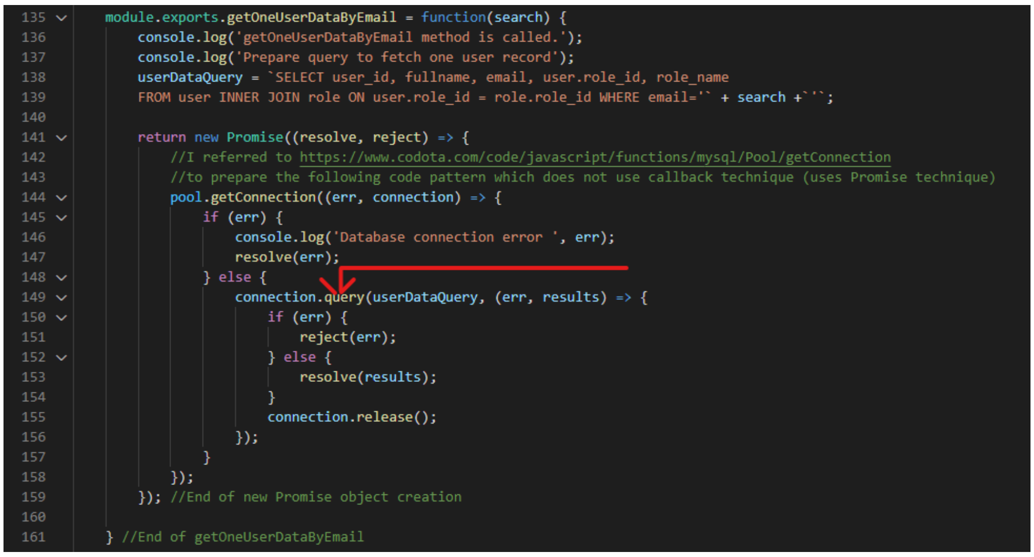
```
exports.processGetSubmissionsbyEmail = async (req, res, next) => {
    let pageNumber = req.params.pagenumber;
    let search = req.params.search;
    let userId = req.body.userId;
    try {
        //Need to search and get the id information from the database
        //first. The getOneuserData method accepts the userId to do the search.
        let userData = await userManager.getOneUserDataByEmail(search);
        console.log('Results in userData after calling getOneUserDataByEmail');
        console.log(userData);
        console.log(userData[0].user_id);
        if (userData) {
            let results = await fileDataManager.getFileDataByUserId(userData[0].user_id, pageNumber);
            console.log('Inspect result variable inside processGetSubmissionsbyEmail code\n', results);
            if (results) {

                //Changes here to protect against XSS Attacks
                for (var i = 0; i < results[0].length; i++) {
                    results[0][i].design_title = validator.escape(results[0][i].design_title);
                    results[0][i].design_description = validator.escape(results[0][i].design_description);
                }

                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);
            }//Check if there is any submission record found inside the file table
        }//Check if there is any matching user record after searching by email
    } catch (error) {
        let message = 'Server is unable to process your request.';
        return res.status(500).json({
            message: error
        });
    }
}
```

Found in controllers/userController.js in the original version.

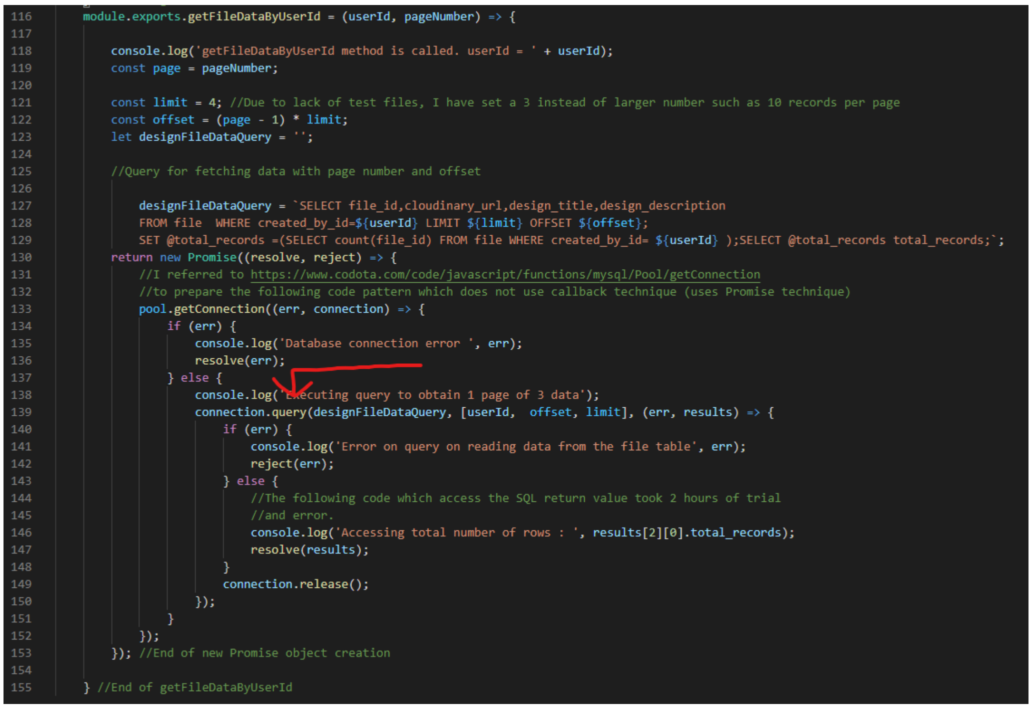
(Continued Below)

The connection.query() is the function that interacts with the SQL database.

Checks also absent inside the individual function definitions.



```
module.exports.getOneUserDataByEmail = function(search) {
    console.log('getOneUserDataByEmail method is called.');
    console.log('Prepare query to fetch one user record');
    userDataQuery = `SELECT user_id, fullname, email, user.role_id, role_name
    FROM user INNER JOIN role ON user.role_id = role.role_id WHERE email='` + search +`'`;

    return new Promise((resolve, reject) => {
        //I referred to https://www.codota.com/code/javascript/functions/mysql/Pool/getConnection
        //to prepare the following code pattern which does not use callback technique (uses Promise technique)
        pool.getConnection((err, connection) => {
            if (err) {
                console.log('Database connection error ', err);
                resolve(err);
            } else {
                connection.query(userDataQuery, (err, results) => {
                    if (err) {
                        reject(err);
                    } else {
                        resolve(results);
                    }
                    connection.release();
                });
            }
        });
    }); //End of new Promise object creation

} //End of getOneUserDataByEmail
```



```
module.exports.getFileDataByUserId = (userId, pageNumber) => {

    console.log('getFileDataByUserId method is called. userId = ' + userId);
    const page = pageNumber;

    const limit = 4; //Due to lack of test files, I have set a 3 instead of larger number such as 10 records per page
    const offset = (page - 1) * limit;
    let designFileDataQuery = '';

    //Query for fetching data with page number and offset

        designFileDataQuery = `SELECT file_id,cloudinary_url,design_title,design_description
        FROM file  WHERE created_by_id=${userId} LIMIT ${limit} OFFSET ${offset};
        SET @total_records =(SELECT count(file_id) FROM file WHERE created_by_id= ${userId} );SELECT @total_records total_records;`;
    return new Promise((resolve, reject) => {
        //I referred to https://www.codota.com/code/javascript/functions/mysql/Pool/getConnection
        //to prepare the following code pattern which does not use callback technique (uses Promise technique)
        pool.getConnection((err, connection) => {
            if (err) {
                console.log('Database connection error ', err);
                resolve(err);
            } else {
                console.log('Executing query to obtain 1 page of 3 data');
                connection.query(designFileDataQuery, [userId,  offset, limit], (err, results) => {
                    if (err) {
                        console.log('Error on query on reading data from the file table', err);
                        reject(err);
                    } else {
                        //The following code which access the SQL return value took 2 hours of trial
                        //and error.
                        console.log('Accessing total number of rows : ', results[2][0].total_records);
                        resolve(results);
                    }
                    connection.release();
                });
            }
        });
    }); //End of new Promise object creation

} //End of getFileDataByUserId
```

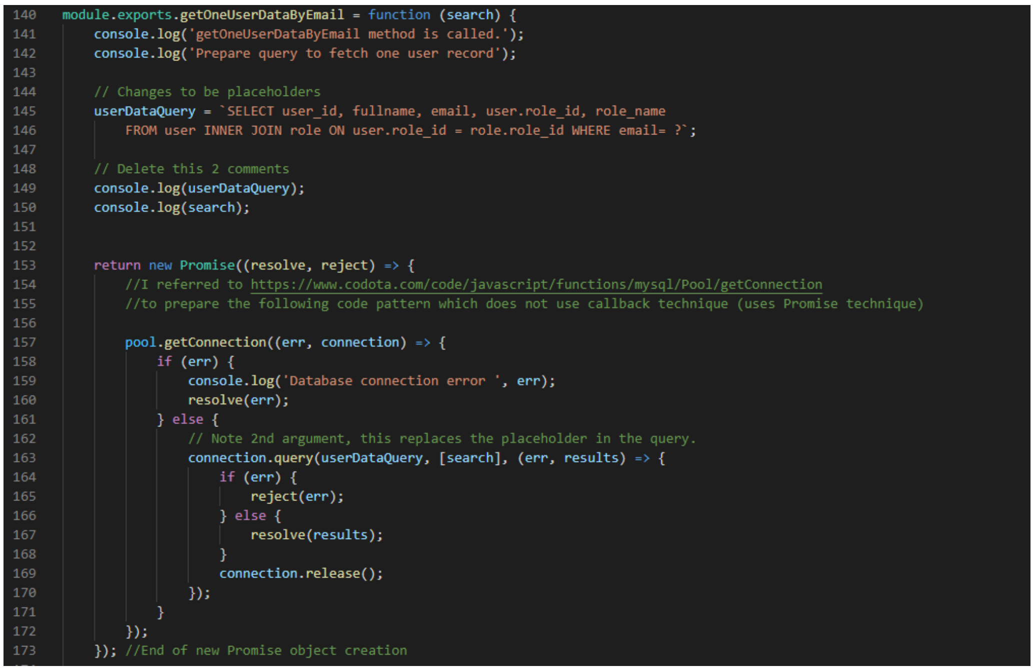
Found in services/userService.js and services/fileService.js respectively.

(Continued Below)

Recommendation:

Use placeholders for all MySQL queries. By using placeholders, the server escapes the characters and thus the server takes the values ‘literally’.

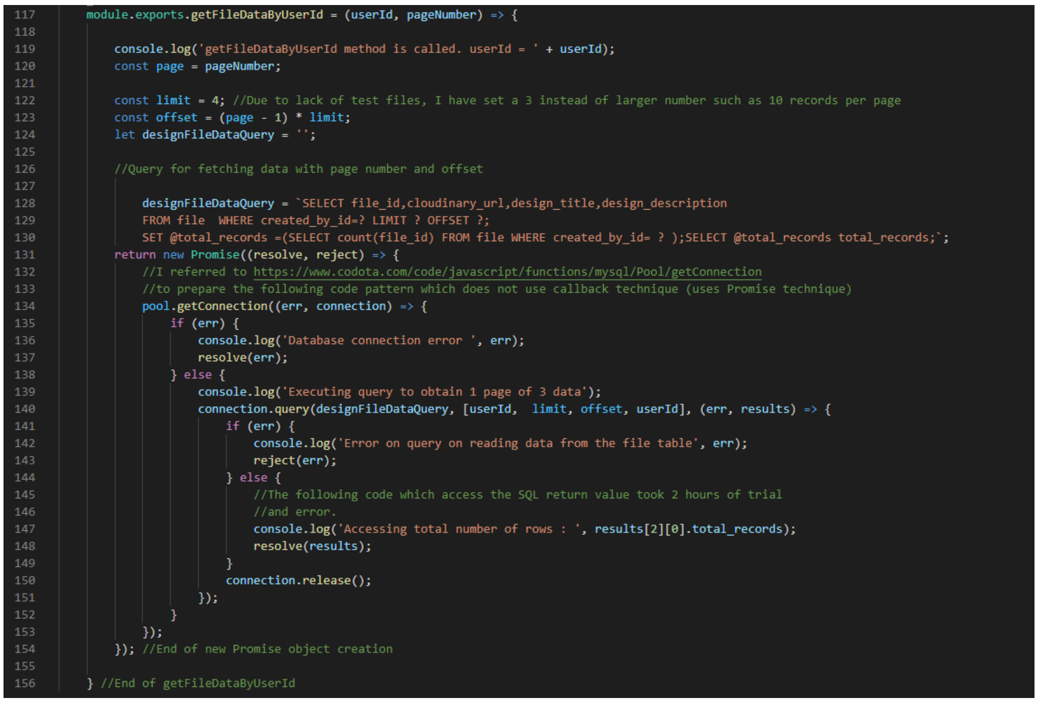
Changes in line 146 and 163.



```
140 module.exports.getOneUserDataByEmail = function (search) {
141     console.log('getOneUserDataByEmail method is called.');
142     console.log('Prepare query to fetch one user record');
143 
144     // Changes to be placeholders
145     userDataQuery = `SELECT user_id, fullname, email, user.role_id, role_name
146         FROM user INNER JOIN role ON user.role_id = role.role_id WHERE email= ?`;
147 
148     // Delete this 2 comments
149     console.log(userDataQuery);
150     console.log(search);
151 
152 
153     return new Promise((resolve, reject) => {
154         //I referred to https://www.codota.com/code/javascript/functions/mysql/Pool/getConnection
155         //to prepare the following code pattern which does not use callback technique (uses Promise technique)
156 
157         pool.getConnection((err, connection) => {
158             if (err) {
159                 console.log('Database connection error ', err);
160                 resolve(err);
161             } else {
162                 // Note 2nd argument, this replaces the placeholder in the query.
163                 connection.query(userDataQuery, [search], (err, results) => {
164                     if (err) {
165                         reject(err);
166                     } else {
167                         resolve(results);
168                     }
169                     connection.release();
170                 });
171             }
172         });
173     }); //End of new Promise object creation
```

Found in controllers/userController.js in the edited version.

Changes in line 126-129 and line 140.



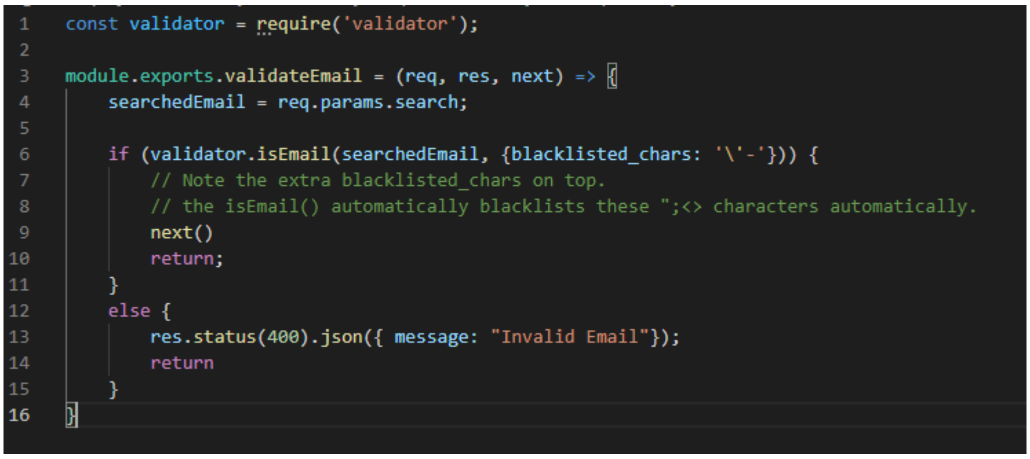
```
117  module.exports.getFileDataByUserId = (userId, pageNumber) => {
118
119      console.log('getFileDataByUserId method is called. userId = ' + userId);
120      const page = pageNumber;
121
122      const limit = 4; //Due to lack of test files, I have set a 3 instead of larger number such as 10 records per page
123      const offset = (page - 1) * limit;
124      let designFileDataQuery = '';
125
126      //Query for fetching data with page number and offset
127
128          designFileDataQuery = `SELECT file_id,cloudinary_url,design_title,design_description
129          FROM file  WHERE created_by_id=? LIMIT ? OFFSET ?;
130          SET @total_records =(SELECT count(file_id) FROM file WHERE created_by_id= ? );SELECT @total_records total_records;`;
131      return new Promise((resolve, reject) => {
132          //I referred to https://www.codota.com/code/javascript/functions/mysql/Pool/getConnection
133          //to prepare the following code pattern which does not use callback technique (uses Promise technique)
134          pool.getConnection((err, connection) => {
135              if (err) {
136                  console.log('Database connection error ', err);
137                  resolve(err);
138              } else {
139                  console.log('Executing query to obtain 1 page of 3 data');
140                  connection.query(designFileDataQuery, [userId,  limit, offset, userId], (err, results) => {
141                      if (err) {
142                          console.log('Error on query on reading data from the file table', err);
143                          reject(err);
144                      } else {
145                          //The following code which access the SQL return value took 2 hours of trial
146                          //and error.
147                          console.log('Accessing total number of rows : ', results[2][0].total_records);
148                          resolve(results);
149                      }
150                      connection.release();
151                  });
152              }
153          });
154      }); //End of new Promise object creation
155
156  } //End of getFileDataByUserId
```

Found in services/fileService.js in the edited version.

As another precaution (theory of defence-in-depth), another thing that can be done is to check whether the user's input contains any 'special' characters which might interfere with the SQL query.

Characters like **' " ; < > / \ -** should never be allowed to interact with the SQL query as they are the characters that can interfere with the SQL query.

We can use an external validator to check whether the email input for check submission is an email. Or whether it contains any of the illegal characters above.



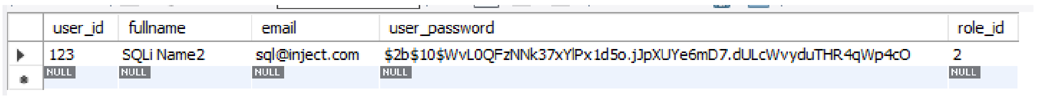
```
const validator = require('validator');

module.exports.validateEmail = (req, res, next) => {
    searchedEmail = req.params.search;

    if (validator.isEmail(searchedEmail, {blacklisted_chars: '\'-'})) {
        // Note the extra blacklisted_chars on top.
        // the isEmail() automatically blacklists these ";<> characters automatically.
        next()
        return;
    }
    else {
        res.status(400).json({ message: "Invalid Email"});
        return
    }
}
```

Note: All our middlewares are reusable, please look at Section 4.3

We don’t usually suggest using an external module, as it is always best to make a middleware function that is catered to what the application needs. But validator is a trustworthy module that garners 24 million uploads every month.

Results:

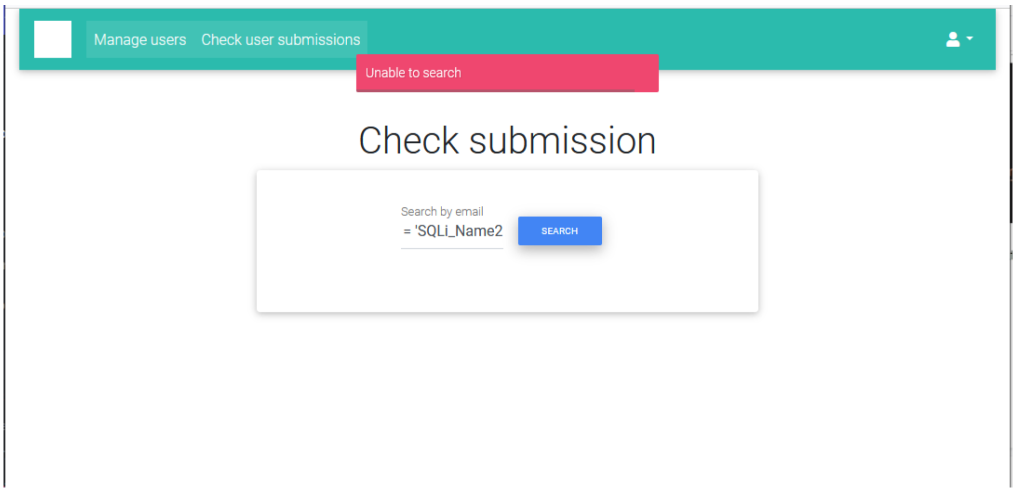
To showcase that the code is working, we’ve created a new user named ‘SQLi Name2’



user_id	fullname	email	user_password	role_id
123	SQLi Name2	sql@inject.com	$2b$10$WvL0QFzNNk37xYlPx1d5o.jJpXUYe6mD7.dULcWvyduTHR4qWp4cO	2
NULL	NULL	NULL	NULL	NULL

Now let’s go to the admin’s check submissions page again.
On this page, input the SQL code below.

'; DELETE FROM user WHERE fullname = 'SQLi_Name2

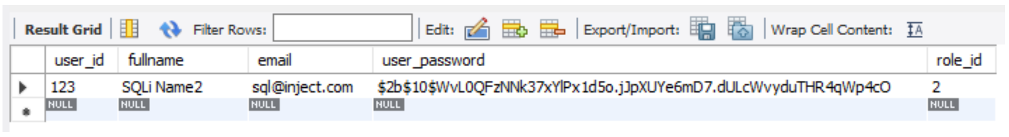


Unable to search

Check submission

Search by email
= 'SQLi_Name2

And we can verify that the new user is still there.



user_id	fullname	email	user_password	role_id
123	SQLi Name2	sql@inject.com	$2b$10$WvL0QFzNNk37xYlPx1d5o.jJpXUYe6mD7.dULcWvyduTHR4qWp4cO	2
NULL	NULL	NULL	NULL	NULL

3.2 Broken Authentication

- Broken Authentication is when a person's identity is not verified properly and gives unintended access to that person, which can be dangerous if given administrative access. An example would be a member of the public who can access a company's financial system as the system doesn't check for that person's identity properly.

3.2.1 Broken Authentication for the Public

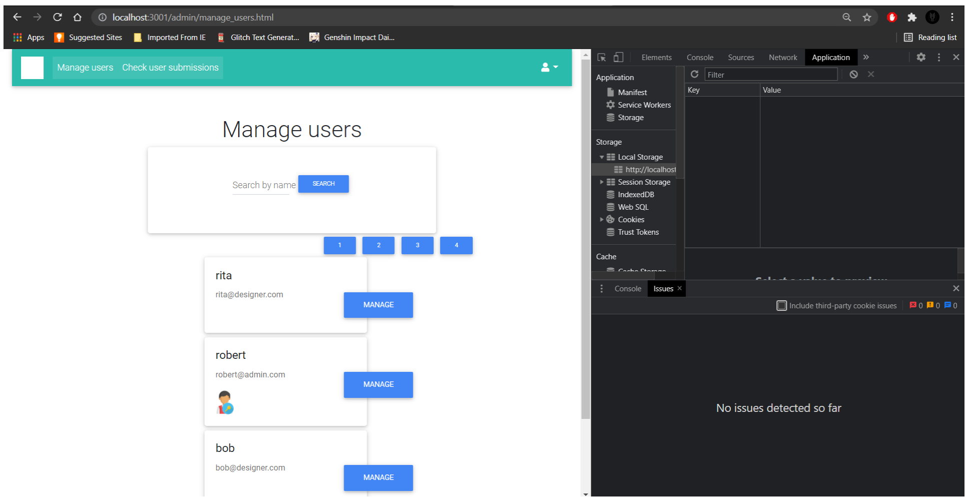
- Both user pages and administrator pages can be accessed without logging into the system.

Impact: High

The IT Department has given this vulnerability an impact rating of High as it is easy to exploit and when exploited can be used to affect the competition on a large scale. The attacker has free access to all of the sites, access to the majority of the functions (such as getting the list of users) as well as the same authority as the intended user (User or Admin, depending on the page) without having to log in.

How to exploit flaw:

Enter the user or admin website into the search bar and hit enter to load the page. The page should load even though the user is not logged in, and still be able to run admin functions, such as managing users as shown below.



Manage users

rita
rita@designer.com

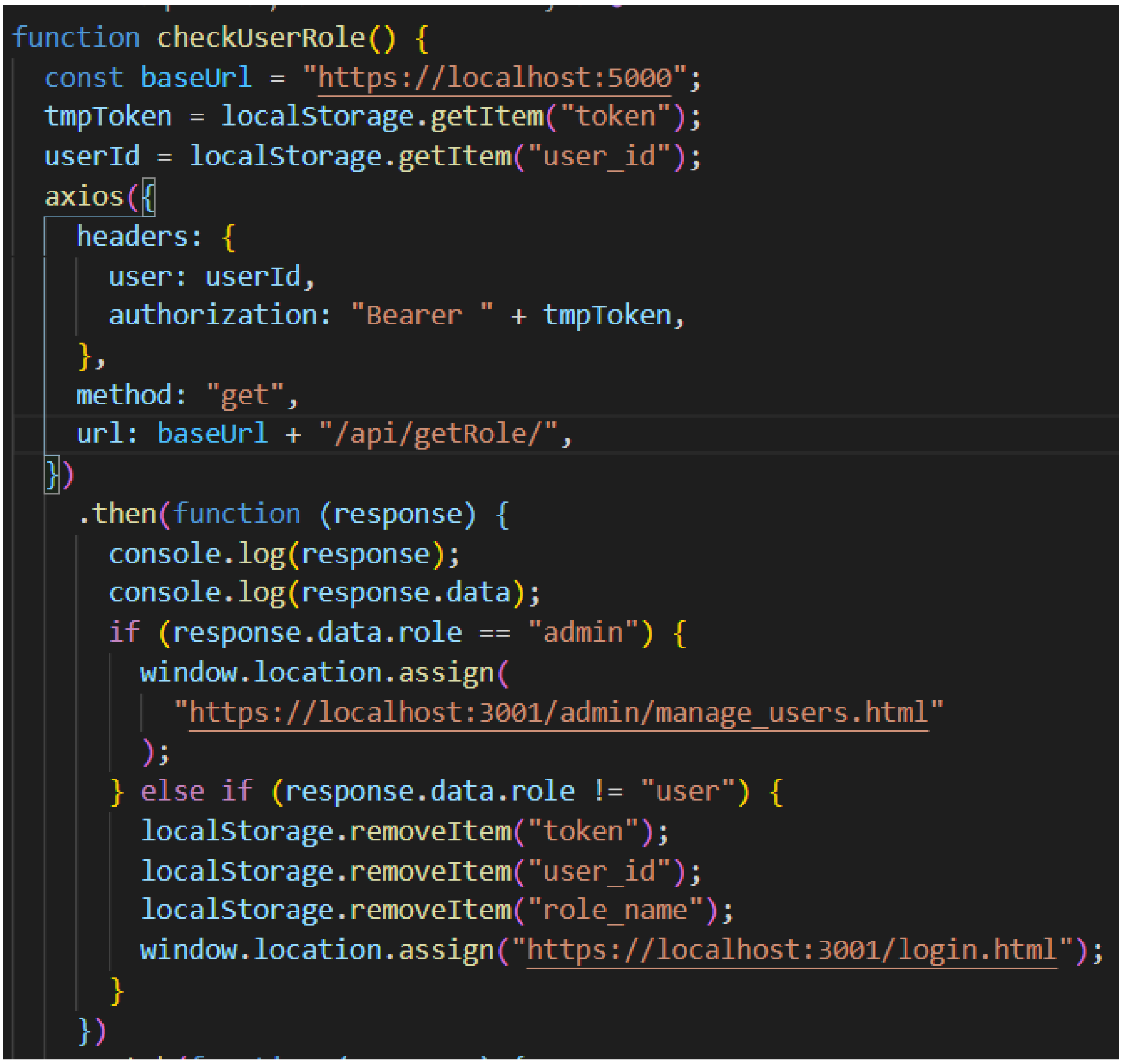
robert
robert@admin.com

bob
bob@designer.com

No issues detected so far

Where the flaw is found:
The flaw is due to a lack of code for verification of the user's identity rather than a flaw in the existing code.

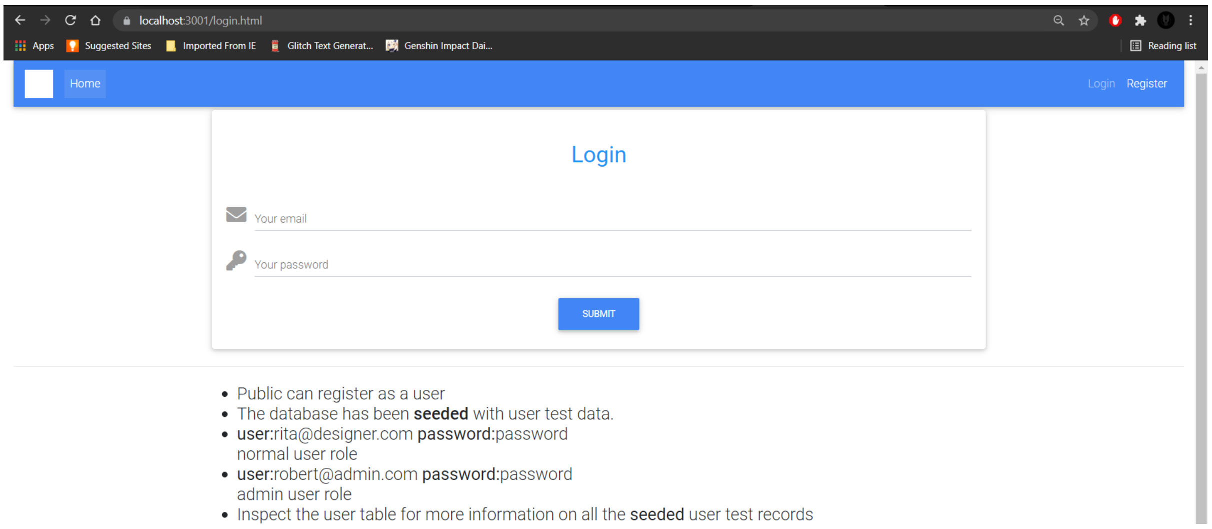
Recommendation:
When accessing any site intended for users or administrators, check for the user's token and role. If any of them are not present or the role does not match the website's intended user, they will be redirected automatically.



```
function checkUserRole() {
  const baseUrl = "https://localhost:5000";
  tmpToken = localStorage.getItem("token");
  userId = localStorage.getItem("user_id");
  axios({
    headers: {
      user: userId,
      authorization: "Bearer " + tmpToken,
    },
    method: "get",
    url: baseUrl + "/api/getRole/",
  })
    .then(function (response) {
      console.log(response);
      console.log(response.data);
      if (response.data.role == "admin") {
        window.location.assign(
          "https://localhost:3001/admin/manage_users.html"
        );
      } else if (response.data.role != "user") {
        localStorage.removeItem("token");
        localStorage.removeItem("user_id");
        localStorage.removeItem("role_name");
        window.location.assign("https://localhost:3001/login.html");
      }
    })
```

Found in /public/js/checkUserRole.js in the edited code.
Note: This Javascript code is reusable. See Section 4.4 for more information.

Now, if we try to exploit the broken authentication again…



- Public can register as a user
- The database has been **seeded** with user test data.
- **user:**rita@designer.com **password:**password
  normal user role
- **user:**robert@admin.com **password:**password
  admin user role
- Inspect the user table for more information on all the **seeded** user test records

The user is redirected to the login page now.

3.3 Sensitive Data Exposure

- Sensitive Data Exposure is when information that is sensitive gets exposed. A real-life example would be your email and password being leaked onto the internet.

3.3.1 Sensitive Data Exposure JSON Web Token

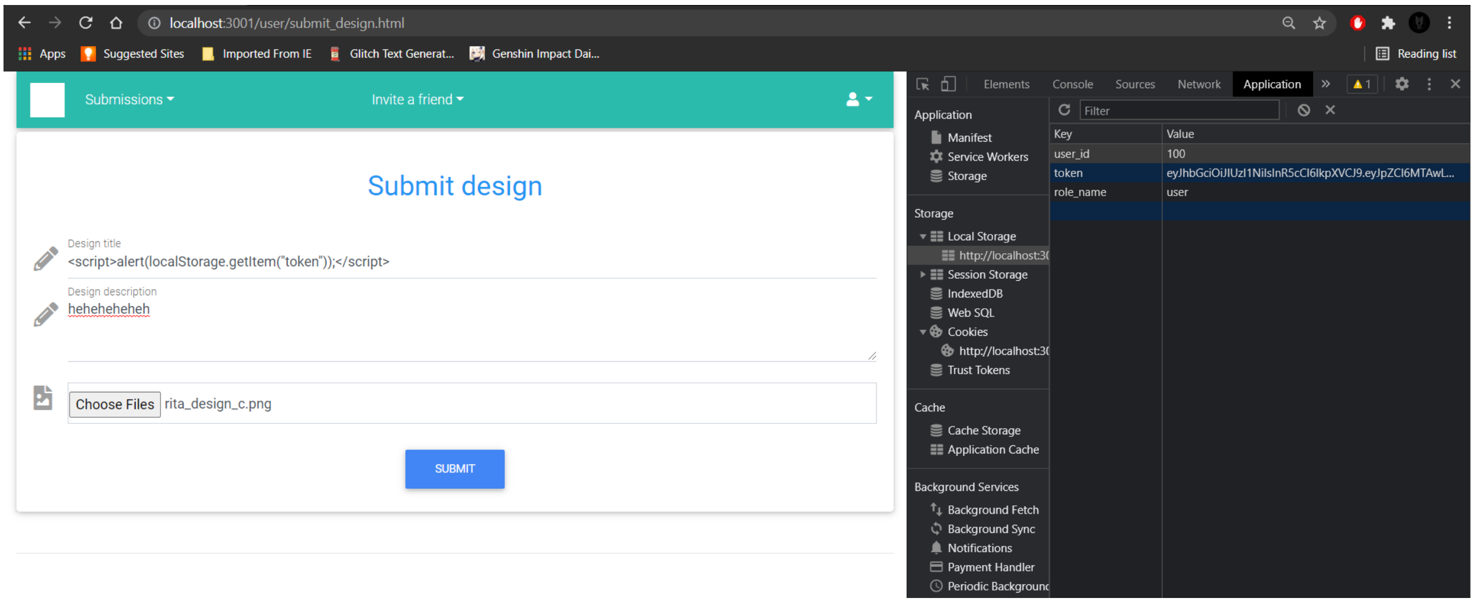
- When a user logs into the Bee Design Award Competition system, a JSON Web Token (JWT) is generated. The JWT can be considered a security key for identifying a person's identity and role (User or Administrator).
- This JWT can be accessed in the Local Storage of the web browser, and attackers can run XSS attacks to get the JWT from the Local Storage.

Impact: Low

The IT Department Team has settled for an impact rating of Low because even though the JWT does contain sensitive information, it requires the attacker to have access to both the user and the administrator's account.

How to exploit flaw:

Log in to the user account by exploiting the Broken Authentication mentioned in part 3.2 and head over to submit a design. Type in the following into the design title:


localhost:3001/user/submit_design.html

Submit design

Design title
<script>alert(localStorage.getItem("token"));</script>

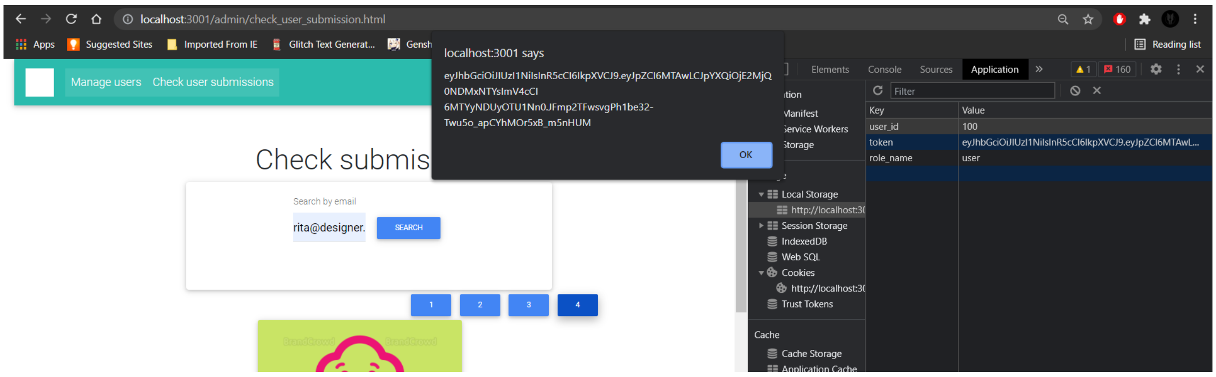
Design description
hehehehehe h

Choose Files rita_design_c.png

SUBMIT

Key	Value
user_id	100
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTAwL...
role_name	user

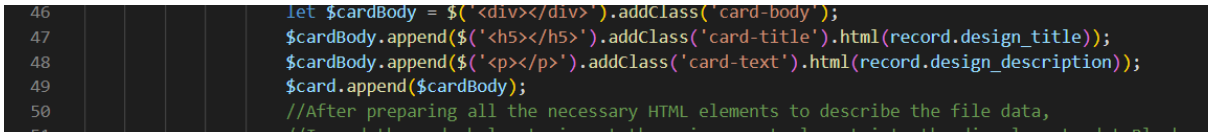
Hit “Submit” and head over to the Administrator’s website. Click on “Check User Submissions” and type in the user account’s email you used to upload the design.


localhost:3001 says

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTAwLCJpYXQiOjE2MjQ0NDMxNTYsImV4cCI
6MTYyNDUyOTU1Nn0.JFmp2TFwsvgPh1be32-Twu5o_apCYhMOr5xB_m5nHUM

Key	Value
user_id	100
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTAwL...
role_name	user

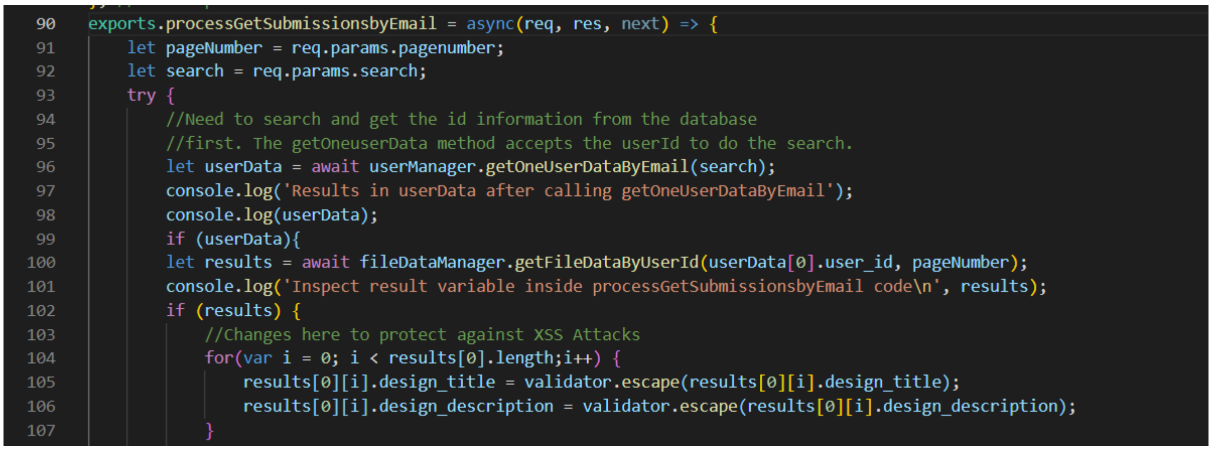
Where the flaw is found:


```
46              let $cardBody = $('<div></div>').addClass('card-body');
47              $cardBody.append($('<h5></h5>').addClass('card-title').html(record.design_title));
48              $cardBody.append($('<p></p>').addClass('card-text').html(record.design_description));
49              $card.append($cardBody);
50              //After preparing all the necessary HTML elements to describe the file data,
```

Found in public/admin/admin_check_user_submission in the original code.

The flaw is found on lines 47 and 48 of admin_check_user_submission.js, where the design title and the design description are displayed onto the page as HTML codes, allowing an XSS attack to happen.

Recommendation:

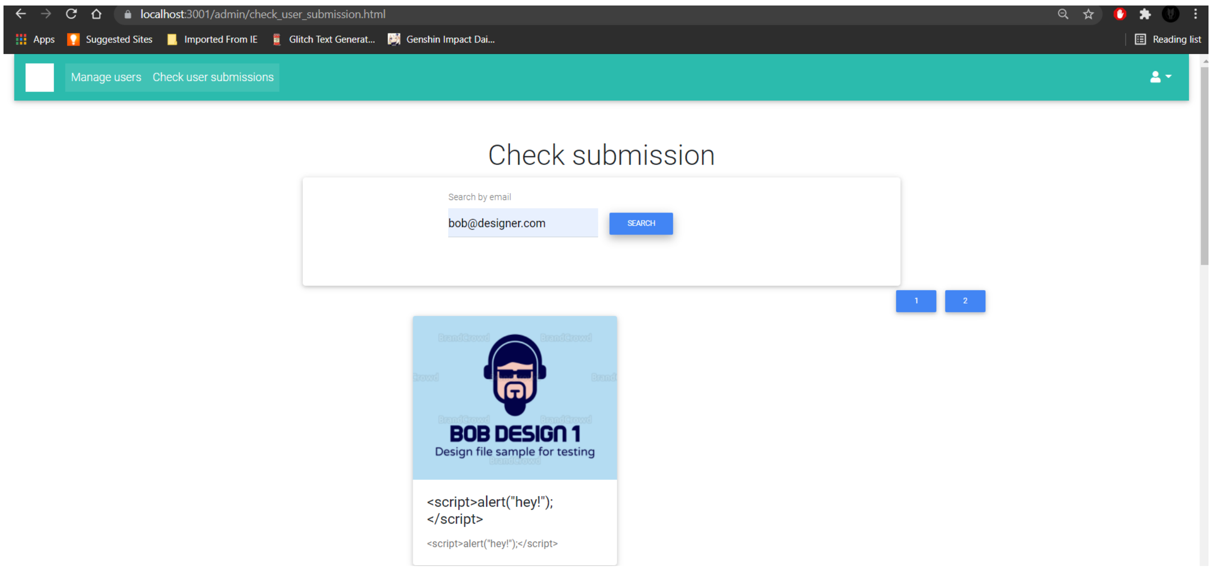


```
 90 exports.processGetSubmissionsbyEmail = async(req, res, next) => {
 91     let pageNumber = req.params.pagenumber;
 92     let search = req.params.search;
 93     try {
 94         //Need to search and get the id information from the database
 95         //first. The getOneuserData method accepts the userId to do the search.
 96         let userData = await userManager.getOneUserDataByEmail(search);
 97         console.log('Results in userData after calling getOneUserDataByEmail');
 98         console.log(userData);
 99         if (userData){
100         let results = await fileDataManager.getFileDataByUserId(userData[0].user_id, pageNumber);
101         console.log('Inspect result variable inside processGetSubmissionsbyEmail code\n', results);
102         if (results) {
103             //Changes here to protect against XSS Attacks
104             for(var i = 0; i < results[0].length;i++) {
105                 results[0][i].design_title = validator.escape(results[0][i].design_title);
106                 results[0][i].design_description = validator.escape(results[0][i].design_description);
107             }
```

Found in /src/controllers/userController.js in the edited code.

After getting the results from the middleware function processGetSubmissionbyEmail in userController.js, use validator.escape to ensure that any characters that may form harmful lines of codes (such as < and >) are replaced with HTML entities in the title and the description, which can be displayed safely. This can be seen in lines 105 and 106 in the image shown above.

If we try to exploit the vulnerability again…



```
<script>alert("hey!");
</script>
<script>alert("hey!");</script>
```

The script no longer runs and instead is displayed safely onto the website.

3.3.2 Sensitive Data Exposure of Transmitted Information

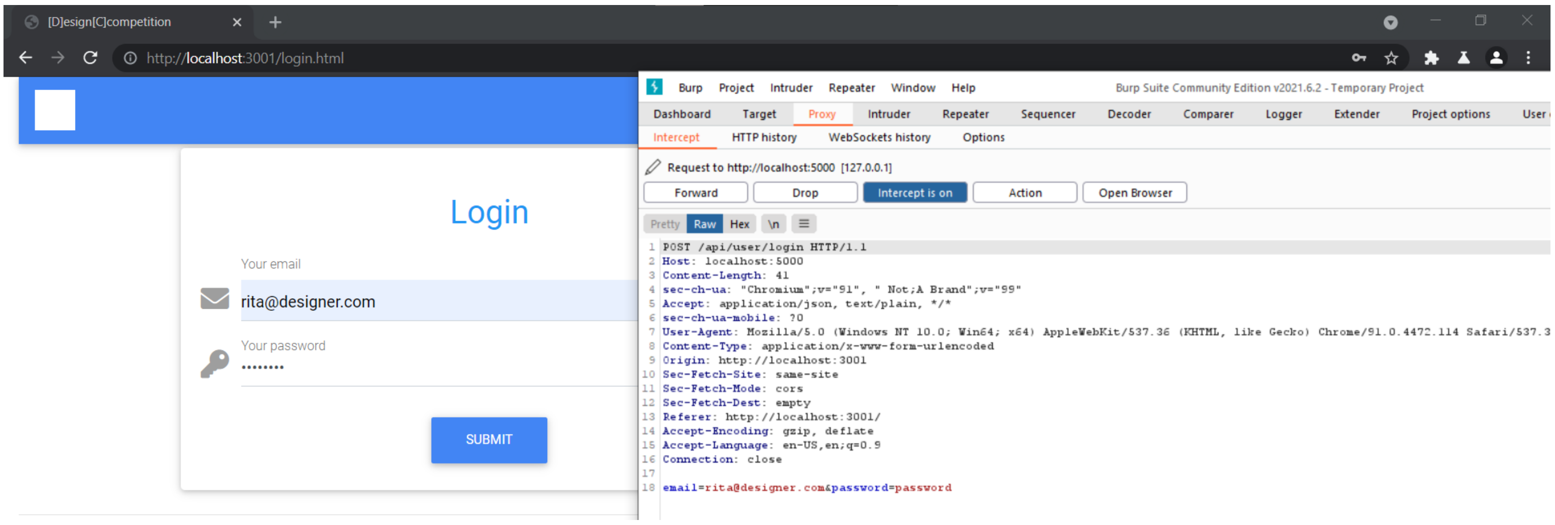
- User's data is sent through HTTP (Hypertext Transfer Protocol) instead of HTTPS. HTTP and HTTPS are protocols that web browsers use to transmit information. HTTPS is the secure version of HTTP, with the S standing for Secure.
- Transmitting information using HTTP allows attackers to use a tool like BURP Suite to tap into the transmitted information.
- Using HTTPS, the information transmitted is encrypted first before being transmitted.

Impact: Medium

The IT Development Team has settled on an impact rating of medium as the chance of having an attacker intercepting the information transmitted is low. The attack would need a way to intercept the connection between the user and the system, which is hard to do.

How to exploit flaw:

Open Burp Suite, and click on Open Browser under Proxy > Intercept. Then attempt to log into the Bee Design Award Competition system.


[D]esign[C]competition
http://localhost:3001/login.html
Login
Your email
rita@designer.com
Your password
••••••••
SUBMIT
Burp Project Intruder Repeater Window Help
Burp Suite Community Edition v2021.6.2 - Temporary Project
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User
Intercept HTTP history WebSockets history Options
Forward Drop Intercept is on Action Open Browser
Pretty Raw Hex \n

Request to http://localhost:5000 [127.0.0.1]

```
POST /api/user/login HTTP/1.1
Host: localhost:5000
Content-Length: 41
sec-ch-ua: "Chromium";v="91", " Not;A Brand";v="99"
Accept: application/json, text/plain, */*
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.114 Safari/537.3
Content-Type: application/x-www-form-urlencoded
Origin: http://localhost:3001
Sec-Fetch-Site: same-site
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3001/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

email=rita@designer.com&password=password
```

The email and password are shown in the Burp Suite application, which is a sensitive data exposure.0

Where the flaw is found:

The flaw is due to a lack of code to enforce HTTPS over HTTP, and not due to a fault in the existing source code.

Recommendation:
Transmit data over HTTPS by employing SSL to encrypt information transmitted.

3. Open Command Prompt with Administrative rights and type in the following to install chocolatey, which will be used later.

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

   Hit the Enter key on your keyboard and continue with the installation of chocolatey.

4. After that, run the following lines of code in the Command Prompt with Administrative right:

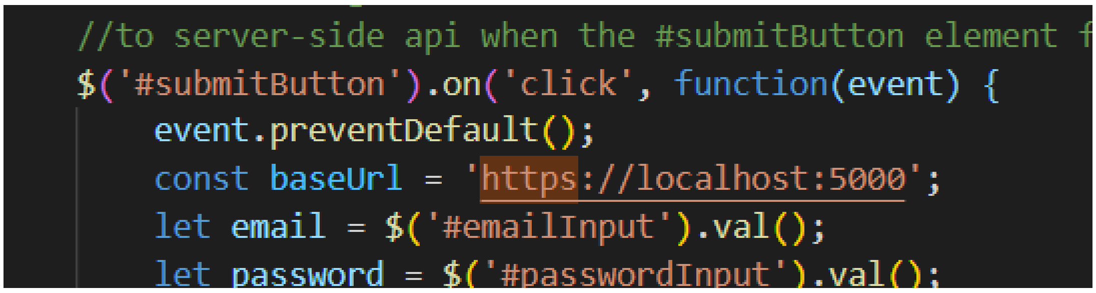
   ```
   choco install mkcert
   mkcert -install
   mkcert localhost
   ```

   Note down the directory of the key and the certificate starting from your C: drive (or the drive where the Command Prompt was running from)

5. Open index.js for both the front end client and the back end server. Insert these lines of code where necessary.

```
let app = express();
const https = require('https');
const fs = require('fs');
const options = {
  key: fs.readFileSync('INSERT_KEY_DIRECTORY_HERE'),
  cert:
fs.readFileSync('INSERT_CERTIFICATE_DIRECTORY_HERE'),
};
https
  .createServer(options,
app).listen({port:INSERT_PORT_NUMBER});
```

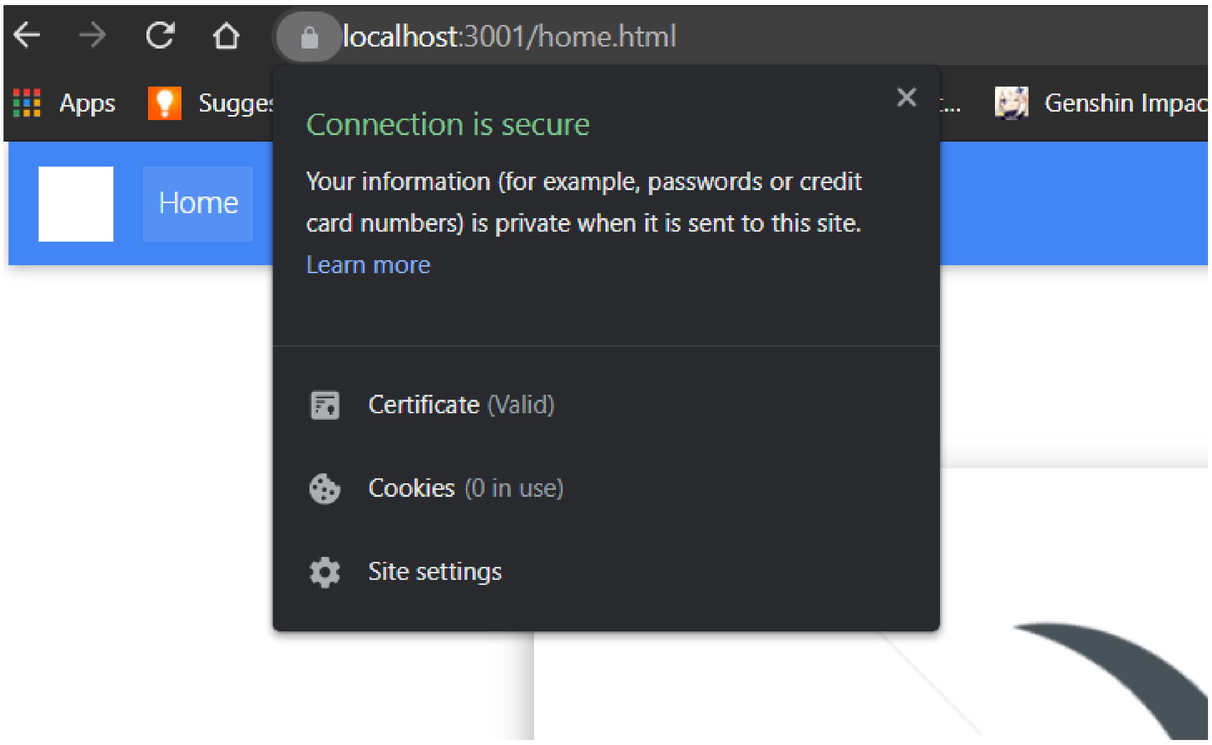
6.
    a. Hold the Windows key and R at the same time. Type in mmc.exe and hit the Enter key. Click Yes when prompted whether to allow the application to make changes.
    b. Go to File > Add/Remove Snap-In
    c. Double click on Certificates on the left-hand column > Computer Account > Local Computer > Finish
    d. Click Ok to dismiss the Snap-In window.
    e. Click on the arrow next to Certificates in the left-hand column and go to Personal > Certificates
    f. Right-click on the certificate named ‘localhost’ and Copy the certificate.
    g. Navigate to the Trusted Root Certification Authorities folder in the same directory as the Personal folder. Right-click on the Certificates folder in the Trusted Root Certification Authorities folder in the left-hand column and click Paste.
7. Head back to the front end js files and ensure that the base URLs for all the js files are changed to HTTPS.



```
//to server-side api when the #submitButton element f
$('#submitButton').on('click', function(event) {
    event.preventDefault();
    const baseUrl = 'https://localhost:5000';
    let email = $('#emailInput').val();
    let password = $('#passwordInput').val();
```

Found in /public/js/login.js in the edited code.

After fixing, you should be able to see the following:



Connection is secure

Your information (for example, passwords or credit card numbers) is private when it is sent to this site. Learn more

Certificate (Valid)

Cookies (0 in use)

Site settings

Seeing this means that the connection is secure and the vulnerability is fixed.

3.4 Broken Access Control

- Broken Access Control is when a user or attacker is granted the right to perform an action that they are not granted permission to do. A real-life example would be an employee being able to view the salaries of everyone in the company.

3.4.1 Broken Access Control for Users and Administrators.

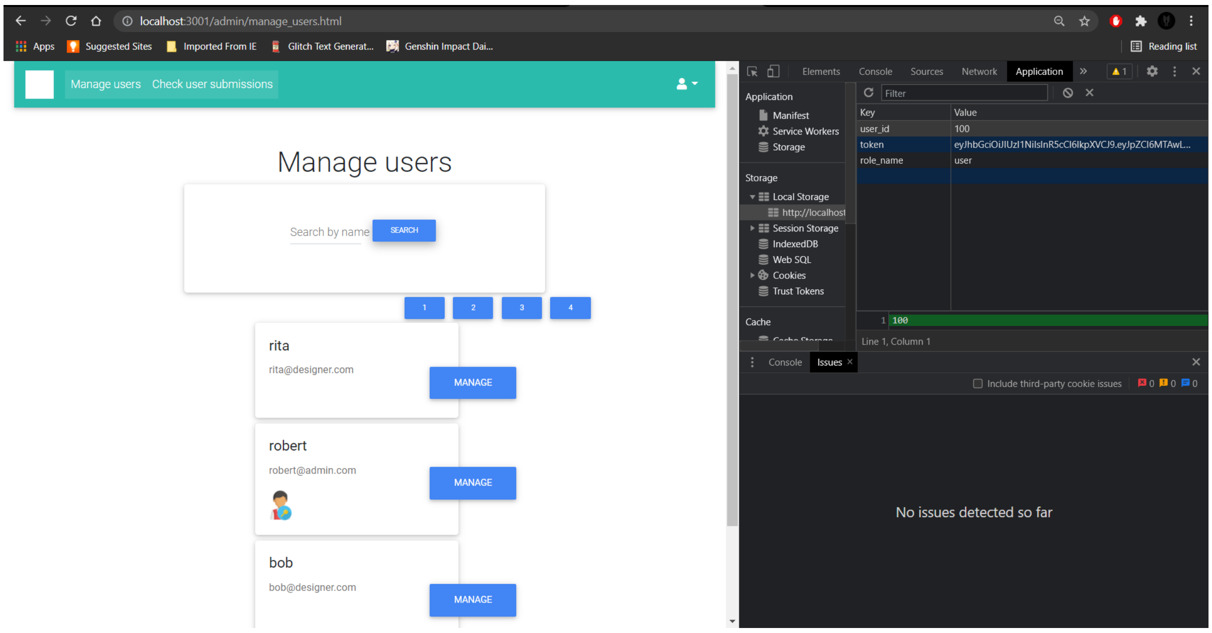
- Currently, in the Bee Design Award Competition system, users are allowed to access pages that have administrative functions, such as the page to allow administrators to view and edit user roles, and the page to allow administrators to view the submissions of a user.

Impact: High

The IT Department Team has given this vulnerability an impact rating of High as administrators and users alike could sabotage the competition using this vulnerability.

How to exploit flaw:

Log in to the user account and type in the Administrator's website URL.


localhost:3001/admin/manage_users.html

Manage users

Key	Value
user_id	100
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTAwL...
role_name	user

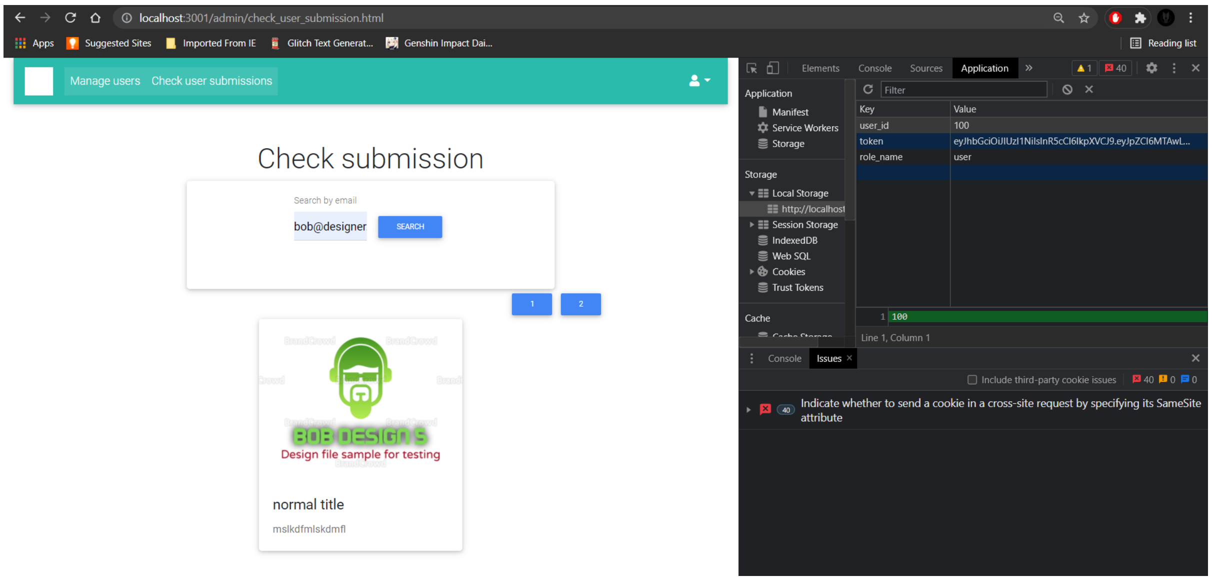
rita
rita@designer.com

robert
robert@admin.com

bob
bob@designer.com

No issues detected so far

This is a problem as the user could change his/her role to an administrator, or worse still abuse the administrative rights to check on user's submissions as shown below, where rita is checking on bob's work:

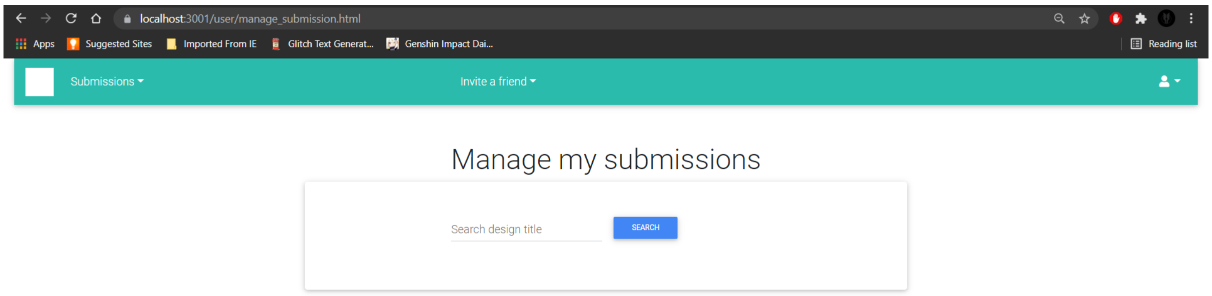


Key	Value
user_id	100
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTAwL...
role_name	user

```
1 100
Line 1, Column 1
```

Indicate whether to send a cookie in a cross-site request by specifying its SameSite attribute

If we try to exploit the vulnerability again:



The user gets redirected back to the pages that they have access to.

3.5 Cross-Site Scripting

- Cross-site scripting(XSS) is a type of attack that enables attackers to ‘inject’ client-side scripts into web pages to be viewed by others.
- An example of this is when attackers send malicious scripts to the server where it gets stored. Later when a user sends a request to the server, the server might send that malicious script back to the user. Once the user has received the malicious script, the browser will run the malicious script.
- Through the malicious script, attackers can choose to enact actions like send the security token to him or make the website practically unusable (e.g. infinite alerts).

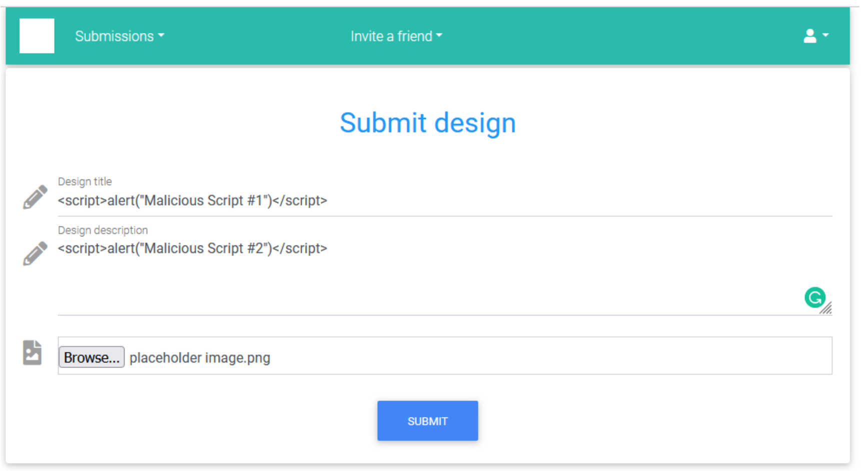
3.5.1 User Submission XSS

Impact: High

The application is vulnerable to “Persistent” Cross-site scripting (XSS). Persistent cross-site scripting is simply attackers sending malicious scripts which get inadvertently saved in the server’s database. Then when a normal user requests something from the server, the malicious code is sent to them and thus gets run by the browser. The team chose to give this vulnerability a rating of “High” as through XSS, attackers can get access to sensitive information or make the application unusable(infinite alerts).

How to exploit flaw:

In the image below, we can see that users can send code to the server.

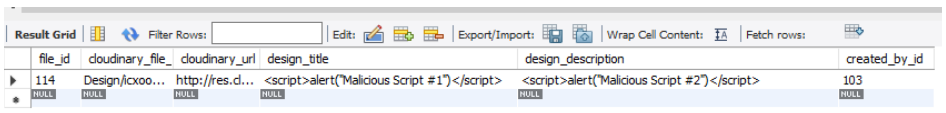


Design title: `<script>alert("Malicious Script #1")</script>`

Design description: `<script>alert("Malicious Script #2")</script>`

placeholder image.png

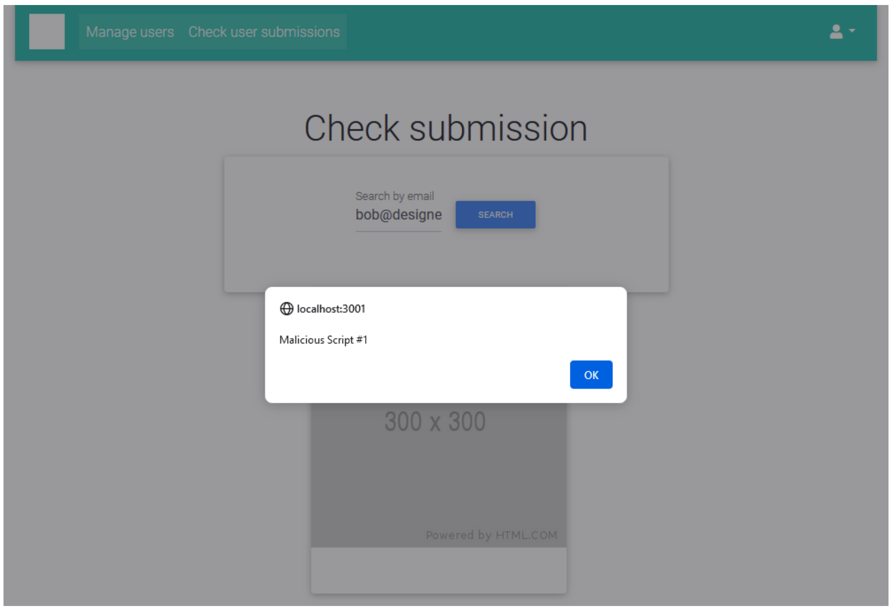
Currently, there's no check to ensure that scripting elements, like <script>, are not saved in the database. So the 'malicious script' is saved. You can see this under the field design_title and design_description in the image below.



file_id	cloudinary_file_	cloudinary_url	design_title	design_description	created_by_id
114	Design/icxoo...	http://res.cl...	<script>alert("Malicious Script #1")</script>	<script>alert("Malicious Script #2")</script>	103
NULL	NULL	NULL	NULL	NULL	NULL

(Continued Below)

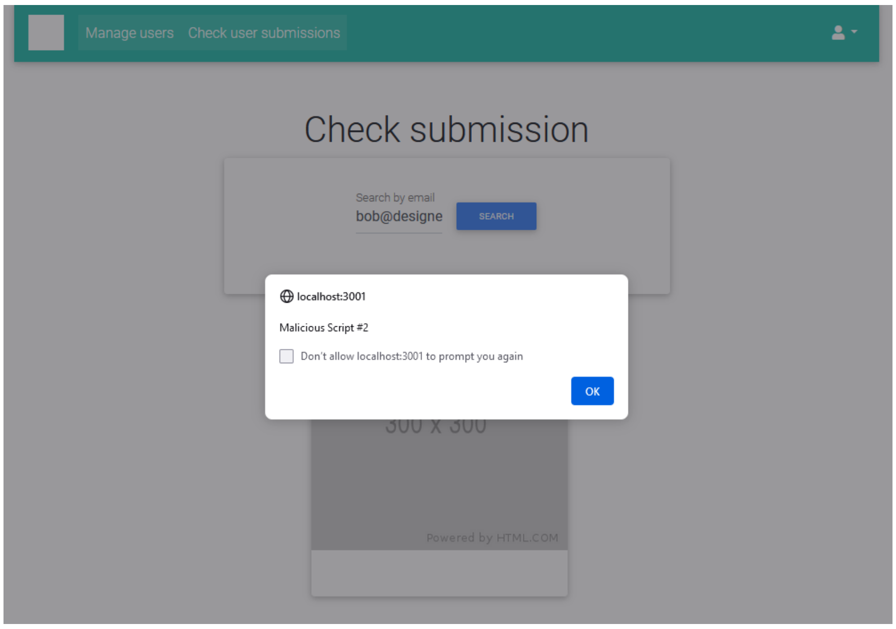
Later when an admin requests for the user's submission. The 'malicious scripts' are run.



localhost:3001

Malicious Script #1

OK



localhost:3001

Malicious Script #2

Don't allow localhost:3001 to prompt you again

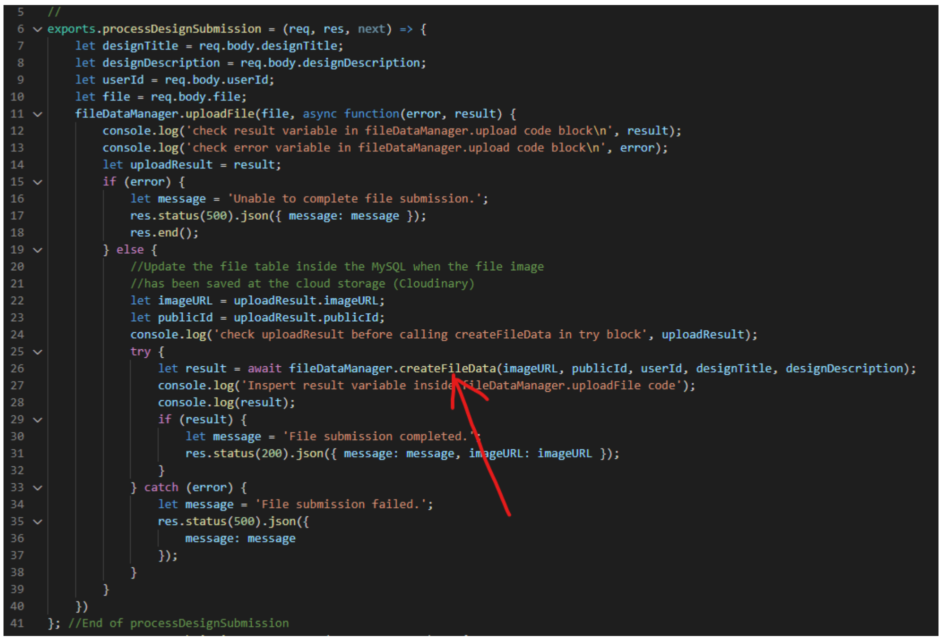
OK

The alert script example above is relatively harmless and can be cleared when the admin presses 'OK'. But the attacker can apply more harmful scripts. Like infinite looping the alert or sending the 'security token' to him.

(Continued Below)

Where flaw is found:

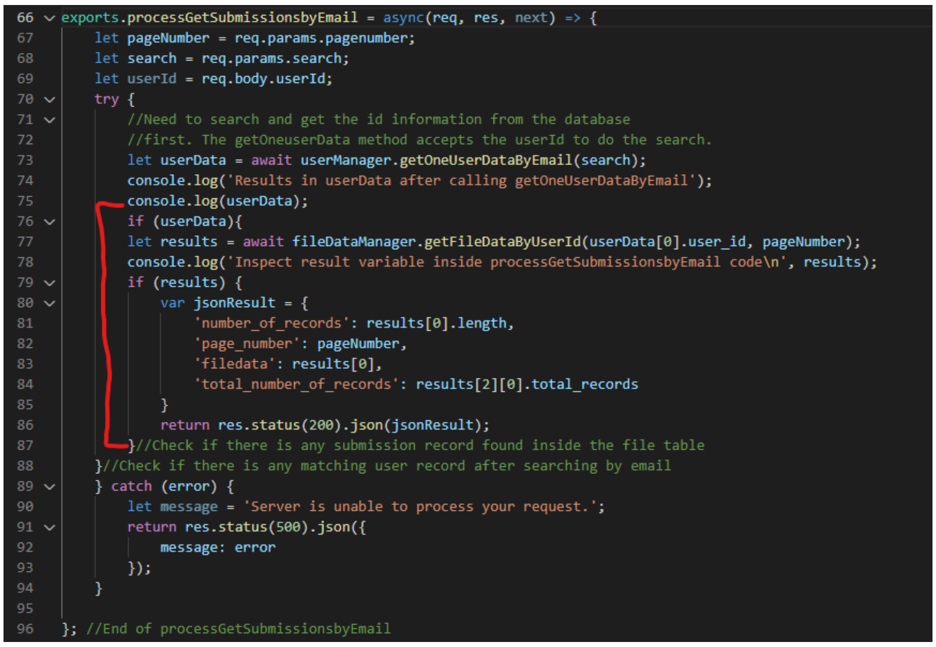
As seen below there's no check for the designTitle and designDescription to ensure that it doesn't contain scripts before submissions get accepted. So 'malicious scripts' get saved in the database through the createFileData() function in line 27.



```
5   //
6   exports.processDesignSubmission = (req, res, next) => {
7       let designTitle = req.body.designTitle;
8       let designDescription = req.body.designDescription;
9       let userId = req.body.userId;
10      let file = req.body.file;
11      fileDataManager.uploadFile(file, async function(error, result) {
12          console.log('check result variable in fileDataManager.upload code block\n', result);
13          console.log('check error variable in fileDataManager.upload code block\n', error);
14          let uploadResult = result;
15          if (error) {
16              let message = 'Unable to complete file submission.';
17              res.status(500).json({ message: message });
18              res.end();
19          } else {
20              //Update the file table inside the MySQL when the file image
21              //has been saved at the cloud storage (Cloudinary)
22              let imageURL = uploadResult.imageURL;
23              let publicId = uploadResult.publicId;
24              console.log('check uploadResult before calling createFileData in try block', uploadResult);
25              try {
26                  let result = await fileDataManager.createFileData(imageURL, publicId, userId, designTitle, designDescription);
27                  console.log('Inspert result variable inside fileDataManager.uploadFile code');
28                  console.log(result);
29                  if (result) {
30                      let message = 'File submission completed.';
31                      res.status(200).json({ message: message, imageURL: imageURL });
32                  }
33              } catch (error) {
34                  let message = 'File submission failed.';
35                  res.status(500).json({
36                      message: message
37                  });
38              }
39          }
40      })
41  }; //End of processDesignSubmission
```

Found in controller/userController.js in the original version.

Furthermore, when admins request a user's submissions, the server doesn't **escape** the result. When the result isn't escaped, the client's web browser takes the <script> literally and therefore runs it.



```
exports.processGetSubmissionsbyEmail = async(req, res, next) => {
    let pageNumber = req.params.pagenumber;
    let search = req.params.search;
    let userId = req.body.userId;
    try {
        //Need to search and get the id information from the database
        //first. The getOneuserData method accepts the userId to do the search.
        let userData = await userManager.getOneUserDataByEmail(search);
        console.log('Results in userData after calling getOneUserDataByEmail');
        console.log(userData);
        if (userData){
        let results = await fileDataManager.getFileDataByUserId(userData[0].user_id, pageNumber);
        console.log('Inspect result variable inside processGetSubmissionsbyEmail 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);
        }//Check if there is any submission record found inside the file table
    }//Check if there is any matching user record after searching by email
    } catch (error) {
        let message = 'Server is unable to process your request.';
        return res.status(500).json({
            message: error
        });
    }

}; //End of processGetSubmissionsbyEmail
```

Found in controller/userController.js in the original version

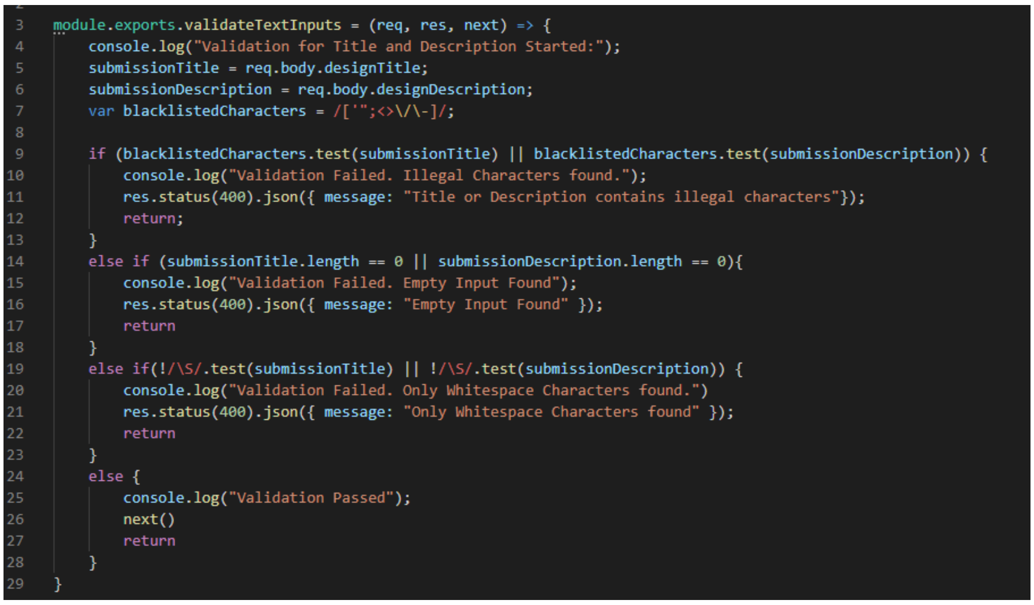
Recommendations:

There are a few ways to solve the XSS above. The most efficient ways to secure the vulnerability is to:

1. Input Validation - by performing input validation, we're sure that <scripts> and their associated illegal characters like '<' and '>' are not saved in the database and therefore returned to users.
2. Output Sanitization - if attackers can bypass the input validation, we can fall back on output sanitization (defence in depth) by escaping the output from the server. This ensures that if scripts are returned to users, their web browser doesn't execute the malicious script.

Input Validation

Middleware function which checks the title and description of user's submissions for illegal characters, whether the input is empty, or only contains spaces.



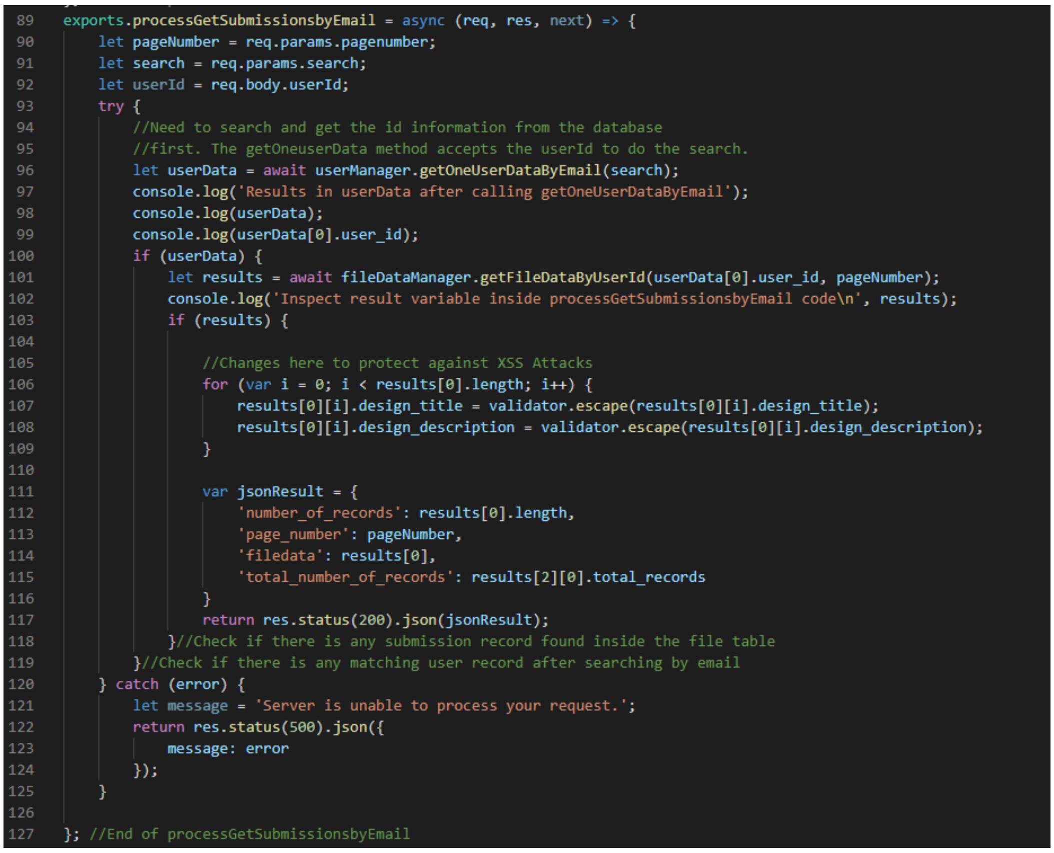
```
module.exports.validateTextInputs = (req, res, next) => {
    console.log("Validation for Title and Description Started:");
    submissionTitle = req.body.designTitle;
    submissionDescription = req.body.designDescription;
    var blacklistedCharacters = /['";<>\/\-]/;

    if (blacklistedCharacters.test(submissionTitle) || blacklistedCharacters.test(submissionDescription)) {
        console.log("Validation Failed. Illegal Characters found.");
        res.status(400).json({ message: "Title or Description contains illegal characters"});
        return;
    }
    else if (submissionTitle.length == 0 || submissionDescription.length == 0){
        console.log("Validation Failed. Empty Input Found");
        res.status(400).json({ message: "Empty Input Found" });
        return
    }
    else if(!/\S/.test(submissionTitle) || !/\S/.test(submissionDescription)) {
        console.log("Validation Failed. Only Whitespace Characters found.")
        res.status(400).json({ message: "Only Whitespace Characters found" });
        return
    }
    else {
        console.log("Validation Passed");
        next()
        return
    }
}
```

Found in middlewares/checkValidTitle&Description.js in the edited version

Note: The above code can also be used to check inputs that might interact with SQL code (SQL Injections) in the future. Just replace the variable 'submissionTitle' or 'submissionDescription' with the value that you want to get checked.

Output Sanitization

Changes in line 105 to 109.



```
89   exports.processGetSubmissionsbyEmail = async (req, res, next) => {
90       let pageNumber = req.params.pagenumber;
91       let search = req.params.search;
92       let userId = req.body.userId;
93       try {
94           //Need to search and get the id information from the database
95           //first. The getOneuserData method accepts the userId to do the search.
96           let userData = await userManager.getOneUserDataByEmail(search);
97           console.log('Results in userData after calling getOneUserDataByEmail');
98           console.log(userData);
99           console.log(userData[0].user_id);
100          if (userData) {
101              let results = await fileDataManager.getFileDataByUserId(userData[0].user_id, pageNumber);
102              console.log('Inspect result variable inside processGetSubmissionsbyEmail code\n', results);
103              if (results) {
104
105                  //Changes here to protect against XSS Attacks
106                  for (var i = 0; i < results[0].length; i++) {
107                      results[0][i].design_title = validator.escape(results[0][i].design_title);
108                      results[0][i].design_description = validator.escape(results[0][i].design_description);
109                  }
110
111                  var jsonResult = {
112                      'number_of_records': results[0].length,
113                      'page_number': pageNumber,
114                      'filedata': results[0],
115                      'total_number_of_records': results[2][0].total_records
116                  }
117                  return res.status(200).json(jsonResult);
118              }//Check if there is any submission record found inside the file table
119          }//Check if there is any matching user record after searching by email
120      } catch (error) {
121          let message = 'Server is unable to process your request.';
122          return res.status(500).json({
123              message: error
124          });
125      }
126
127  }; //End of processGetSubmissionsbyEmail
```

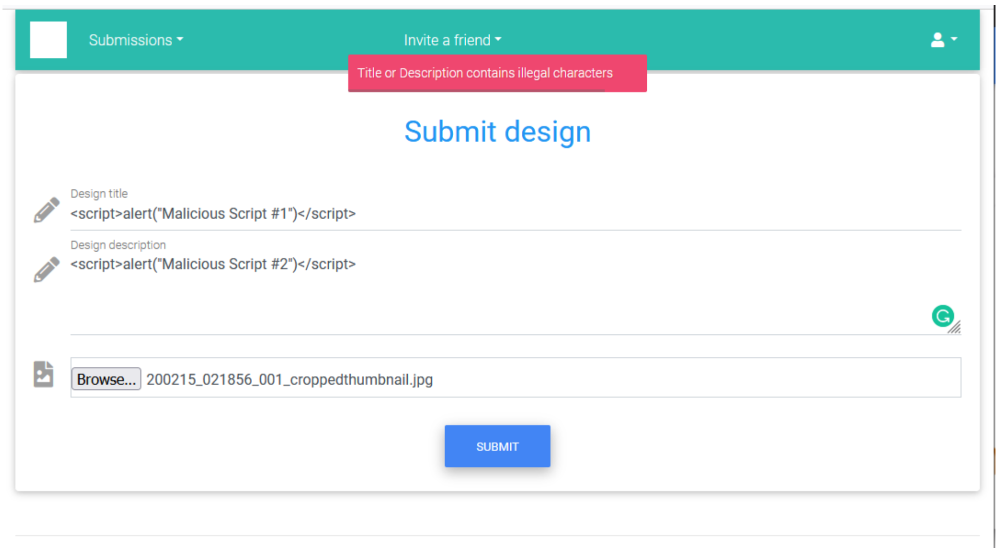
Found in controllers/userController.js in the edited version.

We can use validator’s escape function to escape the results before sending them back to users. By escaping the results, the web browser doesn’t take any code literally and thus doesn’t run any malicious code that attackers might’ve stored in the database.

(Continued Below)

Results:

Firstly, we can tell that the input validation works. As of now when a user tries to input into the title and description one of the illegal characters, it wouldn't get accepted and the user will receive an error saying "Title or Description contains illegal characters".



Title or Description contains illegal characters

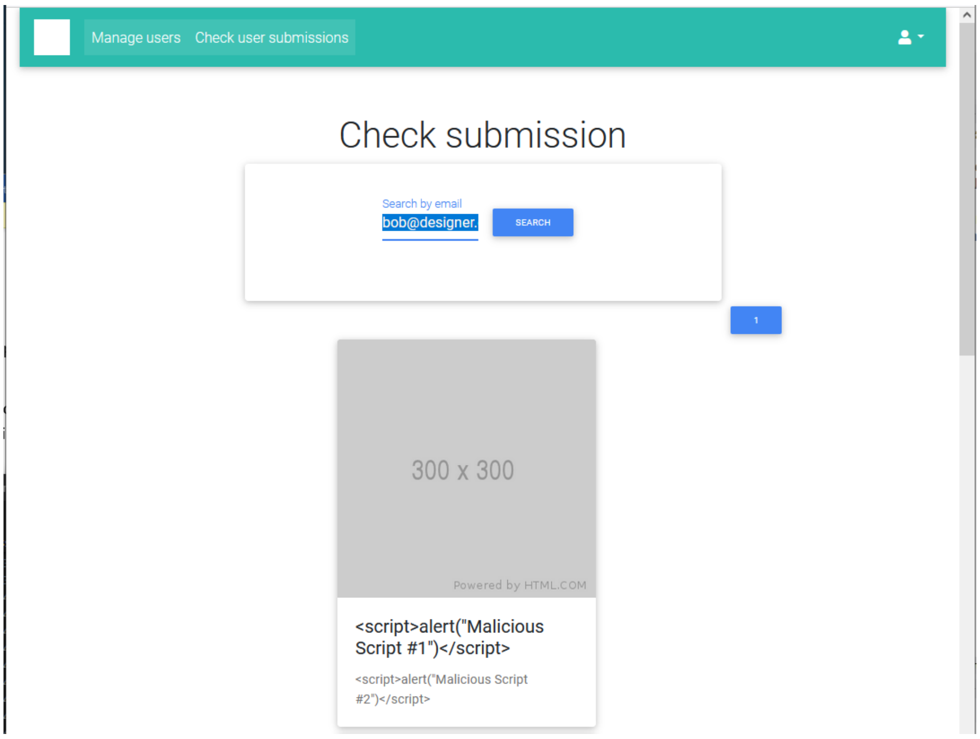
Submit design

Design title
<script>alert("Malicious Script #1")</script>

Design description
<script>alert("Malicious Script #2")</script>

Browse... 200215_021856_001_croppedthumbnail.jpg

Secondly, since we escaped the server's response before it's sent. The web browser doesn't take the <script> element literally as shown below.



Check submission

Search by email
bob@designer.

<script>alert("Malicious Script #1")</script>

<script>alert("Malicious Script #2")</script>

3.6 Insufficient Logging and Monitoring

- Logging is storing a record of events after users try to perform sensitive interactions with the application and back-end server.
- Monitoring is what alerts the server administrators that something has gone wrong and how to fix it.

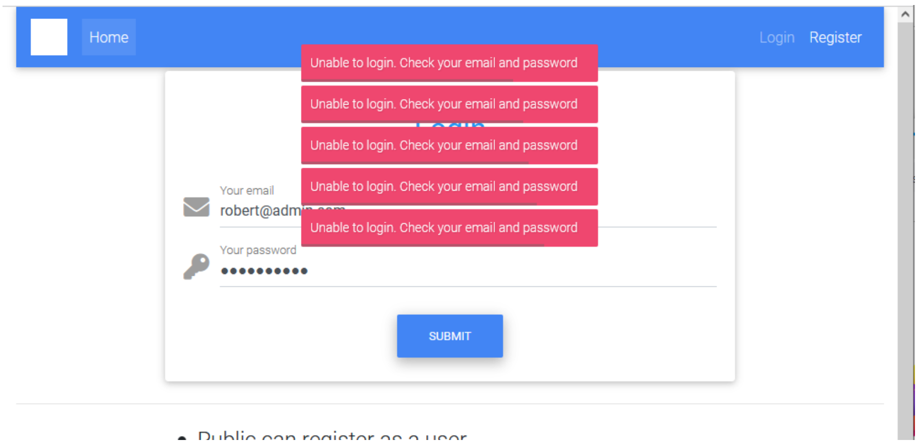
3.6.1 Logging and Monitoring in Login Page

Impact: High

Through brute-forcing the login page, attackers can eventually login to any account. Not only that, but users can also hang the system by spamming requests to the server.

How to exploit flaw:

On the login page, there's no limit on how many times you try to log in.

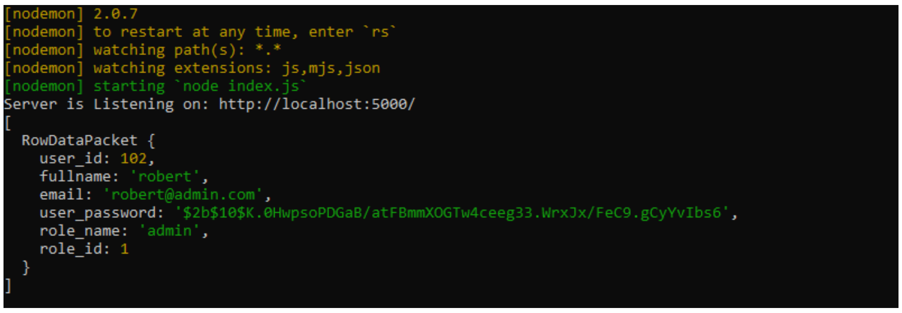


Unable to login. Check your email and password
Unable to login. Check your email and password
Unable to login. Check your email and password
Unable to login. Check your email and password
Unable to login. Check your email and password

This makes the application vulnerable to brute force attacks.
For example, the attacker can have a list of common passwords.
Then, use an automated tool that automatically fills in the password field above and submits. Given enough time, the attacker would be able to gain access to the admin user, robert@admin.com, shown above. Thus enabling him to enact admin actions that he wouldn't have otherwise had permission to do.

(Continued Below)

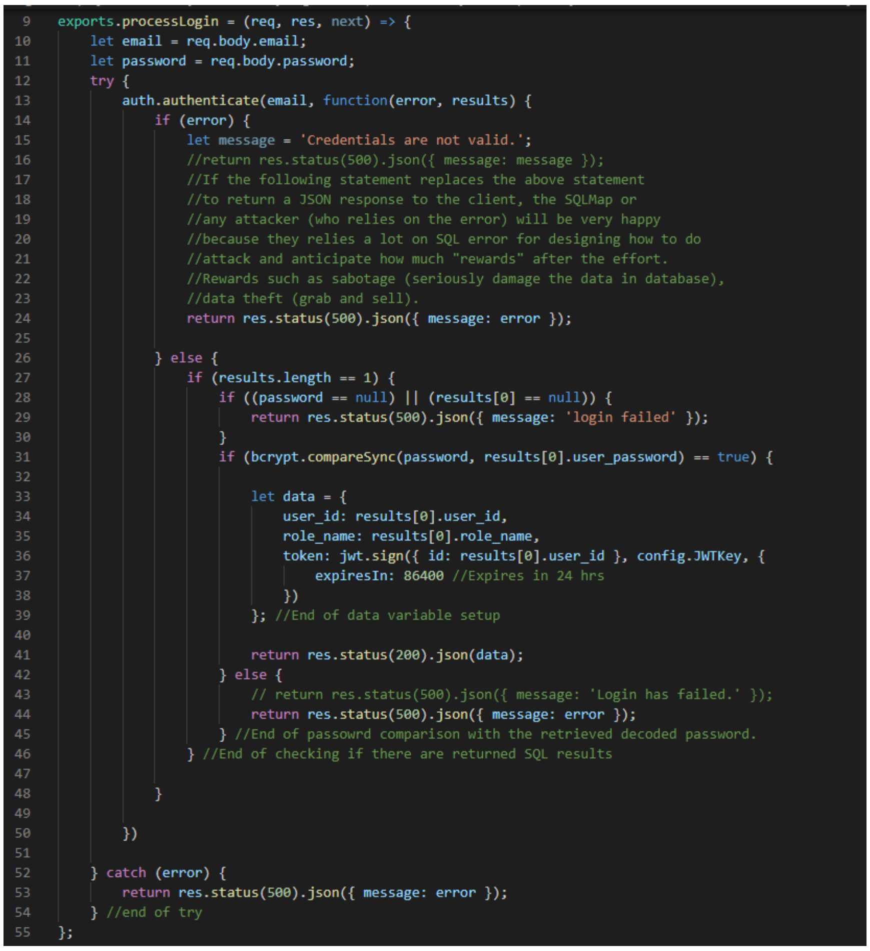
Whenever users try to login into the application, the console only shows the details of the user with that email (i.e. robert@admin.com's details)



```
[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node index.js`
Server is Listening on: http://localhost:5000/
[
  RowDataPacket {
    user_id: 102,
    fullname: 'robert',
    email: 'robert@admin.com',
    user_password: '$2b$10$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg33.WrxJx/FeC9.gCyYvIbs6',
    role_name: 'admin',
    role_id: 1
  }
]
```

The console doesn't show whether the login was successful or not.
Furthermore, the login isn't logged or noted anywhere.
So there isn't any way for server administrators to check that the password is being brute forced or exploited.

Where the flaw is found:

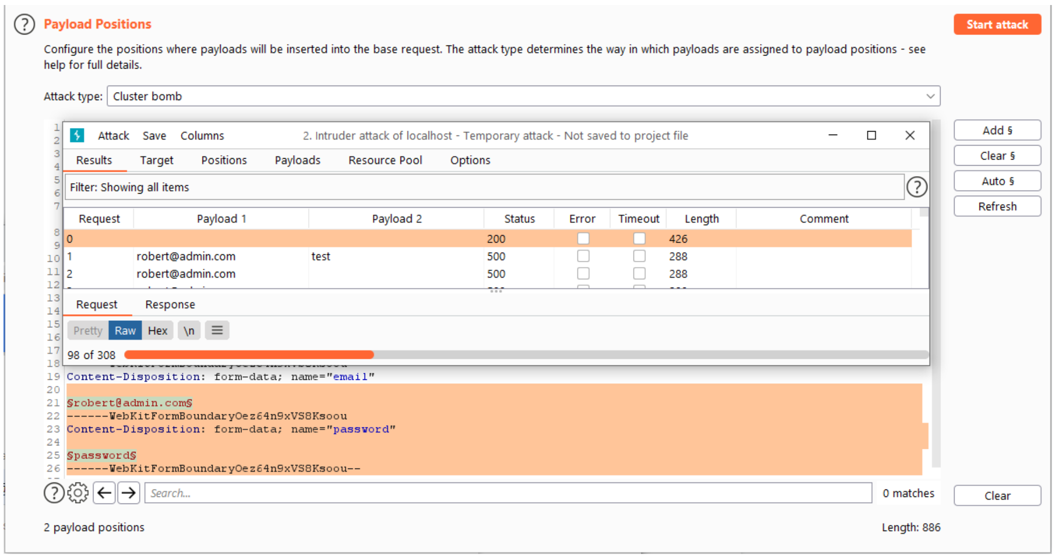


```
exports.processLogin = (req, res, next) => {
    let email = req.body.email;
    let password = req.body.password;
    try {
        auth.authenticate(email, function(error, results) {
            if (error) {
                let message = 'Credentials are not valid.';
                //return res.status(500).json({ 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
};
```

Found in controller/authController.js in the original version

From the above code, which processes the login, there aren't any attempts to log, monitor, or limit the number of times users try to log in.
In addition to brute force attacks above, having no limit to the number of login attempts allows attackers to exploit the application by using automated tools to 'spam' requests onto the server. As long as the attacker can send requests faster than the server can handle one request, the server will hang perpetually.

For example (similar to the 'request spam' on the login page):
Using Burp Suite's Intruder feature to spam requests, we managed to slow down the submission speed from nearly instant (normal submissions) to take as long as 12 seconds to handle each new login request (87th request). And the server fully crashes at the 98th request.



Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions - see help for full details.

Attack type: Cluster bomb

2. Intruder attack of localhost - Temporary attack - Not saved to project file

Filter: Showing all items

Request	Payload 1	Payload 2	Status	Error	Timeout	Length	Comment
0			200	☐	☐	426	
1	robert@admin.com	test	500	☐	☐	288	
2	robert@admin.com		500	☐	☐	288	

98 of 308

```
Content-Disposition: form-data; name="email"

§robert@admin.com§
------WebKitFormBoundaryOez64n9xVS8Ksoou
Content-Disposition: form-data; name="password"

§password§
------WebKitFormBoundaryOez64n9xVS8Ksoou--
```

2 payload positions

Length: 886

Recommendation:

To stop the spamming of logins, we first need to know that they've tried to log in recently.

Create a new table in MySQL for login attempts using the settings below.

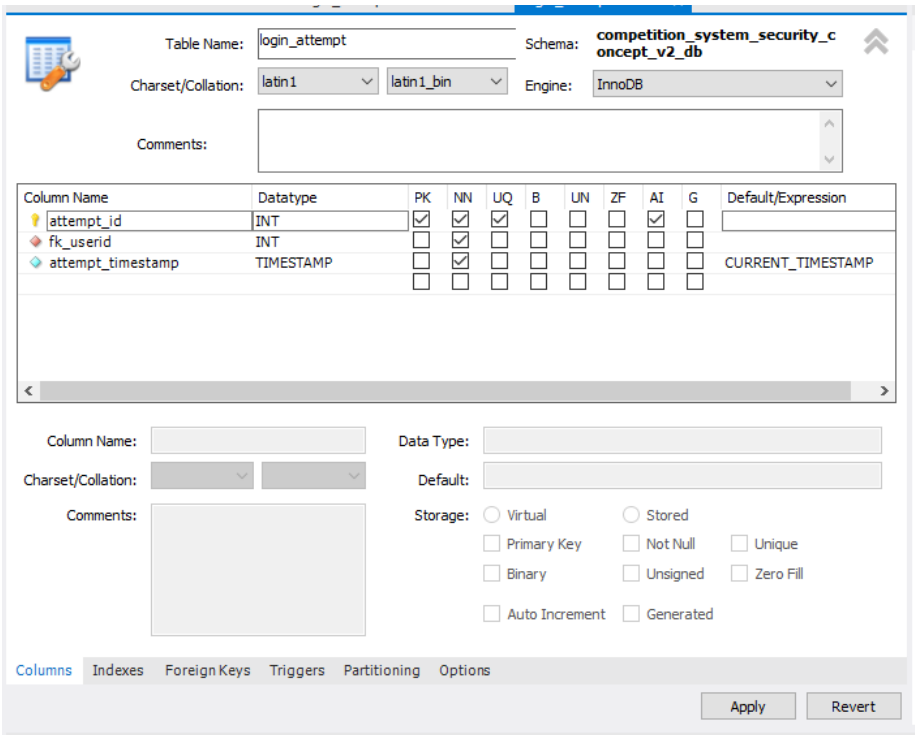


Table Name: login_attempt
Schema: competition_system_security_concept_v2_db
Charset/Collation: latin1 latin1_bin
Engine: InnoDB
Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
attempt_id	INT	✓	✓	✓				✓		
fk_userid	INT		✓							
attempt_timestamp	TIMESTAMP		✓							CURRENT_TIMESTAMP

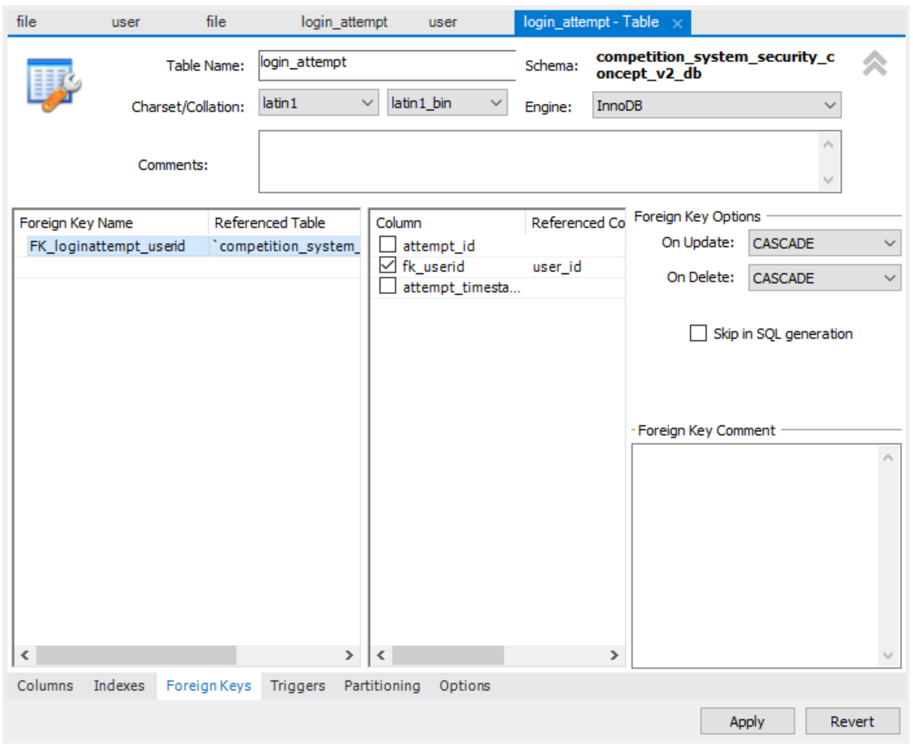


Table Name: login_attempt
Schema: competition_system_security_concept_v2_db
Charset/Collation: latin1 latin1_bin
Engine: InnoDB
Comments:

Foreign Key Name	Referenced Table
FK_loginattempt_userid	`competition_system_

Column	Referenced Co
attempt_id	
✓ fk_userid	user_id
attempt_timesta...	

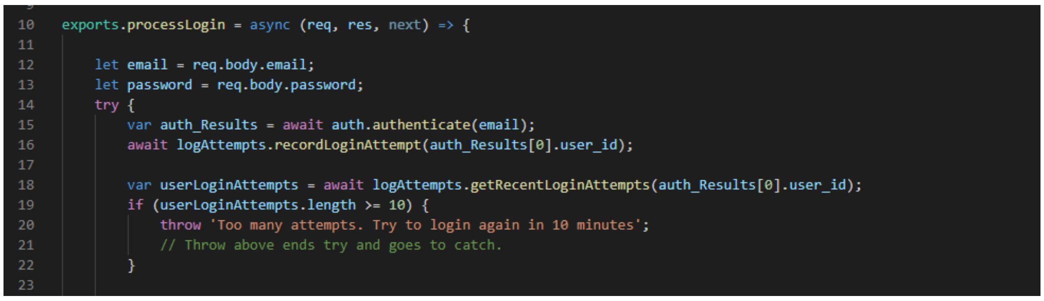
Foreign Key Options
On Update: CASCADE
On Delete: CASCADE
Skip in SQL generation

file user file login_attempt user login_attempt - Table
Foreign Key Comment
Columns Indexes Foreign Keys Triggers Partitioning Options
Apply Revert

Note: The referenced table is the database's ***user*** table.

Then, add two functions that run after the email authentication (line 15), that:

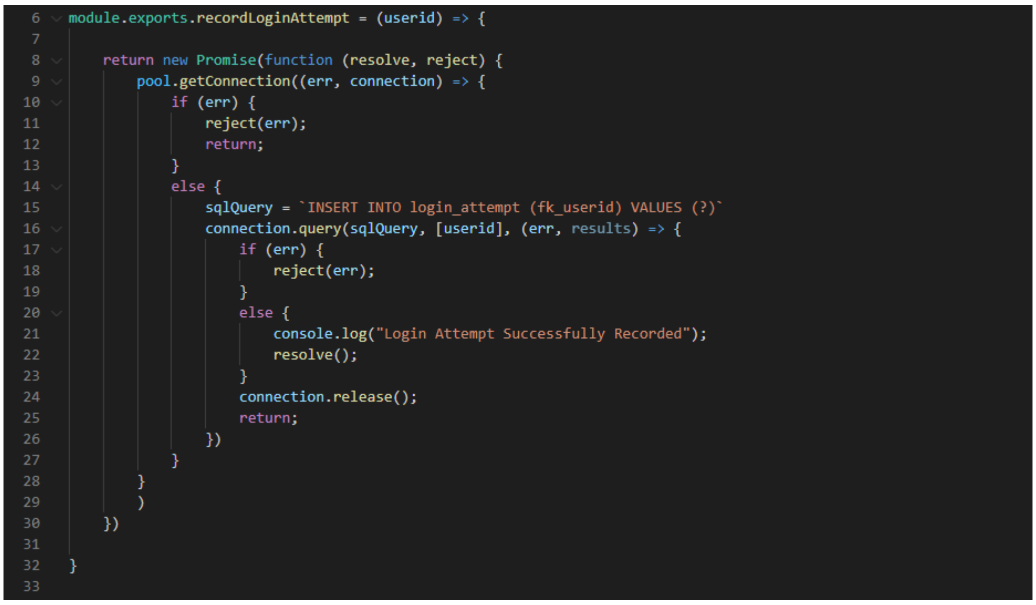
1. Records login attempts (line 16)
2. Get's users **recent** login attempts (line 18)
3. If the user has tried to log in more than 10 times in the timeframe, then discard the login attempt(line 19 - 22).



```
10  exports.processLogin = async (req, res, next) => {
11
12      let email = req.body.email;
13      let password = req.body.password;
14      try {
15          var auth_Results = await auth.authenticate(email);
16          await logAttempts.recordLoginAttempt(auth_Results[0].user_id);
17
18          var userLoginAttempts = await logAttempts.getRecentLoginAttempts(auth_Results[0].user_id);
19          if (userLoginAttempts.length >= 10) {
20              throw 'Too many attempts. Try to login again in 10 minutes';
21              // Throw above ends try and goes to catch.
22          }
23
```

Found in controller/authController.js in the edited version

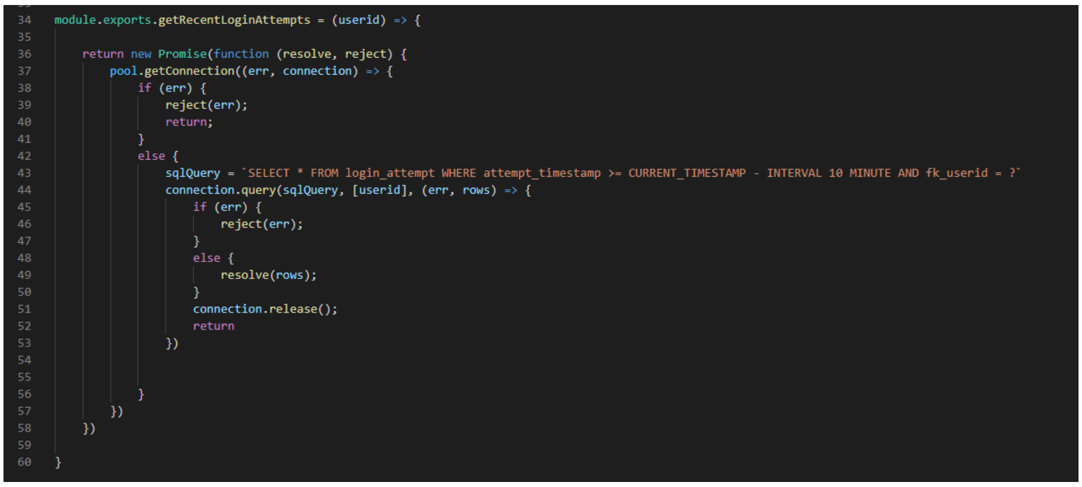
The code/function that records a user's login attempts.



```
module.exports.recordLoginAttempt = (userid) => {

    return new Promise(function (resolve, reject) {
        pool.getConnection((err, connection) => {
            if (err) {
                reject(err);
                return;
            }
            else {
                sqlQuery = `INSERT INTO login_attempt (fk_userid) VALUES (?)`
                connection.query(sqlQuery, [userid], (err, results) => {
                    if (err) {
                        reject(err);
                    }
                    else {
                        console.log("Login Attempt Successfully Recorded");
                        resolve();
                    }
                    connection.release();
                    return;
                })
            }
        }
        )
    })

}
```

Found in middlewares/recordLogin.js in the edited version.

The code/function that gets all of a user’s login attempts in the past 10 minutes.



```javascript
module.exports.getRecentLoginAttempts = (userid) => {

    return new Promise(function (resolve, reject) {
        pool.getConnection((err, connection) => {
            if (err) {
                reject(err);
                return;
            }
            else {
                sqlQuery = `SELECT * FROM login_attempt WHERE attempt_timestamp >= CURRENT_TIMESTAMP - INTERVAL 10 MINUTE AND fk_userid = ?`
                connection.query(sqlQuery, [userid], (err, rows) => {
                    if (err) {
                        reject(err);
                    }
                    else {
                        resolve(rows);
                    }
                    connection.release();
                    return
                })


            }
        })
    })

}
```

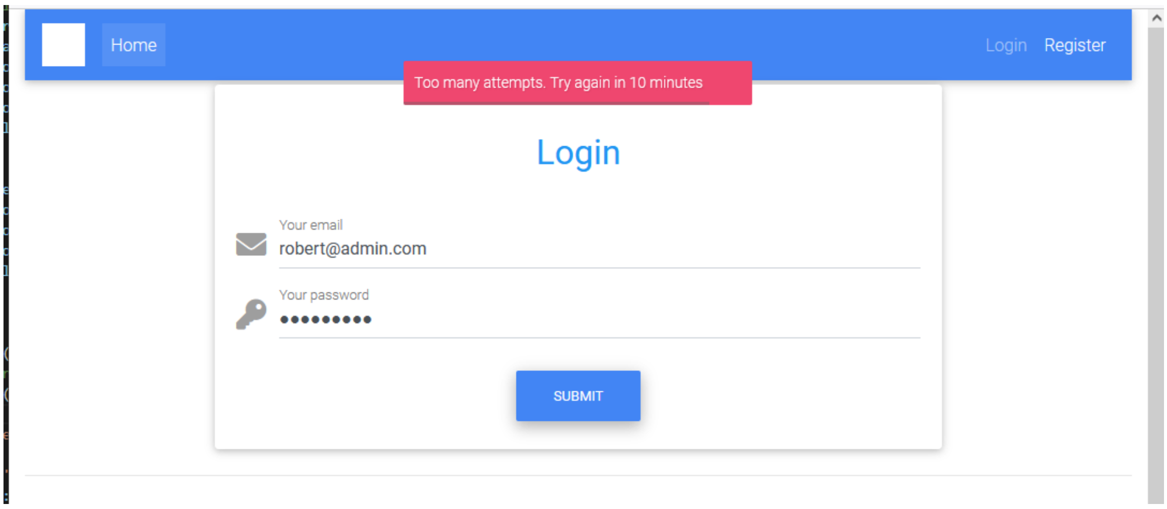
Found in middlewares/recordLogin.js in the edited version.

By having the condition statement (line 19-22 in the first image) which checks that the user hasn’t tried to log in more than 10 times recently, we practically stop all brute force attacks as any attempts to ‘spam’ requests will be immediately rejected.

(Continued Below)

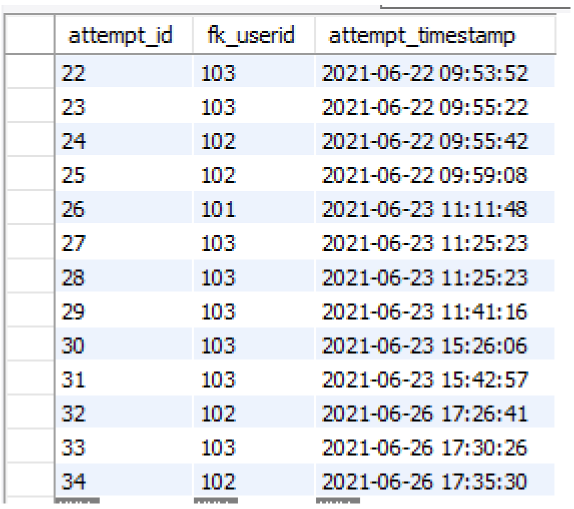
Results:

We can tell that there's now a limit to how many times a user tries to log in because when we attempt to log in many times, the error message below is shown,



Too many attempts. Try again in 10 minutes

If any server administrators want to monitor the login attempts in the future, they can see all previous login attempts in the login_attempt table.



attempt_id	fk_userid	attempt_timestamp
22	103	2021-06-22 09:53:52
23	103	2021-06-22 09:55:22
24	102	2021-06-22 09:55:42
25	102	2021-06-22 09:59:08
26	101	2021-06-23 11:11:48
27	103	2021-06-23 11:25:23
28	103	2021-06-23 11:25:23
29	103	2021-06-23 11:41:16
30	103	2021-06-23 15:26:06
31	103	2021-06-23 15:42:57
32	102	2021-06-26 17:26:41
33	103	2021-06-26 17:30:26
34	102	2021-06-26 17:35:30

3.6.2 Logging and Monitoring for Design Submissions

Impact: High

By using automated tools, attackers can submit many requests inside the design submission page. By submitting enough requests, attackers can slow down or even crash the server. Because it's easy to submit enough requests to crash the server, our team has decided to put the impact as "High".

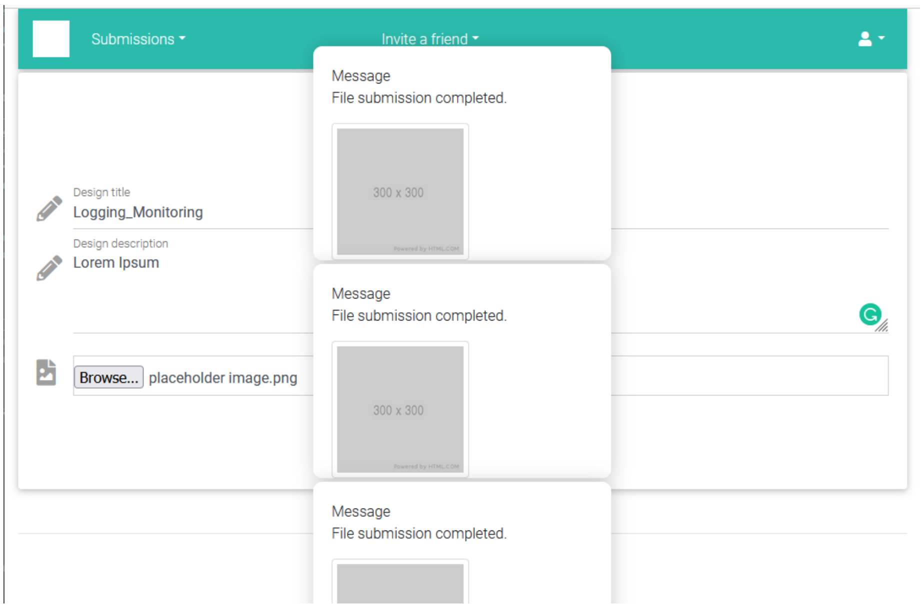
How to exploit flaw:

The design submission is a slow and heavy service.
Not only is the application receiving digitally dense images, but the application is also using an external cloud-based management service, Cloudinary.

Images have to be downloaded and then be uploaded to Cloudinary.
Thus making it slow as it takes time to upload images to Cloudinary and then have to wait for Cloudinary to respond back to say they've successfully saved the submission image without problems.

Only after the response can we tell the user that his submission was successful.

Attackers can exploit this by using automated tools to 'spam' requests onto the server. As long as the attacker can send requests faster than the server can handle one request, the server will hang perpetually.



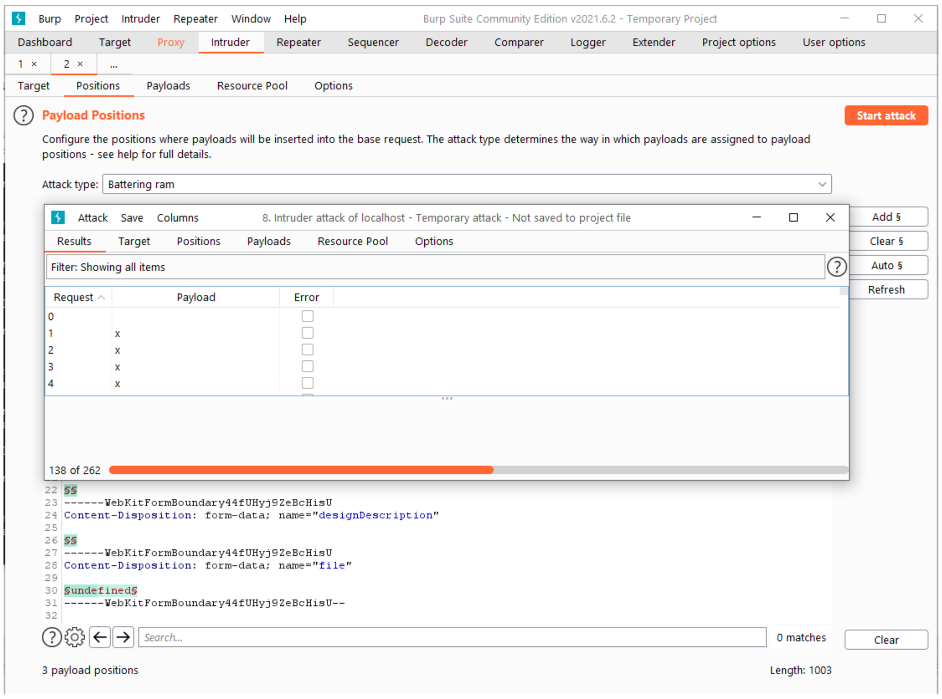
Message
File submission completed.

Message
File submission completed.

Message
File submission completed.

For Example (similar to the ‘request spam’ in the login page):
Using Burp Suite’s Intruder feature to spam requests, we managed to slow down the submission speed from nearly instant (normal submissions) to take as long as 5 seconds to handle each new request.

Note: This is only after the 138th request. The time taken to handle each new request will keep on increasing as the number of ‘spam’ requests increases. At some point, it’ll be practically unusable because of the time it takes to submit new designs. The server crashed after the 212th request in the test run.


Burp Project Intruder Repeater Window Help
Burp Suite Community Edition v2021.6.2 - Temporary Project
Dashboard Target Proxy Intruder Repeater Sequencer Decoder Comparer Logger Extender Project options User options
1 × 2 × ...
Target Positions Payloads Resource Pool Options
Start attack
Attack Save Columns
Results Target Positions Payloads Resource Pool Options
Add § Clear § Auto § Refresh
Search...
Clear

Payload Positions

Configure the positions where payloads will be inserted into the base request. The attack type determines the way in which payloads are assigned to payload positions - see help for full details.

Attack type: Battering ram

8. Intruder attack of localhost - Temporary attack - Not saved to project file

Filter: Showing all items

Request	Payload	Error
0		☐
1	x	☐
2	x	☐
3	x	☐
4	x	☐

138 of 262

```
§§
------WebKitFormBoundary44fUHyj9ZeBcHisU
Content-Disposition: form-data; name="designDescription"

§§
------WebKitFormBoundary44fUHyj9ZeBcHisU
Content-Disposition: form-data; name="file"

§undefined§
------WebKitFormBoundary44fUHyj9ZeBcHisU--

```

0 matches

3 payload positions

Length: 1003

(Continued Below)

Recommendations:

The premise of solving the ‘spamming’ of design submission requests is similar to the solution found on the login page.

Firstly, we need to know that they’ve tried to log in recently.

Create a new table in MySQL for design submission attempts using the settings below.

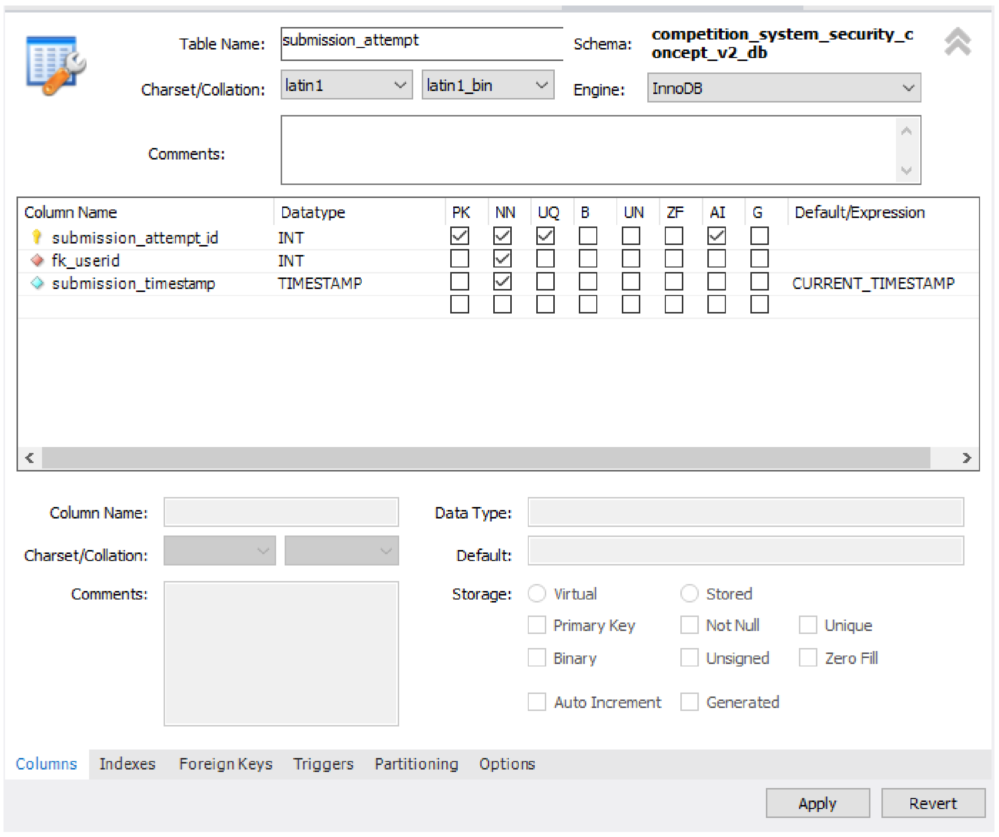


Table Name: submission_attempt

Schema: competition_system_security_concept_v2_db

Charset/Collation: latin1 latin1_bin

Engine: InnoDB

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
submission_attempt_id	INT	✓	✓	✓				✓		
fk_userid	INT		✓							
submission_timestamp	TIMESTAMP		✓							CURRENT_TIMESTAMP

(Continued Below)

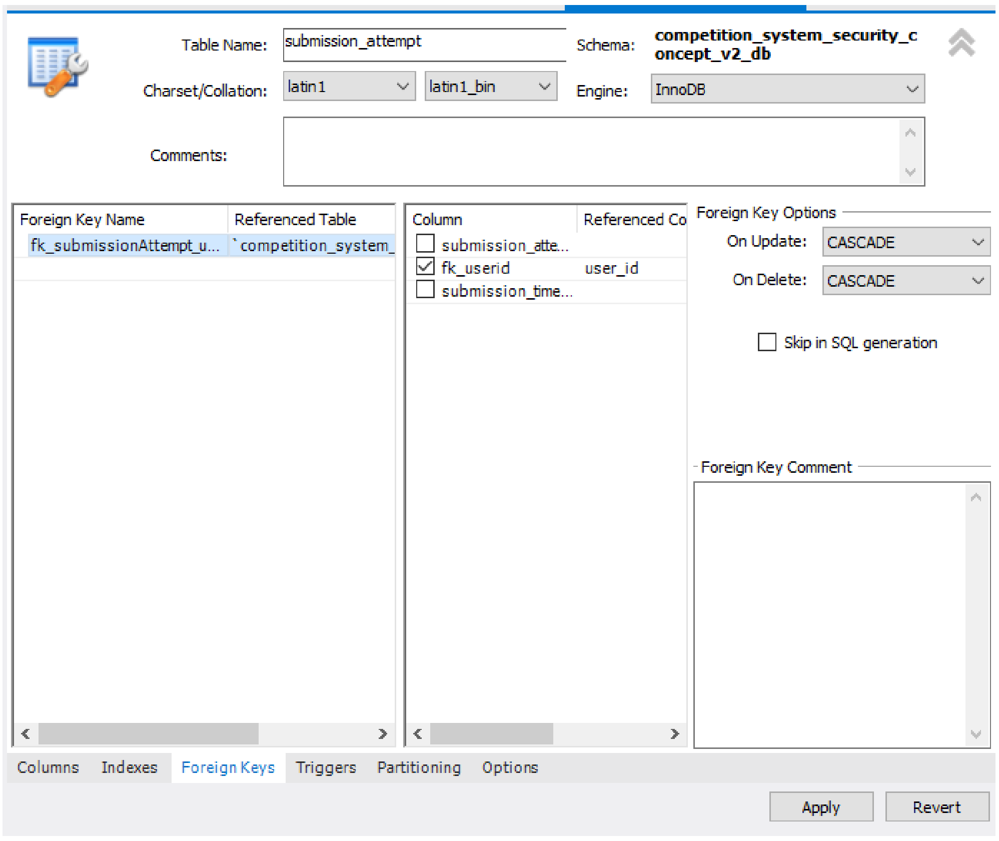


Table Name:	submission_attempt	Schema:	competition_system_security_concept_v2_db
Charset/Collation:	latin1 / latin1_bin	Engine:	InnoDB
Comments:			

Foreign Key Name	Referenced Table
fk_submissionAttempt_u...	`competition_system_

Column	Referenced Co
☐ submission_atte...	
✓ fk_userid	user_id
☐ submission_time...	

Foreign Key Options

On Update: CASCADE

On Delete: CASCADE

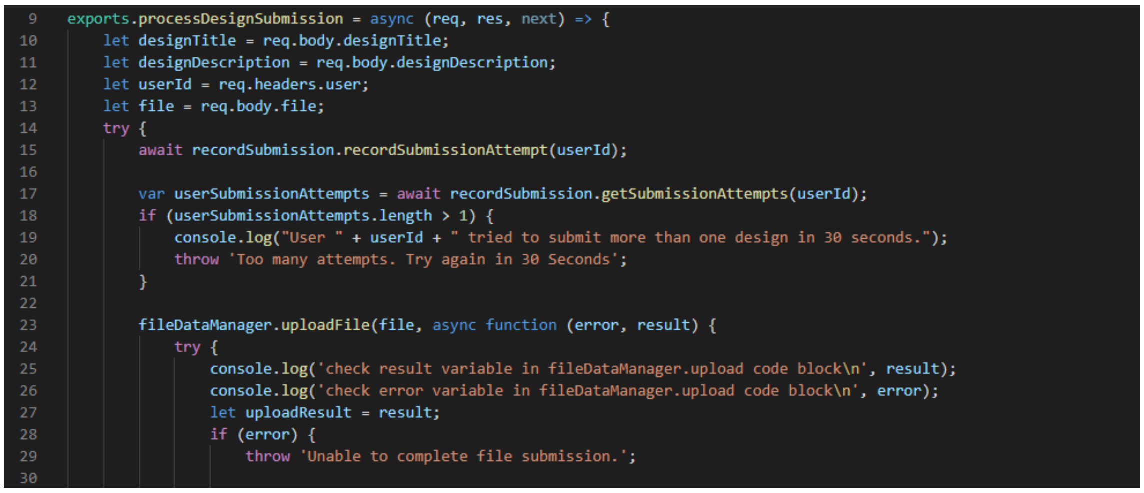
☐ Skip in SQL generation

Foreign Key Comment

Note: The referenced table is the database's ***user*** table.

Afterwards, add two functions that run before the file gets uploaded(line 23), that:

1. Records submission attempts (line 15)
2. Get's users **recent** submission attempts (line 17)
3. If the user has tried to submit more than 1 new design in the timeframe, then discard the submission attempt(line 18 - 21).

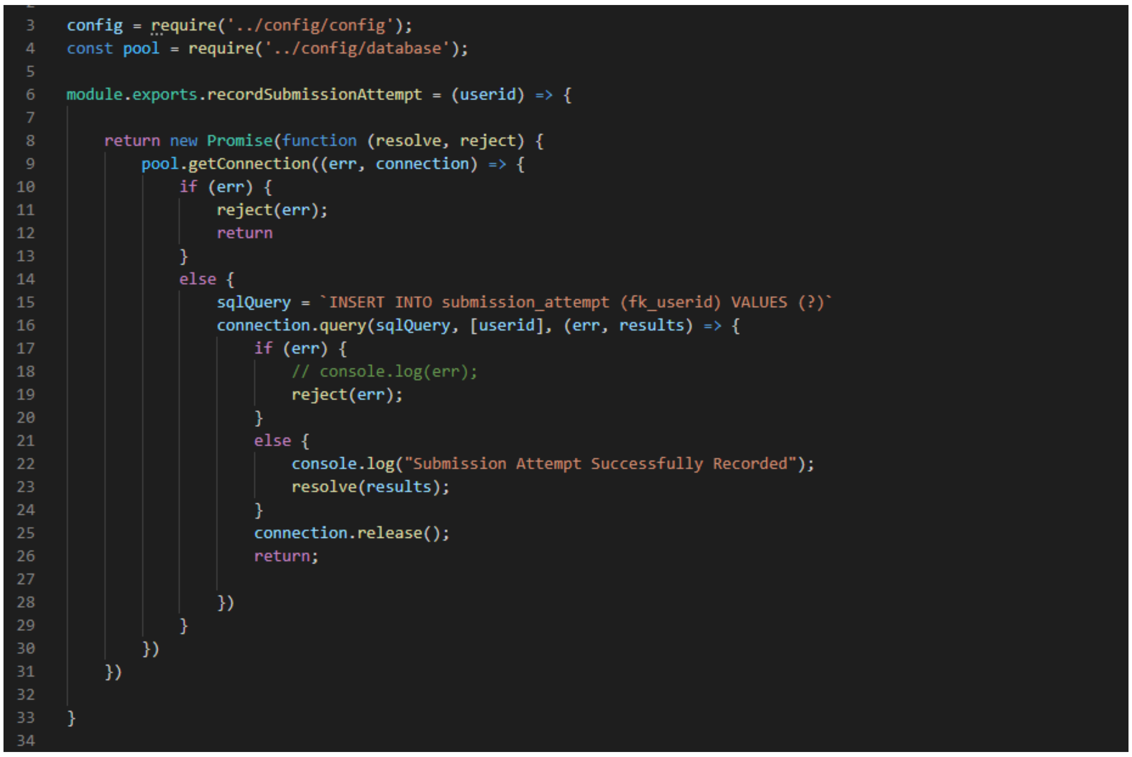


```
 9  exports.processDesignSubmission = async (req, res, next) => {
10      let designTitle = req.body.designTitle;
11      let designDescription = req.body.designDescription;
12      let userId = req.headers.user;
13      let file = req.body.file;
14      try {
15          await recordSubmission.recordSubmissionAttempt(userId);
16
17          var userSubmissionAttempts = await recordSubmission.getSubmissionAttempts(userId);
18          if (userSubmissionAttempts.length > 1) {
19              console.log("User " + userId + " tried to submit more than one design in 30 seconds.");
20              throw 'Too many attempts. Try again in 30 Seconds';
21          }
22
23          fileDataManager.uploadFile(file, async function (error, result) {
24              try {
25                  console.log('check result variable in fileDataManager.upload code block\n', result);
26                  console.log('check error variable in fileDataManager.upload code block\n', error);
27                  let uploadResult = result;
28                  if (error) {
29                      throw 'Unable to complete file submission.';
30
```

Found in controllers/userController.js in the edited version.

(Continued Below)

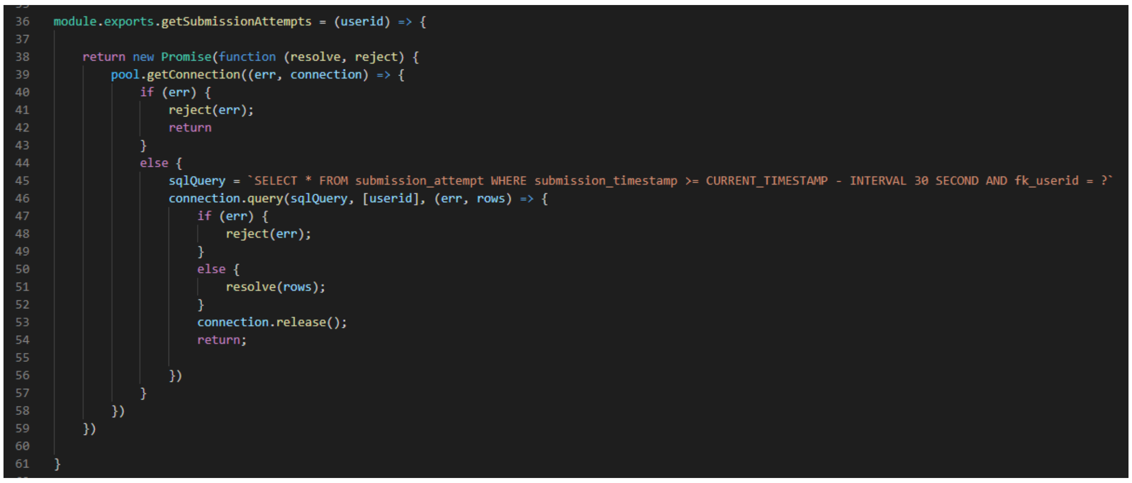
The function/code which **logs** the user's submission attempts.



```
3   config = require('../config/config');
4   const pool = require('../config/database');
5
6   module.exports.recordSubmissionAttempt = (userid) => {
7
8       return new Promise(function (resolve, reject) {
9           pool.getConnection((err, connection) => {
10              if (err) {
11                  reject(err);
12                  return
13              }
14              else {
15                  sqlQuery = `INSERT INTO submission_attempt (fk_userid) VALUES (?)`
16                  connection.query(sqlQuery, [userid], (err, results) => {
17                      if (err) {
18                          // console.log(err);
19                          reject(err);
20                      }
21                      else {
22                          console.log("Submission Attempt Successfully Recorded");
23                          resolve(results);
24                      }
25                      connection.release();
26                      return;
27
28                  })
29              }
30          })
31      })
32
33  }
34
```

Found in middlewares/recordSubmission.js in the edited version.

The function/code which **gets** the user's recent submission attempts.



```
36  module.exports.getSubmissionAttempts = (userid) => {
37
38      return new Promise(function (resolve, reject) {
39          pool.getConnection((err, connection) => {
40              if (err) {
41                  reject(err);
42                  return
43              }
44              else {
45                  sqlQuery = `SELECT * FROM submission_attempt WHERE submission_timestamp >= CURRENT_TIMESTAMP - INTERVAL 30 SECOND AND fk_userid = ?`
46                  connection.query(sqlQuery, [userid], (err, rows) => {
47                      if (err) {
48                          reject(err);
49                      }
50                      else {
51                          resolve(rows);
52                      }
53                      connection.release();
54                      return;
55
56                  })
57              }
58          })
59      })
60
61  }
```

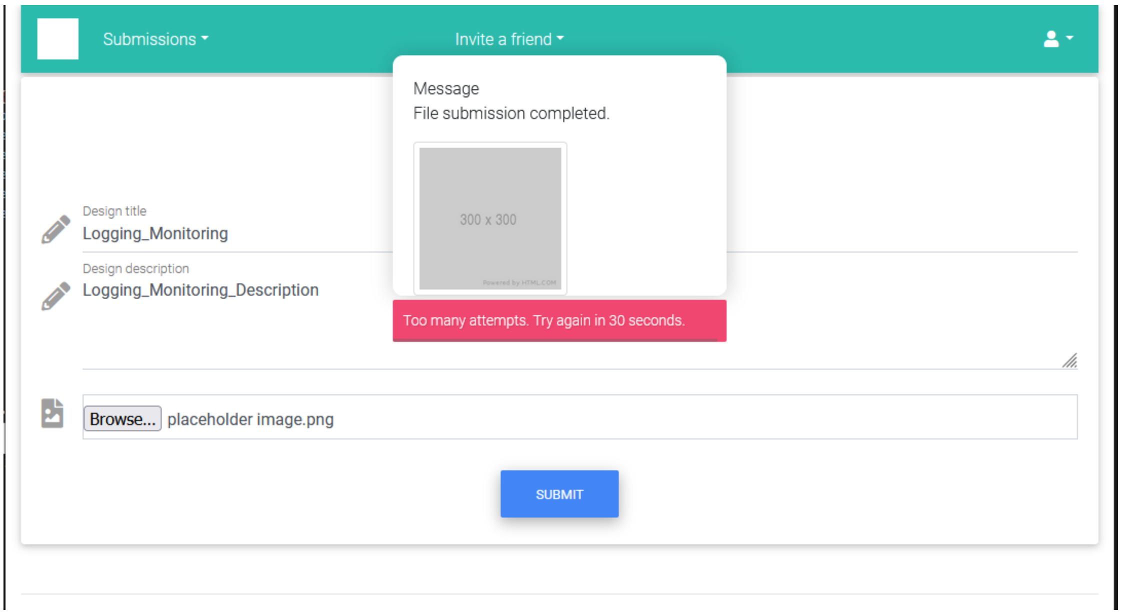
Found in middlewares/recordLogin.js in the edited version.

By having the condition statement (line 18 - 21 in the first image) which checks that the user hasn't tried to log in more than 10 times recently, we practically stop all brute force attacks as any attempts to 'spam' requests will be immediately rejected.

Results:

We can tell that there's now a limit to how many times a user tries to submit designs because when we attempt to submit a design too many times, the error message below is shown.

"Too many attempts. Try again in 30 seconds"



Message
File submission completed.

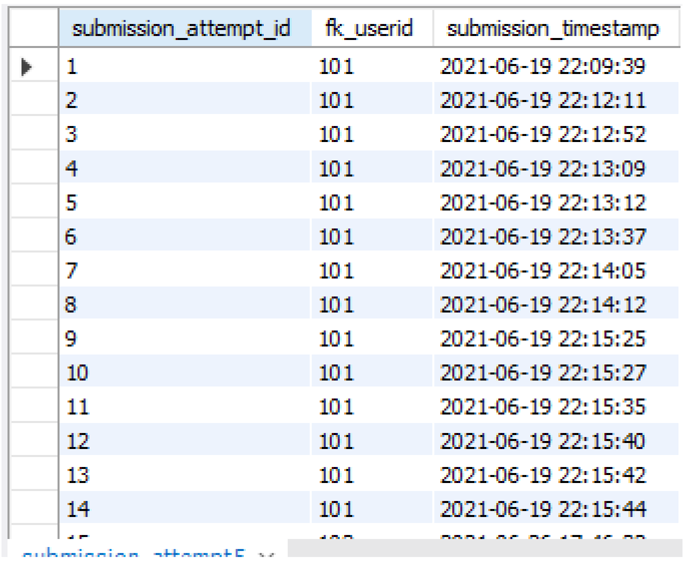
Logging_Monitoring

Logging_Monitoring_Description

placeholder image.png

Too many attempts. Try again in 30 seconds.

If any server administrators want to monitor the design submission attempts in the future, they can see all previous submission attempts in the submission_attempt table.



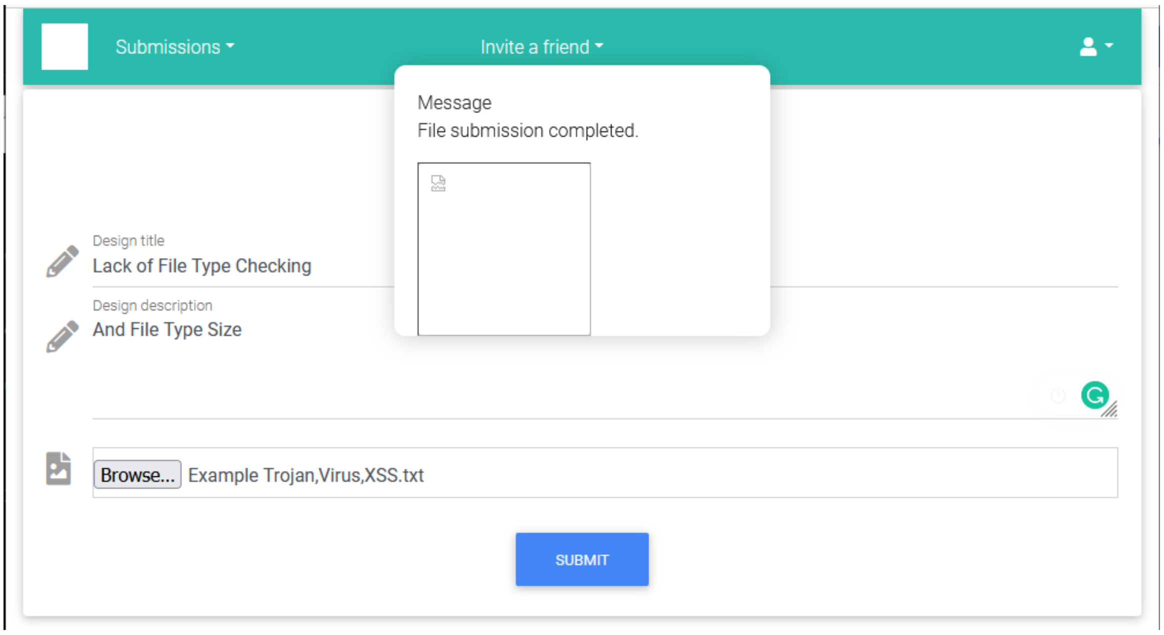
submission_attempt_id	fk_userid	submission_timestamp
1	101	2021-06-19 22:09:39
2	101	2021-06-19 22:12:11
3	101	2021-06-19 22:12:52
4	101	2021-06-19 22:13:09
5	101	2021-06-19 22:13:12
6	101	2021-06-19 22:13:37
7	101	2021-06-19 22:14:05
8	101	2021-06-19 22:14:12
9	101	2021-06-19 22:15:25
10	101	2021-06-19 22:15:27
11	101	2021-06-19 22:15:35
12	101	2021-06-19 22:15:40
13	101	2021-06-19 22:15:42
14	101	2021-06-19 22:15:44

3.6.3 Design Submission Functionality Allows for Potentially Dangerous File Types and Files of Ludicrous Sizes

Impact: Medium

Currently, there's no indication of file type filtering or checking to ensure that colossal files don't get accepted into the server. Because there's no check, attackers can send trojans, viruses and canned XSS exploits which might break the server or harm users. Furthermore, attackers can also send big files which would take a long period of time to download and upload (to Cloudinary). Therefore, slowing the server down. The team however puts this impact at a "Medium" as attackers would need to have a considerable amount of knowledge about the subject and the application server in order to do harm.

How it can be exploited:



Message
File submission completed.

Design title
Lack of File Type Checking

Design description
And File Type Size

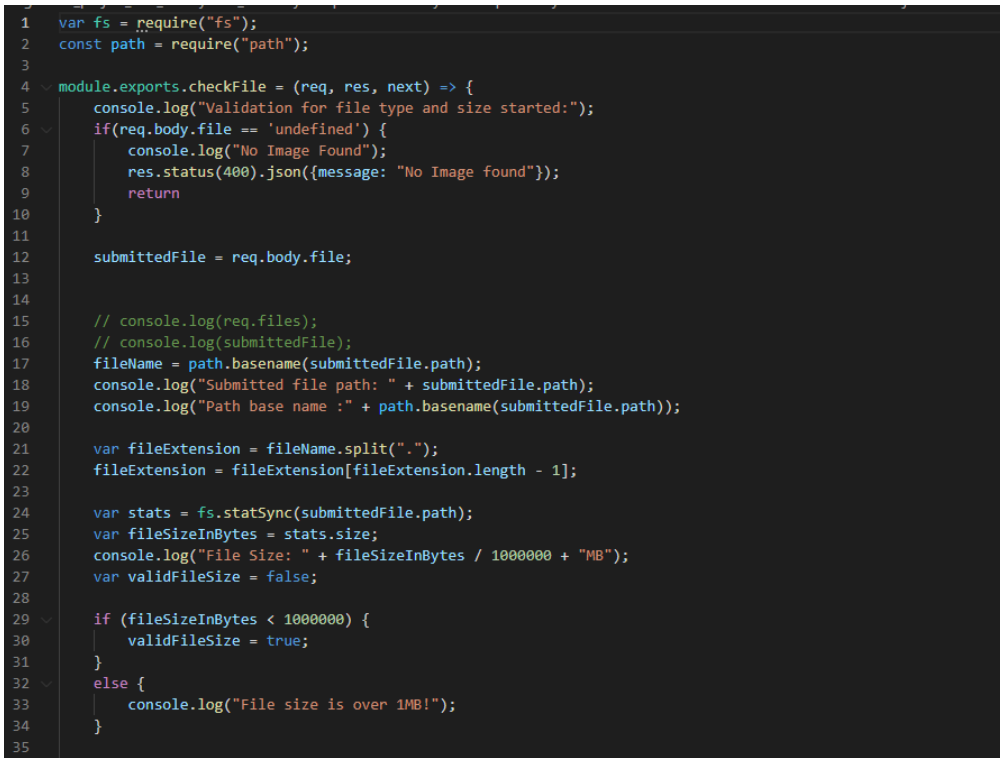
Example Trojan,Virus,XSS.txt

As seen above, we can upload .txt files on the design submission page even though it is clearly not an image type. This is dangerous as attackers can exploit this by passing in custom trojans, viruses and canned cross-site scripts.

Recommendation:

We’ve decided that a middleware function that checks the files extension and size is the best course of action.

The first half of the function checks that the file size is below 10MB.
Line 24 - 31

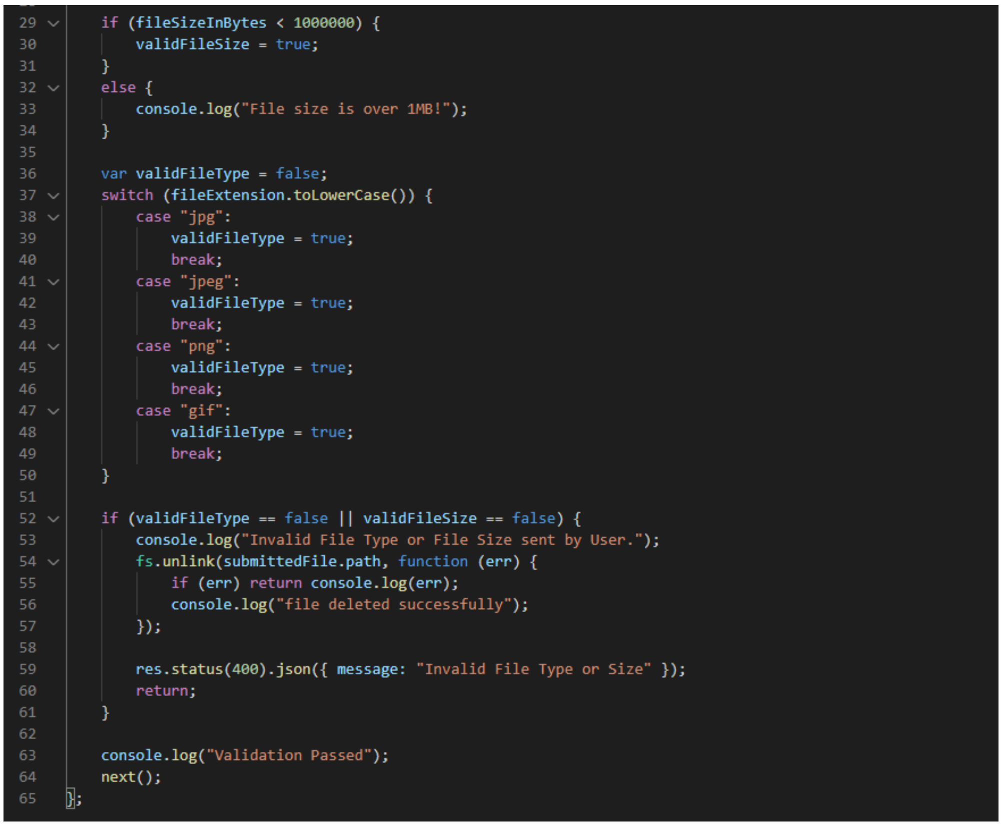


```
1   var fs = require("fs");
2   const path = require("path");
3
4   module.exports.checkFile = (req, res, next) => {
5       console.log("Validation for file type and size started:");
6       if(req.body.file == 'undefined') {
7           console.log("No Image Found");
8           res.status(400).json({message: "No Image found"});
9           return
10      }
11
12      submittedFile = req.body.file;
13
14
15      // console.log(req.files);
16      // console.log(submittedFile);
17      fileName = path.basename(submittedFile.path);
18      console.log("Submitted file path: " + submittedFile.path);
19      console.log("Path base name :" + path.basename(submittedFile.path));
20
21      var fileExtension = fileName.split(".");
22      fileExtension = fileExtension[fileExtension.length - 1];
23
24      var stats = fs.statSync(submittedFile.path);
25      var fileSizeInBytes = stats.size;
26      console.log("File Size: " + fileSizeInBytes / 1000000 + "MB");
27      var validFileSize = false;
28
29      if (fileSizeInBytes < 1000000) {
30          validFileSize = true;
31      }
32      else {
33          console.log("File size is over 1MB!");
34      }
35
```

Found in middlewares/checkSubmissionFile.js in the edited version.

(Continued Below)

The second half of the function checks whether the submitted file is of an image file type (e.g. .jpg, .jpeg, .png, .gif). Line 36 - 50.

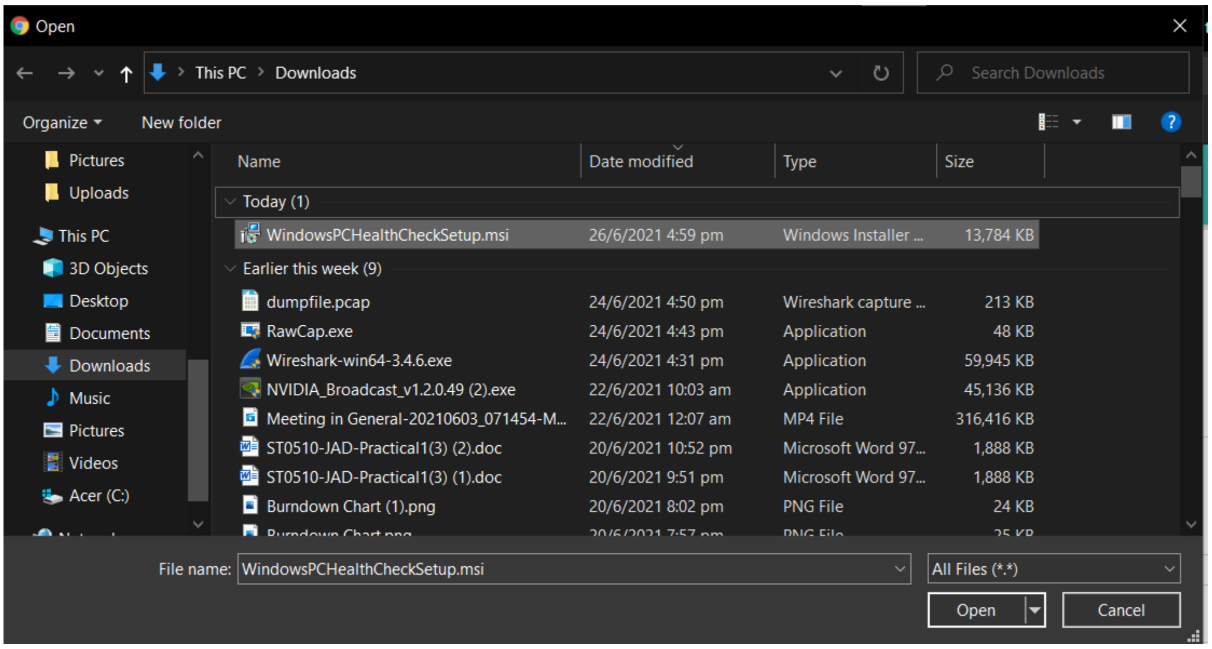


```
29      if (fileSizeInBytes < 1000000) {
30          validFileSize = true;
31      }
32      else {
33          console.log("File size is over 1MB!");
34      }
35
36      var validFileType = false;
37      switch (fileExtension.toLowerCase()) {
38          case "jpg":
39              validFileType = true;
40              break;
41          case "jpeg":
42              validFileType = true;
43              break;
44          case "png":
45              validFileType = true;
46              break;
47          case "gif":
48              validFileType = true;
49              break;
50      }
51
52      if (validFileType == false || validFileSize == false) {
53          console.log("Invalid File Type or File Size sent by User.");
54          fs.unlink(submittedFile.path, function (err) {
55              if (err) return console.log(err);
56              console.log("file deleted successfully");
57          });
58
59          res.status(400).json({ message: "Invalid File Type or Size" });
60          return;
61      }
62
63      console.log("Validation Passed");
64      next();
65  };
```

Found in middlewares/checkSubmissionFile.js in the edited version.

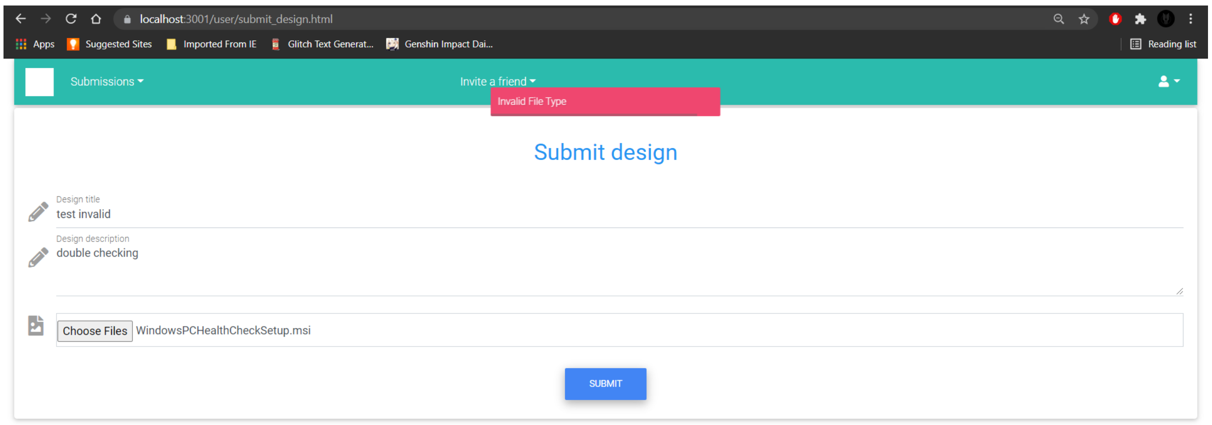
Line 52 - 61, deletes the file from the server if the submitted file is of the wrong file type or too big. Afterwards, the server immediately returns a response. Thus the upload is not being executed.

If we try to submit an invalid file now…



Name	Date modified	Type	Size
Today (1)			
WindowsPCHealthCheckSetup.msi	26/6/2021 4:59 pm	Windows Installer ...	13,784 KB
Earlier this week (9)			
dumpfile.pcap	24/6/2021 4:50 pm	Wireshark capture ...	213 KB
RawCap.exe	24/6/2021 4:43 pm	Application	48 KB
Wireshark-win64-3.4.6.exe	24/6/2021 4:31 pm	Application	59,945 KB
NVIDIA_Broadcast_v1.2.0.49 (2).exe	22/6/2021 10:03 am	Application	45,136 KB
Meeting in General-20210603_071454-M...	22/6/2021 12:07 am	MP4 File	316,416 KB
ST0510-JAD-Practical1(3) (2).doc	20/6/2021 10:52 pm	Microsoft Word 97...	1,888 KB
ST0510-JAD-Practical1(3) (1).doc	20/6/2021 9:51 pm	Microsoft Word 97...	1,888 KB
Burndown Chart (1).png	20/6/2021 8:02 pm	PNG File	24 KB

File name: WindowsPCHealthCheckSetup.msi
All Files (*.*)


localhost:3001/user/submit_design.html

Invalid File Type

Submit design

Design title
test invalid

Design description
double checking

Choose Files WindowsPCHealthCheckSetup.msi

The submission is rejected because it has an invalid file type.

4. Preventing Future Vulnerabilities

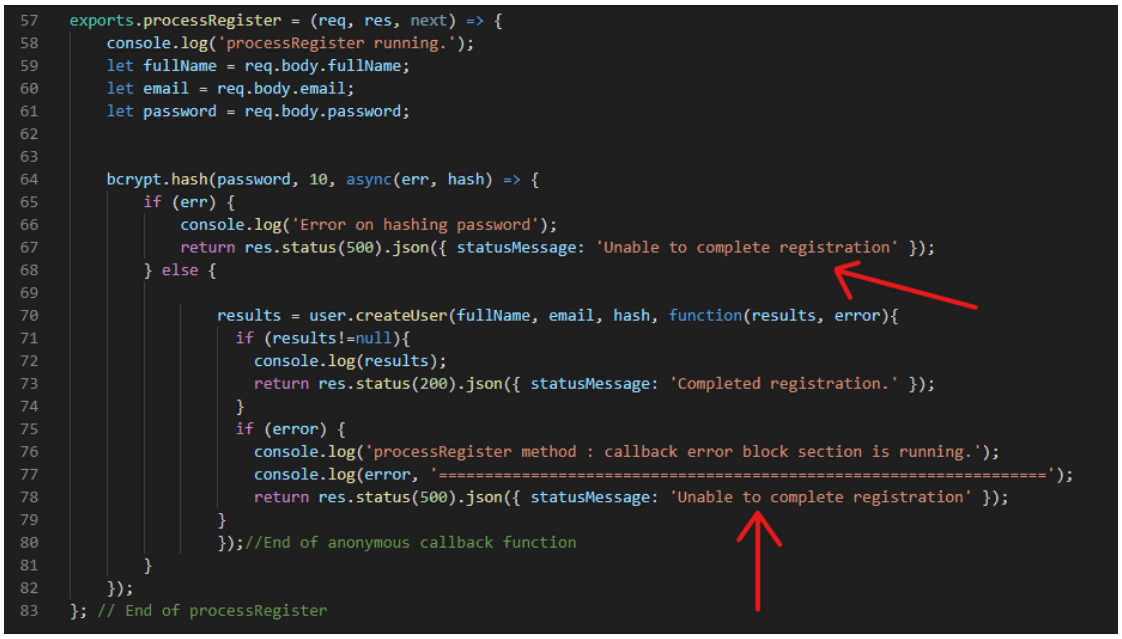
Most security vulnerabilities can be easily spotted through code walkthrough or testing. So why are they so prevalent? In our modern world, everyone is chasing deadlines and everyone wants the new thing. Therefore, we understand the need for ‘more features’ and the lack of time for quality assurance (QA). But, these are also the common factors that lead to security vulnerabilities. Therefore, we have come up with a list of things your developers can do to help spot vulnerabilities easier.

4.1 Error Handling Practices and Nested Callbacks

The application, at this moment, contains many ways to handle errors.

Some endpoints handle the errors immediately at the point of error through the use of:

return res.status(500).json({ statusMessage: ‘Unable to complete registration’ });

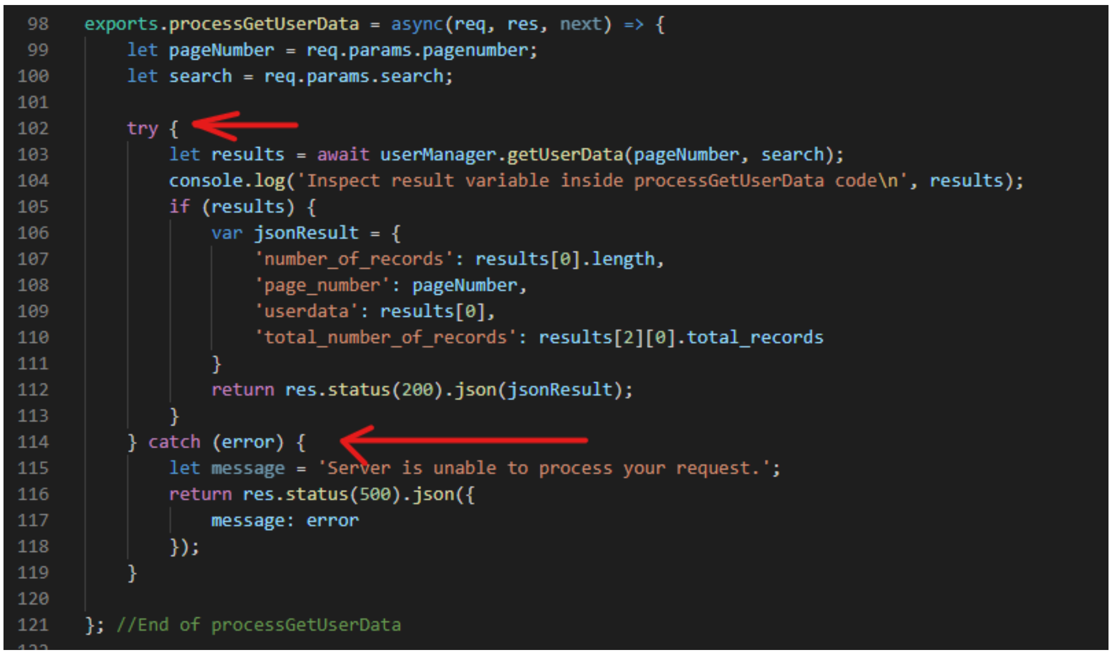


```javascript
exports.processRegister = (req, res, next) => {
    console.log('processRegister running.');
    let fullName = req.body.fullName;
    let email = req.body.email;
    let password = req.body.password;


    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
```

Found in controllers/authController.js in the original version.

(Continued Below)

While other endpoints handle the errors at the end through the use of try/catch:



```
exports.processGetUserData = async(req, res, next) => {
    let pageNumber = req.params.pagenumber;
    let search = req.params.search;

    try {
        let results = await userManager.getUserData(pageNumber, search);
        console.log('Inspect result variable inside processGetUserData code\n', results);
        if (results) {
            var jsonResult = {
                'number_of_records': results[0].length,
                'page_number': pageNumber,
                'userdata': 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 processGetUserData
```

Found in controller/userController.js in the original version.

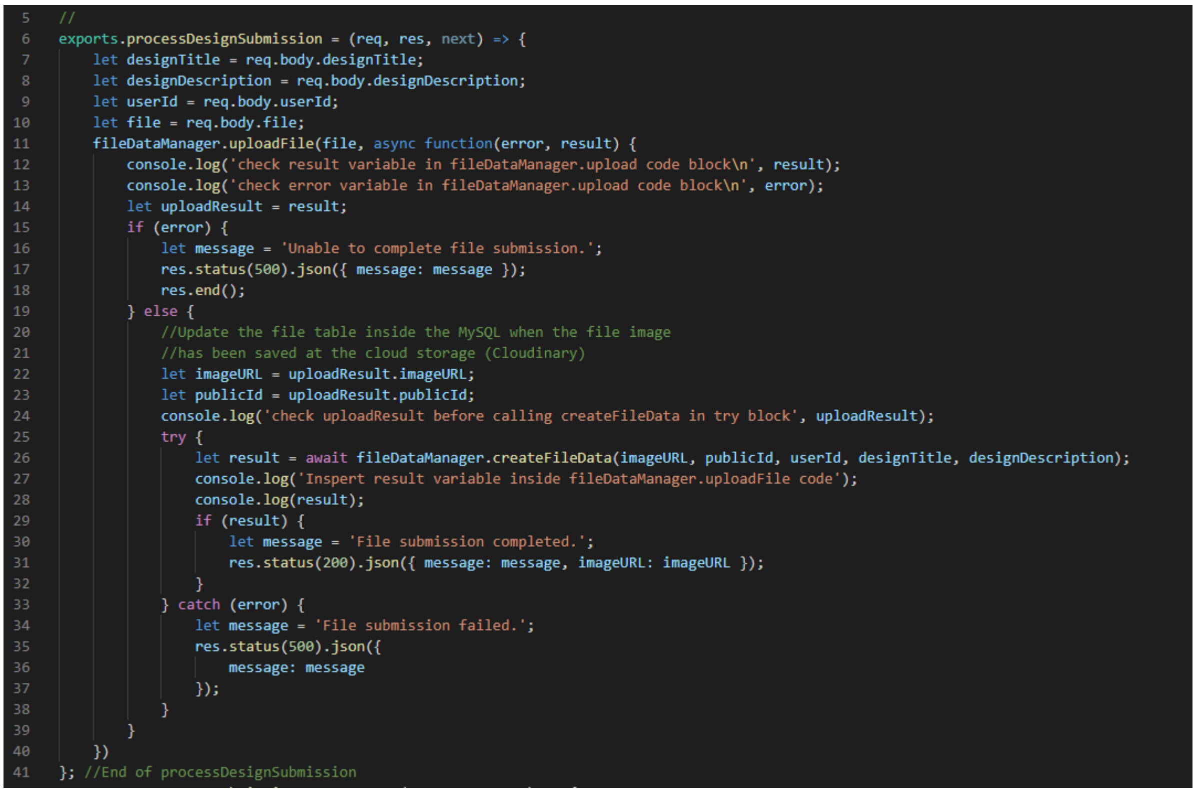
Firstly we highly suggest the usage of the try/catch method in the second picture.

Now you may ask, “Why?”.

It is easier to handle errors when **all errors resolve at the bottom**.
This is because when developers have to inevitably check for vulnerabilities in the future, it’s easier and faster to trace the code as the function concludes at the bottom.

(Continued Below)

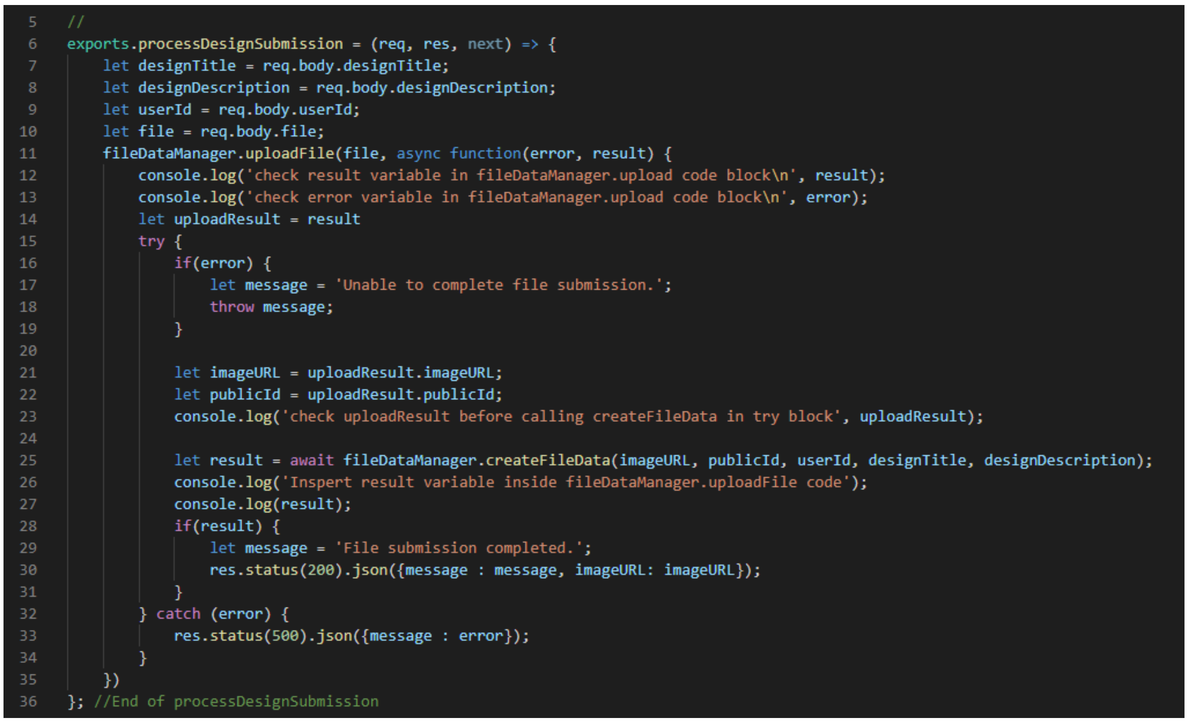
Another example is this:
Compare this code which **uses nested callbacks** and handles **errors on the spot**:



```
//
exports.processDesignSubmission = (req, res, next) => {
    let designTitle = req.body.designTitle;
    let designDescription = req.body.designDescription;
    let userId = req.body.userId;
    let file = req.body.file;
    fileDataManager.uploadFile(file, async function(error, result) {
        console.log('check result variable in fileDataManager.upload code block\n', result);
        console.log('check error variable in fileDataManager.upload code block\n', error);
        let uploadResult = result;
        if (error) {
            let message = 'Unable to complete file submission.';
            res.status(500).json({ message: message });
            res.end();
        } else {
            //Update the file table inside the MySQL when the file image
            //has been saved at the cloud storage (Cloudinary)
            let imageURL = uploadResult.imageURL;
            let publicId = uploadResult.publicId;
            console.log('check uploadResult before calling createFileData in try block', uploadResult);
            try {
                let result = await fileDataManager.createFileData(imageURL, publicId, userId, designTitle, designDescription);
                console.log('Inspert result variable inside fileDataManager.uploadFile code');
                console.log(result);
                if (result) {
                    let message = 'File submission completed.';
                    res.status(200).json({ message: message, imageURL: imageURL });
                }
            } catch (error) {
                let message = 'File submission failed.';
                res.status(500).json({
                    message: message
                });
            }
        }
    })
}; //End of processDesignSubmission
```

Found in controllers/userController.js in the original version

Against this code which uses the **try/catch** method and handles **errors at the end**:



```
//
exports.processDesignSubmission = (req, res, next) => {
    let designTitle = req.body.designTitle;
    let designDescription = req.body.designDescription;
    let userId = req.body.userId;
    let file = req.body.file;
    fileDataManager.uploadFile(file, async function(error, result) {
        console.log('check result variable in fileDataManager.upload code block\n', result);
        console.log('check error variable in fileDataManager.upload code block\n', error);
        let uploadResult = result
        try {
            if(error) {
                let message = 'Unable to complete file submission.';
                throw message;
            }

            let imageURL = uploadResult.imageURL;
            let publicId = uploadResult.publicId;
            console.log('check uploadResult before calling createFileData in try block', uploadResult);

            let result = await fileDataManager.createFileData(imageURL, publicId, userId, designTitle, designDescription);
            console.log('Inspert result variable inside fileDataManager.uploadFile code');
            console.log(result);
            if(result) {
                let message = 'File submission completed.';
                res.status(200).json({message : message, imageURL: imageURL});
            }
        } catch (error) {
            res.status(500).json({message : error});
        }
    })
}; //End of processDesignSubmission
```

Found in controllers/userController.js in the edited version

(Continued Below)

The code structure on the first image looks like a pyramid, making it difficult to read and maintain. This structure is what developers call callback hell. And this problem will only get exacerbated as more features get added and more callbacks are used.

By making it more difficult to read, it becomes much harder to comprehend what the code is trying to achieve. This is dangerous as when developers don't fully understand the code, they can't tell whether it's going to produce a security vulnerability.

Now, compare it with the second image which looks more synchronous and thus less 'clever' and more 'readable'. Not only that, but the try/catch method also makes it easier to check for security vulnerabilities as developers only need to look at the bottom of the function where **all of the server's responses (and therefore response data/body)** are sent. This structure not only makes it much easier to comprehend the code's logic but also makes it easy to find where the response data, information that is sent back to the user, is sent.
The response data is important as most of the time, it is where attackers can 'clue' in or guess the server's vulnerabilities.

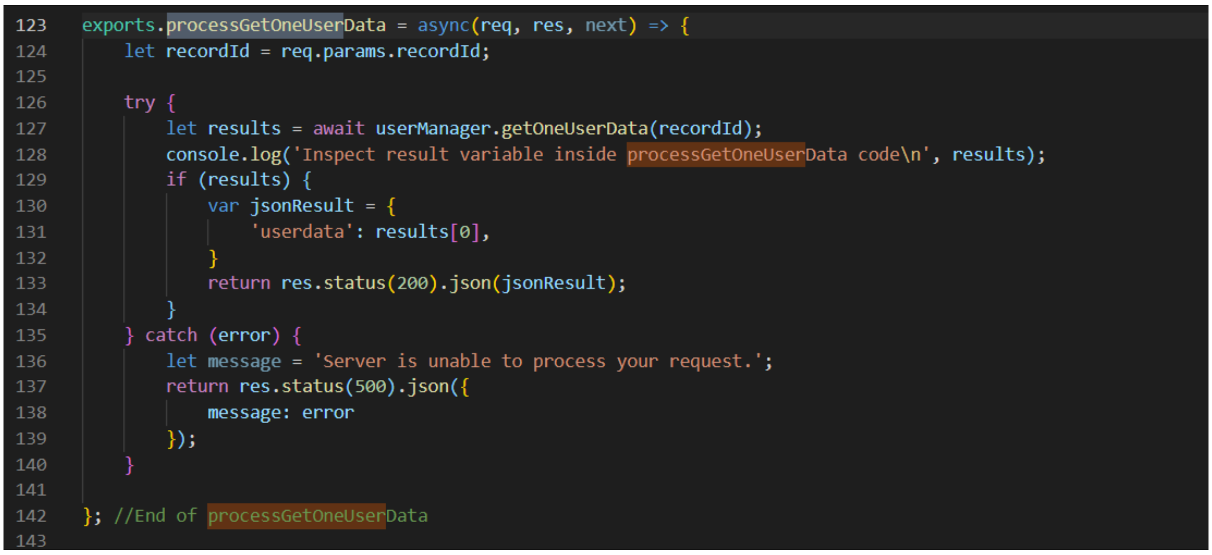
Hence, our team strongly recommends your developers refactor the code to use the try/catch (and promises) format to lessen the probabilities of future security vulnerabilities.

Note: We've refactored most of the code inside the authController.js and userController.js to use the try/catch and promises format. Please take a look for future reference.

4.2 Standardised JSON Response Data

The application contains many different ways of sending back JSON Response Data at the moment.

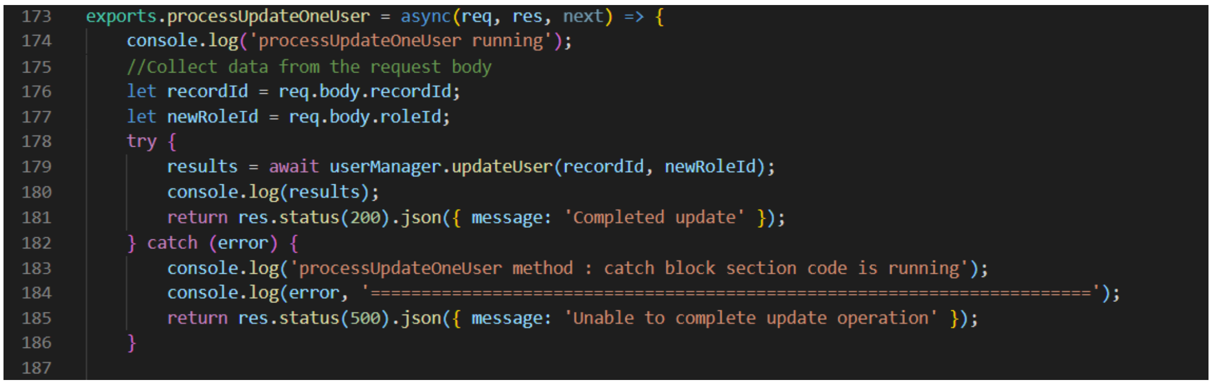
In the userController.js file, the processGetOneDesignData function is using a declared variable to send back the Response data, such as jsonResult in line 133 and error in line 138. Even though the error message in line 136 was declared, it was never used.



```
123  exports.processGetOneUserData = async(req, res, next) => {
124      let recordId = req.params.recordId;
125  
126      try {
127          let results = await userManager.getOneUserData(recordId);
128          console.log('Inspect result variable inside processGetOneUserData code\n', results);
129          if (results) {
130              var jsonResult = {
131                  'userdata': results[0],
132              }
133              return res.status(200).json(jsonResult);
134          }
135      } catch (error) {
136          let message = 'Server is unable to process your request.';
137          return res.status(500).json({
138              message: error
139          });
140      }
141  
142  }; //End of processGetOneUserData
143  
```

Found in /src/controllers/userController.js in the original code.

However, in the same file, the return statement inside the try block differs from that of the above image, which saves an entire JSON object into a variable.

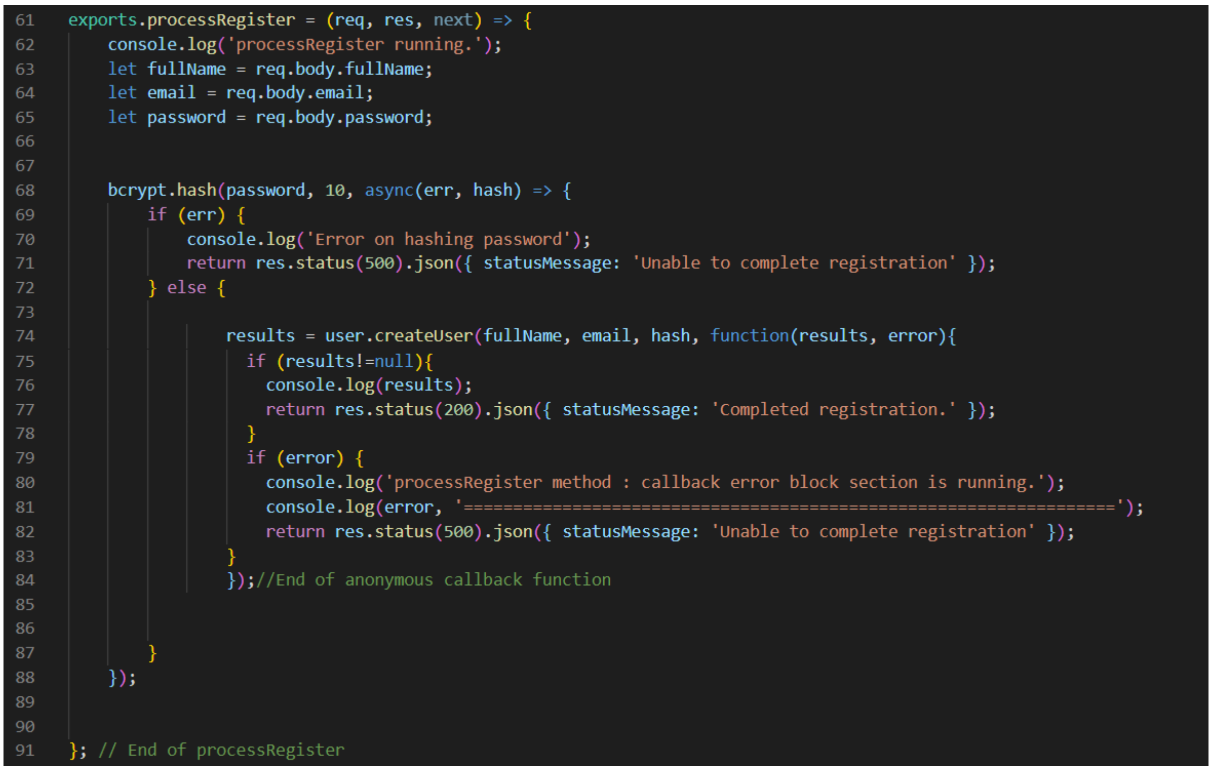


```
173  exports.processUpdateOneUser = async(req, res, next) => {
174      console.log('processUpdateOneUser running');
175      //Collect data from the request body
176      let recordId = req.body.recordId;
177      let newRoleId = req.body.roleId;
178      try {
179          results = await userManager.updateUser(recordId, newRoleId);
180          console.log(results);
181          return res.status(200).json({ message: 'Completed update' });
182      } catch (error) {
183          console.log('processUpdateOneUser method : catch block section code is running');
184          console.log(error, '=======================================================================');
185          return res.status(500).json({ message: 'Unable to complete update operation' });
186      }
187  
```

Found in /src/controllers/userController.js

In the processUpdateOneUser function shown above, the JSON object is created while returning the statement in line 181, whilst the processGetOneDesignData function stores the JSON object to be returned inside a variable before returning it.

authController.js also uses different standards when it comes to returning JSON Response Data.



```
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
```

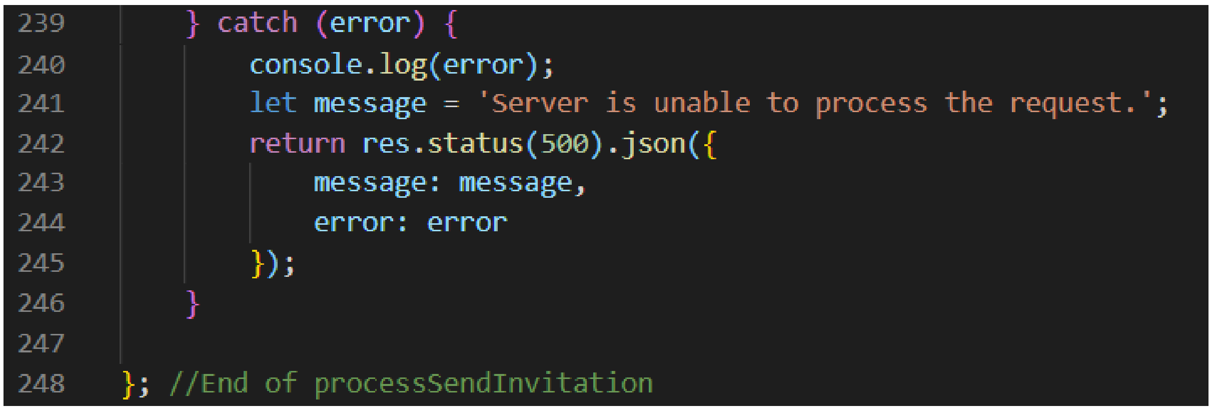
Found in /src/controllers/authController.js in the original code.

Note that the return statements in line 77 and 82 have their message typed out instead of using a variable to store it, which is different from userController.js.

Furthermore, the key used in the JSON Response Data in userController.js is Message, while the key used in the JSON Response Data in authController.js is statusMessage.
This could lead to confusion among the code developers as they might be trying to get the value of statusMessage in authController.js, but because they keep seeing Message being used in userContoller.js so often they use that key instead and that will lead to issues.
We know this may sound confusing, so below is an example of this in a real-world situation:
A person is trying to get the address of a place he has never been to before named statusMessage in his country. He searched the internet for details of this place but a place named Message was the most seen result and he assumed that Message was the right address.

The team recommends standardising all JSON Response Data as it is a good coding practice and it will help to reduce the chance of the mistakes mentioned above.

Below is the JSON Response Data standard the IT Department Team suggests:



```
239        } catch (error) {
240            console.log(error);
241            let message = 'Server is unable to process the request.';
242            return res.status(500).json({
243                message: message,
244                error: error
245            });
246        }
247
248    }; //End of processSendInvitation
```

Found in /src/controller/userController.js

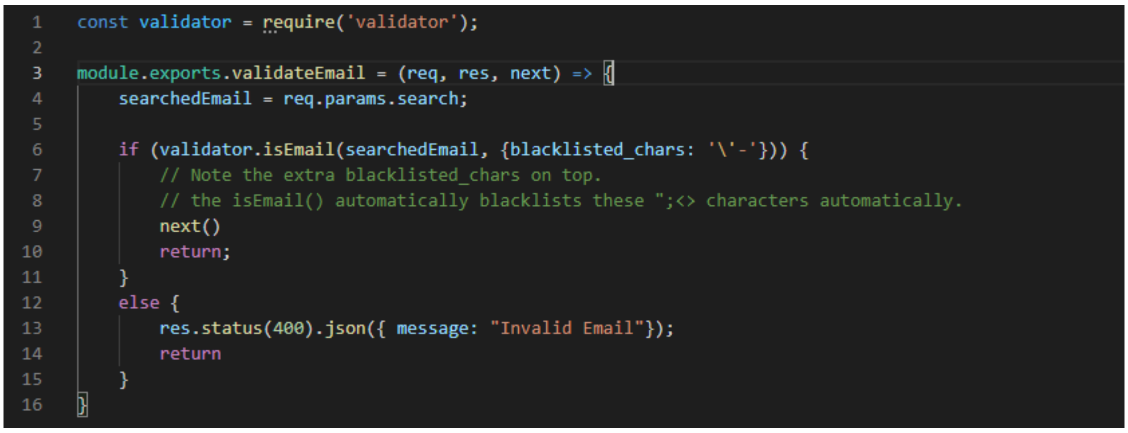
The standard that the IT Department Team has come up with is to store the items to be returned into a variable (error and message in this case) and then put it into a json object in the return statement, as seen in lines 243 and 244.

4.3 Reusable Middleware Functions

We have also accumulated a set of middleware functions that your developers can use to help check user's inputs in the future. These functions will not only help protect your application from security vulnerabilities but also will help expedite your development process through their reusability.

Before we begin, we highly suggest looking into all of the middleware functions we've provided inside the src/middlewares folder so that you can understand and use the middleware which is right for the situation.
Note: We've already applied our middleware functions where appropriate. To see how to use middleware functions and where we used them, please see section 4.3.1 below.

middlewares/checkSearchEmail.js



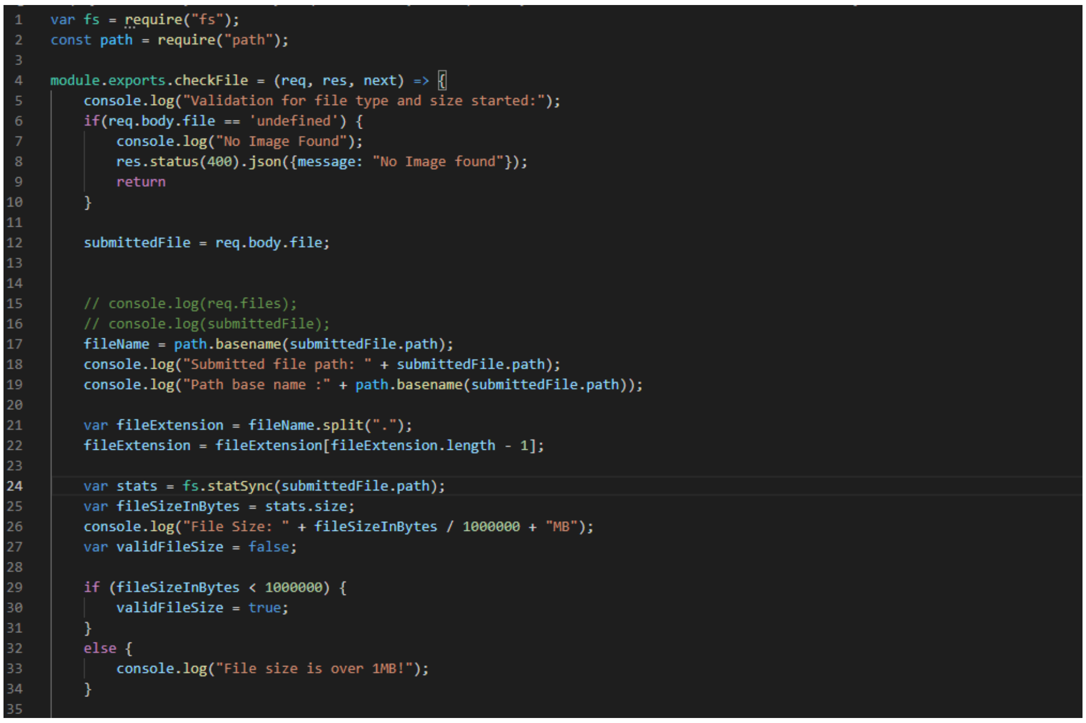
```
const validator = require('validator');

module.exports.validateEmail = (req, res, next) => {
    searchedEmail = req.params.search;

    if (validator.isEmail(searchedEmail, {blacklisted_chars: '\'-'})) {
        // Note the extra blacklisted_chars on top.
        // the isEmail() automatically blacklists these ";<> characters automatically.
        next()
        return;
    }
    else {
        res.status(400).json({ message: "Invalid Email"});
        return
    }
}
```

The function above checks whether the input is an email, is empty, or only contains spaces. If it fails the check, then it stops the request and returns to users a message that says they've inputted an invalid email. Simply replace the searchedEmail variable to use the function.

middlewares/checkSubmissionFile.js

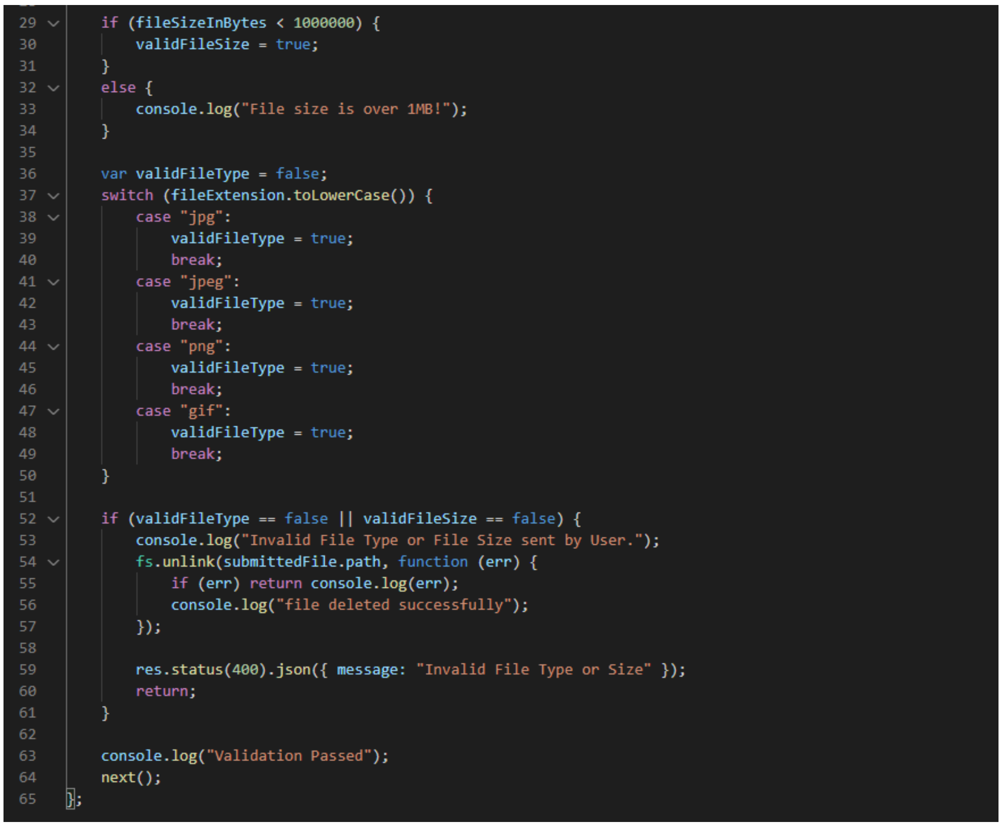
The first half of the function checks that the file size is below 10MB.

Line 24 - 31



```
1   var fs = require("fs");
2   const path = require("path");
3
4   module.exports.checkFile = (req, res, next) => {
5       console.log("Validation for file type and size started:");
6       if(req.body.file == 'undefined') {
7           console.log("No Image Found");
8           res.status(400).json({message: "No Image found"});
9           return
10      }
11
12      submittedFile = req.body.file;
13
14
15      // console.log(req.files);
16      // console.log(submittedFile);
17      fileName = path.basename(submittedFile.path);
18      console.log("Submitted file path: " + submittedFile.path);
19      console.log("Path base name :" + path.basename(submittedFile.path));
20
21      var fileExtension = fileName.split(".");
22      fileExtension = fileExtension[fileExtension.length - 1];
23
24      var stats = fs.statSync(submittedFile.path);
25      var fileSizeInBytes = stats.size;
26      console.log("File Size: " + fileSizeInBytes / 1000000 + "MB");
27      var validFileSize = false;
28
29      if (fileSizeInBytes < 1000000) {
30          validFileSize = true;
31      }
32      else {
33          console.log("File size is over 1MB!");
34      }
35
```

The second half of the function checks whether the submitted file is of an image file type (e.g. .jpg, .jpeg, .png, .gif). Line 36 - 50.

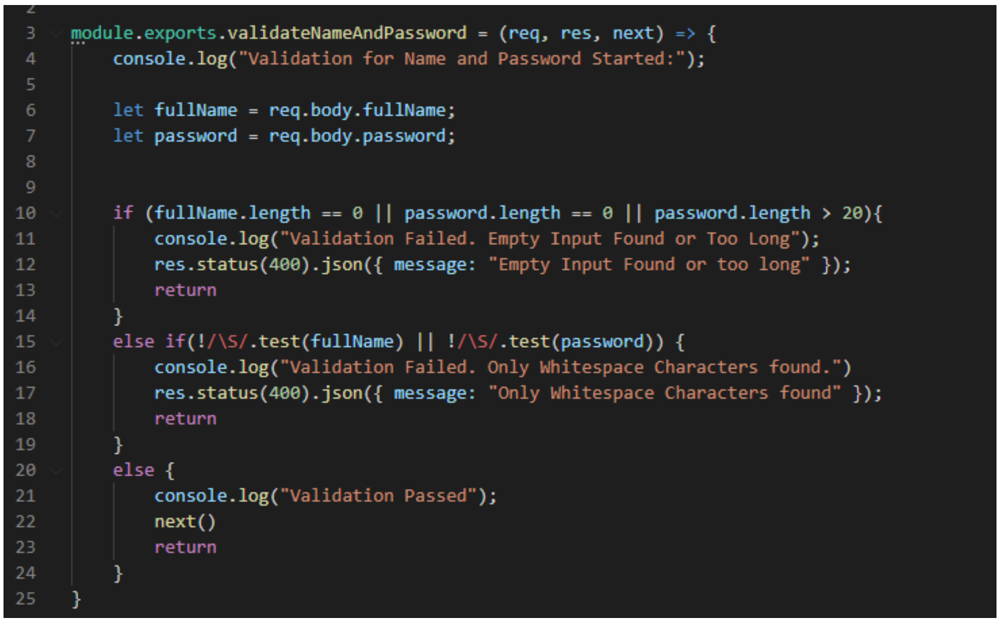


```
29      if (fileSizeInBytes < 1000000) {
30          validFileSize = true;
31      }
32      else {
33          console.log("File size is over 1MB!");
34      }
35
36      var validFileType = false;
37      switch (fileExtension.toLowerCase()) {
38          case "jpg":
39              validFileType = true;
40              break;
41          case "jpeg":
42              validFileType = true;
43              break;
44          case "png":
45              validFileType = true;
46              break;
47          case "gif":
48              validFileType = true;
49              break;
50      }
51
52      if (validFileType == false || validFileSize == false) {
53          console.log("Invalid File Type or File Size sent by User.");
54          fs.unlink(submittedFile.path, function (err) {
55              if (err) return console.log(err);
56              console.log("file deleted successfully");
57          });
58
59          res.status(400).json({ message: "Invalid File Type or Size" });
60          return;
61      }
62
63      console.log("Validation Passed");
64      next();
65  };
```

Line 52 - 61, deletes the file from the server if the submitted file is of the wrong file type or too big. Afterwards, the server immediately returns a response. Thus the upload not being executed.

In the future, you can reuse the functions by replacing the submittedFile variable with the file that you want to get checked (line 12).

middlewares/<u>checkName&Password.js</u>



```
 3  module.exports.validateNameAndPassword = (req, res, next) => {
 4      console.log("Validation for Name and Password Started:");
 5
 6      let fullName = req.body.fullName;
 7      let password = req.body.password;
 8
 9
10      if (fullName.length == 0 || password.length == 0 || password.length > 20){
11          console.log("Validation Failed. Empty Input Found or Too Long");
12          res.status(400).json({ message: "Empty Input Found or too long" });
13          return
14      }
15      else if(!/\S/.test(fullName) || !/\S/.test(password)) {
16          console.log("Validation Failed. Only Whitespace Characters found.")
17          res.status(400).json({ message: "Only Whitespace Characters found" });
18          return
19      }
20      else {
21          console.log("Validation Passed");
22          next()
23          return
24      }
25  }
```

The function above checks whether the name and password are empty, or only contains spaces. If so, the request is immediately stopped and the user needs to ‘remake’ the request with valid inputs.

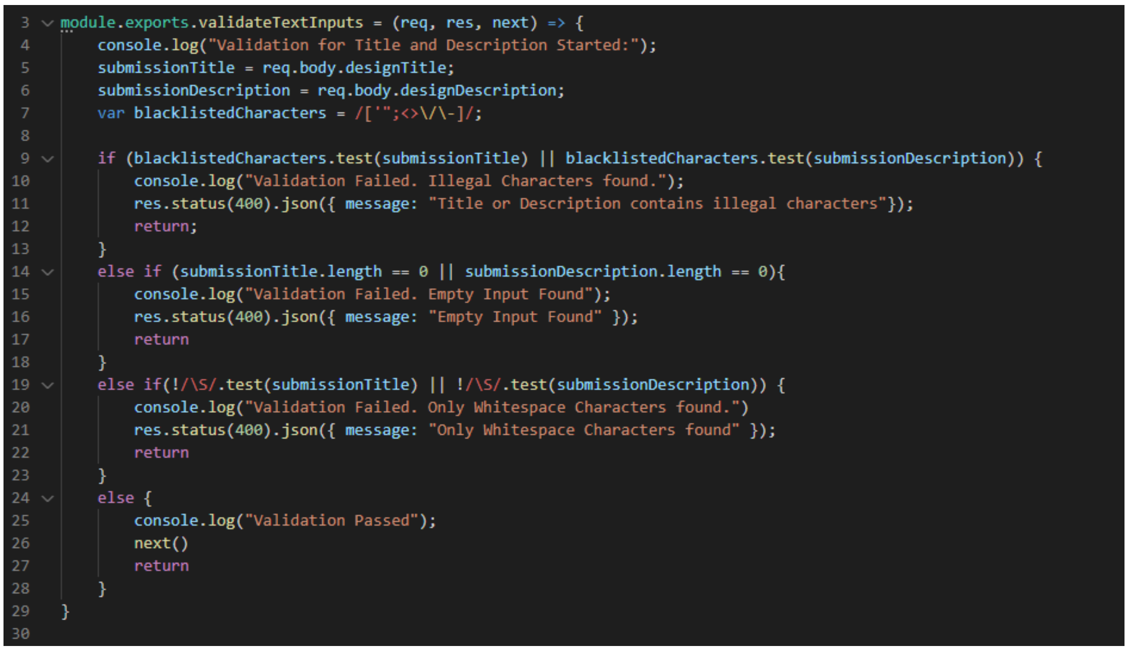
Note: The function can be modified to add more stringent checks to the name and password. As an example the password is checked to ensure that its length is greater than 0 but less than 21(line 10), but if you want to make all passwords be at least 7 length. You can modify the code in line 10 from

password.length == 0

To

password.length < 6

middlewares/checkValidTitle&Description.js



```javascript
module.exports.validateTextInputs = (req, res, next) => {
    console.log("Validation for Title and Description Started:");
    submissionTitle = req.body.designTitle;
    submissionDescription = req.body.designDescription;
    var blacklistedCharacters = /['";<>\/\-]/;

    if (blacklistedCharacters.test(submissionTitle) || blacklistedCharacters.test(submissionDescription)) {
        console.log("Validation Failed. Illegal Characters found.");
        res.status(400).json({ message: "Title or Description contains illegal characters"});
        return;
    }
    else if (submissionTitle.length == 0 || submissionDescription.length == 0){
        console.log("Validation Failed. Empty Input Found");
        res.status(400).json({ message: "Empty Input Found" });
        return
    }
    else if(!/\S/.test(submissionTitle) || !/\S/.test(submissionDescription)) {
        console.log("Validation Failed. Only Whitespace Characters found.")
        res.status(400).json({ message: "Only Whitespace Characters found" });
        return
    }
    else {
        console.log("Validation Passed");
        next()
        return
    }
}
```

Characters like **' " ; < > / \ -** should never be allowed to interact with the SQL query as they are the characters that can interfere with the SQL query (Section 3.1 SQL Injections).

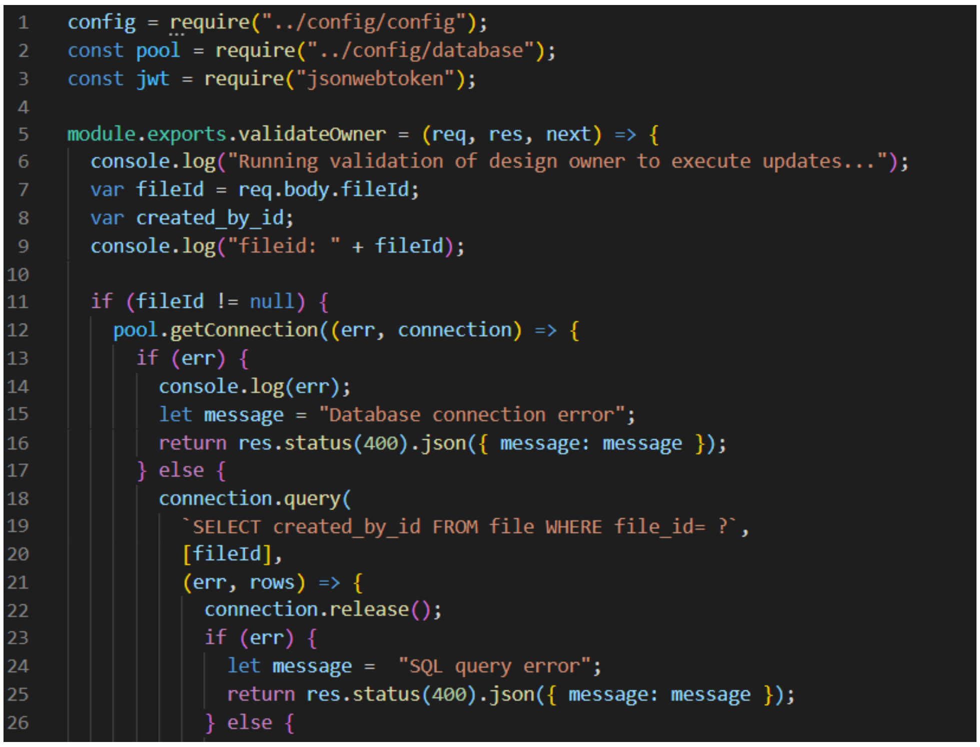
The above code can be reused to check inputs that might interact with SQL code in the future. Just replace the variable ‘submissionTitle’ or ‘submissionDescription’ with the value that you want to check.

middlewares/checkDesignOwner.js (validateOwner)

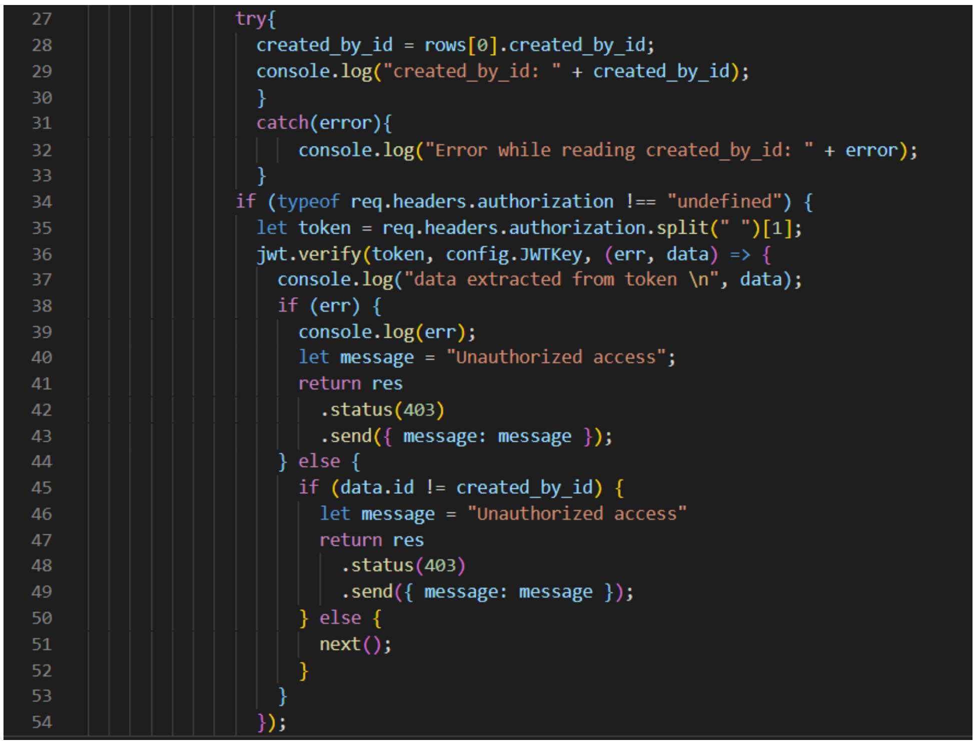
This middleware function checks that only the design's owner can update the design details from the user site.____

Lines 18-32 handle the retrieval of the owner of the design.

Lines 34-58 checks if the token exists and then compares the id of the token with the design owner's id in the database.

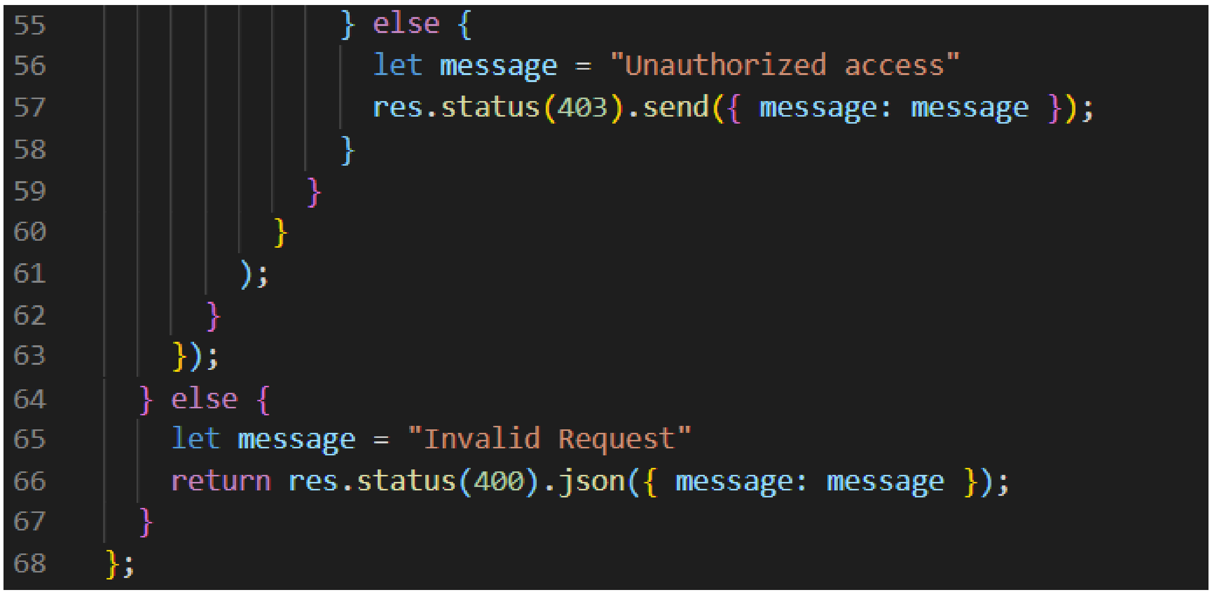


```
1   config = require("../config/config");
2   const pool = require("../config/database");
3   const jwt = require("jsonwebtoken");
4
5   module.exports.validateOwner = (req, res, next) => {
6     console.log("Running validation of design owner to execute updates...");
7     var fileId = req.body.fileId;
8     var created_by_id;
9     console.log("fileid: " + fileId);
10
11    if (fileId != null) {
12      pool.getConnection((err, connection) => {
13        if (err) {
14          console.log(err);
15          let message = "Database connection error";
16          return res.status(400).json({ message: message });
17        } else {
18          connection.query(
19            `SELECT created_by_id FROM file WHERE file_id= ?`,
20            [fileId],
21            (err, rows) => {
22              connection.release();
23              if (err) {
24                let message =  "SQL query error";
25                return res.status(400).json({ message: message });
26              } else {
```



```
27                try{
28                  created_by_id = rows[0].created_by_id;
29                  console.log("created_by_id: " + created_by_id);
30                }
31                catch(error){
32                    console.log("Error while reading created_by_id: " + error);
33                }
34                if (typeof req.headers.authorization !== "undefined") {
35                  let token = req.headers.authorization.split(" ")[1];
36                  jwt.verify(token, config.JWTKey, (err, data) => {
37                    console.log("data extracted from token \n", data);
38                    if (err) {
39                      console.log(err);
40                      let message = "Unauthorized access";
41                      return res
42                        .status(403)
43                        .send({ message: message });
44                    } else {
45                      if (data.id != created_by_id) {
46                        let message = "Unauthorized access"
47                        return res
48                          .status(403)
49                          .send({ message: message });
50                      } else {
51                        next();
52                      }
53                    }
54                  });
```

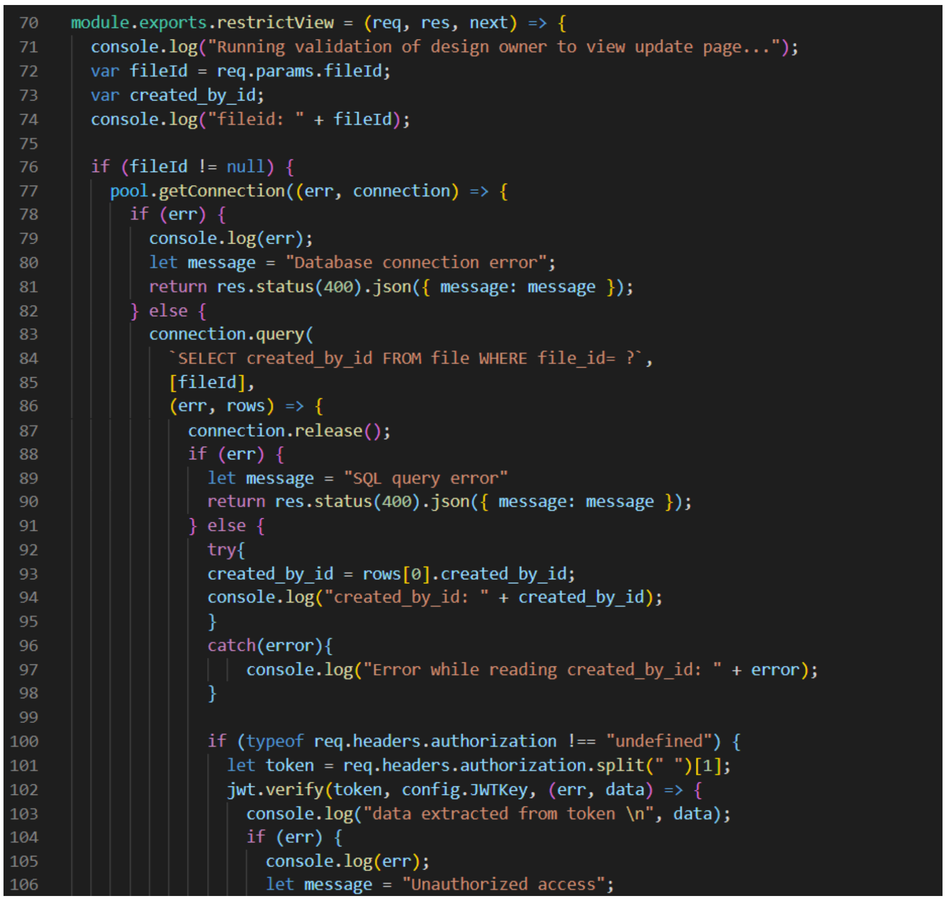
Line 64-68 returns an error message if the fileld requested is invalid.



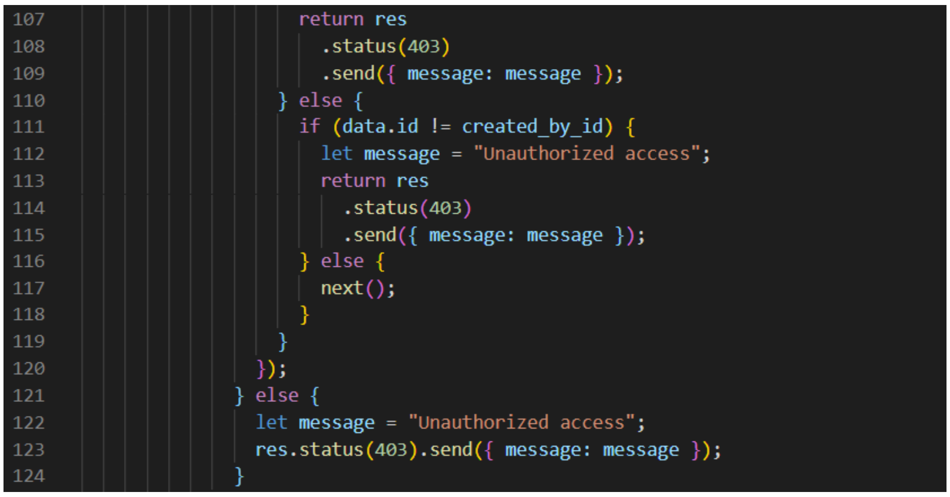
```
55                  } else {
56                    let message = "Unauthorized access"
57                    res.status(403).send({ message: message });
58                  }
59                }
60              }
61            );
62          }
63        });
64      } else {
65        let message = "Invalid Request"
66        return res.status(400).json({ message: message });
67      }
68    };
```

middlewares/checkDesignOwner.js (restrictView)
This middleware function checks that only the design's owner can update the design details from the user site.

Lines 83-98 handle the retrieval of the owner of the design.
Lines 100-123 checks if the token exists and then compares the id of the token with the design owner's id in the database.

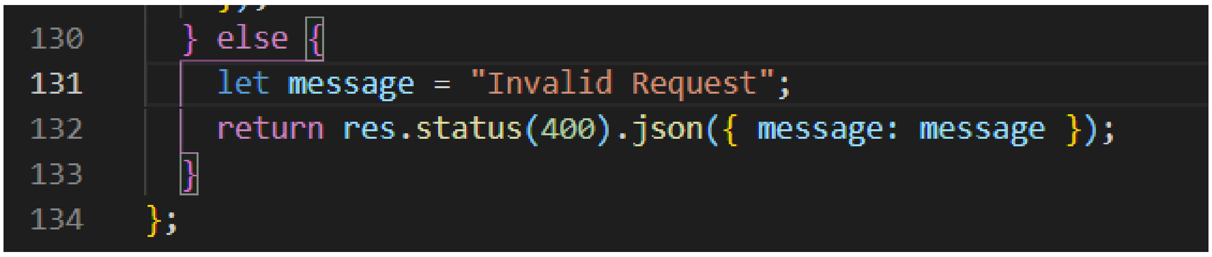


```
70  module.exports.restrictView = (req, res, next) => {
71    console.log("Running validation of design owner to view update page...");
72    var fileId = req.params.fileId;
73    var created_by_id;
74    console.log("fileid: " + fileId);
75
76    if (fileId != null) {
77      pool.getConnection((err, connection) => {
78        if (err) {
79          console.log(err);
80          let message = "Database connection error";
81          return res.status(400).json({ message: message });
82        } else {
83          connection.query(
84            `SELECT created_by_id FROM file WHERE file_id= ?`,
85            [fileId],
86            (err, rows) => {
87              connection.release();
88              if (err) {
89                let message = "SQL query error"
90                return res.status(400).json({ message: message });
91              } else {
92                try{
93                created_by_id = rows[0].created_by_id;
94                console.log("created_by_id: " + created_by_id);
95                }
96                catch(error){
97                    console.log("Error while reading created_by_id: " + error);
98                }
99
100               if (typeof req.headers.authorization !== "undefined") {
101                 let token = req.headers.authorization.split(" ")[1];
102                 jwt.verify(token, config.JWTKey, (err, data) => {
103                   console.log("data extracted from token \n", data);
104                   if (err) {
105                     console.log(err);
106                     let message = "Unauthorized access";
```



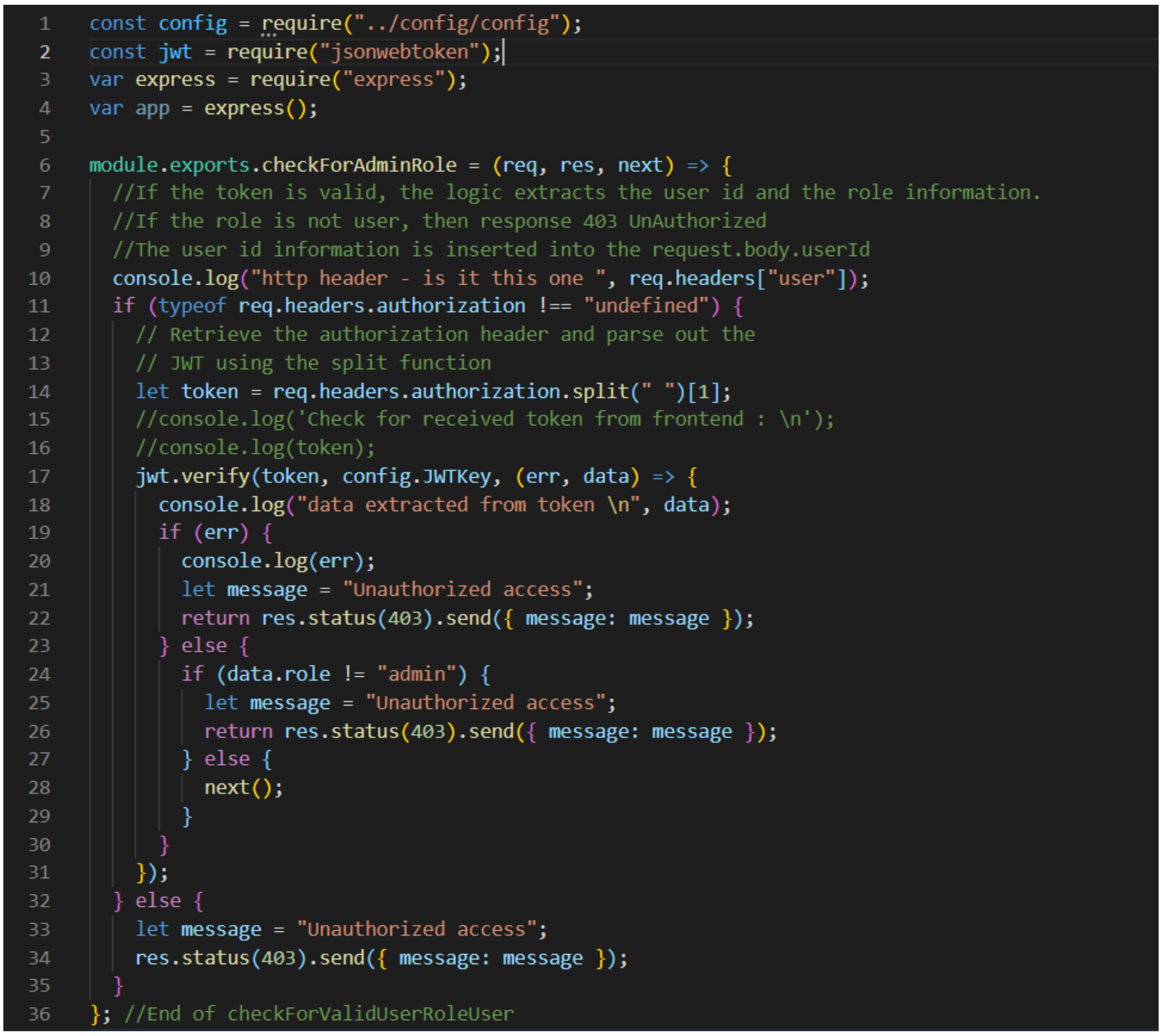
```
107                     return res
108                       .status(403)
109                       .send({ message: message });
110                   } else {
111                     if (data.id != created_by_id) {
112                       let message = "Unauthorized access";
113                       return res
114                         .status(403)
115                         .send({ message: message });
116                     } else {
117                       next();
118                     }
119                   }
120                 });
121               } else {
122                 let message = "Unauthorized access";
123                 res.status(403).send({ message: message });
124               }
```

Lines 130-134 returns an error message if the fileld requested is invalid.



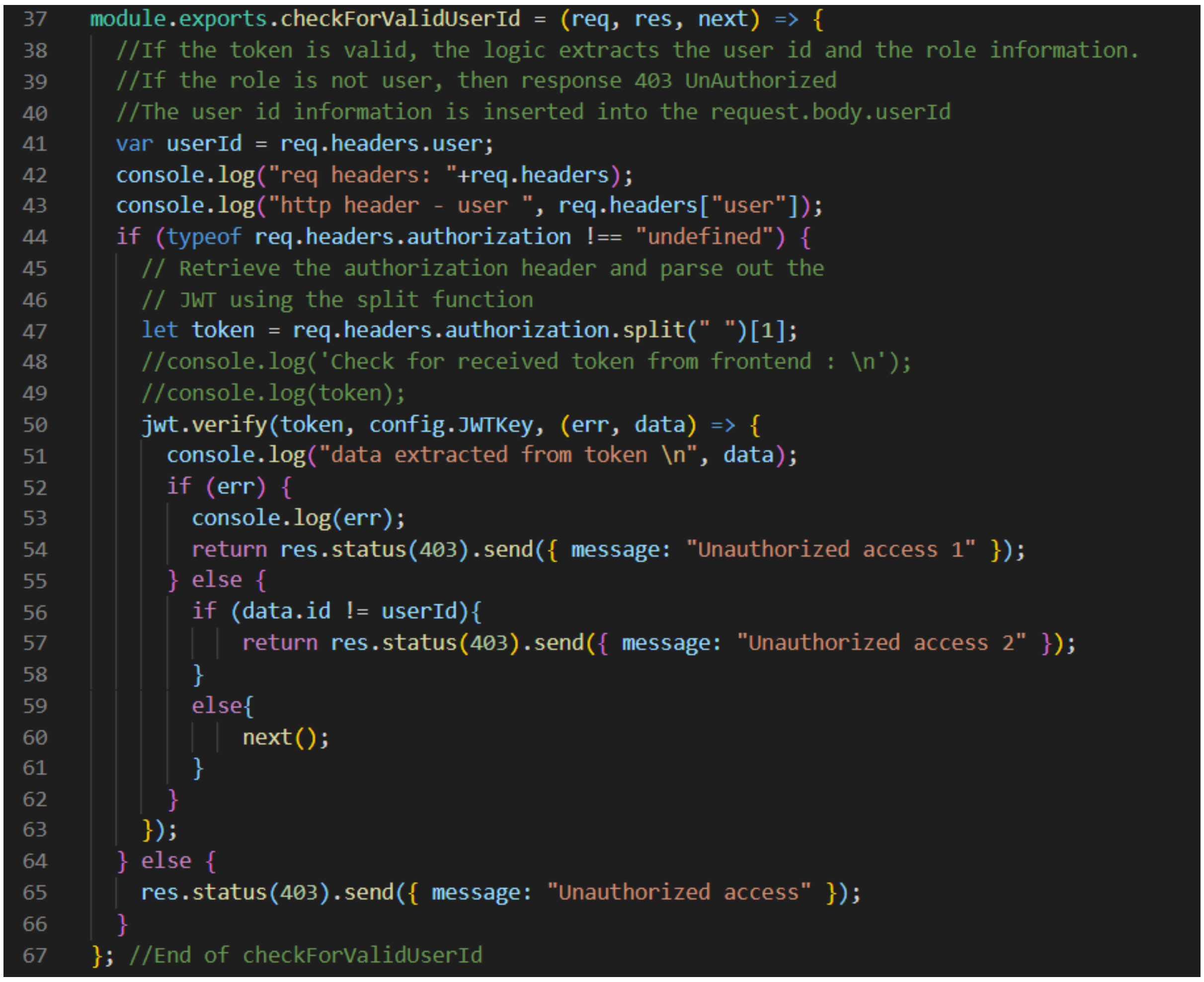
```
130      } else {
131        let message = "Invalid Request";
132        return res.status(400).json({ message: message });
133      }
134    };
```

middlewares/<u>checkUserFnSolution.js (checkForAdminRole)</u>
This middleware checks that the user is an admin.
Lines 11-22 check if the token exists and extracts the data in the token.
Lines 23-31 checks that the role in the token has the value of ‘admin’
Lines 32-35 returns an error message if no token is present in the headers.



```
1   const config = require("../config/config");
2   const jwt = require("jsonwebtoken");
3   var express = require("express");
4   var app = express();
5
6   module.exports.checkForAdminRole = (req, res, next) => {
7     //If the token is valid, the logic extracts the user id and the role information.
8     //If the role is not user, then response 403 UnAuthorized
9     //The user id information is inserted into the request.body.userId
10    console.log("http header - is it this one ", req.headers["user"]);
11    if (typeof req.headers.authorization !== "undefined") {
12      // Retrieve the authorization header and parse out the
13      // JWT using the split function
14      let token = req.headers.authorization.split(" ")[1];
15      //console.log('Check for received token from frontend : \n');
16      //console.log(token);
17      jwt.verify(token, config.JWTKey, (err, data) => {
18        console.log("data extracted from token \n", data);
19        if (err) {
20          console.log(err);
21          let message = "Unauthorized access";
22          return res.status(403).send({ message: message });
23        } else {
24          if (data.role != "admin") {
25            let message = "Unauthorized access";
26            return res.status(403).send({ message: message });
27          } else {
28            next();
29          }
30        }
31      });
32    } else {
33      let message = "Unauthorized access";
34      res.status(403).send({ message: message });
35    }
36  }; //End of checkForValidUserRoleUser
```

middlewares/checkUserFnSolution.js (checkForValidUserId)
This middleware checks that the userId in the request matches the token's userId in the token.
Line 42 gets the userId to be checked.
Lines 44 - 50 checks that the token exists and extracts the data within the token.
Lines 52-61 checks that the userId in the request matches the userId in the token.
Lines 64-67 returns an error message if no token is present in the headers.



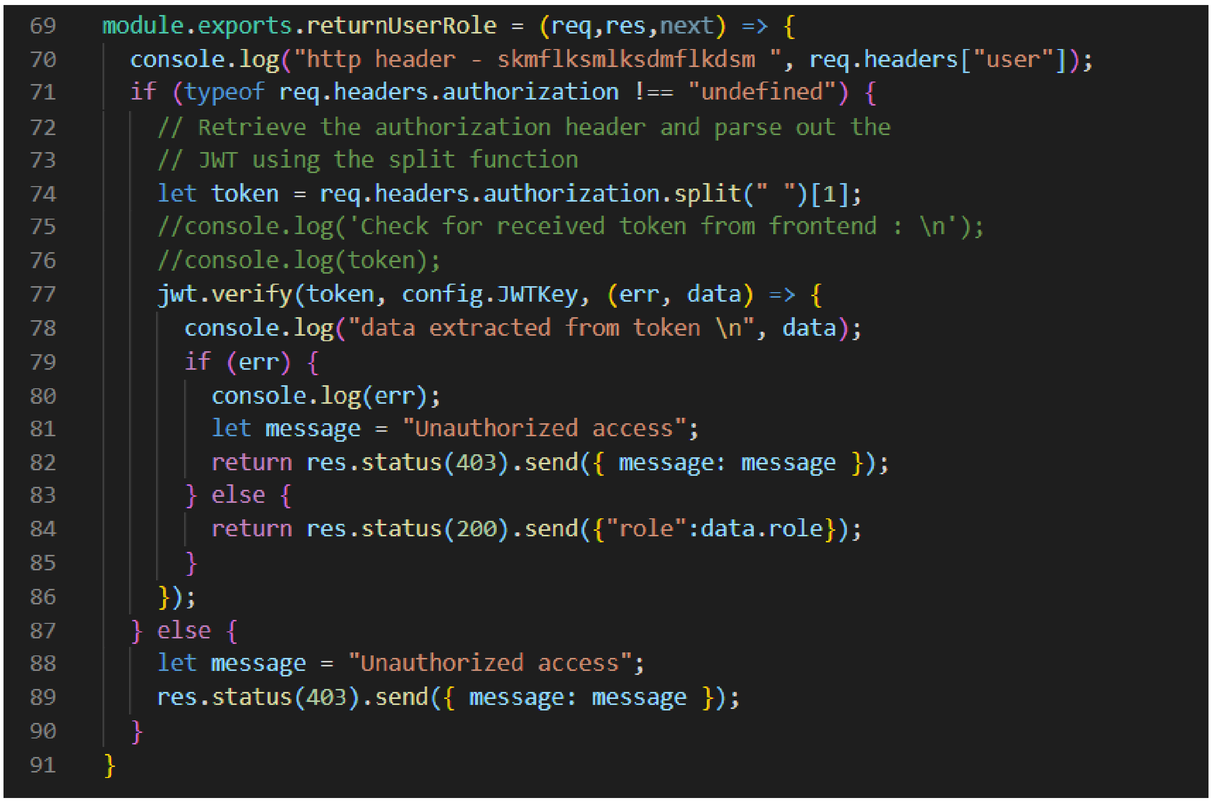
```
37  module.exports.checkForValidUserId = (req, res, next) => {
38    //If the token is valid, the logic extracts the user id and the role information.
39    //If the role is not user, then response 403 UnAuthorized
40    //The user id information is inserted into the request.body.userId
41    var userId = req.headers.user;
42    console.log("req headers: "+req.headers);
43    console.log("http header - user ", req.headers["user"]);
44    if (typeof req.headers.authorization !== "undefined") {
45      // Retrieve the authorization header and parse out the
46      // JWT using the split function
47      let token = req.headers.authorization.split(" ")[1];
48      //console.log('Check for received token from frontend : \n');
49      //console.log(token);
50      jwt.verify(token, config.JWTKey, (err, data) => {
51        console.log("data extracted from token \n", data);
52        if (err) {
53          console.log(err);
54          return res.status(403).send({ message: "Unauthorized access 1" });
55        } else {
56          if (data.id != userId){
57            return res.status(403).send({ message: "Unauthorized access 2" });
58          }
59          else{
60            next();
61          }
62        }
63      });
64    } else {
65      res.status(403).send({ message: "Unauthorized access" });
66    }
67  }; //End of checkForValidUserId
```

middlewares/checkUserFnSolution.js (returnUserRole)

This middleware returns the role inside the token to the client to ensure that administrators cannot access user pages and vice versa.

Line 71-77 checks if the token exists and extracts the data from the token.

Line 79-86 checks if there was an error extracting the information from the token. If there isn't any error, the role value in the token is returned to the client side.



```
69  module.exports.returnUserRole = (req,res,next) => {
70    console.log("http header - skmflksmlksdmflkdsm ", req.headers["user"]);
71    if (typeof req.headers.authorization !== "undefined") {
72      // Retrieve the authorization header and parse out the
73      // JWT using the split function
74      let token = req.headers.authorization.split(" ")[1];
75      //console.log('Check for received token from frontend : \n');
76      //console.log(token);
77      jwt.verify(token, config.JWTKey, (err, data) => {
78        console.log("data extracted from token \n", data);
79        if (err) {
80          console.log(err);
81          let message = "Unauthorized access";
82          return res.status(403).send({ message: message });
83        } else {
84          return res.status(200).send({"role":data.role});
85        }
86      });
87    } else {
88      let message = "Unauthorized access";
89      res.status(403).send({ message: message });
90    }
91  }
```

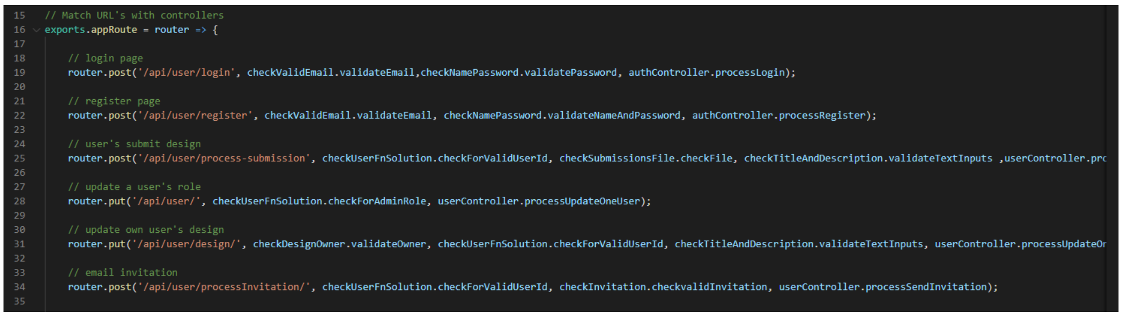
4.3.1 How to use Middleware Functions

For how to make additional expressjs' middleware functions, please go to their website for their documentation. They'll be better at explaining their code than anybody else.

https://expressjs.com/en/guide/writing-middleware.html

For the middleware functions which we have already created, please refer to the ***routes.js*** file inside the folder.

We simply added our middleware functions before the final request.

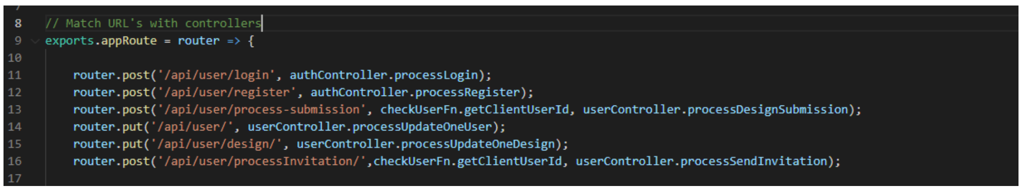


```javascript
// Match URL's with controllers
exports.appRoute = router => {

    // login page
    router.post('/api/user/login', checkValidEmail.validateEmail,checkNamePassword.validatePassword, authController.processLogin);

    // register page
    router.post('/api/user/register', checkValidEmail.validateEmail, checkNamePassword.validateNameAndPassword, authController.processRegister);

    // user's submit design
    router.post('/api/user/process-submission', checkUserFnSolution.checkForValidUserId, checkSubmissionsFile.checkFile, checkTitleAndDescription.validateTextInputs ,userController.prc

    // update a user's role
    router.put('/api/user/', checkUserFnSolution.checkForAdminRole, userController.processUpdateOneUser);

    // update own user's design
    router.put('/api/user/design/', checkDesignOwner.validateOwner, checkUserFnSolution.checkForValidUserId, checkTitleAndDescription.validateTextInputs, userController.processUpdateOr

    // email invitation
    router.post('/api/user/processInvitation/', checkUserFnSolution.checkForValidUserId, checkInvitation.checkvalidInvitation, userController.processSendInvitation);

```

Note: This is just a snippet of the middleware functions we added to the routes. We heavily recommend having a look at the **routes.js** file to have a look at all the functions we added to each endpoint.

Another thing to note is that if you're adding additional middleware functions, you can also simply add them in here as well.

To better illustrate the point this is the same snippet of **route.js** before the addition of our middleware functions.



```javascript
// Match URL's with controllers
exports.appRoute = router => {

    router.post('/api/user/login', authController.processLogin);
    router.post('/api/user/register', authController.processRegister);
    router.post('/api/user/process-submission', checkUserFn.getClientUserId, userController.processDesignSubmission);
    router.put('/api/user/', userController.processUpdateOneUser);
    router.put('/api/user/design/', userController.processUpdateOneDesign);
    router.post('/api/user/processInvitation/',checkUserFn.getClientUserId, userController.processSendInvitation);

```

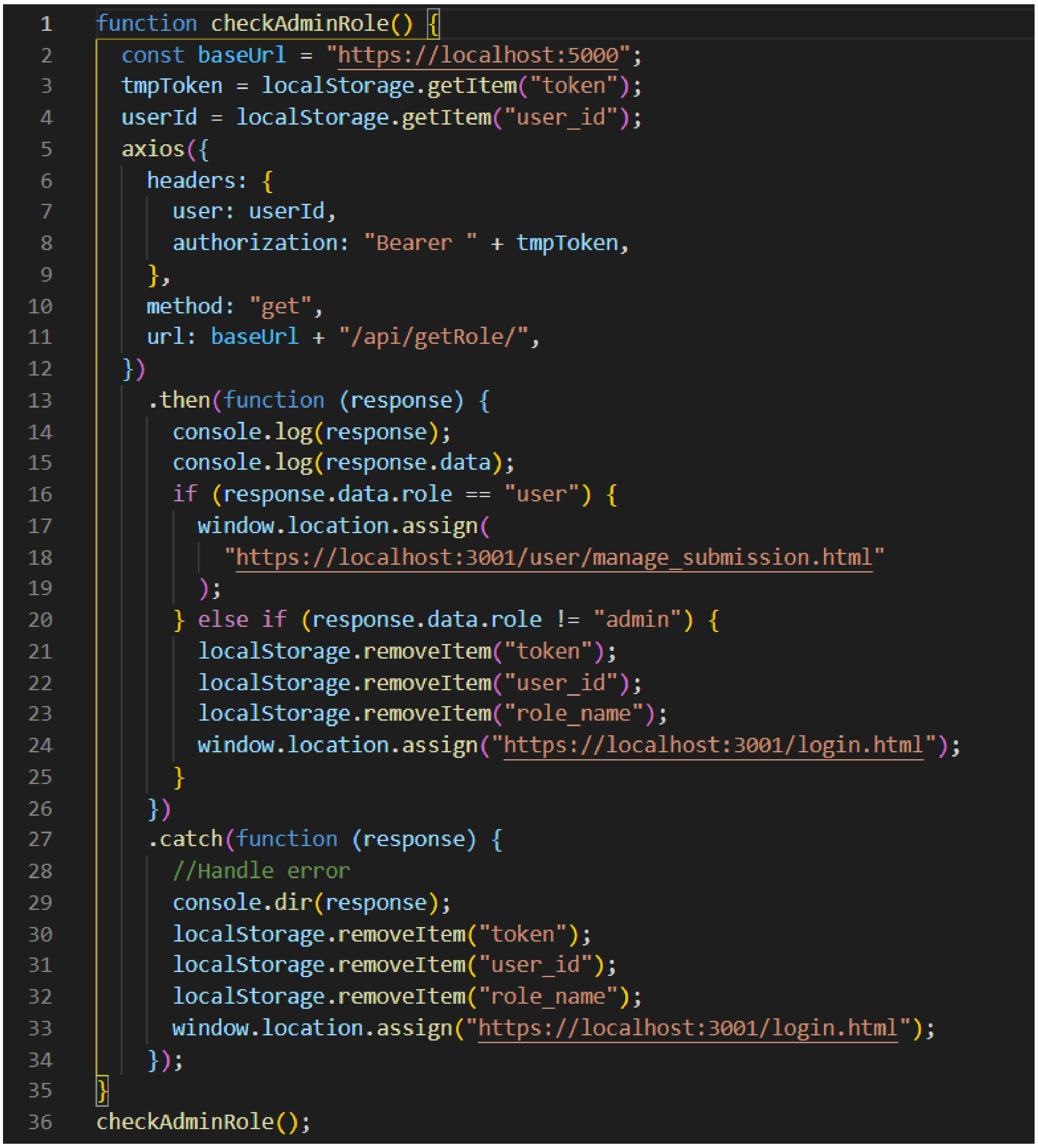
Note how in our version of the **processLogin** endpoint, we have a validateEmail and validatePassword middleware.

4.4 Reusable JavaScripts in the Front End Client

We also have 2 reusable JavaScript codes for the front end client to ensure that users cannot access administrator pages and vice versa.
These JavaScript codes will not only help prevent the Bee Design Award Competition from being sabotaged but also will help expedite your development process through their reusability.

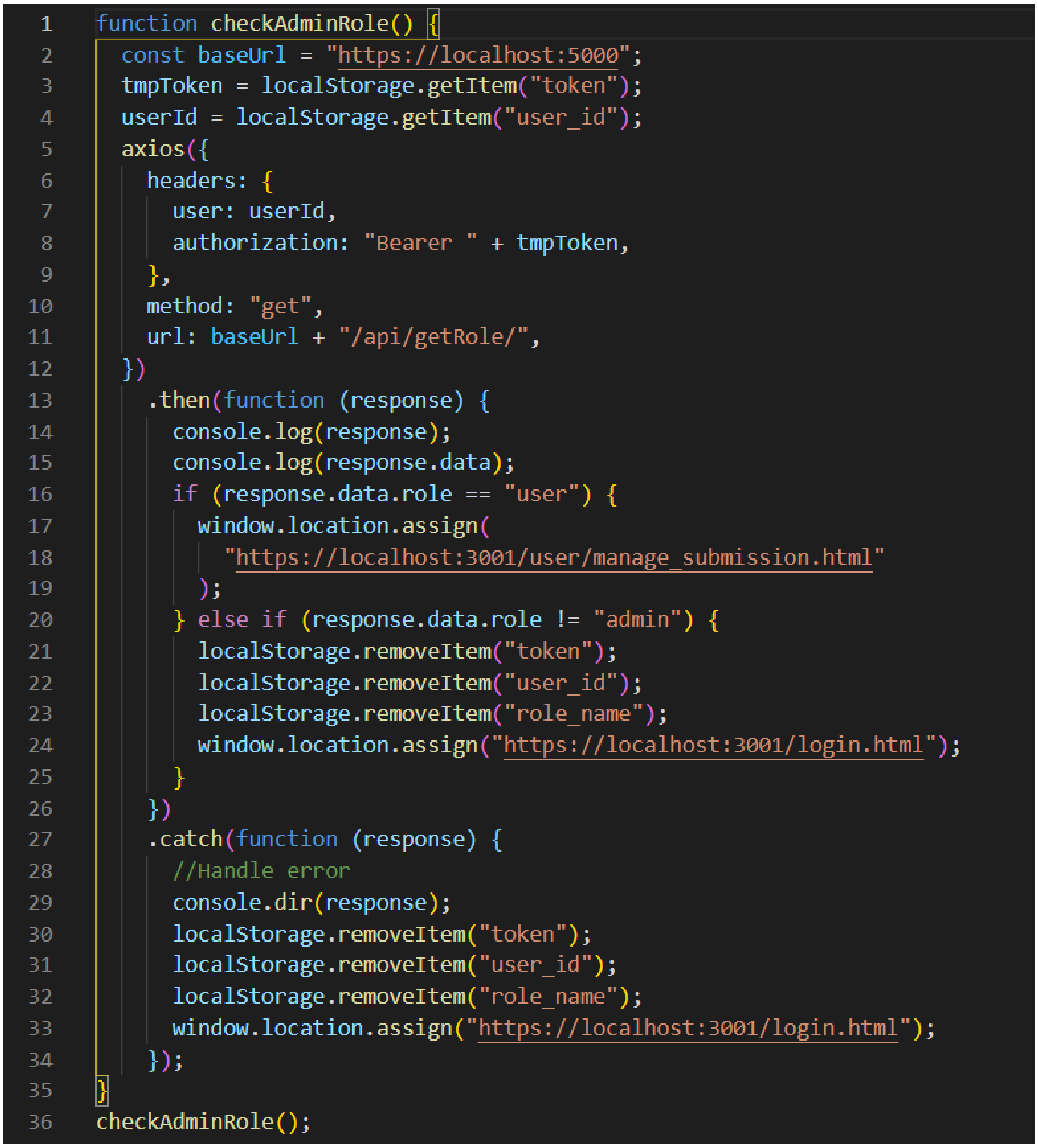
Before referring to the reusable codes below, we highly suggest looking into all of the Javascripts inside the SimulatedFrontEnd/public/js folder so that you can understand and use the middleware which is right for the situation.
Note: We've already incorporated the Javascript codes where appropriate. To see how to implement Javascripts into the Front End Client, please see section 4.4.1 below.

js/checkAdminRole.js
This javascript calls the back end server to check the role of the user via the token. If the user is an admin, he/she proceeds to the administrator's page. If the user is a user, he/she is redirected to the user's page.
Else, the user is kicked out to the login page.
Lines 3 and 4 get the token and the userId from the localStorage respectively.
Lines 5-12 sends the request to get the role of the token from the backend.
Lines 13-19 redirects the user to the user page if his/her role is a user.
Lines 20-25 redirects the user to the login page if his/her role is not an admin.
Lines 27-35 redirects the user to the login page if an error occurs.



```
 1  function checkAdminRole() {
 2    const baseUrl = "https://localhost:5000";
 3    tmpToken = localStorage.getItem("token");
 4    userId = localStorage.getItem("user_id");
 5    axios({
 6      headers: {
 7        user: userId,
 8        authorization: "Bearer " + tmpToken,
 9      },
10      method: "get",
11      url: baseUrl + "/api/getRole/",
12    })
13      .then(function (response) {
14        console.log(response);
15        console.log(response.data);
16        if (response.data.role == "user") {
17          window.location.assign(
18            "https://localhost:3001/user/manage_submission.html"
19          );
20        } else if (response.data.role != "admin") {
21          localStorage.removeItem("token");
22          localStorage.removeItem("user_id");
23          localStorage.removeItem("role_name");
24          window.location.assign("https://localhost:3001/login.html");
25        }
26      })
27      .catch(function (response) {
28        //Handle error
29        console.dir(response);
30        localStorage.removeItem("token");
31        localStorage.removeItem("user_id");
32        localStorage.removeItem("role_name");
33        window.location.assign("https://localhost:3001/login.html");
34      });
35  }
36  checkAdminRole();
```

js/checkUserRole.js
This javascript calls the back end server to check the role of the user via the token. If the user is a user, he/she proceeds to the user's page. If the user is an admin, he/she is redirected to the administrator's page.
Else, the user is kicked out to the login page.
Lines 3 and 4 get the token and the userId from the localStorage respectively.
Lines 5-12 sends the request to get the role of the token from the backend.
Lines 13-19 redirects the user to the user page if his/her role is an admin.
Lines 20-25 redirects the user to the login page if his/her role is not a user.
Lines 27-35 redirects the user to the login page if an error occurs.



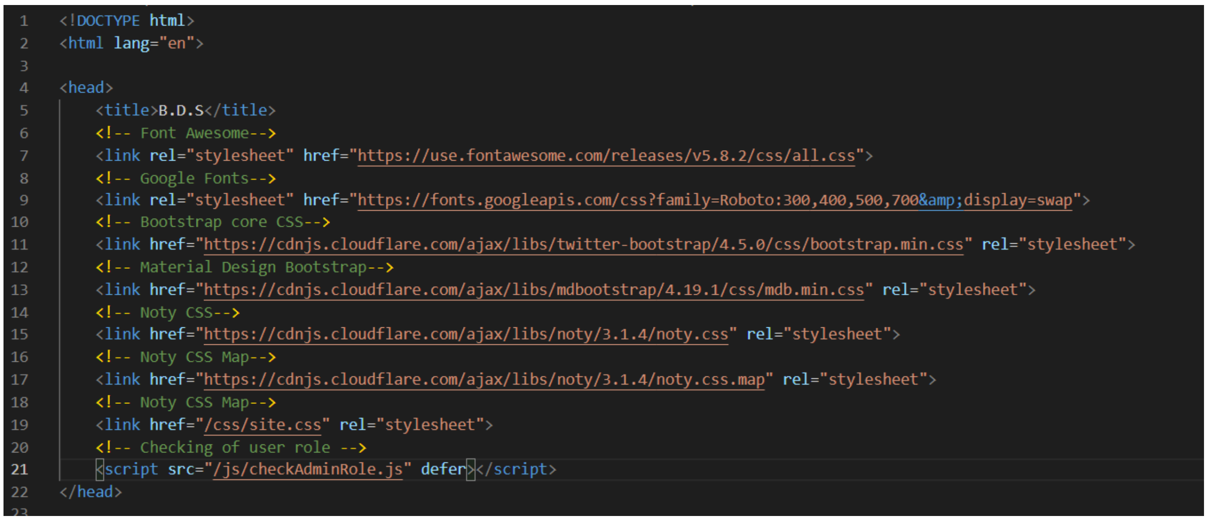
```
 1  function checkAdminRole() {
 2    const baseUrl = "https://localhost:5000";
 3    tmpToken = localStorage.getItem("token");
 4    userId = localStorage.getItem("user_id");
 5    axios({
 6      headers: {
 7        user: userId,
 8        authorization: "Bearer " + tmpToken,
 9      },
10      method: "get",
11      url: baseUrl + "/api/getRole/",
12    })
13      .then(function (response) {
14        console.log(response);
15        console.log(response.data);
16        if (response.data.role == "user") {
17          window.location.assign(
18            "https://localhost:3001/user/manage_submission.html"
19          );
20        } else if (response.data.role != "admin") {
21          localStorage.removeItem("token");
22          localStorage.removeItem("user_id");
23          localStorage.removeItem("role_name");
24          window.location.assign("https://localhost:3001/login.html");
25        }
26      })
27      .catch(function (response) {
28        //Handle error
29        console.dir(response);
30        localStorage.removeItem("token");
31        localStorage.removeItem("user_id");
32        localStorage.removeItem("role_name");
33        window.location.assign("https://localhost:3001/login.html");
34      });
35  }
36  checkAdminRole();
```

4.4.1 How to use Javascript codes

The Javascript codes that have been created are in the public/js folder, namely checkUserRole.js and checkAdminRole.js.

The HTML files where the Javascript codes are added to can be found under public/user as well as public/admin, both containing pages for the user and the admin respectively.

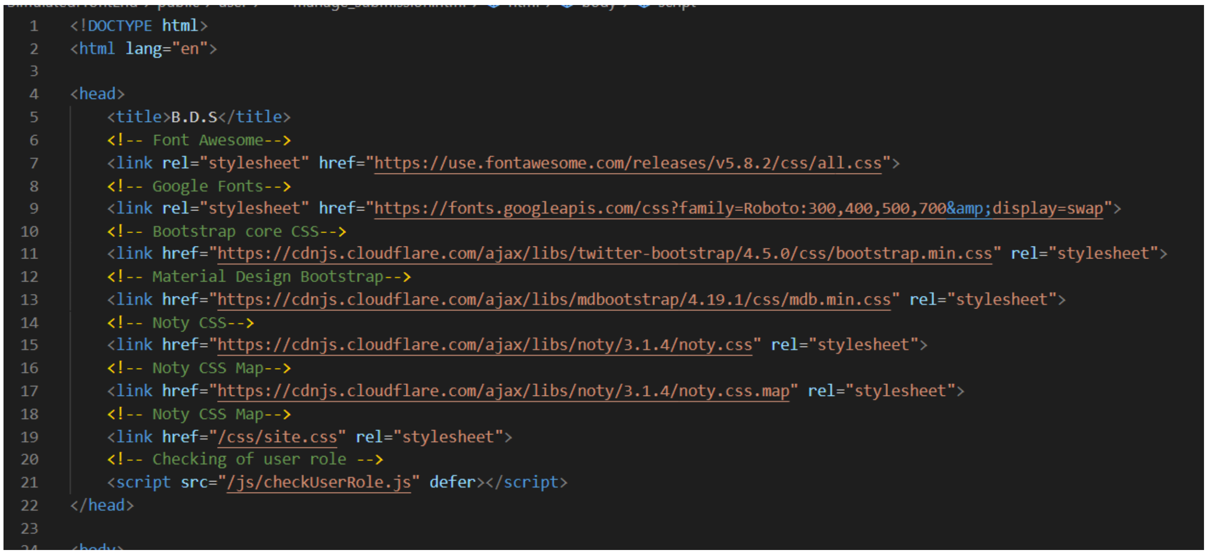
public/admin/check_admin_submission.html



```
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/v5.8.2/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/mdbootstrap/4.19.1/css/mdb.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      <!-- Checking of user role -->
21      <script src="/js/checkAdminRole.js" defer></script>
22  </head>
```

As seen above, the Javascript code to check for the admin role is inserted in line 21 above.
You may add this line wherever you'd want the page to be restricted to a certain role, such as a user or an admin.
Using defer will run checkAdminRole.js before loading the lines of code after it.

public/user/manage_submission.html



```html
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/v5.8.2/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/mdbootstrap/4.19.1/css/mdb.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      <!-- Checking of user role -->
21      <script src="/js/checkUserRole.js" defer></script>
22  </head>
23
```

Similar to public/admin/check_admin_submission.html shown earlier, the Javascript code to check for the user role is inserted in line 21 above.

You may add this line wherever you'd want the page to be restricted to a certain role, such as a user or an admin.

Using defer will run checkAdminRole.js before loading the lines of code after it.