

**SINGAPORE
POLYTECHNIC**



School Of Computing

ST505

ESDE (SECURE CODING)

WEB APPLICATION AND PENETRATION TESTING REPORT

AY2021S2 2A/23

Sara Golkaran (P1935657)

Yap Min Ru Vicki (P1904556)

Table of Contents

Brief Introduction	3
1. Security of the Website	3
1.1 Injection (SQL Injection)	3
1.1.1 Detailed example on how the flaw can be exploited	4
1.1.2 Where the flaw is found	5
1.1.3 Recommendations	5
1.1.4 Solutions	5
1.1.5 Final Outcome (Evidence on the results after solving the flaw)	6
1.2 Broken Authentication	6
1.2.1 Detailed example on how the flaw can be exploited	7
1.2.2 Where the flaw is found	9
1.2.3 Recommendations	10
1.2.4 Solutions	10
1.2.5 Final Outcome (Evidence on the results after solving the flaw)	15
1.3 Sensitive Data Exposure	16
1.3.1 Detailed example on how the flaw can be exploited	16
1.3.2 Where the flaw is found	17
1.3.3 Recommendations	17
1.3.4 Solutions	17
1.3.5 Final Outcome (Evidence on the results after solving the flaw)	19
1.4 Broken Access Control	20
1.4.1 Detailed example on how the flaw can be exploited	20
1.4.2 Where the flaw is found	22
1.4.3 Recommendations	22
1.4.4 Solutions	22
1.4.5 Final Outcome (Evidence on the results after solving the flaw)	24
1.5 Cross-Site Scripting (XSS)	26
1.5.1 Detailed example on how the flaw can be exploited	26
1.5.2 Where the flaw is found	27
1.5.3 Recommendations	27
1.5.4 Solutions	28
1.5.5 Final Outcome (Evidence on the results after solving the flaw)	30
1.6 Insufficient Logging & Monitoring	31
1.6.1 Detailed example on how the flaw can be exploited	31
1.6.2 Where the flaw is found	32
1.6.3 Recommendations	32
1.6.4 Solutions	32
1.6.5 Final Outcome (Evidence on the results after solving the flaw)	34
1.7 Others (Using Components with Known Vulnerabilities)	34
1.7.1 Detailed example on how the flaw can be exploited	35
1.7.2 Where the flaw is found	35
1.7.3 Recommendations	35
1.7.4 Solutions	35
1.7.5 Final Outcome (Evidence on the results after solving the flaw)	36
2. Code Review	36
2.1 Inconsistent JSON Response Structure	36
2.2 Poor Error Handling	41

Brief Introduction

In this project, one of the issues faced by us is improper handling of errors. This flaw allows attackers to easily introduce the above mentioned variety of security problems in the web application server and environment. In programming, Javascript is an operating language that processes only one statement at a time. However, actions such as requesting for a design through an API calling will take an indefinite amount of time to execute and since the API was written to perform in a synchronous manner, the browser will not be able to handle other user inputs like clicking of the 'search' button until the previous request has finish executing. Therefore, in order to prevent such blocking behaviors, it is better practice to write Javascript in an asynchronous manner by replacing the use of callbacks with promise-based techniques, so that the requests can run in parallel with other operations instead of sequentially. Another of the issues faced by us is varying JSON response output. This flaw means that developers are accustomed to their own type of responses so if they take over other developers' portions, they may not have the same level of code interpretation. Therefore, we will implement basic data structure to return HTTP status codes and relevant database rows or error messages regardless of the APIs. In Section 2 of Code Review, we will be modifying the codes to fix poor error handling and inconsistent JSON data structure.

1. Security of the Website

1.1 Injection (SQL Injection)

Definition: A type of web security vulnerability that allows an attacker to interfere with the queries that an application makes to its database, which convinces the application to run SQL code that was not intended. A successful SQL injection exploit is able to read sensitive data from the database. It is also able to modify the database data (Insert/Update/Delete) and execute administration operations on the database. This injection is able to recover the content of a given file and in some cases issues commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands. There are two types of SQL Injections: In-Band and Blind. In-Band SQL allows the attacker to get a direct output from the web server and is easier to exploit as the response will be shown in the form of an error message or database data, while Blind SQL forces the attacker to ask the database server a series of "true" or "false" questions, with the response indicated as one of the options, in order to derive the actual data.

	Vulnerability	Likelihood
Level	High	High
Explanation	A successful SQL injection exploit allows the attacker to read sensitive data such as passwords from the database, execute unauthorized operations on the database such as giving themselves or removing others from Administrator privileges, and even recovering, modifying or destroying the contents of a certain file. In the case of this project, the attacker can access all designs without needing to filter by its title or description. He just needs to input a payload that always returns true and the SQL will be forced to execute the	Injection flaws are commonly found in database-driven websites, where even a software package with minimal user base can be subjected to such an attack. It depends on the fact that the SQL does not make any real distinction between the control and data planes.


	string exactly the way it was input.	
--	--------------------------------------	--

1.1.1 Detailed example on how the flaw can be exploited


Step 1: We log in using Rita’s email and password credentials.

Login

Your email

rita@designer.com

Your password

*****

SUBMIT

rita

rita@designer.com

BACK

Step 2: We enter ‘OR 1=1--’ into the search box and this results in all the design pieces being displayed, including other user’s design pieces. As shown below, it is possible to launch a SQL injection from the **manage_submission.html** page. When the user is on that page, he will be able to verify the vulnerability by supplying a payload which involves inserting a query that always returns true into the design titles search input, thus successfully bypassing the design titles search without inputting the corresponding numbers.

Manage my submissions

Search design title

OR 1=1--

SEARCH

1

2

3


4

5

6

7


Rita Design A



Design A

Description A

UPDATE



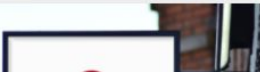
BOB DESIGN 2

Design file sample for testing

Testing

Testing

UPDATE



1.1.2 Where the flaw is found

The vulnerability allows the user to search for designs without inputting the corresponding numbers, which results in the display of all the design titles. In this Javascript file, placeholders were not used when writing the query in the database. The code below is exposed to a SQL Injection attack where the attacker's malicious input will be treated as part of the query and executed.

Code Snippet, located in fileService.js file

```
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= ${userId} );SELECT @total_records total_records;`;
} else {
designFileDataQuery = `SELECT file_id,cloudinary_url,design_title,design_description
FROM file WHERE created_by_id=${userId} AND design_title LIKE '%${search}%' LIMIT ${limit} OFFSET ${offset};
SET @total_records =(SELECT count(file_id) FROM file WHERE created_by_id= ${userId} AND design_title LIKE '%${search}%' );SELECT @total_records total_records;`;
}
```

By inputting ' OR 1=1--' into the search box, the design pieces are not filtered and will all be displayed on **Manage_submission.html** page as the SQL query executed becomes **SELECT file_id, cloudinary_file_id, cloudinary_url, design_title, design_description FROM file where created_by_id = 100 AND design_title LIKE '% OR 1=1--%' LIMIT....** Since 1=1 returns true, the database will return all the rows from the 'file' tables.

1.1.3 Recommendations

1. Prepared Statements (with Parameterized Queries)

The main cause of SQL Injection attacks is due to the lack of used placeholders. When you don't properly escape user input, the SQL will execute the string exactly the way it was entered. This means any characters that include SQL syntax will be considered as part of the overall SQL query. One way of which this can be prevented is by using ? as placeholders instead of \${variable_name}. The use of prepared statements ensure that the attacker is not able to change the intent of the query, even if SQL commands are inserted by the attacker. For example, if an attacker were to enter any random user ID of Harry' or '1'=1, the parameterized query would not be vulnerable and would search for a username which literally matched the entire string harry' or '1'= '1.

2. Stored Procedures

Stored Procedures have the same effect as the use of parameterized queries when implemented safely which is normal for most stored procedures languages. The difference between prepared statements and stored procedures is that the SQL code for stored procedure is defined and stored in the database itself, and then called from the application. We create procedures in MySQL to create specific views with restricted access to parts of the database rather than using normal queries. With normal queries, we can select all columns data from a table but if we utilise stored procedures, we are able to embed multiple SQL statements and even implement if else statements or looping logic to add a layer of security to sensitive column data.

1.1.4 Solutions

1. Prepared Statements (with Parameterized Queries)

The main cause of SQL Injection attacks is due to the lack of indicating placeholders thus it is necessary to use placeholders in order to prevent SQL Injection attacks. With the use of this placeholder, the attacker is not able to change the intent of the query. Even if the attacker wished to insert any random file_id, the parameterized query would not be vulnerable and would search for the file_id that matched the entire string. With placeholders, the query function will escape the input from the user such that the sql query will search for file_id with values '100' or 1=1;-- -which really does not exist in the database. With placeholders, escaping is a form of sanitisation is done on the input before the query is executed on the database server. Thus, we modified the **fileService.js file within the services folder of experimentsecuritywithcompetitionsystem** by replacing all SQL queries with '?' placeholders to prevent SQL injection. Hence, by changing all the SQL queries to prepared statements.

Code Snippet, located in fileService.js file

```

17 |         if ((search == '') || (search == null)) {
18 |             console.log('Prepare query without search text');
19 |             designFileDataQuery = `SELECT file_id,cloudinary_url,design_title,design_description
20 | FROM file WHERE created_by_id=? LIMIT ? OFFSET ?;
21 | SET @total_records =(SELECT count(file_id) FROM file WHERE created_by_id= ? );SELECT @total_records total_records; `;
22 |         } else {
23 |             designFileDataQuery = `SELECT file_id,cloudinary_url,design_title,design_description
24 | FROM file WHERE created_by_id=? AND design_title LIKE ? LIMIT ? OFFSET ?;
25 | SET @total_records =(SELECT count(file_id) FROM file WHERE created_by_id= ? AND design_title LIKE ? );SELECT @total_records total_records; `;
26 |         }

```

We chose to use **Prepared Statements (with Parameterized Queries)** instead of Stored Procedures to stop the SQL Injection attack as Prepared Statements simply requires us to write queries with placeholders instead of actual values while Stored Procedures requires us to have special DBMS permissions in order to call them. As Prepared Statements are pre compiled and can be executed multiple times, we find it easier to apply.

1.1.5 Final Outcome (Evidence on the results after solving the flaw)

As seen in the **manage_submission.html** page below, the code is fixed by using **Prepared Statements (with Parameterized Queries)** as the attacker is not able to perform any more SQL injection attacks and a message stating 'no submission records found' will also be thrown out if the user tries to input a random string.

1.2 Broken Authentication

Definition: A type of web security vulnerability that allows an attacker to bypass the authentication methods or session management systems that are used by a web application, which can lead to impersonation of other users and gain access to functionalities they should not have. Broken authentication attacks aim to take one or more accounts to grant the attacker the same privileges as the victim. Authentication is then “broken” when the attackers are able to successfully compromise passwords, keys or sessions tokens, user accounts information and other personal details to assume user identities. It also permits the attackers to attack the victims with a variety of methods like credential stuffing, unhashed passwords and misconfigured session timeouts. These will then allow attackers to easily gain control of user information such as tokens, ids and even their passwords. For example, many websites require the users to log into avail access to its services. Usually, the login system possesses a username and an associated password. This pair of right credentials will generate a unique session ID for each of account holders and are then combined to identify the unique identity of the associated user.

	Vulnerability	Likelihood
Level	High	High
Explanation	A successful broken authentication exploit allows the attacker to use automated tools with password lists and gain access to valid username and password combinations which can further lead to money laundering, social security frauds, stolen identity and disclosure of	Broken authentication flaws are commonly found in the design and implementation of most identity and access controls. It is made possible due to the permit of brute force or automated attacks and default weak or well-known passwords like passwords. Improper validation of

http://localhost:5000/api/user/100

GET http://localhost:5000/api/user/100 Send

Params Authorization Headers (10) Body Pre-request Script Tests Settings

Headers 9 hidden

KEY	VALUE	DESCRIPTION	...	Bulk
Key	Value	Description		

Body Cookies Headers (8) Test Results

Status: 200 OK Time: 58 ms Size: 373 B

Pretty Raw Preview Visualize JSON

```

1 {
2   "userdata": {
3     "user_id": 100,
4     "fullname": "rita",
5     "email": "rita@designer.com",
6     "role_id": 2,
7     "role_name": "user"
8   }
9 }

```

For Brute Changing of UserId

Step 1: We log in using Rita's email and password credentials.

[Home](#)
[Login](#)
[Register](#)

Login

Your email
rita@admin.com

Your password
.....

SUBMIT

Step 2: We then inspect the local storage by pressing ctrl+shift+i then navigating under application tab to check if the user id matches Rita's user id. It displays 100 which matches the database record of Rita's user id.

[Profile](#)

rita

rita@designer.com

BACK

About this interface: User usually visit this interface to check whether their profile data. For large scale system, the interface will even show their last login and logout date information

- guided links to help them reach other interfaces to manage their profile etc such as password reset.

Application

- Manifest
- Service Workers
- Clear storage

Storage

- Local Storage
 - http://localhost:3001
 - Session Storage
 - IndexedDB
 - Web SQL
 - Cookies
 - http://localhost:3001
- Cache
 - Cache Storage
 - Application Cache
- Background Services
 - Background Fetch

Filter

Key	Value
user_id	100
token	eyJhbGciOiJIUzI1NiIsInR5cCI6I...
role_name	user

Step 3: We changed the user id from 100 to 105, which is John's user id. By performing this action, we will know if we are able to switch to another user account without logging in.


```

module.exports.getClientUserId = (req, res, next) => {
  console.log('http header - user ', req.headers['user']);
  req.body.userId = req.headers['user'];
  console.log('Inspect user id which is planted inside the request header : ', req.body.userId);
  if (req.body.userId != null) {
    next()
    return;
  } else {
    res.status(403).json({ message: 'Unauthorized access' });
    return;
  }
}

} //End of getClientUserId

```

1.2.3 Recommendations

1. Improving Session Management

We can also use a secure server side built-in session manager that generates a new random session ID after each login session. This means that the session IDs would not be revealed in the URL, and will be invalidated after every logout, idle, and absolute timeouts, instead of having the token still shown under local storage.

2. Password Enumeration

The main cause of broken authentication attacks is due to weak passwords. When a user uses a weak password, the account is more susceptible to hacking. One way of which to prevent this is to use strong passwords which follow a high level of complexity: a combination of alphanumeric characters consisting of capital and small letters, numbers, punctuation marks, mathematical and other conventional symbols. Authentication failure responses should not explicitly indicate which part of the authentication data was incorrect. For instance, use "Invalid username and/or password" instead of "Invalid username" or "Invalid password". The error responses must be identical in both display and source code and account disabling after an established number of invalid login attempts should also be enabled to discourage brute force guessing of credentials, but not so long until a denial-of-service attack can be performed.

1.2.4 Solutions

We chose to use both **Session Management** and **Password Enumeration** to stop the Broken Authentication attack as the solutions work hand in hand.

1. Improving Session Management

We created a **blacklistedJWT token table** in MySQL database. The purpose of creating this table is to store the token of other users that have previously logged in, so as to prevent the hacker from retrieving this token and using it to access the account.

	primarykey	fk_user_id	jwt_token_blacklist	Timestamp
▶	35	102	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-29 08:45:39
	36	102	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 01:17:14
	37	100	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDAsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 01:29:52
	38	102	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 01:36:21
	39	124	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 01:45:32
	40	100	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDAsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 01:52:35
	41	100	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDAsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 07:17:20
	42	102	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJ1c2VyIiwiaXNjaXJlci...	2020-12-30 07:23:20
	43	101	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJhZG1pb2IsImV4cGly...	2020-12-30 07:39:25
	44	101	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJhZG1pb2IsImV4cGly...	2020-12-30 07:40:06
	45	101	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJhZG1pb2IsImV4cGly...	2020-12-30 07:42:40
	46	101	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMDIsInJvbGUiOiJhZG1pb2IsImV4cGly...	2020-12-30 07:56:10

We also created a new file called **verifyToken.js** within the **middleware** folder of **experimentsecuritywithcompetitionsystem**. The purpose of creating this file is to replace the previous **checkUserFn.js** file and properly authenticate the validity of the user's token upon login.

Code Snippet, located in verifyToken.js file

```
const jwt = require('jsonwebtoken');
const JWT_SECRET = process.env.JWTKEY;
const auth = require('../services/authService');

async function verifyToken(req, res, next){
  console.log("verifyToken is Running")
  let blacklisted_tokens;
  // console.log(req.headers)
  let token = req.headers['authorization']; //retrieve authorization header's content
  // console.log(token);
  if(!token || !token.includes('Bearer')){ //process the token
    res.status(403);
    return res.send({auth:'false', message:'Not authorized!'});
  }else{
    token=token.split('Bearer ')[1]; //obtain the token's value
    // console.log(token);
    try {
      blacklisted_tokens = await auth.getBlacklistedTokens();
      blacklisted_tokens = blacklisted_tokens.map((token)=>{ return token.jwt_token_blacklist });

    } catch (error) {
      let message = 'Server is unable to process your request.';
      return res.status(500).json({ message: error });
    }
    // console.log(blacklisted_tokens)

    if (blacklisted_tokens.length > 0) {
      for (tkn of blacklisted_tokens) {
        if (tkn == token) {
          res.status(403);
          return res.send({auth:'false', message:'Not authorized!'})
        }
      }
    }
  }
  jwt.verify(token, JWT_SECRET, function(err, decoded){ //verify token
    if(err){
      res.status(403);
      return res.end({auth:false, message:'Not authorized!'});
    }else{
      req.body.userId=decoded.user_id; //extracted from token
      req.role = decoded.role_name; //extracted from token
      // console.log(decoded);
      req.tokenExpiry = decoded.expiresOn;
      req.token = token;
      next();
      console.log("VALID TOKEN ACCEPTED")
    }
  });
}

module.exports = verifyToken;
```

We edited the **authController.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem**. The purpose is to create the signature part of the token by encoding in the user id and role name and converting the javascript expiry to the ISO standard of YYYY-MM-DDTHH:mm:ss.sssZ. We also include the private key to verify that the sender of the JWT is really the user logged in. By doing so, there is no need to implement local storage to store the user id and role name of the logged in user.

Code Snippet, authController.js file

```
exports.processLogin = async (req, res, next) => {
  let email = req.body.email;
  let password = req.body.password;

  try {
    let results = await auth.authenticate(email);
    console.log("GOT RESULTS")
    if (results.length == 1) {
      if ((password == null) || (results[0] == null)) {
        res.status(500).json({ "message": 'login failed' });
        logger.error(`${res.statusCode} - Login Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      }
      if (bcrypt.compareSync(password, results[0].user_password) == true) {
        console.log("CORRECT PASSWORD")
        let today = new Date()
        // console.log(new Date())
        today.setDate(today.getDate() + 1);
        let data = {
          "success": true,
          "message": "design updated successfully",
          "data": {
            role_name: results[0].role_name,
            token: jwt.sign({ user_id: results[0].user_id, role: results[0].role_name, expiresOn: today.toISOString().slice(0, 19).replace('T', ' '), config.JWTKey, {
              expiresIn: 86400 //Expires in 24 hrs
            })
          }
        }
      }
    }
    res.status(200).json(data);
    logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  } else {
    res.status(500).json({ "message": 'error' });
    logger.error(`${res.statusCode} - Login Failed (Catch) - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  }
} // end
} //end
} catch (error) {
  res.status(500).json({ "message": "error" });
  logger.error(`${res.statusCode} - Login Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
  return
} //end of try/catch
};
```

We edited the **authController.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** by adding a new asynchronous logout function. It retrieves the token of the user id and sets a countdown to expire based on the timestamp that the user logs in.

Code snippet, located at authController.js file

```
exports.processLogout = async (req, res, next) => {
  console.log('Process logout is currently running');
  // console.log(req.headers['authorization'])
  let token = req.token;
  let user_id = req.body.userId;
  let timestamp = req.tokenExpiry;

  // console.log(user_id)
  // console.log(token)
  // return res.status(200).json({ message: 'Completed Logout' });
  try {
    let results = await auth.logout(user_id, token, timestamp);
    console.log(results);
    res.status(200).json({
      "success": true,
      "message": 'Logout Successfully',
      "data": {}
    });

    logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  } catch (error) {
    console.log('processLogout method : catch block section code is running');
    console.log(error, '=====');
    res.status(500).json({ "message": 'Unable to complete logout' });
    logger.error(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  }
}

}; //End of process
```

We also modified the **routes.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** by including another POST API to execute the logout function after the user ended his session.

Code Snippet, located in route.js file

```
router.post('/api/user/logout', verifyToken, authController.processLogout);
```

We also modified the **global.js** file within the **js** folder of **SimulatedFrontEnd**. The purpose of this file is to clear the user's local storage upon logout and redirect them back to the home page.

Code Snippet, located in global.js file

```
// Logout
$( '#logoutButton' ).on( 'click', function ( event ) {
    event.preventDefault();
    const baseUrl = 'https://localhost:5000';
    let token = localStorage.getItem( 'token' );
    axios({
        method: 'post',
        url: baseUrl + '/api/user/logout',
        headers: {
            // 'Content-Type': 'multipart/form-data',
            "Authorization": "Bearer " + token
        }
    })
    .then(function (response) {
        console.log(response)
        localStorage.clear();
        window.location.replace('/home.html');
    })
    .catch(function (response) {
        console.log(response)
        new Noty({
            type: 'error',
            layout: 'topCenter',
            theme: 'sunset',
            timeout: '6000',
            text: 'Error logging out. Please Try Again!',
        }).show();
    });
});
})
```

We also added in two more module exports within the **authService.js** file within the **services** folder of **experimentsecuritywithcompetitionsystem**. The purpose of the logout module is to insert all blacklisted tokens into the blacklistedjwt table in MySQL database after the user has ended his application session so that there will be a future reference list which will be called by the export module explained after this.

Code Snippet, located in authService.js file

```
module.exports.logout = (userID, token, timestamp) => {
    console.log("LOGOUT FUNCTION CALLED")
    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(`INSERT INTO blacklistedjwt (fk_user_id, jwt_token_blacklist, Timestamp) VALUES (?, ?, ?)`, [userID, token, timestamp], (err, rows) => {
                    if (err) {
                        reject(err);
                    } else {
                        resolve(rows);
                    }
                });
                connection.release();
            }
        });
    });
}; //End of new Promise object creation
} //End of Logout
```

The purpose of the getBlacklistedTokens module is to make sure that the hacker is not able to use back the same token to log into the user's account as the login token will be validated against the tokens stored in the table. Thus, this can prevent session hijacking.

Code snippet, authService.js file


```

module.exports.getBlacklistedTokens = () => {
  console.log("getBlacklistedTokens FUNCTION CALLED")
  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(`SELECT jwt_token_blacklist FROM blacklistedjwt`, (err, rows) => {
          if (err) {
            reject(err);
          } else {
            resolve(rows);
          }
          connection.release();
        });
      }
    });
  }); //End of new Promise object creation
} //End of getBlacklistedTokens

```

We also modified the **index.js** file of **experimentsecuritywithcompetitionsystem**. The purpose of this file is to implement a 24 hour session timeout. This means that after a day has passed, the user will automatically be logged out of his account.

Code Snippet, located in index.js file

```

session({
  secret: ['alllearnershavetobethebest'],
  name: "jwt",
  cookie: {
    httpOnly: true,
    secure: false,
    sameSite: true, //carry the session on the same site
    maxAge: 1000*60*24 //session last for 24hrs before it times you out
  }
});

```

2. Using Password Enumeration

We also created a new file called **validationFn.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem**. The purpose of this file is to ensure that the user's choice of password fulfills the requirements based on the regular expressions.

Code Snippet, located in validationFn.js file

```

validateRegister: function (req, res, next) {

    var fullName = req.body.fullName;
    var email = req.body.email;
    var password = req.body.password;

    reUserName = new RegExp(`^[a-zA-Z\\s,']+$`);
    rePassword = new RegExp(`^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#\\$%\\^&\\*])(?=.{8,})`);

    console.log(fullName, email, password);

    if (reUserName.test(fullName) && rePassword.test(password) && validator.isEmail(email)) {
        console.log('validate is running');
        next();
    } else {
        console.log("validation error")
        logger.error(`${res.statusCode} || 500} - Register Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        res.status(500).json({ 'message': 'Invalid Name and/or email and/or password' });
    }
},

```

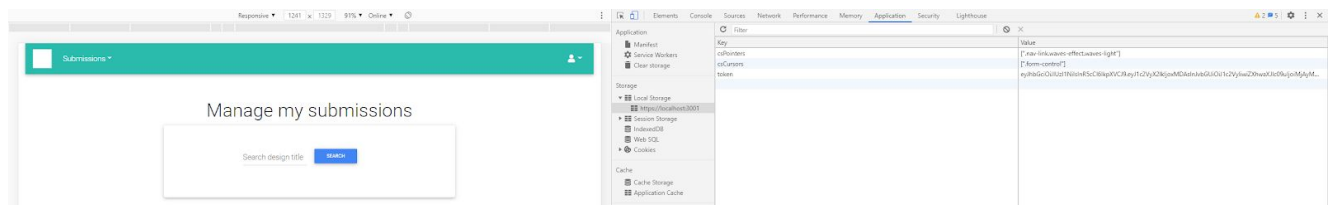
1.2.5 Final Outcome (Evidence on the results after solving the flaw)

As seen in the image below, the code is fixed by using both **Session Management** and **Password Enumeration** as the attacker is not able to perform any more broken authentication attacks by changing the user id of the logged in user.

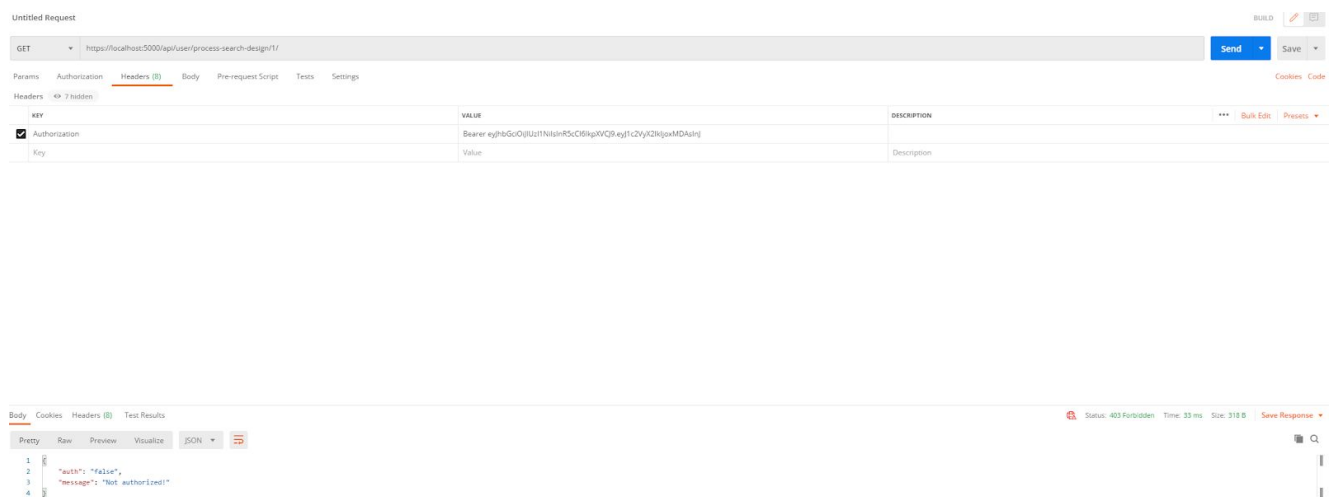
Improving Session Management

Step 1: We log in using Rita's email and password credentials.

Step 2: At `manage_submission.html` page, we press F12 to inspect the application and see that both the userid and role is not shown in the local storage. Thus, the hacker will not be able to manually change the user id of the logged in user as he doesn't know what the correct digits are.



As shown in the image below, even if the hacker manages to retrieve the token, he would not be able to use the token to do any search. It will then print out a message saying he is not authorised to do so.



1.3 Sensitive Data Exposure

Definition: A type of web security vulnerability that allows an attacker to steal and access information due to the web application having inadequate security over protected data. It occurs as a result of not adequately protecting a database where information is stored. This might result in a multitude of things such as weak encryption, no encryption, software flaws, or when someone mistakenly uploads data to the wrong database. Attackers are then able to perform such attacks because the sensitive data is not encrypted and obtain users' personal information and even access their accounts if they manage to get the passwords of other user's accounts.

	Vulnerability	Likelihood
Level	High	High
Explanation	<p>A successful sensitive data exposure exploit allows the attacker to execute middleware attacks, steal keys and clear text data from the user's browser. It often results in the compromise of protected data which includes customer information, employee data, intellectual property and trade secrets, operational and inventory information and industry specific data. In the case of this project, we can use Burp to intercept the HTTP traffic and sensitive data such as the email and password used to login are explicitly stated in the raw response. When the attacker gets the login details, he can then use it to hack into the account.</p>	<p>Sensitive data exposure flaws are commonly found due to implementation of weak key generation, password hashing and storage techniques, algorithm, protocol and cipher or totally not encrypting data. This is especially applicable towards data in transit as server side weaknesses can be easily detected but pose a difficulty for data at rest.</p>

1.3.1 Detailed example on how the flaw can be exploited

Step 1: We log into Rita's account and use Burp to intercept the submission in order to see Rita's personal information such as her password. We deduced that HTTP is not secure enough as Rita's personal information is stated explicitly in the raw response so an attacker can make use of that for further hacking.


```
1 POST /api/user/login HTTP/1.1
2 Host: localhost:5000
3 Content-Length: 266
4 Accept: application/json, text/plain, */*
5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.121 Safari/537.36
6 Content-Type: multipart/form-data; boundary=----WebKitFormBoundarymBSLCJUGOCHSsoH
7 Origin: http://localhost:3001
8 Sec-Fetch-Site: same-site
9 Sec-Fetch-Mode: cors
10 Sec-Fetch-Dest: empty
11 Referer: http://localhost:3001/
12 Accept-Encoding: gzip, deflate
13 Accept-Language: en-US,en;q=0.9
14 Connection: close
15
16 ----WebKitFormBoundarymBSLCJUGOCHSsoH
17 Content-Disposition: form-data; name="email"
18
19 ritaBdesigner.com
20 ----WebKitFormBoundarymBSLCJUGOCHSsoH
21 Content-Disposition: form-data; name="password"
22
23 password
24 ----WebKitFormBoundarymBSLCJUGOCHSsoH--
25
```

1.3.2 Where the flaw is found

The server has no implementation of SSL certificate and the connection is only HTTP, which makes it not as secure as HTTPS, especially in private networks. Although HTTP takes a shorter time to execute requests per second, the packet travelling through it is not encrypted so the attacker can just directly read its contents using third party applications such as Burp to intercept the traffic.

1.3.3 Recommendations

1. Front End Encryption and Back End Decryption

Encryption is a two way function where encryption is scrambled in such a way that it can be unscrambled later on. We can apply a one way Asymmetric Encryption where one key at the front end encrypts and the other key at the back end decrypts. This concept forms the foundation for PKI (public key infrastructure), which is the trust model that undergirds SSL/TLS. We can also apply Symmetric Encryption, which is closer to a form of private key encryption where each party has its own key that can both encrypt and decrypt. After the asymmetric encryption that occurs in the SSL handshake, the browser and server then re-communicate using the symmetric session key that is passed along.

2. Using HTTPS Protocol and SSL Certificate

Hypertext transfer protocol secure (HTTPS) is the secure version of HTTP, which is the primary protocol used to send data between a web browser and the website. HTTPS is encrypted in order to enhance the security of any transferring of data. This is extremely important when any users transmit sensitive and personal data. For example, logging in to a bank account, email services, etc. Websites that require any login credentials should definitely use HTTPS as an encryption protocol to encrypt communications. The protocol is called Transport Layer Security (TLS), which was formerly known as Secure Socket Layer (SSL). SSL (Secure Socket Layer) Certificates enable the websites to move from HTTP to HTTPS which is more secure. A SSL certificate is a data file hosted in a website's origin server, containing the website's public key and identity, along with relevant information. Devices that attempt to communicate with the origin server will reference this file to obtain the public key and verify the server's identity. The private key will be kept secret and secure only to verify the server's identity. Thus, it is important for a website to have a SSL certificate in order to keep the user data secure, verify ownership of the website, prevent any attackers from creating a fake version of the website and gain user's trust. This is made possible due to the public-private key pairing that SSL certificates facilitate.

1.3.4 Solutions

We chose to use **HTTPS Protocol and SSL Certificate** instead of Front End Encryption and Back End Decryption to stop the Sensitive Data Exposure attack as it can prevent phishing of data by encrypting the data from server to client. As the protection is 256-bits compared to the usual 128-bits in terms of security level, we find it more suitable to apply.

Step 1: Open up command prompt, change directory to the backend folder and insert the following command “openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365”. It generates a new root SSL certificate which is then eventually saved into the file.

Step 2: Type password in terminal which saves the cert.pem and key.pem in the backend folder.

Step 3: Answer the following questions:

Country Name (2 letter code) []:

State or Province Name (full name) []:

Locality Name (eg, city) []:

Organization Name (eg, company) []:

Organizational Unit Name (eg, section) []:

Common Name (eg, fully qualified host name) []:

Email Address []:o

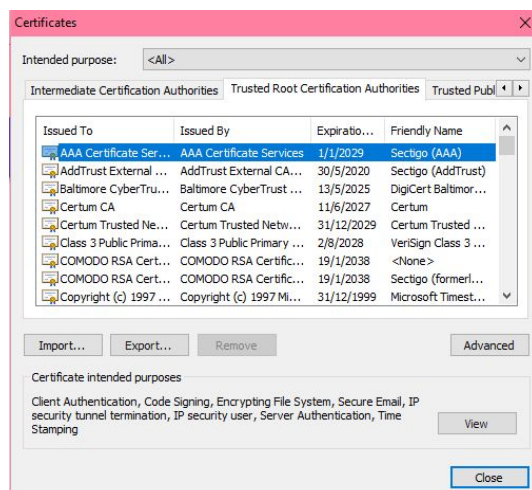
Step 4: Download SSL certificate import into the browser by clicking on ‘manage certificates’.

Manage certificates

Manage HTTPS/SSL certificates and settings



Step 5: Before importing the certificate, click on ‘Trusted Root Certification Authorities’.



Step 6: Import the SSL Certificate into the browser.



cert.pem

23/12/2020 3:25 PM

PEM File

3 KB

After managing to implement our own SSL certificate through the above mentioned steps, we modified the **index.js file of SimulatedFrontEnd**. The purpose of this file is to create a HTTPS server, which is made more secure than the current HTTP server with the use of secret passphrase.

Code Snippet, located in index.js file

```
https.createServer({
  key: fs.readFileSync('./certs/key.pem'),
  cert: fs.readFileSync('./certs/cert.pem'),
  passphrase: 'sweetie1'
}, app).listen(PORT, err => {
  if (err) return console.log(`Cannot Listen on PORT: ${PORT}`);
  logger.info(`Server started and running on https://localhost:${PORT}/`);
  console.log(`Server is Listening on: https://localhost:${PORT}/`);
});
```

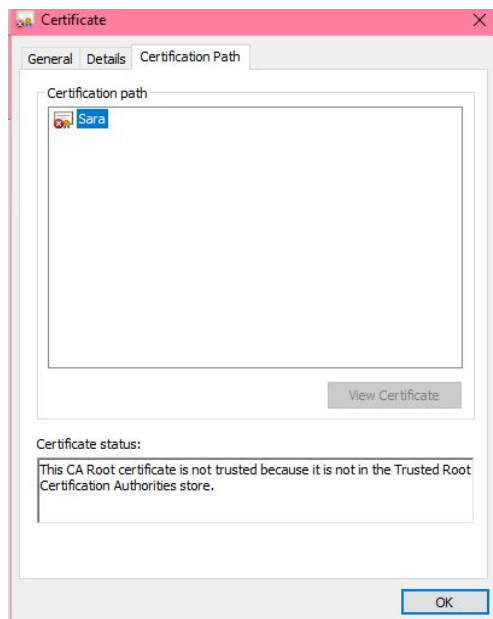
We also modified the **index.js** file of **experimentsecuritywithcompetitionsystem** by adding in a function to listen to the env port of the user's computer to import the SSL certificate created previously.

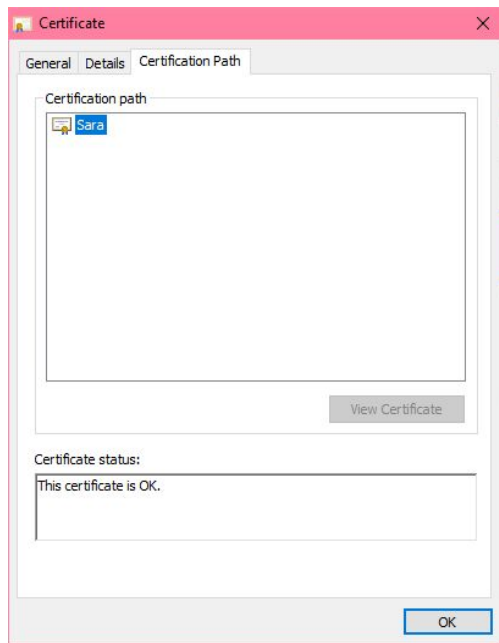
Code Snippet, located in index.js file

```
https.createServer({
  key: fs.readFileSync('./certs/key.pem'),
  cert: fs.readFileSync('./certs/cert.pem'),
  passphrase: 'sweetie1'
}, app).listen(port, err => {
  if (err) return console.log(`Cannot Listen on PORT: ${port}`);
  console.log(`Server hosted at https://${hostname}:${port}`);
});
```

1.3.5 Final Outcome (Evidence on the results after solving the flaw)

As seen in the image below, the code is fixed by using **HTTPS Protocol and SSL Certificate** as the implemented SSL certificate and its certification path is trusted by the webpage, so the attacker is not able to perform any more sensitive data exposure attacks by intercepting HTTP traffic with Burp.





1.4 Broken Access Control

Definition: A type of web security vulnerability that allows an attacker to act outside of their intended permissions. Attackers are able to perform a broken access control attack due to the lack of automated detection, and lack of effective functionality testing by application developers. Since the attackers are not authenticated, it allows them to have unauthorized access to the resources or to perform such action that only web application's admins or even other users have the ability to do it. It can also lead to attackers acting as the users or admin thus using the privileged functions or creating, accessing, updating or deleting every record. This can also lead to the attackers getting unauthorised access to sensitive data that is stored.

	Vulnerability	Likelihood
Level	Moderate	Moderate
Explanation	A successful broken access control exploit allows the attacker to act as users with privileged functions to create, view, update or delete every record or as administrators to conduct a business function outside the limits of an ordinary user. In the case of this project, admin privileges can be granted and removed by a normal user by manually changing the user id of the logged in account.	Broken access control flaws are commonly found due to the lack of automated detection and effective functional testing by application developers. It depends on the protection needs of both the application and its data.


1.4.1 Detailed example on how the flaw can be exploited

Step 1: We log in using Robert's email and password credentials.


[Home](#)[Login](#)[Register](#)

Login

Your email

 robert@admin.com

Your password




[SUBMIT](#)

Step 2: From Part 2, we know that the website is susceptible to broken authentication attacks so we followed the similar steps to change Robert's user id from 101 to 100 which is Rita's user id. We know that unlike Robert, Rita is a normal user, so she should have no administrator privileges such as managing other user profiles.

Key	Value
token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ...
user_id	100
role_name	admin

Step 3: As seen in the images below, we can navigate to the **manage_users.html** page as Rita and give ourselves administrator privileges like Robert, by clicking Administrator in the drop down menu.

 localhost:3001/admin/manage_users.html

Manage users

Search by name [SEARCH](#)

[1](#)[2](#)[3](#)[4](#)[5](#)[6](#)[7](#)


rita

rita@designer.com

[MANAGE](#)

robert

robert@admin.com



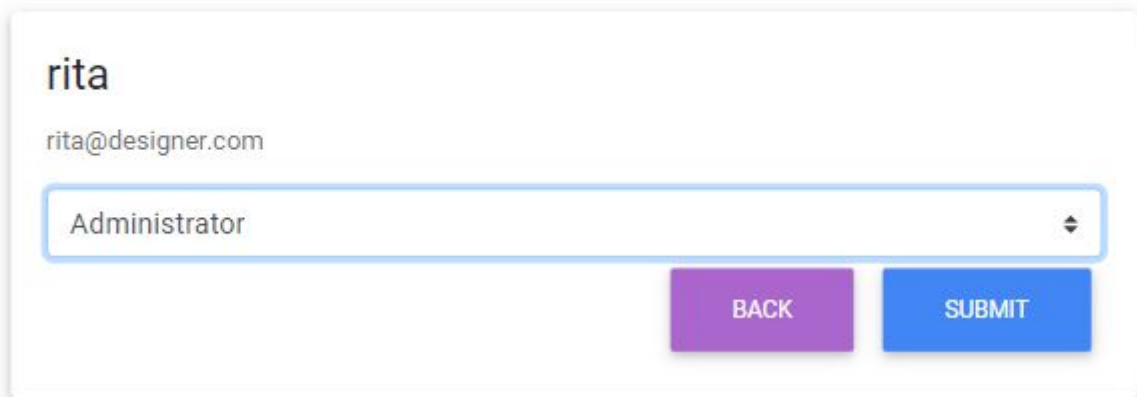
[MANAGE](#)

bob

bob@designer.com

[MANAGE](#)

Update user



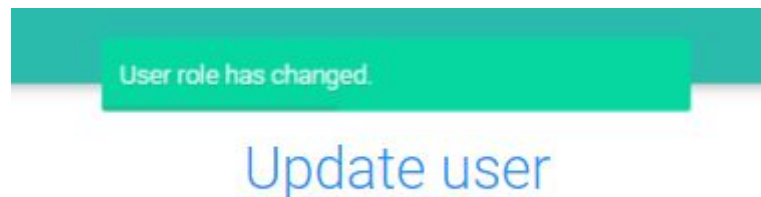
rita

rita@designer.com

Administrator

BACK SUBMIT

Step 4: As seen in the image below, we received an alert box that the database record of Rita is successfully updated with her role_name changed from that of user to admin and her role_id changed from 2 to 1.



user_id	fullname	email	user_password	role_id
100	rita	rita@designer.com	\$2b\$10\$K.0HwpsoPDGaB/atFBmmXOGTw4ceeg...	1

1.4.2 Where the flaw is found

There was no authorisation middleware to validate if the logged in user's role name is Admin or just a normal User. This means that by changing the user id from a normal User to that of an Admin, it will result in unauthorized entry as the normal User can be granted administrative privileges without the consent of the real Admin.

1.4.3 Recommendations

1. Middleware Functions for Authorization

The main cause of broken access control attacks is due to metadata manipulation. When a user tampers with the JSON Web Token (JWT) access control token, cookie or hidden field to elevate current privileges or misconfigured CORS, it allows unauthorized API access. One way of which this can be prevented is to use authorization middlewares to by default deny access to all functionalities and use role based mechanisms to validate login credentials. This involves extracting the user id and role name from the information encoded within the token to check its validity. Thus, omitting the need for local storage.

1.4.4 Solutions

1. Using Middleware Functions for Authorization

We created a new file called **verifyToken.js** within the **middleware** folder of **experimentsecuritywithcompetitionsystem**. The purpose of creating this file is to authenticate the validity of the user's token upon login.

Code Snippet, located in verifyToken.js file

```
const jwt = require('jsonwebtoken');
const JWT_SECRET = process.env.JWTKEY;
const auth = require('../services/authService');

async function verifyToken(req, res, next){
  console.log("verifyToken is Running")
  let blacklisted_tokens;
  // console.log(req.headers)
  let token = req.headers['authorization']; //retrieve authorization header's content
  // console.log(token);

  if(!token || !token.includes('Bearer ')){ //process the token
    res.status(403);
    return res.send({auth:'false', message:'Not authorized!'});
  }else{
    token=token.split('Bearer ')[1]; //obtain the token's value
    // console.log(token);
    try {
      blacklisted_tokens = await auth.getBlacklistedTokens();
      blacklisted_tokens = blacklisted_tokens.map((token)=>{ return token.jwt_token_blacklist });

    } catch (error) {
      let message = 'Server is unable to process your request.';
      return res.status(500).json({ message: error });
    }
    // console.log(blacklisted_tokens)

    if (blacklisted_tokens.length > 0) {
      for (tkn of blacklisted_tokens) {
        if (tkn == token) {
          res.status(403);
          return res.send({auth:'false', message:'Not authorized!'})
        }
      }
    }
  }

  jwt.verify(token, JWT_SECRET, function(err, decoded){ //verify token
    if(err){
      res.status(403);
      return res.end({auth:false, message:'Not authorized!'});
    }else{
      req.body.userId=decoded.user_id; //extracted from token
      req.role = decoded.role_name; //extracted from token
      // console.log(decoded);
      req.tokenExpiry = decoded.expiresOn;
      req.token = token;
      next();
      console.log("VALID TOKEN ACCEPTED")
    }
  });
}

module.exports = verifyToken;
```

We also modified the **routes.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** by including another POST API to execute the logout function after the user ended his session.

Code Snippet, located in routes.js file


```
router.post('/api/user/logout', verifyToken, authController.processLogout);
```

We also modified the **userController.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** by including an if-else loop to check the role name of the logged in user and a sanitization function to print the json result in such a way that it is harmless to the frontend. For example in html, all of the "<" should be escaped out as "<" so that the output will look messier to an attacker with malicious intentions. We also included an if statement to check if the logged in user has the role name of "Admin", so that they will not be allowed to access certain web functionalities as they are just normal users with no special privileges.

Code Snippet, located in userController.js file

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

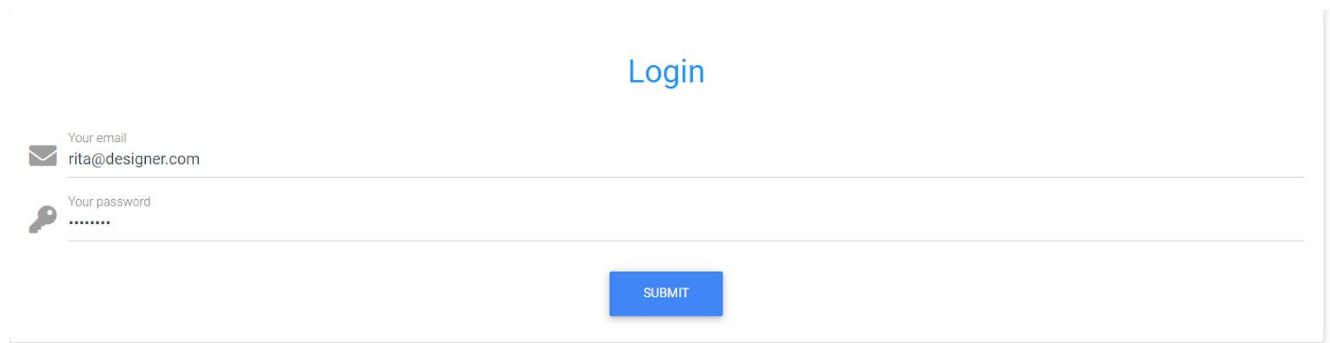
  if (req.role !== 'admin') {
    return res.status(403).json({message:'Not authorized!'})
  }

  try {
    let results = await userManager.getUserData(pageNumber, search);
    console.log('Inspect result variable inside processGetUserData code\n', results);
    if (results) {
      var jsonResponse = {
        'number_of_records': results[0].length,
        'page_number': pageNumber,
        'userdata': results[0],
        'total_number_of_records': results[2][0].total_records
      }
      //sanitise
      jsonResponse = validationFn.sanitizeResult(jsonResponse);
      res.status(200).json(jsonResponse);
      logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
      return
    }
  } catch (error) {
    let message = 'Server is unable to process your request.';
    res.status(500).json({
      message: error
    });
    logger.error(`${res.statusCode} || 500} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  }
}; //End of processGetUserData
```

1.4.5 Final Outcome (Evidence on the results after solving the flaw)

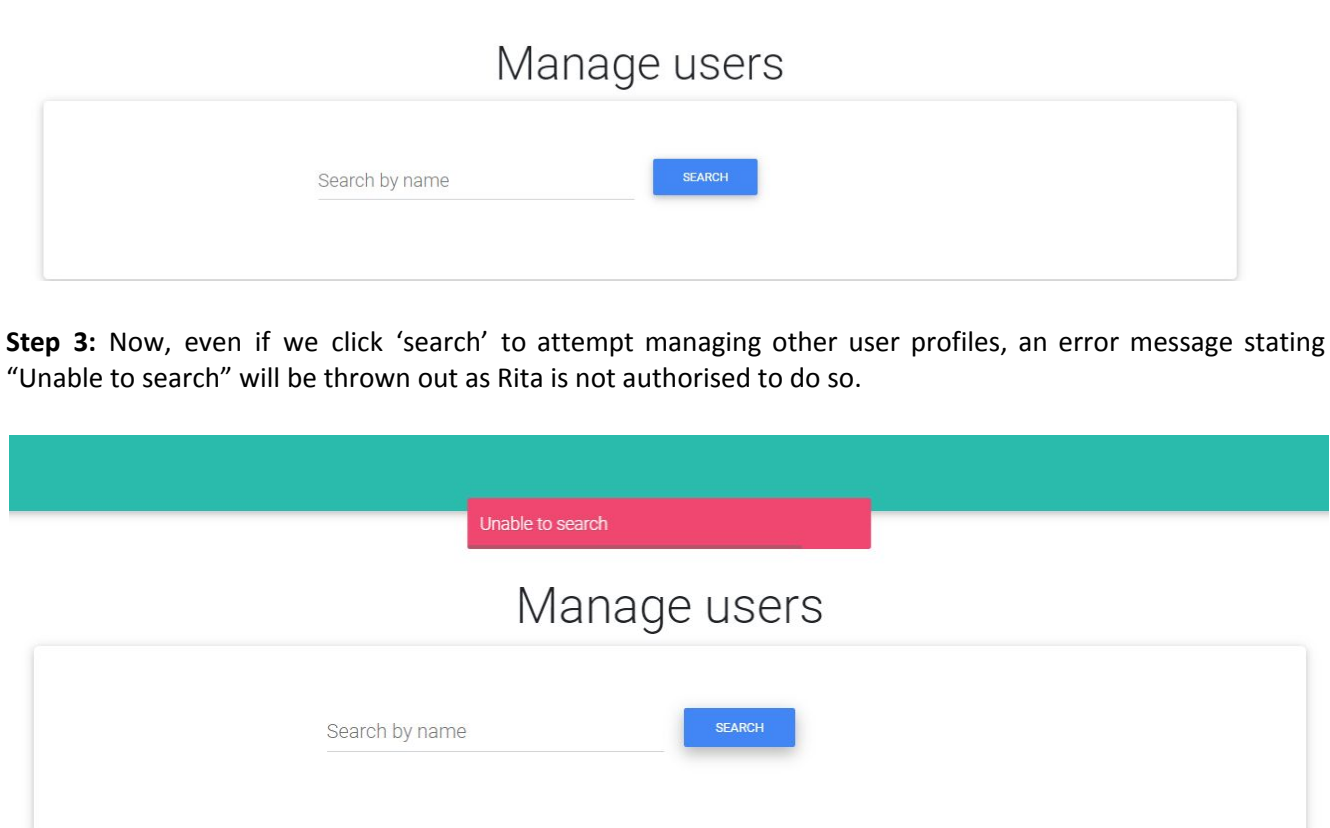
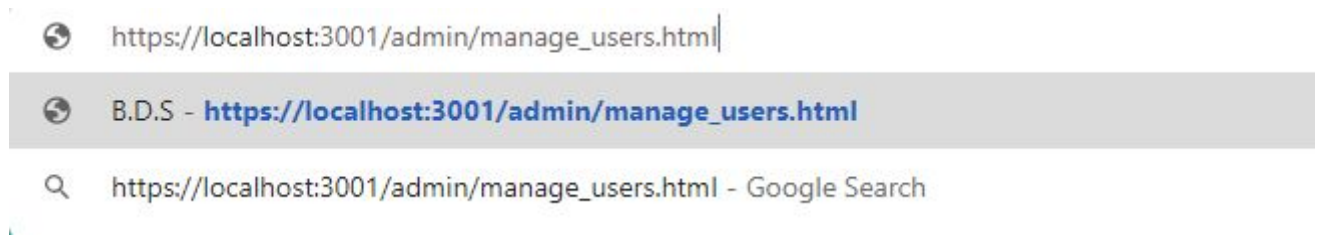
As seen in the image below, the code is fixed by using **Middleware Functions for Authorization** as the attacker is not able to perform any more broken access controls attacks by changing the user id to access as an admin and grant admin privileges to other normal users.

Step 1: We log in using Rita's email and password credentials.



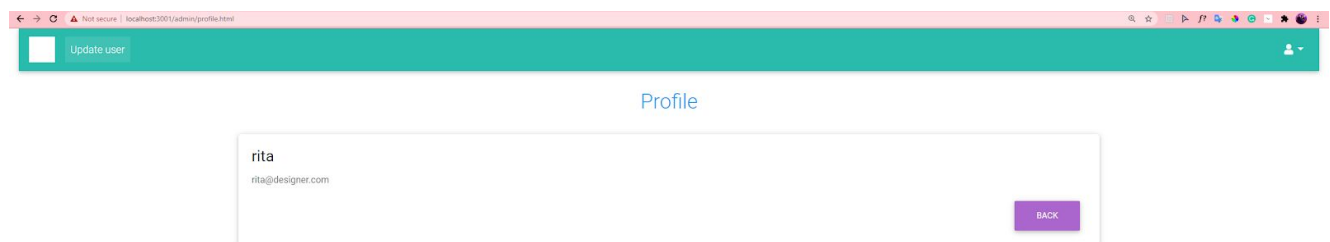
A login form with a white background and a light gray border. At the top center is the word "Login" in blue. Below it are two input fields: "Your email" with the text "rita@designer.com" and "Your password" with masked characters "*****". A blue "SUBMIT" button is at the bottom center.

Step 2: While still in Rita's account, we navigated to the **manage_users.html** page which should only be accessible by a logged in Admin.



A screenshot of the "Manage users" page. The page has a teal header. Below the header is a white box with the text "Search by name" and a blue "SEARCH" button. A red error message "Unable to search" is displayed in the center of the page.

Step 4: As shown in the image below, we navigated to the **admin/profile** page and saw that we are still in Rita's account even though we are under the admin webpage.



A screenshot of the "admin/profile" page. The page has a teal header with a white "Update user" button. Below the header is a white box with the text "rita" and "rita@designer.com". A purple "BACK" button is at the bottom right.

1.5 Cross-Site Scripting (XSS)

Definition: Cross-Site Scripting (XSS) is a type of web security vulnerability that allows an attacker to inject typically javascript client side scripts into the web pages of an unsuspecting user that are viewed by others. This form of attack is usually self-propagating and anyone who visits the site will be at risk. XSS attacks can be classified into two categories which are known as stored and reflected. The other one is known as DOM Based XSS. In general, the attacker sends an URL with malicious script and lures the victim to click on the url. The victim clicks on the URL and sends a request with malicious scripts to the website. The website will then send a request to the victim. Lastly, the attacker receives sensitive data from the victim.

Stored XSS attacks: The injected script is permanently stored on the target servers, such as in the database, message forum, visitor log, comment field and reviews. The malicious code is executed by a victims' browser, and the payload is stored on the victim's server. It is returned as part of the response from the HTML that a server sends. The attack is relatively more dangerous as it is self propagating, and anyone that now visits the site is at risk. To conduct such an attack, the attacker posts his submission with the malicious script which is stored in the database. The victim then sends a request to the website and loads the content containing malicious script from the database. The website sends the response back to the victim and the attacker receives sensitive data from there.

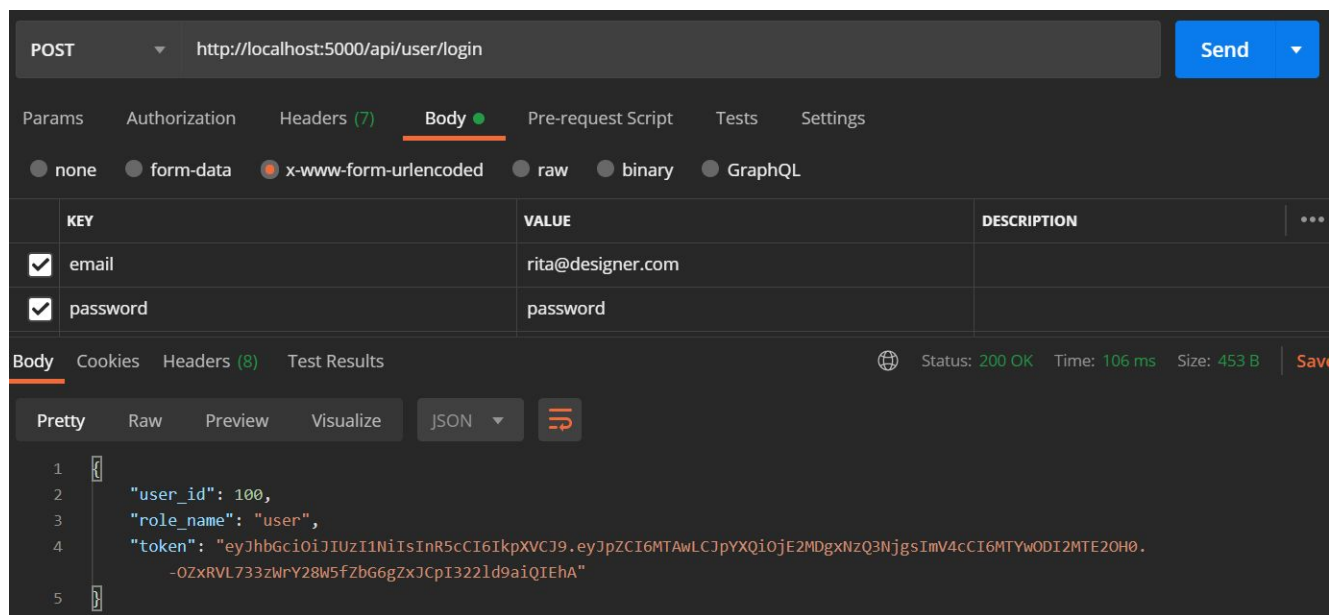
Reflected XSS attacks: The injected script is reflected off the server, for example an error message, search result, or other responses that included some or all of the input that was being sent to the server as part of the request. This attack is delivered to the victim through another route, such as in an email message or other websites. When the victim is tricked into clicking on a malicious link, submitting a specially crafted form, or even browsing to a malicious site. The injected code will then travel to the vulnerable site, which will then reflect the attack back to the user's browser. The browser then executes this malicious code because it came from a 'trusted' source. To conduct such an attack, the attacker sends a flawed URL to the victim, then the victim 'clicks' on the URL. The attack is sent to the application on behalf of the victim and 'reflected' back to the victim before being executed. The victim's browser sends personal information to the attacker.

DOM Based XSS attacks: The data flow never leaves the browser thus allowing the attacker to read malicious data through a page URL or HTML element. Cross-Site Scripting became possible when webpages output user supplied data in the HTML response without sanitising the data first.

	Vulnerability	Likelihood
Level	High	High
Explanation	A successful XSS exploit allows the attacker to access any cookies, session tokens and other sensitive information retained by the browser and used with that website or even rewrite the contents of a HTML page. In the case of this project, the design title can be updated to a javascript code so it means the attacker can also input more malicious code and it will also be accepted and displayed on all front end sites.	XSS flaws are commonly found as it can happen anywhere that a web application requires user input within the generated output without validating or encoding it. As the browser believes that the script came from a trusted source, it'll execute it.

1.5.1 Detailed example on how the flaw can be exploited

Step 1: We log in using Rita's email and password credentials.



When updating design, there is no validation for the design title and description inputs. This means that any text forms can be accepted so if an attacker enters a malicious code written in any programming language, the damage will be more severe than just a simple output of an alert box.

1. Create Middleware for Input Validation and Output Sanitization

The main cause of XSS attacks is due to malicious data included in dynamic content entering a web application through an untrusted source, most frequently a web request. When you don't properly validate user input and

sanitize user output, the client side will be affected as the browser will reflect the output exactly the way it was entered. One way of which this can be prevented is to use regular expressions to validate all user controlled input including form fields, GET and POST parameters, headers and cookies, and should only allow whitelisted characters, not special HTML characters. White list validation is appropriate for all input fields provided by the user and simply involves defining exactly what is authorized.

2. External Library Installation

We can install external libraries such as validator to help us sanitize strings in the application. Common functions such as `isAlphanumeric()` and `isEmail()` that validates emails and usernames respectively are already defined within the library and just need to be applied as an API.

1.5.4 Solutions

1. Create Middleware for Input Validation and Output Sanitization

For Input Validation

We also modified **validationFn.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem**. The purpose of this file is to ensure that the user's choice of design title and description fulfills the requirements based on the regular expressions.

Code Snippet, located in validationFn.js file

```
var validationFn = {
  validateUpdateDesign: function (req, res, next) {
    var designTitle = req.body.designTitle;
    var designDescription = req.body.designDescription;

    designTitlepattern = new RegExp('^[\w.-\s]+$');

    if (designTitlepattern.test(designTitle) && designTitlepattern.test(designDescription)) {
      next();
    } else {
      console.log("validation error")
      logger.error(`${res.statusCode} || 500} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
      res.status(500);
      res.send(`{'message': 'Invalid Design Title and/or Design Description'}`);
    }
  },
};
```

We also modified the **routes.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** by including another function to validate the update input when the user changed the design details.

Code Snippet, located in routes.js file

```
router.put('/api/user/design/', verifyToken, validationFn.validateUpdateDesign, userController.processUpdateOneDesign);
```

For Output Sanitisation

We also added a `sanitizeResult` function in **validationFn.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem**. The purpose of this function is to convert the string to a portable one that can be transmitted across any network that supports ASCII characters, thereby rendering codes in the frontend as harmless.

Code snippet, located at validationFn.js

```

    sanitizeResult: function (result) {
        if (result.filedata) {
            for (file of result.filedata) {
                file.design_title = validator.escape(file.design_title);
                file.design_description = validator.escape(file.design_description);
            }
        } else if (result.userdata) {
            if (result.userdata.length > 1) {
                for (user of result.userdata) {
                    user.fullname = validator.escape(user.fullname);
                    user.email = validator.escape(user.email);
                    user.role_name = validator.escape(user.role_name);
                }
            } else {
                result.userdata.fullname = validator.escape(result.userdata.fullname);
                result.userdata.email = validator.escape(result.userdata.email);
                result.userdata.role_name = validator.escape(result.userdata.role_name);
            }
        }
        return result;
    }
}

```

We also modified the **userController.js** file of **experimentsecuritywithcompetitionsystem** by calling the above mentioned function in the GET submission data API.

Code snippet, located at userController.js file

```

exports.processGetSubmissionData = async (req, res, next) => {
    console.log("processGetSubmissionData called")
    let pageNumber = req.params.pageNumber;
    let search = req.params.search;
    let userId = req.body.userId;

    try {
        let results = await fileDataManager.getFileData(userId, pageNumber, search);
        // console.log('Inspect result variable inside processGetSubmissionData code\n', results);
        if (results) {
            var jsonResponse = {
                "success": true,
                "message": "Get submission data successfully",
                "data": {
                    'number_of_records': results[0].length,
                    'page_number': pageNumber,
                    'filedata': results[0],
                    'total_number_of_records': results[2][0].total_records
                }
            }
            //sanitise
            jsonResponse = validationFn.sanitizeResult(jsonResponse);
            res.status(200).json(jsonResponse);
            logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
            return
        }
    } catch (error) {
        let message = 'Server is unable to process your request.';
        res.status(500).json({
            "message": message
        });
        logger.error(`${res.statusCode} || 500 - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
    }
}

```

We chose to **Create Middleware for Input Validation and Output Sanitization** instead of Installing External Libraries to stop the XSS attack as it simply requires us to validate the input with a Regular Expression without needing to install anything additional. As this was something we learnt before in class, we find it easier to apply and we validated all the necessary input fields.

1.5.5 Final Outcome (Evidence on the results after solving the flaw)

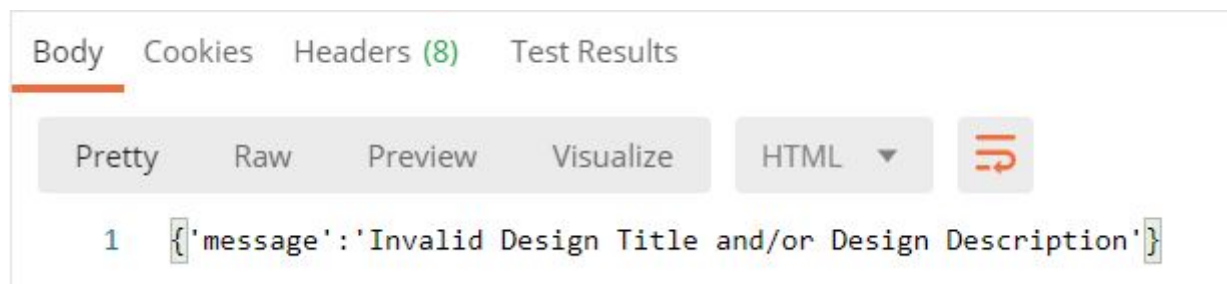
As seen in the image below, the code is fixed by using **Middleware for Input Validation and Output Sanitization** as the attacker is not able to perform anymore XSS attacks since the design title cannot be updated despite the previous input of Javascript code and the error message is thrown out accordingly via an alert box.

Evidence for Input Validation

Step 1: We made use of Rita's Design 6 for the XSS attack scenario by changing its Design Title to `<script>alert('XSS')</script>` which is typed in Javascript language.

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> email	rita@designer.com	
<input checked="" type="checkbox"/> password	password	
<input checked="" type="checkbox"/> field	103	
<input checked="" type="checkbox"/> designTitle	<script>alert('XSS')</script>	
<input checked="" type="checkbox"/> designDescription	#	
Key	Value	Description

Step 2: As shown in the image below, an error message will be thrown out and it does not specifically say which field is invalid so it will take a longer time for the attacker to debug what was the issue.

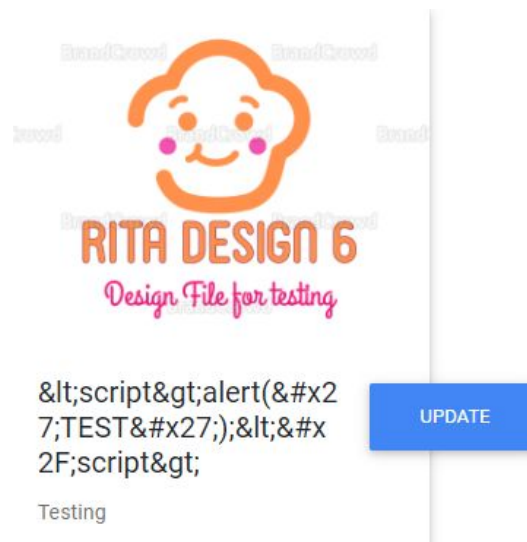


Evidence for Output Sanitisation

Step 1: Update the Design Title and Description by doing a cross site attack which involves typing `<script>alert('XSS')</script>` and 'testing' in the input fields respectively.



Step 2: We notice that after sanitizing the output code, the malicious codes are printed out in a messy string and will be rendered as harmless on the frontend as it will not be considered as Javascript code.



1.6 Insufficient Logging & Monitoring

Definition: A type of web security vulnerability that allows an attacker to conduct their suspicious activities without being monitored by the developer. When errors are not tracked by the developers, the hackers can easily get away with their attacks and if an application has many errors, it will be more prone to security vulnerabilities.

	Vulnerability	Likelihood
Level	Moderate	Low
Explanation	A successful insufficient logging and monitoring exploit allows the attacker to probe for vulnerabilities and identify a breach by relying on the lack of monitoring and timely response. In the case of this project, there is insufficient console.log() to output the problems so attacks may get away with their exploits. In the case of this project, the user does not have a clear idea of the level of programming error so he may not execute necessary actions to fix it.	Insufficient logging and monitoring flaws are commonly found when auditable events such as APIs, logins and high-value transactions are not logged, or only stored locally or when warnings and errors generate no, inadequate or unclear log messages.

1.6.1 Detailed example on how the flaw can be exploited

There are currently no loggers being implemented in this project except to warn developers of errors. However, the error responses are printed in a variety of ways such as HTTP status codes or wrong messages so developers which are not used to that form of output may overlook the level of danger the application is in and will not execute appropriate actions to fix the flaw.

1.6.2 Where the flaw is found

The error is seen to be output as a message to the developer and this means that if an attacker gets hold of the code and runs the application, he will know which parts are vulnerable and use it to his advantage. Having no proper logger to display clear records can also lead to overlooking of errors and the error may end up not getting fixed.

Code Snippet, located in userController.js file

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

1.6.3 Recommendations

1. Implement logging and monitoring

The main cause of insufficient logging and monitoring is due to the lack of timely responses which allow the attacker to perform their attacks without being watched. When logs are not created to detect auditable events, the web application will not be able to be alerted to active attacks in real time. One way of which this can be prevented is to install Winston, a universal logging library that utilizes transport configured at different levels to record entries. In terms of log formatting, it supports flexibility as we can customize our own logging levels in addition to the predefined ones such as 'npm' and 'cli'. As no transports are set by default, we can enable the number of transports by simply calling the add() and remove() methods. Overall, the logs are stored locally with sufficient user context and generated in a format that can be easily consumed by a centralized log management. The data stored in the logs include the timestamp of log entry, HTTP status code and data from GET and POST request endpoints.

1.6.4 Solutions

1. Implement logging and monitoring

Step 1: We installed Winston within the **experimentsecuritywithcompetitionsystem** folder using the command "npm install --save winston".

Step 2: We created a **loggingService.js** in the services folder of **experimentsecuritywithcompetitionsystem**.

Code Snippet, located in loggingService.js file


```
const { createLogger, format, transports } = require('winston');

//create logger
module.exports = createLogger({
  transports: [
    new transports.File({
      filename: 'logs/monitoring.log',
      format: format.combine(
        format.timestamp({ format: 'MMM-DD-YYYY HH:mm:ss' }),
        format.align(),
        format.printf((info) => `${info.level}: ${[info.timestamp]}: ${info.message}`)
      ),
    }),
    new transports.File({
      filename: 'logs/error.log',
      format: format.combine(
        format.timestamp({ format: 'MMM-DD-YYYY HH:mm:ss' }),
        format.align(),
        format.printf((info) => `${info.level}: ${[info.timestamp]}: ${info.message}`)
      ),
      level: 'error',
    }),
  ],
});
```

Step 3: We modified the **userController.js** file of **experimentsecuritywithcompetitionsystem** by adding the logger call line into the code to return the relevant information such as IP address, API method and HTTP status code, so that the exact error can be recorded and debugged later on.

Code Snippet, located in userController.js file

```
exports.processGetSubmissionData = async (req, res, next) => {
  console.log("processGetSubmissionData called")
  let pageNumber = req.params.pageNumber;
  let search = req.params.search;
  let userId = req.body.userId;

  try {
    let results = await fileDataManager.getFileData(userId, pageNumber, search);
    // console.log('Inspect result variable inside processGetSubmissionData code\n', results);
    if (results) {
      var jsonResult = {
        "success": true,
        "message": "Get submission data successfully",
        "data": {
          'number_of_records': results[0].length,
          'page_number': pageNumber,
          'filedata': results[0],
          'total_number_of_records': results[2][0].total_records
        }
      }
      //sanitise
      jsonResult = validationFn.sanitizeResult(jsonResult);
      res.status(200).json(jsonResult);
      logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
      return
    }
  } catch (error) {
    let message = 'Server is unable to process your request.';
    res.status(500).json({
      "message": message
    });
    logger.error(`${res.statusCode || 500} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  }
}
```

Step 4: We created two logs named **monitoring.log** which is used to record all logs, including the error logs, and **error.log** which is used to record only the error logs.

Code Snippet , located in monitoring.log file

```

info: Dec-31-2020 21:56:31: 200 - OK - /api/user/login - POST - ::1
info: Dec-31-2020 21:56:33: 200 - OK - /api/user/process-search-design/1/ - GET - ::1
info: Dec-31-2020 21:56:47: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:56:58: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:57:35: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:57:44: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:57:46: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:57:57: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:57:59: Server started and running on https://localhost:5000/
info: Dec-31-2020 21:58:07: 200 - OK - /api/user/logout - POST - ::1

```

Code snippet, located in error.log file

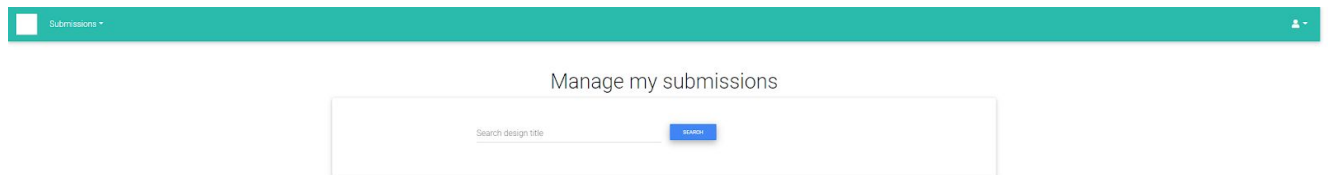
```

error: Dec-30-2020 14:20:12: 500 - Internal Server Error - /api/user/process-search-design/1/ - GET - ::1
error: Dec-30-2020 23:28:18: 500 - Internal Server Error - /api/user/process-submission - POST - ::1
error: Dec-30-2020 23:34:12: 500 - Login Failed (Catch) - /api/user/login - POST - ::1
error: Dec-30-2020 23:34:37: 500 - Login Failed (Catch) - /api/user/login - POST - ::1
error: Dec-30-2020 23:36:25: 500 - Login Failed (Catch) - /api/user/login - POST - ::1
error: Dec-31-2020 21:24:17: 200 - undefined - /api/user/process-submission - POST - ::1
error: Dec-31-2020 21:25:12: 200 - Submit Failed - /api/user/process-submission - POST - ::1
error: Dec-31-2020 21:27:12: 200 - Register Failed - /api/user/register - POST - ::1
error: Dec-31-2020 21:27:36: 200 - Register Failed - /api/user/login - POST - ::1
error: Dec-31-2020 21:27:59: 200 - Login Failed - /api/user/login - POST - ::1
error: Dec-31-2020 21:31:24: 200 - Login Failed - /api/user/login - POST - ::1
error: Dec-31-2020 21:31:41: 500 - Login Failed (catch) - /api/user/login - POST - ::1
error: Dec-31-2020 21:55:58: 403 - Login Failed (Catch) - /api/user/process-search-design/1/ - GET - ::1
error: Dec-31-2020 21:58:15: 403 - User not authorized - /api/user/process-search-design/1/ - GET - ::1

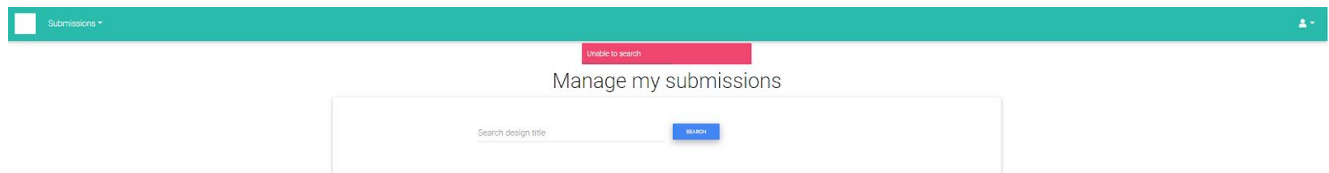
```

1.6.5 Final Outcome (Evidence on the results after solving the flaw)

Step 1: Without logging into any account, navigate to the `manage_submission.html` page and click on the search button.



Step 2: We noticed that an error message printed “unable to search” in the form of an alert box is displayed.



Step 3: We navigated back to the terminal in Visual Studio Code and saw that the error has been recorded.

```

95 error: Dec-31-2020 21:58:15: 403 - User not authorized - /api/user/process-search-design/1/ - GET - ::1

```

1.7 Others (Using Components with Known Vulnerabilities)

Definition: A type of web security vulnerability that allows an attacker to exploit protected assets and breach of data due to limited knowledge of software versions and irregular platform upgrades.

	Vulnerability	Likelihood
Level	Moderate	High
Explanation	A successful usage of components with known vulnerabilities exploit allow the attacker to identify a breach by relying on the lack of knowledge of dependencies versions. In the case of this project, several modules listed in package.json were left unused, as such attackers may use automated tools to find these unpatched configurations and run them using the same privileges as the web application level, resulting in serious flaws like intentional coding errors.	Insufficient usage of components with unknown components are commonly found when developers do not test the compatibility of updated libraries, do not subscribe to reliable security bulletins, do not secure configurations or do not upgrade underperforming platforms in a timely fashion, which will cause the overall software to become unsupported in current runtime environments.

1.7.1 Detailed example on how the flaw can be exploited

There are several undefined modules in package.json which are not used in the codes. As such, if an attacker knows which redundant modules are installed but not used in the code, he can implement it with other malicious codes.

1.7.2 Where the flaw is found

The code calls for an uninstalled “validator” dependency which is another component that is used intentionally despite knowing its vulnerability.

Code Snippet, located in validationFn.js file

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

1.7.3 Recommendations

1. Install OWASP dependency-check

The main cause of using components with known vulnerabilities is due to a lack of knowledge about installed dependencies, which may degrade a web application. By installing this Software Composition Analysis tool, the assigned Common Platform Enumeration for each dependency can be identified and publicly disclosed vulnerabilities contained within a project’s dependency will be detected more easily. This will aid developers in removing unused modules rather than leaving it within the package json, and allow the attacker to exploit it.

1.7.4 Solutions

1. Install OWASP dependency-check

Step 1: We noted that the node.js version used must be 14 or above so if our version is below that, we have to upgrade it. We checked our current version by running the command “node -v” and the version is suitable.

```
C:\Users\vicki\Documents\SP\Y2S2\ESDE\CA1\codes>node -v
v14.15.3
```

Step 2: We installed OWASP dependency-check within the **experimentsecuritywithcompetitionsystem** folder using the command “npm install --save dependency-check”.

```
C:\Users\vicki\Documents\SP\Y2S2\ESDE\CA1\codes\experimentsecuritywithcompetitionsystem>npm install --save dependency-check
npm WARN! experimentsecuritywithdesignerstudio@1.0.0 No repository field.
npm WARN! optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\fsevents):
npm WARN! notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})
+ dependency-check@4.1.0
added 10 packages from 6 contributors and audited 1198 packages in 7.06s

39 packages are looking for funding
  run `npm fund` for details

found 8 low severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
```

Step 3: We run the command “dependency-check ./package.json” to check for unregulated modules within the package.json file.

```
C:\Users\vicki\Documents\SP\Y2S2\ESDE\CA1\codes\experimentsecuritywithcompetitionsystem>dependency-check ./package.json
Fail! Modules in package.json not used in code: @babel/core, axios, babel-preset-es2015, cookie-parser, cssnano, del, express-session, gulp, gulp-autoprefixer, gulp-babel, gulp-concat, gulp-css, gulp-minify-css, gulp-postcss, gulp-replace, gulp-sass, gulp-sourcemaps, gulp-uglify, gulp-uglify-es, gulp-watch, multier, node, normalize-scss, pug, run-sequence, session-file-store, nodemon
Fail! Dependencies not listed in package.json: validator
```

Step 4: We ran the command “npm uninstall <module_name>” for the above mentioned modules to remove them permanently from the package.json file and “npm install --save validator” to install the missing module.

1.7.5 Final Outcome (Evidence on the results after solving the flaw)

As seen in the image below, the code is fixed by installing **OWASP dependency-check** as all the modules listed in the package.json are called for and used in the codes so there are no components that the developers are unaware of and the attacker will not be able to exploit on this vulnerability.

```
C:\Users\vicki\Documents\SP\Y2S2\ESDE\CA1\codes\experimentsecuritywithcompetitionsystem>dependency-check ./package.json --missing
C:\Users\vicki\Documents\SP\Y2S2\ESDE\CA1\codes\experimentsecuritywithcompetitionsystem>
```

2. Code Review

2.1 Inconsistent JSON Response Structure

As this system is developed by more than one student, each individual has their own unique way of printing JSON responses. Thus, the structure output will vary across the project where for some APIs, the console.log is for results[0] while for another, the console.log is for an array called jsonResult. Regardless of the APIs, the most common data structure should include the HTTP status code and the JSON data body. For error scenarios, there should be an uniform definition where the HTTP status code of 500, an error message and the json data body is returned to describe what went wrong. For success scenarios, the HTTP status code of 200, message and only the relevant rows from the database should be returned to prevent attackers from getting more than the necessary information. Therefore, we referred to the other GET APIs and changed the PUT APIs in **UserController.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** to print out the array of data called jsonResult, instead of just results[0] or a success message which does not provide as meaningful response as the other GET and POST APIs.

For Success Scenario (RES 200)

Initially for update of design, the successful response printed out is just “design update successfully”.

PUT ▼ https://localhost:5000/api/user/design Send Save ▼

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies Code

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> fileId	104	
<input checked="" type="checkbox"/> designTitle	Design 4	
<input checked="" type="checkbox"/> designDescription	four	
<input type="checkbox"/>	Design 4	
<input type="checkbox"/>	four	
Key	Value	Description

Body Cookies Headers (8) Test Results Status: 200 OK Time: 263 ms Size: 297 B Save Response

Pretty Raw Preview Visualize JSON ≡

```

1 {
2   "message": "Completed update"
3 }
```

Code Snippet, located in userController.js file

```

exports.processUpdateOneDesign = async(req, res, next) => {
  console.log('processUpdateOneFile running');
  //Collect data from the request body
  let fileId = req.body.fileId;
  let designTitle = req.body.designTitle;
  let designDescription = req.body.designDescription;

  try {
    results = await userManager.updateDesign(fileId, designTitle, designDescription);
    console.log(results);
    res.status(200).json({ message: 'Completed update' });
    logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  } catch (error) {
    console.log('processUpdateOneUser method : catch block section code is running');
    console.log(error, '=====');
    res.status(500).json({ 'message': 'Unable to complete update operation' });
    logger.error(`${res.statusCode || 500} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  }
}

}; //End of processUpdateOneDesign

```

However, after changing the API response to return the relevant input fields, the response is more meaningful as we can see the updated design title and description for that file id.

Code snippet, located in UserController.js file

```

exports.processUpdateOneDesign = async (req, res, next) => {
  console.log('processUpdateOneFile running');
  //Collect data from the request body
  let fileId = req.body.fileId;
  let designTitle = req.body.designTitle;
  let designDescription = req.body.designDescription;

  try {
    results = await userManager.updateDesign(fileId, designTitle, designDescription);
    console.log(results);

    if (results) {
      var jsonResult = {
        "success": true,
        "message": "design updated successfully",
        "data": {
          'file_id': fileId,
          'new_design_title': designTitle,
          'new_design_description': designDescription
        }
      }
    }

    res.status(200).json(jsonResult);
    // res.status(200).json({ message: 'Completed update' });
    logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  } catch (error) {
    console.log('processUpdateOneUser method : catch block section code is running');
    console.log(error, '=====');
    res.status(500).json({ 'message': 'Unable to complete update operation' });
    logger.error(`${res.statusCode} || 500} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
    return
  }
}; //End of processUpdateOneDesign

```

PUT https://localhost:5000/api/user/design Send Save

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> fileId	104	
<input checked="" type="checkbox"/> designTitle	Design 4	
<input checked="" type="checkbox"/> designDescription	four	
<input type="checkbox"/>	Design 4	
<input type="checkbox"/>	four	
Key	Value	Description

Body Cookies Headers (8) Test Results Status: 200 OK Time: 35 ms Size: 411 B Save Response

Pretty Raw Preview Visualize JSON ≡

```

1 {
2   "success": true,
3   "message": "design updated successfully",
4   "data": {
5     "file_id": "104",
6     "new_design_title": "Design 4",
7     "new_design_description": "four"
8   }
9 }

```

For Error Scenario (RES 500)

Initially for login of users, the error response printed out is just “null” and we have to refer back to the terminal in Visual Studio Code to see the full error message printed out.

POST http://localhost:5000/api/user/login Send Save

Params Authorization Headers (7) **Body** Pre-request Script Tests Settings Cookies Code

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	email	rita@designer.com			
<input checked="" type="checkbox"/>	password	password1			
	Key	Value	Description		

Body Cookies Headers (8) Test Results Status: 500 Internal Server Error Time: 230 ms Size: 302 B Save Response

Pretty Raw Preview Visualize JSON ↺

```

1  {
2    "message": null
3  }

```

Code Snippet, located in authController.js file

```

exports.processLogin = async (req, res, next) => {
  let email = req.body.email;
  let password = req.body.password;

  try {
    let results = await auth.authenticate(email);
    console.log("GOT RESULTS")
    if (results.length == 1) {
      if ((password == null) || (results[0] == null)) {
        res.status(500).json({ "message": 'login failed' });
        logger.error(`${res.statusCode} || 500 - Login Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      }
      if (bcrypt.compareSync(password, results[0].user_password) == true) {
        console.log("CORRECT PASSWORD")
        let today = new Date()
        // console.log(new Date())
        today.setDate(today.getDate() + 1);
        // console.log(today.toISOString().slice(0, 19).replace('T', ' ')); //timestamp of token creation/Login
        let data = {
          // user_id: results[0].user_id,
          role_name: results[0].role_name,
          token: jwt.sign({ user_id: results[0].user_id, role: results[0].role_name, expiresOn: today.toISOString().slice(0, 19).replace('T', ' ') }, config.JWTKey, {
            expiresIn: 86400 //Expires in 24 hrs
          })
        }; //End of data variable setup

        res.status(200).json({ 'userdata': data });
        logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      } else {
        res.status(500).json({ 'message': 'error' });
        logger.error(`${res.statusCode} || 500 - Login Failed (Catch) - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      }
    } // end
  } //end
} catch (error) {
  res.status(500).json({ "message": "error" });
  logger.error(`${res.statusCode} || 500 - Login Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
  return
} //end of try/catch
};

```

However, after changing the API response to return the relevant error message, the response is more meaningful as we can see it directly in Postman.

POST https://localhost:5000/api/user/login Send Save

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies Code

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> email	rita@designer.com	
<input checked="" type="checkbox"/> password	111	
Key	Value	Description

Body Cookies Headers (8) Test Results Status: 500 Internal Server Error Time: 400 ms Size: 305 B Save Response

Pretty Raw Preview Visualize **JSON**

```

1 {
2   "message": "error"
3 }
```

Initially for registration of new users, the error response printed out is just “unable to complete registration” which is literally common sense.

POST http://localhost:5000/api/user/register Send Save

Params Authorization Headers (7) **Body** Pre-request Script Tests Settings Cookies Code

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> fullName	rita	
<input checked="" type="checkbox"/> email	rita@designer.com	
<input checked="" type="checkbox"/> password	password	

Body Cookies Headers (8) Test Results Status: 500 Internal Server Error Time: 116 ms Size: 331 B Save Response

Pretty Raw Preview Visualize **JSON**

```

1 {
2   "message": "Unable to complete registration"
3 }
```

However, to see the full error message printed out, we had to refer back to the terminal in Visual Studio Code.

```

processRegister running
rita rita@designer.com $2b$10$RTT023n7KkJJzK9hQ7c1l055jnuRF1yI4KbUM060F.GH2LSMalouW
processRegister method : catch block section code is running
Error: ER_DUP_ENTRY: Duplicate entry 'rita@designer.com' for key 'user.email'
    at Query.Sequence._packetToError (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\sequences\Sequence.js:47:14)
    at Query.ErrorPacket (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\sequences\Query.js:79:18)
    at Protocol._parsePacket (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Protocol.js:291:23)
    at Parser._parsePacket (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Parser.js:433:10)
    at Parser.write (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Parser.js:43:10)
    at Protocol.write (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Protocol.js:38:16)
    at Socket.<anonymous> (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\Connection.js:88:28)
    at Socket.<anonymous> (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\Connection.js:526:10)
    at Socket.emit (events.js:314:20)
    at addChunk (_stream_readable.js:298:12)
    at Protocol._enqueue (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Protocol.js:144:48)
    at PoolConnection.query (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\Connection.js:198:25)
    at C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\src\services\userService.js:13:32
    at Handshake.onConnect (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\pool.js:64:7)
    at Handshake.<anonymous> (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\Connection.js:526:10)
    at Handshake._callback (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\Connection.js:488:16)
    at Handshake.Sequence.end (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\sequences\Sequence.js:83:24)
    at Handshake.Sequence.OkPacket (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\sequences\Sequence.js:92:8)
    at Protocol._parsePacket (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Protocol.js:291:23)
    at Parser._parsePacket (C:\Users\Vicki\Downloads\backend_frontend_project_files\experimentsecuritywithcompetitionsystem\node_modules\mysql\lib\protocol\Parser.js:433:10) {
  code: 'ER_DUP_ENTRY',
  errno: 1062,
  sqlMessage: 'Duplicate entry 'rita@designer.com' for key 'user.email'',
  sqlState: '23000',
  index: 0,
  sql: 'INSERT INTO user ( fullname, email, user_password, role_id) VALUES ('rita','rita@designer.com','$2b$10$RTT023n7KkJJzK9hQ7c1l055jnuRF1yI4KbUM060F.GH2LSMalouW',2) '
}
```

Code Snippet, located in authController.js file


```

exports.processRegister = async (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');
      res.status(500).json({ 'message': 'Unable to complete registration' });
      logger.error(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
      return
    } else {
      try {
        results = await user.createUser(fullName, email, hash);
        console.log(results);

        res.status(200).json({
          "success": true,
          "message": 'Completed registration',
          "data": {}
        });
        logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      } catch (error) {
        console.log('processRegister method : catch block section code is running');
        console.log(error, '=====');
        res.status(500).json({ 'message': 'Unable to complete registration' });
        logger.error(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      }
    }
  });
}; //End of processRegister

```

However, after changing the API response to return the relevant error message, the response is more meaningful as we can see it directly in Postman.

The screenshot shows a Postman interface for a POST request to `https://localhost:5000/api/user/register`. The request body is raw JSON with the following data:

KEY	VALUE	DESCRIPTION
fullName	David Tan	
email	david@designer.com	
password	111	
Key	Value	Description

The response status is **500 Internal Server Error** with a time of 1211 ms and size of 341 B. The response body is:

```

{
  "message": "Invalid Name and/or email and/or password"
}

```

2.2 Poor Error Handling

The use of nested callbacks are seen inside the project. As a result, attackers who use automated tools will “hang” the hosted project for several days. We noticed that the REST API code which handles the pagination feature has more stable error handling. Therefore, we replaced the callbacks in **authController.js** file and **authService.js** file within the **src** folder of **experimentsecuritywithcompetitionsystem** with the **async** and **await** technique that enable promise-based behavior through the use of **try** and **catch** blocks. By doing so, we avoid the need to explicitly configure promise chains and wait for other functions to finish executing before fulfilling the callback.

Original Code Snippet, located in authController.js file (BEFORE)

```
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: 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 password 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
};
```

Fixed Code Snippet, located in authController.js file (AFTER)

```

exports.processLogin = async (req, res, next) => {
  let email = req.body.email;
  let password = req.body.password;

  try {
    let results = await auth.authenticate(email);
    console.log("GOT RESULTS")
    if (results.length == 1) {
      if ((password == null) || (results[0] == null)) {
        res.status(500).json({ "message": 'login failed' });
        logger.error(`${res.statusCode} - Login Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      }
      if (bcrypt.compareSync(password, results[0].user_password) == true) {
        console.log("CORRECT PASSWORD")
        let today = new Date()
        // console.log(new Date())
        today.setDate(today.getDate() + 1);
        let data = {
          "success": true,
          "message": "design updated successfully",
          "data": {
            role_name: results[0].role_name,
            token: jwt.sign({ user_id: results[0].user_id, role: results[0].role_name, expiresOn: today.toISOString().slice(0, 19).replace('T', ' '), config.JWTKey, {
              expiresIn: 86400 //Expires in 24 hrs
            })
          }
        }
        res.status(200).json(data);
        logger.info(`${res.statusCode} - ${res.statusMessage} - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      } else {
        res.status(500).json({ 'message': 'error' });
        logger.error(`${res.statusCode} - Login Failed (Catch) - ${req.originalUrl} - ${req.method} - ${req.ip}`);
        return
      }
    } // end
  } //end
} catch (error) {
  res.status(500).json({ "message": "error" });
  logger.error(`${res.statusCode} - Login Failed - ${req.originalUrl} - ${req.method} - ${req.ip}`);
  return
} //end of try/catch
};

```

Original Code Snippet, located in authService.js file (BEFORE)

```

module.exports.authenticate = (email, callback) => {
  pool.getConnection((err, connection) => {
    if (err) {
      if (err) throw err;
    } else {
      try {
        connection.query(`SELECT user.user_id, fullname, email, user_password, role_name, user.role_id
        FROM user INNER JOIN role ON user.role_id=role.role_id AND email='${email}'`, {}, (err, rows) => {
          if (err) {
            if (err) return callback(err, null);
          } else {
            if (rows.length == 1) {
              console.log(rows);
              return callback(null, rows);
            } else {
              return callback('Login has failed', null);
            }
          }
        });
        connection.release();
      });
    } catch (error) {
      return callback(error, null);
    }
  }); //End of getConnection
} //End of authenticate

```

Fixed Code Snippet, located in authService.js file (AFTER)

```

module.exports.authenticate = (email) => {
  return new Promise((resolve, reject) => {
    pool.getConnection((err, connection) => {
      if (err) {
        console.log('Database connection error ', err);
        resolve(err);
      } else {
        connection.query(`SELECT user.user_id, fullname, email, user_password, role_name, user.role_id
        FROM user INNER JOIN role ON user.role_id=role.role_id AND email= ?`, [email], (err, results) => {
          if(err) {
            reject(err);
          } else {
            if(results.length == 1){
              console.log(results);
              resolve(results);
            } else {
              reject('Login has failed');
            }
          }
        });
        connection.release();
      }
    });
  });
}); //End of new Promise object creation
} //end of authenticate

```