Secure Coding (ST2515)

Assignment

Web Application Security Assessment Report

Written By:

Andrew Poh

DISM/2B/01

# 

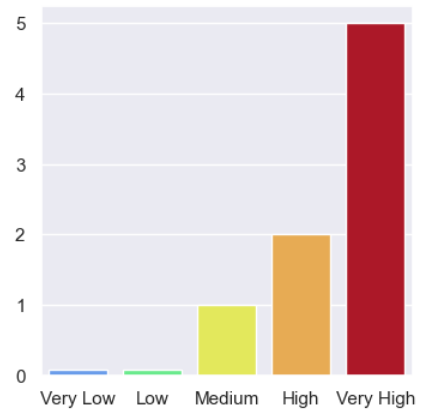
# Executive Summary

In this assignment, I have been engaged to secure a web application. I have been provided with the web application source code and will test the web program as well as analyse the source code in order to discover security vulnerabilities within the program.

This report will identify the webpages and/or source codes that present a security vulnerability. It will show an example of exploitation of that vulnerability if applicable and suggest a way to resolve them.

This document details the scope of testing conducted, a summary of findings which includes the severity of significant findings. The findings are categorised into 4 categories and contain 2 vulnerabilities highlighted per category as well as additional recommendations. The ‘Findings’ section will be a detailed section aimed at a technical audience which covers the exploitation and resolution of the various vulnerabilities discovered.

Below is a visualization of the web application vulnerabilities found by impact. The level of impact is defined in [Appendix A](#_37jnhhwry1t3).



[**Introduction 1**](#_y9duftz0z2v7)

[Overview 1](#_4ddfjcwqyz1a)

[Scope 1](#_h3t5k0rrlvtd)

[Findings 1](#_qul9ppkrk76k)

[**Findings 3**](#_lyjjbxv4trkt)

[SQL Injection 4](#_qmjjod2oafta)

[Vulnerability 1: Delete Reviews API SQLi (Detailed) 4](#_nlw8s79803jb)

[Vulnerability 2: Add Reviews API SQLi (Brief) 12](#_ie5s22bso0e2)

[Additional Recommendations 13](#_qokwvdwtxzh1)

[Cross-Site Scripting (XSS) 14](#_hyrhq027k4q1)

[Vulnerability 1: Product Review Input XSS (Detailed) 14](#_1dpohz25ax29)

[Vulnerability 2: Register User Input XSS (Brief) 25](#_8chl7sjnjvmj)

[Additional Recommendations 31](#_gpr8j6l6gavh)

[Broken Access Control 32](#_copuf37sfauv)

[Vulnerability 1: Non-Admins can Add Products (Detailed) 32](#_ckx3qvb9wuzz)

[Vulnerability 2: Non-Admins can Delete Products (Brief) 37](#_ryxvpzmq11b1)

[Additional Recommendations 38](#_w78pdk7j1z4f)

[Cryptographic Failures 39](#_gzkkk9s786gb)

[Vulnerability 1: Lack of Network Traffic Encryption (Detailed) 39](#_ugjslww49ww1)

[Vulnerability 2: Stored Passwords are not Hashed (Brief) 46](#_j5nt1u1kk54o)

[Additional Recommendations 51](#_r7z6299fb3yo)

[**Tools 52**](#_evb6nrmwrns)

[**Conclusion 53**](#_tfgash20o4r1)

[**Appendix A 54**](#_37jnhhwry1t3)

# 

# Introduction

## Overview

This report documents the findings for the Web Application Security Assessment application provided in the assignment. The purpose of the engagement was to utilise exploitation techniques in order to identify and validate potential vulnerabilities across all systems within scope.

## Scope

The scope of the assessment is limited to the web application frontend, backend and SQL database provided. The following web server hosts will be used for testing:

* localhost:3001 (Front-end)
* localhost:8081 (Back-end)

## Findings

The following section will contain detailed findings are grouped under the following categories:

* SQL Injection (SQLi)
* Cross-Site Scripting (XSS)
* Broken Access Control (BAC)
* Cryptographic Failures (CF)

Below is a summary of the vulnerabilities and their severity rating out of 5:

| **Category** | **Vulnerability** | **Impact** | **Ease of Exploitation** | **Severity** |
| --- | --- | --- | --- | --- |
| SQLi | Delete Reviews SQLi | High | High | 5 |
| Add Reviews SQLi | High | High | 5 |
| XSS | Product Review XSS | High | High | 5 |
| Register User XSS | High | High | 5 |
| BAC | Non-Admins can Add Products | Medium | High | 4 |
| Non-Admins can Delete Products | Medium | High | 4 |
| CF | Lack of Network Traffic Encryption | High | High | 5 |
| Stored Passwords are not Hashed | High | Medium | 3 |

The severity ratings in the above table were calculated using the severity matrix found in [Appendix A](#_37jnhhwry1t3).

As you can see many of the vulnerabilities are actually quite severe. The following section will highlight how to exploit 1 vulnerability from each category as well as how to fix them.

# 

# Findings

In this section, the 4 categories will consist of 2 vulnerabilities found each; 1 detailed and 1 brief. For the detailed vulnerability, I will go through:

1. Exploitation of the vulnerability
2. Vulnerable Code
3. Remediated Code
4. Testing if the exploitation has been prevented.

For the brief vulnerability, I will only cover:

1. Vulnerable Code
2. Remediated Code

## 

## SQL Injection

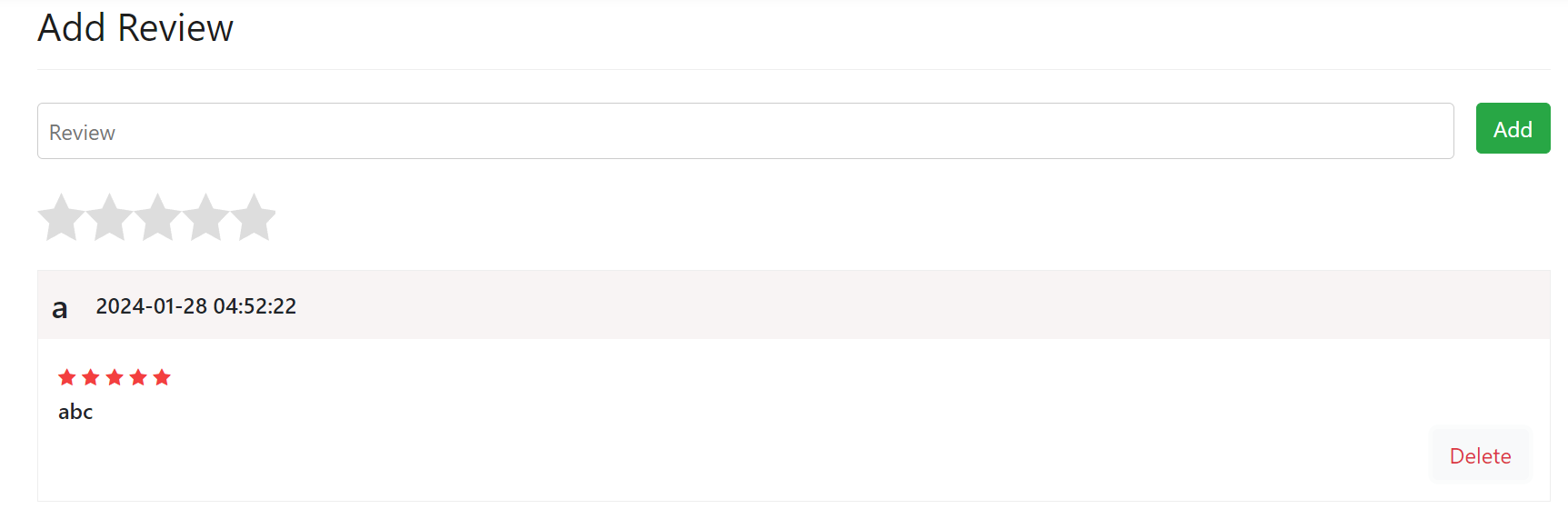
Structured Query Language or SQL is a language used for database creation and manipulation. It is a very commonly used database language. SQL injection is the placement of malicious code in SQL statements, via web page input. This code is sent from the client to the backend, which places the code into SQL queries and sends the query to the SQL database which runs the malicious code. SQL injection is a very dangerous exploit that can cause the destruction of databases, breaking of authentication and pose many other dangers to a web server.

### Vulnerability 1: Delete Reviews API SQLi (Detailed)

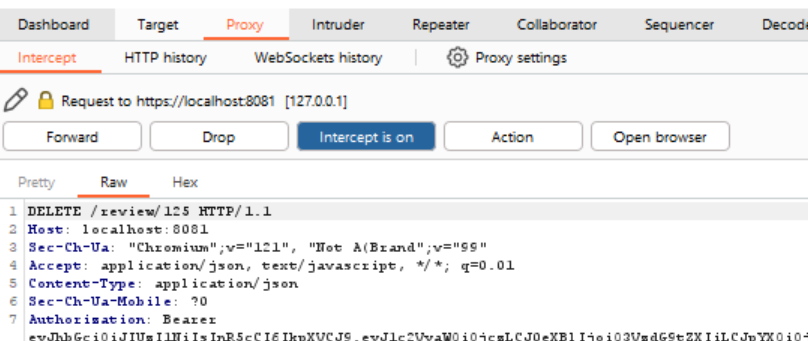
**1. Exploitation**

There were a few instances where SQL injection could have been exploited, but I will only go into detail for 1 exploit.

Seeing that we are able to delete our reviews in the products page, we know that the deletion process likely contains an SQL query.



If we can modify our request to input malicious code into the query, it can allow us to exploit this SQL query. We know that the delete review API uses the http DELETE method for /review/:reviewid. This is because we can see it in the request sent to the web server using a tool like BurpSuite as shown below:



This in itself is not a major vulnerability as it is difficult to prevent. The attackers can then assume that the :reviewid parameter is then used as a variable in the SQL query that is sent to the database. Hence, the attacker can append their own code behind the reviewid. It is fairly simple to guess that the SQL query goes something similar to

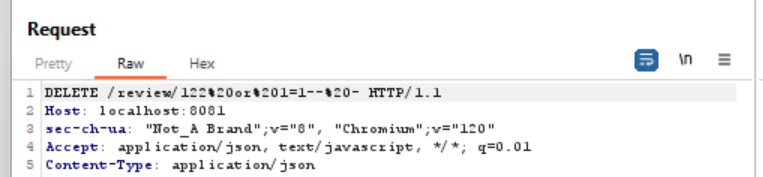
**“DELETE FROM review WHERE reviewid = ${reviewid} …”**

Knowing this, attackers can append code to delete other reviews as well. Below, I have used burpsuite to send a request with the appended code

“**or 1 = 1 -- -**”.

Hence the full request call is:

**“DELETE /review/122%20or%201=1--%20-**”



On the backend, this code will be injected into the SQL query, causing the query to be:

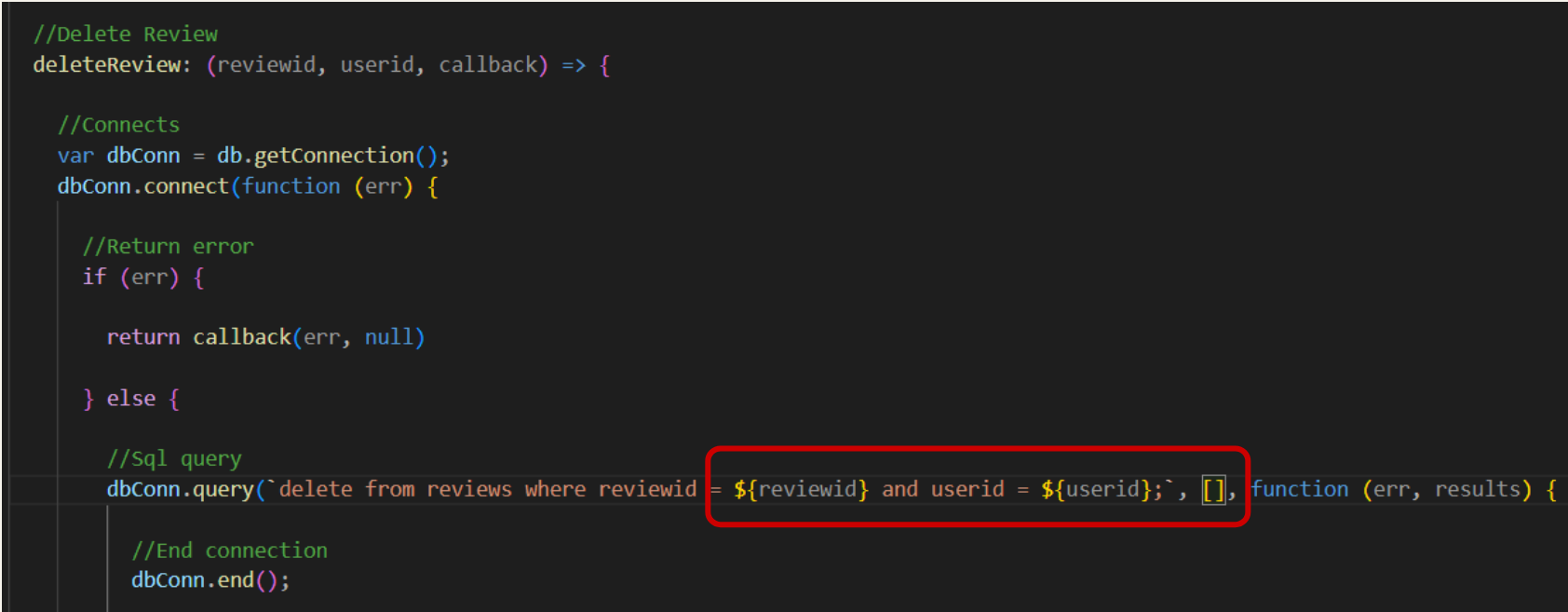
**“DELETE FROM review WHERE reviewid = 122 or 1 = 1 -- - …”**

Because 1 = 1 is a conditional statement which is True, this query will delete all reviews in the table. The “-- -” that follows behind is used to comment out any subsequent code which renders it unprocessed by the SQL database. This prevents any other code from interfering with the query. Product reviews show buyers’ confidence in the individual products as well as show customers the popularity of the company as many other legitimate buyers have bought from their store before. With all the reviews deleted, it can cause a loss of reputation to the company and hurt customer confidence.

**2. Vulnerable Code**

The following code snippet is found in the folder *BackEnd > model*

**review.js > deleteReview function:**



In the function above, the SQL query does not use prepared statements. Instead, they put the variable into the query string and send it to the SQL database. This allows data to be mixed up with code and hence results in a vulnerability which allows for SQL injection.

**3. Remediated Code**

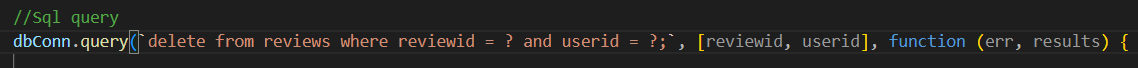
Using the principle of defense-in-depth, we will create 3 layers of defense against SQL injection:

1. Prepared Statements
2. Input Validation
3. Logging SQL Errors

3.1 Prepared Statements:

To change the query to a prepared statement, we should change the queries to use the ‘?’ placeholder instead of passing the variable into the query string. We can then place the variables that contain the data into a list which is put as a parameter of the .query function.

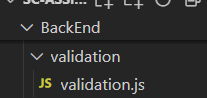
**review.js > deleteReview function:**



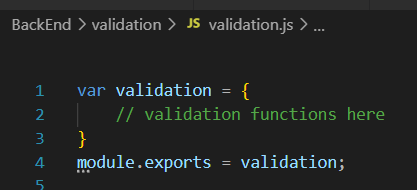
While prepared statements do create a strong defense against SQL injections, using the theory of defense-in-depth, we should also sanitize the inputs for SQL queries. As such, I will create a middleware to validate these inputs.

3.2 Input Validation

Input validation refers to validating user input and rejecting it if it does not meet certain requirements. In the BackEnd folder, I created a folder ‘validation’ and a file inside the folder called ‘validation.js’. This file will contain validation functions that can be called as middleware by our API.



Inside validation.js, create a validation dictionary which contains the validation functions



I then added 2 functions to validate userid and reviewid respectively. Validation is done via Regular Expression

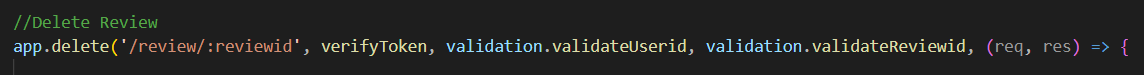


* For userid:
  + Regex: **^[0-9]+$**
  + Only digits are allowed
* For reviewid:
  + Regex: **^[0-9]+$**
  + Only digits are allowed

After creating the validation function, we import the validation library into app.js as *validation*



These middleware are then called on by the delete review API:



With this, any non-digit input will not be further processed by the server, adding an additional layer of defense against SQL injections on this API.

3.3 Logging:

Lastly, we will set up a logging service. I will be using the morgan node module to achieve this. In order to not clutter the logs, we should only log important information. In the case of SQL injections, I will log SQL errors.

Firstly, install the *morgan* and *rotating-file-stream* node modules by running the following command in the Terminal in the BackEnd folder:

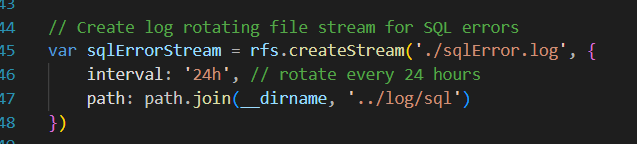
**npm install morgan**

**npm install rotating-file-stream**

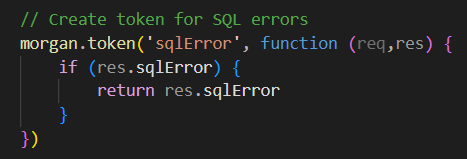
Next, import morgan and rotating-file-stream at the top of the BackEnd > app.js folder

****

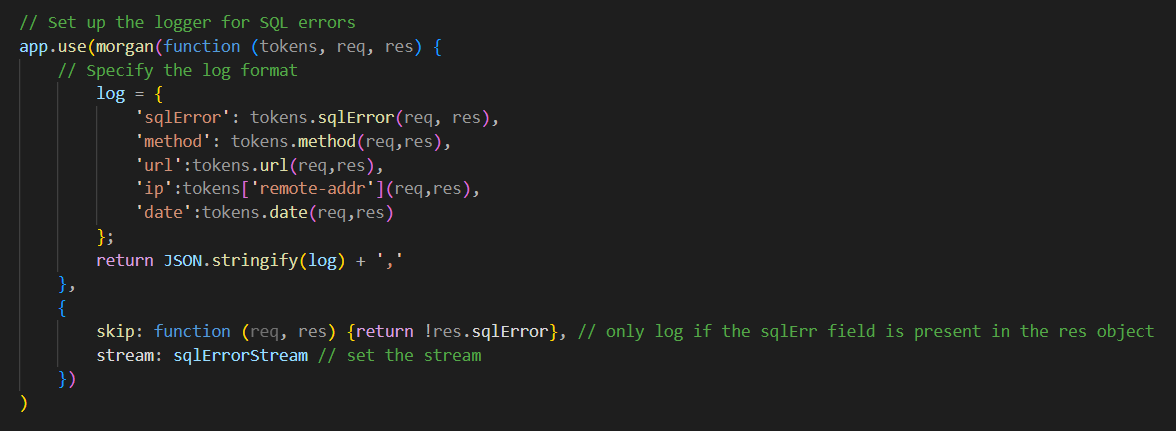
I can then create the rotating file stream for the logs. The logs will be stored in the BackEnd > log folder and I have set it to rotate every 24 hours. This can be adjusted accordingly.



I create a token that checks if the response object contains an sqlError field. The value in the sqlError field will be used in the log.



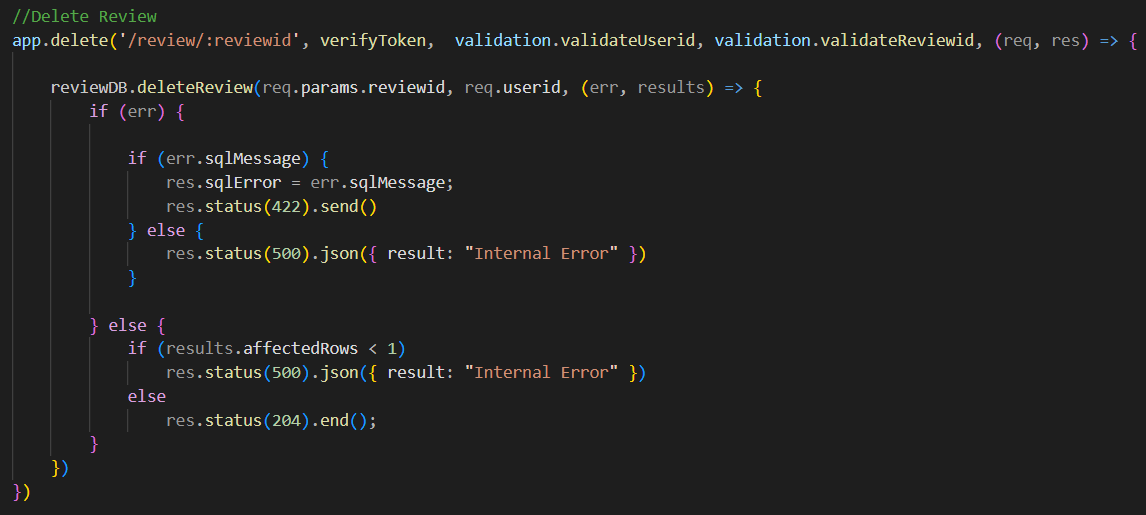
Finally we can set up the logger itself. I have set up a format to contain the sql error, the request method, the url, the source IP and the date. All the logs will be sent in JSON format to the stream specified earlier. This logger will skip all responses that do not have an sqlError; this means that the logger only logs sql errors. By only logging sql errors, it reduces the cluttering of logs and increases visibility.



As seen in the above code, the log format will contain the following information:

* sqlError message
* Request method
* Request URL
* Request Source IP address
* Date

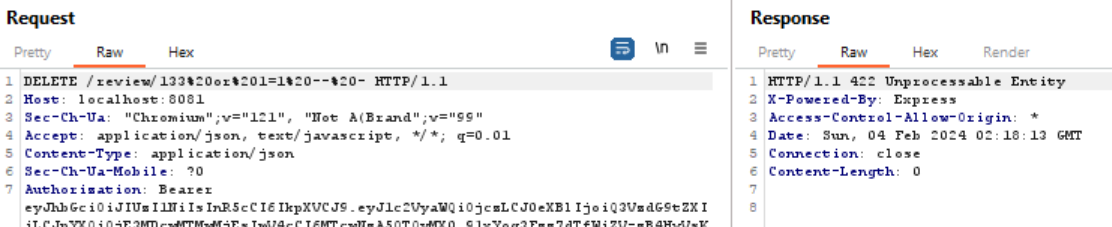
Now, in order to apply the logging to my API, I will just have to define the sqlError field as the sqlMessage in the response object when there is an error. This is shown below in the line **res.sqlError = err.sqlMessage.** I have also changed the response status to be 422 in the case of an SQL error.



**4. Testing**

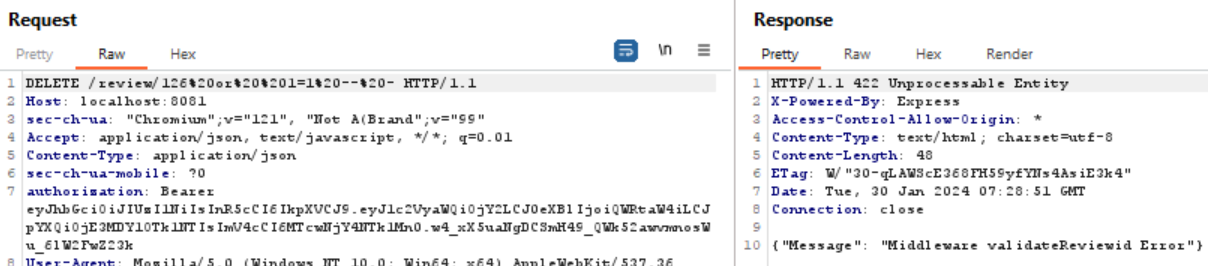
4.1 Testing (Prepared Statements)

If an SQL injection attempt is made with just the prepared statement in place (we are removing input validation for this test), the entire string is read as data and hence will result in an error, preventing a successful attack:



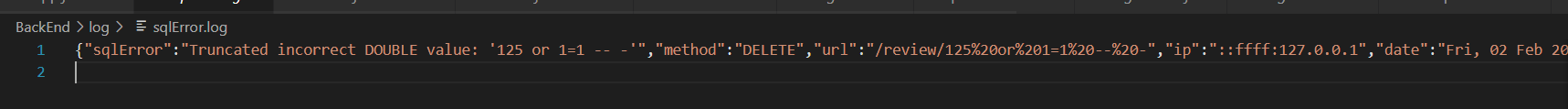
4.2 Testing (Input Validation)

If an SQL injection attempt is made with the input validation in place, the validation middleware will detect that the review id contains more than just numerical values and hence result in an error, preventing a successful attack:



4.3 Testing (Logging)

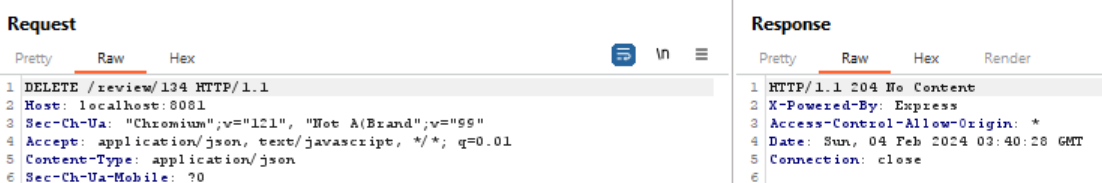
Now, if the validation of the reviewid somehow fails, the attempted sql injection will still be logged as it will produce an sql error since the prepared statements are in place. The error log can be found in Backend > log > sqlError.log



With this information, repeated errors from the same source or around the same time can be investigated and action can be taken before the threat has time to grow.

4.4 Testing (Valid Input)

In the case of a normal valid input, the review will be successfully deleted.

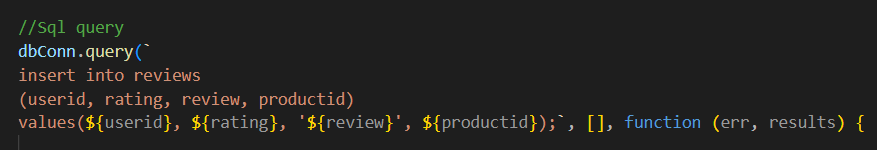


### Vulnerability 2: Add Reviews API SQLi (Brief)

**1. Vulnerable Code**

The following code snippets are found in the folder *BackEnd > model*

**review.js > addReview function:**



Similar to the first vulnerability, the above SQL query does not use prepared statements. Instead, they put the user input into the query string and send it to the SQL database. This allows data to be mixed up with code which allows for SQL injection.

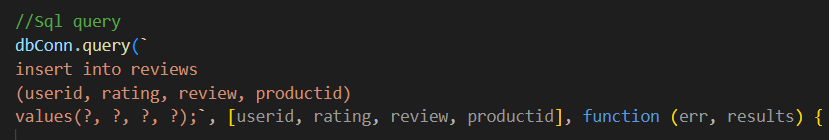
1. **Remediated Code**

2.a Prepared Statements

In order to fix these, we should change the queries to use the ‘?’ placeholder instead, making it a prepared statement. We then place the variables that contain the data into a list which is put as a parameter of the .query function.

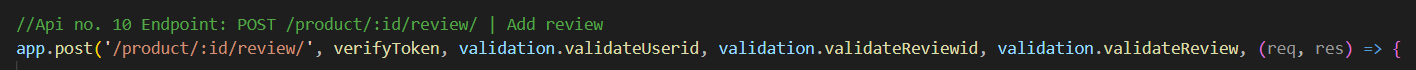
The following code snippets are found in the folders *BackEnd > model*

**review.js > addReview function:**

****

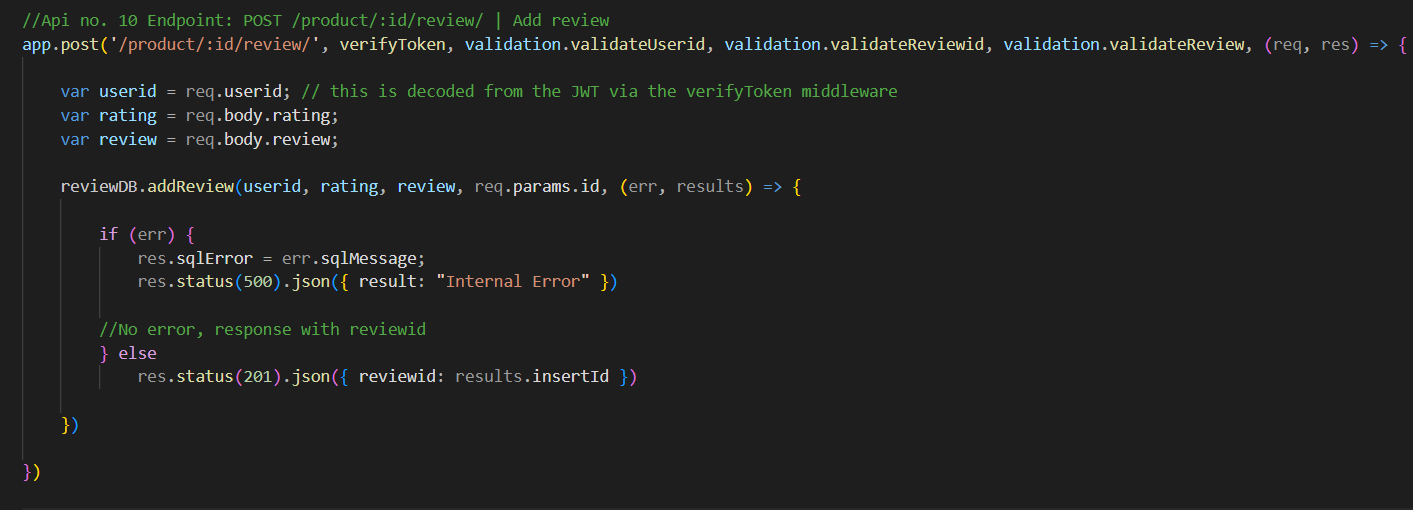
2.b Input Validation

In this case, input validation is not so relevant for SQL injection of the Add Review API, but it will be important later for XSS. More information about the validateReview function used here will be covered under the XSS section.



2.c Logging

The logging from the previous section will cover most of the setup required in order to log SQL errors for this API as well. All that needs to be done to apply the SQL error logging for this API is to add the res.sqlError field into the response object when the API encounters an error:



### Additional Recommendations

For all future SQL queries, I would recommend that for all parameterized queries, only prepared statements should be used. Alternatively, *Stored Procedures* can be used which can be found in the SQL workbench. These are queries that can be stored by the database and called by the backend multiple times. The backend will only send data to the SQL instead of the full queries, separating data from code.

## Cross-Site Scripting (XSS)

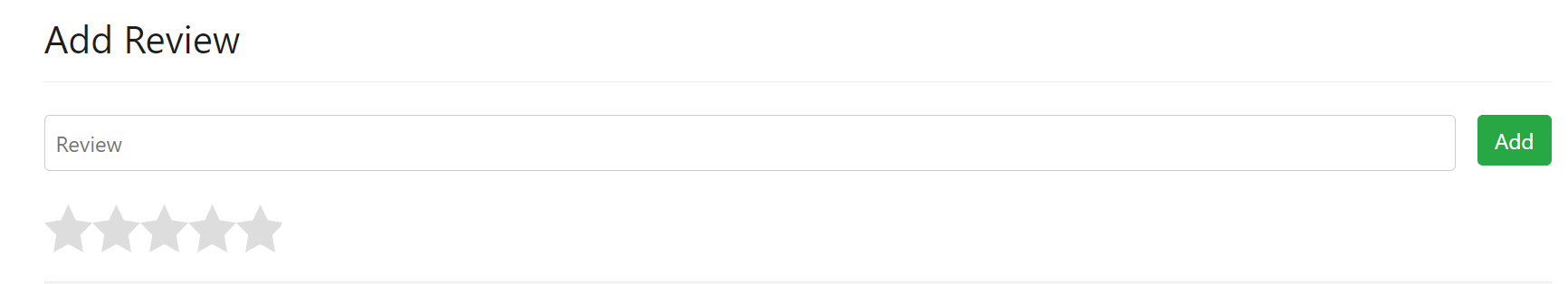
Cross-Site Scripting or XSS attacks involve the injection of malicious code into a web browser. This code, generally in the form of a browser side script, is then reflected or stored by the browser. The code is then run on the client of a different end user, resulting in a successful XSS attack.

### Vulnerability 1: Product Review Input XSS (Detailed)

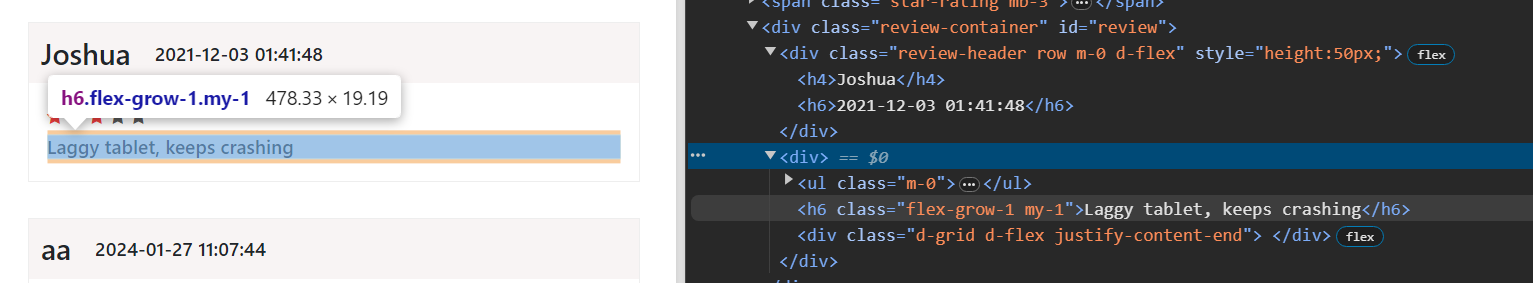
**1. Exploitation**

There were a few instances where XSS could have been exploited, but I will only go into detail for 1 exploit.

On the product review page, users can write reviews using any characters which will then be displayed in the reviews section.

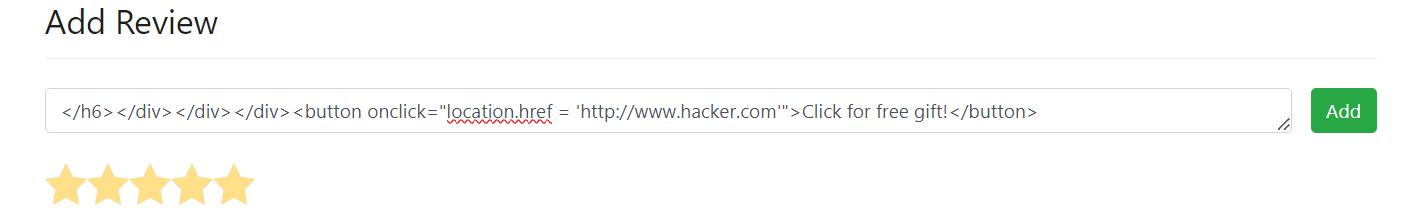


I looked at the html code of the reviews and saw what <header> tags or <div> tags i needed to close before I could input my own code.



From this I can see that if I want to add a html element outside of the review box, I would have to close a <h6> tag as well as 3 <div> tags. I wanted to put a button that would redirect users who clicked on it to a malicious page. As such, I entered the following XSS payload into the review input:

**</h6></div></div></div><button onclick="location.href = 'http://www.hacker.com'">Click for free gift!</button>**

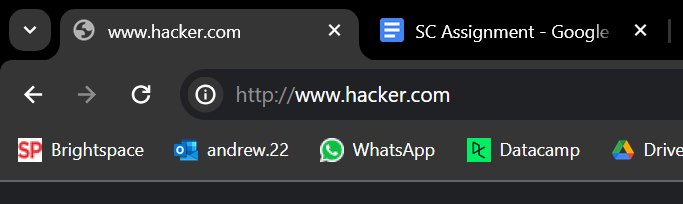
****

After submitting the review, the review was posted on the reviews section and I can see my button below the reviews section.



Users would think that links or buttons on a trusted website are safe to click and thus click on this button, redirecting them to the malicious website.

Upon button click:



1. **Vulnerable Code**

All pieces of code with user inputs as well as lack input validation and output sanitization are potential sources for XSS. For this report, I will only be focusing on 2 instances where XSS is exploitable. In each of the following code snippets, no input validation nor output sanitization has been performed, which allows attackers to enter malicious payloads such as “<script>...” to run scripts on the website, resulting in a successful XSS attack.

2.1 Client-side vulnerable code

The following code snippet is found in the folder *FrontEnd > public*

**products.html (write review)**

Lack of input sanitization

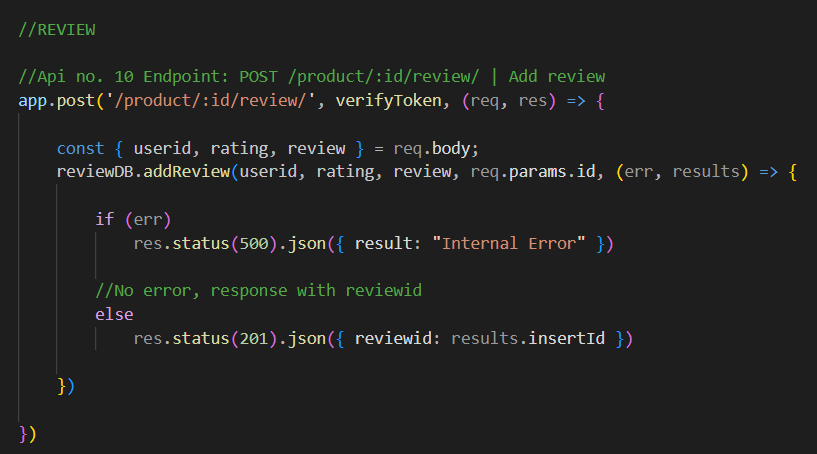


2.2 Server-side vulnerable code

The following code snippet is found in the file *BackEnd > controller > app.js*

**Endpoint: POST /product/:id/review/ | Add review**

Lack of input sanitization



**3. Remediated Code**

For XSS, it is useful to ensure that the website uses the theory of “defense-in-depth”. As such, we will apply 4 layers of defense.

1. Client-side validation
2. Server-side validation
3. Output Sanitization.
4. Logging

3.1 Client-side validation

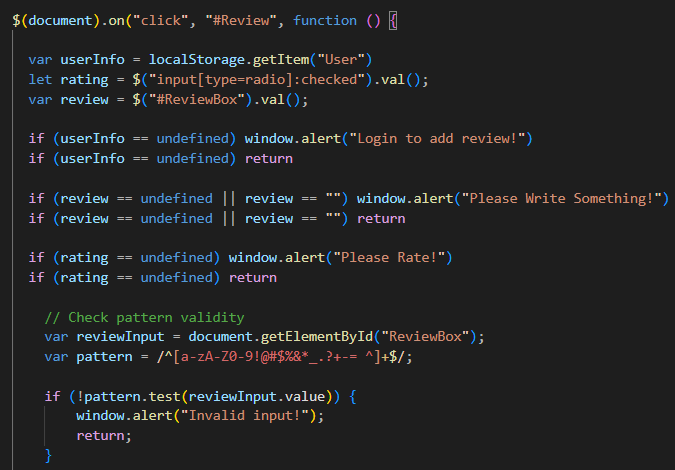
For client-side validation, I will just be adding a ‘pattern’ attribute to all the input fields and setting a certain regular expression that defines which characters the field can accept. *For more information about what the regular expressions mean, refer to b. Server-side validation.*

**products.html (write review)**

The html <textarea> element does not allow for pattern regex, so I shall change it to an <input> element with similar looks to that of a textarea. I have also added the pattern attribute with the regex which only allows for alphanumeric characters and selected special characters.



On top of adding the pattern attribute, I had to make sure the website would check for the pattern before allowing the form to be submitted. Hence, upon the ‘Add’ button being clicked, i added a pattern checker to the currently existing button click listener:



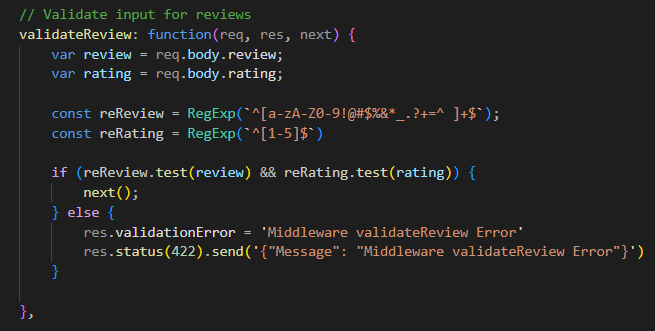
However, client-side validation is not enough as attackers can bypass it easily by modifying their requests. Hence, server-side validation is very important.

3.2 Server-side validation

We will need to install 2 node modules for this. In the backend folder, run the following:

* npm install validator
* npm install is-url

Using the ‘‘validation.js’ file which I created earlier in the SQL injection section, we will add the following validateReview function:



This function validates the characters input for the review. It works via Regular Expressions or ‘Regex’ to whitelist certain characters.

* For reviews:
  + Regex: **^[a-zA-Z0-9!@#$%&\*\_.?+=^ ]$**
  + Only allow alphanumeric characters and selected special characters
* For ratings:
  + Regex: **[1-5]$**
  + Only allow 1 digits from 1 to 5

If all the validation requirements are met, the middleware will go on to the next() middleware.

**Calling the middleware**

Apply the validateReview middleware on the POST /product/:id/review/ API

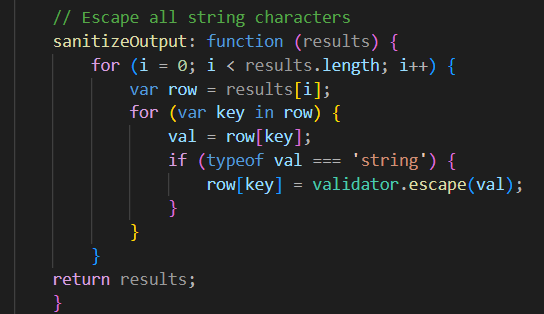


With this, all review inputs are validated before being stored in the database.

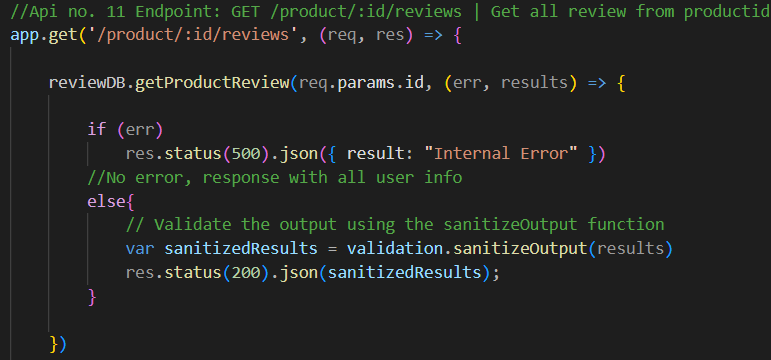
3.3 Output Sanitization

The final step for our defense-in-depth is output sanitization. For this, I will use the *validator* library’s **escape()** function.

In Backend > validation > validation.js, I have added the function sanitizeOutput that will serve as a function that sanitizes the review output before it gets displayed:



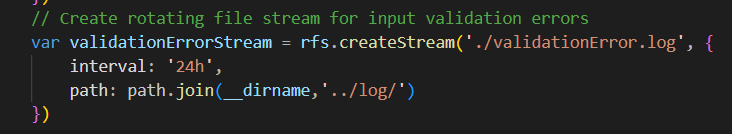
We will use this function in the GET /product/:id/reviews endpoint.



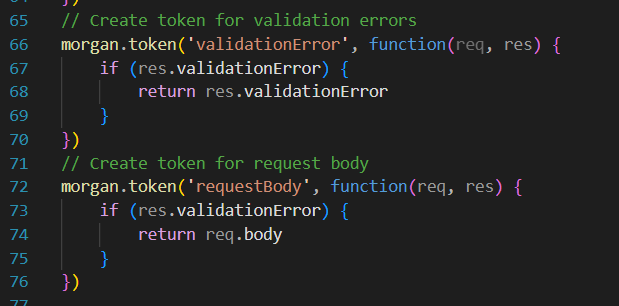
3.4 Logging

We have already set up Morgan by installing as well as importing the morgan and rotating-file-stream node modules in *SQL Injection Vulnerability 1: Delete Reviews SQLi.* We can then go straight to setting up the file stream, token and logger for input validation errors. For this case, we will only log input validation errors from the validation function created above which is validateReview. The steps will be very similar to the logging in SQL injection, except for the final step where something must be added to the validation.js file.

First, create the rotating file stream for the logs. The logs will be stored in the BackEnd > log folder and I have set it to rotate every 24 hours. This can be adjusted accordingly.



This time, I created 2 custom tokens. 1 token will be for the validation error message from the validation function. The second token will allow the log to contain the request body when there is a validation error.



Finally we can set up the logger itself. I have set up a format to contain the validation function error, the request body, the request method, the url, the source IP and the date. All the logs will be sent in JSON format to the stream specified earlier. This logger will skip all responses that do not have an sqlError; this means that the logger only logs sql errors. By only logging sql errors, it reduces the cluttering of logs and increases visibility.

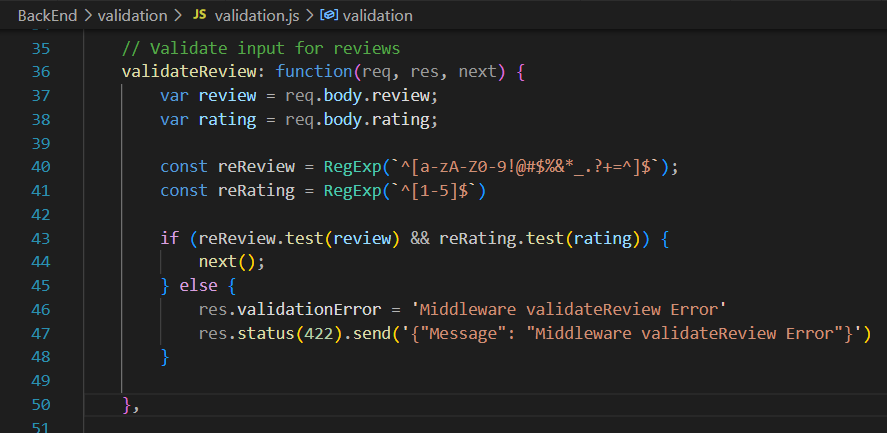


As seen in the above code, the log format will contain the following information:

* Validation function where error occurred
* Request body
* Request method
* Request URL
* Request Source IP address

DateFinally, in the BackEnd > validation > validation.js file, we will add a line to the validateReview function that adds a validationError field to the response when the validation fails.

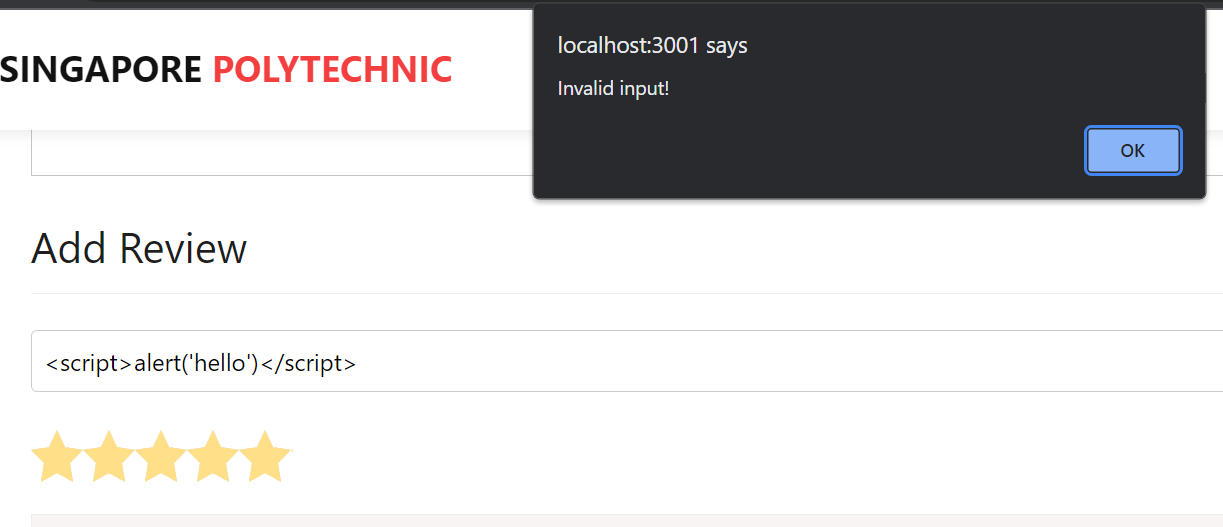
**res.validationError = ‘Middleware validateReview Error’**



**4. Testing**

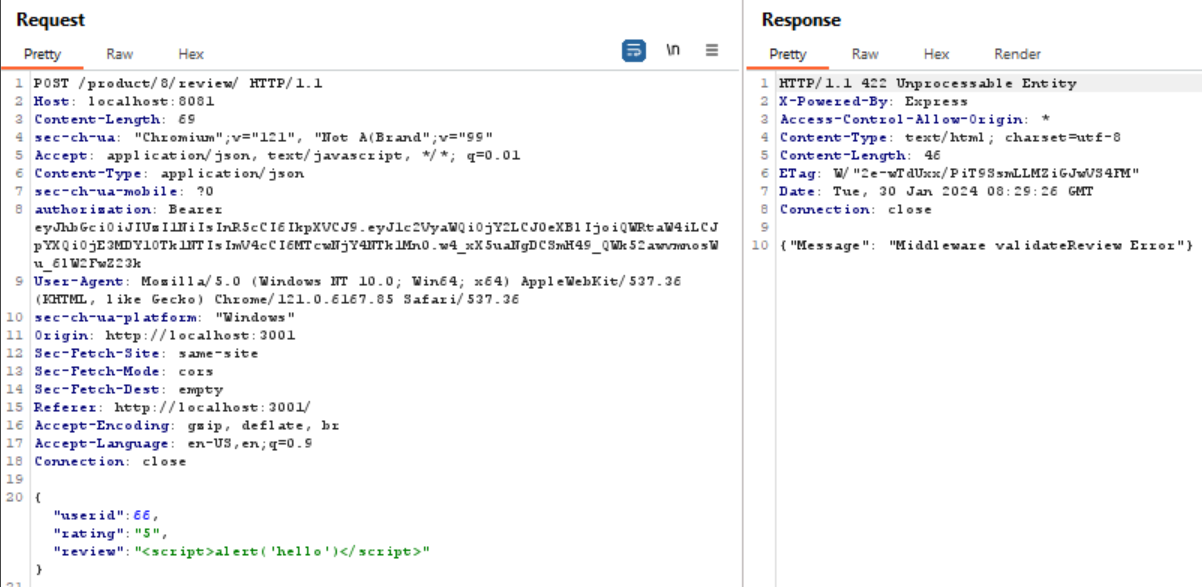
4.1 Testing (Client-Side Validation)

If a user were to type invalid inputs into the review field, the client will notify them that a character that they had input is invalid.



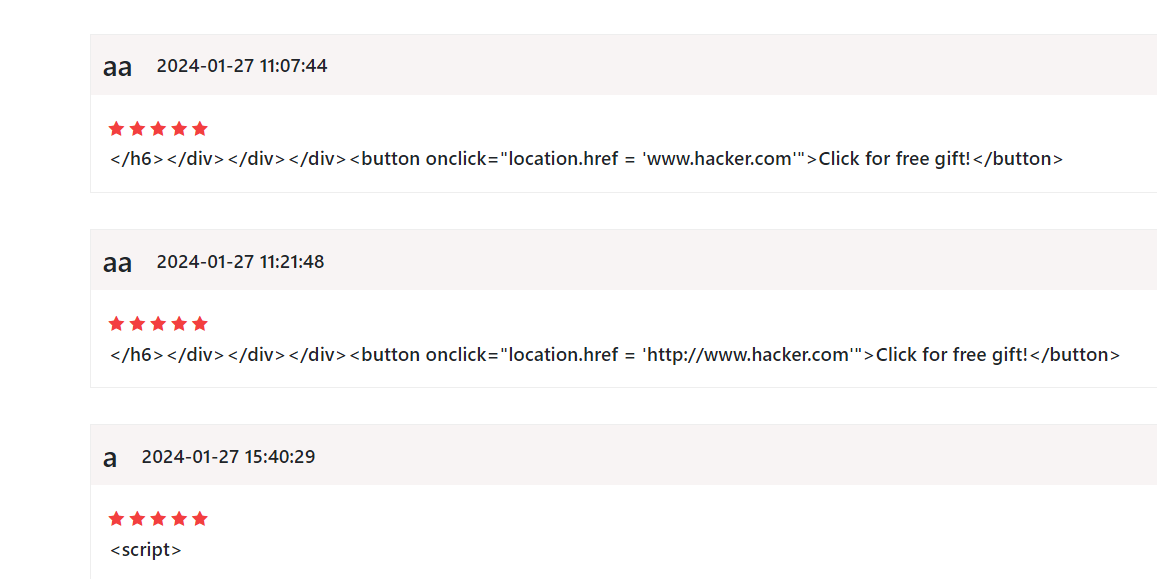
4.2 Testing (Server-Side Validation)

If a client bypasses the client-side validation and still has invalid inputs in their request, the back-end will give them an error:



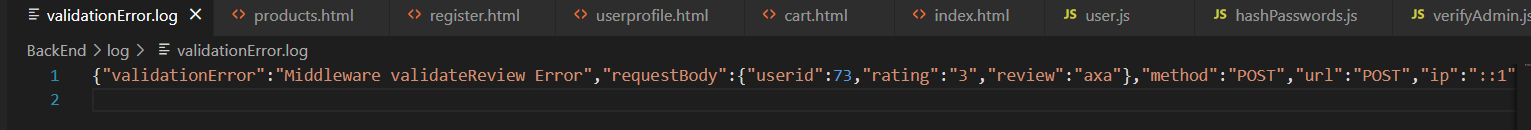
4.3 Testing (Output Sanitization)

If an XSS payload somehow makes its way into the database or was already within the database, reviews that include an attempted XSS show up like this:



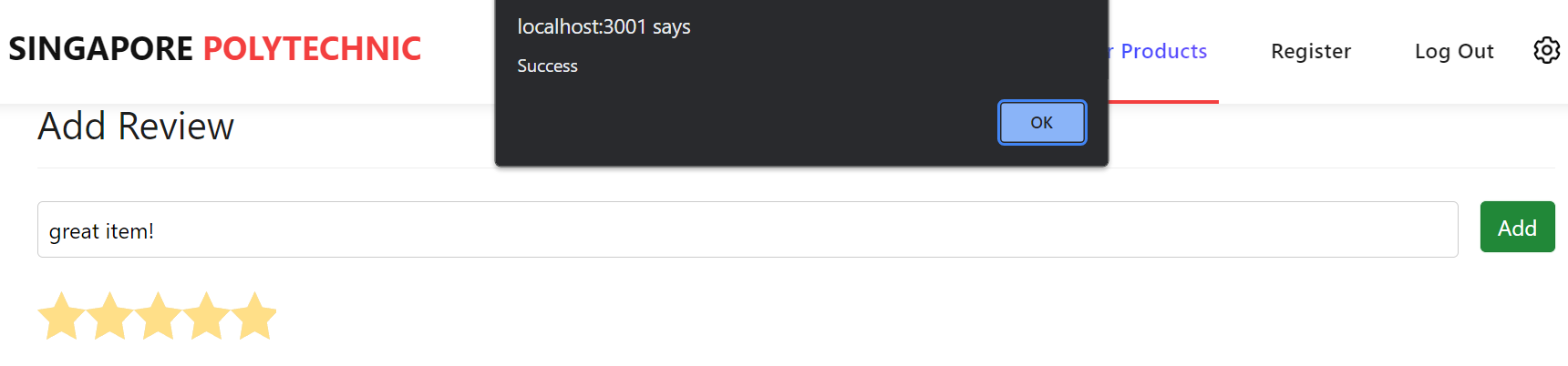
4.4 Testing (Logging)

Now when the validateReview middleware encounters an error, the error is logged with all the relevant information.



4.5 Testing (Valid Input)

In the case of valid input, the review will be added.



### Vulnerability 2: Register User Input XSS (Brief)

**1. Vulnerable Code**

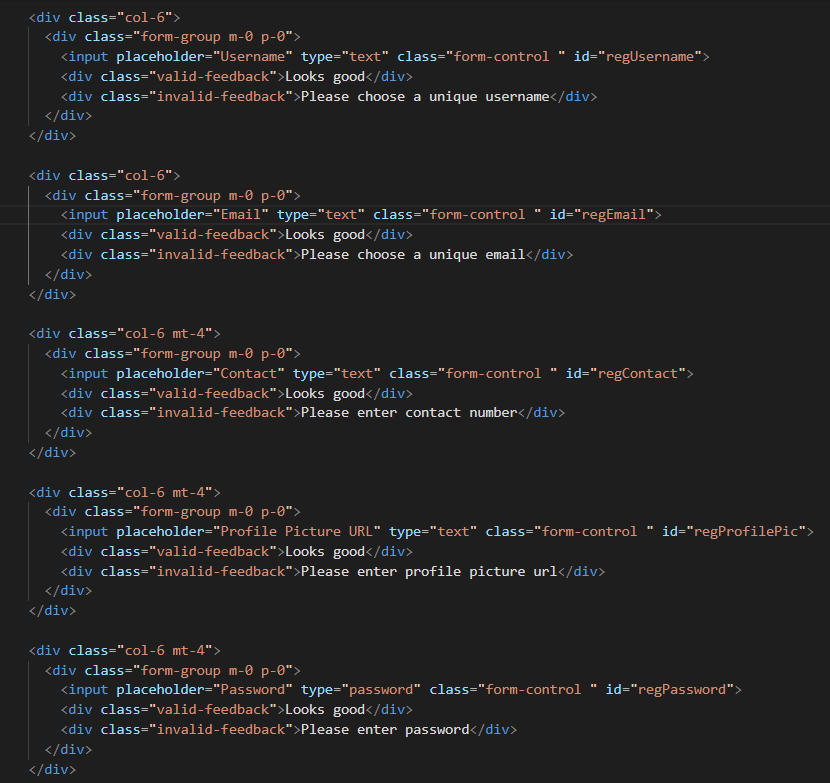
1.1 Client-side vulnerable code

The following code snippets are found in the folder *FrontEnd > public* and lack input sanitization

1. **register.html**



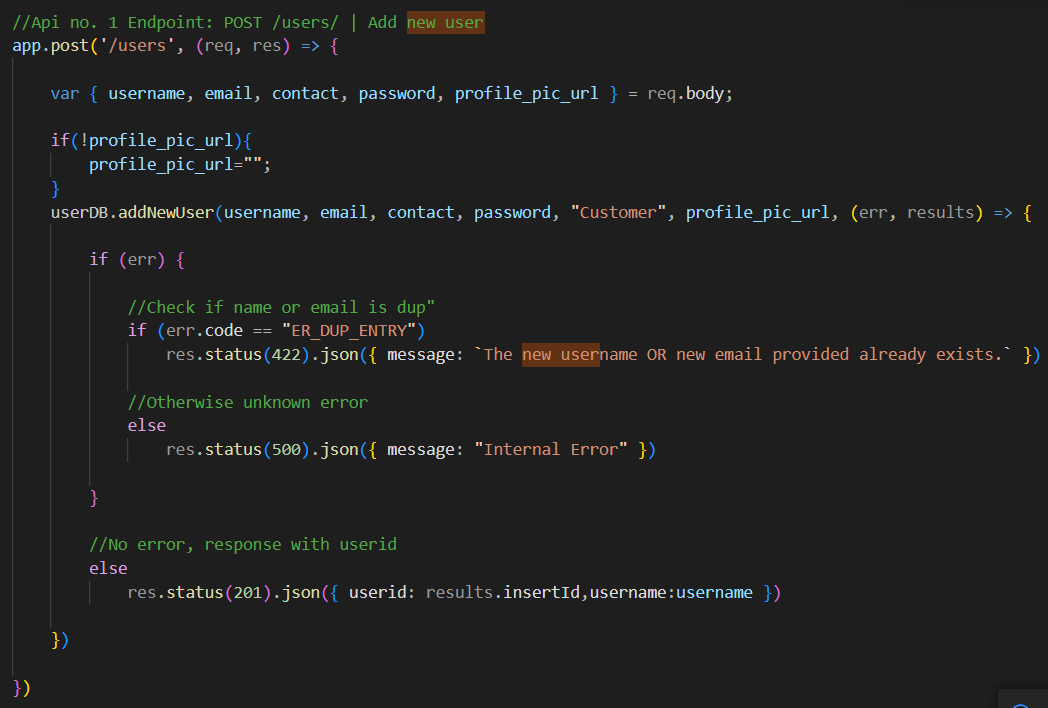
1. **login.html**



2.2 Server-side vulnerable code

The following code snippet is found in the file *BackEnd > controller > app.js* and lacks input sanitization.

**Endpoint: POST /users/ | Add new user**



**2. Remediated Code**

As mentioned earlier, XSS is best defended against using a defense-in-depth principle:

1. Client-side validation
2. Server-side validation
3. Output Sanitization.

2.1 Client-side validation

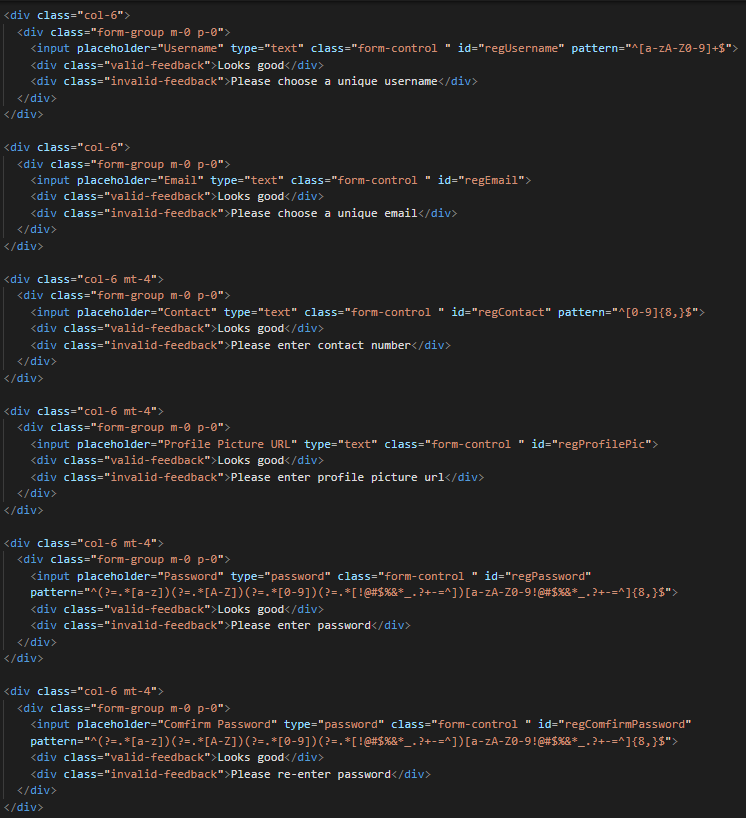
Similar to the first XSS vulnerability, I will just be adding a ‘pattern’ attribute to all the input fields and setting a certain regular expression that defines which characters the field can accept.

1. **register.html**

The regular expressions added for each input field is the same as in b. Server-side validation > B.2 which can be found below this section. The only exception is the email input as it already has input validation.



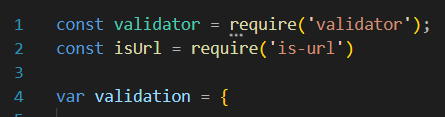
1. **login.html**



However, client-side validation is not enough as attackers can bypass it easily by modifying their requests. Hence, server-side validation is very important.

2.2 Server-side validation

Using the ‘‘validation.js’ file which I created earlier, we will then import these node modules at the top of the validation.js file



**Function for endpoint: POST /users/ | Add new user**

Below is the validateRegister function:



This function will be used to validate user inputs for registration. Regular expressions are used to whitelist the characters allowed for username, password and contact. For emails, the *validator* library’s .isEmail() function was used. For the profile picture URL, it can either be left blank or it is validated by the is-url library.

* For usernames:
  + Regex: **^[a-zA-Z0-9]+$**
  + only lowercase, uppercase and digit characters are allowed
* For passwords:
  + Regex:**^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*[0-9])(?=.\*[!@#$%&\*\_.?+=^])[a-zA-Z0-9!@#$%&\*\_.?+=^]{8,}$**
  + at least 1 lowercase, 1 uppercase, 1 digit, 1 special character and minimum 8 characters are required.
* For contacts:
  + Regex: **^[0-9]{8,}$**
  + only digits are allowed and a minimum of 8 characters are required
* For emails:
  + validating using the *validator* library: **.isEmail()** function
* For URLs:
  + Either left blank or validated using the *is-url* library.

If all the validation requirements are met, the middleware will go on to the next() middleware.

**Calling the middleware**

Apply the validateRegister middleware on the POST /users/ API



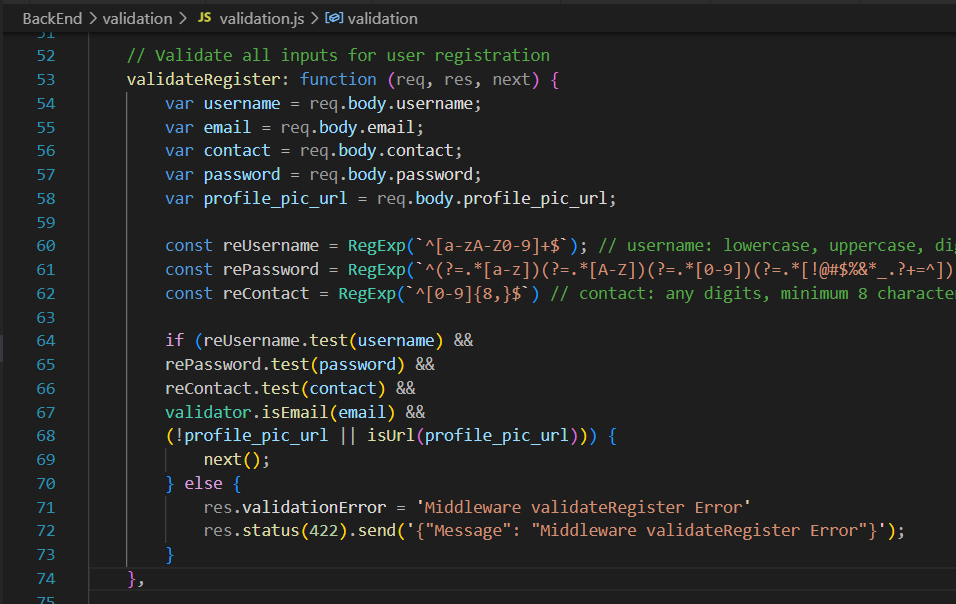
With this, all user registration inputs are validated before being stored in the database.

2.3 Output Sanitization

For output sanitization, the sink of the first vulnerability and second vulnerability are both in the product reviews page. Hence, the earlier output sanitization for XSS vulnerability 1 already solved this layer of defense.

2.4 Logging

We have set up the input validation logging with Morgan in the previous vulnerability under XSS. Hence, all that is needed to apply the logging for this vulnerability as well is to add a res.validationError line in the middleware validateRegister. To do this, go the BackEnd > validation > validation.js and add **res.validationError = middleware validateRegister Error’** under the else statement. This means that if the regular expression test fails, it will add this validationError field to the response object, triggering the logging.



### Additional Recommendations

There are still more possible areas for XSS attacks on this website. In the future when user input is displayed or taken in, there could also be more potential sources or sinks for XSS exploitation. Hence, the developer should make sure that all user input is validated on both the client and server sides (server-side is the most important) and output is sanitized before it is displayed.

## 

## Broken Access Control

Broken Access Control refers to users being allowed to perform certain actions that should be beyond their permissions. This usually stems from insufficient or a lack of proper authorization for actions that should require some level of privilege.

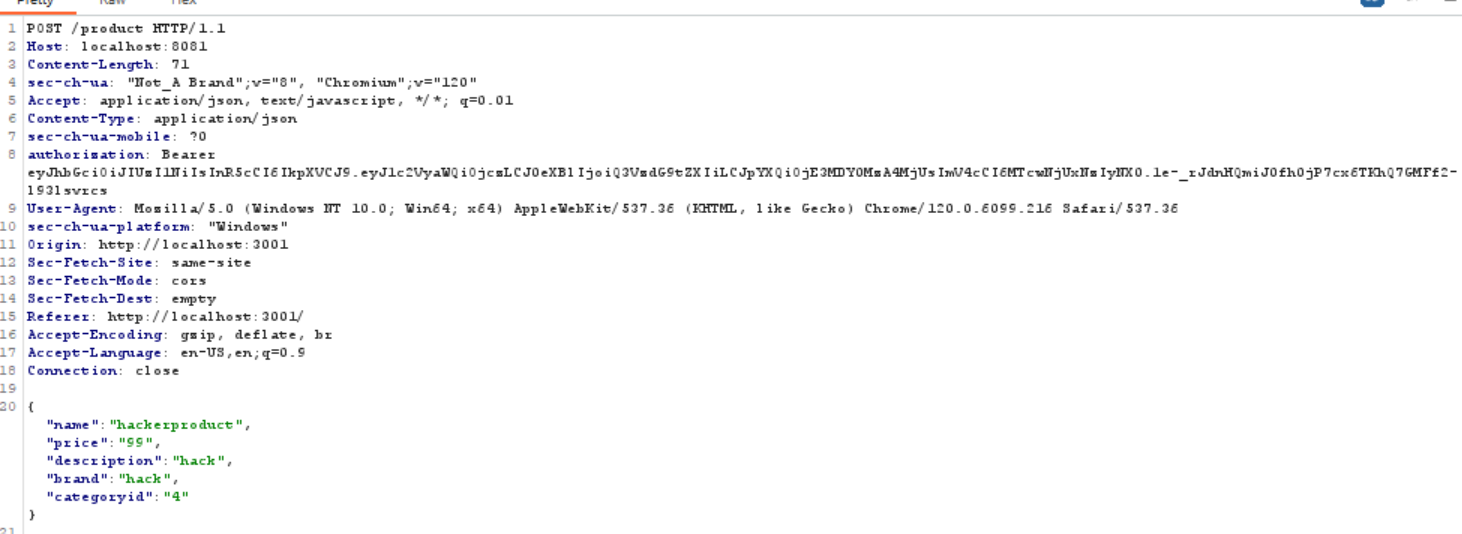
### Vulnerability 1: Non-Admins can Add Products (Detailed)

**1. Exploitation**

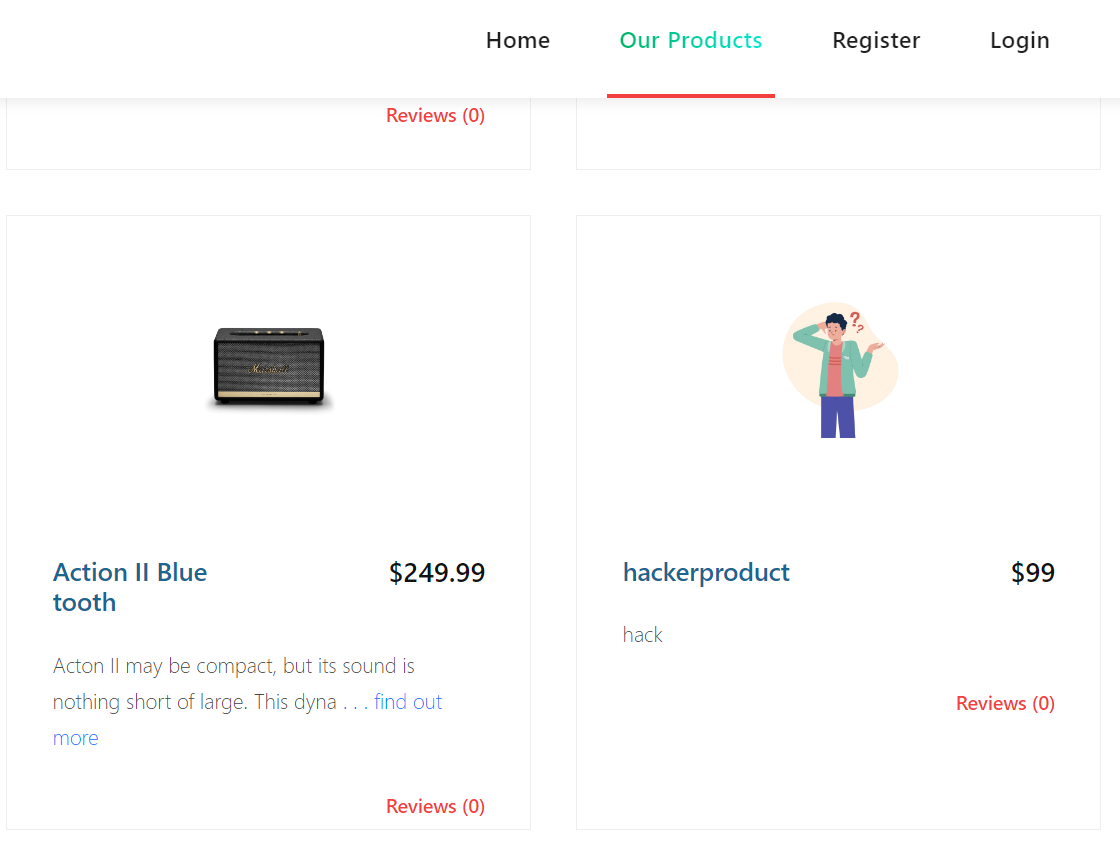
There were a few instances where Broken Access Control could have been exploited, but I will only go into detail for 1 exploit.

Here, I will exploit the lack of admin verification by the server to add our own product to the database. While users cannot access the admin page, they can still directly send requests to the webserver’s backend APIs.

In the below example, I am logged in as a normal customer user. I can guess the API for adding a new product as it is just “POST /product”, although this is not really a vulnerability in itself. I can then create whatever product parameters I want and send it to the API.



The API does check if I have a valid token, which I do since I am logged in as a customer. However, the API does not verify if I am an admin. Hence, this adding of a new product will be successful and result in the website displaying it. With that, the website now lists my ‘hackerproduct’ on the products page. Hackers can also use this for XSS attacks.

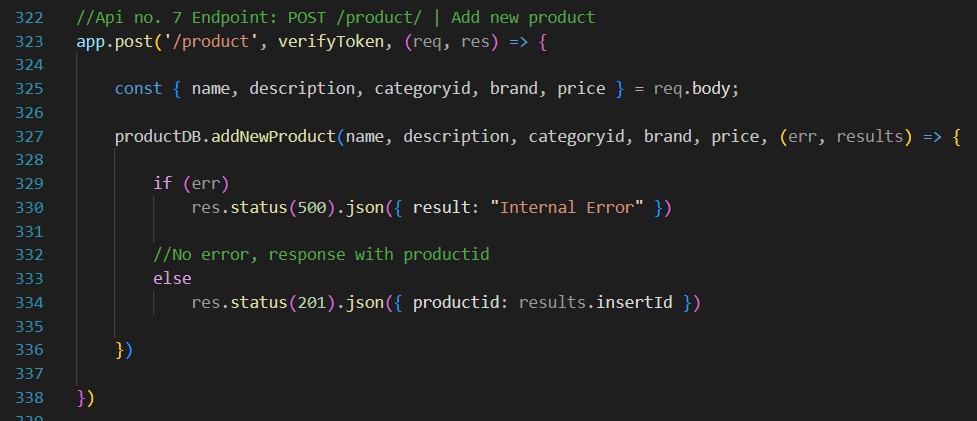


Allowing attackers to have access to APIs that only admins should have access to is very dangerous as it effectively grants them admin privileges. In the next section, we will see that other APIs are also affected like adding categories as well as deleting products.

**2. Vulnerable Code**

The following code is the API to add a new product that should only be accessible to admins. However, they are missing any admin role checks, allowing unauthorized users to use this API.

**Endpoint: POST /product/ | Add new product**



**3. Remediated Code**

Following the principle of defense-in-depth, we will have 2 layers of defense for this:

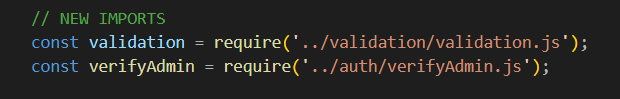
1. Admin Verification
2. Logging Unauthorized Attempts

3.1 Admin Verification

In order to fix this code, we need to add some form of admin check to authenticate admin users which will authorize them to use the APIs. To do that, I will create a new authentication function in BackEnd > auth > verifyAdmin.js called ‘verifyAdmin’. This function will use the role which was decoded by the verifyToken middleware and check if the user’s role is ‘Admin’. Because it is checked from the JWT, it is secure as a tampered JWT will not be successfully verified by verifyToken. Below is the function:

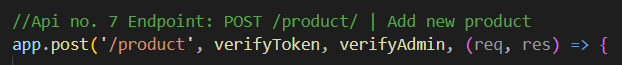


Now we can import this function into app.js



And call this middleware on all the relevant API after the verifyToken middleware:

**Endpoint: POST /product/ | Add new product**

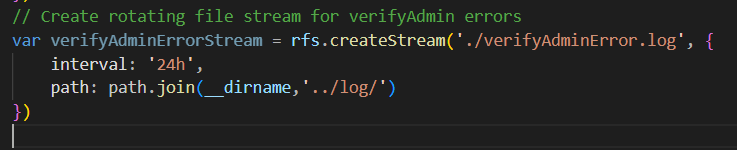
****

With this middleware in place, the APIs will check if the user’s JWT is legitimate and then check if they have the admin role in their JWT. This eliminates the previous authentication failure.

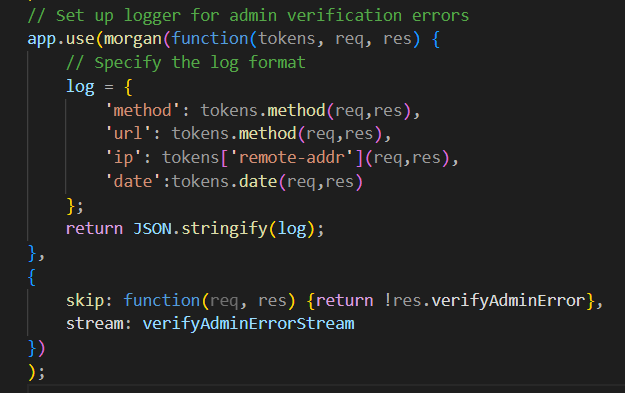
3.2 Logging Unauthorized Attempts

With most of the Morgan and rotating-file-stream node modules set up after fixing SQLi vulnerability 1, we can get straight to creating the file stream and logger for verifyAdmin.

We’ll start off creating the rotating file stream as shown below



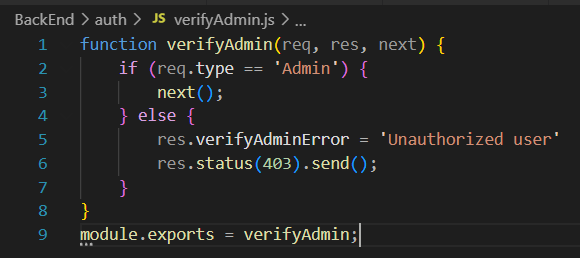
For this logger, I don’t really need a custom token. Hence we can move on to creating the logger. The will only trigger if the verifyAdmin field is present in the response, thus only logging when admin verification fails.



As seen in the above code, the log format will contain the following information:

* Request method
* Request URL
* Request Source IP address
* Date

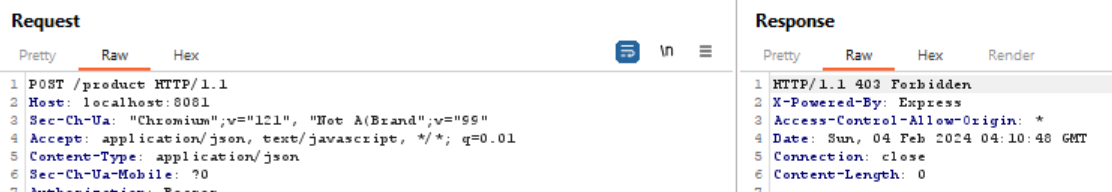
Finally, add to BackEnd > auth >verifyAdmin.js a line of code that fills in the response ‘verifyAdminError’ field if the user is not admin.



**4. Testing**

4.1 Testing (Admin Verification)

Now, if a normal user tries to use an admin API, the server will respond with 403 Forbidden.

****

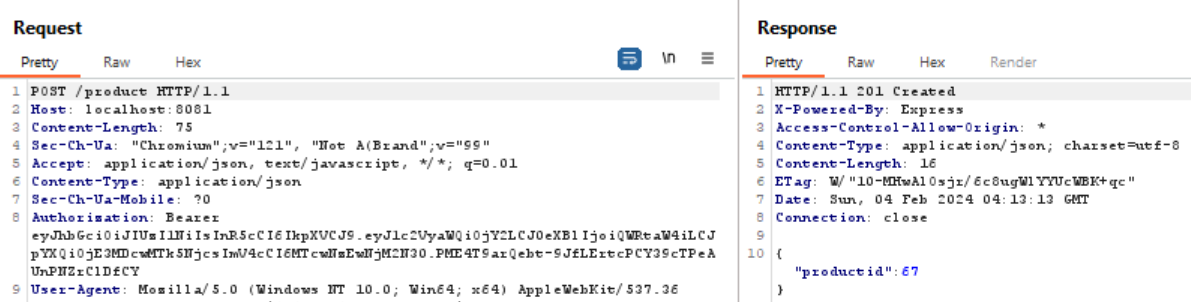
4.2 Testing (Logging Unauthorized Attempts)

On top of denying the user access to the resource, it will also log the failed attempt:



4.3 Testing (Legitimate Admin User)

In the case of an actual Admin user using the API, it will be successful.

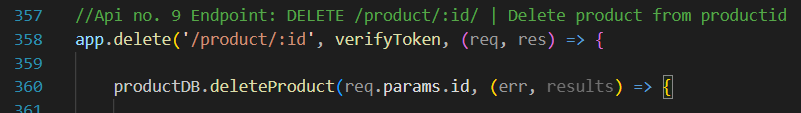


### Vulnerability 2: Non-Admins can Delete Products (Brief)

**1. Vulnerable Code**

The below code is another admin-only API, delete product, that has no admin checks. This allows any user with a valid token (meaning they just need to log in as a customer) to send a request to this API and the API will accept it.

**Endpoint: DELETE /product/:id/ | Delete product from productid**



**2. Remediated Code**

2.1 Admin Verification

Since we have already created the verifyAdmin middleware in the previous part, all we have to do is call the verifyAdmin middleware for this API and it will check for the admin role in the valid JWT. Thus, this problem is solved as well.

**Endpoint: DELETE /product/:id/ | Delete product from productid**

****

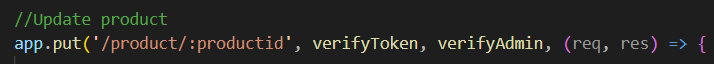
2.2 Logging Unauthorized Attempts

After the logging has been set up as specified in the previous vulnerability (Non-Admins can add product), it will automatically apply to all the APIs with the verifyAdmin middleware. Thus, no further configurations must be made for logging to work for this API as well.

### Additional Recommendations

There are some other admin-only APIs that are vulnerable. I have applied the verifyAdmin middleware to them as well as shown below:

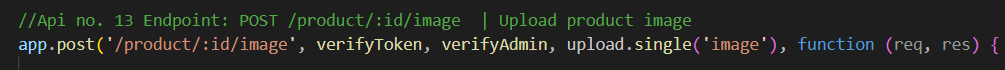
1. **Endpoint: PUT /product/:productid | Update Product**

****

1. **Endpoint: POST /category | Add new category**

****

1. **Endpoint: POST /product/:id/image | Upload product image**

****

Some of these APIs are not even in use anymore. These should be removed if they are no longer used as it clutters the code and can result in unmaintained APIs which may later be attacked.

Role-based Access Control (RBAC) is a simple and effective way to manage the permissions in the web server. This refers to giving users roles like ‘Customer’ and ‘admin’ which has already been done, but it must then be followed up by using these roles to grant permissions to resources. By using middleware like verifyToken combined with verifyAdmin, you can secure your APIs to only be used by authorized users like mentioned above. This should be the standard practice for all future APIs which should only be called by authorized users.

## Cryptographic Failures

The focus of this category is on failures related to cryptography (or lack thereof), which often leads to exposure of sensitive data. This includes weak encryption algorithms, broken hashes or the lack of any encryption/hashes at all.

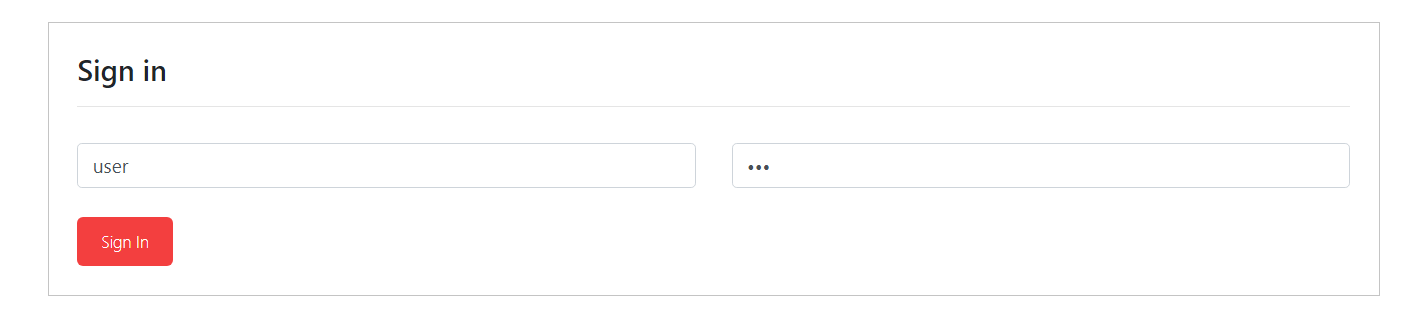
### Vulnerability 1: Lack of Network Traffic Encryption (Detailed)

**1. Exploitation**

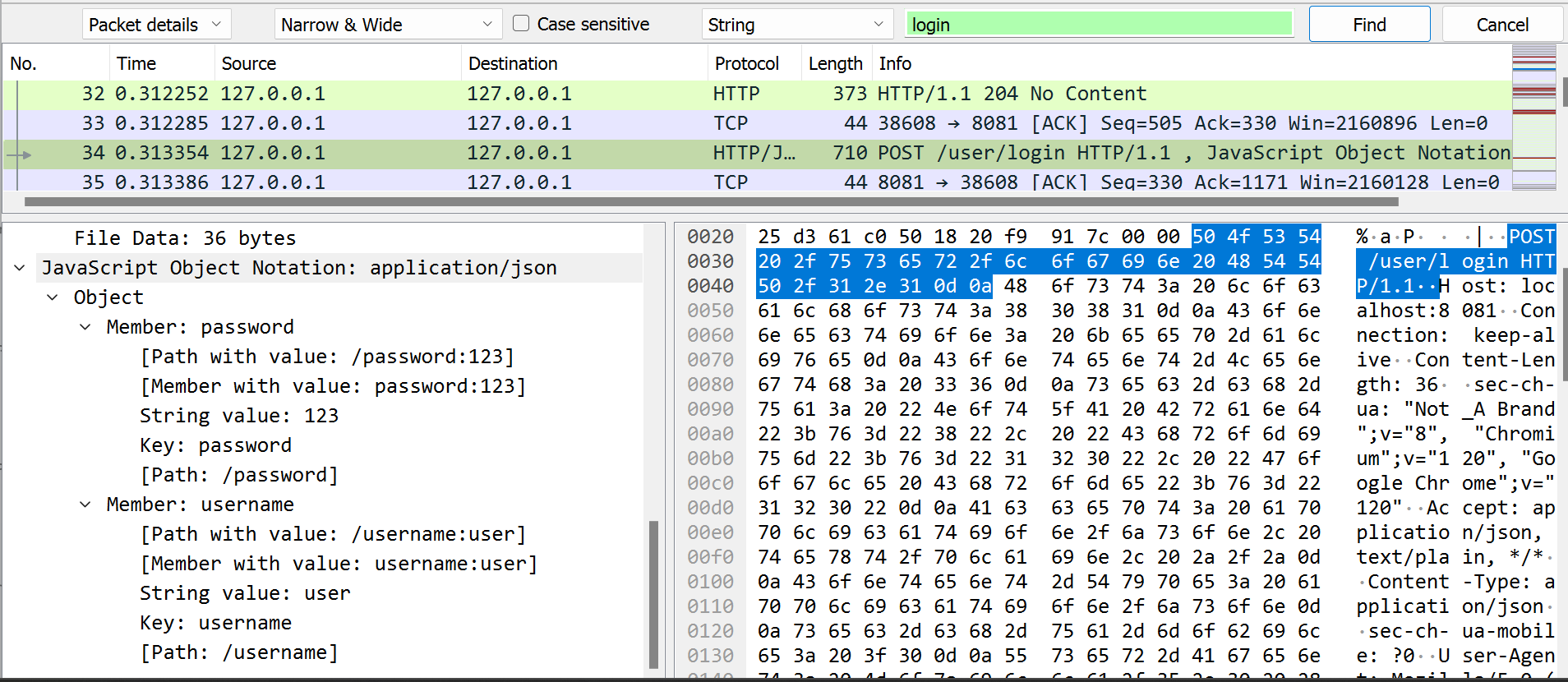
There were a few instances where Cryptography Failures could have been exploited, but I will only go into detail for 1 exploit.

We will exploit the lack of network traffic encryption. This means that if attackers are able to tap into the connection between the client and the web-server back-end, they are able to view all the traffic sent in plaintext. In the following example, I will use Wireshark to intercept the network packets between the client and the back-end.

I have opened Wireshark to listen to packets between the front-end and back-end. Now, if an unsuspecting user logs in to the website with their login credentials:



Using wireshark, I can see the network traffic in plaintext



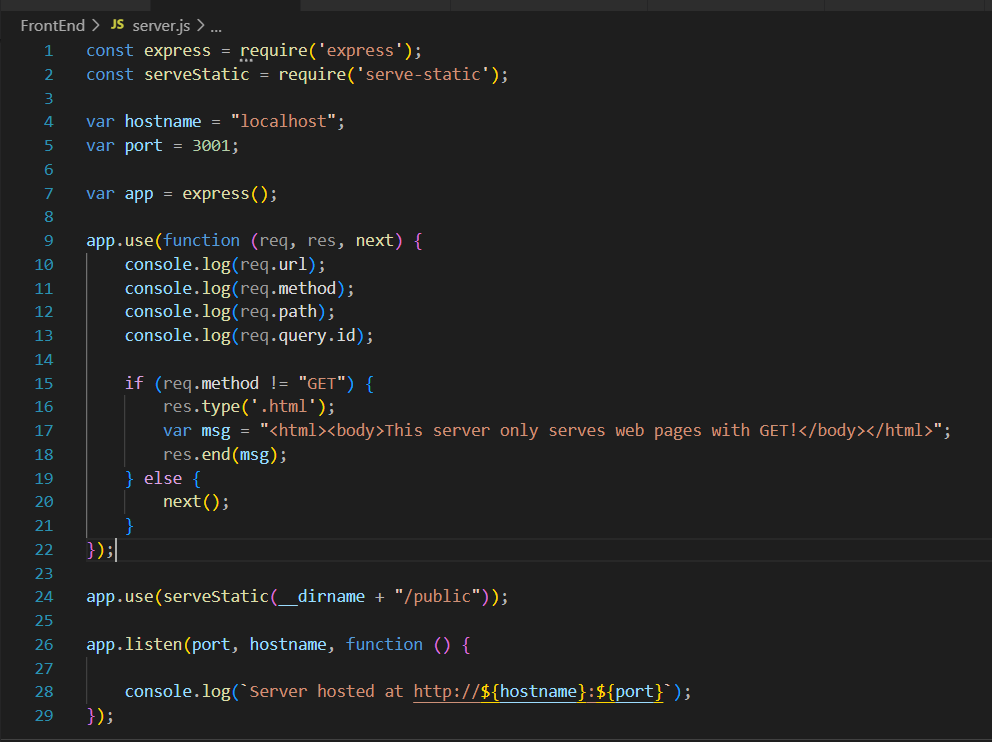
Just like that, I have obtained the user’s username and password in plaintext, allowing me to log in as that user. The user in this example also happens to be the Admin user, meaning the attacker gains the credentials of an account with administrator privileges.

This is highly dangerous network traffic between the client and server can be easily intercepted by attackers. Without proper encryption or integrity checks, all this information will be displayed in plaintext to the attackers.

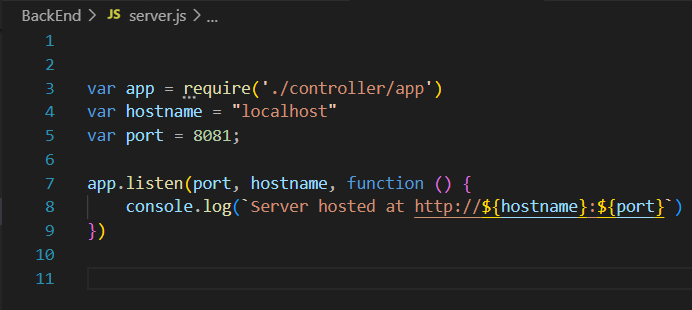
**2. Vulnerable Code**

Neither server.js makes use of SSL encryption to convert the traffic into HTTPS.

FrontEnd > server.js:



BackEnd > Server.js:



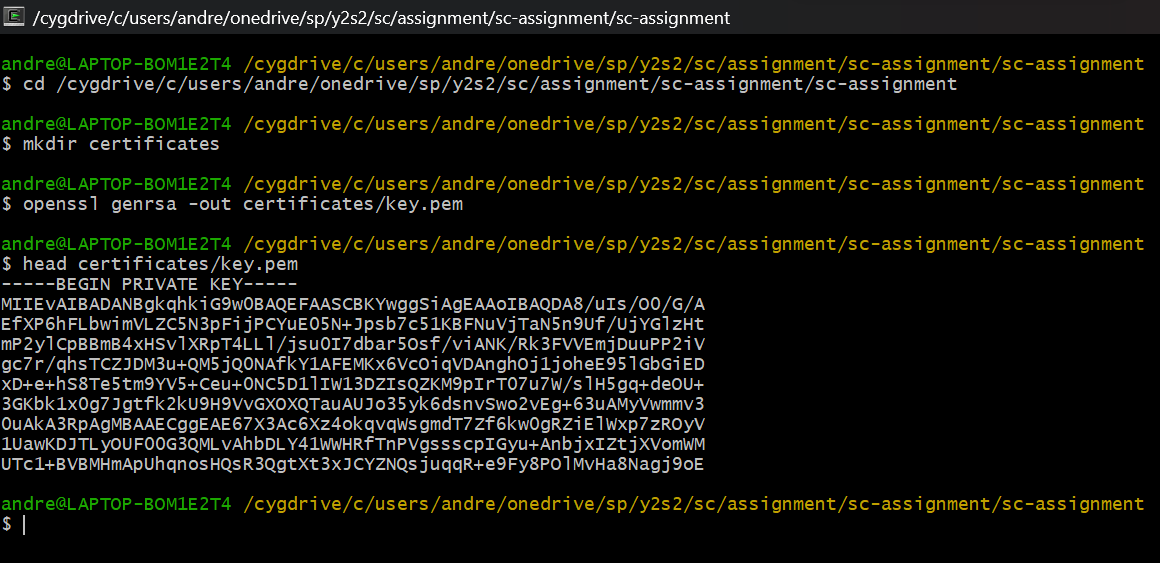
**3. Remediated Code**

In order to add SSL encryption for all the traffic between the client and the backend, we will need to implement HTTP running over SSL/TLS. This is known as HTTPS.

First, we will generate the key and certificate to be used for the encryption. You can use a Linux terminal to generate the key and certificates using the openssl tool. In my case, I used Cygwin on my Windows machine, which emulates a Linux terminal. I first changed the terminal directory into the sc-assignment directory which contains my FrontEnd and BackEnd folders. Here, I created a ‘certificates’ directory. I then ran the command to generate a key inside the certificates directory which is as follows:

**openssl genrsa -out certificates/key.pem**

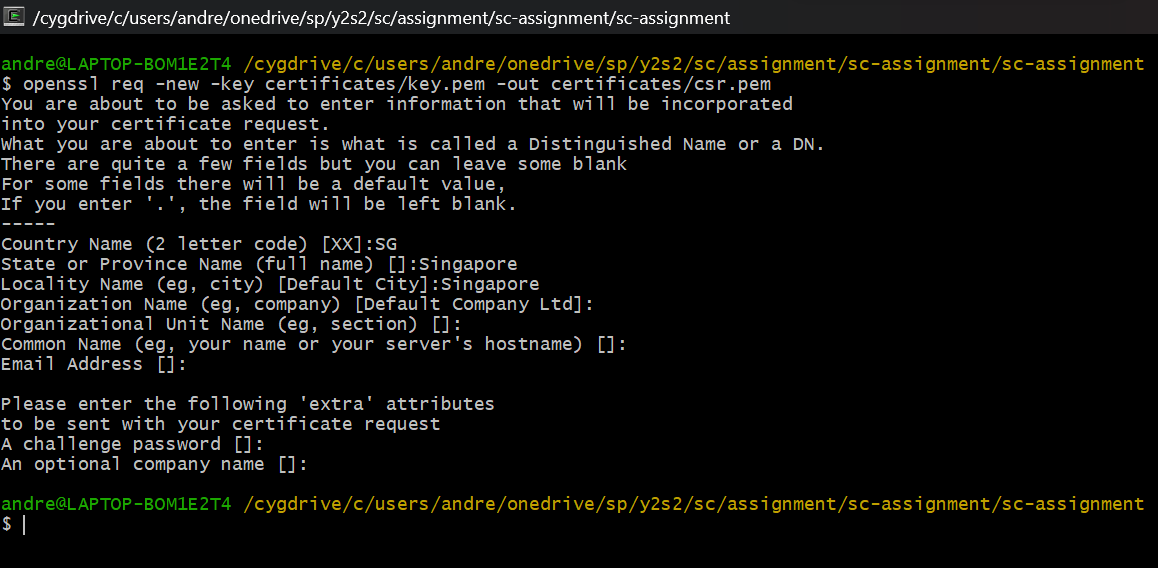
This creates a 2048 bit private key using the RSA algorithm.We can then see using the Linux ‘head’ tool that the key has been generated successfully as key.pem

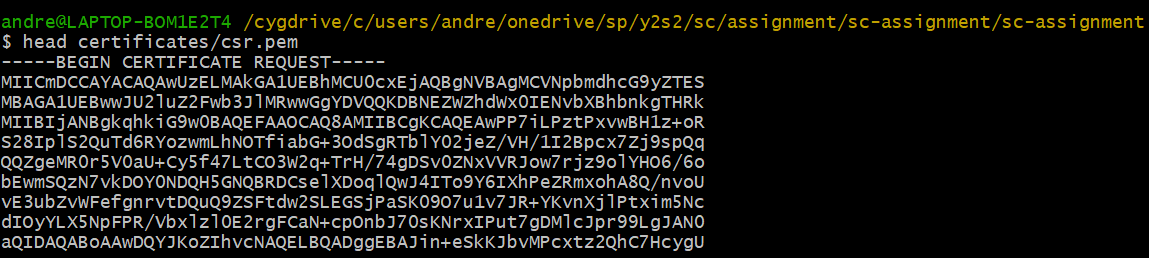
****

Next, we will need to create the Certificate Signing Request (CSR). I used the same Cygwin terminal to run the following:

**openssl req -new -key certificates/key.pem -out certificates/csr.pem**

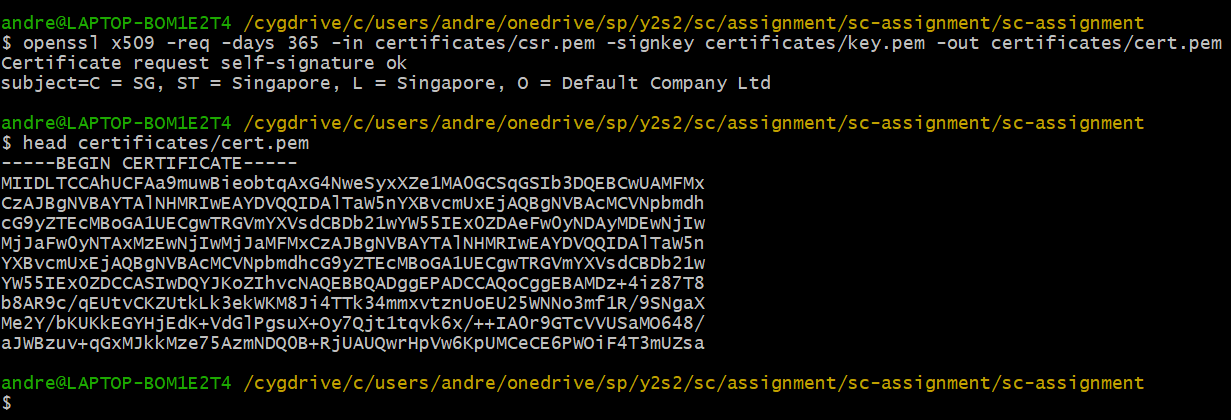
This command uses the key created above to generate the CSR and store it as csr.pem in the certificates directory. I entered some simple parameters for the CSR such as country code (SG) and State name (Singapore).

Using the ‘head’ tool, we can verify that the csr has been created successfully.

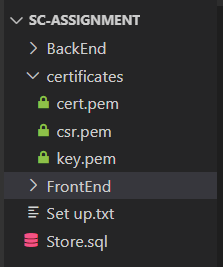


Now, to create the SSL certificate. Once again using the openssl command, I have made the SSL certificate valid for 365 days and used the csr and key which were just created.

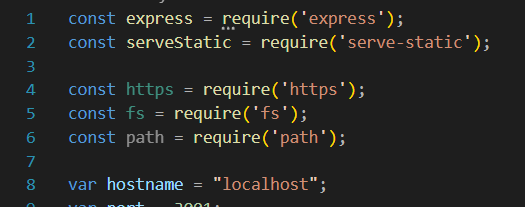
**openssl x509 -req -days 365 -in certificates/csr.pem -signkey certificates/key.pem -out certificates/cert.pem**



With that, the SSL key, CSR and certificate have all been created in the certificates folder

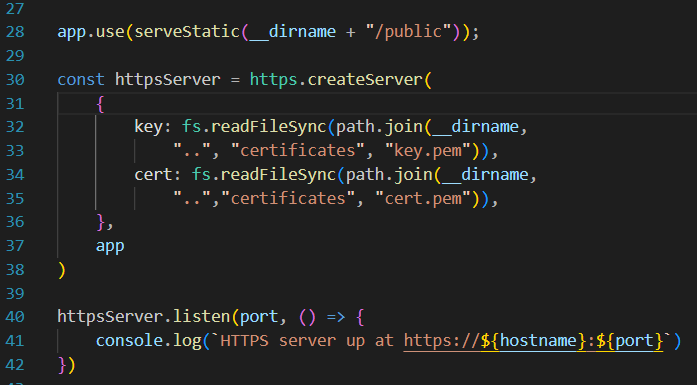


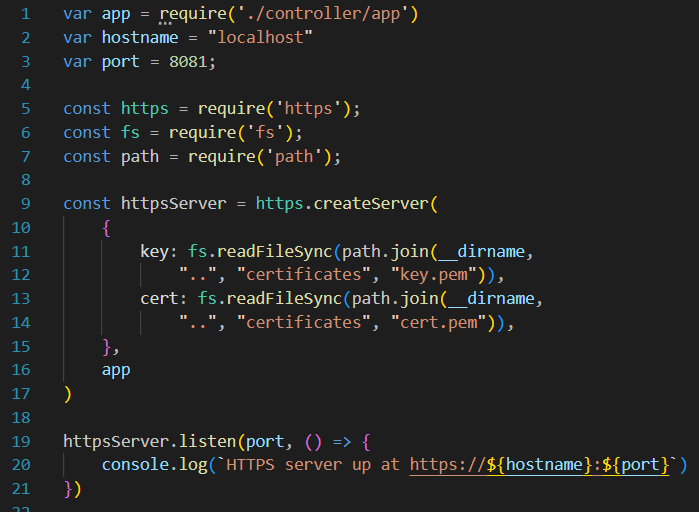
Now, back in the FrontEnd > server.js, we will have to implement HTTPS. First, import the libraries that will be needed for the code:



Next, we will set the https server options. Below the line with code ‘serveStatic’, we will add configurations for https.createServer by defining the key and certificate for the server to use as ‘key.pem’ and ‘cert.pem’ from the certificates folder.

Lastly, we will replace the previous ‘app.listen’ function with the httpsServer.listen function.

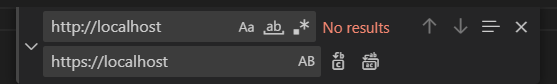


We will add the above items to the server.js in the BackEnd folder as well:  


We can then change all the “http://localhost” redirects in the FrontEnd > Public folder to “<https://localhost>”.

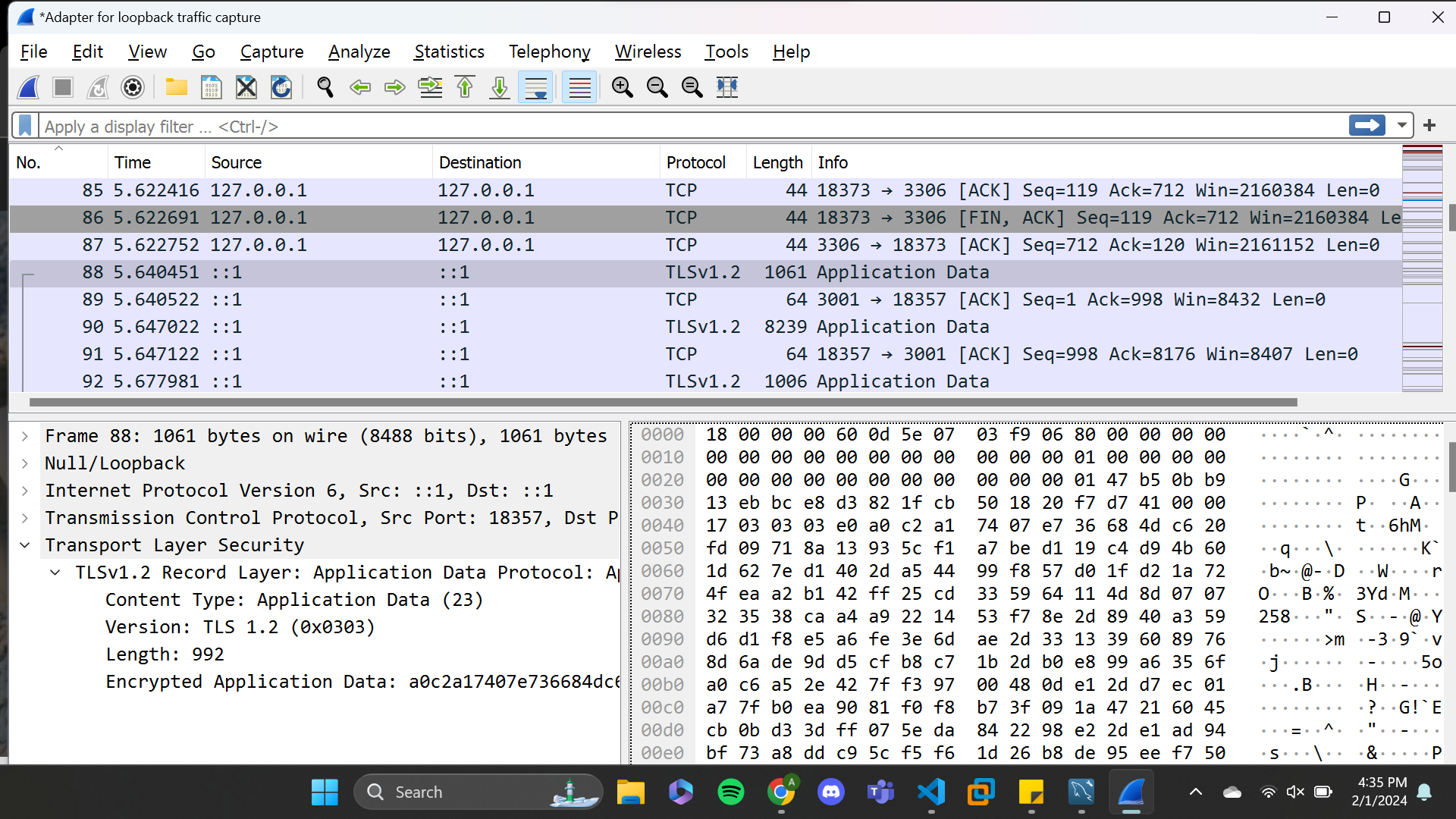




I did this using vscodes Find and Replace All function for all the .html and .js files in the Public folder.  


**4. Testing**

Now if I were to use Wireshark to intercept the packets between the client and the backend, all application data is encrypted as seen below

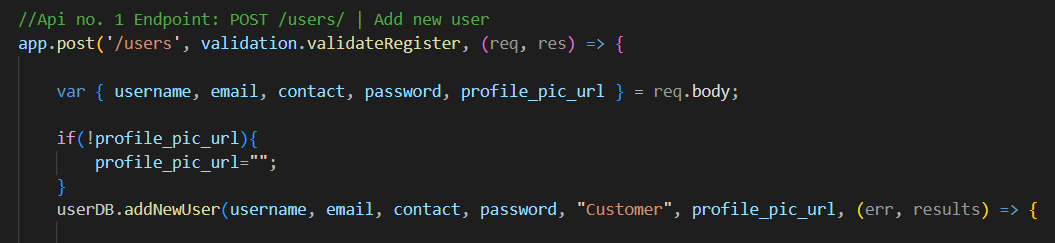


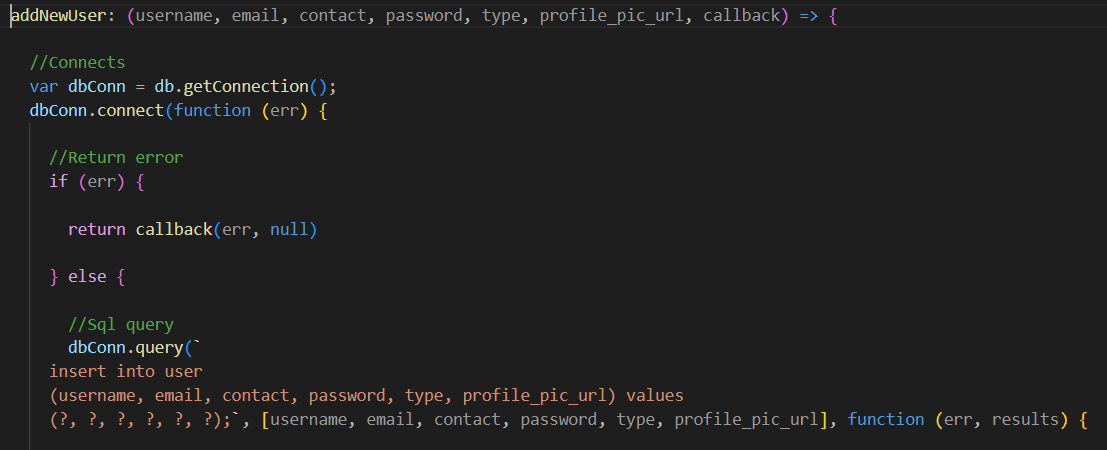
Hence, attackers will not be able to view the data in plaintext anymore. Only the client and backend which have the SSL certificate and key are able to decrypt each others’ messages.

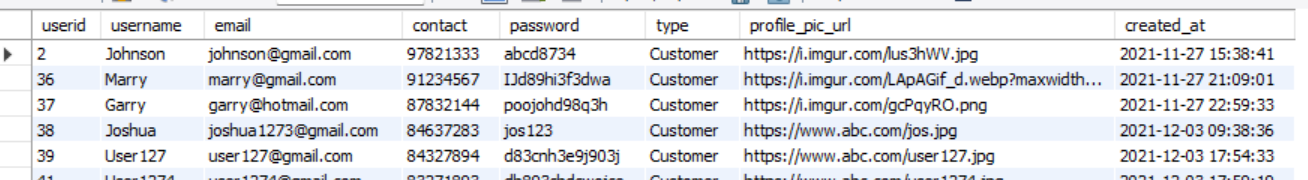
### Vulnerability 2: Stored Passwords are not Hashed (Brief)

**1. Vulnerable Code**

In the code below, we see that the password is not hashed at any point before it is sent in the SQL query.





This results in the password beings stored in the database in plaintext:  


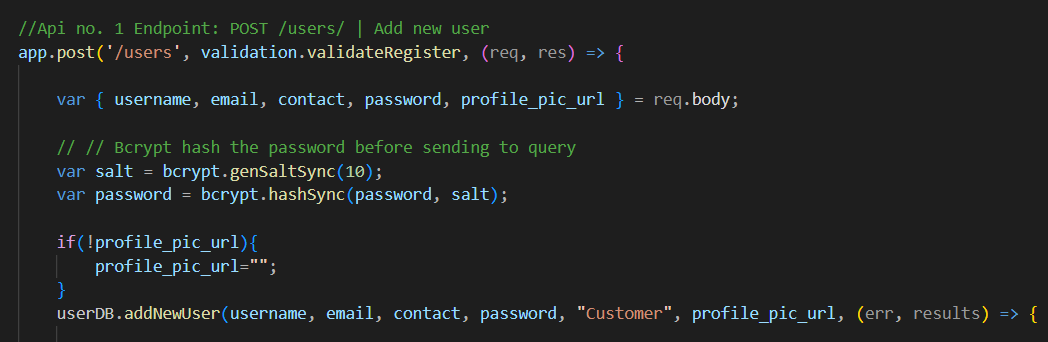
**2. Remediated Code**

In order to resolve this, some form of hashing should be performed before the password is stored. We will use **bcrypt** to hash the passwords as it is designed to have a slow runtime which makes it take much longer for attackers to brute force.

In the terminal of the backend folder, run **npm install bcryptjs**

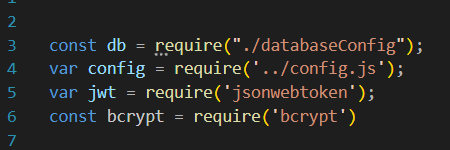
Import the bcryptjs library at the top of the backend’s app.js file



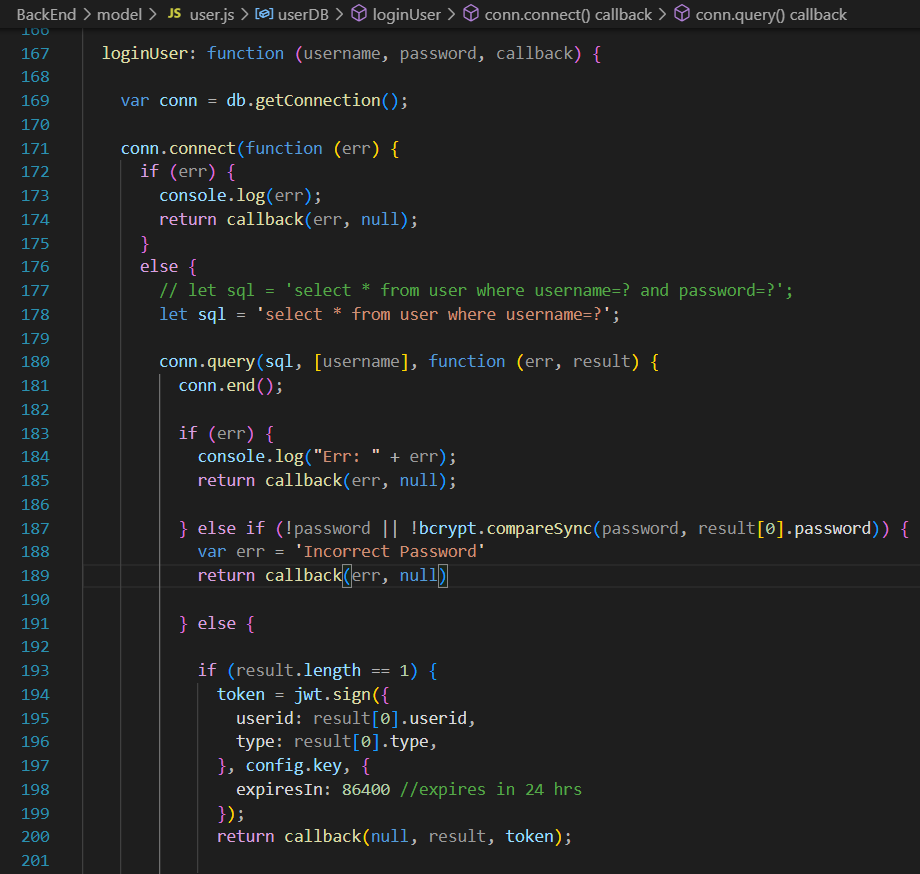
In the add new user API, add a line to generate a salt using the bcrypt library then generate a bcrypt hash of the password. This hashed password will then be stored instead of the plaintext.

Now, users’ passwords are stored hashed instead

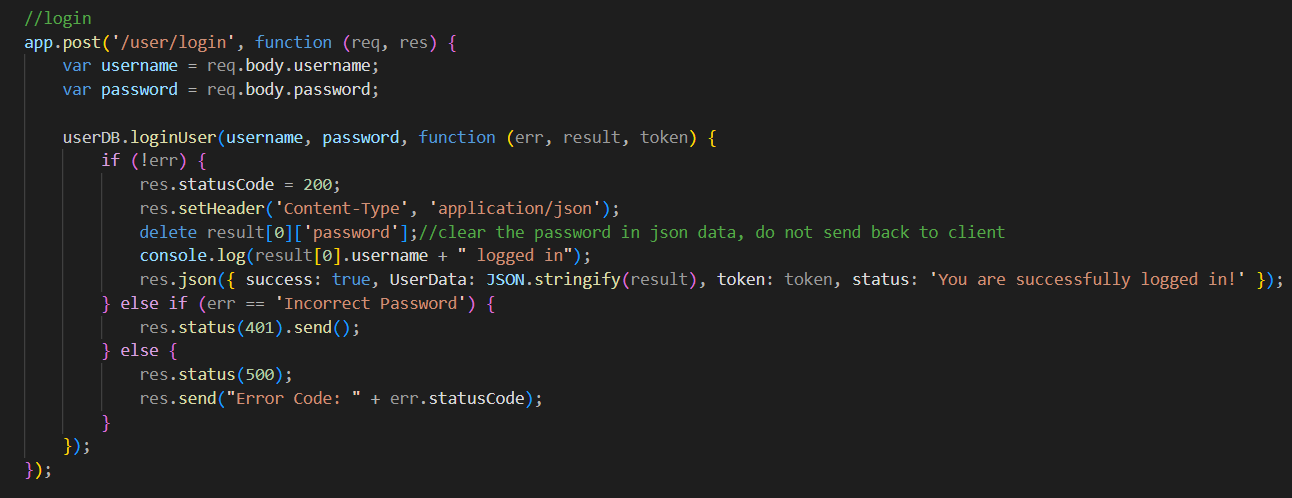
We also need to make sure that the application knows how to compare password hashes during login. To achieve that, we must compare the hashes during the login process. First, import bcrypt into the user.js file where the password comparison will be made:



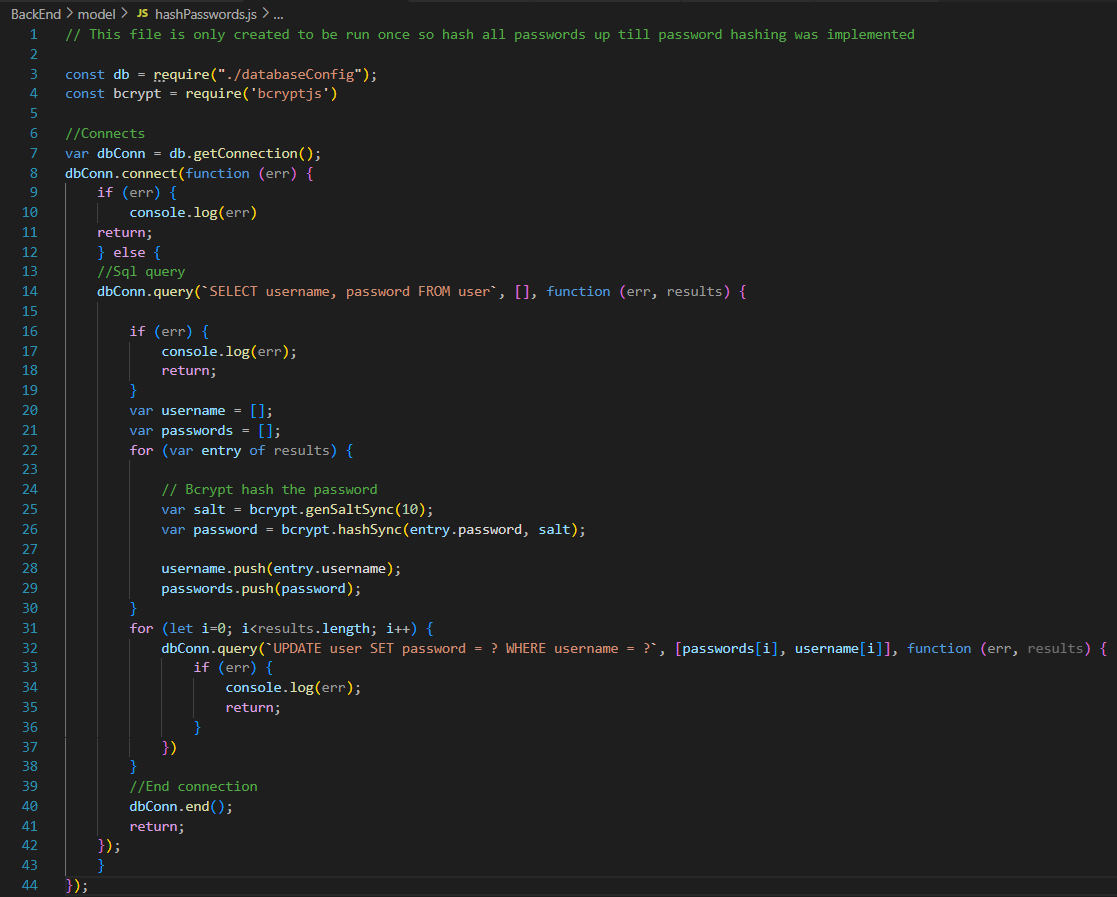
Next is changing the loginUser function (found in BackEnd > model > user.js). I have changed the SQL query for the function to get all information from the table where the username equals to the user’s input username. Then, the hashed password with its salt is retrieved from the database and compared with the password that the user input which will also be hashed with the salt. If the password is correct, the usual login processes will continue. If the password does not match, it will send the error ‘Incorrect Password’ back to the app.js API.



When the app.js API receives the ‘Incorrect Password’ error, it will send a response to the client with the HTTP code of 401:



We then need to reconfigure all the previous passwords to become hashed versions of themselves. Hence, I have created a script called ‘hashPasswords.js’ in the model folder that will take the passwords from the sql database, hash them, and update the users with the new hashed passwords:



With all these in place all passwords past and future will be hashed. Users who had previously signed up will be able to log in, users who will be registering in the future can log in, and incorrect logins will still be rejected as usual.

### Additional Recommendations

For all future network traffic, it should be redirected to the HTTPS protocol before being sent between client and server. In the case that any of the above cryptographic algorithms used are ever broken or expired, it is advised to update the algorithms to whatever is the industry standard.

# Tools

In the above report, the following tools were used for exploitation, testing and/or identifying of vulnerabilities.

| **Tool** | **Description** |
| --- | --- |
| Visual Studio Code | Used to run the web application as well as modify the code in the web application |
| BurpSuite | Used to read and modify requests to the web application, as well as read responses from the web application |
| Wireshark | Used to analyze network packets between the ‘user’ and web application |

# Conclusion

The web server had many vulnerabilities that posed a huge threat to the web servers. Using the theory of defense-in-depth, we have applied a multi-layer approach to harden the web server and protect its information against attackers. Despite the current measures put in place, future development of this web project should maintain the level of security that has been shown in the remediated code above. In general:

* input validation and output sanitization should be applied for all user input that enters and is displayed by the servers to prevent injection attacks like SQL injection and XSS.
* Checking of admin tokens should be applied as middleware on APIs to prevent Broken Access Control vulnerabilities.
* Encryption and hashing should be applied where appropriate. This is applicable to data-in-transit, data-at-rest and data-in-use.

I suggest that the vulnerabilities should be patched as quickly as possible as many of them are highly dangerous. This concludes the web application security assessment.

# Appendix A

Severity Matrix

| Impact / Ease of Exploitation | Low Ease of Exploitation | Medium Ease of Exploitation | High Ease of Exploitation |
| --- | --- | --- | --- |
| Low Impact | Very Low Severity | Low Severity | Medium Severity |
| Medium Impact | Low Severity | Medium Severity | High Severity |
| High Impact | Medium Severity | High Severity | Very High Severity |