Secure Programming
COMP.SEC.300
Exercise Work

Martin Yordanov

martin.yordanov@tuni.fi

Disclosure of AI usage

AI tools were utilized in this project work for the purpose of correct citation in the reference section. ChatGPT version 4 was utilized to correctly format the references in the reference section of this report.

In addition, Grammarly was used to fix punctuation, grammar, and delivery of sentences in the entire work.

Table of contents

General Description

The project is a minimalistic web application with basic CRUD functionality. Its purpose is to mimic the core features of a social media web application. Users can register, create and interact with posts and comments, and delete their own profiles.

Although the UI is minimalistic, attention has been paid to usability, so that the user does not press something they did not mean to. For instance, having the user accidentally press “Delete my profile” and instantly have their profile deleted. Currently, the user is asked to confirm if they want to do that.

User Interface

The guest home page is what users see without an active session. They cannot like a post or leave comments. They cannot go to a specific post page.

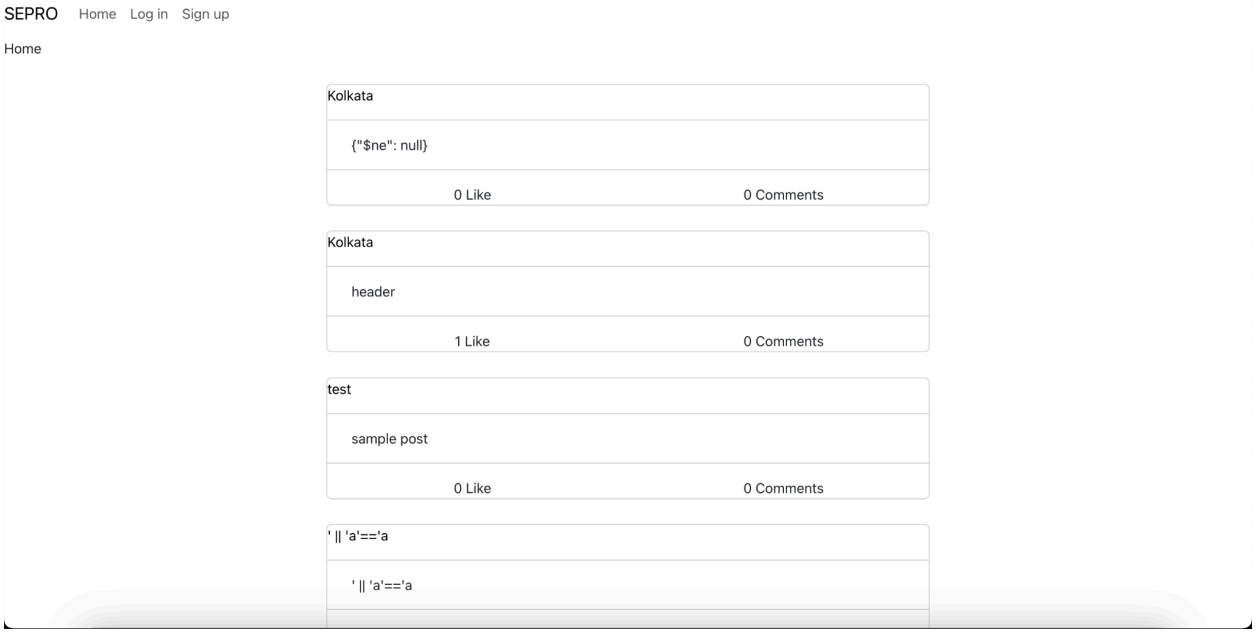

Figure 1. Guest home page

The log in and register pages are accessible only to users without an active session.

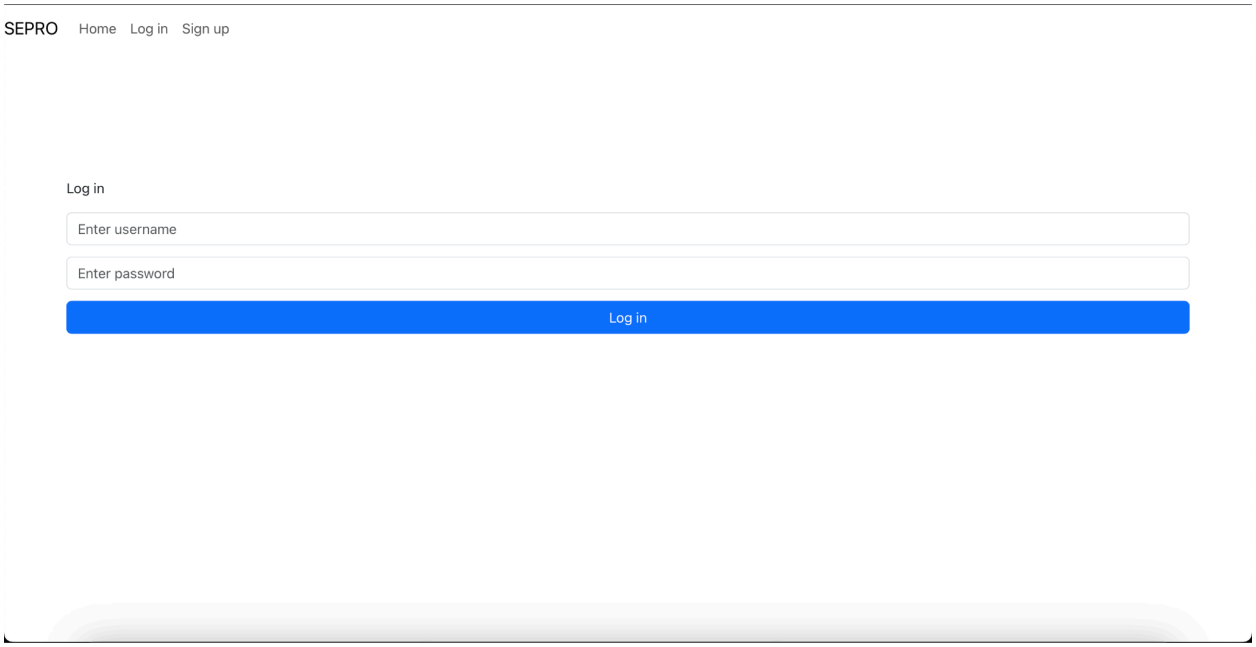

Figure 2. Log in page

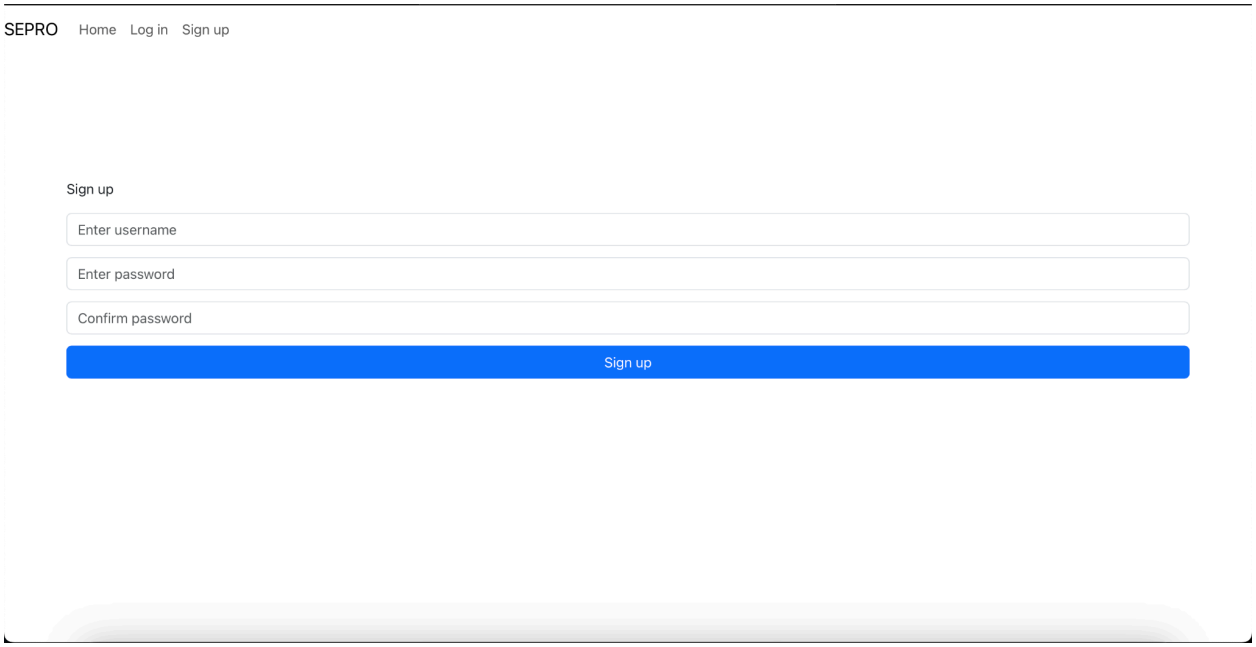

Figure 3. Register page

The fields about creating a post and writing a comment shown on figure 4,5 & 6 are only visible to users with an active session.

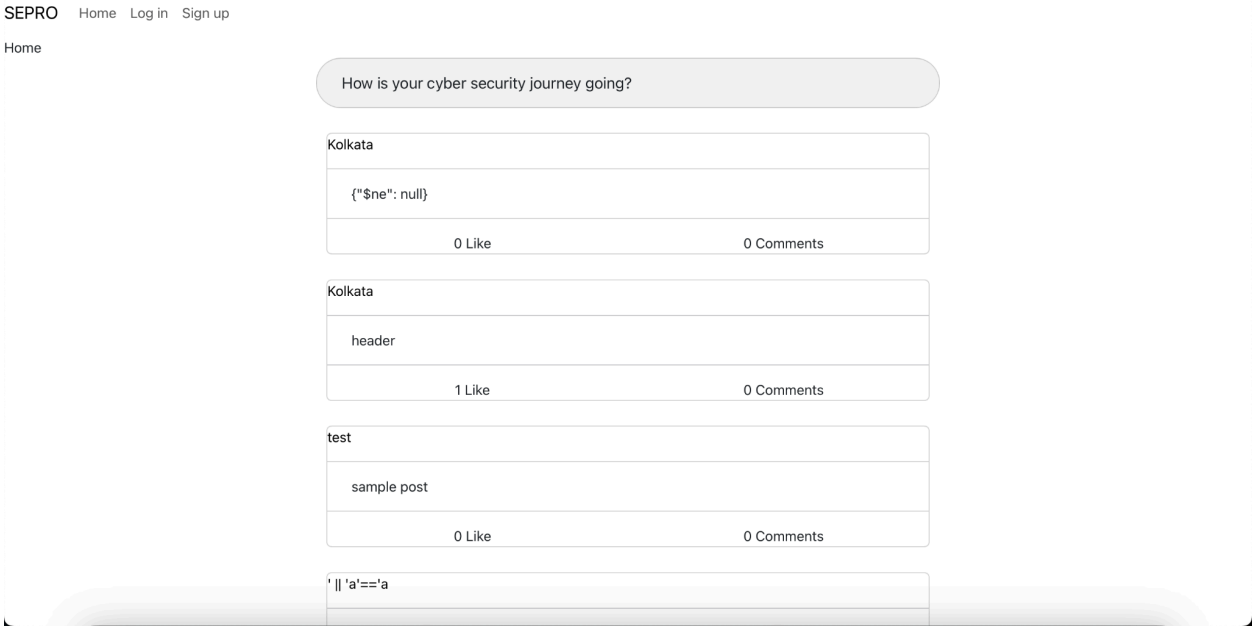

Figure 4. Registered user home page

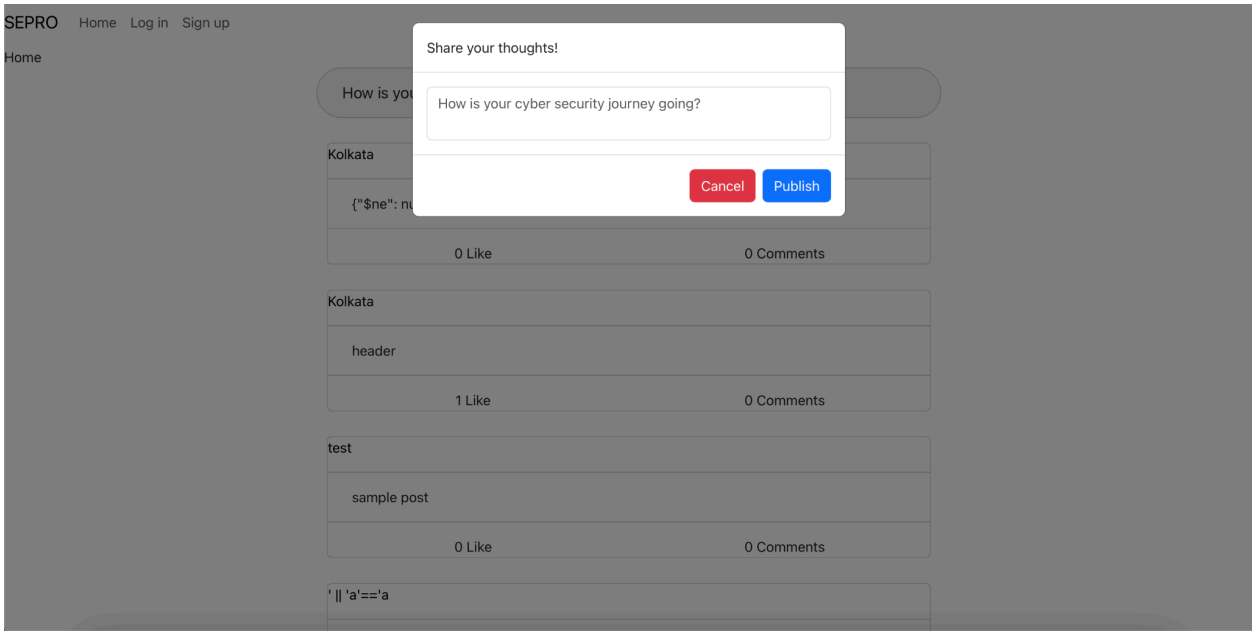

Figure 5. The modal that shows when you press the prompt field.

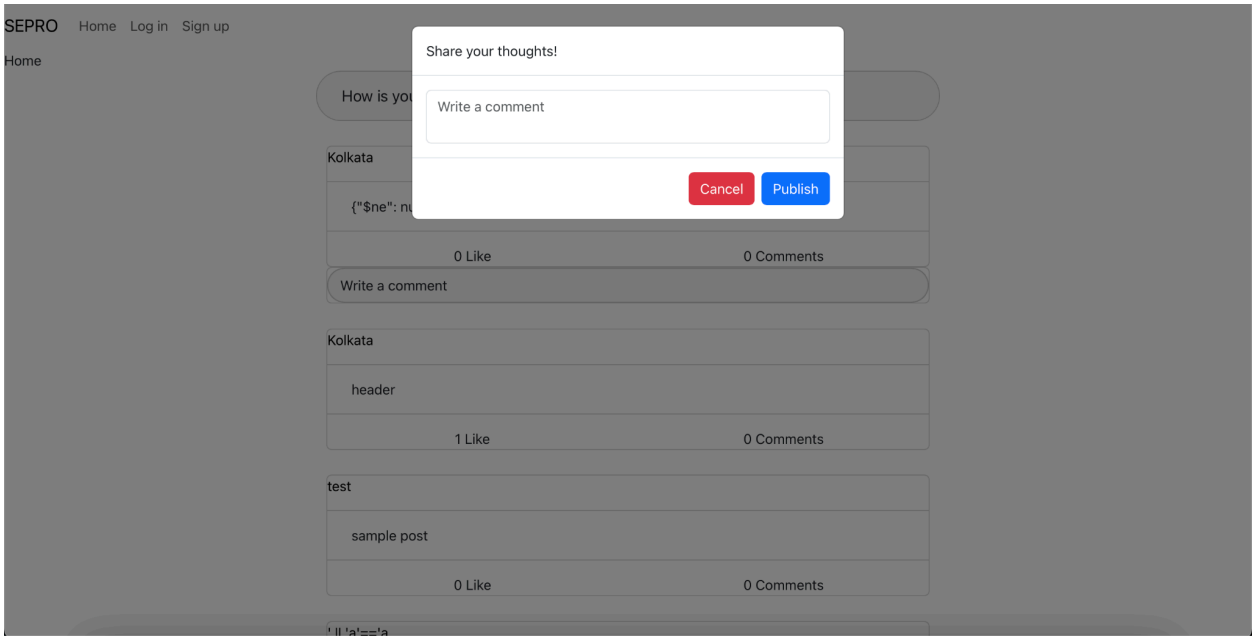

Figure 6. The modal that appears when you press the comment prompt field.

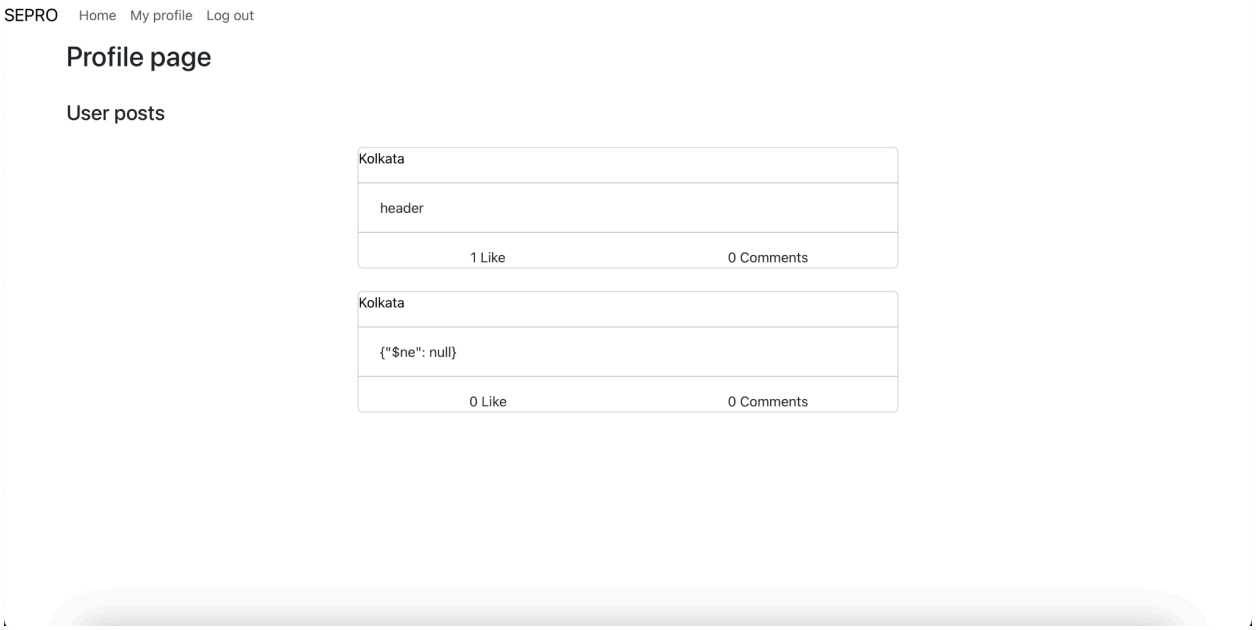

Figure 7. This is the view of when one visits another person's profile page

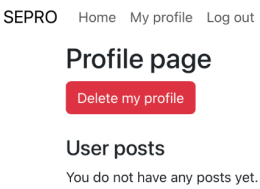

Figure 8. This is the view of when one visits their own profile page

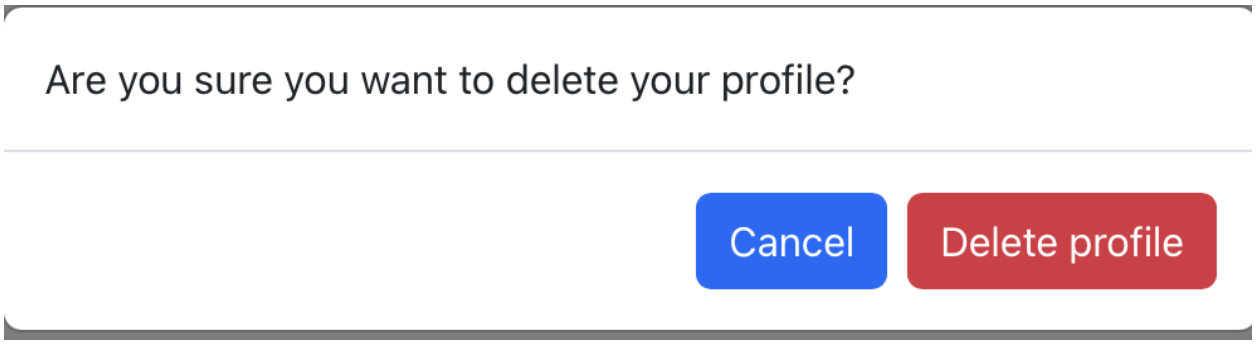

Figure 9. A confirmation modal asking the user to confirm if they want to delete their profile.

Structure of the program

Frontend module

The frontend is built with React and accompanying libraries such as *react-bootstrap* for responsive, mobile-first design, and *react-toastify* for real-time UI notifications on backend response. The frontend consists of eight functional React components - *CreatePostField, Error, Home, Login, Register, Navbar, Post, Profile* - and two *higher-order components* - withAuth, withGuest- used to implement authentication guards for pages such as the profile page, which is only accessible for registered users.

Finally, the frontend utilizes a custom React hook called *UseAuth.* It fetches the */user/isRegistered* endpoint, whose response depends on whether the requesting user is logged in or not. *UseAuth* is used in the two *higher-order components* and in the *Home*, *Navbar*, and *Post* components for conditional rendering.

Backend module

The backend consists of a node with Express server, a non-relational database(MongoDB) with schemas for both a user and a post, two services - user and post services - and a utility file, called Messages. *Figure 10* represents the data flow of the application in a simplified manner.

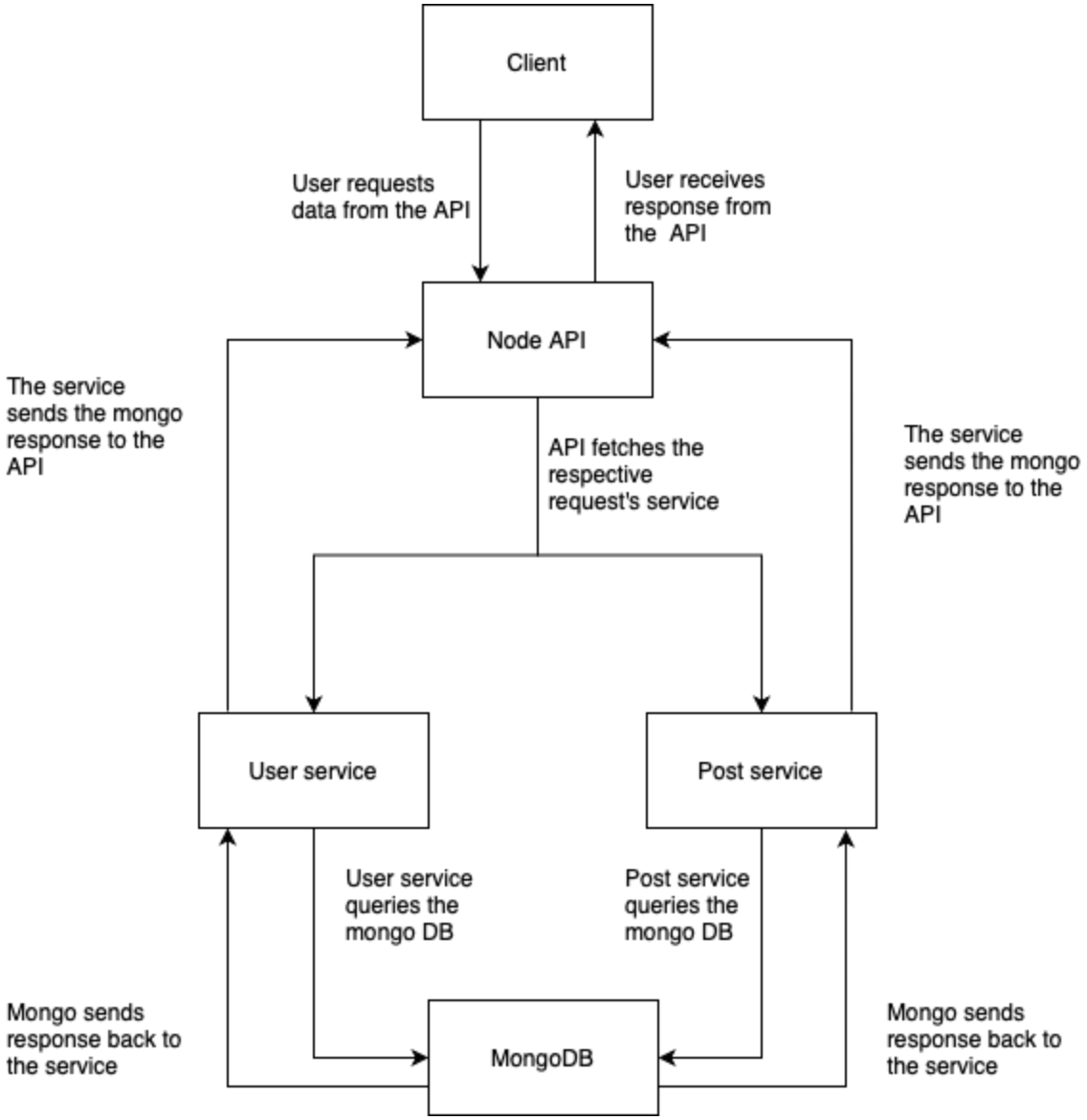

Figure 10. An overview of the client-API/Backend-Database communication

First, let's explore the database.

Database

The User Schema

The user schema is composed of only two string fields - username and password. Both of which are required.

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

const UserSchema = new mongoose.Schema({
    username:  {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
    },

})

const User = mongoose.model('user', UserSchema)

module.exports = User
```

Figure 11. The user schema

The Post Schema

The post schema is composed of five fields - postOwner, postBody, postParent, likes, comments. Note that a comment is a post that has a postParent field, hence the reference to the ‘post’ model in the comment field.

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

const PostModel = new mongoose.Schema({
    postOwner:  {
       type: mongoose.Types.ObjectId,
       ref: 'user',
       required: true
    },
    postBody: {
        type: String,
        required: true
    },
    postParent: {
        type: mongoose.Types.ObjectId,
        ref: 'post',
        required: false
    },
    likes:[{
            type: mongoose.Types.ObjectId,
            ref: 'user'
        }],
    comments:[{
            type: mongoose.Types.ObjectId,
            ref: 'post'
        }]
})

const post = mongoose.model('post', PostModel)

module.exports = post
```

Figure 12. The post schema code.

Next, let's explore the server.

Server

The server consists of twelve user and post-related endpoints. More information about the logic behind the actions performed at the endpoints below can be found in the services section.

User endpoints

1. **/user/login**

This endpoint takes in two *string parameters* from the *request's body* - *username* and *password*. The password is validated using the following regex:

^(?=.*\w)(?=.*[A-Z]){1,}(?=.*\W).{8,}$

It checks if the password contains at least one word character, at least one upper-case letter, at least one non-word/special character and that the password is at least 8 characters long. The stronger the password is, the longer it would take for somebody to "break" the hash(the plaintext passwords are hashed with **bcrypt** by the time they reach the database). Moreover, if the password is unique enough, it would be hard for it to be cracked by password wordlists.

Once the regex validation passes successfully, the user input is sent to the user service, where checks are performed as to whether a user with the provided username exists in the database.

In case that all of the checks have passed, the user is successfully logged in and a server-side session is created. It is later used in other endpoints for authentication and authorization. The said session is stored in the database.

2. **/user/register**

This endpoint takes in the username and password from the request's body. The same regex as in the /login endpoint is applied here. Upon successful regex validation of the password, the data is sent to the *CreateUser* method of the user service where checks are done as to whether a user with a specified username exists. If it does, the plaintext password is hashed with *bcrypt* and sent to the database as a part of the said user's object.

3. **/user/delete/:userID [7]**

This endpoint requires a single request parameter - userID. It uses that parameter's value to compare it with the requester's server-side session userID value. If they match, the user is authorized to execute the delete action and the profile(along with all of the posts associated with it) are deleted. That way broken auth is prevented.

4. **/user/isRegistered**

This endpoint checks if there is an active server-side session, containing userID. All of this is done thanks to session cookies which are HTTP-Only, meaning that javascript itself cannot access the cookie. That way, anybody trying to steal the user's cookies would have to have either physical or system-level access to the said machine, so that they can take the cookie from the browser's dev tools for instance.

5. **/user/get/currentUser**

This endpoint checks if there is an active session corresponding to the requester's session cookie. If such a session exists, the user's ID is returned.

6. **/user/logout**

This endpoint destroys the active server session corresponding to the requester's session cookie. If the requester does not have a session cookie with a session ID corresponding to an active server-side session in the database, then nothing happens.

Post endpoints

1. **/post/create [6]**

This endpoint creates a post. It requires two parameters from the request's body - postContentId and postParentId(not necessary as if a post has a parentId then it is a comment, otherwise it is a post). The postOwnerId is taken from the requester's session. The logic of creating a post or a comment is taken care of in the post service.

2. **/post/delete/:postId**

This endpoint deletes a post given a postId. A check is performed as to whether the requester's session contains the same ID as that of the owner in the database, preventing broken auth.

3. **/post/like/:postId**

This endpoint is used when a user likes or removes their like from a post. It works based on server-side sessions just like the previous endpoints.

4. **/post/getAll**

This endpoint returns all of the posts that are within the database, excluding comments. No authentication or authorization is required here.

5. **/post/fetch/:postId**

This endpoint fetches a specific post based on its postId without a need for any sort of auth. It only takes in the query parameter postId.

6. **post/fetch/owner/:userID**

This endpoint fetches all posts by the specified user id in the query parameter. It does not require any other parameters. It does not need auth.

Now that we have covered the endpoints section, let us take a look at the services, which are an essential part of this project.

Services

User service

The user service is composed of five functions that the node server relies on - CreateUser, LoginUser, DeleteUser, IfUserExists, IfUserExists_Username

CreateUser

This function takes in a username and password as parameters. A check is done as to whether a username with the provided username already exists or not through the **IfUserExists** function. Once the check passes successfully, the plaintext password is hashed with bcrypt and the user object is saved in the database. A respective response is sent to the API, which sends it back to the client.

```
async function CreateUser(username, password){
    let UserObject = await IfUserExists_Username(username)
    if( UserObject.doesUserExist )
    {
        return{status: 409, message: USERNAME_TAKEN}
    }
    else
    {
        const hashedPass = await bcrypt.hash(password, SALT_ROUNDS)
        try{
            let NewUser = await User({
                username: username,
                password: hashedPass
            })

            await NewUser.save()
            return {status: 200, message: REGISTRATION_SUCCESSFUL}
        }
        catch
        {
            return {status: 500, message: REGISTRATION_ERROR}
        }
    }
}
```

Figure 13. The CreateUser method

LoginUser

This function accepts a username and password as parameters. First, a check is performed as to whether the specified username is taken by another user or not. This is done by the **IfUserExists_Username** function. If the check passes successfully, the plaintext password is hashed and the result is compared to the said user's hashed password in the database. If the hashes match, then the login is successful and a server session is created for that said user.

Consequently, an HTTP-Only session cookie is sent to the client.

```
async function LoginUser(username, password)
{
    let UserObject = await IfUserExists_Username(username)
    if(UserObject.doesUserExist)
    {
        if(await bcrypt.compare(password, UserObject.targetUser[0].password))
        {

            return {status: 200, message: LOGIN_SUCCESS, UserId: UserObject.targetUser[0]._id.toString()}
        }
        else
        {
            return {status: 409, message: WRONG_PASSWORD}
        }

    }
    else
    {

        return {status: 404, message: USER_NOT_FOUND}
    }
}
```

Figure 14. The LoginUser method

DeleteUser

This function only takes in userID as a parameter. A check is performed as to whether a user with the said id exists or not. If it does, all of the said user's posts are deleted and then the user itself is deleted. Note that before executing all of that, a check is done on the node backend that the supplied userID parameter matches the userID stored in the requester's

server-side session, thus broken auth is mitigated.

```
async function DeleteUser(userID){
    let UserObject = await IfUserExists(userID)
    if(UserObject.doesUserExist)
    {
        try{
            let targetUserPostIds = (await GetPostsByUser(userID)).targetPost.map((post) => {return post._id})
            await Post.deleteMany({
                _id: {
                    $in: targetUserPostIds
                }
            })
            .then((res) => {
                // console.log(res)
            })
            .catch((err) => {
                // console.log(err)
            })
            await User.findOneAndDelete({_id: userID})

            return {status: 200, message: USER_DELETE_SUCCESS}
        }
        catch(err)
        {
            return {status: 500, message: USER_DELETE_ERROR}
        }
    }
    else
    {
        return {status: 404, message: USER_DELETE_NOT_FOUND}
    }

}
```

Figure 15. The DeleteUser method

IfUserExists

That function takes in a user ID, “queries” the database to check if a user exists with this ID and returns a true or false, depending on the user’s existence. Moreover, if the user exists, the user’s data is sent back to the API endpoint that called this function.

Note that this function cannot be called by itself. The login, register and delete user endpoints are the only ones that can call this function and receive its data. Moreover, the API does not return all of the data to the client. In addition, sensitive information like passwords are hashed, so that even if they fall in the hands of a malicious user, they cannot directly exploit that

piece of information.

```
async function IfUserExists(UserID){
    const targetUser = await User.findById({_id: UserID})
    if(targetUser?._id)
    {
        return {'doesUserExist': true, 'targetUser': targetUser}
    }
    else
    {
        return {'doesUserExist': false}
    }
}
```

Figure 16. The IfUserExists method

IfUserExists_Username

The way this function works is the same as the IfUserExists with the only difference that instead of an id, it takes in a username as a parameter.

```
async function IfUserExists_Username(Username){
    const targetUser = await User.find({username: Username})
    if(targetUser.length > 0)
    {
        return {'doesUserExist': true, 'targetUser': targetUser}
    }
    else
    {
        return {'doesUserExist': false}
    }
}
```

Figure 17. The IfUserExists_Username method

Post service

The post service is composed of six functions - CreatePost, DeletePost, LikePost, GetPostById, GetPostsByUser, GetPostsByUser, GetAllPosts

CreatePost

This function creates a post or a comment. Note that a comment is also a post with the distinction that the comment has a parentPostId field. Without further ado, this function takes in between 2 and 3 parameters if we're creating a comment. In the case of a post, the function takes in a postContent parameter and an ownerId parameter, which comes from the session data. If we're creating a comment, a parentPostId parameter is also supplied. The way that the function works is that first, the post parent is fetched from the database. In the case that it

exists, a post is created and then the post is added to the list of comments of the parent post. If we're creating a post, then a post object is only created without adding anything to any other post's comments array field.

```
async function CreatePost(content, parentPostId, ownerId){
    let postParent = await GetPostById(parentPostId)
    let newPost;
    try{
        if(postParent?.targetPost?.length > 0)
        {
            newPost = await Post({
                postOwner: ownerId,
                postBody: content,
                postParent: parentPostId
            })
            await Post.findByIdAndUpdate(
                postParent?.targetPost[0]._id,
                { $push: { comments: newPost._id } },
                { new: true }
            );
        }
        else
        {
            newPost = await Post({
                postOwner: ownerId,
                postBody: content
            })
        }
        await newPost.save()
        return {status: 200, message: POST_CREATED_SUCCESSFULLY}
    }
    catch(err)
    {
        console.log(err)
        return {status: 501, message: QUERY_ERROR}
    }

}
```

Figure 18. The CreatePost method

DeletePost

This function takes in two parameters - requestUserId and targetPostId. A check is performed as to whether the post owner's ID is the same as the requester's ID. If the check

passes, the post is deleted. Note that the userId is passed from the session id.

```
async function DeletePost(requestUserID, targetPostId){

    try{
        const result = await GetPostById(targetPostId)
        if(result.targetPost[0]?.postOwner._id.toString() == requestUserID)
        {
            let res = await Post.findByIdAndDelete({_id: targetPostId})
            return {status: 200, message: POST_DELETED_SUCCESSFULLY}
        }
        else
        {
            return {status: 401, message: NOT_AUTHORIZED}
        }


    }
    catch(err)
    {
        return {status: 500, message: 'Something went wrong. Check the delete post method logic.', err}
    }

}
```

Figure 19. The DeletePost method

LikePost

This function takes in two parameters - postId and likerId. The way the function works is that it checks and finds if the target post exists. If it does, it inserts the liker id in the likes array field of the said post. Upon a second like on the same post by the same user, the like is

removed thanks to MongoDB's operators, treating the likes field as a set.

```
async function LikePost(postId, likerId){
    try{
        const likerObjectId = new mongoose.Types.ObjectId(likerId);
        await Post.findOneAndUpdate(
            { _id: postId },
            [{
                $set:{
                    likes:{
                        $cond: {
                            if: { $in: [likerObjectId, "$likes"] },
                            then: { $setDifference: ["$likes", [likerObjectId]] },
                            else: { $concatArrays: ["$likes", [likerObjectId]] }
                        }
                    }
                }
            }],
            { new: true }
        );
        return {status: 200, message: POST_LIKE_SUCCESS}
    }
    catch(err)
    {
        return {status: 500, message: CHECK_LIKE_POST_LOGIC, err}
    }
}
```

Figure 20. The LikePost method

GetPostById

This function takes in one parameter - postId. The way the function works is that it finds the post with the given id and returns the response with populated fields for postOwner and comments, so that the client does not only see objectIds for the said fields, but the actual

objects that the said field is referring to.

```
async function GetPostById(postId){
    try
    {
        let targetPost = await Post.find({_id: postId})
        .populate('postOwner', 'username')
        .populate({path:'comments',
            populate: {
                path: 'postOwner',
                select: 'username'
            }})
        return {status: 200, message: POST_FETCHED_SUCCESS, targetPost}
    }
    catch(err)
    {
        return {status: 500, message: POST_FETCH_ERROR, err}
    }
}
```

Figure 21. The GetPostById method

GetPostsByUser

This function works the same way as the **GetPostById** one with the exception that it takes in a userId parameter instead of postId. Another difference is that all of the posts that the user has with the postParent field are not fetched, so that the user's comments are not returned as a part of the response.

```
async function GetPostsByUser(UserID){
    try
    {
        let targetPost = await Post.find({postOwner: UserID, postParent: { $exists: false }})
        .populate('postOwner', 'username')
        .populate({path:'comments',
            populate: {
                path: 'postOwner',
                select: 'username'
            }})
        return {status: 200, message: POST_FETCHED_SUCCESS, targetPost}
    }
    catch(err)
    {
        return {status: 500, message: POST_FETCH_ERROR, err}
    }
}
```

Figure 22. The GetPostsByUser method

GetAllPosts

This function takes no parameters. It fetches the database to return all of the posts that have no parentPost field. The population is the same as in **GetPostById.**

```
async function GetAllPosts()
{
    try
    {
        let allPostsList = await Post.find({
            postParent: { $exists: false }
        })
        .populate('postOwner', 'username')
        .populate({path: 'comments',
            populate: {
              path: 'postOwner',
              select: 'username'
            }})

        return {status: 200, message: POST_FETCHED_SUCCESS, allPostsList}
    }
    catch(err)
    {
        return {status: 500, message: POST_FETCH_ERROR, err}
    }
}
```

Figure 23. The GetAllPosts method

Docker

Docker has been utilized in this project to ensure cross-platform compatibility and the latest image versions of Node and mongo.

Dockerhub is regularly updated with the latest releases of MongoDB and Node images, which are the only images used in this project. That means that when building the project with Docker Compose, all of the images utilized will be of the latest version, compared to the version utilized at the time of building the project.

Not only that, Docker Compose is also utilized to simplify running and building the project. In addition, all of the containers created with Docker Compose are automatically created on the same docker network, making it easier for me to implement the communication between the containers because I do not have to write local IP addresses(or public in the case of deploying the app) and such. We would have to use Docker Compose services' names instead.

Secure programming solutions

Great attention was paid to the security aspect of this project during development. I have followed the OWASP top 10 web application security risks and analyzed the project's attack vectors to the best of my ability. For this purpose, I have created a detailed diagram shown on figure 24. It covers the API and security flow of the application, which is useful when looking for new potential attack vectors and vulnerabilities that could be exploited later on. For instance,

one of the vulnerabilities found thanks to the diagram led to NoSQL injection in the security testing phase.

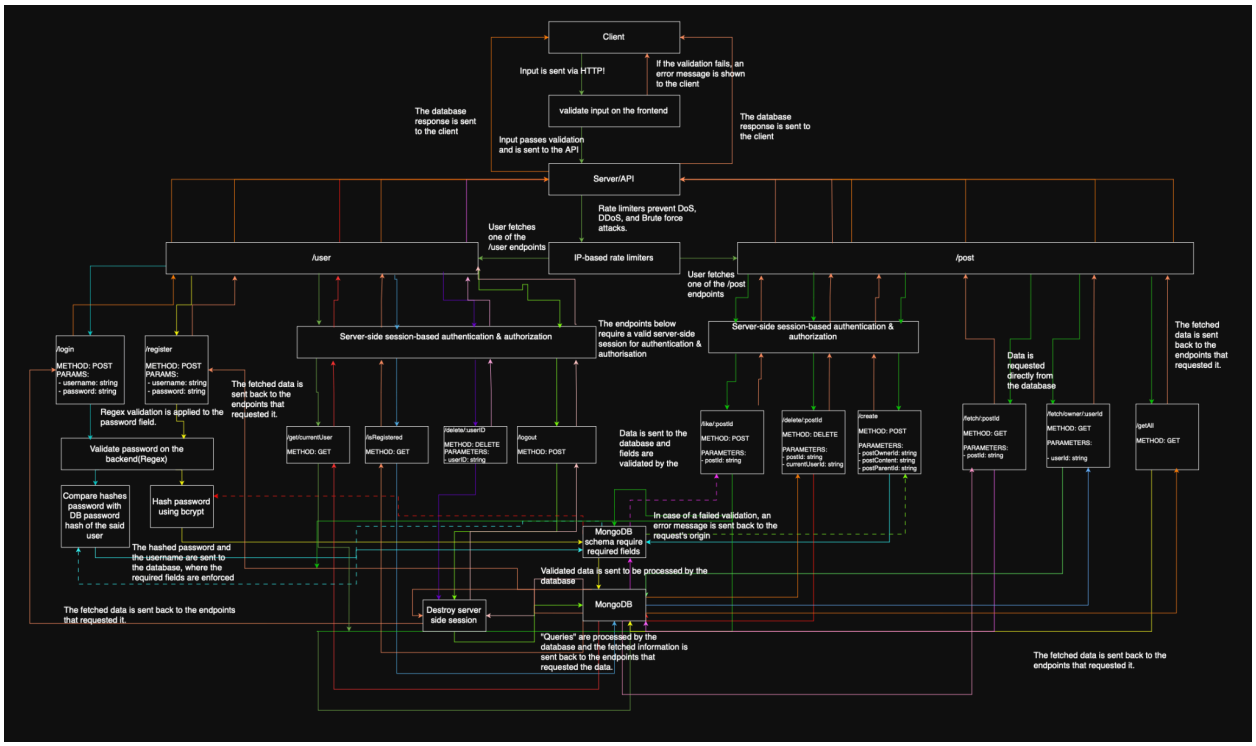

Figure 24. API and Security flow of the web application

I also implemented input validation on the backend as an attempt to mitigate injection attacks. It works by applying the aforementioned regex to the password field. During the testing phase, no NoSQL injection payloads passed the password regex.

```
app.post('/user/register', AuthRateLimiter, async (req,res) => {
    try
    {
        let username = req.body.username
        let password = req.body.password.trim()
        if(!PASSWORD_REGEX.test(password))
        {
            res.status(403).send({message:PASSWORD_CRITERIA_NOT_MET})
        }
        else
        {
            let result = await CreateUser(username, password)

            res.status(result.status).send({message: result.message})
        }
    }
    catch(err)
    {
        res.status(500).send({'message': INTERNAL_SERVER_ERROR, err})
    }

})
```

Figure 25. Regex is applied on the password field on the register endpoint.

```
app.post('/user/login', AuthRateLimiter, async (req,res)  => {
    try{
        let username = req.body.username
        let password = req.body.password.trim()

        if(!PASSWORD_REGEX.test(password))
        {
            res.status(403).send({message: PASSWORD_CRITERIA_NOT_MET})
        }
        else
        {
            let result = await LoginUser(username, password)
            if(result.status == 200)
            {
                const userID = result.UserId
                req.session.userID = userID
                req.session.save(()=>{});
            }
            res.status(result.status).send({message: result.message})
        }
    }
```

Figure 26. Regex is applied on the password field in the login endpoint.

Server-side sessions are implemented for session-based authentication and authorization.

```
app.use(session({
    store: MongoSessionStore,
    cookie: {
        maxAge: 3600*60*24*7,
        path: '/',
        httpOnly: true
    },
    saveUninitialized: false,
    secret: process.env.SESSION_SECRET
}))
```

Figure 27. The server-side session and session cookie configurations.

Special attention has been paid to the security session cookies that are used to identify a user's corresponding server-side session. The cookies have been configured to expire after a week since their creation and they have been set up as HttpOnly cookies, meaning that they are not accessible by javascript running on the client's browser, mitigating the threat of a hacker stealing the cookies on the client's browser through cross-site scripting.

Cross-origin resource sharing(CORS) was also set up to only allow requests from specified origins only, mitigating the risk of requests coming from unwanted hosts, leading to potential breaches.

```
app.use(cors({
    origin: [process.env.ORIGIN, process.env.REMOTE_ORIGIN],
    credentials: true
}))
```

Figure 28. The CORS configuration.

IP-based rate limiters were set up, using the *express-rate-limit* library. Their purpose is to prevent brute force, DoS, or DDoS attacks. Three rate-limiters were set up for the auth, post & any other endpoints.

```
const AuthRateLimiter = rateLimit({
    windowMs: 1*60*1000,
    max: 20,
    message: {message: RATE_LIMIT_MESSAGE}
})

const PostRateLimiter = rateLimit({
    windowMs: 2*60*1000,
    max: 100,
    message: {message: RATE_LIMIT_MESSAGE}
})

const RegularRateLimiter = rateLimit({
    windowMs: 1*60*1000,
    max: 1000,
    message: {message: RATE_LIMIT_MESSAGE}
})
```

Figure 29. The three rate limiters.

In addition to all of the tools utilized, GitHub's dependabot has been utilized to detect vulnerabilities and secrets in the github repository of the project.

During the test, security misconfigurations were found and later remediated. For instance, an environment file was created to avoid leaking secrets and other sensitive data on github, such as addresses & connection strings. The environment file's extension has been added to the .gitignore file, so that the .env file does not end up on github by accident later on, exposing sensitive information.

As for cross-site scripting, react is known to take care of XSS thanks to JSX.[1] XSS was not observed during the testing.

Security testing

The security testing consisted of two stages - the vulnerability assessment and penetration testing.

The goal of the first was to perform static analysis to find vulnerabilities in the project in terms of docker images, docker files, dependencies, etc. As for the penetration test, the goal was to exploit the found vulnerabilities during the vulnerability assessment and look for other vulnerabilities that were to be later remediated. It's important to note that both manual and automated penetration testing has been performed.

Vulnerability assessment

The vulnerability assessment was performed with the following tools:

1. **Tenable Nessus** for automated vulnerability scanning of the web application,
2. **Trivy** for vulnerability scanning the Dockerfile-created images, and
3. **Snyk** for finding vulnerabilities in the project's dependencies.
4. **Metasploit** was utilized to search databases of vulnerability exploits in regards to the dependencies of the project.
5. **npm audit** was executed to find vulnerabilities within the package.json dependencies.

No vulnerabilities were found with Tenable Nessus except for a warning about a missing or permissive http header is a potential attack vector.

Here is the nessus configuration that was used when performing the basic network scan.

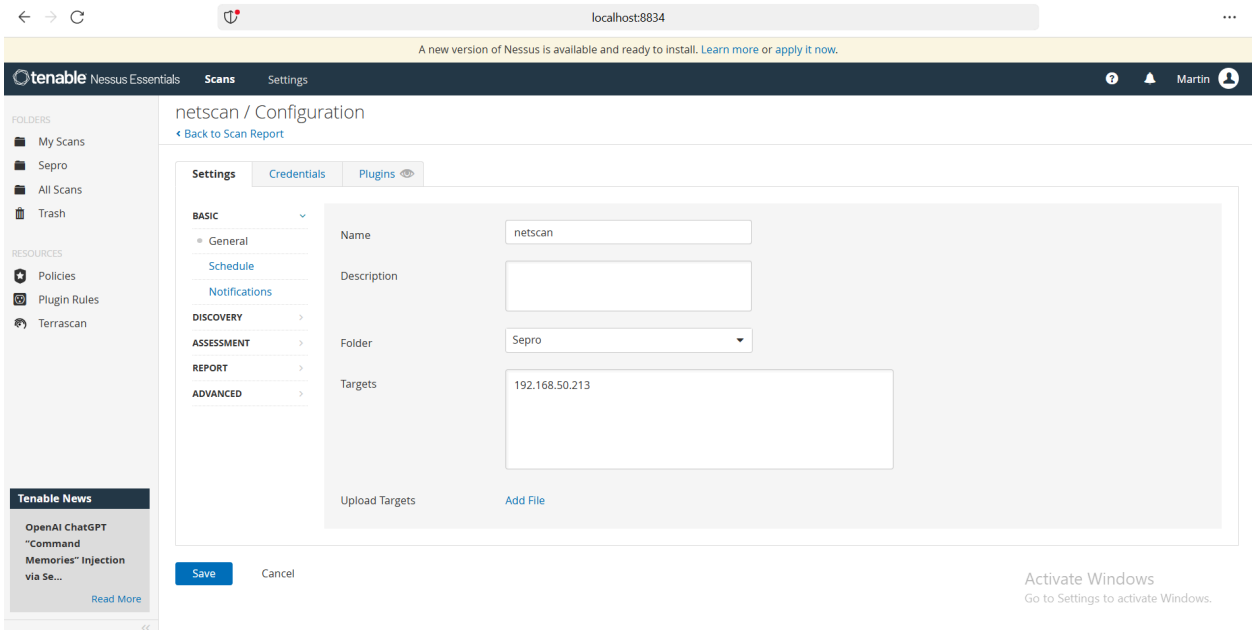

Figure 30. Setting the netscan target to the project's host

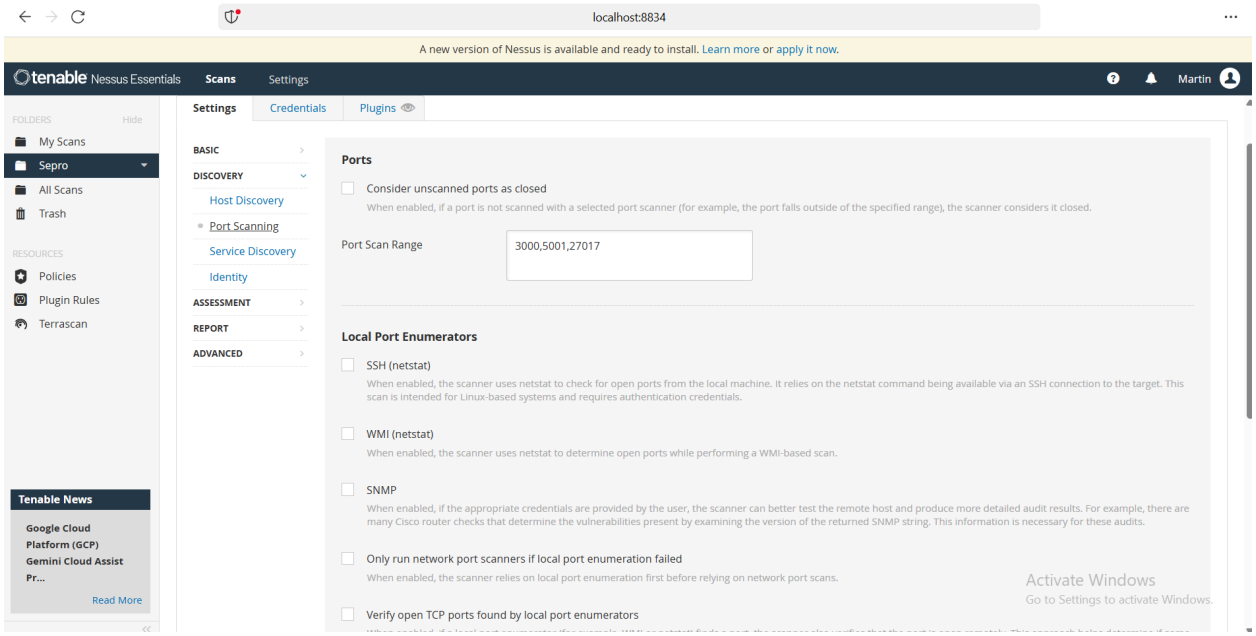

Figure 31. Configuring Nessus to only scan the frontend, backend and mongo ports.

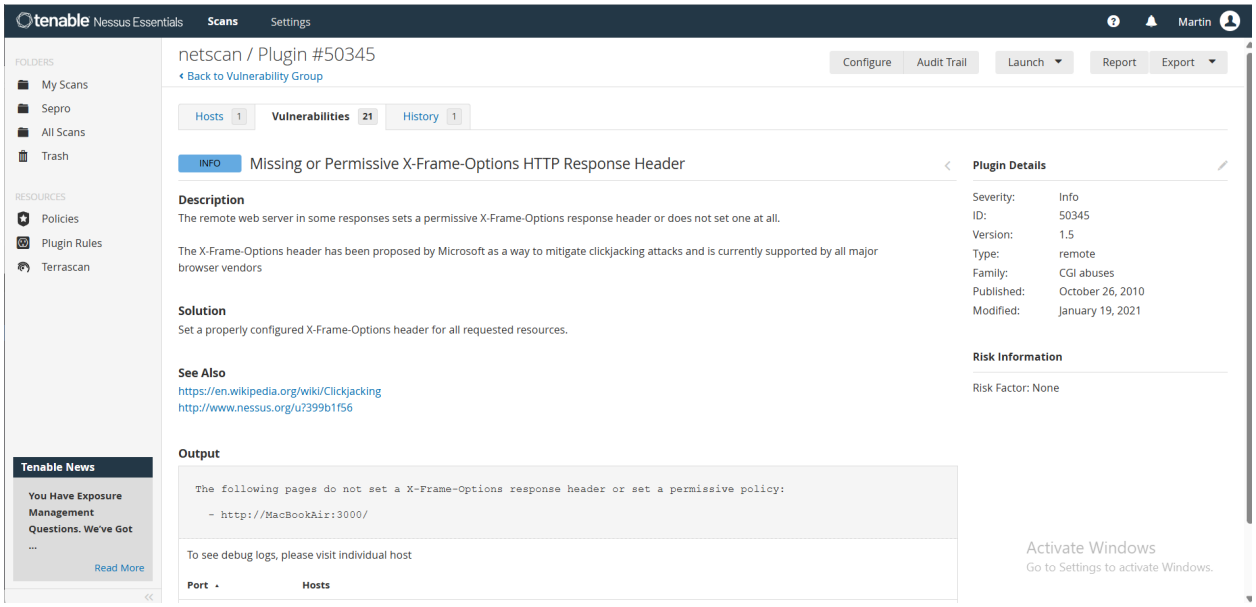

Figure 32. Nessus informs us about a potential clickjacking issue.

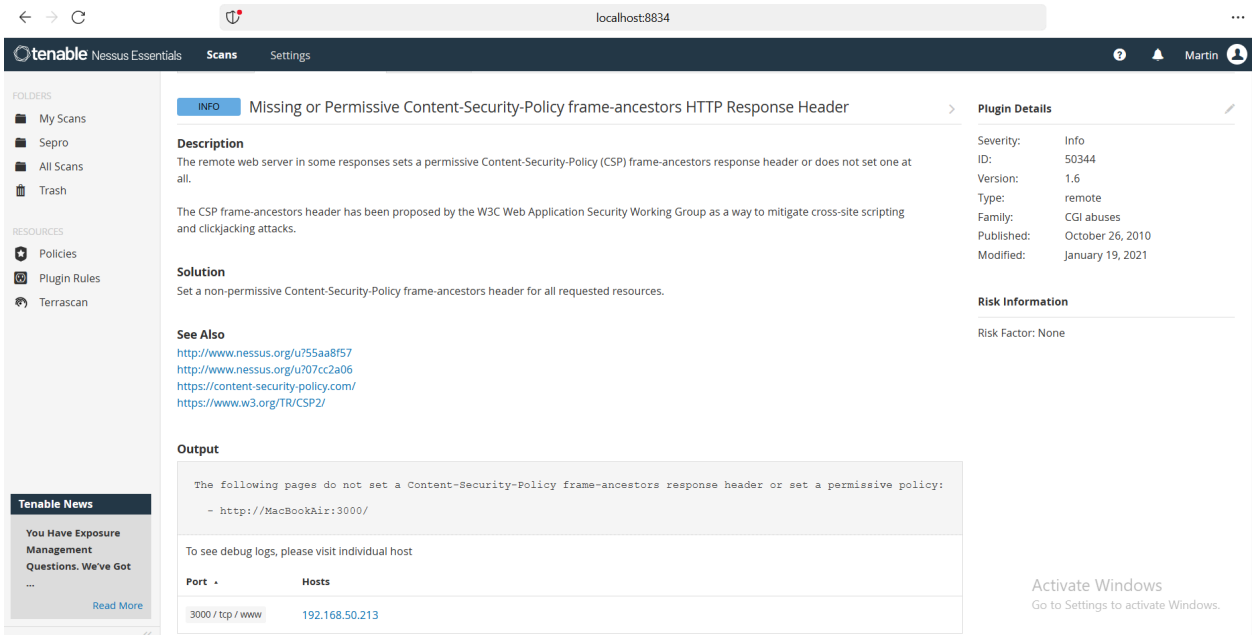

Figure 33. Nessus informs us about another potential attack vector.

However, a single tool cannot be trusted by itself to find or rule out vulnerabilities, as false positives and false negatives exist, thus, more scanning and research are needed. One thing that Nessus did, however, was to successfully enumerate the project's MongoDB version, which could have been useful if the project utilized an older image for MongoDB and we were looking for publicly known vulnerabilities in vulnerability databases.

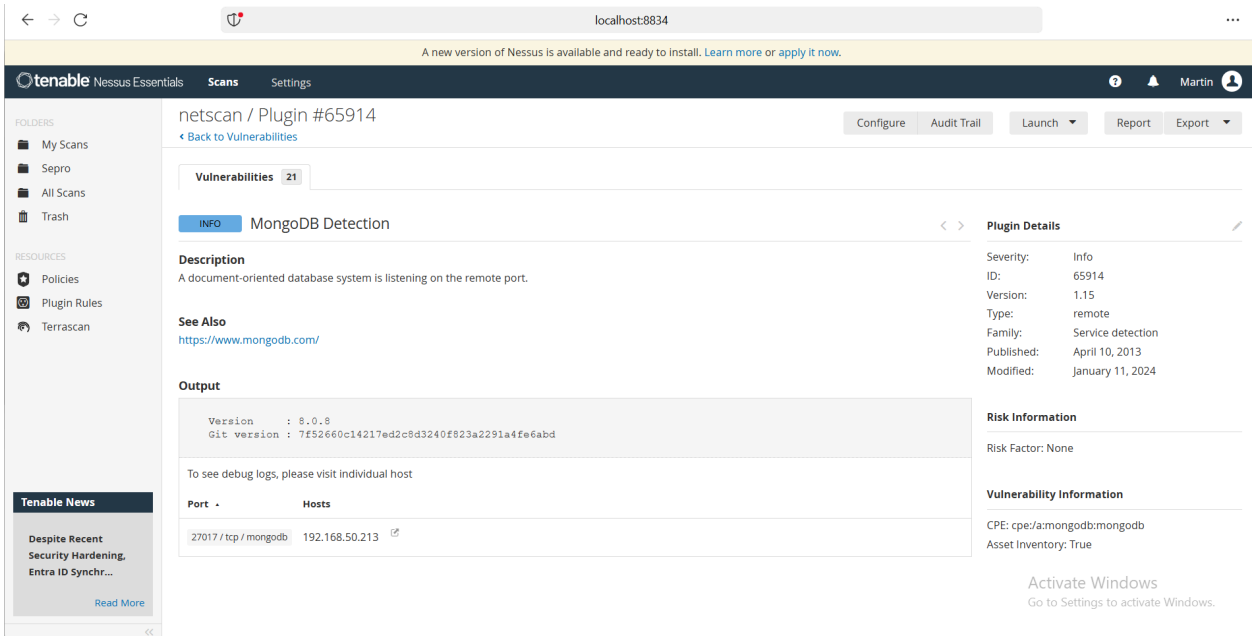

Figure 34. Nessus detected the used MongoDB version

Moreover, Nessus also found that the robots.txt file is accessible. That is not a direct security concern by itself but it could become one if paths to sensitive directories or files are listed there. This, however, does not concern this project.

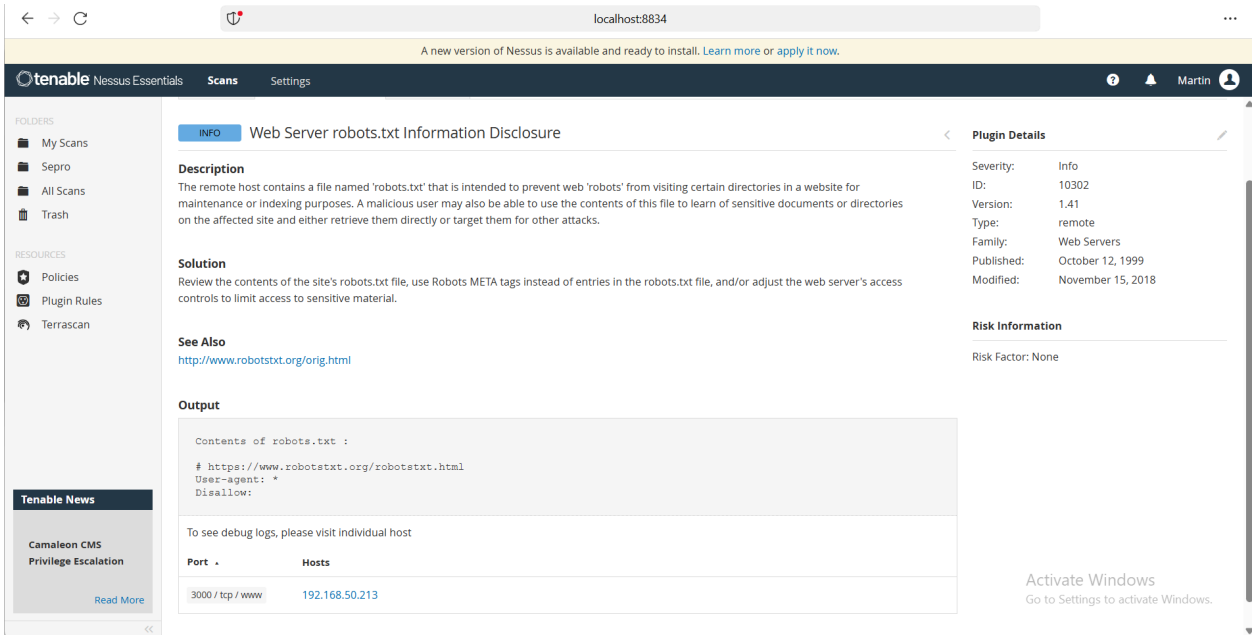

Figure 35. Nessus detects the robots file.

ExploitDB (a database of publicly-known vulnerability exploits) was searched for known exploits for the detected version of MongoDB, but none were found. Metasploit's search was also used for the same purpose.

Figure 36. Start metasploit with the msfconsole command

Figure 37. Search ExploitDB's records for the detected MongoDB version

```
msf6 > search mongo

Matching Modules
================

   #  Name                                                  Disclosure Date  Rank     Check  Description
   -  ----                                                  ---------------  ----     -----  -----------
   0  post/linux/gather/enum_users_history                  .                normal   No     Linux Gather User History
   1  auxiliary/scanner/mongodb/mongodb_login               .                normal   No     MongoDB Login Utility
   2  auxiliary/gather/mongodb_js_inject_collection_enum    2014-06-07       normal   No     MongoDB NoSQL Collection Enumeration Via Injection
   3  auxiliary/gather/mongodb_ops_manager_diagnostic_archive_info  2023-06-09       normal   Yes    MongoDB Ops Manager Diagnostic Archive Sensitive Information Retriever
   4  exploit/linux/misc/mongod_native_helper               2013-03-24       normal   No     MongoDB nativeHelper.apply Remote Code Execution
   5  exploit/linux/local/netfilter_nft_set_elem_init_privesc  2022-02-07       average  Yes    Netfilter nft_set_elem_init Heap Overflow Privilege Escalation

Interact with a module by name or index. For example info 5, use 5 or use exploit/linux/local/netfilter_nft_set_elem_init_privesc
```

Figure 38. Search Metasploit's records for a known MongoDB vulnerability exploit, corresponding to the version used in the project or later

However, it is important to keep in mind that this does not rule out any potential zero-day vulnerabilities.

Trivy was utilized to scan the docker images generated by the Dockerfiles for the mongo, frontend, and backend services from the docker compose file. Note that both Dockerfiles use the node image

☐	seproproject-frontend e11de2b2c9b8	latest	In use	less than a mi	1.55 GB	▶ ⋮ 🗑
☐	seproproject-backend 6ad57de2696f	latest	In use	less than a mi	1.55 GB	▶ ⋮ 🗑

Figure 39. The two images that Trivy was ran on

```
martin — -zsh — 204×55
(base) martin@MacBookAir ~ % trivy image seproproject-backend --format table --severity CRITICAL,HIGH,MEDIUM --ignore-unfixed
2025-04-25T20:50:15+03:00       INFO    [vuln] Vulnerability scanning is enabled
2025-04-25T20:50:15+03:00       INFO    [secret] Secret scanning is enabled
2025-04-25T20:50:15+03:00       INFO    [secret] If your scanning is slow, please try '--scanners vuln' to disable secret scanning
2025-04-25T20:50:15+03:00       INFO    [secret] Please see also https://trivy.dev/v0.61/docs/scanner/secret#recommendation for faster secret detection
2025-04-25T20:50:16+03:00       INFO    Detected OS     family="debian" version="12.10"
2025-04-25T20:50:16+03:00       INFO    [debian] Detecting vulnerabilities...   os_version="12" pkg_num=413
2025-04-25T20:50:16+03:00       INFO    Number of language-specific files       num=1
2025-04-25T20:50:16+03:00       INFO    [node-pkg] Detecting vulnerabilities...
2025-04-25T20:50:16+03:00       WARN    Using severities from other vendors for some vulnerabilities. Read https://trivy.dev/v0.61/docs/scanner/vulnerability#severity-selection for details.
2025-04-25T20:50:16+03:00       INFO    Table result includes only package filenames. Use '--format json' option to get the full path to the package file.

Report Summary
```

Figure 40. Run Trivy on the backend image [3]

Node.js (node-pkg)

Total: 6 (UNKNOWN: 0, LOW: 0, MEDIUM: 3, HIGH: 3, CRITICAL: 0)

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version	Title
http-proxy-middleware (package.json)	CVE-2025-32996	MEDIUM	fixed	2.0.7	2.0.8, 3.0.4	http-proxy-middleware: Always-Incorrect Control Flow Implementation in http-proxy-middleware https://avd.aquasec.com/nvd/cve-2025-32996
	CVE-2025-32997				2.0.9, 3.0.5	http-proxy-middleware: Improper Check for Unusual or Exceptional Conditions in http-proxy-middleware https://avd.aquasec.com/nvd/cve-2025-32997
nth-check (package.json)	CVE-2021-3803	HIGH		1.0.2	2.0.1	nodejs-nth-check: inefficient regular expression complexity https://avd.aquasec.com/nvd/cve-2021-3803
postcss (package.json)	CVE-2023-44270	MEDIUM		7.0.39	8.4.31	PostCSS: Improper input validation in PostCSS https://avd.aquasec.com/nvd/cve-2023-44270
react-router (package.json)	CVE-2025-43864	HIGH		7.4.0	7.5.2	React Router allows a DoS via cache poisoning by forcing SPA mode... https://avd.aquasec.com/nvd/cve-2025-43864
	CVE-2025-43865					React Router allows pre-render data spoofing on React-Router framework mode https://avd.aquasec.com/nvd/cve-2025-43865

```
(base) martin@MacBookAir Backend %
```

Figure 41. Result from the Trivy's scan for the latest node image version at the time of writing this annotation.

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version
libperl5.36	CVE-2024-56406	HIGH	fixed	5.36.0-7+deb12u1	5.36.0-7+deb12u2
linux-libc-dev	CVE-2024-26982 CVE-2024-35866 CVE-2024-50246 CVE-2024-53166 CVE-2024-58002 CVE-2025-21702 CVE-2025-21855 CVE-2025-21858 CVE-2025-21905 CVE-2025-21919 CVE-2025-21920 CVE-2025-21926 CVE-2025-21928 CVE-2025-21934 CVE-2025-21945 CVE-2025-21968 CVE-2025-21979 CVE-2025-21991 CVE-2025-21993	MEDIUM	fixed	6.1.129-1	6.1.133-1
perl	CVE-2024-56406	HIGH	fixed	5.36.0-7+deb12u1	5.36.0-7+deb12u2
perl-base	CVE-2024-56406	HIGH	fixed	5.36.0-7+deb12u1	5.36.0-7+deb12u2
perl-modules-5.36	CVE-2024-56406	HIGH	fixed	5.36.0-7+deb12u1	5.36.0-7+deb12u2

Table 1. Trivy's result for the found and fixed vulnerabilities with a severity higher than or equal to "MEDIUM".

Note that none of the vulnerabilities were directly in the images that the service is built on, i.e. the backend service relies on the latest NodeJS image released on Dockerhub. However, the vulnerabilities that Trivy found are related to the dependencies that those images rely on.

As for Snyk, it found mostly the same vulnerabilities(which have an available fix) as Trivy did.

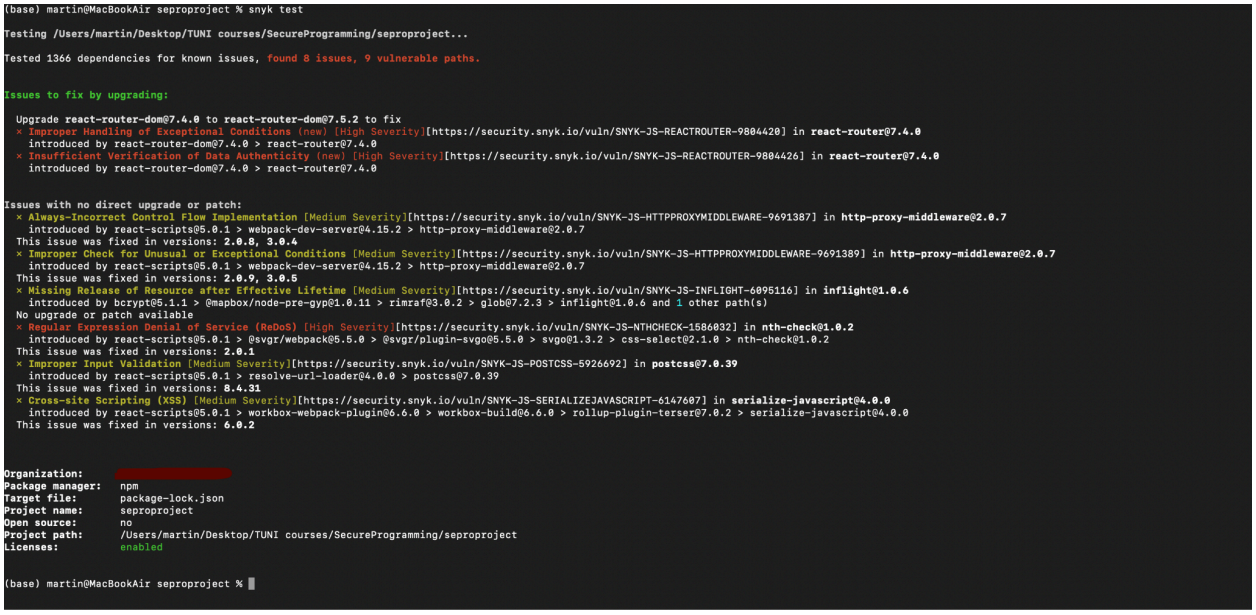

Figure 42. Snyk's test result.

npm audit was also utilized to look for vulnerabilities.

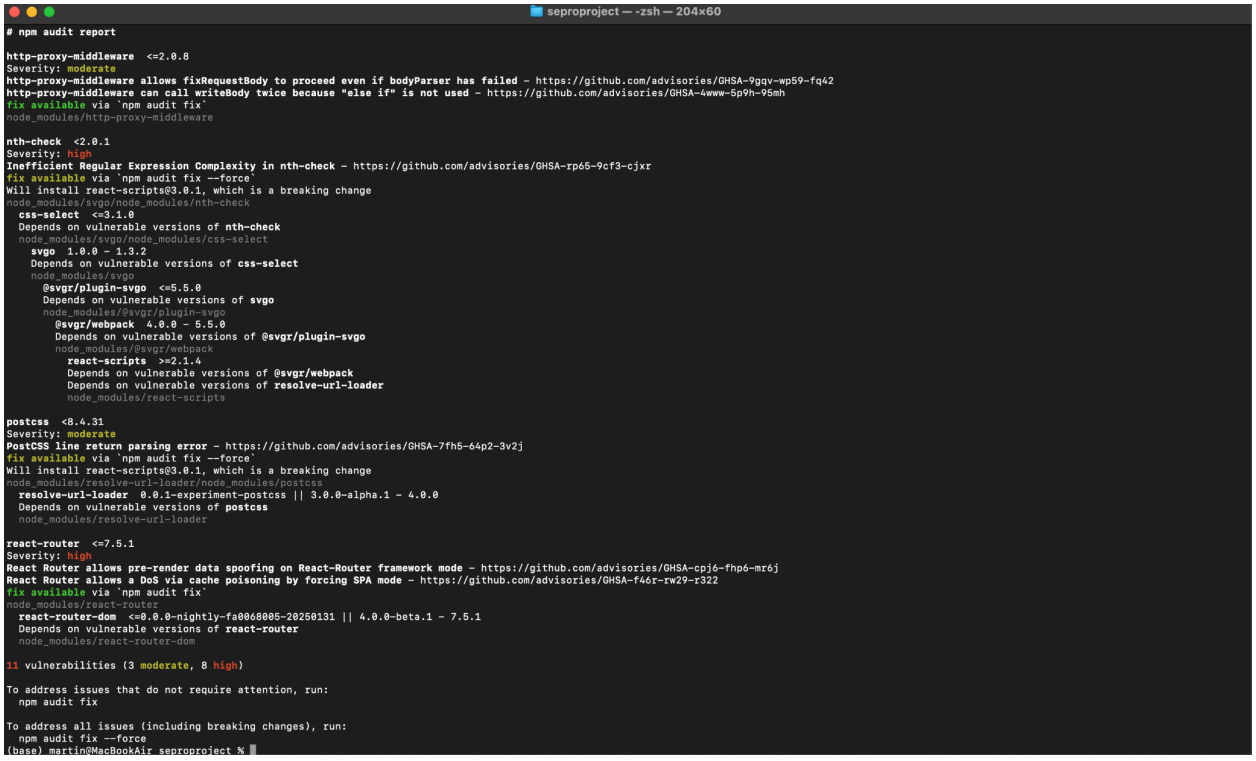

Figure 43. The result of npm audit

Finally, GitHub's Dependabot was also set up on the GitHub repository for this project. It looks for vulnerabilities as well as secrets in the code.

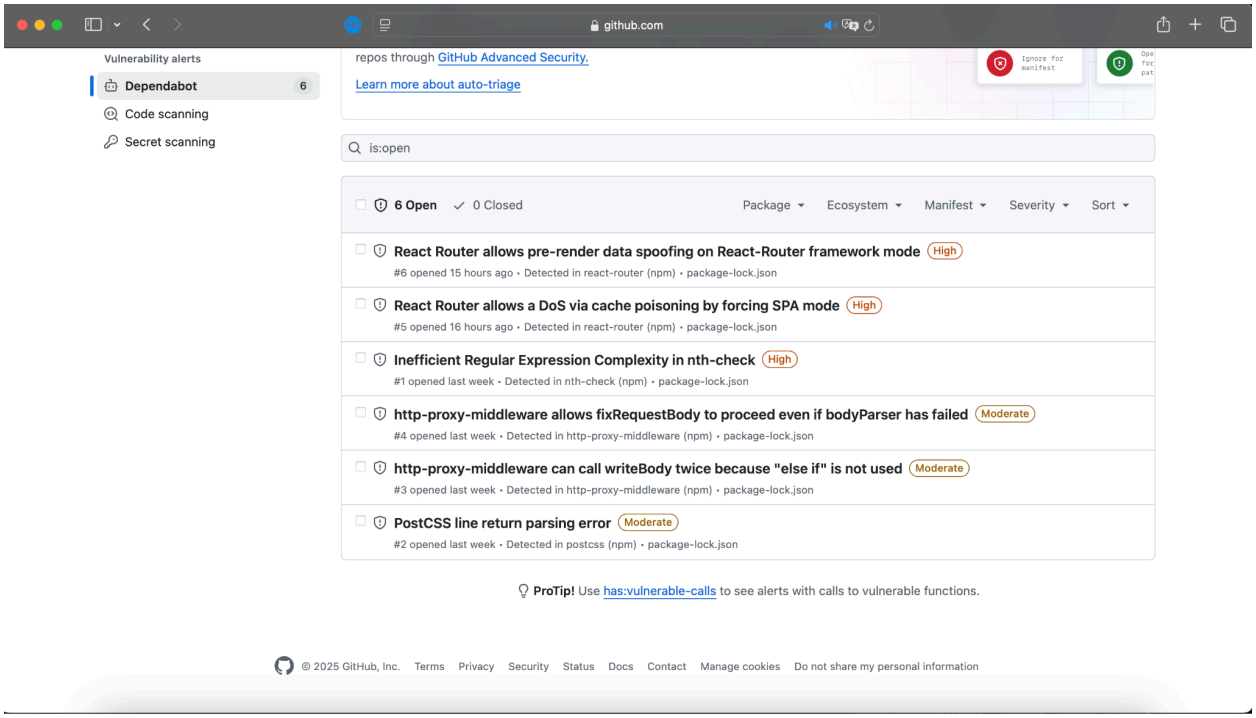

Figure 44. The result of GitHub's Dependabot scans.

Penetration test

Executive summary

Time period: April 18th, 2025 - April 24th, 2025
Scope: Local network(192.168.50.213)

Tools and goals
The penetration test relied on tools such as:

- **NMAP** for port scanning
- **Metasploit** for vulnerability exploit searching
- **Wireshark** for intercepting network traffic
- **Hashcat** for attempts at brute-forcing potential weak passwords
- **Postman** for API testing and attempting to bypass front-end sanitization

Goals: Find *and exploit* vulnerabilities from the **OWASP TOP 10** list with priority as well as any other vulnerability that may be discovered during the test. Note that the test was performed in a ***white box scenario***, removing the need for a complete port scan, directory enumeration, etc.

Found and exploited vulnerabilities

1. Unauthorized database access, which led to account takeover regardless of password hashing (OWASP TOP 10 security misconfiguration[4])

2. User credentials are transmitted through HTTP, thus a MITM attack was successfully executed, leading to stolen user credentials. (OWASP TOP 10 cryptographic failures[4])

3. NoSQL injection was discovered in all authentication input fields. Although it could not be exploited to gain sensitive data, it still poses a security risk and has to be mitigated. (OWASP TOP 10 Injection[4])

Unauthorized database access

Figure 45. Connecting to mongo without any authentication requirement

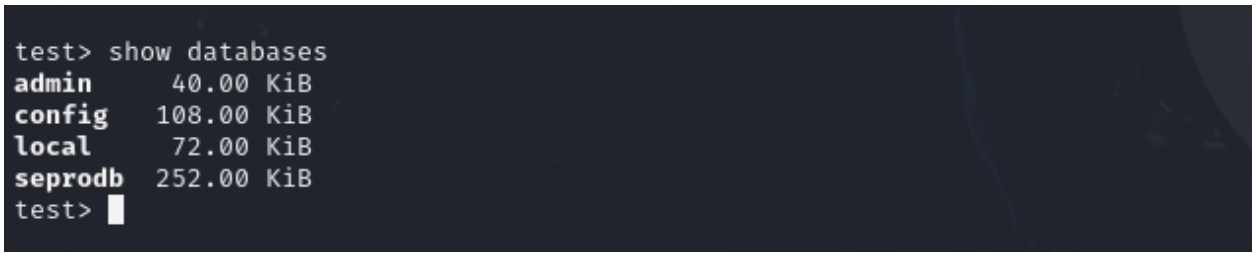

Figure 46. Check all of the databases on the mongo server

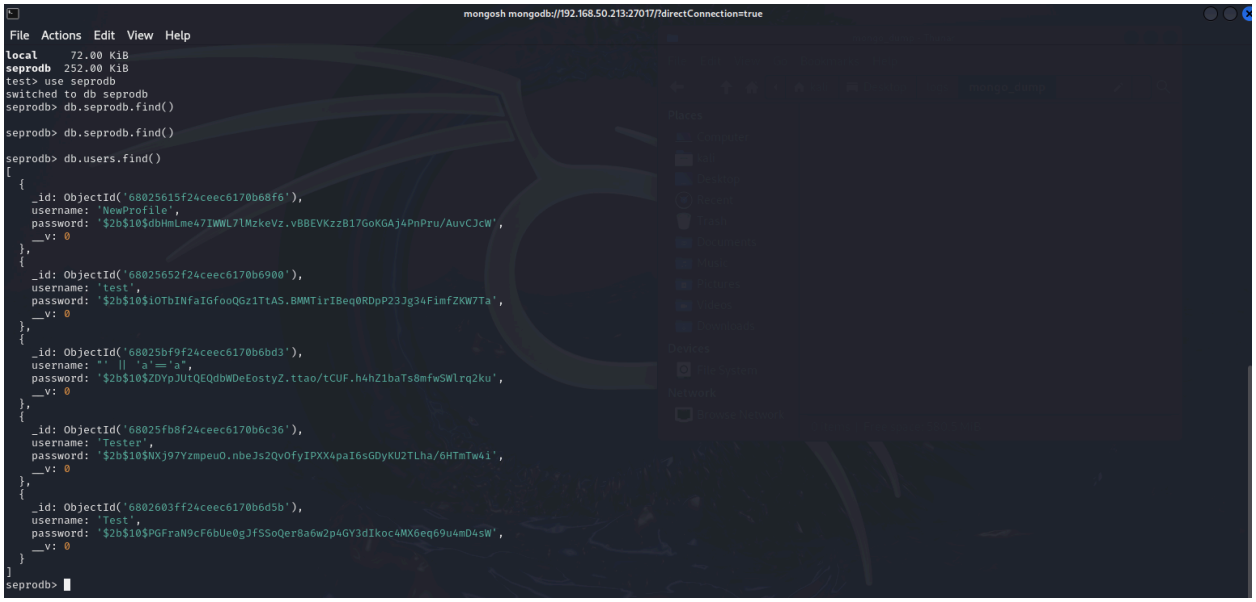

Figure 47. Access the sepro database with *“use seprodb”* and fetch all of the users in the database, using *“db.users.find()”*

Note that hashcat was used in an attempt to “break” some of those hashes, but all of the attempts have failed. Perhaps using a large enough wordlist would have worked eventually. That, however, was not attempted.

Figure 48. Change the password of user “Test” with the plaintext “ChangedPass123!”

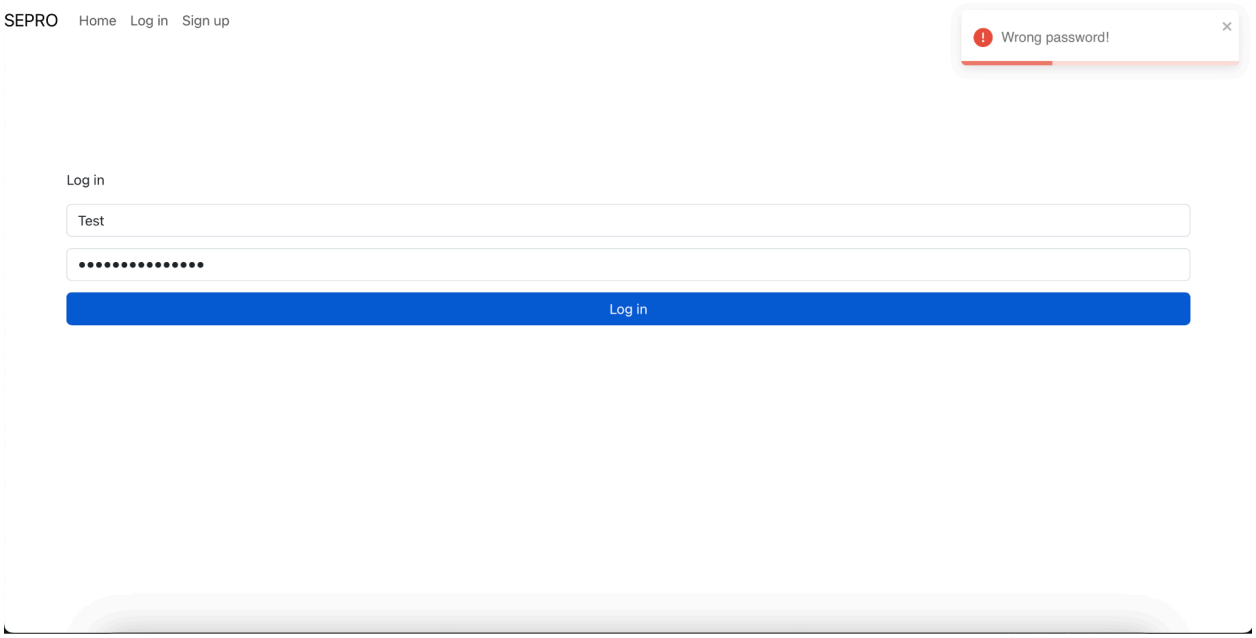

Figure 49. Attempt to log in with the newly changed password fails because the backend is comparing the database password value(which is assumed to be hashed, but in this case isn't) with the provided plaintext password's hash

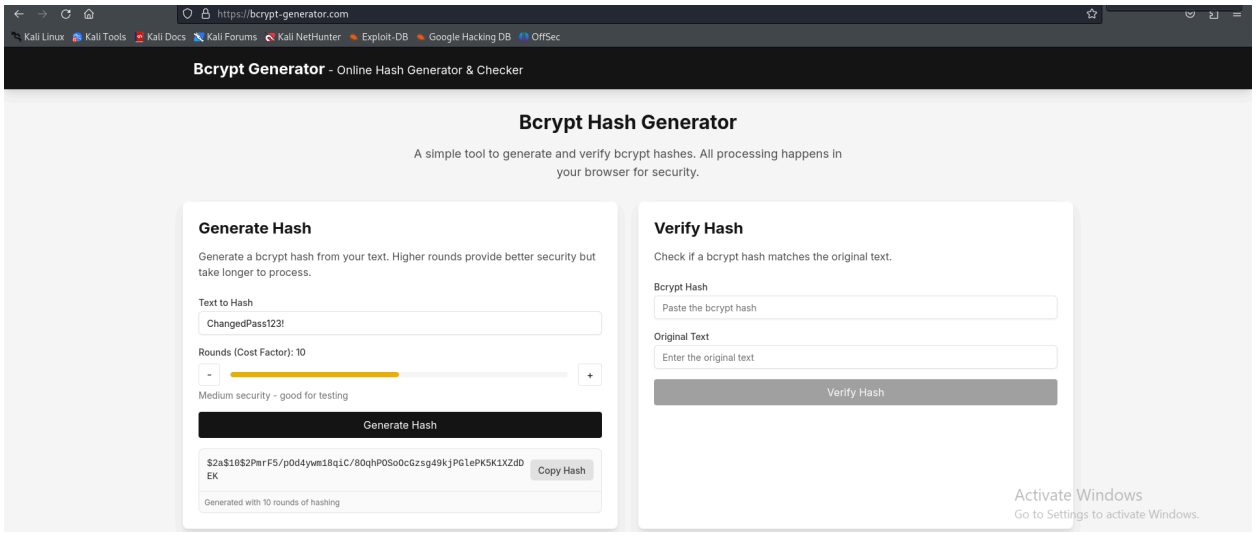

Figure 50. Inferring from all of the hashes in the database, they all have the same first six characters - "$2a$10", which tells us the hashing algorithm($2a)[2]. This allows us to use online bcrypt hash generators to hash our plaintext password with the same "parameters", so that the validations pass later on.

Next, update the password of user "Test" with the hashed password from the last step the same way the password was changed in step 4.

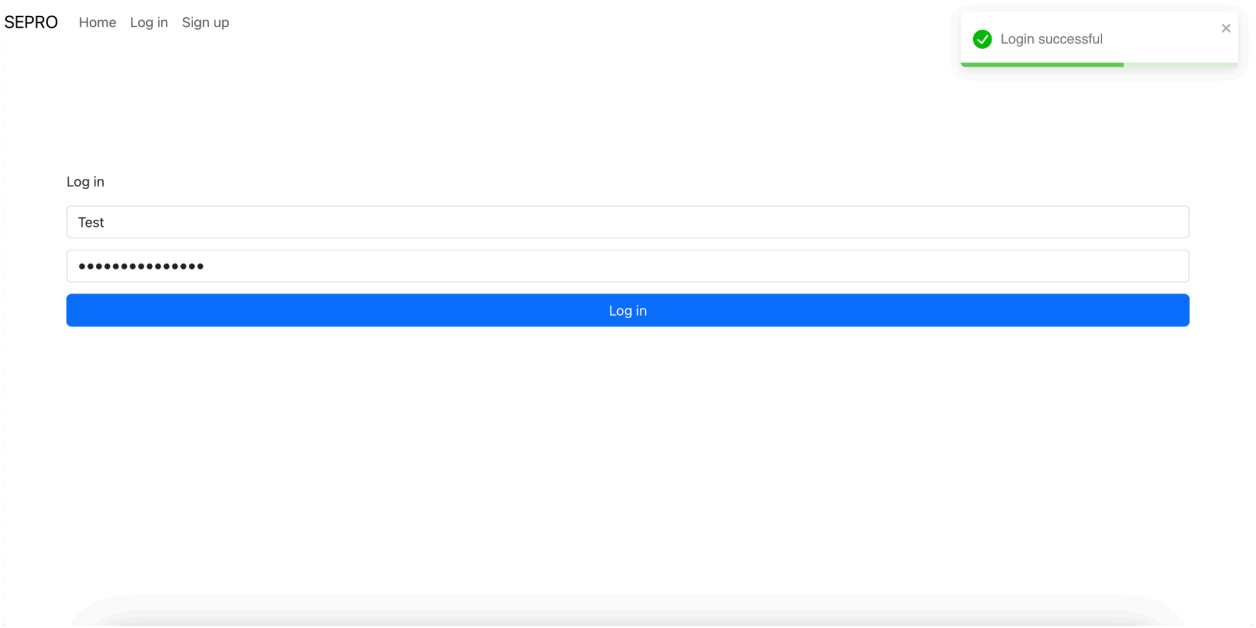

Figure 51. Login with “ChangedPass123!” for the “Test” user

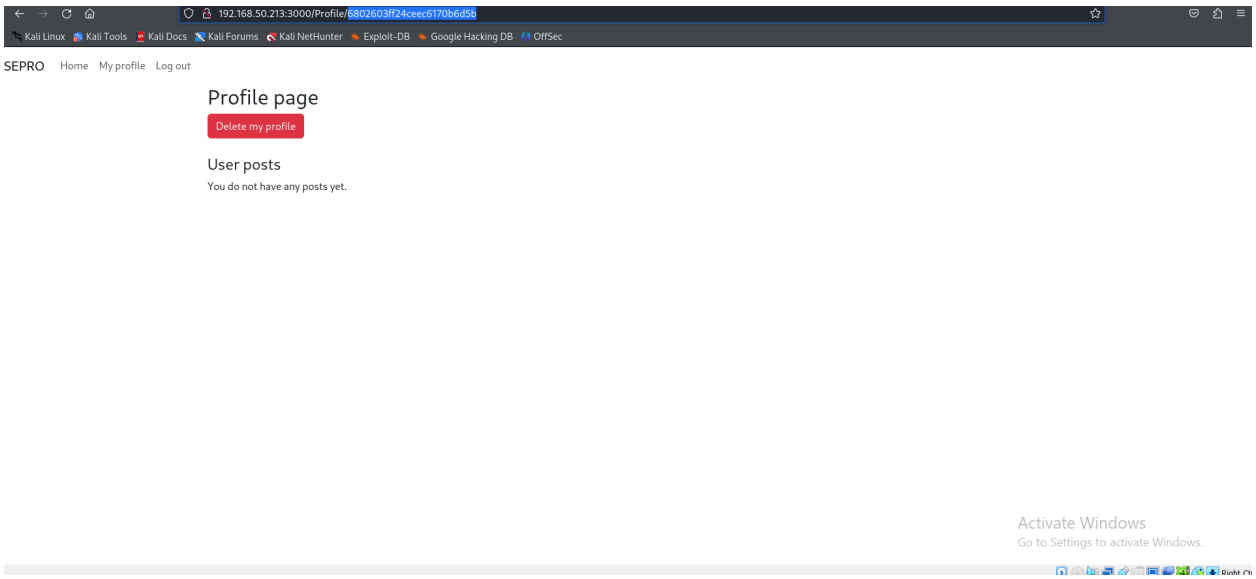

Figure 52. User “Test” has been breached, their account has been taken over, if you go back to step 3 and compare the id of user “Test” with the id parameter in the search bar, they match and the delete profile button is visible, indicating that we are indeed logged in as “Test”.

NoSQL injection

The only way NoSQL injection was observed during the penetration test was through postman.

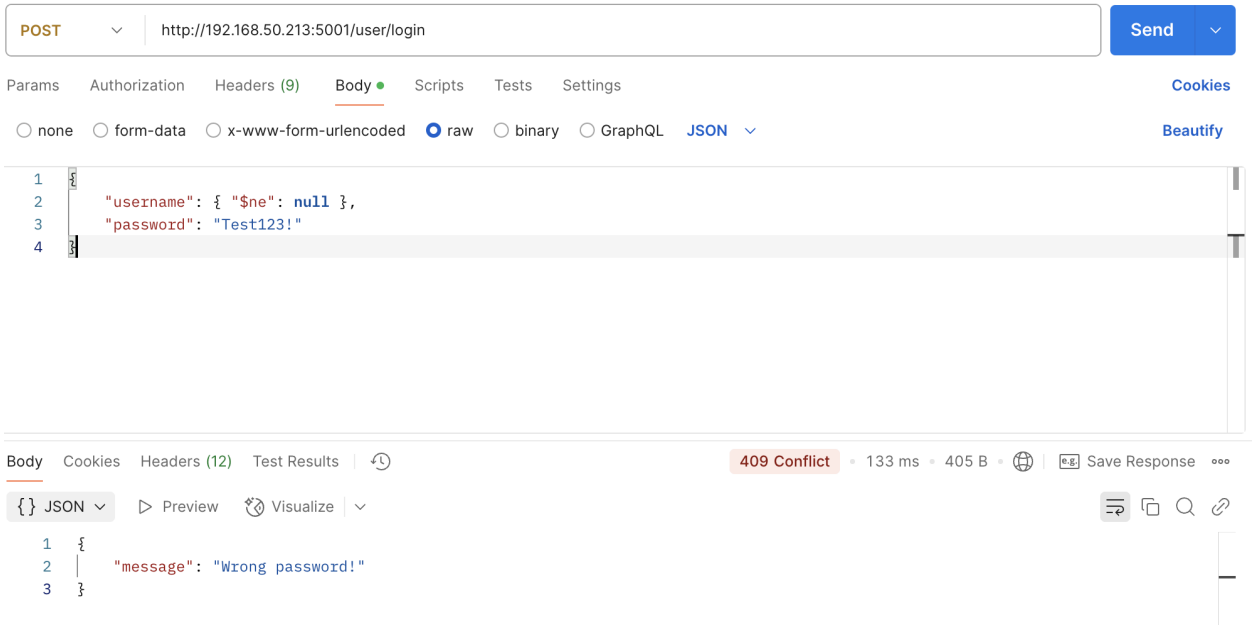

Figure 53. Make an HTTP request and insert a mongo operator in one of the fields.

In this case, inserting an operator that states “username is not equal to null”, i.e. all usernames in the user collection, results in “Wrong password”. That means that the provided password was tried against all of the users in the users collection! The reason we did not get a result different than “Wrong password” is that the server returns only a text response in case of a wrong password and the way the query is structured it is enough for one of the users to not match this password, so that the “Wrong password” response is sent.

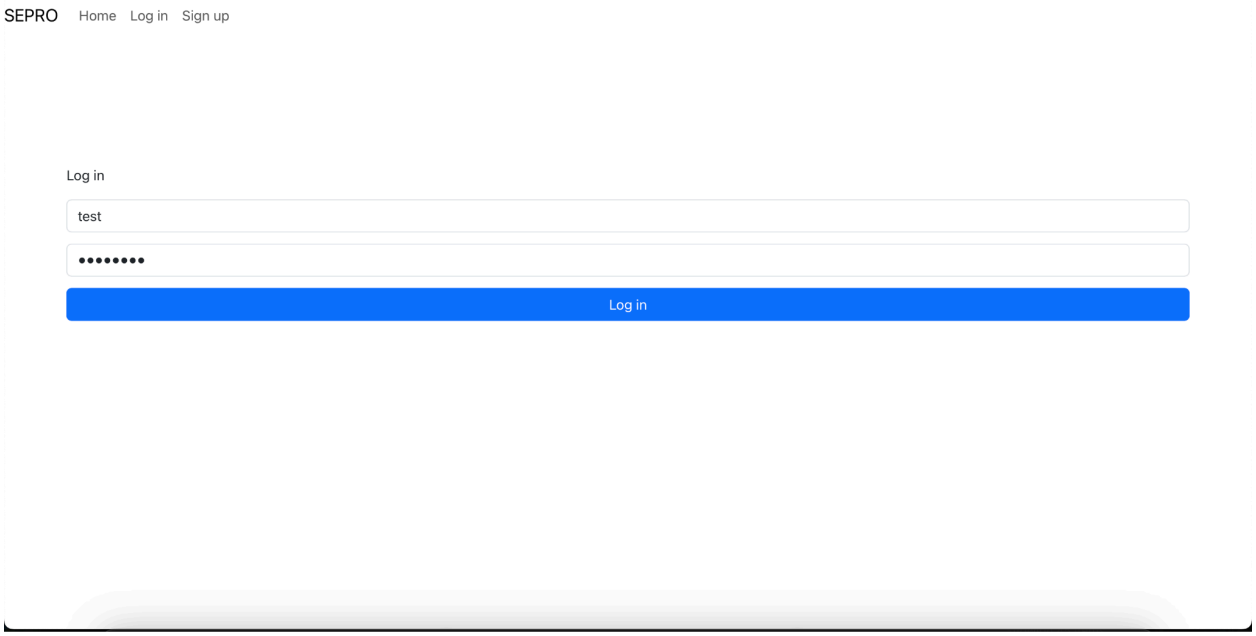

Figure 54. Enter username “test” and password “Test123”

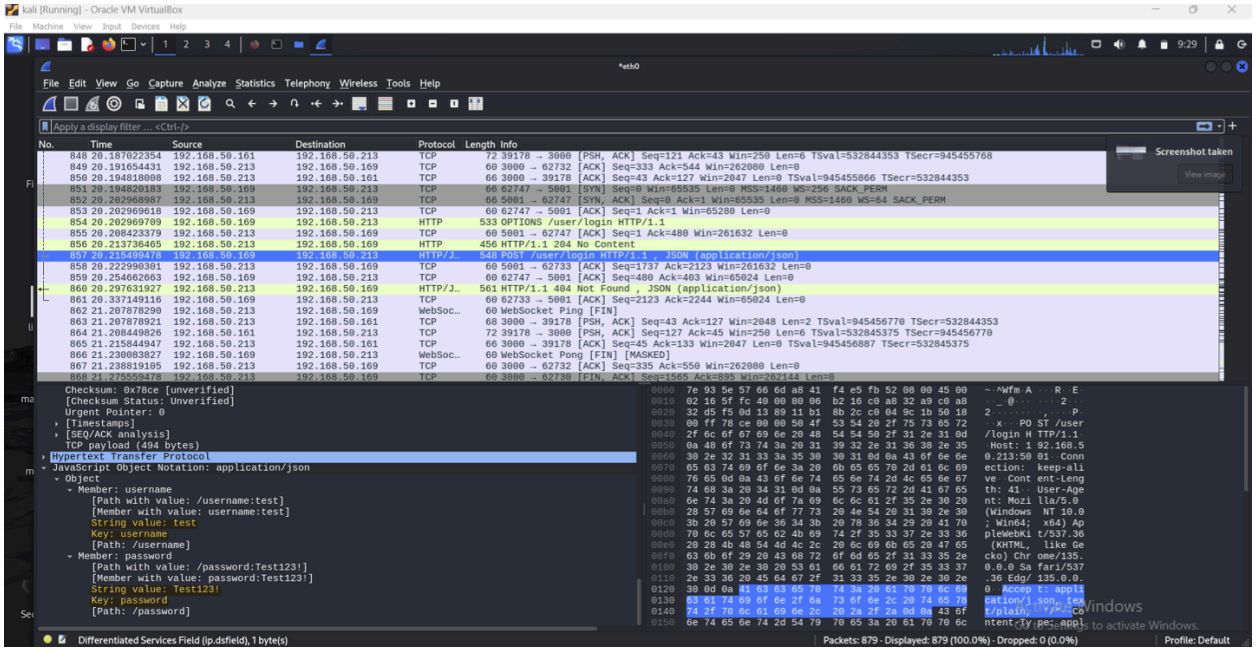

Figure 55. Utilizing Wireshark, start “monitoring” the network and look for an HTTP request with a source address corresponding to the address on which the project is running on. Inspect the data and check the JSON field, the username and plaintext password are there.

It is important to note that this is one way of performing MITM and stealing the credentials. Other attack vectors such as ARP spoofing are also possible.

Unsuccessful attempted exploits

Unsuccessful attempts were made at brute force attacks, cross-site scripting, session hijacking, and auth bypassing.

Brute-force attack

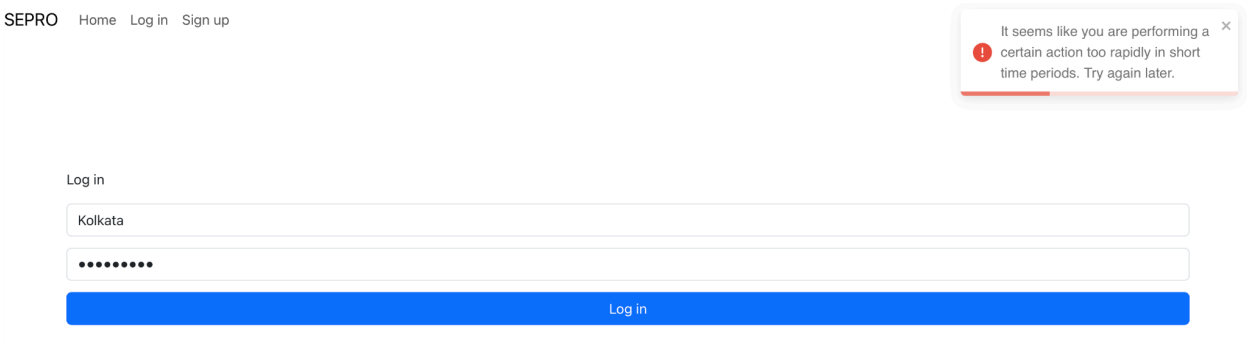

Figure 56. A failed brute-force attack, indicated by the rate limiter message.

XSS Attack

Figure 57. A failed XSS attack.

Broken auth

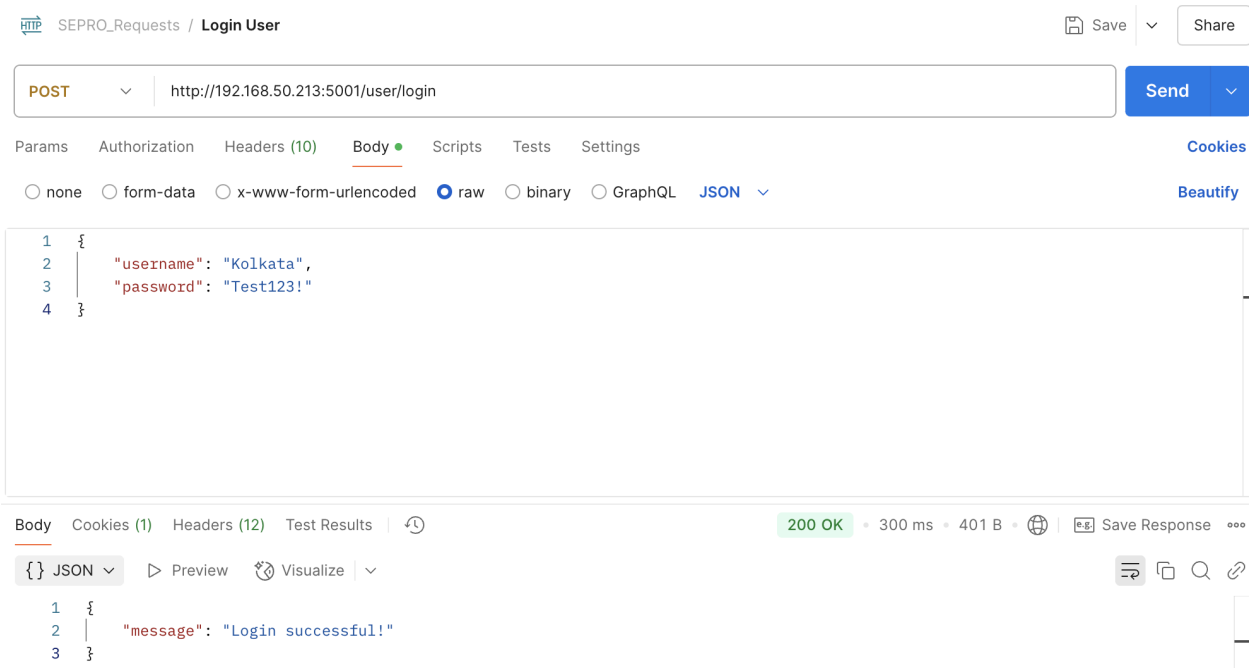

Figure 58. Login as “Kolkata” through postman.

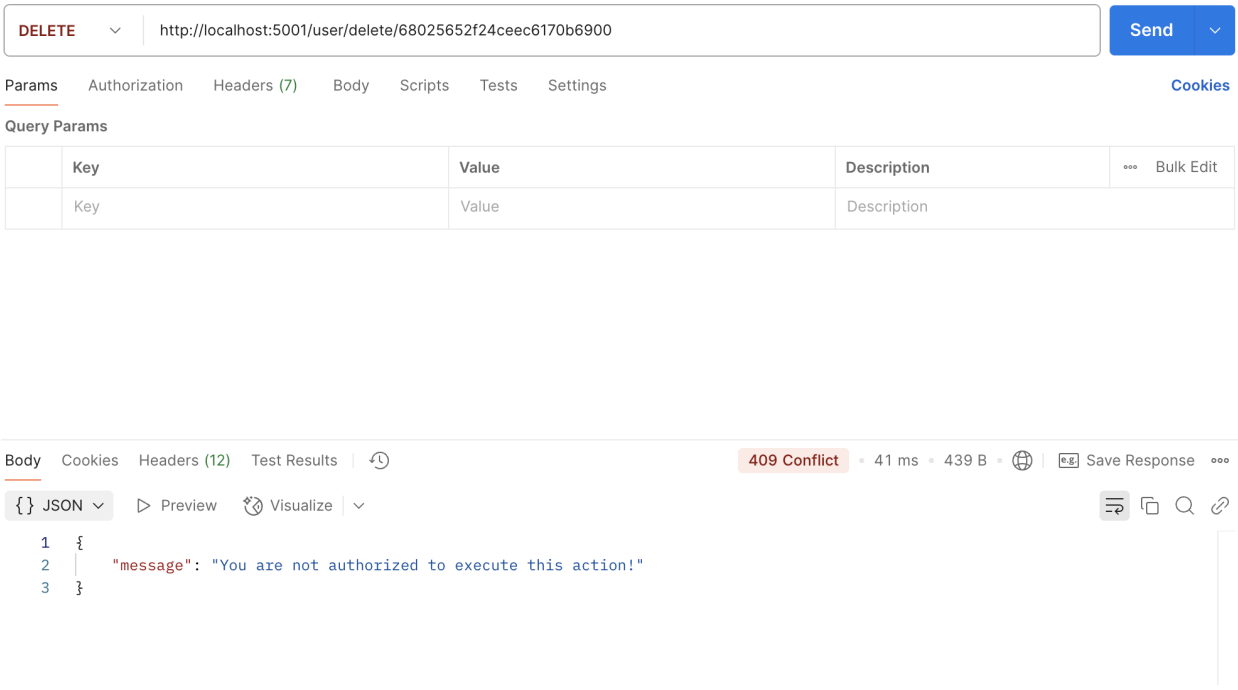

Figure 59. A failed broken auth attempt. We logged in as “Kolkata” and attempted to delete user “test” through Postman.

Remediations

Prevent CSRF attacks

The sameSite attribute has been added to the session cookie configuration. That means that the cookie is going to be available only in the context of the website it was sent to, preventing CSRF attacks. [8]

```
app.use(session({
    store: MongoSessionStore,
    cookie: {
        maxAge: 3600*60*24*7,
        path: '/',
        httpOnly: true,
        sameSite: true
    },
    saveUninitialized: false,
    secret: process.env.SESSION_SECRET
}))
```

Figure 60. SameSite has been added to the session cookie configuration, preventing CSRF attacks from occurring.

Prevent NoSQL Injection [9]

```
app.post('/user/login', AuthRateLimiter, async (req,res)  => {
    try{
        let sanitized_body = sanitize(req.body)
        let username = sanitized_body.username
        let password = sanitized_body.password
        if(typeof username == 'string' && typeof password == 'string')
        {
            if(!PASSWORD_REGEX.test(password))…
                }
            else…
                }
        }
        else
        {
            res.status(500).send({'message': 'Try using a different username or password'})
        }
```

Figure 61. Set up sanitization for the login endpoint, using *mongo-sanitize*. Check if all of the incoming request object body values are strings. If nosql injection payload has been detected, the data type is an object, thus the login attempt fails. The exact same logic is applied to the register endpoint.

Run npm audit

```
added 343 packages, removed 937 packages, changed 376 packages, and audited 1455 packages in 10s

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

# npm audit report

nth-check  <2.0.1
Severity: high
Inefficient Regular Expression Complexity in nth-check - https://github.com/advisories/GHSA-rp65-9cf3-cjxr
fix available via `npm audit fix --force`
Will install react-scripts@3.0.1, which is a breaking change
node_modules/svgo/node_modules/nth-check
  css-select  <=3.1.0
  Depends on vulnerable versions of nth-check
  node_modules/svgo/node_modules/css-select
    svgo  1.0.0 - 1.3.2
    Depends on vulnerable versions of css-select
    node_modules/svgo
      @svgr/plugin-svgo  <=5.5.0
      Depends on vulnerable versions of svgo
      node_modules/@svgr/plugin-svgo
        @svgr/webpack  4.0.0 - 5.5.0
        Depends on vulnerable versions of @svgr/plugin-svgo
        node_modules/@svgr/webpack
          react-scripts  >=2.1.4
          Depends on vulnerable versions of @svgr/webpack
          Depends on vulnerable versions of resolve-url-loader
          node_modules/react-scripts

postcss  <8.4.31
Severity: moderate
PostCSS line return parsing error - https://github.com/advisories/GHSA-7fh5-64p2-3v2j
fix available via `npm audit fix --force`
Will install react-scripts@3.0.1, which is a breaking change
node_modules/resolve-url-loader/node_modules/postcss
  resolve-url-loader  0.0.1-experiment-postcss || 3.0.0-alpha.1 - 4.0.0
  Depends on vulnerable versions of postcss
  node_modules/resolve-url-loader

8 vulnerabilities (2 moderate, 6 high)

To address all issues (including breaking changes), run:
  npm audit fix --force
(base) martin@MacBookAir seproproject %
```

Figure 62. The result after running “npm audit fix –force”. Dependencies which were known to be vulnerable have been updated.

Prevent unauthorized database access [5]

Docker compose environment variables for the mongo service have been set up, creating the administrator credentials upon creating the container for the first time. That way, unauthorized database access is restricted.

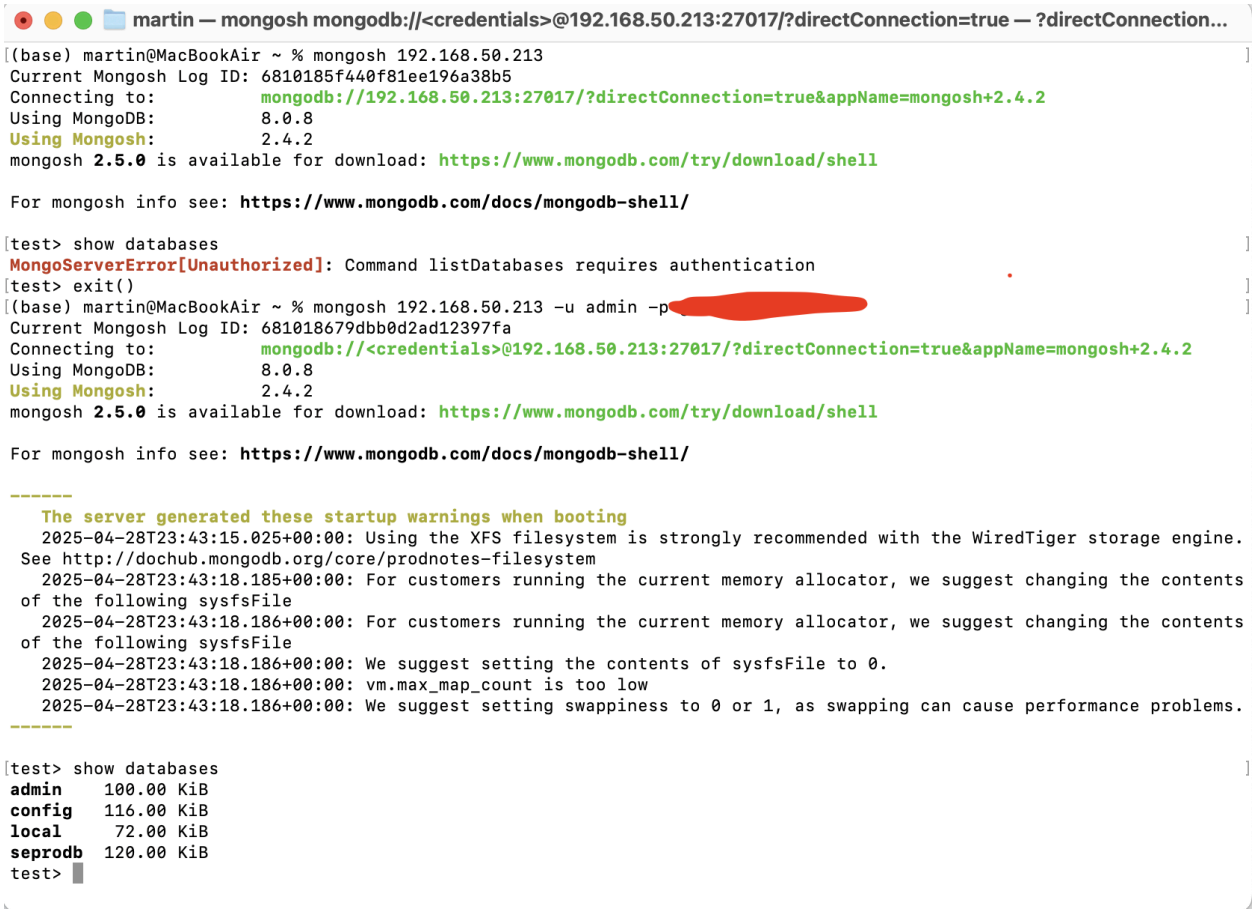

Figure 63. The database cannot be interacted with without authentication now

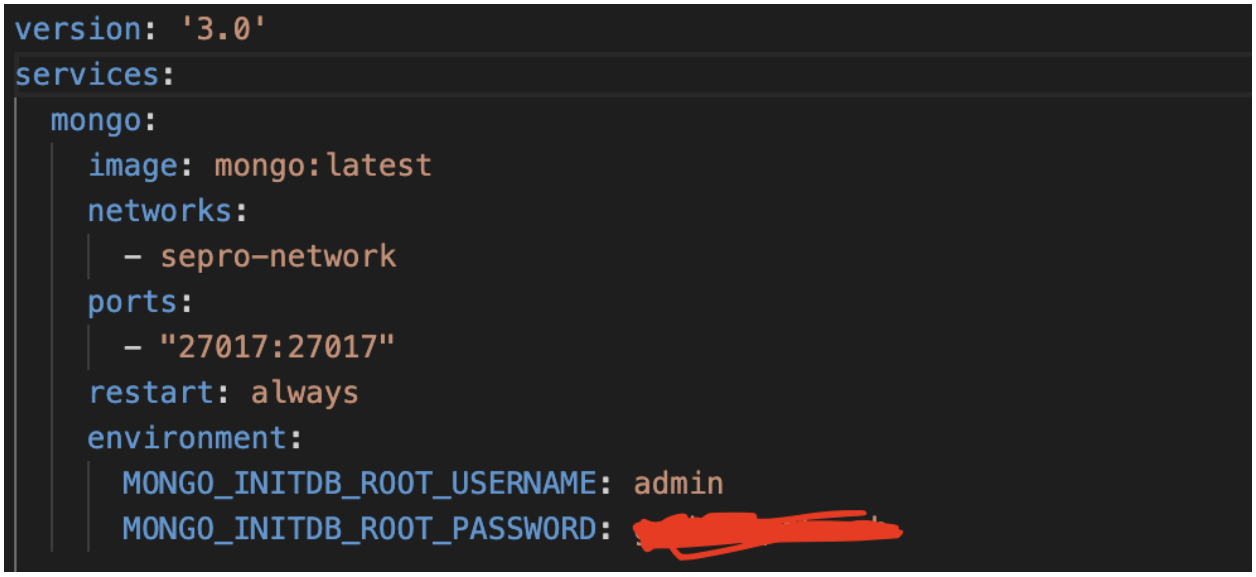

Figure 64. The docker compose mongo service authentication configuration

Suggestions for improvement

Cloud services could be utilized for improved security. For instance, deploying the application on a cloud service would ensure HTTPS is in place, preventing MITM attacks. Load balancers could be configured to handle higher traffic, lowering the risk of a DoS and DDoS attacks to succeed. In addition, the majority of the vulnerability assessment process could be automated by setting up a CI pipeline that will perform the same processes.

Conclusion

First, the application was assessed for publicly-known vulnerabilities. Those with available fixes were prioritized the most. Next, a penetration test was performed whose focus was to find OWASP TOP 10 vulnerabilities. The test was thorough enough to test for every single one of them. The exploited ones were:

- Injection [4]
- Cryptographic failures [4]
- Security misconfiguration [4]

Tests were performed for the rest of the OWASP TOP 10 vulnerabilities but none was found and exploited. All of the exploited vulnerabilities were later remediated.

References

[1] React. (n.d.). *Introducing JSX*. React. https://legacy.reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks

[2] npm. (2023, August 16.). *bcrypt*. https://www.npmjs.com/package/bcrypt#hash-info
[3] Trivy. (n.d.). *Filtering - Trivy*. https://trivy.dev/v0.47/docs/configuration/filtering/
[4] OWASP. (n.d.). *OWASP Top Ten*. https://owasp.org/www-project-top-ten/
[5] jbochniak. (2017, March 23). *How to enable authentication on MongoDB through docker?* Stack Overflow. https://stackoverflow.com/a/42973849
[6] Young Shun. (2022, October 5). *How to retrieve records which didn't have a particular property in MongoDB?* Stack Overflow. https://stackoverflow.com/a/73955334
[7] JohnnyHK. (2017, May 4). *How to delete multiple documents in MongoDB?* Stack Overflow. https://stackoverflow.com/a/43788902
[8] Sjoerd. (2021, October 21). *Do I still need CSRF protection when SameSite is set to Lax?* Security Stack Exchange. https://security.stackexchange.com/a/239842
[9] npm. (2020, March 2). *mongo-sanitize*. https://www.npmjs.com/package/mongo-sanitize