

WEB-APPLICATION SECURITY

Forum application with MERN-stack
COMP.SEC.300

Olli Sund (150162212)

SISÄLLYSLUETTELO

1.THE PROGRAMS FEATURES BRIEFLY	III
1.1 Login and sign up	iii
1.2 Dashboard and topics	iii
1.3 Topic page.....	iv
1.4 What wasn't implemented due to time constraints	v
2.SECURITY FEATURES OF THE APPLICATION.....	VI
2.1 Password security	vi
2.2 Authentication and refresh tokens	vi
2.3 User input sanitation	vii
2.4 Request throttling	vii
2.5 What wasn't implemented due to time constraints	viii
3.SECURITY TOOLS.....	IX
3.1 Snyc.....	ix
3.2 CodeQL	x
SOURCES.....	1

NO TABLE OF FIGURES ENTRIES FOUND.

BRIEF DESCRIPTION OF THE PROJECT AND LEARNING GOALS

I chose the programming project option as my exercise work. My goal was to learn more about web-application security. The project is a simple web forum application in which users can post topics and messages related to those topics. The application is a simple CRUD application, and more emphasis was put into secure programming practices and techniques than complex features. The application itself is implemented with the MERN-stack. The front end is a React app. The backend is an Express.js and Node application. The database is a MongoDB NoSQL document-oriented database.

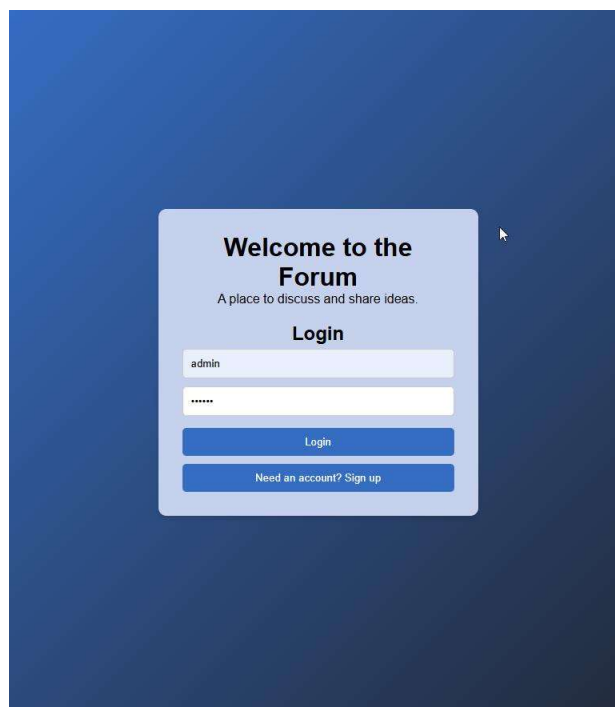
The goal was to focus on the OWASP top 10 vulnerabilities. Most of my time went to learning and working on authentication so I may not have had enough time to focus on all of the top 10 vulnerabilities fully. I do feel though that I learned quite a lot even if I didn't get to work on all the aspects of web application security as widely as I would have liked. So, all in all, I'd say this project served me as a student very well.

1. THE PROGRAMS FEATURES BRIEFLY

The following is a brief description of the functional features of the application project. The application is a very simple discussion forum web application. As mentioned in the previous section the technologies used were React for the frontend, Node and Express.js for the backend, and MongoDB for the database.

1.1 Login and sign up

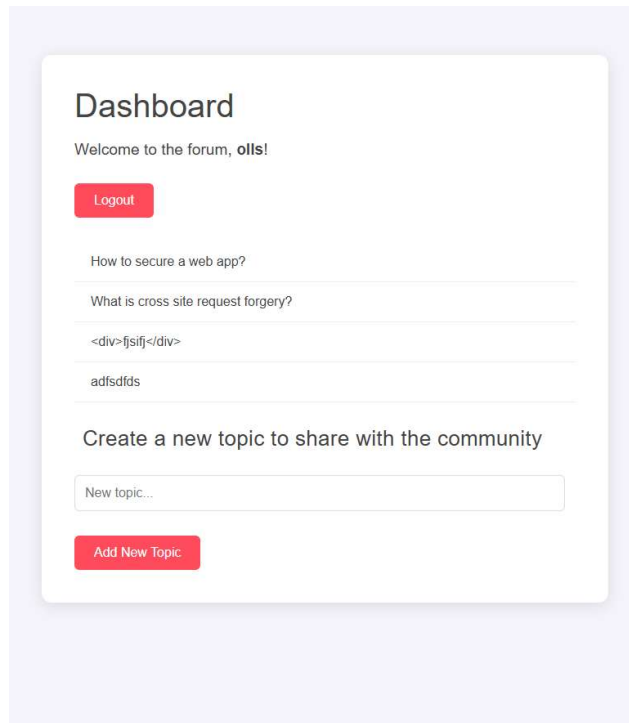
The landing page is a login and sign-up screen. The user can toggle between the login and sign-up forms. There are no requirements for passwords because I felt that it is such a basic feature that I could omit it completely. This also makes testing easier. Obviously in a production application I would have implemented validation for safe passwords and other form validations. Login credentials are **username and password**. Everyone is familiar with basic login and sign-up forms, so I don't think it is necessary to explain these further. The security features of user management are discussed in a later chapter.



1.2 Dashboard and topics

After successful login, the user is directed to a dashboard page. The dashboard page contains a list of discussion topics posted by users. A user can add a new topic from the bottom of the page using a text input and submit button. Users can drill down to view the

messages posted in the topics by clicking on a topic. Logout can be performed by clicking on the logout button shown in the dashboard. Admin users can also navigate to an administrator portal page. I did not have time to implement actual functionality for deleting or modifying users.



1.3 Topic page

Users can drill down to a specific topic from the dashboard by clicking on a topic of their choosing. Users can post messages to the selected topic from the text input and submit button much the same way as posting topics. I did also not have time to implement deleting or modifying messages.

1.4 What wasn't implemented due to time constraints

I would have liked to add more CRUD operations such as modifying and deleting messages. Also, admin functionalities such as proper user management weren't really added.

Error handling and especially informing users of error situations is minimal and I would have developed error handling to be more robust and user friendly.

2. SECURITY FEATURES OF THE APPLICATION

The following are descriptions of the security considerations and features implemented in this project. Also discussed are some of the security features I did not have time to implement but would have liked to spend some time learning.

2.1 Password security

The user passwords are saved in the “users” collection of the database. The passwords are hashed using bcrypt. Bcrypt is a password hashing function that incorporates a salt to protect against rainbow table attacks and uses a work factor to make brute-force attacks computationally expensive. It’s designed to be slow and adaptable, which makes it ideal for securely storing passwords [1]. In this application the passwords are hashed using both salt and pepper. Below you can see the hashed password stored in the database. You can also see the refresh token array which stores hashed ids for the tokens.

```
_id: ObjectId('680d21e9ccf10c1f56fd1199')
username: "olls"
firstname: "Olvi"
lastname: "Snud"
email: "Olvi.Snud@email.com"
password: "$2b$10$iRjFoN8Rd6FqJuAW90d5ugXwjBTfro2kG3u3/I5uXuv74SwWdC6"
type: "user"
createdAt: 2025-04-26T18:11:53.718+00:00
updatedAt: 2025-05-21T13:34:54.407+00:00
__v: 1866
refreshTokens: Array (5)
```

2.2 Authentication and refresh tokens

The application uses authentication tokens and refresh tokens. Both are JSON Web Tokens. The authentication tokens are used to authenticate the user when making api calls to the backend. They are short lived (15 minutes), which minimizes the potential harm caused by a leaked token. The backend uses authentication middleware to validate the auth tokens. There is also middleware to check valid tokens regarding role-based actions (admin vs. Regular user). The refresh tokens are longer lived and are used to provide a new authentication token after the old token expires. The refresh tokens have unique identifiers (jti). The id i.e. jti portion of the refresh token is hashed and stored in the users collection in the database. A user can have five refresh tokens simultaneously which means a user could have sessions with five different devices. The refresh tokens are validated against the jtis on every use and can only be used once. This rotation of jtis

prevents a leaked refresh token from being reused to get an authentication token. Also, only the id portion of the refresh token is saved in the database in a hashed form so the damage from a leaked token id is very minimal. The tokens are “http-only” so they cannot be accessed using Javascript in the browser. The authentication token is stored only in the React application memory so it also should be resistant to cross site request forgery.

```
// Set the access token in the axios instance headers for all requests
// This will be used to authenticate the user for all requests that require authentication
instance.interceptors.request.use(
  async (config) => {
    if (accessToken) {
      config.headers.Authorization = `Bearer ${accessToken}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
)
```

2.3 User input sanitation

User input is sanitized both in the front and backends to combat against XSS attacks. The frontend uses a library called “DOMPurify” the cleanup user inputs such as topics and messages. “DOMPurify sanitizes HTML and prevents XSS attacks. You can feed DOMPurify with string full of dirty HTML and it will return a string (unless configured otherwise) with clean HTML” [2].

The backend uses a library called “express-validator”. The library was used to implement middleware which chains validation rules and checks and escapes invalid messages. The backend also validates MongoDB ids used in queries.

```
const { body } = require('express-validator');

const messageValidator = [
  body('content')
    .trim()
    .escape()
    .notEmpty().withMessage('Content is required')
    .isLength({ min: 1, max: 500 }).withMessage('Content must be between 1 and 500 characters long'),
];

module.exports = messageValidator;
```

2.4 Request throttling

The backend uses middleware to limit the number of requests in a given timeframe. Login requests have stricter rules than other requests. The throttling is used to protect against denial-of-service attacks and brute-force attacks.


```
var app = express();

var limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 500, // max 500 requests per windowMs
});
// apply to all requests
app.use(limiter);
```

2.5 What wasn't implemented due to time constraints

I think the authentication token strategy used already has some protection againsts cross site request forgery attacks. I would still have liked to implement middleware specifically meant to protect against csrf using csrf tokens with the requests.

Also, like with the frontend, I would have liked to spend more time on proper, safe, and unified error handling strategy for the application as a whole.

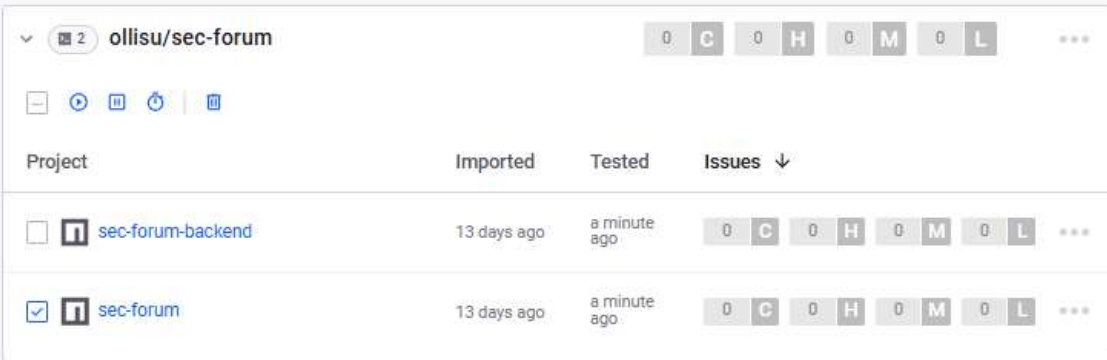
3. SECURITY TOOLS

The project uses GitHub actions to run security scans on the front- and backend. The checks are run every time a commit is pushed to the main branch.

3.1 Snypk

GitHub Actions workflow is used to automatically run security checks on both the frontend and backend using Snypk, a security tool that scans for vulnerabilities in code, dependencies, and configurations. Whenever code is pushed to or a pull request is made against the main branch, the workflow triggers two parallel jobs: one for the backend and one for the frontend. Each job checks out the code and runs three Snypk commands—test to identify known vulnerabilities, monitor to enable ongoing tracking of security issues, and code test to analyze the source code for insecure patterns. The workflow uses a secure token stored in GitHub Secrets to authenticate with Snypk’s API, ensuring that the scans are authorized and integrated into the CI/CD pipeline.

Currently Snypk doesn’t find any third-party vulnerabilities. Earlier there was one dependency in the bcrypt librarys dependency chain, but it was fixed by updating some of the packages to a newer version.



Project	Imported	Tested	Issues ↓
<input type="checkbox"/> sec-forum-backend	13 days ago	a minute ago	0 C 0 H 0 M 0 L ...
<input checked="" type="checkbox"/> sec-forum	13 days ago	a minute ago	0 C 0 H 0 M 0 L ...

There is currently one open security issue dealing with missing csrf middleware. As mentioned earlier this was one of the features not implemented due to time constraints. Also, it might in a way be false positive because the issue is based on the fact that the application does not register any csrf middleware, but it doesn’t explicitly mean that there is a problem or a found vulnerability.

```

9
10 Testing sec-forum-backend ...
11
12 X [Medium] Cross-Site Request Forgery (CSRF)
13   Path: app.js, line 16
14   Info: CSRF protection is disabled for your Express app. This allows the attackers to execute requests on a user's behalf.
15
16
17 ✓ Test completed
18
19 Organization:    ollisu
20 Test type:       Static code analysis
21 Project path:    sec-forum-backend
22
23 Summary:
24
25 1 Code issues found
26 1 [Medium]

```

3.2 CodeQL

GitHub Actions workflow is used to run CodeQL analysis on both the frontend and backend whenever code is pushed to the main branch. CodeQL is a static analysis tool developed by GitHub that scans code for security vulnerabilities and coding errors by querying the codebase like a database. The workflow defines two jobs—one for the backend and one for the frontend. In both jobs, the code is first checked out, then CodeQL is initialized with JavaScript as the target language and the appropriate source directory specified (sec-forum-backend or sec-forum-frontend). Finally, the analysis is performed, and the results are uploaded to GitHub’s security tab, in which developers can identify and fix potential security issues early in the development lifecycle. There is currently one open security issue dealing with missing csrf middleware. **This is the same issue that was found in the Snyk checks.**

Code scanning

✓ All tools are working as expected

🔍 is:open branch:main

☐ ⚠️ 1 Open ✓ 26 Closed

☐ ⚠️ Missing CSRF middleware High

#22 opened 2 weeks ago • Detected by CodeQL in app.js :51

Other issues that were found during the project were things like missing request rate limiting and unsanitized user input. Rate limiting issues were fixed with middleware that limits request rates. User input validation was added to mitigate untrusted inputs.

<input type="checkbox"/>	<input checked="" type="checkbox"/>	Missing rate limiting	High
#15 closed as fixed 2 weeks ago • Detected by CodeQL in routes/api.js :8 2 weeks ago			
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Missing rate limiting	High
#14 closed as fixed 2 weeks ago • Detected by CodeQL in routes/users.js :7 2 weeks ago			
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Database query built from user-controlled sources	High
#26 closed as fixed 2 weeks ago • Detected by CodeQL in routes/api.js :97 2 weeks ago			
<input type="checkbox"/>	<input checked="" type="checkbox"/>	Database query built from user-controlled sources	High
#25 closed as fixed 2 weeks ago • Detected by CodeQL in routes/api.js :91 2 weeks ago			

AI USE IN THE PROJECT

ChatGPT and copilot were used to debug some coding issues. Also, the ai tools were used to get brief and concise of some key concepts and technologies for improve understanding.

SOURCES

- [1] <https://en.wikipedia.org/wiki/Bcrypt>
- [2] <https://github.com/cure53/DOMPurify>

