Tampere University

Tech Blog

Secure Programming Report

Eero Tarri

Table of contents

1. Description

Tech Blog is a web application where users can share their knowledge and interests about current events in technology. Other users can comment and spark up a conversation of any post they desire with intuitive comment section.

Registration is possible on the top right corner of the page when user is not already signed in. Next to registration the user can sign in if already registered. By signing in these buttons change to single “log out” –button. Registration is only needed if the user wants to post their own blogpost or comment on a post.

The front page of the application displays all the posts in descending order based on the time and date. Any post can be read on the front page of the application. By clicking on a post, the application navigates to the designated page for the clicked post where users can interact with others by writing their thoughts on the comment section.

2. Structure of the program

The application was created with npx create-next-app. This initializes the project on the machines file system with some options. One of the most important of those options are the choice of language used and for this project TypeScript was chosen. Rest were the options to use ESLint for code styling tests, Tailwind CSS for UI, which I did not opt for, and App Router for more modern routing.

The command mentioned above creates a NextJS application skeleton. NextJS is a popular React framework that simplifies the programming of modern web applications and provides some extra functionality such as server-side rendering (SSR). Along with NextJS comes NextAuth, which is an easy way to add authentication to NextJS applications. NextAuth supports a number of OAuth2.0 providers.

MaterialUI was chosen to implement UI on this project. MaterialUI is a component library that is based on Google’s Material Design principles. It gives a set of pre-designed React components that are fully customizable. MaterialUI provides a great way to develop professional looking websites with a lot less code than it would take to implement the same with just HTML, JS and CSS.

NextJS is a great for web application security because it has hybrid rendering. This means that the server can render some of the content before it is sent to frontend. NextJS does this by incorporating two types of components, client components, and server components. Client components work in a same way that normal React components work. They are sent to frontend as JavaScript files and rendered there into HTML. Server components however are rendered into HTML already on the server side, which gives one important advantage. Server never needs to expose tokens and other important secrets. Server can for example fetch data from API, render the content and then send it to user client. All the valuable information was held in server the whole time and client doesn’t even know it exists. Client and server can communicate using route handlers which are essentially API-endpoints on the server.

NextJS routing is file structure -based, so for each nested folder there is a similarly named path on the website. By defining a page.tsx, the developer can create visible page and by defining a route.ts, the developer can create an API endpoint that allows user to read and write posts to the server.

All the data is saved on the server in the project folder. This makes the project a sandbox and very easy to spin up. Only optional configuration that can be made is to include own Google Client ID and Secret in an .env file. This enables signing in with Google's OAuth, but it is just an extra functionality and not necessary for the sake of this project.

3. Secure programming solutions

In this chapter I will follow checklists like OWASP 10 and React Best Security practices to list how the application has been secured from external attacks.

3.1 Authentication

Authentication is secured with widely used salt-and-hash technique. During registration, the application uses authentication utility functions to first generate a pseudorandom 16-byte long salt string encoded in base64. Then it adds the salt to the password user provided and hashes it with SHA256 algorithm. Both of these actions are implemented in the JavaScript built-in library called Crypto.

In addition to those, Crypto provides a 36-character long v4 Universally Unique Identifier (UUID) that is used to uniquely identify users. All of this information is stored in a JSON-file after encryption. This way plain-text password will never be stored as such.

Similarly sign in will ask for username and password. Username will be matched with a corresponding user object in the JSON and password provided by user and salt stored in the object are used to generate a hash that is then matched with the hash stored in the object. If the hashes are not a match, authentication will fail.

3.2 Whitelisting and sanitization

The application allows users to write blog posts with HTML tags. To implement this, the blog text is dangerously set in to a Materia UI box. By default, this is not a good practise, as that would allow users to save malicious code on the backend that any client that comes to the website would run that code.

To prevent this kind of attack, whitelisting is used. Firstly, text boldness is chosen at the block level. Users can only choose between H1-H6 and P tags to categorize the text into paragraphs. This way only the header and text tags are whitelisted. Inside those blocks any text can be written, even HTML. To prevent any malicious code written, the text is always sanitized before sending to backend with a library called *sanitizeHtml.* SanitizeHtml takes in the text to sanitize and options as parameters. The options defines the allowed tags that in this application are:

```
["b", "i", "em", "strong", "ul", "ol", "li", "br"]
```

Only emphazising, lisiting and linebreaks are allowed at the moment. No tags are allowed inside the tags but that could be changed if for example style attribute wanted to be whitelisted.

3.3 File traversal

When posting a blog post users can upload a single image that represents the subject at hand. The image will be shown under the title in the post. The image is saved on the server with the same filename as the id of the post so post with id 715991bc-02dc-4ae3-920b-9ac869071391 would have a corresponding image with filename 715991bc-02dc-4ae3-920b-9ac869071391.png in the images folder. User has no say in the name of the file as it is automatically generated by the application.

File traversal should not be a problem because of this, but as an extra layer of protection the pathname is sanitized. If the pathname was set by user to point to a path on the server to extract critical information the application would shorten it to the basepath. This way the file traversal is constrained to the images folder only and all those files are public anyways.

3.4 Server side rendering

The application is built with NextJS, which is a hybrid rendering framework. Server-side rendering means that some of the content can be rendered from React components to HTML already on the server and sent as HTML, as normal React can only send the bundled JavaScript on the client that would then render the content.

The security benefit of server-side rendering is that secrets never need to be exposed to the client. The server can do any fetching that requires tokens on the server and only then sending the received content as parsed HTML to the client. This way the client never even knows where the content came from and what kind of secrets it required.

3.5 Character escaping

The application uses React components to render content. React handles character escaping by default to prevent any unwanted behaviour set by the users. This default can be bypassed by using *dangerouslySetInnerHtml* property which should only be used in cases where necessary. Only the blog posts itself uses this property to enable usage of HTML tags in the posts but this is mitigated with sanitization and whitelisting as described above. Any other display box uses React default behaviour. For example, comment section handles character escaping properly and will not run any code even if users comment text such as ***<script>alert('1')</script>***. It will rather just display that as a plain text.

4. Security testing

The security testing will be manual. Tests will be conducted by following user stories that could pose security attacks. From these tests the way to replicate, expected outcome and actual outcome will be documented.

Test ID	User story	Expected outcome	Actual outcome
1	I want to ensure that all user inputs are properly sanitized and escaped to prevent Cross-Site Scripting (XSS) attacks.	All user inputs are escaped and rendered as plain text, not as code. No script tags or other potentially harmful inputs are executed when rendered.	All inputs that are not known to use dangerous display all html-tags as plain text.
2	I want to verify that the application does not use dangerouslySetInnerHTML with unsanitized user input.	The application either does not use dangerouslySetInnerHTML at all, or if it does, the input is properly sanitized before being used.	The application does use dangerous, but it only accepts allowed html-tags.
3	I want to ensure that the application does not expose sensitive information in its error messages.	All error messages are generic and do not reveal any information about the application's internal workings or data.	The application will not display sensitive data as any errors on the server-side will show in server logs and errors in client will follow the bundled files that

			does not contain sensitive information
4	I want to verify that the application uses secure HTTP headers to prevent Clickjacking attacks.	The application sends the X-Frame-Options: SAMEORIGIN or Content-Security-Policy: frame-ancestors 'self' HTTP headers in its responses.	The application does not use those headers but it does use Sec-Fetch headers.

5. Future work

Saving images to server and rendering them on the post was implemented during the project. It proved to be very laborious to blend images with the text data so a single image per post approach was taken. The original idea was to allow post content field to somehow hold images in it similarly to HTML *<img>* tags but persisting and sourcing those images would be very hard to implement. This would take away from the focus of security, so it was left to future development.

Another idea that time did not allow to implement was to enable adding files such as JS files and the application would render it like Markdown would. This would pose a risk of XSS that would need to be handled separately but it would also add a nice visual for potential snippets shared on blog posts.

6. Known security issues

Currently the sanitization is done at the time of posting the blog post the only content displayed is that sanitized text. The dangerous setting of HTML however is done later in the component that displays the content. Between these two actions there is room to set the content for example by changing the text manually in the server. That is a slight vulnerability as the application could expand to accept more sources of content that would not be sanitized in the current form. The content should be sanitized right before the dangerous setting. That would require more implementation that there is no time for, and it is not currently vulnerable. For these reasons this is only documented and not implemented.

7. Suggestions for improvement

One clear improvement that can not be in the scope of this project is the use of external database. As mentioned before, during the course this project will be as sandbox as possible to ease spin up and testing. After the project is over, there is a high chance that the data persistence of this application is moved to a database. To ease the use and increase the security, Prisma should be used in between. Prisma is an Object Relational Mapping library that simplifies creating schemas for the objects, such as posts and comments.

Similarly to the application specific data, I have plans to separate user information to another service. This service would handle user data in a secure manner by accepting encoded user credentials and retrieving token for the application to use. This service would authorize the user to certain actions, and it could handle authentication providing across other systems too.

8. Use of AI

In this project I used artificial intelligence very conservatively as the environment I was working on was quite high level which makes artificial intelligence make a lot of mistakes. Artificial intelligence does usually not know how to handle a lot of framework code because it would require so much context that the interaction does not have without prompting with tons of code which is not feasible.

ChatGPT was mostly used to gain ideas. ChatGPT is not very good in generating working code, but it does know how to solve problems. Therefore, in this project I used ChatGPT to generate ideas on what features could be implemented that are relevant to blogging and what kinds of security problems they could potentially have.

GitHub Copilot was a lot more present in this project. I have Copilot enabled at all times as it enhances productivity a lot. Copilot is a lot better than ChatGPT in generating code, but even it was mostly on the backburner on this project. Copilot handles near perfectly cases, where I'm halfway through writing a statement and already know what it is going to include. In these cases, it acts as a more advanced IntelliSense in a way. In cases where I'm just beginning to write a state, the Copilot auto generation usually fumbles and suggests something irrelevant.

Because of these notions, I would say that nearly all of the code in the project is my own as NextJS is a new framework for me and I could not trust almost any code generated by the AIs. That said, AI was still used throughout the project to finish my sentences and expedite the process in the ways it can without interference to my learning.

9. Sources

[1] OWASP TOP 10: https://owasp.org/www-project-top-ten/
[2] 10 React security best practices: https://snyk.io/blog/10-react-security-best-practices/
[3] File Upload Vulnerabilities: https://portswigger.net/web-security/file-upload