Assignment 3

Topicus Team 1

Umair Aamir Mirza (s2928558)

Martin Demirev (s2965046)

Condu Alexandru-Stefan (s2769549)

Narendra Setty (s2944200)

Alexandru Lungu(s3006301)

Teodor Pintilie (s2920344)

Introduction

This report serves to give a detailed insight into how the development team was able to set up the security modules for the project, and an insight into how the functionality of the software was tested effectively to prevent unforeseen usage circumstances. It also goes into detail regarding the design choices that were made to ensure that the user base for the project was capable of performing the functions that they required.

Security Analysis

Independent Security Modules:

One of the decisions taken during the planning of the security implementation was to have two independent modules for security; one for front-end, and one for back-end. Another decision taken collectively was to use any available, credible security packages that would assist in our efforts to minimize the security risks of the application. Some packages that we looked into were:

- **DOMpurify:** sanitizing HTML input to prevent any XSS attacks.
- **de.mkammerer Argon2:** more secure and robust algorithm to protect the user credentials on the database against brute force attacks.
- **org.bouncycastle bcprov-jdk15on:** for the use of cryptographic algorithms for data protection and handling.
- **org.springframework.security spring-security-crypto:** for the use of encryption and password management.
- **com.autho java-jwt AND com.okta.jwt okta-jwt-verifier:** to implement JWT tokens in the application, which will be discussed below.

As stated, we have made some initial research into such packages, and will await further instructions whether all of them can be used. The JWT tokens were already allowed for usage, and have been implemented in the application, but the other packages were researched so that their methods can be included once clearance has been provided.

General Structure of Application Security:

As stated, the independent security modules on the front-end and on the back-end allow for focus on more specific elements. The contents of both modules are described in the list below:

- **Front-End Module:**
  - Input Sanitization with Libraries, Methods:
    - SQL Injection.
    - XSS Protection.
  - Authentication:
    - JWT Token in each Request.

- **Back-End Module:**
  - Input Sanitization with Libraries, Methods, Prepared Statements:
    - SQL Injection.
    - Stored XSS Protection.
  - Password Protection:
    - Argon2 for Password Hashing.
  - Authentication Management:
    - JWT Token Provision and Validation.

JWT Token-based Authentication and Access Management:

Beginning with the uppermost layer of the security implementation, authentication and access management is conducted through the use of JWT tokens. There were several reasons for using JWT tokens, predominantly being that they are the modern mechanism of authenticating users and roles in an application. Additionally, JWT tokens are created with a special seed that only the server knows. Thus, in order to deduce if a token originates from the server, the appropriate seed must be used for decoding the token. If the token cannot be decoded by the server, then the user is using a falsely supplied JWT token, and should not be authenticated. Furthermore, JWT tokens enable secure transportation of parameters and other information in the token body itself. In our case, we will transport the ID of the user, along with their role level in

the token which will be later decoded to deduce which user is requesting which data from the server, and whether they have the sufficient permissions to carry out this request or not. JWT tokens will be sent with every user request, and the server will process them accordingly. Currently, with regards to the management of the tokens, we are investigating the possibility of putting a filter class using the @Provider annotation in Jersey on the path "/api", which will validate the token in one place, instead of having the validation at all the resources separately.

The functionalities of the JWT token are conducted through the **CookieService** and **TokenService** classes in the **Service** package.

Front-end SQL Injection and XSS Protection:

To prevent SQL injection and XSS attacks, the **fe-security** package includes the **inputSanitizer.js** file. The code, for now, includes two arrays, namely "sqlRejectWords" and "xssRejectWords" have been created which contain commonly used SQL terms (such as SELECT, --) and XSS terms (such as <, src) respectively. These arrays serve as a blacklist for potentially dangerous terms. However, it is important to note that blacklisting is quite ineffective due to the fact that different encoding schemes can surpass the blacklisting itself. Hence, the function **inputSanitizer()** also includes conversion to UTF-8, along with other means that are being investigated. For now, the security requires more advanced tuning, and if permitted to use DOMPurify, it will enable more secure implementation through trusted libraries. We are also investigating appropriate ways to prevent reflected XSS attacks from occurring to modifications to the URL.

Front-end Input Validation and Sanitization:

Within the relevant functions for both signing up and logging in, the inputSanitizer() function is called to validate the username, email and password. If any of these inputs fail the sanitization check by returning false, an alert is displayed on the screen notifying the user of the invalid input. In the signing-up process, the code provides an additional check to ensure that the email is in the correct format. Finally, a basic password confirmation check is also performed to check whether the input of the entered password matches the confirmed password.

Back-End Security Against SQLi:

SQL injection is a serious threat to a lot of applications, especially ones such as ours which rely heavily on a database in order to function properly. After a meeting in which we discussed how we should protect our database, we have decided on using the following methods:

- Prepared Statements: this ensures that user input is not directly embedded into SQL statements, but rather it is supplied during runtime and replaces the placeholders that were previously set in the queries. This ensures a separation between data obtained via user input(which can be maliciously altered), from SQL logic.
- Input Validation and Sanitization: We plan on enforcing strict validation rules to restrict the data type and format of input data, preventing the execution of potentially malicious SQL code. All of the data fields received from user input are checked, first against data type validators, which ensure both type safety and user error management, and second against several regular expression matchers, which ensure that the format of the data corresponds to the standards requested by our back-end definition.
    - Format checking and request validation are done on the request object themselves on a basic level. However, we aim to create two classes, **InputValidator** and **InputSanitizer** which will be able to handle any obscure, potentially harmful data from entering the database and creating issues with the application.
- Input Filtering: In addition to validation and sanitization, we also check the input against a list of predefined blacklisted values, such as common SQL keywords. Black-listing is a relatively low-level mechanism for preventing SQLi, but the inclusion of it only serves to make our security implementation stronger.
    - As stated earlier, if we are allowed to use existing security packages for input sanitization and validation as well, then it would ensure that the security of the application is as robust as possible.
- Special Character Escaping: To prevent SQL injection attacks, we implemented techniques to escape special characters. We used escaping functions provided by our programming language to neutralize the special ability of these characters in SQL queries.

Password Hashing:

For storing passwords inside the database we are using Argon2 which is considered a state-of-the-art, memory-hard password-hashing function that provides robust protection against various security threats. In order to implement this we created a few components such as a HashedPasswordContainer that has two fields for storing a hashed password and its salt which we use to add them to the database. Therefore, instead of storing plain text, we store a hash and a salt which later can be verified with the user input to check if the password matches the hash that we stored. All of these functionalities are provided in the PasswordService class where we implemented the methods for hashing, verifying and creating the container for the hash and salt.

Software Testing

API/Web Service Testing:

We performed thorough Web Service Testing on our API resources using Postman. This tool allowed us to send HTTP requests to our web servers and receive adequate responses, enabling us to verify the functionality and reliability of our API.

In order to have a centralized API testing environment, we have created a shared Postman collection, which contains all the requests necessary for testing the web service resources. This has multiple benefits. The obvious one is the ability to add requests which can be seen by every member of the team. Another very important benefit is the ability to give examples of API usage. Since we've split our team into front-end and back-end development, Postman has enabled us to communicate more freely in regard to web resource access/usage.

Since we have multiple resources, we created folders for each of them and began testing. Each and every individual resource has been tested on the following use cases:

- retrieval: using GET requests with adequate JSON body responses
- creation: using POST requests with JSON body and adequate headers
- update: using PUT requests with JSON body and adequate headers
- deletion: using DELETE requests

During the testing process, we encountered several bugs which wouldn't be very easy to detect without actually verifying the respective HTTP requests. Postman has definitely made our job a lot easier, and by using it, we have managed to create, thoroughly test and complete all the API resources necessary for our project, such that they are fully functional and ready to be deployed.

This level of API testing enabled us to also change certain structural details of our application, some of which are listed below:

- User-Account Interaction: changed so that it includes a more efficient way of accessing the account, a separate path for performing the log-in, and also the introduction of JWT Tokens.
- RegistrationViewContainers: the use of containers to send large amounts of data at the same time enabled the presentation of registration lists for the school administrator, and also cards for the parents on the front-end.
- Creation of Request Objects: many of these can be seen in the **Payload** package of the project, where request objects are sent and response objects are provided to reveal only the necessary information to clients.

System Unit Testing (Back-end):

For the course of the project thus far, unit testing was not investigated extensively, though an implementation plan was devised. Research has been conducted into how code can be written to test the functionalities of the back-end and interaction with the database. The planning can be summarized in the list below:

- Creation of independent JUnit Testing classes per Resource:
    - CRUD Testing:
        - With the same record that is created, using the capabilities of the resource to manipulate the data.
    - Invalid Request Parameter Testing:
        - Testing invalid parameters and observing the reaction of the system.
    - Invalid Access Level for Request:
        - Testing requests when access level is not appropriate.

The reason that these tests were not implemented thus far was because of the Postman testing package that was created and used, enabling the development team to work on independent components and test them in a mobile fashion. However, moving forward, as the functionalities of the application are growing in number, having unit tests on standby will enable finalizing the product.

User Black-Box Testing (Front-End):

For the front-end, testing was done through using actual users rather than the development team themselves, to prevent bias in the functionality assessment. The development team requested friends and family members to use the application features in a given contextual situation, and collected the necessary feedback to improve the application. Some of the improvements that were suggested are included below:

- Inclusion of Instructional Pop-ups.
- Color-Coding of Buttons and Registration Statuses.
- First Response to Server Causes No Information on the Page (inconvenience).
- Instructions or Descriptions for Each Page.

The feedback above is not particularly difficult to implement, barring the aspect with the first request to the server, which we will attempt to fix by reinstalling the drivers for PostGreSQL. However, this enabled the development team to understand that the state of the front-end is good, as the functionalities are being performed as required. The enhancements mentioned will be included in the fourth and final SPRINT.

Conclusion

The security and software testing report enabled the development team to re-assess, and redistribute the workload with a greater focus on making sure that the system is as robust as possible. Overall, the planning is at a good state, and with the last of the user stories undergoing completion and testing, a greater impetus can be placed on the testing and deployment of security measures and application functionality management.