Faculty of Sciences of the University of Porto
Computer Science Department

Security in Software Engineering (CC4078)

Semester Project

Secure Learning Platform

Project participants:

António Pedro Pinheiro (201704931)
Manuel Veiga (201905924)
Simão Ribeiro (202309188)

Teachers: Hugo Pacheco & Rui Maranhão

Contents

1 Introduction

In the contemporary era, software-based systems have become integral to our daily lives. We rely extensively on these systems across various domains, including financial services, telecommunications, electronics, transportation, and household appliances, among others. [6] Given that software systems are pervasive in numerous aspects of society, security has emerged as a critical concern and a vital requirement. [8, 9] Fundamental security principles, such as confidentiality, availability, and integrity, must be upheld for software to be deemed secure. [3]

Traditionally, software security is addressed primarily during the later stages of development, often treating security concerns as an afterthought. [2] Consequently, this exacerbates the risk of introducing new vulnerabilities throughout various phases of the software development lifecycle. Adherence to traditional protection methods has fostered the "Penetrate and Patch" approach, wherein security specialists evaluate the software—often isolating it from its operational environment—by attempting to exploit common vulnerabilities. Successful penetration necessitates the development of patches and the remediation of identified vulnerabilities. Security is frequently treated as an ancillary feature within the software development lifecycle, managed by security professionals utilizing antivirus software, platform security measures, proxies, firewalls, and intrusion prevention systems. [1, 5]

Defense mechanisms appended to a software system at the conclusion of the development lifecycle, such as intrusion detection systems and firewalls, are frequently inadequate and may result in costly rework. [8] Research further indicates that such supplementary approaches to addressing security concerns are insufficient and can necessitate significant modifications, notwithstanding the intangible consequences of a security breach. [1] To mitigate such rework and modifications, security challenges must be addressed from the inception of the software development lifecycle (i.e., from software requirements elicitation through to maintenance). [1] To this end, secure software engineering has recently emerged as a prominent field of research. Software engineering employs distinct phases for development. The primary stages include: software requirements, software design, coding, testing, integration, and maintenance. [6] These phases are depicted in Figure 1.

The objective of secure software engineering is to address software security vulnerabilities by integrating security concerns and development methodologies throughout the software life cycle. Consequently, secure software engineering is a process aimed at achieving security objectives through the construction, design, and testing of the software. Software security differs from application security, inasmuch as application security focuses on protecting the software post-development and deployment. It typically encompasses various protection mechanisms, such as firewalls, antivirus software, and intrusion detection systems. [2, 3]

Thus, traditional security approaches often result in reactive measures, leaving applications susceptible to attacks. DevSecOps emerges as a proactive solution, integrating security into the DevOps pipeline from the inception of development. [10]

By adopting the "shift left" principle regarding security practices, organizations aim to address security issues as early as possible, ideally during the design and development phases. This proactive approach facilitates the identification and mitigation of security risks before they escalate, resulting in more secure and resilient software deployments. Key components of the DevSecOps pipeline include static code analysis, Dynamic Application Security Testing (DAST), container security analysis, vulnerability assessment, compliance validation, and security monitoring. These components function synergistically to identify, remediate, and prevent security vulnerabilities and compliance violations throughout the software development pro-

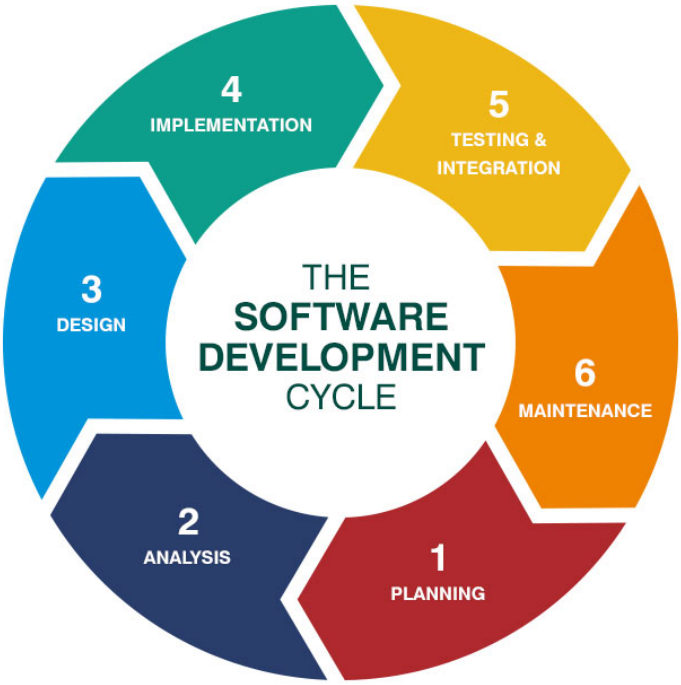


Figure 1: Software Development Phases

cess. [4]

This report is structured as follows: Section 1 provides a brief introduction to the role of security in software development, theoretical principles, and the definition of the "shift left" security principle. Section 2 outlines the security and functional requirements of the application to be developed. Section 3 details the specification of the application's design features, specifically its architecture, actors, use and misuse cases, as well as potential security threats. Section 4 describes the practical implementation of the application, and Section 5 presents a comprehensive security analysis. Finally, Section 6 offers the conclusion.

2 Functional and Security Requirements

In terms of functional requirements, we have categorized these into three groups regarding the interface, authentication, and access control, in accordance with the project specifications. [7]

2.1 Interface

- It must be possible to create, configure, edit, and delete courses;
- Each course must include:
  - A list of topics;
  - Each topic must contain:
    - A status, which may be set to draft (visible only to the owner) or published;
    - Published topics may be visible to all users or restricted to specific users;
  - A forum where registered users can discuss and ask questions regarding the topics:

    - It must be possible to upload files or images;

- There must be a personal page for each student displaying the courses in which they are registered.

2.2 Authentication

- Users wishing to forgo anonymity must authenticate with the application;
- Implement authentication based on strong password policies (minimum length, complexity, etc.);

2.3 Access Controls

- Only users with special permissions may create or delete courses;
- A course has one or more administrators capable of managing its content and student registration;
- Courses may be public or private, being accessible only to students registered therein.
- Ensure proper session management to prevent unauthorized access via:
  - Secure Session Identifiers (random, long, and unique);
  - Session Validity Expiration and Logout Mechanisms.

2.4 Injection Protection

- Protect the site against attacks involving the injection of any type of malicious code (SQL, XSS, ...);
- Implement input validation and sanitization for all information, removing or escaping any potentially malicious characters prior to data processing;
- Ensure the parameterization of database queries;
- Properly parameterize HTTP requests, specifically headers, to restrict undesirable modifications.
- Ensure secure file and image uploads, including:
  - Limiting file types and sizes;
  - Scanning files for malware;
  - Storing files appropriately with restricted access control.

2.5 Data Encryption

- Utilize secure storage for user credentials, such as hashing passwords using high-security algorithms.
- Encrypt sensitive user data, such as "personal identifiable information" (PII), both in the database and in transit between client and server.

3 Website Design

This section addresses the website design, including the system architecture, involved actors, use/misuse cases, and potential security threats. This analysis will provide a detailed understanding of how the system was structured to meet the defined functional and security requirements.

3.1 System Architecture

The architecture of the course management system was designed to be modular and scalable, ensuring that each component can be developed, tested, and maintained independently. The architecture is based on a clear separation between the frontend, backend, and database, with communication between components occurring via REST endpoints.

3.1.1 Main Components

- **Frontend:** Implemented in React with Bootstrap, responsible for all user interactions. The frontend sends HTTP requests to the backend and processes responses to display data in an intuitive and organized manner.
- **Backend:** Developed with FastAPI, manages the platform logic, authentication, authorization, and data manipulation. The backend exposes a RESTful API consumed by the frontend.
- **Database:** Uses SQLAlchemy as an ORM to interact with the SQLite database. It stores all information regarding users, courses, topics, and forum posts.
- **Upload Folder:** Manages files uploaded by users, storing them in an organized manner and facilitating access via generated URLs.

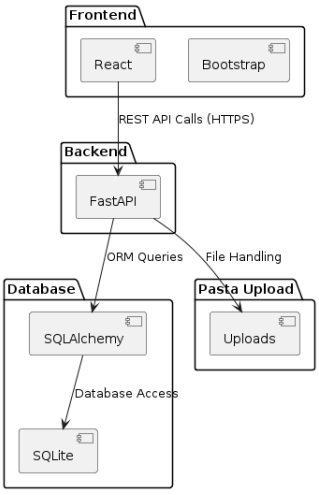


Figure 2: Component Diagram

3.1.2 Class Diagram

Figure 3 presents the class diagram representing the implementation developed within the scope of this project.

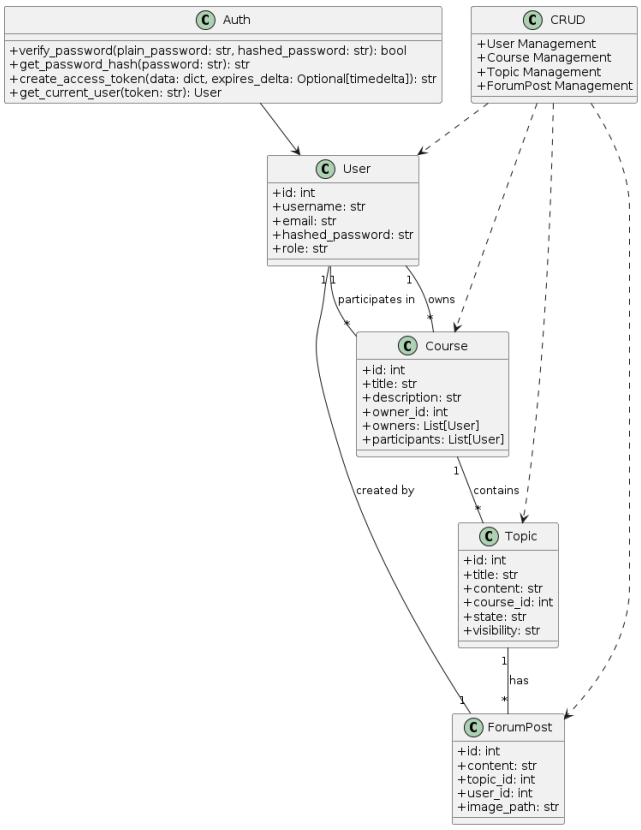


Figure 3: Class Diagram

1. **User** The `User` class represents an application user. Key attributes include:

- `id`: unique user identifier.
- `username`: username.
- `email`: user email address.
- `hashed_password`: encrypted user password.
- `role`: user role within the application, which may be "admin", "teacher", or "student".

2. **Course** The `Course` class represents a course on the platform. Key attributes include:

- `id`: Unique course identifier.
- `title`: Course title.
- `description`: Course description.

- `owner_id`: Identifier of the course owner.
- `owners`: List of users who are owners of the course.
- `participants`: List of users participating in the course.

3. Topic The `Topic` class represents a topic within a course. Key attributes include:

- `id`: Unique topic identifier.
- `title`: Topic title.
- `content`: Topic content.
- `course_id`: Identifier of the course to which the topic belongs.
- `state`: Topic status (e.g., "draft", "published").
- `visibility`: Topic visibility (e.g., "public", "private").

4. ForumPost The `ForumPost` class represents a post within a topic's forum. Key attributes include:

- `id`: unique post identifier.
- `content`: post content.
- `topic_id`: identifier of the topic to which the post belongs.
- `user_id`: identifier of the user who created the post.
- `image_path`: path to the image attached to the post.

5. Auth The `Auth` class handles user authentication and authorization. Key methods include:

- `verify_password`: verifies if the password corresponds to the stored hash.
- `get_password_hash`: generates a hash for the provided password.
- `create_access_token`: creates a JWT access token.
- `get_current_user`: retrieves the current user based on the JWT token.

6. CRUD The `CRUD` class is responsible for Create, Read, Update, and Delete operations for various entities. It is divided into four main categories:

- **User Management**: Methods for managing users, such as `get_user`, `create_user`, `update_user`, and `delete_user`.
- **Course Management**: Methods for managing courses, such as `get_course`, `create_course`, `update_course`, and `delete_course`.
- **Topic Management**: Methods for managing topics, such as `get_topic`, `create_topic`, `update_topic`, and `delete_topic`.
- **ForumPost Management**: Methods for managing forum posts, such as `get_forum_posts`, `create_forum_post`, and `delete_forum_post`.

Relationships Between Classes

- **Association**:
  - A `User` can own multiple `Courses` (`owns`).
  - A `User` can participate in multiple `Courses` (`participates in`).
  - A `Course` contains multiple `Topics`.
  - A `Topic` has multiple `ForumPosts`.
  - A `ForumPost` is created by a `User`.
- **Dependency**:
  - The `CRUD` class depends on the `User`, `Course`, `Topic`, and `ForumPost` classes to perform management operations.
  - The `Auth` class depends on the `User` class to perform authentication and authorization operations.

3.2 Actors and Use/Misuse Cases

This section describes the actors interacting with the system, the primary use cases defining expected functional interactions, and the misuse cases representing potential exploitation attempts or inappropriate system usage.

Main Actors:

- **Administrator:** Responsible for user and course creation and management. Possesses permissions to perform all system operations.
- **Teacher:** May create and manage courses, topics, and forums. Adds and removes participants from their courses.
- **Student:** Participates in courses, views topics and forum posts, and may post questions and answers in the forums.

Regarding the application use cases, a comprehensive list is provided below:

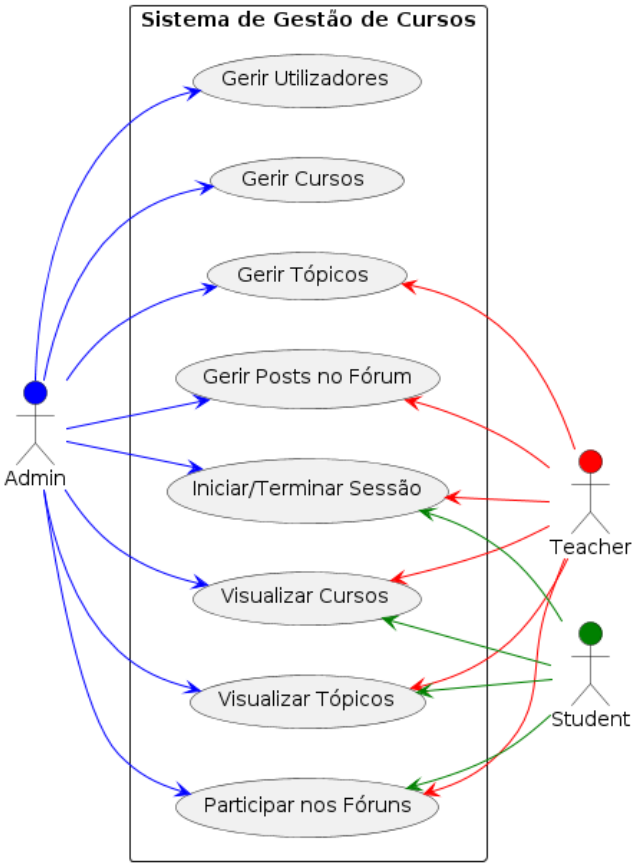


Figure 4: Use Case Diagram

- **Administrator Use Cases:**
  - **Create course:** An administrator creates a new course.
  - **Delete course:** An administrator deletes an existing course.
- **Course Owner Use Cases:**
  - **Configure course:** The owner of a course configures course details (e.g., title, description).
  - **Edit course:** The owner of a course edits the course content and topics.
  - **Delete course:** The owner of a course deletes a course they own.
  - **Enroll students:** The owner of a course enrolls students in the course.
  - **Configure forum:** The owner of a course defines forum settings, including access levels and permissions.
- **Enrolled Student Use Cases:**
  - **View course:** An enrolled student views the course and its topics.
  - **Participate in forum:** An enrolled student asks questions and discusses topics in the forum.
  - **Upload files:** An enrolled student uploads files to the forum.

Finally, regarding misuse cases—specifically abusive or malicious usage—a list of such cases follows:

- **Administrator Misuse Cases:**
  - **Unauthorized Course Deletion:** An attacker obtains administrator privileges and maliciously deletes courses.
  - **Privilege Escalation:** An attacker exploits a vulnerability to escalate privileges to the administrator level.
- **Course Owner Misuse Cases:**
  - **Unauthorized Content Modification:** An attacker impersonates a course owner and modifies or deletes course content.
  - **Improper Enrollment Management:** An attacker adds or removes students from courses without authorization.
- **Enrolled Student Misuse Cases:**
  - **Unauthorized Access to Private Topics:** An enrolled student accesses topics they are not authorized to view.
  - **Forum Abuse:** An enrolled student posts inappropriate or malicious content in the forum.
- **System Misuse Cases:**
  - **SQL Injection:** An attacker exploits a system vulnerability to execute unauthorized SQL commands.
  - **Cross-Site Scripting (XSS):** An attacker injects malicious scripts into web pages viewed by other users.
  - **Data Leakage:** An attacker obtains unauthorized access to sensitive user data via a vulnerability.
  - **Denial of Service (DoS):** An attacker overloads the system with requests, rendering it unavailable to legitimate users.

3.3 Security Threats

- **Injection Attacks:**
  - **SQL Injection:** An attacker executes arbitrary SQL code in the database by exploiting vulnerabilities in inputs.
  - **Command Injection:** An attacker injects and executes arbitrary system commands.
- **Cross-Site Scripting (XSS):**
  - **Stored XSS:** Malicious scripts are stored on the server (e.g., in forum posts) and executed in the browser of any user viewing the affected content.
  - **Reflected XSS:** Malicious scripts are reflected off a web server and executed immediately, usually via a crafted URL.

- **Cross-Site Request Forgery (CSRF):**
  - An attacker tricks a user into performing unintended actions (e.g., submitting a form or making a purchase) by exploiting the user's authenticated session.
- **Broken Authentication and Session Management:**
  - **Session Hijacking:** An attacker assumes control of a user session by stealing session tokens.
  - **Credential Stuffing:** Attackers use username/password combinations stolen from other breaches to gain unauthorized access.
- **Broken Access Control:**
  - **Privilege Escalation:** An attacker obtains higher-level access than intended (e.g., a student gains administrator privileges).
  - **Unauthorized Access:** An attacker gains access to unauthorized resources (e.g., viewing private course content).
- **Security Misconfiguration:**
  - **Default Credentials:** Use of default usernames and passwords for administrator accounts.
  - **Unnecessary Features:** Enabling unnecessary features or services that may be exploited.
  - **Improper Error Handling:** Detailed error messages that provide useful information to attackers.
- **Sensitive Data Exposure:**
  - **Unencrypted Storage:** Sensitive data stored in plaintext.
  - **Insecure Transmission:** Sensitive data transmitted without encryption, for example, via HTTP instead of HTTPS.
  - **Insecure File Uploads:** Storage of uploaded files in an insecure manner.
- **Insecure Direct Object References:**
  - An attacker obtains access to unauthorized data by manipulating object references (e.g., changing a course ID in the URL to access another course).
- **Denial of Service (DoS) and Distributed Denial of Service (DDoS):**
  - Overloading the application with excessive requests, rendering it unavailable to legitimate users.
- **Man-in-the-Middle (MitM) Attacks:**
  - Interception and alteration of communications between the client and the server without the knowledge of either party.
- **Insufficient Logging and Monitoring:**
  - **Lack of Audit Logs:** No record of user actions or security events, hindering incident response and forensic analysis.

  - **Failure to Detect and Respond to Attacks:** Slow or non-existent detection and response to security breaches.

- **Social Engineering:**
  - **Phishing:** Deceiving users into divulging credentials or other sensitive information.
  - **Spear Phishing:** Targeted phishing attacks against specific individuals, such as administrators or course owners.

- **Insecure APIs:**
  - **Unauthorized Access:** Poorly protected APIs, allowing unauthorized access to backend data.
  - **Excessive Data Exposure:** APIs exposing more data than necessary.

- **Malware:**
  - **File Uploads:** Uploading malicious files disguised as legitimate content to exploit vulnerabilities in server or client systems.

- **Insufficient Input Validation:**
  - **Buffer Overflow:** Inadequate input validation, resulting in a buffer overflow.
  - **Directory Traversal:** Inadequate validation of file paths, allowing access to unauthorized files and folders on the server.

- **Vulnerabilities in Third-Party Components:**
  - **Outdated Libraries:** Use of outdated libraries or frameworks with known vulnerabilities.
  - **Third-Party Attacks:** Compromised third-party components compromising application security.

4 Implementation

4.1 System Components

The developed course management system aims to provide a robust and efficient platform for the creation, management, and participation in online courses. This system is composed of diverse components that interact cohesively to offer an intuitive and functional user experience. The main components of the system include the frontend, implemented with React and Bootstrap; the backend, developed with FastAPI; the database managed by SQLAlchemy with SQLite; and an uploads folder for file management. Below, we describe each of these components in detail, as well as the interfaces that interconnect them.

4.1.1 Frontend (React, Bootstrap)

The system frontend is implemented using the React library, which facilitates the construction of interactive and dynamic user interfaces. Bootstrap is used to style the interface, providing a responsive and visually appealing design.

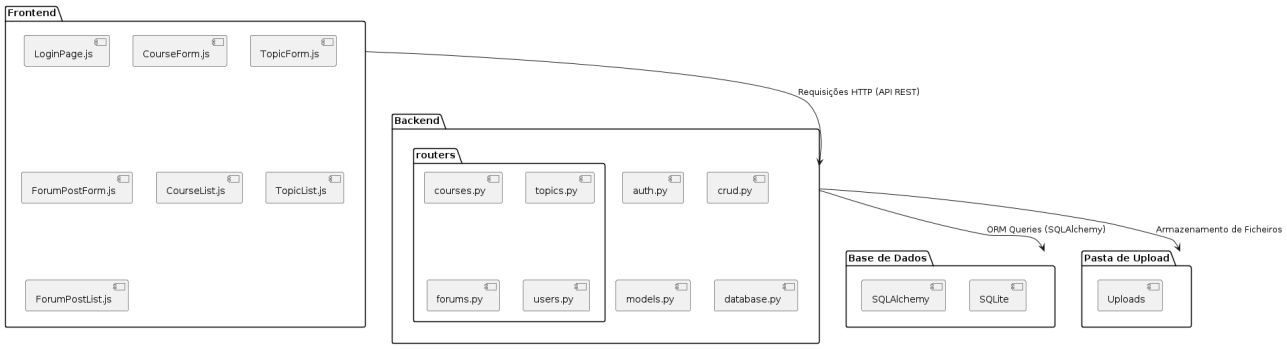


Figure 5: Component Diagram with Files

Main Functions:

- **User Interaction**: Allows users to interact with the system via forms, buttons, and other interface components.
- **Communication with the Backend**: Sends HTTP requests to the backend to perform operations such as login, course creation, topic publication, etc.
- **Data Display**: Receives data from the backend and displays it within the user interface.

Main Components:

- `LoginPage.js`: Manages the user authentication process.
- `CourseForm.js`: Allows for the creation and editing of courses.
- `TopicForm.js`: Allows for the creation and editing of topics.
- `ForumPostForm.js`: Allows for the creation of questions and answers in forums.
- `CourseList.js`: Displays the list of available courses.
- `TopicList.js`: Displays the list of topics within a course.
- `ForumPostList.js`: Displays forum posts.

4.1.2 Backend (FastAPI)

The backend is implemented using the FastAPI framework, known for its efficiency and ease of use in building RESTful APIs. FastAPI is used to manage the platform logic, authentication, authorization, and database interactions.

Main Functions:

- **Authentication and Authorization Management**: Verifies user credentials and generates JWT tokens for authenticated sessions.
- **Data Manipulation**: Performs CRUD operations (Create, Read, Update, Delete) for courses, topics, and forum posts.
- **Database Communication**: Interacts with the database via SQLAlchemy.

Main Components:

- `auth.py`: Manages authentication and JWT token generation.
- `crud.py`: Contains CRUD operations for the different data models.
- `models.py`: Defines the data models used in the system.
- `routers`:
  - `auth.py`: Routing for authentication endpoints.
  - `courses.py`: Routing for course-related operations.
  - `topics.py`: Routing for topic-related operations.
  - `forums.py`: Routing for forum post operations.
  - `users.py`: Routing for user-related operations.

4.1.3 Database (SQLAlchemy / SQLite)

Main Functions:

- **Data Storage**: Stores persistent data such as user information, courses, topics, and forum posts.
- **Queries and Updates**: Enables efficient data queries and updates via SQLAlchemy.

Main Components:

- `models.py`: Defines tables and relationships within the database.
- `database.py`: Configures the database connection and initializes the database.

4.1.4 Upload Folder

Main Functions:

- **File Storage**: Stores files uploaded by users in an organized manner.
- **File Access**: Allows files to be accessed and displayed in forum posts.

Main Components:

- **Backend Upload Configuration**: Configures FastAPI to accept and store uploaded files.
- **Frontend Integration**: Allows users to upload files via the frontend interface, which are then sent to and stored by the backend.

4.2 System Interfaces (REST Endpoints)

The interfaces between the different system components are established through REST endpoints, which facilitate communication between the frontend and the backend. These endpoints are responsible for receiving requests, processing data, and returning appropriate responses.

Main Endpoints:

- **Authentication:**
  - `POST /login`: Verifies user credentials and returns a JWT token.
  - `POST /logout`: Terminates the user session.
- **Users:**
  - `GET /users/me`: Returns details of the authenticated user.
  - `POST /users`: Creates a user.
  - `PUT /users/{id}`: Updates an existing user.
  - `DELETE /users/{id}`: Removes a user.
- **Courses:**
  - `GET /courses`: Lists all courses.
  - `POST /courses`: Creates a course.
  - `PUT /courses/{id}`: Updates an existing course.
  - `DELETE /courses/{id}`: Removes a course.
- **Topics:**
  - `GET /courses/{course_id}/topics`: Lists all topics within a course.
  - `POST /courses/{course_id}/topics`: Creates a topic.
  - `PUT /topics/{id}`: Updates an existing topic.
  - `DELETE /topics/{id}`: Removes a topic.
- **Forum Posts:**
  - `GET /topics/{topic_id}/forum_posts`: Lists all posts within a topic.
  - `POST /topics/{topic_id}/forum_posts`: Creates a post.
  - `DELETE /forum_posts/{id}`: Removes a post.

The system components, from the frontend to the database, with interfaces established via REST endpoints, form the foundation of an efficient and robust course management system. Each component plays an essential role in the system's functionality, ensuring an intuitive and efficient user experience. The adopted methodology guarantees that all system parts are well-integrated yet independent, achieving the project objectives.

4.3 Authentication Flow

Authentication is one of the fundamental processes in any course management system, ensuring that only authorized users have access to system functionalities. The following sequence diagram illustrates the detailed flow of the authentication process within the system, from the submission of the login form to the verification and storage of the JWT token.

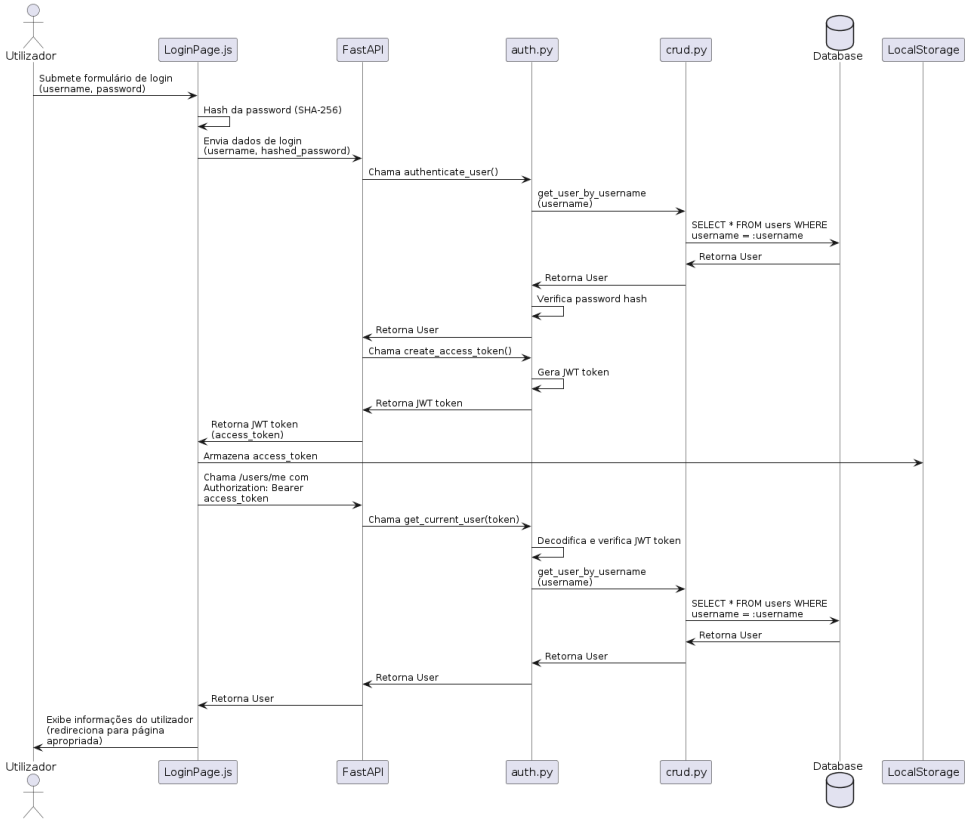


Figure 6: Sequence Diagram: Authentication

Description of the Sequence Diagram: Authentication

1. Login Form Submission: The user enters the username and password and submits the login form via the interface provided by `LoginPage.js`.

2. Password Hashing: The `LoginPage.js` applies a hash to the password using the SHA-256 algorithm before sending the data to the backend. This measure is adopted to ensure that the password is never transmitted in clear text, adding an additional layer of security.

3. Transmission of Login Data: The login data, comprising the username and the hashed password, are sent to the backend, specifically to the endpoint managed by FastAPI.

4. User Authentication:
   - FastAPI calls the function `authenticate_user()` defined in `auth.py` to initiate the authentication process.
   - `authenticate_user()`, in turn, calls `get_user_by_username()` in `crud.py` to retrieve user data.

5. Database Query:
   - `crud.py` performs a database query to search for a user with the provided username.
   - The database returns the found user data, which is sent back via `crud.py` to `auth.py`.

6. Password Verification:

- `auth.py` compares the provided hashed password with the password stored in the database.
- If the passwords match, the user is successfully authenticated.

7. JWT Token Generation:
   - `auth.py` calls `create_access_token()` to generate a JWT (JSON Web Token).
   - The JWT token is generated and returned to FastAPI.
8. Return of JWT Token: FastAPI sends the JWT token back to `LoginPage.js`.
9. JWT Token Storage: The `LoginPage.js` stores the JWT token in LocalStorage for use in future requests, ensuring that the user does not have to re-authenticate with every request.
10. JWT Token Verification:
    - The `LoginPage.js` calls the endpoint `/users/me` with the JWT token included in the authorization header.
    - FastAPI calls `get_current_user(token)` in `auth.py` to decode and verify the JWT token.
    - `auth.py` decodes the token and verifies its validity, then calls `get_user_by_username()` again to obtain updated user data.
11. Return of User Information:
    - After verifying the token, `auth.py` returns the user data to FastAPI, which in turn sends it back to `LoginPage.js`.
    - The `LoginPage.js` displays the user information and redirects to the appropriate page based on permissions and user type (Administrator, Teacher, Student).

4.4 Course Creation

Course creation is a fundamental functionality within the course management system, enabling administrators and teachers to create courses and add owners and participants. The following sequence diagram illustrates the detailed flow of the course creation process, from the submission of the creation form to the addition of owners and participants.

Description of the Sequence Diagram: Course Creation

1. Submission of the Course Creation Form: The user completes the course creation form in the frontend (`CourseForm.js`) and submits the course data, including the title, description, owners, and participants.
2. Transmission of Course Data: The frontend sends the course data to the backend, specifically to the endpoint managed by FastAPI (`courses.py`).
3. Course Creation in the Database:
   - FastAPI calls the `create_course()` function defined in `crud.py`, which inserts a new course into the database.

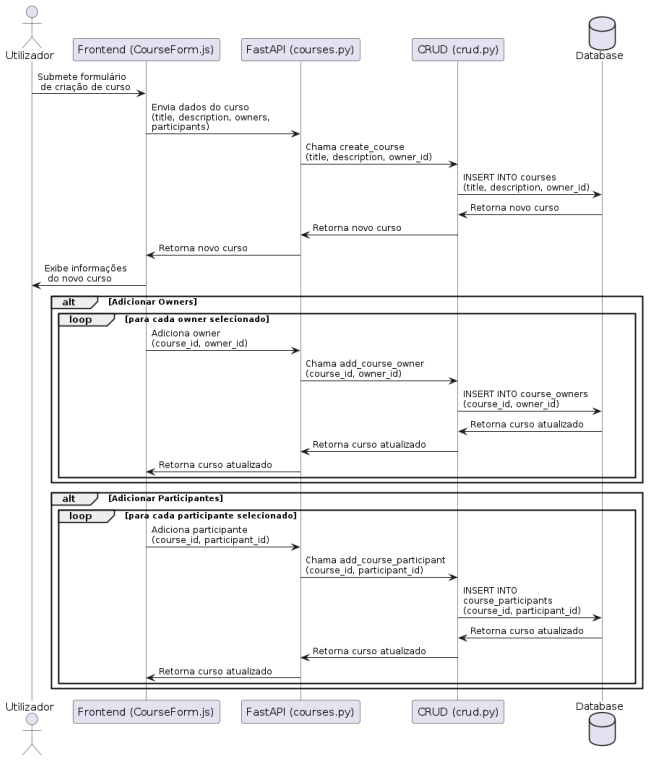


Figure 7: Sequence Diagram: Course Creation

- The `create_course()` function executes an `INSERT` command in the courses table, adding the title, description, and the initial owner ID.
- The database returns the newly created course to `crud.py`, which in turn returns this information to FastAPI.

4. Return of the New Course: FastAPI returns the information of the newly created course to the frontend (`CourseForm.js`), which displays the course details to the user.

5. Addition of Owners:
   - If additional owners were selected, the frontend sends requests to add each owner to the course.
   - For each owner, FastAPI calls the `add_course_owner()` function in `crud.py`, which inserts a new record into the `course_owners` table.
   - The database returns the updated course after each insertion, and this information is returned to the frontend.

6. Addition of Participants:
   - Similarly, if participants were selected, the frontend sends requests to add each participant to the course.
   - For each participant, FastAPI calls the `add_course_participant()` function in `crud.py`, which inserts a new record into the `course_participants` table.
   - The database returns the updated course after each insertion, and this information is returned to the frontend.

4.5 Topic Publication

Topic publication is a key functionality allowing teachers to make content available to students in an organized manner. The following sequence diagram illustrates the detailed flow of the topic publication process, from the user's action to the update of the topic status in the database.

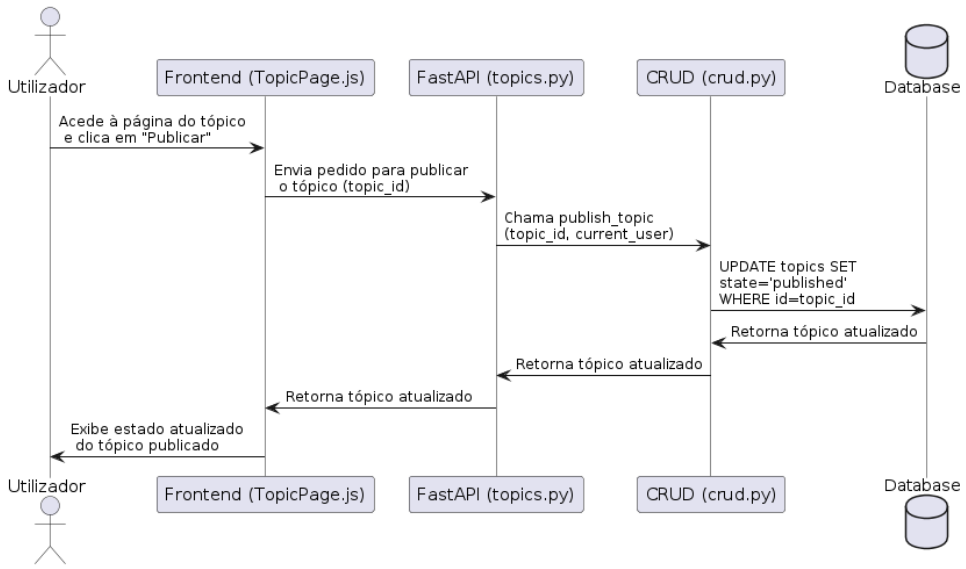


Figure 8: Sequence Diagram: Topic Publication

Description of the Sequence Diagram: Topic Publication

1. User Action: The user (teacher) accesses the topic page on the frontend (`TopicPage.js`) and clicks the "Publish" button to change the topic status from draft to published.

2. Submission of Publication Request: The frontend (`TopicPage.js`) sends a request to the backend to publish the topic, including the `topic_id` in the request.

3. Call to Publication Function: The backend (`FastAPI` in `topics.py`) receives the request and calls the `publish_topic()` function defined in `crud.py`, passing the `topic_id` and the current user (`current_user`) as parameters.

4. Update of Topic Status:
   - The `publish_topic()` function in `crud.py` executes an `UPDATE` command in the database to change the topic status to 'published'.
   - The database updates the topic record and returns the updated topic to `crud.py`.

5. Return of Updated Topic: `crud.py` returns the updated topic to `FastAPI` (`topics.py`), which in turn sends this information back to the frontend.

6. Display of Updated Status: The frontend (`TopicPage.js`) receives the updated topic and displays the new topic status (published) to the user.

4.6 Forum Question Submission

Submitting questions in the forum allows users (students and teachers) to interact and discuss specific topics collaboratively. The following sequence diagram illustrates the detailed flow of the forum question submission process, from user submission to the display of the new question on the frontend.

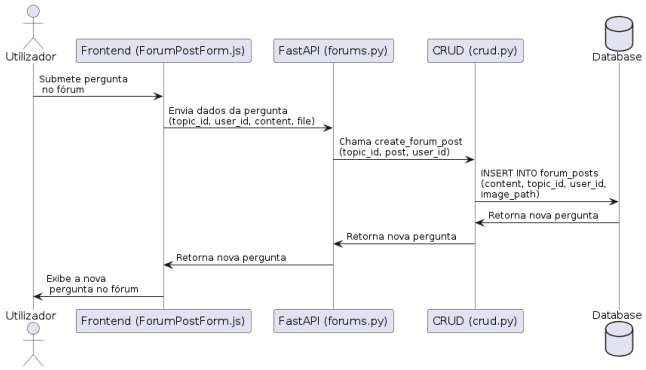


Figure 9: Sequence Diagram: Forum Question Submission

Description of the Sequence Diagram: Forum Question Submission

1. User Question Submission: The user (student or teacher) completes the forum question submission form in the frontend (`ForumPostForm.js`) and submits the question.
2. Transmission of Question Data: The frontend (`ForumPostForm.js`) sends the question data to the backend, including `topic_id`, `user_id`, `content`, and `file` (if applicable).
3. Call to Post Creation Function: The backend (`FastAPI` in `forums.py`) receives the data and calls the `create_forum_post()` function defined in `crud.py`, passing the question data as parameters.
4. Insertion of the Question into the Database:
   - The `create_forum_post()` function in `crud.py` executes an `INSERT` command in the `forum_posts` table, adding the question content, `topic_id`, `user_id`, and `image_path` (if applicable).
   - The database inserts the new record and returns the created question to `crud.py`.
5. Return of the New Question: `crud.py` returns the new question to `FastAPI` (`forums.py`), which in turn sends this information back to the frontend.
6. Display of the New Question: The frontend (`ForumPostForm.js`) receives the new question and displays it in the forum, making it visible to all users with access to the topic.

4.7 File/Image Upload

File or image uploading is an essential functionality allowing users to attach documents or images to their forum posts. The following sequence diagram illustrates the detailed flow of the file or image upload process, from user submission to the display of the file URL in the frontend.

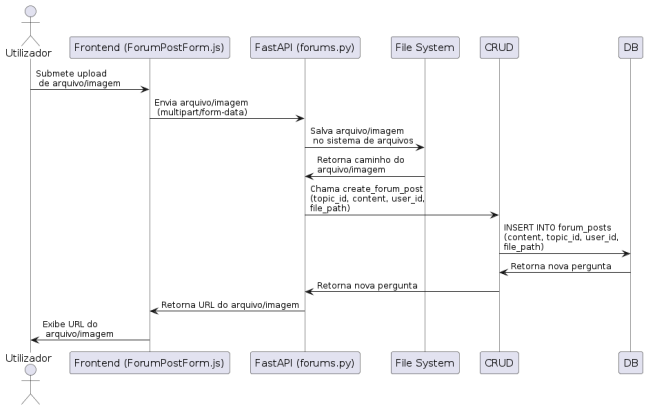


Figure 10: Sequence Diagram: Forum Post and File/Image Upload

Description of the Sequence Diagram: File/Image Upload

1. User Upload Submission: The user selects a file or image and submits the upload form in the frontend (`Frontend`).
2. File/Image Transmission: The frontend sends the file or image to the backend (`FastAPI`), using a multipart/form-data HTTP request.
3. Saving File/Image:
   - The backend (`FastAPI`) receives the file or image and saves it to the file system (`File System`).
   - The file is stored in a folder dedicated to uploads, with a unique filename to prevent conflicts.
4. Return of File/Image Path: The file system (`File System`) returns the path where the file or image was saved to the backend.
5. Return of File/Image URL: The backend (`FastAPI`) constructs the access URL for the file or image based on the returned path and sends this URL back to the frontend.
6. Display of File/Image URL: The frontend receives the file or image URL and displays it to the user, allowing the uploaded file or image to be viewed or accessed.

5 Security Analysis

Security is a critical aspect in any course management system. To ensure data integrity and confidentiality, various security measures have been implemented. Below, the main strategies and analyses performed to fortify system security are described.

5.1 Injection

To prevent injection attacks, several techniques have been implemented:

- File Extension Restriction: Only files with specific extensions are permitted for upload in the forum. This assists in preventing the injection of malicious files that could compromise the system.
- Regular Expressions to Prevent XSS: Regular expressions were utilized to sanitize inputs in forum comments, preventing Cross-Site Scripting (XSS) attacks. The pattern used is `r'[<>]'`, which removes characters that could be used to inject malicious scripts.

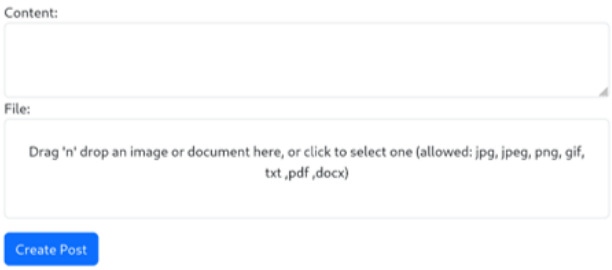


Figure 11: File Extension Restriction

5.2 Vulnerability Analysis

Various vulnerability analysis tools, both static and dynamic, were utilized to identify and mitigate potential security flaws within the system.

- Dynamic Analysis with Wappalyzer and Nuclei:
  - The Wappalyzer tool was utilized to identify technologies and potential vulnerabilities within the application layers. Adopting the role of a pen-tester, this step facilitates subsequent phases of the penetration test.
  - Nuclei was employed to scan the system for known vulnerabilities, enabling the proactive remediation of issues. Scans were executed both generically and utilizing specific templates. Consequently, information regarding vulnerabilities within the transmitted headers was obtained.
- Static Analysis with Node's Packet Vulnerability Search: This tool was utilized to identify vulnerabilities within Node.js package dependencies, ensuring that all utilized libraries remain free of known flaws. Eight vulnerabilities were identified in the packages:

  Complementing the information obtained via Nuclei, and in order to remediate these flaws, we applied patches; specifically, we overrode the versions of vulnerable packages within the `packages.json` file, updating them to versions where the vulnerabilities had been resolved.
- Static Analysis with Sonar Cloud: Sonar Cloud provided a detailed analysis of the source code, identifying no vulnerabilities, as it assigned a "Security Rating" of A across all analyzed pages and files.

Figure 12: Nuclei Tool Analysis

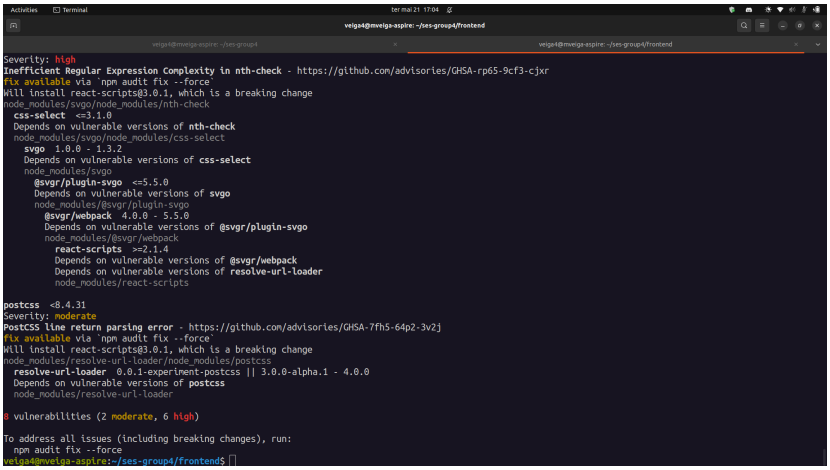


Figure 13: Node's Packet Vulnerability Analysis before patching

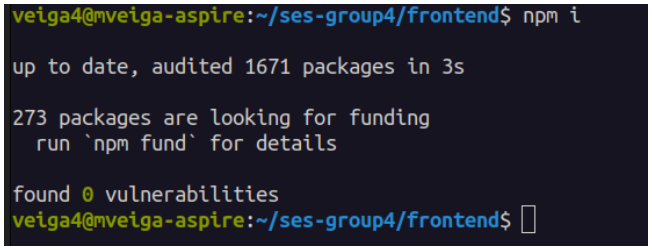


Figure 14: Node's Packet Vulnerability Analysis after patching

- Dynamic Analysis with SQLMap: SQLMap was utilized to test the application against SQL Injection attacks. The tool automated the process of detecting and exploiting SQL injection vulnerabilities, helping to ensure that database queries are secure. The results

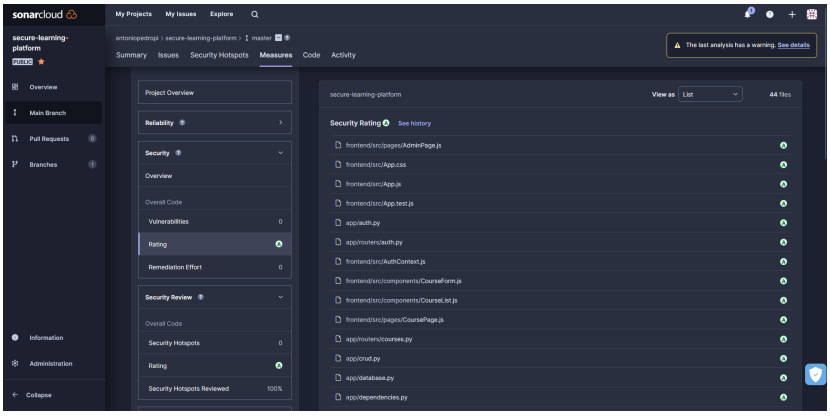
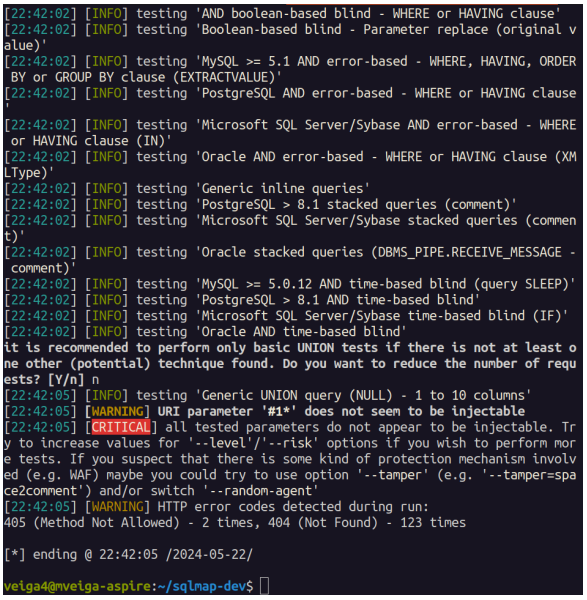


Figure 15: Sonar Cloud Analysis

of the test performed on the login page were positive, demonstrating that the parameters were properly parameterized.

(a) SQL Queries made in order to try an injection



(b) Global Results

Figure 16: SQLMap Analysis

- Dynamic Analysis with ZAP: OWASP ZAP (Zed Attack Proxy) was utilized to conduct dynamic penetration tests on the web application. The analysis provided by SQLMap regarding injection testing is not exhaustive, given the potential for XSS vulnerabilities and the possibility of SQL code injection extending beyond the login page to include topic content. Consequently, this tool was employed to assess these potential vectors. The results were positive, demonstrating that the system was successfully protected against injection attacks.

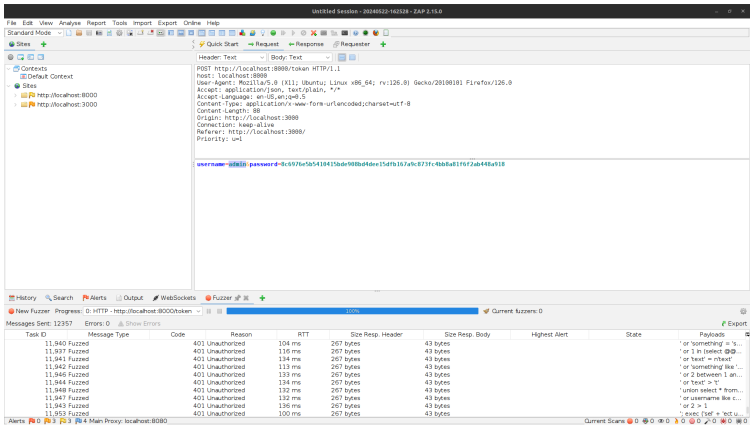


Figure 17: ZAP Analysis for the 'username' field

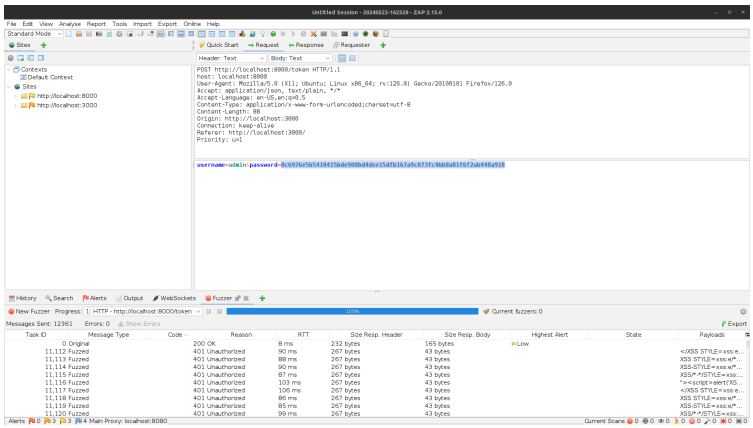


Figure 18: ZAP Analysis for the 'password' field

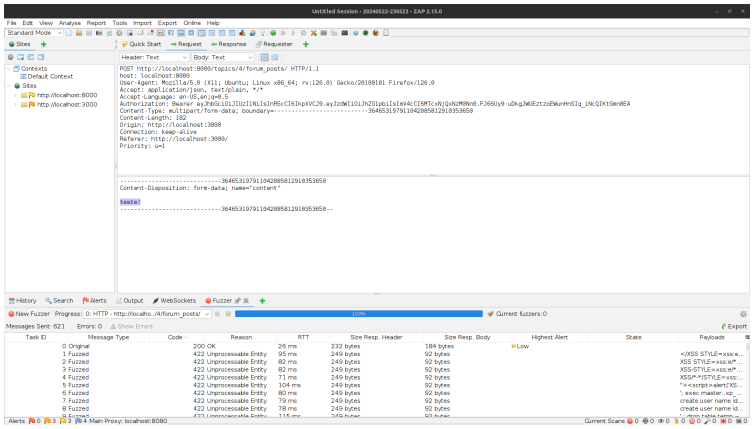


Figure 19: ZAP Analysis for the 'topic' field

6 Conclusion

Within the scope of this project, we had the opportunity to reassess the concept of software security, viewing it not merely as a reactive measure but as a proactive strategy, integrating it throughout all phases of code development. Consequently, we designed, implemented, and analyzed a learning application developed from scratch specifically for this project. We success-

fully constructed a secure application that incorporates protection against a variety of attacks capable of compromising its confidentiality and integrity. We also wish to highlight that, while the results from the security tools tested were positive, they exhibit certain limitations, particularly regarding resistance to XSS injection. The regular expression implemented to prevent such attacks filters the '<' and '>' characters, as we proceeded under the assumption that XSS payloads generally contain these specific characters. However, this is not invariably the case, and there may exist XSS payloads lacking these characters that could potentially be injected. Finally, we conclude by emphasizing the importance of integrating and considering security from the initial phase of software development, in order to safeguard and facilitate subsequent phases that benefit from this methodology.

References

[1] J. Jürjens. Secure systems development with uml. In *Security and Cryptology*, page 318. Springer, 2005.

[2] M.U.A. Khan and M. Zulkernine. Quantifying security in secure software development phases. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference*, pages 955–960, 2008.

[3] M.U.A. Khan and M. Zulkernine. On selecting appropriate development processes and requirements engineering methods for secure software. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, pages 353–358, 2009.

[4] Infosys Limited. Speed and security: How to achieve both in agile development. `https://www.infosys.com/agile-devops/documents/speed-security.pdf`. Accessed: 2024-05-20.

[5] G. McGraw. *Software Security: Building Security In*. Addison Wesley, 2006.

[6] Nabil M. Mohammed, Mahmood Niazi, Mohammad Alshayeb, and Sajjad Mahmood. Exploring software security approaches in software development lifecycle: A systematic mapping study. *Computer Standards Interfaces*, 50:107–115, 2017.

[7] Hugo Pacheco. Project documentation. `https://github.com/hpacheco/ses/blob/main/project/Project.md`, 2024. Accessed: 2024-05-20.

[8] P. Salini and S. Kanmani. Security requirements engineering process for web applications. *Procedia Engineering*, 38:2799–2807, 2012.

[9] P. Salini and S. Kanmani. Survey and analysis on security requirements engineering. *Computers & Electrical Engineering*, 38(6):1785–1797, 2012.

[10] Kiran Kumar Voruganti. Implementing security by design practice with devsecops shift left approach. *Journal of Technological Innovations*, 2(1), 2021.