Report of the Web Project by Vafa Gafarzadeh, Sevinj Rustamova, Nargis Bakshaliyeva, Shahana Huseynzada, Narmin Valiyeva

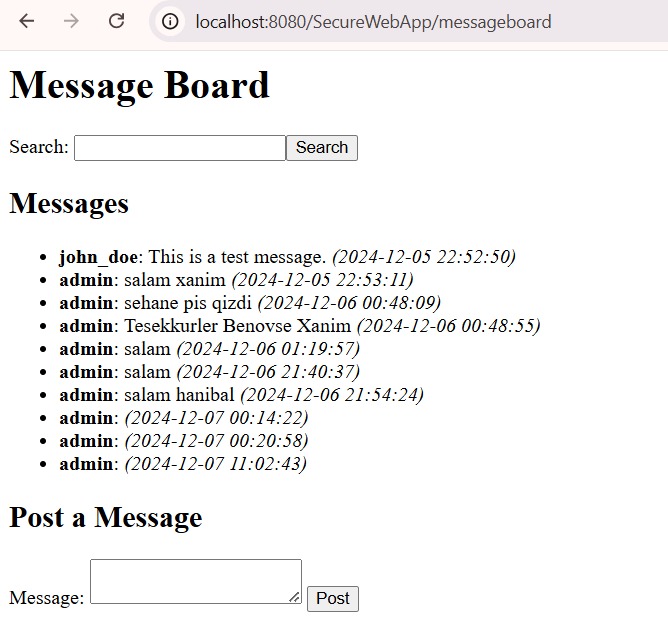
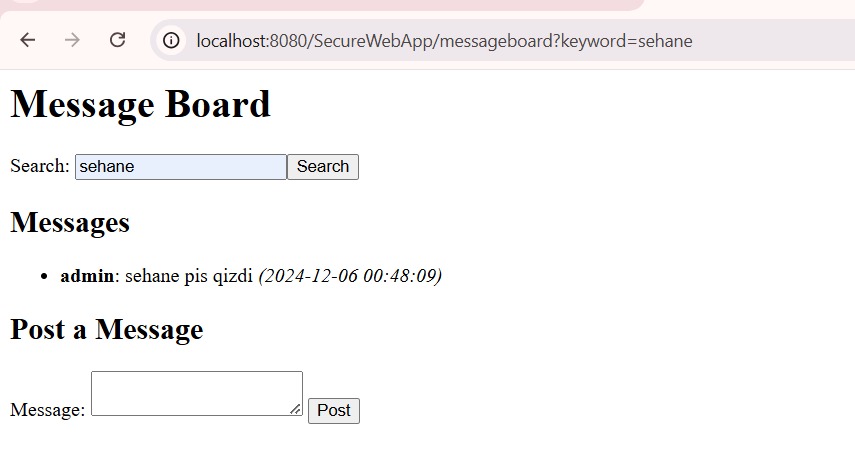
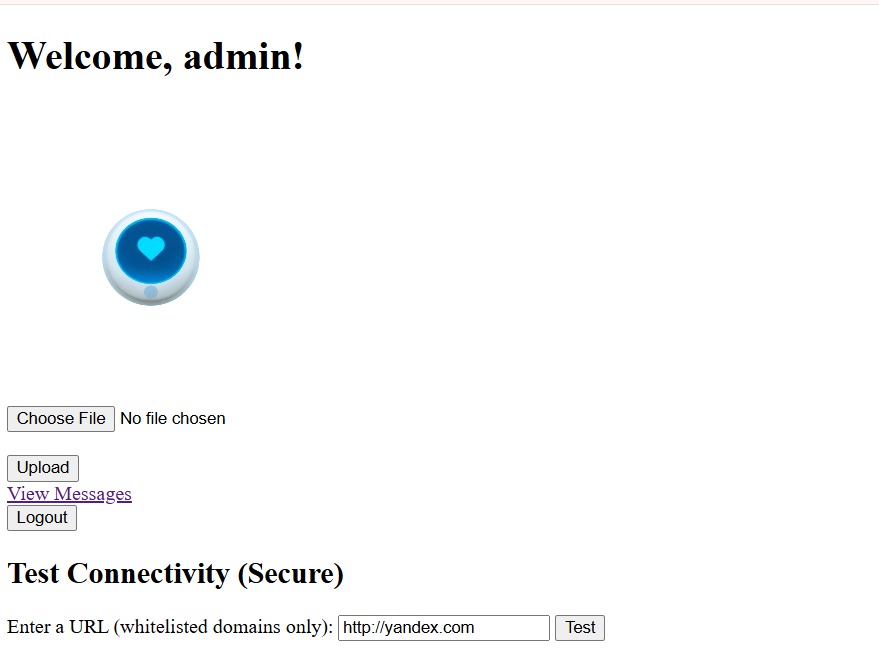
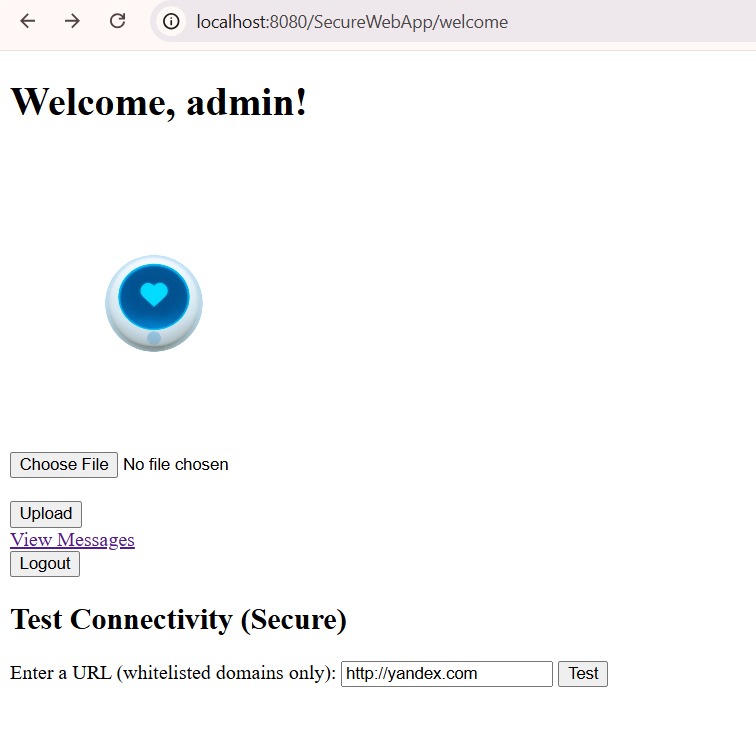
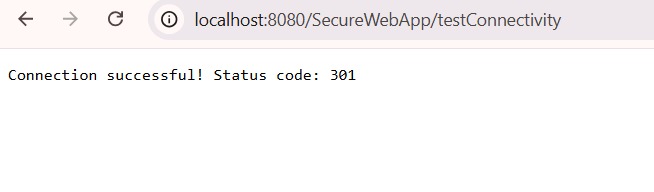
**Objective, Scenario and Requirements**

As part of our group project on Web Application Vulnerabilities and Security Best Practices, we developed a Java-based web application for "SecureWeb Inc." - a mock company. The application will demonstrate some common security vulnerabilities along with their secure implementations.

**Login Page**: Implemented the functionality of a login to allow users to authenticate themselves via a username and password. Simulated an attack, such as SQL Injection, and introduced secure coding practices, such as parameterized queries.

**Message Board**: Implemented a message board to post/view messages by authenticated users. Identified/patched various vulnerabilities such as Cross-Site Scripting (XSS) by sanitizing and validating user input before being displayed on the webpage.

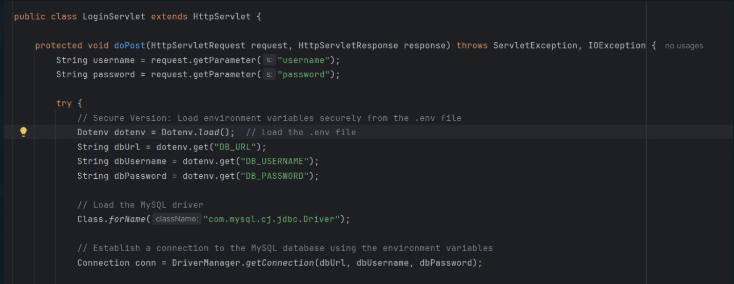
**File Upload Feature**: Created the file upload functionality for users to upload their profile pictures. Showcased how file upload vulnerabilities, such as uploading malicious files, could be used to take advantage of the web application and mitigated the issue using file type validation, size limits, and stored them outside the web root.

In the process, we demonstrated a series of web application vulnerabilities and best practices to secure each feature.**Tasks**

1. **Implementing Secure HTTP Requests and Session Management**

We started by creating a LoginServlet that processed POST requests to validate user credentials. For simplicity, we used hardcoded credentials (admin/admin) as the login details. When a user submitted their credentials via the login form, the servlet would validate the input and initiate a session if the credentials matched.

However, we knew exposing sensitive data in GET requests could lead to security vulnerabilities, such as information leakage in the URL or logs. To mitigate this, we ensured that the login form transmitted the credentials securely via POST requests only.



Once a user successfully logged in, the application created a session for them using the HttpSession object. This session was key to tracking the user's authentication state and granting them access to the message board.

We knew that effective session management was critical to preventing unauthorized access, so we implemented the following best practices:

1. **Restricted Access**: The message board servlet checked whether a valid session existed before granting access. If a user tried to bypass the login process, they were redirected to the login page.
2. **Session Timeout**: To minimize security risks like session hijacking, we set a session timeout of 5 minutes. After this period of inactivity, users were required to log in again.

We created a LogoutServlet to handle user logouts. When the user clicked "Logout," the servlet invalidated the current session using session.invalidate(). This ensured that any session data was cleared, preventing unauthorized access if someone else used the same browser.

A screenshot of a computer program

Description automatically generated

**Secure Implementation Highlights**

* We avoided exposing sensitive data in URLs or GET requests, ensuring that credentials were not accidentally logged or cached.
* By setting a session timeout and implementing logout functionality, we reduced the risk of lingering active sessions being exploited.

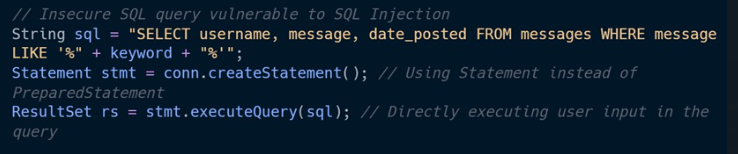
**2. Implementing a Secure Search Feature to Prevent Injection Attacks**

To implement the search feature, we allowed users to search messages by keyword. Initially, the query we used was vulnerable to SQL Injection, which could allow malicious inputs to manipulate the database.

To secure it, we replaced the approach with a prepared statement that treated user input as data rather than executable code. This change effectively safeguarded the database from malicious attacks.

After testing with various inputs, we confirmed the feature worked flawlessly and securely. This task highlighted the critical role of secure coding practices in protecting applications from injection attacks, ensuring *SecureWeb Inc.* remains resilient

Here's a vulnerable implementation of the MessageBoardServlet where the query is constructed using direct string concatenation, making it susceptible to SQL injection.



To prevent SQL Injection, always use Prepared Statements. Here's how to securely implement the feature:

A computer screen shot of code

Description automatically generated

**3. Broken Access Control**

To fix broken access control, we made sure that only authenticated users could access the message board and upload files. This included implementing a Servlet filter to validate user sessions and block access to sensitive endpoints.

Initially, it was possible to directly access /messageBoard.jsp without any authentication, as the login mechanism was bypassed. This is a serious security breach because users who are not authorized can access features that should be restricted to them.

web.xml - Deployment descriptor to configure the servlet and filter mappings.

AuthenticationFilter.java - Filter that intercepts requests to secure URLs and checks user authentication.



In this respect, a Servlet filter was created to block requests from reaching sensitive endpoints. This filter verified whether the user had an active session and was authenticated. If this was not the case, then the user would be forwarded to the login page.

A screenshot of a computer screen

Description automatically generated

We have added some session validation checks in servlets handling file upload and message board accesses. By doing so, there was another layer of assurance that unauthenticated requests cannot pass the filter or reach these features.

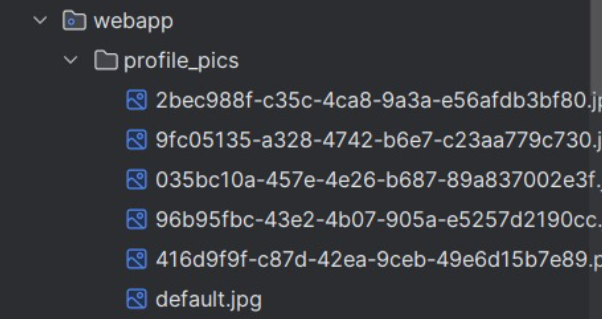
**4. Sensitive Data Exposure**

To handle file uploads securely, we stored uploaded files in a dedicated folder and saved their metadata in the database. This approach separated the actual files from the application logic and reduced the risk of exposing sensitive data.

Initially, files were served directly from the /uploads directory. This allowed anyone with the file's URL to access it, bypassing authentication and access control.

To handle this, we created a secure download servlet. Instead of serving the files directly, the servlet checked the session and permissions of the user before granting access to any uploaded file. Unauthorized requests were rejected, and users were redirected where appropriate.

Should store uploaded files outside of the webroot (i.e., outside of the webapps directory). This way, files are not directly accessible from the URL.



We also followed best practices for handling files, including storing uploads outside the web root and making sure that only authenticated users could access files through the servlet.

A computer screen shot of a program code

Description automatically generated

We first ensure the user is logged in before allowing them to upload pictures.

By implementing these measures, file upload and download became secure, and only the legitimate user could access sensitive files. This further strengthened the application in terms of commitment towards user data security and maintaining the security of SecureWeb Inc.

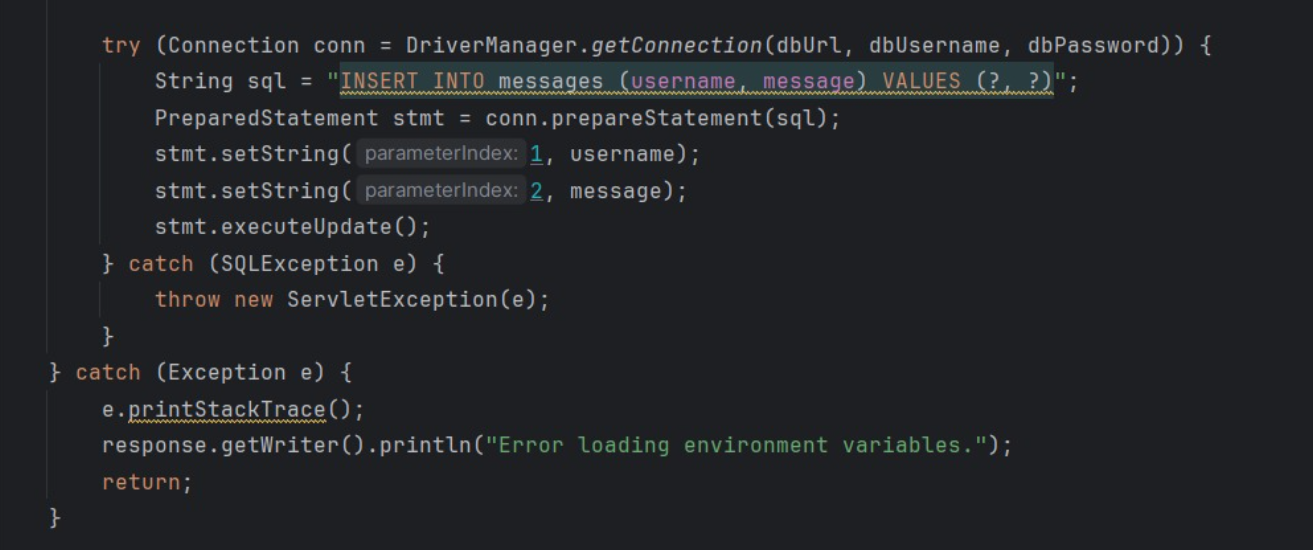
**5.** **Cross-Site Scripting (XSS)**

To enable users to post messages on the message board, we needed to ensure that the functionality was both dynamic and secure. Initially, user input was displayed directly on the browser without any filtering, creating a vulnerability to XSS attacks. Malicious users could inject scripts into their messages, potentially compromising other users’ browsers and data.

**Secure Implementation**

To address this, we implemented input sanitization to remove potentially harmful code from user submissions. Using libraries like the OWASP Java HTML Sanitizer, we ensured that only safe content was allowed, stripping out any scripts or malicious tags. Additionally, we encoded the output before rendering it in the browser, further mitigating risks.

Before:



After:

A computer screen shot of a program

Description automatically generated

Before pop up was:

A computer screen shot of a message

Description automatically generated

A white rectangular object with black text

Description automatically generated

A computer screen shot of a program

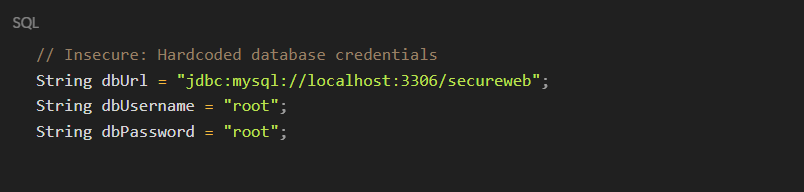
Description automatically generated

The sanitization process effectively neutralized malicious inputs while preserving legitimate user content. This approach ensured a secure and user-friendly experience on the message board, reinforcing the trustworthiness of *SecureWeb Inc.*.

**6. Security Misconfiguration**

To manage database credentials and application settings securely, we avoided storing sensitive information, such as credentials, directly within the Servlet code. Initially, storing credentials as plain text within the code posed a significant risk, as unauthorized access to the source code could expose sensitive details.

The insecure version directly **hardcodes** the database credentials (URL, username, and password) into the code, as seen in the commented-out section:



A screenshot of a computer screen

Description automatically generated

**Secure Version:**

In the secure implementation, the database credentials are **stored in environment variables** and are loaded using the Dotenv library:

A screen shot of a computer code

Description automatically generated

A screen shot of a computer

Description automatically generated

To enhance security, we utilized a configuration file to store the credentials and other application settings. The file was securely stored and encrypted to prevent unauthorized access. Additionally, environment variables were used to retrieve sensitive information, ensuring that the credentials were not hardcoded or exposed in the application repository.

This method improves security by storing the credentials in a separate, **non-version-controlled .env file** that is not included in the source code repository. The credentials are loaded at runtime from environment variables, meaning they are not exposed in the code itself. This is a more secure approach because it prevents the accidental exposure of sensitive data in source code, and also allows for different configurations in different environments (development, staging, production).

**7. Insecure Design**

Password Storage in Plaintext:

* If the BEFORE INSERT and BEFORE UPDATE triggers are not implemented, the passwords would be stored in plaintext in the database.

Secure Design

1. Using Triggers for Hashing:
   * The implementation of triggers (before\_users\_insert and before\_users\_update) ensures that passwords are always stored in a hashed format using SHA2:
     + Before inserting a new user, the trigger checks if the password length is not 64 (indicating it's not hashed) and hashes it.
     + During updates, the same logic is applied to ensure newly updated passwords are also hashed.

**A computer screen shot of a program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

In the login feature, the password comparison happens as follows:

* The user-provided password is hashed using SHA2 during the login query:

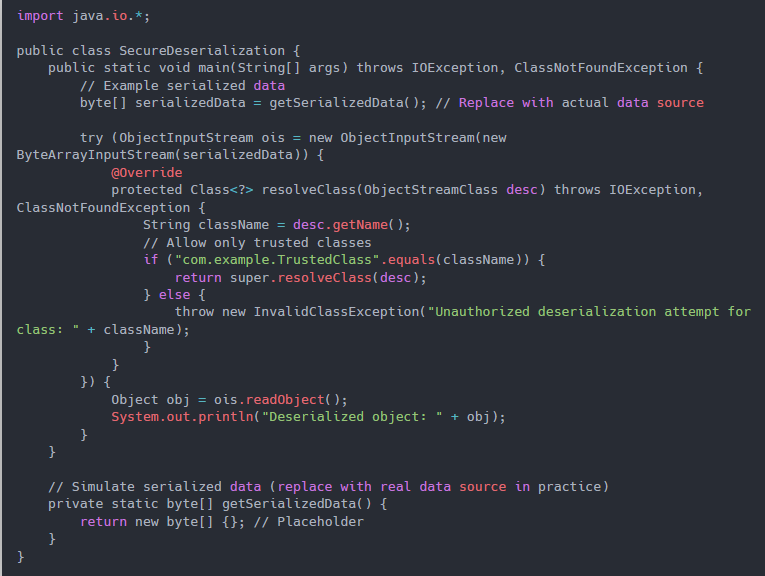
A screen shot of a computer code

Description automatically generated

* This hashed value is compared with the pre-hashed value in the database, ensuring secure authentication.

**8**. **Insecure Deserialization**

Insecure deserialization occurs when user-provided data is deserialized without proper validation. Malicious users could modify the serialized data, leading to security vulnerabilities such as remote code execution or other attacks. To prevent this, we must ensure that the data being deserialized is properly validated and originates from a trusted source.



Secure Implementation:

To mitigate the risks of insecure deserialization, we must:

1. Validate Input: Ensure that only trusted data is deserialized.
2. Use Safe Deserialization Libraries: Some libraries provide safer methods to deserialize data by rejecting unsafe data or only allowing certain classes to be deserialized.

**9. SSRF (Server-Side Request Forgery)**

Server-Side Request Forgery (SSRF) is a vulnerability where an attacker can manipulate a server into making HTTP requests on their behalf. This can lead to attacks like unauthorized access to internal resources, data leaks, or exploitation of internal services. In the context of a feature where users can test connectivity to a URL, if there’s no restriction on the URLs they can test, attackers could use it to send requests to internal services or other malicious endpoints.

**Insecure Version**

* **Behavior:**
  + The code does not validate the user input (urlString) against a whitelist of allowed domains.
  + Any URL entered by the user is processed directly, and the HTTP connection is opened to check the status code.
  + For example, if the user inputs https://www.bing.com, the application will proceed to make a connection and return the status code, without checking whether the domain is safe or allowed.
* **Key Risk:**
  + This behavior makes the application vulnerable to **Server-Side Request Forgery (SSRF)** attacks.
  + An attacker could provide malicious URLs pointing to internal systems or unauthorized endpoints, leading to data leakage or system compromise.
* **Example Code (Vulnerable):**

A screen shot of a computer code

Description automatically generated

**Secure Implementation:**

To mitigate SSRF, we must:

1. **Restrict URL input**: Only allow URLs from trusted domains (e.g., whitelisting domains like example.com).
2. **Validate URL format**: Ensure that the input is a valid and properly formed URL.
3. **Limit allowed protocols**: Restrict the protocols to http, https, etc., and avoid allowing protocols like ftp, file, gopher, etc.
4. **Behavior:** 
   1. The code includes validation to ensure that the domain of the provided URL (host) is included in a predefined whitelist.
   2. If the domain is not in the whitelist, the request is rejected, and the user receives the message:URL is not allowed. Please use a whitelisted domain.For example, inputting https://www.bing.com would trigger this message because bing.com is not part of the whitelist.
5. **Secure Validation:** 
   1. The code normalizes the domain by stripping the www. prefix if it exists and compares the normalized domain against the whitelist.
   2. This ensures only allowed domains are processed further.
6. **Example Code (Secure):**

**A computer screen shot of a computer code

Description automatically generated**

**A screen shot of a computer

Description automatically generated**

**10. Insufficient Logging and Monitoring**

Insufficient logging and monitoring can leave an application vulnerable by not recording important security-related events. Without proper logging, failed login attempts, file uploads, or suspicious activity may go unnoticed, making it easier for attackers to exploit the system.





A screen shot of a computer code

Description automatically generated

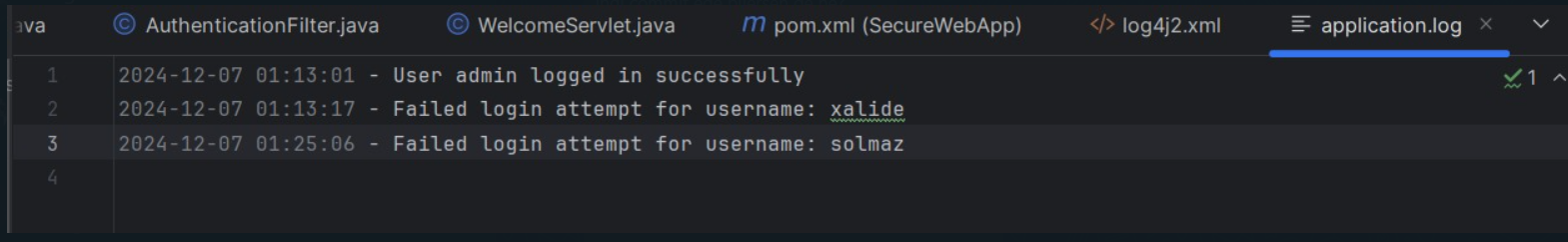
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated





To ensure proper logging, we must:

1. **Log Failed Login Attempts**: Capture and log failed login attempts, including the username, timestamp, and reason for failure.
2. **Sanitize Logs**: Ensure sensitive data, such as passwords, are never logged.
3. **Store Logs Securely**: Logs should be stored in a secure, tamper-proof location, and access should be restricted to authorized personnel.

Here is an implementation demonstrating how to log failed login attempts securely using Java and a logging framework like SLF4J with Logback:

**Explanation:**

1. **Logging Failed Login Attempts**:
   * In the login method, we log a successful login attempt with logger.info and a failed login attempt with logger.warn.
   * The log includes the username and timestamp of the failed attempt, which can be valuable for detecting brute-force attacks or suspicious behavior.
2. **Sanitizing Logs**:
   * We ensure that sensitive information, such as passwords, are **never logged**. In the case of a failed login, only the username and timestamp are logged.
3. **Log Output**:
   * Logs are output to both the console and a log file (logs/app.log).
   * Log entries include the timestamp, log level (INFO, WARN), and message.

**Deliverables and Evaluation**

The project involves creating a web application with several features, including login, message board, and file upload. The main objective is to demonstrate common security vulnerabilities and then apply secure coding practices to fix them.

**Deliverables:**

1. Running Web Application: The application must have the following features:
   * A login page with user authentication.
   * A message board where authenticated users can post and view messages.
   * A file upload feature allowing users to upload profile pictures.
2. Report: A brief explanation of:
   * Vulnerable Implementations: Discusses how the original application contains security flaws (e.g., no session management, SQL injection vulnerabilities, lack of input validation).
   * Secure Implementations: Describes how each vulnerability was mitigated with secure coding practices, such as using prepared statements, session validation, input sanitization, and safe file handling.
   * Key Takeaways: Summarizes lessons learned about secure web application development, including the importance of user input validation, logging, and monitoring.

**Evaluation:**

* Functional Implementation: The web application must work as intended, including all core features (login, message board, file upload).
* Identification of Vulnerabilities: The project should include an analysis of security weaknesses in the initial version of the application.
* Successful Application of Secure Coding Practices: Ensure that secure implementations have been applied to resolve vulnerabilities.
* Clear Documentation: Document all changes made, detailing how vulnerabilities were identified and mitigated. This documentation is key to demonstrating understanding and improvements made to the original code.

This project will test your ability to identify and fix security issues, implement best practices, and document your process clearly.