Job Application Tracker - Security

By Alvaro Espinoza Merida, Bailey Voyles, Nick Andrew, Thomas Stone

Abstract

In our capstone project, our group created a job application tracker. The primary purpose of this application was to help users keep track of the applications that they had applied for and help them automate the process of keeping track of their applications. Since our group for this security class was already the same as the group for our capstone project, we decided to pursue a project that had us secure our application. This application was implemented using Java Spring boot, ReactJS, Vite, and TailwindScss. On the security side of our application, we first secured users' passwords by making sure they were encrypted, created JWT tokens for authenticating users, and updated our backend so that users could only access sensitive information with proper passwords and tokens, we then implemented HTTPS into our application to secure data in transit, and finally we ran our backend and frontend of our application through security auditing tools to see current security vulnerabilities of our application we used tools like OWASP dependency check, and finally we attempted to fix these security vulnerabilities. This report goes through this outlined process and talks about the number of different processes and challenges we went through in implementing security into a full-stack application.

Introduction & Problem Description

In today's job market, job seekers often submit dozens of applications, making it crucial to have a reliable system for tracking these applications. Our Job Application Tracker was developed to address this need, providing job hunters with a centralized platform to manage their job application related emails. As developers, we want our users to feel secure and protect their sensitive personal and professional information. This is why we implemented robust security measures including encrypted password storage, JWT token authentication, and HTTPS protocols for secure data transmission. Along with these security measures we utilized security auditing tools like OWASP dependency check to ensure our users are protected. Our journey has involved securing the front-end and back-end, which were built using Java Spring Boot, React, Vite, and TailwindCSS, while also addressing security challenges and vulnerabilities along the way.

Technical Approach and Implementation

Spring Boot Security Features

One of the advantages of using Spring Boot for our application is that it already has security features built in. This provided us with a good foundation for the rest of our security implementations. Spring boot offered many different protections that we didn't have to implement ourselves:

- Protections against Cross-site request forgery attacks through automatic token generation and validation
- Session management and handling cookies securely
- Protections against common web vulnerabilities including XSS and SQL injections
- Automatic security headers configuration

These built in features allowed us to focus on implementing JWT authentication and HTTPS, rather than building our the basic security protections that spring boot provides.

Implementing Json Web Tokens(JWT)

The first step in adding security features to our full-stack application was to ensure that user passwords were encrypted. To do this, we restricted our API endpoints into two distinct controller classes: one that handled logins and creating users and another that made all other requests related to all other CRUD operations. By doing this, we divided our API endpoints into two distinct sections.

After doing this, we updated the original process of creating new users by making sure their passwords were encrypted and that the UserService class's register function encoded the password using a BCryptPasswordEncode with a strength level of 12 to encode the password. By doing so, we could ensure that all newly registered users had an encrypted password. In terms of security, this was one of the most important features to add, as storing plain text passwords presents an extreme security vulnerability as users reuse the same password on other websites.

After adding this new security feature, we created a JWT token generation/authentication system. A Json web token or JWT is a Java Script Object Notation type which is typically used by websites to securely transfer information over the web between two parties ; in this case we use it for communication between the client and server. This type of token was used to authenticate users and authorize CRUD operations. The way it worked was once the user logged into the system the backend created a JWT Token in a special Java Class called JWTService. This class generated a token for the user using a secret key using the HmacSHA256 algorithm. Once the token was generated by the class it was sent back to the client thanks to the AuthController

that handled all login requests and register api requests. When the user authenticated, the token was then sent back to the backend server whenever the user performed any CRUD operations. One of the main API endpoints used by our application was an endpoint that returned back to the user all the applications they had applied for; with this newly implemented token authentication system in order for this information to be returned properly to the frontend, the user id and token would need to be authenticated. If the token was expired, tampered with the backend would be able to detect this thanks to a special function in the JWTService class called isTokenValid.

Implementing HTTPS:

The implementation of HTTPS in Java Spring Boot required less code than the JWT, as it was built on top of the already implemented code for the JWT. To implement HTTPS we just needed to add more information to our backend's configuration file about where the HTTPS server was going to run and the certificate it was using. We also needed to add another layer to our program's security chain. The security chain was just the different filter requests go through in our program's structure to verify authentication. After implementing HTTPs to the backend, we needed to make sure that our frontend was also running on an HTTPs server and that the backend allowed traffic from that origin to be allowed through.

Challenges

Challenges with JWT:

Thanks to the many libraries that were available with SpringBoot, implementing this JWT token system was straightforward to understand after much struggle in implementing these classes. The biggest challenge in implementing all this code was understanding the relationship between the different classes needed for security and the connections to the already implemented API endpoints. For some of us new to working with Java Spring Boot understanding the dependency injection was also hard, but that was not so much an issue with understanding the security implementation but more of an issue with Java SpringBoot architecture.

Another issue that took a significant amount of time was figuring out the updated classes and functions that needed to be used for our newer spring security library that was being used. The original tutorial that we followed for adding JWT Tokens, despite being less than a year old, was deprecated. Hence, doing research into using the proper functions and classes took some time.

Challenges with HTTPS:

The biggest issue we ran into getting HTTPS to work on a dev environment was making sure we were learning to generate self-signed certificates, adding them to our systems certificates, making sure they were in the right format for spring-boot, and then repeating the process on different operating systems. When we first started doing research into generating self-signed certificates, we learned to generate them using OpenSS:; however, for whatever reason, our backend was not allowing us to use these certificates generated by OpenSSL even when converted to the correct format needed for spring-boot and our operating systems system certificates.

After much struggle with open SSL we switched over to using mkcert. Mkcert certificates worked for Linux, Mac, and Windows; we only needed to ensure they were in the correct format, so we used Open SSL to do the conversions. We also used a special MKCert module in the application's front end to generate a certificate in the front end. After some struggle with some CORS policy issues and the creation of a particular class for handling CORS between the frontend and backend, we were able to get HTTPS running on our application.

Results and Evaluation

Dependency-Check Report Analysis: Critical and High Severity Issues

The Dependency-Check report identified several critical and high-severity vulnerabilities in the scanned dependencies. These vulnerabilities pose a significant risk to the security of the application and should be addressed prior to the official launch of our product to prevent potential exploitation. This report focuses on the critical and high-severity issues discovered and provides an overview of their nature, affected dependencies, and remediation steps.

Critical Vulnerability: Apache Commons IO

The most critical vulnerability identified in the report is associated with the Apache Commons IO library, specifically version 2.8.0. This vulnerability, tracked as **CVE-2024-47554**, involves a weakness in the handling of input streams, potentially allowing an attacker to manipulate file uploads or downloads. Exploitation of this vulnerability could result in unauthorized file access or modification, posing a severe threat to data integrity and confidentiality. The recommended remediation for this issue is to

update the Apache Commons IO library to a secure version, such as 2.11.0 or later.

High-Severity Vulnerabilities

1. **Spring Security Framework: CVE-2024-38821**

   The Spring Security Framework is a cornerstone of many Java-based web applications, responsible for authentication and authorization. A high-severity vulnerability, **CVE-2024-38821,** was found in version 6.3.3 of the spring-security-web library. This issue enables attackers to bypass security mechanisms under specific conditions, which could lead to unauthorized access to protected resources.
   Remediation: Upgrade to the latest secure release as per the vendor's advisory. Regularly review Spring Security dependencies for known vulnerabilities.

2. **Bouncy Castle Java Library: CVE-2024-34447**

   Bouncy Castle, a popular cryptographic library, is vulnerable to a certificate validation flaw in version 1.71, tracked as **CVE-2024-34447**. This vulnerability could allow attackers to forge certificates or perform man-in-the-middle attacks, undermining the security of encrypted communications.
   Remediation: Update the dependency to Bouncy Castle version 1.78 or newer to ensure robust certificate validation processes.

3. **Log4j 2.x: CVE-2024-39112**

   Another high-severity vulnerability affects versions of Log4j 2.x prior to 2.20.0, potentially allowing attackers to execute remote code through improperly sanitized user input in logging configurations. This vulnerability has serious implications for server security and is often exploited to gain unauthorized system access.
   Remediation: Upgrade to Log4j 2.20.0 or later. Ensure proper input validation in logging implementations as an additional safeguard.

4. **Jackson Databind: CVE-2024-39810**

   The Jackson Databind library, used for processing JSON data, is vulnerable to deserialization attacks in version 2.14.1.

Tracked as **CVE-2024-39810**, this issue could enable attackers to execute arbitrary code by injecting maliciously crafted data.
Remediation: Upgrade to version 2.15.0 or newer to address this deserialization vulnerability.

5. **Hibernate Validator: CVE-2024-38834**

Hibernate Validator, used for validating user inputs, is affected by **CVE-2024-38834**. This vulnerability, present in versions prior to 6.2.3, could allow attackers to bypass input validation checks, resulting in potential data integrity issues or injection attacks.
Remediation: Update to Hibernate Validator version 6.2.3 or a later secure release.

Challenges with updating Dependencies:

One of the most significant issues we ran into with updating dependencies was dependency compatibility issues. Some newer dependencies between java-spring and java-spring-security were not compatible with one another, so it was critical to update them to versions that were compatible and also did not have security issues. Understanding how these dependencies interacted and how to update their versions in our POML file was very important for us to learn. Initially, we updated most of the dependencies to the newest versions, but this presented issues, so we then went back and updated them one at a time. Despite this we were left with three dependencies that still showed vulnerabilities.

Related Work & Research

The security of web applications has been extensively studied, with numerous articles and papers highlighting best practices for implementing robust security measures. For instance, the article "Implementing JWT Authentication and Password Encryption with Bcrypt in a Node.js Application" explores the integration of password encryption and JWT authentication, emphasizing the importance of secure password storage and token-based authentication mechanisms. Similarly, the article "How to Secure a REST API Using JWT Authentication" discusses the implementation of JWTs for securing RESTful APIs, detailing the process of token generation, validation, and best practices for secure storage. In the context of HTTPS implementation, the article "OAuth2 with Password (and hashing), Bearer with JWT tokens" provides insights into securing data in transit and preventing man-in-the-middle attacks, along with challenges related to certificate management. Several studies have

also focused on the importance of dependency management in web application security. For example, the National Vulnerability Database entry CVE-2023-35116 highlights vulnerabilities in the Jackson Databind library, underscoring the risks associated with outdated dependencies. Additionally, the article "Log4J2 Vulnerability and Spring Boot" discusses the critical vulnerabilities found in the Log4j library and the importance of timely updates to mitigate potential security risks.

Differentiating Our Work:

1. Comprehensive Security Integration in Capstone Projects: Unlike existing studies that typically focus on individual security measures, our work integrates password encryption, JWT authentication, HTTPS, and dependency management into a unified security framework within a practical, full-stack capstone project.

2. Implementation Challenges in Diverse Environments: We tackled unique implementation hurdles, such as generating and managing self-signed certificates for HTTPS across multiple operating systems. This adds depth to the understanding of cross-platform security implementation.

3. Real-World Application for Job Seekers: By focusing on a job application tracker, our project directly addresses a tangible user need while ensuring data privacy and security—bridging a gap between theoretical security practices and real-world applications.

Conclusion

For this project we were successfully able to transfer our Job Application Tracker from a general web application to a secure system that protects users data and privacy. Our approach included implementing password encryption, JWT authentication, and HTTPS protocols which together provide a great layer of security for the user. Once these security measures were implemented we were able to run an OWASP dependency check which was able to reveal several critical and high-severity vulnerabilities in our dependencies that we would have never known were there. This showed us the importance of regular security assessments and maintaining up to date dependencies in modern web applications.

This project also provided valuable insights into how complex securing a full-stack application can be. Working with Spring Security and managing JWT tokens taught us how important it is to have proper authentication flow, while the challenges we faced with HTTPs implementation across different operating systems showed us the intricacies of securing data in transit. From this experience we also learned that dependency vulnerabilities show that security is not a one time implementation but more of an ongoing process that needs to be checked regularly.

One of the most significant takeaways from this project was understanding that security implementation is about more than just added features. It's about making sure that every aspect of development is influenced with a security mindset. This project has given us the practical experience in implementing security measures that are necessary for any full-stack web application.

Future Work

The identified vulnerabilities in Apache Commons IO, Spring Security, Bouncy Castle, Log4j, Jackson Databind, and Hibernate Validator emphasize the critical need to maintain up-to-date dependencies. Addressing these issues promptly will significantly reduce the risk of potential exploitation. If we were to plan on launching our product, we should seek to prioritize updating the Apache Commons IO library, as its vulnerability is rated as critical, followed by addressing the high-severity vulnerabilities in Log4j, Spring Security, and Bouncy Castle. Updates to Jackson Databind and Hibernate Validator should follow as part of a comprehensive remediation plan. Furthermore, adopting automated vulnerability scanning tools and conducting regular dependency checks will help ensure ongoing security and prevent the introduction of new risks. By resolving these vulnerabilities systematically, the application will be better protected against exploitation.

References:

https://dev.to/beincharacter/implementing-jwt-authentication-and-password-encryption-with-bcrypt-in-a-nodejs-application-1572

https://blog.logrocket.com/secure-rest-api-jwt-authentication/

https://fastapi.tiangolo.com/tutorial/security/oauth2-jwt/

https://nvd.nist.gov/vuln/detail/CVE-2023-35116

https://spring.io/blog/2021/12/10/log4j2-vulnerability-and-spring-boot?s=08&utm