

Title:

“Develop and test a secure software application”

Program: BSc in Computing with Foundation (Cybersecurity)

Course: Secure Application Development (DEV6003)

Student name: Thomais Paraskevaidi

Student Code: 2119593

Word Count: 1497

Contents

[Introduction 2](#_Toc165249800)

[Main Body 3](#_Toc165249801)

[Installation - Instructions 3](#_Toc165249802)

[Screenshots from the scan 4](#_Toc165249803)

[Fixing the warnings 7](#_Toc165249804)

[Conclusion 9](#_Toc165249805)

[References 10](#_Toc165249806)

# Introduction

The annual increases in the quantity of application deployments is accompanied by a corresponding increase in the complexity of their interfaces, resulting in heightened difficulties in the administration of their security. As these applications gain popularity, they also become more susceptible to exploitation by malicious entities. The presence of undiscovered code vulnerabilities that elude early detection in the initial deployment stages due to limitations of conventional hardware is a significant source of concern. For organizations to adequately safeguard their assets, they must possess a thorough comprehension of the essential security tools and demonstrate proficiency in seamlessly integrating these tools into their intricate security frameworks. Regrettably, a substantial level of manual intervention is still necessary for numerous of these technologies. Conversely, current trends are leaning towards greater utilisation of automation in security protocols. When new code is submitted, it activates the automation of repetitive tasks, such as clicking and entering data, using tools like Selenium. In addition to the existing problem, advanced Continuous Integration (CI) tools like Jenkins enable the automation of the entire development cycle. Once the developer has submitted code to a Git repository, the continuous integration system will automatically initiate the process by creating a temporary container, compiling the application, and verifying the success of the compilation. If any errors occur, the developer is immediately notified, allowing for timely correction and recommitment of the code.

# 2.Installation

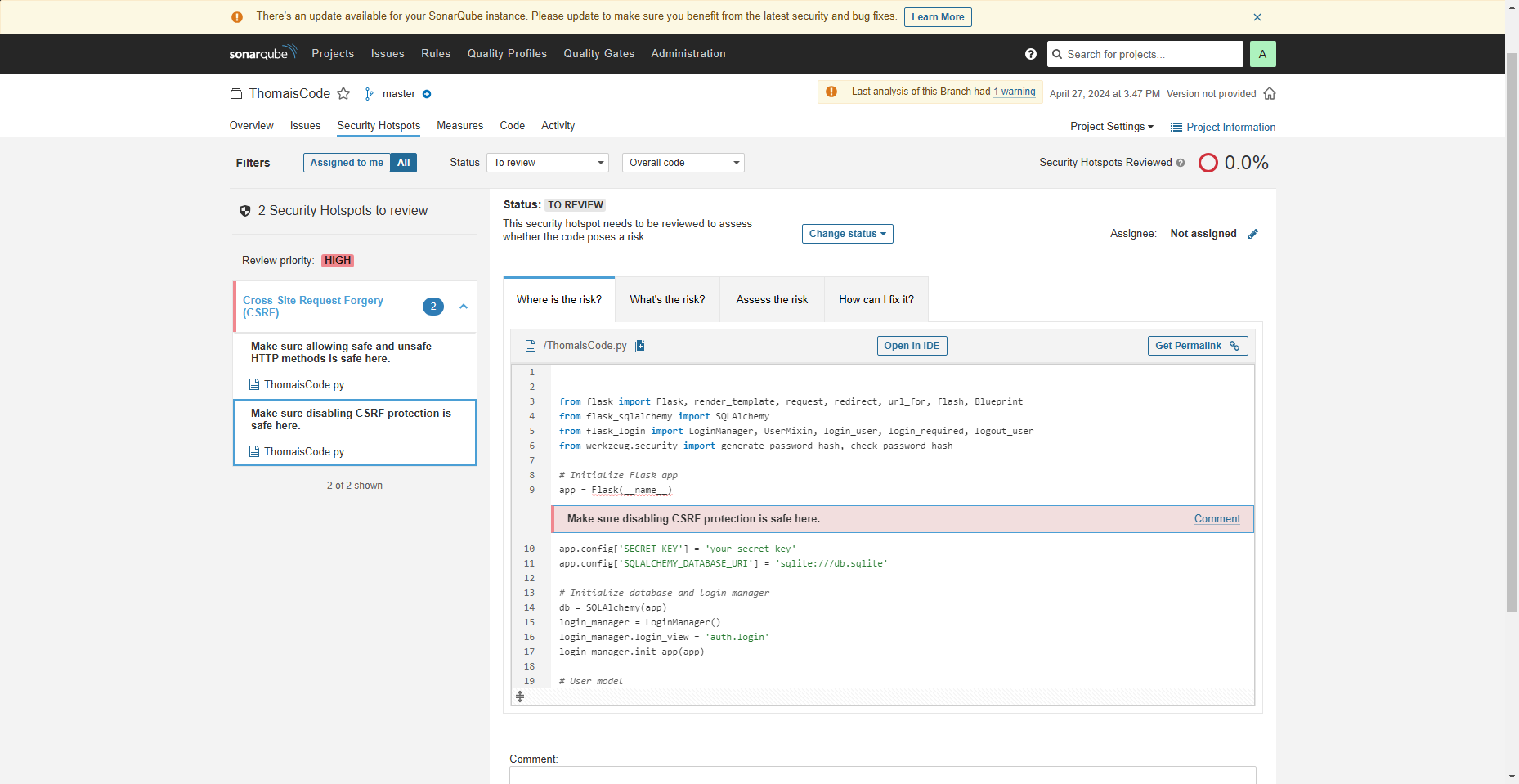
## To complete this assignment, you must install a specific software application to perform a code quality inspection on a software project. The inspection will search for bugs, including code issues, vulnerabilities, security issues. For the purpose of this assignment, we will only be using SonarQube, a widely recognised software often used in conjunction with Jenkins for continuous integration and continuous delivery (CI/CD). Our focus will be on implementing static code analysis on a local project. Since SonarQube will be run as a containerised application, Docker is a crucial component for conducting the tests. Once we have successfully replicated the image from the official website, our container will be operational on the localhost port 9000. Consequently, we will be able to conveniently access the interface via our web browser. After the successful compilation of our code, we will establish communication with SonarQube to carry out our tests. After creating a new project and choosing the option to create a local project, we will proceed to generate a token, establish a build tool, and install Sonarscanner. During the subsequent step, the page will undergo a refresh and present the outcomes of the scan, revealing any vulnerabilities or code issues that were identified in our endeavour.

## Material from Sonar

(1) A screenshot of a computer

Description automatically generated

(2)



(3)

A screenshot of a computer

Description automatically generated

The results where:

1. **Security Hotspots**: The first image makes it quite evident which two security concerns need to be reviewed immediately. These areas are connected to cross-site request forgery, or CSRF, vulnerabilities. Two problems have been found: the first concerns using both safe and unsafe HTTP methods; the second concerns turning off CSRF protection.
2. **Code View**: Additionally, there is a code segment in the first picture. This code is part of a Flask-using Python application, as decorators like @auth\_blueprint.route show. A login function handling both GET and POST requests is shown when enabled. The user logs in if the function successfully authenticates them by comparing their username and password to a database. Should authentication fail, the feature will send the user back to the homepage.
3. **Quality Gate Status**: The overall state of the codebase is shown in the second image as the Quality Gate Status. The code is rated as "Passed" by SonarQube, meaning that it satisfies the requirements for quality established specifically for this project. On the other hand, there is a section that is red, indicating that just 0.0 percent of the Security Hotspots have been examined.
4. **Quality Metrics**: The second image also includes several quality metrics:   
   There is a complete absence of bugs or vulnerabilities.   
     
   Two security hotspots necessitate examination.

* No debt
* No code smells
* All three ratings for maintainability, reliability, and security are assigned a 'A' rating, which is typically the highest possible rating.  
  Both the unit test coverage and duplication blocks metrics are currently at 0.0%. This implies that there might be a deficiency in unit tests or code coverage analysis.

# 3.Solution

The warning regarding "allowing safe and unsafe HTTP methods" does not explicitly refer to HTTPS. Nevertheless, it is imperative to guarantee that the application exclusively permits HTTPS requests, especially for hazardous HTTP methods that have the potential to alter the system's state (POST, PUT, DELETE).

1. **Enforcement of HTTPS**:
   * Its setup guarantees automatic redirection to the HTTPS version when accessed via HTTP by redirecting all HTTP traffic to HTTPS.
   * Set up on the server, HTTP Strict Transport Security (HSTS) forbids SSL stripping attacks by telling browsers via an HTTP header to use HTTPS only for a predetermined amount of time.
2. **Cookie Security**:
   * Cookies with the Secure label will only be transmitted over HTTPS connections used by the application.
   * Setting the HTTPOnly flag on cookies is thought to reduce the risk of XSS attacks by preventing client-side script access.
3. **Forms and API Calls**:
   * Updated forms guarantee that they post to HTTPS endpoints.
4. **SSL/TLS Certificate**:
   * If a valid SSL/TLS certificate is not already in possession, one may obtain one for free using services like Let's Encrypt.
   * TLS/SSL certificates are maintained current by routine renewals as needed.
5. **Backend Configuration**:
   * Application backends are set up to only accept HTTPS requests for sensitive operations or routes.
   * HTTPS enforced in development environments like Flask is done by middleware or configuration that confirms the scheme of incoming requests.
6. **Content Security Policy (CSP)**:
   * There is an implemented Content Security Policy that allows content to be loaded only over HTTPS.

The Implementation:

1. First, install Flask-WTF if you haven’t already:

A black and white screen with white text

Description automatically generated

1. Then, in your Flask application, enable CSRF protection:

A screenshot of a computer

Description automatically generated

1. In your forms, you'll need to include the CSRF token:

A screenshot of a computer program

Description automatically generated

1. For AJAX requests, send the CSRF token within the request header:

A computer screen shot of a black background

Description automatically generated

1. **Enforcing HTTPS**: To redirect HTTP traffic to HTTPS and to ensure your Flask application always uses HTTPS, you can do the following:

Set up a middleware to redirect HTTP to HTTPS in production. For local development, it's usually okay to use HTTP:

A computer screen shot of a program

Description automatically generated

1. In addition, you can enforce secure cookies by setting the **SESSION\_COOKIE\_SECURE** flag to **True**:



1. Lastly, set the **Strict-Transport-Security** header to enforce HSTS:

A computer screen with white text

Description automatically generated

# Conclusion

An important part of the continuous work to improve software security and lessen the threat of cyberattacks is secure code analysis. Creating safe applications requires a number of steps, one of which being a thorough code analysis with both automated and human inspection. Organisations may greatly reduce the likelihood that hackers will be able to successfully exploit vulnerabilities by regularly implementing robust coding techniques at every phase of the development process. This assignment stresses the need of protecting digital assets by doing an analysis and ongoing education in secure coding standards. Adopting secure coding techniques has to be given top priority because software development is becoming more and more necessary. Taking care of this need is a duty to stakeholders and users as well as a technological need.

# References

1. **Jammeh, B. (2020)** 'DevSecOps: Security Expertise a Key to Automated Testing in CI/CD Pipeline', *Bournemouth University*, December.
2. **Fernández González, D., Rodríguez Lera, F. J., Esteban, G., & Fernández Llamas, C. (2021)** 'SecDocker: Hardening the Continuous Integration Workflow', *SN Computer Science*, 3(80). DOI: 10.1007/s42979-021-00939-4.
3. **Chandramouli, R., Kautz, F., and Torres-Arias, S. (2024)** 'Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD Pipelines', *NIST Special Publication (SP) NIST SP 800-204D*. National Institute of Standards and Technology, Gaithersburg, MD.
4. **Dakic, V., Redzepagic, J., and Basic, M. (2022)** 'CI/CD Toolset Security', *Proceedings of the 33rd DAAAM International Symposium on Intelligent Manufacturing and Automation*. DAAAM International, Vienna, Austria. DOI: 10.2507/33rd.daaam.proceedings.xxx.
5. **Pan, Z. et al. (2024)** 'Ambush From All Sides: Understanding Security Threats in Open-Source Software CI/CD Pipelines', *IEEE Transactions on Dependable and Secure Computing*, 21(1). https://doi.org/10.1109/TDSC.2023.xxxxx (Accessed: 30 April 2024).
6. **Abiola, O. and Olufemi, O. G. (2023)** 'An Enhanced CICD Pipeline: A DevSecOps Approach', *ResearchGate*.
7. **Smedinga, R. and Biehl, M. (eds.) (2020)** '17th SC@RUG 2020 proceedings 2019-2020', *University of Groningen*.
8. **Jammeh, B. (2020)** 'DevSecOps: Security Expertise a Key to Automated Testing in CI/CD Pipeline', *ResearchGate*.
9. **Fernández González, D. et al. (2022)** 'SecDocker: Hardening the Continuous Integration Workflow', *SN Computer Science*, 3(80).
10. **Hussain, N. (2021)** 'Gaps and Improvements in Secure Development – In Practice', *MSc Project Report*, National College of Ireland.
11. **Putra, A.M. and Kabetta, H. (Date Unknown)** 'Implementation of DevSecOps by Integrating Static and Dynamic Security Testing in CI/CD Pipelines', *IEEE Xplore*.