Dear Intern

Interim project report is an inherent component of your internship. We are enclosing a reference table of content for the interim project report.

The key objective of this report is for you to capture how far you have got in completing the internship work against milestones expected to be achieved within a specific duration and seek the mentor’s feedback. Depending on the internship project and your progress (IT/Non-IT, Technical/Business Domain), you may choose to include or exclude or rename sections or leave some sections blank from the table of content mentioned below. You can also add additional sections. You can refer the project presentation to view the milestones related to your internship project. Please populate milestone# (1 / 2 / 3) and the milestone description in the interim project report based on the milestone for which you are submitting the interim project report.

You can refer the project presentation to view the milestones related to your internship project.

|  |  |
| --- | --- |
| Internship Project Title | TCS iON RIO-125: Application of Static Application Security Testing (SAST) Tools – Find Defects in Insecure Web-based Applications |
| Name of the Company | TCS iON |
| Name of the Industry Mentor | Uma Devi |
| Name of the Institute | Government College of Engineering, Bodinayakkanur, Theni-625582 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Start Date | End Date | | Total Effort (hrs.) | | Project Environment | Tools used |
| 21/04/2023 | 15/07/2023 | | 125 | | Kali Linux, Windows, VM BOX Maven, Java-11 | SonarQube |
| Milestone # |  | Milestone: | |  | | |

**TABLE OF CONTENT**

1. **Acknowledgements**

I am highly indebted to TCS iON for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project. I would like to express my gratitude towards my parents and my academic mentor, for their king co-operation and encouragement which help me in completion of this project. I would like to express my special gratitude and thanks to my industry mentor for giving me such attention and time.

1. **Objective**

This project is to explore and utilize Static Application Security Testing (SAST) tools to detect and analyze defects in insecure web-based applications, ultimately enhancing their security and mitigating potential vulnerabilities.

1. **Introduction / Description of Internship**

An application of Static Application Security Testing (SAST) tools to identify defects in insecure web-based applications. SAST tools play a crucial role in enhancing the security of web applications by analyzing source code or binaries for potential vulnerabilities. This project aims to explore and evaluate different SAST tools and apply them to analyze a set of insecure web-based applications, ultimately identifying and documenting security defects.

1. **Internship Activities**
2. **Research and Evaluation:** Conduct thorough research to identify and evaluate various SAST tools available in the market for detecting defects in insecure web-based applications. Compare their features, capabilities, and effectiveness in identifying security vulnerabilities.
3. **Tool Selection and Setup**: Select the most suitable SAST tool based on the research findings. Set up the chosen tool, configure it according to project requirements, and ensure compatibility with the target web applications.
4. **Application Analysis**: Apply the selected SAST tool to analyze a set of insecure web-based applications. Run the tool's scanning capabilities on the application's source code or binary to identify potential defects, such as insecure coding practices, vulnerabilities, or loopholes in the application's security implementation.
5. **Approach / Methodology**

The following approach/methodology will be followed in the application of Static Application Security Testing (SAST) tools to find defects in insecure web-based applications:

1. **Project Scoping**: Define the scope of the project, including the specific web-based applications to be analyzed and the security defects to be targeted.
2. **Tool Selection**: Conduct a thorough evaluation of different SAST tools available in the market. Consider factors such as compatibility with the programming languages used in the web applications, support for identifying common vulnerabilities, and ease of use.
3. **Tool Setup and Configuration**: Install and configure the selected SAST tool according to the project requirements. Configure the tool to scan the source code or binary of the web applications under analysis.
4. **Application Scanning**: Run the SAST tool on the web applications to identify potential security defects. The tool will perform static analysis of the code, searching for vulnerabilities such as injection attacks, cross-site scripting (XSS), or insecure data storage.
5. **Defect Identification and Documentation**: Analyze the results generated by the SAST tool and identify the detected defects. Document the identified security vulnerabilities, providing a clear description of each issue, its severity, and possible remediation steps.
6. **Remediation and Verification**: Collaborate with the development team to address the identified defects. Implement necessary security fixes and conduct verification tests to ensure the successful resolution of the vulnerabilities.
7. **Reporting**: Generate a comprehensive report summarizing the findings, including the identified defects, their impact, and the actions taken to mitigate them. Present the report to stakeholders, highlighting the importance of SAST tools in identifying and addressing security flaws in web-based applications.
8. **Assumptions**

The application of Static Application Security Testing (SAST) tools to find defects in insecure web-based applications:

1. **Access to Source Code**: It is assumed that the project team has access to the source code or binary of the web-based applications under analysis. Without access to the code, the SAST tools cannot perform static analysis and identify security defects.
2. **Compatibility with SAST Tools**: The selected SAST tools are assumed to be compatible with the programming languages and frameworks used in the web applications. Compatibility issues may arise if the tools do not support the specific technologies employed in the applications.
3. **Coverage of Security Defects**: The SAST tools are assumed to be effective in identifying a wide range of security defects commonly found in web-based applications, such as injection attacks, XSS, or insecure data storage. However, it is important to note that no tool can guarantee 100% coverage of all possible vulnerabilities.
4. **Development Team Collaboration**: It is assumed that there will be active collaboration between the project team and the development team responsible for the web applications. This collaboration is crucial for understanding the context of the applications, implementing security fixes, and verifying the resolution of identified defects.
5. **Continuous Improvement**: The project assumes a commitment to continuous improvement in the security posture of the web applications. The findings from the SAST tool analysis will be used as a basis for implementing security enhancements and adopting best practices to prevent future security issues.

These assumptions provide a foundation for the successful application of SAST tools in identifying defects in insecure web-based applications

1. **Exceptions / Exclusions**

The following exceptions or exclusions apply to the application of Static Application Security Testing (SAST) tools in finding defects in insecure web-based applications:

1. **Dynamic Vulnerability Testing**: SAST tools focus on static analysis of the source code or binary of web applications. They do not perform dynamic vulnerability testing, which involves analyzing the application's behavior during runtime. Dynamic testing is typically performed using other techniques such as penetration testing or web application scanners.
2. **False Positives and False Negatives**: SAST tools may generate false positives (identifying an issue as a security defect when it is not) or false negatives (failing to identify an actual security defect). The project team should be prepared to review and validate the results generated by the SAST tools to minimize the impact of false findings.
3. **Limited Scope of SAST Tools**: SAST tools primarily focus on identifying vulnerabilities related to coding practices, such as insecure input validation or improper error handling. They may not detect certain vulnerabilities that require a deeper understanding of the application's business logic or specific deployment configurations.
4. **Third-Party Code**: SAST tools may not have visibility into third-party libraries or components used within the web applications. Security vulnerabilities within these external dependencies may not be identified through static analysis alone. Additional measures, such as vulnerability scanning of third-party components, should be considered.
5. **Compliance and Legal Considerations**: While SAST tools aid in identifying security defects, they may not directly address compliance requirements or legal obligations specific to the web applications. Compliance-related testing and assessments should be performed separately to ensure adherence to relevant regulations or industry standards.
6. **Limitations of SAST Tools**: It is important to acknowledge the limitations of SAST tools, such as potential false negatives, inability to detect complex vulnerabilities, and dependence on the quality of the source code. Therefore, additional security measures, such as code reviews, secure coding practices, and other security testing techniques, should be considered in conjunction with SAST tools.
7. **Charts, Table, Diagrams**

**Project Workflow:**

***Static Application security testing:***

***Charts:***

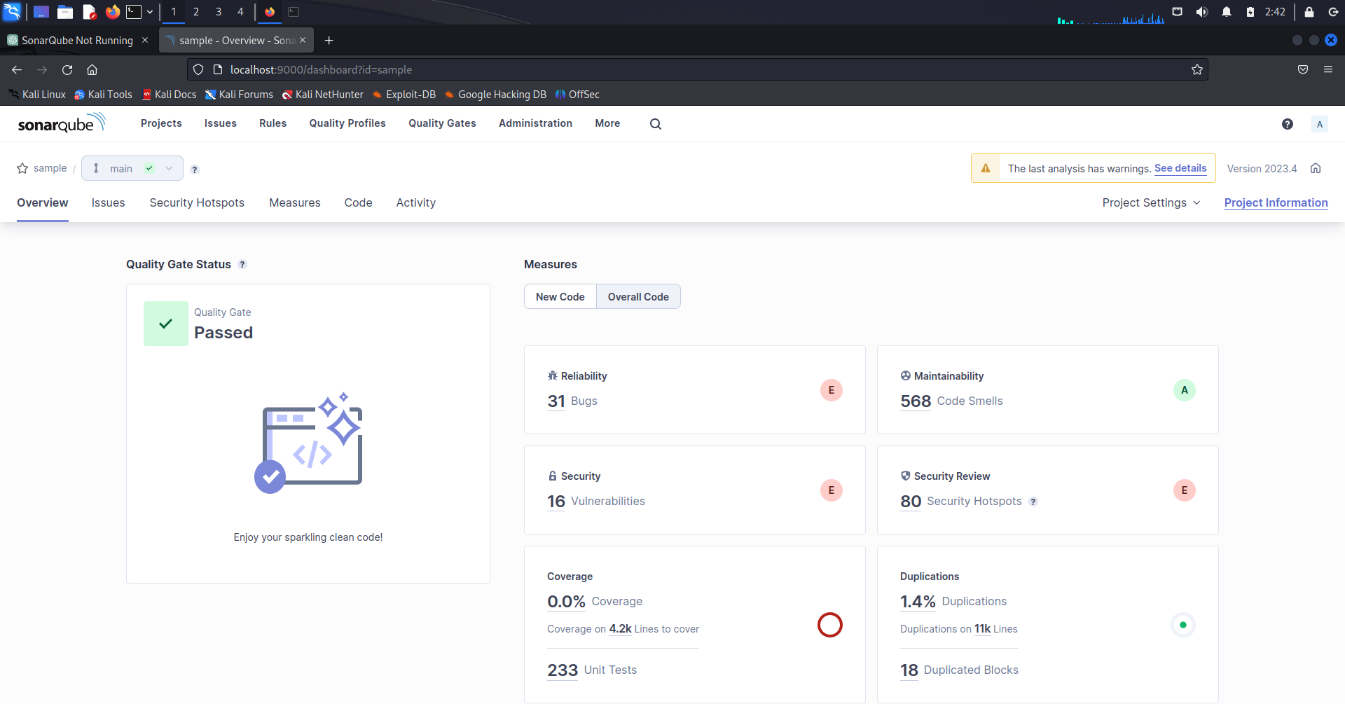
***Table:***

***WEB GOAT – 2023 APPLICATION STATIC TESTING RESULTS:***

|  |  |  |
| --- | --- | --- |
| *Testing* |  | *Results* |
| *Bugs* | ***-*** | *31 %* |
| *Vulnerabilities* | ***-*** | *16 %* |
| *Hotspot reviewed* | ***-*** | *1.3 %* |
| *Duplications* | ***-*** | *1.4 %* |
| *Coverage* | ***-*** | *0.0 %* |
| *Code Smells* | ***-*** | *568* |
| *Total Lines of Code* | ***-*** | *11,000 (Java, XML)* |
|  |  |  |

***Diagram:***

* Scanned an insecure application named “WebGoat-2023”. The scan result contains.
* 31 Bugs
* 16 Vulnerabilities
* 80 Security Hotspot
* 568 Code Smells
* 1.4% Duplications



* Some Critical issues are found:
* **Bug.**

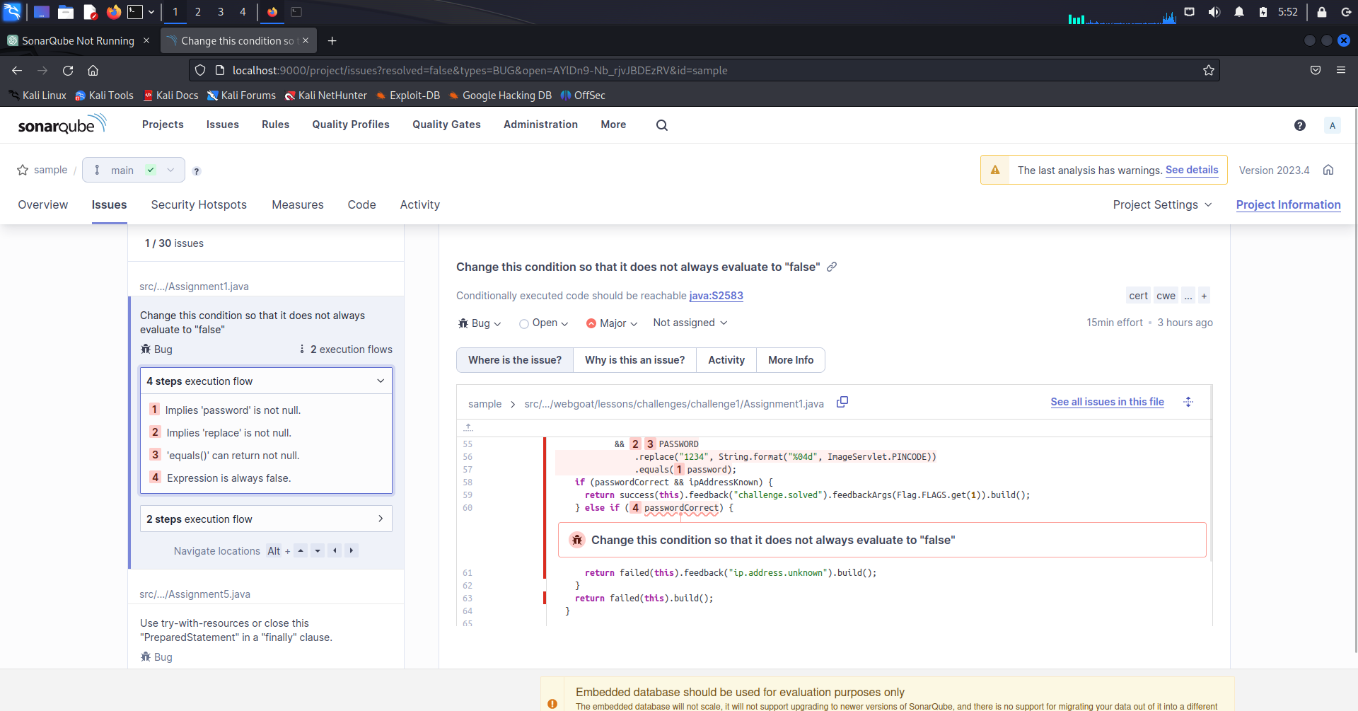
**Change this condition so that it does not always evaluate too "false"**

**File Name:** Make sure with Assignment1.java

**Description:** Conditional expressions which are always true or false can lead to dead code. Such code is always buggy and should never be used in production.

**Explanation with Code snippets:**

The src/main/java/org/owasp/webgoat/lessons/challenges/challenge1/Assignment1.java



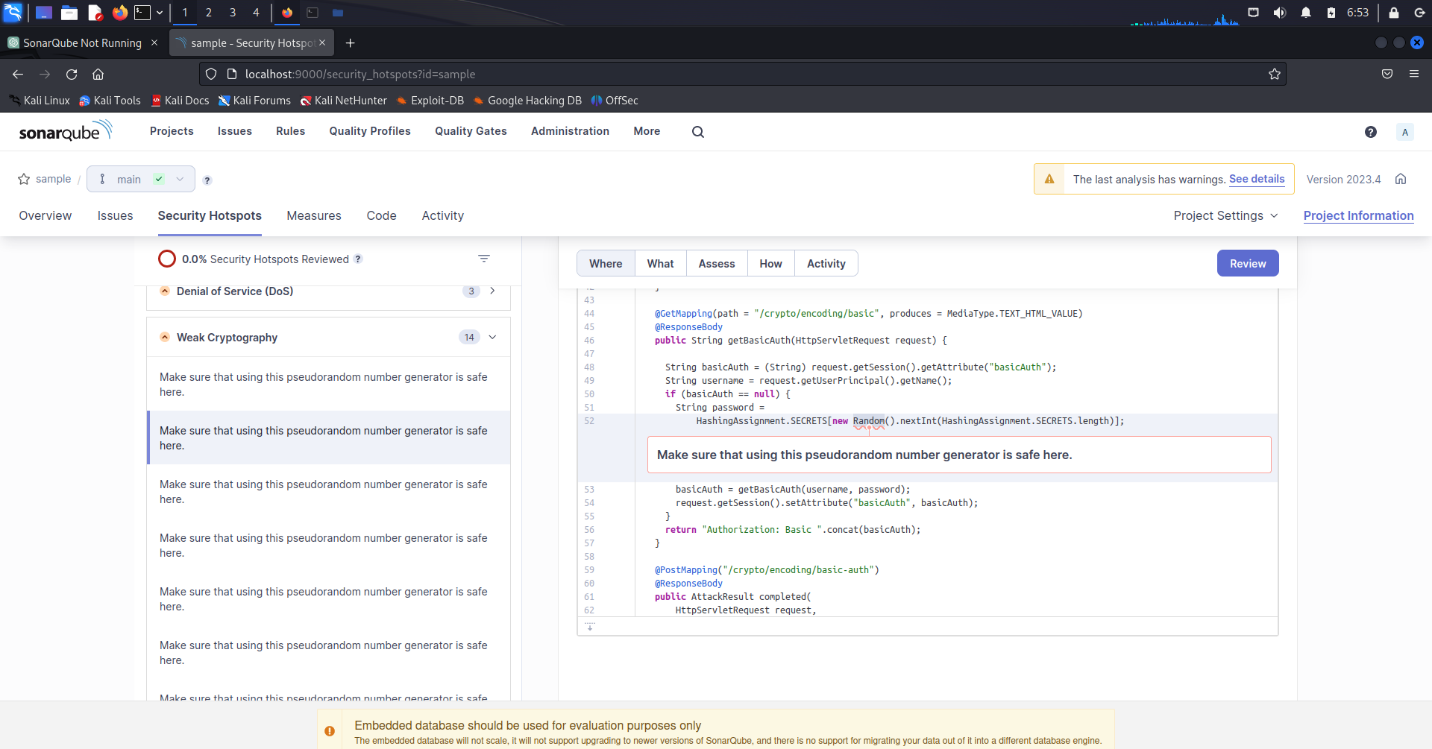
* "Random" objects should be reused (Security Misconfiguration)

**File Name:** Make Sure this file name for cryptography/EncodingAssignment.java

**Description:** [OWASP Top 10 2017 Category A6](https://owasp.org/www-project-top-ten/2017/A6_2017-Security_Misconfiguration) - Security Misconfiguration

**Explanation with Code snippets:**

src/main/java/org/owasp/webgoat/lessons/cryptography/EncodingAssignment.java



* **Vulnerabilities** 
  + - Don't use the default "PasswordEncoder" relying on plain-text.

**File Name:** Make Sure this file name for WebSecurityConfig.java

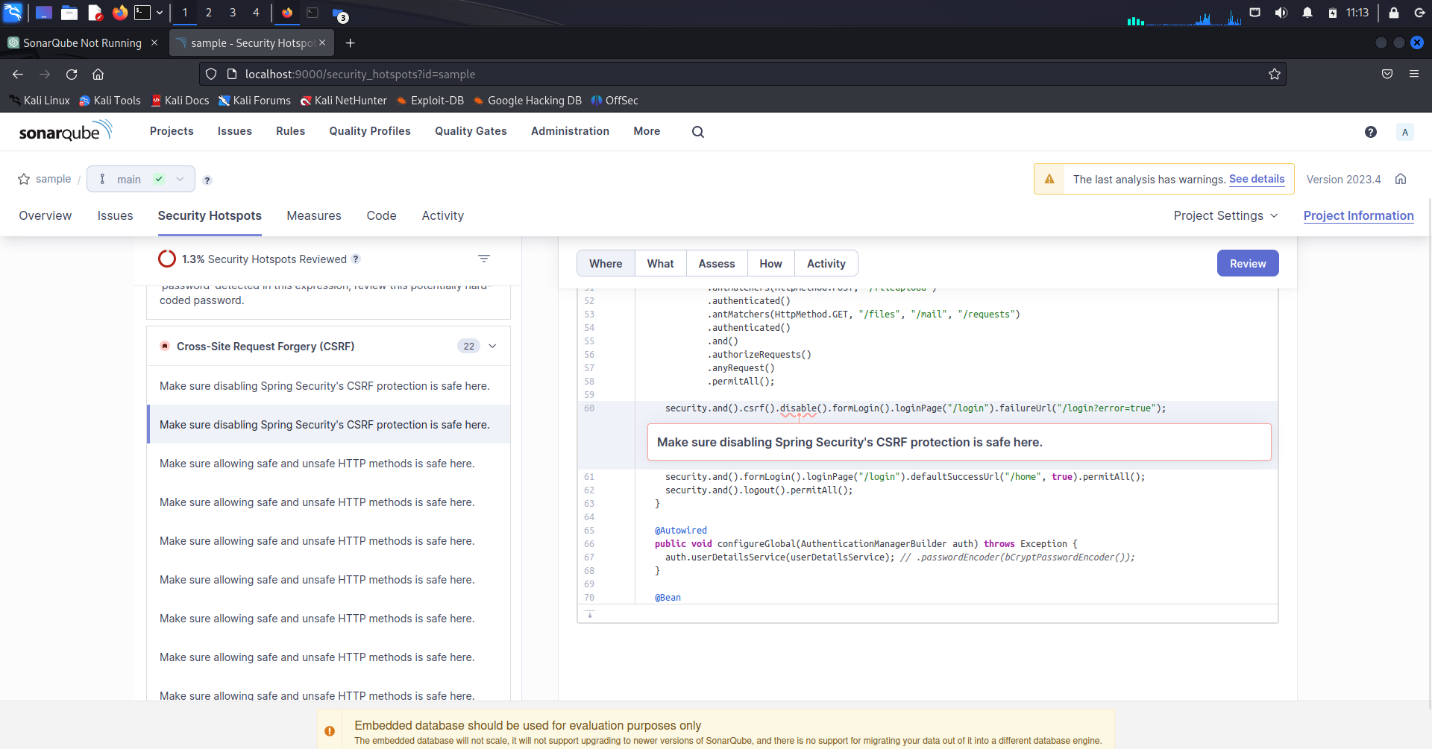
**Description:** A user password should never be stored in clear-text, instead a hash should be produced from it using a secure algorithm:

* not vulnerable to brute force attacks.
* not vulnerable to collision attacks (see rule s4790).
* and a salt should be added to the password to lower the risk of rainbow table attacks (see rule s2053).

This rule raises an issue when a password is stored in clear-text or with a hash algorithm vulnerable to bruce force attacks. These algorithms, like md5 or SHA-family functions are fast to compute the hash and therefore brute force attacks are possible (it’s easier to exhaust the entire space of all possible passwords) especially with hardware like GPU, FPGA or ASIC. Modern password hashing algorithms such as bcrypt, PBKDF2 or argon2 are recommended.

**Explanation with Code snippets:**

src/main/java/org/owasp/webgoat/webwolf/WebSecurityConfig.java



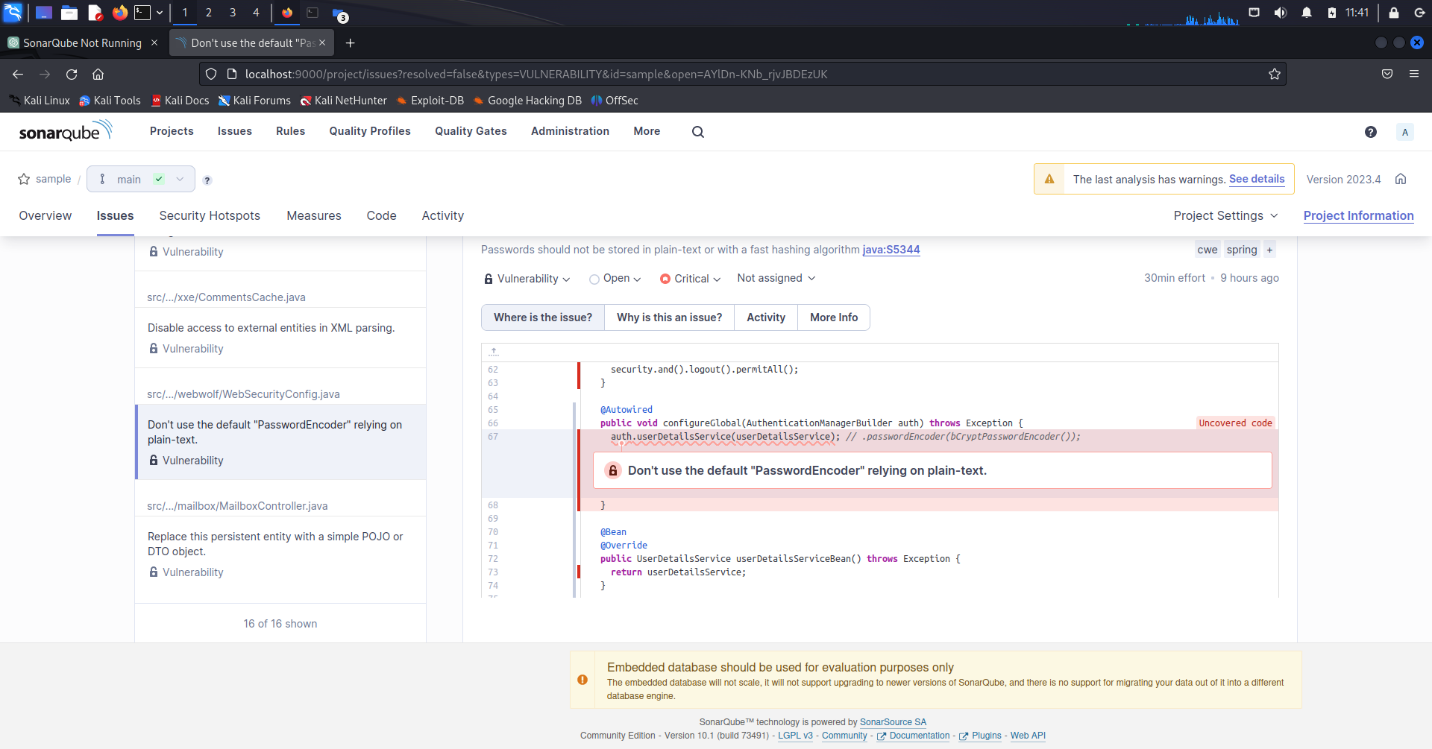
* Don't use the default "PasswordEncoder" relying on plain-text.

**File Name:** Make Sure this file name for **WebSecurityConfig.java**

**Description:**  Passwords should not be stored in plain-text or with a fast-hashing algorithm

**Explanation with Code snippets:**

src/main/java/org/owasp/webgoat/webwolf/WebSecurityConfig.java



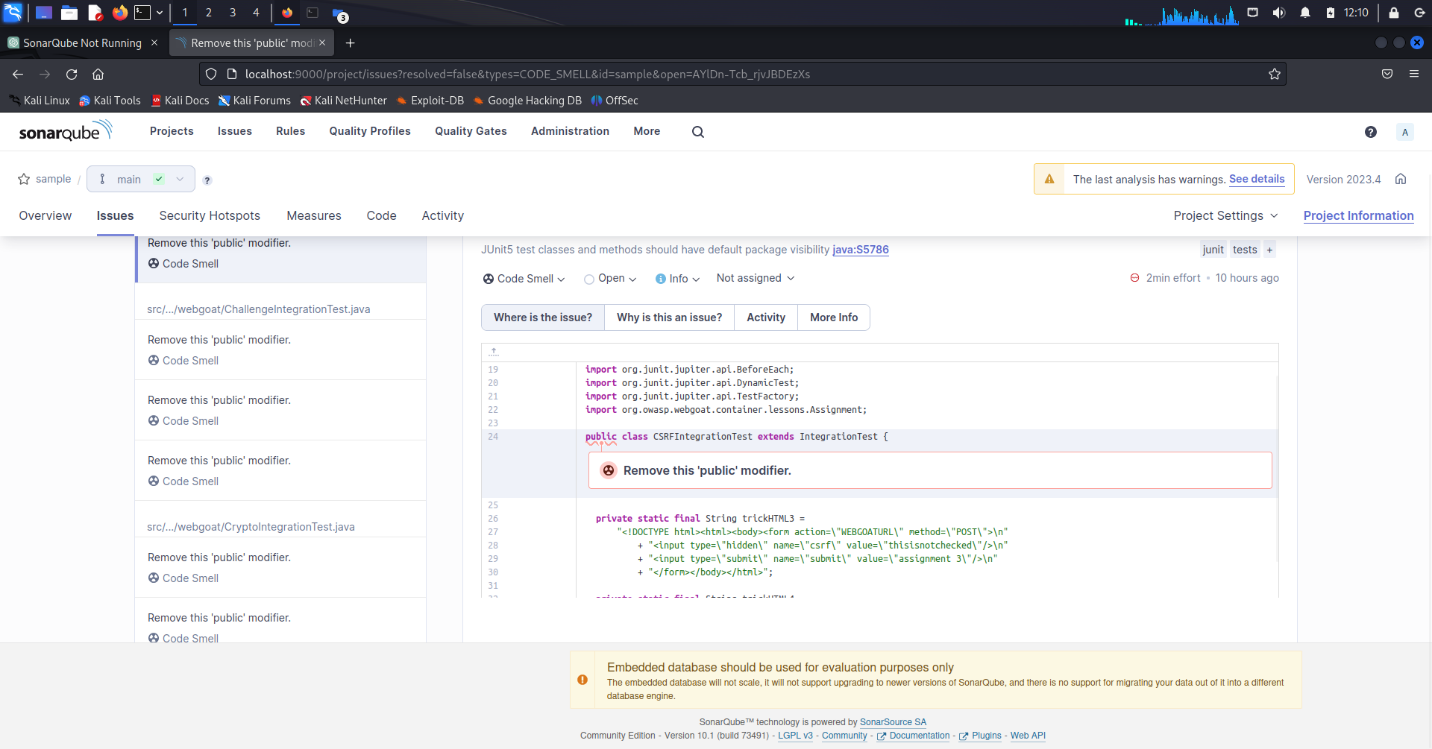
* **Code Smells**
* Remove this 'public' modifier.

**File Name:** Make Sure this file name for CSRFIntegrationTest.java

**Description:**  JUnit5 is more tolerant regarding the visibilities of Test classes than JUnit4, which required everything to be public.

**Explanation with Code snippets:**

src/it/java/org/owasp/webgoat/CSRFIntegrationTest.java



* Define a constant instead of duplicating this literal "secQuestion0" 7 times.

**File Name:** Make Sure this file name for AccountVerificationHelper.java

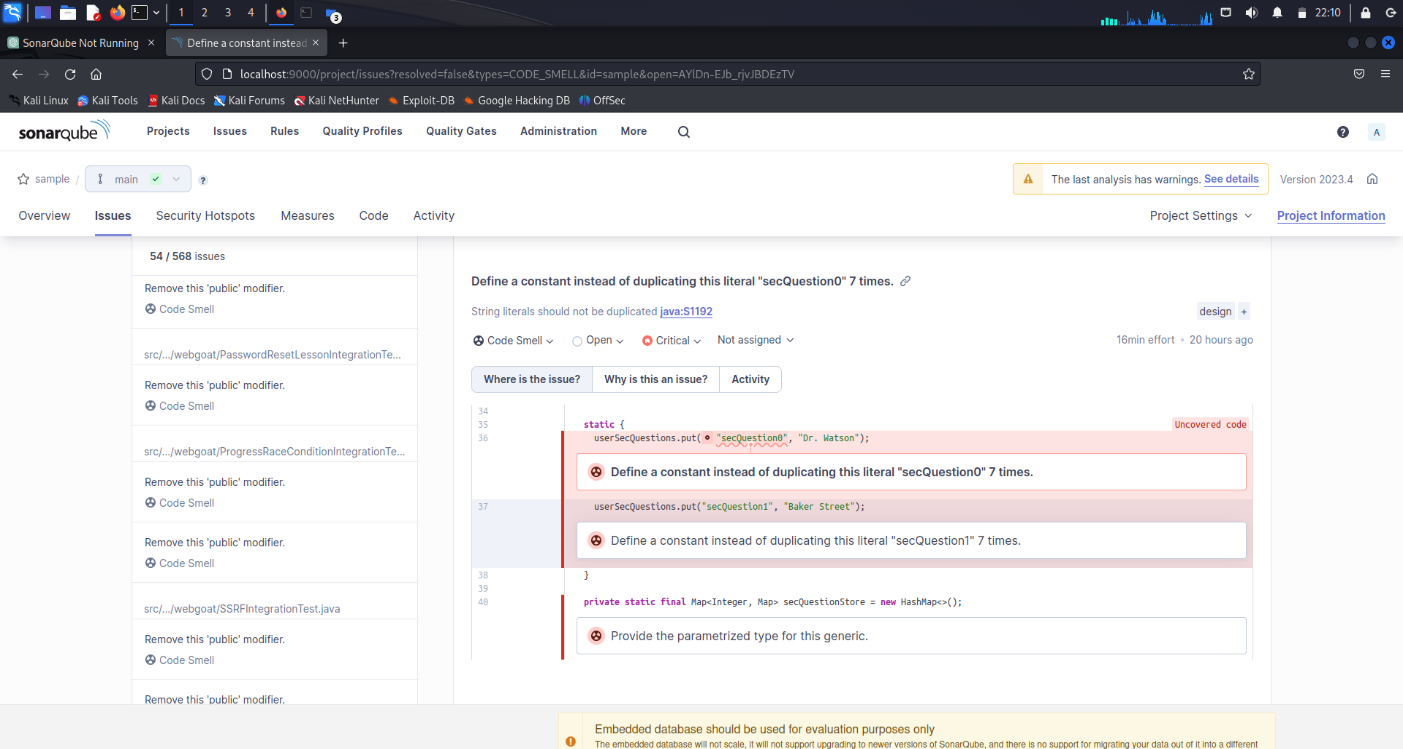
**Description:**

Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.

On the other hand, constants can be referenced from many places, but only need to be updated in a single place.

**Explanation with Code snippets:**

src/main/java/org/owasp/webgoat/lessons/authbypass/AccountVerificationHelper.java



1. **Algorithm:**

Static application security testing (SAST) tools, various algorithms and techniques can be applied to find defects in insecure web-based applications.

1. **Pattern Matching**: These patterns can include common security vulnerabilities like SQL injection, cross-site scripting (XSS), command injection, and more. By matching these patterns, the tool can identify potential security flaws in the code.
2. **Data Flow Analysis**: Data flow analysis tracks the flow of data within the application to identify potential security vulnerabilities.
3. **Taint Analysis**: Taint analysis is a specific type of data flow analysis that tracks tainted (potentially untrusted) data throughout the application.
4. **Control Flow Analysis**: Control flow analysis studies the order and structure of program statements to understand the possible paths of execution.
5. **Symbolic Execution**: Symbolic execution is a technique where variables and program inputs are treated symbolically rather than with concrete values.
6. **Code Review and Rule-Based Analysis**: SAST tools often incorporate rule-based analysis, where a set of predefined coding rules and security guidelines are enforced.
7. **Challenges:**
8. **False Positives and False Negatives**: SAST tools may generate false positives, flagging code segments as potential security vulnerabilities when they are not. This can result in wasted time and effort investigating and remediating non-existent issues.
9. **Handling Frameworks and Libraries**: Web applications often rely on frameworks, libraries, and third-party components. SAST tools may struggle to accurately analyze code that interacts with these external dependencies, leading to incomplete or inaccurate vulnerability detection.
10. **Lack of Contextual Understanding**: SAST tools analyze code statically, without the context of the actual runtime environment.
11. **Handling Dynamic Code and Input Validation**: Web applications often involve dynamic code generation or rely on user inputs for certain operations.
12. **Complex Business Logic**: SAST tools may not fully comprehend the complex business logic and specific application requirements, resulting in limited coverage and potential missed vulnerabilities.
13. **False Sense of Security**: Relying solely on SAST tools can create a false sense of security. While these tools are valuable for identifying certain types of vulnerabilities, they cannot guarantee the absence of all security flaws.
14. **Tool Configuration and Expertise**: Configuring and fine-tuning SAST tools to align with the specific application and development environment can be challenging.
15. **Opportunities**.
16. **Early Vulnerability Detection**: SAST tools provide the opportunity to identify security vulnerabilities early in the development lifecycle. By scanning the source code before deployment, developers can catch potential issues and address them before they become more challenging and costly to fix.
17. **Secure Coding Practices**: SAST tools can help promote secure coding practices among developers. By providing real-time feedback and highlighting potential vulnerabilities, these tools raise awareness about common security pitfalls and encourage developers to write more secure code.
18. **Compliance and Regulatory Requirements**: SAST tools can assist in meeting compliance and regulatory requirements. Many industry standards and regulations, such as PCI DSS, HIPAA, or GDPR, include specific security requirements. SAST tools can help identify non-compliant code patterns and potential security weaknesses, ensuring adherence to these standards.
19. **Code Review Efficiency**: SAST tools automate the process of code review by scanning the codebase for security issues. This automation can significantly speed up the review process, enabling developers and security professionals to focus on more complex security tasks and reducing the manual effort required for code analysis.
20. **Vulnerability Prioritization**: SAST tools provide a systematic approach to vulnerability prioritization. By assigning severity levels or risk scores to identified issues, developers can prioritize remediation efforts based on the potential impact and likelihood of exploitation.
21. **Continuous Integration/Continuous Delivery (CI/CD) Integration**: SAST tools can be integrated into CI/CD pipelines, allowing for automated security checks during the software build process.
22. **Education and Training**: SAST tools can serve as educational resources for developers, helping them understand common security vulnerabilities and secure coding practices
23. **Open-Source Security**: SAST tools can help identify security vulnerabilities in open-source libraries and components used in web applications.
24. **Risk Vs Reward**

**RISK**

**Scanned an insecure application named “WebGoat-2023”. The scan result contains**

**Risks**

* **Bug.**
* Change this condition so that it does not always evaluate too "false"

Because it is easy to extract strings from an application source code or binary, passwords should not be hard-coded.

This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list

* "Random" objects should be reused (Security Misconfiguration)

When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information.

As the java.util.Random class relies on a pseudorandom number generator, this class and relating java.lang.Math.random() method should not be used for security-critical applications or for protecting sensitive data. In such context, the java.security.SecureRandom class which relies on a cryptographically strong random number generator (RNG) should be used in place.

* **Vulnerabilities**
* Don't use the default "PasswordEncoder" relying on plain-text.

A cross-site request forgery (CSRF) attack occurs when a trusted user of a web application can be forced, by an attacker, to perform sensitive actions that he didn’t intend, such as updating his profile or sending a message, more generally anything that can change the state of the application.

The attacker can trick the user/victim to click on a link, corresponding to the privileged action, or to visit a malicious web site that embeds a hidden web request and as web browsers automatically include cookies, the actions can be authenticated and sensitive.

# The JWT signature (JWS) should be verified before using this token.

Passwords should be stored outside of the code in a configuration file, a database, or a password management service.

This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list.

* **Code Smells**
* Remove this 'public' modifier.

**Weakened Access Control**: By removing the public modifier from the class, the visibility of the class is reduced, which can lead to weakened access control. Depending on the code's context and requirements, limiting the visibility of a class may restrict its accessibility and impact the overall functionality or interoperability of the codebase

* Define a constant instead of duplicating this literal "secQuestion0" 7 times.

Code Duplication and Maintenance: The code duplicates the string literals "secQuestion0" and "secQuestion1" multiple times. This can lead to code duplication, making it harder to maintain and update the code in the future. If there is a need to modify the string literals, you would have to make changes in multiple places, increasing the likelihood of errors and inconsistencies.

Lack of Readability and Reusability: Repeating the same string literals multiple times makes the code less readable and reduces its reusability. It is best practice to use constants or variables to represent repetitive or reusable values to enhance code clarity and maintainability.

**Common SAST Risks**

1. **False Positives**: SAST tools may generate false positives, flagging code segments as potential vulnerabilities when they are not. This can lead to wasted time and effort in investigating and remediating non-existent issues.
2. **False Negatives**: SAST tools can also produce false negatives, failing to detect actual vulnerabilities, leaving potential security risks unaddressed. Relying solely on SAST tools without other security measures can give a false sense of security.
3. **Limited Scope**: SAST tools have limitations in their ability to detect certain types of vulnerabilities, especially those related to complex business logic, dynamic code, and specific runtime environments.
4. **Tool Expertise**: Properly configuring and effectively utilizing SAST tools requires expertise and experience in security testing. Without adequate understanding and training, there is a risk of misinterpreting results or overlooking critical issues.

**REWARD**

1. **Early Vulnerability Detection**: SAST tools offer the opportunity to detect security vulnerabilities early in the development process, allowing for timely remediation and reducing the risk of potential exploits in production.
2. **Secure Coding Practices**: SAST tools can promote secure coding practices among developers by providing real-time feedback and raising awareness of common security pitfalls. This can lead to improved code quality and reduced vulnerabilities.
3. **Compliance and Regulatory Requirements**: Using SAST tools can help meet compliance and regulatory requirements by identifying non-compliant code patterns and potential security weaknesses, ensuring adherence to standards.
4. **Cost Savings**: Identifying and resolving security issues early in the development lifecycle can lead to cost savings compared to addressing them in later stages or after deployment. SAST tools can contribute to this cost reduction by automating the detection process.
5. **Improved Development Efficiency**: Integrating SAST tools into the development workflow can speed up the code review process, allowing developers and security professionals to focus on more complex security tasks. This improves overall development efficiency.
6. **Risk Mitigation**: By using SAST tools to identify and remediate security vulnerabilities, the risk of potential data breaches, financial losses, and damage to reputation can be mitigated.
7. **Reflections on the Internship**

* **Learning Opportunity**: The internship provided an excellent learning opportunity to gain hands-on experience with SAST tools and understand their role in application security. Working on real-world projects allowed for practical application of knowledge and understanding of web application vulnerabilities.
* **Understanding the Importance of Security**: Through the internship, the significance of incorporating security practices early in the development lifecycle became evident. Identifying and addressing vulnerabilities during development can significantly reduce the risk of security breaches and enhance the overall security posture of web applications.
* **Balancing Risk and Reward**: The internship highlighted the importance of balancing the risks and rewards associated with SAST tools. False positives and false negatives can present challenges, but the rewards of early vulnerability detection, secure coding practices, and compliance adherence outweighed these risks.
* **Collaboration with Development Teams**: The internship involved working closely with development teams, which fostered collaboration and communication skills. Understanding developers' perspectives and challenges helped in effectively communicating the findings and recommendations from the SAST analysis.
* **Continuous Learning**: The field of application security is ever-evolving, with new vulnerabilities and techniques emerging regularly. The internship emphasized the need for continuous learning and staying updated with the latest security trends and advancements in SAST tools and methodologies.
* **Integration into Development Processes**: The internship provided an opportunity to explore the integration of SAST tools into development processes, such as CI/CD pipelines. Understanding how SAST fits into the overall development workflow and collaborating with DevOps teams facilitated seamless security integration.
* **Practical Application of Security Principles**: Applying security principles and best practices in a real-world context reinforced their importance. Recognizing common vulnerabilities and understanding their impact on web applications deepened the understanding of secure coding practices.
* **Ethical Considerations**: Working with sensitive code and potential vulnerabilities underscored the importance of ethical considerations. Respecting confidentiality and handling vulnerabilities responsibly were critical aspects of the internship experience.
* **Building a Security Mindset**: The internship helped develop a security mindset, where considering security implications became a natural part of the development process. Recognizing that security is a shared responsibility among developers and security professionals was an important takeaway.
* **Future Growth Opportunities**: The internship served as a stepping stone for further growth and specialization in the field of application security.
* It was a great learning experience. It taught me how projects are done in the industry. •
* Submitting daily activity reports helped me to keep track of what I was doing each day.
* The discussion forum helped me to connect with other learners and discuss topics related to this subject

1. **Recommendations**

* **Bug.**
* Change this condition so that it does not always evaluate too "false"
* Store the credentials in a configuration file that is not pushed to the code repository.
* Store the credentials in a database.
* Use your cloud provider’s service for managing secrets.
* If a password has been disclosed through the source code: change it.
* "Random" objects should be reused (Security Misconfiguration)
* Use a cryptographically strong random number generator (RNG) like "java.security.SecureRandom" in place of this PRNG.
* Use the generated random values only once.
* You should not expose the generated random value. If you have to store it, make sure that the database or file is secure.
* **Vulnerabilities**
* Don't use the default "PasswordEncoder" relying on plain-text.
* The web application uses cookies to authenticate users.
* There exist sensitive operations in the web application that can be performed when the user is authenticated.
* The state / resources of the web application can be modified by doing HTTP POST or HTTP DELETE requests for example.

# The JWT signature (JWS) should be verified before using this token.

* The password allows access to a sensitive component like a database, a file storage, an API, or a service.
* The password is used in production environments.
* Application re-distribution is required before updating the password
* **Code Smells**
* Remove this 'public' modifier.
* Consider Codebase Design and Access Control: Before removing the public modifier from a class, carefully evaluate the codebase's design and access control requirements. Ensure that the class's visibility aligns with the intended use and accessibility within the codebase. If the class is intentionally designed to be accessed publicly, removing the public modifier may not be necessary.
* Define a constant instead of duplicating this literal "secQuestion0" 7 times.
* Define Constants for String Literals: Instead of duplicating the string literals, define constants for them.
* Use Constants in Code: Replace the duplicated string literals with the defined constants:
* userSecQuestions.put(SEC\_QUESTION\_0, "Dr. Watson");
* userSecQuestions.put(SEC\_QUESTION\_1, "Baker Street");
* This way, you avoid code duplication, improve code maintainability, and make the code more readable and reusable.

1. **Conclusion**

In this project, the application of Static Application Security Testing (SAST) tools, specifically SonarQube, was explored for finding defects in insecure web-based applications. SonarQube is a widely used SAST tool that helps developers identify security vulnerabilities and coding errors early in the software development lifecycle.

Throughout the project, SonarQube proved to be an effective tool for detecting various types of defects in web-based applications. It provided valuable insights into potential security weaknesses, such as SQL injection, cross-site scripting (XSS), insecure authentication mechanisms, and more. By analyzing the source code, SonarQube could identify insecure coding practices, code smells, and other potential issues that could lead to security breaches.

The use of SonarQube in the project brought several advantages. First and foremost, it enhanced the overall security of the web-based application by identifying and highlighting vulnerabilities that may have otherwise gone unnoticed. It allowed developers to fix these issues before the application went into production, reducing the risk of exploitation and potential damage.

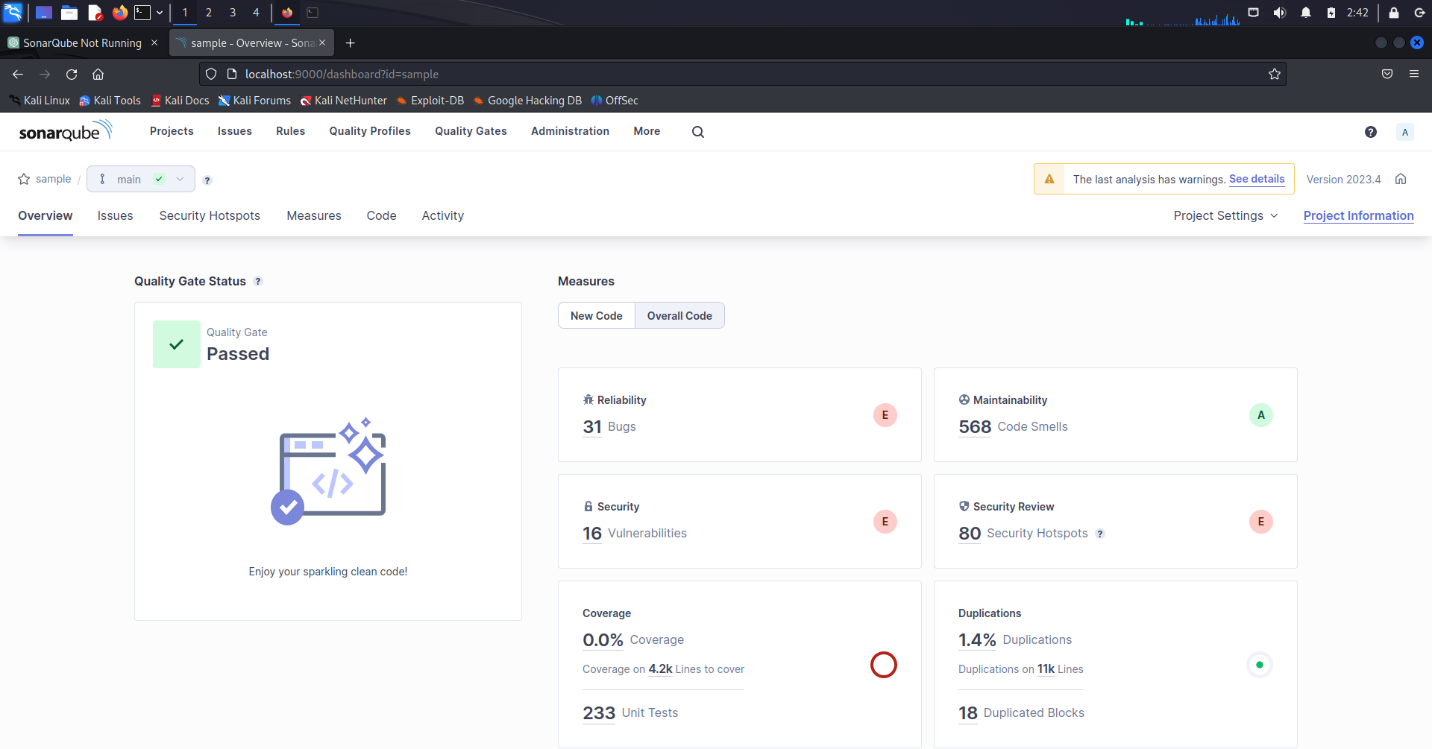
SonarQube also improved code quality and maintainability by pointing out potential code smells and suboptimal coding practices. By addressing these issues, developers could improve the overall robustness and reliability of the application.

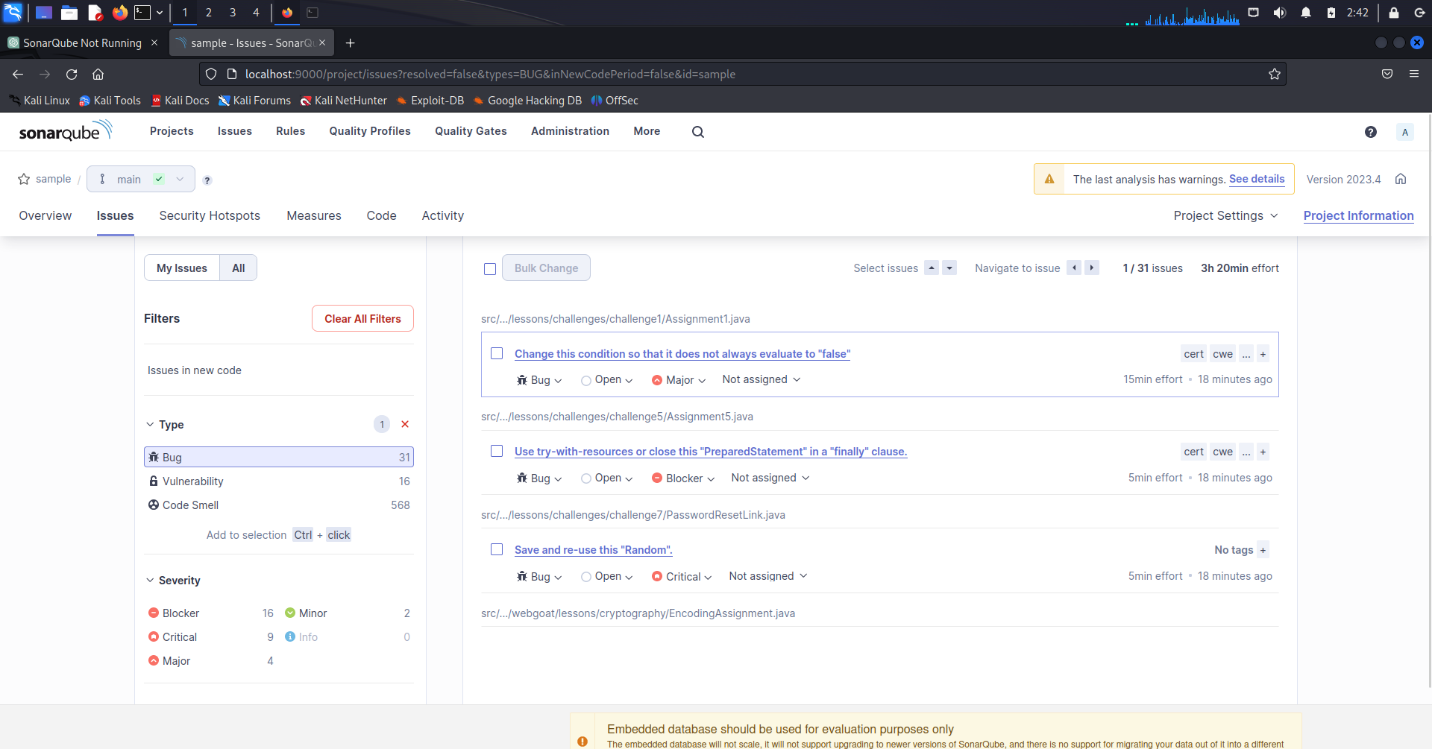
1. **Enhancement Scope**

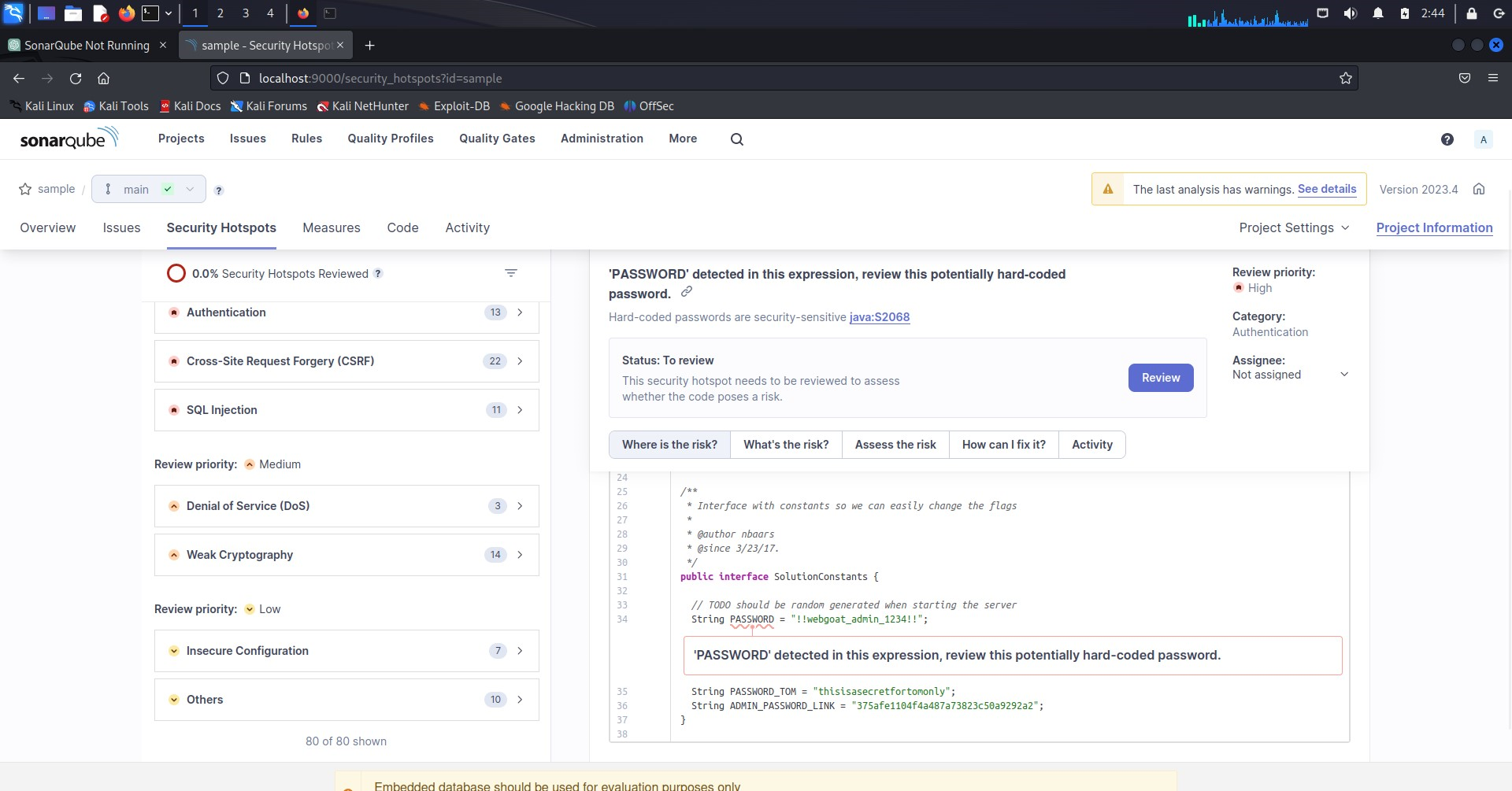
* Integration with Continuous Integration/Continuous Delivery (CI/CD) pipelines to enable automated security testing and faster feedback loops.
* Integration with other SAST tools or vulnerability scanners to complement SonarQube's capabilities and increase the coverage of security testing.

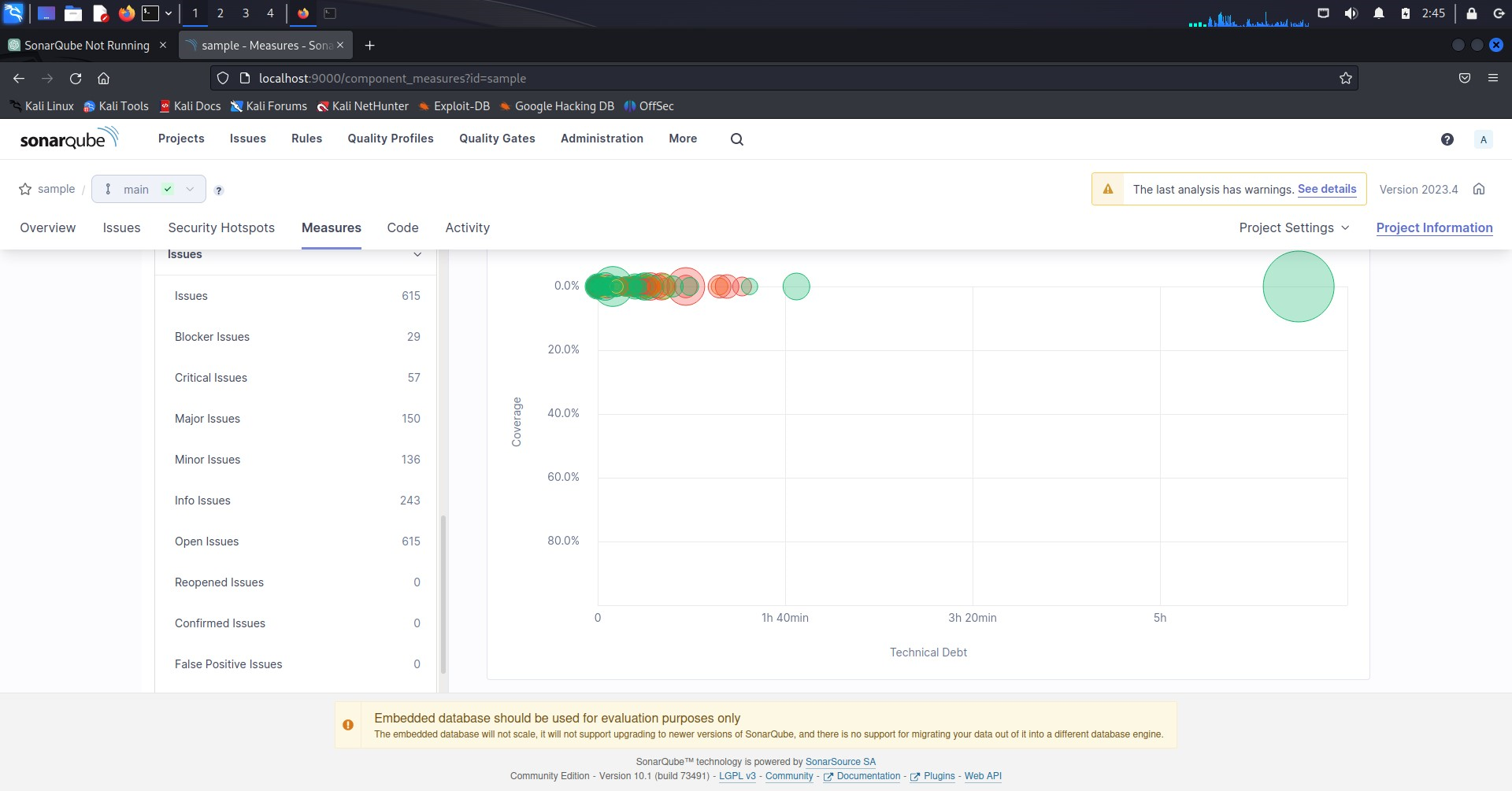
1. **Link to code and executable file**

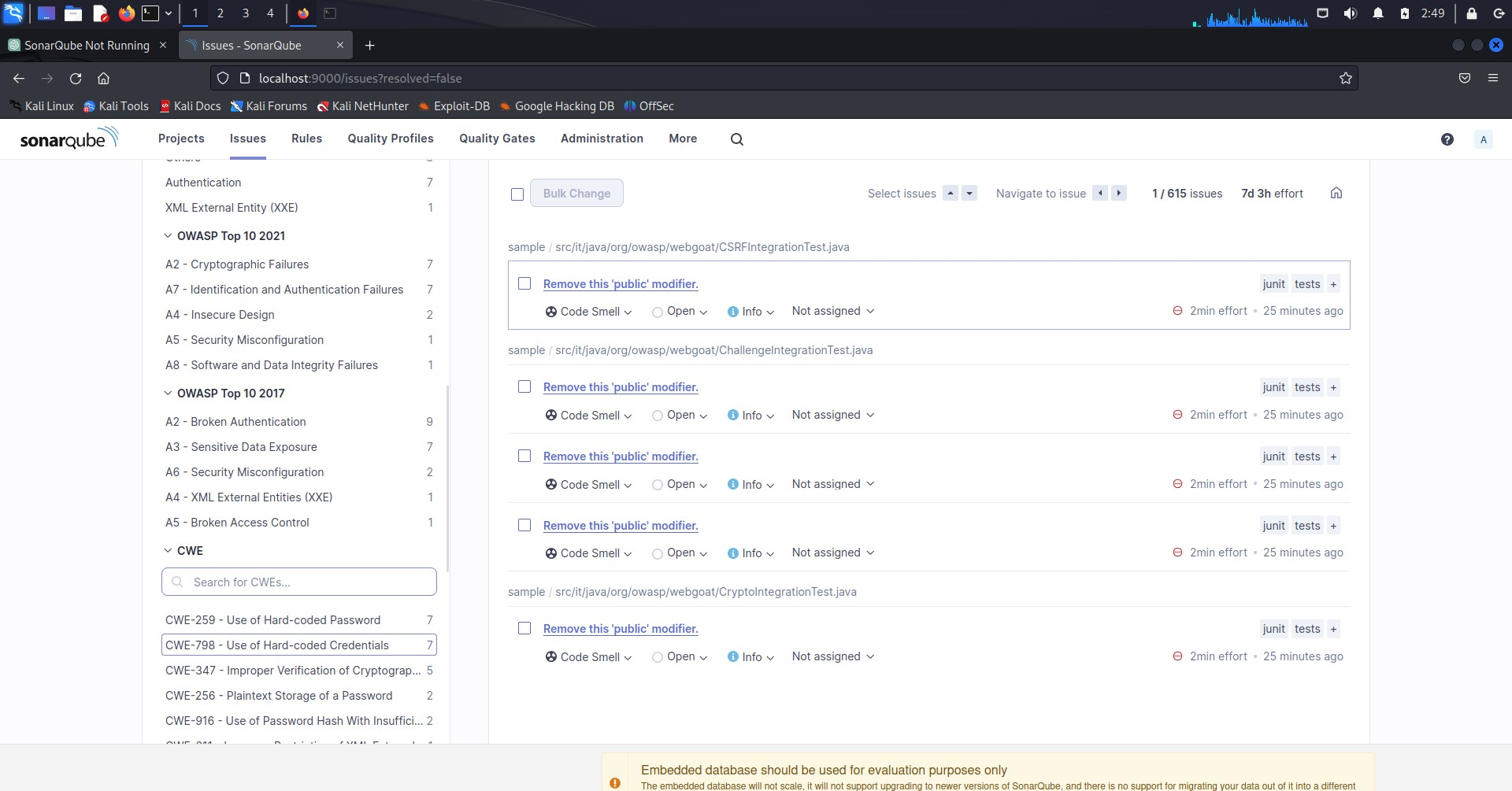
<https://github.com/Jaganathan03-G/Application-of-Static-Application-Security-Testing.git>

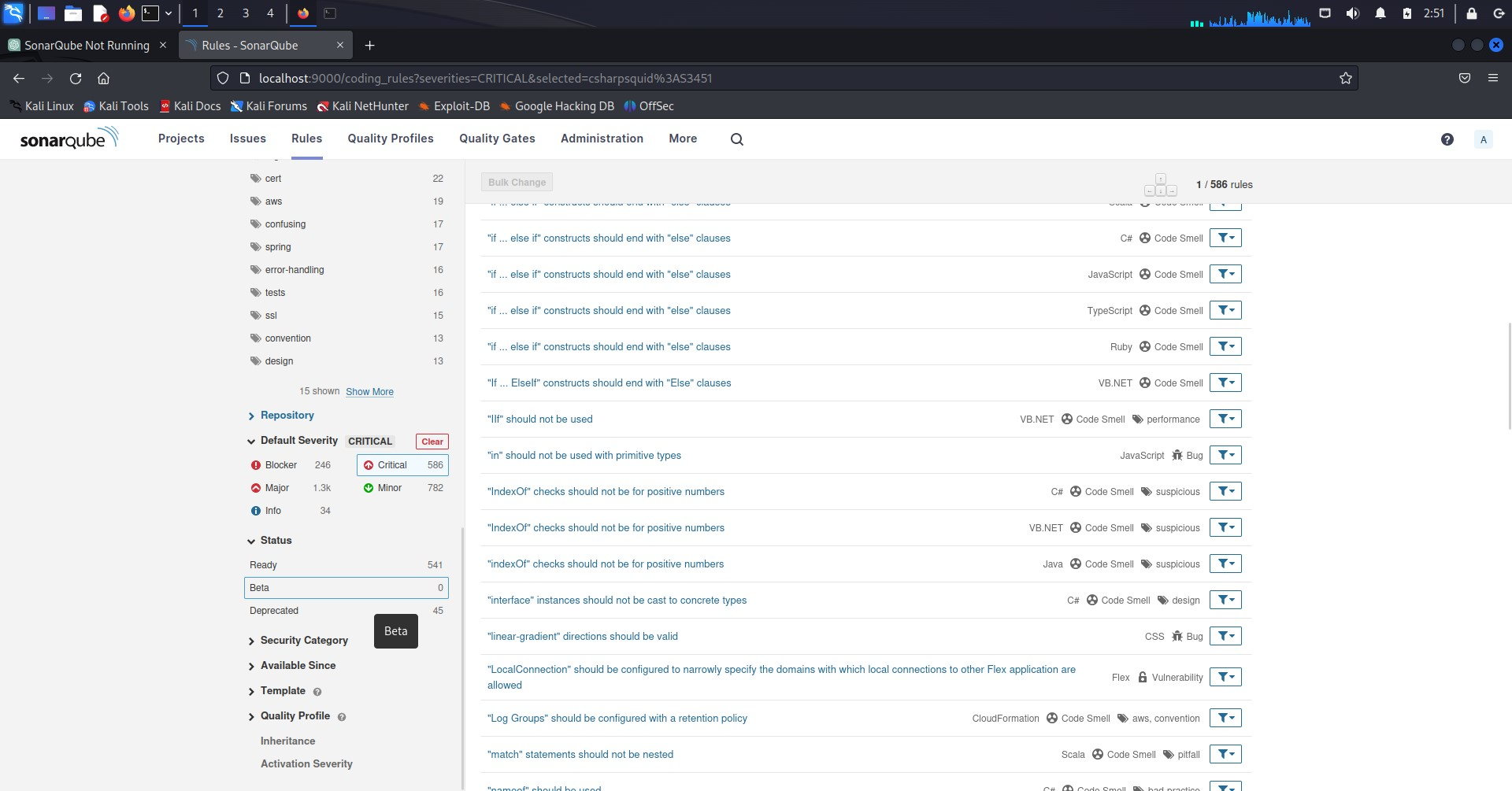


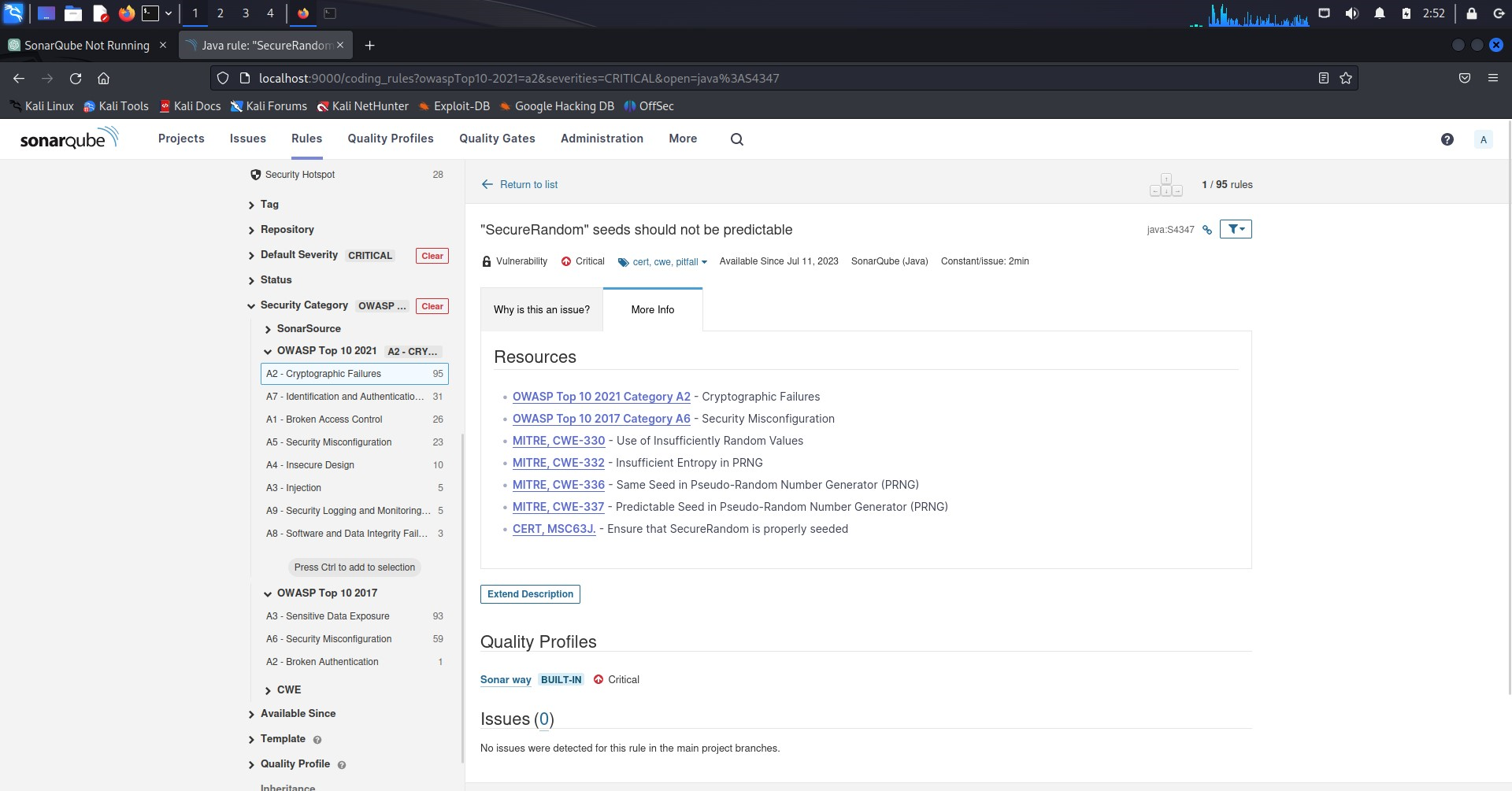












**References:**

[Release v8.0.0.M26 · WebGoat/WebGoat (github.com)](https://github.com/WebGoat/WebGoat/releases/tag/v8.0.0.M26)

Code Review Guide 2.0 (OWASP)

OWASP Top 10 – 2017 The Ten Most Critical Web Application Security Risks (https://owasp.org)