Tunisian Republic
Ministry of Higher Education and Research
Private Higher School of Technologies & Engineering

Summer Internship Report

Level : SSIR

Code Security Assessment: Utilizing SonarQube for Bitbucket-Hosted Projects and Blackbox Penetration Testing

By GHASSEN LANDOULSI

realized within HEXAGONE Afrique SARL

Professional supervisor : Quentin LE HENAFF

Scholar year: 2023 - 2024

Contents

General Introduction

As part of my engineering studies at the Private Higher School of Technologies Engineering, I sought an internship that would allow me to deepen my expertise in cybersecurity, particularly in code security assessment and penetration testing. HEXAGONE has committed itself to ensuring that modern organizations are well-equipped to tackle cyber threats, employing advanced methodologies to secure their codebases and systems. During my internship, I benefited immensely from the collaborative and knowledge-driven environment fostered by the company's experienced team, who supported me in developing both technical and analytical skills.

This report begins with a description of the hosting company and its mission. The second chapter explores SonarQube's role in code security assessment, covering its integration within development pipelines, key security features, and configuration for project-specific needs. The third chapter provides an in-depth look at blackbox penetration testing, where I describe the methodology, tools, and findings relevant to securing real-world applications. Finally, the report concludes with a summary of my experiences and the knowledge I gained during this project, highlighting its contribution to my growth in cybersecurity practices.

Chapter 1

The General framework of the internship

1.1 The General presentation of the host organization

1.1.1 Presentation of HEXAGONE

Hexagone is a leading provider of strategic digital solutions, specializing in the life sciences sector. The company leverages extensive industry experience and deep expertise to deliver innovative, results-driven solutions. With a comprehensive understanding of clients' challenges, Hexagone is uniquely positioned to offer tailored strategies that address the complexities of daily operations.

Focused on developing digital marketing materials for the life sciences, Hexagone streamlines processes, reduces inefficiencies, and optimizes spending. By partnering with global and local brands, the company integrates advanced technologies to drive operational excellence, build long-term business value, and enhance overall performance. Committed to future-proofing businesses, Hexagone designs forward-thinking solutions that meet the evolving demands of tomorrow.

Figure 1.1: Hexagone

1.1.2 Hexagone's services:

Hexagone is committed to delivering meaningful, transformative work with its clients, developing agile digital strategies that are both operational and sustainable over the long term.

Global Marketing managers Design and share best practices across regions and affiliates. Ensure and maintain Brand consistency customer engagement while being mindful of global compliance.

Local Marketing managers Map a full integrated and consistent customer experience.

Create compelling and relevant content through content strategy. Improve materials to develop more engaging content.
Digital managers Provide efficient support in digital campaign management. Orchestrate the creation and roll out of multichannel marketing initiatives. Implement global digital marketing strategy at affiliate level.

1.2 Organizational chart of the company

Hexagone is structured as such (here I'm only describing my superiors:

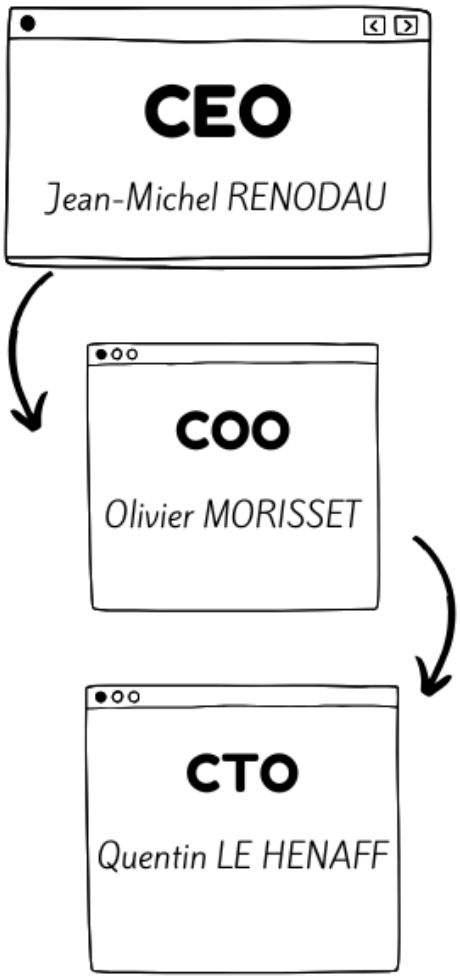

Figure 1.2: Hexagone's organizational Chart

1.3 Conclusion

In this chapter, I present Hexagone, its missions, and its organizational chart.

Chapter 2

SonarQube Code Security Assessment

2.1 Introduction to Code Security Assessment

2.1.1 Definition and importance of code security assessments

Code security assessment is a process of analyzing and evaluating code to identify potential vulnerabilities, security risks, and compliance issues that could expose applications to attacks.
This practice is essential in modern software development as it helps organizations to proactively detect and mitigate security flaws before deployment.
Conducting regular security assessments ensures that applications meet security standards and reduces the likelihood of exploitation, contributing to a more secure and resilient software environment.

2.1.2 Overview of static code analysis in software development security

Static code analysis is a method of examining source code without executing it, aimed at identifying potential security vulnerabilities, code quality issues, and compliance violations.
In software development security, static analysis plays a crucial role by providing developers with early insights into flaws that could be exploited if left unaddressed.
This approach leverages automated tools to scan codebases and generate reports, allowing for efficient identification of issues such as code injections, buffer overflows, and insecure data handling.
By integrating static code analysis into the development lifecycle, organizations can enhance code reliability and minimize security risks.

2.2 Overview of SonarQube

2.2.1 What is SonarQube?

SonarQube is an open-source platform designed for continuous code quality and security inspection.
It performs static code analysis to detect vulnerabilities, bugs, and code smells across

multiple programming languages.
SonarQube is widely used in development pipelines to help teams maintain high standards of code quality and security by identifying potential issues early in the development lifecycle.
Its dashboard provides detailed metrics, allowing developers to quickly review and address security and quality issues, making it an essential tool for maintaining robust, secure, and maintainable software.

Figure 2.3: SonarQube

2.2.2 Key features for security analysis and quality assurance

SonarQube offers a comprehensive set of features that enhance both security analysis and overall quality assurance.
It includes customizable rules for identifying security vulnerabilities such as SQL injection, cross-site scripting (XSS), and hardcoded credentials.
The platform uses security-focused quality gates that help teams enforce security standards by blocking insecure code from progressing through the pipeline.
SonarQube also provides detailed reports on code smells, bugs, and duplications, allowing developers to improve code maintainability and readability.
Additionally, its integration with CI/CD pipelines ensures continuous monitoring, enabling teams to detect and resolve issues early in the development process, thereby reducing the risk of security breaches.

2.2.3 Benefits of integrating SonarQube into a CI/CD pipeline

Integrating SonarQube into a CI/CD pipeline brings several benefits that enhance both security and development efficiency.
By automating code analysis within the pipeline, SonarQube enables immediate detection of security vulnerabilities, bugs, and code quality issues as part of the build process.
This approach promotes a proactive security culture, allowing developers to address problems early, thereby reducing the cost and complexity of fixing issues later in development.
Continuous monitoring also ensures that new changes align with established quality and security standards, providing a streamlined workflow that improves overall code reliability and reduces the risk of deploying insecure applications.

2.3 Configuring SonarQube for Bitbucket Integration

2.3.1 Setup process for integrating SonarQube with Bitbucket

To integrate SonarQube with Bitbucket, I hosted the SonarQube Scanner on an AWS EC2 container, providing a secure and scalable environment for running static code analysis.
The setup involved configuring authentication between Bitbucket and the SonarQube server using generated tokens, ensuring that only authorized requests from Bitbucket could trigger scans.
Additionally, a webhook was set up on Bitbucket to initiate the SonarQube analysis upon each commit or pull request, facilitating continuous monitoring of code changes.
This integration enables seamless and automated scanning, allowing code to be analyzed for security vulnerabilities and quality issues as part of the CI/CD pipeline.

2.3.2 Configuring Bitbucket pipelines for automated code scanning

To enable automated code scanning with SonarQube in Bitbucket, I configured a Bitbucket pipeline that triggers the SonarQube Scanner on each code push or pull request.
This setup involves defining a 'bitbucket-pipelines.yml' file within the repository, where the SonarQube scanning stage is specified.
The pipeline is configured to use the SonarQube token and server URL, enabling secure communication with the SonarQube instance hosted on AWS EC2. Upon execution, the pipeline automatically runs SonarQube analysis, providing immediate feedback on code security and quality issues.
This continuous integration approach helps maintain high standards by identifying and addressing vulnerabilities early in the development lifecycle.

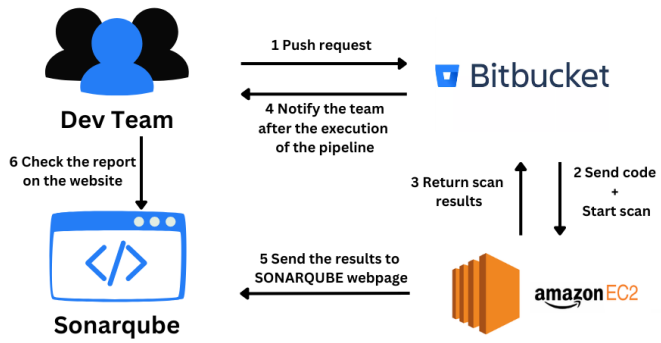

Figure 2.4: SonarQube scan workflow in CI/CD integration

2.3.3 Security considerations and access control for integrations

When integrating SonarQube with Bitbucket, it is crucial to implement robust security measures to safeguard both the code and the analysis process.
One key consideration is the use of secure authentication methods, such as personal access tokens, to prevent unauthorized access to the SonarQube server.
Tokens should be stored securely in Bitbucket environment variables, ensuring they are not exposed in the codebase.

Additionally, access control to the SonarQube instance should be carefully managed, with user roles and permissions defined to limit access to sensitive data and settings.
It is also essential to configure SonarQube to use secure connections (HTTPS) for communication between Bitbucket and the SonarQube server.
Regular audits of access logs and security configurations should be performed to detect any anomalies or potential vulnerabilities in the integration.

2.4 Running Security Scans with SonarQube

2.4.1 Types of vulnerabilities and code smells identified by SonarQube

SonarQube is designed to identify a wide range of security vulnerabilities and code quality issues, including common threats such as SQL injection, cross-site scripting (XSS), and improper authentication mechanisms.
It also detects code smells, which are suboptimal coding practices that, while not necessarily harmful, can lead to maintenance challenges or potential future vulnerabilities.
Examples of code smells include excessive code duplication, long methods, and overly complex class structures.
SonarQube's ruleset can be customized to target specific vulnerabilities and quality issues relevant to a given project.
This allows development teams to focus on high-priority concerns while maintaining a healthy, maintainable codebase.

2.4.2 Key security rules and metrics in SonarQube

SonarQube provides a set of predefined security rules designed to identify common vulnerabilities such as injection flaws, improper input validation, and insecure data handling.
Key security rules include checks for SQL injection, cross-site scripting (XSS), hardcoded credentials, and insufficient encryption. In addition to security rules, SonarQube offers various metrics to evaluate the overall security health of the codebase, such as vulnerability severity, the number of security hotspots, and the code coverage by tests.
The platform also provides a "Security Rating" metric, which classifies the code as secure, moderate, or insecure based on the detected vulnerabilities.
By enforcing these rules and monitoring security metrics, teams can ensure that security concerns are continuously addressed and that their codebase adheres to industry best practices.

2.4.3 Interpreting SonarQube scan results

Interpreting SonarQube scan results involves analyzing the identified issues in terms of their severity, impact, and the necessary remediation steps.
SonarQube categorizes findings into three main types: Bugs, Vulnerabilities, and Code Smells.
Bugs represent functional issues that could break the application, while vulnerabilities are security-related flaws that could be exploited by attackers.
Code smells refer to areas of the code that, although not immediately problematic, may lead to long-term maintenance issues or introduce future bugs.

Each issue is assigned a severity level—Critical, Major, Minor, or Info—which helps prioritize remediation efforts.
Additionally, SonarQube provides detailed descriptions for each issue, along with suggested fixes and links to relevant security guidelines, enabling developers to quickly understand and resolve the problems.

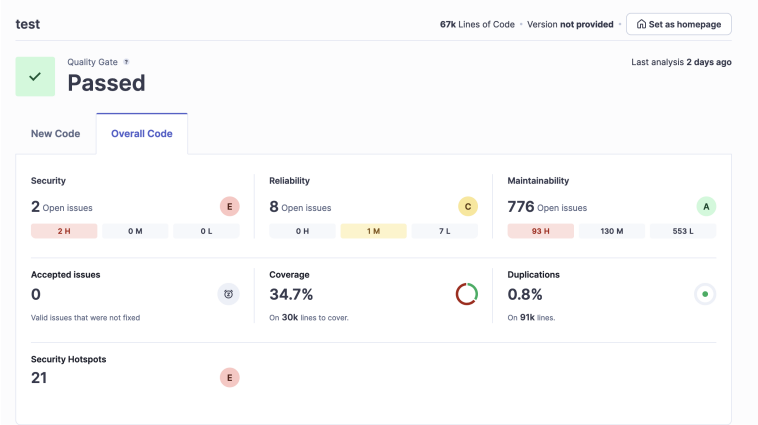

Figure 2.5: SonarQube scan Results UI

2.5 Exclusions and Customization in SonarQube

2.5.1 Setting up exclusions for specific files or modules

In some cases, it may be necessary to exclude certain files or modules from SonarQube scans, such as third-party libraries, generated code, or files that are not relevant to the security assessment.
SonarQube provides an easy way to configure exclusions through its user interface or configuration files.
To exclude specific files, you can define patterns in the ‘sonar-project.properties‘ file, such as ‘sonar.exclusions=**/vendor/**‘ to ignore all files in the ‘vendor‘ directory.
Alternatively, exclusions can be set in the SonarQube dashboard under the project settings.
By properly configuring exclusions, teams can ensure that scans focus on relevant code, improving performance and avoiding unnecessary warnings for non-critical files.

2.5.2 Customizing rules and quality gates for project-specific requirements

SonarQube allows for extensive customization of rules and quality gates to align with the specific needs and standards of a project.
Custom rules can be added or adjusted to target particular coding practices, security requirements, or industry regulations relevant to the project.
This can be done through the SonarQube UI or by creating custom plugins.
Quality gates, which define the criteria for passing or failing a project based on code quality, can also be tailored to enforce project-specific requirements such as a maximum allowable number of critical vulnerabilities or a minimum test coverage percentage.
By customizing rules and quality gates, teams can ensure that the code adheres to internal

coding standards and security practices, while maintaining the flexibility to adapt as project requirements evolve.

2.6 Challenges and Limitations of Using SonarQube

2.6.1 Limitations in security detection

While SonarQube provides robust static analysis for identifying many common security vulnerabilities, it has certain limitations in detecting more complex or subtle security issues.
For example, SonarQube primarily focuses on known vulnerability patterns and may miss newly emerging attack vectors or custom vulnerabilities specific to a particular application.
Additionally, SonarQube's static analysis may not fully account for runtime issues such as logic flaws, race conditions, or vulnerabilities that only manifest during dynamic execution.
False positives and false negatives can also occur, where non-issues are flagged as problems or actual vulnerabilities are overlooked.
To complement SonarQube's capabilities, manual code reviews, dynamic analysis tools, and penetration testing are recommended for a more comprehensive security assessment.

2.6.2 Challenges in configuring and interpreting results

Configuring SonarQube to suit specific project requirements can present several challenges.
One issue is fine-tuning the ruleset to avoid excessive false positives or negatives, especially when working with complex or legacy codebases.
Defining custom rules and quality gates that accurately reflect the security and quality needs of the project requires careful planning and expertise.
Interpreting the results can also be challenging, as SonarQube provides a wealth of information that may require context and deeper analysis.
Developers must prioritize vulnerabilities based on severity and potential impact, which can sometimes be subjective.
Additionally, integrating SonarQube into a CI/CD pipeline and ensuring smooth interaction with version control systems like Bitbucket may require addressing access control issues and configuring authentication tokens securely.
Despite these challenges, the benefits of automated code analysis and early vulnerability detection make SonarQube an essential tool for maintaining secure and high-quality software.

2.6.3 Suggestions for supplementing SonarQube with additional tools

While SonarQube is an effective tool for static code analysis and identifying a wide range of security vulnerabilities, supplementing it with additional tools can provide a more comprehensive security and quality assurance approach.
For example, **dynamic application security testing (DAST)** tools like OWASP ZAP or Burp Suite can be used to identify runtime vulnerabilities such as authentication flaws

and session management issues, which SonarQube might miss.
Additionally, integrating **software composition analysis (SCA)** tools like Snyk or WhiteSource can help detect vulnerabilities in third-party libraries and dependencies.
Manual code reviews, penetration testing, and fuzz testing should also be performed regularly to identify vulnerabilities that automated tools may overlook.
By combining SonarQube with these complementary tools, development teams can achieve a multi-layered approach to security, enhancing the overall reliability and safety of the software.

2.7 Conclusion

In conclusion, SonarQube provides a powerful and comprehensive solution for static code analysis, allowing teams to proactively identify and address security vulnerabilities, code quality issues, and potential maintenance challenges.
The integration of SonarQube with Bitbucket and its seamless inclusion in CI/CD pipelines enhances the efficiency of the development process by ensuring that code is continuously analyzed and assessed for quality and security risks.
While SonarQube excels in detecting common vulnerabilities such as SQL injections and XSS, it is important to recognize its limitations in detecting more complex or runtime-specific issues.
Supplementing SonarQube with additional tools, such as dynamic analysis and software composition analysis tools, further strengthens the security posture of the project.
Despite challenges in configuration and result interpretation, SonarQube's customization capabilities and detailed feedback make it an indispensable tool for maintaining high standards in software security and quality assurance.

Chapter 3

Blackbox Penetration Testing

3.1 Introduction to Blackbox Penetration Testing

3.1.1 Definition and Purpose of Blackbox Testing

Blackbox penetration testing, commonly referred to as external testing, involves assessing the security of an application or system from an external perspective with no prior knowledge of the internal architecture or source code.
The purpose of blackbox testing is to simulate an attacker's approach, allowing the tester to discover security vulnerabilities that could be exploited without internal access.
This approach provides valuable insights into the real-world security posture of the system and helps organizations mitigate risks that external attackers might exploit.

3.1.2 Differences Between Blackbox, Whitebox, and Greybox Testing

Penetration testing can vary widely based on the level of access and information provided to the tester.
In **blackbox testing**, the tester has no prior knowledge, simulating an external attacker's perspective.
Whitebox testing, in contrast, provides the tester with complete access to the system's internal architecture and source code, allowing a more thorough and exhaustive security review.
Greybox testing strikes a balance between the two, where the tester has partial knowledge or access, often mimicking an insider threat.
Each approach has unique advantages, with blackbox testing focusing on external threats, whitebox testing ensuring in-depth code review, and greybox testing combining aspects of both to address insider and outsider vulnerabilities.

3.2 Blackbox Testing Methodology

3.2.1 Planning and Scoping a Blackbox Test

The planning phase of a blackbox test is crucial to establish the scope, objectives, and rules of engagement.
During scoping, testers and stakeholders define the systems, networks, or applications in

scope, ensuring that sensitive or non-testable systems are excluded.
This phase also involves clarifying boundaries and permissions to avoid unintended disruptions, as well as agreeing on reporting and escalation protocols.
Clear scoping helps prevent issues and ensures that the test aligns with the organization's security goals.
In this project I was tasked with conducting a Pen-test on the company's main product which is a website

3.2.2 Tools and Techniques Commonly Used in Blackbox Penetration Testing

Blackbox penetration testing requires a variety of tools and techniques to simulate external attacks.
Commonly used tools include **network scanners** (e.g., Nmap) for discovering open ports and services, **vulnerability scanners** (e.g., Nessus, OpenVAS) for identifying weaknesses, and **web application security tools** (e.g., Burp Suite, OWASP ZAP) for testing web-based systems.
In addition, **reconnaissance tools** (e.g., theHarvester, Shodan) are employed to gather information about the target.
Manual testing techniques are also essential to verify findings and exploit vulnerabilities where possible.

3.2.3 Steps of the Testing Process: Reconnaissance, Scanning, Exploitation, and Reporting

The blackbox testing process follows a series of structured steps: reconnaissance, scanning, exploitation, and reporting.
Reconnaissance involves gathering public and passive information about the target, such as IP addresses and domain information.
Scanning aims to identify open ports, services, and versions, providing insights into potential attack vectors.
Exploitation is the process of attempting to exploit identified vulnerabilities within the defined scope and limitations.
Finally, reporting involves documenting findings, providing severity assessments, and recommending mitigations, offering a clear path for remediation.

3.3 Setting Up the Testing Environment

3.3.1 Preparing for an External Assessment (Tools, Permissions, and Boundaries)

Setting up the environment for a blackbox test requires selecting the appropriate tools, obtaining necessary permissions, and defining clear boundaries.
Tools for scanning, exploitation, and documentation should be organized and tested in advance to ensure they function effectively during the assessment.
Permissions are essential, particularly when testing a live environment, as they help prevent unintended disruptions and confirm that actions are within scope.

Boundaries must also be defined to ensure that testing remains focused on permitted areas, reducing the risk of impacting unrelated systems.

3.3.2 Approach to Test Targets Like Web Applications and Network Services

Blackbox testing techniques vary depending on the target type.
For web applications, testing often involves checking for common vulnerabilities such as SQL injection, XSS, and authentication flaws.
Network services are assessed for open ports, misconfigured services, and potential access points.
Each target requires tailored methodologies and tools, with special considerations for sensitive or critical infrastructure that may require more cautious testing approaches to avoid disruptions.

3.4 Reconnaissance and Information Gathering

3.4.1 Techniques for Collecting Public and Passive Information (OSINT)

Open Source Intelligence (OSINT) is a crucial step in blackbox testing, involving the collection of publicly available information about the target.
This can include domain names, IP addresses, contact information, and exposed technologies, obtained through search engines, WHOIS databases, and tools like Shodan and theHarvester.
Passive information gathering helps build an attacker's perspective, providing insights that inform subsequent testing phases.

3.4.2 Scanning for Open Ports, Services, and Versions

After gathering public information, the next step involves scanning the target for open ports, services, and versions to identify potential entry points.
Tools like Nmap are commonly used for this phase, allowing testers to map the network and discover exposed services.
Scanning results provide a foundation for vulnerability identification, as each open port and service represents a potential vector for exploitation.

3.5 Vulnerability Identification

3.5.1 Methods for Identifying Common Vulnerabilities (e.g., SQL Injection, XSS)

Identifying vulnerabilities is a core aspect of blackbox testing.
Methods include automated scanning for known vulnerabilities, manual testing for issues like SQL injection and cross-site scripting (XSS), and testing for misconfigurations.
These common vulnerabilities can often lead to severe security risks if left unaddressed, making their identification and prioritization critical to the testing process.

3.5.2 Tools for Automated and Manual Vulnerability Discovery

To discover vulnerabilities effectively, a combination of automated and manual methods is recommended.
Tools like Burp Suite, OWASP ZAP, and Nessus are used for automated scanning, identifying low-hanging vulnerabilities quickly.
Manual testing allows for more detailed exploration, verifying findings and uncovering weaknesses that automated tools may miss.
This balanced approach ensures comprehensive vulnerability discovery.

3.6 Exploitation and Vulnerability Verification

3.6.1 Process for Validating Identified Vulnerabilities

Validating vulnerabilities is essential to confirm the severity and exploitability of findings.
Testers attempt to reproduce identified issues under controlled conditions, ensuring that each vulnerability is legitimate.
Validation helps in prioritizing vulnerabilities based on their actual impact, guiding remediation efforts effectively.

3.6.2 Safely Exploiting Vulnerabilities Within Scope Limitations

Exploitation in blackbox testing is conducted carefully, respecting the agreed-upon scope and limitations to avoid causing damage.
Safe exploitation techniques are employed to demonstrate the existence of vulnerabilities without compromising system integrity.
For instance, testers might perform proof-of-concept exploits that show the vulnerability without altering data or causing disruptions.

3.6.3 Handling Sensitive Data and Maintaining Confidentiality

During blackbox testing, handling sensitive data with care is critical.
Testers must follow strict confidentiality protocols, ensuring that any accessed data remains secure and is not disclosed or misused.
Sensitive data handling practices are outlined in advance, often including data redaction and secure storage, to protect the organization's information.

3.7 Reporting and Recommendations

3.7.1 Structuring a Blackbox Testing Report (Findings, Severity, Recommendations)

A well-structured report is essential to convey the results of a blackbox test.
Each finding is documented with details on its severity, potential impact, and specific recommendations for mitigation.
The report provides a clear overview of vulnerabilities, their risk levels, and actionable steps to enhance security, making it a valuable resource for remediation planning.

3.7.2 Prioritizing Findings and Suggesting Mitigation Strategies

In the reporting phase, findings are prioritized based on their severity and potential impact.
Critical vulnerabilities are highlighted, with immediate mitigation strategies suggested to address the most pressing issues.
Recommendations may include patches, configuration changes, or enhanced security controls to minimize risk and improve the organization's overall security posture.

3.7.3 Documenting Attack Paths and Exploitation Scenarios

Attack paths and exploitation scenarios are documented to provide a clearer understanding of how vulnerabilities could be exploited by attackers.
This documentation includes details on each step taken during testing, illustrating potential exploitation chains and demonstrating the consequences of unpatched vulnerabilities.
These scenarios help the organization comprehend the importance of remediation and the potential impact of an attack.

3.8 Challenges and Considerations

3.8.1 Common Challenges in Blackbox Testing (e.g., Limited Visibility, False Positives)

Blackbox testing presents unique challenges, including limited visibility into the system's internal workings and a reliance on external analysis.
False positives are also common, as the lack of full access can lead to misinterpretations.
Addressing these challenges requires careful tool selection, verification processes, and a methodical approach to testing.

3.8.2 Ethical and Legal Considerations

Ethical and legal considerations are integral to blackbox testing, ensuring that all actions comply with regulations and respect organizational boundaries.
Testers must adhere to ethical guidelines, obtaining proper permissions and avoiding unauthorized access or data exposure.
Legal compliance is vital to avoid liability and ensure that testing is conducted responsibly.

3.8.3 Limitations of Blackbox Testing Compared to Other Methodologies

While blackbox testing is effective for identifying external vulnerabilities, it has limitations compared to whitebox and greybox testing.
The lack of internal access means that certain vulnerabilities may go undetected, and some complex security issues cannot be fully assessed.
Combining blackbox testing with other methodologies provides a more comprehensive security assessment.

3.9 Conclusion

This chapter outlined the essentials of blackbox penetration testing, a method that simulates external attacks to uncover vulnerabilities visible to outsiders.
Beginning with the planning and scoping phases, we examined tools and techniques for reconnaissance, vulnerability identification, and safe exploitation.
Effective blackbox testing relies on a structured approach and careful consideration of scope, permissions, and limitations, with each step aimed at thoroughly evaluating the system's defenses.

While blackbox testing presents challenges like limited visibility and potential false positives, it remains a powerful tool for understanding external threats.
Through careful reporting and prioritization, this method helps organizations address vulnerabilities effectively, enhancing overall security posture against real-world attacks.

General Conclusion

In this report, we have explored critical aspects of modern cybersecurity practices through code security assessment and blackbox penetration testing. Each chapter focused on a unique area, starting with an in-depth analysis of SonarQube's role in automated code security assessments. By integrating SonarQube into a CI/CD pipeline, we demonstrated how organizations can maintain code quality and identify potential vulnerabilities before deployment, ensuring that security is woven into the development lifecycle.

The examination of blackbox penetration testing provided insight into the process of simulating real-world attacks to evaluate external threats to an organization's infrastructure. From reconnaissance to exploitation, this approach highlights how attackers view an organization's defenses, enabling security professionals to uncover and mitigate risks that would otherwise remain undetected.

Together, these methods illustrate a holistic approach to cybersecurity, addressing risks from both code vulnerabilities and external threats. By combining automated code analysis with thorough penetration testing, organizations can strengthen their defenses across the board. This report underscores the importance of proactive security measures and continuous improvement to stay resilient in the evolving landscape of cyber threats.

List of Figures

Liste des acronymes

OSINT : *Open Source Intelligence*

CI/CD : Continuous Integration / Continuous Deployment

OSINT : Open-Source Intelligence

XSS : Cross-Site Scripting

SQL : Structured Query Language

AWS : Amazon Web Services

IP : Internet Protocol