INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT (IJSREM)
VOLUME: 05 ISSUE: 04 | APRIL - 2021 SJIF RATING: 6.714 ISSN: 2582-3930

Systematic Approach to Prevent Code Vulnerabilities using CI/CD Pipelines

Kamalakar Reddy Ponaka
kamalakar.ponaka@gmail.com

***Abstract* — This paper discusses a systematic approach to integrating Static Application Security Testing (SAST), Software Composition Analysis (SCA), Code Coverage, and Code Quality Checks into Continuous Integration/Continuous Delivery (CI/CD) pipelines. Modern CI/CD pipelines accelerate software delivery but introduce significant security and quality challenges. By incorporating SAST and SCA for security testing, along with code coverage and quality checks, organizations can prevent code vulnerabilities and ensure the maintainability and reliability of their applications. This approach helps development teams shift security and quality controls left, catching issues early in the development lifecycle.**

***Keywords* —** *CI/CD, DevSecOps, Static Application Security Testing (SAST), Software Composition Analysis (SCA), Code Coverage, Code Quality, Vulnerability*

I. INTRODUCTION

Continuous Integration and Continuous Delivery (CI/CD) pipelines have become essential for rapid software development and delivery. However, as speed increases, security vulnerabilities and code quality issues often emerge, creating risks for production environments. To mitigate these risks, organizations must integrate robust security and quality measures into their CI/CD workflows. This paper presents a systematic approach to preventing code vulnerabilities and enforcing code quality by integrating Static Application Security Testing (SAST), Software Composition Analysis (SCA), code coverage, and code quality checks within the CI/CD pipeline.

II. PIPELINE PHASES

A typical CI/CD pipeline includes the following stages:

Source Code Management: Developers commit code to a version control system (e.g., Git).

Build: The application is compiled, and build artifacts are generated.

Verify/Test: Automated testing, including unit tests, security scans, and code quality checks, is performed.

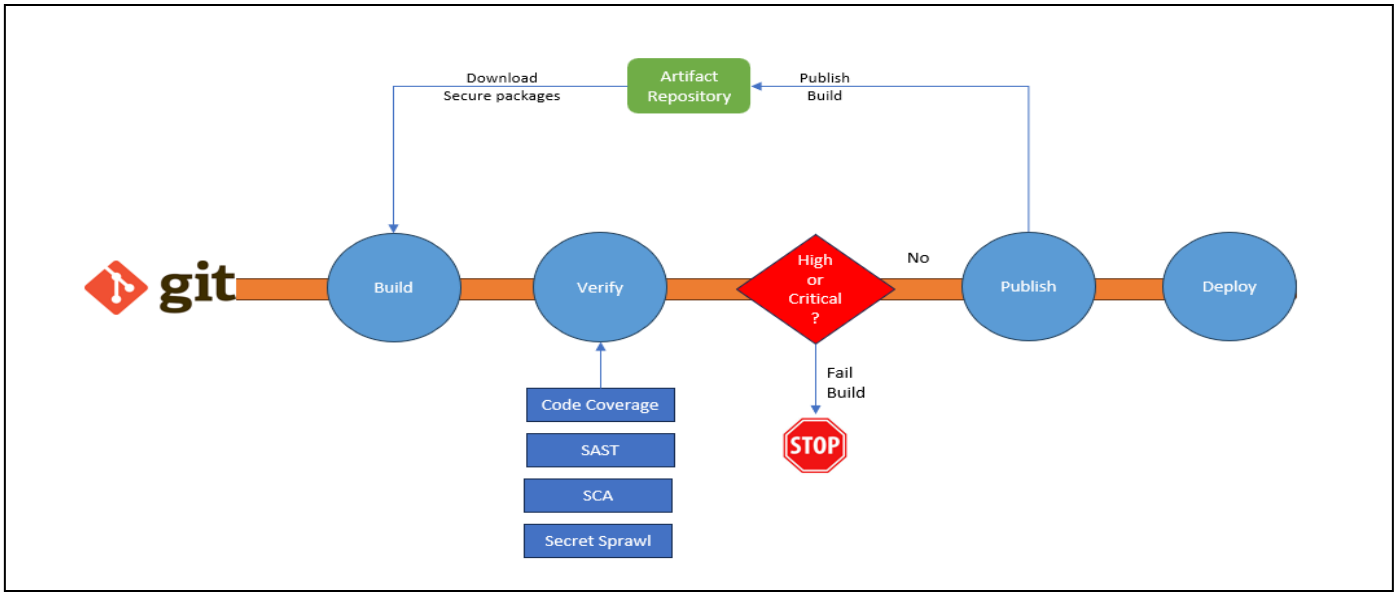
Publish: Publish the generated package and tag with version number.

Deployment: The application is deployed to testing or production environments.

III. SECURITY AND QUALITY CHALLENGES IN CI/CD PIPELINES

CI/CD pipelines consist of several stages, including code integration, automated testing, and deployment. While this process increases development speed, it also exposes software to various security risks and quality issues, such as:

a) *Insecure Code:* Poor coding practices lead to vulnerabilities like SQL injection and Cross-Site Scripting (XSS).



b) *Third-Party Dependencies:* Vulnerabilities in third-party libraries introduce risks into applications, which are often overlooked in security testing.
c) *Low Test Coverage:* Insufficient test coverage can leave critical parts of the application untested, resulting in undetected bugs and vulnerabilities.
d) *Poor Code Quality:* Poorly structured, unmaintainable code increases technical debt and the likelihood of security breaches due to hidden bugs and complexity.

IV. STATIC APPLICATION SECURITY TESTING (SAST)

SAST involves the static analysis of source code for security vulnerabilities before the code is compiled or executed. As a white-box testing technique, SAST identifies issues such as SQL injection, buffer overflows, and cross-site scripting (XSS) in the early stages of development.

A. *Benefits of SAST*

- **Early Detection of Vulnerabilities:** SAST scans code as it is written, enabling the early detection of vulnerabilities.
- **Integration in CI/CD:** SAST can be integrated into CI/CD pipelines to scan code after each commit, ensuring continuous security.
- **Comprehensive Coverage:** SAST tools analyze code for a wide range of vulnerabilities across different programming languages.

B. *Challenges of SAST*

- **False Positives:** SAST tools may generate false positives, requiring manual review to confirm the relevance of detected vulnerabilities.
- **Complexity:** Interpreting SAST results requires skilled developers who understand security issues and how to resolve them.

V. SOFTWARE COMPOSITION ANALYSIS (SCA)

SCA tools focus on analyzing third-party dependencies for known vulnerabilities. Many modern applications rely on open-source libraries and frameworks, which can introduce vulnerabilities if not properly managed.

A. *Benefits of SCA*

- **Dependency Vulnerability Management**: SCA ensures that third-party libraries are free from known vulnerabilities.
- **License Compliance**: SCA helps organizations ensure that open-source licenses comply with legal requirements.
- **Continuous Monitoring**: SCA tools continuously monitor dependencies for new vulnerabilities, alerting teams when remediation is required.

B. *Challenges of SCA*

- **Complex Dependency Trees**: Managing vulnerabilities in large dependency trees, where libraries depend on other libraries, can be challenging.
- **False Positives**: Some vulnerabilities identified by SCA tools may not be exploitable in the context of the application, leading to false positives.

VI. CODE COVERAGE

Code coverage measures the percentage of source code executed during testing. By increasing code coverage, organizations can ensure that a larger portion of the codebase is tested, reducing the risk of untested vulnerabilities or bugs.

A. *Types of Code Coverage*

a) *Line Coverage:* Measures the percentage of lines of code executed during testing.
b) *Branch Coverage:* Measures how many control structures (e.g., if statements) are executed.
c) *Function Coverage:* Measures the percentage of functions or methods executed during testing.

B. *Benefits of Code Coverage*

- **Better Test Assurance**: Ensures that critical parts of the application are tested.
- **Early Bug Detection**: Uncovers untested code paths that may harbor bugs or vulnerabilities.

C. *Challenges of Code Coverage*

- **Quality vs. Quantity**: High coverage percentages do not necessarily mean effective testing. It is essential to ensure that tests are meaningful and validate critical functionality.

VII. CODE QUALITY CHECKS

Code quality checks ensure that code is maintainable, readable, and follows industry best practices. Tools such as **SonarQube** and **CodeClimate** evaluate code based on metrics like cyclomatic complexity, duplication, and maintainability.

A. Key Code Quality Metrics

a) *Cyclomatic Complexity:* Measures the complexity of code by evaluating control flow paths. Higher complexity makes code harder to maintain and test.

b) *Code Duplication:* Identifies duplicate code, which can increase maintenance overhead.

c) Maintainability Index: Provides an overall score indicating how maintainable the code is.

B. Benefits of Code Quality Checks

- **Improved Maintainability**: Enforcing best practices reduces technical debt and increases code readability.
- **Reduced Risk of Bugs**: Cleaner, more maintainable code is less prone to bugs.

C. Challenges of Code Quality Checks

- **Developer Overhead**: Enforcing strict code quality checks can slow down development if not optimized.
- **Balancing Style and Functionality**: Focusing too much on style-related issues can detract from addressing functional bugs.

VIII. INTEGRATING SAST, SCA, CODE COVERAGE, AND CODE QUALITY IN CI/CD PIPELINES

The following steps describe how to systematically integrate SAST, SCA, code coverage, and code quality checks into CI/CD pipelines:

A. SAST Integration

a) *Pre-Commit Hooks:* Run SAST scans before code is committed to the repository to detect early vulnerabilities.

b) Automated Scans in Pipeline: Configure automated SAST scans to run during the build phase.

c) Feedback and Remediation: Notify developers immediately when vulnerabilities are detected and require remediation before progressing.

B. SCA Integration

a) *Automated Dependency Scanning:* Use SCA tools to scan dependencies for known vulnerabilities during the build process.

b) *Fail on Critical Vulnerabilities*: Block deployments if high/critical vulnerabilities are detected in third-party libraries.

c) *Automated Dependency Updates:* Implement tools like Dependabot or Renovate to automatically update vulnerable dependencies.

C. Code Coverage Integration

a) *Test Coverage Tools:* Use tools such as JaCoCo for Java or pytest-cov for Python to measure code coverage.

b) Coverage Thresholds: Set coverage thresholds (e.g., 80%) and fail builds if the threshold is not met.

c) *Reporting:* Generate detailed coverage reports and share them with development teams to improve test coverage.

D. Code Quality Integration

a) *Automated Quality Analysis:* Use SonarQube or CodeClimate to automatically analyze code for quality metrics.

b) *Quality Gates:* Set quality gates that block the build if critical code smells or high complexity issues are detected.

c) Track Technical Debt: Use code quality tools to track and manage technical debt over time.

CONCLUSION

Integrating SAST, SCA, code coverage, and code quality checks into CI/CD pipelines ensures that security and quality are addressed continuously throughout the development lifecycle. By shifting left and incorporating these tools early, development teams can reduce vulnerabilities, improve code quality, and deliver more secure, maintainable software.

REFERENCES

[1] A. M. Davis, "Cross-site scripting vulnerabilities," *IEEE Security & Privacy*, vol. 9, no. 5, pp. 77-80, 2011.

[2] S. M. Bellovin, "Static analysis and software security: A work in progress," *IEEE Security & Privacy*, vol. 5, no. 4, pp. 76-79, 2007.

[3] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An empirical study of CI build failures," in *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)*, 2017, pp. 123-131.

[4] H. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: Learning to classify vulnerabilities and predict exploits," in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 105-114.

[5] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Boston, MA: Addison-Wesley, 2010.

[6] A. L. Bucchiarone, S. Gnesi, P. Pelliccione, and G. Polini, "Continuous integration of model-based techniques into DevOps," *IEEE Software*, vol. 35, no. 1, pp. 66-71, 2018.

[7] R. Jenkins, "Automating security checks in CI/CD pipelines," *IEEE Computer*, vol. 52, no. 11, pp. 30-37, 2019.

[8] O. Alhazmi and Y. Malaiya, "Quantitative vulnerability assessment of systems software," in *Proceedings of the 2005 International Symposium on Software Reliability Engineering (ISSRE)*, 2005, pp. 323-333.

[9] G. Catolino, F. Palomba, D. A. Tamburri, and F. Ferrucci, "Improving code quality with code smell diversity-aware analysis," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 282-301, 2022.

[10] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006, pp. 452-461.