Usability of Static Application Security Testing Workflows

Bhagya Chembakottu
School of Computer Science
McGill University
Montreal, Canada
bhagya.chembakottu@mail.mcgill.ca

Martin P. Robillard
School of Computer Science
McGill University
Montreal, Canada
robillard@acm.org

***Abstract*—The usability of static application security testing tools (SASTs) can facilitate the development of secure code within GitHub workflows. We report on our experience applying these tools with Spring and Django web development frameworks, analyzing aspects such as setup complexity, build integration, and the utility of the generated vulnerability reports. A key observation is that Django projects require less effort to integrate, whereas Spring projects involve significant setup challenges, particularly with SonarCloud, due to build environment dependencies. Furthermore, we observed usability issues such as ambiguous error messages and inconsistent warnings. By examining setup time and error incidence, we provide insights for improving SAST usability and recommendations for easier installation and clearer notifications.**

***Index Terms*—Software Security, Static Analysis, Web App**

I. Introduction

Static analysis tools have proven invaluable in identifying vulnerabilities in web applications. According to established software engineering practices, incorporating security awareness during the initial stages of development has become a critical priority [1]. Static analysis tools are employed as a proactive approach to security testing by detecting potential vulnerabilities without requiring the software to be executed, which can speed up the vulnerability identification process [2]. Traditional static analysis tools rely on predefined rules and advanced techniques such as query languages to analyze source code and identify potential security risks [3]. However, developers often encounter obstacles when setting up the tools, interpreting notifications, and making decisions informed by the analysis output, especially when integrating the tools into development pipelines [4], [5].

Integrating static analysis into software engineering processes can be challenging due to the need for configuration, setup, and maintenance [6]. Additionally, the effort required to understand notifications, prioritize them, and determine appropriate remediation can distract developers and impact their productivity and willingness to adopt these tools [7]. Recognizing these limitations, there have been initiatives to incorporate security practices into the software development life cycle so as to minimize developer burden and encourage usage [8].

One such initiative is the integration of static analysis as part of continuous integration and continuous delivery (CI/CD) *workflows*, such as those offered through GitHub Actions [9]. GitHub workflows allow developers to automate processes, including running tests, deploying code, and performing static analysis within their repositories without manual intervention. In recent developments, GitHub introduced code scanning as a supported feature, thus further simplifying the process of incorporating static analysis tools directly into the CI/CD pipeline [10]. By using templates for code scanning workflows, GitHub aims to reduce the efforts of developers to set up security scans configuration.

Despite these advancements, questions remain about whether these workflows improve the usability of static analysis tools, particularly in web application projects with varying languages, frameworks, and build environments. We are currently analyzing how static application security testing tools (SASTs) can be used along with GitHub repositories to improve security. We deployed SASTs to identify security vulnerabilities in two web application frameworks and documented our experience. Our analysis focused on two aspects:

a) Impact of Development Environments on Workflow Setup: We analyzed how software dependencies impacted the ease or difficulty of using static analysis workflows.

b) Quality of the Notifications: We evaluated how understandable and actionable the notifications were to determine whether these messages help users efficiently address issues or add unnecessary complexity.

II. Background and Related Work

Previous work includes studies on the usability of static analysis tools and investigations of the factors influencing secure development practices.

Usability of Static Analysis Tools: Static code analysis tools aim to assist users in detecting defects, enforcing standards, and improving security. However, poor usability limits their adoption in workflows [5], [7], [11]. Common challenges include inadequate feedback, unclear warning messages, and insufficient integration with developer workflows [4]. Smith et al. identified usability gaps in popular tools, such as limited support for managing vulnerabilities and resolving issues effectively [7]. Nachtigall et al. highlighted issues with insufficient warning messages and lack of actionable suggestions across 46 SASTs [11]. Piskachev et al. demonstrated that

TABLE I
SETUP TIME AND CONFIGURATION STEPS FOR CODEQL AND SONARCLOUD ACROSS FRAMEWORKS

Language/Framework	CodeQL Setup Time (min)	SonarCloud Setup Time (min)	Average Configuration Steps
Python (Django)	15	20	5
Java (Spring)	25	35	8

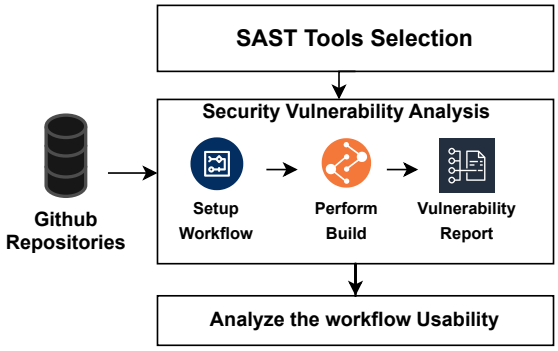

Fig. 1. Overview of wokflow setup and analysis

customizable configurations enhance vulnerability resolution, emphasizing the need for adaptable tool settings [4]. Tahaei et al. noted the importance of meaningful notifications in aiding developers, although challenges persist in implementing suggested resolutions accurately [5]. Collectively, these studies underscore the need to prioritize usability to leverage the full potential of static analysis tools.

Secure Software Development Practices: Nurgalieva et al. proposed a model for privacy and security practices in software development, identifying five influencing factors: environmental, organizational, product-related, process-related, and individual [12], [13]. Regulatory frameworks and industry standards shape environmental factors, while organizational culture and resources drive internal prioritization. Product-related factors balance security as a competitive differentiator against business needs. Process-related challenges, such as lack of automation and evaluation metrics, hinder integration, while individual factors highlight developers' expertise and attitudes. Assal et al. categorized developers into *security adopters*, who integrate security across the software development lifecycle (SDLC), and *security inattentive*, who delegate these responsibilities [6]. These findings highlight gaps between best practices and real-world applications, suggesting the need for a multi-level approach to promote pragmatic and adaptable security practices.

III. CONTEXT

We evaluated the integration and usability of SASTs in GitHub workflows by analyzing the setup process, build execution, and effectiveness of generated vulnerability reports. The objective was to identify practical challenges and usability concerns when integrating and using SASTs.

We followed the process, illustrated in Figure 1. The first step involved selecting SASTs based on criteria such as popularity, GitHub compatibility, multi-language support, and the ability to produce detailed vulnerability reports. We chose CodeQL and SonarCloud due to their compatibility with GitHub workflows and comprehensive programming language support. We focused on five web applications developed using Spring and five using Django, as these are two popular web application frameworks leveraging different development technologies.

In a second step, we integrated CodeQL and SonarCloud in GitHub workflows using a consistent setup process for the selected repositories. We standardized the configurations for both tools, minimizing variability caused by differences in repository structures, build environments, or dependency configurations. This involved defining uniform workflows, such as consistent trigger events (e.g., pull requests or pushes) and dependency management strategies. This controlled testing ensured that differences in setup or configuration across repositories would not impact the tools' performance or the accuracy of the vulnerability detection. During the setup phase, we documented the time required, steps taken, and any issues encountered, such as compatibility or configuration challenges. During the build phase, we applied the tools to identify vulnerabilities, generating logs and vulnerability data for analysis. In the reporting phase, we evaluated the usability of vulnerability reports based on clarity, comprehensiveness, and actionability.

In a final step we analyzed usability by evaluating the quality of notifications provided by CodeQL and SonarCloud during various stages of the workflow. This included assessing the clarity, specificity, and actionability of the messages, as well as their impact on resolving configuration errors, setup issues, and identifying bugs in the target repositories.

IV. RESULTS

We organized our observations along two dimensions: the impact of development environments on workflow setup and the quality of the notifications.

A. Impact of Development Environments on Workflow Setup

Setup Time and Configuration Complexity: Table I outlines the setup times and the number of configuration steps required for CodeQL and SonarCloud when applied to Spring and Django. A *step* refers to a configuration task that a user needs to complete to make the tool operational. Examples of these steps include setting up dependency resolution, adjusting custom property files, and integrating the tool into the project build pipeline.

For compiled languages like Java in the Spring Framework, additional setup challenges arise because the tools must analyze not only source code but also the compiled artifacts. CodeQL requires modifications to the GitHub workflow, by specifying the build commands (`mvn compile` or `gradle build` for Spring projects) to ensure CodeQL can generate a database for analysis. Moreover, dependency resolution must be configured to include all third-party libraries. SonarCloud also requires configuring the `sonar-project.properties` file with build instructions, paths to compiled artifacts, and by setting up the appropriate language analyzers for Java.

For Django projects, configurations are generally more straightforward. For CodeQL, workflow changes involve ensuring that Python dependencies are properly installed (e.g., `pip install -r requirements.txt`) and that the environment is prepared for the application's structure, such as setting up paths to source files and any custom configurations for the analysis. Similarly, for SonarCloud, enabling the Python plugin is essential. Additionally, the configuration must specify the exact paths to the source code directories (`sonar.sources`) and test directories (`sonar.tests`). Since Python is an interpreted language, there are no compiled artifacts to manage, and dependency resolution is less complex.

Impact of Build Environments: Workflows for Spring projects required additional adjustments to handle custom dependencies and configurations, particularly for SonarCloud. These steps introduced variations in setup processes due to differences in build tools and frameworks.

Dependency Management: In Spring, dependency management relies on build tools such as Maven[1] and Gradle.[2] Maven requires explicitly adding plugins like `sonar-maven-plugin` to `pom.xml` and configuring parameters such as `sonar.projectKey`, `sonar.organization`, and authentication tokens. These settings can be stored in external files like `settings.xml` or passed as command-line arguments. Gradle simplifies this process by embedding configurations inline within `build.gradle`, using the `org.sonarqube` plugin. In contrast, Django uses a straightforward approach with `requirements.txt` or `Pipfile` for dependencies and relies on environment variables for configurations, avoiding the need for build descriptors.

Build Tools and Mechanisms: Spring projects depend on build tools for defining the build lifecycle, from dependency resolution to generating artifacts. Maven's XML-based `pom.xml` files are verbose and can be complex, while Gradle's Groovy and Kotlin DSL in `build.gradle` offer flexibility but require familiarity with their syntax. These build tools introduce variability and complexity depending on the tool chosen. Django, in contrast, does not require specialized build tools. Dependency resolution is managed through a `pip install` process, making the workflows simpler and avoiding the configuration complexity.

[1]https://maven.apache.org/

[2]https://gradle.org/

Code Analysis Preparation: For Spring projects, bytecode generation is a prerequisite for CodeQL. This involves running specific build commands such as `mvn clean install` for Maven or `gradle build` for Gradle to ensure that compiled Java classes are available for analysis. This step can be time-consuming and error-prone if the environment is not configured correctly. For Django, however, bytecode generation is optional. CodeQL and SonarCloud can directly analyze Python source files, removing the need for intermediate steps.

Configuration Complexity: Spring projects involved more configuration efforts. Maven requires defining dependencies and plugins in `pom.xml` and sometimes `settings.xml` for external settings such as authentication tokens. Gradle adds configurations within `build.gradle`, but inline settings are also complex to manage. Workflow execution in Spring typically involved configuring dependencies, generating bytecode, and then running analysis. Django, by contrast, involved minimal configuration. Most setups relied on environment variables and single command-line invocations.

Lesson: Programming languages and frameworks influenced the setup process, with Spring projects requiring more steps and complexity than Django projects. Spring workflows for SonarCloud involved higher configuration effort due to dependency resolution and build tool variability, while Django's simpler dependency management and direct analysis capabilities resulted in a simpler process with fewer opportunities for introducing errors.

B. Quality of Notifications

Table II provides an overview of both the total number of notifications and the number of unclear notifications detected by CodeQL and SonarCloud. We define *unclear notifications* through personal assessment as those that lack sufficient detail or context to be easily understood and actionable. These notifications often use vague or generic language, such as "Configuration error detected," without specifying the exact issue, its location, or the steps to resolve it.

Notifications During Workflow Setup and Error Resolution: When setting up workflows for Spring projects using CodeQL, the notifications related to configuration errors were specific and actionable. For example, when an environment variable like `DATABASE_URL` was missing, CodeQL provided a clear error message such as: "Environment variable `DATABASE_URL` is missing or improperly set in `settings.json`. Ensure a valid URL format is used." This allowed to identify the issue and resolve it.

For SonarCloud in Spring projects, the notifications during setup were less descriptive. For instance, if the `sonar.projectKey` parameter was missing in the pom.xml file, the warning only stated: "Configuration error detected in `pom.xml`. Review the configuration settings." This generic message did not specify which key or parameter was causing the issue, leading to longer troubleshooting times.

In Django projects, CodeQL also provided detailed guidance during setup. For instance, if dependencies in

TABLE II
NUMBER OF NOTIFICATIONS AND UNCLEAR NOTIFICATIONS IN DJANGO AND SPRING REPOSITORIES WITH LOC

Repository	Framework	LOC	CodeQL Notifications				SonarCloud Notifications			
			Setup/ Errors	Bugs/ Analysis	Unclear Setup	Unclear Bugs	Setup/ Errors	Bugs/ Analysis	Unclear Setup	Unclear Bugs
django-beginners-guide	Django	1240	4	6	1	2	6	9	3	3
Django-Projects-for-beginners	Django	13059	6	21	2	3	9	37	3	4
DJANGO_COURSE	Django	5165	6	8	2	3	8	12	7	1
DJANGO-TUTORIAL	Django	1143	1	4	0	0	5	11	2	4
django-intro-tutorial	Django	193	1	2	0	1	2	2	0	0
gs-serving-web-content	Spring	84	0	0	0	0	3	1	1	0
gs-rest-service	Spring	85	-	-	-	-	0	0	0	0
gs-spring-boot	Spring	102	-	-	-	-	3	4	1	2
tutorials	Spring	1164	10	99	3	6	-	-	-	-
gs-consuming-rest	Spring	88	1	1	0	0	0	1	0	0

- Indicates that the workflow did not run successfully for the project.

`requirements.txt` were not properly installed, the notification stated: "Dependency django-extensions missing. Run `pip install -r requirements.txt` to resolve.

Notifications on Bugs and Analysis Results: For actual bug detection and analysis, CodeQL produced precise and actionable insights for both Spring and Django projects. In a Spring project, CodeQL flagged a SQL injection vulnerability with the message: "Potential SQL injection detected in function `validate_user_input()` on `line 45`. Validate input using parameterized queries." This notification contains the line number and function, along with remediation steps. Similarly, for Django, a message like: "Cross-site scripting vulnerability detected in `views.py`, `line 102`. Sanitize user inputs before rendering in templates," provided both the location and the nature of the issue.

SonarCloud's notifications on bugs were comparatively less detailed. For example, in a Spring project, a warning about input handling was phrased as: "Enable server hostname verification on this SSL/TLS connection." This generic message did not specify the specific line number, function. In Django projects, a similar pattern emerged, with warnings such as: "Revoke and change this password, as it is compromised." that offered no contextual details about where the issue occurred.

Lesson: CodeQL provided detailed and precise notifications across both Python and Java projects. In contrast, SonarCloud notifications were overall less specific.

V. CONCLUSION

This experience report highlights the usability challenges and integration complexities of CodeQL and SonarCloud within GitHub workflows, for Spring and Django projects.

While Django projects could be integrated with reasonable effort, Spring projects involved challenges due to their more complex build environments. Unclear error messages and inconsistent notifications in both tools often hindered implementation efficiency, underscoring the need for better-designed feedback mechanisms. Future work could explore automating configuration detection, improving error message clarity. Additionally, conducting studies across diverse programming languages and frameworks would provide deeper insights into optimizing static application security testing tools for continuous integration workflows and proactive vulnerability mitigation.

ACKNOWLEDGMENT

This work is funded by NSERC

REFERENCES

[1] M. Souppaya, K. Scarfone, and D. Dodson, "Secure software development framework (ssdf) version 1.1: Recommendations for mitigating the risk of software vulnerabilities," Tech. Rep. SP 800-218, National Institute of Standards and Technology (NIST), 2022. Supersedes: CSWP 13 (04/23/2020).

[2] M. Esposito, V. Falaschi, and D. Falessi, "An extensive comparison of static application security testing tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, pp. 69–78, 2024.

[3] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen, "Comparison and evaluation on static application security testing (sast) tools for java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pp. 921–933, 2023.

[4] G. Piskachev, M. Becker, and E. Bodden, "Can the configuration of static analyses make resolving security vulnerabilities more effective? - a user study," *Empirical Softw. Engg.*, vol. 28, sep 2023.

[5] M. Tahaei, K. Vaniea, K. K. Beznosov, and M. K. Wolters, "Security notifications in static analysis tools: Developers' attitudes, comprehension, and ability to act on them," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, 2021.

[6] H. Assal and S. Chiasson, "Security in the software development lifecycle," in *Fourteenth symposium on usable privacy and security (SOUPS 2018)*, pp. 281–296, 2018.

[7] J. Smith, L. N. Q. Do, and E. R. Murphy-Hill, "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security," in *SOUPS @ USENIX Security Symposium*, 2020.

[8] OWASP, "Owasp application security verification standard (asvs)," 2024. Accessed: 2024-08-07.

[9] GitHub, *Workflow syntax for GitHub Actions*, 2023. Accessed: 2024-11-07.

[10] GitHub, "About code scanning," 2024. Accessed: 2024-11-07.

[11] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pp. 532–543, 2022.

[12] L. Nurgalieva, A. Frik, and G. Doherty, "A narrative review of factors affecting the implementation of privacy and security practices in software development," *ACM Comput. Surv.*, vol. 55, jul 2023.

[13] R. Walker, M. Cooke, A. Henderson, and D. K. Creedy, "Characteristics of leadership that influence clinical learning: a narrative review," *Nurse education today*, vol. 31, no. 8, pp. 743–756, 2011.