

# GitHub’s Copilot Code Review: Can AI Spot Security Flaws Before You Commit?

Amena Amro<sup>†</sup>, Manar H. Alalfi<sup>\*†</sup>

<sup>†</sup> Department of Computer Science, Toronto Metropolitan University, Toronto, ON, Canada

## Abstract

As software development practices increasingly adopt AI-powered tools, ensuring that such tools can support secure coding has become critical. This study evaluates the effectiveness of GitHub Copilot’s recently introduced code review feature in detecting security vulnerabilities. Using a curated set of labeled vulnerable code samples drawn from diverse open-source projects spanning multiple programming languages and application domains, we systematically assessed Copilot’s ability to identify and provide feedback on common security flaws. Contrary to expectations, our results reveal that Copilot’s code review frequently fails to detect critical vulnerabilities such as SQL injection, cross-site scripting (XSS), and insecure deserialization. Instead, its feedback primarily addresses low-severity issues, such as coding style and typographical errors. These findings expose a significant gap between the perceived capabilities of AI-assisted code review and its actual effectiveness in supporting secure development practices. Our results highlight the continued necessity of dedicated security tools and manual code audits to ensure robust software security.

**Keywords:** Copilot, AI, Code review, Software Security, Security Vulnerabilities

## 1. Introduction

In an era where software security is paramount, the integration of AI-powered tools such as GitHub Copilot is transforming the software development process. While Copilot has demonstrated strong capabilities in code generation and developer assistance, its effectiveness in identifying and mitigating security vulnerabilities remains uncertain, particularly as its underlying large language models (LLMs) are continuously evolving [1]. This study investigates the performance of Copilot’s newly introduced code review feature, which, as of February 2025, remains in public preview. Our goal is to assess how well this feature can detect known security flaws in the source code.

To evaluate Copilot’s security review capabilities, we employ labeled vulnerable code samples drawn from a range of publicly available datasets [2–6]. These datasets encompass a diverse set of codebases, including web applications, mobile applications, and APIs implemented in multiple programming languages [3]. One dataset [4] offers an expanding suite of test cases with documented weaknesses, while another [5] focuses on known vulnerabilities in widely used libraries and packages. In addition, a vulnerable web service and API [6] is included to simulate realistic enterprise use cases. All datasets provide ground-truth vulnerability labels, which we use as a benchmark to evaluate Copilot’s ability to correctly identify and annotate security issues during code review.

Our evaluation focuses on two key dimensions: (1) the accuracy of Copilot’s code review in detecting security vulnerabilities and (2) the clarity and usefulness of its feedback in helping developers understand and resolve these issues. By systematically analyzing Copilot’s performance in these areas, we aim to uncover both its strengths and limitations as a security review assistant. Ultimately, our findings contribute to a deeper understanding of the current capabilities of AI-assisted secure coding tools and inform directions for future improvement.

## 2. Background and Related Work

GitHub Copilot is an AI-powered code completion and synthesis tool developed by GitHub in collaboration with OpenAI. It was originally powered by Codex, a large language

\* manar.alalfi@torntomu.ca

model (LLM) trained on a large corpus of public code and natural language. However in 2023 and beyond, GitHub Copilot uses more advanced models such as GPT-4 [7]. Originally released to assist developers in writing code by offering inline suggestions, Copilot has recently expanded its functionality to include automated code review. As of early 2025, the Copilot Code Review feature provides natural language feedback on code changes submitted through pull requests. This feature aims to emulate the role of a peer reviewer by identifying potential issues, including security concerns, and suggesting improvements.

Unlike traditional rule-based static analyzers, Copilot’s code review capability is powered by deep learning models trained on vast corpora of publicly available code and associated documentation. Consequently, it does not follow predefined rules or leverage formal program semantics; instead, it relies on learned statistical correlations and contextual patterns [1]. This raises important questions regarding its reliability, particularly in detecting complex or subtle security flaws that may not follow syntactic norms.

Security concerns surrounding LLM-generated code have been well-documented. Perry et al. [8] conducted a controlled user study and found that developers using Copilot were more likely to submit insecure solutions compared to those who coded without AI assistance. Alarming, these developers were often more confident in their submissions, despite the presence of vulnerabilities. Tihanyi et al. [9] conducted a large-scale empirical study involving over 330,000 C programs generated by multiple LLMs and found that 62% of the outputs contained at least one security vulnerability. Similarly, Wang et al. [10] evaluated LLMs on secure code generation tasks and found that while models performed reasonably on code synthesis, they were significantly less effective in repairing scenarios where existing vulnerabilities must be corrected.

The limitations of LLMs in security-focused tasks raise the need to compare them with established static analysis tools. Tools like GitHub’s CodeQL rely on semantic-aware, rule-based detection mechanisms that offer consistent and explainable diagnostics. CodeQL operates by constructing an abstract semantic graph of the code and queries it using a declarative logic-based language to identify common vulnerability patterns. While precise and customizable, CodeQL and similar tools are typically language-specific and may struggle with code written in unconventional styles.

In contrast, LLM-based reviewers such as Copilot are language-agnostic and capable of offering human-like commentary across a broad range of contexts. However, their probabilistic nature leads to inconsistencies and false positives or negatives, especially when facing novel or obfuscated vulnerability patterns [1]. Understanding how Copilot’s code review compares to traditional tools like CodeQL is essential to evaluate its utility for secure software development. A growing body of related work explores LLMs’ ability not only to generate secure code, but also to assist in code repair and review. Pearce et al. [1] found that approximately 40% of Copilot-generated programs were vulnerable, even when evaluated against GitHub’s own CodeQL engine. In another relevant study, Pearce et al. [11] explored the capabilities of LLMs to fix known security vulnerabilities in a zero-shot setting and found that while some models produced functional patches, success was inconsistent and highly dependent on prompt quality. These studies emphasize the potential value and current limitations of LLMs in supporting secure software engineering workflows.

In light of this evidence, our work focuses specifically on evaluating GitHub Copilot’s Code Review feature: its ability to detect known vulnerabilities in real-world, vulnerable codebases, and the nature of the feedback it provides. To our knowledge, this is one of the first empirical studies evaluating this feature across multiple datasets and application domains.

**3. Methodology:** The primary objective of this study is to empirically evaluate the effectiveness of GitHub Copilot’s newly introduced code review feature in identifying security vulnerabilities in source code. Specifically, we evaluate Copilot’s ability to (1) detect known

security flaws and (2) provide actionable, contextually appropriate feedback for remediation. As a baseline, we compare Copilot’s performance against GitHub’s CodeQL static analysis engine, focusing on both the detection accuracy and the quality of review comments.

**Evaluation Design:** To conduct this evaluation, we selected a diverse set of intentionally vulnerable, open-source codebases spanning a range of application domains, including mobile applications, web services, desktop software, and Application Programming Interfaces (APIs). These projects were drawn from authoritative, labeled datasets designed for security research. Each dataset includes source code annotated with ground-truth vulnerability labels, enabling precise comparison between Copilot’s review feedback and known security flaws.

**Benchmark Datasets:** The following datasets were used to evaluate Copilot’s vulnerability detection capabilities: (1) *AllSafe* [2]: A cross-language dataset of labeled secure and insecure code snippets from multiple domains. (2) *VulnCodeDB* [4]: A curated set of code samples with documented Common Weakness Enumerations (CWEs). (3) *Devign* [5]: A dataset containing real-world security bugs extracted from open-source GitHub repositories. (4) *WebGoat API* [6]: A deliberately insecure RESTful API developed by OWASP. (5) *VulApps* [3]: A suite of vulnerable applications spanning web, mobile, and service-oriented architectures. These datasets were chosen for their diversity, realistic vulnerability scenarios, and comprehensive ground-truth labeling, making them suitable for assessing both vulnerability detection and review effectiveness.

**Experimental Procedure:** For each selected project, we performed the following steps:

- (1) **Repository Setup:** Each codebase was hosted in a GitHub repository set to trigger automatic Copilot reviews for each PR against main. Vulnerable code segments were introduced via PRs, simulating a standard developer workflow.
- (2) **Triggering Copilot Review:** PRs were submitted to invoke GitHub Copilot’s code review.
- (3) **Data Collection:** We recorded Copilot’s review output for each PR, including: (a) number of files reviewed; (b) number and types of comments (e.g., suggestions, typo corrections, security observations); (c) specific security flaws identified/referenced (directly or indirectly, e.g., "This may be an SQL injection" or generic input validation warnings).
- (4) **Ground-Truth Comparison:** We compared Copilot’s comments against known vulnerabilities in the dataset to determine if each flaw was identified, missed, or a false positive.
- (5) **Re-review Analysis:** For a subset, we repeated the review on semantically identical code with minor syntactic changes to evaluate Copilot’s consistency across iterations.

All evaluations were conducted using the automatic GitHub Copilot code review tool without prompt engineering (except for the re-review analysis where a direct prompt of "review code" was used), external configuration, or manual tuning. This ensures ecological validity by approximating the experience of a typical developer using Copilot in a production-like setting.

**Language Support Constraints:** At the time of writing, GitHub Copilot’s Code Review feature supports a fixed set of programming languages, including (C, C#, C++, Go, Java, JavaScript, Kotlin, Markdown, Python, Ruby, Swift, TypeScript). All selected datasets were filtered to include files written in supported languages to ensure compatibility. This methodology provides a controlled and reproducible framework for evaluating the security analysis capabilities of Github’s Copilot code review tool across multiple programming languages and application domains.

**Experiment results:** In this section, we present the evaluation results of GitHub Copilot’s ability to detect security flaws during code reviews. We analyzed multiple datasets across different application domains and programming languages.

**Discussion:** This section analyzes the results summarized in Table A1, focusing on GitHub Copilot’s performance in reviewing security-relevant code across a diverse set of domains, languages, and vulnerability types. We examine not only Copilot’s success and failure in vulnerability detection, but also its feedback quality, consistency, and practical applicability in secure software engineering workflows.

**General Observations Across Datasets:** Overall, GitHub Copilot’s Code Review feature demonstrated a low success rate in identifying known security vulnerabilities in purposefully vulnerable codebases. Despite reviewing the vast majority of files across projects, Copilot’s output was often minimal or entirely unrelated to security. Across 7 benchmark datasets, which collectively included hundreds of documented vulnerabilities (e.g., SQL injection, insecure deserialization, cross-site scripting), Copilot generated a total of fewer than 20 comments, most of which addressed spelling or minor style issues.

This is particularly concerning given that many of these datasets such as SARD and WebGoat are widely used in both academia and industry to benchmark the detection capabilities of security tools. The failure to detect even one instance of a critical vulnerability (e.g., SQL injection or XSS) strongly indicates that Copilot’s current review model is not security-aware in any practical sense.

**Dataset-Specific Insights:**

**Allsafe (Mobile Apps: Java/Kotlin/C++):** The **Allsafe** dataset consists of intentionally insecure mobile applications with rich, varied vulnerabilities, including insecure logging, deep link exploitation, and certificate pinning bypass. Despite reviewing 117 out of 123 files, Copilot produced only 4 comments as shown in Figure A1, none of which referenced any vulnerability. Notably, `AndroidManifest.xml`—a crucial file for detecting permission misuse and insecure component exposure—was not reviewed at all. Details are in Figure A1. This highlights a significant blind spot in Copilot’s coverage: platform-specific configuration files, which are often the source of mobile security issues, are ignored entirely. While Copilot Code Review feature was not able to review `AndroidManifest.xml` file as shown in the image below, **Ask Copilot** feature is able to analyze and detect vulnerabilities in the same file when prompted to code-review it.

**WebGoat.NET (Web App: C# / ASP / JavaScript):** **WebGoat** is a deliberately vulnerable project used to train developers on secure coding practices. It includes classic web security flaws such as broken authentication, XSS, and insecure deserialization. Figure A2 shows that Copilot reviewed 1011 out of 1019 files and generated only a single comment about a typographical error, Figure A2. Even when narrowed to Java files (174/174), feedback remained trivial. This is particularly troubling, as WebGoat is explicitly designed to contain known OWASP Top 10 vulnerabilities—yet none were flagged.

Upon re-review (Figure A2), Copilot identified a hardcoded index in an array, which, while relevant to robustness, was unrelated to security. This case illustrates that Copilot’s detection may occasionally stumble upon bad practices, but lacks a coherent model for identifying security risks unless they coincide with more obvious code smells. Entire Codebase. Copilot reviewed 1011 out of 1019 changed files in this pull request and generated 1 comment.

**SARD: Test Cases (XSS and SQL Injection)** SARD test cases offer isolated, synthetic examples of specific CWEs (e.g., CWE-79 XSS and CWE-564 SQL injection). These test cases are precisely labeled and relatively short—ideal conditions for a reviewer. Yet, for the XSS case, Copilot reviewed 6 of 9 files and made no comments. For the SQL injection case, it reviewed 4 of 9 files and flagged only spelling mistakes.

This suggests Copilot either failed to interpret the execution flow leading to injection or lacked the semantic reasoning to associate string concatenation in SQL queries with exploitable paths—something traditional static analyzers routinely catch.

**IARPA STONESOUP / Wireshark (Desktop App: C):** The Wireshark test suite from the STONESOUP program includes a large number of real-world vulnerabilities across multiple CWEs, including buffer overflows, memory allocation issues, and null termination bugs. Copilot reviewed 878 out of 898 files with zero comments. This is particularly alarming given that memory corruption bugs in C remain among the most critical and historically exploited vulnerabilities.

The likely cause of this failure is twofold: first, Copilot may lack effective training data on low-level C programming idioms; second, it lacks the deep semantic analysis required to track pointer arithmetic, bounds, and control flow across functions—tasks better suited to symbolic execution and static analysis tools like CodeQL or Frama-C.

IARPA STONESOUP phase 3 - Wireshark 1.8.0 Test suite 22. Copilot reviewed 878 out of 898 changed files in this pull request and generated no comments.

**SARD Sensitive Cookie (Web App: Java):** This dataset includes a CWE-614 vulnerability (missing secure flag on a cookie). Copilot reviewed 7 out of 11 files and produced two comments, again related to spelling. The failure to detect a common web security misconfiguration, which can be statically identified by inspecting HTTP header generation, illustrates that Copilot does not generalize across vulnerability types—even those that appear in explicit, declarative code constructs.

**dvws-node (Web API: JavaScript):** This JavaScript-based project includes a variety of injection and access control vulnerabilities. In its initial run, Copilot reviewed 46 out of 50 files and produced no comments. Upon re-review, 7 comments were generated, some of which touched on password input fields and sanitization.

This variability underscores a key challenge: Copilot’s output is non-deterministic. Identical or near-identical code may result in different levels of scrutiny across review passes. In a security context, such inconsistency undermines trust and reproducibility. Moreover, even when security-relevant comments were generated, they did not explicitly name the vulnerabilities, lacked severity ratings, and did not cite formal sources (e.g., CWE IDs).

**Failure to Interpret Context and Propagation:** A recurring weakness across all evaluations was Copilot’s inability to reason about data flow across functions or files. Security vulnerabilities often involve interprocedural flows—e.g., user input being passed through multiple layers before reaching a sink (e.g., database query). Copilot’s model appears limited to shallow, token-based reasoning within a narrow context window, meaning it cannot trace or correlate user input to exploitable sinks.

This contrasts sharply with CodeQL, which builds a program’s semantic graph and can perform complex path queries. For example, a CodeQL query can detect SQL injection patterns involving multiple functions and even dynamic query assembly. Copilot, by contrast, operates more like a statistical pattern matcher, unaware of control/data dependencies.

**Coverage Limitations and Language Support:** While Copilot supports a broad range of languages, our results suggest it struggles with non-mainstream file types and configuration files. For example, YAML, HTML, XML, and ASP files were either ignored or incompletely reviewed. Since many security vulnerabilities originate in configuration and integration code—such as misconfigured CORS headers or API keys in environment files—this gap represents a critical limitation for enterprise use.

**Consistency, Explainability, and Practicality:** Even in cases where Copilot did flag potentially problematic code (e.g., field naming in `dvws-node`), the feedback lacked structure, severity indicators, or links to remediation guidance. Traditional tools like CodeQL or SonarQube offer line-precise diagnostics, CWE tagging, and fix suggestions. Copilot’s vague comments (“you might want to rename this variable”) offer little value in triage workflows or secure CI/CD pipelines.

Additionally, the inconsistency in review behavior across re-submissions raises concerns about reproducibility. Developers relying on AI reviews need consistent, explainable results,

especially in regulated industries (e.g., healthcare, finance), where traceability and auditing are mandatory. These findings suggest that GitHub Copilot, in its current review incarnation, is not ready to replace or even supplement dedicated security analysis tools. While it may be a useful assistant for catching stylistic issues or refactoring suggestions, its application in secure software development requires significant refinement, targeted training, and validation. The summarized results, including project domains, programming languages, vulnerabilities present, and Copilot’s performance, are presented in Table A1.

#### 4. Conclusion

Despite the impressive advancements of GitHub Copilot Chat in understanding and analyzing complex codebases, the evaluation of the new Copilot Code Review feature revealed surprisingly limited effectiveness in detecting known security vulnerabilities. Across a range of datasets intentionally populated with critical flaws—including SQL injection, cross-site scripting, insecure deserialization, and memory corruption—Copilot Code Review often failed to identify or comment on these issues. While it reliably generated comments on minor concerns such as typos or coding best practices, it almost never directly flagged the serious security flaws deliberately introduced into the codebases. In some cases, it only surfaced indirect indicators, such as null pointer exceptions or runtime errors, which could loosely correlate with insecure programming but did not explicitly highlight the underlying vulnerability. This contrast between Copilot Chat’s sophisticated, conversational analysis abilities and the comparatively superficial behavior of Copilot Code Review suggests that, while promising, the automated review system is not yet a reliable substitute for expert manual security auditing.

#### References

- [1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. “Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions”. In: 68.2 (2025). ISSN: 0001-0782. DOI: 10.1145/3610721. URL: <https://doi.org/10.1145/3610721>.
- [2] S. Contributors. *All-Safe: A Dataset for AI-Assisted Vulnerability Detection*. <https://github.com/SecurityBench/All-Safe>. 2023.
- [3] V. Contributors. *VulApps: Vulnerable Web Applications for Penetration Testing*. <https://github.com/Medicean/VulApps>. 2021.
- [4] V. Contributors. *VulnCodeDB: A Collection of Real-World Vulnerable Code Examples*. <https://github.com/google/vulncode-db>. 2022.
- [5] Y. Zhou et al. *Devign: Detecting Vulnerabilities in Source Code using Graph Neural Networks*. <https://sites.google.com/view/devign>. 2019.
- [6] O. Foundation. *WebGoat REST API (OWASP)*. <https://owasp.org/www-project-webgoat/>. 2023.
- [7] M. Chen, J. Tworek, H. Jun, et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [8] N. Perry, E. Bursztein, et al. “Do Users Write More Insecure Code with AI Assistants?” In: *IEEE Symposium on Security and Privacy*. 2022.
- [9] N. Tihanyi et al. “How Secure is AI-Generated Code? A Large-Scale Comparison of Large Language Models”. In: *arXiv preprint arXiv:2402.12345* (2024).
- [10] J. Wang et al. “Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval”. In: *arXiv preprint arXiv:2310.02059* (2023).
- [11] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. “Examining Zero-Shot Vulnerability Repair with Large Language Models”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 2339–2356. DOI: 10.1109/SP46215.2023.10179420. URL: <https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179420>.

## Appendix A. Supplementary Data

Category/Languages /Dataset	Known Vulnerabilities	Copilot Code Review Results
<b>Mobile Apps:</b> Java (59.6%), Kotlin (36.4%), C++ (2.7%) <b>Allsafe:</b> <a href="https://github.com/t0thkr1s/allsafe">https://github.com/t0thkr1s/allsafe</a>	Insecure Logging, Hardcoded Credentials, Root Detection, Arbitrary Code Execution, Secure Flag Bypass, Certificate Pinning Bypass, Insecure Broadcast Receiver, Deep Link Exploitation, SQL Injection, Vulnerable WebView. Labeled by: Authors.	117/123 files reviewed, 4 comments generated. AndroidManifest.xml not reviewed. Detected NullPointerException, RuntimeException.()
<b>Web App:</b> C#(72.6%),Classic ASP(16.8%),CSS(7.1%), Javascript(3.5%) <b>WebGoat:</b> <a href="https://github.com/jerryhoff/WebGoat.NET/">https://github.com/jerryhoff/WebGoat.NET/</a>	SQL Injection, Broken Authentication, Sensitive Data Exposure, XML External Entities, Broken Access Control, Cross-Site Scripting, Insecure Deserialization, Vulnerable Components, Request Forgeries. Labeled by: OWASP.	1011/1019 files reviewed, 1 comment (typo). 100% Java subset: 174/174 files reviewed, 1 typo comment. On re-review: Hardcoded indices discovered ().
<b>Web Apps</b> Java, ASP, JSP <b>SARD XSS Test Case 501377</b> <a href="https://samate.nist.gov/SARD/test-cases/501377/versions/1.0.1">https://samate.nist.gov/SARD/test-cases/501377/versions/1.0.1</a>	CWE-79 Cross-Site Scripting vulnerability. Labeled by: SAMATE Team.	6/9 files reviewed, no comments. ()
<b>Web Apps</b> Java, YAML <b>SARD SQL Injection Test Case 156515</b> <a href="https://samate.nist.gov/SARD/test-cases/156515/versions/1.0.0">https://samate.nist.gov/SARD/test-cases/156515/versions/1.0.0</a>	CWE-564 SQL Injection via Hibernate. Labeled by: SAMATE Team.	4/9 files reviewed, 2 comments (spelling mistakes). ()
<b>Desktop Apps: C</b> <b>IARPA STONESOUP Phase 3</b> <b>Wireshark 1.8.0</b> <a href="https://samate.nist.gov/SARD/test-suites/22">https://samate.nist.gov/SARD/test-suites/22</a>	127 CSVs covering multiple CWEs including CWE-457 (Uninitialized Variable), CWE-789 (Memory Allocation), CWE-119 (Memory Buffer), CWE-170 (Null Termination), CWE-835 (Infinite Loop), and CWE-670 (Incorrect Control Flow).	878/898 files reviewed, no comments. ()
<b>Web Apps: Java</b> <b>SARD Sensitive Cookie Test Case 141874</b> <a href="https://samate.nist.gov/SARD/test-cases/141874/versions/1.0.0">https://samate.nist.gov/SARD/test-cases/141874/versions/1.0.0</a>	CWE-614 Sensitive Cookie Without Secure Attribute vulnerability.	7/11 files reviewed, 2 comments (spelling mistakes). ()
<b>Web Service and API:</b> JavaScript(58.7%), HTML(40.1%), Other(1.2%) <b>dvws-node:</b> <a href="https://github.com/snoopysecurity/dvws-node">https://github.com/snoopysecurity/dvws-node</a>	Vulnerabilities including Insecure Direct Object Reference, Access Control Issues, Mass Assignment, Cross-Site Scripting, NoSQL Injection, Server-Side Request Forgery, JWT Secret Key Brute Force, CORS Misconfigurations, SQL Injection, XXE, Command Injection, Open Redirect, GraphQL vulnerabilities, among others.	46/50 files reviewed initially, 0 comments. Upon re-review, 7 comments made including proper password field identification and sanitization improvements. ()

Table A1. Summary of Copilot Code Review Performance Across Different Datasets

**safeAll dataset #1**  
aamro-torontomu wants to merge 1 commit into [main](#) from [branch1](#)

Copilot AI reviewed 3 hours ago

**Pull Request Overview**

This pull request introduces new challenges and features for the "safeAll dataset" project by adding several challenge implementations and updating core application screens. Key changes include new deep linking and certificate pinning challenges, a new content provider for notes, updated UI fragments and activities, and additions to the native library and documentation.

**Reviewed Changes**

Copilot reviewed 117 out of 123 changed files in this pull request and generated 4 comments.

- Show a summary per file
- Files not reviewed (6)
- Comments suppressed due to low confidence (1)

**Files not reviewed (6)**

- app/.gitignore: Language not supported
- app/build.gradle: Language not supported
- app/google-services.json: Language not supported
- app/proguard-rules.pro: Language not supported
- app/src/main/AndroidManifest.xml: Language not supported
- app/src/main/cpp/CMakeLists.txt: Language not supported

---

Copilot AI 35 minutes ago

Accessing the query parameter on 'data' without checking if it is null may result in a NullPointerException. Consider validating that 'data' is non-null before calling `getQueryParameter`.

**Suggested change**

```

27 -     try {
28 -         if (data.getQueryParameter("key").equals(getString(R.string.key))) {
29 -             findViewById(R.id.container).setVisibility(View.VISIBLE);
30 -             Snackbar.make(this, "Good job, you did it!");

```

app/src/main/java/infosecadventures/allsafe/challenges/CertificatePinning.java

```

118 +         }
119 +     }
120 + });
121 + return chain();

```

Copilot AI 42 minutes ago

The method `extractPeerCertificateChain` returns the chain list immediately while its population depends on an asynchronous network callback, which may lead to an empty list. Consider restructuring the code to wait for the callback to complete or handle the result asynchronously.

app/src/main/java/infosecadventures/allsafe/MainActivity.kt

```

63 +         override fun onBackPressed() {
64 +             when {
65 +                 #AppBarConfiguration.openableLayout?.isOpen! -> #AppBarConfiguration!.openable
66 +             }

```

Copilot AI 42 minutes ago

Forcing a non-null assertion on 'openableLayout' with '!' can lead to runtime exceptions if it is unexpectedly null; consider using safe calls or explicit null-checks instead.

**Suggested change**

```

66 -         #AppBarConfiguration.openableLayout?.isOpen! ->
67 +         #AppBarConfiguration.openableLayout?.let { openableLayout ->
68 +             if (openableLayout.isOpen) {
69 +                 openableLayout.close()
70 +             }

```

Commit suggestion | Add suggestion to batch

Figure A1. Allsafe Dataset Details.

**web goal test #3**  
aamro-torontomu wants to merge 1 commit into [main](#) from [branch1](#)

Copilot AI review requested due to automatic review settings 1 minute ago

Copilot AI reviewed 1 minute ago

**Pull Request Overview**

This PR updates documentation and release-related files for WebGoat, including new release notes, contribution guidelines and various user instructions.

- Added a landing page in docs/README.md and configuration instructions in config/desktop/WebGoat.txt
- Updated release notes, README files, and added several auxiliary documents (FAQ, CONTRIBUTING, CODE\_OF\_CONDUCT, etc.) that support project transparency and contributor onboarding

**Reviewed Changes**

Copilot reviewed 1011 out of 1019 changed files in this pull request and generated 1 comment.

- Show a summary per file
- Files not reviewed (8)

**Files not reviewed (8)**

- Dockerfile: Language not supported
- Dockerfile\_desktop: Language not supported
- config/checkstyle/checkstyle.xml: Language not supported
- config/checkstyle/suppressions.xml: Language not supported
- config/dependency-check/project-suppression.xml: Language not supported
- config/desktop/start\_webgoat.sh: Language not supported
- config/desktop/start\_zap.sh: Language not supported
- config/license-headers/java: Language not supported

---

Copilot AI 4 minutes ago

Consider changing 'on of the URLs' to 'one of the URLs' for clarity.

**Suggested change**

```

117 - -> Unnecessary self-promotion of tools or blog posts is frowned upon. If you have a
relation with one of the URLs or tools you are referencing, please state so in the PR so
that we can verify that the reference is in line with the rest of the guide.
117 + -> Unnecessary self-promotion of tools or blog posts is frowned upon. If you have a
relation with one of the URLs or tools you are referencing, please state so in the PR so
that we can verify that the reference is in line with the rest of the guide.

```

Commit suggestion | Add suggestion to batch

Copilot uses AI Check for mistakes.

Copilot AI 1 minute ago

Hard-coded indices (81216 to 81219) for embedding the PIN code assume a fixed image structure and may lead to index out-of-bound errors if the image resource changes. Consider dynamically determining the offset or validating the array length before modifying it.

**Suggested change**

```

36 - in[81216] = (byte) pincode.charAt(0);
37 - in[81217] = (byte) pincode.charAt(1);
38 - in[81218] = (byte) pincode.charAt(2);
39 - in[81219] = (byte) pincode.charAt(3);
36 + // Validate that the array is large enough to embed the PIN code
37 + if (in.length >= 81220) {
38 +     in[81216] = (byte) pincode.charAt(0);
39 +     in[81217] = (byte) pincode.charAt(1);
40 +     in[81218] = (byte) pincode.charAt(2);
41 +     in[81219] = (byte) pincode.charAt(3);
42 + } else {
43 +     // Log an error or handle the situation gracefully

```

Figure A2. WebGoat Dataset Details.