Academic Research Journal of Science and Technology
Nallamuthu Gounder Mahalingam College
publications.ngmc.ac.in
e-ISSN: 3048-9644

BugShield: AI-Based Web Attack Surface Mapper For Automated Security Assessment

Ritesh Thorve[1,*], Amruta Jawade[2]

[1] Department of Master of Computer Applications, Anantrao Pawar College of Engineering & Research (APCOER), Pune

[2] Assistant Professor at Department of Master of Computer Applications, Anantrao Pawar College of Engineering & Research (APCOER), Pune

***Corresponding author:** Ritesh Thorve, riteshthorve387@gmail.com

Abstract

Web application security is an important concern in modern software development because insecure coding practices and exposed credentials can introduce risks to application confidentiality, integrity, and availability. This paper presents BugShield, a source-code security scanning and findings-management platform designed to support preliminary security assessment of web applications. The system combines project management, automated source-code analysis using implemented detection rules and integrated security tools, vulnerability classification, structured findings, dashboard visualization, AI-assisted explanations, and PDF report generation. The platform aims to help developers and security researchers identify potential weaknesses, review supporting code evidence, prioritize findings, and understand possible remediation measures. The evaluation methodology considers detection outcomes against known test cases, severity classification, scan execution time, and report-generation functionality. Experimental findings are to be reported using verified measurements collected during testing. The system's capabilities depend on supported programming languages, implemented detection rules, and integrated tools. Potential false positives and false negatives require further verification, and source-code scanning alone cannot identify every runtime or business-logic vulnerability. BugShield is intended to complement established security testing practices by providing an integrated workflow for preliminary vulnerability analysis and findings management.

Keywords: Web Application Security; Vulnerability Detection; Static Analysis; Attack Surface Analysis; Secure Coding; AI-Assisted Security

Received: 23 October 2026 · Accepted: 3 November 2026 · Published: 3 November 2026

How to cite this article: Ritesh Thorve, & Amruta Jawade (2026). BugShield: AI-Based Web Attack Surface Mapper For Automated Security Assessment. *Academic Research Journal of Science and Technology*, *5*(07). https://doi.org/10.63300/arjst0507112026.01

Copyright and licence: © 2026 The Author(s). This is an open access article distributed under the terms of the Creative Commons Attribution 4.0 International License (CC BY 4.0), which permits unrestricted use, distribution and reproduction in any medium, provided the original work is properly cited.

1. Introduction

Web applications handle sensitive data and are continuously exposed to attack, so security checking must happen during development and not only after deployment. Common weaknesses such as injection, unsafe deserialization, weak cryptography and insecure configuration are catalogued in the OWASP Top 10 [1] and the Common Weakness Enumeration [7], and structured guidance exists for testing [2], verification [3] and secure development practice [4]. In practice, however, a student or small team who wants a quick first look at a codebase must either configure a full static-analysis engine or review code by hand, and then organise the output into a form that others can act on.

BugShield targets this gap at the workflow level. It combines project management, source acquisition, a pattern-based scanner, a searchable findings list and PDF reporting in one application. It does not attempt data-flow or taint analysis. Its value is in making an initial pass quick to run and easy to review, while being explicit that its output is a set of leads for a human reviewer and not a verdict.

1.1 Contributions

- A working, integrated workflow that takes a public GitHub repository or an uploaded source file through scanning, findings review and PDF export in a single application.
- A findings model that records severity, description, file and line location, matched-code excerpt and status for every match, so each result can be traced to its source.
- A qualitative review of real scanner output on an intentionally vulnerable PHP project, identifying concrete false-positive patterns (matches on comments and a name collision with SQL execution calls) that guide the next version of the rules.
- A candid security assessment of the platform itself, including the missing ownership scoping, and a reproducible protocol for a quantitative benchmark.

1.2 Paper Organisation

Section 2 reviews related work. Section 3 describes the design and implementation. Section 4 presents the qualitative case study. Section 5 assesses the security of the platform. Section 6 discusses limitations and threats to validity. Section 7 gives the proposed evaluation protocol and future work, and Section 8 concludes.

2. Background and Related Work

Static application security testing analyses source code without executing it. Early and influential work argued that static analysis can find real security defects but that its usefulness depends on rule quality and on managing false positives [5], and research on scripting languages such as PHP showed that detecting injection-style flaws reliably requires modelling how untrusted input flows to sensitive operations [6]. Modern maintained engines such as Semgrep [14] express rules over syntax trees, and dedicated secret scanners such as Gitleaks [15] focus on exposed credentials. These tools are more precise than line-level pattern matching but need setup, rule selection and result handling.

Guidance documents define what to look for and how to organise the work. The OWASP Web Security Testing Guide [2] and Application Security Verification Standard [3] describe test procedures and verification requirements, and the NIST Secure Software Development Framework [4] recommends integrating security practices across the development lifecycle. BugShield does not implement these frameworks. It supports a narrow part of that lifecycle, an early automated triage with documented output.

We therefore do not claim detection superiority or novelty of the individual techniques. The contribution is the integration of a simple scanner with persistence, review and reporting in one deployable application, together with an honest account of its behaviour on real output.

3. System Design and Implementation

3.1 Workflow

A signed-in user creates a project by supplying a public GitHub repository URL or by uploading a text source file. The server obtains the source, runs the scanner, stores the project, the scan record and the findings, and exposes them through a dashboard and a findings page from which a PDF report can be downloaded. Figure 1 summarises the workflow.

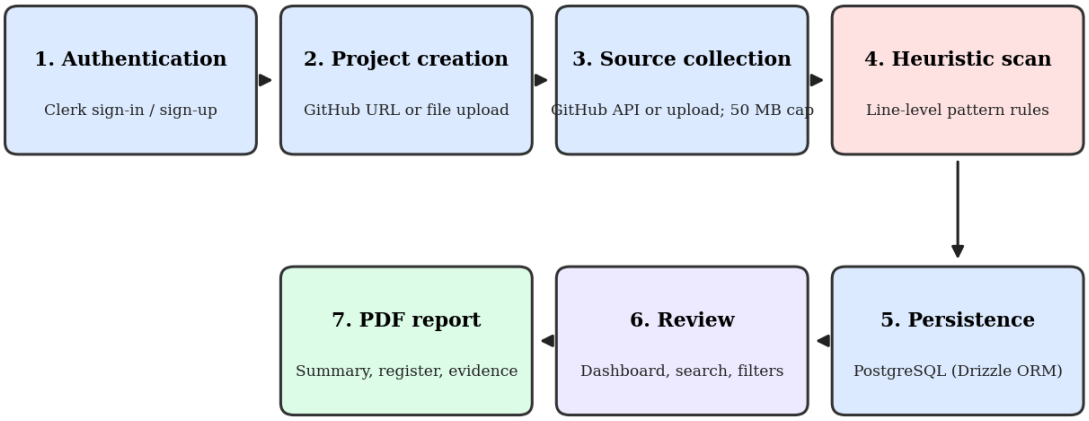


Figure 1. BugShield workflow from sign-in to PDF report.

3.2 Technology Stack

Table 1. Technology stack.

Layer	Implementation
Application framework	**Next.js 16 (App Router, server actions, API handlers), React 18, TypeScript**
User interface	**Tailwind CSS; dashboard, projects, vulnerabilities and settings pages**
Authentication	**Clerk sign-in and sign-up; middleware redirects signed-out visitors to sign-in [11]**
Database	**PostgreSQL 14 or newer with Drizzle ORM and Drizzle Kit [10][13]**
Report generation	**pdf-lib [12]**
Source acquisition	**GitHub REST API for public repositories [8]; direct upload of text source files**
Runtime	**Node.js 20.9.0 or newer**

3.3 Source Acquisition

For a GitHub project, the server reads the default branch through the GitHub API, downloads supported source and configuration files, and skips common generated and dependency directories. A combined-source limit of 50 MB is enforced. If GitHub truncates its file listing or any source file cannot be downloaded, the scan stops with an error instead of reporting a partial result, which avoids presenting an incomplete scan as a clean one. Private repositories are not supported, even though the Settings page can store provider tokens. For uploads, the file contents are sent to a server action; upload scanning is intended for text source files.

3.4 Heuristic Scanning

The scanner checks source lines against built-in patterns for common risky code, including dynamic evaluation, injection-like query construction, unsafe shell calls and insecure configuration. The case study in Section 4 also shows rules for insecure deserialization, weak cryptographic hashes and unsafe HTML injection. A match becomes a finding with a title, severity, description, file and line location and

a matched-code excerpt. Matching is line-based, so the scanner has no knowledge of language grammar, control flow or data flow.

3.5 Persistence

The Drizzle schema (lib/db/schema.ts) defines three entities. The projects table holds the project name, repository, stored source, scan status, issue count and last-scan time. The scans table holds scan status and start and completion timestamps. The vulnerabilities table holds the finding title, severity, description, source excerpt, location and status. Migrations are kept under lib/db/migrations; push-based schema updates are used for local development and reviewed, versioned migrations are recommended for production.

3.6 Review and Reporting

The dashboard presents project and scan aggregates, finding trends, priority findings and recent scan activity. The Vulnerabilities page lists findings with their severity, description, location, matched code and status, and supports text search and filtering by severity and status. A report endpoint builds a PDF from the stored findings containing a summary, a findings register, source evidence and suggested remediation.

4. Qualitative Case Study

To examine real scanner output, we scanned an intentionally vulnerable PHP demonstration project named vuln-php-demo using a local instance of BugShield. Figure 2 shows the resulting Vulnerabilities page. This is a single, small, illustrative run. It is not a benchmark, no ground-truth labelling of the project was used, and the observations below must not be read as precision or recall figures.

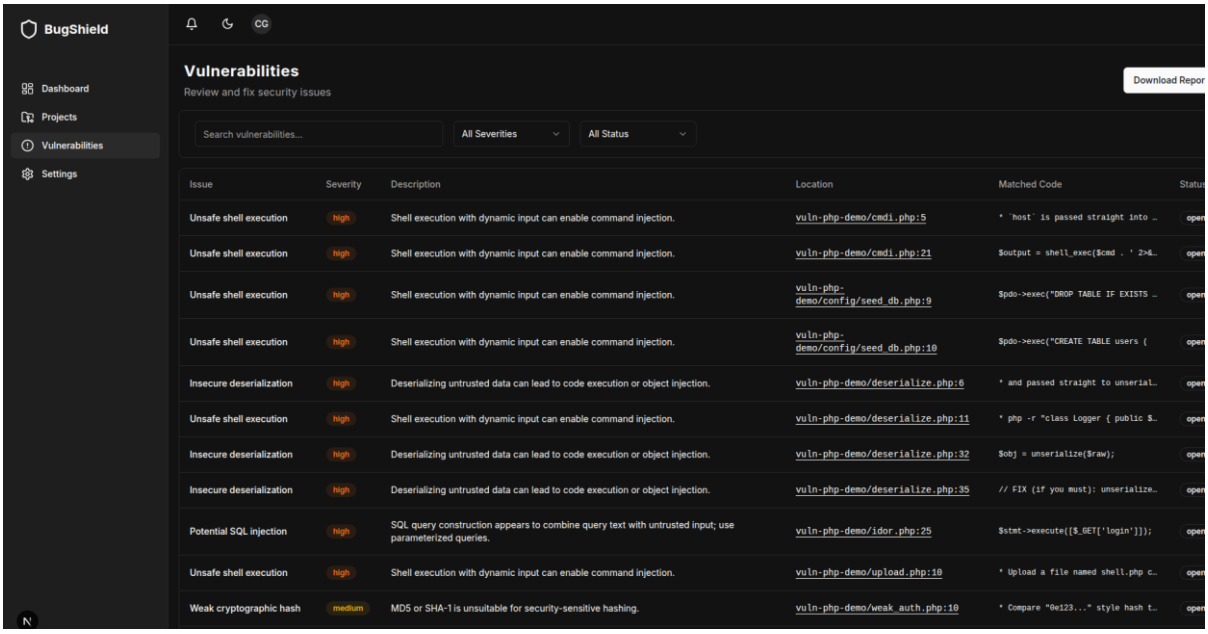


Figure 2. Vulnerabilities page after scanning vuln-php-demo, showing severity, description, location, matched code and status.

We manually reviewed the eleven findings fully visible in Figure 2 (ten high, one medium) using the displayed excerpts. Table 2 records the review notes. Because excerpts are truncated in the interface, notes marked as comment matches are based on the excerpt beginning with a comment marker.

Table 2. Manual review of the eleven visible findings.

Location	Reported category	Severity	Reviewer note
cmdi.php:5	Unsafe shell execution	High	Excerpt is a comment line describing the vulnerability; comment match.
cmdi.php:21	Unsafe shell execution	High	Code line calls shell_exec with a concatenated command; consistent with a command-injection weakness.
config/seed_db.php:9	Unsafe shell execution	High	Excerpt is $pdo->exec(...), which executes SQL through PDO, not a shell; misclassification.
config/seed_db.php:10	Unsafe shell execution	High	Same as above; PDO exec call matched by a shell-execution rule.
deserialize.php:6	Insecure deserialization	High	Excerpt is a comment line; comment match.
deserialize.php:11	Unsafe shell execution	High	Excerpt is a comment line containing an example php -r command; comment match.
deserialize.php:32	Insecure deserialization	High	Code line unserializes a raw variable; consistent with an insecure-deserialization weakness.
deserialize.php:35	Insecure deserialization	High	Excerpt is a comment containing remediation text; comment match.
idor.php:25	Potential SQL injection	High	Excerpt shows a prepared-statement execute with a bound value; needs manual triage and is probably not injectable through this line.
upload.php:10	Unsafe shell execution	High	Excerpt is a comment line describing an upload scenario; comment match.
weak_auth.php:10	Weak cryptographic hash	Medium	Excerpt is a comment line; the hash call itself is not visible in the excerpt.

In this sample, two findings point at code that is consistent with a genuine weakness, six match comment text, two are misclassified because a SQL execution method shares a name pattern with a shell call, and one requires manual triage. The sample also confirms the scanner does find dangerous constructs in this project, and the categories and locations make each result quick to verify. It also shows that a line-based matcher without comment handling or call-type awareness produces a high proportion of noise on documented demonstration code, which reinforces the README's instruction that all matches require human review.

These observations directly suggest improvements: skip or down-rank matches inside comments and documentation blocks, distinguish PDO and database exec methods from operating-system command execution, treat prepared statements with bound parameters differently from concatenated queries, and move from line patterns towards syntax-tree rules.

5. Security Assessment of the Platform

A security tool is itself an attack surface, and the project documentation records several limitations that we state here explicitly.

- **Missing ownership scoping.**The projects table has no owner or user identifier, and server actions can operate on globally queried project and finding records. Any signed-in user could therefore potentially read or modify another user's data. This is a broken-access-control weakness of the kind described in the OWASP Top 10 [1]. The application must not be deployed for multiple real users until projects and findings are scoped to the authenticated owner through an owner column, scoped queries and a reviewed migration.
- **Sensitive data at rest.**Uploaded source is stored in the database, and the Settings page can store provider tokens. Before any real deployment, the protection of stored source and tokens (encryption, access control and retention) should be reviewed.
- **Untrusted input.**Repository contents and uploaded files are untrusted. The 50 MB combined-source limit, the supported-file filtering and the fail-on-truncation behaviour reduce resource-exhaustion and silent-partial-scan risks, but matched excerpts are rendered in the interface and embedded in PDFs and should be treated as untrusted output.
- **Quality assurance.**The lint script is currently not operational with the installed Next.js version and there is no automated test suite, so neither should be reported as passing. TypeScript checking and dependency auditing (npm audit) are available and should be part of release practice.
- **Responsible use.**The scanner should be used only on code the user owns or is authorised to assess. Its output is not a security certification.

6. Limitations and Threats to Validity

- **Detection coverage.**Only constructs covered by the built-in patterns are reported. Runtime behaviour, business logic, authorization flaws, dependency vulnerabilities and deployment configuration are out of scope.
- **False positives and negatives.**Table 2 shows concrete false-positive patterns. Missed vulnerabilities cannot be counted without a ground-truth benchmark.
- **Evidence base.**The case study covers one project and eleven findings reviewed by one person from truncated excerpts. It illustrates behaviour and does not support generalisation.
- **Scope of inputs.**Only public GitHub repositories and uploaded text files are supported, from the default branch only, and large repositories are rejected by the 50 MB limit.
- **No comparison.**BugShield has not been compared with established tools, so no claim is made that it is faster, more accurate or otherwise superior.

7. Evaluation Protocol and Future Work

7.1 Proposed Benchmark Protocol

A quantitative evaluation should use test applications with known, documented vulnerabilities, such as OWASP Juice Shop, DVWA or other intentionally vulnerable projects, with expected findings defined before scanning. For each run, record the BugShield commit, rule set, environment, input and raw

output, and map each reported finding to a known case. A finding is a true positive if it points to a known vulnerable location and category, a false positive otherwise, and a known case that is not reported is a false negative. Precision is TP/(TP+FP), recall is TP/(TP+FN) and F1 is their harmonic mean. Scan time should be measured over repeated runs on fixed hardware with the method stated. The same inputs should be scanned with at least one maintained tool such as Semgrep [14] under documented configuration, so that results can be compared fairly.

7.2 Planned Improvements

- Add owner scoping to projects, scans and findings, with a reviewed migration and authorization checks on every server action.
- Reduce noise: ignore comments, separate SQL and shell execution rules, and handle parameterized queries.
- Move from line patterns to syntax-aware rules, and integrate established scanners and dependency auditing as optional sources of findings.
- Add automated tests and a working lint step, support private repositories safely, and add CI integration.

8. Conclusion

BugShield provides an end-to-end workflow for a first-pass security review: project registration from a public repository or an uploaded file, heuristic scanning, searchable findings with severity and code evidence, persistent scan history and PDF reporting. A manual review of its output on a vulnerable PHP project showed that it surfaces genuinely dangerous calls but also produces avoidable noise from comments and misclassified calls, and the platform itself needs ownership scoping before multi-user use. We therefore position BugShield as a triage and reporting aid whose findings require human review. The benchmark protocol and the planned improvements define the work needed to move from a demonstrated workflow to a measured one.

Declarations

Ethics and authorisation. The scan reported in this paper was run on an intentionally vulnerable demonstration project for which authorisation was not required, and no third-party systems were tested. No credentials, personal data or confidential source code appear in this paper.

Conflict of interest. The author declares no conflict of interest.

Funding. The author received no external funding for this work.

Acknowledgements

None

Funding

This research received no external funding.

Conflict of Interest

The author declares no conflict of interest.

Data Availability

The experimental data consist of security scan results and generated reports. Anonymized results may be made available upon reasonable request, subject to confidentiality, authorization, and security restrictions.

References

OWASP Foundation. (n.d.). *OWASP Top 10*. https://owasp.org/www-project-top-ten/

OWASP Foundation. (n.d.). *Web Security Testing Guide*. https://owasp.org/www-project-web-security-testing-guide/

OWASP Foundation. (n.d.). *Application Security Verification Standard*. https://owasp.org/www-project-application-security-verification-standard/

Souppaya, M., Scarfone, K., & Dodson, D. (2022). *Secure Software Development Framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities* (NIST SP 800-218). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-218

Chess, B., & McGraw, G. (2004). Static analysis for security. *IEEE Security & Privacy, 2*(6), 76–79.