iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** XI **Month of publication:** November 2026

DOI: https://doi.org/10.22214/ijraset.2026.85569

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue XI Nov 2026- Available at www.ijraset.com

OmniGuard AI: A Dual-Engine Platform for Code Leakage and Phishing Defense

Santhosh V[1], Yuvan Sundar R[2], Vishwa Kumar R[3]
Department of CSE, Adhiyamaan College of Engineering, Hosur, Tamilnadu, India

Abstract: ***In modern software development environments, sensitive credential exposure and phishing attacks remain two of the most critical security vulnerabilities threatening enterprise infrastructure. Traditional security solutions often operate in isolation and identify vulnerabilities long after code has been committed or applications deployed. This paper presents OmniGuard AI, an integrated dual-engine security platform that unifies static secret detection, automated remediation and phishing threat analysis. Engine 1 (Code Secret Scanner) combines regular-expression signatures with Shannon entropy scoring to detect hardcoded API keys, database connection strings and tokens, while Engine 2 (Phishing Threat Analyzer) evaluates URLs with structural and lexical feature heuristics to assign risk scores. To enforce a "shift-left" security model, the platform provides a local Git pre-commit hook that intercepts credential leaks before they are committed, and Credential Shield JS, a lightweight browser-side runtime monitoring module. An automated refactoring engine then remediates detected vulnerabilities by replacing hardcoded credentials with secure environment-variable calls (os.getenv()). The prototype is implemented in Python Flask with a browser dashboard and is validated functionally on backend and frontend routes: the scanner reports simulated credentials with file, line, signature, entropy and severity, no false positives were observed on the clean templates tested, and the pre-commit hook is implemented to abort commits that contain credential patterns. By bridging pre-commit prevention with application-level threat analysis, OmniGuard AI provides a scalable, developer-centric framework for enterprise security assurance.***

Keywords: ***secret leakage detection, shift-left security, pre-commit hooks, phishing detection, credential shield, automated refactoring, Shannon entropy, software security.***

I. INTRODUCTION

Modern software development relies heavily on cloud infrastructure, third-party APIs, microservice architectures and distributed version control. These technologies have accelerated delivery cycles, but they have also enlarged the enterprise attack surface. Two distinct yet interconnected threats have become dominant attack vectors: source-code credential leakage and phishing-based social engineering.

Source repositories, public and private, frequently suffer accidental disclosure of AWS access keys, database connection URIs, Stripe secret tokens and private cryptographic keys. When developers hardcode such credentials for local testing or rapid prototyping, they risk staging and committing them to Git. Once pushed to a public platform or a compromised corporate repository, the secrets can be harvested by automated scraping bots within seconds, leading to unauthorized access, cloud-resource exploitation and costly data breaches. At the same time, users and developers are targeted by phishing campaigns that rely on brand impersonation, deceptive top-level domains (TLDs), long subdomain chains and obfuscated URL structures to capture session tokens and access credentials.

Traditional paradigms treat these two domains separately: static application security testing (SAST) tools analyse repositories, while perimeter or endpoint tools handle web threats. OmniGuard AI bridges this gap with a unified, dual-engine platform that prevents source-code leaks at the developer workstation while identifying phishing threats across application runtimes. The main contributions are:

1) *Dual-engine detection core:* Engine 1 applies regular-expression signatures and Shannon entropy scoring to uploaded source files; Engine 2 scores URLs from structural, lexical and heuristic features.
2) *Shift-left developer controls:* a downloadable Git pre-commit hook blocks commits that contain credential patterns, and Credential Shield JS monitors input values in the browser at runtime.
3) *Automated remediation:* an in-place refactoring route rewrites flagged hardcoded assignments into os.getenv() calls and returns an updated source preview.

©IJRASET: All Rights are Reserved

4) *Telemetry and compliance reporting:* a live findings ledger on the dashboard and a CSV audit export (/export/csv) give administrators a complete record of detections.

The rest of the paper is organised as follows. Section II surveys related work, Section III analyses existing and proposed systems, Section IV presents the architecture, Section V explains the methodology, Section VI describes the implementation, Section VII reports results, Section VIII discusses limitations and future work, and Section IX concludes.

II. LITERATURE SURVEY

Smith and Johnson [1] study modern phishing attacks, which bypass traditional blacklist filters through dynamic URL structures and newly registered domains. They evaluate supervised classifiers trained on structural URL properties such as path length, suspicious query parameters, subdomain depth and lexical randomness, and show that structural heuristics combined with lightweight lexical parsing give high accuracy at sub-second latency. This supports the use of structural and lexical heuristics in Engine 2.

Patel and Gomez [2] address the high false-positive rate of regex-based secret scanners, which often match dummy variables, comments or public test keys. Their context-aware models analyse the surrounding code together with Shannon entropy, checking whether a high-entropy string is assigned to a sensitive variable or merely used as mock data. This motivates supplementing the regex signatures of Engine 1 with entropy metrics.

Zhang and Wang [3] tackle zero-day phishing, where blacklists are ineffective, with an optimised ensemble feature-selection framework. Combining lexical, host-based and content features through weighted voting improves robustness against unseen attack patterns, which guides the weighted scoring and risk assessment of the phishing analyzer.

Davis and Taylor [4] argue that vulnerabilities found late in production raise remediation cost sharply. They discuss automated security hooks in version-control systems, where local pre-commit validation intercepts vulnerability injections before code is staged or sent to remote repositories. This directly validates the local Git pre-commit hook of OmniGuard AI. Anderson and Lee [5] propose a hybrid prevention architecture that combines static pattern matching, entropy analysis and runtime monitoring, and stress that detection alone is insufficient without automated remediation that converts exposed hardcoded keys into environment variables.

Further work covers entropy-based static analysis for credential leakage [6], browser runtime monitoring and credential shielding [7], interactive security-operations dashboards for automated remediation [8], lexical feature extraction with heuristic evaluation for phishing analysis [9], and automated compliance auditing with telemetry export [10]. Individually, these studies address detection, prevention, remediation or reporting; few combine them. OmniGuard AI targets this gap by integrating secret scanning, phishing analysis, pre-commit enforcement, runtime shielding, automated refactoring and audit export in one platform.

III. SYSTEM ANALYSIS

A. *Existing System and Its Limitations*

Many organisations still rely on manual code review, isolated static-analysis tools or reactive post-commit scanning to find credential leaks, while phishing detection is handled by separate firewalls or external threat-intelligence feeds. Developer remediation and client-side credential monitoring are largely informal. The resulting limitations are as follows.

1) *Lack of integration:* code scanning, phishing detection and remediation are handled by separate tools, giving fragmented workflows and inconsistent policy enforcement.
2) *Reactive, delayed detection:* most SAST systems analyse repositories only after code is committed or pushed, leaving a window in which exposed credentials can be scraped.
3) *Developer friction:* scanners report problems but do not fix them, so developers manually rewrite code and configure environment variables, which costs time and invites error.
4) *No runtime protection:* few solutions protect the browser, leaving applications exposed to accidental token entry, clipboard leaks and exposed form inputs.
5) *Poor telemetry:* disparate tools produce scattered logs and inconsistent audit reports, which limits operational visibility.

B. *Proposed System*

OmniGuard AI is an integrated web-based platform that unifies static code scanning, automated refactoring, pre-commit enforcement and phishing analysis in one developer-centric interface built with Python Flask and standard web technologies. Developers upload code files for Engine 1 and submit URLs to Engine 2, trigger refactoring from the findings ledger, download the pre-commit hook and shield.js from the dashboard, and export the findings as a CSV audit report. The advantages over the existing approach are shift-left enforcement on the developer workstation, a single dashboard for both engines, automated remediation instead of mere flagging, runtime browser protection, and centralised telemetry with exportable reports.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

C. Problem–Solution Fit and Ideation

The ideation phase (Fig. 1) moved from problem identification, through brainstorming of a dual-engine core with shift-left hooks and a browser shield, to technology selection: Python Flask for the backend, HTML, CSS and JavaScript for the dashboard, regex pattern matching with Shannon entropy scoring, and local Git hook automation. Feature planning fixed regex matching, entropy calculation, URL lexical feature extraction, os.getenv() refactoring and CSV export as the core features, with emphasis on speed, no false positives on clean templates and easy developer integration.

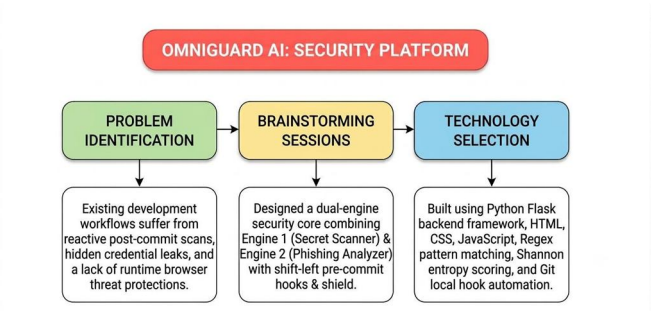


Fig. 1 Ideation and brainstorming flow of the OmniGuard AI project: problem identification, brainstorming sessions and technology selection.

IV. SYSTEM ARCHITECTURE AND DESIGN

A. Architecture Overview

The architecture (Fig. 2) has three parts. The data sources are internal code repositories (Git, SVN) containing .py, .js, .csv and .docx files, and external threat vectors such as URLs, websites and phishing links. The core processing backend hosts Engine 1 for internal scanning, with a multi-format parser using regex search, AST tokenization and Shannon entropy, and Engine 2 for external scanning, with a phishing classifier (Random Forest model) and WHOIS telemetry, all backed by database storage. The outputs and remediation layer provides a real-time alerts dashboard with a risk score, audit reporting in PDF and CSV, and proactive remediation tools: the Git pre-commit hook and os.getenv() auto-sanitization.

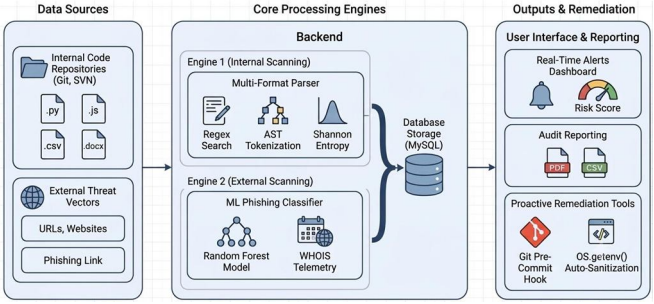


Fig. 2 Architectural design of OmniGuard AI: data sources, core processing engines and outputs/remediation.

B. Description of Modules

1) *User authentication and dashboard:* users log in or register with verified credentials and sessions are maintained securely. The dashboard is the central interface for scans, URL analysis, tool downloads and CSV export.
2) *Engine 1, Code Secret Scanner:* inspects uploaded source files for AWS access keys, Stripe live tokens, database connection URIs and private API keys, combining regular-expression matching with Shannon entropy analysis to separate real secrets from random text.
3) *Automated refactoring module:* reads the flagged file, locates the vulnerable string with regex, replaces it in place with an os.getenv() call, persists the file to disk and returns a success confirmation.
4) *Engine 2, Phishing Threat Analyzer:* breaks a URL into structural, lexical and heuristic indicators (excessive subdomain nesting, deceptive TLDs, brand-impersonation keywords and IP-address hosting) to compute a risk score and classify the threat.
5) *Shift-left developer tools:* the Git pre-commit hook (pre-commit) blocks commits that contain staged secrets, and Credential Shield JS (shield.js) monitors browser inputs and logs security warnings to the console.

C. Data Flow

Fig. 3 shows how data moves through the system. The user or developer authenticates and submits code files and URLs to the core security application. Code files go to Engine 1, which returns scan results and flagged detections, and URLs go to Engine 2, which returns threat intelligence and scores. The core application writes to four data stores: the security findings database (audit log data), the detections database (SQL), the threat-intelligence cache and the shift-left compliance repository. A remediation command routes to the automated remediation process and returns the refactoring status, while hook installation and configuration, together with the Credential Shield JS tools, feed the hook-installation process that produces the exported security report.

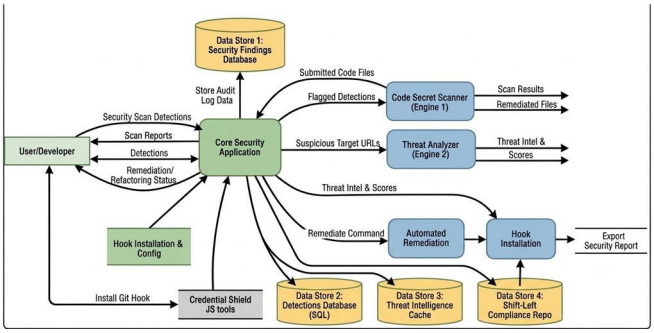


Fig. 3 Data flow diagram of OmniGuard AI showing the user, core application, both engines, remediation, hook installation and four data stores.

D. Technology Stack

Table I lists the technologies used. The backend is Python Flask [11] with Flask-SQLAlchemy and Flask-Login; Git hooks [12] provide local enforcement; and the dashboard uses Bootstrap [13] with custom CSS and JavaScript.

TABLE I
TECHNOLOGY STACK OF OMNIGUARD AI

Layer	Technology
Backend	Python 3.8+, Flask 3 [11], Flask-SQLAlchemy, Flask-Login, python-dotenv
Data	SQLAlchemy; SQLite by default, MySQL through PyMySQL
Analysis	Built-in re, csv, io, os; scikit-learn, joblib, numpy, pandas; python-whois
Documents	python-docx (.docx input), reportlab (PDF reports)
Frontend	HTML5, CSS3, JavaScript ES6, Bootstrap 5.3 [13]
Dev tools	Git hooks [12], pytest, Visual Studio Code

V. METHODOLOGY

A. Engine 1: Signature and Entropy Scoring

Engine 1 reads an uploaded file line by line and applies regular-expression signatures for AWS access keys (AKIA/ASIA prefixes), Stripe tokens (sk_live_), database connection URIs and password assignments. Each matched string is also scored with its Shannon entropy [14],

$$H(s) = -\Sigma\, p(c) \cdot \log_2 p(c) \qquad (1)$$

where p(c) is the relative frequency of character c in string s and the sum runs over the distinct characters of s. High entropy indicates random, key-like text, whereas ordinary identifiers score lower. For each finding the scanner records the file path, line number, flagged signature, entropy score and severity (Table II lists the signature classes). As a check, the score the ledger shows for AKIAIOSFODNN7EXAMPLE, 3.6842, equals the value of (1) computed in bits per character for that 20-character string.

TABLE II
CREDENTIAL SIGNATURE CLASSES AND OBSERVED SEVERITY

Signature	Pattern / indicator	Severity
AWS access key ID	AKIA or ASIA + 16 chars [0-9A-Z]	critical
AWS secret key	40-char base64-like string	high
Stripe live key	sk_live_ (hook also rk_live_) + 16 or more chars	high
Slack token	xox[baprs]- + 10 or more chars	high
DB connection URI	scheme://user:pass@host	high
JSON Web Token	eyJ… three-part token	high
Password/secret assignment	password, secret, api_key, access_token = value of 8 or more chars	high

B. Automated Refactoring

When the user presses Refactor in the ledger, the dashboard posts the file path to /refactor. The route validates that the path exists, reads the file, and applies the substitution

([A-Z0-9_]+_KEY|[A-Z0-9_]+_SECRET|PASSWORD)\s*=\s*["'][^"']+["']

which turns an assignment such as STRIPE_SECRET_KEY = "sk_live_…" into STRIPE_SECRET_KEY = os.getenv("STRIPE_SECRET_KEY"). The rewritten text is written back to the same file, and the route returns a JSON status, a message and the updated source, which the dashboard shows as a preview (Fig. 8).

C. Engine 2: Phishing Threat Analysis

Engine 2 parses a submitted URL and extracts structural and lexical indicators: domain length, subdomain nesting depth, suspicious TLDs, IP-address hosting and brand-impersonation keywords. These indicators are combined into a standardised risk score, shown as a percentage with a severity label together with a domain-status flag (Fig. 9). Following the literature [1], [3], [9], the design favours cheap features that can be computed in real time.

D. Shift-Left Developer Tools

1) *Git pre-commit hook:* served at /download/hook and installed as .git/hooks/pre-commit, the script runs git diff --cached --unified=0 and examines only added lines. It searches them with a case-insensitive extended regex for AKIA/ASIA keys, sk_live_/rk_live_ keys, URIs with embedded user:pass@, and password, secret, api_key or access_token assignments of eight or more characters. A match prints a block message to standard error and exits with status 1, which makes Git abort the commit (Fig. 4). A companion CLI, omniguard_hook_scan.py, runs Engine 1 on given files and exits 1 when any finding is high or critical; install_pre_commit_hook.py installs the hook in the current workspace.

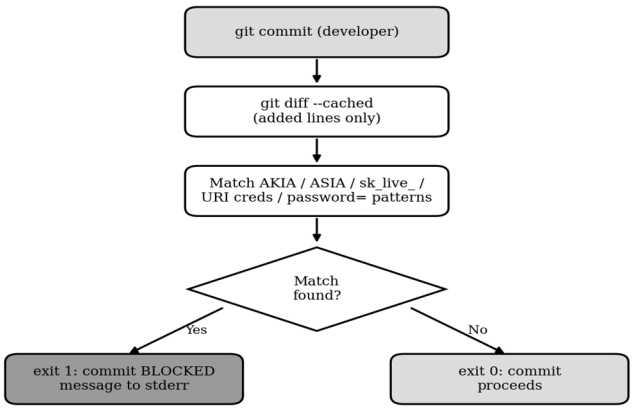


Fig. 4 Decision flow of the OmniGuard pre-commit hook.

2) *Credential Shield JS:* served at /download/shield, shield.js is an immediately invoked function expression (under 2 KB) that exposes window.OmniGuardCredentialShield.scan(value). It tests a string against AWS-key, sk_live_ and Slack-token (xox[baprs]-) patterns, logs a console warning and returns true on a match, so that host pages can inspect form data and block unsafe submissions.

E. Audit Reporting and REST Interface

The /export/csv route builds the report with Python's csv and io.StringIO modules. Its header row is File Target, Line, Flagged Signature, Risk Level and Severity, followed by one row per stored finding, and it is returned as an attachment named omniguard_audit_report.csv. Table III lists the routes.

TABLE III
MAIN HTTP ROUTES OF THE FLASK SERVICE

Route	Method	Purpose
/	GET	Dashboard (127.0.0.1:5000)
/scan	POST	Upload file; run Engine 1; return findings as JSON
/refactor	POST	Replace secrets with os.getenv() in the file
/export/csv	GET	Download CSV audit report
/download/hook	GET	Download pre-commit hook script
/download/shield	GET	Download shield.js

VI. IMPLEMENTATION

A. Environment and Requirements

The system was developed and tested in a local environment (Python Flask served at 127.0.0.1:5000, Chrome browser). Table IV gives the hardware and software requirements.

TABLE IV
SYSTEM REQUIREMENTS

Item	Requirement
Server CPU/RAM	1.4 GHz dual-core or higher; 2 GB RAM minimum
Storage	20 GB or more free disk space (SSD recommended)
Client	Desktop or laptop, 1 GHz or faster, 1 GB RAM; Chrome, Edge or Firefox (HTML5, CSS3, ES6)
Operating system	Ubuntu 20.04+, Windows 10/11 or macOS
Runtime	Python 3.8+ with Flask; built-in csv, io, re, os
Tools	Git for hook testing; VS Code; pytest for route and component tests

B. Configuration and Project Structure

Settings are read from environment variables, in keeping with the platform's own advice: SECRET_KEY, DATABASE_URL (default SQLite), MAX_UPLOAD_BYTES (default 25 MB), UPLOAD_FOLDER, WHOIS_TIMEOUT (default 4 s) and RISK_MODEL_PATH.

A TestConfig class uses an in-memory SQLite database with CSRF disabled so that pytest runs are fast and isolated. The entry point app.py calls create_app(), and pytest.ini sets the test path. Dependencies are pinned by major version in requirements.txt.

C. User Interface

The dashboard is a dark-themed Bootstrap 5.3 page with custom CSS. A header carries the product name and three buttons: Download Pre-Commit Hook, Download Credential Shield JS and Export CSV Audit Report. Two cards hold the Engine 1 file upload form and the Engine 2 URL form. Beneath them, the Security Audit Findings table lists file or target, line or indicator, flagged signature, risk score, severity and an automated-action button. The upload handler posts the file to /scan, fills the table from the JSON response, or shows a clean-file message when no findings are returned; the Refactor button posts to /refactor, shows the server message and reloads the page.

VII. RESULTS AND DISCUSSION

Evaluation was carried out in a local development environment to verify that files with hardcoded credentials are flagged, remediated in place and logged for telemetry and compliance reporting. Two test files containing simulated credentials were uploaded; the figures below are screenshots of the running prototype.

A. Dashboard and Engine Panels

The dashboard initialises with the Engine 1: Code Secret Scanner and Engine 2: Threat & Phishing Detector panels and a “System Online & Secured” status indicator (Fig. 5). Engine 1 accepts .py, .js, .csv and .docx uploads.

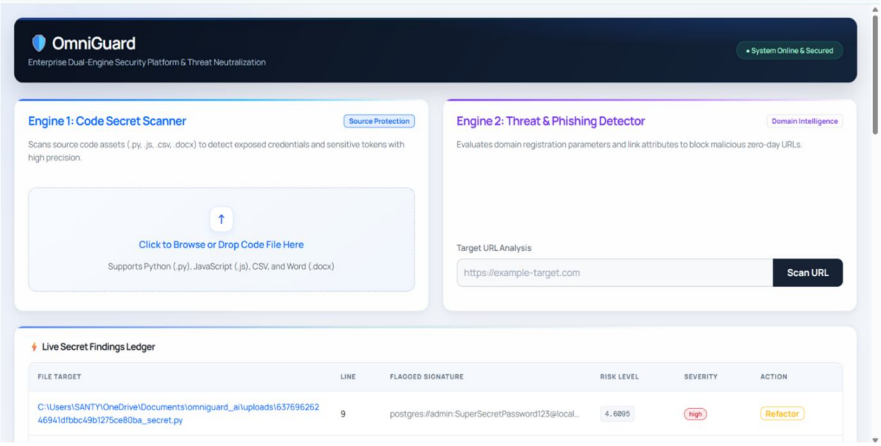


Fig. 5 Main dashboard with Engine 1 upload panel, Engine 2 URL panel and the Live Secret Findings Ledger.

B. Engine 1 Scanning Results

After an upload, the dashboard confirms “Scan successful! Found 4 secret(s)” and lists the findings (Fig. 6). The ledger records file path, line number, flagged signature, risk level (entropy) and severity, with a Refactor action per row (Fig. 7). Table V transcribes the eight findings from the two test files; each file produced four findings.

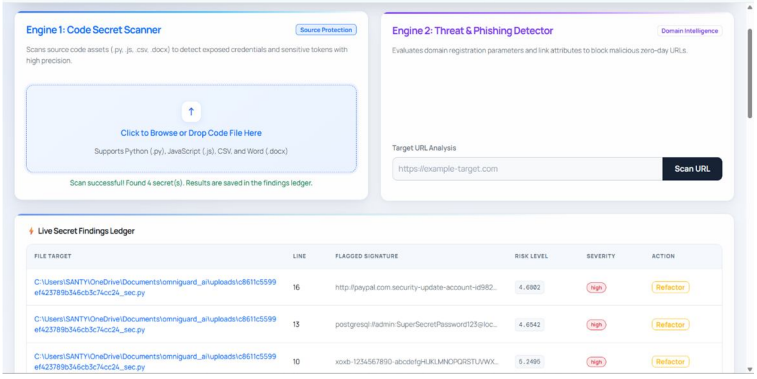


Fig. 6 Code Secret Scanner after upload: four secrets found and listed in the findings ledger.

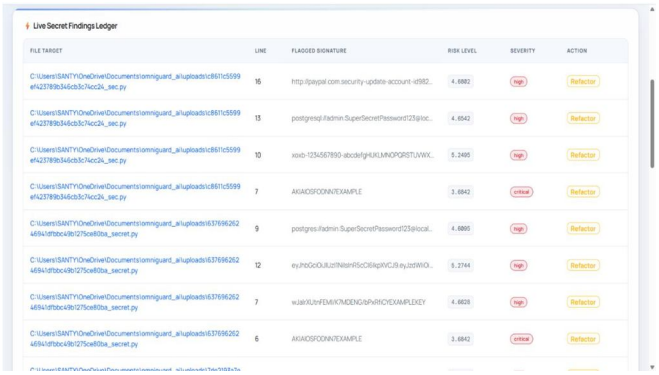


Fig. 7 Live Secret Findings Ledger showing file target, line, flagged signature, risk level, severity and Refactor action.

TABLE V
FINDINGS REPORTED FOR THE TWO TEST FILES

File	Line	Flagged signature	Entropy	Severity
A	16	http://paypal.com.security-update-account-id982…	4.6802	high
A	13	postgresql://admin:SuperSecretPassword123@loc…	4.6542	high
A	10	xoxb-1234567890-abcdefgHIJKLMNOPQRSTUVWX…	5.2495	high
A	7	AKIAIOSFODNN7EXAMPLE	3.6842	critical
B	9	postgres://admin:SuperSecretPassword123@local…	4.6095	high
B	12	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOi…	5.2744	high
B	7	wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY	4.6628	high
B	6	AKIAIOSFODNN7EXAMPLE	3.6842	critical

File A is c8611c5599ef423789b346cb3c74cc24_sec.py and file B is 63769626246941dfbbc49b1275ce80ba_secret.py, both in the upload folder. Signatures are truncated by the dashboard.

C. Automated Refactoring

Selecting Refactor calls the remediation route, which parses the vulnerable file and rewrites raw secret assignments into os.getenv() calls. The dashboard then displays the updated source in a preview dialog, which starts with import os and import requests and lists the Stripe live-key pattern and the AWS access-key-ID pattern (Fig. 8).

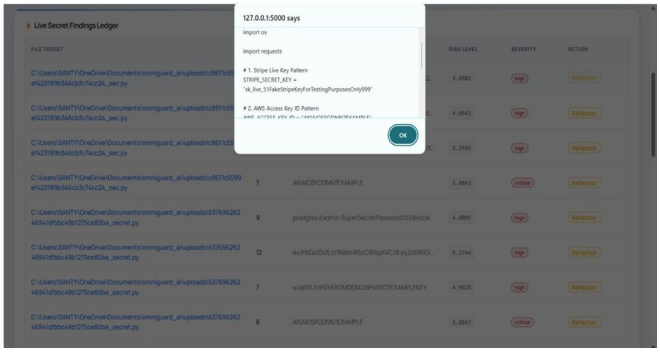


Fig. 8 Automated refactoring execution: success preview dialog showing the rewritten source.

D. Engine 2 Phishing Analysis

Engine 2 parsed the user-submitted URL https://pathfinder.w3schools.com/?checkoutRedirect=true, extracted its structural and lexical features, and returned a Threat Risk Score of 15% with the domain status “Verified” (Fig. 9). The low score is consistent with a legitimate-looking host. This is a single illustrative probe, not a detection-rate measurement.

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue XI Nov 2026- Available at www.ijraset.com

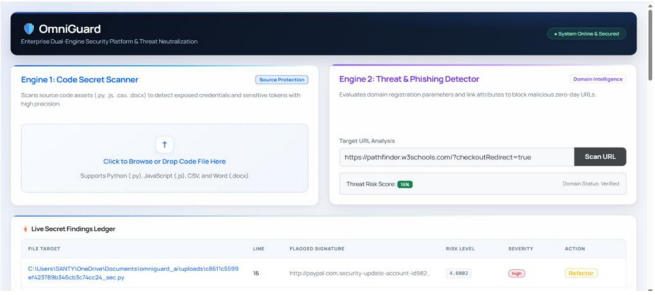


Fig. 9 Phishing threat analysis and risk scoring for a submitted URL.

E. Shift-Left Controls and Audit Export

The compliance footer of the dashboard gives one-click access to Export CSV Audit, Git Pre-Commit Hook and Credential Shield JS (Fig. 10). The ledger above the footer also shows that findings from further uploaded files accumulate in the same table.

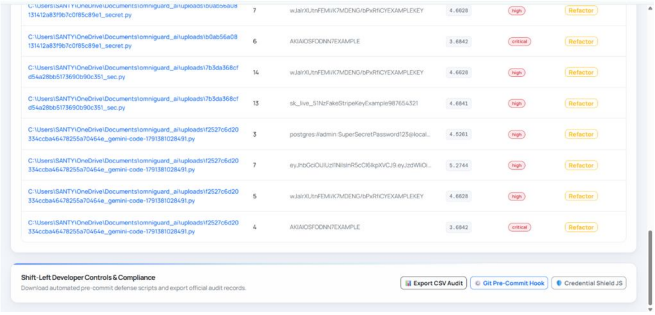


Fig. 10 Shift-Left Developer Controls and Compliance panel with CSV audit export, pre-commit hook and Credential Shield JS.

F. Entropy Analysis

Fig. 11 plots the entropy of the eight findings in Table V. High-severity matches lie between 4.61 and 5.27 bits per character, while the AWS access-key IDs have the lowest entropy, 3.68, yet are rated critical. In these runs, therefore, severity follows the matched signature class, and entropy is a supplementary measure of how random the string is, in line with the combined regex-plus-entropy design suggested in [2] and [5].

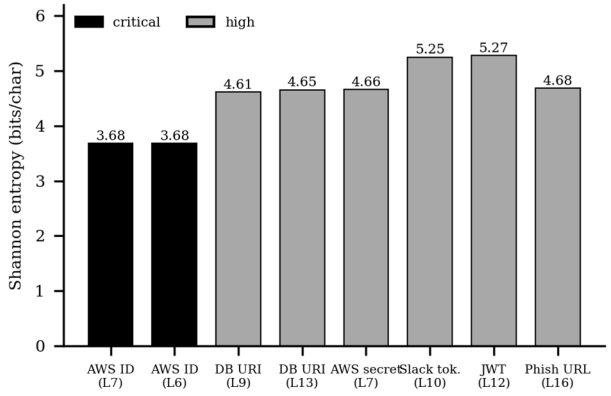


Fig. 11 Shannon entropy of the eight findings in Table V (black: critical, gray: high).

©IJRASET: All Rights are Reserved

G. Functional Validation Summary

TABLE VI
FUNCTIONAL VALIDATION OF OMNIGUARD AI COMPONENTS

Component	Input	Outcome
Engine 1	Two files with simulated credentials	Four findings per file with line, signature, entropy, severity (Figs. 6, 7)
Engine 1	Clean code templates	No false positives observed
Refactoring	Refactor on flagged file	Rewrite to os.getenv(); preview shown (Fig. 8)
Engine 2	One test URL	Risk score 15%; domain status Verified (Fig. 9)
Pre-commit hook	Staged credential patterns	Exit status 1 and block message, per script logic
Shield JS	AWS, Stripe, Slack patterns	Console warning and true, per script logic
CSV export	/export/csv	Five-column report, per route code

The hook, shield and CSV rows describe the behaviour of the implemented scripts and routes; Fig. 10 confirms that the three controls are exposed on the dashboard.

VIII. LIMITATIONS AND FUTURE WORK

OmniGuard AI is a working prototype, and the following limitations should be addressed.

1) *Evaluation scope:* results are functional checks on a few test files and one URL. Detection rate, precision and recall on a labelled corpus, and a statement that clean templates produce no false alarms, are still to be measured on larger and more varied code bases.
2) *Refactoring coverage:* the substitution matches only names ending in _KEY or _SECRET, or PASSWORD, and generates Python os.getenv() calls. Connection strings and tokens under other names are not rewritten, no import os is inserted when missing, and the file is overwritten in place without a backup.
3) *State and hardening:* findings are held in a process-wide list, so they are lost on restart and shared across users. Untrusted file names should be escaped before insertion into the dashboard, and the default SECRET_KEY must be replaced before deployment.
4) *Engine 2 calibration:* the risk score is heuristic and should be calibrated on labelled phishing datasets; the classifier and WHOIS components shown in the architecture need separate evaluation.
5) *Tool coverage:* the hook and shield.js rely on regex patterns only, and shield.js covers three credential formats.

Future work includes: (a) AI-driven contextual secret detection with transformer-based models to separate production keys from test fixtures; (b) CI/CD integration through GitHub Actions, GitLab CI and Jenkins for centralised gating of pull requests and builds; (c) live threat-intelligence feeds such as PhishTank, VirusTotal and Google Safe Browsing for domain reputation lookups; and (d) refactoring templates for Node.js (process.env), Java (System.getenv), Go and C#.

IX. CONCLUSION

This paper presented OmniGuard AI, an integrated, developer-centric platform that unifies static secret detection, automated remediation, shift-left developer tooling and heuristic phishing analysis. Engine 1 combines regex signatures with Shannon entropy to report hardcoded credentials with file, line and severity; the Git pre-commit hook and Credential Shield JS move enforcement to the developer workstation and the browser; the refactoring module turns detection into remediation by rewriting secrets into os.getenv() calls; and Engine 2 scores URLs from structural and lexical features. Telemetry and CSV audit export give administrators a complete record. The prototype shows that these controls can be combined in one lightweight Flask application, and the limitations above define the evaluation and hardening needed before field use.

©IJRASET: All Rights are Reserved

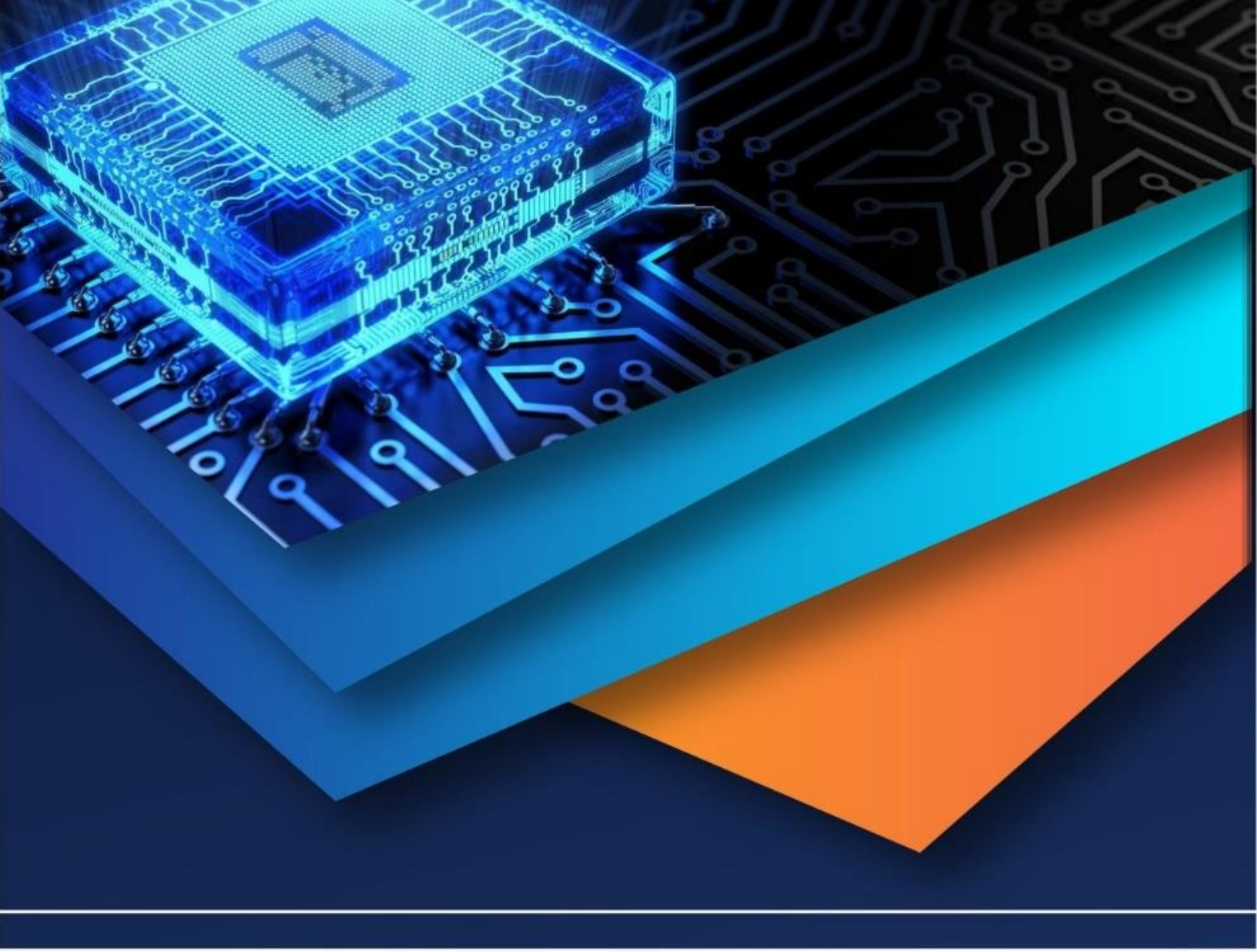
X. ACKNOWLEDGMENT

The authors thank B. Sathya, Assistant Professor, and G. Fathima, Professor and Head, Department of Computer Science and Engineering, Adhiyamaan College of Engineering, Hosur, for their guidance and encouragement, and R. Radhakrishnan, Principal, for the opportunity and environment provided for this work.

REFERENCES

[1] A. Smith and B. Johnson, “Phishing URL detection using machine learning and structural features,” *J. Cybersecurity Res.*, vol. 14, no. 1, pp. 12–25, 2026.
[2] K. Patel and R. Gomez, “Contextual machine learning for secret leakage detection,” *Int. J. Softw. Eng. Secur.*, vol. 12, no. 3, pp. 110–124, 2025.
[3] L. Zhang and H. Wang, “Real-time phishing detection via ensemble feature optimization,” *IEEE Trans. Inf. Forensics Security*, vol. 20, pp. 3450–3462, 2025.
[4] M. Davis and S. Taylor, “Shifting security left with Git pre-commit automation,” *J. Syst. Softw. Eng.*, vol. 9, no. 4, pp. 88–97, 2024.
[5] R. Anderson and C. Lee, “Hybrid credential exposure prevention in code assets,” *Comput. Secur.*, vol. 138, pp. 103–115, 2024.
[6] A. Kumar and R. Sharma, “Entropy-based static code analysis for credential and secret leakage detection in modern software assets,” *J. Syst. Softw. Secur.*, vol. 15, no. 4, pp. 112–128, 2022.
[7] R. Williams and D. Singh, “Browser runtime monitoring and credential shielding in web applications,” in *Proc. Int. Conf. Innovative Comput. Commun. (ICICC)*, 2023, paper 4523.
[8] T. Brown and L. Davis, “Interactive security operations dashboards for automated vulnerability remediation,” *IEEE Trans. Learn. Technol.*, vol. 17, no. 1, pp. 25–35, 2024.
[9] B. Ahmed and S. N. Malik, “Lexical feature extraction and heuristic evaluation systems for phishing analysis,” in *Proc. IEEE Int. Conf. Signal Process. Commun. (ICSPC)*, 2022, pp. 410–415.
[10] V. Chandra and P. Nair, “Automated compliance auditing and telemetry export frameworks for software security,” *Int. J. Artif. Intell. Appl. (IJAIA)*, vol. 11, no. 2, pp. 45–53, 2020.
[11] Pallets Projects, “Flask documentation.” [Online]. Available: https://flask.palletsprojects.com/
[12] Git SCM, “githooks – Hooks used by Git.” [Online]. Available: https://git-scm.com/docs/githooks
[13] Bootstrap Team, “Bootstrap.” [Online]. Available: https://getbootstrap.com/
[14] C. E. Shannon, “A mathematical theory of communication,” *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul. 1948.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)