RESEARCH ARTICLE OPEN ACCESS

WEBSHIELDX: Ethical Hacking and Pentest Insights

Ancil Teril, Deepu Raj D, Arjun M Babu, Devanand V A
Dept. of Computer Science & Engineering Toc H Institute of Science & Technology Kerala, India TOC22CS037

Ms. Aswathy Ramakrishnan
Assistant Professor, Dept. of CSE Toc H Institute of Science & Technology Kerala, India

***Abstract*—**SQL injection and brute-force attacks are among the oldest tricks in the book, yet they keep working because most deployed web applications have no reliable way to catch them early. This paper describes WebShieldX, a security monitoring system we built to close that gap. The setup is fairly straight- forward: we wrote scripts to simulate both attack types against a Flask web application, collected and stored the resulting logs, and trained a Random Forest model to tell malicious requests apart from normal ones. We used a Kaggle SQL injection dataset alongside a brute-force dataset we put together ourselves. Once trained, the model was plugged directly into the application so logs get classified as they arrive, and the results show up on an admin dashboard with priority labels so whoever is watching knows where to look first. The whole thing runs on a regular laptop with no GPU needed, and our tests showed it catches attacks noticeably more reliably than static rule-based filters.
***Index Terms*—**Web Application Security, SQL Injection, Brute- Force Attack, Random Forest, Penetration Testing, Machine Learning, Intrusion Detection, Flask Framework.

Date of Submission: 20-03-2026 Date of acceptance: 03-04-2026

I. INTRODUCTION

Cyberattacks against web applications have grown both in frequency and sophistication over the past several years. Incidents such as large-scale ransomware campaigns and data breaches affecting millions of users [9], [10] have made it clear that organizations relying purely on perimeter defenses are not adequately protected. At the application layer, attacks like SQL injection and brute-force login attempts remain stubbornly common, not because they are technically sophisticated, but because many deployed applications still lack the monitoring infrastructure to catch them early.

Traditional detection approaches tend to rely on predefined rules or signature matching, which works reasonably well against known patterns but struggles when attackers vary their payloads or slow down their attempts to avoid thresholds. Machine learning addresses this by learning what normal and abnormal traffic look like from data, making it better suited to detecting variations that a fixed rule would miss. The challenge, however, is that most intrusion detection research focuses on model performance in isolation, evaluated on benchmark datasets. Comparatively little work demonstrates these models working inside a real web application that an administrator can actually monitor and act on.

That gap is what motivated this project. WebShieldX is a complete prototype, not just a classifier, but a web application with an authentication flow, an attack log database, a trained Random Forest model, and a dashboard that surfaces results in a usable form.

A. Problem Statement

Despite improvements in security tooling, many web ap- plications still depend on manual log review and static in- put validation to catch attacks. These methods are time- consuming, inconsistent, and tend to generate noisy alerts that are difficult to prioritize. Without a smarter classification layer, administrators often end up either missing genuine attacks or spending too much time chasing false positives. The core problem is the absence of an integrated system that combines detection, classification, and structured reporting within the same application environment.

B. Proposed Solution

WebShieldX addresses this by combining four components into a single deployable system:

- **Flask-Based Web Application:** Handles authentication, records all request activity, and hosts both the monitoring dashboard and the ML analysis interface.
- **Attack Simulation and Log Collection:**

SQL injection and brute-force attacks are simulated using dedicated scripts, and every attempt is logged to a database in the same format as real traffic.

- **Random Forest Classification:** A model trained on labeled SQL injection and brute-force data classifies incoming logs and assigns each detection a confidence- based priority level.
- **Dashboard Reporting:** Detected threats are displayed through an admin dashboard with priority tiers, giving administrators a clear view of what needs attention and when.

II. LITERATURE REVIEW

A. Penetration Testing Methodologies

Kumar et al. [1] conducted a practical vulnerability assessment using Kali Linux and ZAP Proxy, demonstrating a successful exploit of a broken session management vulnerability on a real target site. While the approach was effective, the authors noted that automated scanners tend to produce a large number of false positives and are unable to identify zero-day vulnerabilities. This points to a broader limitation of purely signature-based tools and partly motivates the use of learned classifiers that can adapt to new patterns.

Al-Sinani and Mitchell [2] took a different direction with PenTest++, integrating ChatGPT-4o into the penetration test- ing workflow to automate tasks from reconnaissance through report generation. The tool reduces effort significantly, though the authors acknowledge that AI-generated outputs need care- ful human validation, and routing sensitive scan data through an external API introduces privacy considerations. Alhamed and Rahman [3] reviewed 39 papers published between 2018 and 2022, finding a clear trend toward automation and AI-driven approaches, with deep reinforcement learning identified as the most promising direction for scalable future systems.

B. Security Control Bypass and Tools

Cisar and Pinter [4] gave a broad survey of ethical hacking capabilities available in Kali Linux, covering tools for network scanning, exploitation, wireless testing, and web application assessment. Complementing this, Sinha and Thakare [5] fo- cused specifically on bypassing existing security controls — NAC, antivirus, and Windows UAC — using tools like the Veil Framework and Shellter. Their main argument is that testing whether controls can be evaded is just as important as testing whether initial access is possible. Li et al. [6] provide a structured walkthrough of the five-stage penetration testing process, with the useful observation that information gathering can take up more than 60% of total test time in practice.

C. Machine Learning for Intrusion Detection

Lu et al. [7] proposed an enhanced Random Forest com- bining Bald Eagle Search-optimized kernel PCA for dimensionality reduction with a cost-sensitive classifier to handle class imbalance. Evaluated on the UNSW-NB15 dataset, their method improved specificity by 11.7% over standard RF and reduced training time substantially. Azhar et al. [8] developed IDRandom-Forest, which applies stratified feature sampling and an Accuracy Sliding Window to prune the ensemble down to its most useful trees. On both UNSW-NB15 and CICIDS2017, the method achieved competitive accuracy while outperforming deep learning models on inference latency — an important property for real-time monitoring. These two works directly informed the classification design used in WebShieldX.

III. SYSTEM DESIGN AND METHODOLOGY

WebShieldX is structured around four modules that interact in a clear pipeline: requests enter through the Flask applica- tion, activity is logged to the database, the Random Forest classifier processes those logs, and results are pushed to the dashboard. Figure 1 illustrates this flow.

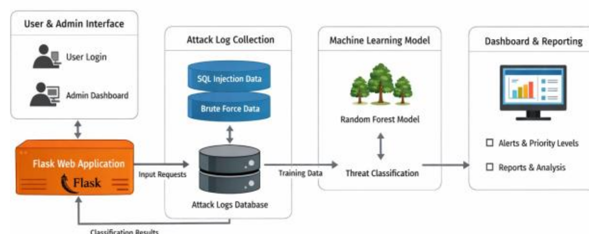

Fig. 1. WebShieldX framework architecture showing the four interconnected modules: Web Application Layer, Attack Log Collection, Machine Learning Classification, and Dashboard Reporting.

A. Module 1: Web Application Module

The Flask application is the entry point for all system activity. It hosts the admin login page, the monitoring dashboard, and the ML analysis interface. Access to these pages is protected with JWT-based authentication so that the monitoring system itself cannot be casually accessed.

Every request that passes through the application — a login attempt, a form submission, a query input — is recorded in the database with its username, input content, timestamp, IP address, and outcome. This logging happens uniformly for both real user activity and simulated attacks, which means the stored logs reflect realistic traffic rather than purely synthetic data.

B. Module 2: Attack Log Collection and Database Layer

Two datasets underpin the classification system. The first is a public SQL injection dataset from Kaggle, which con- tains query strings labeled as benign or malicious, covering patterns such as `UNION SELECT` statements, comment-based bypasses, and boolean blind injection attempts. The second is a custom brute-force dataset we built to capture repeated authentication behavior. Its features include username, IP address, timestamp, attempt count within a sliding window, and login outcome. Keeping the two datasets separate during preprocessing allowed us to evaluate the model on each attack type independently before combining them.

C. Module 3: Machine Learning Classification Module

The classifier is a Random Forest model with stratified feature sampling, based on the design in [8]. The core idea is that features vary considerably in how much they contribute to separating attacks from normal traffic, so weighting them accordingly during tree construction leads to better use of the available signal.

1) Stratified Feature Sampling: For each feature F_i in the training dataset, an informativeness score ϕ_i is computed and normalized to produce a weight y_i:

$$\phi_i = \text{informativeness}(F_i), \quad y_i = \text{normalize}(\phi_i) \quad (1)$$

A threshold τ then divides features into two groups:

$$\text{Group}(F_i) = \begin{cases} F_s & \text{if } y_i \le \tau \quad \text{(lower importance)} \\ F_w & \text{if } y_i > \tau \quad \text{(higher importance)} \end{cases} \quad (2)$$

Each decision tree draws features proportionally from both groups, and cost-sensitive splitting is applied to reduce the cost of missing attack-class instances, which are the minority in realistic traffic distributions.

2) Classification and Priority Assignment: When a log entry is submitted for classification, the model outputs a label — normal or malicious — alongside a confidence score. This is mapped to a three-tier priority system. High priority is assigned to SQL injection detections above a confidence threshold. Medium priority covers brute-force sequences that exceed a login frequency cutoff. Low priority captures flagged entries that do not fit clearly into either category. This tiering was added specifically to help administrators focus their at- tention rather than treating every alert as equally urgent.

D. Module 4: Dashboard and Reporting Module

The admin dashboard consolidates classification output into a readable summary. It shows running counts of total login attempts, failed attempts, unique source IPs, and successful logins, alongside a live alert panel for high-priority detections. The ML analysis page goes further, listing individual detections with their classification labels, source IPs, and timestamps, and showing the live status of both trained models. Figure 1 shows the overall data flow; the interface screens are shown in Section V.

IV. IMPLEMENTATION

A. Data Collection and Preprocessing

1) SQL Injection Dataset: We pulled the SQL injection dataset from Kaggle, which gave us **6,186 labeled query records** to work with. The malicious side covers a good range of techniques — from blunt `DROP TABLE` commands to quieter boolean-based patterns that try to extract data without raising obvious flags. An 80/20 stratified split left us with around 4,949 training samples and 1,237 for testing.

2. *Brute-Force Dataset:* Since no public brute-force dataset

matched what our application actually logs, we built one ourselves. It ended up with **1,127 records**, each capturing a username, IP address, timestamp, how many login attempts came from that IP within a five-minute window, and whether the attempt succeeded. Normal entries look like what you would expect from a real user; attack entries are rapid, repeated attempts across many username–password combinations from the same source. The same 80/20 split gave us around 901 training samples and 282 for testing — 188 normal and 94 brute-force — which is what the confusion matrix in Figure 2 reflects.

3. *Preprocessing Steps:* Both datasets went through the same preprocessing pipeline:

- **Data Cleaning:** Duplicate and incomplete records were removed. Rows missing the label, query string, or times- tamp fields were dropped entirely rather than imputed.
- **Missing Value Handling:** Where minor fields were absent, appropriate imputation was applied to maintain the dataset size without introducing bias.
- **Categorical Encoding:** Text and categorical fields were label-encoded to numerical representations that preserve ordinal relationships where they exist.
- **Feature Selection:** Pearson correlation between each feature and the target label was computed; features below a threshold of 0.05 were excluded. This reduced noise and kept training efficient.
- **Train-Test Split:** An 80/20 stratified split was used, ensuring that the class ratio was preserved in both the training and test sets.

B. Attack Simulation and Log Generation

A standalone Python script drives attack simulation sepa- rately from the main application. Brute-force simulation works by submitting repeated POST requests to the login endpoint using randomized username and password pairs drawn from a common wordlist, with request timing varied to cover both fast and slow attack patterns. SQL injection simulation injects a curated set of known payloads into the application's input fields. Every simulated request is logged to the database in exactly the same format as a genuine request, so the classifier encounters no formatting difference between training data and live data.

C. Training Configuration

The model was trained using Python's `scikit-learn` library on a standard development machine. No GPU was required. Key configuration details are summarized in Table I.

TABLE I
WEBSHIELDX SYSTEM COMPONENTS

Component	Detail
Web Framework	Flask (Python 3.x)
ML Library	scikit-learn
Algorithm	Enhanced Random Forest
SQL Injection Data	Kaggle (6,186 samples)
Brute-Force Data	Custom-created (1,127 samples)
Database	SQLite / MySQL
Frontend	HTML, CSS, JavaScript
Attack Types	SQL Injection, Brute-Force
Priority Tiers	High, Medium, Low
Evaluation Metrics	Accuracy, Precision, Recall, F1-Score
Hardware	Intel i5/i7, 8 GB RAM

- **Accuracy:** Proportion of all instances classified correctly.
- **Precision:** $TP/(TP + FP)$ — of everything the model flagged as an attack, how much actually was one.
- **Recall:** $TP/(TP + FN)$ — of all real attacks in the test set, how many the model caught.
- **F1-Score:** Harmonic mean of precision and recall, useful when the class distribution is uneven.

B. Detection Results

SQL injection detection was the stronger of the two tasks. The query-level features — keyword patterns, structural anomalies — are distinct enough that the classifier had little difficulty separating malicious from benign inputs. Brute-force detection depended more heavily on the temporal features, particularly login attempt frequency within the sliding window. Tuning that window size during feature engineering had a visible effect on recall: a window that was too wide smoothed over the attack signal, while one that was too narrow missed slower-paced attacks. The final configuration gave a good balance between the two.

Looking at the numbers in Table II, the SQL injection model did about as well as we could reasonably hope for. On 6,184 test samples, it hit an overall accuracy of 1.00, with a precision of 0.99 on normal traffic and a perfect 1.00 on injection queries. Recall was 1.00 and 0.99, respectively, giving a macro F1 of 0.99. In plain terms, the model missed 25 actual injections and incorrectly flagged 5 normal queries — 30 mistakes out of over six thousand samples.

TABLE II
SQL INJECTION MODEL – CLASSIFICATION REPORT

Class	Precision	Recall	F1-Score	Support
Normal (0)	0.99	1.00	1.00	3893
SQL Injection (1)	1.00	0.99	0.99	2291
Accuracy			1.00	6184
Macro avg	1.00	0.99	0.99	6184
Weighted avg	1.00	1.00	1.00	6184

The brute-force model tells a similarly clean story. Table III shows the full classification report: precision, recall, and F1-score all came out at 1.00 across both classes, with a test accuracy of 1.00 on 282 samples. Every one of the 188 normal instances and all 94 brute-force instances were correctly labeled, as the confusion matrix in Figure 2 confirms. That kind of result is unusual, but it makes sense given how distinctive the temporal features are once the sliding window size is right — legitimate users simply do not send dozens of login attempts from the same IP within a five-minute window.

TABLE III
BRUTE-FORCE MODEL – CLASSIFICATION REPORT

Class	Precision	Recall	F1-Score	Support
Normal (0)	1.00	1.00	1.00	188
Brute Force (1)	1.00	1.00	1.00	94
Accuracy			1.00	282
Macro avg	1.00	1.00	1.00	282
Weighted avg	1.00	1.00	1.00	282

The SQL injection confusion matrix is shown in Figure 3. From a security standpoint, the number that matters most is false negatives — attacks the model quietly lets through — and both models kept that low. The 25 missed injections on the SQL side are the only real gap, and they would show up as undetected events rather than false alarms, which is worth watching in a production setting. The handful of false positives would appear as minor noise in the alert feed, but not at a volume that would cause an administrator to start ignoring them.

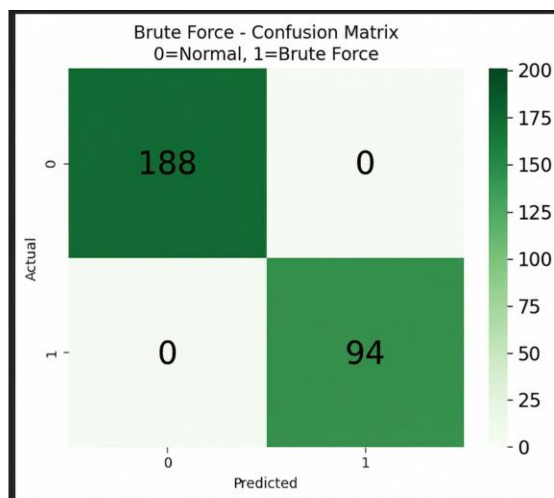

Fig. 2. Confusion matrix for the Brute-Force detection model on the held-out test set (282 samples: 188 normal, 94 brute-force). Zero misclassifications in either direction.

C. User Interface

1) *Admin Login Interface:* The login page is kept deliberately simple. Credential fields are clearly labeled, and a small notice informs users that login activity is logged for security purposes. Figure 4 shows the interface.

2) *Admin Dashboard:* The main dashboard, shown in Figure 5, presents running totals for login attempts, failures, successful logins, and unique source IPs. The right-hand panel

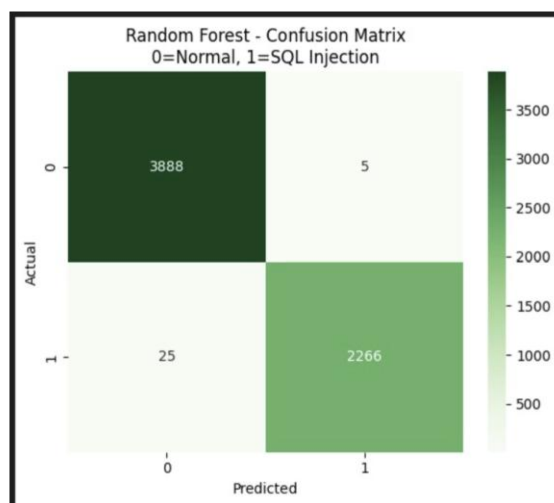

Fig. 3. Confusion matrix for the SQL Injection detection model on the held- out test set (6,184 samples). Only 30 misclassifications out of 6,184 instances: 5 false positives and 25 false negatives.

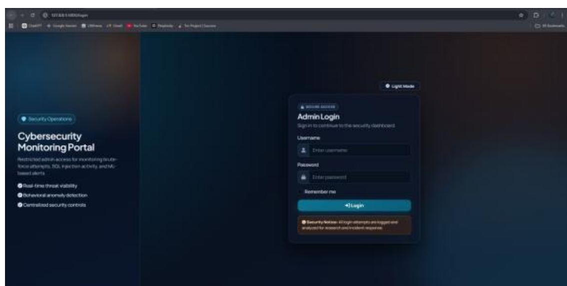

Fig. 4. Admin login interface. All login attempts, including failed ones, are recorded in the attack log database and fed to the classifier.

lists active security alerts with timestamps and severity. This layout means an administrator can scan the screen in a few seconds and know whether anything unusual is happening.

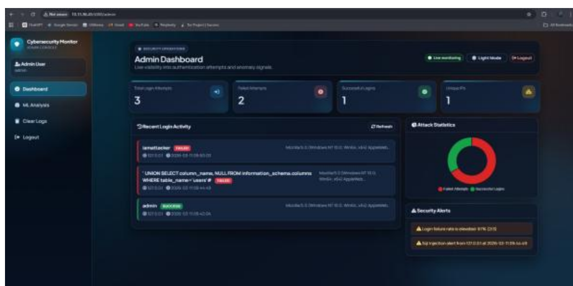

Fig. 5. Admin dashboard showing live security monitoring with summary counters and a real-time alert panel.

3) ML Analysis Dashboard: Figure 6 shows the ML analysis page. Both trained models — brute-force and SQL injection — are listed with their status, and a table of recent detections shows each event's classification label, source IP, and detection time. This page is where an administrator would go to investigate a specific alert in more detail or look for patterns across multiple events.

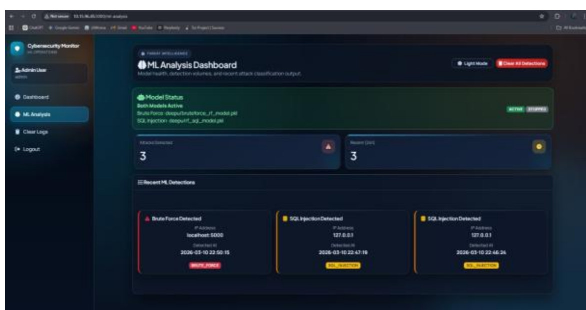

Fig. 6. ML analysis dashboard showing model status, total detections, and a per-event breakdown with attack-type labels and source details.

V. CONCLUSION AND FUTURE SCOPE

This paper presented WebShieldX, a Flask-based web ap- plication security monitoring system that combines attack simulation, Enhanced Random Forest classification, and a structured admin dashboard to detect SQL injection and brute-force threats. The system was built with practical usability in mind: it runs on ordinary hardware without specialized infrastructure and presents detection results in a way that administrators can interpret and act on without needing to understand the underlying model.

The results confirmed that tying the classifier directly into the application environment, rather than running it as an offline analysis step, improves detection timeliness and makes the system genuinely useful in a monitoring context. The priority- level scheme further reduces the effort required to triage alerts.

Future work will focus on:

- **Extended Attack Coverage:** Adding detection for Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and command injection to make the system useful against a broader threat surface.
- **Real-Time Stream Processing:** Moving from batch log analysis to stream-based processing to reduce the gap between when an attack occurs and when it is flagged.
- **Deep Learning Integration:** Exploring LSTM networks or Transformer-based models for better detection of temporally distributed attacks, such as slow-rate brute-force attempts that stay under per-request frequency thresholds.
- **Automated Response Mechanisms:** Adding the ability to automatically block or rate-limit detected attack sources, rather than only alerting and waiting for a human to act.
- **Cloud Deployment:** Adapting the system for cloud- hosted web applications where traffic volumes and in- frastructure constraints differ from a local deployment.

REFERENCES

[1] B. Kumar, S. P. Bejo, R. Kedia, P. Banerjee, P. Jha, and M. K. Dehury, "Kali Linux based Empirical Investigation on Vulnerability Evaluation using Pen-Testing tools," in *Proc. 2023 World Conf. on Communication & Computing (WCONF)*, Raipur, India, 2023, pp. 1–6.

[2] H. S. Al-Sinani and C. J. Mitchell, "PenTest++: Elevating Ethical Hacking with AI and Automation," *arXiv preprint arXiv:2502.09484v1*, 2025.

[3] M. Alhamed and M. M. H. Rahman, "A Systematic Literature Review on Penetration Testing in Networks: Future Research Directions," *Applied Sciences*, vol. 13, no. 12, p. 6986, 2023.
[4] P. Cisar and R. Pinter, "Some ethical hacking possibilities in Kali Linux environment," *J. Applied Technical and Educational Sciences (jATES)*, vol. 9, no. 4, pp. 129–149, 2019.
[5] V. A. Sinha and V. M. Thakare, "A Study of Bypassing Security Control Methods in Kali Linux Environment," *Int. J. Engineering Research & Technology (IJERT)*, vol. 11, no. 8, pp. 56–61, 2022.
[6] W. Zhang, J. Xing, and X. Li, "Penetration Testing for System Security: Methods and Practical Approaches," *arXiv preprint arXiv:2505.19174v1*, 2025.
[7] C. Lu, Y. Cao, and Z. Wang, "Research on Intrusion Detection Based on an Enhanced Random Forest Algorithm," *Applied Sciences*, vol. 14, no. 2, p. 714, 2024.
[8] M. Azhar, S. Perveen, A. Iqbal, and B. Lee, "IDRandom-Forest: Ad- vanced Random Forest for Real-time Intrusion Detection," *IEEE Access*, vol. 12, pp. 1–17, 2024.
[9] J. Tidy, "Swedish Coop supermarkets shut due to US ran- somware cyber-attack," *BBC News*, Jul. 2021. [Online]. Available: https://www.bbc.com/news/technology-57707530
[10] The Associated Press, "T-Mobile says breach exposed personal data of 37 million customers," *NPR*, Jan. 2023. [Online]. Available: https://www.npr.org/2023/01/20/1150215382/t-mobile-data-37-million- customers-stolen
[11] Panadiya et al., "ML-Based SQL Injection Detection," 2024 https://www.researchgate.net/publication/387123538
[12] Vinayakumar et al., "Deep Learning Approach for IDS," IEEE Access, 2019 https://ieeexplore.ieee.org/document/8768877
[13] Xin et al., "Machine Learning and Cybersecurity," IEEE Access, 2018 https://ieeexplore.ieee.org/document/8352613
[14] Sarhan et al., "ML-Based Intrusion Detection Systems," Elsevier, 2024 https://doi.org/10.1016/j.future.2022.12.016
[15] M. Javaid, A. Niyaz, W. Sun, and M. Alam, "A Deep Learning Approach for Network Intrusion Detection System," in *Proc. IEEE Conf.*, 2016. [Online]. Available: https://ieeexplore.ieee.org/document/7874638