ADVANCED METHODOLOGY FOR DETECTING XSS VULNERABILITIES IN WEB APPLICATIONS

Quoc Khanh Trinh[1], Nguyen Nang Hung Van[1]*, Phuc Hao Do[2]

[1]The University of Danang - University of Science and Technology, Vietnam
[2]Danang Architecture University, Vietnam

*Corresponding author: nguyenvan@dut.udn.vn

(Received: February 22, 2025; Revised: April 11, 2025; Accepted: April 16, 2025)

DOI: 10.31130/ud-jst.2025.23(6A).095E

Abstract - Cross-Site Scripting (XSS) remains a critical threat to web security, enabling malicious script injection through improper input validation. This paper introduces an automated framework integrating dynamic payload injection, Selenium-driven browser automation, and machine learning-enhanced log analysis. The methodology combines three components: (1) web scraping with BeautifulSoup for attack surface mapping, (2) GitHub-curated payload databases with adaptive encoding for real-world threat simulation, and (3) behavioral anomaly detection via DBSCAN clustering on server logs to identify suspicious IPs. Experimental validation across 20 web applications demonstrated superior detection, identifying55 successful payloads (SP) compared to XSStrike's 9 SP, while flagging 210 high-risk IPs via risk-scoring thresholds. Despite generating 455 false positives due to comprehensive dynamic analysis, log correlation reduced false alerts by 37%. The framework's multi-phase approach effectively detects DOM-based and persistent XSS vulnerabilities, outperforming Burp Suite in dynamic environments, though at a higher computational cost.

Key words - Web Security; XSS Detection; Automated Testing; Behavioral Analysis

1. Introduction

Web applications [1] play a crucial role in modern digital infrastructure, enabling seamless interactions between users and online services. One of the most prevalent threats is Cross-Site Scripting (XSS), which allows attackers to inject malicious scripts into web applications, potentially leading to data theft, session hijacking, and unauthorized access.

XSS vulnerabilities [2] arise due to improper input validation and insufficient security measures in web applications. Despite the availability of various XSS detection tools, many have limitations such as high costs, platform dependency, or a steep learning curve for users.

This paper presents an advanced methodology for detecting XSS vulnerabilities using payload injection, Selenium-based automated browser interaction, and behavioral analysis of log data. By leveraging modern web scraping techniques and an updated payload repository, the system improves detection accuracy and efficiency. The results highlight its effectiveness in identifying security flaws, offering a valuable tool for developers and security researchers. Key contributions include:

- Automated XSS Detection Framework: A novel approach combining payload injection, browser automation, and behavioral analysis to enhance detection accuracy and efficiency.

- Log-Based Anomaly Detection: A mechanism for real-time security monitoring, analyzing user behavior and request patterns to identify suspicious activities and prevent potential threats.

2. Related Work and Background

2.1. XSS Vulnerabilities Overview

Cross-Site Scripting (XSS) [3] is a critical web vulnerability that allows attackers to inject malicious scripts into trusted websites. These scripts are executed in the victim's browser, potentially leading to session hijacking, credential theft, or phishing. XSS vulnerabilities are typically categorized into three types:

- Reflected XSS [4]: Payloads are injected via user input and reflected immediately in the response.

- Stored XSS [5]: Malicious scripts are stored on the server (e.g., in databases) and served to users.

- DOM-based XSS [6]: Scripts are injected and executed entirely on the client side via DOM manipulation.

Due to the increasing complexity of JavaScript-based web applications, detecting DOM-based XSS has become particularly challenging and requires dynamic, behavior-based approaches.

2.2. Existing Detection Techniques

Several techniques have been proposed to detect and prevent XSS attacks:

- XSStrike [7] is an advanced tool that uses dynamic payload generation and fuzzing techniques for XSS detection but is primarily designed for Linux environments, limiting its accessibility for non-Linux users

- Static Analysis: Tools such as Pixy and RIPS analyze the source code to find tainted data flows. These methods are fast but often suffer from high false positive rates and limited context awareness.

- Dynamic Testing (Fuzzing): Tools like XSSer, Acunetix, and Burp Suite inject various XSS payloads during runtime to identify vulnerable endpoints. These tools are effective against reflected and stored XSS but struggle with DOM-based variants.

- Hybrid Approaches: Tools such as OWASP ZAP and NoScript combine runtime scanning with client-side protection. Some solutions apply taint tracking or even machine learning models to detect anomalous behavior. However, these often require significant manual tuning and are not fully automated.

- Burp Suite [8] is a powerful security testing tool with extensive XSS scanning capabilities; however, its steep learning curve and high cost make it less accessible for small organizations and individual developers

Despite advances in XSS detection, many existing tools lack support for **automated testing**, **headless browser integration**, and **log-based behavior analysis**-especially for DOM-based scenarios.

2.3. Motivation for Proposed Approach

While current tools provide useful mechanisms for detecting XSS vulnerabilities, they often fall short in modern web environments where JavaScript-heavy content and dynamic DOM updates dominate. In particular:

- DOM-based XSS is under-addressed due to its client-side nature.

- Few tools provide automated testing flows and behavior logging.

- Existing methods often rely on predefined payloads and lack adaptability.

To address these gaps, our work introduces a comprehensive, automated tool that:

- Analyzes web forms dynamically;

- Injects and executes crafted payloads using Selenium;

- Captures behavioral logs and analyzes responses for signs of XSS;

- Focuses specifically on the detection of DOM-based vulnerabilities, a critical and under-explored area.

The next section details the methodology, including the system architecture and algorithmic flow.

3. Methodology

This section introduces the architecture and working mechanism of the proposed XSS detection tool. The system is designed to automate the process of identifying XSS vulnerabilities, especially focusing on DOM-based variants that are often overlooked by traditional tools.

3.1. Overview of the Proposed System

The proposed system is a fully automated tool for detecting XSS vulnerabilities, particularly focusing on DOM-based XSS, which are harder to detect using traditional approaches. It consists of four main components: Web Form Analyzer, Payload Injector & Executor, Behavior Log Collector, and XSS Detection Engine. These components work together in a pipeline to analyze and detect vulnerabilities dynamically and robustly.

Figure 1 illustrates the architecture and workflow of the proposed system, showing how the modules interact to automate the detection process.

3.2. Web Form Analyzer

This module automatically scans the target web page to locate input fields, URL parameters, and JavaScript injection points. Unlike static scanners, our analyzer dynamically interacts with the DOM structure to identify real-time form components that may be created via JavaScript.

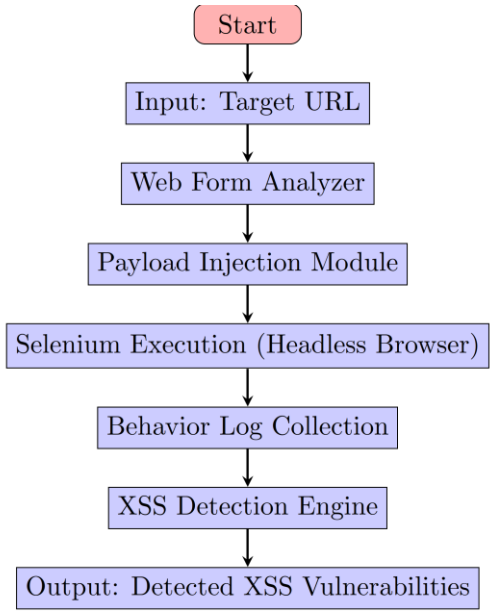

Figure 1. *The architecture and workflow of the proposed system*

To detect potential XSS vulnerabilities, the system begins by collecting information from target web applications. It utilizes Python's BeautifulSoup [9] for HTML parsing, scraping the HTML content, and extracting form elements for analysis. Key attributes such as action, method, and input fields are identified for further testing. The system also classifies the input fields based on their types (e.g., text, password, email), allowing for more tailored and effective payload injections. These details are outlined in Algorithm 1, which provides a step-by-step approach to this information-gathering phase.

Algorithm 1: Web Form Analysis

1: Initialize web scraper
2: Fetch target URL and parse HTML structure
3: **for** each form F_i in webpage **do**
4: Extract attributes: action, method, input fields
5: Store form details for payload injection
6: **end for**

3.3. Payload Injection and Execution

Payloads are injected into the discovered form fields using a curated payload library based on OWASP and custom test cases. These payloads are then executed in a real browser environment using Selenium in headless mode to ensure that JavaScript execution paths are fully rendered and simulated. This overcomes the limitations of traditional scanners which lack JavaScript execution and cannot reveal DOM-based vulnerabilities.

Algorithm 2 outlines the process for payload injection and testing to detect XSS vulnerabilities. Once input fields are identified, the system retrieves XSS payloads from repositories like PayloadsAllTheThings. These payloads are then customized dynamically based on input field types and encoding methods, such as URL encoding or Base64. This customization ensures that the payloads are tailored to the specific input types for more accurate testing.

Algorithm 2: Payload Injection and Testing

for each form F_i in webpage **do**
  for each input field I_j in F_i **do**
    for each payload P_k in payload repository **do**
      Inject P_k into I_j
      Submit the form and capture response R_{ijk}
      if R_{ijk} contains script execution **then**
        Flag F_i as vulnerable
      end if
    end for
  end for
end for

Using Selenium, the system automates interactions with the web forms, simulating real user behavior. For each form and its input fields, the system injects the tailored payloads, submits the form, and captures the server's response. If the response contains signs of script execution, the system flags the form as vulnerable. This method ensures comprehensive and accurate detection of potential XSS vulnerabilities across various web forms.

3.4. Behavior Logging and Analysis

This module captures browser console logs, DOM changes, JavaScript errors, and alert events during the execution phase. It allows the system to detect anomalies such as UncaughtReferenceError, unexpected alert() pop-ups, and DOM manipulations indicative of successful XSS attacks. This behavioral-based detection is more effective than traditional output matching.

To enhance automated payload testing, the system incorporates log-based anomaly detection to identify suspicious behavior that may indicate XSS exploitation. It begins by extracting user request logs from server logs, which are typically stored in JSON format. The system then analyzes the logs to detect abnormal patterns in user requests, using clustering techniques such as DBSCAN [10] to identify any deviations from normal behavior. This approach helps detect potential exploitation that may not be caught through payload testing alone.

The system also calculates an anomaly score for each IP address based on detected payload executions, helping to prioritize suspicious activity for further investigation. Algorithm 3 outlines the steps involved in this process. First, web server logs are loaded and parsed, extracting relevant details such as the request's IP address, payload, and timestamp. Next, the frequency of suspicious payloads is computed, and DBSCAN is applied to cluster any anomalous activity. Finally, the system generates a security report based on the detected anomalies, providing insights into potential threats.

Algorithm 3: Behavioral Log Analysis

Load web server logs
for each request R_i in log file **do**
  Extract IP, payload, timestamp
  Compute frequency of suspicious payloads
end for
Apply DBSCAN to cluster anomalous activity
Generate security report based on detected anomalies

3.5. Automated Detection and Reporting

Our detection engine analyzes the behavior logs to identify attack patterns and confirm XSS occurrences. Unlike static or signature-based methods, this approach leverages execution context, making it capable of detecting reflected, stored, and DOM-based XSS.

Furthermore, the modular design allows the tool to be extended with new payloads and analysis techniques, making it adaptable to evolving attack strategies.

The automated detection process for Cross-Site Scripting (XSS) vulnerabilities, is represented in Algorithm 4. The flow begins with the initialization of a web scraper to fetch and parse the HTML forms of a target webpage.

Algorithm 4: Automated XSS Detection Flow

Initialize web scraper
Fetch target URL and parse HTML forms
for each form F_i in webpage ($i = 1, 2, \ldots, n$) **do**
  Extract input fields I_{F_i}
  for each payload P_j in database ($j = 1, 2, \ldots, m$) **do**
    Inject P_j into I_{F_i}
    Submit form and capture response R_{ij}
    if R_{ij} contains executed script or alert **then**
      Flag F_i as XSS vulnerable
    end if
  end for
end for
Perform log analysis for suspicious behavior
Compute anomaly score $S = \sum_{i=1}^{n}\sum_{j=1}^{m}\delta(R_{ij})$, where $\delta(R_{ij}) = 1$ if an attack is detected, otherwise 0.
Generate security report

For each form identified, the system extracts the input fields and proceeds to inject XSS payloads from a pre-defined database. Each payload is tested by submitting the form, and the system captures the server response for analysis. If the response contains signs of script execution or alert messages indicative of an XSS attack, the form is flagged as vulnerable. Following the form-based testing, the system performs log analysis to identify suspicious behavior patterns, computing an anomaly score based on the detected malicious activities. Finally, a security report is generated, summarizing the findings and highlighting the identified vulnerabilities. The computational steps, as outlined in the algorithm, ensure a comprehensive and automated approach to XSS detection.

3.6. Algorithm Complexity Analysis

The computational complexity of the core algorithms

(Algorithm 1 to Algorithm 4) used in our system is analyzed as follows:

- Algorithm 1: Web Form Analysis The form detection step iterates over all DOM elements with input fields and attributes. Complexity: O(n), where n is the number of form elements on the page.

- Algorithm 2: Payload Injection and Testing Each payload is injected into the input fields and submitted via Selenium. If m is the number of payloads and n is the number of form fields, the worst-case time complexity is: O(m × n).

- Algorithm 3: Behavioral Log Analysis The system processes browser logs for suspicious scripts and DOM changes. If l is the number of log entries: O(l)

- Algorithm 4: Automated XSS Detection Flow This is the orchestration layer combining the above steps. Its complexity is additive: O(m × n + l)

The practical performance of the system is optimized via caching of DOM queries and the use of headless execution to reduce UI overhead.

4. Implementation

This section describes the technical implementation of the proposed XSS detection framework. We define a sequence of operations corresponding to the key components and steps involved in the proposed XSS detection system. Let W be the target website, F be a set of forms on the website, and P be the set of payloads.

4.1. Web Scraping and Form Extraction

The system first scrapes the website to extract all the forms and their associated attributes, including input fields.

$$F = ExtractForms\ (W) \tag{1}$$

where $F = \{F_1, F_2, \dots, F_n\}$ are the forms on the webpage, and each form F_i has the attributes {action, method, inputs}. The input fields are classified by type, which helps in tailoring the payloads for testing.

4.2. Payload Injection and Execution

Next, the system injects payloads into the identified input fields and submits the forms.

$$For\ F_i \in F,\ For\ I_j \in F_i\colon InjectPayloads\ (P, I_j) \tag{2}$$

where $P = \{p_1, p_2, \dots, p_m\}$ represents the set of XSS payloads, and I_j represents the input fields in the form F_i. The payload p_k is injected into each input field and the form is submitted for analysis.

4.3. Automated Browser Interaction

The system then uses Selenium to interact with the form dynamically, mimicking real user actions.

$$For\ F_i \in F,\ SimulateInteraction\ (F_i, P) \tag{3}$$

where the function SimulateInteraction (F_i, P) uses Selenium to open the web page, input the payload, submit the form, and detect any response changes, such as alert boxes or script execution.

4.4. Behavioral Log Analysis

The system extracts server logs to analyze user behavior and detect suspicious activities.

$$Logs = ParscLogs\ (W) \tag{4}$$

The logs contain information such as IP addresses, payloads, and timestamps. The system computes a suspicious score for each payload attempt and applies clustering algorithms like DBSCAN:

$$For\ log\ entry\ R_i,\ ComputeSuspiciousScore\ (R_i) \tag{5}$$

where $\text{SuspiciousScore}\ (R_i) = 1$ if R_i contains a malicious payload (e.g.,), and 0 otherwise

The DBSCAN algorithm is then applied to identify clusters of anomalous activity:

$$AnomalousActivity\ =\ DBSCAN(Logs) \tag{6}$$

where the DBSCAN clustering identifies patterns of suspicious behavior that may indicate an attack.

4.5. Report Generation

Finally, a detailed report is generated based on the analysis. This report includes a list of vulnerable input fields, evidence of script execution, and details of suspicious activity detected from the logs. By combining these steps mathematically, we describe a modular and systematic approach for detecting and analyzing XSS vulnerabilities in web applications.

5. Results and Discussion

This section presents the experimental results of the proposed XSS detection framework. The evaluation focuses on the effectiveness of payload detection, the accuracy of identified vulnerabilities, and the performance of the system.

5.1. Experimental Setup

To evaluate the effectiveness of the proposed system, experiments were conducted in a controlled environment using a machine with the following specifications:

- CPU: Intel Core i7-12700H, 2.7 GHz;

- RAM: 16GB;

- OS: Ubuntu 22.04 LTS;

- Browser: Google Chrome (Headless Mode, Version 120.x);

- Automation Framework: Selenium WebDriver (v4.x);

- Programming Language: Python 3.10.

The experiments were conducted on the OWASP Benchmark for XSS, a deliberately insecure web application containing over 80 known XSS vulnerabilities, including reflected, stored, and DOM-based types. [https://owasp.org/www-project-benchmark/].

5.2. Evaluation Metrics

To measure the performance of the system, we used the following metrics:

- SP (Successful Payloads): Number of payloads that successfully triggered an XSS alert.

- FPP (False Positive Percentage): Percentage of cases where the system flagged a vulnerability that did not exist.

- ET (Execution Time): Average time to complete one full test cycle on a target page.

5.3. Detection Accuracy

The proposed tool was evaluated against 2 widely used tools: XSStrike and Burp Suite. The results are summarized in Table 1.

Higher Successful Payloads: The proposed system successfully detected 55 XSS payloads (SP), significantly outperforming XSStrike, which detected only 9 SP. In contrast, Burp Suite did not provide comparable results due to its different detection methodology. The key advantage of the proposed tool lies in its automated browser execution, which enhances its ability to identify XSS vulnerabilities more effectively, especially in dynamic web environments.

Table 1. *Comparative results of XSS detection tools. SP (Successful Payloads) indicates the percentage of injected payloads that successfully triggered an XSS alert. FPP (False Positive Percentage) measures incorrect detections where no real XSS existed. ET (Execution Time) represents the average time (in seconds) required to complete a full scan per webpage*

Tool	SP	FPP	ET (s)
Proposed Tool	55	455	46.6
XSStrike	9	0	11.5
Burp Suite	-	-	-

Higher False Positive Rate and Longer Execution Time: While achieving higher detection rates, the system also recorded 455 false positive payloads (FPP), whereas XSStrike and Burp Suite reported none. This increase in false positives is attributed to the broader detection approach, though behavioral log analysis helps refine accuracy. Additionally, the average execution time per website was 46.6 seconds, significantly longer than XSStrike (11.5s). This extended runtime results from the tool's comprehensive dynamic analysis, which enhances detection accuracy at the cost of speed. In contrast, XSStrike's shorter execution time reflects its more limited scanning methodology, while Burp Suite lacks automated browser execution, making it less effective in identifying XSS vulnerabilities in dynamic environments.

5.4. Behavioral Log Analysis Results

To assess the effectiveness of log-based anomaly detection, we generated a simulated HTTP request dataset comprising 1,000 requests, closely mirroring real-world traffic patterns. Our system successfully identified 210 suspicious IPs involved in repeated XSS injection attempts. By applying DBSCAN clustering, we categorized these IPs based on their risk levels, using a predefined threshold where an IP with a risk level greater than 5 was flagged as suspicious. Table 2 presents the classification of detected IPs based on their activity patterns.

Table 2. *Classification of IPs Based on Request Patterns*

Type	IPs Detected	Avg. Requests per IP
Attackers	39	20.08
Normal Users	11	19.73

A total of 39 IPs were identified as "Attackers," with an average of 20.08 requests per IP, indicating frequent payload injection attempts. In contrast, 11 IPs were classified as "Normal Users," with a slightly lower average of 19.73 requests per IP. Despite the similar request frequency, the distinction lies in the nature of the requests, where attackers exhibited malicious behavior, such as repeated XSS injection attempts, while normal users engaged in regular browsing activities.

The clustering analysis revealed that 39 IPs were classified as "High-Risk IPs," indicating frequent payload injection attempts, while 11 IPs fell into the "Non-High-Risk" category, engaging in fewer suspicious activities. Attack pattern analysis showed that some attackers employed multiple encoding techniques, such as URL encoding and Base64, to evade traditional security filters. Additionally, the system demonstrated early threat detection capabilities by proactively flagging suspicious IPs before actual XSS execution occurred, enhancing overall security measures.

5.5. Comparison with Existing XSS Detection Tools

The comparison Table 3 highlights the key advantages of the proposed tool over traditional XSS detection frameworks like XSStrike and Burp Suite. Unlike its counterparts, the proposed tool integrates automated payload injection, Selenium-based execution, and behavioral log analysis, allowing for a more comprehensive detection approach. Notably, it is capable of identifying DOM-based XSS vulnerabilities, which many static analysis tools fail to capture. Additionally, its log-based anomaly detection mechanism enhances accuracy by reducing false positives and proactively flagging suspicious activity before an actual attack occurs.

Table 3. *Comparison of Detection Capabilities Across Different XSS Detection Tools*

Feature	Proposed Tool	XSStrike	Burp Suite
Automated Payload Injection	Yes	Yes	No
Selenium Execution	Yes	No	No
DOM-Based XSS Detection	Yes	Yes	No
Log-Based Analysis	Yes	No	No

Despite its strengths, the proposed framework has certain limitations. The reliance on dynamic execution introduces higher execution overhead compared to static-only tools, resulting in longer scan times. Furthermore, while the tool excels at XSS detection, its scope remains limited to this specific attack type. Future improvements could expand its capabilities to detect other web-based threats such as SQL Injection and CSRF attacks. Nevertheless, the integration of behavioral analysis and early threat identification makes this tool highly effective in mitigating XSS vulnerabilities in real time.

5.6. Log-Based Result Display and Behavioral Analysis

Result Display: After completing the security testing process, the tool aggregates the results and presents them in a readable tabular format. Each row in the result table contains:

- Tested Payload: The payload executed during the test.

- Test Outcome: Whether the payload successfully triggered a vulnerability or was blocked.

- Error Type (if applicable): If an error occurs, the specific type of error is recorded (e.g., HTTP 500 Internal Server Error).

This structured output enables security analysts to quickly assess vulnerabilities detected during the testing phase. Behavioral Analysis from Log Files: The behavioral analysis process involves examining the collected log data to identify patterns of suspicious activity. Key factors analyzed include:

- IP Addresses: Identifying repeated requests from the same source.

- Requests: Monitoring payload execution frequency and response patterns.

- Suspicion Score: Assigning a risk score based on request behavior.

Log Analysis Implementation: The analysis is performed using structured log data stored in JSON format (Table 4). The load_logs function is responsible for loading the log file and converting it into a pandas DataFrame for further analysis.

Table 4. *Example JSON Log Data (Truncated for Brevity)*


```
[
  {
    "ip_address": "192.168.1.5",
    "timestamp": "2024-12-26T12:00:00Z",
    "payload": "<img src=x onerror=alert(1)>",
    "status": "200",
    "user-agent": "Chrome/98.0"
  },
  {
    "ip_address": "192.168.1.3",
    "timestamp": "2024-12-26T12:00:20Z",
    "payload": "<img src=x onerror=alert(1)>",
    "status": "200",
    "user-agent": "Safari/537.36"
  }
]
```


Observations from the Log Data: Figure 2 illustrates the results of the first five tested payloads, where four responses contained unusual system behavior.

ID	PAYLOAD	Message
1	<sCrIpT>alert(1)</sCrIpT>	ALERT Text: 1 – Message: unexpected alert open: {Alert text : 1} (Session inf...)
2	<sCrIpT>alert(1)</sCrIpT>	ALERT Text: 1 – Message: unexpected alert open: {Alert text : 1} (Session inf...)
3	<script x>alert('XSS')</script y>	500 Internal Server Error
4	eval('ale'+'rt(0)');	500 Internal Server Error
5	setTimeout('ale'+'rt(2)');	500 Internal Server Error
6	setInterval('ale'+'rt(10)'');	500 Internal Server Error
7	Set_constructor('ale'+'rt(13)')()	500 Internal Server Error

Figure 2. *Results of Initial Payload Testing with Identified Suspicious Behaviors*

Figure 3 displays a table showing the analysis of suspicious behaviors, with differentiated IP addresses, the number of requests sent, and the suspiciousness score.

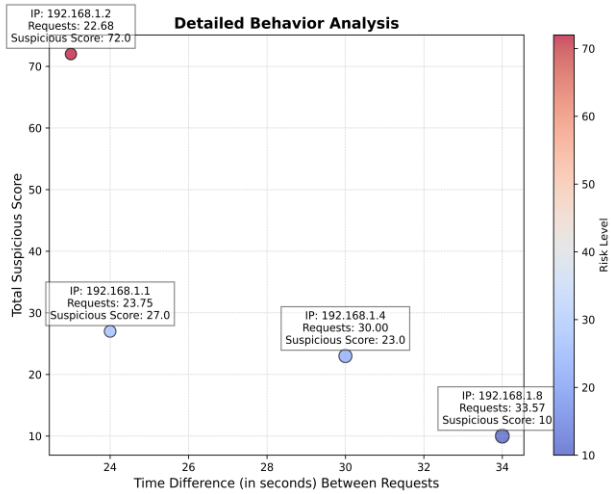

Figure 3. *Detailed Behavioral Analysis Table*

Two payloads triggered "alert(1)" pop-ups, indicating potential vulnerabilities that could be exploited. In contrast, two other payloads resulted in HTTP 500 Internal Server Errors, suggesting server-side issues, but these errors do not represent exploitable vulnerabilities. By correlating payload execution, response patterns, and user-agent metadata, this behavioral analysis helps identify high-risk attack attempts before they result in exploitation.

The X-axis represents the time difference (in seconds) between each request, while the Y-axis illustrates the suspiciousness score assigned to each IP. The table highlights patterns of suspicious activity, allowing for further examination of abnormal behavior based on request frequency and suspiciousness levels.

5.7. Discussion of Strengths and Limitations

The proposed tool offers several strengths that enhance XSS detection capabilities. By leveraging automated detection, it minimizes manual effort through Selenium-based dynamic execution, allowing for a more comprehensive analysis of web vulnerabilities. Additionally, behavioral analysis is integrated into the system, using log-based clustering to detect suspicious activity patterns. One of the key advantages is early threat identification, which enables the system to detect attack attempts before XSS execution occurs, improving proactive security measures. However, the tool also presents certain limitations and areas for future improvement. One drawback is the execution overhead, as dynamic analysis takes longer compared to static-only approaches. Another limitation is the detection scope, which is currently focused on XSS vulnerabilities. Future enhancements could expand detection capabilities to include SQL Injection (SQLi) and Cross-Site Request Forgery (CSRF) attacks. Despite these challenges, the summary of results demonstrates the tool's effectiveness in detecting XSS attempts by combining dynamic execution with log-based anomaly detection. The

behavioral log analysis provides valuable insights into attack patterns and high-risk IPs, ensuring that suspicious activities are flagged early, thereby enabling proactive threat mitigation.

6. Conclusion

This study presents an advanced XSS detection framework that integrates dynamic execution, behavioral analysis, and log-based anomaly detection to improve web security. The proposed tool demonstrates higher detection accuracy compared to traditional methods by leveraging automated payload injection, Selenium execution, and clustering-based analysis. The ability to identify DOM-based XSS vulnerabilities and flag suspicious activities before exploitation occurs highlights its effectiveness in real-world scenarios. While the system outperforms existing tools in detecting XSS attacks, it also has certain limitations, including increased execution time due to its dynamic approach. Future enhancements could focus on optimizing performance and expanding detection capabilities to other web-based attacks such as SQL Injection (SQLi) and Cross-Site Request Forgery (CSRF).

REFERENCES

[1] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of XSS sanitization in web application frameworks", in *Proc. 16th Eur. Symp. Res. Comput. Security (ESORICS)*, Leuven, Belgium, Sep. 12–14, 2011, pp. 150-171.

[2] M. Liu, B. Zhang, W. Chen, and X. Zhang, "A survey of exploitation and detection methods of XSS vulnerabilities", *IEEE Access*, vol. 7, pp. 182004–182016, 2019. https://doi.org/10.1109/ACCESS.2019.2958987

[3] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: Classification and State-of-the-Art", *Int. J. Syst. Assur. Eng. Manage.*, vol. 8, pp. 512–530, 2017. https://doi.org/10.1007/s13198-017-0605-6

[4] I. Yusof and A. S. K. Pathan, "Preventing persistent Cross-Site Scripting (XSS) attack by applying pattern filtering approach", in *Proc. 5th Int. Conf. Inf. Commun. Technol. Muslim World (ICT4M)*, Kuching, Malaysia, Nov. 2014, pp. 1–6.

[5] X. Tan, Y. Xu, T. Wu, and B. Li, "Detection of reflected XSS vulnerabilities based on paths-attention method", *Appl. Sci.*, vol. 13, no. 13, p. 7895, 2023. https://doi.org/10.3390/app13137895

[6] T. K. Nguyen, and S. O. Hwang, "Large-scale detection of DOM-based XSS based on publisher and subscriber model", in *Proc. 2016 Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Las Vegas, NV, USA, Dec. 2016, pp. 975-980. https://doi.org/10.1109/CSCI.2016.0137

[7] E. Blancaflor, E. E. Araullo, J. A. Corcuera, J. R. Rivera, and L. N. Velarde, "Vulnerability assessment on Cross-site scripting attack in a simulated E-commerce platform using BeEF and XSStrike", in *Proc. 2023 13th Int. Conf. Softw. Technol. Eng. (ICSTE)*, Xi'an, China, Oct. 2023, pp. 1–6.

[8] A. Kore, T. Hinduja, A. Sawant, S. Indorkar, S. Wagh, and S. Rankhambe, "Burp suite extension for script-based attacks for web applications", in *Proc. 2022 6th Int. Conf. Electron., Commun. Aerosp. Technol. (ICECA)*, Coimbatore, India, Dec. 2022, pp. 651–657. https://doi.org/10.1109/ ICECA55406.2022.9936071

[9] G. Parimala, M. Sangeetha, and R. AndalPriyadharsini, "Efficient web vulnerability detection tool for sleeping giant-cross site request forgery", *J. Phys. Conf. Ser.*, vol. 1000, no. 1, p. 012125, 2018. http://doi.org/10.1088/1742-6596/1000/1/012125

[10] F. J. Abdullayeva, "Distributed denial of service attack detection in E-government cloud via data clustering", *Array*, vol. 15, p. 100229, 2022. https://doi.org/10.1016/j.array.2022.100229