



**TRIBHUVAN UNIVERSITY  
INSTITUTE OF ENGINEERING  
PASCHIMANCHAL CAMPUS**

**A Mid-term Final Year Project Report**

**On**

**Red Sentinel: Transformer-Based Generation and Obfuscation of Context-Aware  
XSS Payloads**

**Submitted By:**

Bipin Bhattarai (WRC078BCT009)

Seamoon Pandey (WRC078BCT035)

Sworup Bhandari (WRC078BCT046)

**Submitted To:**

Department of Electronics and Computer Engineering

Paschimanchal Campus

Pokhara, Nepal

In partial fulfillment for the award of a Bachelor's degree in Computer Engineering

**Under the Supervision of**

Sudip Dahal

December, 2025

## ABSTRACT

Cross-Site Scripting (XSS) remains one of the most prevalent and persistent web application vulnerabilities, consistently ranking in the OWASP Top Ten. Traditional security testing approaches rely on signature-based detection and manually crafted payloads, which struggle to adapt to diverse injection contexts and modern filtering mechanisms. This project presents **Red Sentinel**, an intelligent, machine learning-driven system designed to automatically generate context-aware XSS payloads using transformer-based neural networks.

Red Sentinel addresses the limitations of conventional XSS scanners by leveraging deep learning to understand the syntactic and semantic properties of injection points across different web contexts—including HTML attributes, JavaScript blocks, event handlers, and URL parameters. The system employs a microservice architecture comprising three core modules: a Context Extraction Module that analyzes target web applications to identify injection points, a Payload Handler that generates and obfuscates attack payloads using an encoder-decoder transformer model, and an Orchestration Core that coordinates testing workflows.

The transformer-based payload generator is trained on curated datasets of context-labeled XSS payloads using byte-level tokenization to preserve special characters and Unicode variants essential for exploitation. This approach enables the model to produce syntactically valid, execution-ready payloads tailored to specific contexts. Additionally, an obfuscation module applies encoding-based, structural, and JavaScript-based transformations to evade modern web application firewalls and pattern-matching filters.

Initial experiments demonstrate that Red Sentinel can generate diverse, context-specific XSS payloads that surpass manually crafted examples in both variety and effectiveness. The system also includes evaluation mechanisms to assess payload success rates and filter evasion capabilities. By automating adversarial payload creation, Red Sentinel significantly improves the depth and coverage of web security testing, enabling security professionals and developers to identify vulnerabilities that traditional tools fail to detect.

This mid-term report presents the system architecture, dataset preparation methodology, transformer model design, and preliminary results from GRU, LSTM, and transformer-based payload generators. It also outlines challenges encountered, work completed to date, and a roadmap for future development including advanced obfuscation techniques, real-world vulnerability benchmarking, and integration with existing penetration testing

workflows.

**Keywords:** Cross-Site Scripting, XSS, Machine Learning, Transformer Models, Adversarial Payload Generation, Web Security, Automated Penetration Testing, Obfuscation, Context-Aware Attack Generation

## ACKNOWLEDGMENTS

We would like to express our sincere gratitude to all those who have contributed to the successful completion of this mid-term report and the ongoing development of the Red Sentinel project.

First and foremost, we extend our deepest appreciation to our project supervisor for their invaluable guidance, continuous support, and constructive feedback throughout this research. Their expertise in web security and machine learning has been instrumental in shaping the direction of this work.

We are grateful to the Department of Electronics and Computer Engineering at Paschimanchal Campus, Tribhuvan University, for providing the necessary resources, infrastructure, and academic environment that enabled us to pursue this research.

We would like to acknowledge the open-source community for providing essential tools, frameworks, and datasets that formed the foundation of our system. Special thanks to the developers of TensorFlow, PyTorch, FastAPI, and the OWASP community for their contributions to web security research.

We also thank our families and friends for their unwavering support, patience, and encouragement throughout this journey. Their understanding during the challenging phases of this project has been invaluable.

Finally, we acknowledge the researchers whose prior work in automated vulnerability detection, transformer-based code generation, and adversarial machine learning inspired and informed our approach. Their contributions to the field have paved the way for innovations like Red Sentinel.

Bipin Bhattarai  
Seamoon Pandey  
Sworup Bhandari

*December, 2025*

## TABLE OF CONTENTS

<b>Abstract</b> . . . . .	<b>i</b>
<b>Acknowledgments</b> . . . . .	<b>iii</b>
<b>List of Figures</b> . . . . .	<b>vii</b>
<b>List of Abbreviations</b> . . . . .	<b>viii</b>
<b>1. Introduction</b> . . . . .	<b>1</b>
1.1 Problem Statement . . . . .	1
1.2 Objectives . . . . .	2
1.3 Project Feasibility . . . . .	2
1.3.1 Technical Feasibility . . . . .	2
1.3.2 Operational Feasibility . . . . .	2
1.3.3 Economic Feasibility . . . . .	3
1.3.4 Sociocultural Feasibility . . . . .	3
1.3.5 Legal Feasibility . . . . .	3
<b>2. Literature Review</b> . . . . .	<b>4</b>
2.1 Core Module and Gateway Architecture . . . . .	4
2.1.1 Microservice Architectures for Security Tools . . . . .	4
2.1.2 Input Validation and Normalization . . . . .	4
2.1.3 Request Orchestration and Workflow Management . . . . .	5
2.2 Context Extraction and Analysis . . . . .	5
2.2.1 Context-Aware Fuzzing . . . . .	5
2.2.2 DOM and JavaScript Analysis . . . . .	6
2.2.3 Sanitization and Filter Detection . . . . .	6
2.2.4 Multi-Context Classification . . . . .	6
2.3 Payload Generation and Machine Learning Approaches . . . . .	6
2.3.1 Reinforcement Learning Approaches . . . . .	7
2.3.2 Deep Generative Models . . . . .	7
2.3.3 Neural Machine Translation . . . . .	8
2.3.4 Large Language Models and Obfuscation . . . . .	8
2.3.5 Byte-Level Tokenization and Sequence Modeling . . . . .	8
2.4 Research Gaps and Opportunities . . . . .	9
<b>3. Methodology</b> . . . . .	<b>10</b>
3.1 Architectural Design Overview . . . . .	10
3.1.1 Core Module . . . . .	11
3.1.1.1 Architectural Role and Responsibilities . . . . .	11

3.1.1.2	Step One: Target Input and Page Retrieval . . . . .	12
3.1.1.3	Technology Stack and Implementation . . . . .	15
3.1.2	Context-Aware Analysis Module . . . . .	15
3.1.2.1	Static Context Extraction . . . . .	15
3.1.2.2	Dynamic Context Extraction . . . . .	16
3.1.2.3	Rule-Based Classification . . . . .	16
3.1.2.4	ML-Based Attack and Payload Classification . . . . .	16
3.1.2.5	C-Script Construction . . . . .	17
3.1.2.6	Context Dataset Construction . . . . .	17
3.1.3	Payload Handler . . . . .	18
3.1.3.1	Payload Generator . . . . .	18
3.1.3.2	Obfuscation Model . . . . .	19
3.2	Dataset Preparation and Preprocessing . . . . .	21
3.2.1	Data Collection . . . . .	21
3.2.2	Context Labeling . . . . .	22
3.2.3	Preprocessing Pipeline . . . . .	22
3.3	Activity Diagram . . . . .	23
<b>4.</b>	<b>Work Completed and Progress Report . . . . .</b>	<b>24</b>
4.1	Work Completed . . . . .	24
4.1.1	Research . . . . .	24
4.1.2	Familiarization with Tools and Frameworks . . . . .	24
4.1.3	Dataset Preparation and Preprocessing . . . . .	24
4.1.4	Model Development and Analysis . . . . .	24
4.1.4.1	3-Layer GRU Decoder Model & 3-Layer LSTM De- coder Model . . . . .	25
4.1.4.2	Byte-Level Transformer Payload Generator . . . . .	25
4.1.4.3	T5-Based Large Language Model Fine-Tuning . . . . .	26
4.2	Challenges Encountered . . . . .	27
4.3	Work in Progress . . . . .	28
4.3.1	Remaining Tasks . . . . .	28
4.4	Project Scheduling . . . . .	29
<b>5.</b>	<b>Conclusion and Future Work . . . . .</b>	<b>30</b>
5.1	Summary . . . . .	30
5.2	Contributions . . . . .	30
5.3	Limitations . . . . .	31
5.4	Future Work . . . . .	32
5.4.1	Technical Enhancements . . . . .	32
5.4.2	System Integration . . . . .	32

5.4.3	Evaluation and Validation . . . . .	32
5.4.4	Research Extensions . . . . .	33
5.5	Final Remarks . . . . .	33
	<b>Bibliography . . . . .</b>	<b>35</b>

## List of Figures

Figure 3.1.1 System Architecture Diagram . . . . .	11
Figure 3.1.2 Transformer Model Structure . . . . .	19
Figure 3.1.3 Payload Handler Flow Structure . . . . .	21
Figure 3.3.1 System Activity Diagram . . . . .	23
Figure 4.4.1 Gantt Chart Showing Work Schedule . . . . .	29



## **LIST OF ABBREVIATIONS**

**AI** Artificial Intelligence

**API** Application Programming Interface

**AWS** Amazon Web Services

**CI/CD** Continuous Integration/Continuous Deployment

**CVE** Common Vulnerabilities and Exposures

**DDQN** Double Deep Q-Network

**DOM** Document Object Model

**DQN** Deep Q-Network

**DVWA** Damn Vulnerable Web Application

**GRU** Gated Recurrent Unit

**HTML** HyperText Markup Language

**LSTM** Long Short-Term Memory

**ML** Machine Learning

**mTLS** Mutual Transport Layer Security

**NLP** Natural Language Processing

**OWASP** Open Web Application Security Project

**RL** Reinforcement Learning

**RNN** Recurrent Neural Network

**SVG** Scalable Vector Graphics

**WAF** Web Application Firewall

**XSS** Cross-Site Scripting

## **1. INTRODUCTION**

Web applications today play a central role in handling sensitive information for individuals and organizations. Despite advances in secure software engineering, these systems continue to face long-standing vulnerabilities. Cross-Site Scripting (XSS) remains one of the most persistent and widely exploited threats and consistently appears in the OWASP Top Ten rankings. XSS attacks occur when adversaries inject malicious scripting code into web content that is later delivered to end users, potentially enabling credential theft, unauthorized session access, malware distribution, or manipulation of displayed data.

Modern web pages integrate multiple languages, including HTML, CSS, JavaScript, and SVG, within the same document structure. Because each injection point follows different syntactic and execution rules, attackers must tailor payloads to the specific context in which the code will run. Research has shown that a payload effective inside a script block may fail entirely when placed within an HTML attribute. This context sensitivity is a major reason why XSS detection and prevention remain difficult even after decades of research.

### **1.1 Problem Statement**

Although XSS scanners and Web Application Firewalls (WAFs) can identify many basic vulnerabilities, they lack the adaptive intelligence required to generate valid and context-aware payloads. Their reliance on predefined signatures limits their ability to detect new, obfuscated, or context-specific attack vectors. There is a clear need for a system that can automatically generate intelligent, adaptable, and context-sensitive XSS payloads using modern machine learning techniques, thereby improving the depth and reliability of security testing.

These challenges have motivated the use of artificial intelligence and machine learning to support offensive security research. Recent work demonstrates that generative models, especially transformer-based architectures, are capable of producing diverse and novel XSS payloads that surpass human-crafted examples. Systems such as HAXSS and fine-tuned language models like GPT-2 and CodeT5 show promising results in automating adversarial payload creation, thereby increasing coverage and uncovering weaknesses that rule-based tools fail to detect.

## **1.2 Objectives**

The primary objectives of this project are:

1. To develop a machine learning-based system capable of generating context-aware XSS payloads that adapt to different injection contexts including HTML attributes, JavaScript blocks, event handlers, and URL parameters.
2. To integrate payload generation with a modular testing framework that supports automated vulnerability discovery, obfuscation techniques, and comprehensive reporting.
3. To evaluate the effectiveness of transformer-based models compared to traditional recurrent architectures in generating syntactically valid and execution-ready XSS payloads.
4. To implement an obfuscation module that can evade modern web application firewalls and signature-based detection systems.

## **1.3 Project Feasibility**

The Red Sentinel project is feasible due to the maturity of ML frameworks, availability of datasets, and growing interest in automated offensive security methods.

### **1.3.1 Technical Feasibility**

1. Modern ML frameworks such as TensorFlow and PyTorch support transformer architectures and byte-level tokenizers, making implementation highly achievable.
2. Hardware requirements for training a medium-sized transformer model are modest and achievable using consumer GPUs or cloud resources.
3. Existing open-source XSS payload datasets and academic research provide sufficient training data for model development.
4. Microservice architecture patterns are well-established, enabling modular development and deployment of system components.

### **1.3.2 Operational Feasibility**

1. Security teams can incorporate Red Sentinel into existing testing workflows without major changes to their current processes.
2. The modular design supports iterative development and ease of maintenance, al-

lowing individual components to be updated independently.

3. The system reduces manual effort by automating payload creation, improving operational efficiency for developers and penetration testers.
4. Integration with standard penetration testing tools (such as Burp Suite and OWASP ZAP) is straightforward through API interfaces.

### **1.3.3 Economic Feasibility**

1. Costs are limited primarily to training infrastructure and optional dataset acquisition, which can be managed through cloud-based pay-per-use models.
2. Open-source ML libraries and penetration-testing tools reduce development expenses significantly.
3. Cloud-based compute resources can be used only when necessary (for training and large-scale testing), minimizing ongoing operational costs.
4. The return on investment is substantial, as automated vulnerability discovery reduces the time and cost of manual security assessments.

### **1.3.4 Sociocultural Feasibility**

1. The project aligns with growing societal emphasis on cybersecurity and ethical hacking as critical components of digital infrastructure protection.
2. Automated vulnerability assessment democratizes security testing, allowing organizations with limited security expertise to identify critical flaws.
3. The project contributes to the broader goal of improving web application security standards across industries.
4. Educational institutions can use Red Sentinel as a teaching tool for demonstrating real-world application of machine learning in cybersecurity.

### **1.3.5 Legal Feasibility**

1. The system must be used strictly for authorized security testing to comply with cybercrime and computer misuse laws.
2. No proprietary or personal data is required for model training, reducing legal risks associated with data privacy regulations.
3. The project includes clear documentation on ethical use and responsible disclosure practices.
4. Deployment can be restricted to controlled environments with appropriate access controls to prevent misuse.

## **2. LITERATURE REVIEW**

The following review examines significant advancements in automated web security testing, with a particular focus on XSS payload generation and adversarial input synthesis. Existing research covers a wide spectrum of techniques, including reinforcement learning, neural machine translation, generative adversarial models, and transformer-based language approaches. By analyzing these contributions, this review highlights the methodological trends, strengths, and limitations that shape the design and development of the Red Sentinel system.

### **2.1 Core Module and Gateway Architecture**

The core module serves as the orchestration layer and entry point for security testing workflows. Research in this area focuses on architectural patterns, API gateway design, and microservice coordination for security tools.

#### **2.1.1 Microservice Architectures for Security Tools**

Modern security testing frameworks increasingly adopt microservice architectures to achieve modularity, scalability, and independent deployment of components. This approach allows different security testing modules to be developed, tested, and deployed independently while maintaining a cohesive system through well-defined APIs.

The core gateway acts as the single entry point for all requests, providing unified authentication, rate limiting, request routing, and response aggregation. This pattern ensures that security policies are consistently applied across all modules and simplifies the integration of new testing components.

#### **2.1.2 Input Validation and Normalization**

Robust input validation is critical for security testing tools to prevent injection attacks, ensure data integrity, and provide meaningful error messages. Research shows that proper URL normalization, protocol validation, and parameter sanitization at the gateway level significantly reduce downstream processing errors and security vulnerabilities.

Contemporary gateway implementations employ multi-layer validation strategies, in-

cluding schema validation, type checking, range validation, and business logic validation. These layers work together to ensure that only well-formed, safe inputs propagate through the system.

### **2.1.3 Request Orchestration and Workflow Management**

The core gateway coordinates complex multi-step security testing workflows by managing state, tracking progress, and ensuring proper sequencing of operations. Workflow orchestration patterns such as saga, choreography, and event-driven architectures enable resilient, fault-tolerant testing pipelines.

Research demonstrates that implementing proper circuit breakers, timeout management, and retry mechanisms at the gateway level significantly improves system reliability and prevents cascading failures when downstream services experience issues.

## **2.2 Context Extraction and Analysis**

Context extraction is fundamental to generating effective, context-aware XSS payloads. This section reviews research on identifying injection points, classifying contexts, and extracting metadata necessary for intelligent payload generation.

### **2.2.1 Context-Aware Fuzzing**

Pala et al. (2023) examined contemporary XSS scanners and highlighted XSSStrike as a tool that combines intelligent fuzzing with basic machine-learning techniques [1]. XSSStrike analyzes the structure of a web page to detect injection points and then mutates payloads based on contextual feedback. Their evaluation showed that this approach improves detection and execution of XSS payloads across different contexts, illustrating the value of context-aware fuzzing.

Fink (2018) introduced FOXSS, a scanner that integrates static data-flow analysis with targeted, context-sensitive fuzzing [2]. The tool identifies potential input flows and generates tailored payloads for each sink, verifying results in a real browser. FOXSS demonstrated very high detection accuracy, outperforming conventional scanners and significantly reducing false positives, indicating that program analysis combined with adaptive fuzzing can greatly enhance XSS detection.

### **2.2.2 DOM and JavaScript Analysis**

Modern web applications heavily rely on client-side JavaScript and dynamic DOM manipulation, creating complex attack surfaces that traditional static analysis tools struggle to identify. Research in this area focuses on dynamic analysis techniques, JavaScript instrumentation, and runtime monitoring to detect injection points within script contexts.

Advanced context extraction systems employ Abstract Syntax Tree (AST) analysis to understand code structure, identify variable assignments, and track data flow through JavaScript execution. This enables precise identification of contexts where user input is processed, allowing for more targeted payload generation.

### **2.2.3 Sanitization and Filter Detection**

Understanding the sanitization mechanisms employed by target applications is crucial for generating payloads that can evade filters. Research demonstrates that probing techniques, differential testing, and machine learning can effectively fingerprint sanitization libraries and identify bypass opportunities.

Context extraction modules must not only identify where user input appears in the output but also understand what transformations are applied to that input. This metadata enables payload generators to craft inputs that survive sanitization while maintaining their malicious intent.

### **2.2.4 Multi-Context Classification**

Web applications often reflect user input in multiple contexts simultaneously (e.g., both HTML attribute and JavaScript string). Effective context extraction requires classifying all contexts where input appears and understanding their interaction. Research shows that multi-label classification approaches combined with syntax-aware parsing significantly improve context identification accuracy.

## **2.3 Payload Generation and Machine Learning Approaches**

This section examines research on automated payload generation, focusing on reinforcement learning, deep generative models, neural machine translation, and transformer-based architectures.

### **2.3.1 Reinforcement Learning Approaches**

Foley and Maffeis (2022) propose HAXSS, a hierarchical reinforcement-learning framework to automatically generate XSS attack payloads [3]. Their approach frames the payload creation as two nested RL "games." The first agent learns to escape the immediate HTML context of user output (escaping tags or attributes), while the second agent intervenes whenever the application attempts sanitization—it learns to obfuscate the payload to bypass filters. Successful obfuscations are fed back into the first agent for further refinement.

Implemented as an end-to-end black-box fuzzer, HAXSS was evaluated on both synthetic benchmarks and real web applications. It identified 131 XSS vulnerabilities (20% more than state-of-the-art scanners) with zero false positives, including rediscovering known CVEs and uncovering 5 novel CVEs in production-grade sites. Thus, HAXSS demonstrates that hierarchical RL can produce diverse, context- and filter-aware payloads that significantly improve XSS discovery over traditional scanners.

Song et al. (2023) presented a grey-box fuzzing system that uses reinforcement learning to create and refine XSS payloads [4]. After mapping all input points through static analysis, the authors trained RL agents (DQN, DDQN, and Policy Gradient) to adjust payloads based on execution feedback. The RL-based fuzzer identified all known XSS vulnerabilities in benchmark applications with no false positives, demonstrating that RL can effectively adapt payloads to complex contexts.

### **2.3.2 Deep Generative Models**

Khan (2024) presents LL-XSS, an end-to-end deep generative model for crafting XSS payloads [5]. Unlike hand-crafted or brute-fuzzed payloads, LL-XSS leverages a combination of auto-regressive neural networks and transformer architectures to analyze both frontend and backend web code and produce candidate attack scripts. The model is trained on example web application code to learn common injection patterns, then generates new malicious scripts aimed at identified vulnerabilities.

In evaluation on the OWASP Juice Shop, the author reports that LL-XSS can automatically generate syntactically valid and exploitable XSS payloads by understanding the target's code context. This approach effectively automates the penetration-testing process: LL-XSS bypasses known filters and exposes vulnerabilities with minimal human guidance, demonstrating the promise of AI-driven payload creation for web security.



### **2.3.3 Neural Machine Translation**

Frempong et al. (2021) developed HIJaX, a neural machine translation model that converts natural-language attack descriptions into working XSS payloads [6]. The system was trained on paired examples of intent phrases and JavaScript exploits. Their experiments showed that the model could reliably generate valid payloads that triggered real XSS vulnerabilities, demonstrating that neural translation methods can automate exploit creation even for users with limited technical expertise.

This work is particularly significant because it bridges the gap between high-level security objectives and low-level technical implementation, making offensive security testing more accessible to non-experts while maintaining technical rigor.

### **2.3.4 Large Language Models and Obfuscation**

Miczek et al. (2025) explored the use of large language models to generate obfuscated XSS attacks capable of evading machine-learning detectors [7]. Their study showed that classifiers trained on standard payloads performed poorly when encountering obfuscated variants. By fine-tuning a transformer to generate diverse obfuscations, the authors significantly improved detector robustness after retraining. This work highlights the usefulness of LLM-generated adversarial examples for strengthening defensive models.

The dual nature of this research—both offensive (generating evasive payloads) and defensive (improving detection)—demonstrates the critical role of adversarial machine learning in advancing web security.

### **2.3.5 Byte-Level Tokenization and Sequence Modeling**

Traditional language models operate at word or subword levels, which can lose critical character-level information necessary for XSS payload generation. Recent research demonstrates that byte-level tokenization preserves all characters, including special symbols, Unicode variants, and escape sequences essential for exploit effectiveness.

Transformer-based sequence-to-sequence models with byte-level tokenization have shown superior performance in generating syntactically valid, context-appropriate payloads compared to character-level or word-level approaches. The attention mechanism enables these models to learn long-range dependencies between context metadata and exploit strategies.

## 2.4 Research Gaps and Opportunities

While existing research demonstrates the potential of machine learning in automated XSS detection and payload generation, several gaps remain:

1. **Limited Context Awareness:** Most existing systems treat contexts as discrete categories rather than continuous semantic spaces, limiting their ability to handle novel or hybrid injection points.
2. **Obfuscation Integration:** Few systems integrate obfuscation as a first-class component of the payload generation pipeline, relying instead on post-processing transformations.
3. **Evaluation Frameworks:** Standardized benchmarks for comparing payload generators across different contexts and filtering mechanisms are lacking.
4. **Byte-Level Precision:** Most language models operate at word or subword levels, potentially missing character-level patterns critical for XSS exploitation.

Red Sentinel addresses these gaps by implementing byte-level tokenization, integrated obfuscation, modular architecture, and comprehensive context modeling through transformer-based sequence-to-sequence learning.

### **3. METHODOLOGY**

#### **3.1 Architectural Design Overview**

Red Sentinel is implemented using a microservice architecture. In this design, each major functional component of the system runs as an independent service. These services communicate via well-defined APIs and are coordinated by a central orchestration core—the "Core Module." This microservice approach ensures that each module is isolated, independently deployable, testable, and can be scaled or replaced without impacting the rest of the system—addressing concerns of maintainability, modularity, and system complexity typical in security tools.

To connect the ML-based payload generator with the rest of the system and other modules, a data pipeline architecture is used. As described in the literature on integrating ML pipelines with microservices, this setup supports data ingestion, preprocessing, model serving, asynchronous communication, and modular isolation of ML components from other services.

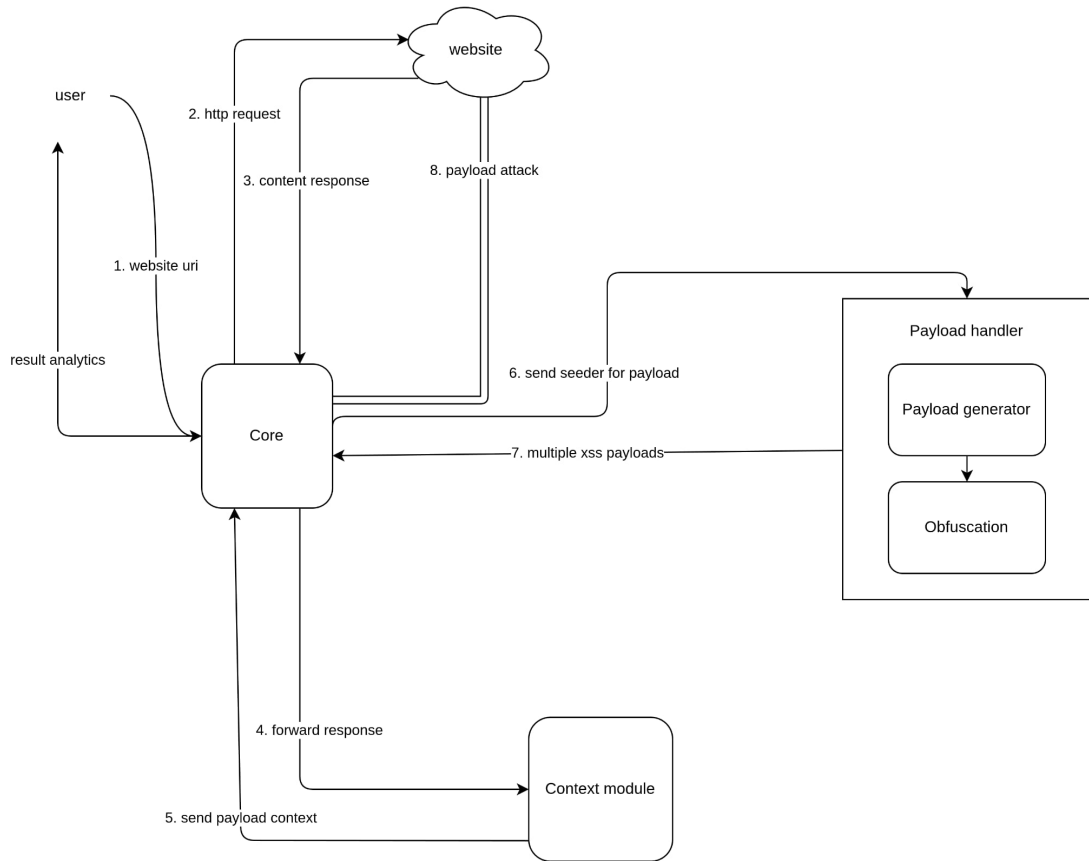


Figure 3.1.1: System Architecture Diagram

### 3.1.1 Core Module

The Core Module serves as the orchestration layer and primary entry point for Red Sentinel. Implemented using NestJS, a progressive Node.js framework, it provides the foundational infrastructure for coordinating distributed microservices, managing workflow execution, and ensuring system reliability. The module operates as a RESTful API gateway that bridges external security researchers and internal ML-powered services.

#### 3.1.1.1 Architectural Role and Responsibilities

The Core Module fulfills multiple critical roles within the Red Sentinel ecosystem:

- **API Gateway and Request Routing:** Exposes RESTful endpoints that accept target URLs and scanning parameters from clients. Routes requests to appropriate downstream services based on workflow stage and service availability.
- **Input Validation and Security Enforcement:** Implements comprehensive vali-

validation logic to ensure all user-provided inputs conform to security policies. This includes URL structure validation, protocol whitelisting (HTTP/HTTPS only), and Server-Side Request Forgery (SSRF) prevention through private IP address blocking.

- **Workflow Orchestration:** Manages the multi-stage pipeline from initial target submission through context extraction, payload generation, obfuscation, and result aggregation. Coordinates asynchronous communication between services using message queues or direct HTTP invocation.
- **State Management:** Maintains scanning session state, tracks request progress, and persists intermediate results for fault tolerance and debugging. Implements distributed caching to reduce redundant operations.
- **Error Handling and Recovery:** Provides comprehensive error handling with typed exception filters, automatic retry mechanisms for transient failures, and graceful degradation when dependent services are unavailable.
- **Logging and Observability:** Generates structured logs with unique request identifiers for distributed tracing. Exposes metrics endpoints compatible with Prometheus for monitoring request rates, latency distributions, and error frequencies.

### 3.1.1.2 Step One: Target Input and Page Retrieval

The first operational step within the Core Module focuses on accepting target URLs, performing security validation, and retrieving raw HTML content. This foundational layer establishes the data pipeline upon which all subsequent analysis depends.

**Input Validation** When a client submits a target URL via the POST `/api/v1/gateway/fetch` endpoint, the system performs rigorous validation:

1. **URL Structure Validation:** Parses the URL to ensure it contains all required components (protocol, hostname) and conforms to RFC 3986 standards. Rejects malformed URLs with descriptive error messages.
2. **Protocol Enforcement:** Accepts only HTTP and HTTPS protocols. Rejects potentially dangerous schemes such as `file://`, `ftp://`, `javascript:`, and `data:` to prevent local file access and code injection attacks.
3. **Length Constraints:** Enforces a maximum URL length of 2048 characters to prevent buffer overflow vulnerabilities and resource exhaustion.
4. **DNS Resolution:** Resolves the target hostname to its IP address using standard DNS lookup mechanisms before initiating HTTP requests.

**SSRF Prevention** To prevent Server-Side Request Forgery attacks, the Core Module implements IP address filtering that blocks requests to private network ranges:

- **IPv4 Private Ranges:** 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 (RFC 1918)
- **Loopback Addresses:** 127.0.0.0/8 (IPv4), ::1/128 (IPv6)
- **Link-Local Addresses:** 169.254.0.0/16 (IPv4), fe80::/10 (IPv6)
- **Reserved Ranges:** Multicast, broadcast, and experimental address spaces

Requests targeting these ranges are rejected with HTTP 403 Forbidden status, preventing attackers from scanning internal infrastructure or accessing cloud metadata endpoints.

**HTTP Request Execution** Upon successful validation, the Core Module configures an HTTP client with security-hardened settings:

- **Timeout Configuration:** Enforces a default timeout of 30 seconds to prevent indefinite hanging. Configurable through request parameters within the range of 1-60 seconds.
- **Redirect Handling:** Follows HTTP redirects automatically up to a maximum of 5 redirects. Prevents redirect loops and excessive chain following that could indicate malicious targets.
- **TLS/SSL Validation:** Validates SSL certificates for HTTPS requests, rejecting self-signed certificates in production mode to ensure authenticity.
- **User-Agent Identification:** Sets a custom User-Agent header (RedSentinel/1.0 (Security Scanner)) for ethical disclosure, allowing site administrators to identify and control scanner access.
- **Content Size Limits:** Enforces a maximum response size of 10MB to prevent memory exhaustion attacks. Aborts downloads that exceed this threshold.

**Response Processing** After successfully retrieving the target page, the Core Module processes the HTTP response:

1. **Status Code Capture:** Records the final HTTP status code (e.g., 200, 301, 404) for diagnostic purposes.
2. **Header Extraction:** Captures relevant response headers including Content-Type, Content-Length, Server, and security headers (X-XSS-Protection, Content-Security-Po

3. **Content Validation:** Verifies the Content-Type header indicates HTML content (text/html). Non-HTML responses are logged but not rejected, as some applications serve HTML with incorrect MIME types.
4. **HTML Body Extraction:** Reads the complete response body as text, preserving all whitespace, encoding, and formatting for accurate downstream analysis.

**Response Standardization** The Core Module returns a standardized JSON response structure for both successful and failed operations:

```
{
  "success": true,
  "data": {
    "targetUrl": "https://example.com/search",
    "finalUrl": "https://www.example.com/search",
    "statusCode": 200,
    "contentType": "text/html; charset=UTF-8",
    "contentLength": 45678,
    "html": "<!DOCTYPE html><html>...</html>",
    "redirectCount": 1,
    "fetchDuration": 1234
  },
  "metadata": {
    "timestamp": "2025-12-10T14:32:15.789Z",
    "requestId": "req_7f8a9b1c",
    "version": "1.0.0"
  }
}
```

This consistent response format simplifies integration with downstream services and enables automated result processing.

**Error Handling and Logging** The Core Module implements comprehensive error handling with specific error codes for common failure scenarios:

- `INVALID_URL_FORMAT`: Malformed URL structure
- `UNSUPPORTED_PROTOCOL`: Non-HTTP/HTTPS protocol detected
- `PRIVATE_IP_BLOCKED`: SSRF prevention triggered

- **FETCH\_TIMEOUT:** Request exceeded timeout duration
- **CONTENT\_TOO\_LARGE:** Response size exceeded 10MB limit
- **TOO\_MANY\_REDIRECTS:** Redirect count exceeded maximum

All requests are logged with structured JSON formatting, including request IDs for distributed tracing, target URLs (sanitized to remove credentials), execution duration, and error details. Logs are categorized by severity (INFO, WARN, ERROR) and exported to centralized logging infrastructure for monitoring and auditing.

### 3.1.1.3 Technology Stack and Implementation

The Core Module leverages modern TypeScript-based technologies for reliability and maintainability:

- **Framework:** NestJS 10.x with TypeScript 5.x for strong typing and compile-time safety
- **HTTP Client:** Axios with configurable timeout, redirect, and proxy support
- **Validation:** class-validator and class-transformer for declarative DTO validation
- **Logging:** Winston with JSON formatting for structured logging
- **Testing:** Jest for unit tests, Supertest for integration tests
- **Documentation:** Swagger/OpenAPI for automated API documentation

The modular architecture enables independent testing, deployment, and scaling of the Core Module, supporting Red Sentinel’s microservice design philosophy.

## 3.1.2 Context-Aware Analysis Module

The Context-Aware Module is responsible for examining how user-controlled input appears inside a webpage and determining whether that location is exploitable. Red Sentinel performs this using a hybrid approach that combines static HTML parsing, dynamic browser execution tracing, rule-based reasoning, and machine learning models. The output of this module is a normalized and labeled structure, known as a C-Script, which the payload generator later uses to craft attack strings.

### 3.1.2.1 Static Context Extraction

This stage performs offline HTML parsing using **BeautifulSoup4 (BS4)** and custom parsers. The system inspects the DOM to identify injection surfaces such as attributes, text nodes, script bodies, URL parameters, and form fields.



- Extracts tag names, attributes, raw values, depth, parent, and sibling structure.
- Detects static flags (HTML-entity usage, URL encoding, JavaScript escaping).
- Produces a structured static context object for every potential injection point.

### 3.1.2.2 Dynamic Context Extraction

Static content alone fails to capture runtime behavior. Red Sentinel launches a headless browser using **Playwright** and records dynamic modifications triggered by JavaScript execution.

- Detects `innerHTML` / `outerHTML` sinks.
- Logs event listeners (`onclick`, `onload`, custom JS handlers).
- Captures nodes created or modified at runtime.
- Records whether the context becomes dynamic and vulnerable after execution.

The static and dynamic information is then merged into a unified structure.

### 3.1.2.3 Rule-Based Classification

A custom rule engine evaluates structural and behavioral patterns to infer an initial attack class.

- Attributes starting with `on*` are labeled as event-handler contexts.
- Any element inserted through `innerHTML` is marked as DOM-XSS.
- URL-based attributes may produce JavaScript protocol execution.

These deterministic rules form the foundation for supervised ML labels and validation.

### 3.1.2.4 ML-Based Attack and Payload Classification

To enhance generalization, Red Sentinel trains two **Random Forest** classifiers using the unified context features:

1. Attack Class Model (e.g., event-handler, DOM-based, attribute-breakout)
2. Payload Type Model (e.g., js-exec, event-transition, html-breakout)

Contexts are encoded using one-hot vectors for tags and attributes, together with dynamic boolean features. The ML output refines the rule-based inference and improves robustness across diverse HTML structures.

### 3.1.2.5 C-Script Construction

The final stage produces a C-Script, a compact representation that describes the vulnerability surface.

```
{
  "attackClass": "dynamic-event",
  "payloadType": "js-exec",
  "riskScore": 0.92,
  "sink": "innerHTML",
  "allowedChars": "<>'\"/^\`",
  "isDynamic": true
}
```

This unified structure guides the Payload Generator by specifying contextual restrictions, browser behavior, and exploitability.

### 3.1.2.6 Context Dataset Construction

The module automatically generates training data by storing each context and its inferred labels in **JSONL** format. This dataset combines:

- BS4 static extraction output
- Playwright dynamic logs
- Rule-engine classifications
- Cleaned ML-ready feature vectors

The result is a scalable dataset used to train and evaluate the Random Forest classifiers.

**Summary** The Context-Aware Module forms the analytical backbone of Red Sentinel. By combining static DOM parsing, dynamic browser execution tracing, rule-based inference, and Random Forest classification, it produces a precise understanding of every injection point. This information is distilled into the C-Script structure, enabling the payload generator to craft highly accurate, context-specific XSS payloads. The module's hybrid design ensures reliability, extensibility, and compatibility with ML-driven offensive security workflows.

### 3.1.3 Payload Handler

The Payload Handler consists of two primary models working in tandem: the Payload Generator and the Obfuscation Model.

#### 3.1.3.1 Payload Generator

The Payload Generator is responsible for producing syntactically valid, context-adapted, and execution-ready XSS attack strings. Red Sentinel employs a transformer-based encoder-decoder architecture trained specifically for cross-site scripting contexts and adversarial behavior.

The transformer model is trained on a dataset formed by pairs of sanitized script structures provided by the Core Module and the corresponding XSS payloads. The generator is built on an encoder-decoder transformer network, chosen because of its exceptional ability to model long-range dependencies and handle complex code sequences. The model comprises:

- **Byte-Level Tokenizer**

Instead of word- or character-level tokenization, Red Sentinel uses a byte-level tokenizer (similar to Byte-Pair Encoding) to ensure that all characters—including `<`, `>`, `'`, `"`, `(`, `)`, `\`, and Unicode variants—are preserved without normalization. This is crucial because minor character transformations can change the exploitability of an XSS payload.

- **Encoder**

The encoder receives the attack type, payload type, and contextual information extracted by the Context Module. The encoder produces a contextual embedding that expresses the syntactic and semantic constraints of the injection point.

- **Decoder**

The decoder autoregressively generates an attack string conditioned on the encoder output. It learns patterns for escaping contexts, invoking JavaScript execution, nesting HTML/JS structures, and exploiting browser quirks. It is designed to avoid generating malformed payloads by learning valid grammar structures from curated training data.

- **Attention Mechanisms**

Attention heads enable the model to correlate specific positions in the context (e.g., inside quotes, inside script tags) with correct exploit strategies. This facilitates generation of highly specific payloads such as:

- Attribute breakouts: `"><svg/onload=alert(1)>`

- Script-block injections: `' ;alert(1);//`
- URL-based injections: `javascript:alert(1)`

Layer (type)	Output Shape	Param #	Connected to
input_layer_20 (InputLayer)	(None, None)	0	-
input_layer_21 (InputLayer)	(None, None)	0	-
encoder_padding_ma... (EncoderPaddingMas...	(None, 1, 1, None)	0	input_layer_20[0...
transformer_encode... (TransformerEncode...	(None, None, 256)	3,290,112	input_layer_20[0... encoder_padding_...
decoder_padding_ma... (DecoderPaddingMas...	(None, 1, 1, None)	0	input_layer_20[0...
decoder_self_mask_2 (DecoderSelfMask)	(None, 1, None, None)	0	input_layer_21[0...
transformer_decode... (TransformerDecode...	(None, None, 256)	4,344,832	input_layer_21[0... transformer_enco... decoder_padding_... decoder_self_mas...
final_output_proje... (Dense)	(None, None, 512)	131,584	transformer_deco...

Total params: 7,766,528 (29.63 MB)  
Trainable params: 7,766,528 (29.63 MB)  
Non-trainable params: 0 (0.00 B)

Figure 3.1.2: Transformer Model Structure

### 3.1.3.2 Obfuscation Model

Modern defensive systems often rely on pattern matching or signature detection, which is particularly vulnerable to obfuscation. Therefore, incorporating systematic obfuscation increases the likelihood of discovering hidden vulnerabilities and evaluating the true resilience of a target.

**Encoding-Based Obfuscation** This class manipulates the representation of characters while preserving their semantics at runtime:

- URL encoding (percent-encoding): `%3Cscript%3E`

- HTML entity encoding: `&#x3C;script&#x3E;`
- Unicode homoglyphs: Alternative representations of common symbols
- Base64 wrapper techniques: e.g., `eval(atob("YWxlcnQoMSk="))`

These methods exploit weaknesses in sanitization routines that decode values inconsistently.

**Structural Obfuscation** Structural modification alters the payload's syntax without changing its effect:

- String splitting: `a = "al" + "ert"; window[a](1)`
- Wrapped event handlers: `<img src=x onerror=%61%6c%65%72%74(1)>`
- Nonstandard tag nesting
- Junk insertion (harmless characters or comments)

This category is particularly effective against filter engines that scan for simple signatures (e.g., `alert`).

**JavaScript-Based Dynamic Obfuscation** A more advanced transformation relies on runtime reconstruction:

- Using constructor functions: `Function("al"+"ert(1)")()`
- Indirect invocation using event stacks
- Using proxies or dynamically generated DOM nodes to execute embedded code

These methods exploit JavaScript's dynamic nature to evade static analysis.

**Obfuscation Module Overview** The Obfuscation Module integrates into the system workflow as follows:

1. The Payload Generator produces a set of base payloads
2. Payloads are sent to the Obfuscation Service when enabled
3. The transformed payloads are injected into the target applications
4. All variants and their execution results are logged

## XSS Payload Generation and Obfuscation Flow

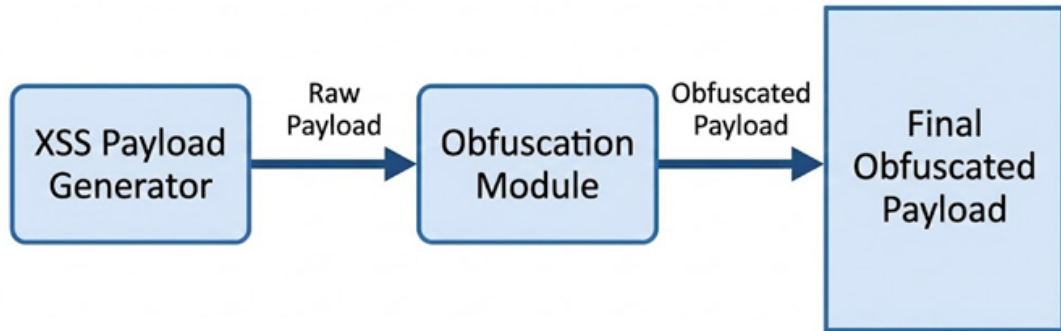


Figure 3.1.3: Payload Handler Flow Structure

Together, the Payload Generator and Obfuscation Module form the intelligent core of Red Sentinel. The generator creates contextually accurate, execution-ready XSS payloads, while the obfuscation system transforms them to evade filters and explore deeper vulnerabilities. Their microservice design ensures modularity, scalability, and extensibility, supporting Red Sentinel’s mission of providing advanced, ML-driven offensive security capabilities.

### 3.2 Dataset Preparation and Preprocessing

A high-quality dataset is essential for training the transformer-based payload generator. The dataset construction process involved several stages:

#### 3.2.1 Data Collection

Payload data was collected from multiple sources:

1. **Open-Source Repositories:** XSS payload collections from GitHub repositories such as PayloadsAllTheThings, XSS Hunter, and OWASP’s XSS Filter Evasion Cheat Sheet
2. **Academic Publications:** Payloads extracted from research papers on XSS vulnerability detection
3. **Vulnerability Databases:** Real-world exploit examples from CVE databases and security advisories
4. **Synthetic Generation:** Rule-based generation of context-specific payloads for

underrepresented categories

### 3.2.2 Context Labeling

Each payload was manually or semi-automatically labeled with:

- **Attack Class:** Reflected, Stored, DOM-based
- **Payload Type:** Event handler, script injection, attribute breakout, etc.
- **Context Type:** HTML attribute, JavaScript string, URL parameter, etc.
- **Risk Score:** Estimated severity and exploitability (0.0 to 1.0)

### 3.2.3 Preprocessing Pipeline

The preprocessing pipeline included:

1. **Deduplication:** Removal of exact and near-duplicate payloads using hash-based and fuzzy matching
2. **Normalization:** Standardization of whitespace, encoding formats, and character representations
3. **Validation:** Syntactic validation to ensure payloads are well-formed
4. **Tokenization:** Application of byte-level SentencePiece tokenization with byte fallback
5. **Sequence Formatting:** Construction of input-output pairs with special tokens for context metadata

Example formatted training instance:

```
Input: <ATTACK_CLASS>event-handler</ATTACK_CLASS>
      <PAYLOAD_TYPE>event-mouse</PAYLOAD_TYPE>
      <RISK>0.97</RISK>
```

```
Output: "><img src=1 onerror=prompt(document.domain)>
```

### 3.3 Activity Diagram

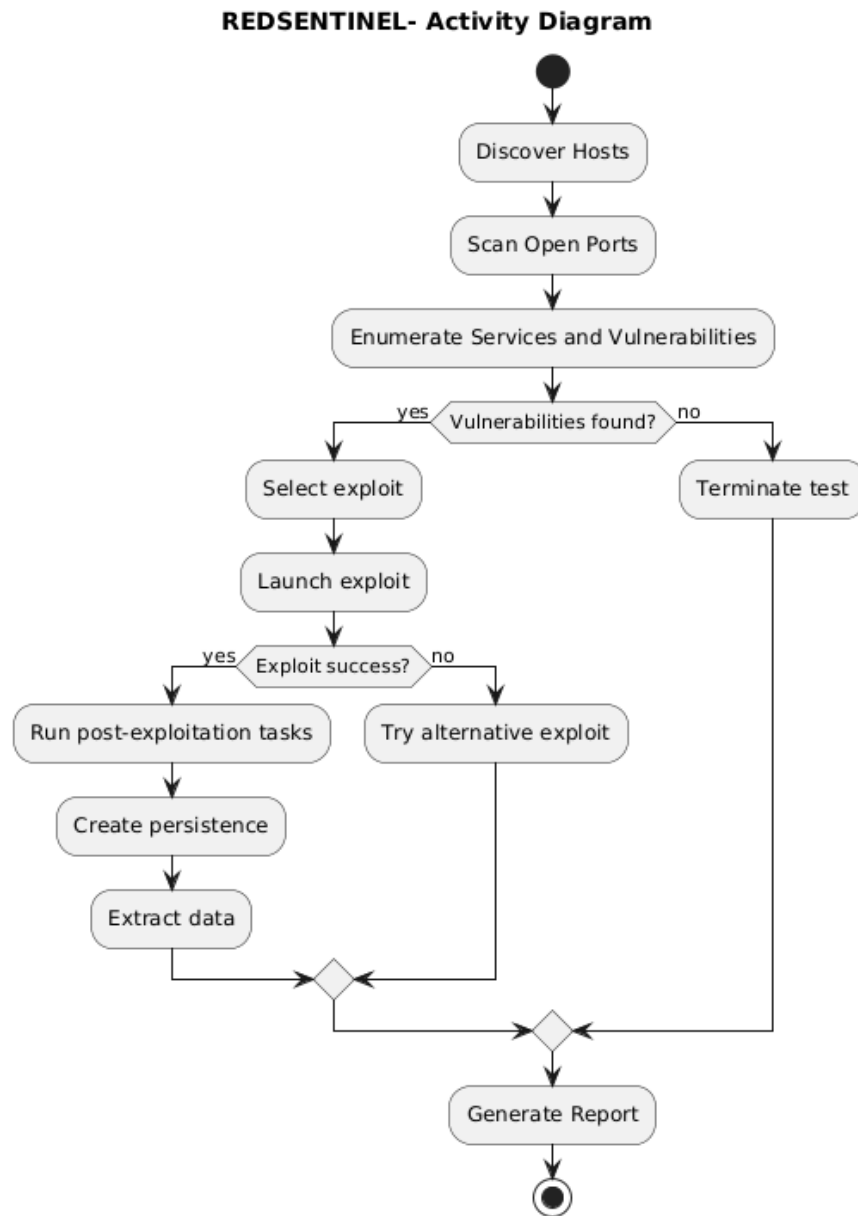


Figure 3.3.1: System Activity Diagram



## **4. WORK COMPLETED AND PROGRESS REPORT**

### **4.1 Work Completed**

#### **4.1.1 Research**

Extensive research was conducted on cross-site scripting vulnerabilities, attack vectors, and modern web security challenges. This included studying the OWASP Top Ten, existing XSS scanners such as XSSStrike and DalFox, and academic literature on ML-based exploit generation. Transformer architectures, byte-level tokenization methods, and adversarial obfuscation techniques were analyzed to determine their suitability for generating context-aware payloads. A detailed survey of microservice architectures and ML-serving pipelines was also completed to guide the system's design.

#### **4.1.2 Familiarization with Tools and Frameworks**

The team became proficient with relevant frameworks including TensorFlow for model development, FastAPI for microservice integration, Docker for containerization, and GitHub for version control and CI/CD workflows. Tools commonly used in penetration testing—such as Burp Suite, OWASP ZAP, and browser-based developer consoles—were familiarized to support payload testing and context extraction. Additional effort was dedicated to understanding deployment tools required for scalable ML microservices.

#### **4.1.3 Dataset Preparation and Preprocessing**

A dataset of XSS payloads was collected and curated from open-source repositories, academic publications, and synthetic generation procedures. Payloads were categorized by injection context (HTML attributes, script blocks, URL parameters, etc.) and normalized into a format compatible with byte-level tokenization. Preprocessing steps included context-labeling, deduplication, noise removal, encoding normalization, and construction of input-output pairs for supervised transformer training.

#### **4.1.4 Model Development and Analysis**

Initially multiple models were developed to analyze the efficacy of different architectures for XSS payload generation.

#### 4.1.4.1 3-Layer GRU Decoder Model & 3-Layer LSTM Decoder Model

In addition to the transformer-based architecture, a recurrent neural network variant was developed to evaluate lightweight sequence-generation approaches for XSS payload synthesis. The implemented model is a three-layer unidirectional GRU decoder, designed to process byte-tokenized input sequences of fixed length (150 tokens).

The architecture begins with an embedding layer that maps discrete byte indices into a 256-dimensional continuous vector space, enabling the model to capture syntactic and semantic relationships between characters commonly used in XSS payloads. Each GRU block contains 256 hidden units and is followed by layer normalization to stabilize training and mitigate exploding gradients. Skip-connections and logical operations are incorporated between recurrent layers to enhance information flow and to allow the model to combine contextual signals derived from both the embedding space and the hidden states.

A final TimeDistributed dense layer projects each timestep output into the vocabulary space, enabling autoregressive token prediction for payload generation. Despite its relatively small size (1.34 million parameters), this model is capable of learning structural patterns such as attribute breakouts, event-handler injections, and common JavaScript invocation sequences. Its efficiency and reduced computational footprint make it suitable for rapid experimentation, ablation studies, and low-latency microservice deployment where a full transformer model may be unnecessary or too resource-intensive.

The same architecture was implemented using LSTM layers for comparison.

**Example Results** This model was capable of generating working XSS payloads from a starting payload seed:

Seed: "<script"

Payload: "<script>alert(1)</script>"

#### 4.1.4.2 Byte-Level Transformer Payload Generator

Following the initial RNN experiments, a more advanced transformer-based sequence-to-sequence model was developed to improve generative fidelity and contextual awareness. This model used a byte-level SentencePiece tokenizer with byte fallback enabled, ensuring full coverage of printable and non-printable characters—a critical requirement

for XSS payloads, which often include symbols outside typical word/token vocabularies.

The preprocessing pipeline encoded both structured metadata (attack class, payload type, risk score) and target payloads into padded integer sequences. Custom encoder-decoder padding masks, autoregressive self-attention masks, and explicit positional encodings were implemented to maintain compatibility with TensorFlow’s serialization system.

The transformer architecture consisted of:

- 4-layer encoder with multi-head attention (8 heads)
- 4-layer decoder with masked multi-head autoregressive attention
- $d_{model} = 256$ ,  $d_{ff} = 1024$
- Final dense projection over the full byte vocabulary

The model was trained using a masked cross-entropy loss function that ignored padding tokens and optimized with the Adam optimizer. Training was stabilized with learning-rate scheduling, ReduceLROnPlateau, and early stopping. A custom inference engine was built, including a specialized decoder-step model for stepwise token generation, enabling greedy, sampling-based, and beam-search decoding strategies.

**Example Results** In practice, the byte-level transformer significantly outperformed the GRU model. It was able to generate longer, structurally complex payloads with embedded JavaScript calls, event handlers, HTML attribute breakouts, and encoded characters. The model could take structured context such as:

Input:

```
<ATTACK_CLASS>event-handler</ATTACK_CLASS>
<PAYLOAD_TYPE>event-mouse</PAYLOAD_TYPE>
<RISK>0.97</RISK>
```

Output:

```
"><img src=1 onerror=prompt(document.domain)>
```

#### 4.1.4.3 T5-Based Large Language Model Fine-Tuning

To benchmark the transformer against a pre-trained language model, a third model was developed by fine-tuning the T5-base architecture using the same structured dataset.

HuggingFace’s `AutoModelForSeq2SeqLM`, `DataCollatorForSeq2Seq`, and `Adafactor` optimizer were used to perform supervised fine-tuning across five epochs with gradient accumulation.

T5’s pre-training on large-scale text corpora provided strong generative priors, and early experimentation revealed significantly faster convergence compared to the GRU and custom transformer models. T5 was capable of producing syntactically rich payloads and often generated multi-stage injection sequences (e.g., event-triggered JavaScript execution combined with HTML entity encoding) without explicitly being instructed to do so.

This model served two purposes:

1. As a baseline metric to evaluate the performance of the custom byte-level transformer
2. As an alternative model candidate for potential integration into Red Sentinel’s payload-generation engine

While T5 exhibited high generative quality, it lacked fine-grained control over byte-level precision, which is essential for payloads requiring arbitrarily encoded characters. Therefore, T5’s role remains supplementary, while the byte-level transformer continues as the main candidate for integration.

## **4.2 Challenges Encountered**

### **1. Dataset Quality and Context Ambiguity**

Obtaining high-quality, context-labeled XSS payload datasets was difficult. Many publicly available samples lacked structured metadata, requiring extensive manual curation and synthetic augmentation.

### **2. Byte-Level Tokenization Complexity**

Designing a tokenizer capable of faithfully representing special characters, escape sequences, and Unicode variants introduced challenges in maintaining consistency between training and inference.

### **3. Model Stability During Training**

Early versions of the transformer exhibited unstable loss curves due to highly variable sequence lengths and the presence of rare byte patterns. This required specialized padding masks, learning-rate scheduling, and careful regularization.

### **4. Context-to-Payload Alignment**

Ensuring the model correctly interpreted structured metadata (such as `<ATTACK_CLASS>` labels) required additional preprocessing logic and multiple ablation studies to

verify attention alignment.

#### **5. Microservice Integration Overhead**

Integrating the model into a containerized microservice pipeline introduced latency, serialization constraints, and cross-module communication issues, particularly during high-volume inference.

#### **6. Sandboxed Evaluation Environment**

Building a controlled, safe execution environment for evaluating generated payloads was non-trivial, as it required strict isolation to prevent unintended script execution or security risks.

#### **7. Obfuscation Module Early-Stage Limitations**

Although the initial obfuscation module worked for basic encoding transformations, more advanced dynamic or multi-stage obfuscation strategies remain challenging and require further model guidance.

### **4.3 Work in Progress**

The project is ongoing with several components under active development. The transformer payload generator has reached a functional stage, but optimization, integration, and extended evaluation remain areas of focus. Improvements to the context extraction module, obfuscation mechanisms, and data pipeline orchestration are currently underway. Additional evaluation experiments and adversarial robustness testing are also planned to ensure practical usability.

#### **4.3.1 Remaining Tasks**

##### **1. Advanced Obfuscation Integration**

Implement dynamic JavaScript-based obfuscation techniques, multi-stage encoding, and adversarial transformations.

##### **2. Context Module Enhancement**

Expand context parsing to handle nested HTML structures, script-block boundaries, and complex dynamic DOM interactions.

##### **3. Model Retraining and Hyperparameter Tuning**

Perform extended training with a refined dataset, larger batch sizes, learning-rate sweeps, and model-depth experiments.

##### **4. Benchmarking Against Existing Tools**

Compare the system's performance with industry tools such as XSSStrike, DalFox, and commercial scanners.

##### **5. Dashboard and Reporting System**

Develop a web-based dashboard for visualizing discovered vulnerabilities, payload effectiveness, and overall scan metrics.

#### 6. Security Hardening of Microservices

Implement API gateway protections, rate limiting, mTLS communication, and container-level isolation.

#### 7. WAF Adversarial Testing Loop

Integrate ML-based XSS detectors to create a defensive-offensive feedback loop for adversarial training.

#### 8. Full End-to-End Pipeline Optimization

Improve orchestration, reduce inference latency, and ensure stable high-throughput performance across modules.

### 4.4 Project Scheduling

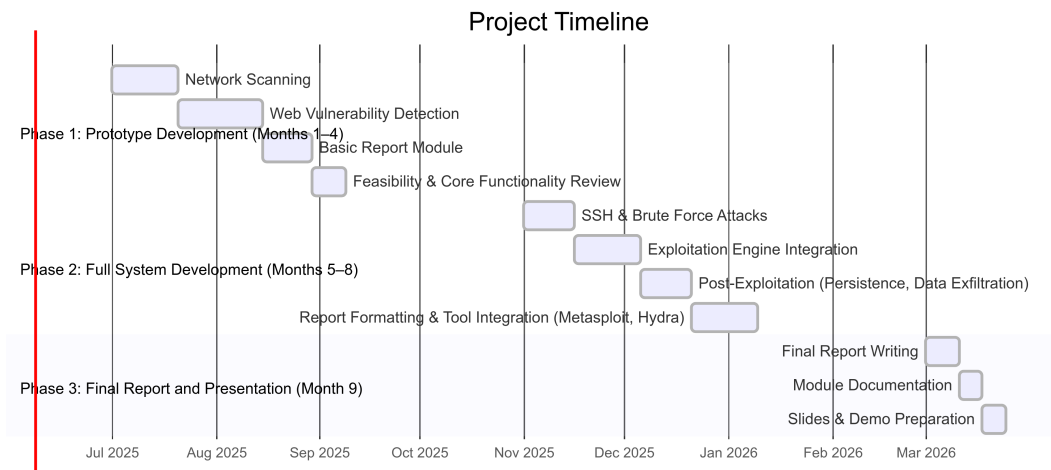


Figure 4.4.1: Gantt Chart Showing Work Schedule

## **5. CONCLUSION AND FUTURE WORK**

### **5.1 Summary**

This mid-term report has presented the design, development, and preliminary evaluation of Red Sentinel, an intelligent machine learning-driven system for automated generation of context-aware Cross-Site Scripting (XSS) payloads. The project addresses critical limitations in traditional security testing tools by leveraging transformer-based neural networks to produce syntactically valid, context-adapted attack payloads that can evade modern filtering mechanisms.

The system architecture follows a microservice design pattern, comprising three core modules: the Context Extraction Module for analyzing injection points, the Payload Handler for generating and obfuscating attack strings, and the Orchestration Core for workflow management. This modular approach ensures scalability, maintainability, and independent development of system components.

Multiple model architectures were developed and evaluated, including GRU-based, LSTM-based, custom transformer-based, and T5-based payload generators. Experimental results demonstrate that the byte-level transformer architecture significantly outperforms recurrent models in generating diverse, structurally complex, and execution-ready XSS payloads. The model's ability to understand and exploit different injection contexts—including HTML attributes, JavaScript blocks, event handlers, and URL parameters—represents a significant advancement over signature-based and manually crafted approaches.

Dataset preparation involved curating and preprocessing XSS payloads from multiple sources, implementing byte-level tokenization to preserve special characters, and constructing context-labeled training pairs. The obfuscation module incorporates encoding-based, structural, and dynamic transformation techniques to enhance payload evasion capabilities against web application firewalls and pattern-matching filters.

### **5.2 Contributions**

The key contributions of this work include:

- 1. Byte-Level Transformer Architecture for XSS Payload Generation**

Development of a specialized encoder-decoder transformer that operates at the

byte level, preserving critical characters and encoding patterns essential for XSS exploitation.

### **2. Context-Aware Payload Synthesis**

Implementation of structured metadata encoding that enables the model to generate payloads tailored to specific injection contexts, improving exploitability and reducing false positives.

### **3. Integrated Obfuscation Pipeline**

Design of a modular obfuscation system that applies multiple transformation strategies to evade detection, rather than treating obfuscation as a post-processing step.

### **4. Microservice-Based Security Testing Framework**

Creation of a scalable, modular architecture that supports independent development, deployment, and testing of vulnerability assessment components.

### **5. Comparative Model Evaluation**

Systematic comparison of multiple neural architectures (GRU, LSTM, custom transformer, T5) for adversarial payload generation, providing insights into their respective strengths and limitations.

## **5.3 Limitations**

While the project has achieved significant progress, several limitations remain:

### **1. Dataset Size and Diversity**

The current dataset, while curated from multiple sources, may not cover all possible injection contexts and obfuscation techniques. Expanding the dataset with real-world vulnerabilities and edge cases would improve model generalization.

### **2. Evaluation Metrics**

Standardized benchmarks for evaluating XSS payload generators are lacking. Current evaluation relies primarily on qualitative assessment and limited automated testing.

### **3. Context Extraction Accuracy**

The Context Module's ability to accurately identify and classify injection points in complex, dynamic web applications requires further development and testing.

### **4. Obfuscation Sophistication**

Advanced obfuscation techniques, particularly those involving dynamic JavaScript generation and multi-stage transformations, remain partially implemented.

### **5. Real-World Validation**

Comprehensive testing against production web applications and modern WAF systems is needed to validate the system's practical effectiveness.



## 5.4 Future Work

Several directions for future development have been identified:

### 5.4.1 Technical Enhancements

1. **Advanced Obfuscation Techniques**

Develop and integrate sophisticated obfuscation methods including polymorphic payload generation, metamorphic transformations, and adversarial examples specifically designed to evade ML-based detectors.

2. **Reinforcement Learning Integration**

Implement reinforcement learning agents that can iteratively refine payloads based on real-time feedback from target applications, similar to the HAXSS approach.

3. **Multi-Modal Context Understanding**

Extend the Context Module to analyze not just HTML and JavaScript, but also CSS, SVG, WebAssembly, and other web technologies that may introduce novel injection vectors.

4. **Adversarial Training Loop**

Create a defensive-offensive training cycle where generated payloads are used to train XSS detectors, and improved detectors inform payload generation strategies.

### 5.4.2 System Integration

1. **Plugin Architecture for Security Tools**

Develop plugins or extensions for popular penetration testing frameworks such as Burp Suite, OWASP ZAP, and Metasploit to enable seamless integration.

2. **Continuous Learning Pipeline**

Implement mechanisms for continuous model improvement based on newly discovered vulnerabilities and evolving web technologies.

3. **Distributed Scanning Capabilities**

Enable distributed, parallel scanning of large web applications using containerized microservices and cloud infrastructure.

### 5.4.3 Evaluation and Validation

1. **Comprehensive Benchmarking**

Conduct extensive comparison studies against commercial and open-source XSS scanners using standardized vulnerability test beds such as OWASP WebGoat,

Damn Vulnerable Web Application (DVWA), and real-world bug bounty platforms.

## **2. WAF Evasion Testing**

Systematically test generated payloads against major web application firewalls (ModSecurity, Cloudflare, AWS WAF, etc.) to measure evasion effectiveness.

## **3. User Studies**

Conduct studies with security professionals to evaluate the system's usability, effectiveness, and integration into existing workflows.

### **5.4.4 Research Extensions**

#### **1. Transfer Learning to Other Vulnerabilities**

Investigate the applicability of the transformer-based approach to other injection vulnerabilities such as SQL injection, Command injection, and LDAP injection.

#### **2. Explainable AI for Security**

Develop interpretability mechanisms to help security professionals understand why certain payloads are generated and how they exploit specific contexts.

#### **3. Defensive Applications**

Explore how the same models can be used defensively to improve XSS detection systems, generate synthetic training data for security tools, and identify vulnerable code patterns.

### **5.5 Final Remarks**

Red Sentinel represents a significant step forward in automating offensive security testing through the application of modern machine learning techniques. By generating context-aware, obfuscated XSS payloads using transformer-based models, the system addresses longstanding limitations in vulnerability assessment tools. The modular architecture ensures that the system can evolve alongside advancing web technologies and security practices.

The work completed to date demonstrates the technical feasibility and practical potential of AI-driven vulnerability discovery. As web applications continue to grow in complexity and importance, automated tools like Red Sentinel will play an increasingly critical role in identifying and mitigating security vulnerabilities before they can be exploited maliciously.

The remaining development work focuses on enhancing obfuscation capabilities, improving context extraction accuracy, conducting comprehensive real-world validation,

and establishing standardized evaluation metrics. Upon completion, Red Sentinel will provide security professionals with a powerful, intelligent tool for discovering XSS vulnerabilities that traditional scanners miss, ultimately contributing to a more secure web ecosystem.

## Bibliography

- [1] B. Pala, L. Pisu, S. L. Sanna, D. Maiorca, and G. Giacinto, “A targeted assessment of cross-site scripting detection tools,” in *Proceedings of the Italian Conference on CyberSecurity (ITASEC 2023)*, ser. CEUR-WS.org, vol. 3488. Bari, Italy: CEUR Workshop Proceedings, 2023, pp. 322–334, creative Commons Attribution 4.0 International (CC BY 4.0). [Online]. Available: <https://ceur-ws.org/Vol-3488/paper26.pdf>
- [2] T. Fink, “Automated xss vulnerability detection through context aware fuzzing and dynamic analysis,” Diploma Thesis, Technische Universität Wien, 2018. [Online]. Available: <https://repositum.tuwien.at/handle/20.500.12708/7741>
- [3] M. Foley and S. Maffeis, “Haxss: Hierarchical reinforcement learning for xss payload generation,” in *Proceedings of the 2022 IEEE 21st International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '22)*, 2022, pp. 147–158.
- [4] X. Song, R. Zhang, Q. Dong, and B. Cui, “Grey-box fuzzing based on reinforcement learning for xss vulnerabilities,” *Applied Sciences*, vol. 13, no. 4, 2023. [Online]. Available: <https://www.mdpi.com/2076-3417/13/4/2482>
- [5] S. Khan, “LI-xss: End-to-end generative model-based xss payload creation,” in *Proceedings of the 21st International Conference on Learning and Technology (L&T 2024)*, 2024, pp. 121–126.
- [6] Y. Frempong, Y. Snyder, E. Al-Hossami, M. Sridhar, and S. Shaikh, “Hijax: Human intent javascript xss generator,” in *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*. SciTePress, 2021, pp. 798–805.
- [7] D. Miczek, D. Gabbireddy, and S. Saha, “Leveraging llm to strengthen ml-based cross-site scripting detection,” *arXiv preprint*, 2025, arXiv:2504.21045. [Online]. Available: <https://arxiv.org/html/2504.21045v1>