# **Final Project Submission Web Application Vulnerability Scanner**

## 1. Introduction

With the rapid growth of web applications, security has become a critical concern. Many websites are vulnerable to attacks such as Cross-Site Scripting (XSS), SQL Injection (SQLi), and Cross-Site Request Forgery (CSRF). The goal of this project is to develop an automated vulnerability scanner capable of detecting these common security issues in web applications. The scanner crawls through web pages, identifies input fields, injects test payloads, and analyzes responses for possible vulnerabilities. The tool also features a Flask-based web interface that allows users to manage scans and view detailed vulnerability reports.

## 2. Objective

To design and implement a Python-based Web Application Vulnerability Scanner that:  
- Detects common OWASP Top 10 vulnerabilities (XSS, SQLi, CSRF, etc.).  
- Provides a user-friendly web interface for scan management.  
- Generates detailed reports with evidence, severity, and remediation suggestions.

## 3. Tools and Technologies

Tools / Libraries used:  
- Python: Core programming language  
- requests: Handling HTTP requests and responses  
- BeautifulSoup (bs4): Parsing and crawling HTML content  
- re (regex): Pattern matching for detecting vulnerabilities  
- Flask: Building the web interface  
- python-docx: (used for report generation)  
- OWASP Top 10 Checklist: Guideline for identifying key web vulnerabilities

## 4. System Design

Architecture Overview:  
1. Crawling Module:  
 - Uses requests and BeautifulSoup to discover pages, URLs, and input fields.  
 - Extracts parameters and form data for further testing.  
  
2. Attack Injection Module:  
 - Injects crafted payloads for:  
 • XSS: <script>alert(1)</script>  
 • SQLi: ' OR '1'='1' --  
 • CSRF: Custom payload to check request forgery.  
 - Monitors responses for changes or indicators of successful attacks.  
  
3. Detection Module:  
 - Uses regex and response pattern analysis to confirm vulnerabilities.  
 - Assigns severity levels (Low, Medium, High).  
  
4. Reporting and Logging Module:  
 - Logs findings (URL, parameter, payload, severity).  
 - Generates structured reports in HTML or JSON and this Word document.  
  
5. Flask Web Interface:  
 - Dashboard for initiating scans.  
 - Displays ongoing scan progress.  
 - Shows vulnerability details with evidence and mitigation tips.

## 5. Implementation Steps

Step 1: Web Crawling  
- Use requests to fetch web pages.  
- Parse HTML with BeautifulSoup to extract forms, links, and parameters.  
  
Step 2: Payload Injection  
- For each discovered input field, inject XSS, SQLi, and CSRF test payloads.  
- Capture server responses.  
  
Step 3: Vulnerability Detection  
- Analyze the returned HTML/HTTP responses using regex for:  
 - Script reflections (<script> tags).  
 - SQL error patterns (e.g., 'You have an error in your SQL syntax').  
- Mark endpoints as vulnerable if patterns match.  
  
Step 4: Logging and Reporting  
- Store scan data (target URL, affected parameter, payload, vulnerable evidence, severity).  
- Export reports to JSON, HTML, and Word (.docx).

## 6. Payload Examples and Detection Patterns

XSS Payloads:  
- <script>alert(1)</script>  
- "'><script>alert(1)</script>  
Detection: Check if payload appears unescaped in response or triggers script execution indicators.  
  
SQLi Payloads:  
- ' OR '1'='1' --  
- " OR "" = "  
Detection: Look for SQL error messages in responses or abnormal content changes.  
  
CSRF Tests:  
- Attempt to submit state-changing requests without CSRF token.  
Detection: Check server response for success when token omitted or verify presence/absence of anti-CSRF tokens in forms.

## 7. Flask Web Interface (UI)

Key Features:  
- Scan Management: Start, pause, and stop scans.  
- Target Input: Provide target base URL and scope settings.  
- Scan Settings: Configure payload sets, depth, and rate limits.  
- Results Dashboard: View discovered vulnerabilities with severity, evidence, and remediation steps.  
- Export: Download report in JSON, HTML, and DOCX formats.

## 8. Reporting and Severity Levels

Each vulnerability entry includes:  
- Target URL and parameter  
- Vulnerability type (XSS, SQLi, CSRF, etc.)  
- Payload used  
- Evidence (response snippet or HTTP log)  
- Severity: Low / Medium / High  
- Recommended Fix: Brief remediation suggestion  
  
Severity Criteria:  
- High: Direct evidence of exploitability (e.g., reflected XSS that executes)  
- Medium: Indicators or errors suggesting injection but not directly exploitable  
- Low: Informational or potential issues (missing security headers)

## 9. Testing and Validation

Testing Approach:  
- Use purposely vulnerable test applications (e.g., OWASP Juice Shop, DVWA) for validation.  
- Compare findings with known vulnerabilities and manual verification.  
  
Validation Steps:  
- Reproduce each finding manually.  
- Confirm whether exploit leads to impact (data disclosure, session theft).  
- Record false positives and refine detection rules.

## 10. Ethical Considerations

- Only test targets you own or have explicit permission to test.  
- Obtain written consent for any security testing on third-party systems.  
- Log and handle sensitive data responsibly; do not expose collected data publicly.  
- Follow responsible disclosure practices when reporting vulnerabilities.

## 11. Limitations and Future Work

Limitations:  
- May produce false positives and false negatives; manual verification needed.  
- Highly dynamic single-page applications (SPAs) may require JavaScript execution (use of headless browsers like Selenium or Playwright).  
- Advanced injection techniques and authentication-bypassing methods are out of scope for the initial version.  
  
Future Enhancements:  
- Integrate a headless browser to handle JS-heavy sites.  
- Add authenticated scanning (login sequences, session handling).  
- Expand payload library and use fuzzing techniques.  
- Add API scanning support and rate-limiting handling.  
- Implement user roles and multi-tenant scanning in Flask UI.

## 12. Conclusion

This project builds a foundational Python-based web application vulnerability scanner that automates discovery, payload injection, detection, and reporting for common vulnerabilities like XSS, SQLi, and CSRF. Combined with a Flask UI, it provides a usable interface for security testing and reporting. With future enhancements, it can evolve into a robust security testing tool.

## 13. References

- OWASP Top Ten: https://owasp.org/www-project-top-ten/  
- OWASP Juice Shop: https://owasp.org/www-project-juice-shop/  
- DVWA (Damn Vulnerable Web Application)  
- Requests documentation  
- BeautifulSoup documentation  
  
Appendix: Sample Code Snippets and Database Schema  
(Include in project submission repository)