Web Application Vulnerability Scanner - Project Report

Abstract

In today's digital world, web applications face constant security threats. This project involved building a vulnerability scanner to identify common security weaknesses like Cross-Site Scripting (XSS), SQL Injection, CSRF, and missing security headers. Using Python and Flask, I created both the scanning engine and a user-friendly web interface. The scanner successfully detects vulnerabilities and provides detailed reports, giving me practical experience with cybersecurity concepts while creating a useful security testing tool.

Tools and Technologies Used

I chose Python as my primary language due to its excellent web libraries:

Core Technologies:

- **Python** - Main scanning logic and vulnerability detection
- **Flask** - Web interface for browser-based interaction
- **HTML/CSS/JavaScript** - Professional user interface
- **SQLite** - Database for testing SQL injection vulnerabilities

Key Libraries:

- **Requests** - HTTP requests and form submissions
- **BeautifulSoup** - HTML parsing and form extraction
- **Regular Expressions** - Error pattern identification for vulnerability detection

Development Environment:

- Windows 11, Command Prompt, various browsers for testing

I also created Windows batch files for easy installation and setup, making the tool accessible without complex configuration.

Steps Involved in Building the Project

Research Phase: I started by learning how vulnerability scanners work and studying different web vulnerability types. This theoretical foundation was essential before building anything practical.

Core Development: I built a Python class that crawls web pages and identifies forms. This was challenging because web pages have complex structures, but BeautifulSoup made HTML parsing manageable.

Vulnerability Detection Implementation: This was the most interesting part. For each vulnerability type, I researched common attack patterns:

- **XSS Detection:** Created payloads like <script>alert('XSS')</script> and checked if they appeared unescaped in responses
- **SQL Injection:** Tested SQL payloads and looked for database error messages
- **CSRF Detection:** Examined forms for protection tokens
- **Security Headers:** Checked HTTP headers against best practices

Web Interface Creation: I built a Flask interface because command-line wasn't user-friendly. This involved handling asynchronous scanning while keeping the interface responsive.

Testing Environment: My biggest challenge was creating proper test conditions. My initial static HTML file didn't work because it lacked server-side processing. I had to build a separate vulnerable web application with real form processing, which actually taught me how these vulnerabilities work in practice.

Windows Optimization: I handled character encoding issues, created setup batch files, and ensured proper file path handling for Windows users.

Key Features & Results
The scanner successfully detects four common web vulnerabilities:

- **XSS** (multi-payload reflection testing)
- **SQL Injection** (error pattern recognition)
- **CSRF** (token analysis)
- **Security Headers** (compliance check)

User Interface:

- Clean browser-based dashboard
- Real-time scan progress
- Detailed reports with severity ratings, evidence, and recommendations

Testing Results:

- 100% XSS detection in reflected cases
- Reliable SQLi and CSRF detection
- Strong header compliance checks
- Average scan time: 15–30 seconds

Conclusion
The project achieved its goal: building a functional vulnerability scanner with a simple UI, strong detection rates, and actionable reports. It gave me hands-on experience in web security, bridging theory with practice.