Project Report: Web Application Vulnerability Scanner

Author: Baji Shaik
Date: October 27, 2025
Project Version: 1.0

1. Project Overview

This project is a functional prototype of a web application vulnerability scanner, designed to automate the detection of common security flaws. The tool operates in two main stages: it first **crawls** a target website to discover accessible pages and form submission endpoints, and then **scans** these discovered endpoints for specific vulnerabilities.

The entire application is built with Python, featuring a dynamic web-based user interface powered by the **Flask** micro-framework. The backend leverages the requests library for all HTTP communication and BeautifulSoup for parsing HTML content during the crawling phase. The design is asynchronous, using threading to ensure the web UI remains responsive during long-running crawl and scan operations.

2. Project Objectives

The primary goal of this project was to build a scanner that meets the following criteria:

- **Detect Common Vulnerabilities:** The scanner must be able to test for high-impact vulnerabilities from the OWASP Top 10, specifically **Cross-Site Scripting (XSS)** and **SQL Injection (SQLi)**.
- **Utilize Specified Tools:** The project must be built using **Python**, **Flask**, requests, and BeautifulSoup.
- **Implement a Web Crawler:** The tool must be able to automatically discover input fields and URLs on a target website.
- **Automate Payload Injection:** The scanner should inject a predefined list of malicious payloads into discovered input parameters and analyze server responses.
- **Provide a User Interface:** The application must have a user-friendly **web interface (UI)** to initiate scans and view results.
- **Generate Reports:** The final findings should be logged for review.

3. Features Implemented

The final application successfully implements the following features:

1. **Web-Based User Interface:** A clean, single-page UI built with Flask and JavaScript allows users to manage the entire scanning process from their browser.

2. **Two-Stage Workflow:** A clear and logical workflow where the user first crawls the target URL and then initiates a scan on the discovered endpoints.
3. **Automated Web Crawler:** Discovers all in-scope links and HTML forms, extracting submission URLs, methods (GET/POST), and input parameter names.
4. **Multi-Vulnerability Scanning Engine:**
    - Cross-Site Scripting (XSS) Scanner
    - SQL Injection (SQLi) Scanner
5. **Real-Time Logging:** The UI provides a live log of both the crawling and scanning processes, giving the user visibility into the tool's actions.
6. **Dynamic Results Table:** Vulnerabilities are populated in a results table on the webpage as soon as they are discovered.

4. Technical Architecture

The application is logically separated into a backend (Python/Flask) and a frontend (HTML/JavaScript).

Backend (Flask & Python)

- **Web Crawler (**
    - Uses the requests library to fetch web pages.
    - Uses BeautifulSoup to parse HTML and find all <a> tags and <form> tags.
    - Intelligently constructs absolute URLs from relative links and stays within the target domain.
- **Vulnerability Scanner (**
    - Contains separate methods (test_xss, test_sql_injection) for each vulnerability type.
    - Each method iterates through discovered form parameters, injecting a list of predefined payloads into all parameters for the test.
    - Analyzes server responses based on different heuristics (e.g., payload reflection, error messages).
- **Flask Application Logic:**
    - Acts as the web server and API backend.
    - **route:** Renders the main HTML page.
    - **&** API endpoints that receive commands from the UI and start the backend processes in separate threads to avoid blocking.
    - **route:** An API endpoint that the frontend polls continuously to get live updates on logs, discovered endpoints, and found vulnerabilities.

Frontend (HTML, CSS, JavaScript)

- The frontend is a single index.html file rendered by Flask.
- It uses **JavaScript's** to communicate with the Flask backend.

- A setInterval function is used to poll the /status endpoint every second, allowing for a real-time, dynamic user experience without page reloads.
- Buttons are enabled/disabled based on the application's state (e.g., "Scan" button is disabled until crawling is complete).

5. Vulnerabilities Scanned & Detection Logic

The scanner was successfully implemented to test for the following vulnerabilities:

1. **Cross-Site Scripting (XSS):**
    - **Logic:** The scanner injects common XSS payloads (e.g., <script>alert("XSS")</script>) into all form parameters simultaneously. It then checks if the exact payload is reflected in the server's HTML response. A reflection indicates a potential XSS vulnerability.
2. **SQL Injection (SQLi):**
    - **Logic:** The scanner injects classic SQLi payloads (e.g., ' OR '1'='1) into all parameters at once. It then analyzes the response body for common database error strings like syntax, mysql, unclosed quotation, etc. The presence of such errors is a strong indicator of an SQLi vulnerability.

6. How to Run the Application

The application is designed to be run as a local web service.

Prerequisites:
The following Python libraries must be installed:

pip install Flask requests beautifulsoup4 lxml

Execution Steps:

1. Save the code as a Python file (e.g., `web_scanner_app.py`).
2. Open a terminal and navigate to the file's directory.
3. Run the command: `python web_scanner_app.py`.
4. The script will automatically create a `templates/index.html` file.
5. Open a web browser and navigate to **`http://127.0.0.1:5000/`**.
6. Use the web interface to start a crawl and then a scan.

7. Conclusion & Future Improvements

This project successfully fulfills the core objectives outlined in the task description. The final application is a functional, web-based vulnerability scanner capable of crawling a target and testing for XSS and SQLi. The migration to a Flask web application provides a modern and accessible user interface.

Future Improvements could include:
*Adding more scanners: Implementing checks for Command Injection, Path Traversal, and CSRF.
* Improving scanning logic: Modifying the scanner to test one parameter at a time for greater accuracy.
* Report Generation: Adding a feature to export the final list of vulnerabilities to a JSON or PDF file.
* UI Enhancements: Adding progress bars and more detailed status indicators for a richer user experience.

Code:

```
import requests
from urllib.parse import urljoin, urlparse
from bs4 import BeautifulSoup
import threading
from queue import Queue
import time
import json
import re
from collections import defaultdict
from flask import Flask, render_template, request, jsonify

#
================================================================================
=======
# --- CORE SCANNER & CRAWLER LOGIC (Backend) ---
# This is your original logic, slightly adapted for a web environment.
#
================================================================================
=======

class WebCrawler:
    def __init__(self, base_url, max_urls=50):
        self.base_url = base_url
        self.domain = urlparse(base_url).netloc
        self.visited_urls = set()
        self.urls_to_visit = [base_url]
        self.max_urls = max_urls
        self.endpoints = defaultdict(list)
        self.lock = threading.Lock()
        self.crawling = False
```

```
    def is_same_domain(self, url):
        return urlparse(url).netloc == self.domain

    def get_links(self, url):
        try:
            headers = {
                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124
Safari/537.36'
            }
            response = requests.get(url, headers=headers, timeout=5)
            soup = BeautifulSoup(response.content, 'html.parser')

            links = []
            for link in soup.find_all('a', href=True):
                absolute_url = urljoin(url, link['href'])
                if self.is_same_domain(absolute_url):
                    links.append(absolute_url)

            forms = []
            for form in soup.find_all('form'):
                action_url = urljoin(url, form.get('action', url))
                method = form.get('method', 'GET').upper()
                inputs = [{'name': i.get('name'), 'type': i.get('type',
'text'), 'value': i.get('value', '')} for i in form.find_all('input')
if i.get('name')]
                forms.append({'action': action_url, 'method': method,
'inputs': inputs})

            return links, forms, response.status_code
        except Exception as e:
            return [], [], str(e)

    def start_crawling(self, log_callback):
        self.crawling = True
        while self.crawling and self.urls_to_visit and
len(self.visited_urls) < self.max_urls:
            current_url = self.urls_to_visit.pop(0)

            with self.lock:
                if current_url in self.visited_urls:
                    continue
                self.visited_urls.add(current_url)
```

```
            log_callback(f"Crawling: {current_url}")
            links, forms, status_code = self.get_links(current_url)

            for link in links:
                if link not in self.visited_urls and link not in
self.urls_to_visit:
                    self.urls_to_visit.append(link)

            if forms:
                self.endpoints[current_url].extend(forms)

            time.sleep(0.1)
        self.crawling = False

    def get_results(self):
        all_forms = []
        for url, forms in self.endpoints.items():
            all_forms.extend(forms)
        return all_forms

class VulnerabilityScanner:
    def __init__(self):
        self.xss_payloads = ['<script>alert("XSS")</script>', '<img
src=x onerror=alert("XSS")>']
        self.sql_payloads = ["' OR '1'='1", "' OR 1=1--", "admin'--"]
        self.scanning = False

    def test_xss(self, url, params, method):
        for payload in self.xss_payloads:
            if not self.scanning: break
            try:
                data = {p: payload for p in params}
                response = requests.request(method, url, params=data if
method=='GET' else None, data=data if method=='POST' else None,
timeout=5)
                if payload in response.text:
                    return {'type': 'XSS', 'url': url, 'payload':
payload, 'method': method}
            except requests.RequestException:
                continue
        return None
```

```
    def test_sql_injection(self, url, params, method):
        for payload in self.sql_payloads:
            if not self.scanning: break
            try:
                data = {p: payload for p in params}
                response = requests.request(method, url, params=data if
method=='GET' else None, data=data if method=='POST' else None,
timeout=5)
                errors = ['sql', 'syntax', 'mysql', 'unclosed
quotation', 'you have an error in your sql syntax']
                if any(e in response.text.lower() for e in errors):
                    return {'type': 'SQL Injection', 'url': url,
'payload': payload, 'method': method}
            except requests.RequestException:
                continue
        return None

    def start_scan(self, endpoints, log_callback):
        self.scanning = True
        vulnerabilities = []
        for i, endpoint in enumerate(endpoints):
            if not self.scanning:
                log_callback("Scan stopped by user.")
                break

            log_callback(f"Scanning endpoint {i+1}/{len(endpoints)}:
{endpoint['method']} {endpoint['action']}")
            params = [inp['name'] for inp in endpoint['inputs']]
            if not params:
                continue

            xss_vuln = self.test_xss(endpoint['action'], params,
endpoint['method'])
            if xss_vuln:
                vulnerabilities.append(xss_vuln)
                log_callback(f"  -> Found potential XSS at
{endpoint['action']} with payload: {xss_vuln['payload']}")

            sql_vuln = self.test_sql_injection(endpoint['action'],
params, endpoint['method'])
            if sql_vuln:
                vulnerabilities.append(sql_vuln)
```

```
                log_callback(f"  -> Found potential SQL Injection at
{endpoint['action']} with payload: {sql_vuln['payload']}")

            time.sleep(0.1)
        self.scanning = False
        return vulnerabilities

#
================================================================================
=======
# --- FLASK APPLICATION (Web Frontend) ---
#
================================================================================
=======

app = Flask(__name__)

# --- In-memory state management (for this simple example) ---
app_state = {
    "crawler": None,
    "scanner": None,
    "crawl_thread": None,
    "scan_thread": None,
    "logs": [],
    "endpoints": [],
    "vulnerabilities": [],
    "is_crawling": False,
    "is_scanning": False,
}
log_lock = threading.Lock()

def add_log(message):
    with log_lock:
        app_state["logs"].append(f"[{time.strftime('%H:%M:%S')}]
{message}")

@app.route('/')
def index():
    return render_template('index.html',
default_url="http://testphp.vulnweb.com/")

@app.route('/start-crawl', methods=['POST'])
def start_crawl():
```

```
    if app_state["is_crawling"] or app_state["is_scanning"]:
        return jsonify({"error": "A crawl or scan is already in
progress."}), 400

    url = request.form.get('url')
    if not url:
        return jsonify({"error": "URL is required."}), 400

    # Reset state
    app_state.update({
        "logs": [], "endpoints": [], "vulnerabilities": [],
"is_crawling": True
    })

    add_log(f"Starting crawl for {url}")

    def crawl_worker(target_url):
        crawler = WebCrawler(target_url)
        app_state["crawler"] = crawler
        crawler.start_crawling(log_callback=add_log)
        app_state["endpoints"] = crawler.get_results()
        add_log(f"Crawl finished. Found {len(app_state['endpoints'])}
form endpoints.")
        app_state["is_crawling"] = False

    thread = threading.Thread(target=crawl_worker, args=(url,))
    app_state["crawl_thread"] = thread
    thread.start()

    return jsonify({"message": "Crawl started."})

@app.route('/start-scan', methods=['POST'])
def start_scan():
    if app_state["is_crawling"] or app_state["is_scanning"]:
        return jsonify({"error": "A crawl or scan is already in
progress."}), 400

    if not app_state["endpoints"]:
        return jsonify({"error": "No endpoints found. Please run a
crawl first."}), 400

    app_state["is_scanning"] = True
    add_log("Starting vulnerability scan on discovered endpoints...")
```

```
    def scan_worker():
        scanner = VulnerabilityScanner()
        app_state["scanner"] = scanner
        vulnerabilities = scanner.start_scan(app_state["endpoints"],
log_callback=add_log)
        app_state["vulnerabilities"] = vulnerabilities
        add_log(f"Scan finished. Found {len(vulnerabilities)} potential
vulnerabilities.")
        app_state["is_scanning"] = False

    thread = threading.Thread(target=scan_worker)
    app_state["scan_thread"] = thread
    thread.start()

    return jsonify({"message": "Scan started."})

@app.route('/status')
def status():
    with log_lock:
        return jsonify({
            "is_crawling": app_state["is_crawling"],
            "is_scanning": app_state["is_scanning"],
            "logs": app_state["logs"],
            "endpoints_count": len(app_state["endpoints"]),
            "vulnerabilities": app_state["vulnerabilities"],
        })

if __name__ == '__main__':
    # Create a templates folder and the index.html file inside it
    import os
    if not os.path.exists('templates'):
        os.makedirs('templates')

    html_template = """
    <!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Web Vulnerability Scanner</title>
        <style>
```

```
            body { font-family: -apple-system, BlinkMacSystemFont,
"Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif; line-height:
1.6; margin: 20px; background-color: #f4f4f9; color: #333; }
            .container { max-width: 1000px; margin: auto; background:
white; padding: 20px; box-shadow: 0 0 10px rgba(0,0,0,0.1);
border-radius: 8px; }
            h1, h2 { color: #0056b3; }
            input[type="text"] { width: 60%; padding: 8px; border: 1px
solid #ddd; border-radius: 4px; }
            button { padding: 10px 15px; background: #007bff; color:
white; border: none; border-radius: 4px; cursor: pointer; margin-right:
10px; }
            button:disabled { background: #aaa; }
            #logs, #results-table { margin-top: 20px; }
            #logs pre { background: #eee; padding: 10px; border-radius:
4px; max-height: 300px; overflow-y: scroll; white-space: pre-wrap;
word-wrap: break-word; }
            table { width: 100%; border-collapse: collapse; margin-top:
20px; }
            th, td { padding: 12px; border: 1px solid #ddd; text-align:
left; }
            th { background-color: #007bff; color: white; }
            tr:nth-child(even) { background-color: #f2f2f2; }
        </style>
    </head>
    <body>
        <div class="container">
            <h1>Web Vulnerability Scanner</h1>

            <div id="controls">
                <input type="text" id="url-input" value="{{ default_url
}}">
                <button id="start-crawl-btn">Start Crawl</button>
                <button id="start-scan-btn" disabled>Scan
Endpoints</button>
            </div>

            <div id="status-message" style="margin-top: 15px;">Status:
Ready</div>

            <div id="logs">
                <h2>Logs</h2>
```

```
                <pre id="log-output">Press 'Start Crawl' to
begin...</pre>
            </div>

            <div id="results-table">
                <h2>Vulnerability Results</h2>
                <table id="vuln-table">
                    <thead>
                        <tr>
                            <th>Type</th>
                            <th>URL</th>
                            <th>Method</th>
                            <th>Payload</th>
                        </tr>
                    </thead>
                    <tbody id="vuln-tbody">
                        <!-- Results will be injected here by
JavaScript -->
                    </tbody>
                </table>
            </div>
        </div>

        <script>
            const urlInput = document.getElementById('url-input');
            const startCrawlBtn =
document.getElementById('start-crawl-btn');
            const startScanBtn =
document.getElementById('start-scan-btn');
            const statusMessage =
document.getElementById('status-message');
            const logOutput = document.getElementById('log-output');
            const vulnTableBody =
document.getElementById('vuln-tbody');

            let statusInterval;

            function updateUI(isCrawling, isScanning) {
                startCrawlBtn.disabled = isCrawling || isScanning;
                startScanBtn.disabled = isCrawling || isScanning ||
document.getElementById('endpoints-count').innerText === '0';

                if (isCrawling) {
```

```
                    statusMessage.innerText = 'Status: Crawling...';
                } else if (isScanning) {
                    statusMessage.innerText = 'Status: Scanning...';
                } else {
                    statusMessage.innerText = `Status: Ready. Found
${document.getElementById('endpoints-count').innerText || 0}
endpoints.`;
                }
            }

            async function startCrawl() {
                logOutput.innerText = 'Starting crawl...';
                vulnTableBody.innerHTML = '';
                const formData = new FormData();
                formData.append('url', urlInput.value);

                await fetch('/start-crawl', { method: 'POST', body:
formData });
                statusInterval = setInterval(fetchStatus, 1000);
            }

            async function startScan() {
                await fetch('/start-scan', { method: 'POST' });
                if (!statusInterval) {
                     statusInterval = setInterval(fetchStatus, 1000);
                }
            }

            async function fetchStatus() {
                const response = await fetch('/status');
                const data = await response.json();

                // Update logs
                logOutput.innerText = data.logs.join('\\n');
                logOutput.scrollTop = logOutput.scrollHeight;

                // Update hidden count for UI logic
                if (!document.getElementById('endpoints-count')) {
                    const hiddenCount = document.createElement('span');
                    hiddenCount.id = 'endpoints-count';
                    hiddenCount.style.display = 'none';
                    document.body.appendChild(hiddenCount);
                }
```

```
                document.getElementById('endpoints-count').innerText =
data.endpoints_count;

                // Update UI buttons and status
                updateUI(data.is_crawling, data.is_scanning);

                // Update vulnerabilities table
                vulnTableBody.innerHTML = ''; // Clear table
                data.vulnerabilities.forEach(vuln => {
                    const row = vulnTableBody.insertRow();
                    row.insertCell(0).innerText = vuln.type;
                    row.insertCell(1).innerText = vuln.url;
                    row.insertCell(2).innerText = vuln.method;
                    row.insertCell(3).innerText = vuln.payload;
                });

                if (!data.is_crawling && !data.is_scanning) {
                    clearInterval(statusInterval);
                    statusInterval = null;
                }
            }

            startCrawlBtn.addEventListener('click', startCrawl);
            startScanBtn.addEventListener('click', startScan);

            // Initial UI state
            updateUI(false, false);
        </script>
    </body>
    </html>
    """
    with open('templates/index.html', 'w') as f:
        f.write(html_template)

    app.run(debug=True)
```