Lab 5: Automated Web Vulnerability Analysis Using Python

Course: FSCT 8561 – Security Applications
Instructor: Dr. Maryam R. Aliabadi
Lab Duration: 3 Hours

Overview

In Lab 4, you played the role of a forensic analyst, passively inspecting network traffic to detect attacks. In Lab 5, you transition into the role of an Application Security Engineer. Most modern breaches occur at the application layer, where business logic and user input meet. Instead of waiting for traffic to arrive, you will use Python to actively "interrogate" a web server. By automating HTTP requests, you will probe the OWASP Juice Shop—a modern, sophisticatedly insecure application—to identify critical flaws like Injection and Cross-Site Scripting (XSS).

Learning Objectives

- **Automate** the reconnaissance of web endpoints using Python's requests library.
- **Identify** and exploit Reflected XSS and SQL Injection vulnerabilities.
- **Analyze** HTTP response headers to detect server-side information leakage.
- **Develop** custom Python logic to programmatically flag security misconfigurations.
- **Bridge** the gap between network-layer security and application-layer defense.

Pre-requisite

- Completed Labs 1–4
- Understanding of TCP/IP protocols
- Basic understanding of HTML and the HTTP request/response cycle.
- Understanding OWASP TOP TEN

Required Reading & Tutorials

- Mastering Python for Networking and Security – Chapter 9 and 10
    - https://learning.oreilly.com/library/view/mastering-python-for/9781839217166/
    - Source code on: https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition
- Web Security Academy (PortSwigger): * What is an Interception Proxy?
- OWASP Top 10 Reference: A01:2021-Broken Access Control, A03:2021-Injection, …

Lab Scenario

You have been hired to perform a Point-in-Time Vulnerability Assessment on "Juice Shop," a new e-commerce platform. The development team suspects that their input validation is weak and that the server is "chatty"—leaking too much information in its headers. Your mission is to build a custom Python-based scanner that can crawl the site, test common attack vectors, and generate a report on the server's security posture before the platform goes live.

Environment Setup

To ensure a controlled and reproducible environment, you must deploy OWASP Juice Shop locally using Docker.

1.1 Install Docker

Run the following commands in Kali Linux:

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y docker.io
sudo systemctl enable docker --now
```

1.2 Deploy OWASP Juice Shop

Pull and run the official Juice Shop image:

```
sudo docker pull bkimminich/juice-shop
sudo docker run -d -p 3000:3000 bkimminich/juice-shop
```

Verify by visiting http://localhost:3000 in Firefox. You should see the OWASP Juice Shop storefront.

Part 1 – Reconnaissance of the Vulnerable Server

Before attacking, you must understand the "surface area." Identify the application attack surface.

1. Explore the Juice Shop interface.
2. Identify:
    - Input fields and forms
    - URL parameters
    - API endpoints
2. Observe:
    - Error messages
    - HTTP status codes
    - Response headers

Expected Observations: Verbose error handling, predictable endpoints, exposed metadata.

Part 2 – Python-Based HTTP Request Automation

You goal in this part is to automate interaction with the server. Develop a Python script (http_scanner.py) that:

1. Sends HTTP GET and POST requests to selected (e.g., /rest/products/search) endpoints
2. Injects varying input values into request parameters
3. Logs:
    - Endpoint tested
    - HTTP method
    - Payload used
    - Status code
    - Response length

Part 3 – XSS Vulnerability Detection

Cross-Site Scripting (XSS) occurs when the server "reflects" uncleaned input back to the browser.

1. **Scripting:** Feed your scanner a list of payloads, such as <script>alert(1)</script> or <u>test</u>.
2. **Detection:** Have Python check if the exact string <script> appears in the r.text output.
3. **Expected Observations:** If the script tag is returned unchanged, the endpoint is vulnerable to Reflected XSS.

Part 4 – SQL Injection Detection

SQLi allows an attacker to interfere with the queries an application makes to its database. Identify SQL injection vulnerabilities.

1. **Scripting:** Inject syntax-breaking characters like ', --, or ') OR 1=1--.
2. **Detection:** Monitor for HTTP 500 status codes or specific database error strings (e.g., "SQLITE_ERROR").
3. **Expected Observations:** Successful bypass of the login screen or a detailed error message revealing the database type.

Part 5 – Server & Configuration Analysis

A secure server should hide its identity and enforce strict browser rules.

- **Task:** Extend your script to check for the **absence** of these headers:
    - Content-Security-Policy (CSP)
    - Strict-Transport-Security (HSTS)
    - X-Content-Type-Options
- **Scripting Logic:** If a header is missing, print a "Low Severity" warning to the console.

Part 6 – Automated Detection Logic

Apply IDS-style logic at the application layer. Extend your Python scripts to:

- Automatically flag:
  - XSS indicators
  - SQL injection indicators
  - Configuration issues
- Avoid duplicate alerts
- Output a concise vulnerability summary

Part 7 - Reflection Questions

1. In Lab 4, we saw plaintext passwords in a PCAP. In Lab 5, we saw them potentially leaked via SQLi. Which is harder to defend, and why?
2. How does **Parameter Tampering** in Python differ from simply typing in a browser URL bar?
3. If an application is encrypted (HTTPS), can Scapy (Lab 4) still see the SQLi payloads? Can your Python script (Lab 5) still see the responses?
4. Why is "Automated Scanning" often followed by "Manual Verification"?

Part 7 – Security Analysis

Write 300–400 words synthesizing your findings: If you were the CISO of Juice Shop, what is the *first* thing you would fix based on your Python script's output?

- Discuss the "Noise" factor: Does automated scanning create enough noise for the IDS (from Lab 4) to catch it?
- Compare the risk of **Information Leakage** (Headers) vs. **Direct Exploitation** (SQLi).

Deliverables

Submit **one PDF file** containing:

- `http_scanner.py-` HTTP Endpoint and Configuration Scanner
- `vulnerability_detector.py` – Automated Web Vulnerability Detection Script
- Screenshots or recordings demonstrate successful detection
- Security analysis and reflection report

Filename format: Lab**5**-FirstName-LastName-StudentNumber.pdf

Good Luck!

Lab 5 Grading Rubric (Total: 100 points)

Component	Points	Full Credit	Partial Credit	Minimal/No Credit
http_scanner.py (HTTP Scanning Script)	20	Script runs correctly; enumerates endpoints; logs status codes, headers, and responses clearly	Runs with limited coverage or minor logging errors	Script missing, does not run, or incorrect
vulnerability_detector.py (Vulnerability Detection Script)	25	Correctly tests for SQLi and XSS; flags findings; avoids duplicates; produces a clear summary	Partial detection or logic/output issues	Script missing or non-functional
Server & Configuration Analysis	15	Identifies misconfigurations, missing security headers, and SSL/TLS weaknesses	Partial identification or minor issues	Analysis missing or incorrect
Evidence & Reproducibility	10	Clear screenshots/logs; steps reproducible; results traceable to scripts	Incomplete evidence or minor clarity issues	No evidence or not reproducible
Reflection & Security Discussion	30	300–400 words; clear analysis of findings, risks, and mitigations tied to Juice Shop	Partial coverage or weak linkage to findings	Missing, generic, or incomplete