

جامعة آل البيت

**Al-Al Bayt University**

**Web Vulnerability Scanner**

**Presented by:**

|  |  |
| --- | --- |
| Ahmad Ali Shwaiyat: | 2100906056 |
| Mohannad Abdallah Alzoubi: | 2100906042 |
| Yousef Mohammad Yousef Hjooj: | 2100906076 |
| Abdalrahman Reda Albeshtawi: | 2100906014 |

**Supervisor: Dr. Khaled Batiha**

**Submitted in Partial Fulfillment of the Requirements for bachelor’s degree in Cybersecurity**

**Prince Al Hussein bin Abdullah Faculty of Information Technology**  
**Computer Science**

**Al-Mafraq- Jordan**

**DECLARATION**

As part of the requirements for a bachelor's degree in Cybersecurity, we Mohammad Alzoubi, Ahmad Shwaiyat, Yousef Hjooj, and Abd Abdalrahman Albeshtawi

state that the project titled “**Web Vulnerability Scanner**” is our creation. We confirm that all data, sources, and information used in this project have been appropriately referenced and acknowledged.

Furthermore, we declare that this project has not been previously submitted for credit, towards another program or test at any institution.

**Table of Contents**

[Chapter 1: Introduction 4](#_Toc197809022)

[1.1 Project Problem 4](#_Toc197809023)

[1.2 Project Goals 4](#_Toc197809024)

[1.3 What is an automated web vulnerability ? 5](#_Toc197809025)

[1.4 Why are vulnerability scanners Important? 5](#_Toc197809026)

[1.5 What the project covers 6](#_Toc197809027)

[1.6 Beneficiaries 7](#_Toc197809028)

[Chapter 2: Overview of Target vulnerabilities 8](#_Toc197809029)

[2.1 Server-side Request Forgery (SSRF) 8](#_Toc197809030)

[2.2 Server-side Template Injection (SSTI) 8](#_Toc197809031)

[2.3 Path Traversal 9](#_Toc197809032)

[2.4 Cross-Site Scripting (XSS) 9](#_Toc197809033)

[Chapter 3: Detection Methodologies 13](#_Toc197809034)

[3.1 SSRF Detection Technique 13](#_Toc197809035)

[3.2 SSTI Detection Technique 24](#_Toc197809036)

[3.3 Path Traversal Detection Technique 24](#_Toc197809037)

[3.4 XSS Detection Technique 25](#_Toc197809038)

[Chapter 4: Building the Vulnerability Scanner 30](#_Toc197809039)

[4.1 How the Scanner works? 30](#_Toc197809040)

[4.2 Choice of Programming and Libraries 30](#_Toc197809041)

[4.3 User Graphic interface (GUI) 30](#_Toc197809042)

[4.4 Comparisons with other vulnerability 30](#_Toc197809043)

[Chapter 5: Conclusion 30](#_Toc197809044)

[5.1 Summary of Achievements 30](#_Toc197809045)

[5.2 Potential Enhancements 30](#_Toc197809046)

[5.3 References 30](#_Toc197809047)

# Introduction

## Project Problem

Modern web applications are increasingly targeted by cyberattacks due to vulnerabilities such as SQL injection, cross-site scripting (XSS), insecure APIs, and misconfigured servers. Manual vulnerability detection is time-consuming, error-prone, and requires specialized expertise. Many organizations, especially small-to-medium enterprises (SMEs), lack the resources to implement robust security practices, leaving their systems exposed to breaches. This project addresses the critical need for an automated, accessible, and efficient web vulnerability scanner to identify and mitigate risks proactively.

## Project Goals

The main goals of this project are:

1. To develop an automated web vulnerability scanner that identifies and reports common security weaknesses.
2. To improve the security posture of web applications by enabling early detection and remediation of vulnerabilities.
3. To provide a user-friendly tool that can be used by developers, security teams, and organizations regardless of their technical expertise.

## What is an automated web vulnerability ?

An automated web vulnerability scanner is a software tool designed to systematically identify security weaknesses in web applications, APIs, and servers by combining predefined rules, machine learning (ML), and simulated attack patterns. It eliminates the need for manual penetration testing, enabling rapid, scalable, and repeatable security assessments.

## Why are vulnerability scanners Important?

1. Cost Efficiency: Reduce expenses associated with manual security audits.
2. Proactive Defense: Identify vulnerabilities before attackers exploit them.
3. Compliance: Meet regulatory standards (e.g., GDPR, PCI-DSS).
4. Reputation Protection: Prevent data breaches that damage organizational trust.
5. Continuous Monitoring: Enable real-time scanning in DevOps pipelines (shift-left security).

## What the project covers

Path Traversal

* Definition: Exploits improper input sanitization to access unauthorized files (e.g., /../../etc./passwd).
* Impact: Data theft, system compromise.
* Detection: Inject traversal sequences (e.g., ../, %2e%2e%2f) and analyze server responses for file disclosures.

Cross-Site Scripting (XSS)

* Definition: Injects malicious scripts into web pages (e.g., <script>alert(1)</script>).
* Types: Stored (persistent), Reflected (URL-based), DOM-based (client-side).
* Detection: Submit payloads and check for unencoded output in HTML/JS contexts.

Server-Side Request Forgery (SSRF)

* Definition: Forces a server to make unauthorized internal requests (e.g., to AWS metadata endpoints).
* Impact: Internal network reconnaissance, cloud credential theft.
* Detection: Send URLs with internal IPs (e.g., http://169.254.169.254) and monitor responses.

1.5.4 Server-Side Template Injection (SSTI)

* Definition: Injects malicious code into templating engines (e.g., Jinja2, Smarty).
* Impact: Remote code execution (RCE), data leaks.
* Detection: Test with template syntax (e.g., {{7\*7}} → 49 indicates vulnerability).

## Beneficiaries

* Developers: Integrate security into CI/CD pipelines.
* Penetration Testers: Accelerate vulnerability discovery.
* Organizations: Reduce breach risks and audit costs.
* End Users: Safeguard personal data from exploits.

# Overview of Target vulnerabilities

## Server-side Request Forgery (SSRF)

A Server-side Request Forgery (SSRF) vulnerability occurs when an attacker manipulates a server-side application into making HTTP requests to a domain of their choice. This vulnerability exposes the server to arbitrary external requests directed by the attacker.

**What is SSRF?**

Server-side request forgery is a web security vulnerability that allows an attacker to cause the server-side application to make requests to an unintended location.

In a typical SSRF attack, the attacker might cause the server to make a connection to internal-only services within the organization's infrastructure. In other cases, they may be able to force the server to connect to arbitrary external systems. This could leak sensitive data, such as authorization credentials.

**What is the impact of SSRF attacks?**

A successful SSRF attack can often result in unauthorized actions or access to data within the organization. This can be in the vulnerable application, or on other back-end systems that the application can communicate with. In some situations, the SSRF vulnerability might allow an attacker to perform arbitrary command execution.

An SSRF exploit that causes connections to external third-party systems might result in malicious onward attacks. These can appear to originate from the organization hosting the vulnerable application.

## Server-side Template Injection (SSTI)

**What is server-side template injection?**

Server-side template injection is when an attacker is able to use native template syntax to inject a malicious payload into a template, which is then executed server-side.

Template engines are designed to generate web pages by combining fixed templates with volatile data. Server-side template injection attacks can occur when user input is concatenated directly into a template, rather than passed in as data. This allows attackers to inject arbitrary template directives in order to manipulate the template engine, often enabling them to take complete control of the server. As the name suggests, server-side template injection payloads are delivered and evaluated server-side, potentially making them much more dangerous than a typical client-side template injection.

## Path Traversal

**What is path traversal?**

Path traversal is also known as directory traversal. These vulnerabilities enable an attacker to read arbitrary files on the server that is running an application. This might include:

Application code and data.

Credentials for back-end systems.

Sensitive operating system files.

In some cases, an attacker might be able to write to arbitrary files on the server, allowing them to modify application data or behavior, and ultimately take full control of the server.

## Cross-Site Scripting (XSS)

XSS attacks rely on injecting a malicious script in a benign website to run on a user’s browser. In other words, XSS attacks exploit the user’s trust in the vulnerable web application, hence the damage.

**Types of XSS**

1. **Reflected XSS**: This attack relies on the user-controlled input reflected to the user. For instance, if you search for a particular term and the resulting page displays the term you searched for (reflected), the attacker would try to embed a malicious script within the search term.
2. **Stored XSS**: This attack relies on the user input stored in the website’s database. For example, if users can write product reviews that are saved in a database (stored) and being displayed to other users, the attacker would try to insert a malicious script in their review so that it gets executed in the browsers of other users.
3. **DOM-based** XSS: This attack exploits vulnerabilities within the Document Object Model (DOM) to manipulate existing page elements without needing to be reflected or stored on the server. This vulnerability is the least common among the three.

**Causes and Implications**

Cross-site scripting (XSS) is a web security vulnerability that allows an attacker to inject malicious scripts into a web page viewed by other users. As a result, the unsuspecting users end up running the unauthorized script in their browsers, although the website they are visiting is trusted to be benign. Therefore, XSS can be a severe threat because it exploits users’ trust in a site.

**What Makes XSS Possible**

There are many reasons why XSS vulnerabilities are still found in web apps. Below, we list a few of them.

**1) Insufficient input validation and sanitization**

Web applications accept user data, e.g., via forms, and use this data in the dynamic generation of HTML pages. Consequently, malicious scripts can be embedded as part of the legitimate input and will eventually be executed by the browser unless adequately sanitized.

**2) Lack of output encoding**

The user can use various characters to alter how a web browser processes and displays a web page. For the HTML part, it is critical to properly encode characters such as <, >, ", ', and & into their respective HTML encoding. For JavaScript, special attention should be given to escape ', ", and \. Failing to encode user-supplied data correctly is a leading cause of XSS vulnerabilities.

**3) Improper use of security headers**

Various security headers can help mitigate XSS vulnerabilities. For example, Content Security Policy (CSP) mitigates XSS risks by defining which sources are trusted for executable scripts. A misconfigured CSP, such as overly permissive policies or the improper use of unsafe-inline or unsafe-eval directives, can make it easier for the attacker to execute their XSS payloads.

**4) Framework and language vulnerabilities**

Some older web frameworks did not provide security mechanisms against XSS; others have unpatched XSS vulnerabilities. Modern web frameworks automatically escape XSS by design and promptly patch any discovered vulnerability.

**5) Third-party libraries**

Integrating third-party libraries in a web application can introduce XSS vulnerabilities; even if the core web application is not vulnerable.

**Implications of XSS**

There are many implications of XSS. Below, we list a few of them.

**1) Session hijacking**

As XSS can be used to steal session cookies, attackers can take over the session and impersonate the victim if successful.

**2) Phishing and credential theft**

Leveraging XSS, attackers can present a fake login prompt to the user. In one recent case, the browser’s page was partially hidden by a dialogue box requesting users to connect to their cryptocurrency wallet.

**3) Social engineering**

Using XSS, an attacker can create a legitimate-looking pop-up or alert within a trusted website. This can trick users into clicking malicious links or visiting malicious websites.

**4) Content manipulation and defacement**

In addition to phishing and social engineering, an attacker might use XSS to change the website for other purposes, such as inflicting damage on the company’s reputation.

**5) Data exfiltration**

XSS can access and exfiltrate any information displayed on the user’s browser. This includes sensitive information such as personal data and financial information.

**6) Malware installation**

A sophisticated attacker can use XSS to spread malware. In particular, it can deliver drive-by download attacks on the vulnerable website.

**Reflected XSS** is a type of XSS vulnerability where a malicious script is reflected to the user’s browser, often via a **crafted URL** or **form submission**. Consider a search query containing <script>alert(document.cookie)</script>; many users wouldn’t be suspicious about such a URL, even if they look at it up close. If processed by a vulnerable web application, it will be executed within the context of the user’s browser.

One simple reflected XSS vulnerability is when the user searches for some term, and the search string is included verbatim in the results page. This simple scenario provides an easy target for the attacker to exploit.

Although discovering such vulnerabilities is not always easy, fixing them is straightforward. User input such as <script>alert('XSS')</script> should be santized or HTML-encoded to &lt;script&gt;alert('XSS')&lt;/script&gt;.

**Stored XSS**, or Persistent XSS, is a web application security vulnerability that occurs when the application stores user-supplied input and later embeds it in web pages served to other users without proper sanitization or escaping. Examples include web forum posts, product reviews, user comments, and other data stores. In other words, stored XSS takes place when user input is saved in a data store and later included in the web pages served to other users without adequate escaping.

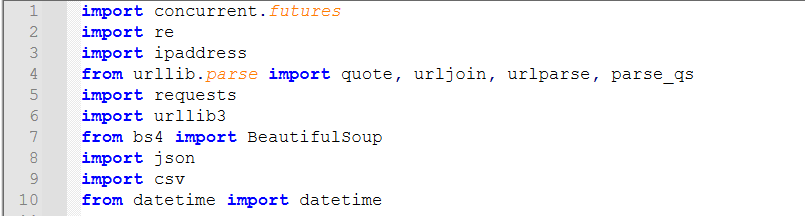
Stored XSS begins with an attacker injecting a malicious script in an input field of a vulnerable web application. The vulnerability might lie in how the web application processes the data in the comment box, forum post, or profile information section. When other users access this stored content, the injected malicious script executes within their browsers. The script can perform a wide range of actions, from stealing session cookies to performing actions on behalf of the user without their consent.

# Detection Methodologies

## SSRF Detection Technique

The SSRF Scanner is a multi-functional tool designed to detect Server-Side Request Forgery (SSRF) vulnerabilities. Key features include:

* **Content Fetching**: Retrieves HTML content from URLs.
* **HTML Parsing**: Extracts links, form actions, input values, and input names.
* **SSRF Testing**: Injects payloads to test for vulnerabilities.
* **Request Crafting**: Sends HTTP requests with custom headers and payloads (GET/POST).
* **OS Detection**: Identifies server OS via response headers (e.g., Server header).
* **Collaboration Support**: Detects out-of-band interactions using domains like Burp Collaborator.
* **Output Formats**: Generates results in JSON or CSV.
* **Efficiency:** Supports multithreading, URL lists, and proxy configurations



**concurrent . futures**  
Used for running tasks concurrently using threads or processes (helps with parallel execution).

**re**  
Provides support for **regular expressions**, allowing pattern matching and text searching.

**ipaddress**  
Allows manipulation and validation of **IPv4 and IPv6 addresses**.

**urllib.parse (quote, urljoin, urlparse, parse\_qs)**  
Tools for **manipulating and parsing URLs**, such as encoding, joining, and extracting components or query parameters.

**requests**  
A popular library for making **HTTP requests** (GET, POST, etc.) to communicate with web servers.

**urllib3**  
A powerful, low-level HTTP client, often used internally by libraries like requests.

**bs4 (BeautifulSoup)**  
Used for **parsing and extracting data from HTML or XML** documents (web scraping).

**json, csv**: Result formatting/output.

**datetime**  
Provides classes for working with **dates and times**, useful for timestamps and scheduling.

**Disable SSL Warnings**



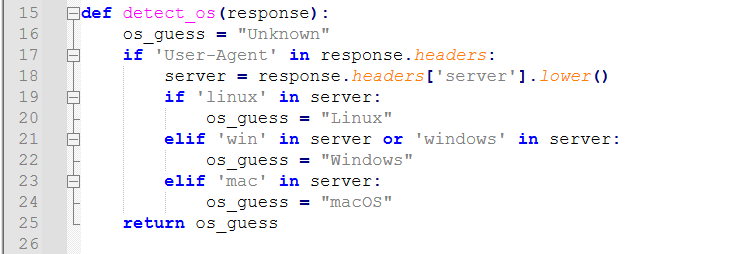
Suppresses SSL certificate warnings when making insecure HTTPS requests.

Global Variables



Stores scan results as a list of dictionaries containing details about successful SSRF attempts.

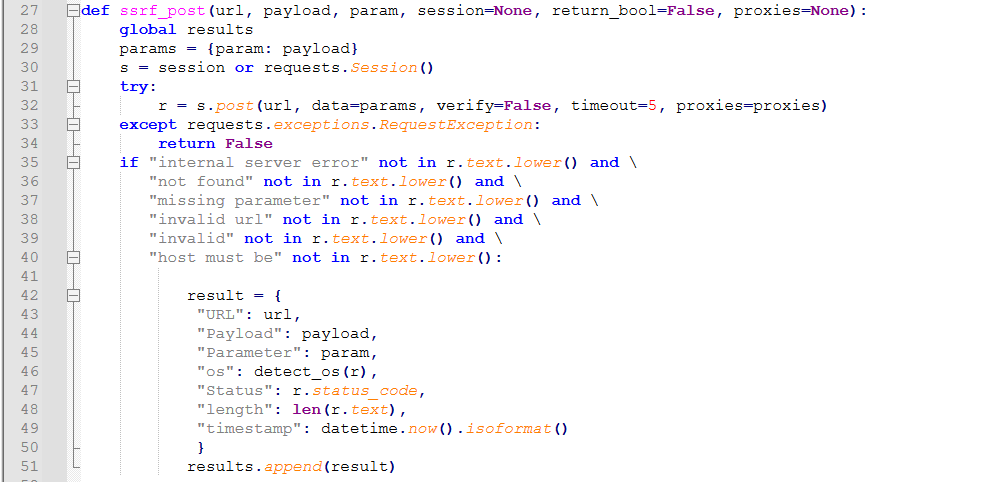
**Detection Functions**

1. detect\_os(response)  


Detect the operating system of the server by inspecting the Server header.

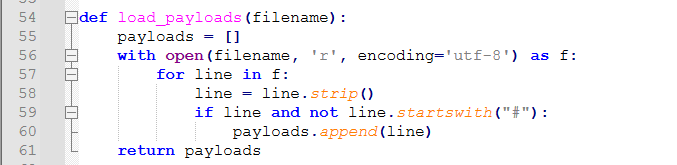
**Attack Functions**

1) ssrf\_post(url, payload, param, session=None, return\_bool=False, proxies=None)



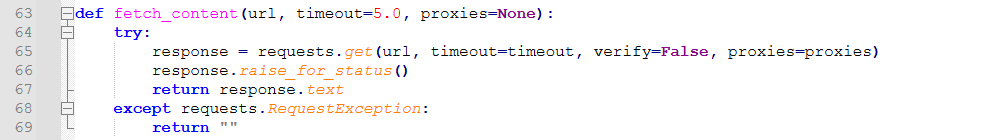
Performs an SSRF POST request with a given payload injected into a parameter.

2. load\_payloads(filename)



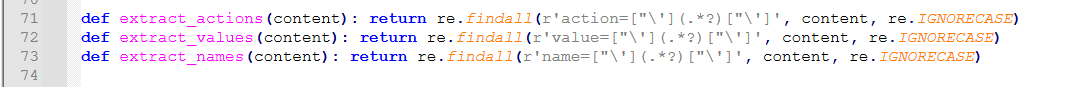
Loads payloads from a file (e.g., payload.txt) line by line.

3. fetch\_content(url, timeout=5.0, proxies=None)



Fetches page content using GET requests.

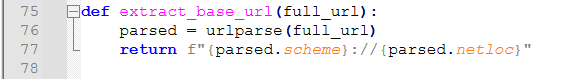
4. extract\_actions(content), extract\_values(content), extract\_names(content)



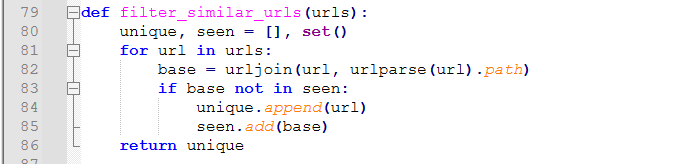
Use regex to extract:

Form action attributes, Input value attributes , Input name attributes Used to identify where to inject payloads.

5. extract\_base\_url(full\_url)

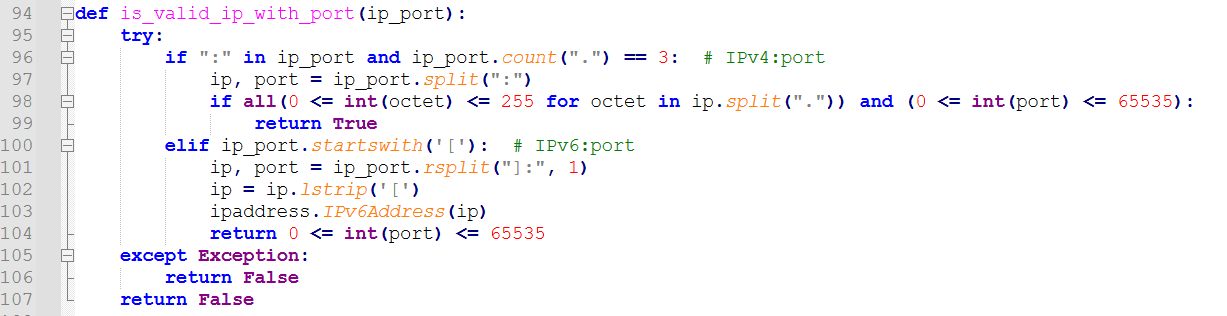


Returns the base URL (scheme + domain) of a full URL.  
6. filter\_similar\_urls(urls)

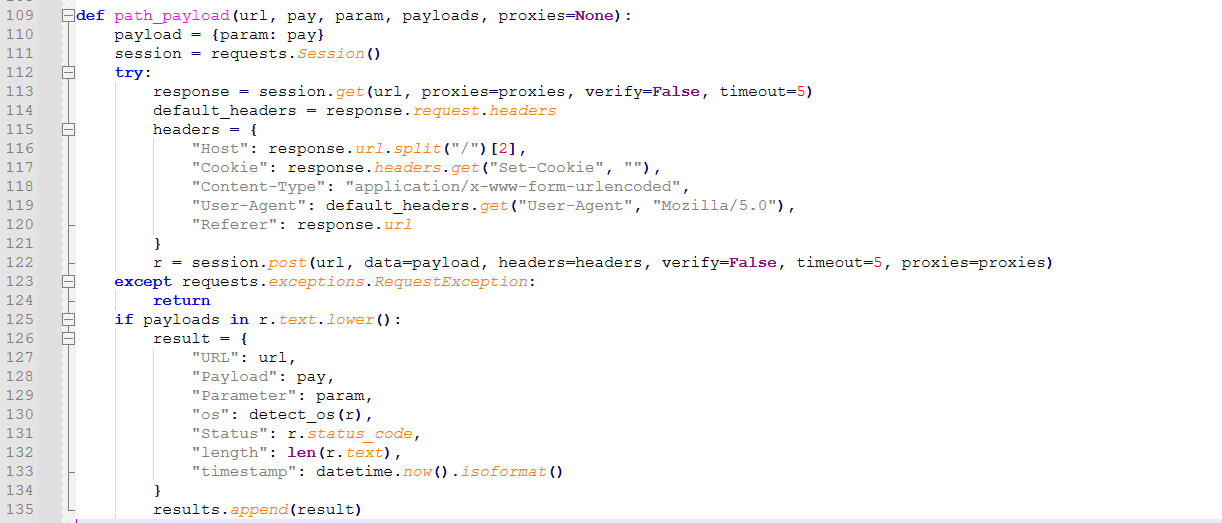


Filters duplicate URLs by comparing base paths.

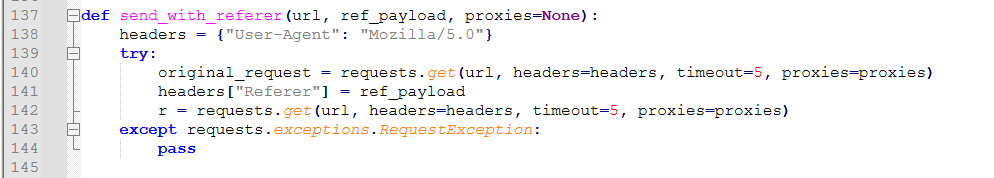
7. is\_valid\_ip\_with\_port(ip\_port)



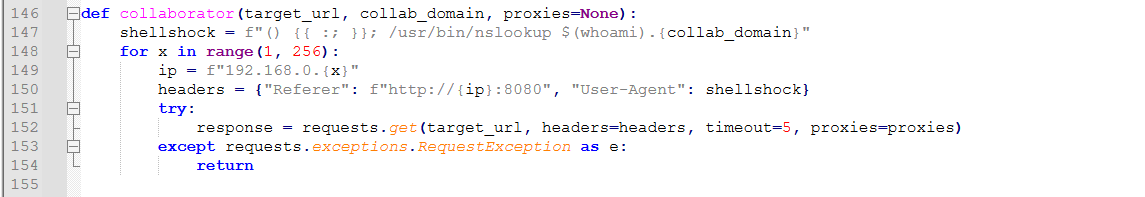
Validates IPv4 or IPv6 addresses with optional port numbers.  
8. path\_payload(...)



Injects payloads into the path of a URL and sends POST requests with custom headers.  
9. send\_with\_referer(...)

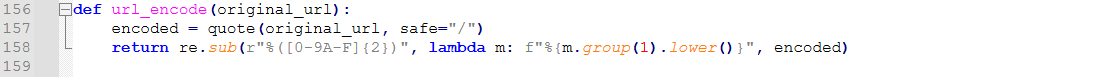


Sets a custom Referer header in a GET request.  
10. collaborator(target\_url, collab\_domain, proxies=None)



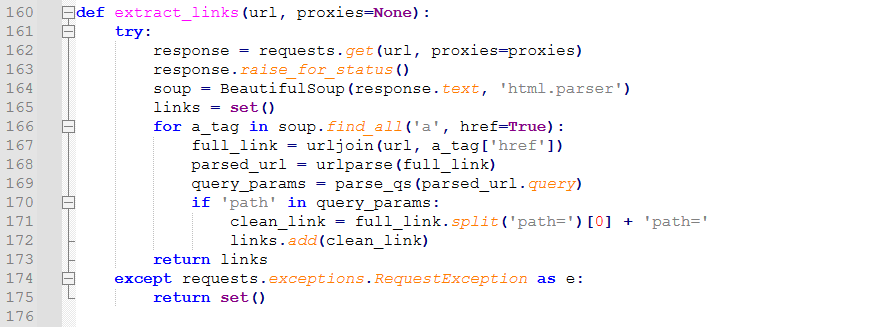
Attempts to trigger an external DNS lookup to a collaborator domain (e.g., Burp Collaborator).

11. url\_encode(original\_url)



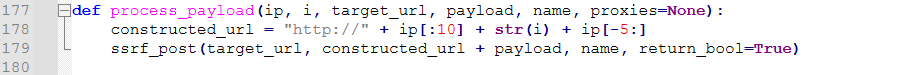
URL-encodes a string.

12. extract\_links(url, proxies=None)



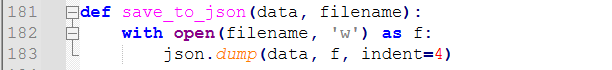
Uses BeautifulSoup to extract all <a href> links and looks for path= query parameters.

13.process\_payload(…)



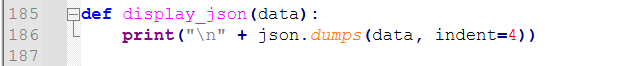
Takes an IP template and modifies the middle part with a number i to generate a new internal IP address and send it to ssrf\_post().(Bruteforce attack)

14. save\_to\_json(data, filename)



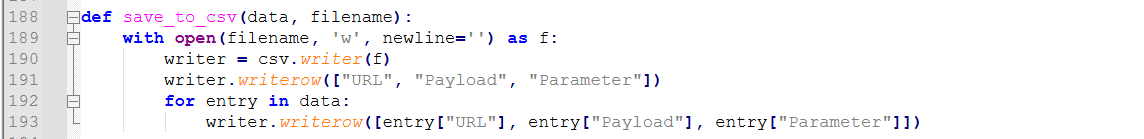
Saves results to a JSON file.

15. display\_json(data)



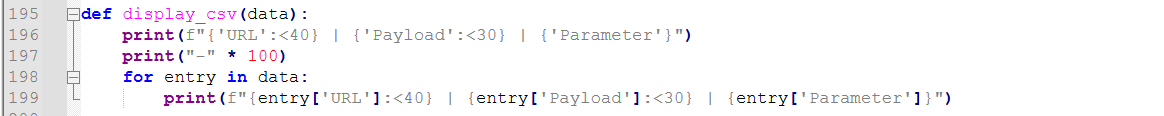
Prints JSON-formatted results to console.

16. save\_to\_csv(data, filename)



Saves results to a CSV file.

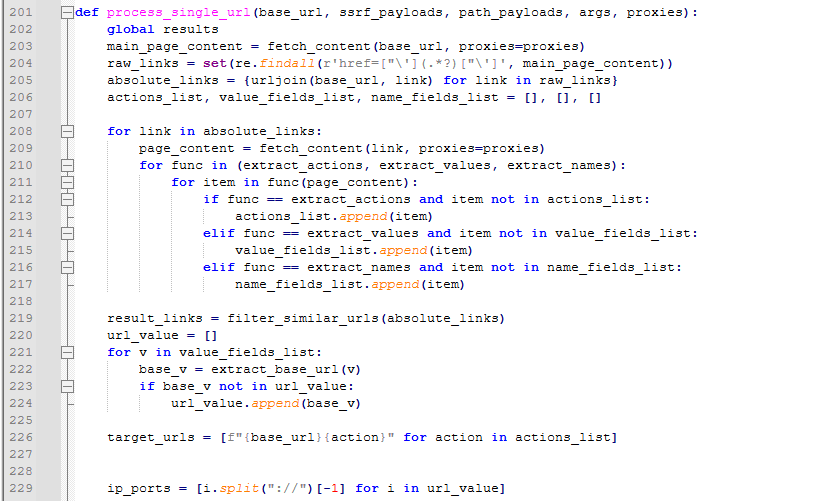
17. display\_csv(data)

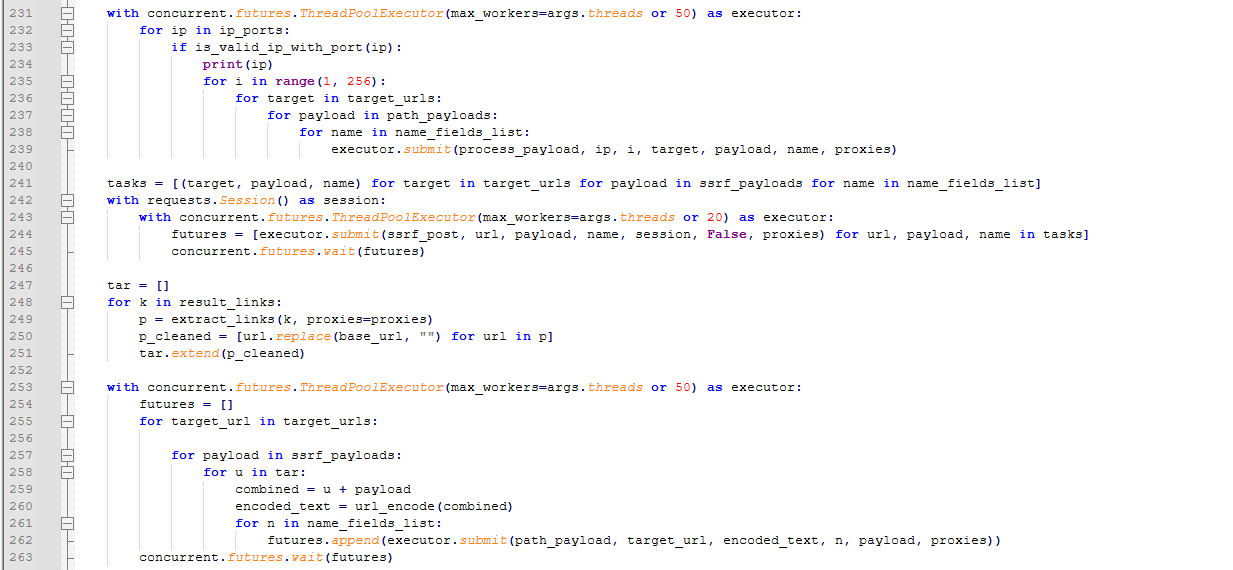


Prints CSV-style formatted results to console.

**Main Scanner Logic**

1. process\_single\_url(base\_url, ssrf\_payloads, path\_payloads, args, proxies)







This is the heart of one-target scanning:

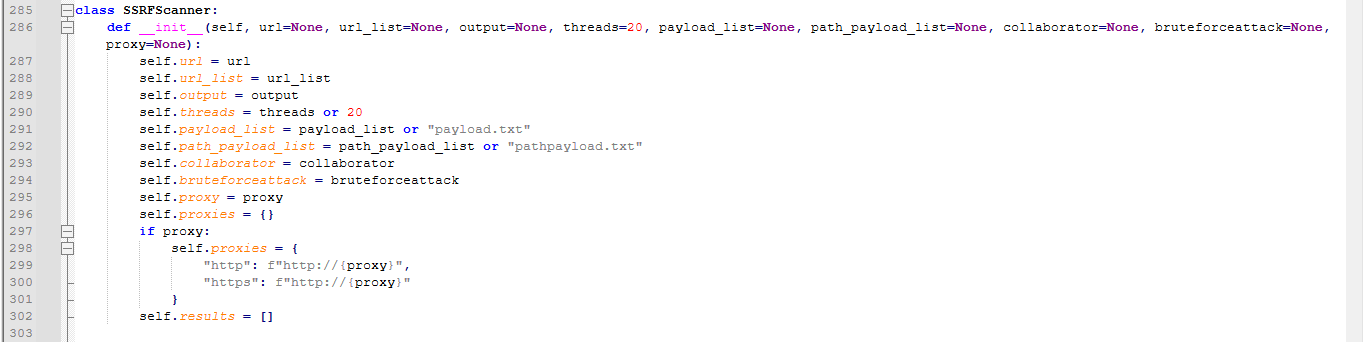
Crawl main page to collect all links (href=) and normalize to absolute URLs.

For each link:

1. Fetch page content.
2. Extract all action=, value=, name= fields.
3. Deduplicate similar URLs and base URLs.
4. Build list of SSRF targets by combining base URL + each extracted form action.
5. Brute‑force path-based SSRF
6. For each discovered hostname:port from URL values, try varying trailing numbers 1–255.
7. Concurrently submit path\_payload to each form endpoint.
8. Using all (form‑action, ssrf\_payload, param\_name) combinations, POST concurrently.
9. Chained SSRF via “path=”
10. Extract additional “path=” links from all result pages, append each SSRF payload, URL‑encode, re‑POST.
11. Blind SSRF (Collaborator) If configured, run send\_with\_referer and full collaborator scan.
12. Output Print or save to JSON/CSV, depending on CLI flags.

2. class SSRFScanner:

A) def \_\_init\_\_(….)

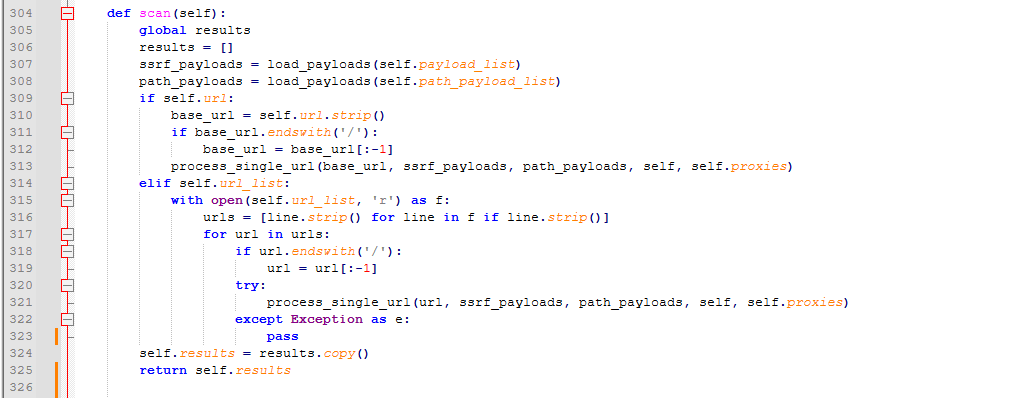


Initializes the scanner with:

Target URL or list

Payload files, Collaborator domain, Proxy settings, Thread count, Output format

B) scan(self)



Runs the scanner against each URL.

## SSTI Detection Technique

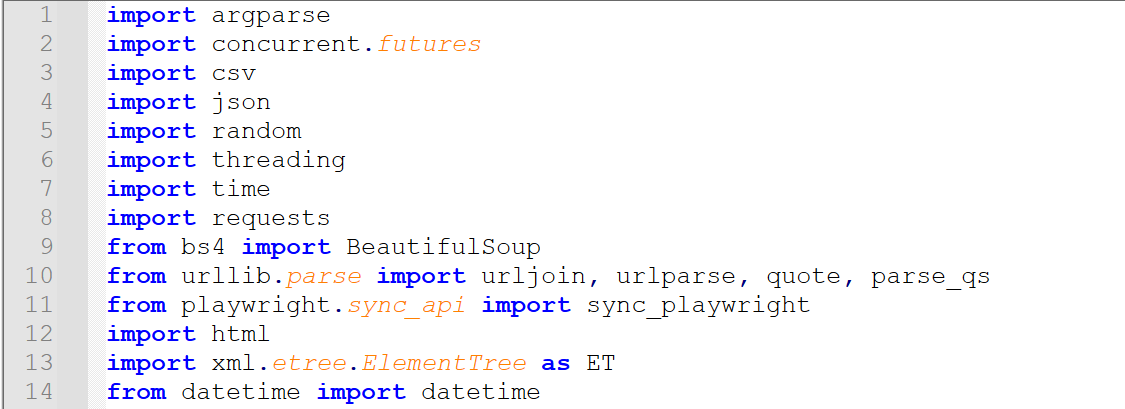
## Path Traversal Detection Technique

## XSS Detection Technique

The XSSHunter is an advanced tool designed to detect Cross-Site Scripting (XSS) vulnerabilities. Key features include:

* **Crawling**: Discovers and processes links, forms, and URL parameters on the target site.
* **Payload Injection**: Tests reflected, stored, DOM-based, and polyglot XSS using context-specific payloads.
* **Concurrent Execution**: Uses multithreading for efficient crawling and testing.
* **DOM Testing**: Leverages Playwright for browser-based DOM XSS detection.
* **Proxy Integration**: Supports Burp Suite proxy for traffic interception.
* **Output Formats**: Generates reports in JSON, CSV, or XML.
* **Rate Limiting**: Configurable delays between requests to avoid detection.

**Imports & Setup**



**argparse**  
Handles command-line argument parsing (target URL, workers, output format).

**concurrent.futures**  
Manages multithreading for parallel crawling and testing.

**requests**  
Sends HTTP requests to interact with the target application.

**bs4 (BeautifulSoup)**  
Parses HTML content to extract forms, links, and input fields.

**urllib.parse (urljoin, urlparse, quote, parse\_qs)**  
Manipulates URLs and encodes payloads.

**playwright**  
Automates Chromium for DOM XSS detection (e.g., hashchange, sink testing).

**json, csv, xml.etree.ElementTree**  
Formats and saves scan results.

**datetime**  
Adds timestamps to vulnerability reports.

**Key Methods**

1. get\_xss\_payloads(context)  
   Returns XSS payloads based on context:
   * reflected: Classic payloads (e.g., " onmouseover="alert(1)).
   * dom: Hash-based and JavaScript URI payloads.
   * polyglot: Multi-context payloads.
   * sink\_specific: Payloads targeting sinks like innerHTML or document.write.
2. crawl(url, depth)  
   Recursively crawls the target URL:
   * Extracts links and forms using BeautifulSoup.
   * Tests forms and URL parameters with payloads.
   * Limits depth to avoid infinite loops.
3. test\_form(form\_details, payload)  
   Injects payloads into form fields and submits requests. Detects:
   * Reflected XSS via response content.
   * Stored XSS using keywords like "thank you".
4. test\_url\_param(base\_url, param, payload)  
   Tests URL parameters by appending payloads (e.g., ?param=<script>alert(1)</script>).
5. detect\_vulnerabilities(response, payload, context, source)  
   Checks responses for:
   * Reflected payloads (encoded/decoded).
   * Event handlers (e.g., onload=, onmouseover=).
   * Stored XSS indicators.
6. run\_dom\_test(payload, test\_type, sink)  
   Uses Playwright to test DOM-based XSS:
   * Triggers hashchange events.
   * Injects payloads into JavaScript sinks.
   * Detects alert/print execution via browser automation.
7. report\_vulnerability(vuln\_type, payload, location)  
   Logs findings and stores them in vulnerabilities list.
8. export\_json()**,**export\_csv()**,**export\_xml()  
   Saves results to files in specified formats.

**Main Logic**

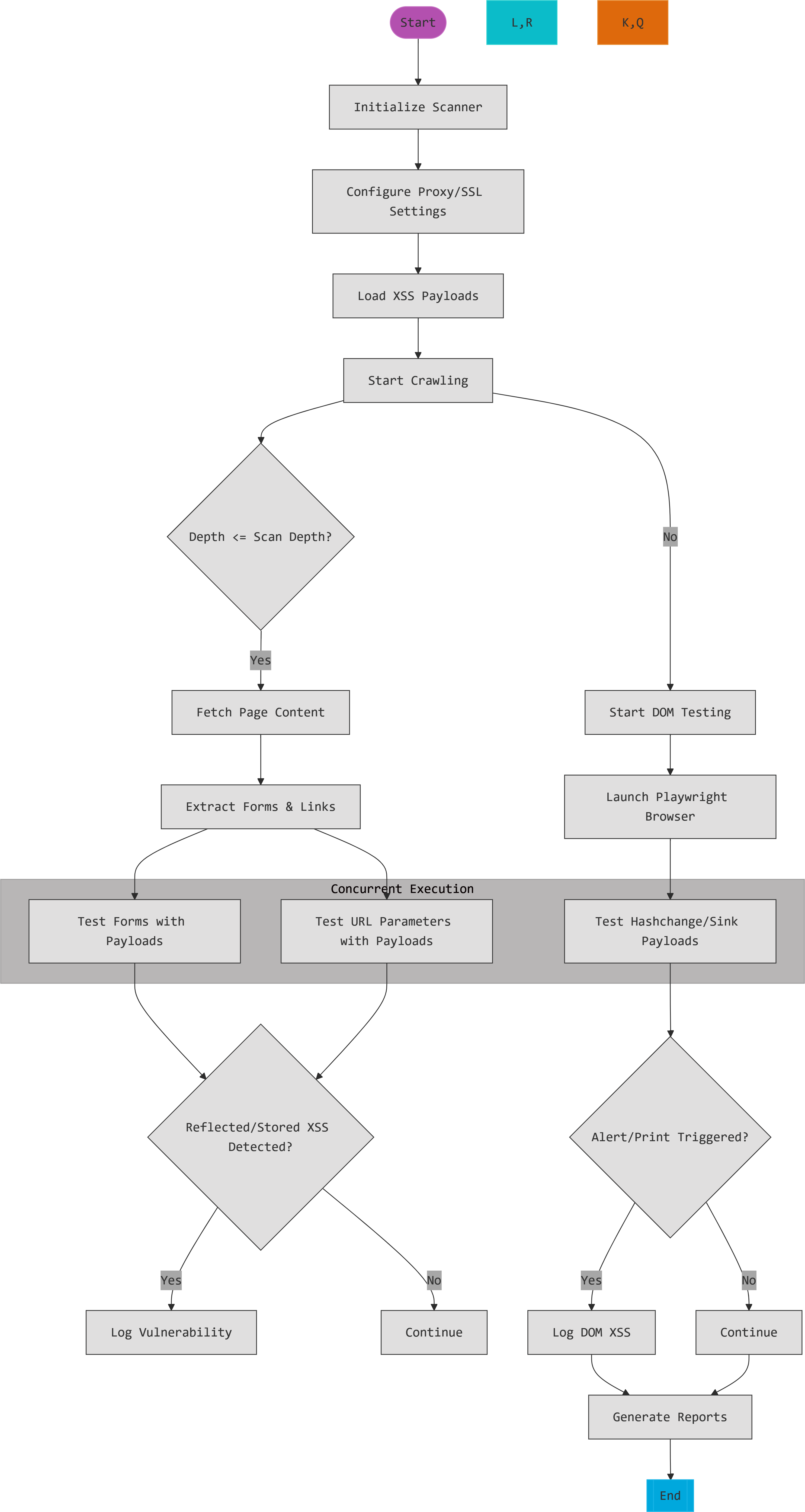
1. start\_scan()
   * Initiates crawling and DOM testing concurrently.
   * Uses ThreadPoolExecutor for parallel execution.
   * Triggers report generation after completion.
2. run\_dom\_tests()  
   Executes DOM tests in parallel:
   * Standard DOM payloads.
   * Sink-specific tests (e.g., innerHTML, hashchange).

**Output**

Results include:

* Vulnerability type (e.g., Reflected, DOM, Stored).
* Payload used.
* Location (URL or form action).
* Timestamp.

**Here’s a flowchart to visualize the workflow of the XSSHunter scanner.**

****

# Building the Vulnerability Scanner

## How the Scanner works?

## Choice of Programming and Libraries

## User Graphic interface (GUI)

## Comparisons with other vulnerability

# Conclusion

## Summary of Achievements

## Potential Enhancements

## References