

جامعة آل البيت

**Al-Al Bayt University**

**Web Vulnerability Scanner**

**Presented by:**

|  |  |
| --- | --- |
| Ahmad Ali Shwaiyat: | 2100906056 |
| Mohammad 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](#_Toc198285370)

[1.1 Project Problem 4](#_Toc198285371)

[1.2 Project Goals 4](#_Toc198285372)

[1.3 What is an automated web vulnerability ? 5](#_Toc198285373)

[1.4 Why are vulnerability scanners Important? 5](#_Toc198285374)

[1.5 What the project covers 6](#_Toc198285375)

[1.6 Beneficiaries 7](#_Toc198285376)

[Chapter 2: Overview of Target vulnerabilities 8](#_Toc198285377)

[2.1 Server-side Request Forgery (SSRF) 8](#_Toc198285378)

[2.2 Server-side Template Injection (SSTI) 8](#_Toc198285379)

[2.3 Path Traversal 9](#_Toc198285380)

[2.4 Cross-Site Scripting (XSS) 9](#_Toc198285381)

[Chapter 3: Detection Methodologies 13](#_Toc198285382)

[3.1 SSRF Detection Technique 13](#_Toc198285383)

[3.2 SSTI Detection Technique 24](#_Toc198285384)

[3.3 Path Traversal Detection Technique 24](#_Toc198285385)

[3.4 XSS Detection Technique 25](#_Toc198285386)

[Chapter 4: Building the Vulnerability Scanner 30](#_Toc198285387)

[4.1 How the Scanner works? 30](#_Toc198285388)

[**4.1.1** SSRF 30](#_Toc198285389)

[**4.1.2** **SSTI** 30](#_Toc198285390)

[4.2 Choice of Programming and Libraries 30](#_Toc198285391)

[4.3 User Graphic interface (GUI) 30](#_Toc198285392)

[4.4 Comparisons with other vulnerability 30](#_Toc198285393)

[Chapter 5: Conclusion 30](#_Toc198285394)

[5.1 Summary of Achievements 30](#_Toc198285395)

[5.2 Potential Enhancements 30](#_Toc198285396)

[5.3 References 30](#_Toc198285397)

# Introduction

## Project Problem

Because of vulnerabilities like SQL injection, cross-site scripting (XSS), unsecured APIs, and incorrectly setup servers, cyberattacks are increasingly targeting modern web applications. Manual vulnerability identification takes a lot of effort, is prone to mistakes, and calls for specific knowledge. Many businesses, particularly small-to-medium-sized businesses (SMEs), lack the funding necessary to put strong security procedures in place, leaving their systems vulnerable to intrusions. In order to proactively identify and reduce risks, this project tackles the urgent need for an automated, easily available, and effective web vulnerability scanner.

## Project Goals

The main goals of this project are:

1. Create an automated web vulnerability scanner that finds and reports common security flaws.
2. To enhance web applications' security posture by making it possible to identify and fix problems early.
3. To offer an easy-to-use solution that companies, security teams, and developers of all technical backgrounds can employ.

## What is an automated web vulnerability ?

An automated online vulnerability scanner is a software program that combines pre-established rules, machine learning (ML), and simulated attack patterns to systematically find security flaws in servers, web apps, and APIs. By doing away with human penetration testing, it makes security assessments quick, scalable, and repeatable.

## Why are vulnerability scanners Important?

1. Cost Efficiency: Lower the costs related to manual security audits.
2. Proactive Defense: Find weaknesses before they are used by attackers.
3. Compliance: Meet Adhere to legal requirements (such as GDPR and PCI-DSS).
4. Reputation Protection: : Avoid data breaches that undermine corporate confidence..
5. Continuous Monitoring: : Make DevOps pipelines (shift-left security) capable of real-time scanning.

## What the project covers

Path Traversal

* Definition: Gains access to unauthorized files (such as /../../etc./passwd) by taking advantage of inadequate input sanitization
* Impact: System breach and data theft.
* Detection: Check server responses for file disclosures by injecting traversal sequences (such as../, %2e%2e%2f).

Cross-site scripting (XSS):

* Definition: : Inserts harmful scripts, such as (e.g., <script>alert(1)</script>), into web pages.
* Types include DOM-based (client-side), stored (permanent), and reflected (URL-based).
* Detection: In HTML/JS contexts, submit payloads and look for unencoded output.

Server-Side Request Forgery(SSRF)

* Definition: Forces a server to make unauthorized internal requests (e.g., to AWS metadata endpoints).
* Impact: Theft of cloud credentials and internal network reconnaissance.
* Detection: Send internal IP-based URLs (such as http://169.254.169.254) and track the replies.

1.5.4 Server-Side Template Injection (SSTI)

* Definition: inserting malicious code into templating engines, such as Jinja2 and Smarty.
* Impact: Data breaches and remote code execution (RCE).
* Detection: Use template syntax to test (for example, {{7\*7}} → 49 indicates vulnerability).

## Beneficiaries

* Developers: Make sure that CI/CD pipelines incorporate security.
* Penetration testers: Quickly identify vulnerabilities.
* Organizations: Lower audit expenses and breach risks.
* End Users: Protect private information from fraud.

# Overview of Target vulnerabilities

## Server-side Request Forgery (SSRF)

When an attacker tricks a server-side application into sending HTTP requests to a desired domain, this is known as a server-side request forgery (SSRF) vulnerability. Due to this vulnerability, the attacker can send arbitrary external requests to the server..

**What is SSRF?**

A web security flaw known as "server-side request forgery" enables a hacker to make requests to an unauthorized location from a server-side application.   
  
The attacker may force the server to connect to internal-only services within the architecture of the company in a standard SSRF attack. In other situations, they might have the power to compel the server to establish connections with any external systems. Sensitive information, including authorization credentials, may be exposed.

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

Unauthorized acts or access to data within the business are frequently the outcome of a successful SSRF attack. This may occur on other back-end systems that the application can connect to or within the susceptible application itself. The SSRF vulnerability may occasionally give an attacker the ability to execute commands arbitrarily.   
  
Malicious follow-on attacks may arise from an SSRF exploit that establishes links to external third-party systems. These can appear to originate from the entity hosting the vulnerable application.

## Server-side Template Injection (SSTI)

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

The ability of an attacker to insert a malicious payload into a template using native template syntax and have it run server-side is known as server-side template injection.   
  
The purpose of template engines is to create web pages by fusing dynamic input with set templates. When user input is concatenated straight into a template instead of being sent in as data, server-side template injection attacks may be launched. This gives attackers the ability to alter the template engine by injecting arbitrary template directives, which frequently gives them total control over the server. Compared to a standard client-side template injection, server-side template injection payloads are potentially far more harmful because they are supplied and analyzed server-side, as the name implies.

## Path Traversal

**What is path traversal?**

Directory traversal is another name for path traversal. These flaws give an attacker the ability to read any file on the server hosting an application. This could consist of:   
  
Data and application code.   
Back-end system credentials.   
operating system files that are sensitive.   
An attacker may occasionally be able to write to any file on the server, changing application data or behavior and eventually gaining complete control of the system.

## Cross-Site Scripting (XSS)

In order for XSS attacks to function on a user's browser, a malicious script must be injected into a benign website. To put it another way, XSS attacks cause harm by taking advantage of the user's faith in the weak web application

**Types of XSS**

1. **Reflected XSS**: This attack depends on the user seeing their own input that they have manipulated. An attacker would attempt to insert a malicious script into the search term, for example, if you type in a phrase and the page that results shows the term you typed in (reflected).
2. **Stored XSS**: This attack uses user input that is kept in the database of the website. For instance, an attacker might attempt to include a malicious script in a user's product review so that it runs in other users' browsers if users are able to write reviews that are saved in a database (stored) and visible to other users.
3. **DOM-based** XSS: This attack modifies pre-existing page components without requiring them to be reflected or saved on the server by taking advantage of flaws in the Document Object Model (DOM). Of the three vulnerabilities, this one occurs the least frequently.

**Causes and Implications**

A web security flaw known as cross-site scripting (XSS) enables a hacker to insert harmful code into a page that other users are viewing. As a result, even if the website they are visiting is thought to be harmless, the unwary users wind up running the illegal script in their browsers. Therefore, because it preys on users' trust in a website, XSS might pose a serious threat.

**What Makes XSS Possible**

Web apps continue to have XSS vulnerabilities for a variety of reasons. We list some of them below.

**1) Inadequate input sanitization and validation**   
Web apps use user data—such as that obtained via forms—to dynamically generate HTML pages. As a result, dangerous scripts might be included in the legitimate input and, if the browser is not sufficiently cleaned, will eventually be executed.

**2) lack of output encoding**

To change how a web browser interprets and presents a page, the user can utilize different characters. Characters like <, >, ", ', and & must be correctly encoded into their corresponding HTML encoding for the HTML portion. JavaScript requires extra care with escape characters ', ", and \. One of the main causes of XSS vulnerabilities is incorrect encoding of user-supplied data.

**3)Improper use of security headers**

XSS vulnerabilities can be lessened with the use of different security headers. For instance, by specifying which sources are reliable for executable scripts, the Content Security Policy (CSP) reduces the possibility of cross-site scripting attacks. An attacker may find it simpler to carry out their XSS payloads if the CSP is configured incorrectly, for example, by using unsafe-inline or unsafe-eval directives incorrectly or by having excessively permissive rules.

**4) Framework and language vulnerabilities**

Some ancient web frameworks have unpatched XSS vulnerabilities, while others lacked XSS security measures. Contemporary web frameworks are built to automatically avoid XSS and quickly fix any vulnerabilities that are found

**5) Third-party libraries**

Even if the main web application is not susceptible, adding third-party libraries can result in XSS vulnerabilities.

**Implications of XSS**

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

**1)Session hijacking**

If successful, attackers can take over the session and assume the identity of the victim because XSS can be used to steal session cookies.

**2)Phishing and credential theft**

Attackers can employ XSS to provide the user a phony login prompt. In one recent instance, a dialogue box asking users to login to their cryptocurrency wallet partially obscured the browser's content.

**3) Social engineering**

An attacker can use XSS to make a pop-up or alert that appears authentic on a reliable website. This may deceive visitors into visiting malicious websites or clicking on harmful links.

**4) Content manipulation and defacement**

An attacker may use XSS to alter the website for additional objectives, such as harming the company's reputation, in addition to phishing and social engineering.

**5) Data exfiltration**

Any data that is visible in the user's browser can be accessed and exfiltrated by XSS. Sensitive data including financial and personal information are included in this.

**6) Malware installation**

XSS can be used to distribute malware by a skilled attacker. It is specifically capable of delivering drive-by download attacks on the website that is at risk.

A malicious script that is reflected to the user's browser, frequently through a contrived URL or form submission, is known as a "reflected XSS" vulnerability. Think considering a search query that has <script>alert(document.cookie)</script>; even if they were to examine it closely, many users wouldn't be wary of such a URL. It will be run within the user's browser context if it is processed by a web application that is susceptible.

When a user searches for a term and the search string appears exactly as it is on the results page, this is a straightforward reflected XSS issue. The attacker can easily exploit this straightforward situation.   
Finding these vulnerabilities can be challenging, but addressing them is simple. User input like <script>alert('XSS')</script> needs to be HTML-encoded or sanitized to &lt;script&gt;alert('XSS')&lt;/script&gt;..

A online application security flaw known as stored XSS, or persistent XSS, arises when the program saves user-supplied information and then incorporates it into other users' web pages without properly sanitizing or escaping it. User comments, product reviews, web forum discussions, and other data warehouses are a few examples. To put it another way, stored XSS occurs when user input is preserved in a data store and then used in websites that are seen by other users without sufficient escaping.

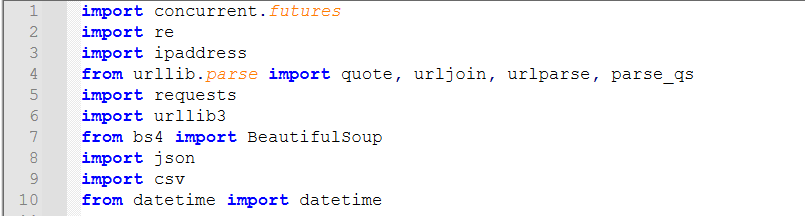
An attacker starts stored cross-site scripting (XSS) by inserting a malicious script into a web application's input field. The way the online application handles the data in the forum post, profile information area, or comment box may be the weak point. The malicious script that was injected runs in other users' browsers when they view this cached material. The script can carry out a variety of tasks, including as stealing session cookies and acting on the user's behalf without getting their permission.

# 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**



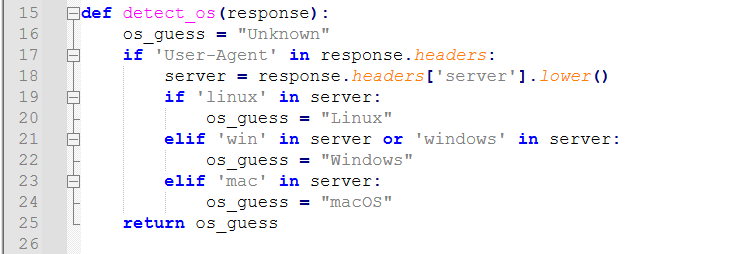
Suppresses SSL certificate warnings when making insecure HTTPS requests.

Global Variables



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

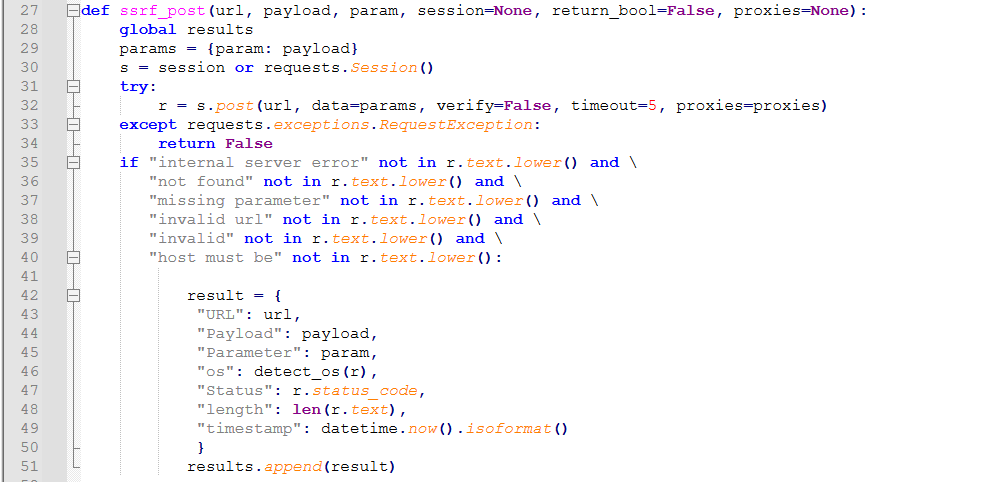
**Detection Functions**

1. detect\_os(response)  


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

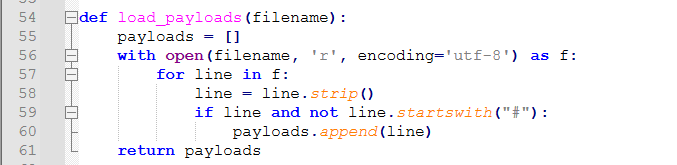
**Attack Functions**

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



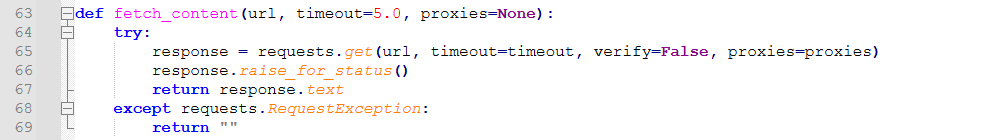
Performs an SSRF POST request with a given payload injected into a parameter.

2. load\_payloads(filename)



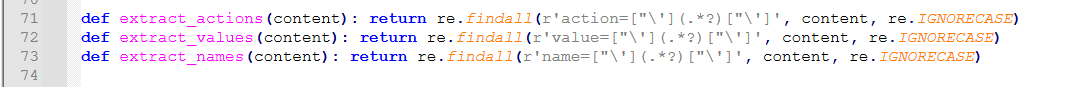
Loads payloads from a file (e.g., payload.txt) line by line.

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



Fetches page content using GET requests.

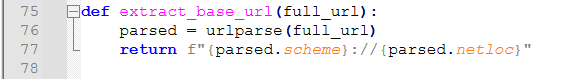
4. extract\_actions(content), extract\_values(content), extract\_names(content)



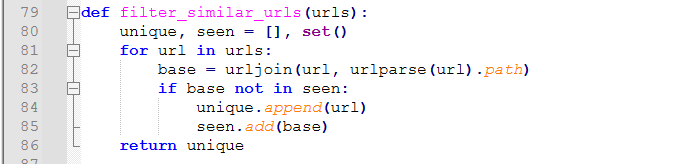
Use regex to extract:

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

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

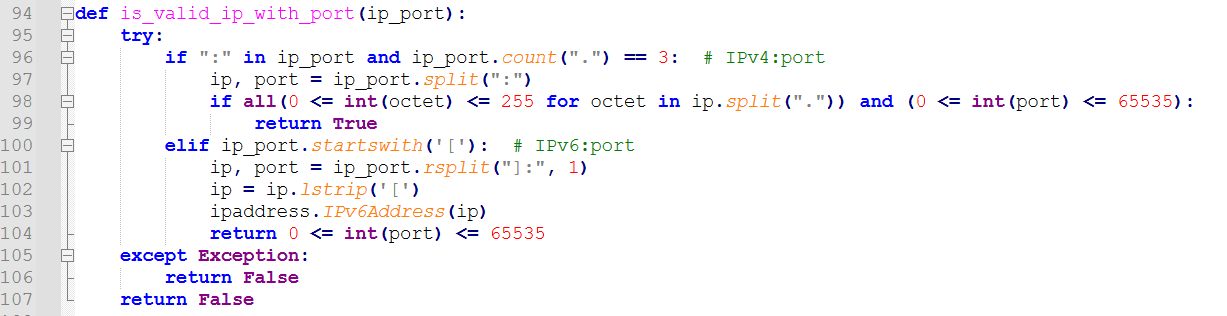


Returns the base URL (scheme + domain) of a full URL.  
6. filter\_similar\_urls(urls)

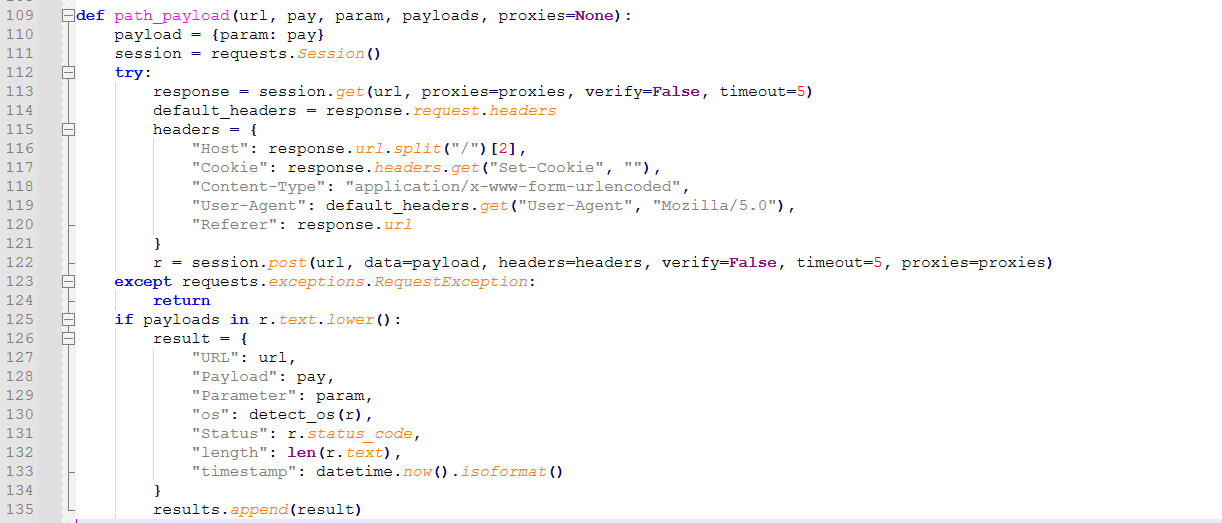


Filters duplicate URLs by comparing base paths.

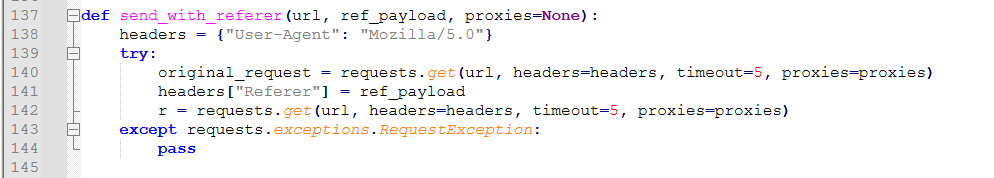
7. is\_valid\_ip\_with\_port(ip\_port)



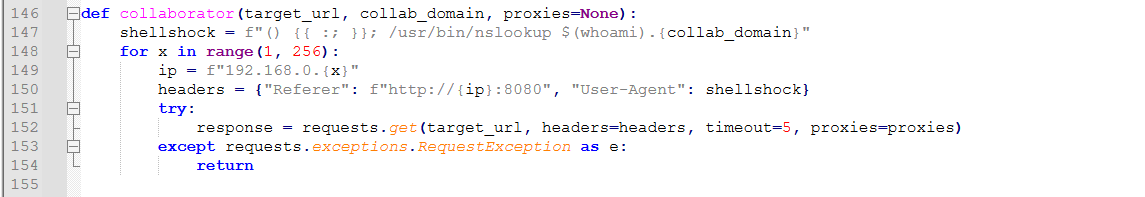
Validates IPv4 or IPv6 addresses with optional port numbers.  
8. path\_payload(...)



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

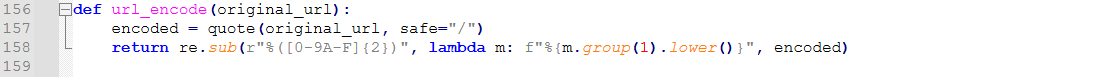


Sets a custom Referer header in a GET request.  
10. collaborator(target\_url, collab\_domain, proxies=None)



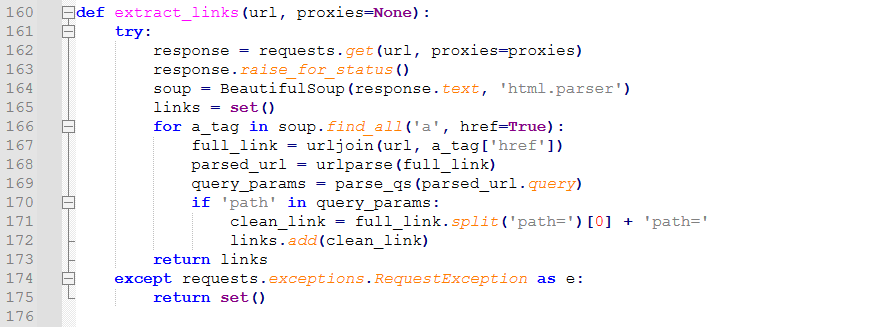
Attempts to trigger an external DNS lookup to a collaborator domain (e.g., Burp Collaborator).

11. url\_encode(original\_url)



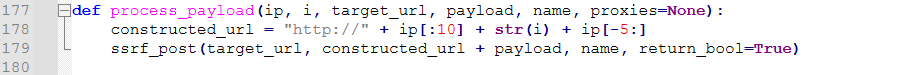
URL-encodes a string.

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



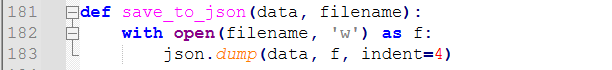
Uses BeautifulSoup to extract all <a href> links and looks for path= query parameters.

13.process\_payload(…)



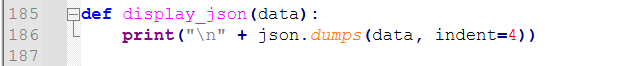
Takes an IP template and modifies the middle part with a number i to generate a new internal IP address and send it to ssrf\_post().(Bruteforce attack)

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



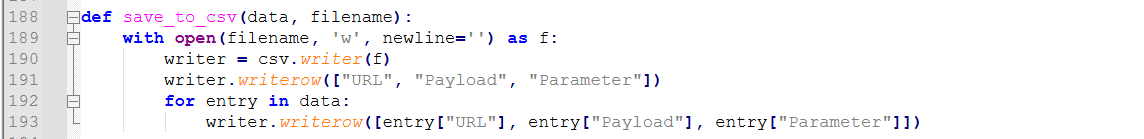
Saves results to a JSON file.

15. display\_json(data)



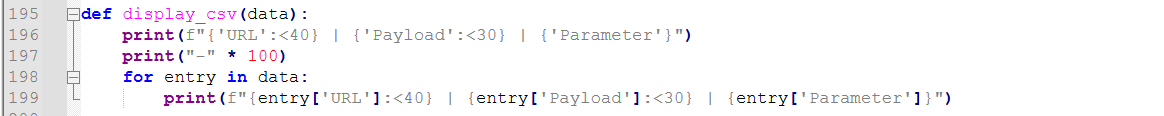
Prints JSON-formatted results to console.

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



Saves results to a CSV file.

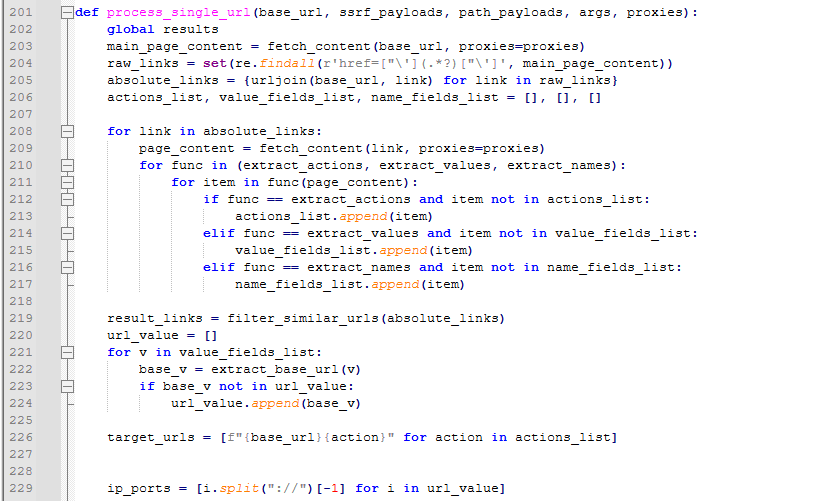
17. display\_csv(data)

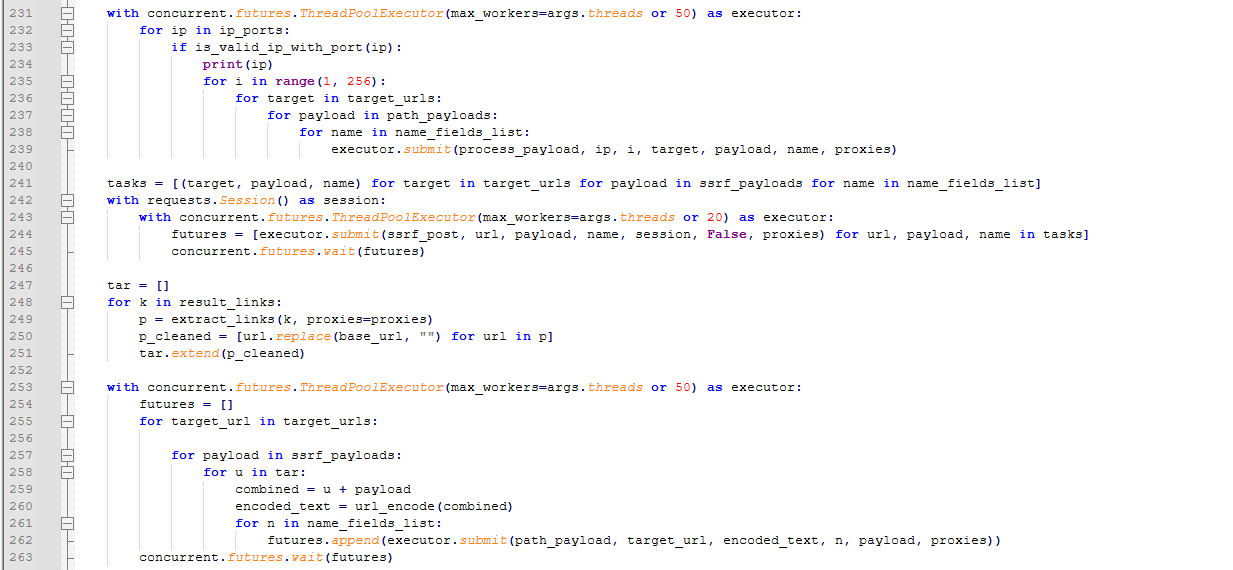


Prints CSV-style formatted results to console.

**Main Scanner Logic**

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







This is the heart of one-target scanning:

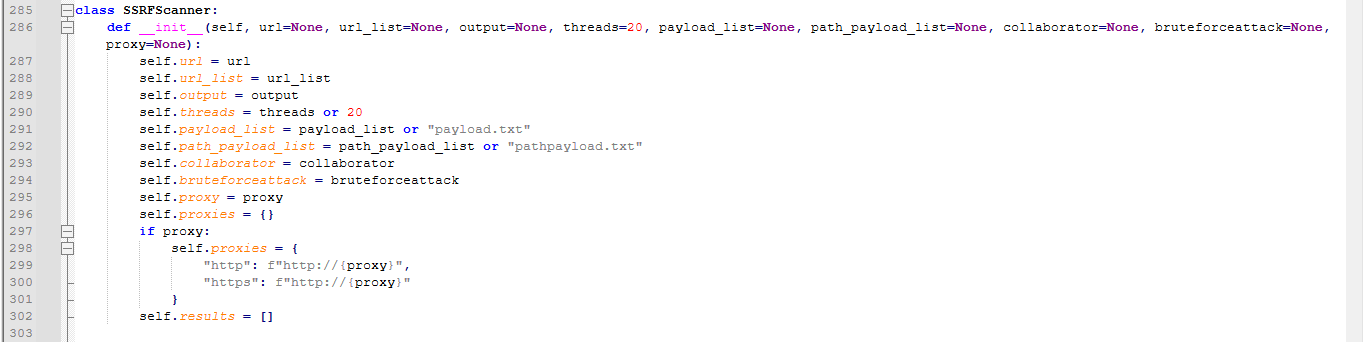
Crawl main page to collect all links (href=) and normalize to absolute URLs.

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\_\_(….)

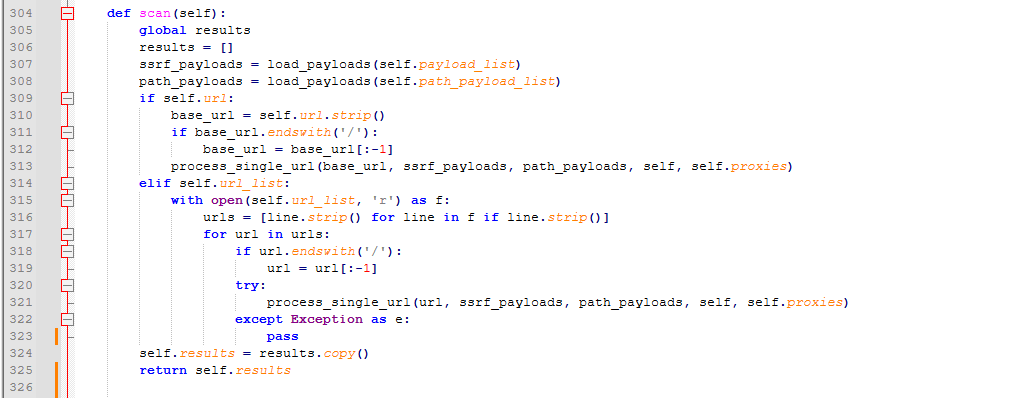


Initializes the scanner with:

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

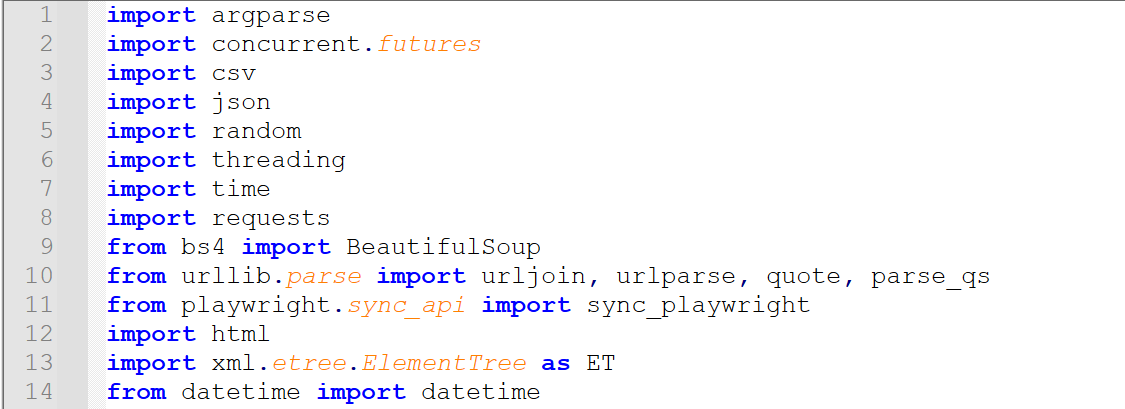
## Path Traversal Detection Technique

## 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**

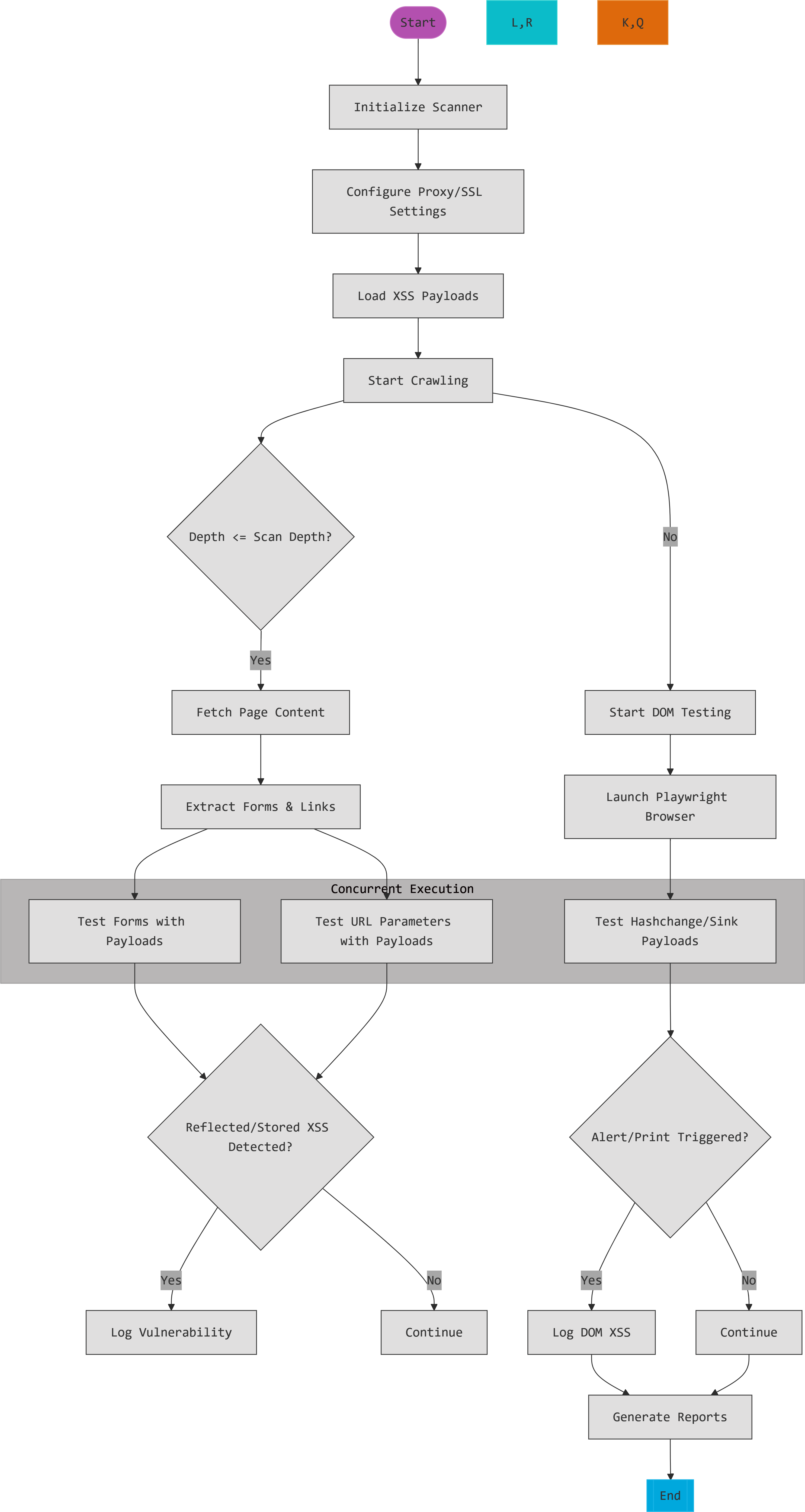
1. start\_scan()
   * Initiates crawling and DOM testing concurrently.
   * Uses ThreadPoolExecutor for parallel execution.
   * Triggers report generation after completion.
2. run\_dom\_tests()  
   Executes DOM tests in parallel:
   * Standard DOM payloads.
   * Sink-specific tests (e.g., innerHTML,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

**Output**

Results include:

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

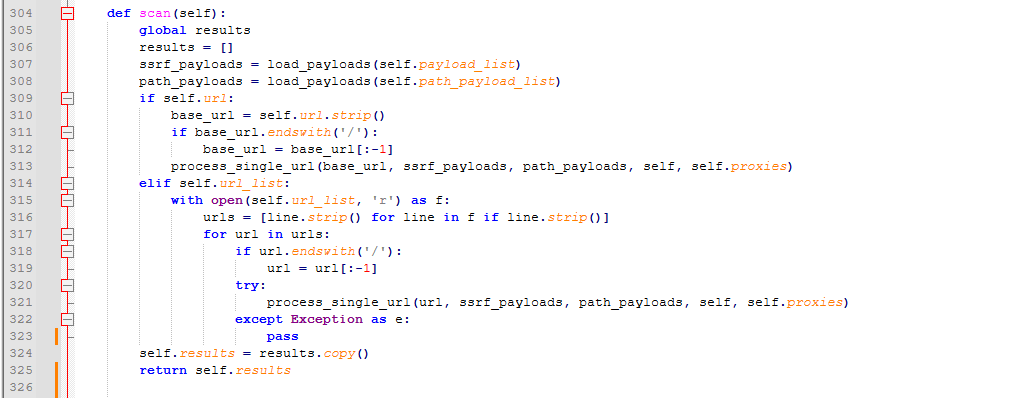
**Here’s a flowchart to visualize the workflow of the XSSHunter scanner.**

****

# Building the Vulnerability Scanner

## How the Scanner works?

### SSRF



### **SSTI**

#### Architectural Foundation

1. Core Components

System Building Blocks

| **Component** | **Code Reference** | **Purpose** |
| --- | --- | --- |
| Main Scanner Class | SSTIScanner | Coordinates scanning workflow |
| Parameter Discovery | SiteCrawler | Finds input fields/URL parameters |
| Vulnerability Detectors | ErrorBasedEngineDetector + EvaluationBasedEngineDetector | Confirms SSTI existence |
| Exploitation System | SSTIExploiter | Executes attacks post-detection |
| Payload Repository | TemplateDB | Stores engine-specific payloads |
| Scenario System | ScenarioHandler | Handles special cases/lab environments |

**Technology Stack**

*Critical dependencies and their roles in the system:*

| **Library** | **Purpose** | **Usage Context** |
| --- | --- | --- |
| requests | HTTP request handling with session persistence | Target interaction, vulnerability probing |
| BeautifulSoup (bs4) | HTML/XML parsing for parameter discovery | Site crawling, form extraction |
| argparse | Command-line interface (CLI) configuration | User input handling |
| concurrent.futures | Parallel execution of scanning tasks | Multi-threaded parameter testing |
| re | Regex pattern matching | Error analysis, payload detection |
| urllib3 | Low-level HTTP client with SSL/TLS controls | Proxy configuration, warning suppression |
| json/csv | Report generation in structured formats | Output serialization |
| datetime | Timestamping of findings | Report metadata |

**Key Library Explanations**

**requests**

Manages all HTTP(S) communications

Handles cookies, redirects, and proxy configurations

Provides timeout controls for safe scanning

**BeautifulSoup**

Analyzes HTML structure during site crawling phase

Extracts form parameters and hidden inputs

Identifies reflection points in rendered content

**concurrent.futures.ThreadPoolExecutor**

Enables parallel scanning of multiple parameters

Manages thread pool for reflection tests (Phase 2)

Accelerates engine detection (Phase 3)

**argparse**

Processes command-line arguments (URLs, output format, etc)

Implements help system and input validation

Configures proxy settings and thread counts

**re**

Identifies template engine fingerprints in error messages

Extracts executed command output between TINJ\_\* markers

Validates parameter reflection through pattern matching

2. Template Payload Database

*Centralized repository for vulnerability patterns and exploit templates*

**Key Roles**:

Stores engine fingerprints (regex patterns)

Contains validation payloads (e.g., {{7\*7}})

Provides exploit templates (command/file access)

Documents engine-specific constraints

**Critical Integration Points**:

Used by ErrorBasedEngineDetector and EvaluationBasedEngineDetector

Accessed by SSTIExploiter during attack execution

Version-controlled via template\_db.json

Schema Overview

{  
 "engines": {  
 "EngineName": {  
 "detection": {  
 "error\_regex": ["Pattern1", "Pattern2"],  
 "evaluation": {  
 "payload": "TEST\_EXPRESSION",  
 "expected": "EXPECTED\_RESULT"  
 },  
 "reflection\_markers": ["DELIMITERS"]  
 },  
 "exploit": {  
 "command\_exec": {  
 "payload": "EXPLOIT\_TEMPLATE",  
 "description": "Usage context"  
 },  
 "file\_read": {  
 "payload": "FILE\_ACCESS\_TEMPLATE"  
 }  
 }  
 }  
 }  
}

Key Components

**Detection Signatures**

error\_regex: Patterns to identify engines from error messages

evaluation: Mathematical tests to confirm template execution

reflection\_markers: Character sequences that indicate template context

**Exploit Templates**

command\_exec: OS command injection payloads

file\_read: File system access patterns

reverse\_shell: Pre-built connection templates

Integration with Scanning Workflow

**Detection Phase**

# ErrorBasedEngineDetector uses error\_regex  
if re.search(pattern, response.text):  
 identify\_engine()

**Verification Phase**

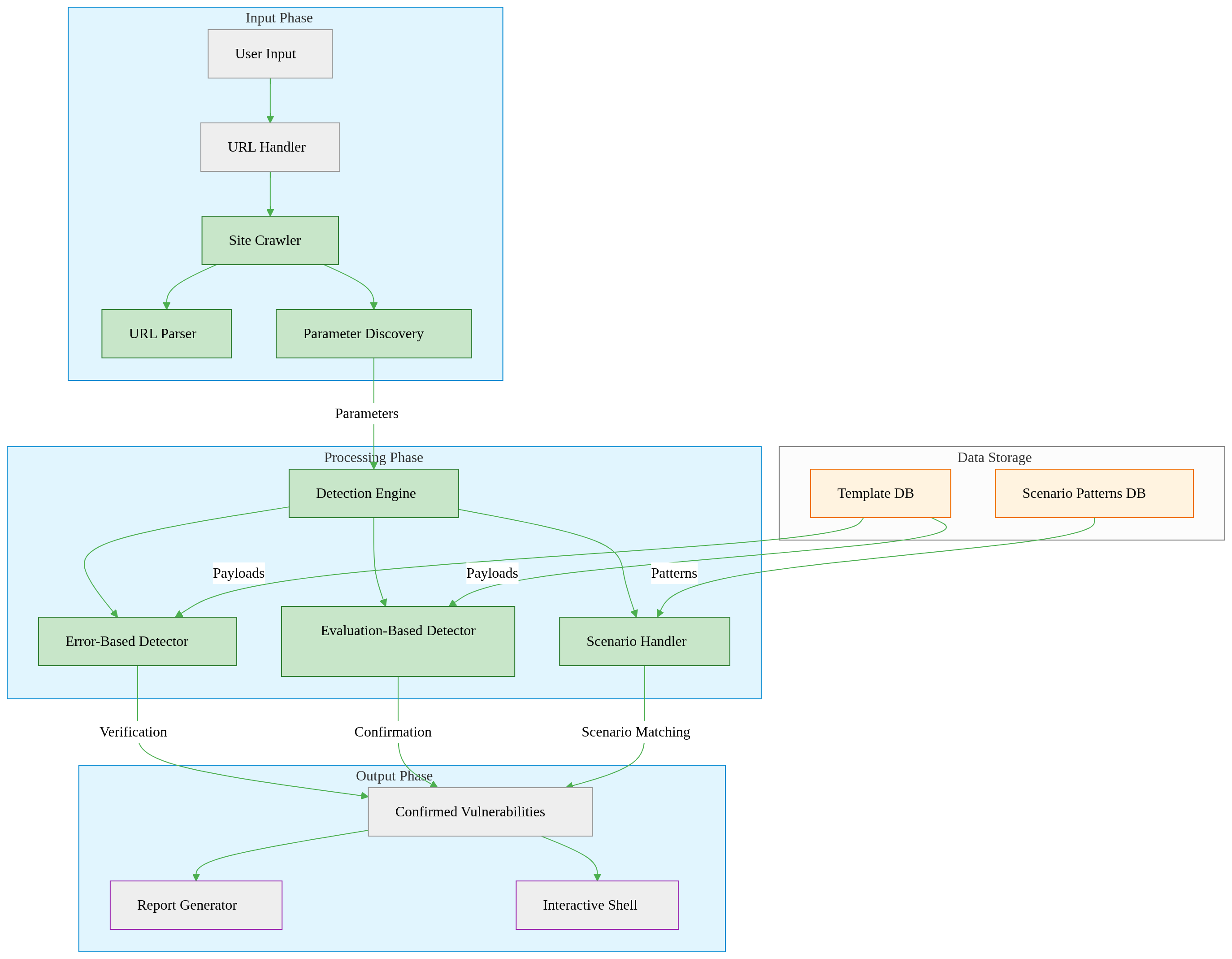
# EvaluationDetector uses evaluation payloads  
send(TEMPLATE\_DB["Jinja2"]["detection"]["evaluation"]["payload"])

**Exploitation Phase**

# SSTIExploiter retrieves payload templates  
payload = TEMPLATE\_DB[engine]["exploit"]["command\_exec"]["payload"].replace("COMMAND", user\_input)

3. Component Hierarchy

Structural Relationships



**The scanner consists of 4 essential components:**

**SSTIScanner**

The brain of the system

Manages the scanning workflow:  
URL Input → Parameter Discovery → Engine Detection → Exploitation

Handles threading and session management

**Detection Tools**

**Error Detector**: Looks for template engine fingerprints in error messages

**Evaluation Detector**: Sends math payloads (e.g., {{7\*7}}) to detect execution

Work together to confirm vulnerabilities

**SSTIExploiter**

Takes verified vulnerabilities and:

Executes system commands

Reads files

Maintains interactive sessions

**Support Systems**

**Site Crawler**: Discovers parameters by analyzing pages/forms

**Template Database**: Stores engine-specific attack payloads

**ScenarioHandler**: Special logic for lab environments

4.1.2.2 Scanning Workflow

Phase 1: Target Initialization

*Prepares the scanning environment and validates the target*

GUI Interface Overview

GUI Input Interface  
*Caption: Scanner input interface showing configuration fields*

User Inputs

**Target Specification**:

Single URL (e.g., https://example.com/search)

*OR* text file containing multiple URLs (one per line)

**Configuration Options**:

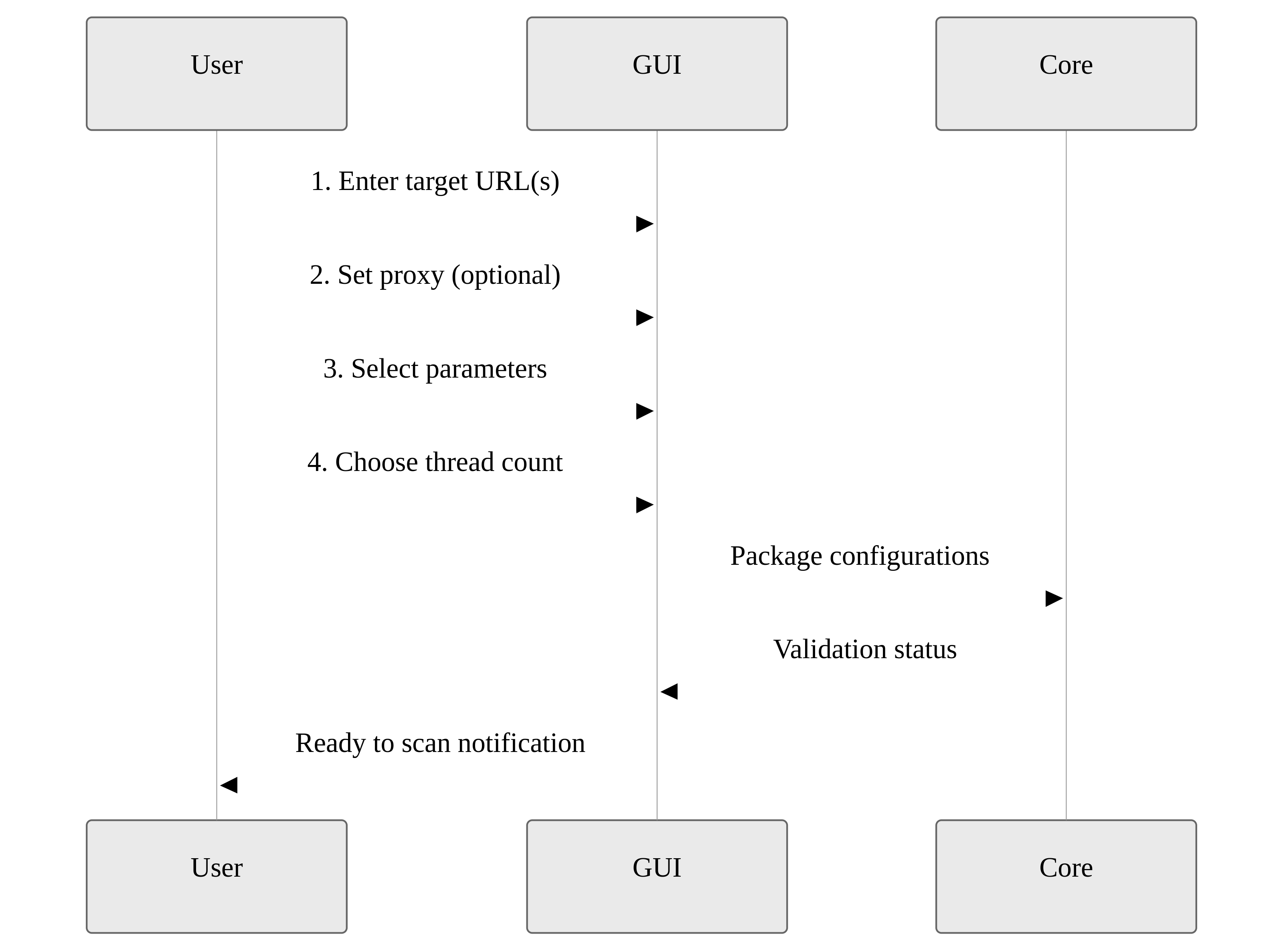
Number of parallel scanning threads

Proxy server address/port (optional)

Output report format (JSON/CSV/XML)

Specific parameters to test (optional list)

Input Flow Sequence



Step 1: Session Setup

# SSTIScanner.\_\_init\_\_  
self.session = requests.Session()  
self.session.proxies = proxies

Creates persistent HTTP connection preserving:

Authentication cookies

Custom headers

SSL/TLS session state

Disables SSL certificate verification for debugging

Step 2: Proxy Configuration

Applies user-provided proxy settings to all traffic

Supports both HTTP/HTTPS proxy types

Maintains connection reuse for efficiency

Step 3: Reachability Check

Sends test request to first URL

Validates response:

**Success**: HTTP 200 status → Continue scan

**Failure**:

Invalid URL format → Terminates immediately

Connection timeout → Logs error, skips URL

Step 4: Scenario Pre-Check

# ScenarioHandler.detect\_scenario  
if re.search(r"Basic server-side template injection", response\_text):  
 return Scenario2

Scans initial response content for:

Predefined vulnerability markers

Decision logic:  
*If scenario pattern detected* → Specialized handling  
*If no patterns found* → Standard scanning

Failure Handling

**Invalid URLs**: Terminates with format examples

**Unreachable targets**: Displays troubleshooting tips

**Proxy failures**: Attempts direct connection fallback

*This phase only establishes working conditions - no vulnerability testing occurs here.*

Phase 2: Parameter Discovery

*Identifies potential injection points through comprehensive analysis*

Key Objective:

Find all user-controllable inputs that reflect values back in responses

Step 1: Site Crawling

# SiteCrawler.crawl()  
forms = soup.find\_all('form')  
inputs = [input\_tag['name'] for form in forms for input\_tag in form.find\_all('input')]  
  
# SiteCrawler.\_extract\_params\_from\_url()  
query = urlparse(url).query  
params = parse\_qs(query).keys()

**Initial URL Analysis**

Starts at user-provided URL

Follows same-domain links up to configurable depth

Preserves session cookies for authenticated areas

**Multi-Source Extraction**

**HTML Forms**:

<input> fields (text/email/search types)

<textarea> content

Hidden parameters (<input type="hidden">)

**URL Query Strings**:

Parameters after ? (e.g., ?search=test)

**Link Parameters**:

URL parameters in <a href=""> tags

Step 2: Reflection Verification

# test\_parameter\_reflection()  
test\_res = session.get(url, params={param: "TINJ\_REFL\_TEST"})  
verify\_res = session.get(url, params={param: "TINJ\_VERIF\_123"})  
if test\_value in test\_res.text and verify\_value in verify\_res.text:  
 return True

**Strict Two-Stage Validation:**

**Initial Test**

Injects unique string (e.g., TINJ\_REFL\_7BxY9z)

Checks exact match in:

Response body content

HTTP headers

Redirect URLs

**Confirmation Test**

Sends different verification string (e.g., TINJ\_VERIF\_4QwP2m)

Requires both strings to appear unmodified

**Filter Logic:**  
Parameter survives only if:

Reflects both test values exactly

No encoding/truncation detected

Output Handling

**Passed Parameters**: Queued for engine detection

**Failed Parameters**: Permanently excluded

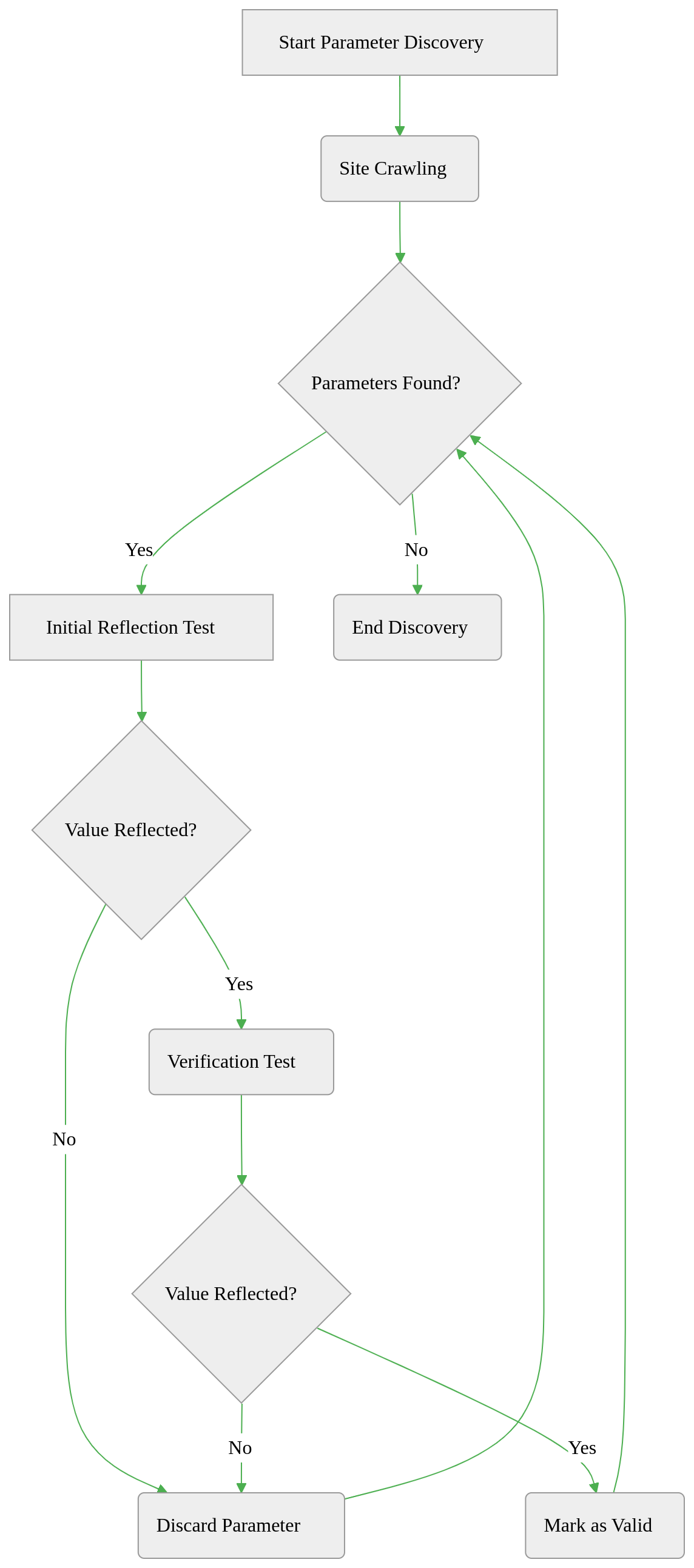
**Discovery Report**:

{  
 "discovered\_parameters": ["search", "email", "filter"],  
 "reflected\_parameters": ["search", "email"]  
}

Special Considerations

Avoids destructive operations during crawling

Limits recursion depth to prevent infinite loops



Phase 3: Engine Identification

*Determines the template engine through pattern analysis and execution testing*

Key Objective:

Confirm the server-side template engine (e.g., Jinja2, Twig) using two verification methods

Step 1: Error Pattern Analysis

# ErrorBasedEngineDetector.detect()  
response = session.get(url, params={param: error\_payload})  
for pattern in engine\_patterns:  
 if re.search(pattern, response.text):  
 return engine

**Method**:

Sends malformed template syntax to trigger errors

Example payload: ${<\%[%'}}%\

Analyzes error messages for fingerprints:

**Jinja2**: Contains "jinja2.exceptions.TemplateSyntaxError"

**Twig**: Shows "Twig\_Error\_Syntax"

Records partial matches for secondary verification

**Outcome**:

If **Error Fingerprint Found** → High-confidence engine identification  
Else → Proceed to Step 2

Step 2: Evaluation-Based Detection

# EvaluationBasedEngineDetector.detect()  
payload = self.payload\_map[engine]["payload"]  
response = session.get(url, params={param: payload})  
if data["expected"] in response.text:  
 return engine

**Method**:

Sends engine-specific arithmetic payloads:

**Jinja2**: {{7\*'7'}} → Expected output: "7777777"

**Twig**: {{7\*7}} → Expected output: "49"

Checks for mathematical execution evidence:

Direct output match (e.g., "49" appears in response)

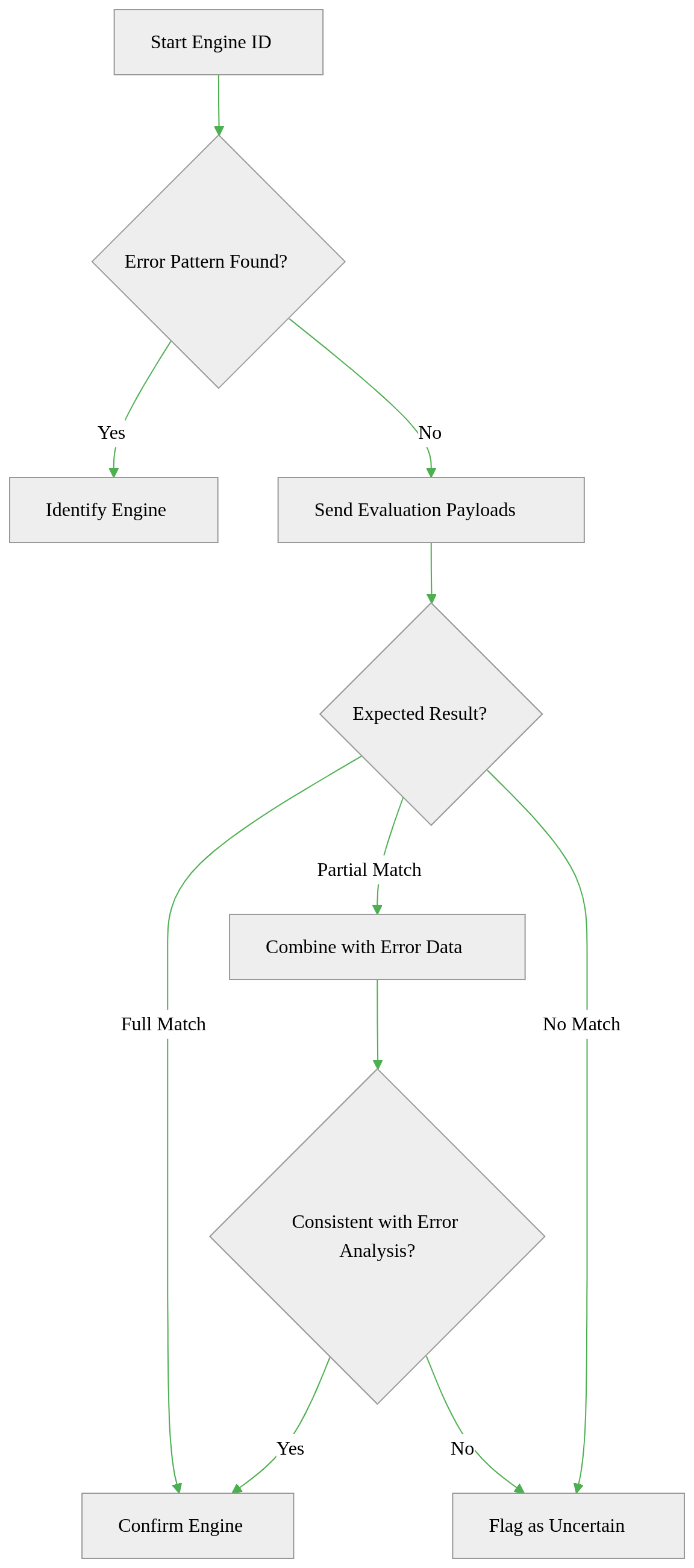
Partial pattern match (e.g., "777" in truncated responses)

**Verification Logic**:

**Full Match**: Confirms template engine definitively

**Partial Match**: Cross-references with error analysis results

**No Match**: Marks engine as "Unknown" for manual review



Phase 4: Post-Detection Actions

*Executes payloads and delivers results after confirming vulnerabilities*

Key Objective:

Leverage identified vulnerabilities for controlled exploitation and reporting

Step 1: Exploitation Launch

# SSTIExploiter.execute\_command()  
payload = self.payloads["exec"].replace("COMMAND", command)  
response = self.session.get(url, params={self.param: payload})

**Process**:

Retrieves engine-specific payloads from TemplateDB:

**Command Execution**: {{ self.\_\_init\_\_.\_\_globals\_\_... }} (Jinja2)

**File Read**: ${product.getClass()...} (Freemarker)

Injects payload into confirmed vulnerable parameter

Validates command execution/output sanitization

**Safety Measures**:

Timeout enforcement (15s default)

Payload sandboxing to prevent accidental damage

Step 2: Interactive Shell

# interactive\_shell()  
while True:  
 cmd = input("tinj-shell» ")  
 if cmd.startswith("read "):  
 print(exploiter.read\_file(cmd[5:]))  
 else:  
 print(exploiter.execute\_command(cmd))

**Features**:

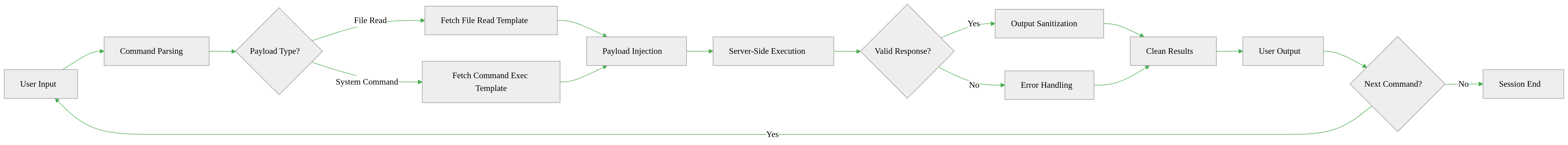
Real-time command execution interface

Context-aware parsing:

read /etc/passwd → Auto-uses file read payload

whoami → Uses command execution template

Output filtering to remove template artifacts



**Example Flow**:

User types whoami

Tool selects command execution template for detected engine

Injects {{ self.\_\_init\_\_...popen('whoami') }}

Server returns TINJ\_STARTrootTINJ\_END

Tool displays cleaned output: root

Step 3: Report Generation

**Output Formats**:

| **Format** | **Structure** | **Use Case** |
| --- | --- | --- |
| JSON | Nested objects with full technical details | Developer analysis |
| CSV | Tabular data with key fields | Spreadsheet integration |
| XML | Hierarchical vulnerability listing | Enterprise system ingestion |

**Report Contents**:

Vulnerable URL(s)

Confirmed parameter(s)

Template engine identified

Evidence snippets

Timestamp of discovery

Concurrency Management

Thread pool reused from earlier phases

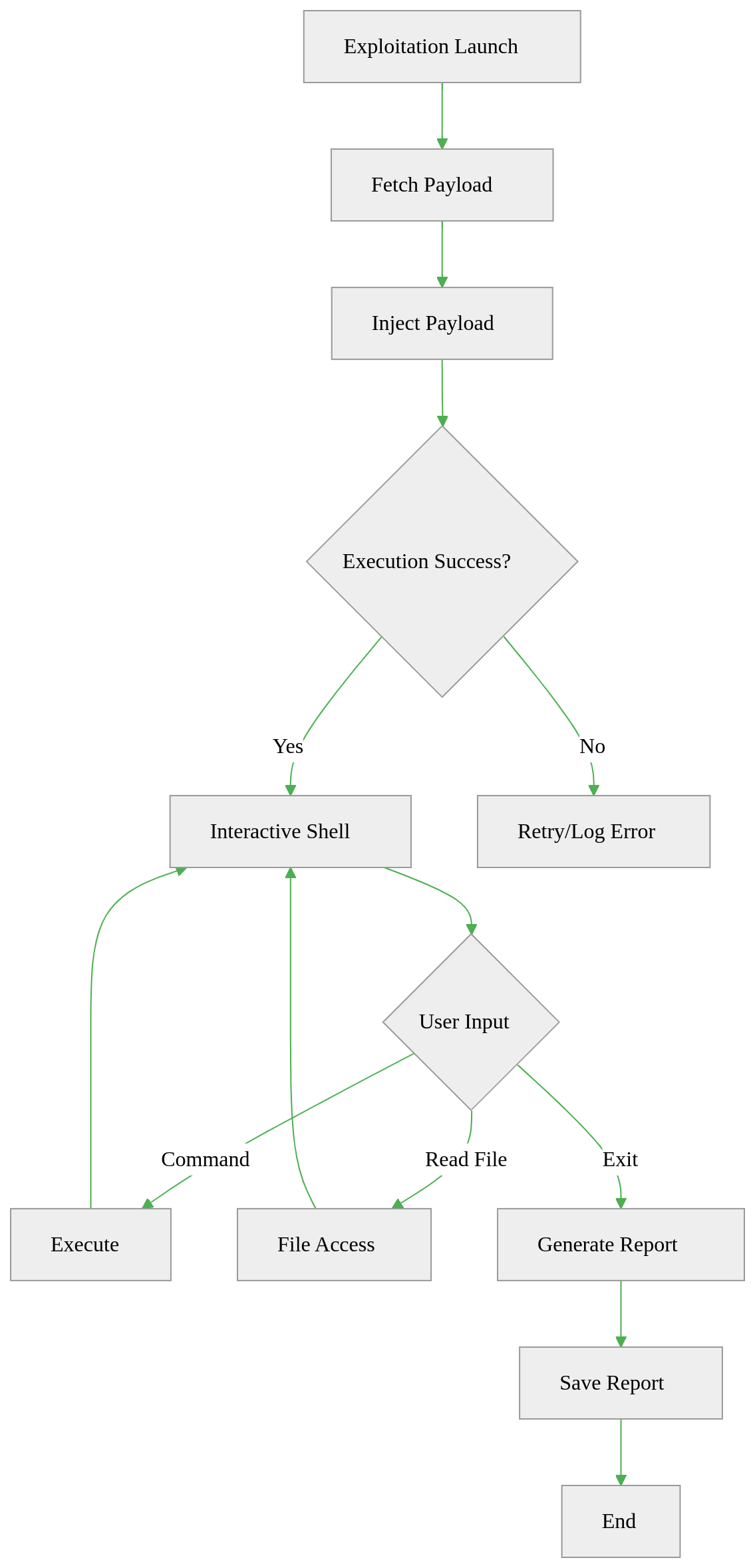
Parallel tasks:

Background report writing

Shell input/output handling

Payload retries on timeout

This phase transforms detection results into actionable outcomes while maintaining operational safety.



## Choice of Programming and Libraries

## User Graphic interface (GUI)

## Comparisons with other vulnerability

# Conclusion

## Summary of Achievements

## Potential Enhancements

## References