

جامعة آل البيت3333333

Al al-Bayt University

**“Web Vulnerability Scanner”**

**Presented by**

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

Supervisor: Dr.

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  Mohammad Abdallah Alzoubi | 21000906056  2100906042 |
| Yousef Mohammad Hjooj | 2100906076 |
| Abdalrahman Reda Albeshtawi | 2100906014 |

# **Abstract**

**TABLE OF CONTENTS**

**CHAPTER.1 Introduction**

**1.1** Project Problem ---------------------------------------------------------6

**1.2** Project Goals ------------------------------------------------------------6

**1.3**What is an automated web vulnerability? ----------------------------6

**1.4** Why are vulnerability scanners Important -------------------------- 7

**1.5** What the project covers? ------------------------------------------------8

**1.6** Beneficiaries-------------------------------------------------------------- 9

**1.7** SDLC Phases-------------------------------------------------------------10

**CHAPTER.2 Overview of Target vulnerabilities  
 2.1** Server-side Request Forgery (SSRF)------------------------------------11  
 **2.2** Server-side Template Injection (SSTI)  
 **2.3** Path Traversal  
 **2.4** Cross-Site Scripting (XSS)

**CHAPTER.3 Detection& Exploitation Methodologies**

**4.1** SSRF Detection & Exploitation Technique   
 **4.2** SSTI Detection & Exploitation Technique   
 **4.3** Path Traversal Detection & Exploitation Technique  
 **4.4** XSS Detection & Exploitation Technique

**CHAPTER.4 Building the Vulnerability Scanner**

**3.1** How the Scanner works?

**3.2** Choice of Programming and Libraries

**3.3** User Graphic interface (GUI)

**3.4** Comparisons with other vulnerability

**CHAPTER .5 Conclusion**

5.1 Summary of Achievements

5.2 Potential Enhancements

5.3 References

|  |
| --- |
| Table of Figures  [Figure I SDLC Phases 9](#_Toc186572006)  [Figure II Identify SSTI 1](#_Toc186572007)7  [Figure III Who Pathe travesal work 20](#_Toc186572008)  [Figure IV SSRF Detection 36](#_Toc186572009)  [Figure V Do you have cybersecurity experience? 37](#_Toc186572010)  [Figure VI What are the most important challenges in cyber security? 37](#_Toc186572011)  [Figure VII How To Prefer Website Design? 38](#_Toc186572012)  [Figure VIII Have you dealt with another platform, mention it? 38](#_Toc186572013)  [Figure IX Do you have any suggestions to include? 39](#_Toc186572014)  [Figure X Logic flaw of website 43](#_Toc186572015)  [Figure XI Database structure 44](file:///C:\Users\Marwa\OneDrive\Desktop\DOC%20NEW%2090.docx#_Toc186572016)  [Figure XII Home page 45](#_Toc186572017)  [Figure XIII Scrollable content 45](#_Toc186572018)  [Figure XIV Sign UP page 46](#_Toc186572019)  [Figure XV Login page 46](#_Toc186572020)  [Figure XVI Main categories of vulnerabilities 47](#_Toc186572021)  [Figure XVII Categories for reflected values within XSS 47](#_Toc186572022)  [Figure XVIII User profile page 48](#_Toc186572023)  [Figure XIX Help&support page 48](file:///C:\Users\Marwa\OneDrive\Desktop\DOC%20NEW%2090.docx#_Toc186572024)  [Figure XX Classification of labs based on category 49](#_Toc186572025)  [Figure XXI Submit the flag and access to the lab 50](#_Toc186572026)  [Figure XXII Red box explains error in submit the flag 50](#_Toc186572027)  [Figure XXIII Box Described Indirectly Lap Solution Method 51](#_Toc186572028)  [Figure XXIV How to solve a lab 52](#_Toc186572029)  [Figure XXV Enquire about the username to solve the lab 52](#_Toc186572030)  [Figure XXVI Green box explains correct in submit the flag 53](#_Toc186572031)  [Figure XXVII Leaderborad of the top ten user 53](#_Toc186572032) |

**Ch1:Introdaction**

* 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. **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.

**1.7 SDLC Phases**

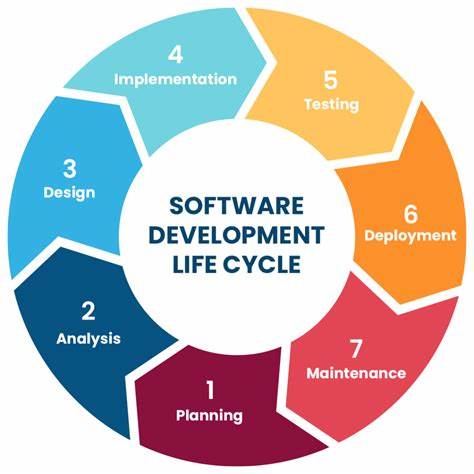
****

Figure I SDLC Phases

1. Planning
   * Define scope (path traversal, XSS, SSRF, SSTI).
   * Choose tools: Python, Requests, Beautiful Soup.
   * Set up test environments (Docker, Kali Linux).
2. Analysis
   * Study OWASP Top 10 patterns.
   * Map attack vectors (e.g., ../ for path traversal).
3. Design
   * Modularize components:
     + Scanner Engine (payload injection).
     + Reporting Module (PDF/HTML outputs).
4. Implementation
   * Code payload generators (e.g., SSTI {{7\*7}}).
   * Build response parsers (regex, DOM analysis).
5. Testing
   * Validate with OWASP Juice Shop.
   * Benchmark false positives/negatives.
6. Deployment
   * Package as CLI tool (Python Installer).
   * Publish on GitHub.
7. Maintenance
   * Update payload databases (CVE tracking).
   * Add new vulnerability checks (e.g., Log4j).

**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).

1. 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](https://portswigger.net/web-security/essential-skills/obfuscating-attacks-using-encodings#obfuscation-via-double-url-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.

1. 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.

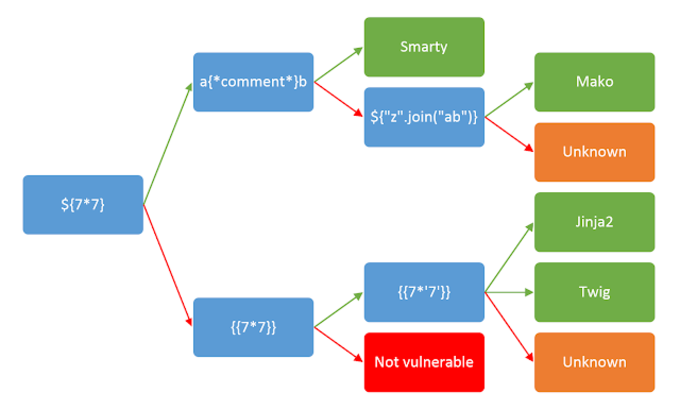


Figure II Identify SSTI

**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.

1. **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.

1. **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.

1. **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.

1. **Utilize a**[**Docker**](https://www.wallarm.com/cloud-native-products-101/what-is-docker)**Container**

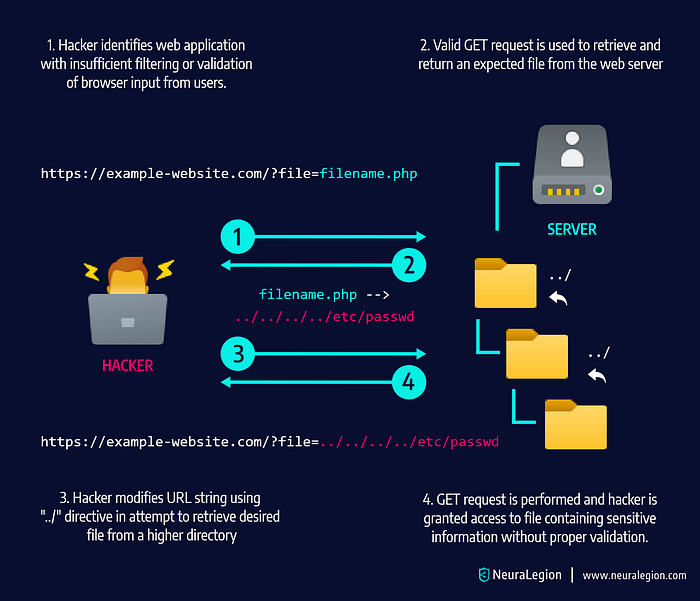
If none of the above solutions work then defenders must admit that the remote code execution is inevitable and should try to trim its impact by implementing customized sandboxing by executing the template engine in a completely locked Docker container.

**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?**

[Figure III Who Pathe travesal work](#_Toc186572008)

**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**](https://www.indusface.com/products/was-platform.php), 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.](https://thesecmaster.com/14-things-to-check-when-a-system-gets-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](https://thesecmaster.com/what-is-ssl-tls-how-ssl-tls-1-2-and-tls-1-3-differ-from-each-other/) 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.

**CH.3Detection & Exploitation Methodologies**

**4.1** SSRF Detection & Exploitation Technique

**Detection Technique**

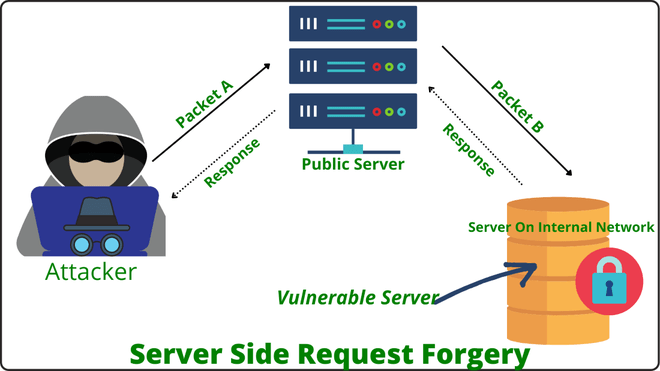
SSRF (Server Side Request Forgery) is a security vulnerability that allows an attacker to make unauthorized HTTP requests from the backend of a vulnerable web application by manipulating the URL/domain/path parameter of the request. The injected URL can come from either an internal network or a third-party 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.



[Figure IV **SSRF Detection**](#_Toc186572009)

**Exploitation Technique**

In the [previous section](https://blog.blueinfy.com/2023/04/ssrf-detection-exploitation-and.html), 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*

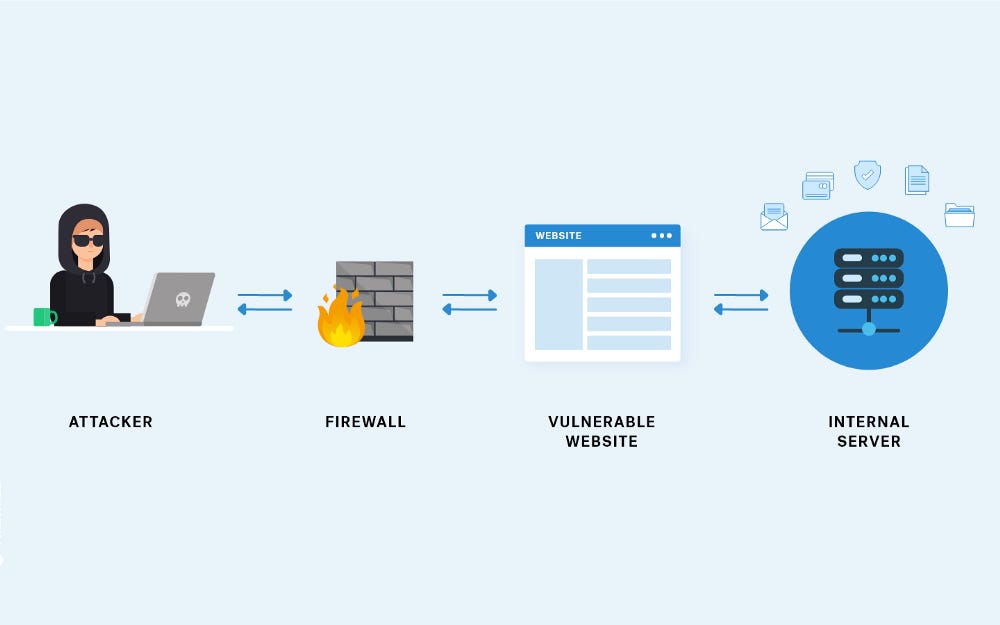
Azure (https://learn.microsoft.com/en-us/azure/virtual-machines/instance-metadata-service?tabs=windows)

* *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.



[Figure V](#_Toc186572010)

**4.1** SSRF Detection & Exploitation Technique

**Detection Technique**