Name:Kaushik Kotian Roll No.:29 Div:D20B Batch:B

EXPERIMENT NO:- 8

AIM:- Cross-Site Scripting (XSS) vulnerability and OS Command vulnerability.

THEORY:-

Cross-Site Scripting (XSS) Vulnerability

Definition: Cross-Site Scripting (XSS) is a security vulnerability commonly found in web applications, where an attacker is able to inject malicious scripts into web pages viewed by other users. These scripts typically execute in the context of the victim's browser, potentially leading to the theft of sensitive information, such as cookies, session tokens, or personal data. XSS attacks can also be used to hijack user sessions, redirect users to malicious websites, or carry out phishing attacks.

Types of XSS

There are three primary types of XSS attacks, each with a distinct method of injection and impact:

Stored XSS (Persistent XSS): In Stored XSS, the malicious script is permanently stored on the target server (e.g., in a database, comment field, or message board). When a user requests a page containing this script, it is served to them as part of the page content. This type of attack can have a broad impact because every user who accesses the affected page will be exposed to the malicious code.

Example:
A user posts a comment containing a malicious <script> tag on a public forum. Every time someone views that comment, the script is executed in their browser, stealing their session cookie.

Reflected XSS (Non-Persistent XSS): Reflected XSS occurs when the injected script is reflected off a web server, most commonly through query parameters or form inputs. The malicious code is executed immediately as part of the response and does not persist on the server. This type of attack usually requires social engineering, where the attacker tricks a user into clicking a specially crafted URL.

Example:
An attacker sends a link to a user that contains a malicious script embedded in the URL. When the user clicks the link, the server reflects the script back as part of the page response, executing it in the victim's browser.

DOM-Based XSS: DOM-based XSS is a client-side attack where the vulnerability exists in the browser's DOM (Document Object Model) rather than on the server. The malicious script is

injected through client-side JavaScript code that dynamically modifies the web page without proper sanitization.

Example:
A web page dynamically updates the content of a div element based on user input from the URL. If this input is not properly sanitized, an attacker could inject a script into the URL, which would then be executed by the client-side JavaScript.

Impact of XSS
The impact of XSS attacks can vary based on the type of attack and the privileges of the targeted users. Some potential consequences include:

Session Hijacking: The attacker can steal session cookies and impersonate the victim to perform unauthorized actions.
Credential Theft: The attacker can capture login credentials or other sensitive data by manipulating the page content.
Defacement: The attacker can alter the content of the web page, leading to defacement or misinformation.
Phishing Attacks: The attacker can redirect users to malicious websites designed to steal personal information.

XSS Prevention Techniques
Input Validation and Sanitization: One of the most important steps in preventing XSS attacks is to validate and sanitize all user inputs. Input validation ensures that the data provided by the user conforms to expected formats (e.g., no HTML or JavaScript code). Sanitization removes or escapes any potentially dangerous characters, such as <, >, and &.

Example Techniques:

Use server-side frameworks that automatically sanitize inputs (e.g., HTML sanitizers in Python, Java, PHP).
Use input filtering to reject suspicious inputs containing script tags or special characters.
Output Encoding: Always encode data before rendering it in the browser. This ensures that user-supplied content is treated as text rather than executable code. Context-specific encoding should be applied depending on where the output is being placed (e.g., in HTML, JavaScript, or URL).

Example Techniques:

HTML Encode characters such as <, >, &, and ".
Use JavaScript escaping for dynamic content injected into scripts.
Content Security Policy (CSP): Implementing a Content Security Policy (CSP) is a robust defense mechanism that restricts the sources from which a page can load resources (e.g.,

scripts, images, stylesheets). By defining a strict CSP, you can prevent unauthorized or malicious scripts from being executed.

Content-Security-Policy: script-src 'self' https://trustedsource.com;
HttpOnly and Secure Cookies: Cookies that contain sensitive information (such as session tokens) should be marked with the HttpOnly attribute, preventing them from being accessed by JavaScript. The Secure attribute ensures that cookies are only transmitted over HTTPS, reducing the risk of interception.

OS Command Injection Vulnerability
Definition: OS Command Injection is a security vulnerability that allows an attacker to execute arbitrary operating system (OS) commands on the server hosting the application. This occurs when user inputs are improperly sanitized or used directly in system-level commands. OS Command Injection can be highly dangerous, as it can lead to full system compromise, allowing attackers to execute commands with the privileges of the vulnerable application.

Types of OS Command Injection
There are two primary types of OS Command Injection based on the injection method:

Direct Command Injection: In direct command injection, the attacker appends their own commands directly to the vulnerable input field. The system executes these commands along with the legitimate ones.

Example: If an application uses a system command like ping to test network connectivity, an attacker might input:

127.0.0.1; ls -la
This would result in the execution of both ping 127.0.0.1 and ls -la, allowing the attacker to list files on the server.

Blind Command Injection: In blind command injection, the attacker doesn't directly see the output of their commands but can infer it through side effects or timing differences. For example, the response time might indicate whether a command has executed successfully.

Example: An attacker might send:

127.0.0.1 && sleep 10
If the server takes 10 seconds to respond, the attacker knows that the command was executed.

Conclusion :- Hence we studied about the Cross Site Scripting vulnerabilities and OS commands vulnerabilities.