**SURE TRUST ORGANIZATION**

**Domain : Cyber Security**

**Trainer : Mr. Himaneesh sir**

**Presented by :**

**L. Bhavani Shankar**

**XSS PAYLOADS**

INTRODUCTION

Web security is paramount in safeguarding online assets and user data from a multitude of threats. Confidentiality, one of the core tenets of web security, underscores the importance of preventing unauthorized access to sensitive information such as user credentials and personal details. Maintaining data integrity is equally critical to ensure that information presented on a website is accurate and has not been tampered with. Without proper safeguards, attackers could manipulate data, compromising the reliability of web content. Furthermore, the availability of online services is contingent on robust security measures. Protection against denial-of-service (DoS) attacks is imperative to guarantee that websites remain accessible to legitimate users, contributing to a seamless and uninterrupted user experience.

Authentication and authorization mechanisms form the foundation of secure web practices. Weak authentication processes can lead to unauthorized access, underscoring the significance of robust login mechanisms. Additionally, proper authorization controls dictate the level of access granted to users based on their roles, preventing unauthorized privileges. Ensuring secure data validation is crucial in mitigating injection attacks, such as SQL injection and Cross-Site Scripting (XSS), which could compromise user data and compromise the confidentiality of sensitive information. Effective session management is essential to thwart session hijacking and fixation attacks, preserving the security of user sessions and preventing unauthorized access.

Specific vulnerabilities like Cross-Site Scripting (XSS) and SQL Injection (SQLi) pose significant risks to web security. XSS vulnerabilities can compromise user data confidentiality and integrity by allowing attackers to inject malicious scripts into web pages. Mitigating XSS risks involves implementing robust input validation, output encoding, and Content Security Policy (CSP). SQL injection, on the other hand, threatens the security of databases, potentially leading to unauthorized access and data manipulation. Techniques such as parameterized queries, prepared statements, and stringent input validation are essential in preventing SQL injection attacks. In summary, a comprehensive approach to web security involves addressing confidentiality, integrity, and availability, while implementing secure authentication, authorization, and data validation practices to counteract specific vulnerabilities that may be exploited by malicious actors.

Cross-Site Scripting (XSS) is a type of security vulnerability that occurs when an attacker injects malicious scripts into web pages viewed by other users. This occurs when a web application allows users to input untrusted data, which is then included in the output sent to the browser without proper validation or escaping. XSS can have severe consequences, including the theft of sensitive information, session hijacking, or defacement of websites. There are three main types of XSS:

1. Reflected XSS (Non-Persistent): In this scenario, the malicious script is embedded in a URL or other input, and it's immediately reflected in the response sent to the user. For example, an attacker might craft a phishing link that, when clicked, executes a script on the victim's browser.

[https://example.com/page?name=<script>alert('XSS')](https://example.com/page?name=%3cscript%3ealert('XSS')) </script> ```

2. Stored XSS (Persistent) : With stored XSS, the injected script is permanently stored on the target server, often in a database. When a user accesses the compromised page, the script is served along with the legitimate content. This type of XSS can have a long-lasting impact as it affects all users who visit the compromised page.

Malicious Post Content: <script>alert('XSS')</script>

3. DOM-based XSS: This type occurs when the client-side script manipulates the Document Object Model (DOM) of a web page. Instead of targeting the server's response, the attacker exploits the way the page is processed on the client side. The malicious script modifies the DOM, leading to unintended consequences .

<script> document.getElementById("demo").innerHTML = "<img src='malicious-site.com'>"; </script>

Exploiting XSS typically involves injecting scripts that steal user cookies, redirect users to phishing sites, or perform actions on behalf of the victim. To prevent XSS, web developers should implement input validation, output encoding, and use security mechanisms like Content Security Policy (CSP) to restrict the execution of scripts to trusted sources. User awareness and responsible disclosure also play crucial roles in mitigating the impact of XSS vulnerabilities.

Below are the different types of basic XSS payloads to understand how they are working and how they are used for vulnerability assessment of web applications to provide more security to the users for their data . In this payloads it also includes the STORED XSS , REFLECTIVE XSS and DOM BASED XSS as well to know the functionalities of them by using different types in the creation of web pages called JAVA SCRIPT language called TAGS , EVENTS etc .

Below are examples for all the tags mentioned in the previous paragraph:

1. Script Tags (`<script>`):

html <script>alert('XSS');</script>

Explanation: This is a basic example of an XSS payload. If input is not properly validated, an attacker could inject malicious scripts like this to execute arbitrary code on the victim's browser.

1. Link Tags (`<a>`):

html <a href="https://legitimate-site.com/redirect?url=https://malicious-site.com">Click me</a>

Explanation: This could lead to an open redirect vulnerability if the url parameter is not properly validated, allowing an attacker to redirect users to a malicious site.

1. Input Tags (`<input>`):

html <input type="text" name="username" value="'; DROP TABLE users; --">

Explanation: An example of an SQL injection attempt. If not handled properly, this could manipulate the SQL query and cause unintended actions, like deleting the users table.

1. Image Tags (`<img>`):

html <img src="malicious-site.com/malicious-image.jpg onerror="alert('XSS')">

Explanation: If user input is used to construct the src attribute, an attacker might inject a script that executes when the image fails to load.

1. Iframe Tags (`<iframe>`):

html <iframe src="https://malicious-site.com"></iframe>

Explanation: Iframes can be used for embedding external content. If not properly controlled, this can lead to Clickjacking, where the attacker tricks the user into interacting with unintended elements.

1. Form Tags (`<form>`):

html <form action="/delete-account" method="post"> <input type="hidden" name="action" value="delete"> <button type="submit">Delete Account</button> </form>

Explanation: If CSRF protection is lacking, an attacker might trick a user into submitting a form that performs actions like deleting their account without their consent.

7. Link Relations (`<link rel="stylesheet">`):

html <link rel="stylesheet" href="https://malicious-site.com/malicious- styles.css">

Explanation: Loading stylesheets from untrusted sources could lead to CSS injection vulnerabilities if the styles are not sanitized.

8. Cookie Handling (`Set-Cookie`):

http Set-Cookie: session\_id=abc123; Secure; HttpOnly

Explanation: Proper cookie handling involves attributes like Secure to ensure it's transmitted over secure channels and HttpOnly to prevent access via client-side scripts.

9. HTTP Methods (`GET`, `POST`, etc.):

http GET /reset-password?token=malicious-token HTTP/1.1

Explanation: Sensitive data in URLs (like password reset tokens) should be transmitted securely, preferably using POST requests and proper session management.

10. Textarea Tags (`<textarea>`):

html <textarea name="comment">'; DROP TABLE comments; --</textarea>

Explanation: Similar to input fields, text areas should be validated to prevent injection vulnerabilities.

11. XML Tags (`<xml>`, `<xpath>`):

xml <xml version="1.0" encoding="UTF-8"> <data> <user>admin</user> <password>input</password> </data>

Explanation: This example showcases a piece of XML content. However, if user input is not properly validated and sanitized, an attacker might inject malicious XML content to manipulate XML-based functionalities, potentially leading to XPath injection vulnerabilities.

12. AJAX Requests (`XMLHttpRequest`, Fetch API):

javascript const xhr = new XMLHttpRequest(); xhr.open('GET', 'https://malicious-site.com/data', true); xhr.send();

Explanation: This JavaScript code represents an XMLHttpRequest to fetch data from a remote server. If user input is directly used in constructing the URL or payload, it could lead to security issues like Cross-Site Request Forgery (CSRF) or unauthorized access.

13. File Upload Tags (`<input type="file">`):html <form action="/upload" method="post" enctype="multipart/form-data"> <input type="file" name="file"> <input type="submit" value="Upload"> </form>

Explanation: This HTML form allows users to upload files. If file uploads are not properly validated and sanitized, an attacker might upload malicious files, leading to security risks such as code execution or storing malware on the server.

14. Error Messages (`<error>`, `404`, etc.):

html <error>Error: This page does not exist.</error>

Explanation: Custom error messages might unintentionally reveal sensitive information about the application's structure or functionality. This could aid attackers in understanding potential vulnerabilities or misconfigurations.

15. HTTP Headers (`Content-Security-Policy`, `X-Frame-Options`):

http Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted-scripts.com;

Explanation: This HTTP header is part of Content Security Policy (CSP), which helps prevent Cross-Site Scripting (XSS) attacks. The example specifies that scripts should only be loaded from the same origin and a trusted external source.

16. SQL Query Tags (`SELECT`, `INSERT`, etc.):

sql SELECT \* FROM users WHERE username = 'input';

Explanation: This SQL query selects user data based on the provided username. If user input is directly incorporated into SQL queries without proper validation, it can lead to SQL injection vulnerabilities.

17. API Endpoints (`<fetch>`, `<axios>`):

javascript fetch('https://api.malicious-site.com/data', { method: 'GET', headers: { 'Authorization': 'Bearer malicious-token' } });

Explanation: This JavaScript code represents an API request using the Fetch API. If user input is used to construct the URL or headers without proper validation, it could lead to security issues such as unauthorized access or injection attacks.

18. Audio and Video Tags (`<audio>`, `<video>`):

html <audio controls> <source src="malicious-site.com/malicious-audio.mp3" type="audio/mp3"> Your browser does not support the audio tag. </audio>

Explanation: If user input is used to construct the source URL, an attacker might inject malicious audio or video content. This could lead to various security risks, including unintended content playback or malicious file execution.

19. DOM Manipulation (`document.createElement`, `innerHTML`):

javascript const div = document.createElement('div'); div.innerHTML = '<p>Malicious content</p>'; document.body.appendChild(div);

Explanation: This JavaScript code dynamically creates a DOM element and injects potentially malicious content. If user input is directly incorporated into DOM manipulation functions without proper validation, it could lead to Cross-Site Scripting (XSS) vulnerabilities.

20. Content-Disposition Header: http Content-Disposition: attachment; filename="malicious-file.exe"

Explanation: This HTTP header suggests that the browser should treat the response content as an attachment and prompts the user to download a file. If user input is used to construct filenames without proper validation, it could lead to security risks such as arbitrary file downloads.

These examples illustrate potential security issues associated with each tag and emphasize the importance of secure coding practices to mitigate such vulnerabilities.

Mitigating Cross-Site Scripting (XSS) vulnerabilities is crucial for ensuring the security of web applications. Here are some effective measures to guard against XSS attacks:

1. \*\*Input Validation:\*\* - Validate and sanitize all user inputs on both the client and server sides. - Use appropriate validation mechanisms to ensure that input adheres to expected formats.

2. Output Encoding:- Encode user-generated content before rendering it in the HTML context. - HTML escape characters such as `<`, `>`, `&`, etc., using libraries or frameworks.

3. Content Security Policy (CSP): - Implement a Content Security Policy header to restrict the sources from which certain types of content can be loaded. - Specify allowed domains for scripts, styles, images, and other resources.

4. HTTP-Only Cookies: - Set the `HttpOnly` flag on cookies to prevent them from being accessed via JavaScript. - This reduces the risk of session theft through XSS attacks.

5. Secure Cookies: - Set the `Secure` flag on cookies to ensure they are only transmitted over HTTPS connections. - This prevents attackers from intercepting cookies during transit.

6. X-Content-Type-Options Header: - Set the `X-Content-Type-Options` header to `nosniff` to prevent browsers from interpreting files as a different MIME type. - This helps in preventing certain types of XSS attacks.

7. X-XSS-Protection Header: - Enable the XSS filter in the browser by setting the `X-XSS-Protection` header. - This helps the browser to detect and block some types of XSS attacks.

8. Use Frameworks with Built-in Protections: - Choose web application frameworks that have built-in security features, such as automatic output encoding. - Frameworks like Ruby on Rails, Django, and Express.js have strong security mechanisms.

9. Regular Security Audits and Penetration Testing: - Conduct regular security audits of your web application's codebase. - Perform penetration testing to identify and address vulnerabilities proactively.

10. Context-Specific Output Encoding: - Apply context-specific output encoding based on where the data is being used (e.g., HTML, attributes, JavaScript, CSS). - Different contexts require different encoding techniques.

11. Browser Security Features: - Stay informed about the security features provided by modern browsers. - Utilize browser security mechanisms, such as the `referrer-policy` header, to control how much information is included with requests.

Regularly updating and patching software, staying informed about the latest security best practices, and testing applications thoroughly contribute to a robust security posture.