**Introduction**

In an XSS attack, malicious content (malicious script) is introduced into the trusted context of a vulnerable Web application. The victim, on executing the Web application, is served with the malicious content which masquerades as a part of the legitimate code of the application. The victim's browser ends up inadvertently executing the malicious script because of its inability to differentiate between malicious and legitimate content

The severity of the XSS threat is well documented within the web application development industry. It is the third most prevalent issue in the Open Web Application Security Project, found in the injection category. According to OWASP, XSS flaws are found in roughly two-thirds of all applications today.

**Reflected XSS**

Reflected XSS occurs when a web application immediately returns a user input in an error message or other form of output on the page. This does not get saved into the server. Typically use a phishing email or malicious advert to trick a user into clicking a fraudulent link. When the user clicks this link, the script executes in their browser. For example, a website has a search box, and that search is reflected to user, as in you searched for. What an attacked can do now is edit that box, and send this link, this will now when victim enters that link, the current session cookie will then be reflected to the attacker’s website.

**Stored XSS**

Stored XSS is a type of XSS attack where the attacker’s malicious script gets saved inside a database, such as that for a message forum, visitor log, comment field, etc. For example, consider a website that has a comments section in which end users can enter comments. Suppose one user has entered a malicious script in the comments, and the website saves that comment inside the database without using proper techniques used for preventing XSS attacks. Then whenever another user tries to retrieve comments, this malicious script is likely to be executed.

**DOM-based XSS attack(Document Object Model)**

Just like a reflected cross-site scripting attack, a DOM-based attack can also be delivered via a URL containing a dangerous script. However, instead of loading the payload in the HTTP response, the attack is carried out by maliciously manipulating the DOM environment. It is worth noting that the attacker crafted malicious payload cannot be found in the response as in the case of the other two types of attack. It can be found by either scrutinizing the DOM or when the Web page is loaded, i.e., at runtime. For example, suppose you are taking parameters directly from the URL and passing it to the eval function. The URL and script will be as follows. Continue explaining…

**Defensive methods**

**Do not trust any user input**

Since every user input on a web page introduces a risk of a cross-site scripting attack, you should treat them as untrusted and validate them as much as possible. A simple yet effective technique you can use is to pass every inputted data via a security filter that detects and removes harmful keywords. You can configure the filter to identify some common scripting strings that are not expected from the inputted data and prevent them from being executed. An example of such a scripting string is the HTML <SCRIPT> tag. But this does not protect from other attack possibilities.

**Use escaping/encoding**

This technique modifies certain standard characters in the user’s data and ensures they are not interpreted as active content. Based on where the user’s data is to be used, this may involve implementing a combination of HTML, CSS, JavaScript, and URL encoding.

**Sanitize HTML**

If the user’s generated content contains HTML, escaping or encoding them could break all the valid tags. You need to use a security-focused library for parsing and cleaning the HTML formatted text in such cases. Node.js has a special XSS prevention package.

**Use Content Security Policy (CSP)**

CSP is a browser-side mechanism that lets you whitelist the trusted sources in HTTP headers. You can use the technique to approve sources of content that browsers are permitted to load on your website.

**Cookie Security**: We should specify the httpOnly flag of cookies to be true so that they cannot be accessed on the client-side. This method can also be helpful in preventing XSS attacks.

**Same Site Attribute**: We should use the same site attributes for cookies as it can be helpful in preventing XSS attacks. When a cookie’s same site attribute is set to “strict”, it is stripped from all the cross-origin requests and when it is set to “lax” it is stripped from all cross-origin requests other than GET, OPTIONS and TRACE requests.