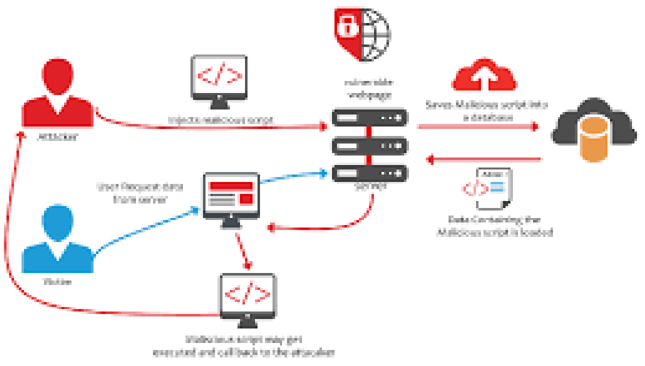
Cross Site Scripting (XSS)

- Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.
- It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other.
- Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data.
- If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

How Cross-site Scripting Works



Attacker
Injects malicious script
vulnerable webpage
Server
Saves Malicious script into a database
Data Containing the Malicious script is loaded
User Request data from server
Victim
Malicious script may get executed and call back to the attacker

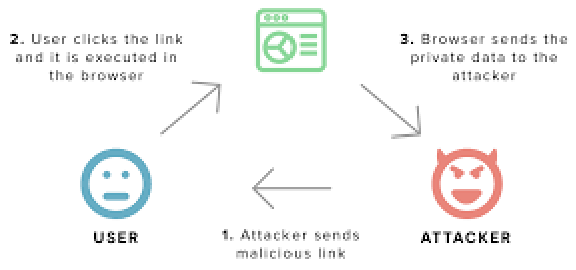
There are two stages to a typical XSS attack:

1) To run malicious JavaScript code in a victim's browser, an attacker must first find a way to inject malicious code (payload) into a web page that the victim visits.
2) After that, the victim must visit the web page with the malicious code. If the attack is directed at particular victims, the attacker can use social engineering and/or phishing to send a malicious URL to the victim.

For step one to be possible, the vulnerable website needs to directly include user input in its pages. An attacker can then insert a malicious string that will be used within the web page and treated as source code by the victim’s browser. There are also variants of XSS attacks where the attacker lures the user to visit a URL using social engineering and the payload is part of the link that the user clicks.

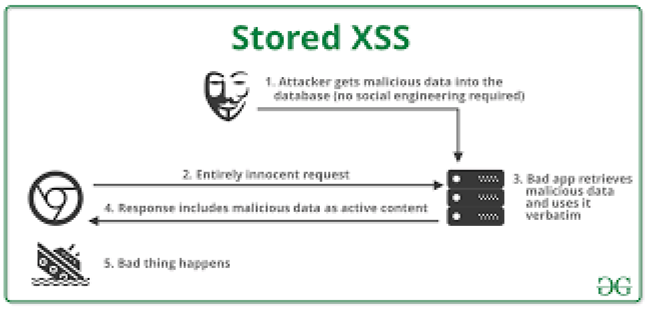
Types of cross-site scripting :-

Reflected XSS


2. User clicks the link and it is executed in the browser
3. Browser sends the private data to the attacker
USER
1. Attacker sends malicious link
ATTACKER

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other website. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user’s browser. The browser then executes the code because it came from a “trusted” server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS.

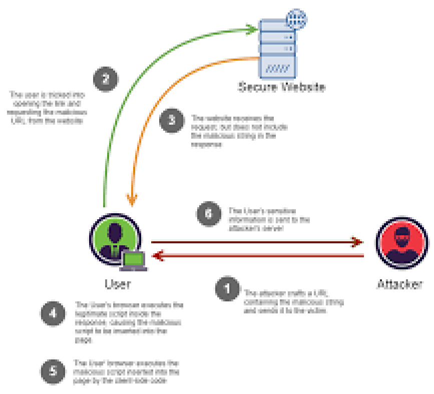
Stored XSS


Stored XSS
1. Attacker gets malicious data into the database (no social engineering required)
2. Entirely innocent request
3. Bad app retrieves malicious data and uses it verbatim
4. Response includes malicious data as active content
5. Bad thing happens
GG

Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

The data in question might be submitted to the application via HTTP requests; for example, comments on a blog post, user nicknames in a chat room, or contact details on a customer order. In other cases, the data might arrive from other untrusted sources; for example, a webmail application displaying messages received over SMTP, a marketing application displaying social media posts, or a network monitoring application displaying packet data from network traffic.

DOM-based XSS


Secure Website
User
Attacker
1 The attacker crafts a URL containing the malicious string and sends it to the victim
2 The user is tricked into opening the link and requesting the malicious URL from the website
3 The website receives the request, but does not include the malicious string in the response
4 The User's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page
5 The User' browser executes the malicious script inserted into the page by the client-side code
6 The User's sensitive information is sent to the attacker's server

DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute

In a typical case, the input field would be populated from part of the HTTP request, such as a URL query string parameter, allowing the attacker to deliver an attack using a malicious URL, in the same manner as reflected XSS.

Blind Cross-site Scripting

Blind Cross-site Scripting is a form of persistent XSS. It generally occurs when the attacker's payload saved on the server and reflected back to the victim from the backend application. For example, in feedback forms, an attacker can submit the malicious payload using the form, and once the backend user/admin of the application will open the attacker's submitted form via the backend application, the attacker's payload will get executed. Blind Cross-site Scripting is hard to confirm in the real-world scenario but one of the best tools for this is XSS Hunter.

Impact of XSS vulnerabilities

An attacker who exploits a cross-site scripting vulnerability is typically able to:

- Impersonate or masquerade as the victim user.
- Carry out any action that the user is able to perform.
- Read any data that the user is able to access.
- Capture the user's login credentials.
- Perform virtual defacement of the web site.
- Inject trojan functionality into the web site.

The actual impact of an XSS attack generally depends on the nature of the application, its functionality and data, and the status of the compromised user. For example:

- In a brochureware application, where all users are anonymous and all information is public, the impact will often be minimal.
- In an application holding sensitive data, such as banking transactions, emails, or healthcare records, the impact will usually be serious.

- If the compromised user has elevated privileges within the application, then the impact will generally be critical, allowing the attacker to take full control of the vulnerable application and compromise all users and their data.

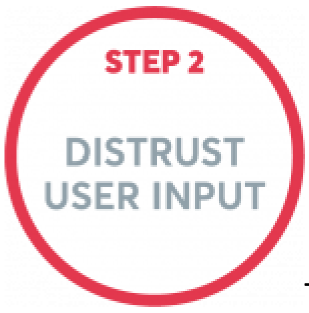
How to Prevent XSS

Preventing Cross-site Scripting (XSS) is not easy. Specific prevention techniques depend on the subtype of XSS vulnerability, on user input usage context, and on the programming framework. However, there are certain general strategic principles that you should follow to keep your web application safe.

Step 1: Train and maintain awareness

To keep your web application safe, everyone involved in building the web application must be aware of the risks associated with XSS vulnerabilities. You should provide suitable security training to all your developers, QA staff, DevOps, and SysAdmins. You can start by referring them to this page.

Step 2: Don't trust any user input



Treat all user input as untrusted. Any user input that is used as part of HTML output introduces a risk of an XSS. Treat input from authenticated and/or internal users the same way that you treat public input.

Step 3: Use escaping/encoding

Use an appropriate escaping/encoding technique depending on where user input is to be used: HTML escape, JavaScript escape, CSS escape, URL escape, etc. Use existing libraries for escaping, don't write your own unless absolutely necessary.

Step 4: Sanitize HTML

If the user input needs to contain HTML, you can't escape/encode it because it would break valid tags. In such cases, use a trusted and verified library to parse and clean HTML. Choose the library depending on your development language, for example, HtmlSanitizer for .NET or SanitizeHelper for Ruby on Rails.

Step 5: Set the HttpOnly flag

To mitigate the consequences of a possible XSS vulnerability, set the HttpOnly flag for cookies. If you do, such cookies will not be accessible via client-side JavaScript.

Step 6: Use a Content Security Policy

To mitigate the consequences of a possible XSS vulnerability, also use a Content Security Policy (CSP). CSP is an HTTP response header that lets you declare the dynamic resources that are allowed to load depending on the request source.

Reference

https://www.acunetix.com/websitesecurity/cross-site-scripting/

https://infosecwriteups.com/all-about-cross-site-scripting-xss-406a2db6c330

https://portswigger.net/web-security/cross-site-scripting

https://owasp.org/www-community/attacks/xss/#:~:text=Cross%2DSite%20Scripting%20(XSS),to%20a%20different%20end%20user.