What is cross-site scripting (XSS)?

Cross-site scripting (usually written XSS) is the vulnerability where an attacker injects JavaScript into a page that another user later loads. The attacker's code runs with that user's permissions inside the application. Three flavors exist (stored, reflected, DOM-based) and they share a single root cause: somewhere in the rendering pipeline, attacker text was treated as code instead of data.

HOW IT WORKS

01 What are the three types of XSS?

Stored XSS. The attacker plants their payload into the application's data (a comment, a profile field, a chat message). Every user who later views that data runs the payload. Worst variant because the attacker fires it once and gets every visitor.

Reflected XSS. The payload travels in a URL parameter or form submission and gets echoed back into the page that loads next. Requires the victim to click a crafted link. Common in search boxes and error pages.

DOM-based XSS. The vulnerable code lives entirely in client-side JavaScript: the page reads a value (from the URL, a localStorage entry, a message event) and writes it into the DOM through an unsafe sink like `innerHTML`. Server logs may show nothing because the payload never reaches the server.

02 What do attackers actually do with XSS?

Real exploit patterns we see on engagements:

- Session hijack. Steal the auth cookie or token and replay it from the attacker's browser.
- Action-on-behalf-of-user. Submit forms, transfer funds, change account settings using the victim's session.
- Credential capture. Replace the password field with one that POSTs to the attacker's server.
- Multi-stage compromise. XSS on an admin page leads to admin-account takeover, then to a backdoor across the application.
- Watering hole inside the app. A stored XSS in a popular team page silently captures every employee who visits it.

HOW TO DEFEND

- Context-aware output encoding. When the application writes user content into the page, escape it for the specific context (HTML body, HTML attribute, JavaScript string, URL parameter). Modern frameworks (React, Vue, Angular) do this by default for most cases. Audit any place the framework's default is bypassed (`dangerouslySetInnerHTML-`, `v-html`, `bypassSecurityTrustHtml`, raw template literals).
- Content Security Policy (CSP). A header that tells the browser which sources may load scripts. A strict CSP turns most XSS findings from full compromise into low-severity. Worth doing even if existing code is well-escaped.
- Input validation at the boundary. Reject inputs that contain characters that have no business being there (a phone number field should not accept HTML tags). Secondary defense layer.

SOURCES

[1] OWASP A03:2021 Injection

[2] OWASP Cross Site Scripting Prevention Cheat Sheet

[3] CWE-79: Cross-Site Scripting

[4] MDN Content Security Policy

03 How does SecureLayer7 test for XSS?

Every web application engagement covers XSS in three layers.

- Automated layer. Fuzz every input field and URL parameter for common payload shapes. Catches reflected XSS on unauthenticated endpoints fast.
- Manual layer. Test stored XSS in every place user content gets written and later rendered (profiles, comments, chat, descriptions, file uploads with metadata, email subject lines forwarded to a web view). Test DOM-based XSS in every place client-side code reads URL parameters, hash fragments, postMessage events, or localStorage and writes them into the DOM.
- Bypass layer. When the application has a sanitizer or CSP, we test their specific bypass patterns: encoded payloads, mutation XSS, template injection sneaking around the filter.

Deliverable maps findings to OWASP A03:2021 (Injection) and includes the specific encoding or CSP change required to fix each one.

Find XSS before someone else does.

securelayer7.net/learn/application-security/cross-site-scripting

Open online