GRIDINSOFT
HELP CENTER

OFFICIAL SUPPORT GUIDE

X

Cross-Site Scripting (XSS): Types, Impact, and Prevention

Cross-site scripting lets untrusted data execute as browser code in a trusted site. Learn the major XSS types, impact, testing principles, and effective defenses.

PUBLISHER	UPDATED	READ TIME
Gridinsoft Support	Oct 8, 2026	5 min

CONTENTS

In this guide

01 How XSS works

02 Reflected, stored, and DOM-based XSS

03 What an attacker may achieve

04 Preventing XSS correctly

05 Common defenses that are insufficient

06 Testing and responding

07 Frequently asked questions

Quick answer: Cross-site scripting (XSS) is a web vulnerability in which untrusted data is interpreted as active browser code inside a trusted site. A successful attack runs with the affected site's origin and can alter what a user sees or does. The main defenses are safe framework APIs, context-aware output encoding, HTML sanitization when markup is allowed, and avoiding dangerous browser sinks. Content Security Policy (CSP) is valuable defense in depth, not a substitute for fixing injection.

01 How XSS works

Web applications routinely combine templates, user input, database content, URLs, and third-party data. XSS appears when the application places untrusted data into HTML, an attribute, JavaScript, CSS, or a URL without the protection required for that exact context. The browser cannot tell that the resulting instructions were not intended by the site owner.

The name is historical: an attack does not need to cross between two websites. The essential issue is that data crosses a boundary and becomes executable content. The OWASP XSS overview explains current terminology and the relationship between server-side and client-side injection.

02 Reflected, stored, and DOM-based XSS

- **Reflected XSS:** malicious input arrives in a request and is immediately included in the response. A victim may have to open a crafted link or submit a manipulated form.
- **Stored XSS:** untrusted content is saved, for example in a profile, ticket, comment, or product record, and later rendered to one or many users.
- **DOM-based XSS:** client-side JavaScript reads attacker-controlled data and sends it to an unsafe DOM API. The vulnerable transformation can occur entirely in the browser.

These categories can overlap. "Stored versus reflected" describes where input persists, while "server versus client" describes where injection happens. During investigation, document both the data source and the execution sink rather than forcing every issue into one label.

03 What an attacker may achieve

Impact depends on the affected page, user privileges, browser controls, and application design. XSS can modify page content, submit actions as the signed-in user, read data available to page scripts, capture form input, redirect visitors, or impersonate the interface. An XSS flaw in an administrative console can be much more severe than the same primitive on an isolated public page.

An HttpOnly cookie cannot be read by JavaScript, but XSS may still send authenticated requests from the victim’s browser. SameSite cookies and anti-CSRF tokens address other risks and do not generally cure XSS. Avoid absolute claims such as “XSS always steals cookies” or “HttpOnly prevents XSS.”

04 Preventing XSS correctly

1. **Use framework auto-escaping.** Keep templates in their safe default mode and review any feature that renders raw HTML or bypasses escaping.
2. **Encode for the output context.** HTML text, HTML attributes, URLs, CSS, and JavaScript require different handling. One generic replacement function is not enough.
3. **Sanitize allowed HTML.** If users need rich text, use a maintained allow-list sanitizer designed for that markup. Encoding everything would display markup as text; accepting it unchanged would be unsafe.
4. **Use safe DOM APIs.** Prefer text-only setters and create elements through DOM methods. Avoid sending untrusted strings to APIs that interpret HTML or code.
5. **Validate URLs and attributes.** Allow expected schemes and values rather than assuming encoding makes every destination safe.
6. **Deploy CSP as a backstop.** A nonce- or hash-based policy can limit script execution and produce violation reports, but weak allow-lists and unsafe inline code reduce its value.

The OWASP XSS Prevention Cheat Sheet provides rules for each output context. Apply them at the final rendering boundary; early transformations can be decoded or placed into a different context later.

05 Common defenses that are insufficient

- Blocking a short list of tags or words; browsers have many parsing contexts and encodings.
- Encoding input before storing it; the same stored value may later be used in HTML, JavaScript, JSON, email, or logs.
- Relying on a web application firewall; it can reduce commodity probes but lacks full application context.
- Using CSP alone; a policy may contain bypasses and does not correct unsafe data handling.
- Using the obsolete `X-XSS-Protection` header as the primary control; modern protection should not depend on legacy browser filters.

06 Testing and responding

Test only applications you own or are authorized to assess. Review data flows from request, storage, and browser-controlled sources into templates and DOM sinks. Combine code review, framework-specific checks, automated scanning, and manual validation with harmless proof strings. Test different roles because stored content may execute only for moderators or administrators.

If XSS is confirmed, remove or neutralize malicious stored content, fix the unsafe sink, deploy the corrected code, and review CSP and application logs for exploitation. Consider revoking sessions if sensitive pages were affected. Determine which users viewed the payload and what their roles could access. A patch closes the flaw but does not undo actions already performed through a victim’s session.

07 Frequently asked questions

Is XSS the same as CSRF?

No. XSS injects executable content into a trusted origin. CSRF causes a browser to send an unwanted authenticated request. XSS can often bypass CSRF defenses because it runs inside the site.

Does input validation stop XSS?

Validation reduces unexpected data, but context-aware output handling is still required. Valid text can contain characters that become meaningful in a later rendering context.

Can modern frameworks prevent all XSS?

They reduce risk through default escaping, but raw HTML features, unsafe DOM APIs, third-party components, and incorrect contexts can reintroduce it.

NEED MORE HELP?

We are here when the guide is not enough.

Visit the Gridinsoft Support Center for current product guidance or submit a support request with the details of your case.

Open this guide online at support.gridinsoft.com