XSS, CSRF, SSRF: Three Ways to Make Something Else Do the Asking

Wed, Sep 23, 6:00 PM EDT · https://scottslab.io/posts/xss-csrf-and-ssrf

> **TL;DR**
>
> Cross-site scripting, cross-site request forgery, and server-side request forgery are three different bugs with one shared trick: the attacker never makes the request directly. They convince a browser or a server that already has trust, a session, or network access to make it for them. Defend all three the same way: never let a trusted party act on unvalidated input or instructions from an untrusted place.

The trick they all share

Say it up front, because it's the one idea worth carrying out of this article: an attacker who can't reach your data directly will look for something that can, and trick it into asking on their behalf. A victim's browser has a valid session cookie the attacker doesn't have. Your server has network access to internal systems the attacker can't reach from outside. XSS, CSRF, and SSRF are three different ways of hijacking that borrowed trust instead of trying to forge it from scratch.

XSS: hijacking the browser that's already logged in

Cross-site scripting embeds a script tag inside content a site accepts from one user and displays to another: a comment field, a profile name, a search box reflected back on the results page. The browser loading the page has no way to know the script came from an attacker rather than the site itself, so it runs silently, with whatever access and cookies the victim's session already has.

That's what makes XSS worse than it looks at first glance. The malicious code never runs as the attacker. It runs as the victim, inside the victim's own authenticated session, which is exactly the access the attacker was missing. The defense is server-side, on output: strip or encode script content before it's rendered back to any other user, rather than trusting whatever validation happened client-side, which an attacker posting straight to the API never goes near.

Encoding isn't one operation. Which scheme applies depends on where the value ends up. HTML encoding replaces characters that carry meaning in markup with their entity form, ampersand notation like `<` for a less-than sign, so a browser renders the character instead of reading it as the start of a tag. URL encoding does the equivalent job for characters inside a URL, percent notation like `%27` for a single quote, so the character travels as inert data in the query string instead of breaking whatever is parsing that URL. A value sitting in an HTML body, inside a tag attribute,

inside a URL, and inside a script block each needs a different encoding pass, and mixing them up is itself a common source of filter bypasses.

You can watch URL encoding happen in real time on any search engine. Type a single quote into the search box and submit it, then look at the address bar: the quote comes back as `%27` inside the query string. The site didn't forward your raw character to whatever handles the search, it encoded it into a form that's safe to carry inside a URL first, which is the same transformation that has to happen to attacker-supplied input before a page ever renders it back.

CSRF: hijacking the cookie that's already there

Cross-site request forgery, pronounced "sea surf," exploits a different piece of browser behavior: an auth cookie stays live across every tab you have open, not just the one where you logged in. If you're signed into your bank in one tab and an attacker's page is open in another, that page can embed something as simple as a hidden image tag pointing at your bank's transfer endpoint. The browser attaches your session cookie automatically, because that's what cookies are for, and the bank's server sees a request from your authenticated session with no way to tell you never clicked anything.

The fix is making the request carry something the attacker's page can't know in advance: a cryptographically strong token generated per exchange and checked on the server before the action runs. Two habits close the gap further. Never let a state-changing action run off a plain HTTP GET, since a GET request is exactly what an image tag or a link can trigger with zero interaction. And auto-logout idle sessions, so a cookie sitting open in a forgotten tab doesn't stay valid indefinitely.

SSRF: hijacking the server itself

Server-side request forgery is the same idea aimed at your own infrastructure instead of a victim's browser. Instead of tricking a person's browser into making a request, the attacker tricks your server into making one. A URL parameter meant to fetch a profile image or preview a link gets pointed somewhere the attacker chose instead, and the server dutifully fetches it, because the server is the one making the outbound call.

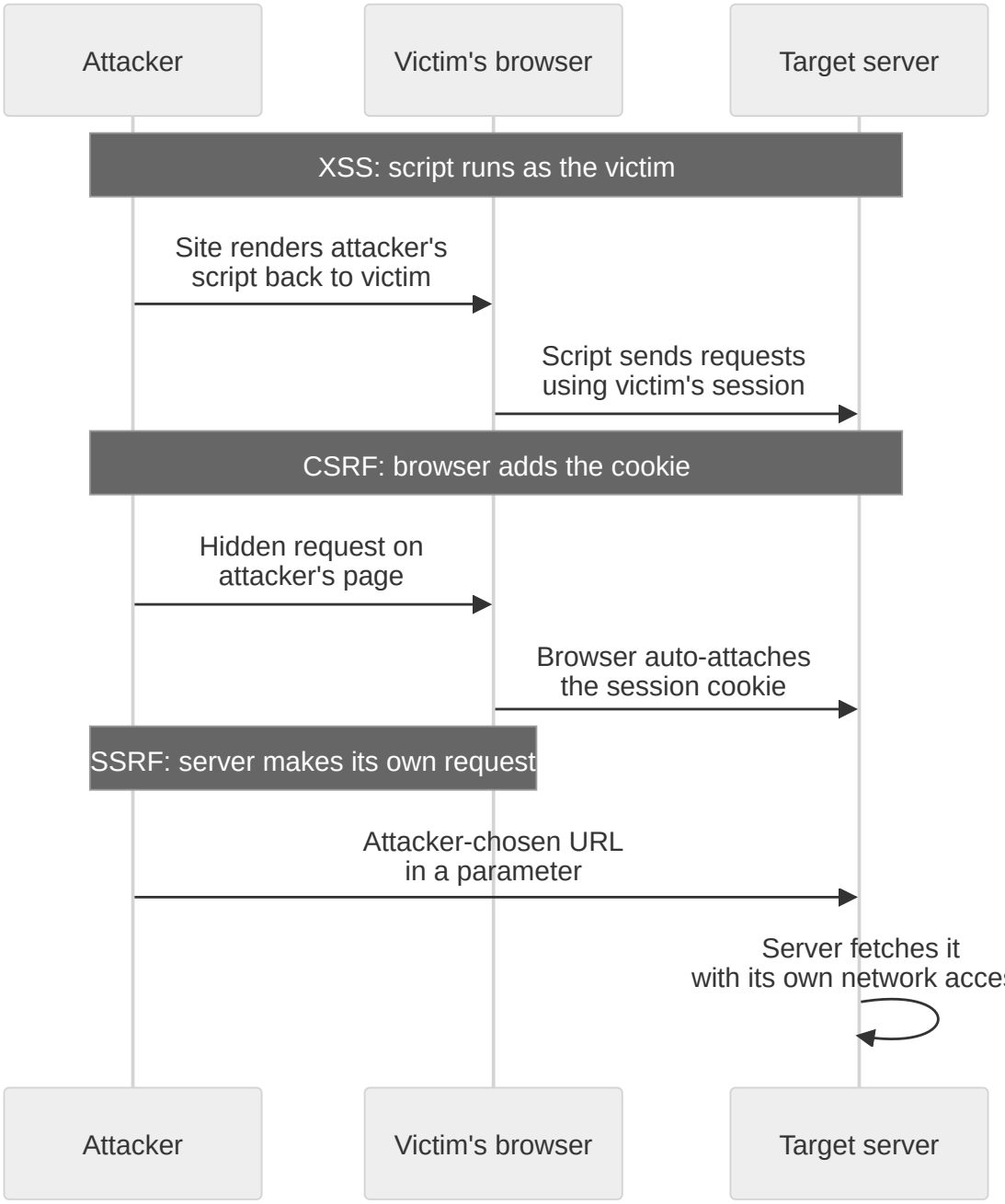
What makes this dangerous is whose position that request runs from. It comes from inside your network, carrying whatever trust internal systems extend to traffic arriving from that host: cloud metadata endpoints, internal admin panels, services that were never given their own auth because they were never meant to be reachable from outside. The server was never supposed to be tricked into asking on the attacker's behalf, and once it is, it's making requests from a network position the attacker could never have reached directly on their own.

The defense mirrors the other two: validate the destination server-side, and don't trust a redirect either, since a URL that passes your check can still redirect somewhere it wouldn't have. Allowlist

the destinations a given feature is actually allowed to fetch rather than trying to blocklist every dangerous one, and block outbound requests to internal and link-local address ranges by default so the server can't be turned into a proxy into its own network just because nobody thought to ask it not to.

Why the naming feels off, until it doesn't

The three names read like a checklist item each, which is part of why they get taught side by side and forgotten as three unrelated acronyms. They're more useful remembered as one move, applied to three different victims. Once you're looking for "who did the asking, and did anyone check," a webhook fetching an attacker-supplied callback URL or a PDF generator rendering attacker-supplied HTML stops looking novel. Same question again: what is this component willing to fetch on someone else's word, and who checked before it did.



Why the naming feels off, until it doesn't

One rule covers all three

The defenses read differently: output encoding, anti-forgery tokens, blocking GET for state changes. But they're the same instruction pointed at three different actors. Never let a browser or a server act on input or instructions from a place you haven't verified, no matter how much implicit trust that browser or server already carries. XSS launders through a victim's browser. CSRF launders through a victim's cookie. SSRF launders through your own server's network position. Different vector, same borrowed authority, same fix: verify before you act, every time, regardless of who's asking.
