Engineer

CSRF

an attack that tricks a browser into an unwanted request

What it is

CSRF, short for cross-site request forgery, is the attack in which a page the victim happens to open makes the victim's own browser send a request to another site, one where they are logged in. The browser attaches that site's cookies as it always does, so the request arrives carrying the victim's session and looks like something they chose to do. The Common Weakness Enumeration catalogues it as CWE-352 and describes the flaw from the server's side: the application does not, or cannot, check well enough whether a request was intentionally made by the user it appears to come from.

The OWASP prevention cheat sheet lists where the attack can start, from a malicious web site to an email, a blog post, an instant message or a program, and what it can achieve: a funds transfer, a changed password, a purchase, a grant of privileges. Each is an action on the site, a change of state, and that is where the cheat sheet puts its defences: a token on every state-changing request. CWE explains that the victim's permissions determine how far the attack can reach, because the attacker borrows the victim's identity. Against an administrator, the result can be complete control of the web application.

CWE treats CSRF as a composite, a weakness that exists only where several others meet. Among them are an origin validation error, CWE-346, and the unintended proxy or "confused deputy", CWE-441: the browser holds the victim's credentials and spends them on behalf of a page that has none. That sets CSRF apart from cross-site scripting, which CWE lists as a peer. With XSS the attacker's code runs inside the target site; with CSRF it stays on the attacker's page and only sends requests. It also sets it apart from CORS, which governs what a page may read from another origin. OWASP points out that a plain HTML form sends a simple request, one that needs no preflight, and that browsers permit such

requests to any origin. CWE's own title spells the term Cross-Site Request Forgery, and the entry lists other names for the same thing, among them XSRF and Session Riding.

Use the concept in a decision

Put "What is CSRF?" 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

CSRF has stayed on both of the best-known weakness lists. CWE-352 was in the CWE Top 25 in 2009, 2010 and 2011, and in every edition from 2019 to 2025. OWASP gave it a category of its own in the Top 10 of 2007 and 2010, as A5, and of 2013, as A8. Since 2021 it has been part of A01, Broken Access Control, which sits at the top of the 2025 edition. OWASP reports that every one of the applications tested had some form of broken access control, and names CWE-352 among its notable weaknesses.

The attack is hard to stamp out because the browser does the work. The W3C Fetch Metadata draft starts from that point: browsers are easily convinced to send requests to a site's endpoints and to include the user's ambient credentials, such as cookies or a privileged position on an intranet, so applications have to be very careful about how those endpoints behave, and being careful turns out to be hard. CWE notes that almost any HTML element can launch the attack, image tags and links included. Its observed examples read like a list of administrative actions carried out from someone else's page: user accounts added through a URL in an `img` tag (CVE-2004-1703), administrator privileges gained the same way (CVE-2004-1842), and a web interface that let a forged request change passwords or stop a virtual machine (CVE-2009-3759).

Scanners are little help. CWE says CSRF is currently difficult to detect reliably with automated techniques, because every application has its own implicit policy about which requests an outsider may cause and which need strong confidence

that the user meant them. A tool cannot know that one form is harmless and the next one moves money. That makes CSRF a design question: decide, request by request, what proves that the user meant it.

Browsers have moved in the defender's favour. OWASP's cheat sheet says current mainstream browsers enforce SameSite=Lax on cookies by default, which keeps them off most cross-site `POST` requests, and that every major browser has sent the Fetch Metadata headers since March 2023. Both help, and neither settles the matter. The cookie draft that defines SameSite calls Lax reasonable defence in depth against attacks that rely on unsafe methods such as `POST`, but not a robust defence against CSRF as a category, and advises that even Strict mode should not be the whole of a site's defence. Users on older or embedded browsers, OWASP adds, may get cookies that behave as if no SameSite value were set at all.

What it looks like in practice

Take a web application at `app.example.com` where a logged-in user changes the email address on their account with a form post to `/account/email`. The session cookie is set with `SameSite=None; Secure`, because the same dashboard is also embedded in a partner's portal, and None is the value that lets a browser send a cookie in a cross-site context. An attacker publishes a page at `prizes.example.net` and gets the victim to open it:

```
<form action="https://app.example.com/account/email" method="post">
  <input type="hidden" name="email" value="mallory@example.net">
</form>
<script>document.forms[0].submit();</script>
```

The browser submits the form as soon as the page loads, and since the cookie allows it, the session goes along:

```
POST /account/email HTTP/1.1
Host: app.example.com
Origin: https://prizes.example.net
Sec-Fetch-Site: cross-site
Content-Type: application/x-www-form-urlencoded
Cookie: session=...

email=mallory%40example.net
```

To the server this is a logged-in user changing their address. If nothing else is checked, the address changes, and a password reset sent to the new one hands the account to the attacker.

Two headers in that request belong to the browser, not to the attacker. The Fetch Standard lists `Origin` among the forbidden request headers, along with every header whose name starts with `Sec-`, so a script on the page cannot set or change them. `Origin` names the origin that sent the request, and the Fetch Metadata draft defines four values for `Sec-Fetch-Site`: `cross-site`, `same-site`, `same-origin`, and `none` for a navigation the user started, by typing an address or choosing a bookmark. A check can lean on both, and still fall back to a token for browsers that send neither. A sketch in Python:

```
import hmac
import secrets

SAFE_METHODS = {"GET", "HEAD", "OPTIONS", "TRACE"}
OUR_ORIGIN = "https://app.example.com"


def new_csrf_token(session: dict) -> str:
    session["csrf_token"] = secrets.token_urlsafe(32)
    return session["csrf_token"]


def csrf_ok(method: str, headers: dict, form: dict, session: dict) -> bool:
    if method in SAFE_METHODS:
        return True
    if headers.get("Sec-Fetch-Site") == "cross-site":
        return False
    origin = headers.get("Origin")
    if origin is not None and origin != OUR_ORIGIN:
        return False
    expected = session.get("csrf_token", "").encode()
    sent = form.get("csrf_token", "").encode()
    return bool(expected) and hmac.compare_digest(sent, expected)
```

The token comes from `secrets.token_urlsafe(32)`. Python’s documentation says the `secrets` module draws on the most secure source of randomness the operating system provides, and that 32 bytes, 256 bits, is believed to be enough. The server puts the token in a hidden field in every form it renders and keeps a copy in the session. It never puts it in a cookie, a URL or a log, which are the places OWASP’s cheat sheet rules out for a synchronizer token.

`hmac.compare_digest` is designed not to stop at the first byte that differs, so the

time a comparison takes reveals nothing about how much of a guess was right. Both sides are encoded to bytes first, because the function accepts text strings only when they are pure ASCII. A request with no `Origin` header falls through to the token, and `Origin: null` fails, since only the exact origin passes.

In a real application, reach for the framework's protection before writing your own; that is the first item on OWASP's list. Django's middleware, for example, requires both its CSRF cookie and a `csrfmiddlewaretoken` field on every method other than `GET`, `HEAD`, `OPTIONS` and `TRACE`, answers 403 when either is missing or wrong, and checks `Origin` against the host and the `CSRF_TRUSTED_ORIGINS` setting. Where nothing needs to embed the application, the session cookie can also go out as `__Host-session=...; Path=/; Secure; SameSite=Lax`. OWASP notes that a cookie with the `__Host-` prefix cannot be overwritten from another subdomain.

Browser settings are a layer, not the defence. Our write-up on hardening the application with nonce-based CSP notes that the proxy middleware sets HSTS and SameSite cookies alongside the Content Security Policy, and that entitlements are re-checked on the server on every request, with the frontend gates treated as UX only.

Where it goes wrong

State changes behind GET. SameSite=Lax still sends the cookie on a cross-site top-level navigation with a safe method, so a link the victim clicks, or a page that sends the whole window to `https://app.example.com/account/delete?confirm=yes`, arrives logged in. OWASP calls this the most common way SameSite-based defences fail in practice, and Django builds its protection on the same premise: `GET` and the other safe methods should have no side effects. Keep `GET`, `HEAD`, `OPTIONS` and `TRACE` free of side effects, and put every change behind a method your checks cover.

SameSite as the whole defence. The cookie draft explains why Lax is not enough on its own: an attacker can still open a new window or trigger a top-level navigation to make a request count as same-site, and prerendering can do the same without the user noticing. OWASP adds that SameSite looks at the registrable domain rather than the full origin. A request from any host under example.com counts as same-site for `app.example.com`, so a forgotten subdomain that someone takes over is already inside the fence. Treat SameSite as a second layer, and decide on the server with a token or an origin check.

A token the attacker can plant. In the naive double-submit pattern the server only checks that a cookie and a form field carry the same random value. OWASP now marks that pattern as discouraged, because anyone who can write cookies on the domain, through a vulnerable sibling subdomain, a DNS takeover or a cookie injected over plain HTTP, can set both halves. CWE's entry still lists the plain double-submitted cookie among its mitigations, so the two sources disagree here, and OWASP's warning is the one to follow. Keep the token in the server-side session, or use OWASP's signed double-submit cookie, bound to the session with an HMAC.

Cross-site scripting on the same site. A script that runs inside your own pages can read the token from the form and send a request that is same-origin by every measure. OWASP's cheat sheet puts it plainly: XSS "can defeat all CSRF mitigation techniques". CWE and Django say the same, and CWE's best-known example shows the two working together. The Samy worm used XSS to plant itself in a MySpace profile and add the attacker as a friend, and friends who viewed the profile had their own profiles changed the same way; since no victim chose to make that change, CWE counts CSRF as a root cause. Treat every XSS finding as a CSRF finding too, and fix it first.

An origin check that matches too much. A test like `origin.startswith("https://app.example.com")` also accepts `https://app.example.com.attacker.example`, the trap OWASP's cheat sheet warns about. The value `null` is another. Under the Fetch Standard, a form posted from a page whose referrer policy is `no-referrer` arrives with `Origin: null`, and OWASP adds that redirects across origins and sandboxed frames can produce it too. Compare the whole origin with a fixed list of exact strings, and treat a missing header or `null` as unknown, never as your own origin.

Trusting the preflight too far. A custom request header is a sound defence for an API, because browsers send a preflight before any cross-origin request that carries one. It only holds if the API refuses everything else. OWASP warns that a JSON API which also accepts `text/plain` can be reached by a form, since that is one of the three content types a simple request may use, and that a CORS policy allowing credentials from every subdomain through a regular expression lets anyone who takes over a subdomain send the custom header. Reject content types the endpoint does not expect, and list the exact origins that may send credentialed requests.

Forgetting the login form. Login CSRF runs the other way round: the attacker signs the victim in to the attacker's own account. On a shopping site, OWASP's example goes, the victim then enters card details, and the attacker can later buy things with them. Django's documentation names login CSRF among the attacks its protection covers. Protect the login form with a token tied to a pre-session, and destroy that session and start a new one once the user has signed in, as OWASP advises.

Requests built from attacker input. Client-side CSRF happens when the application's own JavaScript builds a request from input the attacker controls, such as the URL, the window name or a `postMessage`. The script adds the token itself and the request is same-site, so neither tokens nor SameSite stop it; OWASP describes it as the confused deputy coming back. Keep request targets and parameters out of reach of attacker-controlled input, or choose them from a fixed list in the code.

Further reading

- CWE-352: Cross-Site Request Forgery (CSRF) — the weakness entry: the composite structure, consequences, mitigations and observed CVEs.
- OWASP Cheat Sheet Series: Cross-Site Request Forgery Prevention — tokens, Fetch Metadata, origin checks, the limits of SameSite and login CSRF.
- W3C: Fetch Metadata Request Headers — the `Sec-Fetch-Site` header and the four values a browser can send.
- IETF draft: Cookies: HTTP State Management Mechanism (draft-ietf-httpbis-rfc6265bis) — the SameSite attribute and why Lax is defence in depth rather than a cure; still an Internet-Draft.
- Django documentation: Cross Site Request Forgery protection — a framework's built-in protection, what it assumes and where subdomains get round it.
- OWASP Top 10:2025: A01 Broken Access Control — where CWE-352 sits in the 2025 edition.

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 CSRF?” 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/csrf/