Engineer

CSRF

en attack som får webbläsaren att skicka en oönskad begäran

Vad det är

CSRF, kort för cross-site request forgery, är attacken där en sida som offret råkar öppna får offrets egen webbläsare att skicka ett anrop till en annan webbplats, en där offret är inloggat. Webbläsaren bifogar den webbplatsens kakor som den alltid gör, så anropet kommer fram med offrets session och ser ut som något offret själv valt att göra. Common Weakness Enumeration för upp den som CWE-352 och beskriver felet från serverns sida: applikationen kontrollerar inte, eller kan inte kontrollera, tillräckligt väl om ett anrop medvetet skickades av den användare det verkar komma från.

OWASP:s fusklapp om att förebygga CSRF (cheat sheet) räknar upp var attacken kan börja, från en illasinnad webbplats till ett mejl, ett blogginlägg, ett snabbmeddelande eller ett program, och vad den kan åstadkomma: en överföring av pengar, ett ändrat lösenord, ett köp, tilldelade privilegier. Var och en är en handling på webbplatsen, en ändring av tillstånd, och det är där fusklappen sätter sitt försvar: en token på varje anrop som ändrar tillstånd. CWE tillägger att omfattningen bara begränsas av offrets privilegier, eftersom angriparen lånar offrets identitet. Riktad mot en administratör kan följden bli full kontroll över webbapplikationen.

CWE behandlar CSRF som en sammansatt svaghet, en som bara finns där flera andra möts. Bland dem finns ett fel i valideringen av ursprung, CWE-346, och den oavsiktliga proxyn eller ”confused deputy”, CWE-441: webbläsaren håller offrets inloggningsuppgifter och spenderar dem åt en sida som saknar egna. Det skiljer CSRF från cross-site scripting, som CWE för upp som en jämbördig svaghet. Vid XSS körs angriparens kod inne på målwebbplatsen; vid CSRF stannar den på angriparens sida och skickar bara anrop. Det skiljer det också från CORS, som styr vad en sida får läsa från ett annat ursprung. OWASP påpekar att ett vanligt HTML-formulär skickar ett enkelt anrop, ett som inte behöver något föranrop

(preflight), och att webbläsare tillåter sådana anrop till vilket ursprung som helst. CWE stavar själv termen Cross-Site Request Forgery i titeln, och posten nämner andra namn på samma sak, bland dem XSRF och Session Riding.

Använd begreppet i ett beslut

Använd ”Vad är CSRF?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Konton, lösenord och behörigheter ordnade så att alla har exakt den åtkomst deras arbete kräver, och de som slutar förlorar den samma dag.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Skrev 1 000 RBAC-regler som säkrar Linux-servrar — 1,000 åtkomstregler, skrivna för att täppa till en säkerhetslucka

Skrev 1 000 RBAC-regler som säkrar Linux-servrar Hör av er →

Varför det spelar roll

CSRF har hållit sig kvar på båda de mest kända listorna över svagheter. CWE-352 fanns med på CWE Top 25 år 2009, 2010 och 2011, och i varje utgåva från 2019 till 2025. OWASP gav den en egen kategori på Top 10 år 2007 och 2010, som A5, och år 2013, som A8. Sedan 2021 ingår den i A01, Broken Access Control, som ligger högst upp i 2025 års utgåva. OWASP rapporterar att varenda en av de testade applikationerna hade någon form av bristande åtkomstkontroll, och nämner CWE-352 bland de mest framträdande svagheterna i kategorin.

Attacken är svår att utrota eftersom det är webbläsaren som gör jobbet. W3C:s utkast om Fetch Metadata utgår just från det: webbläsare låter sig lätt övertalas att skicka anrop till en webbplats ändpunkter och ta med de inloggningsuppgifter användaren redan har, till exempel kakor eller en privilegierad plats på ett intranät, så applikationer måste vara mycket noga med hur de ändpunkterna beter sig, och det visar sig vara svårt. CWE påpekar att attacken kan iscensättas från praktiskt taget vilken tagg eller HTML-konstruktion som helst, bildtaggar och länkar inräknade. De observerade exemplen i posten läses som en lista över administrativa handlingar utförda från någon annans sida: användarkonton som lagts till via en URL i en `img` -tagg (CVE-2004-1703), administratörsbehörighet som skaffats på samma sätt (CVE-2004-1842) och ett webbgränssnitt där ett förfalskat anrop kunde ändra lösenord eller stoppa en virtuell maskin (CVE-2009-3759).

Skannrar hjälper föga. CWE skriver att CSRF i dag är svår att upptäcka tillförlitligt med automatiska tekniker, eftersom varje applikation har sin egen underförstådda policy för vilka anrop en utomstående får orsaka och vilka som

kräver stark visshet om att användaren menade dem. Ett verktyg kan inte veta att ett formulär är ofarligt och att nästa flyttar pengar. Det gör CSRF till en designfråga: avgör, anrop för anrop, vad som bevisar att användaren menade det.

Webbläsarna har rört sig till försvararens fördel. OWASP:s fusklapp säger att dagens vanliga webbläsare som standard tillämpar SameSite=Lax på kakor, vilket håller dem borta från de flesta `POST` -anrop mellan webbplatser, och att alla stora webbläsare har skickat Fetch Metadata-huvudena sedan mars 2023. Båda hjälper, och ingen av dem avgör saken. Utkastet om kakor som definierar SameSite kallar Lax ett rimligt extra försvarslager mot attacker som bygger på osäkra metoder som `POST` , men inte ett robust försvar mot CSRF som kategori, och avråder från att låta ens Strict-läget vara hela webbplatsens försvar. Användare av äldre eller inbyggda webbläsare, tillägger OWASP, kan få kakor som beter sig som om inget SameSite-värde alls var satt.

Så ser det ut i praktiken

Ta en webbapplikation på `app.example.com` där en inloggad användare ändrar e-postadressen på sitt konto med ett formulär som skickas med `POST` till `/account/email` . Sessionskakan är satt med `SameSite=None; Secure` , eftersom samma instrumentpanel också är inbäddad i en partners portal, och None är värdet som låter en webbläsare skicka en kaka i ett sammanhang mellan webbplatser. En angripare publicerar en sida på `prizes.example.net` och får offret att öppna den:

```
<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>
```

Webbläsaren skickar formuläret så snart sidan har laddats, och eftersom kakan tillåter det följer sessionen med:

```
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
```

För servern är detta en inloggad användare som ändrar sin adress. Kontrolleras inget annat ändras adressen, och en återställning av lösenordet som skickas till den nya adressen lämnar över kontot till angriparen.

Två av huvudena i anropet tillhör webbläsaren, inte angriparen. Fetch-standarden räknar `Origin` till de förbjudna anropshuvudena, tillsammans med alla huvuden vars namn börjar med `Sec-`, så ett skript på sidan kan varken sätta eller ändra dem. `Origin` anger ursprunget som skickade anropet, och utkastet om Fetch Metadata definierar fyra värden för `Sec-Fetch-Site`: `cross-site`, `same-site`, `same-origin` och `none` för en navigering som användaren själv startade genom att skriva en adress eller välja ett bokmärke. En kontroll kan luta sig mot båda och ändå falla tillbaka på en token för webbläsare som inte skickar något av dem. En skiss i 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)
```

Token kommer från `secrets.token_urlsafe(32)`. Pythons dokumentation säger att modulen `secrets` hämtar från den säkraste källan till slumpmässighet som operativsystemet erbjuder, och att 32 byte, alltså 256 bitar, anses räcka. Servern lägger token i ett dolt fält i varje formulär den genererar och sparar en kopia i sessionen. Den hamnar aldrig i en kaka, en URL eller en logg, vilket är de platser OWASP:s fusklapp utesluter för en synkroniseringstoken (synchronizer token). `hmac.compare_digest` är utformad för att inte stanna vid den första byte som skiljer sig, så tiden en jämförelse tar avslöjar ingenting om hur stor del av en gissning som var rätt. Båda sidor kodas först till bytes, eftersom funktionen bara tar emot textsträngar när de är ren ASCII. Ett anrop utan `Origin`-huvud faller igenom till token, och `Origin: null` stoppas, eftersom bara det exakta ursprunget släpps igenom.

I en riktig applikation ska du ta ramverkets skydd innan du skriver ett eget; det är första punkten på OWASP:s lista. Djangos middleware kräver till exempel både sin CSRF-kaka och ett `csrfmiddlewaretoken`-fält för alla metoder utom `GET`, `HEAD`, `OPTIONS` och `TRACE`, svarar med 403 när något av dem saknas eller är fel och kontrollerar `Origin` mot värden och inställningen `CSRF_TRUSTED_ORIGINS`. Där inget behöver bädda in applikationen kan sessionskakan också skickas som `__Host-session=...; Path=/; Secure; SameSite=Lax`. OWASP påpekar att en kaka med prefixet `__Host-` inte kan skrivas över från en annan underdomän.

Webbläsarens inställningar är ett lager, inte själva försvaret. Vår beskrivning av hur vi härdade applikationen med nonce-baserad CSP berättar att proxy-middlewaren sätter HSTS och SameSite-cookies tillsammans med Content Security Policy, och att rättigheterna kontrolleras på nytt på servern vid varje anrop, medan spärrarna i frontenden enbart behandlas som UX.

Var det går fel

Tillståndsändringar bakom GET. SameSite=Lax skickar fortfarande kakan vid en navigering på toppnivå mellan webbplatser med en säker metod, så en länk som offret klickar på, eller en sida som skickar hela fönstret till `https://app.example.com/account/delete?confirm=yes`, kommer fram inloggad. OWASP kallar detta det vanligaste sättet som SameSite-baserade försvar fallerar på i praktiken, och Django bygger sitt skydd på samma premiss: `GET` och de andra säkra metoderna ska inte ha några sidoeffekter. Håll `GET`, `HEAD`, `OPTIONS` och `TRACE` fria från sidoeffekter, och lägg varje ändring bakom en metod som dina kontroller täcker.

SameSite som hela försvaret. Utkastet om kakor förklarar varför Lax inte räcker på egen hand: en angripare kan fortfarande öppna ett nytt fönster eller utlösa en navigering på toppnivå för att få ett anrop att räknas som same-site, och förrendering (prerendering) kan göra samma sak utan att användaren märker det. OWASP tillägger att SameSite gäller den registrerbara domänen, inte ursprunget. Ett anrop från vilken värd som helst under example.com räknas som same-site för `app.example.com`, så en bortglömd underdomän som någon tar över är redan innanför stängslet. Behandla SameSite som ett andra lager, och avgör på servern med en token eller en kontroll av ursprunget.

En token som angriparen kan plantera. I det naiva double-submit-mönstret kontrollerar servern bara att en kaka och ett formulärfält bär samma slumpmässiga värde. OWASP avråder nu från det mönstret, eftersom den som kan skriva kakor på domänen, genom en sårbar underdomän bredvid, ett DNS-övertagande eller en kaka som injicerats över vanlig HTTP, kan sätta båda halvorna. CWE-posten nämner fortfarande den enkla dubbelt inskickade kakan bland sina åtgärder, så de två källorna är oense här, och det är OWASP:s varning som ska följas. Håll token i sessionen på serversidan, eller använd OWASP:s signerade double-submit-kaka, bunden till sessionen med en HMAC.

Cross-site scripting på samma webbplats. Ett skript som körs på dina egna sidor kan läsa token från formuläret och skicka ett anrop som är same-origin enligt alla mått. OWASP:s fusklapp säger det rakt ut: XSS kan slå ut alla tekniker för att motverka CSRF. CWE och Django säger samma sak, och CWE:s mest kända exempel visar de två i samverkan. Samy-masken använde XSS för att plantera sig själv i en MySpace-profil och lägga till angriparen som vän, och vänner som tittade på profilen fick sina egna profiler ändrade på samma sätt; eftersom inget av offren valde att göra den ändringen räknar CWE CSRF som en av grundorsakerna. Behandla varje XSS-fynd som ett CSRF-fynd också, och åtgärda det först.

En ursprungskontroll som släpper igenom för mycket. Ett test som `origin.startswith("https://app.example.com")` godtar också `https://app.example.com.attacker.example`, fällan som OWASP:s fusklapp varnar för. Värdet `null` är en annan. Enligt Fetch-standarden kommer ett formulär som skickas från en sida med referrer-policyn `no-referrer` fram med `Origin: null`, och OWASP tillägger att omdirigeringar mellan ursprung och ramar i sandlåda också kan ge det. Jämför hela ursprunget med en fast lista över exakta strängar, och behandla ett saknat huvud eller `null` som okänt, aldrig som ditt eget ursprung.

För stor tillit till föranropet. Ett eget anropshuvud är ett sunt försvar för ett API, eftersom webbläsare skickar ett föranrop (preflight) före varje anrop mellan ursprung som bär ett sådant huvud. Det håller bara om API:et avvisar allt annat. OWASP varnar för att ett JSON-API som också godtar `text/plain` kan nås från ett formulär, eftersom det är en av de tre innehållstyper som ett enkelt anrop får använda, och att en CORS-policy som tillåter inloggningsuppgifter från alla underdomäner via ett reguljärt uttryck låter den som tar över en underdomän skicka det egna huvudet. Avvisa innehållstyper som ändpunkten inte väntar sig, och räkna upp de exakta ursprung som får skicka anrop med inloggningsuppgifter.

Det bortglömda inloggningsformuläret. Login-CSRF går åt andra hållet: angriparen loggar in offret på angriparens eget konto. På en webbutik, enligt OWASP:s exempel, matar offret sedan in sina kortuppgifter, och angriparen kan senare handla med dem. Djangos dokumentation nämner login-CSRF bland de attacker som skyddet täcker. Skydda inloggningsformuläret med en token som är knuten till en försession (pre-session), och förstör den sessionen och starta en ny när användaren har loggat in, som OWASP råder.

Anrop byggda av angriparens indata. CSRF på klientsidan (client-side CSRF) uppstår när applikationens egen JavaScript bygger ett anrop av indata som angriparen styr, till exempel URL:en, fönstrets namn eller ett `postMessage`. Skriptet lägger själv till token och anropet är same-site, så varken tokens eller SameSite stoppar det; OWASP beskriver det som att confused deputy-problemet kommer tillbaka. Håll anropens mål och parametrar utom räckhåll för indata som angriparen styr, eller välj dem från en fast lista i koden.

Läs mer

- CWE-352: Cross-Site Request Forgery (CSRF) — svaghetsposten: den sammansatta strukturen, följderna, åtgärderna och de observerade CVE:erna.
- OWASP Cheat Sheet Series: Cross-Site Request Forgery Prevention — tokens, Fetch Metadata, ursprungskontroller, gränserna för SameSite och login-CSRF.
- W3C: Fetch Metadata Request Headers — huvudet `Sec-Fetch-Site` och de fyra värden en webbläsare kan skicka.
- IETF draft: Cookies: HTTP State Management Mechanism (draft-ietf-httpbis-rfc6265bis) — attributet SameSite och varför Lax är ett extra försvarslager snarare än ett botemedel; fortfarande ett Internet-Draft.

- Django documentation: Cross Site Request Forgery protection — ett ramverks inbyggda skydd, vad det förutsätter och var underdomäner tar sig runt det.
- OWASP Top 10:2025: A01 Broken Access Control — var CWE-352 står i 2025 års utgåva.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med driftsättningsvägen, driftsriskerna och dem som har jour. Vi kan avgränsa en förändring med kontroller, en väg tillbaka och tydligt ansvar.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är CSRF?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/csrf/