Engineer

CSRF

et angrep som lurer en nettleser til en uønsket forespørsel

Hva det er

CSRF, kort for cross-site request forgery, er angrepet der en side offeret tilfeldigvis åpner, får offerets egen nettleser til å sende en forespørsel til et annet nettsted, et der offeret er logget inn. Nettleseren legger ved nettstedets informasjonskapsler slik den alltid gjør, så forespørselen kommer fram med offerets økt og ser ut som noe offeret selv valgte å gjøre. Common Weakness Enumeration fører det som CWE-352 og beskriver feilen fra serverens side: applikasjonen kontrollerer ikke, eller kan ikke kontrollere, godt nok om en forespørsel bevisst ble sendt av den brukeren den ser ut til å komme fra.

OWASPs jukselapp om forebygging av CSRF (cheat sheet) lister opp hvor angrepet kan starte, fra et ondsinnet nettsted til en e-post, et blogginnlegg, en direktemelding eller et program, og hva det kan oppnå: en pengeoverføring, et endret passord, et kjøp, tildelte privilegier. Hver av dem er en handling på nettstedet, en endring av tilstand, og det er der jukselappen setter forsvaret: et token på hver forespørsel som endrer tilstand. CWE legger til at omfanget bare begrenses av offerets privilegier, fordi angriperen låner offerets identitet. Rettet mot en administrator kan resultatet bli full kontroll over webapplikasjonen.

CWE behandler CSRF som en sammensatt svakhet, en svakhet som bare finnes der flere andre møtes. Blant dem er en feil i validering av opprinnelse, CWE-346, og den utilsiktede proxyen eller «confused deputy», CWE-441: nettleseren har offerets påloggingsinformasjon og bruker den på vegne av en side som ikke har noen. Det skiller CSRF fra krysskripting, som CWE fører opp som en likestilt svakhet. Ved XSS kjører angriperens kode inne i målnettstedet; ved CSRF blir den værende på angriperens side og sender bare forespørsler. Det skiller det også fra CORS, som styrer hva en side kan lese fra en annen opprinnelse. OWASP påpeker at et vanlig HTML-skjema sender en enkel forespørsel, en som ikke trenger noen forhåndsforespørsel (preflight), og at nettlesere tillater slike forespørsler til

enhver opprinnelse. CWE staver selv termen Cross-Site Request Forgery i tittelen, og oppføringen nevner andre navn på det samme, blant dem XSRF og Session Riding.

Bruk begrepet i en beslutning

Bruk «Hva er CSRF?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Kontoer, passord og tilganger ordnet slik at folk har nøyaktig den tilgangen jobben krever, og de som slutter, mister den samme dag.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Skrev 1 000 RBAC-regler for å sikre Linux-servere — 1,000 tilgangsregler skrevet for å tette et sikkerhetshull

Skrev 1 000 RBAC-regler for å sikre Linux-servere Ta kontakt →

Hvorfor det er viktig

CSRF har holdt seg på begge de mest kjente listene over svakheter. CWE-352 var med på CWE Top 25 i 2009, 2010 og 2011, og i hver utgave fra 2019 til 2025. OWASP ga det en egen kategori på Top 10 i 2007 og 2010, som A5, og i 2013, som A8. Siden 2021 har det vært en del av A01, Broken Access Control, som står øverst i 2025-utgaven. OWASP melder at hver eneste av de testede applikasjonene hadde en eller annen form for brutt tilgangskontroll, og nevner CWE-352 blant de mest merkbare svakhetene i kategorien.

Angrepet er vanskelig å utrydde fordi det er nettleseren som gjør jobben. W3C-utkastet om Fetch Metadata tar utgangspunkt i nettopp det: nettlesere lar seg lett overtale til å sende forespørsler til et nettsteds endepunkter og ta med påloggingsinformasjonen brukeren allerede har, for eksempel informasjonskapsler eller en privilegert plass på et intranett, så applikasjoner må være svært nøye med hvordan disse endepunktene oppfører seg, og det viser seg å være vanskelig. CWE påpeker at angrepet kan settes i gang fra så å si hvilken som helst tagg eller HTML-konstruksjon, også bildetagger og lenker. De observerte eksemplene i oppføringen leses som en liste over administrative handlinger utført fra en annens side: brukerkontoer lagt til gjennom en URL i en `img` -tagg (CVE-2004-1703), administratorrettigheter skaffet på samme måte (CVE-2004-1842), og et webgrensesnitt der en forfalsket forespørsel kunne endre passord eller stoppe en virtuell maskin (CVE-2009-3759).

Skannere hjelper lite. CWE skriver at CSRF i dag er vanskelig å oppdage pålitelig med automatiske teknikker, fordi hver applikasjon har sin egen underforståtte policy for hvilke forespørsler en utenforstående kan utløse, og hvilke som krever

sterk visshet om at brukeren mente dem. Et verktøy kan ikke vite at ett skjema er harmløst og det neste flytter penger. Det gjør CSRF til et designspørsmål: avgjør, forespørsel for forespørsel, hva som beviser at brukeren mente det.

Nettleserne har flyttet seg i forsvarerens favør. OWASPs jukselapp sier at dagens utbredte nettlesere som standard håndhever SameSite=Lax på informasjonskapsler, noe som holder dem unna de fleste `POST`-forespørsler på tvers av nettsteder, og at alle de store nettleserne har sendt Fetch Metadata-headerne siden mars 2023. Begge deler hjelper, og ingen av dem avgjør saken. Utkastet om informasjonskapsler som definerer SameSite, kaller Lax et rimelig ekstra forsvarslag mot angrep som bygger på usikre metoder som `POST`, men ikke et robust forsvar mot CSRF som kategori, og råder til at ikke engang Strict-modus skal være hele forsvaret til et nettsted. Brukere av eldre eller innebygde nettlesere, legger OWASP til, kan få informasjonskapsler som oppfører seg som om det ikke var satt noen SameSite-verdi i det hele tatt.

Slik ser det ut i praksis

Ta en webapplikasjon på `app.example.com` der en innlogget bruker endrer e-postadressen på kontoen sin med et skjema som sendes med `POST` til `/account/email`. Øktens informasjonskapsel er satt med `SameSite=None; Secure`, fordi det samme dashbordet også er innebygd i en partners portal, og None er verdien som lar en nettleser sende en informasjonskapsel i en kontekst på tvers av nettsteder. En angriper publiserer en side på `prizes.example.net` og får offeret til å åpne 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>
```

Nettleseren sender skjemaet så snart siden er lastet, og siden informasjonskapselen tillater det, følger økten 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
```

For serveren er dette en innlogget bruker som endrer adressen sin. Blir ingenting annet kontrollert, endres adressen, og en tilbakestilling av passordet som sendes til den nye adressen, gir kontoen til angriperen.

To av headerne i denne forespørselen tilhører nettleseren, ikke angriperen. Fetch-standarden regner `Origin` blant de forbudte forespørselsheaderne, sammen med alle headere der navnet begynner med `Sec-`, så et skript på siden kan verken sette eller endre dem. `Origin` oppgir opprinnelsen som sendte forespørselen, og Fetch Metadata-utkastet definerer fire verdier for `Sec-Fetch-Site`: `cross-site`, `same-site`, `same-origin` og `none` for en navigasjon brukeren selv startet ved å skrive inn en adresse eller velge et bokmerke. En kontroll kan lene seg på begge og likevel falle tilbake på et token for nettlesere som ikke sender noen av dem. En skisse 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)
```

Tokenet kommer fra `secrets.token_urlsafe(32)`. Python-dokumentasjonen sier at modulen `secrets` henter fra den sikreste kilden til tilfeldighet operativsystemet tilbyr, og at 32 byte, altså 256 bit, regnes som nok. Serveren legger tokenet i et skjult felt i hvert skjema den lager, og tar vare på en kopi i økten. Tokenet havner aldri i en informasjonskapsel, en URL eller en logg, som er stedene OWASPs jukselapp utelukker for et synkroniseringstoken (synchronizer token). `hmac.compare_digest` er laget for ikke å stoppe ved den første byten som er ulik, så tiden en sammenligning tar, avslører ingenting om hvor mye av en gjetning som var riktig. Begge sider kodes til bytes først, fordi funksjonen bare tar imot tekststrenger når de er rene ASCII. En forespørsel uten `Origin` -header faller gjennom til tokenet, og `Origin: null` stoppes, siden bare den eksakte opprinnelsen slipper gjennom.

I en ekte applikasjon bør du ta i bruk rammeverkets beskyttelse før du skriver din egen; det er det første punktet på OWASPs liste. Mellomvaren i Django krever for eksempel både sin CSRF-informasjonskapsel og et `csrfmiddlewaretoken` -felt for alle metoder unntatt `GET`, `HEAD`, `OPTIONS` og `TRACE`, svarer med 403 når et av dem mangler eller er feil, og kontrollerer `Origin` mot verten og innstillingen `CSRF_TRUSTED_ORIGINS`. Der ingenting trenger å bygge inn applikasjonen, kan øktens informasjonskapsel også sendes som `__Host-session=...; Path=/; Secure; SameSite=Lax`. OWASP påpeker at en informasjonskapsel med prefikset `__Host-` ikke kan overskrives fra et annet underdomene.

Innstillingene i nettleseren er et lag, ikke selve forsvaret. Beskrivelsen vår av hvordan vi herdet applikasjonen med nonce-basert CSP forteller at proxy-mellomvaren i frontenden setter HSTS og SameSite-cookies i tillegg til Content-Security-Policy, og at rettighetene kontrolleres på nytt på serveren ved hver forespørsel, mens sperrene i frontenden bare behandles som UX.

Hvor det går galt

Tilstandsendringer bak GET. SameSite=Lax sender fortsatt informasjonskapselen ved en navigasjon på toppnivå på tvers av nettsteder med en sikker metode, så en lenke offeret klikker på, eller en side som sender hele vinduet til `https://app.example.com/account/delete?confirm=yes`, kommer fram innlogget. OWASP kaller dette den vanligste måten forsvar basert på SameSite svikter på i praksis, og Django bygger beskyttelsen sin på samme premiss: `GET` og de andre sikre metodene skal ikke ha bivirkninger. Hold `GET`, `HEAD`, `OPTIONS` og `TRACE` fri for bivirkninger, og legg hver endring bak en metode kontrollene dine dekker.

SameSite som hele forsvaret. Utkastet om informasjonskapsler forklarer hvorfor Lax ikke er nok alene: en angriper kan fortsatt åpne et nytt vindu eller utløse en navigasjon på toppnivå for å få en forespørsel til å telle som same-site, og forhåndsrendering (prerendering) kan gjøre det samme uten at brukeren merker det. OWASP legger til at SameSite gjelder det registrerbare domenet, ikke opprinnelsen. En forespørsel fra en hvilken som helst vert under example.com regnes som same-site for `app.example.com`, så et glemt underdomene som noen tar over, er allerede innenfor gjerdet. Behandle SameSite som et ekstra lag, og avgjør på serveren med et token eller en kontroll av opprinnelsen.

Et token angriperen kan plante. I det naive double-submit-mønsteret kontrollerer serveren bare at en informasjonskapsel og et skjemafelt har den samme tilfeldige verdien. OWASP fraråder nå det mønsteret, fordi alle som kan skrive informasjonskapsler på domenet, gjennom et sårbart underdomene ved siden av, en DNS-overtakelse eller en informasjonskapsel injisert over vanlig HTTP, kan sette begge halvdelene. CWE-oppføringen nevner fortsatt den enkle dobbelt innsendte informasjonskapselen blant tiltakene sine, så de to kildene er uenige her, og det er OWASPs advarsel man bør følge. Hold tokenet i økten på serversiden, eller bruk OWASPs signerte double-submit-informasjonskapsel, bundet til økten med en HMAC.

Krysskripting på samme nettsted. Et skript som kjører i dine egne sider, kan lese tokenet fra skjemaet og sende en forespørsel som er same-origin etter alle mål. OWASPs jukselapp sier det rett ut: XSS kan sette alle teknikker mot CSRF ut av spill. CWE og Django sier det samme, og CWEs mest kjente eksempel viser de to i samspill. Samy-ormen brukte XSS til å plante seg selv i en MySpace-profil og legge til angriperen som venn, og venner som så på profilen, fikk sine egne profiler endret på samme måte; siden ingen av ofrene valgte å gjøre den endringen, regner CWE CSRF som en av de grunnleggende årsakene. Behandle hvert XSS-funn som et CSRF-funn også, og rett det først.

En kontroll av opprinnelse som slipper gjennom for mye. En test som `origin.startswith("https://app.example.com")` godtar også `https://app.example.com.attacker.example`, fellen OWASPs jukselapp advarer mot. Verdien `null` er en annen. Etter Fetch-standarden kommer et skjema sendt fra en side med referrer-policyen `no-referrer` fram med `Origin: null`, og OWASP legger til at omdirigeringer på tvers av opprinnelser og rammer i sandkasse også kan gi den. Sammenlign hele opprinnelsen med en fast liste over eksakte strenger, og behandle en manglende header eller `null` som ukjent, aldri som din egen opprinnelse.

For stor tillit til forhåndsforespørselen. En egendefinert forespørselsheader er et solid forsvar for et API, fordi nettlesere sender en forhåndsforespørsel (preflight) før enhver forespørsel på tvers av opprinnelser som har en slik header. Det holder bare hvis API-et avviser alt annet. OWASP advarer om at et JSON-API som også godtar `text/plain`, kan nås fra et skjema, siden det er en av de tre innholdstypene en enkel forespørsel kan bruke, og at en CORS-policy som tillater påloggingsinformasjon fra alle underdomener gjennom et regulært uttrykk, lar alle som tar over et underdomene, sende den egendefinerte headeren. Avvis innholdstyper endepunktet ikke forventer, og list opp de eksakte opprinnelsene som får sende forespørsler med påloggingsinformasjon.

Det glemte innloggingsskjemaet. Login-CSRF går motsatt vei: angriperen logger offeret inn på angriperens egen konto. På en nettbutikk, slik OWASPs eksempel lyder, legger offeret deretter inn kortopplysningene sine, og angriperen kan senere handle med dem. Django-dokumentasjonen nevner login-CSRF blant angrepene beskyttelsen dekker. Beskytt innloggingsskjemaet med et token knyttet til en forhåndsøkt (pre-session), og ødelegg den økten og start en ny når brukeren har logget inn, slik OWASP anbefaler.

Forespørsler bygd av angriperens input. CSRF på klientsiden (client-side CSRF) oppstår når applikasjonens egen JavaScript bygger en forespørsel av input angriperen kontrollerer, for eksempel URL-en, vindusnavnet eller en `postMessage`. Skriptet legger selv ved tokenet, og forespørselen er same-site, så verken tokener eller SameSite stopper den; OWASP beskriver det som at confused deputy-problemet kommer tilbake. Hold mål og parametere for forespørsler utenfor rekkevidden til input angriperen kontrollerer, eller velg dem fra en fast liste i koden.

Les mer

- CWE-352: Cross-Site Request Forgery (CSRF) — svakhetsoppføringen: den sammensatte strukturen, konsekvensene, tiltakene og de observerte CVE-ene.
- OWASP Cheat Sheet Series: Cross-Site Request Forgery Prevention — tokener, Fetch Metadata, kontroll av opprinnelse, grensene for SameSite og login-CSRF.
- W3C: Fetch Metadata Request Headers — headeren `Sec-Fetch-Site` og de fire verdiene en nettleser kan sende.
- IETF draft: Cookies: HTTP State Management Mechanism (draft-ietf-httpbis-rfc6265bis) — attributtet SameSite og hvorfor Lax er et ekstra forsvarslag og ikke en kur; fortsatt et Internet-Draft.

- Django documentation: Cross Site Request Forgery protection — et rammeverks innebygde beskyttelse, hva den forutsetter, og hvor underdomener kommer rundt den.
- OWASP Top 10:2025: A01 Broken Access Control — hvor CWE-352 står i 2025-utgaven.

Ordliste

Velg ut fra deres faktiske rammer

Ta med utrullingsløpet, driftsrisikoene og dem som har vakt. Vi kan avgrense en endring med kontroller, en vei tilbake og tydelig ansvar.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er CSRF?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/csrf/