Engineer

CSRF

amener un navigateur à envoyer une requête non voulue

Ce que c’est

CSRF, pour cross-site request forgery, est l’attaque dans laquelle une page ouverte par la victime fait envoyer à son navigateur une requête vers un autre site, où elle est connectée. Le navigateur y joint les cookies de ce site comme d’habitude ; la requête arrive donc avec la session de la victime et paraît résulter de son choix. La Common Weakness Enumeration la répertorie sous CWE-352 et décrit la faille côté serveur : l’application ne vérifie pas, ou ne peut pas vérifier suffisamment, que la requête a été intentionnellement émise par l’utilisateur dont elle semble provenir.

La fiche de prévention d’OWASP énumère les points de départ, d’un site malveillant à un courriel, un billet de blog, un message instantané ou un programme, et les résultats possibles : transfert de fonds, changement de mot de passe, achat, octroi de privilèges. Chaque cas est une action sur le site, un changement d’état ; c’est là que la fiche place les défenses : un jeton sur toute requête qui change l’état. CWE précise que les permissions de la victime déterminent la portée, car l’attaquant emprunte son identité. Face à un administrateur, le résultat peut être la prise de contrôle complète de l’application web.

CWE traite CSRF comme une faiblesse composite, qui n’existe qu’à la rencontre de plusieurs autres. On y trouve notamment l’erreur de validation d’origine, CWE-346, et le proxy involontaire ou « adjoint confus », CWE-441 : le navigateur détient les informations d’authentification de la victime et les emploie pour une page qui n’en possède aucune. Cela distingue CSRF du cross-site scripting, cité par CWE comme une faiblesse de même niveau. Avec XSS, le code de l’attaquant tourne dans le site cible ; avec CSRF, il reste sur la page de l’attaquant et ne fait qu’envoyer des requêtes. C’est aussi différent de CORS, qui régit ce qu’une page peut lire depuis une autre origine. OWASP rappelle qu’un simple formulaire HTML envoie une requête simple, sans preflight, et que les navigateurs

permettent ces requêtes vers toute origine. Le titre de CWE développe le terme en Cross-Site Request Forgery, et l'entrée donne d'autres noms du même phénomène, dont XSRF et Session Riding.

Utilisez le concept pour prendre une décision

Utilisez « Que signifie CSRF ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des comptes, mots de passe et droits attribués selon les besoins exacts de chaque poste, puis retirés le jour du départ.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 1 000 règles RBAC pour sécuriser Linux — 1,000 règles d'accès écrites pour combler une faille de sécurité

1 000 règles RBAC pour sécuriser Linux Contactez-nous →

Pourquoi c'est important

CSRF est resté dans les deux listes de faiblesses les plus connues. CWE-352 figurait au CWE Top 25 en 2009, 2010 et 2011, puis dans chaque édition de 2019 à 2025. OWASP lui a réservé une catégorie du Top 10 en 2007 et 2010, sous A5, puis en 2013, sous A8. Depuis 2021, il fait partie d'A01, Broken Access Control, en tête de l'édition 2025. OWASP rapporte que toutes les applications testées présentaient une forme de contrôle d'accès défaillant et cite CWE-352 parmi les faiblesses notables.

L'attaque est difficile à éliminer parce que le navigateur fait le travail. Le projet W3C Fetch Metadata part de ce constat : il est facile de convaincre les navigateurs d'envoyer des requêtes aux points d'accès d'un site et d'y joindre les moyens d'accès implicites de l'utilisateur, comme des cookies ou une position privilégiée sur un intranet. Les applications doivent donc être très attentives au comportement de ces points d'accès, ce qui s'avère difficile. CWE note que presque tout élément HTML peut lancer l'attaque, y compris les images et les liens. Les exemples observés ressemblent à une liste d'actions d'administration effectuées depuis la page d'un tiers : ajout de comptes par une URL dans une balise `img` (CVE-2004-1703), obtention de privilèges administrateur de la même manière (CVE-2004-1842), et interface web permettant à une requête forgée de changer des mots de passe ou d'arrêter une machine virtuelle (CVE-2009-3759).

Les scanners aident peu. CWE indique que CSRF reste difficile à détecter de manière fiable par des techniques automatisées, car chaque application a sa politique implicite sur les requêtes qu'un tiers peut provoquer et celles qui exigent une forte assurance de l'intention de l'utilisateur. Un outil ne peut savoir

qu'un formulaire est inoffensif tandis que le suivant transfère de l'argent. CSRF est donc une question de conception : décidez, requête par requête, ce qui prouve l'intention de l'utilisateur.

Les navigateurs ont évolué en faveur des défenseurs. La fiche OWASP indique que les principaux navigateurs actuels appliquent SameSite=Lax aux cookies lorsqu'aucune valeur SameSite valide n'est définie, ce qui les exclut de la plupart des requêtes `POST` intersites, et que tous les grands navigateurs envoient les en-têtes Fetch Metadata depuis mars 2023. Ces deux évolutions aident, sans régler la question. Le projet de norme des cookies, qui définit SameSite, présente Lax comme une défense en profondeur raisonnable contre les attaques utilisant des méthodes non sûres comme `POST`, pas comme une défense robuste contre toute la catégorie CSRF. Il recommande que même le mode Strict ne constitue pas toute la défense du site. OWASP ajoute que, dans des navigateurs anciens ou embarqués, les cookies peuvent se comporter comme si aucune valeur SameSite n'était définie.

À quoi cela ressemble en pratique

Prenez une application à `app.example.com` où un utilisateur connecté change l'adresse courriel de son compte par un formulaire POST vers `/account/email`. Le cookie de session porte `SameSite=None; Secure`, car le même tableau de bord est intégré au portail d'un partenaire, et None permet l'envoi du cookie dans un contexte intersite. Un attaquant publie une page à `prizes.example.net` et fait en sorte que la victime l'ouvre :

```
<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>
```

Le navigateur soumet le formulaire dès le chargement ; puisque le cookie le permet, la session accompagne la requête :

```
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
```

Le serveur voit un utilisateur connecté changer son adresse. Sans autre vérification, l'adresse change, et une réinitialisation de mot de passe envoyée à la nouvelle adresse remet le compte à l'attaquant.

Deux en-têtes de cette requête appartiennent au navigateur, pas à l'attaquant. Le Fetch Standard place `Origin` parmi les en-têtes interdits aux scripts, ainsi que tout en-tête dont le nom commence par `Sec-` ; le script de la page ne peut donc ni les définir ni les modifier. `Origin` nomme l'origine qui a envoyé la requête, et le projet Fetch Metadata définit quatre valeurs de `Sec-Fetch-Site` : `cross-site`, `same-site`, `same-origin` et `none` pour une navigation initiée par l'utilisateur, en saisissant une adresse ou en choisissant un favori. Un contrôle peut s'appuyer sur les deux et se rabattre sur un jeton pour les navigateurs qui n'envoient ni l'un ni l'autre. Voici une esquisse en 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)
```

Le jeton vient de `secrets.token_urlsafe(32)`. La documentation Python dit que `secrets` utilise la source aléatoire la plus sûre fournie par le système d'exploitation et que 32 octets, soit 256 bits, sont considérés comme suffisants. Le serveur place le jeton dans un champ masqué de chaque formulaire rendu et en conserve une copie en session. Il ne le met jamais dans un cookie, une URL ou un journal, destinations exclues par OWASP pour un jeton de synchronisation. `hmac.compare_digest` est conçu pour ne pas s'arrêter au premier octet différent ; la durée de comparaison ne révèle donc rien de la proportion correcte d'une tentative. Les deux côtés sont d'abord encodés en octets, car la fonction n'accepte les chaînes de texte que si elles sont entièrement ASCII. Une requête sans en-tête `Origin` passe au contrôle du jeton, et `Origin: null` échoue, car seule l'origine exacte est acceptée.

Dans une véritable application, utilisez la protection du framework avant d'en écrire une ; c'est le premier conseil d'OWASP. Le middleware de Django, par exemple, exige à la fois son cookie CSRF et un champ `csrfmiddlewaretoken` pour toute méthode autre que `GET`, `HEAD`, `OPTIONS` et `TRACE`, répond 403 si l'un manque ou est incorrect, et compare `Origin` à l'hôte et au réglage `CSRF_TRUSTED_ORIGINS`. Si rien n'a besoin d'intégrer l'application, le cookie de session peut aussi être envoyé comme `__Host-session=...; Path=/; Secure; SameSite=Lax`. OWASP note qu'un cookie préfixé `__Host-` ne peut pas être écrasé depuis un autre sous-domaine.

Les réglages du navigateur sont une couche, pas la défense. Notre récit du durcissement de l'application avec une CSP fondée sur un nonce indique que le middleware proxy définit HSTS et des cookies SameSite avec la Content Security Policy, que les droits sont revérifiés sur le serveur à chaque requête et que les contrôles du frontend ne concernent que l'expérience utilisateur.

Où les problèmes surviennent

GET provoque des changements d'état. SameSite=Lax envoie toujours le cookie lors d'une navigation intersite de premier niveau avec une méthode sûre ; un lien cliqué par la victime, ou une page qui dirige toute la fenêtre vers `https://app.example.com/account/delete?confirm=yes`, arrive donc avec sa session. OWASP en fait la manière la plus courante dont les défenses SameSite échouent en pratique ; Django fonde sa protection sur la même prémisse : `GET` et les autres méthodes sûres ne devraient avoir aucun effet de bord. Gardez `GET`, `HEAD`, `OPTIONS` et `TRACE` sans effets de bord et réservez chaque changement à une méthode couverte par vos contrôles.

SameSite constitue toute la défense. Le projet des cookies explique pourquoi Lax ne suffit pas seul : l'attaquant peut encore ouvrir une fenêtre ou déclencher une navigation de premier niveau pour qu'une requête soit considérée comme same-site, et le prérendu peut le faire sans que l'utilisateur le remarque. OWASP ajoute que SameSite se fonde sur le domaine enregistrable, pas l'origine complète. Une requête venant de n'importe quel hôte sous example.com est considérée comme same-site pour `app.example.com` ; un sous-domaine oublié dont quelqu'un prend le contrôle est donc déjà dans l'enceinte. Traitez SameSite comme une seconde couche et décidez sur le serveur par un jeton ou un contrôle d'origine.

L'attaquant peut déposer le jeton. Dans le schéma naïf de double soumission, le serveur vérifie seulement qu'un cookie et un champ de formulaire portent la même valeur aléatoire. OWASP le déconseille désormais : quiconque peut écrire des cookies sur le domaine, par un sous-domaine voisin vulnérable, une prise de contrôle DNS ou un cookie injecté sur HTTP en clair, peut définir les deux moitiés. CWE cite encore le cookie de double soumission non signé parmi ses mesures d'atténuation ; les deux sources divergent donc ici, et c'est l'avertissement d'OWASP qu'il faut suivre. Conservez le jeton dans la session côté serveur ou utilisez le cookie de double soumission signé d'OWASP, lié à la session par un HMAC.

Le même site présente du cross-site scripting. Un script exécuté dans vos pages peut lire le jeton du formulaire et envoyer une requête de même origine selon tous les critères. La fiche OWASP est explicite : XSS « peut déjouer toutes les techniques d'atténuation CSRF ». CWE et Django le disent aussi, et l'exemple le plus connu de CWE montre les deux ensemble. Le ver Samy utilisait XSS pour s'installer dans un profil MySpace et ajouter l'attaquant comme ami ; les amis qui visitaient ce profil voyaient le leur modifié de même. Comme aucune victime n'avait choisi ce changement, CWE compte CSRF comme cause fondamentale. Traitez chaque constat XSS aussi comme un constat CSRF et corrigez-le d'abord.

Le contrôle d'origine accepte trop de valeurs. Un test comme `origin.startswith("https://app.example.com")` accepte aussi `https://app.example.com.attacker.example`, le piège contre lequel OWASP avertit. La valeur `null` en est un autre. Selon le Fetch Standard, un formulaire envoyé depuis une page dont la politique de provenance est `no-referrer` arrive avec `Origin: null` ; OWASP ajoute les redirections entre origines et les cadres en bac à sable parmi ses autres causes. Comparez l'origine entière à une liste fixe de chaînes exactes et traitez un en-tête absent ou `null` comme inconnu, jamais comme votre propre origine.

On fait trop confiance au preflight. Un en-tête de requête personnalisé est une défense solide pour une API, car les navigateurs envoient un preflight avant toute requête interorigine qui en contient un. Cela ne tient que si l'API refuse tout le reste. OWASP avertit qu'une API JSON acceptant aussi `text/plain` est accessible par formulaire, puisque ce type figure parmi les trois types de contenu permis aux requêtes simples, et qu'une politique CORS autorisant les informations d'authentification depuis tout sous-domaine par une expression régulière permet à quiconque contrôle un sous-domaine d'envoyer l'en-tête personnalisé. Refusez les types de contenu inattendus et énumérez les origines exactes pouvant envoyer des requêtes avec informations d'authentification.

Le formulaire de connexion est oublié. Le CSRF de connexion fonctionne à l'envers : l'attaquant connecte la victime à son propre compte d'attaquant. Dans l'exemple de boutique d'OWASP, la victime saisit ensuite ses coordonnées de carte bancaire, que l'attaquant peut utiliser plus tard pour faire des achats. Django cite le CSRF de connexion parmi les attaques couvertes par sa protection. Protégez le formulaire avec un jeton lié à une pré-session, puis détruisez celle-ci et ouvrez une nouvelle session une fois l'utilisateur connecté, comme le conseille OWASP.

Les requêtes sont construites à partir d'une entrée de l'attaquant. Le CSRF côté client survient lorsque le JavaScript de l'application construit une requête à partir d'une entrée contrôlée par l'attaquant, comme l'URL, le nom de fenêtre ou un `postMessage`. Le script ajoute lui-même le jeton et la requête est same-site ; ni les jetons ni SameSite ne l'arrêtent donc. OWASP y voit le retour de l'adjoint confus. Gardez les cibles et les paramètres de requête hors de portée des entrées contrôlées par l'attaquant, ou choisissez-les dans une liste fixe du code.

Pour aller plus loin

- CWE-352 : Cross-Site Request Forgery (CSRF) — la faiblesse : structure composite, conséquences, mesures d'atténuation et CVE observées.
- Fiches OWASP : prévention de la falsification de requête intersite — jetons, Fetch Metadata, contrôles d'origine, limites de SameSite et CSRF de connexion.
- W3C : en-têtes de requête Fetch Metadata — l'en-tête `Sec-Fetch-Site` et les quatre valeurs qu'un navigateur peut envoyer.
- Projet IETF : Cookies, mécanisme de gestion d'état HTTP (draft-ietf-httpbis-rfc6265bis) — l'attribut SameSite et les raisons pour lesquelles Lax est une défense en profondeur, pas un remède ; encore un Internet-Draft.

- Documentation Django : protection contre la falsification de requête intersite — la protection intégrée du framework, ses hypothèses et la manière dont les sous-domaines la contournent.
- OWASP Top 10:2025 : A01 Broken Access Control — la place de CWE-352 dans l'édition 2025.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le parcours de déploiement, les risques opérationnels et les personnes d'astreinte. Nous pouvons définir un changement avec des vérifications, une procédure de reprise et une responsabilité claire.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Que signifie CSRF ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/csrf/