Client-Side Script Injection (CCS)

Client-Side Script Injection (CCS) désigne toute injection de code (souvent JavaScript, mais aussi HTML, CSS, SVG, etc.) exécutée dans le navigateur de la victime.

Il s'agit d'un sous-ensemble des attaques côté client, dont le but est d'exécuter du code malveillant sans intervention du serveur.

XSS (Cross-Site Scripting) est la forme la plus connue de CCS, mais **CCS inclut aussi** :

- HTML Injection
- CSS Injection
- SVG Injection
- DOM-based Manipulation

L'objectif de l'attaque CCS à plusieurs objectifs comme le vol de données sensibles comme des cookies de sessions Cookies, jetons JWT…

Redirection vers un site de phishing ou malveillant

Phishing via injection d'interface

Keylogging (espionnage des frappes clavier)

Bypass de filtrage, ce terme désigne une technique utilisée pour contourner les systèmes de filtrage mis en place pour bloquer ou limiter certains contenus, actions ou accès.

Requêtes automatiques à l'insu de l'utilisateur (Attaque CSRF)

Escalade de privilèges

Type	Description
Reflected XSS	Payload dans l'URL ou formulaire, affiché immédiatement
Stored XSS	Script enregistré en base, affiché à chaque visite
DOM XSS	Exploitation du DOM côté client
HTML Injection	Injection de balises HTML pour modifier l'interface
CSS Injection	Injection de styles malveillants
SVG Injection	Vecteur SVG utilisé pour charger du JS ou exécuter du code

Scénario d'attaque typique

1. L'attaquant repère un champ de saisie ou URL non sécurisé.
2. Il injecte un script dans ce champ (formulaire de commentaire, champ de recherche...).
3. Ce script est **réinjecté dans la page web sans échappement**.
4. Le navigateur de la victime **interprète le script** et l'exécute.
5. Le script peut voler des cookies ou envoyer des requêtes à un serveur distant de l'attaquant.

🛡 Mesures de protection côté développeur :

- **Échappement de caractères** (<, >, ", ', &)
- **Utilisation de frameworks modernes** (React, Angular, Vue) qui échappent automatiquement
- **Validation côté serveur + client**
- **Stockage des données avec sanitation** le **sanitizing** (ou assainissement) des données consiste à nettoyer une entrée utilisateur avant de la stocker ou de l'utiliser (dans une base de données, dans une page HTML, etc.).

Exemple de sanitation (nettoyage)

Avant **stockage ou affichage**, on remplace certains caractères :

Caractère	Remplacé par
<	<
>	>
"	"
'	'
/	/

<script>alert('Hacked')</script>

Si ce script est stocké en brute comme elle est en base de données, elle sera afficher plus tard dans une page web, si non **nettoyer**, le script sera exécuté dans le navigateur c'est ce qu'on appel une **attaque XSS stockée**.

Nettoyé ça ressemble à ça <script>alert('Hacked')</script>

Résultat : Il s'affiche comme du texte, pas comme un script**.**

- Ne jamais insérer d'entrée utilisateur dans **`innerHTML`**, **`document.write`**, etc.

🛡 Mesures de protection côté navigateur :

CSP (Content Security Policy) :
Empêche le chargement de scripts non autorisés : Content-Security-Policy: default-src 'self'; script-src 'self'

HttpOnly sur les cookies :
Empêche JS de lire les cookies :Set-Cookie: sessionid=xyz; HttpOnly; Secure

SameSite Cookies :
Empêche CSRF : Set-Cookie: session=abc; SameSite=Strict

Cas réels

1. **British Airways (2018)** : vol de données via script injecté dans un formulaire de paiement.
2. **Yahoo** : plusieurs failles XSS signalées entre 2013–2016 sur des services critiques (messagerie, recherche).
3. **WordPress** : plugins vulnérables régulièrement exploités pour stocker du code XSS.

🛡 Bonnes **pratiques de pentester / dev sécurisé**

Côté Pentest	Côté Dev Sécurisé
Identifier tous les points d'entrée	Échapper systématiquement
Tester HTML/JS/DOM Injection	Désactiver innerHTML ou eval()
Utiliser XSS Hunter	Activer CSP et HttpOnly
Utiliser payloads <script>, <img>	Logger tous les inputs suspects