Browser Exploitation Moderne : V8, Blink et les Sandbox

Auteur : Ayi NEDJIMI **Date :** 15 février 2026

À RETENIR

√ VÉRIFIÉMANUELLEMENT

Votre architecture de sécurité repose-t-elle sur une seule couche de défense ?

Introduction

Le navigateur web est le logiciel le plus attaqué au monde. Avec plus de 3 milliards d'utilisateurs de Chrome/Chromium, la surface d'attaque est immense. Les moteurs JavaScript (V8 dans Chrome, SpiderMonkey dans Firefox, JavaScriptCore dans Safari) traitent du code non fiable provenant d'Internet, le compilent en code machine natif via des compilateurs JIT (Just-In-Time), et l'exécutent dans un sandbox censé protéger le système d'exploitation. Une faille dans ce système peut permettre l'exécution de code arbitraire sur la machine de la victime simplement en visitant une page web.

⚡ Retour terrain

Dans les projets techniques complexes, j'ai appris à toujours commencer par auditer la documentation existante plutôt que l'infrastructure elle-même. Dans 80 % des cas, le delta entre la documentation et la réalité est la source première de risques. Une infrastructure bien documentée qui ne correspond pas à la réalité est plus dangereuse qu'une infrastructure sans documentation — parce qu'elle induit une fausse confiance.

En 2025-2026, les exploits navigateur restent un outil de choix pour les acteurs étatiques et les groupes de cyberespionnage. Les vulnérabilités zero-day dans Chrome se vendent entre 500 000 et 3 millions de dollars sur le marché gris (Zerodium, Crowdfense). Google a corrigé plus de 40 zero-days activement exploités dans Chrome depuis 2020, la majorité ciblant V8 et le sandbox.
Cet article analyse l'architecture de sécurité des navigateurs modernes (Chrome/Chromium), les techniques d'exploitation de V8 (type confusion, JIT compilation exploits), les méthodes de sandbox escape, et les mitigations de pointe (V8 Sandbox, MiraclePtr, CFI). Il s'adresse aux chercheurs en sécurité, aux développeurs de navigateurs et aux équipes de threat intelligence.

Élément	Description	Priorite
Prevention	Mesures proactives de reduction de la surface d'attaque	Haute
Detection	Surveillance et alerting en temps reel	Haute
Reponse	Procedures d'incident response et remediation	Critique
Recovery	Plan de reprise et continuite d'activite	Moyenne

Architecture du Browser : V8, Blink, Sandbox

Architecture multi-processus de Chrome

Chrome utilise une architecture multi-processus avec isolation par site (Site Isolation). Chaque site web s'exécute dans un processus de rendu séparé, sandboxé, qui n'a aucun accès direct au système de fichiers, au réseau ou aux autres onglets. Le processus browser (principal) est le seul à avoir des privilèges élevés et sert d'intermédiaire via l'IPC Mojo.

```
 # Architecture des processus Chrome
#
# [Browser Process] (privilégié, unique)
# |-- Network Service
# |-- GPU Process
# |-- Storage Service
# |-- [Renderer Process 1] (sandboxé, site A)
# | |-- V8 JavaScript Engine
# | |-- Blink Rendering Engine
# | |-- DOM, CSSOM, Layout
# |-- [Renderer Process 2] (sandboxé, site B)
# | |-- V8 JavaScript Engine
# | |-- Blink Rendering Engine
# |-- [Renderer Process N] (sandboxé, site N)
#
# Communication via Mojo IPC (interfaces typées)
# Le renderer ne peut PAS :
# - Accéder au filesystem
# - Ouvrir des sockets réseau directement
# - Communiquer avec d'autres renderers
# - Accéder aux périphériques (caméra, micro)
# Sans passer par le browser process via Mojo

# Full chain exploit = 2 étapes minimum :
# 1. RCE dans le renderer (V8/Blink exploit)
# 2. Sandbox escape (Mojo IPC exploit ou kernel exploit)
```

V8 : le moteur JavaScript

V8 est le moteur JavaScript et WebAssembly de Chrome. Il compile le JavaScript en code machine natif via plusieurs niveaux de compilation : Ignition (interpréteur

bytecode), Sparkplug (compilateur baseline), Maglev (compilateur mid-tier), et TurboFan (compilateur optimisant JIT). Chaque niveau introduit des opportunités d'exploitation :

```
// Pipeline de compilation V8
//
// JavaScript Source Code
// |
// v
// [Parser] -> AST (Abstract Syntax Tree)
// |
// v
// [Ignition] -> Bytecode (interprété)
// | Collect type feedback (profiling)
// v
// [Sparkplug] -> Code machine non-optimisé (rapide)
// | Accumulate hot function data
// v
// [Maglev] -> Code machine mid-tier optimisé
// | More type feedback
// v
// [TurboFan] -> Code machine hautement optimisé
// | Speculative optimizations basées sur le type feedback
// | SI les types changent -> Deoptimization (bailout)
// v
// [Code machine natif x86-64/ARM64]

// Le problème de sécurité :
// TurboFan fait des SUPPOSITIONS sur les types
// basées sur le profiling (type feedback)
// Si ces suppositions sont incorrectes mais que le
// code ne vérifie pas (bug dans TurboFan), on obtient
// une type confusion -> corruption mémoire -> RCE
```

AVIS DrXPERT

Notre avis d'expert

La défense en profondeur n'est pas un concept abstrait — c'est une architecture concrète avec des couches mesurables et testables. Chaque couche doit être conçue pour fonctionner indépendamment des autres, car l'hypothèse de défaillance d'une couche est la seule hypothèse réaliste.

Type Confusion dans V8

Principe de la type confusion

Les objets JavaScript dans V8 sont représentés par des structures en mémoire avec un pointeur vers une `Map` (aussi appelée "hidden class" ou "shape") qui décrit le layout de l'objet (quels champs, à quels offsets, quels types). Une type confusion se produit quand V8 traite un objet avec la mauvaise Map, accédant à des champs avec les mauvais types et offsets.

```
 // Exemple conceptuel de type confusion V8
// (simplifié pour la compréhension)

// V8 représente les objets en mémoire comme :
// [Map pointer][Properties pointer][Éléments pointer][Field1][Field2]...

// Objet A (type: {x: SMI, y: SMI})
// [MapA][...][...][0x42 (int)][0x43 (int)]

// Objet B (type: {x: HeapObject, y: SMI})
// [MapB][...][...][ptr vers objet][0x43 (int)]

// Si V8 confond MapA et MapB :
// Il lit le champ x de A (0x42, un entier)
// comme un pointeur vers un objet HeapObject
// -> Lecture/écriture à l'adresse 0x42 en mémoire !

// Exploitation typique :
function trigger_confusion() {
 // Phase 1 : entraîner TurboFan avec des types cohérents
 for (let i = 0; i < 100000; i++) {
 let obj = {x: 42, y: 43}; // Toujours SMI
 foo(obj);
 }
 // TurboFan compile foo() en supposant x est toujours SMI

 // Phase 2 : passer un objet avec un type différent
 let evil = {x: {}, y: 43}; // x est maintenant un HeapObject
 foo(evil);
 // TurboFan traite evil.x (un pointeur) comme un SMI
 // -> Type confusion -> primitive d'addrof/fakeobj
}

// Primitives obtenues par type confusion :
// addrof(obj) : obtenir l'adresse mémoire d'un objet JS
// fakeobj(addr) : traiter une adresse arbitraire comme un objet JS
// -> Combinées = lecture/écriture arbitraire en mémoire
```

√ VÉRIFIÉMANUELLEMENT

Combien de vos contrôles de sécurité ont été testés en conditions réelles cette année ?

JIT Compilation Exploits

Bugs dans les optimisations TurboFan

TurboFan, le compilateur JIT optimisant de V8, effectue des optimisations agressives basées sur les types observés pendant le profiling. Les bugs dans ces optimisations sont la source la plus fréquente de vulnérabilités V8. Les catégories principales sont :

Bounds check elimination (BCE) : TurboFan supprime des vérifications de bornes de tableaux quand il "prouve" qu'elles sont inutiles. Si la preuve est incorrecte, un accès OOB (out-of-bounds) est possible.

Redundancy elimination : TurboFan élimine des opérations qu'il considère redondantes. Si deux opérations ne sont pas réellement équivalentes, cela peut créer un état incohérent.

Escape analysis bugs : TurboFan peut "scalar replace" un objet alloué sur le heap si il détermine qu'il n'échappe pas de la fonction. Un bug dans cette analyse peut corrompre l'état.

Side effect misattribution : TurboFan peut réordonner des opérations s'il considère qu'elles n'ont pas d'effets de bord. Un mauvais jugement peut créer des race conditions.

```
 // Exemple : bug de bounds check elimination
// CVE-2021-21224 (type confusion via integer overflow)

function vuln(arr, idx) {
 // TurboFan voit que idx est toujours < arr.length
 // pendant le profiling, et élimine le bounds check
 return arr[idx];
}

// Phase 1 : warm up avec des index valides
let arr = new Array(100).fill(1.1);
for (let i = 0; i < 100000; i++) {
 vuln(arr, i % 100);
}
// TurboFan compile vuln() SANS bounds check

// Phase 2 : passer un index qui cause un integer overflow
// dans le calcul interne de TurboFan
let oob_idx = -1; // ou un très grand nombre
// L'absence de bounds check permet l'accès OOB

// OOB read -> leak d'adresses mémoire (ASLR bypass)
// OOB write -> corruption d'objets adjacents
// -> Chaîne vers addrof/fakeobj -> RCE dans le renderer
```

RETOUR DrXPÉRIENCE

Cas concret

L'exploitation de Log4Shell (CVE-2021-44228) en décembre 2021 a démontré les risques systémiques liés aux dépendances open-source. Cette vulnérabilité dans la bibliothèque de logging Log4j affectait des millions d'applications Java et a nécessité une mobilisation mondiale de l'industrie pour identifier et corriger tous les systèmes vulnérables.

Sandbox Escape Techniques

Exploitation de Mojo IPC

Mojo est le framework IPC de Chrome qui permet aux processus de rendu sandboxés de communiquer avec le processus browser privilégié. Les interfaces Mojo sont typées (via Mojom IDL), mais des bugs dans leur implémentation peuvent permettre au renderer d'envoyer des messages malformés qui exploitent des vulnérabilités dans le processus browser.

```
 // Exploitation Mojo IPC (conceptuel)
// Depuis le renderer compromis (post-V8 exploit) :

// 1. Le renderer a accès aux interfaces Mojo
// définies dans les fichiers .mojom
// 2. Certaines interfaces permettent des opérations
// qui ne devraient pas être accessibles depuis
// un renderer compromis

// Exemple : interface de navigation
// Le renderer peut demander au browser de naviguer
// vers une URL spéciale (file://, chrome://) qui
// déclenche un bug dans le handling côté browser

// CVE-2023-4863 : WebP heap buffer overflow
// Exploitable depuis le renderer via le décodage d'images
// L'image WebP est traitée par le GPU process ou
// un service dédié, qui a plus de privilèges

// CVE-2024-0519 : V8 OOB access
// + CVE-2024-XXXX : Mojo interface use-after-free
// = Full chain : visite d'une page web -> RCE sur l'OS

// Stratégie d'exploitation typique :
// 1. V8 type confusion -> addrof/fakeobj
// 2. Construire primitives R/W arbitraire
// 3. Identifier les objets Mojo en mémoire
// 4. Corrompre un pointeur d'interface Mojo
// 5. Rediriger un appel Mojo vers du code contrôlé
// 6. Le browser process exécute le code avec ses privilèges
// 7. Sandbox escape complete
```

Kernel exploits depuis le sandbox

Une alternative au bypass du sandbox Chrome est d'exploiter directement une vulnérabilité kernel depuis le processus de rendu sandboxé. Bien que le sandbox limite les appels système disponibles (via seccomp-bpf sur Linux, restricted tokens sur Windows), certaines vulnérabilités kernel sont accessibles depuis le sandbox :

GPU driver bugs : Le processus de rendu communique avec le GPU via des interfaces qui exposent des bugs dans les drivers GPU (NVIDIA, AMD, Intel)

Font rendering bugs : Le parsing de polices (OpenType, TrueType) peut déclencher des vulnérabilités dans le kernel (Windows) ou les bibliothèques système

Syscall bugs : Certains syscalls autorisés par le profil seccomp du sandbox ont des vulnérabilités (io_uring, eBPF avant leur blocage)

Full Chain Exploitation

Anatomie d'un exploit Chrome complet

```
// Full chain Chrome exploit - Structure typique
// (ne contient PAS de code d'exploit fonctionnel)

// Phase 1 : Trigger V8 vulnerability
// Déclencher un bug de type confusion ou OOB dans V8
// via du JavaScript spécialement conçu

// Phase 2 : Build exploitation primitives
// addrof(obj) : leak l'adresse d'un objet JS
// fakeobj(addr) : traiter une adresse comme un objet
// -> Construire un faux ArrayBuffer avec backing store
// pointant vers une adresse arbitraire
// -> Obtenir lecture/écriture arbitraire dans le renderer

// Phase 3 : Bypass ASLR
// Utiliser addrof pour leak des pointeurs
// Calculer la base de V8, libc, Chrome binary
// Identifier les gadgets ROP ou les objets Mojo

// Phase 4 : Code exécution dans le renderer
// Option A : Écrire du shellcode dans une page RWX
// (si JIT pages sont RWX - de moins en moins courant)
// Option B : ROP chain via les gadgets identifiés
// Option C : Corrompre un objet Wasm pour obtenir RWX

// Phase 5 : Sandbox escape
// Exploiter un bug Mojo IPC pour corrompre le browser process
// Ou exploiter un bug kernel accessible depuis le sandbox
// Ou exploiter un bug dans un service intermédiaire (GPU, Network)

// Phase 6 : Payload final
// Exécution de code avec les privilèges du processus browser
// ou les privilèges kernel (selon la technique de sandbox escape)

// Valeur marchande (2026) :
// V8 RCE seul (dans le sandbox) : $200k - $500k
// Full chain (RCE + sandbox escape) : $500k - $3M
// Full chain + persistence : $1M - $5M+
```

Mitigations : MiraclePtr, CFI, V8 Sandbox

V8 Sandbox (Memory Cage)

Le V8 Sandbox (aussi appelé "Memory Cage") est la mitigation la plus significative introduite par Google en 2024-2025. Il confine tous les objets V8 dans une région mémoire réservée de 1 To (virtuel). Les pointeurs au sein de cette région sont convertis en offsets 40-bit relatifs au début de la cage. Cela signifie qu'une corruption de pointeur dans V8 ne peut accéder qu'à la mémoire à l'intérieur de la cage, pas à la mémoire du processus de rendu entier.

```
// V8 Sandbox : transformation des pointeurs
// AVANT le sandbox :
// [Map ptr 64-bit][Éléments ptr 64-bit][Field1]...
// Un attaquant corrompant un pointeur peut lire/écrire
// n'importe où dans l'espace d'adressage du processus

// APRÈS le sandbox :
// [Map offset 32-bit][Éléments offset 32-bit][Field1]...
// Les offsets sont relatifs au début de la cage V8
// Un attaquant ne peut accéder qu'à la mémoire dans la cage
// Les objets Mojo, le code C++, la pile, etc.
// sont EN DEHORS de la cage

// Impact sur les exploits :
// - addrof() ne donne qu'un offset dans la cage, pas une vraie adresse
// - fakeobj() ne peut créer des faux objets que dans la cage
// - Les pointeurs vers du code natif sont "sandboxed pointers"
// (table d'indirection en dehors de la cage)
// - Les ArrayBuffer backing stores pointent EN DEHORS de la cage
// mais via un mécanisme vérifié (external pointer table)

// Le V8 Sandbox rend BEAUCOUP plus difficile de :
// 1. Obtenir une vraie adresse mémoire (ASLR toujours efficace)
// 2. Corrompre des objets C++ (Blink, Mojo) depuis V8
// 3. Exécuter du code arbitraire depuis une corruption V8

// Activation : chrome://flags/#enable-v8-sandbox (par défaut depuis 2025)
```

MiraclePtr (BackupRefPtr)

MiraclePtr est une protection contre les use-after-free (UAF) dans le code C++ de Chrome (Blink, Mojo, etc.). Elle fonctionne en ajoutant un compteur de références à chaque allocation PartitionAlloc. Quand un objet est libéré alors que des pointeurs "raw" le référencent encore, la mémoire n'est pas réellement libérée -- elle est mise en "quarantaine" jusqu'à ce que tous les MiraclePtr soient détruits. Cela rend les UAF inexploitables en empêchant la réallocation de la mémoire.

Control Flow Integrity (CFI)

CFI vérifie que les appels de fonctions indirects (via pointeurs de fonctions, vtables) ciblent des destinations légitimes. Chrome utilise Clang CFI qui vérifie à runtime que la cible d'un appel indirect correspond au prototype de fonction attendu. Cela complique significativement les exploits de type ROP/JOP et la corruption de vtables.

> État des mitigations Chrome en 2026
>
> **V8 Sandbox** : Activé par défaut. Confine les corruptions V8 dans une cage mémoire dédiée.
>
> **MiraclePtr** : Couvre ~50% des allocations C++ de Chrome. Élimine les UAF exploitables.
>
> **CFI** : Activé sur toutes les plateformes. Bloque les détournements de flux de contrôle.
>
> **Site Isolation** : Chaque site dans un processus séparé. Complique le cross-site exploitation.
>
> **Seccomp-BPF** : Filtre strict des syscalls dans le renderer. io_uring bloqué.
>
> **PAC/BTI (ARM64)** : Pointer Authentication et Branch Target Identification sur Apple Silicon et ARM serveur.

> **V8 Maglev** : Compilateur mid-tier réduisant le temps passé dans TurboFan (moins de bugs d'optimisation).

Ressources et références

- Techniques d'Évasion EDR/XDR
- Désérialisation et Gadgets
- Exfiltration Furtive
- Living-off-the-Land

Ayi NEDJIMI

Expert en Cybersécurité & Intelligence Artificielle
Consultant senior avec plus de 15 ans d'expérience en sécurité offensive, audit d'infrastructure et développement de solutions IA. Certifié OSCP, CISSP, ISO 27001 Lead Auditor et ISO 42001 Lead Implementer. Intervient sur des missions de pentest Active Directory, sécurité Cloud et conformité réglementaire pour des grands comptes et ETI.

LinkedIn Profil complet Tous ses articles

Partagez cet Article

Partagez-le avec votre réseau professionnel !

Partager sur X Partager sur LinkedIn

Références et ressources externes

OWASP Testing Guide — Guide de référence pour les tests de sécurité web

MITRE ATT&CK T1189 — Drive-by Compromise

PortSwigger Academy — Ressources d'apprentissage en sécurité web

CWE — Common Weakness Enumeration — catalogue de faiblesses logicielles

NVD — National Vulnerability Database — base de vulnérabilités du NIST

ARTICLE SUIVANT RECOMMANDÉ

Bypass FIDO2 et Passkeys : Attaques sur l'Authentification →

Contournement de FIDO2/WebAuthn : AitM sur l'enrollment, device binding bypass, token theft. Thèmes : passkeys, authenti

📖 DÉFINITION

Surface d'attaque : Ensemble des points d'entrée exploitables par un attaquant pour compromettre un système, incluant les services exposés, les interfaces utilisateur et les API.

⚠️ ATTENTION

Les techniques et outils de sécurité présentés dans cet article sont destinés aux professionnels de la cybersécurité dans un cadre autorisé. Toute utilisation malveillante est interdite et pénalement répréhensible.

💡 BON ÀSAVOIR

Mettez en place un environnement de lab isolé pour pratiquer les techniques décrites. Les plateformes comme HackTheBox, TryHackMe ou un lab Active Directory local sont idéales pour l'apprentissage sécurisé.

Besoin d'un expert cybersécurité ?

Audit, pentest, formation, IA — plus de 25 ans d'expérience, 100+ missions réalisées.

Demander un devis

ayi@ayinedjimi-consultants.fr

Vous souhaitez approfondir ce sujet ou évaluer votre exposition ?

Contacter nos experts ou contactez-nous directement.

Questions frequentes

Comment ce sujet impacte-t-il la sécurité des organisations ? +

Quelles sont les bonnes pratiques recommandees par les experts ? +

Pourquoi est-il important de se former sur ce sujet en 2026 ? +

Pour approfondir ce sujet, consultez notre outil open-source log-analyzer qui facilite l'analyse automatisée des journaux de sécurité.

Conclusion

L'exploitation des navigateurs modernes est devenue un art extrêmement technique, réservé aux chercheurs les plus avancés et aux acteurs étatiques. Les mitigations successives -- V8 Sandbox, MiraclePtr, CFI, Site Isolation -- ont considérablement élevé la barre d'exploitation. Une full chain Chrome en 2026 nécessite potentiellement 3 à 5 vulnérabilités enchaînées et des mois de développement.

Cependant, l'exploitation navigateur reste un vecteur critique pour plusieurs raisons :

- Les navigateurs traitent du contenu non fiable par design -- chaque page web est du code potentiellement hostile.
- La complexité croissante (WebAssembly, WebGPU, WebTransport) élargit continuellement la surface d'attaque.
- Les zero-days navigateur sont activement utilisés par les acteurs étatiques (NSO Group, Candiru, Intellexa) pour le surveillance ciblée.

> La valeur marchande des exploits navigateur continue d'augmenter, incitant la recherche offensive.

Pour les défenseurs, la priorité est de maintenir les navigateurs à jour (les correctifs Chrome sont disponibles en 24-48h après la divulgation), d'activer Site Isolation et toutes les mitigations par défaut, et de surveiller les campagnes d'exploitation zero-day via les feeds de threat intelligence (Google TAG, Microsoft MSTIC, CitizenLab).

Sources et références : MITRE ATT&CK · CERT-FR

© Ayi NEDJIMI Consultants — Tous droits réservés