Évolution des protections du moteur Javascript V8

François Jolivet
francois.jolivet@ssi.gouv.fr

ANSSI

Résumé. Cet article propose une exploration approfondie des techniques utilisées par les attaquants pour exploiter une vulnérabilité ciblant le moteur Javascript V8. L'analyse se concentre sur la démystification des différentes étapes de l'exploitation, depuis le déclenchement de la vulnérabilité jusqu'à l'aboutissement de l'exécution de code arbitraire. Cette investigation se déroule dans le contexte des mécanismes de sécurité en constante évolution de V8, mettant en avant l'implémentation du projet V8 Sandbox.

En examinant en détail les tenants et aboutissants de l'exploitation, l'article met en évidence les défis posés par les mesures de sécurité telles que la compression de pointeurs et l'isolation des objets Javascript, soulignant l'importance cruciale de protéger les pointeurs sensibles pour atténuer les menaces potentielles, tout en mettant en avant la nécessité croissante d'intégrer des protections matérielles.

L'analyse va plus loin en soulignant la capacité d'adaptation des attaquants face aux innovations des développeurs, mettant ainsi en lumière l'évolution continue des attaques et des mesures de protection dans l'écosystème des navigateurs.

1 Introduction

Le moteur JavaScript **V8** [1] [6], développé par Google, joue un rôle essentiel dans l'écosystème des moteurs Javascript avec d'autres moteurs tels que SpiderMonkey [2] (Mozilla) ou encore JavascriptCore [3] (Apple). En tant qu'outil open source, il est largement utilisé dans divers projets tels que Node.js et Electron. De plus, le navigateur Chromium, qui intègre le moteur V8, est également open source, offrant ainsi une base pour le développement d'autres navigateurs tels que Chrome (la version source fermée de Chromium), Edge, Opera, Brave et d'autres initiatives. Au fil des années, V8 a été la cible fréquente d'attaques de type Zero-Day, suscitant des préoccupations légitimes quant à sa sécurité. Toutefois, en 2023, une

[1] https://v8.dev
[2] https://spidermonkey.dev
[3] https://developer.apple.com/documentation/javascriptcore

diminution significative du nombre d'attaques a été observée, ce phénomène pouvant être attribué à l'intégration de nouvelles protections au sein de V8. Par le passé, il a déjà été observé que l'ajout de contre-mesures efficaces tend à augmenter significativement les coûts de développement d'une attaque, incitant ainsi les attaquants à changer de cible. Cette tendance a été particulièrement remarquée dans le cas des langages ActionScript [27] (utilisé principalement pour le développement de sites et d'application ciblant la plate-forme Flash d'Adobe) et Java [8].

Cet article se penche sur l'évolution récente de la sécurité de V8 en commençant par une brève introduction présentant le navigateur Chromium et le moteur V8 (section 2), puis une présentation succincte des éléments du langage Javascript nécessaire à la bonne compréhension de cet article est proposée section 3. Ensuite, un historique et une analyse des techniques d'exploitation couramment utilisées par les attaquants sont détaillés section 4 et 5.

Les mesures de protection nouvellement intégrées à V8 sont alors explorées section 6, examinant en détail leur fonctionnement et leur capacité à neutraliser les techniques d'exploitation prévalentes. Ces mesures de protection forment ensemble un mécanisme de sandbox appelé la sandbox V8. Pour finir, les limitations de ces mesures de sécurité et les tactiques de contournement mises en œuvre par les attaquants en réaction aux nouvelles défenses sont abordées. Une compréhension approfondie de ces éléments est cruciale pour évaluer la robustesse de V8 face aux menaces persistantes dans le paysage de la sécurité informatique où les navigateurs jouent un rôle de plus en plus prépondérant.

2 Le Navigateur Chromium et V8

L'exécution du code Javascript joue un rôle essentiel dans la création de pages web dynamiques, contribuant ainsi à améliorer l'expérience de navigation pour des millions d'utilisateurs à travers le monde. En conséquence, le moteur Javascript au sein des navigateurs est un composant crucial et critique pour assurer une interaction fiable et sécurisée avec ce type de contenu. En se référant aux statistiques globales et mondiales du marché des navigateurs sur tous les types de plateforme (ordinateurs, tablettes et mobiles) [52], il apparaît que Chromium est le navigateur le plus utilisé (65.3% des parts de marché en février 2024), loin devant Safari (18.3%), Edge (5.07%), Firefox (3.04%) ou encore Opera (2.47%). Par extension, le moteur Javascript V8 utilisé au sein des navigateurs Chromium, Edge ou encore Opera est omniprésent dans l'écosystème web.

Au niveau de l'intégration, Chromium utilise, tout comme d'autres navigateurs tels que Firefox [15], une architecture basée sur la séparation des processus, où V8 est exécuté dans un processus distinct appelé le moteur de rendu, ou *renderer* [37]. L'objectif principal de cette approche est d'isoler les différents processus les uns des autres, ainsi que du système d'exploitation sous-jacent. Cette isolation est cruciale pour garantir la stabilité du navigateur et minimiser les risques de sécurité.

Le moteur de rendu sous Chromium, appelé **Blink** [4] [25], est responsable de l'affichage des pages Web dans les onglets du navigateur. V8, en tant que librairie utilisée par Blink, prend en charge l'exécution du code Javascript et WebAssembly [5] (Wasm). Pour renforcer cette isolation, Chromium tire parti des mécanismes de bac à sable [38], également connus sous le nom de *sandbox*. Ces mécanismes permettent de délimiter les actions d'un processus, restreignant son accès aux ressources et protégeant ainsi l'ensemble du système d'exploitation contre les attaques potentielles.

Une analyse plus poussée de l'architecture des processus au sein de Chromium dépasse le cadre de cet article. À titre indicatif, le lecteur intéressé pourra se référer à la documentation officielle.[6]

3 Le langage Javascript et sa gestion au sein de V8

Au cœur du paysage des attaques contre les navigateurs se trouve la manipulation des objets en Javascript. Comprendre les spécificités de ce langage, la façon dont les données associées sont représentées en mémoire où l'organisation du tas au sein de V8 sont autant de concepts essentiels pour appréhender au mieux les techniques d'exploitation.

Le langage Javascript est un langage de programmation, basé sur la norme *EcmaScript* [9], largement utilisé dans le développement web. Il s'appuie sur le typage dynamique qui confère au développeur une grande flexibilité en permettant aux variables de changer de type au cours de l'exécution du programme.

L'exécution de Javascript se fait à travers une machine virtuelle dédiée, notamment Ignition [7] dans le moteur Javascript V8. Le code source Javascript est préalablement compilé en *bytecode*, un ensemble d'instructions intermédiaires, qui est ensuite interprété et exécuté par la machine

[4] `https://www.chromium.org/blink/`
[5] `https://webassembly.org/`
[6] `https://www.chromium.org/developers/design-documents/multi-process-architecture/`
[7] `https://v8.dev/docs/ignition`

virtuelle. Cette approche permet d'obtenir une portabilité du langage sur différentes plates-formes.

Par ailleurs, Javascript tire pleinement parti des API fournies par le navigateur, offrant aux développeurs un accès direct à diverses fonctionnalités et services. L'API *Document Object Model* (DOM) permet la manipulation dynamique des éléments d'une page web, tandis que des API plus récentes, telles que WebGPU,[8] fournissent un accès performant aux fonctionnalités graphiques sur le GPU, élargissant ainsi les possibilités de développement web.

3.1 Fonctions *Built-ins*

Une fonction *Built-in* est une fonction Javascript pré-implémentée dans le moteur V8 pour offrir des fonctionnalités de base et des opérations fréquemment utilisées. Ces fonctions sont généralement plus rapides que celles implémentées en Javascript pur car elles sont directement écrites en langage intermédiaire Torque [9] (ou en CodeStubAssembler qui est une interface pour écrire du code assembleur dans le contexte du moteur V8) puis compilées en code machine optimisé. Les attaquants peuvent cibler ces fonctions pour exploiter des vulnérabilités, souvent en cherchant à manipuler les données ou le comportement de ces fonctions pour atteindre des objectifs malveillants. Un exemple illustrant une telle vulnérabilité est la `CVE-2021-21225` [4, 16], qui affecte la fonction Builtin `array.concat` (utilisée pour concaténer deux ou plusieurs tableaux).

3.2 Types de données

En Javascript, la représentation des données varie en fonction de leur type, avec des catégories distinctes telles que :

- **Entier/SMI :** Dans le contexte de V8, un entier est représenté sous forme de SMI (*SMall Integer*) [10]. Sur une architecture 64 bits, les SMI sont typiquement des mots de 64 bits stockés directement en mémoire permettant ainsi d'économiser de la ressource. En effet, il n'est pas nécessaire de réaliser une allocation dynamique sous forme d'objet Javascript pour représenter l'entier ;
- **Float :** utilisé pour les nombres décimaux à virgule flottante. Un Float sous V8 est représenté via la norme IEEE754 utilisant 8 octets de mémoire. Cette spécificité est abusée par les attaquants afin de fuiter de la mémoire ;

[8] `https://gpuweb.github.io/gpuweb/`
[9] `https://v8.dev/docs/torque`

— **BigInt :** introduit pour représenter des entiers de taille arbitraire, échappant aux limites des entiers Javascript classiques. L'attaquant utilise la représentation en BigInt pour manipuler les pointeurs une fois fuités.

3.3 Modèle d'Objet Javascript (JSObject)

Lorsqu'un développeur instancie un objet en Javascript dans son code, il est ensuite représenté en mémoire, sous forme d'un objet C++ hérité de l'objet parent *JSObject* [5, 56]. Cet objet se compose de plusieurs composants clés comme illustrés sur le schéma en Figure 1.

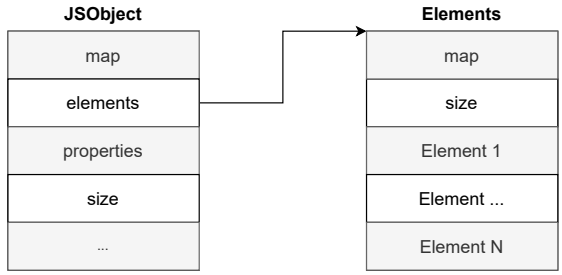

Fig. 1. Structure d'un objet JSObject

— **map :** Le pointeur *map* (carte) est une référence vers une structure de données interne qui définit la forme de l'objet. Il s'agit essentiellement d'une carte décrivant la disposition des propriétés et éléments de l'objet. Lorsqu'un nouvel objet Javascript est créé, V8 lui attribue une carte spécifique qui détermine comment l'objet est structuré en mémoire. La structure d'un objet contient des informations telles que le nombre de propriétés, la disposition des éléments, les emplacements en mémoire où peuvent être retrouvées les données, et d'autres méta-données cruciales. Cette carte est ensuite partagée par tous les objets ayant la même structure, ce qui permet d'économiser de la mémoire ;
— **properties :** pointeur faisant référence à une structure de données interne stockant les informations sur les propriétés définies pour cet objet. Elle contient des détails essentiels tels que le nom de la propriété, sa valeur, son attribut (par exemple, si elle est en lecture seule), et d'autres méta-données associées. On parle ici de propriétés nommées de la forme `"a":1`. Le caractère *a* représente le nom de

la propriété et *1* sa valeur. Ainsi, lorsque des modifications sont apportées sur une propriété, les structures *properties* et *map* sont utilisées. En effet, pour retrouver l'emplacement dans la structure *properties*, la structure *map* est utilisée (plus particulièrement le *descriptor*).

— **element :** une structure stockant les valeurs des éléments de l'objet, notamment les valeurs attribuées aux indices d'un tableau.

— **size :** indique le nombre d'éléments présents dans l'objet.

L'adresse d'un objet est représentée en mémoire via des *tagged pointers* (pointeurs taggés) utilisés conjointement avec la compression de pointeur [11].

— **Tagged Pointer :** Le mécanisme de *Tagged Pointer* consiste à incorporer des informations supplémentaires dans les bits de poid faible d'un pointeur. Plutôt que d'utiliser tous les bits pour représenter l'adresse mémoire, le bit de poid faible est réservé pour stocker un drapeau permettant de distinguer un pointeur d'un entier SMI ;

— **Compression de pointeur :** la *Compression de Pointeur*[10] vise à réduire la taille des pointeurs en utilisant des offsets de 32 bits par rapport à une adresse de base à la place des adresses mémoires complètes de 64 bits. Cette compression permet de réduire l'empreinte mémoire des objets Javascript tout en limitant l'accès aux zones spécifiques de la mémoire. En effet, le principe fondamental de la sandbox V8 repose sur le mécanisme de compression de pointeur. La section 6 de cet article fournira une analyse détaillée de la sandbox V8.

3.4 ArrayBuffer

Un `ArrayBuffer` en Javascript est une structure de données qui représente un tampon mémoire brute, organisée sous la forme d'un tableau d'octets. Le `BackingStore`, ou zone de stockage, de l'`ArrayBuffer` est l'emplacement réel en mémoire où les données sont stockées. Cependant, l'`ArrayBuffer` lui-même ne fournit pas d'interface directe pour manipuler ou accéder aux données. C'est là que les `TypedArray` entrent en jeu. Les `TypedArray` sont des vues spécialisées sur l'`ArrayBuffer` qui permettent l'accès aux données en interprétant les octets de la mémoire selon un format particulier, tel que les entiers signés ou non signés, les nombres à virgule flottante, etc. Les `TypedArray` offrent un moyen efficace d'effectuer

[10] `https://v8.dev/blog/pointer-compression`

des opérations sur des données binaires de manière structurée. La figure 2 permet d'illustrer l'agencement mémoire des structures `BigUInt64Array`, `ArrayBuffer` et le buffer `BackingStore` associé dans le cas de l'utilisation d'un buffer avec un `TypedArray` utilisant le format `BigUint64`.

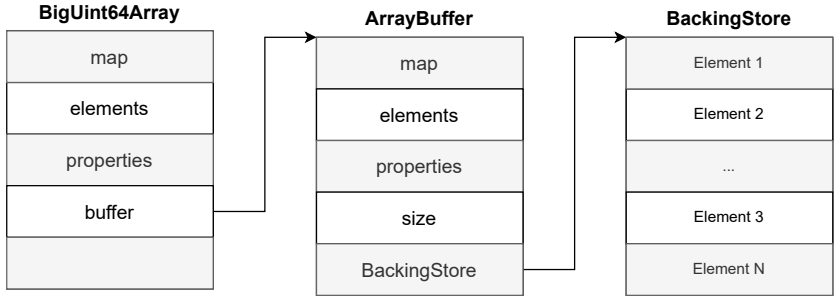

Fig. 2. Agencement mémoire d'un `ArrayBuffer`

En plus des `TypedArray`, l'interface `DataView` offre une flexibilité accrue pour interagir avec les données de l'`ArrayBuffer`. Les `DataView` permettent de spécifier manuellement le type de données et l'ordre des octets lors de l'accès aux données, offrant ainsi un contrôle précis sur la manière dont les informations sont interprétées. Cette fonctionnalité est particulièrement utile lorsque les données sont échangées entre des systèmes avec des encodages différents.

3.5 Le tas V8

La mémoire de V8 est organisée en plusieurs espaces distincts, chacun ayant son propre rôle spécifique dans le stockage des données et des objets Javascript. Les principaux espaces du tas V8 comprennent :

- **Newspace :** cet espace est utilisé pour l'allocation rapide et la gestion des objets de courte durée de vie ;
- **Oldspace :** conçu pour les objets qui ont survécu à plusieurs cycles de *Garbage Collector* dans *newspace*, c'est l'espace principal pour les objets de plus longue durée de vie ;
- **Large-oldspace :** similaire à *oldspace*, mais destiné aux objets particulièrement volumineux ;
- **Codespace :** réservé pour le stockage du code machine généré par le compilateur, cet espace peut se voir attribuer des droits d'exécution au moment de l'exécution du code JIT.

Cette organisation hiérarchique de la mémoire permet à V8 de gérer efficacement la création, la vie et la destruction des objets Javascript tout en optimisant l'utilisation des ressources système.

3.6 Composants clés de la chaîne d'exécution

L'exécution du code Javascript sous V8 repose sur plusieurs composants, chacun jouant un rôle spécifique dans le processus :

1. **Ignition :** Cette machine virtuelle est chargée de l'interprétation du *bytecode* Javascript [39]. Son rôle initial consiste à parser le code Javascript et à produire le *bytecode* correspondant. Ignition est responsable de l'exécution de code Javascript à un niveau intermédiaire, avant toute optimisation significative ;
2. **Sparkplug :** le compilateur Sparkplug [11] est un composant spécialisé axé sur la rapidité de compilation sans pour autant mettre en œuvre des optimisations complexes [53]. Positionné entre Ignition et TurboFan, son principal objectif est de convertir rapidement le *bytecode* généré par Ignition en langage machine natif. Il n'introduit pas de représentation intermédiaire, mais utilise les informations déjà produites lors de la transformation du JavaScript en *bytecode* ;
3. **Maglev :** Maglev [12] est un optimisateur situé entre Sparkplug et TurboFan dont la mission est de réaliser de l'optimisation très rapide sans utiliser d'analyse dynamique [55]. Pour cela, il se repose sur le mécanisme de *feedback* d'Ignition. Enfin, Maglev crée également une représentation intermédiaire sous forme de noeuds appelé *Maglev IR*.
4. **TurboFan :** TurboFan [13] prend en charge la compilation du *bytecode* Javascript en langage machine [54]. Il joue un rôle crucial en optimisant le code pour améliorer les performances d'exécution. TurboFan compile une fonction Javascript si elle est fréquemment appelée, ce que Ignition détermine en la déclarant comme *hot* ;
   (a) **Optimisation JIT (Just-In-Time) :** TurboFan effectue une compilation à la volée des fonctions déclarées comme *hot*. Plusieurs phases d'optimisation sont ensuite appliquées pour générer un code machine efficace.

[11] `https://v8.dev/blog/sparkplug`
[12] `https://v8.dev/blog/maglev`
[13] `https://v8.dev/docs/turbofan`

(b) **Compilation du ByteCode Wasm :** En plus de traiter le code Javascript, TurboFan est également responsable de la compilation du *bytecode* WebAssembly (Wasm) [32], offrant ainsi une polyvalence dans la gestion de l'exécution du code ainsi que du langage utilisé par le développeur.

3.7 Flux d'exécution du code Javascript

Lorsqu'une page Web intègre du code Javascript, le processus d'exécution commence par Ignition. Cette machine virtuelle analyse le code Javascript, produit le *bytecode* associé, puis interprète ce *bytecode* au sein de son environnement. Il est important de noter que le Javascript est un langage typé dynamiquement, ce qui signifie que les types de variables ne sont pas spécifiés lors de la déclaration. Le *bytecode* est ensuite compilé par Sparkplug vers du langage natif.

— **Types dynamiques :** Ignition effectue la détermination des types lors de l'exécution du *bytecode*. Cette caractéristique du Javascript permet une flexibilité accrue lors du développement, mais impose également une charge dynamique sur le moteur d'exécution ;

— **Observations et Feedback Vector :** les observations effectuées par Ignition sont stockées dans ce qui est appelé un *feedback vector* [28]. Ces informations sont cruciales pour optimiser le code plus tard dans le processus.

Si Ignition identifie une fonction comme *hot*, Maglev puis TurboFan entre en jeu pour effectuer la compilation JIT. TurboFan transforme le *bytecode* en une représentation sous forme d'arbre, connue sous le nom de *Sea of Nodes* [14], afin de faciliter plusieurs phases d'optimisation. Ces phases visent à produire un code machine extrêmement efficace pour l'exécution ultérieure de la fonction.

Le processus de compilation est illustré dans la Figure 3.

3.8 Web Assembly (Wasm)

Le WebAssembly (Wasm) est une technologie dont l'objectif est d'offrir des performances accrues et une portabilité élargie des applications web. Ce langage bas niveau permet d'écrire du code dans des langages tels que C, C++, ou Rust, puis de le compiler en un format binaire compact appelé *bytecode* Wasm. L'une des caractéristiques majeures du WebAssembly réside dans son typage statique, apportant une sécurité supplémentaire en identifiant les erreurs potentielles dès la phase de compilation. Cette approche procure des performances proches de celles du code natif, ouvrant

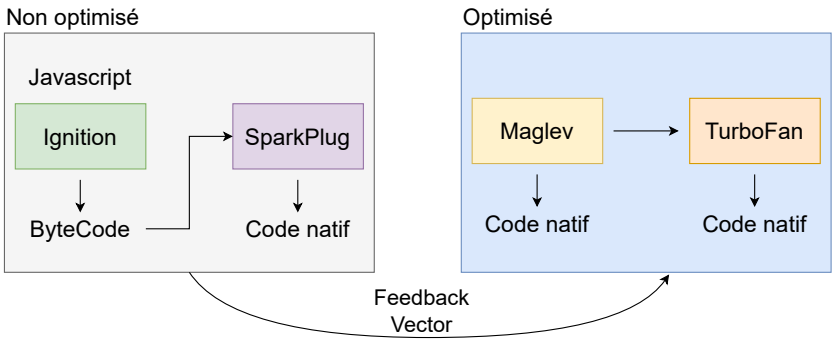

Fig. 3. Compilation du code Javascript

la voie à l'exécution rapide d'applications complexes directement dans le navigateur.

Le *bytecode* Wasm est compilé via un composant de V8 appelé Liftoff[14] [3]. Ce compilateur convertit le code Wasm en *bytecode* puis TurboFan se charge de transformer le *bytecode* en instructions natives. Le WebAssembly dispose également d'une interface d'application (API) qui facilite l'intégration et la communication avec le code JavaScript.

Ainsi, en raison de sa complexité, sécuriser un moteur tel que V8 représente un défi substantiel. De nombreuses vulnérabilités ont été exploitées, entraînant des conséquences néfastes pour les utilisateurs. De plus, de nouvelles techniques d'exploitation ont émergé et sont couramment utilisées dans la plupart des attaques.

Les principes fondamentaux concernant le fonctionnement et la gestion des objets JavaScript au sein du moteur V8, que nous avons précédemment établis, constituent une base solide de compréhension nous permettant d'engager une discussion plus approfondie dans les sections suivantes. Ces sections se concentreront sur l'examen de l'historique des méthodes d'exploitation fréquemment employées par les acteurs malveillants contre les navigateurs, ainsi que sur une analyse détaillée de ces méthodes.

4 Historique des attaques contre les navigateurs

Depuis l'Aurora Attack perpétrée contre Google en 2009 [33], les navigateurs web ont été constamment pris pour cible par des attaquants déterminés. Les attaques visant spécifiquement les navigateurs sont classées comme des attaques *1-click*. Dans ce scénario, l'utilisateur est sollicité

[14] `https://v8.dev/blog/liftoff`

pour accéder à un document ou une page web capable d'exécuter du Javascript. Les victimes peuvent être ciblées à travers des attaques de phishing, mettant en évidence l'ingénierie sociale comme vecteur d'attaque.

En 2021, il a été observé que 50% des attaques de type Zero-Day étaient dirigées contre des navigateurs [17]. Une fois qu'une vulnérabilité est exploitée, l'attaquant exécute généralement un shellcode, inaugurant la deuxième étape de l'exploitation consistant à échapper à la sandbox. En général, le noyau du système d'exploitation est la cible principale de cette phase, ajoutant une complexité significative.

Pour contrer ces attaques, de nombreuses protections ont été développées incitant les attaquants à ajuster leurs tactiques. Avant l'avènement de protections telles que l'*isolated heap* ou le *delayed free*, le *Document Object Model* (DOM) était souvent la cible, résultant sur des vulnérabilités de type *Use-After-Free* (UAF) [27]. Puis, des protections mises en place par exemple sous Windows, comme CFG (*Control Flow Guard*) [29] ou ACG (*Arbitrary Code Guard*) [30], ont réussi à complexifier des techniques d'exploitation telles que l'écrasement d'un pointeur de *vtable* ou la création dynamique de pages exécutables.

Cependant, l'application de ces protections à V8, qui réalise une compilation à la volée, pose des défis particuliers. CFG, par exemple, est une protection implémentée statiquement lors de la compilation, tandis que ACG ne peut être appliqué directement, car le compilateur requiert l'allocation dynamique de pages exécutables.

En conséquence, les attaquants se sont déplacés vers des cibles moins protégées telles que TurboFan comme par exemple au sein de la `CVE-2018-17463` [42]. De nouvelles techniques d'exploitation basées sur l'implémentation de primitives Javascript ont ainsi émergé. La plupart des exploits ciblant V8 utilisent des primitives permettant de dévoiler une adresse mémoire (*addrOf* - cf. section 5.4), puis d'obtenir une lecture et une écriture arbitraires en mémoire (corruption d'un `ArrayBuffer` et *fakeObj* - cf. section 5.5). Cette évolution souligne la nécessité constante d'innovation dans le domaine de la sécurité pour faire face aux tactiques changeantes et adaptatives des attaquants. La section suivante propose une exploration des principales stratégies d'exploitation qui ont été identifiées et utilisées au fil du temps.

5 Analyse des techniques d'exploitation ciblant le moteur V8

5.1 Définition d'une stratégie d'exploitation

L'exploitation d'une vulnérabilité, ouvrant la porte à l'exécution de code arbitraire à distance, suit généralement un ensemble d'étapes bien définies. Leurs exécutions peuvent cependant varier en fonction de la version de V8 ciblée ou de la vulnérabilité exploitée. Dans cette section, nous nous pencherons particulièrement sur l'une des stratégies les plus fréquemment employées lors d'attaques visant les moteurs Javascript, où l'attaquant doit successivement accomplir **trois** grands objectifs :

(I) Tout d'abord, il doit identifier une page mémoire exécutable ou élaborer une stratégie équivalente, par exemple, en construisant une chaîne de ROP (*Return-Oriented programming*).

(II) Ensuite, l'attaquant doit copier un *shellcode* sur une plage mémoire exécutable, ce dernier représentant le code malveillant à exécuter.

(III) Enfin, l'attaquant doit rediriger le flot d'exécution du programme vers le *shellcode*, initiant ainsi l'exécution du code arbitraire.

La réalisation de ces objectifs nécessite des compétences et des *capacités* spécifiques de la part de l'attaquant.

— Tout d'abord, la **capacité de lecture arbitraire** est cruciale. Cela permet à l'attaquant de lire le contenu d'objets en mémoire, récupérer les adresses des objets nécessaires, et, à la fin de l'attaque, obtenir l'adresse de la page mémoire exécutable.

— Ensuite, la **capacité d'écriture arbitraire** est essentielle pour écraser le contenu des pointeurs ciblés, permettant ainsi à l'attaquant d'écrire le *shellcode* dans la mémoire du processus visé.

Ces capacités sont typiquement acquises au fil de plusieurs étapes chaque étape visant à étendre progressivement la portée de la lecture ou de l'écriture. Ce processus complexe permet à l'attaquant, à la fin de l'attaque, de cibler l'entièreté de la mémoire.

Dans le schéma en Figure 4, un cas typique d'instanciation de cette stratégie d'exploitation est proposée, découpée en 6 étapes :

1. **Déclenchement de la vulnérabilité** : Le déclenchement de la vulnérabilité conduit généralement à une corruption de la mémoire, manifestée par des comportements tels que les dépassements d'indices de tableau (*Array Index Out-0f-Bound*) ou des confusions de type (*Type Confusion*). Ces corruptions offrent à l'attaquant la possibilité de lire ou d'écrire à des emplacements spécifiques en

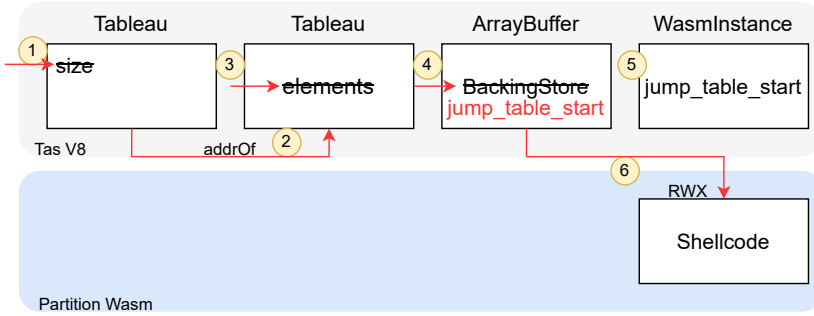

Fig. 4. Illustration de la structure de la mémoire du tas au moment de l'attaque

mémoire créant ainsi une configuration adéquate pour la suite de l'exploitation ;

2. **Fuite de l'adresse d'un objet Javascript** : cette étape est réalisée lors de la construction de la primitive *addrOf*. Il s'agit d'une fonction prenant en argument l'instance d'un objet et qui renvoie l'adresse de l'objet stocké sur le tas V8 (cf. section 5.4).
   Cette fuite d'information est réalisée en écrasant la valeur (par exemple de type *Float*) dans un tableau par l'adresse d'un objet ;
3. **Lecture et écriture dans le tas V8** : les objets Javascript permettent aux développeurs d'écrire des valeurs en mémoire puis de les manipuler ultérieurement. L'attaquant va ici chercher à corrompre la manière dont un objet *JSObject* recherche en mémoire ses éléments. En manipulant le champ *elements* d'un tableau par exemple, l'attaquant peut contrôler l'emplacement où celui-ci lira ou écrira ses éléments. Ces capacités sont pour autant limitées au tas V8. Si la compression de pointeur est activée, l'attaquant doit mettre en place une étape supplémentaire afin d'être en mesure d'écraser le pointeur *elements* qui est transformé en pointeur de 32 bits. C'est la primitive *fakeObj* qui permet à l'attaquant de contourner cette limitation (cf. section 5.5) ;
4. **Lecture et écriture arbitraire en mémoire** : avec la création des primitives précédentes, l'attaquant est désormais en mesure d'écraser le pointeur `BackingStore` de l'`ArrayBuffer`. Il étend alors ses capacités de lecture et d'écriture lui permettant de cibler une adresse en dehors du tas V8 ;
5. **Obtention d'une zone mémoire exécutable** (objectif I) : le contrôle de l'objet `BackingStore` devient particulièrement critique

lors de l'attaque de la page RWX (*Read-Write-Execute*) de l'objet *WasmInstanceObject* qui est utilisé par le moteur pour représenté l'environnement d'exécution du code Wasm. L'attaquant peut alors **écrire un code malveillant (*shellcode*) sur cette page** (objectif II) en utilisant notamment un objet `DataView`. En obtenant le contrôle du `BackingStore`, l'attaquant peut contourner les limitations initiales et finaliser son attaque ;

6. **Redirection du flot d'exécution** (objectif III) : L'invocation d'une fonction WebAssembly (Wasm) peut être effectuée à partir du langage Javascript. Ainsi, le *shellcode* peut être appelé de manière similaire à une fonction Wasm, car le code d'origine a été remplacé par celui-ci. Cette manipulation du flot d'exécution permet à l'attaquant de détourner le contrôle du programme vers le code malveillant, exécutant ainsi le *shellcode* avec les privilèges du processus compromis.

La figure 5 offre une illustration de l'évolution des capacités au fur et à mesure de la construction des primitives, jusqu'à parvenir à l'exécution du code arbitraire.

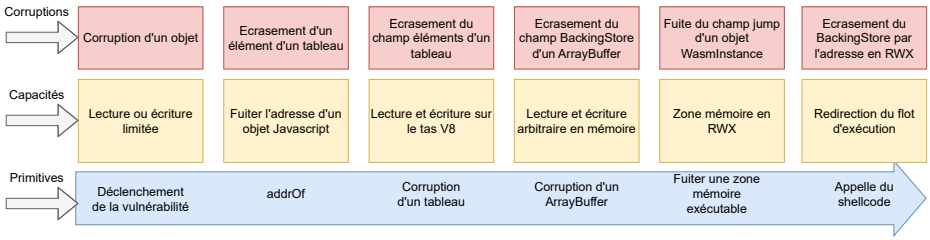

Fig. 5. Évolution des capacités de l'attaquant

L'application de cette stratégie par les attaquants est fortement influencée par les innovations introduites par les développeurs de V8. Cela est particulièrement vrai pour la compression de pointeur, dont l'effet sur la stratégie est examiné dans la section suivante.

5.2 La compression de pointeur

L'introduction de la compression de pointeur (évoquée section 3.3) dans le moteur V8 a été réalisé en 2020.[15] Cependant tous les pointeurs ne

[15] `https://v8.dev/blog/pointer-compression`

sont pas soumis à cette compression. C'est notamment le cas du pointeur `BackingStore` dans l'objet `ArrayBuffer`, qui a conservé un format de 64 bits (par exemple, l'adresse `0x000055a612ff5df0`).

De manière similaire, le pointeur vers la zone mémoire RWX `jump_table_start` dans l'objet *WasmInstanceObject* demeure un pointeur de 64 bits (par exemple : `0x0000266c107c2000`).

L'exploitation d'une vulnérabilité implique notamment la manipulation de ces deux pointeurs par l'attaquant.

Ainsi, dans ce contexte, l'attaquant doit être capable de gérer des adresses en 64 bits ou en 32 bits en fonction de la cible. Les autres objets présents sur le tas V8 utilisent majoritairement des adresses sous forme de pointeurs compressés de 32 bits, incluant les objets de type tableau (`Float`, `ArrayBuffer`, `WasmInstanceObject`).

5.3 Corruption de la taille d'un tableau

Lors de l'exploitation d'une vulnérabilité, l'attaquant va souvent tenter de corrompre la taille d'un tableau, lui donnant ainsi la possibilité de lire de la mémoire en dehors de ses limites et donc de bénéficier d'une situation de OOB (*Out-Of-Bound*). La suite de l'attaque est complexe et nécessite la construction de primitives telles que *addrOf* et *fakeObj*, ainsi que des techniques avancées d'écriture et de lecture arbitraires en mémoire pour transformer la vulnérabilité en exécution de code arbitraire.

5.4 Fuite de l'adresse d'un objet Javascript

La primitive *addrOf*. Afin de parvenir à la fuite de l'adresse de n'importe quel objet instancié dans le code JavaScript de l'exploit, l'attaquant développe une fonction, également appelée primitive, qui prend en paramètre l'instance de l'objet et retourne son adresse en mémoire sous forme de `Float`. Cette primitive est couramment désignée sous le nom de *addrOf*.

- En JavaScript, lorsqu'un objet est stocké dans un tableau d'objet, l'adresse de cet objet est copiée dans la structure *elements* du tableau ;
- de la même façon, un `Float` est également placé dans la structure *elements* d'un tableau de `Float` ;
- en substituant le `Float` par l'adresse de l'objet victime, l'attaquant induit une confusion de type, amenant l'interprétation de l'adresse comme un `Float` lors de sa consultation.

Lorsque cet élément est accédé, le moteur JavaScript, désormais trompé, ne renvoie pas la valeur initiale qui a été écrasée, mais plutôt l'adresse de

l'objet ciblé par l'attaquant. Le `Float` peut ensuite être converti en entier en utilisant les mécanismes de conversion du langage Javascript tel que les `TypedArray`, `Float64Array` et `Uint32Array`.

Construction de la primitive *addrOf*. La création de la primitive *addrOf* repose sur une manipulation habile de la mémoire où les objets nécessaires à l'attaque vont être placés de manière consécutive. La figure 6 permet de représenter cette configuration ainsi que le code Javascript correspondant.

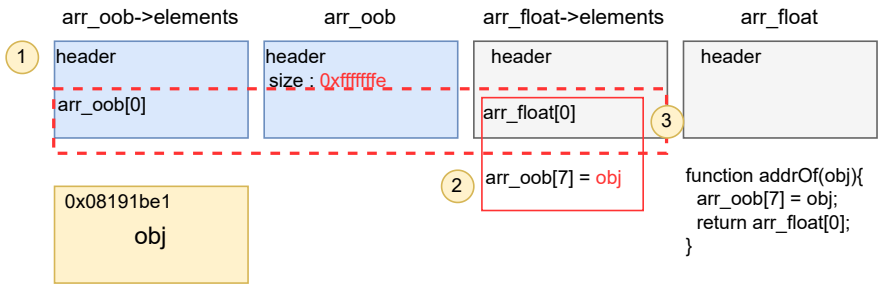

Fig. 6. Primitive *addrOf*

1. **Positionnement adjacent en mémoire** : L'attaquant place consécutivement le tableau en situation de OOB (ici appelé `arr_oob`) et un tableau de float (appelé `arr_float`) ;
2. **Écrasement d'un élément Float du tableau** : par la suite, l'attaquant effectue une indexation incorrecte en dépassant la taille initiale du tableau (`arr_oob`) et provoque un dépassement de ses bornes. Cette manipulation permet à l'attaquant d'écraser le premier élément `Float` du tableau adjacent (`arr_float`) avec l'adresse d'un objet victime ;
3. **Récupération de l'adresse** : La valeur écrite correspond à l'adresse d'un objet Javascript précédemment instancié par l'attaquant (`0x08191be1` sur la figure 6). Ainsi accéder à l'index 0 du tableau `arr_float` permet à l'attaquant de récupérer l'adresse de l'objet ciblé. Cet objet pourra être par exemple de type `ArrayBuffer` ou *WasmInstanceObject*.

Il convient de noter que l'attaquant récupère deux pointeurs compressés de 32 bits lors de la lecture d'un Float de 64 bits. L'attaquant utilise

ensuite des mécanismes bas niveaux tels que *shift* pour ne conserver que le pointeur compressé ciblé. Ainsi, la primitive *addrOf* permet à l'attaquant de récupérer l'adresse mémoire d'un objet donné, ouvrant ainsi la voie à des manipulations plus avancées.

5.5 Lecture et écriture sur le tas V8

Objectif. Pour lire et écrire de manière arbitraire dans la mémoire du tas V8, l'attaquant cherche à obtenir le contrôle du pointeur *elements* d'un tableau de `Float` via la primitive *fakeObj*. En effet, ce pointeur contient l'adresse de la structure *elements* qui stocke en mémoire les données sous forme de `Float`. L'attaquant en mesure de corrompre le pointeur *elements* d'un objet pourrait forcer le tableau à lire ou écrire sous forme de `Float` n'importe où dans la mémoire du tas V8. Cet objet est appelé *fakeObj* ou `fake` et possède les caractéristiques suivantes :

- L'attaquant contrôle complètement l'entête de l'objet et donc son pointeur *elements*. Pour cela, il va utiliser un autre tableau faisant office de conteneur ;
- La primitive *addrOf* est utilisée pour obtenir l'adresse ciblée par l'attaquant. Elle est ensuite utilisée pour écraser le champ *elements* du *fakeObj* ;
- Pour lire ou écrire en mémoire, il suffit à l'attaquant d'utiliser l'indexation du *fakeObj* (par exemple `fake[0]`) ;
- Trois primitives sont alors construites pour élaborer le système : *fakeObj*, *arbRead* et *arbWrite*

La primitive *fakeObj*. Pour construire en mémoire le faux objet, l'attaquant va utiliser comme conteneur un autre tableau de `Float`. En effet, l'attaquant peut placer les valeurs nécessaires à un entête dans la structure *elements* de ce conteneur. La figure 7 représente la mémoire avec l'entête du faux objet stocké dans le tableau conteneur (nommé ici `container`). En utilisant cette configuration, le pointeur *elements* du faux objet peut être modifié par l'attaquant en utilisant l'index 1 du conteneur (`container[1]`).

Pour utiliser l'objet `fake` tel que décrit précédemment, l'attaquant doit être en mesure de transformer son adresse en objet. Pour cela, l'attaquant va leurrer le moteur Javascript en le forçant à considérer l'adresse du début de l'entête comme un véritable objet. Un nouveau tableau, cette fois de type objet, se rajoute à l'équation. Dans la figure 8, il est nommé `arr_obj`. Un tableau d'objet à la particularité de pouvoir stocker dans

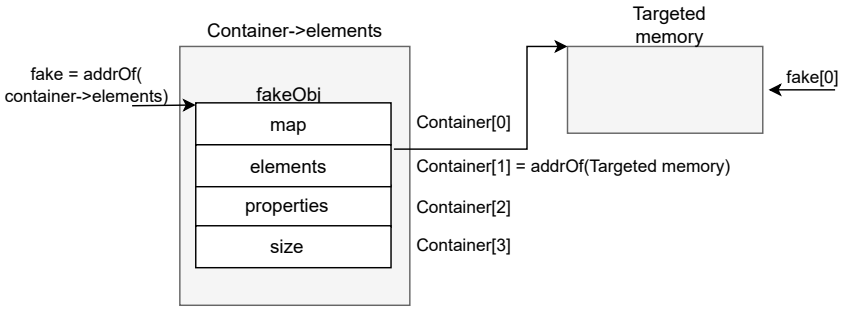

Fig. 7. Utilisation du conteneur pour stocker le faux objet

sa structure *elements* l'adresse d'un objet. La stratégie de l'attaquant est d'écraser l'élément de `arr_obj` par l'adresse du faux objet. Les étapes sont les suivantes :

1. une écriture en dehors des limites du tableau OOB (`arr_oob`) est réalisée pour venir écraser l'élément de `arr_obj` ;
2. le moteur Javascript est trompé et considère l'adresse du faux objet (soit l'adresse de l'élément 0 de l'objet conteneur `container[0]`) comme un objet ;
3. interroger le tableau `arr_obj` permet d'obtenir une référence sous forme d'objet de fake (`fake = arr_obj[0]`).

Ces étapes sont résumées sur la figure 8.

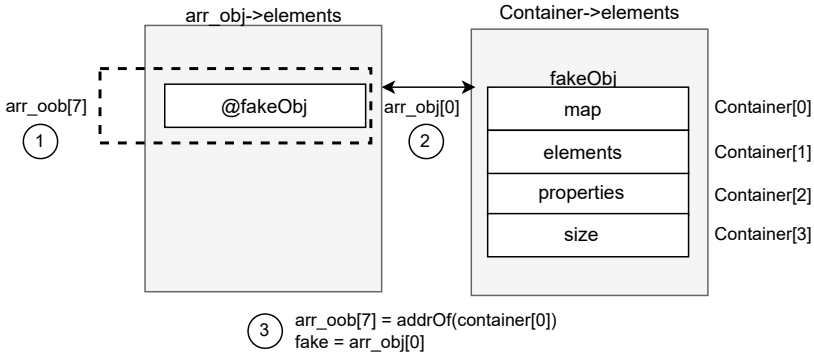

Fig. 8. Utilisation du tableau d'objet pour injecter le faux objet

Les étapes décrites précédemment sont placées dans une fonction Javascript dont le code est le suivant :

```
//FakeObj
  //in :  memory address as an unsigned int
  //out : fake object
  function fakeObj(addr)  {
      arr_oob[7] = itof(addr);
      let fake = arr_obj[0];
      return fake;
  }
  var fake = fakeObj(addrOf(container) - 0x20n);
```

Les Primitives *arbRead* et *arbWrite* Les deux fonctions, *arbRead* et *arbWrite*, utilisent le faux objet pour lire et écrire de manière arbitraire en mémoire. Ces fonctions prennent en paramètre une adresse (offset de 32 bits) qui est placée à l'index 1 du tableau conteneur `container`, ce qui a pour effet de changer le pointeur *elements* du faux objet. La lecture ou l'écriture à cette adresse est réalisé en en utilisant l'indexation du faux objet (`fake[0]` par exemple).
Le code Javascript correspondant est le suivant :

```
//Read Primitive
//in :  object
//out : float
function arbRead(addr) {
    container[1] = itof(addr);
    return (fake[0]);
  }

//Write Primitive
//in : unsigned int memory address, int value
//out : NA
function arbWrite(addr, val) {
    container[1] = itof(addr);
    fake[0] = itof(BigInt(val));
}
```

Cet article ne couvre pas toutes les subtilités techniques liées au *fakeObj*. Son objectif est de fournir aux lecteurs les connaissances nécessaires pour appréhender les évolutions de la sandbox V8 abordées section 6. Voici cependant quelques points d'attention à retenir :

— L'écriture directe d'une adresse en utilisant le tableau `arr_oob` n'est pas possible si celui-ci est de type int. En effet, une opération de décalage est effectuée sur la valeur avant son enregistrement en mémoire. Ainsi, l'attaquant doit disposer d'un tableau de type `Float` en situation de dépassement de capacité (*Out Of Bounds*), avec son champ `size` corrompu par une valeur plus grande ;

— L'adresse stockée dans le tableau d'objet est une adresse compressée de 32 bits. Pour un attaquant capable d'écrire 64 bits, la gestion de l'écrasement d'une valeur adjacente peut parfois poser problème ;
— Lorsque le moteur V8 utilise le champ *elements* d'un *JSObject*, il ajoute ensuite 8 octets pour contourner l'en-tête de la structure et accéder véritablement aux contenus de l'objet. Lorsque l'attaquant utilise *arbRead* ou *arbWrite*, il doit tenir compte de ce décalage.

Le lecteur curieux est encouragé à se référer à des ressources telles que [24] pour obtenir plus de détails.

5.6 Contrôle du *BackingStore* d'un *ArrayBuffer*

À ce stade, l'attaquant est en mesure d'obtenir l'adresse de n'importe quel objet instancié, d'effectuer des opérations arithmétiques sur cette adresse, et d'y écrire. Cependant, pour pouvoir cibler la page RWX d'un objet Wasm, il faut étendre ces capacités afin de pouvoir cibler de la mémoire en dehors du tas V8. Pour cela, l'attaque cible l'objet `ArrayBuffer` dont le buffer de donnée est également situé en dehors du tas V8. C'est le pointeur `BackingStore`, un pointeur non compressé sur 64 bits, qui est utilisé pour référencer le buffer. L'exploitation implique l'écrasement de ce pointeur avec l'adresse de la page en mémoire RWX également un pointeur non compressé de 64 bits.

5.7 Plage mémoire RWX

Zone mémoire du code JIT. La recherche ou l'acquisition d'une page mémoire dotée d'autorisations d'exécution finalise l'attaque. Avant l'intégration de la restriction W⊕X en 2017, les attaques exploitaient fréquemment la page allouée pour exécuter le code JIT. Cependant, ces pages sont désormais restreintes soit en écriture soit en exécution ce qui rend leur utilisation plus complexe [35].

Zone mémoire du code Wasm. Les attaquants ont ensuite exploré une autre opportunité en exploitant l'emplacement utilisé pour exécuter du code Wasm. Cette zone mémoire, bien qu'elle ne soit pas protégée par W⊕X, peut être sécurisée par *wasm-memory-protection-keys* sous Linux avec un processeur Intel compatible [36]. Cette protection permet d'attribuer à une source spécifique différents droits.

— Protection en exécution : la mémoire du code peut se voir attribuer une clé de protection autorisant uniquement l'exécution (mais pas

les opérations de lecture ou d'écriture) depuis le code Wasm uniquement. Cela empêche d'autres parties du programme ou d'autres processus d'exécuter du code Wasm arbitraire ;
— Protection en lecture et d'écriture : la mémoire des données peut se voir attribuer une clé de protection permettant uniquement les opérations de lecture et d'écriture depuis le code Wasm, restreignant davantage les actions possibles des autres parties du programme ou d'autres processus.

Il existe toutefois une technique de contournement permettant de désactiver cette protection en ciblant le drapeau `FLAG_wasm_memory_protection_keys` après l'obtention par l'attaquant des primitives de lecture et d'écriture arbitraire [31].

Récupération de l'adresse d'une Page RWX Pour obtenir l'adresse mémoire de la page en RWX provenant du code Wasm, la primitive *addrOf* est utilisée pour récupérer l'adresse stockée dans l'objet *WasmInstanceObject* (`jump_table_start`). Ensuite, la primitive *arbRead* est employée pour lire le pointeur `jump_table_start` et ainsi récupérer l'adresse de la page RWX.

5.8 Redirection du flot d'exécution

Une fois que le pointeur `BackingStore` est substitué par l'adresse de la page RWX, l'attaquant peut effectuer une opération de copie du *shellcode* sur cette page. Le *shellcode*, qui représente le code malveillant à exécuter, est ainsi enfin positionné dans une zone mémoire permettant son exécution. Le flot d'exécution peut ensuite être redirigé en appelant la représentation du code Wasm sous forme de fonction Javascript. L'attaquant a écrasé le code Wasm initial avec le *shellcode*. Ainsi, lorsque le code Wasm est appelé, le *shellcode* est effectivement exécuté.

L'exploitation met en lumière la vulnérabilité potentielle des pointeurs non sandboxés référençant des zones mémoire en dehors du tas V8, et souligne la nécessité cruciale de renforcer la protection de ces références sensibles. L'attaque a démontré comment la compromission du pointeur `BackingStore` d'un `ArrayBuffer`, un pointeur non compressé sur 64 bits, pouvait permettre à un attaquant d'écraser ce pointeur avec l'adresse d'une page RWX, ouvrant ainsi la voie à des manipulations malveillantes.

Cette nécessité de protéger les pointeurs sensibles tels que le `BackingStore` ou l'adresse de la page RWX Wasm souligne l'importance des mécanismes de sécurité pour prévenir les attaques. Les vulnérabilités

qui permettent la compromission de ces pointeurs sensibles peuvent avoir des conséquences graves, compromettant l'intégrité du flot d'exécution de V8. Les mécanismes de sécurité, notamment la sandbox V8, ont été développés pour mitiger les risques liés à ces vulnérabilités. La sandbox V8, examinée section 6, vise à restreindre l'accès et la manipulation arbitraires au sein du tas V8 confiné.

5.9 Attaques ciblant l'objet *TheHole*

Les attaques ciblant l'objet *TheHole* ont connu une recrudescence en 2023, imposant des techniques d'exploitation incontournables pour qui veut s'intéresser à la sécurité du navigateur V8. *TheHole* est un élément utilisé par V8 pour représenter un type de valeur particulier similaire à `true`, `false`, `null` et `undefined` en JavaScript. Cet objet a été pris pour cible notamment dans plusieurs Zero-Day observées en 2023 et listées à la fin de cette section.

TheHole est principalement utilisé dans la gestion des objets de type `Array` et `Map`, où il représente un élément manquant ou supprimé. Par exemple, lorsqu'un élément est supprimé d'un tableau, V8 utilise *TheHole* pour marquer l'espace, réalisant ainsi de l'optimisation afin de gérer efficacement la représentation de valeurs situées à des index éloignés [5]. En revanche, *TheHole* n'est pas directement accessible depuis un programme JavaScript. Les attaquants parviennent cependant à fuiter l'objet *TheHole* dans l'exploitation de certaines vulnérabilités puis à le manipuler de manières à obtenir de nouvelles capacités d'exploitation. C'est notamment le cas des vulnérabilités `CVE-2021-38003` [7] et `CVE-2022-1364` [44] qui provoquent une fuite de l'objet *TheHole*, entraînant sa manipulation par l'attaquant dans du code Javascript. En exploitant cette inconsistance du moteur, l'attaquant peut ensuite réaliser une compromission complète du navigateur en ciblant notamment les objets de type `Map`. En effet, lorsqu'un élément est supprimé d'un objet `Map`, il est remplacé par la valeur *TheHole*. La suppression d'une paire clé-valeur d'une `Map` contenant déjà comme valeur *TheHole* conduit cet objet à supprimer ses éléments de manière incorrect, résultant sur une mise à jour de sa taille jusqu'à atteindre la valeur de -1. Une condition similaire à la corruption de la taille d'un tableau mentionnée précédemment dans cet article section 5.

Plusieurs correctifs ont été apporté à l'objet `Map` afin d'empêcher son utilisation dans de futur exploitation [45]. Cependant en 2023, l'objet *TheHole* a de nouveau été pris pour cible par les CVEs : `CVE-2023-2033` [21], `CVE-2023-3079` [22] et `CVE-2023-4762` [20]. Les attaquants ont réussi à fuiter l'objet *TheHole* en ciblant l'optimisateur TurboFan, puis lors de

l'exécution des phases d'optimisation, l'erreur induite par la présence de l'objet *TheHole* se propage au reste de l'exécution du script. Ces manipulations permettent à l'attaquant d'utiliser l'objet *TheHole* comme index dans un tableau provoquant une situation similaire à celle des *Typer Bugs* [13]. En effet, lorsque l'optimisateur rencontre l'objet *TheHole* comme index, la vérification des bornes du tableau est supprimée. L'attaquant peut ensuite utiliser le tableau pour construire l'exploit [26] comme décrit section 5.

Pour répondre à ces nouvelles attaques, un durcissement (*hardening*) ciblant l'objet *TheHole* a été mise en place par les développeurs. En particulier, un changement de paradigme a été initié afin de fragmenter l'objet *TheHole* en plusieurs sous objets : un pour chaque type d'objet (`Map`, `Array` etc.) [47]. Par exemple, la fuite d'un objet *TheHole* provenant d'un objet `Map`, ne pourrait plus être réutilisée dans un objet `Array` car celui-ci ne pourrait manipuler que le type *TheHoleArray* ce qui permettrait ainsi de réduire la possibilité de confusion de type.

Les attaques exploitant l'objet *TheHole* engendrent une complexification des étapes d'exploitation, nécessitant des étapes supplémentaires par rapport aux primitives exposées antérieurement dans cet article telles que *addrOf*, *fakeObj*, *arbRead*, et *arbWrite*. Cependant, elles ne permettent pas d'évader la sandbox en utilisant par exemple les techniques décrites dans la section suivante mais introduisent une nouvelle classe d'attaque.

6 La sandbox V8

Les techniques utilisées dans les exploits ciblant V8 mettent en lumière la nécessité de protéger les pointeurs sensibles. Pour répondre à cette nécessité, un dispositif de confinement a été élaboré, connu sous le nom de sandbox V8 [38, 48]. La sandbox V8 est un mécanisme d'isolation conçu pour réduire l'impact de l'exploitation de vulnérabilités en restreignant les capacités de lecture et d'écriture arbitraire de l'attaquant à l'intérieur d'un environnement clos. Cette section propose de détailler l'architecture de la sandbox, d'étudier les tactiques d'évasion utilisées puis de lister les évolutions potentielles.

6.1 Architecture de la sandbox V8

La conception générale de la sandbox V8 vise à empêcher un attaquant exploitant une vulnérabilité de corrompre d'autres parties de la mémoire du processus et d'exécuter un code arbitraire tout en minimisant le surcoût de performance [48].

Comme indiqué précédemment, le moteur Javascript V8 utilise depuis 2020 la compression de pointeur, une évolution permettant d'optimiser l'utilisation de la mémoire. Chaque référence d'un objet V8 dans le tas pointe alors vers un autre objet du tas avec un offset de 32 bits à partir de la base du tas. L'espace mémoire virtuel ainsi obtenu est appelé *Pointer Compression Cage* ou *Main Cage*, une dénomination que nous privilégierons dans la suite de cet article pour marquer la distinction avec d'autres espaces mémoire de la sandbox. Malgré la mise en place de la compression de pointeurs, certains objets continuent d'utiliser des *raw pointer* sur 64 bits (aussi appelé pointeur non sandboxé) pour référencer des objets à l'extérieur du tas V8. L'objectif de la sandbox V8 est donc d'étendre les mécanismes de protection apportés par la compression de pointeur de manière à également protéger ces *raw pointer*.

En effet, les vulnérabilités dans V8 peuvent conduire à des corruptions de mémoire des objets. Cependant, avec l'activation de la compression de pointeurs, un attaquant ayant la capacité de modifier des objets dans le tas V8 est confronté à des limitations significatives, notamment l'incapacité à sortir de la sandbox uniquement en manipulant des pointeurs compressés. Pour accéder à l'extérieur du tas V8, l'attaquant doit alors cibler l'un des *raw pointers* restants dans le tas, tels que les pointeurs de stockage de tableau `ArrayBuffer` (le `BackingStore` – cf. section 3.4) ou `TypedArray`.

Pour renforcer l'efficacité de la sandbox, les *raw pointers* doivent être éliminés du tas V8. Un mécanisme clé pour cela est l'utilisation de l'*External Pointer Table* (EPT). L'EPT est une table de pointeurs créée pour contenir les *raw pointers* de la sandbox. Un *raw pointer* peut aussi correspondre à un *wrapper* vers un objet HTML instancié dans Blink. Dans l'objet V8 sandboxé, le *raw pointer* est transformé en index de l'EPT et en suivant cet index, le *raw pointer* initial peut être retrouvé par l'objet. Pour assurer la protection de la table, celle-ci est placée en dehors de la sandbox. D'autres mesures de protection sont également implémentées, telles que la vérification du type de pointeur lors d'un accès, la protection contre l'accès concurrent par d'autres threads, ou la protection contre les accès hors limites de la table.

La protection du buffer de données (`BackingStore`) d'un `ArrayBuffer` est assurée par un autre espace mémoire virtuel appelé `ArrayBuffer Partition Cage`. Le `BackingStore` est stocké dans cet espace, tandis que l'objet `ArrayBuffer` est situé dans la *Main Cage*. Ainsi, le *raw pointer* vers le `BackingStore` (dans l'objet `ArrayBuffer`) est transformé en un nouveau type de pointeur appelé *sandboxed pointer* [41].

Le *sandboxed pointer* est un offset de 40 bits à partir du début de la sandbox. Par exemple, pour une sandbox de 1TiB=2^{40} octets [46] avec :

- `b` : l'adresse de base de la sandbox
  - `b = 0xa38d00000000` ;
- `p` : l'adresse du `BackingStore` d'un `ArrayBuffer`
  - `p = 0xa44d667df000` ;
- `sp` : le *SandboxedPointer* de `p`
  - `sp` serait référencé comme l'offset sur 40 bits
  - `sp = (p-b) = 0xc0667df000`.
- Pour permettre le stockage du pointeur dans un registre de 64 bits, le *SandboxedPointer* est décalé vers la gauche de 24 bits soit :
  - `sp = (p-b)«24 = 0xc0667df000000000`.

Un attaquant qui corromprait cette valeur n'aurait accès qu'à une zone mémoire *à l'intérieur* de la sandbox de 1TiB (puisque la valeur sera comprise entre `0xa38d00000000` et $2^{40} - 1$) et non en-dehors. Lors de l'accès, la valeur `0xc0667df000000000` est *décodée* en un pointeur brut en le décalant d'abord vers la droite, puis en l'ajoutant à la base de la sandbox permettant d'obtenir l'adresse `0xa44d667df000`.

Ainsi, le terme sandbox fait référence à un vaste espace d'adressage virtuel qui comprend plusieurs espaces isolés, dont la *Main Cage* et l'*ArrayBuffer Partition Cage*. Cette architecture a été renforcée à la suite d'attaques réussies ayant permis d'échapper à la sandbox. La suite de cette section se penche sur l'étude approfondie de ces attaques, mettant en lumière les vulnérabilités exploitées et les réponses apportées pour renforcer la protection de la sandbox. L'analyse des attaques et des contre-mesures offrira un aperçu essentiel de l'évolution des mécanismes de sécurité dans V8.

6.2 Protection du pointeur `BackingStore`

La sandbox est activée par défaut depuis décembre 2022 pour les architectures x64. Ainsi les techniques d'exploitation telles que décrites lors de l'étude des techniques d'exploitation de cet article (section 5) sont en partie neutralisées.

En effet, avec la mise en place de la sandbox telle que décrite précédemment, l'exploitation permet toujours à l'attaquant d'obtenir une lecture et une écriture arbitraire. La sandbox, bien que ne prévenant pas le déclenchement de la vulnérabilité ou la corruption de la taille d'un tableau, limite néanmoins les possibilités d'exploitation en empêchant la corruption des objets `ArrayBuffer` et Wasm. Par conséquent, l'attaquant ne peut pas parvenir à une exécution de code arbitraire. L'attaquant peut

extraire l'adresse de la page RWX d'un objet Wasm à l'aide de la primitive *addrOf*. Cependant, lors de sa tentative d'utilisation de la primitive *fakeObj* pour écraser le pointeur du `BackingStore` avec cette adresse, la sandbox intervient en interprétant l'écrasement du pointeur de 40 bits comme un offset lors de l'accès au `BackingStore`. La configuration des pointeurs de type *sandboxed pointer* réussit donc efficacement à contrer une telle manipulation. Avec l'activation de la sandbox, exploiter une vulnérabilité nécessite désormais que l'attaquant ajoute des étapes supplémentaires pour échapper la sandbox.

6.3 Évolution de la sandbox face aux techniques d'évasion

La robustesse de la sandbox V8 est mise à l'épreuve par des attaquants cherchant à contourner ses mécanismes de sécurité. Plusieurs techniques avancées émergent dans cet objectif : la manipulation du point d'entrée du code JIT dans l'objet `Function` via la technique du *JIT Spray*, la corruption du point d'entrée du *ByteCode* dans l'objet `ByteCodeArray` aussi appelé *Fake Stack*, et l'exploitation du `jump_table_start` de la structure `WasmInstanceObject`, la technique des *Wasm Gadgets*. Ces méthodes sophistiquées permettent aux attaquants de dépasser les limites imposées par la sandbox V8, ouvrant ainsi la voie à l'exécution de code arbitraire. Pour contrecarrer ces attaques, les mécanismes de la sandbox sont étendus avec la mise en place d'une nouvelle table de pointeurs appelée *Code Pointer Table (CPT)*, couplée à l'ajout d'un nouvel espace mémoire isolé : le *Trusted Space*.

La figure 9 propose une vue chronologique de l'évolution des mécanismes de la sandbox dans le temps, en mettant en lumière les réponses apportées face aux différentes attaques. La timeline sert également de repère pour situer les techniques d'échappement de la sandbox ainsi que les Zero-Days ciblant le moteur V8.

Dans la suite de cette section, nous explorerons en détail ces attaques et ces mécanismes de protection en examinant comment elles sont mises en œuvre, leurs implications et leurs limitations.

JIT Spray. L'attaque consiste en la manipulation du point d'entrée du code JIT provenant de l'objet `Function`. Le pointeur est écrasé par l'adresse du début d'une chaîne de gadgets de type JOP (*Jump-Oriented Programming*). En effet, dans une telle chaîne, chaque gadget termine par un saut vers le gadget suivant.

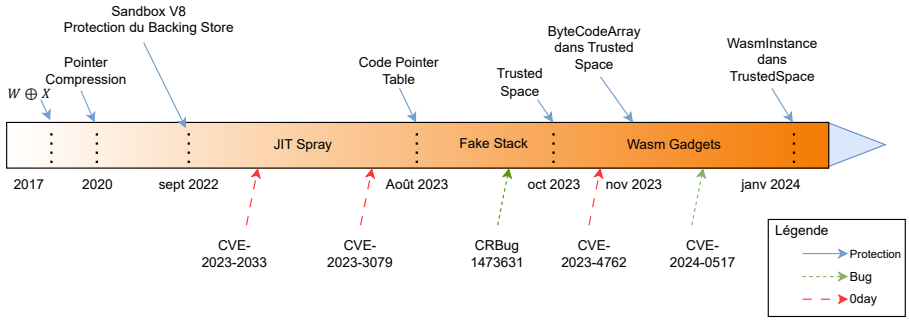

Fig. 9. Chronologie des évolutions architecturales et des attaques affectant la sandbox v8

Les fonctions Javascript sont représentées en mémoire sous la forme d'objets `Function`, essentiels à l'exécution du code. Cependant, leur utilisation peut être détournée par des attaquants cherchant à contrôler le pointeur `code_entry_point`, responsable de l'adresse du point d'entrée du code compilé à la volet par *TurboFan*.

L'objet `Function` est situé sur la *Main Cage* avec les autres objets *JSObject*. Le pointeur `code_entry_point` est un *raw pointer* faisant référence à une zone mémoire exécutable. Cette zone est cependant protégée par W⊕X qui empêche la mémoire d'être accessible en écriture et en exécution simultanément.

Cette situation est propice à l'exécution de code arbitraire car l'attaquant dispose d'une zone mémoire exécutable et de la possibilité de rediriger le flot d'exécution vers cette zone (en écrasant le pointeur `code_entry_point`). Le dernier prérequis est de pouvoir écrire un *shellcode* dans cette zone. Pour réaliser cela, l'attaquant place un *shellcode* sous forme de JOP dans un tableau de `Float`. La représentation en `Float` permet en effet de stocker des opcodes sur 8 octets. Une fois placés dans un tableau, les opcodes du *shellcode* sont placés en mémoire exécutable lors de la compilation à la volée. Chaque élément `Float` du tableau devient ainsi un petit gadget JOP, terminant par un saut vers le gadget suivant. Le point d'entrée de la fonction est écrasé pour pointer vers le premier `Float` du tableau, c'est-à-dire le premier opcode du *shellcode*. L'exécution de la fonction exécute ainsi le *shellcode*. Un attaquant en capacité de lire et d'écrire arbitrairement dans la *Main Cage* peut dès lors cibler le *raw pointer* `code_entry_point` pour échapper la sandbox.

La figure 10 illustre l'attaque en mettant en avant les étapes suivantes :

1. La primitive *addrOf* est utilisée pour récupérer l'adresse du `code_entry_point` dans l'objet `Function` ;
2. le tableau de `Float` est ensuite construit pour entreposer les opcodes du *shellcode* sous forme de `Float` ;
3. la fonction Javascript est appelée de nombreuses fois de manière à déclencher l'optimisation ;
4. les primitives *fakeObj* et *arbWrite* sont utilisées pour écraser le `code_entry_point` en le remplaçant par l'adresse du premier `Float` du tableau ;
5. La fonction est ensuite appelée une nouvelle fois de manière à exécuter le premier gadget JOP.

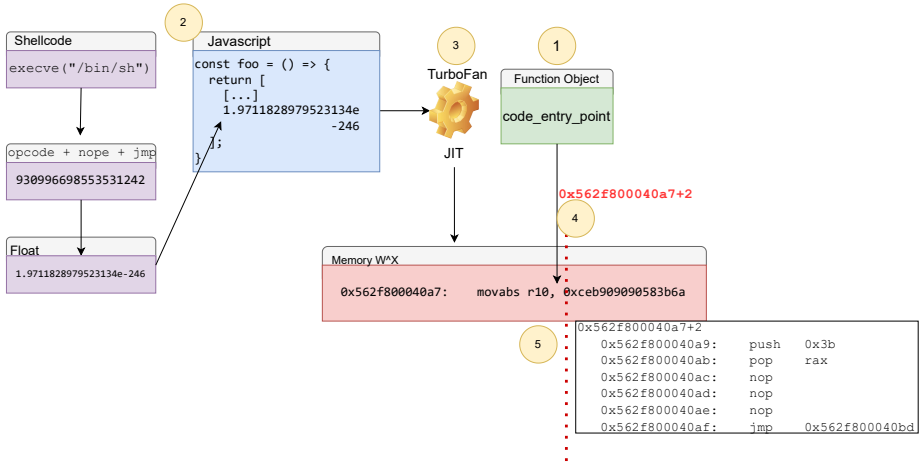

Fig. 10. Corruption du *raw pointer* `code_entry_point`

Code Pointer Table (CPT). Pour ce prémunir de cette attaque, le `code_entry_point` a été déplacé en dehors de la sandbox dans une table externe appelée la *Code Pointer Table* (CPT). Ainsi, le `code_entry_point` n'est plus un *raw pointer*, mais un offset vers la *Code Pointer Table*, qui contient ensuite le pointeur vers le point d'entrée.

La CPT est similaire à l'EPT présentée section 6.1. Ce sont toutes deux des tables de pointeurs, cependant l'EPT est réservée au référencement des objets externes à la sandbox. La CPT joue un rôle différent, car

son objectif est de protéger le code exécutable. Pour cela, une approche par CFI (*Control Flow Integrity*) est réalisée. Le CFI permet de vérifier l'intégrité du code *avant* son exécution, en s'assurant, par exemple, que des appels illégitimes à des fonctions systèmes (*syscall*) ne soient pas présents, et/ou en intégrant des vérifications basées sur la signature des prototypes de fonctions (composée des informations telles que la convention d'appel, les arguments, et le premier opcode).

Fake Stack. Suite à la sécurisation du `code_entry_point` dans la CPT en août 2023, une nouvelle technique d'évasion a été développée, ciblant cette fois le champ `ByteCodeArray`.

À ce jour, aucune analyse publique n'a détaillé le fonctionnement précis de cette technique. Néanmoins, le code d'exploitation complet utilisé pour la vulnérabilité identifiée est disponible sur le site de *ticketing* de Google sous la référence CRbug `#1473631` [19], permettant ainsi d'en dessiner les contours. Cette attaque vise donc à déclencher une séquence classique de ROP au moment de l'instruction de retour d'une fonction *Built-in* (cf. section 3.1). Ce qui rend l'attaque inédite c'est la capacité de l'attaquant à tromper la machine virtuelle Ignition, l'incitant à utiliser une fausse pile lors de l'exécution du *ByteCode*.

En effet, pour exécuter du *ByteCode* Javascript, Ignition enregistre le code résultant de la compilation, ainsi que d'autres informations nécessaires à l'exécution, dans un tableau appelé le `ByteCodeArray`. Ce tableau est référencé par un *raw pointer*, au sein de la structure `SharedInfoFunction`.

L'attaquant en capacité d'écraser le pointeur vers le `ByteCodeArray` va forger un nouveau tableau en y plaçant un enchaînement d'opcodes spécialement conçu pour forcer l'interpréteur, lors du retour à la fonction appelante, à utiliser une fausse pile. Cette fausse pile est entièrement sous le contrôle de l'attaquant et lui permet de placer l'adresse du début de la chaîne ROP à la place de l'adresse de retour.

La figure 11 représente la pile de la fonction Javascript déclarée par l'attaquant, ainsi que la fausse pile utilisée par Ignition lors du retour vers la fonction parente. Les étapes nécessaires à l'attaque, détaillées maintenant, sont également indiquées sur la figure.

1. L'attaquant utilise les primitives *addrOf* et *fakeObj* pour retrouver la représentation en mémoire sous forme d'objet de la fonction `SharedInfoFunction`. L'attaquant écrase le pointeur `ByteCodeArray` pour le remplacer par un tableau d'opcode sous son contrôle. Ce pointeur est ensuite copié par Ignition sur la pile

(*stack*), remplaçant ainsi le *ByteCode* initial de la fonction par celui forgé par l'attaquant ;

2. Lors de l'exécution du *ByteCode* malveillant, le pointeur *Previous Stack Frame* est écrasé par l'adresse de la fausse pile sous le contrôle de l'attaquant ;
3. Dans cette fausse pile, l'adresse de retour est modifiée pour rediriger le flot d'exécution vers une chaîne de ROP.

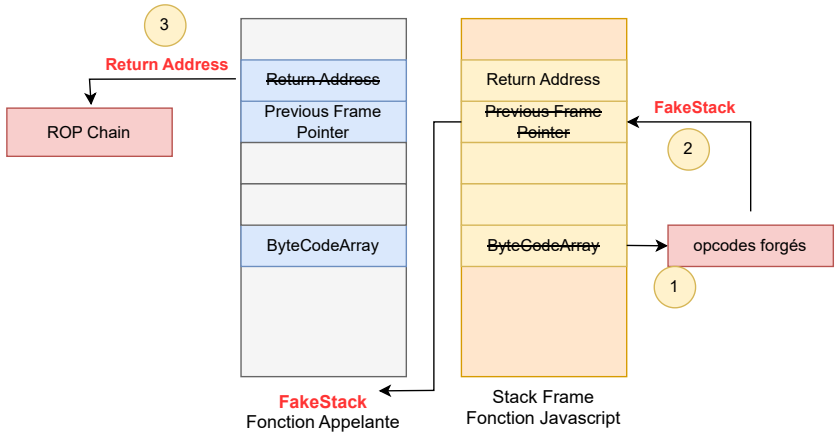

Fig. 11. Étapes du `ByteCodeArray` ROP

Trusted Space. L'attaque contre le `ByteCodeArray` montre la nécessité de protéger des objets critiques internes au tas de V8. En effet, les objets contenant du *ByteCode*, du code machine ou même des meta-données peuvent être ciblés par les attaquants pour échapper la sandbox. Pour pallier à ce problème un nouveau mécanisme appelé *Trusted Space* a été ajouté à la sandbox V8 en octobre 2023 [50]. La figure 12 illustre l'architecture du *TrustedSpace*.

Les objets critiques, appelés *Trusted Objects*, sont désormais déplacés dans un espace isolé appelé *Trusted Space*. Cet espace mémoire est une *Memory Cage* à l'instar de la *Main Cage*. Deux types de pointeurs sont alors créés pour permettre l'accès aux objets sur le *Trusted Space* :

— *Trusted pointer* : un objet sur la *Main Cage* va accéder à un objet sur le *Trusted Space* via une nouvelle table de pointeur appelée la

Trusted Pointer Table (TPT). Ainsi, le pointeur est représenté par un index dans la TPT ;

— *Protected pointer* : un objet dans le *Trusted Space* accède à un autre objet du *Trusted Space* en utilisant un offset sur 32 bits par rapport à l'adresse de base du *Trusted Space*. Il s'agit d'un pointeur compressé comme ceux existant dans la *Main Cage*.

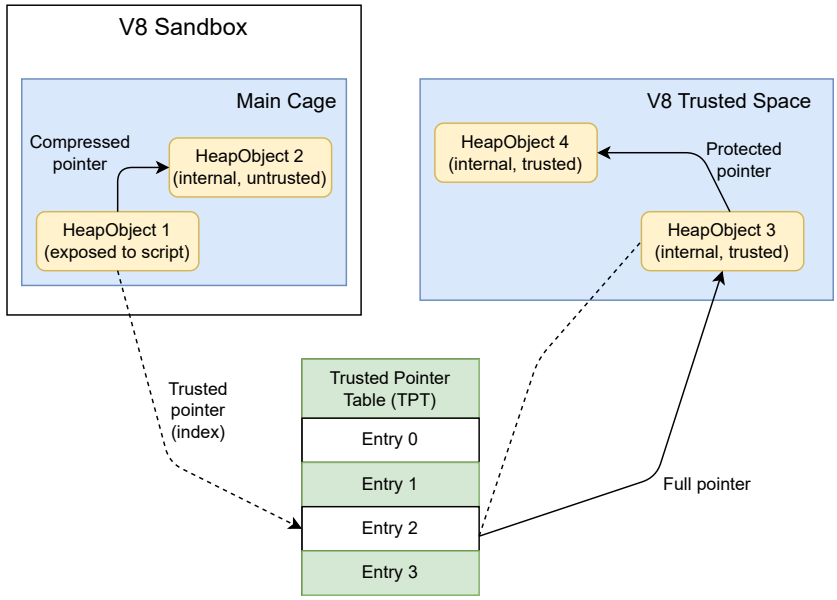

Fig. 12. *Trusted Space*

Wasm Gadgets. Malgré les multiples protections intégrées à la sandbox V8, les attaquants ont démontré leur capacité d'adaptation en développant une nouvelle technique d'évasion, cette fois en ciblant spécifiquement l'environnement d'exécution du code WebAssembly. Cette technique a notamment été exploitée sur la `CVE-2024-0517` impactant le compilateur Maglev [12], et la `CVE-2023-4762` qui permet de faire fuiter l'objet *The-Hole* [20, 23]. En effet, le code *Wasm* compilé par Turbofan est placé dans une page RWX. Cette page est référencée via le champ `jump_table_start` de la structure `WasmInstanceObject` sous forme de raw pointer. En ciblant ce pointeur, l'attaquant peut contourner la sandbox V8 et contrôler le flot d'exécution.

Lors de l'appel à une fonction *Wasm*, le flot d'exécution est d'abord redirigé vers une *Jump Table* composée de plusieurs instructions assembleur *jmp*, dont l'objectif est de sauter vers le code de la fonction correspondante. Lors du premier appel d'une fonction, le code n'est pas encore présent sur la page RWX ; il est compilé lors de ce premier appel, selon une technique connue sous le nom de *lazy compilation*.[16] Cette compilation est réalisée par un appel à la fonction *WasmCompileLazy*. Le champ `jump_table_start` pointe donc vers la jump table, comme représenté en figure 13. Ce champ n'est utilisé qu'une seule fois lors du premier appel de la fonction. Ensuite, la fonction est représentée en mémoire par un objet *function* avec son propre point d'entrée. Lorsqu'une fonction *Wasm* est appelée, ses arguments sont stockés dans les registres `RAX`, `RDX` et `RCX`, ce qui permet à l'attaquant de contrôler ces registres.

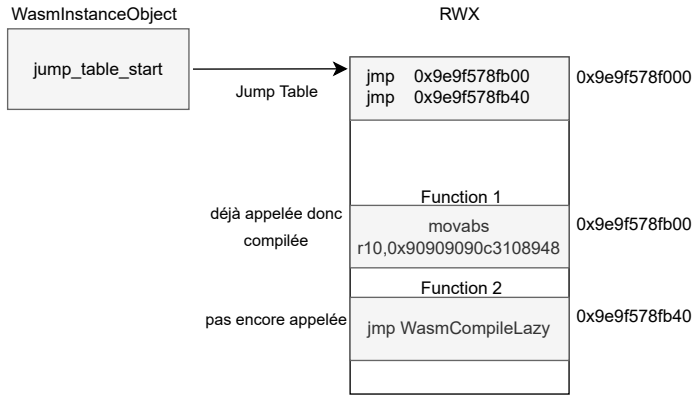

Fig. 13. Représentation de la *Jump Table*

Ensuite, pour copier un *shellcode* sur la page RWX, puis y rediriger le flot d'exécution, l'attaquant doit procéder à plusieurs étapes complexes. La stratégie est la suivante :

— Création d'une primitive d'écriture arbitraire en dehors de la sandbox sous forme de gadget ;
— Utilisation du gadget pour corrompre tous les appels de fonction *Wasm* en les convertissant en une deuxième primitive d'écriture arbitraire ;
— Copie d'un *shellcode* sur la page RWX puis redirection du flot d'exécution dessus.

[16] `https://v8.dev/docs/wasm-compilation-pipeline`

Pour créer la primitive d'écriture arbitraire sous forme de gadget, l'attaquant place un Float dans une fonction *Wasm* pour former, une fois compilée, les opcodes suivants :

```
mov    QWORD PTR [rax],rdx
ret
```

Cependant, avec la présence de l'étape de *lazy compilation* décrite précédemment, l'attaquant ne peut pas directement rediriger le flot d'exécution vers ce gadget, car il n'est présent en mémoire que si la fonction est appelée. Ainsi, la solution consiste à utiliser deux fonctions *Wasm* :

— `Function 1` : compilée lors d'un appel, contient dans son code le gadget (sous forme de Float) ;
— `Function 2` : sert à rediriger le flot d'exécution vers le gadget.

Ainsi, l'attaquant modifie le pointeur `jump_table_start` pour le diriger vers le gadget. Lors de l'appel à la deuxième fonction, le gadget sera exécuté à la place de la *Jump Table* comme cela est illustré sur la figure 14. L'écriture arbitraire ne peut cependant être utilisée qu'une fois. La prochaine étape consiste donc à obtenir une écriture arbitraire persistante.

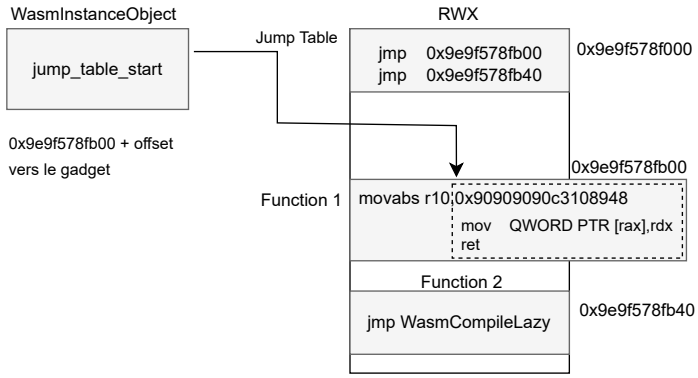

Fig. 14. Appel de `Function 2` redirigé vers le gadget dans `Function 1`

Ainsi, le prochain objectif de l'attaquant est de réécrire par-dessus le code de `Function 2` afin d'obtenir une écriture arbitraire capable d'être appelée à tout moment comme le serait une fonction *Wasm*. La primitive d'écriture arbitraire (le gadget) est appelée avec comme argument l'adresse de `Function 2` et comme contenu l'opcode à placer par-dessus le code de la fonction. Le code Javascript correspondant est le suivant :

```
let initial_rwx_write_at = wasm_rwx + wasm_function2_offset;
  function2(initial_rwx_write_at, 0xc310894890909090n);
```

L'opcode est identique à celui utilisé dans le gadget, il s'agit d'un `mov` plaçant le contenu du registre `RDX` dans l'adresse mémoire contenue par le registre `RAX`. L'attaquant transforme ainsi `Function 2` en une nouvelle primitive qu'il peut appeler comme il le souhaite. Le résultat obtenu est représenté dans la figure 15.

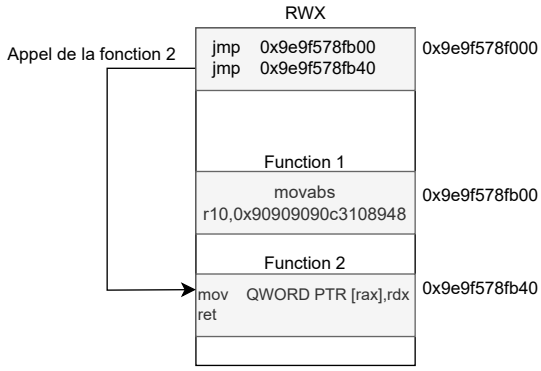

Fig. 15. Réécriture de la Fonction 2

Équipé d'une écriture arbitraire persistante, l'attaquant est en mesure de copier un *shellcode* dans la page RWX. Cependant, comme mentionné précédemment, le pointeur `jump_table_start` n'est utilisable qu'une seule fois. Pour rediriger le flot d'exécution vers le *shellcode*, l'attaquant va donc instancier un nouveau code Wasm pour obtenir un pointeur `jump_table_start` vierge puis l'écraser pour rediriger le flot d'exécution vers le *shellcode*. L'agencement de la mémoire RWX obtenue est représenté en figure 16.

Pour neutraliser cette attaque, la structure `WasmInstanceObject` a été déplacée dans le *Trusted Space* en janvier 2024 [18], permettant ainsi d'empêcher la manipulation du pointeur `jump_table_start`.

6.4 Vue d'ensemble des mécanismes de la sandbox V8

Afin de répondre aux menaces persistantes ciblant V8, la sandbox V8 n'a donc cessé d'évoluer et de s'adapter aux nouvelles techniques d'évasion élaborées par les attaquants. La structure actuelle de la sandbox est illustrée dans la figure 17.

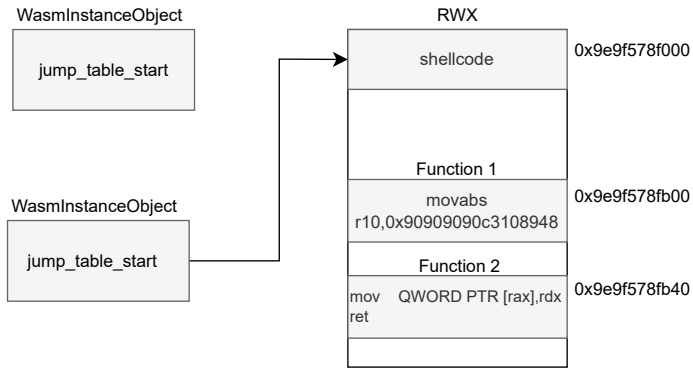

Fig. 16. Redirection vers le *shellcode*

Ainsi, à ce jour, l'architecture est composée des éléments suivants :

— **V8 Pointer Compression Cage :** il s'agit de l'espace désigné par le terme *Main Cage*. Cette zone mémoire du tas est utilisée par V8 pour stocker ses objets classiques. Cependant, ce n'est pas un espace de confiance, car les objets qui y résident sont considérés comme potentiellement corruptibles par un attaquant. Pour référencer des objets situés en dehors de cet espace, des types de pointeurs spécifiques sont utilisés ;

— **TrustedSpace :** les objets situés dans la *Memory Cage Trusted Space* contiennent des données ou du code sensibles. Ce sont des objets V8 classiques, mais avec un type d'instance spécial qui leur permet d'être alloués dans ce tas de confiance, distinct des autres objets V8 susceptibles d'être corrompus. Cette configuration garantit qu'ils n'ont pas été manipulés par un attaquant. Bien qu'ils puissent être lus en toute sécurité, une attention particulière est nécessaire de la part des développeurs lors de leurs manipulations pour éviter toute corruption de mémoire dans cet espace de confiance ;

— **`ArrayBuffer` Partition :** cet espace est dédié au stockage du tampon `BackingStore` des objets `ArrayBuffer`. L'objet `ArrayBuffer` est situé dans la sandbox et pour référencer son `BackingStore` un pointeur de type *Sandboxed Pointer* est utilisé ;

— **Wasm memory cages :** semblable à la `ArrayBuffer Partition`, cet espace est dédié au WebAssembly ;

— **Guarded region :** afin de confiner les tableaux de type `ArrayBuffer` et `TypedArray` dans la sandbox, deux régions dont

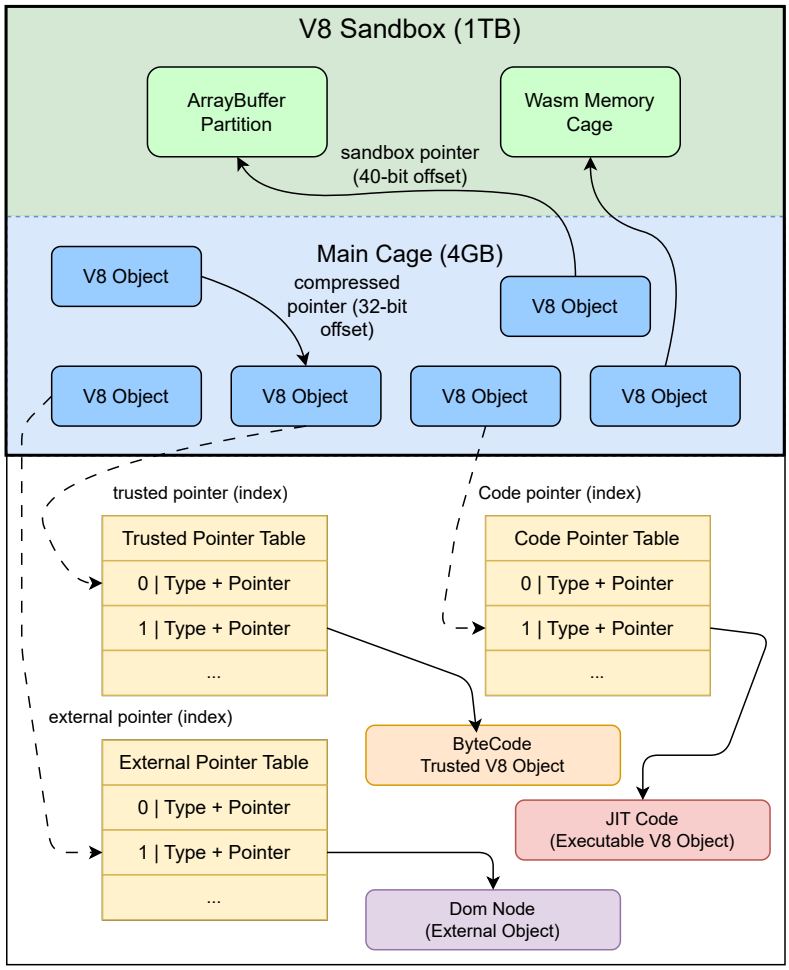

Fig. 17. Vue d'ensemble des mécanismes de la sandbox V8

les pages mémoires sont gardées (via l'option `PROT_NONE` pour `mmap` par exemple) sont ajoutées aux abords de la sandbox [46].

La sandbox V8 utilise différents types de pointeurs pour référencer des objets en mémoire, chacun ayant des implications spécifiques en termes de sécurité et de gestion des accès [49].

1. **Compressed Pointer** ou compression de pointeur, offset sur 32 bits, utilisé pour les objets dans une *Memory Cage*, c'est un offset par rapport à l'adresse de base de la *Memory Cage* ;
2. **Sandboxed Pointer :** Offset de 40 bits à partir du début de la sandbox. Permet de référencer les objets situés dans la sandbox,

mais à l'extérieur de la *V8 Pointer Compression Cage*. Ces objets particuliers comme le `BackingStore` d'un `ArrayBuffer` nécessitent des mesures de sécurité supplémentaires, raison pour laquelle ils sont logés dans une *Memory Cage* distincte ;

3. **External Pointer :** les objets externes à la sandbox sont référencés sous forme d'offset dans une table de pointeurs appelée *External Pointer Table* (EPT). Cette table de pointeurs est située en dehors de la sandbox ;
4. **Trusted Pointer :** offset par rapport à la table *Trusted Pointer Table* (TPT). Utilisé pour référencer les objets placés dans la *Memory Cage Trusted Space*. Cette table de pointeurs est également située en dehors de la sandbox ;
5. **Code Pointer :** offset par rapport à la table *Code Pointer Table* (CPT). Utilisé pour sécuriser le point d'entrée du code d'une fonction ;
6. **Uncompressed/Full Pointer/Pointeur non sandboxé :** il s'agit d'un *raw pointer* sur 64 bits. Bien que les *raw pointers* ne devraient plus être utilisés, la sandbox V8 est encore en cours de développement, ce qui signifie que tous les pointeurs n'ont pas encore été sécurisés, au moment de la rédaction de cet article.

6.5 Futures Améliorations

Les enseignements tirés des tentatives d'évasion de la sandbox V8 ouvrent la voie à des améliorations futures visant à renforcer encore davantage la sécurité de cet environnement d'exécution. La stratégie actuelle de la sandbox consistant à apporter un correctif en réaction à une attaque réussie montre ces limites. En effet, dans un tel modèle de développement, il n'est malheureusement pas impossible de voir émerger de nouvelles techniques d'évasions. Les développeurs impliqués dans le projet semblent en avoir pris conscience et proposent désormais de nouvelles approches basées notamment sur l'utilisation de primitives de sécurité matérielle [40]. Ces considérations sont abordées ci-dessous.

Protections matérielles

1. **Pointer Authentication :** La technologie *Pointer Authentication Code* (PAC), implémentée notamment sur l'architecture ARM, ajoute des informations d'authentification en reliant chaque pointeur à une clé d'authentification. Lors de l'utilisation du pointeur,

la signature est vérifiée par le CPU, garantissant ainsi l'intégrité du pointeur. PAC prévient les attaques visant à corrompre les pointeurs dans la mémoire.

2. **Protection de la pile :**
   — `Shadow Stack` : est une technique qui maintient une pile parallèle pour stocker les adresses de retour des fonctions. Cela permet de détecter les modifications indésirables de la pile d'exécution. En pratique, ce type d'approche est implémenté sur les processeurs Intel (*Control-flow Enforcement Technology* CET [51]) et ARM (*Guarded Control Stack* GCS [2]) ;
   — `PAC RET` : vise à sécuriser les adresses de retour dans la pile en ajoutant des signatures aux adresses de retour. Cette mesure empêche les attaquants de manipuler les retours d'appels de manière malveillante. ARM utilise donc PAC pour signer les adresses de retour.
3. **Landing Pads :**
   Le *Landing Pads* est une technique de *Control-Flow Integrity* (CFI) du type *Forward-Edge CFI*. Comme mentionné section 6.3, le CFI est une mesure de sécurité qui vise à prévenir les attaques basées sur la corruption du flot de contrôle d'un programme. En imposant des restrictions strictes sur les transferts de contrôle, le CFI peut contrer efficacement les tentatives de déviation du flot d'exécution vers des zones indésirables de la mémoire.
   Une implémentation possible consiste à ajouter des instructions assembleur utilisées comme marqueurs, permettant de valider les sauts de code vers des destinations spécifiques. Grâce à ces instructions, le code ne peut effectuer des sauts qu'à des emplacements prédéfinis, connus sous le nom de *Landing Pads*. Des implémentations telles que l'*Intel Indirect Branch Tracking* (IBT) du *Control-Flow Enforcement Technology* (CET) ou l'ARM *Branch Target Identification* (BTI) utilisent cette approche et pourrait améliorer significativement la résilience de la sandbox face aux détournements du flot d'exécution.
4. **JIT Memory Integrity :**
   La préservation de l'intégrité de la mémoire JIT est essentielle pour contrer les attaques visant à corrompre le code généré à la volée. Les solutions telles que l'Intel *Memory Protection Key* (MPK) [34] ou les extensions ARM *Permission Overlay* [1] pourraient permettre le renforcement de la protection de la mémoire utilisée par le moteur d'exécution V8.

L'adoption de ces protections matérielles pourrait offrir une défense multi-couche contre une variété d'attaques avancées, contribuant ainsi à la robustesse de la sandbox V8 face aux menaces émergentes.

Capture The Flag (CTF). Anticiper quels objets pourraient être exploités à des fins malveillantes s'avère être une tâche difficile. Afin de stimuler l'innovation et la recherche en matière de sécurité, Google a lancé un *Capture The Flag* (CTF) pour v8[17] dont l'objectif est de découvrir des techniques d'évasion de la sandbox. Une classe d'objet spécifique a même été ajoutée au code de la sandbox pour permettre aux chercheurs de simuler les primitives *addrOf* et *fakeObj* sans recourir à une vulnérabilité réelle [43]. Ces évolutions témoignent de l'engagement continu de Google à anticiper et à contrer les attaques futures contre la sandbox V8.

7 Conclusion

V8, moteur d'exécution Javascript au cœur des navigateurs modernes, demeure toujours une cible privilégiée pour les attaquants cherchant à obtenir un point d'entrée sur le système d'une victime. Cet article a examiné en détail les techniques d'exploitation couramment utilisées ainsi que les mesures de protection mises en place pour contrer ces attaques.

L'évolution constante des attaques contre V8 a mis en lumière la nécessité de nouvelles mesures de protection. L'analyse des exploits a révélé la sophistication des techniques employées, mettant en évidence les défis auxquels est confrontée la sécurité du moteur.

Ainsi, la mise en place de la sandbox V8 a permis de neutraliser les stratégies d'exploitation habituellement utilisées par les attaquants en isolant les capacités d'écriture et de lecture au sein de la sandbox. Les mécanismes internes de la sandbox V8 ont été analysés en profondeur. La stratégie actuelle repose sur l'isolation des objets Javascript, mais les attaques réussies soulignent la nécessité d'anticiper de nouvelles classes d'objets pouvant être exploitées.

Le lancement d'un *Capture The Flag* (CTF) dédié à v8 par Google témoigne de l'engagement à encourager la recherche en sécurité. Il témoigne également d'un certain niveau de maturité en matière de sécurité.

Enfin pour renforcer la protection, des améliorations futures sont envisagées, notamment la mise en place d'une *Control-Flow Integrity* avec l'introduction de protections matérielles, telles que *Landing Pads*, *Pointer*

[17] `https://github.com/google/security-research/tree/master/v8ctf`

Authentication, *Shadow Stack*, *PAC RET*, et la préservation de l'intégrité de la mémoire JIT, offrant une défense multi-couche contre les attaques sophistiquées. Ces solutions exploitent des fonctionnalités spécifiques des architectures matérielles et pourraient renforcer la sécurité globale de la sandbox.

En conclusion, la lutte constante entre les attaquants et les défenseurs souligne l'importance d'une approche holistique pour garantir la sécurité des environnements d'exécution Javascript. Les efforts conjoints des chercheurs en sécurité, des développeurs et des concepteurs d'architectures matérielles sont essentiels pour maintenir la robustesse des systèmes et protéger les utilisateurs finaux contre les menaces émergentes.

Je souhaite exprimer ma gratitude aux relecteurs pour leur temps et leur expertise : Mr Guillaume Bouffard, Mr Arnauld Michelizza et Mr Sebastien Varrette. Vos commentaires constructifs ont grandement contribué à l'amélioration de ce travail. Merci également à toute l'équipe du LAM, votre soutien et votre collaboration ont été essentiels à la réalisation de cet article.

Références

1. ARM. Permission indirection and permission overlay extensions. `https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions`.
2. ARM. Shadow stacks for 64-bit Arm systems. `https://lwn.net/Articles/940403/`.
3. Clemens Backes. Liftoff : a new baseline compiler for WebAssembly in V8. `https://v8.dev/blog/liftoff`, 2018.
4. Brendon. Exploiting CVE-2021-21225 and disabling W⊕X. `https://tiszka.com/blog/CVE_2021_21225_exploit.html`, 2021.
5. Mathias Bynens. Elements kinds in V8. `https://v8.dev/blog/elements-kinds`, 2017.
6. Mathias Bynens. Celebrating 10 years of V8. `https://v8.dev/blog/10-years`, 2018.
7. Bruce Chen. TheHole New World - how a small leak will sink a great browser (CVE-2021-38003). `https://starlabs.sg/blog/2022/12-the-hole-new-world-how-a-small-leak-will-sink-a-great-browser-cve-2021-38003/`, 2022.
8. Ieu Eauvidoum. Twenty years of Escaping the Java Sandbox. `http://phrack.org/issues/70/7.html`, 2021.
9. ECMA. ECMA-262 : ECMAScript 2023 language specification. `https://ecma-international.org/publications-and-standards/standards/ecma-262/`, 2023.
10. Jaroslav Sevcik et Benedikt et Meurer Georg Neis. Fast arithmetic for dynamic languages. `https://docs.google.com/presentation/d/1wZVIqJMODGFYggueQySdiA3tUYuHNMcyp_PndgXsO1Y`.

11. Igor Sheludko et Santiago Aboy Solanes. Pointer compression. https://v8.dev/blog/pointer-compression, 2020.
12. Javier Jimenez et Vignesh Rao. Google Chrome V8 CVE-2024-0517 Out-of-Bounds Write Code Execution. https://blog.exodusintel.com/2024/01/19/google-chrome-v8-cve-2024-0517-out-of-bounds-write-code-execution/, 2024.
13. Jeremy Fetiveau. Circumventing Chrome's hardening of typer bugs. https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/, 2019.
14. Jeremy Fetiveau. Introduction to TurboFan. https://doar-e.github.io/blog/2019/01/28/introduction-to-turbofan/, 2019.
15. fscholz. Multiprocess Firefox. https://devdoc.net/web/developer.mozilla.org/en-US/docs/Mozilla/Firefox/Multiprocess_Firefox.html, 2017.
16. Google. Stable Channel Update for Desktop Tuesday, April 20, 2021. https://chromereleases.googleblog.com/2021/04/stable-channel-update-for-desktop_20.html.
17. Google. 0day In the Wild. https://docs.google.com/spreadsheets/d/1lkNJ0uQwbeC1ZTRrxdtuPLCIl7mlUreoKfSIgajnSyY, 2023.
18. Google. Issue 14499 : Move Wasm instance data to the Trusted Space. https://bugs.chromium.org/p/v8/issues/detail?id=14499, 2023.
19. Google. Issue 1473631. https://bugs.chromium.org/p/chromium/issues/detail?id=1473631, 2023.
20. Google. Stable Channel Update for Desktop. https://chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop.html, 2023.
21. Google. Stable Channel Update for Desktop Friday, April 14, 2023. https://chromereleases.googleblog.com/2023/04/stable-channel-update-for-desktop_14.html, 2023.
22. Google. Stable Channel Update for Desktop Monday, June 5, 2023. https://chromereleases.googleblog.com/2023/06/stable-channel-update-for-desktop.html, 2023.
23. Google's Threat Analysis Group. 0-days exploited by commercial surveillance vendor in Egypt. https://blog.google/threat-analysis-group/0-days-exploited-by-commercial-surveillance-vendor-in-egypt/, 2023.
24. Jack Halon. Chrome Browser Exploitation, Part 3 : Analyzing and Exploiting CVE-2018-17463. https://jhalon.github.io/chrome-browser-exploitation-3/, 2023.
25. Haraken. How Blink works. https://docs.google.com/document/d/1aitSOucL0VHZa9Z2vbRJSyAIsAz24kX8LFByQ5xQnUg, 2018.
26. jarin@chromium.org. Issue 8806 : Harden Turbofan's bounds check against typer bugs. https://bugs.chromium.org/p/v8/issues/detail?id=8806, 2019.
27. Jungwon Lim, Yonghwi Jin, Mansour Alharthi, Xiaokuan Zhang, Jinho Jung, Rajat Gupta, Kuilin Li, Daehee Jang, and Taesoo Kim. SOK : On the Analysis of Web Browser Security. *CoRR*, abs/2112.15561, 2021. https://arxiv.org/pdf/2112.15561.pdf.
28. Benedikt Meurer. An Introduction to Speculative Optimization in V8. https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8, 2017.

29. Microsoft. Control flow guard for platform security. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2022.
30. Microsoft. Arbitrary code guard. https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection-reference?view=o365-worldwide#arbitrary-code-guard, 2023.
31. Man Yue Mo. Chrome in-the-wild bug analysis : CVE-2021-37975. https://securitylab.github.com/research/in_the_wild_chrome_cve_2021_37975/, 2021.
32. Mozzila. WebAssembly Concepts. https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.
33. Council on Foreign Relations. Operation aurora. https://www.cfr.org/cyber-operations/operation-aurora.
34. Soyeon Park, Sangho Lee, and Taesoo Kim. Memory Protection Keys : Facts, Key Extension Perspectives, and Discussions. *IEEE Security & Privacy (SP)*, 21(3) :8–15, 2023.
35. Chromium project member. Tracking bug for write-protecting code objects. https://bugs.chromium.org/p/v8/issues/detail?id=6792&q=14917b6531596d33590edb109ec14f6ca9b95536&can=1, 2017.
36. Chromium project member. Stabilize wasm code protection (via mprotect and pku). https://bugs.chromium.org/p/v8/issues/detail?id=11974&q=write%20protected%20code%20pages&can=1, 2021.
37. The Chromium Projects. Multi-process Architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture/.
38. The Chromium Projects. Sandbox. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/design/sandbox.md.
39. rmcilroy et oth. Ignition : V8 Interpreter. https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44, 2016.
40. Stephen Röttger. Control-flow Integrity in V8. https://v8.dev/blog/control-flow-integrity, 2023.
41. saelo aka Samuel Groß. V8 Sandbox - Sandboxed Pointers. https://docs.google.com/document/d/1HSap8-J3HcrZvT7-5NsbYWcjfc0BVoops5TDHZNsnko, Feb 2022.
42. saleo aka Samuel Groß. Exploiting Logic Bugs in JavaScript JIT Engines. http://www.phrack.org/issues/70/9.html, 2021.
43. saleo aka Samuel Groß. Add new Memory Corruption API. https://chromium.googlesource.com/v8/v8/+/4a12cb1022ba335ce087dcfe31b261355524b3bf, 2022.
44. saleo aka Samuel Groß. CVE-2022-1364 : Inconsistent Object Materialization in V8. https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2022/CVE-2022-1364.html, 2022.
45. saleo aka Samuel Groß. Harden Map.prototype.delete and related methods. https://chromium-review.googlesource.com/c/v8/v8/+/3593783, 2022.
46. saleo aka Samuel Groß. V8 Sandbox - Address Space. https://docs.google.com/document/d/1PM4Zqmlt8ac5O8UNQfY7fOsem-6MhbsB-vjFI-9XK6w, Feb 2022.
47. saleo aka Samuel Groß. Leaking the_hole should not be a security issue. https://issues.chromium.org/issues/40064521, 2023.

48. saleo aka Samuel Groß. V8 Sandbox. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8, Dec 2023.
49. saleo aka Samuel Groß. V8 Sandbox - Glossary. https://docs.google.com/document/d/10ZVrH2m_cbsjhZmjnWd4K5jpEHWCLourq2dulwN8elI, 2023.
50. saleo aka Samuel Groß. V8 Sandbox - Trusted Space. https://docs.google.com/document/d/1IrvzL4uX_Zv0k2Iakdp_q_z33bj-qlYF5IesGpXW0fM, Dec 2023.
51. Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proc. of the 8th Intl. Workshop on Hardware and Architectural Support for Security and Privacy (HASP'19)*, New York, NY, USA, 2019. ACM. https://doi.org/10.1145/3337167.3337175.
52. Statcounter. Browser Market Share Worldwide - February 2024. https://gs.statcounter.com/browser-market-share/, 2024.
53. Leszek Swirski. Sparkplug — a non-optimizing JavaScript compiler. https://v8.dev/blog/sparkplug, 2021.
54. Ben L Titzer. TurboFan JIT Design. https://docs.google.com/presentation/d/1sOEF4MlF7LeO7uq-uThJSulJlTh--wgLeaVibsbb3tc/htmlpresent.
55. Victor Gomes Olivier Flückiger Darius Mercadier et Camillo Bruni Toon Verwaest, Leszek Swirski. Maglev - V8's Fastest Optimizing JIT. https://v8.dev/blog/maglev, 2023.
56. Chao Wang. V8 engine JSObject structure analysis and memory optimization ideas. https://medium.com/@bpmxmqd/v8-engine-jsobject-structure-analysis-and-memory-optimization-ideas-be30cfcdcd16, 2019.