Fingerprinting de navigateurs

Erwan Abgrall, Martin Monperrus, Yves Le Traon, Sylvain Gombault,
Mario Heiderich et Alain Ribault
erwan.abgrall@telecom-bretagne.eu
martin.monperrus@univ-lille1.fr
yves.letraon@uni.lu
sylvain.gombault@telecom-bretagne.eu
mario.heiderich@rub.de
alain.ribault@kereval.com

[1] Telecom-Bretagne - University of Luxemburg - KEREVAL
[2] Université de Lille 1 & Inria
[3] University of Luxemburg
[4] Institut Mines-Telecom - Telecom Bretagne - UEB - IRISA-RTS-OCIF
[5] Ruhr University Bochum
[6] KEREVAL

Résumé De nombreuses techniques servent à l'identification d'un navigateur. Généralement le serveur web se base sur le champ *User-Agent (UA)* présent dans les en-têtes des requêtes HTTP. En cas de doute sur son authenticité, il est possible de retrouver cette information à partir de l'observation des spécificités de comportement du navigateur au niveau réseau, HTML, DOM, JavaScript ou CSS. Les usages réels de ces spécificités de comportement se limitent essentiellement à l'évasion de filtres anti-XSS et à l'identification du type de navigateur lors d'attaques par *Drive by download*. Dans ce papier, nous présenterons les techniques actuelles d'identification de navigateur web et leurs limites. Nous vous proposerons ensuite une nouvelle technique de prise d'empreinte axée sur l'identification du moteur HTML et son couplage avec un moteur JavaScript donné. L'étude de ces techniques permet de déjouer les mécanismes de détection mis en place par les *Exploit Kits* dans le cadre de la lutte contre les botnets.

1 Introduction

En sécurité informatique, le *fingerprinting* consiste à identifier un système par une approche boite noire [1] en observant soit des comportements spécifiques (prise d'empreinte passive), soit les réponses spécifiques à un ensemble de stimulis (on parle alors de prise d'empreinte active). Un exemple classique de *fingerprinting* d'un service réseau consiste en l'identification du service situé derrière un port réseau donné. Par

exemple derrière le port 22 peut éventuellement se cacher un service SMTP Postfix version 7, en lieu et place du traditionnel serveur SSH.

L'identification de système d'exploitation [5,8,2] est une forme très répandue de *fingerprinting*, notamment dans le cadre de tests intrusifs ou encore d'une cartographie réseau. En envoyant des paquets forgés avec soin, de petites différences dans l'implémentation de la pile réseau révèlent l'identité de la dite pile et du système d'exploitation sous-jacent.

De la même manière, le *fingerprinting* de navigateur consiste en l'identification de l'implémentation d'un navigateur web et de sa version. Tout comme le *fingerprinting* d'OS, la prise d'empreinte d'un navigateur peut caractériser un navigateur de façon unique (cf *ex :* [3]) ou bien simplement identifier le type de navigateur dont il s'agit (*ex :* Firefox ou Internet Explorer) et sa version majeure (*ex :* IE 8 ou IE 9).

Les spécifications du protocole HTTP précisent qu'un navigateur doit envoyer une chaîne de caractères spécifique dans le champ *User-Agent (UA)* de l'en-tête HTTP, pour s'identifier vis à vis du serveur. En pratique tous les navigateurs renseignent et envoient ce paramètre. Ce paramètre existe pour que le serveur puisse adapter les contenus qu'il envoie au traitement des différents navigateurs.

Un serveur ne peut malheureusement pas se baser sur cette en-tête *User-Agent* pour définir l'empreinte d'un navigateur, car cette valeur est affectée par le navigateur et ne peut être crédible dans la mesure ou un attaquant peut le modifier en le patchant (certains navigateurs offre même la possibilité de le modifier dans leur configuration). Le User-Agent est communément employé par les *Exploit Kits* pour attaquer les clients en intégrant des charges utiles malicieuses en fonction du navigateur identifié au travers de cette chaîne.

Nos expérimentations révèlent qu'il est possible de déterminer la version exacte d'un navigateur sur les 77 analysés de façon précise grâce à sa signature. De plus, en utilisant une technique de classifications (c4.5) décrite par Quinlan [10], seuls 6 tests sont nécessaires pour déterminer la famille exacte d'un navigateur.

Les sources d'informations exploitables pour l'identification d'un navigateur web dans un contexte où l'attaquant pousse du code JavaScript chez la victime depuis au moins un serveur web, sont les suivantes :

- le User-Agent et autres en-têtes,
- le comportement réseau,
- le comportement lors du parsing HTML (construction du DOM),
- le comportement lors de modification du DOM,
- les propriétés JavaScript spécifiques,

– les bogues de parsing HTML de parsing HTML
– les bogues de parsing JavaScript

2 Cas d'usage

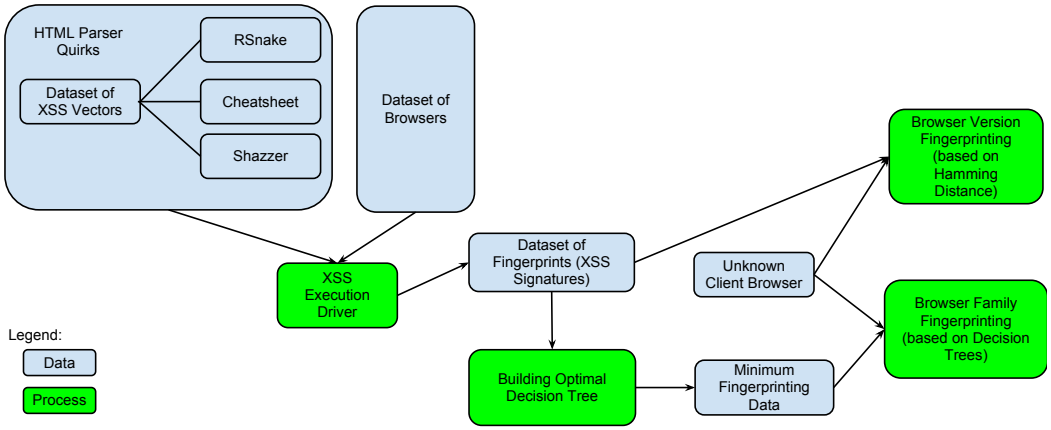

FIGURE 1. Processus de prise d'empreinte

2.1 Se prémunir du vol de session avec l'empreinte du navigateur

Le vol de session se traduit par le vol d'un cookie ou d'un identifiant de session dans le but d'accéder à des ressources non autorisées. L'application serveur est responsable de la détection du vol de session. Cela peut être réalisé à travers la vérification de la présence d'un cookie ou d'un identifiant de session concordant avec l'en-tête User-Agent de HTTP. Cependant, comme dit précédemment, cela ne fonctionne pas si l'attaquant est en mesure de voler à la fois les identifiants et le User-Agent. La vérification des identifiants avec l'adresse IP n'est pas un moyen sûr de se prémunir d'un vol de session en raison de la mobilité des utilisateurs et des mécanismes NAT. Avec l'empreinte du navigateur, à tout moment, l'application serveur peut : 1) Vérifier si l'en-tête HTTP User-Agent concorde bien avec le type de navigateur 2) Vérifier si le navigateur concorde bien avec celui utilisé lors de l'authentification.

2.2 Cibler l'exploitation d'un navigateur en particulier

L'exploitation de vulnérabilités dans les navigateurs est devenu le vecteur le plus utilisé pour propager un malware. La stratégie d'attaque mise en œuvre par les *Exploit Kits* est progressivement passée

d'un bombardement du navigateur à l'aide de multiples exploits à une identification fine du navigateur avant redirection vers un sous-ensemble d'attaques adaptées à la cible. Les techniques employées par les codes JavaScript des *Exploit Kits* pour identifier les navigateurs avant exploitation sont de plus en plus fines et commencent à faire appel à des techniques de *fingerprinting* basés sur des spécificité de chaque famille de navigateur. Ce fut le cas notamment avec l'intégration de plugindetect [7] dans l'Exploit Kit Blackhole. L'étude de ces techniques nous permet donc d'anticiper les évolutions des Exploits Kits et d'envisager d'éventuelles contre-mesures pouvant être implémentées au sein d'un *honeyclients* [8].

2.3 Intérêt d'un *fingerprinting* ciblant l'analyseur HTML et le DOM

Les précédents travaux portant sur le *fingerprinting* de navigateurs portent soit sur le comportement du moteur JavaScript [9,4], soit sur le comportement réseau des navigateurs [11]. Notre technique utilise les spécificités de comportement [9] [10] des analyseurs HTML lors de la construction et de l'evolution du DOM au sein des navigateurs. Ces spécificités peuvent être mises en exergue en employant des portions de code HTML associé à du code JavaScript.

Ces comportements spécifiques à certains navigateurs sont employés pour l'évasion de filtres dans le cadre d'attaques par XSS. Après tout, une attaque par XSS résulte de l'assemblage d'une portion de code HTML avec un appel à une fonction JavaScript contenant la charge utile de l'attaque. Assemblage effectué in-fine au sein du navigateur (DOM-XSS), ou bien au sein du serveur (XSS stocké ou réfléchis). L'avantage de telles spécificités, c'est qu'elles sont observable depuis un serveur grâce au mécanisme de Same Origin Policy.

De plus, Les spécificités des moteurs HTML des navigateurs sont connues et la communauté qui étudie les Cross Sites Scriptings les inventoriés de façon régulière. Ce qui signifie que nous pouvons disposer

7. *Plugindetect* `http://www.pinlady.net/PluginDetect/` est un ensemble de méthodes JavaScript permettant l'identification de la présence ou non d'un plugin donné (flash, java, etc...) ainsi que l'identification du navigateur à partir du User-Agent accessible coté JavaScript.

8. `http://buffer.github.io/thug/`

9. référence de spécificités à l'usage des développeurs : `http://help.dottoro.com/ljrobjtw.php`

10. La page `https://medium.com/the-javascript-collection/ce3645cca083` présente d'autres problèmes auquels font face les développeurs JavaScript lorsqu'il s'agit de rendre un code portable entre navigateurs

d'un très grand nombre de ces spécificités pour mettre en œuvre notre prise d'empreinte.

On peut penser que les spécificités de comportement des navigateurs web que l'on observe ne sont que des bogues mais ce n'est pas si simple que cela. Certaines de ces spécificités sont liées à des implémentations de fonctionnalités non-standardisées, d'autres sont de vraies bogues d'analyse lexicale. Partant de cette constatation, nous traitons ces observations sur un même pied d'égalité, qu'il s'agisse d'une bogue, d'une fonctionnalité non-standardisée, ou bien d'un standard qui n'est pas encore implémenté. Des détails sur l'origine de toutes ces bizzareries sont données dans la section 3 et 3.2. *De plus, nous nommerons par abus de langage l'ensemble de ces spécificité des vecteurs*.

3 Pourquoi tant de vecteurs ?

Les développeurs web le savent (surtout ceux qui ont dû développer des sites compatibles avec IE6), les comportements des navigateurs n'ont eu de cesse de changer ces 10 dernières années. Nous avons donc hérité d'un grand nombre de fonctionnalités diverses. Après la publication des premières normes HTML, et l'apparition de JavaScript dans Netscape a commencé une première guerre entre les navigateurs, avec notamment IE et son moteur JScript pas vraiment compatible avec le JavaScript employé par Netscape.

Après moult efforts de normalisation, les choses se sont stabilisées autour de Html4 puis ont évolué avec les techno XML et l'échec d'XHTML de rendre les pages HTML aisément parsables. Mais c'était sans compter sur la bonne volonté des navigateurs, le plus laxiste rendait les développeurs et donc les internautes heureux car il affichait même des pages HTML incomplètes, malformées, etc... créant un héritage lourd de conséquences. Lorsqu'un navigateur voulait marcher sur les terres d'Internet Explorer, il se devait d'imiter ses défauts pour que les rendus des pages soient les plus fidèles. Ainsi, entre évolution des normes et rétro-compatibilité, les navigateurs sont devenus des cathédrales de code complexes. L'arrivée des feuilles de styles visant à expurger les informations de mise en forme du contenu HTML, et l'ajout d'évènements JavaScript permettant à HTML de rattraper son manque de dynamisme face à Flash, Java et autres scripts VBScript et ActiveX on provoqué l'explosion du nombre de méthodes et techniques permettant de faire appel à du code JavaScript sous diverses conditions. In fine, cette course à l'annihilation de flash et consorts au travers de la norme HTML5 à nécessité

d'accroître les méthode d'appel à JavaScript, augmentant par la même occasion le nombre possible de vecteurs de XSS.

Ajoutons à ce phénomène, les diverses bogues de rendu ou de parsing de certains jeux de caractères, ou la permissivité dont dépend la rétro-compatibilité et vous obtenez une multitude de petites différences dont le fonctionnement ou non peut directement être observé grâce au couplage Moteur HTML / Moteur JavaScript.

L'avènement des navigateurs mobiles et la généralisation des interfaces tactiles n'est que la cerise sur le gâteau des fans de XSS.

3.1 Les sources de vecteurs de XSS

La section suivante présente nos sources de vecteurs de XSS et comment nous avons construit une grande base de spécificités testables pour effectuer notre prise d'empreinte, que cela soient des listings ou des outils de fuzzing.

L'ensemble des sources de vecteurs de XSS et leur contribution en nombre de vecteurs sont indiqués dans la table 1. Notre principale source fut Shazzer avec 291 vecteurs distincts. La liste complète des vecteurs que nous utilisons sont disponibles à l'url suivante `http://xss2.technomancie.net/vectors/`

Table 1. Composition de notre ensemble de XSS (nombre de vecteurs par source)

Rsnake	Html5Sec	Shazzer	Total
69	163	291	523

Antisèche XSS (XSS Cheat Sheet) La XSS Cheat Sheet, qui fut crée par R. 'RSnake" Hansen et al., est riche en ressources pour les développeurs et les pentesteurs. Elle présente une centaine de vecteurs différents. Ce document n'ayant pas été mis à jour depuis des années ; les vecteurs d'attaques basés sur HTML5 et SVG en sont notamment absents. Une version beta d'une version actualisée de la Cheat Sheet fut annoncée en 2010, mais elle ne fut jamais publiée. Le manque de mise à jour de ce document mena à l'apparition de projets communautaires tel que la Cheat Sheet sécurité pour HTML5 (H5SC).

La fameuse XSS Cheat Sheet de rsnake depuis migrée sur le site de l'OWASP : `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`

HTML5 Security Cheat Sheet L'antisèche HTML5 Security (H5SC) est un projet communautaire visant à documenter et catégoriser des vecteurs de XSS connu et divers vecteur d'attaque coté client. La H5SC fournit un stockage JSON et permet aux contributeurs enregistrés d'ajouter de nouveaux vecteurs, de modifier les données existantes et plus important, de mettre à jour les informations sur les versions des navigateurs affectés par chaque vecteur donné. Ce site permet aux professionnels de la sécurité et aux développeurs de se documenter et d'améliorer la protection de leurs application. Voire même d'effectuer de simples analyses de risques. Par exemple lorsqu'un développeur décide de (ou ne peut pas) filtrer telle ou telle sortie. En plus de l'impact sur les navigateurs, chaque vecteur est accompagné d'une explication sur les mécanismes mis en oeuvre pour déclencher cet appel à JavaScript.

Shazzer : Fuzzing collaboratif de navigateurs web Shazzer[11] est un outil de fuzzing collaboratif dont l'objectif est l'identification de vecteurs de XSS et autres bogues dans les analyseurs syntaxiques et les moteurs JavaScript. Son principal intérêt est de fournir un système de *template* de vecteur, des itérateurs et des ensembles de données alimentant ces itérateurs. Il dispose aussi d'un système d'enregistrement des essais réussis. Le *template* est exécuté dans une *iframe*, entraînant le remplacement des différents conteneurs par leur données respectives, le tout est itéré autant de fois que nécessaire jusqu'à ce que la fonction de validation soit appelée. En cas d'appel à ladite fonction, les paramètres du *template* sont enregistrés (doctype, encodage des caractères, etc...), ainsi que le User-Agent employé lors de la session de fuzzing. Shazzer fut employé par un grand nombre de chercheurs et de testeurs en sécurité pour découvrir des bogues connues ou non dans certains navigateurs, ainsi que leur correction.

3.2 Types de Vecteurs de XSS

Nous allons maintenant apporter un éclairage plus spécifique sur les différentes familles de vecteurs et sur l'origine de leur capacité à discriminer une famille d'une autre, ou une version d'un navigateur d'un autre. Cet argumentaire est issue d'expérimentations et d'observations empiriques faites par deux des auteurs. Il s'agit d'une forme de taxonomie des vecteurs de XSS.

11. cf `http://shazzer.co.uk/home`

Vecteurs dépendant de l'éditeur Certains éditeurs embarquent un grand nombre de fonctionnalités qui sont spécifiques à leurs navigateurs (plus particulièrement Opera et Microsoft IE). Cela inclut des expressions CSS spécifiques, le support de VB Script, certains préfixes CSS spécifiques tel que les *-o-link* et d'autres fonctionnalités non-standardisés par le W3C. Certains navigateurs basé sur Gecko équipé d'une JRE et du plugin idoïne supportent une fonctionnalité appelée LiveConnect (`https://developer.mozilla.org/en-US/docs/Java_in_Firefox_Extensions`). Toutes ces fonctions uniques et liées à l'éditeur apportent leur lot de vecteurs et sont une précieuse ressource pour le *fingerprinting*. Par exemple le vecteur #397 selectionné par le classifieur ne fonctionne qu'avec les navigateurs basés sur Gecko (dont les navigateurs Firefox)

Vecteur #397 :

```
<script>({0:#0=eval/#0#/#0#(alert(1))})</script>
```

Vecteurs dépendant de certaines fonctionnalités Certains vecteurs de XSS dépendent de fonctionnalités spécifiques qui ne sont pas forcément liées à un éditeur en particulier. Par exemple, l'exécution de code JavaScript via VML, et les vecteur altérant le DOM qui fonctionnait avec les vieilles versions d'IE. (VML était l'ancêtre de SVG). Le support de cette fonctionnalité à débuté avec la version 5.5 et a pris fin avec la version 8. Les versions suivantes ne peuvent pas effectuer ce rendu VML sans changer de doctype ou sans éléments de configuration additionnels. D'un autre coté, les versions récentes d'IE ne sont pas capable d'afficher correctement les images SVG. Contrairement à IE 9 et 10. Des différences similaires sont constatables avec les autres familles de navigateurs.

Vecteurs dépendant de la version Certains comportements sont liés à certaines versions d'un navigateur. C'est particulièrement vrai avec les vecteurs HTML5 dont certaines fonctionnalités sont partiellement implémentées. Ce support partiel peut être très aisément détecté et permet une discrimination forte entre les versions d'un même navigateur, comme le support de fonctionnalités de *sandboxing* des *iframe* et l'implémentation de *srcdoc*. Chrome et les autres navigateurs en ont un support partiel d'implémenté, et ont effectué plusieurs releases mineures avant que *srcdoc* soit complètement implémenté ; permettant ainsi de distinguer certaines versions mineures entre elles.

Vecteurs dépendant de l'analyseur lexical Certains vecteurs sont spécifiques à certains analyseurs lexicaux voir certaines versions de ces analy-

seur. C'est notamment le cas de certains caractères neutres qui peuvent servir de bourrage. Les premières versions de Chrome permettaient, par exemple, l'utilisation de caractères non imprimables ASCII comme bourrage pour les handlers de protocole des urls. Des phénomènes similaires peuvent être observés lors de tests de tolérance aux espaces et retours chariot. De nombreux navigateurs interprètent des caractères exotiques comme de simples espaces. C'est notamment le cas avec l'espace d'Ogham, parfois affiché sous forme de petits carrés dans vos navigateur. Il s'agit d'une marque blanche pour les idéogrammes. Et pourtant ces caractères ne sont pas de simples espaces et ne devraient pas être interprété comme tels par les analyseurs lexicaux. Les vecteurs 89, 90, 128 et 258 appartiennent à cette catégorie.

Vecteur 89 :

```
--><!-- --\x00> <img src=xxx:x onerror=%(eval_payload)s> -->
```

Vecteur 90 :

```
--><!-- --\x21> <img src=xxx:x onerror=%(eval_payload)s> -->
```

Vecteur 128 :

```
<script src="data:\xCB\x8F,%(eval_payload)s"></script>
```

Vecteur 258 :

```
"‘’><script>\xEF\xBF\xBE%(eval_payload)s</script>
```

Phénomènes de mutation d'éléments Des différences de comportement sont observables entre les navigateurs lorsqu'on interagit avec le DOM et certaines de ses propriétés. Notamment lors d'accès en lecture/écriture sur ces élèments, tel qu' *innerHTML* ou *cssText*. En fonction du contexte et de la version du navigateur, certaines séquences de caractères sont échappées ou non, sont ignorées, décodées ou encodées. Les caractères ASCII non-imprimables peuvent être soit retirés, soit mutés. Tout ceci fournit d'autres caractéristiques observables au travers d'un vecteur et donc exploitable dans notre stratégie de *fingerprint*.

Vecteurs issus d'une bogue Certains vecteurs peuvent se baser sur une bogue pour le fingerprint, et ainsi il est envisageable d'identifier un sous-ensemble de versions d'un navigateur grâce à cette dernière. C'est notamment le cas pour une fuite d'adresses mémoire dans IE `https://www.w3.org/Bugs/Public/show_bug.cgi?id=16757#c10` ayant permis la mise au point du vecteur suivant :

```
<script>
alert(document.createElement("td").cellIndex)
</script>
```

Ce vecteur renvoie -1 quand il s'agit d'un moteur Gecko, 0 pour Chrome, undefined pour Opera, et une adresse mémoire quand il s'agit d'IE. Lorsque cette bogue sera patchée, le test permettra de distinguer les versions postérieures à cette bogue.

Résumons un peu Ce ne sont pas les comportements spécifiques qui manquent lorsqu'un même élément HTML est analysé par un navigateur. Notre approche repose sur ce seul point clef : la mise en exergue de ces différences au travers d'un framework de test sous la forme de vecteurs de XSS.

4 XSS Test Driver

Ce chapitre décrit le fonctionnement de notre framework de test automatisant l'exécution de nos vecteurs de XSS sur les navigateurs, permettant la collecte de nos empreintes de références.

4.1 Description d'un vecteur élémentaire

Une attaque XSS a pour but d'exécuter du code (majoritairement JavaScript) au sein d'un navigateur web par l'injection d'un contenu dans un élément d'une application web qui est retourné à l'utilisateur. Le résultat obtenu dans le navigateur permettant cette éxécution de code constitue un vecteur de XSS. Par exemple le vecteur de XSS le plus connu est le suivant : `<script>alert('xss');</script>` Un vecteur de XSS se décompose de la façon suivante :

1. le contexte : la norme employée influe sur l'exécution d'un vecteur [12], de même que certaines librairies JavaScripts [13].
2. l'encodage : l'analyse lexicale du HTML travaille à partir d'un encodage, elle peut donc etre impactée par certains caractères spéciaux.
3. la partie HTML : un ensemble de balises et de propriétés permettant le déclenchement de l'appel à JavaScript.

12. communiqué au navigateur via la balise DocType, et le type- mime
13. ex : Sandbox JavaScript

4. la charge utile : le code JavaScript Exécuté, dans ce dernier il peut y avoir des interactions avec le DOM pour permettre l'exécution d'un code Arbitraire [14].
5. le format de la charge utile : les propriétés appelées peuvent nécessiter un format particulier (ex : base64).

Ceci est un exemple très simple de vecteur. Les fonctionnalités grandissantes offertes aux développeurs par les navigateurs web autorisent des vecteurs de XSS d'une plus grande complexité. Toute API ou langage capable d'exécuter ou d'appeler du code javascript peut devenir un nouveau vecteur de XSS. Pour plus d'information sur la richesse des vecteurs de XSS, se référer au paragraphe 3.2 et regarder les sources de vecteurs de XSS décrits dans le 3.1

4.2 Fonctionnement de la prise d'empreinte

La prise d'empreinte du navigateur se fait au travers d'une utilisation détournée de notre framework de test de vecteurs de XSS. La prise d'empreinte se résume à exécuter la même suite de test pour l'ensemble des navigateurs. L'ensemble des résultats de tests d'un navigateur sert de signature pour ce dernier. Lorsque l'on souhaite effectuer une prise d'empreinte d'un navigateur inconnu, on passe par une url différente qui rejoue uniquement les vecteurs exécutés par au moins un navigateur.

Logique de test Pour éviter l'utilisation de bibliothèques JavaScript ou toute interaction non désirée avec le DOM, nous avons utilisé la logique suivante pour enchaîner les tests et récupérer les résultats :

1. chaque attaque XSS est servie par une URL différente, avec une charge utile indépendante de toute bibliothèque JavaScript ;
2. la charge utile d'une attaque XSS contient une routine de validation JavaScript ;
3. le moteur d'exécution est constitué d'une ou plusieurs *iframe* avec un mécanisme de rafraîchissement et de timeout ;
4. quand le mécanisme de validation est déclenché, le test est marqué comme réussi ;
5. quand l'URL `/test/next` est appelée, un message 302 Redirect est envoyé au navigateur pour le rediriger vers le test suivant.

14. évasion de filtres, de sandbox JavaScript, etc...

Format de test et charge utile Les cas de test sont fournis sous forme de tuples python, constitués du vecteur avec le format de la charge utile en paramètre et de sa description :

```
("""<script>%(payload)s</script>""","basic script payload")
```

Une suite de tests est composée d'une simple liste de cas de tests à enchaîner. Elle est traitée par XSS Test Driver qui construit les vecteurs et les envoie au navigateur. La charge utile est générée lorsque le test est appelé. Elle contient une routine de redirection vers une adresse de validation, ainsi qu'une requête `XMLHttpRequest` vers une autre adresse de validation pour les navigateurs qui bloquent les redirections JavaScript. En effet, alors que certains navigateurs bloquent les redirections JavaScript, ils ne bloquent pas les appels `XMLHttpRequest` . Une technique de validation basée sur l'adjonction d'une image dans le DOM à été ajoutée pour contourner les problèmes de sandboxing et de limitation d'accès à certaines fonctions JavaScript dans certains navigateurs. Après tout, charger une image, c'est le moins que puisse faire un navigateur.

Nous utilisons deux types de charges utiles : celle que nous avons déjà décrite et une générique contenant `alert('xss')`. Cette dernière est envoyée pour pouvoir confirmer manuellement l'exécution d'un vecteur dans un *WebContext 'quirk'* encodé en utf-8.

Divers formats de charge utile sont disponibles pour couvrir les besoins de vecteurs spécifiques : quelques attaques XSS nécessitent de présenter le JavaScript dans un fichier spécifique pour tromper certains navigateurs (IE6 dans ce cas) :

```
<LINK REL="stylesheet" HREF="http://ha.ckers.org/xss.css">
```

divers formats sont supportés dont voici une liste non-exhaustive :

- payload : code JavaScript à exécuter
- jscript : adresse pour charger un .js contenant la charge utile
- eval_payload : charge utile XSS fournie sous la forme d'une fonction `eval(String.['fromCharCode'](XX,XX,XX))`
- scriptlet : XSS dans une petite page HTML
- css : XSS dans un css
- htc : XSS dans un fichier .htc
- jpg : code JavaScript dans un `.jpg` servi comme un fichier jpeg
- svg : code JavaScript embarqué dans une image svg (3 variantes existent)
- b64 : code JavaScript encodé en base64 (plusieurs variantes).

Afin de tenir compte des paramètres qui peuvent conditionner le déclenchement d'un vecteur, en plus du code JavaScript, nous exécutons les

vecteurs dans différents conteneurs HTML permettant de tenir compte des déclarations du corp de la page HTML comme le doctype ou le type MIME de la page. Nous nommons ce conteneur *WebContext* dans la suite de l'article.

En sus du *WebContext*, le framework de test gère aussi l'encodage de la page et son encodage déclaré afin de permettre le déclenchement de vecteurs dont l'obfuscation réside dans les spécificité de décodage de chaînes de caractères (bogues sur des caractères Unicode, etc...).

Le *WebContext* et l'encodage sont indépendant du vecteur dans la version actuelle du framework, permettant l'exhaustivité des tests lors de l'exécution des prises d'empreintes initiales, mais ce au détriment de la performance lorsque l'on traite un grand nombre de vecteurs (explosion combinatoire) notamment lors des rendus de résultats.

Lors de l'analyse qui va suivre, nous avons utilisé 523 vecteurs, deux WebContext différents (un doctype non déclaré aka quirk mode, et un doctype HTML5 strict), et un seul encodage (utf-8).

Résultat de test et interprétation A la suite d'une exécution, pour un vecteur donné dans un contexte donné avec un encodage donné et un navigateur donné, nous obtenons le verdict de test suivant :

- SENT : Le vecteur à été transmis au navigateur.
- PASS : L'URL de validation à été appelée.
- N/A : Aucun test n'a été effectué sur ce vecteur avec les paramètre indiqués.

L'état *transitoire* SENT qui est le plus sujet à interprétation. En effet, une fois que l'élément de test à été envoyé au navigateur, nous n'avons plus la main ni le contrôle pour juger de l'échec de l'interprétation du vecteur ou non, ou encore de la capacité du navigateur à joindre le serveur ou non, ni même si le vecteur a eu le temps de s'exécuter. Dans le cadre de campagnes de tests où nous contrôlons le navigateur, ce n'est pas un réel problème, mais lorsqu'il s'agit de navigateurs normaux, cela peut impacter et déformer l'empreinte prise. Il en va de même face à un attaquant qui manipulerait manuellement les réponses du navigateur.

La fiabilisation de la prise d'empreinte dans un contexte difficile ou hostile est un sujet qui dépasse la portée du papier actuel. Ce biais dans la prise d'empreinte peut être partiellement compensé par le nombre de tests effectués. On se retrouve alors face à un choix entre fiabilité de la prise d'empreinte et vitesse d'exécution.

5 Identification minimaliste de la famille d'un navigateur

Afin de déterminer automatiquement les tests à effectuer pour savoir si on est dans telle ou telle famille de navigateur, ou tout autre propriété de façon automatique, nous avons fait appel à l'algorithme de classification C4.5 [10] pour qu'il trie et sélectionne les bons vecteurs et leur enchaînement. Weka [6] est un outil d'expérimentation d'algorithmes de fouille de données implémenté en Java. Après passage des résultats au travers de l'outil, nous obtenons l'arbre de décision suivant :

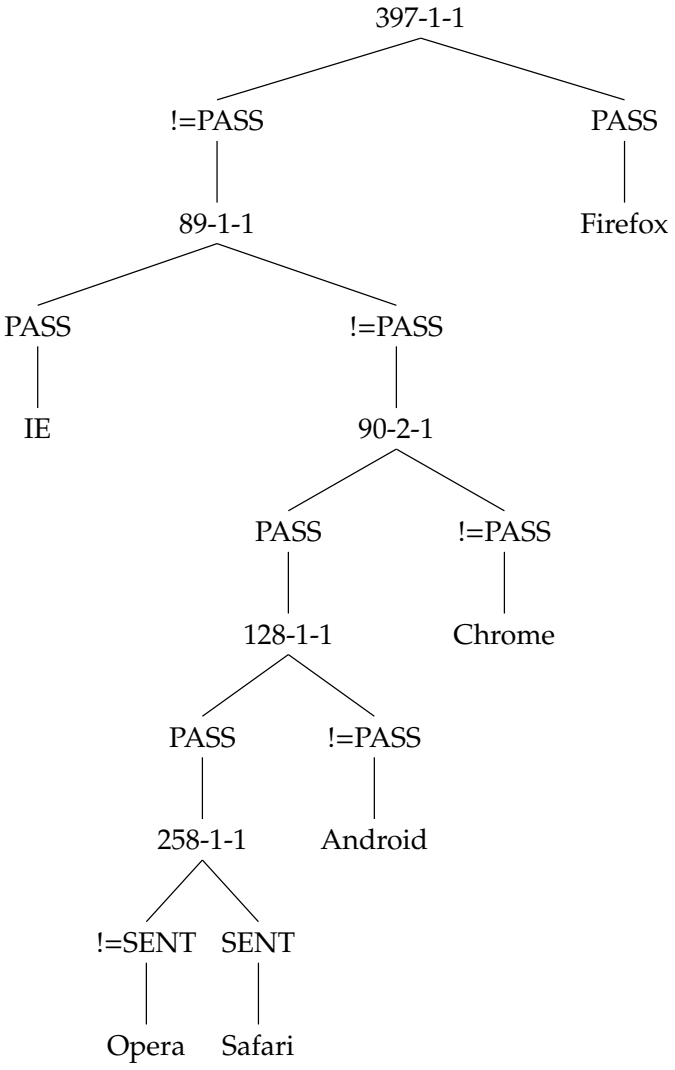

Figure 2. L'exécution de 6 vecteurs de XSS nous permet de déterminer la famille du navigateur avec une précision de 98%

Lors de l'utilisation de ce classifieur, l'outil valide l'arbre en rejouant les données et génère une matrice de confusion 2. De ce fait, il est aisé de contrôler que les tests choisis pour la classification sont les bons et la précision de notre arbre de décision. La diagonale de la matrice contient les navigateurs correctement classifiés. Toute valeur en dehors indique une erreur de classification. La seule erreur que nous avons eue est qu'un navigateur Android à été pris pour un navigateur Chrome avec l'arbre que nous venons de vous présenter.

Table 2. Matrice de confusion

classified as	a	b	c	d	e	f
a = Safari	11	0	0	0	0	0
b = Firefox	0	15	0	0	0	0
c = IE	0	0	6	0	0	0
d = Opera	0	0	0	6	0	0
e = Android	0	0	0	0	14	1
f = Chrome	0	0	0	0	0	19

Les vecteurs 89, 90, 128 et 258 viennent de Shazzer et sont issus de bogues d'analyse syntaxique liés à des caractères spéciaux comme 0x00. Le vecteur 397 est spécifique aux navigateurs basés sur le moteur Gecko (Firefox & Mozilla) et provient de html5sec [15].

6 Identification d'un navigateur par mesure de distance entre les empreintes

L'identification d'un navigateur par mesure de distance se fait de la façon suivante : Pour un ensemble de tests donnés, les résultats de tests de chaque navigateur constituent un vecteur (au sens mathématique du terme) à N dimensions, chaque test constituant une dimension de ce vecteur. [16]

La distance entre deux Navigateurs est calculées en mesurant la similarité entre chaque vecteur représentant les résultats de la prise d'empreinte de chaque navigateur. La fonction de distance que nous avons

15. `http://html5sec.org/#15`

16. Cette vision mathématique n'est pas très utile avec des résultats de tests possédant seulement 3 valeurs, mais pourrais s'avérer plus utile si l'on avait N valeurs possibles pour un test au lieu de 3. Par exemple lorsque l'on récupère la valeur d'un élément du DOM, au lieu d'avoir un booléen, on pourrais vouloir récupérer un nombre d'éléments de tel type ou la valeur d'une propriété (cf : 3.2). Évitant d'avoir à produire plusieurs vecteurs de XSS pour tester de multiples résultats pour une propriété.

choisie est la distance de Hamming. Cette similarité est mesurée en comparant les deux empreintes, et en totalisant le nombre de tests dont les résultats diffèrent.

L'identification d'une empreinte se fait par la recherche de l'empreinte la plus proche dans l'ensemble des empreintes de références (à défaut d'avoir l'empreinte exacte).

6.1 Adaptation de la mesure de distance à la détection de vol de session

Le *MHD* permet l'observation de la dérive d'une signature en cours par rapport à une signature donnée et donc de mettre en oeuvre un seuil de tolérance à cette dérive lorsque l'on souhaite utiliser la vérification de cette signature dans le cadre de la lutte contre le vol de sessions.

En effet, lors de l'utilisation d'une signature partielle que l'on compléterais au fil de la navigation, un grand nombre de résultats de tests se retrouveraient dans un état *N/A.* générant une distance de Hamming très grande avec le navigateur de référence enregistré. Pourtant, si l'on souhaite limiter l'impact sur les performances de tels tests, il pourrait être bon de les effectuer au fil de l'eau au lieu de les enchainer violement. Avoir une mesure de distance capable d'ignorer les comparaisons faites entre un test non-effectué et un résultat de test effectué répond à ce besoin.

Notre distance de Hamming modifiée fonctionne comme la distance de Hamming, sauf qu'elle ignore les cas de tests ou l'un des deux résultats n'est pas disponible (état *N/A*).

L'ensemble de tests de référence est amené à évoluer dans le temps. Et donc on peut se retrouver avec des signatures incomplètes dans le jeu d'empreintes de façon temporaire. De ce fait, une distance de Hamming modifiée permet de ne pas casser le mécanisme d'identification des empreintes. Le rendant ainsi évolutif.

Cette modification apporte cependant un biais dans les mesures de distances lorsque l'empreinte de référence est trop fragmentaire. De ce fait, nous avons introduis une mesure de confiance indiquant le taux de tests correctement exécutés (tests marqués *SENT* ou *PASS*). Nous avons conservé dans notre jeu d'empreintes de référence uniquement des empreintes avec un taux de "réussite" supérieur à 90%.

6.2 Expérimentation

Afin de valider que notre jeu de signatures ne comporte pas trop de colisions possibles (distance nulle alors que les navigateurs sont différents), nous avons calculé le plus proche voisin pour chaque navigateur de notre jeu d'empreintes. Chaque empreinte est ainsi comparée aux autres, ainsi que la distance médiane avec l'ensemble des autres navigateurs de sa famille. Permettant de valider la cohérence des résultats.

Les tables 3 et 4 présentent les résultats de calcul de plus proche voisin utilisant notre distance de Hamming modifiée, ainsi que d'autres métriques facilitant leur analyse.

Le *MHD* correspond à la distance de Hamming modifiée entre le Navigateur et son plus proche voisin dans l'ensemble des empreintes de référence restantes.

Le *MDF* correspond à la distance médiane entre le navigateur et le reste des navigateurs de sa famille. Un MDF faible correspond à un ensemble de navigateurs très proches dans leur comportement.

Le *Fsize* correspond à la taille de la famille du navigateur.

Le *MDD* correspond à la distance médiane entre le navigateur et le l'ensemble des autres navigateurs du jeu de référence. Un *MDD* élevé indique que le navigateur peu être aisément distingué des autres. Un *MDD* faible au contraire indique que peut de tests le distinguent de l'ensemble de référence.

Aucun navigateur dont la *MHD* est nulle ne présente de signature identique avec son plus proche voisin. De plus ces derniers sont proches de navigateurs de la même famille, et d'une version Majeure proche.

7 Etat de l'art du fingerprinting de navigateurs

7.1 p0f : Passive Os Fingerprinting

Principalement développé par Michal Zalewski, p0f est un outil de prise d'empreinte passive basée sur l'analyse de nombreuses propriétés portés par les paquets échangés lors d'une connexion TCP. p0f à fait l'objet d'une évaluation en 2003 par Lippman et al. [7] et de nombreuses évolutions [17]. La version 3 inclue un *fingerprinting* des navigateurs web via l'analyse des headers HTTP [18]. Les signatures sont une combinaison de présence ou d'absence de certains headers, ainsi que l'application de PCRE sur l'entête User-Agent.

17. `http://lcamtuf.coredump.cx/p0f3/`
18. `http://lcamtuf.coredump.cx/p0f3/README`

7.2 Fingerprinting passif utilsant des données *Netflow*

En utilisant du machine learning, Yen et al. ont mit au point un *fingerprinting* passif des navigateurs basé sur leur comportement réseau [11]. Le nombre de connexions TCP lancées simultanément, le nombre de requêtes et leur fréquence, tous ces paramètres dépendent de l'implémentation et du paramétrage du navigateur. Ils fournissent donc une empreinte qui peut être générée automatiquement à partir de réseaux Bayésiens. Le grand avantage de leur technique est de permettre l'identification de la famille d'un navigateurs à partir des éléments *Netflow* issus des routeurs. Ils proposent deux techniques : l'une générique, l'autre spécifique par navigateur avec un écart de précision de 15%. *Le fait d'utiliser une approche active nous permet une identification plus précise de la version du navigateur.*

7.3 Prise d'empreinte en utilisant les capacités de script du navigateur

Fioravanti propose l'utilisation de diverses fonctionnalités et API de javascript pour déterminer la famille du navigateur [4]. Néanmoins l'utilisation de certaines extensions du navigateur (tel User-Agent switcher pour Firefox) ou la réécriture des résultats de tests par des valeurs correctes peuvent altérer les éléments collectés. *La principale différence de notre approche est qu'elle utilise des spécificités de l'analyseur syntaxique HTML, bien plus complexe à falsifier puisqu'elle utilise une base de donnée de bizzarrerie, et que seule la modification de l'analyseur syntaxique ou leur re-implémentation permettrait d'en copier le comportement.*

7.4 Panopticlick : Fingerprinting d'utilisateurs

Dans ce papier, Eckersley et al. ont collecté des bits d'information à partir de diverses propriétés des navigateurs web (User-Agent, résolution, polices de caractère installés, plugins, etc...) permettant de constituer une empreinte de l'utilisateur du navigateur [3]. Les informations sont collectés via Java, Flash et JavaScript. à l'aide de l'ensemble de ces propriétés, la signature obtenue peut s'avérer unique parmi l'ensemble des utilisateurs testés. Cependant, l'identification d'un utilisateur à partir d'options de configuration ne permet pas forcément l'identification stricte du navigateur et de sa version. De plus, le *Panopticlick* utilise Java, Flash et JavaScript, là ou notre technique ne nécessite qu'HTML et JavaScript. Cependant, l'association de notre techniques avec celles présentées dans le *Panopticlick* pourraient mener à un fingerprint encore plus précis.

7.5 Fingerprinting par analyse de performance JavaScript

Mowery et al. utilise des mesures issues de 39 tests de performances pour générer une signature sous la forme d'un vecteur à 39 dimensions [9]. Ils ont ainsi un ratio de détection de la famille du navigateur de 98.2% dans les conditions de l'expérimentation, mais lorsqu'il s'agit d'identifier la version majeure du navigateur, cette précision tombe à 79.8%. La contribution la plus intéressante de ce papier est l'identification de l'architecture matérielle sous-jacente.

8 Conclusion

Au travers de cet article, nous vous avons présenté une nouvelle technique de *fingerprinting* de navigateur web s'appuyant sur un ensemble de vecteurs de XSS. Notre approche donne de très bons résultats dépassant largement ce qui se fait dans ce domaine, et les pistes d'amélioration sont nombreuses. En gardant cette philosophie de prise d'empreinte active, on peut étendre, par exemple, les tests effectués aux bogues disponibles dans les bugtrackers des différents moteurs HTML et JavaScript. Une autre piste de réflexion porte sur le *fingerprinting* de User-Agents ne possédant pas de moteur JavaScript tels que les crawlers et les bots, en se reposant, là encore, sur des spécificités de comportement des analyseurs qui les composent. Notre approche couplée aux autres techniques exposées dans l'état de l'art devrait améliorer la précision de l'identification des navigateurs.

9 Remerciements

Les auteurs remercient KEREVAL, les participants au projet DALI (2008-2012) financé par l'ANR. Des remerciements particuliers à Alexandre Clerici, François Sorin, Maxime Gaudin pour leur aide lors de la traduction de l'article original, Olivier Courtay pour son travail de relecture, Gareth pour avoir créé Shazzer et pour le temps précieux qu'il nous à fait gagner par ce biais, Thibault Henin pour ses conseils sur les mesures de distances.

10 Glossaire

DOM : Document Object Model, représentation interne au navigateur d'une page web. Le DOM est issue de l'analyse lexicale du HTML et évolue au travers de diverses opérations comme l'exécution de scripts JavaScript.

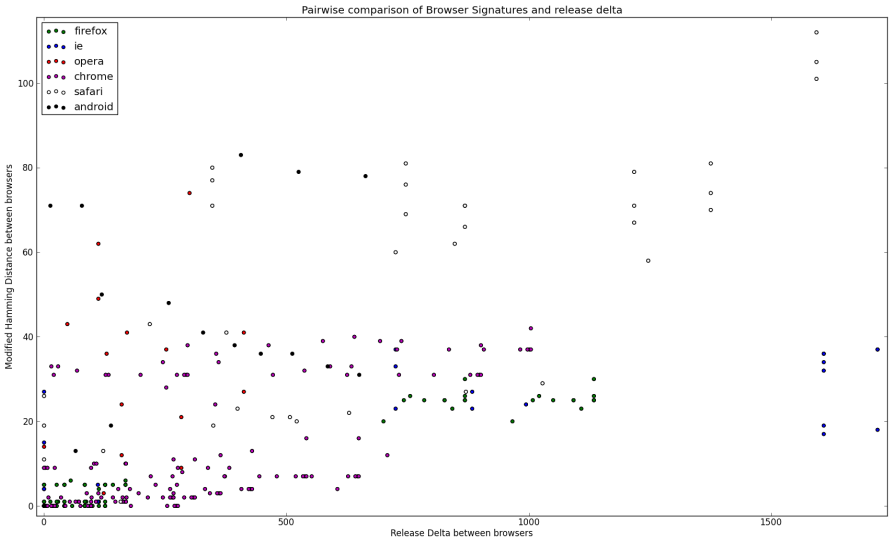

FIGURE 3. Analyse de la relation entre Distance inter-navigateurs et date de Release

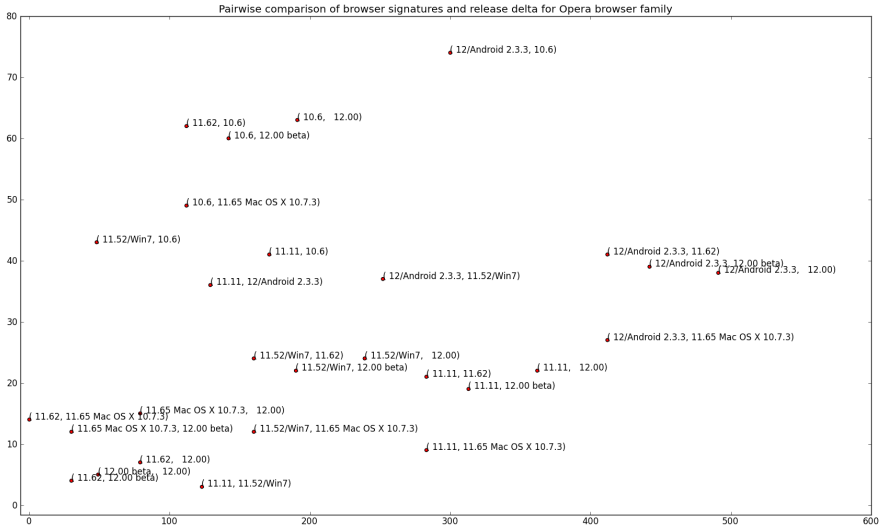

FIGURE 4. Analyse de la relation entre Distance inter-navigateurs et date de Release pour les navigateurs Opera

XSS : Cross Site Scripting, le X est utilisé à la place du C pour éviter la confusion avec les feuilles de style CSS. Il s'agit d'une attaque ayant pour but l'exécution de code coté client (au sein du navigateur). Cette attaque peut être soit stockée sur le serveur (XSS de type 1), soit réfléchie (XSS de type 2), soit lié à une vulnérabilité dans les scripts coté client (DOM-XSS).

SVG : Scalable Vector Graphics, norme du W3C permettant de définir des images vectorielles et des animations au travers d'un document XML. Le SVG permet nottament la gestion d'évenements clavier & souris, et d'attacher des appels à des fonctions JavaScript à ces derniers. SVG à pour but de remplacer Flash au sein des navigateurs.

VML : Vector Markup Language, prédécesseur de SVG dans Internet Explorer et d'autres produits Microsoft sorti en 1998. SVG est issue de la fusion de VML et de PGML (norme concurrente soutenue par Adobe et Sun).

CSS : Cascading Style Sheet : feuilles de style ayant pour but de séparer le contenu du document HTML de sa mise en page afin d'éliminer les propriétés de style des balises HTML.

W3C : World Wide Web Consortium, organisme chargé du développement des standards du web.

IFrame : Inline Frame, déclare un document HTML au sein d'un document HTML. L'inception sauce HTML.

srcdoc : attribut d'une IFrame permettant de spécifier le HTML source à l'intérieur d'une IFrame sous divers formats (dont base64).

Table 3. Evaluation de la distance de Hamming modifiée entre les navigateurs (1ère partie)

Browser	Nearest Neighbor (MHD)	MHD	MDF	Fsize	MDD
89-Origin Browser	28-Safari 5.1.5	3	-	1	129.0
25-fbx v6	8-Safari 5.1.5	7	-	1	127.5
27-Rekonq Linux	40-Safari 5.0.6	15	-	1	131.0
11-Konqueror 4.7.4/KHTML	46-Chrome 3.0.182.2	52	-	1	88.5
5-Firefox 11.0/Win7	39-Firefox 11.0	0	0,5	15	67.5
9-Firefox 10.0/Ubuntu/Linaro	39-Firefox 11.0	0	0,5	15	67.5
16-Mozilla Firefox 11.0 Ubuntu	39-Firefox 11.0	0	0,5	15	67.5
21-Firefox 10	39-Firefox 11.0	0	0,5	15	62.5
39-Firefox 11.0	59-Mozilla Firefox 9.0	0	0,5	15	67.5
51-Mozilla Firefox 8.0	39-Firefox 11.0	0	0,5	15	67.5
59-Mozilla Firefox 9.0	39-Firefox 11.0	0	0,5	15	67.5
60-Mozilla Firefox 10.0	39-Firefox 11.0	0	0,5	15	67.5
4-Firefox 8.0.1	88-Firefox 11.0 linux	0	1	15	68.5
88-Firefox 11.0 linux	4-Firefox 8.0.1	0	1	15	68.5
62-Chrome 12.0.742.91	63-Chrome 13.0.782.99	0	2	19	71.5
63-Chrome 13.0.782.99	62-Chrome 12.0.742.91	0	2	19	71.5
58-Chrome 10.0.648.133	57-Chrome 9.0.597.94	1	3	19	72.5
1-Chrome 18.0	15-Chromium 18.0	0	3,5	19	69.5
15-Chromium 18.0	65-Chrome 16	0	3,5	19	69.5
64-Chrome 14.0.814.0	15-Chromium 18.0	0	3,5	19	69.5
65-Chrome 16	15-Chromium 18.0	0	3,5	19	69.5
70-Chrome 17.0.963.8	15-Chromium 18.0	0	3,5	19	69.5
75-Chrome 18 / Win XP 32	15-Chromium 18.0	0	3,5	19	69.5
66-Chrome 15.0.874.106	15-Chromium 18.0	1	3,5	19	69.5
56-Chrome 8.0.552.215	57-Chrome 9.0.597.94	0	4	19	73.5
57-Chrome 9.0.597.94	56-Chrome 8.0.552.215	0	4	19	73.5
83-Firefox 11.0	4-Firefox 8.0.1	4	5	15	70.0
19-Firefox 7.0	39-Firefox 11.0	5	5	15	70.5
55-Chrome 7.0.517.41	57-Chrome 9.0.597.94	3	7	19	72.5
53-Chrome 6.0.453.1	57-Chrome 9.0.597.94	7	7,5	19	72.0
96-Chrome Nexus S	15-Chromium 18.0	6	8,5	19	69.5
73-Chrome 18.0	15-Chromium 18.0	9	11	19	76.5
68-Opera 11.65	2-Opera 11.11	9	14	6	124.0
107-IE 9	3-IE 9.0	9	17,5	6	69.0
24-IE 7.0	86-IE 7.0	1	21	6	76.0
86-IE 7.0	24-IE 7.0	1	21	6	77.0
2-Opera 11.11	7-Opera 11.52/Win7	3	21	6	136.0
84-IE 7.0	86-IE 7.0	4	22	6	78.0

Table 4. Evaluation de la distance de Hamming modifiée entre les navigateurs (2ème partie)

Browser	nearest neighbor (MHD)	MHD	MDF	Fsize	MDD
7-Opera 11.52/Win7	2-Opera 11.11	3	24	6	134.0
18-Opera 11.62	68-Opera 11.65	14	24	6	133.0
31-Firefox 3.0.17	32-Firefox 3.0.15	0	25	15	79.5
32-Firefox 3.0.15	31-Firefox 3.0.17	0	25	15	79.5
29-Firefox 3.0.6	31-Firefox 3.0.17	2	25	15	81.5
85-IE 8.0	107-IE 9	23	25,5	6	100.0
95-Android 2.3.3	94-ANdroid 2.3.1	13	26	15	160.5
100-Samsung galaxy ace	105-Samsung Galaxy S	13	26,5	15	151.0
104-LG p970	106-Sony Xperia s	11	27	15	142.0
94-ANdroid 2.3.1	95-Android 2.3.3	13	27	15	154.5
106-Sony Xperia s	104-LG p970	11	27,5	15	152.5
101-Samsung galaxy y	100-Samsung Galaxy Ace	13	29	15	154.5
105-Samsung galaxy s	100-Samsung Galaxy Ace	13	30	15	155.0
48-Chrome 4.0.223.11	52-Chrome 5.0.307.1	4	31	19	73.5
52-Chrome 5.0.307.1	48-Chrome 4.0.223.11	4	31	19	75.5
98-Samsung galaxy tab	104-lg p970	15	31	15	157.0
3-IE 9.0	107-IE 9	9	32,5	6	85.0
17-Internet Explorer 9	107-IE 9	15	35	6	80.0
46-Chrome 3.0.182.2	48-Chrome 4.0.223.11	10	37	19	64.5
6-Opera 12/Android 2.3.3	68-Opera 11.65	27	37	6	127.0
79-Android 1.5	80-Android 1.6	19	39	15	144.0
80-Android 1.6	79-Android 1.5	19	41,5	15	147.0
99-HTC Desire hd	100-Samsung Galaxy Ace	40	44,5	15	151.5
82-Android 2.1	95-Android 2.3.3	38	47	15	158.5
37-Opera 10.6	2-Opera 11.11	41	49	6	127.0
92-Safari 3.2.1	91-Safari 3.1.2	1	54	11	149.0
91-Safari 3.1.2	92-Safari 3.2.1	1	56,5	11	148.0
69-Safari 4.0.4	90-Safari 4.0.5	13	60,5	11	148.5
81-Safari 5.0.5	69-Safari 4.0.4	20	64,5	11	152.5
40-Safari 5.0.6	8-Safari 5.1.5	9	65	11	126.0
90-Safari 4.0.5	69-Safari 4.0.4	13	65,5	11	157.0
28-Safari 5.1.5	89-Origin Browser	3	68	11	132.0
8-Safari 5.1.5	25-fbx v6	7	68,5	11	121.5
87-Safari iPhone	40-Safari 5.0.6	25	74	11	138.0
23-Safari 5 Windows 7 64b	8-Safari 5.1.5	19	75	11	119.5
103-Android 3.0	28-Safari 5.1.5	21	78	15	135.0
93-Safari 3.0.4	92-Safari 3.2.1	41	81,5	11	181.0
74-Samsung GT-S5570 Android	11-Konqueror 4.7.4	116	139	15	137.5
97-Google Samsung Nexus	96-Chrome Nexus S	10	151,5	15	69.5

Références

1. R. Adhami and P. Meenen. Fingerprinting for security. *Potentials, IEEE*, 20(3) :33–38, 2001.
2. Patrice Auffret. Sinfp, unification of active and passive operating system fingerprinting. *Journal in computer virology*, 6(3) :197–205, 2010.
3. P. Eckersley. How unique is your web browser ? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.
4. M. Fioravanti. Client fingerprinting via analysis of browser scripting environment. 2010.
5. L.G. Greenwald and T.J. Thomas. Toward undetected operating system fingerprinting. In *Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–10. USENIX Association, 2007.
6. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software : an update. *ACM SIGKDD Explorations Newsletter*, 11(1) :10–18, 2009.
7. R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from tcp/ip packet headers. In *Workshop on Data Mining for Computer Security*, page 40, 2003.
8. Gordon Fyodor Lyon. *Nmap Network Scanning : The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.
9. K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of Web*, volume 2, 2011.
10. J.R. Quinlan. *C4. 5 : programs for machine learning*. Morgan kaufmann, 1993.
11. T.F. Yen, X. Huang, F. Monrose, and M. Reiter. Browser fingerprinting from coarse traffic summaries : Techniques and implications. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 157–175, 2009.