JSITBAD : DÉTECTER DU JAVASCRIPT MALVEILLANT SANS L'EXÉCUTER

Édouard KLEIN – jsitbad@beaver-labs.com
Beaver Labs

C'est théoriquement impossible, et pourtant c'est faisable en pratique. En s'inspirant d'une technique d'apprentissage statistique (Machine Learning) habituellement réservée au traitement du langage naturel, il est possible de déterminer avec une très grande précision si un bout de code en JavaScript est malveillant. Ces résultats s'étendent naturellement à tout langage interprété, mais sont mis en défaut par l'arrivée du WebAssembly.

mots-clés : *MALWARE / MACHINE LEARNING / ANALYSE STATIQUE*

Il y a presque 30 ans, le Web fut créé afin de partager des documents structurés et les lier entre eux **[Web]**. Désormais, la page web médiane approche des 2 Mo **[Size]**, dont près d'un quart est du JavaScript. Le navigateur est devenu une machine virtuelle avec laquelle l'internaute moyen va donc télécharger et exécuter plusieurs dizaines, voire centaines de Mo de JavaScript par jour.

Cette surface d'attaque est monumentale. Un moyen d'analyser statiquement (qui veut exécuter du code malveillant ?) le code qui s'apprête à être exécuté permettrait de la réduire un petit peu.

1. UNE IMPOSSIBILITÉ THÉORIQUE

1.1 L'informatique avant les ordinateurs

En 1936, Alan Turing **[Halting]** démontre qu'on ne peut écrire un programme (l'oracle) qui prendrait en argument n'importe quel autre programme et qui pourrait nous dire si cet autre programme va se terminer un jour. C'est ce qu'on appelle, en science informatique, le *Halting Problem*.

En 1951, Rice **[Rice]** étend ce résultat à toute propriété non triviale du programme donnée en argument. Une propriété non triviale est une propriété qui est vraie pour certains programmes, mais pas pour d'autres. Dans le cas qui nous préoccupe, la propriété non triviale est « Est-ce que ce programme est malveillant ». Ce n'est pas très formel, mais on peut se soucier de propriétés plus objectives comme :

- Est-ce que ce programme va prendre toute ma mémoire vive ?
- Est-ce que ce programme va mettre moins de dix secondes à s'exécuter ?
- Est-ce que ce programme va aller lire tel cookie ?

Il est facile de démontrer le résultat de Rice par l'absurde. Supposons que Rice se soit trompé et que l'on dispose d'un oracle codé en JavaScript sous la forme d'une fonction `is_malicious`, qui nous réponde, quel que soit le code qu'on pourrait lui donner, si ce bout de code est malveillant ou pas.

Il suffit alors de lui donner le code suivant :

```
function lol(js_code){
    if is_malicious(js_code){
        //Don't be malicious
    } else {
        //Be malicious
    }
}
```

On est face à un paradoxe :

- si `is_malicious` répond « vrai », alors le code n'est en pratique pas malveillant ;
- et si `is_malicious` répond « faux », alors le code est malveillant.

Ce paradoxe ne se résout que par l'impossibilité de notre hypothèse de départ : l'existence d'un oracle qui peut déterminer, pour tout programme, s'il est malveillant ou non.

1.2 Des conséquences quotidiennes

Ce résultat d'impossibilité théorique n'est pas juste un exercice intellectuel. Il a des conséquences pratiques chaque fois que quelqu'un écrit du code.

Tout le champ de l'analyse statique des programmes est né du besoin d'obtenir des garanties (souvent des garanties de bon fonctionnement) à partir du code seul, sans avoir à l'exécuter sur toutes les entrées qu'il pourrait rencontrer. Or la recherche de ces garanties se heurte très vite à l'impossibilité démontrée par Rice et Turing.

Cela n'a pas empêché des développements théoriques (par exemple, la théorie des types et les méthodes formelles) et la mise à disposition dans les IDE d'outils pratiques (par exemple, les *types checkers*) permettant d'éliminer certaines classes de bugs.

Il existe également une autre manière d'échapper au théorème de Rice : celui-ci démontre la non-existence d'un oracle pour tous les programmes. Mais il est possible d'analyser un seul programme (ou une classe restreinte de programmes) et d'en prouver des propriétés non triviales. C'est d'ailleurs ce que font de manière informelle tous ceux qui font du reverse engineering.

2. FRÉQUENCE RELATIVE DES TOKENS

Pour évaluer la dangerosité d'un extrait de JavaScript, il nous faut nous aussi contourner le théorème de Rice. Nous nous basons pour cela sur quelques exemples de code malveillant (les exemples positifs) et quelques exemples de code sain (les exemples négatifs).

La méthode que nous proposons calcule certaines propriétés statistiques du code (les *features*) et se base ensuite sur des techniques classiques d'apprentissage statistique (*machine learning*) pour différencier le code sain du code malveillant en se basant sur la valeur des *features* calculées sur les exemples.

2.1 La TF-IDF, une technique classique du traitement du langage naturel

Dans le traitement du langage naturel, un outil courant est le TF-IDF. Un superbe tutoriel à ce sujet est disponible dans la documentation de la libraire scikit-learn **[Scikit]**.

Brièvement, il s'agit d'étudier la fréquence des termes employés dans le document que l'on étudie (TF : *term frequency*) relativement à leur fréquence dans le corpus en général (IDF : *Inverse document frequency*).

Ainsi, les vers suivants :

Les sanglots longs

des violons

de l'automne

ont la fréquence de termes (TF) suivante :

sanglot	25 %
long	25 %
violon	25 %
automne	25 %

Ce qui n'est pas très informatif. En revanche, si l'on pondère ces fréquences en utilisant le nombre d'articles Wikipédia qui contiennent ces mots, l'on obtient le vecteur suivant :

sanglot	2.4
long	0.7
violon	1.5
automne	1.4

Ce vecteur nous informe que le mot le plus saillant est de loin *sanglot*, puis dans une moindre mesure *violon* et *automne*, alors que *long* est commun dans le corpus de référence.

Cette technique est employée avec beaucoup de succès pour par exemple ordonner les résultats d'une recherche par mots clefs, en essayant d'aligner les mots saillants de la requête avec les mots saillants du document.

Typiquement, une recherche « *Recette de bœuf bourguignon* », doit ramener les documents qui contiennent avant tout *bourguignon*, puis *bœuf*, alors que *recette* et à plus forte raison *de* peuvent être ignorés.

Cette technique permet de projeter un morceau de texte, quelle qu'en soit la longueur, dans un espace vectoriel de taille fixe. Cette taille est le nombre de mots du vocabulaire. À chaque mot correspond une composante du vecteur. Le vecteur d'un document donné est constitué du TF-IDF de chacun des mots du vocabulaire : 0 si le mot est absent du document, ou un chiffre de plus en plus grand à mesure que le mot est employé souvent dans le document, relativement à sa présence dans le corpus de référence.

2.2 Inapplicable telle quelle avec du code

Si l'on applique exactement le même processus à du code source, l'on tombe très vite sur un os. Prenons par exemple les deux extraits de code suivants :

```
function greetings(name){
  return "Hello "+name
}
```

```
function i(j){
  return "Hello "+j
}
```

Il s'agit en fait de la même fonction, mais si l'on se base sur les « mots », ici les noms *greetings*, *name*, *i* et *j*, alors les deux vecteurs associés à ces extraits sont extrêmement différents. En effet, le premier extrait aurait pour vecteur quelque chose comme :

greetings	5
name	2
i	0
j	0

Alors que le second aurait quelque chose comme :

greetings	0
name	0
i	0.5
j	0.7

Il est clair que l'on ne peut pas se baser sur les noms des variables et des fonctions pour évaluer le code source. D'une part, cela ferait exploser la taille de notre vocabulaire, et d'autre part ces noms sont choisis pour le lecteur humain du code (que ce soit pour l'aider si l'on veut que le code soit maintenu, ou pour l'induire en erreur si l'on veut au contraire l'obfusquer), alors que notre intérêt se porte sur le comportement de la machine qui exécutera ce code, qui elle se moque des noms choisis.

2.3 L'astuce

Notre astuce consiste à ne pas exploiter le code JavaScript directement, mais à remplacer chaque élément par sa nature lexicale.

L'équivalent pour le langage naturel serait de reconnaître pour chaque mot s'il s'agit d'un déterminant (DET), d'une préposition (PREP), d'un nom (NOM), etc. :

Les	sanglots	longs	des	violons	de	l'	automne
DET	NOM	ADJ	PREP	NOM	PREP	DET	NOM

Cela ne se fait pas en traitement du langage naturel d'une part, car réaliser correctement cette détermination de manière automatique est encore un problème ouvert **[POS]**, et d'autre part, cela élimine toute information sémantique, ne laissant en place qu'une information structurelle extrêmement parcellaire.

En revanche, le code est écrit en suivant une syntaxe formelle qu'il est (comparativement au langage naturel) extrêmement facile de transformer en unités lexicales (cette transformation est d'ailleurs la première étape de la compilation : le lexing, qui va identifier le type de chaque token **[Dragon]**).

NOTE

La surabondance de NOM dans l'extrait de Verlaine devrait permettre à un algorithme d'apprentissage statistique de le reconnaître comme de la poésie. Il serait en revanche impossible d'en déterminer le thème, le champ lexical ou même le sentiment général (ici, nostalgique et déprimant) puisque cette information est détruite lors du passage d'un vocabulaire étendu contenant les mots eux-mêmes à un vocabulaire restreint ne contenant que les différents types de mots utilisés.

Cette transformation est donc parfaitement adaptée à notre problème :

- elle est difficile à calculer sur du langage naturel, mais triviale sur du code ;
- elle détruit l'information sémantique des noms de variables et fonctions, qui est une information n'ayant aucune corrélation avec la dangerosité du code ;
- elle conserve l'information structurelle d'enchaînement des tokens, qui est elle très liée avec le comportement effectif du code lorsqu'on l'exécute.

Le résultat de cette analyse lexicale est le même pour les deux extraits de code :

`function`	`function`	FUNCTION
`greetings`	`i`	ID
`(`	`(`	LPAREN
`name`	`j`	ID
`)`	`)`	RPAREN
`{`	`{`	LBRACE
`return`	`return`	RETURN
`"Hello "`	`"Hello "`	STRING
`+`	`+`	PLUS
`name`	`j`	ID
`}`	`}`	RBRACE

Par exemple, le mot clef `function` est remplacé par FUNCTION, alors que `greetings`, `name`, `i` et `j` sont tous remplacés par ID (pour *identifier*).

Il ne reste ensuite plus qu'à appliquer la TF-IDF à cette suite de noms de token pour obtenir un vecteur de taille raisonnable (il existe une centaine de

tokens en JavaScript), qui décrit la fréquence relative de chaque type de tokens dans un extrait, par rapport au corpus total.

Cette information purement statistique ne tient pas compte de la position relative des tokens : l'on n'accède ni à leur valeur contextuelle (c'est-à-dire le fait par exemple que c'est l'argument de la fonction que l'on utilise dans le *return*, et non un autre ID), ni à la structure de l'arbre syntaxique ainsi créé. Il s'avère qu'en pratique, il a été inutile de pousser l'analyse aussi loin, la simple fréquence relative d'utilisation des tokens suffit.

3. APPRENTISSAGE STATISTIQUE

La transformation de l'objet d'intérêt (ici, un extrait de code JavaScript) en un vecteur de taille fixe est ce que l'on appelle en apprentissage statistique le *feature engineering*. Lorsqu'il est bien fait, il est trivial pour les algorithmes d'apprentissage statistiques d'apprendre comment séparer les objets d'intérêt en deux classes. C'est ici le cas, armé de la transformation décrite ci-dessus, un simple *classifieur* linéaire est en mesure de séparer les exemples malveillants des exemples bénins.

3.1 Un classifieur parfait (du moins sur nos exemples)

Comme le nombre d'exemples malveillants n'était pas excessivement grand (quelques dizaines), nous avons évalué les performances de notre méthode en entraînant toute la chaîne de traitement sur tous les exemples sauf un, puis nous avons comparé la prédiction offerte par le système avec celle déterminée par le reverse engineer.

Ensuite, on recommence en laissant un autre exemple de côté, et ainsi de suite jusqu'à ce que tous les exemples ont servi de test, chacun à son tour.

Nous sommes parvenus ainsi à 100 % de reconnaissance.

Il est évident qu'il est possible d'écrire du code malveillant qui passerait pour bénin aux yeux de notre *classifieur* (ne fut-ce qu'à cause du Théorème de Rice). Par exemple, celui-ci est extrêmement sensible aux attaques adversariales qui modifieraient le code malveillant en ajoutant des parties non fonctionnelles dont le but serait de diluer les statistiques d'usage des tokens pour les ramener vers un profil bénin. Comme l'analyse grâce à TF-IDF est globale, il faudrait alors pour détecter une telle contre-mesure changer la notion de document en divisant chaque extrait en pièces, en espérant tomber sur une fraction malveillante détectable.

Néanmoins, cette approche est la première, à notre connaissance, à réussir à classifier du code source avec une telle précision. Les tentatives antérieures se basaient sur la longueur de l'extrait, le nombre de chaînes de caractères, le niveau d'imbrication maximal, etc. **[Related]**. Nous avons essayé ces approches sur nos exemples, mais les performances n'ont jamais dépassé 80 %.

Enfin, notre méthode à l'avantage d'être extrêmement rapide, l'étape la plus coûteuse, le *lexing*, est de toute façon une étape obligée avant l'exécution du code, elle peut donc être considérée comme gratuite dans un contexte d'évaluation du code à la volée, avant son exécution. Les calculs restants sont proportionnels au nombre de tokens, c'est-à-dire que le temps nécessaire croît linéairement avec la taille du code, ce qui ne gênera pas la performance d'un moteur intégrant cette technique.

NOTE

Les moteurs JS modernes ne parsent pas tout le code, une partie n'est parsée qu'à la demande [v8] au dernier moment. Déterminer la complexité algorithmique exacte est complexe, mais elle ne peut être moins que linéaire : il faut au moins lire tout le fichier une fois !

3.2 Un peu de créativité dans les exemples bénins

Comme tous nos exemples malveillants étaient détectés par notre classifieur, nous avons décidé d'essayer de trouver un *faux positif* : un exemple bénin qui serait reconnu comme malveillant.

Pour cela, nous avons fait appel à divers obfuscateurs et minifieurs, sans succès.

NOTE

Afin de minimiser la taille du code transmis au client, le code JS va souvent être minifié. Tous les espaces sont retirés, les identifiers sont choisis pour être les plus courts possible, etc. Si cela complexifie un tout petit peu le reverse engineering (il suffit de faire appel à un déminifieur), cela n'affecte en rien notre technique qui de toute façon ne fait pas usage des informations supprimées par le minifieur !

Les deux méthodes ayant donné les vecteurs les plus iconoclastes sont :

- la sortie du compilateur ClojureScript. Ce langage qui compile vers le JavaScript opère des transformations magiques comme transformer le code asynchrone en une vaste machine à état synchrone. La position du vecteur associé à ce code était assez éloignée des exemples connus, mais du bon côté de la frontière du classifeur linéaire ;
- ce bout de code magnifique (crédit @cowboy) :

```
!function $(){console.log('!'+$+'()')}()
```

lui aussi justement reconnu comme bénin.

4. QUAND L'APPRENTISSAGE STATISTIQUE RENCONTRE L'ART MODERNE

Notre technique consiste à projeter un extrait de code JavaScript dans un espace à une centaine de dimensions, et à apprendre à glisser un hyperplan (c'est-à-dire un espace d'une dimension de moins que l'espace dans lequel les extraits sont projetés) entre les exemples malveillants et les exemples bénins.

On espère ensuite que tous les exemples malveillants qu'on rencontrera seront du bon côté de l'hyperplan.

On a souhaité visualiser ces résultats :

Un mathématicien et un ingénieur vont assister à une conférence donnée par un ami commun, physicien. Ce dernier explique une théorie physique mettant en jeu un espace à 13 dimensions. L'ingénieur chuchote au mathématicien : « J'arrive à visualiser en 3 dimensions, mais 13... je vois vraiment pas à quoi ça peut ressembler ! ». Le mathématicien répond : « C'est facile, tu visualises en n dimensions, puis tu prends n=13 ».

Heureusement, pour nous, il n'est pas nécessaire de prendre n=100 pour avoir visualisé la position relative d'un extrait de code par rapport à des exemples connus. Il existe des algorithmes d'apprentissage statistique de réduction de dimension, qui permettent de projeter un espace de grande dimension en 2D, en faisant en sorte que la distance en 2D entre les objets considérés ne soit pas trop différente de la distance dans l'espace original.

C'est ainsi que l'on peut obtenir la figure 1. La couleur représente la distance à l'hyperplan qui sépare les exemples malveillants et bénins. On peut la voir comme une feuille qu'on aurait dépliée après avoir fait un origami savant permettant de

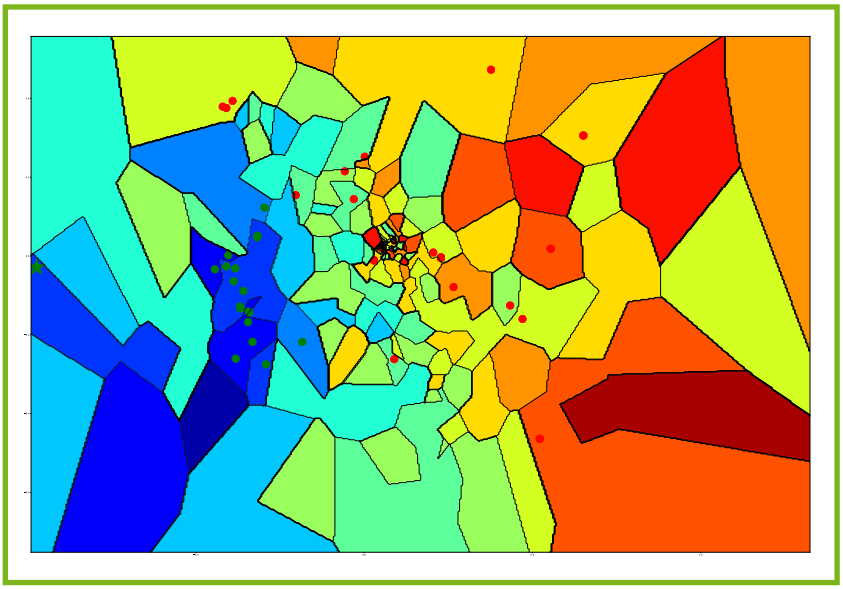

Fig. 1 : Projection en 2 dimensions des exemples bénins (en vert) et malveillants (en rouge) ainsi que du quine de @cowboy (l'étoile verte sur la gauche), qui sont projetés par notre méthode dans un espace à presque 100 dimensions.

lui faire occuper tout l'espace de dimension supérieure (d'où son aspect papier froissé).

Les points verts désignent les exemples bénins, tandis que les points rouges désignent les exemples malveillants (on voit qu'ils sont plus éloignés les uns des autres, il y a plus de variété dans les exploits que dans le code fonctionnel).

L'étoile désigne la position du Quine de @cowboy, justement classifié comme bénin (il ne faisait pas partie du set d'entraînement).

5. PERSPECTIVES

5.1 Contributions

Le code de JsItBad est libre et disponible en ligne. En revanche, les exemples malveillants ne le sont pas.

La création d'une base ouverte de JavaScript malveillant reste épineuse, peut-être cet article lancera-t-il une discussion à ce sujet. En attendant, je peux rajouter les exemples que l'on m'envoie à ma version de la base et publier le modèle ainsi entraîné. Je m'engage bien entendu à respecter les consignes de non-conservation ou de non-divulgation qui viendraient avec les exemples.

5.2 WebAssembly

L'arrivée de WebAssembly n'est que la suite logique de la métamorphose des navigateurs web de lecteurs de documents en plateforme applicative. L'approche proposée ici s'écroule bien évidemment face à du bytecode, puisque toute l'information lexicale est perdue. La plupart des méthodes automatiques destinées à contourner le théorème de Rice s'écroulent également face à un code bas niveau tel que le WebAssembly. S'il est bon pour la performance (de transfert et d'exécution), le WebAssembly est un casse-tête sécuritaire. Il est certain que prévenir l'exploitation d'une faille de navigateur dans un contexte où tous les internautes téléchargent de manière quotidienne plusieurs centaines de Mo de code binaire non audité, est impossible.

CONCLUSION

Nous avons présenté une méthode rapide (O(n)), efficace et novatrice de détection de JavaScript malveillant. Elle souffre probablement d'un manque de généralité faute d'un nombre suffisant d'exemples malveillants, et elle est trivialement contournable par quiconque possède le modèle entraîné. Néanmoins, il s'agit à notre connaissance de la méthode d'apprentissage statistique la plus efficace à ce jour

REMERCIEMENTS

Ces travaux ont été réalisés en 2015, en partie à Sekoia, avec l'aide de Sébastien Larinier et Alexandra Toussaint qui m'ont proposé de travailler sur le problème de la détection de code JS malveillant et de Paul Rascagnères qui a fourni des exemples de code malveillant. ■

RÉFÉRENCES

[Web] Tim Berners-Lee, et al, «World-Wide Web : Information Universe», Electronic : Research, Applications and Policy, April 1992

[Size] **https://www.httparchive.org/reports/page-weight#bytesTotal**

[Halting] Alan Turing, «On Computable Numbers, with an Application to the Entscheidungsproblem», Proceedings of the London Mathematical Society, Series 2, 42 (1936–7), pp 230–265

[Rice] *Rice, H. G. (1953),* «Classes of recursively enumerable sets and their decision problems»

[Scikit] **https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html**

[POS] **https://en.wikipedia.org/wiki/Part-of-speech_tagging**

[Dragon] Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6

[Related] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In 4th International Conference on Malicious and Unwanted Software (MALWARE), 2009, pages 47–54. IEEE

[v8] **https://v8.dev/blog/preparser**