Formalisation et génération d'injections

Eric Alata(eric.alata@laas.fr)*
Didier Le Botlan (didier.le.botlan@laas.fr) *

Abstract: Certains logiciels informatiques, comme les serveurs Web, peuvent se révéler vulnérables aux *injections*, qui permettent de détourner une fonctionnalité initialement anodine en exploitant une trop grande permissivité dans l'interprétation des requêtes. Nous nous intéressons dans cet article à la définition formelle des injections, vues comme des éléments d'un langage formel, décrit par une grammaire. Nous décrivons le problème théorique de la caractérisation des injections d'un langage donné, et proposons un algorithme permettant d'énumérer des injections, connaissant le langage cible.

Keywords: injections, grammaire formelle, générateur de langage

1 Introduction

La prolifération de nouveaux langages, *frameworks* et outils pour Internet permet aux développeurs de concevoir des applications plus rapidement. Cette vitesse de développement, lorsqu'elle n'est pas accompagnée d'une phase de tests rigoureuse, peut entraîner l'apparition de vulnérabilités qui peuvent être exploitables par des *injections*. Les *attaques par injection* désignent des attaques consistant à injecter du code dans des zones de saisie ou des paramètres d'une application dans le but de faire exécuter ce code (par exemple, l'injection SQL vise à injecter du code exécuté par un serveur Mysql). Pour traiter ce problème, de nombreuses approches pour la détection des vulnérabilités ont été implémentées : *a*) l'analyse du code source permet d'identifier des motifs correspondant à la présence de vulnérabilités ; *b*) l'analyse statique emploie des méthodes formelles pour identifier également la présence de vulnérabilités dans le code ; *c*) les outils d'analyse dynamique exécutent le logiciel et observent son fonctionnement afin de détecter des comportements suspects et *d*) les méthodes basées sur le *fuzzing* exécutent le programme avec des données aléatoires en entrée afin d'identifier des comportements indésirables dans l'exécution du programme.

Cette dernière catégorie est bien adaptée pour les applications réparties sur plusieurs machines et qui coopèrent en se basant sur un format d'échange formellement défini. Cet article se focalise sur cette catégorie et propose une nouvelle méthode de *fuzzing* pour l'identification de vulnérabilités basée sur une approche formelle utilisant la théorie des langages.

Le nombre d'outils basés sur le *fuzzing* est important. Pourtant, l'étude de ces différents outils de *fuzzing* montrent qu'il y a encore actuellement des améliorations à leur apporter. En particulier, des études telles que [DCV10, FVM07] montrent que beaucoup de scanners de vulnérabilités dans les applications Web, scanners qui utilisent notamment des

* CNRS, LAAS, 7 Avenue du Colonel Roche, BP 54200, 31031 Toulouse Cedex 4, France
Université de Toulouse, INSA, LAAS, F- 31400 Toulouse Cedex 4, France

techniques d'injection par *fuzzing*, sont encore trop peu efficaces. Nous pensons qu'avec une meilleure compréhension de la vulnérabilité exploitable par injection, ces outils de *fuzzing* pourraient être largement améliorés. En particulier, à notre connaissance, il n'y a pas eu d'étude des injections du point de vue de la théorie des langages, bien que certains outils employent des algorithmes issus de cette théorie. Dans cet article, nous proposons une méthode pour dériver, à partir d'un mot engendré par une grammaire d'un langage, une grammaire des injections. Cette méthode est générique et peut être appliquée à tout logiciel dont le format des données peut être représenté par des grammaires.

La suite de cet article est organisée en quatre sections. Dans la section 2, nous présentons la notion d'injection de manière informelle, en utilisant d'abord un exemple vulgarisé, puis en déclinant un exemple de vulnérabilité issu d'un langage informatique. La section 3 établit un état de l'art concernant les outils de *fuzzing*. Nous présentons en section 4 l'analyse théorique de la notion d'injection, et formalisons divers problèmes liés à la génération systématique des injections. Cette analyse aboutit à un algorithme permettant, pour un langage de référence et une requête donnée, de détecter une éventuelle vulnérabilité dans la requête, et générer alors une famille d'injections correspondantes. La section 5 conclut et présente des perspectives de recherche.

2 Présentation informelle des injections

Dans cette section, nous illustrons le principe des injections par un premier exemple, un jeu de devinette (textes à trou), puis en illustrant plus précisément la notion d'injections dans un langage informatique.

2.1 Présentation intuitive : un texte à trou

Ce jeu nécessite deux joueurs (notés **A** et **B**). Le joueur **A** commence en proposant une phrase contenant un trou (noté —), par exemple : *Je suis — ton balcon, Marie-Christine.* Le joueur **B** doit alors proposer une manière de compléter le trou, de manière à obtenir une phrase complète grammaticalement correcte. À titre d'exemple, le joueur **B** peut répondre correctement avec «*sous*», car il obtient ainsi une phrase complète correcte. Au contraire, la réponse «*saoûl*» ne convient pas, la phrase obtenue n'ayant aucun sens.

Le déroulement du jeu suppose, implicitement, que le joueur B fournira un ou quelques mots qui respecteront la structure initiale de la phrase à trou. La plupart du temps, le joueur **B** se plie à cette règle implicite. Mais, sournoisement, il peut fournir une séquence de mots qui modifie la structure de la phrase, d'une manière que **A** n'avait pas anticipé. Par exemple, le joueur **B** peut proposer de compléter le trou avec «**ravi de pouvoir enfin dormir sur**», qui modifie la structure grammaticale (implicite) initiale. Pire, il peut introduire plusieurs phrases, par exemple avec «**poursuivi par un dangereux gorille ! Aide-moi à grimper jusqu'à ton**», qui introduit une nouvelle phrase sortant totalement du contexte initial.

Notons que la séquence de mots proposée doit toujours donner lieu à une phrase complète grammaticalement correcte, sans quoi la proposition est (automatiquement) rejetée, et pourrait aboutir à l'exclusion du joueur **B** car il ne respecte pas les règles.

Une évolution plus complexe de ce jeu consiste à ne pas exposer la phrase à trou au joueur **B**, mais seulement lui indiquer la nature du ou des mots attendus (ici : préposition de localisation). Le jeu devient plus difficile pour un joueur intègre, et encore plus difficile

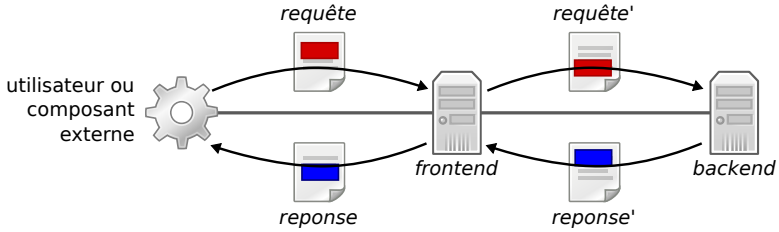

Fig. 1: Architecture informatique de référence

pour un joueur souhaitant tricher en modifiant la structure de la phrase. Dans une telle situation, une des possibilités du joueur **B** est d'énumérer toutes les phrases à trou possibles et, pour chacune, identifier les séquences de mots qui peuvent combler le trou. Plus précisément, il doit identifier l'ensemble des séquences de mots telles que, pour chacune, il existe une phrase à trou qu'elle peut combler correctement et qu'une préposition de localisation peut également combler. Ces séquences sont nommées des injections. Dans la suite, nous nous focalisons sur ce problème.

Cet exemple est simpliste pourtant il est difficile de trouver un algorithme permettant de résoudre ce problème de manière générale. Effectivement, les langages naturels possèdent une grammaire très complexe, contrairement aux langages utilisés en informatique. Toutefois, ce jeu partage beaucoup de points communs avec les systèmes d'information. Dans ces systèmes, plusieurs composants coopèrent et chacun peut construire des requêtes sur la base de bribes de données obtenues depuis d'autres composants. Or, bien souvent, ces données ne sont pas suffisamment contrôlées, permettant alors à un composant malveillant de forger des données de manière à parasiter la requête construite par le composant ciblé.

2.2 Application informatique des injections

En informatique, le langage `SQL` est une référence classique en matière d'injections. Récemment, avec l'évolution des architectures logicielles déployées sur Internet, d'autres langages deviennent des cibles potentielles : `Javascript`, `XML`, `XPath`, etc.

L'étude des injections a fait l'objet de nombreuses recherches. L'une des techniques étudiées consiste à rechercher des vulnérabilités de sites Internet, ou encore à tester les systèmes de détection d'intrusion, en leur soumettant une série d'injections et en observant leur comportement. Dans ces deux cas d'utilisation, les injections sont générées automatiquement par un outil nommé un *fuzzer*. Cette section s'intéresse aux mots qu'ils sont sensés générer.

Il convient de faire un point sur les notations et la définition d'une d'injection. Ces notations et cette définition sont présentées en étant appliquées aux langages informatiques. Pour ce faire, nous nous appuyons sur l'architecture présentée à la figure 1.

Dans cette architecture, le serveur *frontend* obtient les données à analyser depuis un utilisateur externe ou un autre composant. Pour lui permettre de rendre ses services, le serveur *frontend* peut invoquer les services du serveur *backend*. Par exemple, le serveur *frontend* peut être une base de données, un système de fichiers, un serveur `LDAP`, une base `XML`, un service `SOAP`, etc. Les données fournies par l'utilisateur extérieur ou le composant

```
$query = "function(){" .
         "  var name = '" . $name . "';" .
         "  return this.name == name; }";
$result = $users->find(array("$where"=>$query));
```

Fig. 2: Code source `php` pour interroger une base de données avec du `NoSql`

extérieur ont un impact direct sur l'exécution du serveur *frontend* et un impact indirect sur l'exécution du serveur *backend*. En fait, la plupart du temps, ces données sont analysées par le serveur *frontend* dans l'objectif de construire la requête prime à envoyer au serveur *backend*.

Durant la conception d'un logiciel, les développeurs conçoivent un code capable de construire à l'exécution des requêtes destinées au serveur *backend*. Ce code contient entre autre un motif de requêtes qui correspond à une requête avec un trou. Les développeurs choisissent ces motifs de manière à attacher indirectement une même sémantique à toutes les requêtes obtenues après remplissage du trou, quelles que soient les données envoyées par l'utilisateur extérieur ou le composant extérieur. De plus, ces requêtes doivent appartenir au langage des requêtes comprises par le serveur *backend*, sinon elles seront en toute logique rejetées par ce serveur. À titre d'exemple, le langage interprété par le serveur *frontend* pourrait être du `SOAP` au format `XML` et le langage utilisé entre ce serveur et le serveur *backend* pourrait être un langage `NoSql` au format `JSon`, comme indiqué dans la figure 2. Dans cette situation, le développeur s'attend à ce que l'utilisateur externe ou le composant externe fournisse un nom propre, or, il peut tout à fait fournir l'injection suivante, qui réalise un déni de service sur le composant : `test';while(1);a='nom`.

Le code responsable de la construction des requêtes peut être vulnérable. En raison de la présence d'une vulnérabilité, une injection choisie judicieusement par l'utilisateur extérieur ou par le composant extérieur peut changer la structure de la requête envoyée au serveur *backend*. Il peut ainsi changer la sémantique de la requête, tout en conservant le caractère valide de cette requête, qui sera par conséquent dûment effectuée par le serveur *backend*. Au contraire, une injection non judicieuse compromettra la structure de la requête, et sera rejetée par le serveur *backend*.

La connaissance du code exécuté sur le serveur *frontend* permet de guider la construction des injections en se focalisant plus directement sur les injections valides vis-à-vis du serveur *backend*. Il est du ressort d'un outil nommé *fuzzer* de produire des injections valides vis-à-vis du serveur *backend*. La section suivante présente les méthodes et outils disponibles pour la génération des injections.

3 État de l'art sur les *fuzzers*

Le *fuzzing* est une technique qui vise à découvrir des erreurs d'implémentation en analysant le comportement s'un système face à des données corrompues, aléatoires ou invalides de manière à ce qu'elles franchissent les mécanismes de vérification tout en modifiant par la suite le comportement du système. Cette technique peut se focaliser sur le protocole de communication (*fuzzing* de l'interface de l'application) ou sur le format de communication (*fuzzing* du contenu des messages). La cible du *fuzzing* peut être un système distant accessi-

ble par le réseau [JR12, MZR], un compilateur [YCER11], le système d'exploitation sollicité par des appels systèmes [GMICL11, vS05, Jon13], un matériel informatique [GHGT+08], etc. Cette technique peut être employée pour vérifier un code source en complément d'un audit, identifier des vulnérabilités, évaluer l'efficacité des moyens de protection qui accompagnent un système. Noter qu'un *fuzzer* tente de générer une "phrase" complète (un programme complet, un document complet, etc.), en ignorant a priori le contexte précis dans lequel il sera utilisé. Pour la génération des données, les *fuzzers* peuvent se baser sur les algorithmes génétiques [GHGT+08], des memento recensant des exemples d'injections pour un langage donné (*injection sheet cheat*) [MZR] et des *parsers* pour les grammaires. Le *fuzzing* est donc un domaine à cheval entre le test et la sécurité. Ce domaine a été enrichi par de nombreux travaux et outils. Dans la suite, nous nous focalisons sur le *fuzzing* basé sur des langages formels avec une attention particulière sur les approches basées sur des *parsers*.

Dans [MP07], les auteurs ont mené une comparaison entre les approches de *fuzzing* basées sur des mutations et les approches basées sur des générations d'injections. Leur évaluation a porté sur le format des fichiers `PNG` et la capacité des *fuzzer* à générer un format corrompu de manière à franchir les tests de conformité réalisés par l'interpréteur, tout en produisant une erreur dans cet interpréteur. Durant leurs expérimentations, des fichiers `PNG` sains ont été modifiés par des *fuzzer* basés sur des mutations et de nouveaux fichiers `PNG` ont été créés à partir de la spécification du format. En conclusion, les auteurs indiquent que les approches à base de mutation souffrent d'un désavantage lié au manque de diversité dans les fichiers sains.

L'outil `skipfish` [MZR] permet de tester un grand nombre de vulnérabilités sur des sites Internet. Les injections que cet outil permet de cibler couvrent un ensemble varié de langages : `SQL`, `XPath`, `shell script`, etc. L'approche utilisée consiste, pour chaque type de vulnérabilités, à utiliser un ensemble prédéfini d'injections (à l'instar des approches basées sur des *injection cheat sheet*). Cette approche limite grandement la variété des injections testées, au bénéfice de la performance.

Le *fuzzer* `jsfunfuzz` [Rud07] est un outil populaire dédié au test des interpréteurs `javascript` des navigateurs Internet, ayant déjà permis de mettre en évidence de nombreux problèmes. Partant de la grammaire du langage `javascript`, l'outil génère aléatoirement des programmes complets. Ce *fuzzer* a inspiré les travaux menés pour le développement de l'outil `LangFuzz` [HHZ12], également basé sur la grammaire du langage `javascript`, mais qui utilise une technique de génération différente. Quant à l'outil `croff_fuzz` [Zal11], il cible uniquement l'interpéteur `DOM` des navigateurs Internet. Son approche est également basée sur une mutation de la structure initiale du document.

Dans [HHZ12], les auteurs indiquent que, selon eux, il est surprenant que le nombre de *fuzzers* qui génèrent leurs données à partir d'une grammaire soit aussi limité. Cette remarque est d'autant plus pertinente que les approches à base de grammaires semblent particulièrement efficaces [MP07]. Toutefois, dans notre cas, nous nous intéressons aux injections dans les langages. La génération d'une injection à partir d'une grammaire n'est pas aussi simple que la génération d'un mot du langage à partir d'une grammaire. Cette nuance explique peut-être le manque d'approches de construction des injections à partir des grammaires des langages.

Dans la suite de cet article, nous proposons une analyse théorique du problème de la génération automatique d'injections à partir d'une grammaire, ainsi qu'une mise en œuvre

Type 0	Grammaire générale	aucune restriction
Type 1	Grammaire monotone	règles de la forme $\alpha \to \beta$ avec $\lvert\alpha\rvert \leq \lvert\beta\rvert$
	Grammaire contextuelle	règles de la forme $\alpha A \beta \to \alpha \gamma \beta$ avec $A \in N$ et $\lvert\gamma\rvert > 0$
Type 2	Grammaire algébrique	règles de la forme $A \to \beta$ avec $A \in N$
Type 3	Grammaire régulière	règles de la forme $A \to \beta B$ avec $A \in N$, $B \in N$, $\beta \in T^*$

Fig. 3: Classification de Noam Chomsky pour les grammaires

algorithmique.

4 Le problème de génération des injections

4.1 Énoncé du problème général

Une grammaire formelle $G = (T, N, R, S)$ est constituée de quatre éléments : l'ensemble T des symboles terminaux, l'ensemble N des symboles non-terminaux, un ensemble de règles R, et un non-terminal S appelé l'axiome. Les règles sont des règles de réécriture de forme générale $\alpha \to \beta$, avec $\alpha \in (N \cup T)^*$ et $\beta \in (N \cup T)^*$. Un mot est dit dérivable dans la grammaire s'il existe une succession de règles s'appliquant à l'axiome et permettant d'atteindre ledit mot. La suite comprenant, en alternance, les mots intermédiaires et les règles appliquées s'appelle *dérivation*. Par construction, toute dérivation commence par l'axiome S. Le *langage* correspondant à la grammaire est constitué de l'ensemble des mots dérivables composés uniquement de symboles terminaux.

La hiérarchie de Noam Chomsky permet de classer les grammaires en plusieurs types[Cho59], en fonction des restrictions imposées à la forme des règles de production. Le tableau 3 contient une présentation succinte de cette classification, avec les restrictions sur les règles.

La plupart des langages informatiques sont représentés par des langages algébriques [KR88, ISO92]. Plutôt que de construire laborieusement un arbre syntaxique, ou d'utiliser une bibliothèque dédiée (en utilisant par exemple les *prepared request* pour le langage `SQL`), en général, les développeurs se basent directement sur ces syntaxes pour construire les requêtes dans leurs programmes, via des concaténations de chaînes de caractères Par exemple, le code source `php` de la figure 4 permet de consulter le contenu d'une base de données. Dans cet exemple, l'utilisateur extérieur ou le composant extérieur est censé insérer un nom d'utilisateur à la place de la chaîne `$_GET[name]`. Cet emplacement constitue un point (potentiel) d'injection. Du point de vue de la grammaire, un point d'injection est un symbole particulier ayant vocation à être substitué par un mot lors de la construction de la requête. Notre objectif est d'analyser la forme que peuvent prendre les mots à insérer dans ce point d'injection tout en garantissant que la requête finale conservera une syntaxe correcte. Il convient donc de commencer par définir formellement la notion d'injection.

```
$sql = "SELECT Tel FROM users WHERE name = '$_GET[name]'";
$result = mysql_query($sql);
if (mysql_num_rows($result) >= 1) {
  print("Found");
} else {
  print("Not found");
}
```

Fig. 4: Code source `php` pour interroger une base de données avec du `sql`

Injections sur un langage Étant donné un langage $\mathcal{L}$ et un symbole terminal x nommé point d'injection, on définit les *injections de* $\mathcal{L}$ *en* x, noté $I_x(\mathcal{L})$, comme étant le langage définit par

$$\{w \mid \exists a \in T^*, b \in T^* \ s.t. \ axb \in \mathcal{L} \wedge awb \in \mathcal{L}\}$$

Sans perte de généralité, nous supposons que la grammaire initiale du langage est étendue avec un symbole particulier, x, faisant office de point d'injection. Sans rentrer dans les détails, nous supposons que la grammaire garantit au maximum un seul point d'injection par terme. Cette définition indique que, pour le langage $\mathcal{L}$ et le point d'injection x, w constitue une injection s'il existe un mot axb du langage tel que awb est également un mot du langage. Si l'on note $a\backslash m$ l'opération qui retire à m le préfixe a (dite quotient à gauche) et m/b l'opération qui retire à m le suffixe b (dite quotient à droite), la définition ci-dessus est alors équivalente à $I_x(\mathcal{L}) = \{a\backslash m/b \mid axb \in \mathcal{L} \wedge m \in \mathcal{L}\}$. La notation $a\backslash m/b$ représente un quotient bilatéral de m par a et b.

4.2 Résultats sur les langages d'injection

Étant donné un langage $\mathcal{L}$ et un symbole d'injection x, on cherche à caractériser le langage $I_x(\mathcal{L})$. Nous présentons ici deux résultats, sans en fournir les démonstrations détaillées par manque de place.

Les injections d'un langage régulier forment un langage régulier. Lorsque le langage étudié est régulier (de type 3), alors le langage $I_x(\mathcal{L})$ est lui-même régulier. (La démonstration repose sur la notion de quotient, et ne pose pas de difficulté particulière.)

Les injections d'un langage algébrique sont de type 0 dans le cas général. Lorsque le langage étudié est algébrique (de type 2), alors le langage $I_x(\mathcal{L})$ est de type 0 dans le cas général, i.e. il est possible d'exhiber un langage de type 2 dont le langage des injections est de type 0. Notre démonstration constitue en réalité un résultat plus fort :

Tout langage de type 0 peut être décrit comme le langage des injections d'un langage algébrique. En effet, à partir d'un langage quelconque $\mathcal{L}_0$ (de type 0), nous sommes en mesure de construire un langage algébrique correspondant $\mathcal{L}_2$, contenant un point d'injection x, dont le langage des injections $I_x(\mathcal{L}_2)$ est précisément $\mathcal{L}_0$. La technique de construction de $\mathcal{L}_2$ reprend une idée classique utilisée dans de nombreux travaux portant sur la caractérisation des langages [Gef88, LT90].

L'appartenance d'un mot à un langage de type 0 étant indécidable, ainsi que la génération de l'ensemble des mots de $\mathcal{L}_0$ d'une taille finie, il est impossible de construire un *fuzzer* générique capable de générer les injections par ordre croissant de taille, entre autres. Il est également impossible, dans le cas général, de construire un algorithme capable d'indiquer si une grammaire peut faire l'objet d'injections non triviales. Ces résultats théoriques, malheureusement négatifs, constituent une première contribution de cet article.

Toutefois, malgré ces limites, deux problèmes intéressants peuvent tout de même être traitées. Le premier concerne l'identification de l'ensemble des injections associées à une phrase à trou donnée. Le second concerne la construction d'un *fuzzer* permettant de générer des injections valides en se basant uniquement sur la grammaire du langage, bien qu'il s'agisse alors d'un semi-algorithme en raison de l'impossibilité de déterminer quelle est l'injection la plus petite, ou de savoir si l'énumération est terminée (ou même de savoir si l'énumération est vide).

4.3 Identification des injections pour un motif

Rappelons qu'un des objectifs des *fuzzers* est d'identifier la présence de vulnérabilités dans un composant du système. La section précédente a montré qu'il était impossible de répondre à cette question de manière générale. Par contre, si nous disposons du code source du composant, il est alors possible de parcourir ce code à la recherche des routines dédiées à la construction de requêtes à partir des informations fournies par un utilisateur externe ou un composant externe. Cette recherche peut être réalisée sans difficulté avec un analyseur syntaxique du langage utilisé pour le développement du composant (langage `C` ou `php` par exemple). Ces routines sont les motifs discutées précédemment. Pour illustrer notre démarche, nous allons nous focaliser sur la grammaire fictive de la figure 5 qui engendre le langage reconnu entre le serveur *frontend* et le serveur *backend*. Cette grammaire engendre des séquences de messages et de demandes d'exécution. Un message débute par le mot clef **msg** et contient une suite de paramètres représentés sous la forme d'une clef et d'une valeur. Une demande d'exécution débute par le mot-clef **exec** et contient une commande à exécuter. Pour la suite, supposons que notre point d'injection soit représenté par le symbole **value**. Autrement dit, un utilisateur extérieur ou un composant extérieur fournit une valeur au serveur *frontend* et ce dernier construit, par exemple, un message avec cette valeur, à destination du serveur *backend*. Le motif correspondant est :

```
request = "msg key=" + $client_input + "&key=value";
```

Normalement, le client doit fournir la donnée **value** mais il peut fournir l'injection :

```
value;exec command;msg key=value
```

Calculer la grammaire correspondant au résultat du quotient bi-latéral d'une grammaire algébrique par elle-même est indécidable. Par contre, il est possible de calculer la grammaire correspondant au résultat du quotient bi-latéral d'une grammaire algébrique par un mot. Ce cas de figure ramène le problème au calcul de deux quotients (quotient à gauche et quotient à droite) d'une grammaire algébrique par deux mots. Par exemple, pour le motif précédent, le préfixe `msg key=` et le suffixe `&key=value` constituent bien deux mots qui peuvent être utilisés pour des opérations de quotient. Le calcul du quotient gauche d'une grammaire algébrique par un mot peut être réalisé de manière itérative, en partant de la grammaire algébrique et en considérant successivement les différents symboles terminaux du mot. Chaque itération i crée une nouvelle grammaire notée $G_i = (T, N_i, R_i, S_i)$, avec $G_0 = G$, en s'assurant que cette nouvelle grammaire ne génère

S=**S** T={**;**, **msg**, **exec**, **cmd**, **key**, **=**, **value**, **&**} N={**S**, **Msg**, **Exe**, **Params**}

R_1	**S**	$\rightarrow$	**Msg ; S**
R_2	**S**	$\rightarrow$	**Exe ; S**
R_3	**S**	$\rightarrow$	**Msg**
R_4	**S**	$\rightarrow$	**Exe**
R_5	**Exe**	$\rightarrow$	**exec cmd**
R_6	**Msg**	$\rightarrow$	**msg Params**
R_7	**Params**	$\rightarrow$	**key = value**
R_8	**Params**	$\rightarrow$	**key = value & Params**

Fig. 5: Exemple de grammaire algébrique

que des mots commençant par le symbole terminal désiré. Ceci peut être atteint en dupliquant les règles et les symboles non-terminaux, en distinguant les symboles de gauche et les premiers symboles de ces règles et en filtrant ces éléments pour ne conserver que ceux permettant de dériver le symbole terminal désiré en première position. À ce moment, les règles qui engendrent le symbole terminal en première position sont modifiées pour ne plus l'engendrer. Les autres symboles des règles sont inchangés de manière à permettre de poursuivre les dérivations dans le mot en utilisant les autres règles inchangées de la grammaire de départ. L'itération i est réalisée en cinq étapes :

1. création, pour chaque non-terminal $A \in N_{i-1}$ de la grammaire, d'un nouveau non-terminal noté A_i unique ;
2. pour chaque règle de la grammaire, de la forme $A \rightarrow B\alpha$ avec $(A, B) \in N_{i-1}^2$, une nouvelle règle est ajoutée : $A_i \rightarrow B_i\alpha$;
3. pour chaque règle de la grammaire, de la forme $A \rightarrow t\alpha$ avec $A \in N_{i-1}$ et t le symbole en cours d'analyse, une nouvelle règle est ajoutée : $A_i \rightarrow \alpha$;
4. le nouvel axiome est le résultat de la transformation de l'ancien axiome, lors de la première étape ;
5. suppression des $\epsilon - production$, des symboles inutiles, des symboles inaccessibles.

La démarche à suivre pour réaliser le quotient à droite est similaire. Il suffit simplement d'inverser l'ordre des éléments des parties droites des règles pour se ramener à un quotient à gauche et d'appliquer l'algorithme précédent. Ensuite, une réinversion permet d'obtenir la grammaire finale.

L'application de l'algorithme, pour le préfixe `msg key=` et le suffixe `&key=value`, sur la grammaire de la figure 5 aboutit à la grammaire de la figure 6. L'utilisation de cette grammaire peut entraîner, par exemple, la dérivation de la figure 7. Cette grammaire peut être analysée afin d'identifier si des injections sont possibles sur ce motif. Il suffit d'identifier si la grammaire engendre un langage vide. Ce problème est décidable pour les langages algébriques [Lan64]. En l'occurrence, sur notre exemple, le langage engendré n'est pas vide et le motif est donc *vulnérable* si aucun mécanisme de détection ou d'assainissement des données n'est installé.

Ce test peut être généralisé à tous les motifs d'un composant. Concrètement, il suffit de parcourir le code source de ce composant à la recherche de motifs permettant de construire

S=**X'2** T={**key**, **=**, **msg**, **exec**, **cmd**, **value**, **:**, **;**} N={**X'2**, **Exe'12**, **Params'6**, **X'1**, **Exe'11**, **Params'5**, **Exe'8**, **Msg'5**, **Exe'7**, **Params'2**, **Msg**, **Exe'6**, **Exe'3**, **Exe'4**, **Params**, **Params'1**, **Params'0**, **Exe**}

R_1	**X'2**	→	**Exe'12**
R_2	**Exe'12**	→	**Params'6**
R_3	**Params'6**	→	**X'1**
R_4	**X'1**	→	**Exe'11**
R_5	**Exe'11**	→	**Params'5**
R_6	**Params'5**	→	**Exe'8**
R_7	**Exe'8**	→	**Exe'7**
R_8	**Exe'7**	→	**Exe'3 ; Msg'5**
R_9	**Exe'7**	→	**Exe'6**
R_{10}	**Exe'6**	→	**Params'1**
R_{11}	**Params'1**	→	**value : Exe'4**
R_{12}	**Msg'5**	→	**Msg ; Msg'5**
R_{13}	**Msg'5**	→	**Exe ; Msg'5**
R_{14}	**Msg'5**	→	**Params'2**
R_{15}	**Params'2**	→	**msg Exe'4**
R_{16}	**Exe'4**	→	**key = value : Exe'4**
R_{17}	**Params'1**	→	**value**
R_{18}	**Exe'4**	→	**key = value**
R_{19}	**Exe'3**	→	**Params'0**
R_{20}	**Params'0**	→	**value**
R_{21}	**Params'0**	→	**value : Params**
R_{22}	**Exe**	→	**exec cmd**
R_{23}	**Msg**	→	**msg Params**
R_{24}	**Params**	→	**key = value**
R_{25}	**Params**	→	**key = value : Params**

Fig. 6: Exemple de grammaire algébrique – après analyse du préfixe

X'2	⇒	**Exe'12**
	⇒	**Params'6**
	⇒	**X'1**
	⇒	**Exe'11**
	⇒	**Params'5**
	⇒	**Exe'8**
	⇒	**Exe'7**
	⇒	**Exe'3 ; Msg'5**
	⇒	**Params'0 ; Msg'5**
	⇒	**value : Params ; Msg'5**
	⇒	**value : key = value ; Msg'5**
	⇒	**value : key = value ; Exe ; Msg'5**
	⇒	**value : key = value ; exec cmd ; Msg'5**
	⇒	**value : key = value ; exec cmd ; Params'2**
	⇒	**value : key = value ; exec cmd ; msg Exe'4**
	⇒	**value : key = value ; exec cmd ; msg key = value**

Fig. 7: Exemple de dérivation à partir de la grammaire de la figure 6

des requêtes pour un autre composant et d'exécuter cet algorithme avec ce motif et la grammaire associée afin de déterminer si, par construction, le motif est vulnérable. Ce résultat peut également être mis à profit dans le but de tester les mécanismes de détection d'intrusion. Pour ce faire, il suffit de générer des injections qui exploitent réellement des vulnérabilités sur un système témoin. Notre algorithme présente l'avantage de générer un ensemble très varié d'injections valides pour chaque motif du composant testé.

4.4 Algorithme de génération d'injections

Dans le cas d'un composant à tester mais dont on ne dispose pas du code source, la stratégie précédente ne peut pas être appliquée. Il faut donc changer de stratégie et il est alors préférable de générer un grand nombre d'injections en espérant qu'une d'entre elles pourra effectivement exploiter une vulnérabilité sur le composant. Nous pouvons toutefois nous inspirer des résultats précédents pour construire un nouveau type de *fuzzer*.

Ce *fuzzer* prend en entrée une grammaire du langage qu'il cible ainsi qu'un point d'injection dans ce langage. Le point d'injection est choisi arbitrairement mais la connaissance des habitudes de développement permet rapidement de cibler les points d'injection les plus judicieux. Par exemple, dans le langage `sql`, il est plus judicieux de choisir, comme point d'injection, une chaîne de caractères qui apparaît à droite d'une égalité, plutôt que le premier terminal de la requête `sql` (qui peut être `SELECT`, `DROP`, etc.). Sur la base de la grammaire, les *parsers top-down*[GJ90] peuvent aisément être modifiés pour générer des mots du langage qui contiennent forcément le point d'injection. En exécutant plusieurs fois ce *parser*, nous disposons d'un ensemble de mots contenant le point d'injection. Chacun de ces mots peut être un motif proche de ceux utilisés dans le composant à tester. Nous nous ramenons donc au cas précédent.

Il est également envisageable de collecter plusieurs grammaires des injections, une par mot généré par le *parser*, et de réaliser l'union de ces grammaires pour obtenir une grammaire un peu plus générale pour les injections. Cette solution peut être implémentée sous réserve de ne pas réaliser l'union d'un trop grand nombre de grammaires pour deux raisons : *1*) il n'existe pas d'algorithme permettant de minimiser la taille des grammaires algébriques[TK70][1] et *2*) l'union d'une infinité de grammaires algébriques peut être une grammaire contextuelle et il est donc impossible de manière générale de converger vers une grammaire algébrique en réalisant les unions successives[2]. Malgré ces limites, la grammaire ainsi générée peut tout de même servir de base pour la création d'un ensemble d'injections. Cet algorithme constitue un troisième résultat.

5 Conclusion et perspectives

Les approches employées pour la conception des *fuzzers* s'appuient rarement sur le formalisme des grammaires. Pourtant, l'utilisation de ce formalisme semble très intéressante pour vérifier si les constructions utilisées dans un code source sont vulnérables à des attaques par injection. Il est également très intéressant pour la génération d'exemples d'injections aléatoires et variés.

---

[1] la taille d'un grammaire algébrique correspond à la somme des tailles des parties droite et gauche des règles. Par exemple, la taille de la grammaire de la figure 5 est 28.

[2] Le cas le plus flagrant est l'union des mots { **abc**, **aabbcc**, **aaabbbccc**, ... } qui aboutit au langage $\{a^n b^n c^n, n > 0\}$ qui est connu pour ne pas être algébrique.

Dans cet article, nous présentons trois résultats. Le premier résultat est une démonstration sur la difficulté de génération d'une grammaire pour le langage des injections. Le second résultat correspond à un algorithme permettant de générer des exemples d'injections à partir d'une grammaire, d'un point d'injection et d'un motif de construction de mots du langage. Le troisième résultat est un algorithme général pour la génération d'injections associé à une grammaire et un point d'injection.

Nous envisageons de poursuivre les analyses en nous focalisant, dans un premier temps, sur des grammaires algébriques particulières, par exemple les grammaires déterministes. Il s'agit d'un sous ensemble des grammaires algébriques. Aussi, nous souhaitons identifier une construction permettant de générer une grammaire algébrique pour les injections, si une telle grammaire existe. Dans un second temps, nous envisageons de poursuivre l'analyse en déterminant des propriétés sur les grammaires, voire les langages, de manière à pouvoir identifier si une grammaire ou un langage ne peut pas faire l'objet d'injections.

References

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, June 1959.

[DCV10] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 111–131. Springer Berlin Heidelberg, 2010.

[FVM07] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365–372, Dec 2007.

[Gef88] V. Geffert. A representation of recursively enumerable languages by two homomorphisms and a quotient. *Theor. Comput. Sci.*, 62(3):235–249, December 1988.

[GHGT+08] Liu Guang-Hong, Wu Gang, Zheng Tao, Shuai Jian-Mei, and Tang Zhuo-Chun. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, volume 2, pages 491–497, November 2008.

[GJ90] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, Upper Saddle River, NJ, USA, 1990.

[GMICL11] Amaury Gauthier, Clément Mazin, Julien Iguchi-Cartigny, and J Lanet. Enhancing fuzzing technique for okl4 syscalls testing. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 728–733. IEEE, 2011.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 445–458, Berkeley, CA, USA, 2012. USENIX Association.

[ISO92] ISO. *ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL*. 1992. Available in English only.

[Jon13] Dave Jones. Trinity: A linux kernel fuzz tester, February 2013.

[JR12] Russell J. and Cohn R. *W3af*. Book on Demand Ltd., 2012.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[Lan64] Peter S. Landweber. Decision problems of phrase-structure grammars. *Electronic Computers, IEEE Transactions on*, EC-13(4):354–362, Aug 1964.

[LT90] Michel Latteux and Paavo Turakainen. On characterizations of recursively enumerable languages. *Acta Informatica*, 28(2):179–186, 1990.

[MP07] Charlie Miller and Zachary N. J. Peterson. Analysis of Mutation and Generation-Based Fuzzing. Technical report, Independent Security Evaluators, March 2007.

[MZR] Niels Heinen Michal Zalewski and Sebastian Roschke. skipfish: web application security scanner.

[Rud07] Jesse Rudermann. Introducing jsfunfuzz, 2007.

[TK70] Kenichi Taniguchi and Tadao Kasami. Reduction of context-free grammars. *Information and Control*, 17(1):92 – 108, 1970.

[vS05] Ilja van Sprundel. Unix kernel auditing, November 15-16, 2005.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[Zal11] Michal Zalewski. Announcing cross fuzz – blog entry, 2011.