Identification de vulnérabilités Web et génération de scénarios d'attaque

Rim Akrout, Eric Alata, Mohamed Kaâniche, Vincent Nicomette

Identification de vulnérabilités Web et génération de scénarios d'attaque

Mai, 2013

Rim Akrout, Eric Alata, Mohamed Kaâniche, Vincent Nicomette

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse
Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
{rakrout,ealata,kaaniche,nicomett}@laas.fr

ABSTRACT. *Web applications have become increasingly exposed to malicious attacks that could affect essential properties such as confidentiality, integrity or availability of information systems. To cope with these threats, it is necessary to develop efficient security protection mechanisms and testing techniques (firewall, intrusion detection system, Web scanner, etc.).*
This paper presents a new methodology, based on Web pages clustering techniques, that is aimed at identifying the vulnerabilities of a Web application following a black box analysis of the target application. Each identified vulnerability is actually exploited to ensure that the identified vulnerability does not correspond to a false positive. The proposed approach can also highlight different potential attack scenarios including the exploitation of several successive vulnerabilities, taking into account explicitly the dependencies between these vulnerabilities. We have focused in particular on code injection vulnerabilities, such as SQL injections. The proposed method led to the development of a new Web vulnerability scanner and has been validated experimentally based on various vulnerable applications. This methodology has been implemented and has been validated experimentally on several examples of vulnerable applications.

RÉSUMÉ. *Les applications Web sont devenues de plus en plus des cibles de choix pour les attaquants. Pour faire face à ces malveillances, il est donc nécessaire de développer des mécanismes de protection et de test (pare feu, système de détection d'intrusion, scanner Web, etc.) qui soient efficaces.*
Dans cet article, nous proposons une nouvelle méthode, basée sur des techniques de clustering de pages Web, qui permet d'identifier les vulnérabilités à partir de l'analyse selon une approche boîte noire de l'application cible. Chaque vulnérabilité identifiée est réellement exploitée ce qui permet de s'assurer que la vulnérabilité identifiée ne correspond pas à un faux positif. L'approche proposée permet également de mettre en évidence différents scénarios d'attaque

potentiels incluant l'exploitation de plusieurs vulnérabilités successives en tenant compte explicitement des dépendances entre les vulnérabilités. Nous nous sommes intéressés plus particulièrement aux vulnérabilités de type injection de code, comme les injections SQL. Cette méthode s'est concrétisée par la mise en œuvre d'un nouveau scanner de vulnérabilités et a été validée expérimentalement sur plusieurs exemples d'applications vulnérables.

KEYWORDS: *Web application, Vulnerabilities, Attacks, Evaluation, Web Scanner*

MOTS-CLÉS : *Application Web, Attaque, Vulnérabilités, Évaluation, Scanner Web.*

DOI:10.3166/HSP.x.1-33

Contents

1 Introduction **5**

2 Contexte et état de l'art **6**

2.1 Approche par dictionnaire de messages d'erreurs 7

2.2 Approche par comparaison des réponses 8

2.3 Analyse critique des scanners de vulnérabilités web 8

3 Détection de vulnérabilités par classification automatique des pages HTML 9

3.1 Principes 10

3.2 Génération des trois ensembles de requêtes 10

3.3 Regroupement des pages en grappes 11

3.4 Grammaire pour la génération des requêtes 12

3.5 Extension à d'autres classes de vulnérabilités 13

4 Mise en œuvre de l'approche **14**

4.1 Définitions 15

4.2 Principes 15

4.3 Exemple 18

5 Algorithme **20**

6 Expérimentations **22**

6.1 Notation 24

6.2 Expériences avec les applications modifiées 24

6.2.1 Présentation des applications modifiées 24

6.2.2 Résultats 26

6.3 Expériences avec les applications non modifiées 27

6.3.1 Présentation des applications non modifiées 28

6.3.2 Résultats 29

6.4 Synthèse 30

7 Conclusion **31**

References **32**

1. Introduction

De nos jours le développement du Web dynamique et la richesse fonctionnelle qu'offrent les nouvelles technologies du Web permettent de répondre à un grand nombre de besoins. Néanmoins, cette richesse fonctionnelle s'accompagne d'une complexité grandissante des technologies utilisées aujourd'hui pour réaliser les applications Web (Java, JavaScript, PHP, Ruby, J2E, etc.). Aussi, il est particulièrement difficile d'une part d'empêcher l'introduction de vulnérabilités dans ces applications Web (tel qu'illustré par exemple par les statistiques publiées par Mitre[1]) et d'autre part d'estimer ou de prévoir leur présence. Nous pouvons expliquer également la multiplication de vulnérabilités par d'autres raisons : les délais sans cesse plus courts de mise sur le marché de logiciels, les compétences parfois limitées et le manque de culture en sécurité des développeurs. Ces vulnérabilités peuvent être exploitées par les attaquants pour mettre en oeuvre différentes actions malveillantes et dangereuses. Elles peuvent leur permettre, par exemple, d'obtenir des données confidentielles (numéros de cartes de crédit, mots de passe, etc.) qui sont manipulées par l'application, voire même de modifier ou détruire certaines de ces données. Il est donc nécessaire d'auditer régulièrement les applications Web pour vérifier la présence de vulnérabilités exploitables et ceci peut être réalisé notamment par des scanners de vulnérabilités Web.

Dans cet article, nous proposons une méthodologie originale permettant d'identifier automatiquement les vulnérabilités résiduelles d'une application Web à partir de l'analyse dynamique selon une approche boîte noire de l'application cible. L'approche proposée permet d'identifier et d'exploiter automatiquement des vulnérabilités élémentaires. Elle permet aussi de mettre en évidence différents scénarios d'attaque potentiels incluant l'exploitation de plusieurs vulnérabilités successives qui ne sont pas nécessairement indépendantes. Ces dépendances se traduisent par le fait que certaines vulnérabilités ne peuvent être découvertes qu'après l'exploitation réussie d'autres vulnérabilités. L'identification de ces scénarios est basée sur un parcours dynamique de l'application qui se traduit par l'élaboration d'un graphe de navigation. Ce graphe décrit les différentes possibilités pour un attaquant d'activer l'application et les vulnérabilités associées. Il représente aussi de façon explicite les dépendances entre les vulnérabilités du site et par la suite les différents scénarios d'attaques. Cette approche s'est concrétisée par le développement d'un nouvel outil de détection de vulnérabilités que nous avons validé sur plusieurs applications vulnérables, et par comparaison avec d'autres outils existants.

Cet article débute, dans la section 2, par une présentation de l'état de l'art lié aux travaux sur la détection de vulnérabilités dans des applications Web. Dans la section 3 nous détaillons le principe de classification de pages Web sur lequel s'appuie notre approche. Nous présentons ensuite dans la section 4 une vue globale de notre approche et nous décrivons en détail les algorithmes utilisés dans la section 5. Afin de valider notre approche, deux séries d'expérimentations réalisées sur différents types

1. http://www.cve.mitre.org/top25

d'applications sont présentées dans la section 6. La section 7 conclut et propose des perspectives de recherche à nos travaux.

2. Contexte et état de l'art

Selon des statistiques récentes publiées par des sources diverses (par exemple, l'OWASP[2] (**O**pen **W**eb **A**pplication **S**ecurity **P**roject) ou IBM (IBM, 2012)), les attaques les plus courantes ciblant les serveurs Web sont les attaques d'injection SQL (lorsque le serveur Web est connecté à une base de données SQL) et d'injection de code *Javascript* (réalisées sous la forme d'attaques de type *Cross Site Scripting* ou *XSS*). Ces injections de code proviennent de l'exploitation du même type de vulnérabilité des serveurs Web, appellé faille d'injection.

Une faille d'injection se produit quand une donnée non fiable est envoyée à un interpréteur en tant qu'élément d'une commande ou d'une requête. Ces données sont destinées à duper l'interpréteur afin de l'amener à exécuter des commandes malveillantes ou à accéder à des données non autorisées (Halfond, 2006). Ainsi, les attaquants cherchent à tirer profit d'une entrée utilisateur à travers un point d'injection dont l'utilisation n'est pas suffisamment protégée et assainie. Nous définissons *un point d'injection* par une entrée d'une page Web dans laquelle il est possible d'injecter du code : un paramètre d'une URL ou un champ d'un formulaire. L'assainissement d'une entrée utilisateur consiste à transformer le contenu, avant le traitement de celui-ci, de telle manière qu'il ne puisse être exécuté comme du code informatique.

On distingue généralement différentes familles d'injection en fonction du protocole visé (SQL, XML, XPATH, LDAP) ou du type d'injection (par exemple injection d'un programme exécutable dans le cas des failles de type "OsCommanding", ou chargement d'un fichier dans le cas des failles de type "FileUpload").

Pour vérifier si ces attaques d'injection de code sont possibles, les outils de détection de vulnérabilités envoient des requêtes particulières et analysent les réponses retournées par le serveur. Un serveur peut répondre avec une *page de rejet* ou une *page d'exécution*. Une page de rejet correspond à la détection par le serveur de valeurs d'entrée mal-formées ou invalides. Une page d'exécution est renvoyée par le serveur suite à l'activation réussie de la requête. Elle peut correspondre soit au scénario "normal", dans le cas d'une utilisation légitime du site, soit à un détournement de son exécution via l'exploitation réussie d'une injection de code. Pour identifier les vulnérabilités d'un site Web, les outils de détection de vulnérabilités, généralement appelés scanners de vulnérabilités, soumettent au site des requêtes contenant des données non conformes correspondant à des attaques potentielles. Les réponses sont alors analysées afin d'identifier les pages d'exécution. Si une page d'exécution est identifiée, elle est considérée vulnérable. C'est ainsi que les outils détectent l'absence d'assainissement

2. https://www.owasp.org

des paramètres. Tout le problème vient donc de l'analyse des réponses pour déterminer s'il s'agit réellement d'une page de rejet ou d'une page d'exécution.

Prenons l'exemple d'une page d'authentification qui utilise une base de données SQL pour conserver les couples *nom d'utilisateur / mot de passe* valides. Un outil de détection de vulnérabilités doit déterminer si la page est vulnérable à une injection SQL qui permettrait à un attaquant de contourner l'authentification. A la requête d'authentification soumise, le serveur peut retourner deux types de réponses : succès ou échec de l'authentification qui se traduisent généralement par l'affichage de deux pages différentes au niveau du navigateur du client en terme de code HTML. Ces pages peuvent varier dans leur forme, en fonction du langage utilisé, du site lui-même, du développeur, etc. Les outils de détection de vulnérabilités doivent donc automatiquement classifier la réponse retournée afin de déterminer si la vulnérabilité est présente ou pas.

Dans la suite nous analysons les stratégies adoptées par différents outils de détection de vulnérabilités en considérant des outils en source libre tels que *W3af*[3](1.1), *Skipfish*[4](1.9.6b) et *Wapiti*[5](2.2.1). Le choix s'est porté sur des outils en source libre afin de pouvoir analyser leurs algorithmes mis en œuvre. Une telle analyse n'est pas possible avec des outils commerciaux tels que *Acunetix*, *WebInspect*, *AppScan*, etc. On peut distinguer deux principales classes d'approches adoptées par les scanners de vulnérabilités: 1) une approche par dictionnaire basée sur la reconnaissance de messages d'erreurs, et 2) une approche par comparaison des réponses associées aux requêtes envoyées au serveur. Dans la suite nous décrivons les principes de ces deux approches en considérant la détection d'injections SQL comme exemple. On peut noter aussi que certains travaux, par exemple (Huang, 2003), utilisent ces deux approches de façon combinée.

2.1. Approche par dictionnaire de messages d'erreurs

Cette approche consiste à envoyer des requêtes d'un format particulier et chercher des motifs spécifiques dans les réponses tels que les messages d'erreurs de base de données. L'idée est que la présence d'un message d'erreur SQL dans une page HTML de réponse signifie que la requête correspondante n'a pas été vérifiée par l'application Web avant d'être transmise au serveur de bases de données. Par conséquent, le fait que cette requête a été envoyée inchangée au serveur SQL révèle la présence d'une vulnérabilité. Les scanners tels que *W3af* (module SQLI), *Wapiti* et *Secubat* (Stefan, 2006) adoptent une telle approche. Par exemple *Secubat* utilise une liste de messages d'erreurs obtenue par l'analyse de réponses des pages de sites web vulnérables, qui est destinée à couvrir un large éventail de réponses correspondant à des exécutions échouées et une variété de serveurs de base de données.

3. http://w3af.sourceforge.net
4. http://code.google.com/p/skipfish
5. http://wapiti.sourceforge.net

2.2. Approche par comparaison des réponses

Le principe de cette approche consiste à envoyer différentes requêtes spécifiques aux types de vulnérabilités recherchées et à étudier la similitude des réponses renvoyées par l'application en utilisant une distance. En fonction des résultats obtenus et de critères bien définis, on conclut sur l'existence ou non d'une vulnérabilité. Conctrairement à la première approche, elle ne s'appuie pas sur la connaissance a priori d'un dictionnaire de messages d'erreurs Prenons comme exemple l'approche adoptée par *Skipfish* pour détecter les vulnérabilités d'injection SQL. *Skipfish* est un outil développé par Google. Il procède en deux étapes. Dans une première étape, il parcourt le site et collecte toutes les pages qui lui semblent stables. Les autres sont ignorées. Pour détecter si une page est stable, *Skipfish* envoie 15 requêtes identiques et compare les réponses correspondantes. Si les réponses sont similaires, la page est considérée stable. Dans la deuxième étape, plusieurs tests sont réalisés sur ces pages stables, en fonction du type de vulnérabilités recherchées. En particulier, un de ces tests concerne les injections SQL. Cette vulnérabilité est testée grâce à 3 requêtes A, B et C incluant chacune une injection SQL : A) `'"`, B) `\'\"` et C) `\\'\\"`. Les réponses sont comparées deux à deux. Selon *Skipfish*, une vulnérabilité est présente si les réponses associées aux injections A et B ne sont pas similaires ainsi que les réponses associées aux injections A et C. Le test de similarité utilise les fréquences d'apparition des mots dans les réponses.

2.3. Analyse critique des scanners de vulnérabilités web

Les deux approches que nous venons de présenter présentent un certain nombre de limites. L'efficacité des scanners de vulnérabilités est généralement analysée en évaluant expérimentalement le pourcentage de faux positifs (fausses alarmes) et de faux négatifs (absences de détection) observés. L'efficacité de l'approche par dictionnaire de messages d'erreurs est liée à la complétude de la base de connaissance regroupant les messages d'erreurs susceptibles de résulter de l'exécution des requêtes soumises à l'application Web. Généralement, comme c'est dans le cas de *W3af*, on considère principalement les messages d'erreurs issus de la base de données. Cependant, les messages d'erreurs qui sont inclus dans des pages HTML de réponse ne proviennent pas forcément du serveur de bases de données. Le message d'erreur peut également être généré par l'application qui peut aussi reformuler le message d'erreur issu du serveur, par exemple pour le rendre compréhensible par le client. Par ailleurs, même si le message est généré par le serveur de base de données, la réception de ce message n'est pas suffisante pour affirmer que l'injection SQL est possible. En effet, ce message signifie que, pour cette requête particulière, les entrées n'ont pas été assainies, mais ne permet pas de conclure par rapport à d'autres requêtes SQL, en particulier celles qui seraient susceptibles de correspondre à des attaques réussies.

En ce qui concerne l'approche par comparaison des réponses, elle se base sur l'hypothèse que le contenu d'une page de rejet est généralement différent du contenu d'une page d'exécution. Pour que cette comparaison puisse cependant être efficace, il

est important d'assurer une large couverture des différents types de pages de rejet qui pourraient être générés par l'application. Ceci peut être réalisé en générant un grand nombre de requêtes visant à activer le plus grand nombre possible de pages de rejet variées. Cependant, les implémentations existantes de cette approche, en particulier dans *Skipfish*, génèrent trop peu de requêtes. *Skipfish* utilise seulement 3 requêtes. Si les réponses correspondent à différentes pages de rejet, il conclut à tort que la vulnérabilité est présente conduisant ainsi à un faux positif.

Par ailleurs, pour l'approche par comparaison, comme dans tout problème de classification, le choix de la distance est très important. Celle utilisée dans *Skipfish* ne prend pas en compte les ordres des mots dans un texte. Cependant, cet ordre définit généralement la sémantique de la page. Il est donc important d'en tenir compte comme par exemple dans (Huang, 2003).

Enfin, il est à noter qu'aucun des outils que nous avons analysés n'a été conçu pour générer automatiquement des requêtes d'attaque qui mènent à l'exploitation réussie de la vulnérabilité identifiée. Cependant, une telle possibilité serait utile pour déterminer si la vulnérabilité suspectée peut être effectivement exploitée et réduire ainsi le taux de faux positifs. Outre les analyses que nous venons de faire en considérant principalement les algorithmes implémentés dans *Skipfish*, *W3af* et *Wapiti*, d'autres études basées sur des analyses et des approches expérimentales ont aussi fait état de certaines limitations des scanners web, incluant des outils commerciaux (Bau, 2010), (Doupe, 2010) et (Fonseca, 2007). Ces études s'appuient sur l'utilisation d'applications vulnérables et l'activation de ces applications avec des entrées qui sont conçues pour activer ces vulnérabilités.

Ces analyses ont révélé que les outils existants présentent des taux élevés de faux positifs et de faux négatifs. Elles montrent clairement le besoin de développer de nouvelles approches permettant d'améliorer l'efficacité des outils de détection de vulnérabilités et les capacités d'automatisation des campagnes d'évaluation. Les travaux présentés dans le cadre de cet article s'inscrivent dans cette optique.

3. Détection de vulnérabilités par classification automatique des pages HTML

L'analyse effectuée dans la section précédente montre qu'il est nécessaire d'améliorer les performances des différents scanners de vulnérabilités. L'approche que nous présentons dans cette section vise à atteindre cet objectif. Elle permet la détection automatisée des différents types de vulnérabilités Web, correspondant aux attaques de type injections SQL, OsCommanding, File Include et XPath[6]. Elle est basée sur la classification automatique des réponses retournées par les serveurs Web en utilisant les techniques de regroupement de données (ou clustering) et permet d'identifier les requêtes qui sont capables d'exploiter avec succès des vulnérabilités présentes. L'exploitation réussie des vulnérabilités permet ainsi de réduire les faux positifs.

6. voir (Akrout, 2012) pour une description plus détaillée de ces vulnérabilités

3.1. Principes

Notre algorithme de classification a pour but d'identifier, de manière automatique, si un point d'injection contient une vulnérabilité qu'il est possible d'exploiter avec succès. Pour cela, nous soumettons à chaque point d'injection identifié un ensemble de requêtes contenant :

1. des requêtes générées aléatoirement, notées R_a
2. des requêtes d'injection de code syntaxiquement invalides, notées R_{ii}
3. et des requêtes d'injection de code syntaxiquement valides, notées R_{iv}.

Les deux premiers ensembles de requêtes visent à identifier les différents types de pages de rejet qui peuvent être générées par l'application, suite à une exécution non réussie de la requête envoyée par un client. L'identification des requêtes d'injection de code réussies peut être ainsi effectuée en comparant les pages retournées avec celles obtenues à partir des deux premiers ensembles. Celles qui s'en écartent significativement correspondront vraisemblablement à des attaques réussies. C'est cette idée de base qui est mise en œuvre dans notre approche. Grâce à une technique de regroupement en grappes (ou clusters) des réponses associées à ces trois ensembles de requêtes, notre algorithme peut automatiquement déterminer les requêtes qui ont réellement permis l'exploitation de la vulnérabilité, requêtes baptisées *requêtes d'injection réussie*.

Cet algorithme nécessite, en entrée, un point d'injection. Dans la suite, nous considérons comme exemple d'illustration le cas d'une page d'authentification où les entrées correspondent aux couples "login, mot de passe" et nous nous intéressons uniquement aux injections SQL. Nous présentons tout d'abord la génération des trois ensembles de requêtes cités ci-dessus, puis la technique de regroupement des réponses associées. Cette technique est basée sur l'utilisation d'une distance et d'un seuil que nous présentons également.

3.2. Génération des trois ensembles de requêtes

Dans le contexte d'une authentification sur une application Web, notre objectif est d'identifier, parmi un ensemble d'injections SQL possibles, celles qui permettent effectivement de contourner l'authentification. Le principal défi réside dans l'automatisation de ce processus. Nous proposons une méthode qui vise à réduire à la fois le nombre de faux positifs et faux négatifs, comparé aux solutions des outils existants. Cette méthode se base sur plusieurs constats :

– les pages de *rejet* peuvent être différentes entre elles et elles sont différentes des pages d'*exécution* en terme de contenu textuel

– deux pages d'*exécution* peuvent être aussi différentes

Par exemple, les réponses associées à des données valides contiendront des messages de bienvenue et les réponses associées à des données invalides contiendront des messages d'erreur relatifs au langage de programmation (PHP, ruby on rails, etc.) ou

des messages d'erreur SQL. Le point important est l'existence de différences entre les pages de rejet et les pages d'exécution. Notre approche vise à étudier ces différences afin de déterminer, parmi des réponses différentes, celles qui sont des pages d'exécution.

Pour débuter cette classification, nous avons besoin de requêtes initiales dont nous connaissons le type des réponses associées (rejet ou exécution). Clairement, il est plus facile de générer des requêtes qui engendrent une page de rejet. Il suffit par exemple de générer aléatoirement les noms d'utilisateur et mots de passe permettant de renseigner le formulaire d'authentification. Un autre exemple concerne les requêtes volontairement malformées générant des erreurs SQL.

Nous notons S_a, S_{ii} et S_{iv} les réponses associées aux requêtes R_a, R_{ii} et R_{iv} respectivement. Le principe de notre algorithme est alors le suivant :

"les requêtes R_{iv} dont les réponses S_{iv} ne sont similaires à aucune des réponses S_{ii} et S_a, sont considérées comme des injections SQL réussies (i.e., la vulnérabilité a été exploitée avec succès)."

Pour évaluer la similarité entre les pages renvoyées par les différentes requêtes, nous utilisons une technique de classification basée sur le calcul de la distance entre les réponses. La section suivante présente cette distance.

3.3. Regroupement des pages en grappes

Pour analyser la similarité entre deux pages HTML, nous avons besoin d'une distance permettant d'évaluer la différence entre deux chaînes de caractères. L'ordre des mots dans un texte peut avoir une grande importance. En effet, les mêmes mots dans un ordre différent peuvent changer complètement la sémantique de la réponse. Généralement, le principe consiste à calculer le nombre minimal de modifications de la première chaîne permettant d'obtenir la seconde en autorisant les opérations de remplacement, suppression et ajout de caractères. Un des problèmes à résoudre est l'identification de la plus longue sous-chaîne commune entre deux chaînes. Ce problème peut être résolu par différentes approches, notamment par programmation dynamique. La distance de *Levenshtein* en est un exemple (Levenshtein, 1996). La commande `diff` (Hunt, 1976) disponible sur tout système Unix est un autre exemple de programme qui résout ce problème. Une particularité de ce programme est qu'il considère les chaînes de caractères lignes par lignes et non caractère par caractère. L'opérateur de différence (distance) que nous utilisons dans le cadre de notre approche est ainsi une version légèrement modifiée de la distance de *Levenshtein*. Il s'exprime formellement comme suit. Soient a et b deux réponses de longueur n et m. Notons a_i ($i = 1,\cdots, n$) les lignes de la première réponse a, et b_j ($j = 1,\cdots, m$) les lignes de la seconde réponse b. La distance[7] est formalisée dans la figure 1. Notre algorithme de classification des réponses va ainsi soumettre à l'application des requêtes

7. Cette distance respecte les propriétés de base : symétrie, séparation et inégalité triangulaire.

$$\mathsf{diff}(a_i, b_j) = \begin{cases} n - i + m - j & i = n \text{ ou } j = m \\ \mathsf{diff}(a_{i+1}, b_{j+1}) & a_i = b_j, i < n, j < m \\ 1 + \min(\mathsf{diff}(a_{i+1}, b_j), \mathsf{diff}(a_i, b_{j+1})) & \\ & a_i \neq b_j, i < n, j < m \end{cases}$$

$$d(a, b) = \frac{\mathsf{diff}(a_1, b_1)}{(n + m)}$$

Figure 1. Distance de classification

issues des 3 ensembles présentés dans la section précédente et calculer la distance deux à deux entre toutes les réponses associées à ces requêtes. Ensuite, en fonction de la valeur de ces distances, l'algorithme va regrouper les requêtes afin d'obtenir différentes grappes. L'idée sous-jacente est de chercher une grappe ne contenant que des requêtes appartenant à l'ensemble R_{iv}. Cette grappe constitue alors les requêtes ayant réellement permis d'exploiter la vulnérabilité, que nous appelons requêtes d'injection réussie.

De façon générale, les techniques de regroupement s'appuient sur deux stratégies différentes. La première est guidée par le nombre de grappes que l'on désire obtenir. Il s'agit alors de partir d'une grappe unique contenant toutes les requêtes et de la diviser petit à petit, en s'appuyant sur les distances, jusqu'à obtenir le nombre de grappes souhaitées. La seconde est utilisée dans le cas où on ne connaît pas a priori ce nombre. Il s'agit alors de regrouper ensemble au sein d'une même grappe les requêtes dont la distance deux à deux est inférieure à un seuil. Dans notre situation, le nombre de grappes n'est pas déterminé a priori car il dépendre des différents types de pages de rejets qui peuvent être asssociés au point d'injection considéré. Pour cette raison, nous avons opté pour la seconde stratégie, appelée regroupement hiérarchique (Johnson, 1967) qui nécessite le choix d'un seuil.

Le seuil de regroupement des requêtes peut varier d'un point d'injection à l'autre. En effet, il dépend de la taille des réponses et de la quantité de données qui changent entre deux réponses associées à des requêtes du même type. Il varie également d'une application Web à une autre puisque leur mise en œuvre peut être très différente. Ce seuil doit donc être adapté à chaque application Web. Il doit permettre de déterminer la valeur limite permettant de définir les pages qui se ressemblent (quelques caractères modifiés) et les pages qui ne se ressemblent pas (beaucoup de caractères modifiés). Afin de respecter ce principe, notre choix a été de définir ce seuil par la plus petite distance entre i) la distance la plus longue entre deux réponses dans S_a et ii) la distance la plus longue entre deux réponses dans S_{ii}.

3.4. Grammaire pour la génération des requêtes

Un objectif important de l'algorithme proposé est d'identifier la présence d'une vulnérabilité dans un point d'injection en fonction des réponses multiples générées à

partir de ce point d'injection. Pour améliorer la précision des résultats, il est nécessaire de générer un nombre suffisamment grand de requêtes permettant d'assurer une couverture élevée des différentes réponses pouvant être renvoyées par le serveur Web pour chaque point d'injection considéré. Notons que d'autres approches génèrent un nombre réduit de requêtes (par exemple, 3 pour *Skipfish*), ce qui peut donner lieu à des faux positifs, ce que nous voulons éviter dans notre approche.

Une manière possible de procéder est d'enregistrer dans un fichier statique toutes les requêtes que l'on souhaite envoyer. En particulier, pour les requêtes de la catégorie R_{iv}, on peut se baser sur les connaissances provenant d'experts en sécurité ou disponibles sur Internet (par exemple, utiliser les "SQL Sheets" (Kiezun, 2009)). Cette approche présente l'avantage de pouvoir générer des requêtes particulièrement subtiles puisqu'elles sont produites par de réels experts du domaine. Son inconvénient majeur est qu'elle est statique puisqu'il faut saisir une à une toutes les requêtes que l'on veut envoyer. En particulier, si l'on veut pouvoir envoyer des variantes d'un certain type de requêtes, il est nécessaire de toutes les imaginer et les écrire. Il en est de même pour les requêtes syntaxiquement invalides et les requêtes aléatoires.

Une approche plus flexible consiste à définir une grammaire ou un modèle pour automatiser ce processus. Cette approche présente des avantages certains. Elle permet de générer de façon complètement automatique un très grand nombre de requêtes. Dans le cas du langage SQL qui nous intéresse ici en particulier, la grammaire nous permet de générer une multitude de variantes d'un certain type d'injection. Une grammaire peut également facilement être mise à jour si on envisage un nouveau type d'injection que l'on n'avait pas considéré jusqu'à présent. L'avantage de la grammaire est qu'il suffit d'ajouter une règle pour pouvoir automatiquement générer une multitude de variantes de ce nouveau type d'injection. Les grammaires que nous avons définies pour générer les ensembles R_a, R_{ii} et R_{iv} sont présentées dans (Akrout, 2012).

Néanmoins, la mise en œuvre de cette approche peut aussi être confrontée à quelques difficultés. En effet, l'écriture d'une grammaire ou d'un modèle peut s'avérer complexe et nécessite de disposer d'une spécification du langage concerné (SQL par exemple dans notre cas). Ce modèle ou cette grammaire peuvent de plus s'avérer inadaptés si l'implémentation du langage ne respecte pas scrupuleusement les spécifications ou si l'application testée n'utilise qu'une partie de ce langage.

3.5. Extension à d'autres classes de vulnérabilités

Le principe de la détection de la vulnérabilité d'injection SQL peut être généralisé. En effet, de nombreuses attaques adoptent le même comportement : le client envoie une chaîne de caractères qui change la sémantique de la requête forgée. Selon le contexte, la requête forgée est envoyée à un composant spécifique côté serveur Web tel que le moteur XPATH, le système d'exploitation, etc. Les noms des attaques par injection correspondantes sont dérivés du nom de cette composante menant à l'injection : injection XPATH, Os Commanding, etc. Ainsi, l'algorithme de clustering que nous

avons présenté en prenant l'exemple de l'injection SQL dans la section précédente peut aussi être utilisé pour ces types de vulnérabilités.

La vulnérabilité XPATH correspond, comme la vulnérabilité SQL, à la soumission d'entrées dans un formulaire HTML ou paramètres d'URL non assainis, à la différence que cette vulnérabilité peut être exploitée pour exécuter des requêtes XPATH et non les requêtes SQL.

Dans le cas de la vulnérabilité OS Commanding, la chaîne envoyée par le client est utilisée pour créer une commande exécutée par le système d'exploitation. Cette commande est exécutée sous l'identité du processus correspondant au serveur Web. L'exploitation de cette vulnérabilité permet à un attaquant d'exécuter des commandes du système arbitraires et peut permettre également l'accès en lecture ou/et en écriture à certains fichiers.

Comme expliqué précédemment, l'algorithme utilise trois ensembles de requêtes: R_a (requêtes aléatoires), R_{ii} (injections syntaxiquement invalides) et R_{vi} (injections syntaxiquement valides). L'adaptation de l'algorithme à d'autres types de vulnérabilités ne nécessite que la définition de ces trois ensembles pour chaque type de vulnérabilité. Une fois que ces ensembles sont établis, l'algorithme procède de la même manière : envoyer ces requêtes, stocker les résultats correspondants et obtenir des grappes grâce à la distance que nous avons présentée dans la section 3.3.

Des exemples illustrant ces différentes requêtes sont détaillés dans (Akrout, 2012) pour des injections OS Commanding, Xpath et File Include.

4. Mise en œuvre de l'approche

La section précédente nous a permis de définir une méthodologie permettant la détection de vulnérabilités par classification automatique de pages HTML. Dans cette section, nous présentons maintenant une approche globale permettant d'exhiber des scenarios d'attaque combinant l'exploitation de plusieurs de ces vulnérabilités dont certaines peuvent être causalement dépendantes les unes des autres. L'objectif de cette approche est de pouvoir établir ces scenarios à partir de la construction de façon automatique d'un graphe qui représente l'ensemble des navigations possibles sur un site en tenant compte de ses vulnérabilités. L'approche est globalement constituée 1) d'une étape de navigation sur le site et 2) d'une étape d'identification et d'exploitation de vulnérabilités à l'aide de la méthodologie présentée dans la section précédente. Nous adoptons une approche de type boîte noire dans la mesure où aucune information sur la mise en œuvre du code source du site Web n'est requise. L'adresse publique du site web est utilisée comme point de départ pour la découverte de façon dynamique du site et de ses vulnérabilités. Nous commençons cette section par la définition de certains termes qui sont utiles pour la présentation de notre approche. Par la suite nous décrivons le principe de notre approche.

4.1. Définitions

Une *navigation* correspond à une séquence de requêtes envoyées au site par le client. L'ensemble des navigations effectuées par un utilisateur en activant le site peut être représenté sous la forme d'un graphe, appelé *graphe de navigation*. Un *état de navigation* est défini par l'ensemble des requêtes http susceptibles d'être réalisées par le client. Cet ensemble dépend de la page courante et du contenu du *cookie*[8] stocké en local sur le navigateur du client. Un état dépend de l'historique de la navigation sur le site.

Chaque *nœud* du graphe de navigation correspond à un état de navigation. Un *arc* entre deux états correspond à la possibilité d'exécuter une requête permettant, depuis un état de départ, d'atteindre un autre état. Un arc peut correspondre à une requête normale correspondant à une utilisation légitime du site, ou à une requête liée à l'exploitation d'une vulnérabilité sur le site. Un *graphe de vulnérabilités* est un cas particulier de graphe de navigation contenant des arcs correspondant à des exploitations de vulnérabilités.

Il est important de noter qu'un graphe de navigation est différent d'un arbre de pages HTML décrivant la structure du site Web. Dans un arbre de pages HTML, chaque nœud correspond à une page du site et un arc entre deux nœuds correspond à l'existence d'un lien HTML permettant de passer de la première page à la seconde. Ce type de graphe est utile pour représenter la structure d'un site Web. La différence entre un graphe de navigation et un graphe de pages HTML tient principalement du fait qu'un état de navigation ne dépend pas simplement de la page accédée. En effet, il est possible d'accéder à une même page d'un site tout en étant dans des états différents. Par exemple, dans un site marchand, nous pouvons accéder à la page de paiement soit en ayant sélectionné des produits dans le panier soit sans en avoir. Dans le premier cas, le cookie du navigateur contient les informations du panier et l'utilisateur peut poursuivre son paiement, contrairement au second cas où le site Web avertit que le panier est vide. Notons toutefois que, dans certains cas de sites statiques (sans contenu variable), le graphe de navigation et le graphe des pages HTML peuvent être identiques.

4.2. Principes

Nous nous plaçons dans la situation où nous disposons uniquement de l'adresse du site, qui constitue le premier paramètre de notre algorithme. La construction du graphe de navigation doit donc se faire de manière dynamique, par identification des différentes navigations sur le site et des vulnérabilités. De plus, l'exploitation d'une vulnérabilité ouvre de nouvelles possibilités de navigation. La construction doit donc également se faire de manière itérative. Notre approche est donc constituée d'une

8. Espace de stockage côté client permettant d'enregistrer diverses informations sur la session en cours (clefs, contenu du panier, préférence utilisateur, etc.)

étape de navigation appelée *crawling* permettant d'identifier les différentes navigations du site et ainsi les différents états de navigation et d'une étape d'identification des vulnérabilités qui s'appuie sur les principes que nous avons présentés dans la section 3.

La figure 2 présente une vue de haut niveau de l'approche proposée. Elle commence par l'analyse de l'URL initiale (qui correspond, la plupart du temps, à la page principale de l'application). A partir de cette URL, le parcours combinatoire du site permet d'identifier la liste des traces de navigation. Notre approche est basée sur une recherche exhaustive permettant d'obtenir toutes les navigations possibles du site. Pour ce faire, nous prenons soin de partir d'un état initial vide en supprimant les cookies côté client. Cette initialisation est importante pour que les différentes navigations soient indépendantes. En effet, sans suppression des cookies, un effet de mémoire peut rendre la seconde navigation dépendante de la première. Ensuite, nous naviguons sur le site en commençant par la requête initiale et en mémorisant les requêtes envoyées au site. Le choix de la requête à envoyer se fait en analysant le contenu de la page affichée. Si cette page contient plusieurs liens HTML, un de ces liens est choisi pour construire la requête suivante et les autres liens sont mémorisés pour les analyser ultérieurement. Sachant que la navigation à travers un site peut être infinie, nous bornons le nombre de requêtes envoyées. Cette borne représente la profondeur maximale de navigation du site. Elle constitue un second paramètre de notre algorithme.
Lorsque la profondeur maximale est atteinte ou l'état atteint ne permet plus d'envoyer de requêtes, autrement dit la page considérée ne contient plus de liens HTML, la séquence de requêtes mémorisées depuis l'état initial constitue une navigation du site. Nous recommençons alors à zéro en essayant de nouvelles navigations, en se basant sur les choix mémorisés jusqu'à avoir tout essayé compte tenu de la borne fixée. Nous obtenons ainsi un ensemble de séquences de requêtes représentant l'ensemble de navigations du site. Cet ensemble est utilisé pour construire une première version du graphe de navigation dépourvu d'arcs correspondant à des vulnérabilités. L'objectif est d'obtenir un graphe minimal qui permet de représenter l'ensemble des navigations obtenues. Le passage d'un ensemble de séquences de requêtes à un graphe peut être considéré comme un problème d'inférence grammaticale pour lequel nous cherchons à obtenir un automate minimal à partir d'exemples du langage, plus précisément à partir de séquences de symboles du langage. Dans cette analogie, l'automate correspond au graphe de navigation et les symboles correspondent aux requêtes. Notre choix s'est porté en particulier sur l'algorithme RPNI (**R**egular **P**ositive **N**egative Inference)(Dupont, 1996) qui présente une complexité en temps polynomial, raisonnable par rapport aux autres algorithmes, tout en étant simple à implémenter.

A la fin de cette étape de crawling, nous notons que la seule possibilité d'enrichir le modèle est de considérer des attaques permettant de rajouter des arcs voire des nœuds. Ainsi, nous appliquons sur ce graphe notre algorithme d'identification de vulnérabilités (détaillé dans la section 3). Cet algorithme permet l'identification et l'exploitation effective de vulnérabilités présentes. Il permet ainsi de découvrir de nouvelles pages qui peuvent contenir de nouveaux points d'injection qui n'étaient pas accessibles dans la première étape. Par conséquent, un nouveau domaine de l'application devient ac-

cessible. Par la suite, l'approche est ré-exécutée de façon itérative en incluant les nouvelles pages, ce qui conduit à l'obtention d'un nouveau graphe de navigation, jusqu'à satisfaire le critère d'arrêt, qui est spécifié par la profondeur maximale de navigation. Le graphe final est ainsi obtenu.

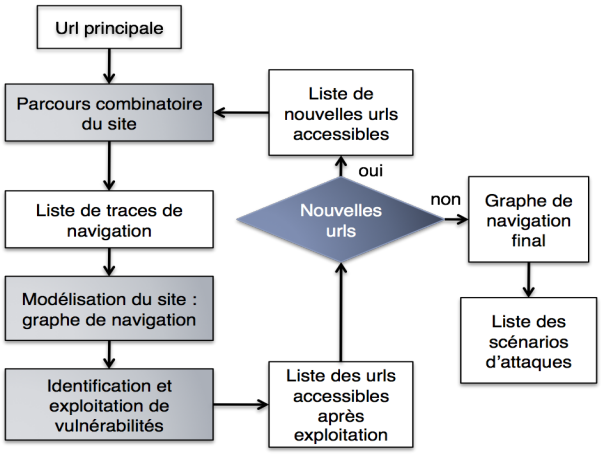


Figure 2. Algorithme d'extraction de points d'injection et de recherche de vulnérabilités

La figure 3 décrit de façon schématique cette approche itérative. Les arcs représentés en rouge identifient des vulnérabilités dont l'exploitation permet de révéler de nouveaux états et arcs du graphe de navigation qui étaient initialement inaccessibles.

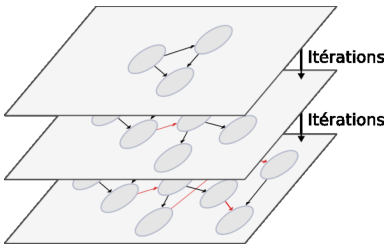


Figure 3. Principe de l'algorithme

Notre algorithme a été implémenté dans un outil développé en utilisant le langage Python, langage qui facilite grandement la gestion des concepts HTTP (cookies, paramètres, etc.). Cet outil utilise également le logiciel d'analyse statistique R[9] qui intègre un ensemble de programmes de clustering que nous avons détaillé dans la section 3. Ils ont été utilisés pour développer notre algorithme de classification. Cet outil est nommé *Wasapy*, qui signifie **W**eb **A**pplication **S**ecurity **A**ssessment in **Py**thon.

9. http://www.r-project.org/

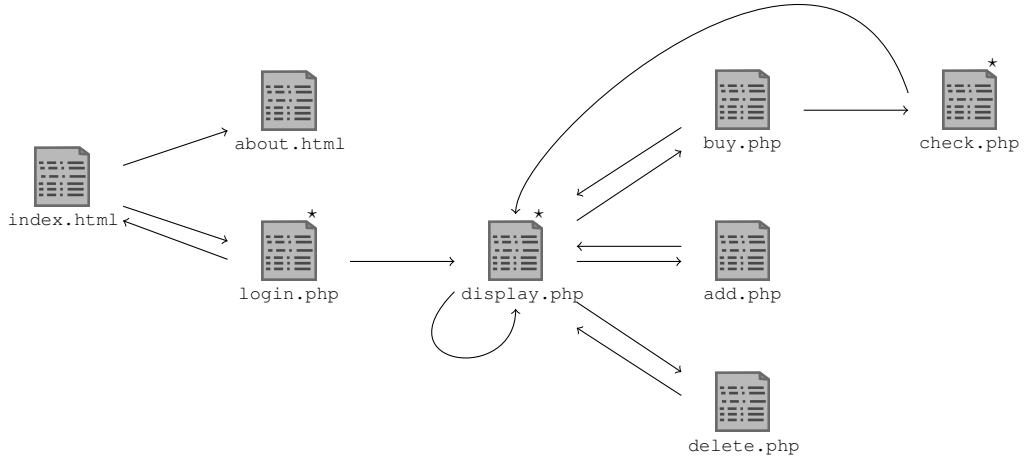


Figure 4. Structure du site Web

4.3. Exemple

Afin d'illustrer notre approche, nous avons développé un site Web en utilisant le langage `php` et la base de données `mysql`. Il s'agit d'un site de commerce électronique qui permet l'achat de livres. Ce site n'est pas fait pour être représentatif de tous les sites disponibles sur l'Internet. Néanmoins, il est conçu pour illustrer les dépendances entre vulnérabilités. Le site Web inclut des pages dynamiques et implémente les principales fonctionnalités qui sont couramment disponibles sur des sites sur Internet : connexion et authentification, recherche, description des livres disponibles, paiement, etc. Le site intègre aussi différentes vulnérabilités dont certaines ne peuvent être atteintes qu'après l'exploitation réussie d'autres vulnérabilités.

La figure 4 donne une vue d'ensemble du site. Une page est représentée par une icône de fichier. La page principale est `index.html`. Si une page contient un lien vers une autre page, une flèche est ajoutée entre les icônes correspondantes. La page `display.php` est associée à un lien réflexif car elle contient un lien utilisé pour mettre à jour le modèle de la recherche. Dans cet exemple, nous considérons trois vulnérabilités. Les pages qui contiennent ces vulnérabilités sont marquées par une étoile en haut à droite de l'icône.

La première vulnérabilité est associée à la page `login.php`. L'exploitation de cette vulnérabilité permet de contourner l'authentification par une injection SQL.

La seconde est associée à la page `display.php` et permet à l'attaquant de faire une copie du contenu de la base de données. La dernière est associée à la page `check.php` et elle permet d'acheter le contenu du panier sans fournir un numéro de carte valide. Elle ne peut être exploitée que si le produit a été ajouté au panier.

Tout d'abord, nous considérons le point de vue d'un utilisateur normal, non malveillant, qui ne dispose pas d'un compte sur le site. Les seules actions que cet utilisateur peut faire sont :

– accéder à la page `index.html`

– remplir le formulaire d'authentification avec des informations dans la page `login.php`

– obtenir des informations à partir de la page `about.html`

Ces actions sont présentées dans la liste des traces de navigation de la figure 6. Une trace de navigation est composée d'actions élémentaires successives où chacune correspond à l'exécution d'un lien de la page courante accédée. L'ensemble de ces traces peuvent être synthétisée par le graphe de navigation de la figure 5. Dans ce graphe, les arcs correspondent aux liens du site (page HTML ou php, etc.), les nœuds correspondent à des états de navigation. L'ensemble des arcs de sortie d'un nœud correspond à l'ensemble de liens accessibles de l'état de navigation correspondant. Cet ensemble est indépendant des liens précédemment utilisés pour atteindre cet état de navigation.

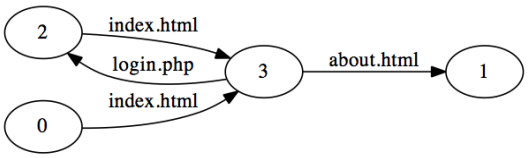


Figure 5. Graphe de navigation d'un utilisateur non enregistré

Ce graphe de navigation est très simple parce que les actions possibles sans (`login` / `password`) valides sont limitées. Toutefois, si nous considérons le point de vue d'un attaquant et s'il est en mesure d'exploiter correctement les vulnérabilités du site, alors il est capable d'exécuter plus d'actions que l'utilisateur normal. Par conséquent, le graphe de navigation associé correspond à une version plus riche que le graphe de la figure 5, de nouveaux arcs et nouveaux nœuds pouvant apparaître.

P_1 : `index.html`
P_2 : `index.html` → `about.html`
P_3 : `index.html` → `login.php`
P_4 : `index.html` → `login.php` → `index.html`

Figure 6. Traces de navigations utilisées pour tester les vulnérabilités

C'est précisément en quoi consiste l'étape suivante. Les navigations de la figure 6 sont testées afin d'identifier les vulnérabilités[10]. L'exploitation d'une nouvelle vulnérabilité peut changer l'état de navigation. Ainsi, elle peut mener à l'insertion d'un nouveau nœud dans le graphe. A cette étape, la seule vulnérabilité accessible correspond à l'injection SQL pour l'authentification, i.e., scénario P_3. Le résultat obtenu est présenté dans la figure 7.

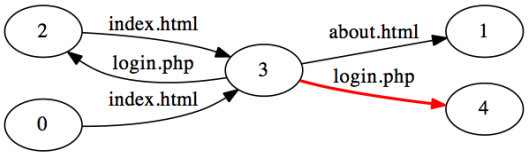


Figure 7. Résultats de la première itération de l'algorithme

Au cours de la deuxième itération, nous identifions les pages qui peuvent être accessibles suite à l'exploitation de la vulnérabilité identifiée lors de l'itération précédente. Afin d'atteindre ces pages, il est nécessaire de traverser les arcs `index.html` et `login.php`*. L'ensemble des traces exécutées pour cette deuxième itération contient 65 traces. Ces traces atteignent `display.php`, `add.php`, `delete.php`, `buy.php` ou `check.php`. Ensuite, la phase d'identification de vulnérabilités est exécutée une nouvelle fois pour chaque nouvel arc de cette seconde itération. Nous enchainons ainsi jusqu'à obtenir le graphe qui couvre la totalité du site, présenté dans la figure 8. Dans le cas de cet exemple, l'algorithme s'arrête après 6 itérations en considérant une profondeur maximale de navigation d'une valeur de 7. Cela signifie qu'il n'y a plus de vulnérabilités découvertes au cours de la sixième itération.

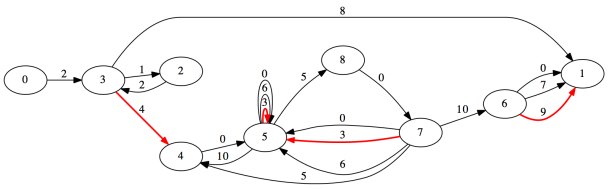
Pour automatiser le processus, nous avons implémenté des algorithmes correspondant aux deux étapes principales de notre approche : le crawling et la recherche de vulnérabilités. Ces algorithmes font l'objet de la section suivante.

5. Algorithme

Nous présentons dans cette section les algorithmes que nous avons développés pour mettre en œuvre à l'approche décrite dans la section précédente, et nous détaillons ses différentes fonctions. Dans ces algorithmes, une trace correspond à une navigation sur le site.

L'algorithme 1 est destiné à l'exploration du site Web. La fonction $crawler$ permet de découvrir une partie d'un site en partant d'une navigation fournie en paramètre notée $path$, d_m étant la profondeur maximale de navigation à travers le site. La variable $remain$ contient l'ensemble des navigations qui viennent d'être découvertes et qui doivent donc être analysées. Cet algorithme prend fin lorsqu'il n'y a plus de navigations à analyser, autrement dit lorsque l'ensemble $remain$ est vide, ou lorsque la

10. Par soucis de simplification, seuls les liens html actifs par la requête sont mentionnés, sans indiquer explicitement les paramètres associés à ces requêtes



0	`display.php`	
1	`login.php`	
2	`index.html`	
3	`display.php`	vuln.
4	`login.php`	vuln.
5	`add.php`	
6	`delete.php`	
7	`check.php`	
8	`about.html`	
9	`check.php`	vuln.
10	`buy.php`	

Figure 8. Graphe de vulnérabilités final

Algorithm 1 $crawler(path, d_m)$

Require: $path, d_m$
Ensure: new_paths
1: $remain \leftarrow \{path\}$
2: $traces \leftarrow \emptyset$
3: $d \leftarrow |path|$
4: **while** $remain \neq \emptyset \wedge d \leq d_m$ **do**
5: $next \leftarrow \emptyset$
6: **for** $trace \in remain$ **do**
7: $free\_cookies()$
8: **for** $i \in 1 \ldots |trace|$ **do**
9: $links \leftarrow response(trace_i)$
10: **end for**
11: **for** $link \in links$ **do**
12: $next \leftarrow next \cup \{trace \oplus link\}$
13: $traces \leftarrow traces \cup \{trace \oplus link\}$
14: **end for**
15: **end for**
16: $remain \leftarrow next$
17: $d \leftarrow d + 1$
18: **end while**
19: **return** $traces_0$

profondeur d'exploration du site nommée d_m est atteinte. Avant l'analyse de chaque navigation, les cookies sont supprimés. Ensuite, la navigation est lancée requête après requête. Le contenu de la réponse associée à la dernière requête est analysé : il détermine les nouvelles requêtes exécutables. De nouvelles navigations, correspondant à la navigation en cours d'analyse enrichie de ces requêtes, devront à leur tour être analysées. Ces nouvelles navigations sont construites en utilisant l'opérateur $\oplus$ qui représente la concaténation d'une requête à une navigation. Elles sont ensuite stockées dans l'ensemble $remain$.

Algorithm 2 $search\_vulns(path)$

Require: $path$ = navigation
Ensure: $vulns$ = ensemble de navigations
1: $vulns \leftarrow \emptyset$
2: **for** $class \in classes$ **do**
3: $\quad vulns \leftarrow vulns \cup wasapy(path)$
4: **end for**
5: **return** $vulns$

L'algorithme 2 est destiné à la recherche des vulnérabilités. La fonction $search\_vulns$ utilise une navigation comme paramètre. L'objectif de cette fonction est d'analyser la dernière requête de cette navigation pour rechercher des vulnérabilités. L'analyse est réalisée avec la fonction $wasapy$, en considérant les différentes classes de vulnérabilités prises en compte. Le retour de cette fonction est une liste de navigations constituée chacune d'un ensemble de requêtes successives dont la dernière permet l'exploitation d'une des vulnérabilités identifiées.

La fonction $main$ de l'algorithme 3 permet d'enchaîner les deux fonctions précédentes. Lors de la première itération, les invocations de la fonction $crawler$ permettent de découvrir le site en ne considérant que les requêtes normales. Ensuite, les navigations obtenues sont condensées en un graphe avec l'algorithme RPNI. Chaque état de ce graphe fait alors l'objet d'une analyse de vulnérabilités avec la fonction $search\_vulns$. A l'issue de la première itération, nous disposons de toutes les navigations contenant au plus une vulnérabilité et finissant par cette vulnérabilité. Si des vulnérabilités ont effectivement été identifiées, leur exploitation peut éventuellement permettre de découvrir de nouvelles parties du site. L'itération suivante peut alors débuter. Le début de chaque itération i de la fonction $main$ correspond à l'exploration d'une sous-partie du site juste après l'exploitation de la $(i-1)$-ème vulnérabilité de la navigation. La fin de chaque itération i de la fonction $main$ correspond à la recherche de vulnérabilités en considérant des navigations contenant $i-1$ vulnérabilités et finissant par une requête normale. A l'issue de la i-ième itération, nous disposons de toutes les navigations contenant au plus i vulnérabilités. La fonction prend fin lorsque la profondeur maximale d'exploration du site est atteinte ou lorsque plus aucune vulnérabilité ne peut être identifiée.

Selon la complexité de l'application cible, le nombre de traces de navigation et la taille du graphe de navigation peuvent être importants. Nous avons effectué une étude de complexité des algorithmes présents ci-dessus dont les détails sont présents dans (Akrout, 2012). Ce point n'est pas développé dans cet article faute d'espace.

6. Expérimentations

Cette section présente les expérimentations que nous avons réalisées afin de valider et d'évaluer notre algorithme de détection de vulnérabilités. Nous avons considéré

Algorithm 3 $main(urls)$

Require: $urls$
Ensure: $(G = (S, N, R), vulns)$
1: $G \leftarrow RPNI(urls)$
2: $ntraces \leftarrow urls$
3: $traces \leftarrow urls$
4: $vulns \leftarrow \emptyset$
5: **while** $|ntraces| \neq 0$ **do**
6: **for** $nt \in ntraces$ **do**
7: **if** $|nt| < d_m$ **then**
8: $traces \leftarrow traces \cup crwl(nt, d_m)$
9: **end if**
10: **end for**
11: $G' \leftarrow RPNI(traces)$
12: $new\_nodes \leftarrow N' \setminus N$
13: $ntraces \leftarrow \emptyset$
14: **for** $nn \in new\_nodes$ **do**
15: $ptnn \leftarrow shortest\_path(R', S', nn)$
16: **if** $|ptnn| < d_m$ **then**
17: $nptv \leftarrow search\_vulns(ptnn)$
18: **for** $np \in nptv$ **do**
19: $new\_vuln \leftarrow np_{|np|}$
20: $vulns \leftarrow vulns \cup \{new\_vuln\}$
21: **end for**
22: $ntraces \leftarrow ntraces \cup nptv$
23: **end if**
24: **end for**
25: $G \leftarrow G'$
26: **end while**
27: **return** $(G, vulns)$

plusieurs applications en utilisant les scanners de vulnérabilités, *W3af 1.1*, *Skipfish 1.9.6b*, *Wapiti 2.2.1* et notre propre scanner de vulnérabilité *Wasapy* décrit dans la section 4.2. Notons qu'il n'y a pas eu de configuration particulière pour les différents outils utilisés sauf pour *Wasapy*, pour lequel nous avons défini le nombre de requêtes injectées par point d'injection à 30 pour chacune des classes de vulnérabilités testées.

Les expériences ont été exécutées sur une machine équipée du système *GNU/Linux* (noyau 2.6) exécutant plusieurs machines virtuelles grâce à l'utilitaire *VirtualBox*. Toutes les machines virtuelles exécutent le serveur web *Apache* 1.3.37 ou 2.2.8 avec *PHP* 4.0 ou 5,0 et *MySQL* 5 comme serveur de base de données.

Nous avons réalisé deux séries d'expériences. Dans la première série d'expériences, nous avons injecté volontairement et manuellement quelques vulnérabilités spéci-

fiques dans cinq applications open source et analysé les capacités de détection de notre algorithme par rapport aux trois outils explorés précédemment. Dans la deuxième série d'expériences, nous avons examiné cinq autres applications vulnérables sans les modifier. Ces expériences nous ont permis d'analyser l'efficacité de notre algorithme.

6.1. Notation

Les résultats de nos expériences sont présentés dans différents tableaux. Par souci de lisibilité, nous utilisons les notations et abréviations suivantes :

✓	La vulnérabilité a été détectée par le scanneur correspondant
✗	La vulnérabilité n'a pas été détectée par le scanneur correspondant
–	Le point d'injection n'a pas été testé par le scanner
SQL*i*	`Injection SQL`
XPa	`Injection XPATH`
OsC	`Os Commanding`
FIn	`File Include`
CVE	La référence CVE de la vulnérabilité considérée si elle existe
NR	La vulnérabilité correspondante n'a pas de référence CVE. Donc, soit elle a été détectée par un des outils testés, et pour vérifier qu'il ne s'agit pas d'un faux positif, nous l'avons testée manuellement, soit nous l'avons notée de la référence MOTH(AnantaSec, 2009) sur laquelle nous nous sommes basés pour certaines expérimentations.

– Une vulnérabilité est considérée comme détectée si le scanner génère une alerte pour cette vulnérabilité, quelle que soit la méthode utilisée pour la détecter.

– Une vulnérabilité est considérée comme non détectée si le scanner teste le point d'injection correspondant sans envoyer une alerte.

– Une vulnérabilité est considérée comme ignorée par le scanner si le point d'injection correspondant n'a pas été testé par le scanner.

Dans la suite nous considérons deux types d'applications avec lesquelles nous avons réalisé des expérimentations : des applications modifiées et des applications non modifiées.

6.2. Expériences avec les applications modifiées

6.2.1. Présentation des applications modifiées

Les cinq applications choisies pour cette première série d'expériences sont décrites dans la suite :

phpBB-3[11] Cette application est un gestionnaire de forum écrit en *PHP* 4 et utilisant une base de données *MySQL* version 4.x.x. Nous avons modifié le formulaire d'authentification de l'application de sorte qu'il inclut la vulnérabilité (`v1`) qui peut être exploitée par une injection SQL. Cette vulnérabilité permet à un attaquant d'atteindre l'accès restreint à l'espace administrateur du forum.

Secure Page[12] Cette application écrite en *PHP*, est conçue pour protéger l'accès d'un site Web grâce à l'authentification. Les couples valides pour cette authentification sont stockés dans une base de données *Mysql*. Une vulnérabilité (`v2`) similaire à (`v1`) a été volontairement injectée.

Hardware Store Nous avons nous-mêmes développé cette application en *PHP* 5.0. Cette application permet à un utilisateur de faire l'inventaire d'équipements informatiques dans une base de données et d'interroger cette base de données. L'utilisateur doit d'abord s'authentifier. Cinq vulnérabilités SQL ont été volontairement injectées dans cette application :

– La vulnérabilité (`v3`) : permet l'injection du code SQL dans un formulaire de recherche, et permet à un attaquant d'accéder à toute la base de données.

– La vulnérabilité (`v4`) : permet l'injection du code SQL dans le formulaire de l'authentification.

– La vulnérabilité (`v5`) : permet l'injection du code SQL dans un paramètre d'une requête HTML. Pour cette page HTML vulnérable, nous avons volontairement désactivé le message d'erreur renvoyé par le serveur, afin de comparer le comportement de W3af et Wapiti avec celui de *Wasapy* dans une telle situation.

– La vulnérabilité (`v6`) : est similaire à la vulnérabilité (`v4`), mais elle est utilisée dans un contexte différent : le message d'erreur renvoyé par le serveur est là-aussi, comme pour (`v5`), désactivé.

– La vulnérabilité (`v7`) : est particulière dans le sens où elle ne peut être exploitée qu'après l'exploitation réussie de la vulnérabilité (`v4`). En effet, cette vulnérabilité est incluse dans une page qui ne peut être consultée qu'après une authentification réussie sur l'application Web ou après un succès de contournement du mécanisme d'authentification (par l'exploitation de la vulnérabilité (`v4`)).

Les vulnérabilités de types XPATH, OS Commanding et File Include ont également été injectées dans cette application.

– La vulnérabilité (`v10`) : permet à un attaquant de contourner l'authentification grâce à une injection XPATH dans la page d'authentification.

– La vulnérabilité (`v11`) : est une vulnérabilité Os commanding qui ne peut être exploitée qu'après exploitation de la vulnérabilité (`v4`). En effet, cette vulnérabilité est incluse dans une page qui est uniquement accessible après authentification (ou

11. http://www.phpbb.com
12. http://www.01php.com/fiche-scripts-126.html

contournement de l'authentification grâce au succès de l'exploitation de (v4)).

– La vulnérabilité (v12) : est une vulnérabilité File Include, elle est incluse dans la même page que (v11) et peut être exploitée dans les mêmes conditions que (v11).

Insecure Cette application a été développée en Ruby on Rails dans le cadre du projet DALI. Il s'agit d'un site de commerce électronique où plusieurs vulnérabilités ont été injectées volontairement. Une vulnérabilité (v8), qui permet à un attaquant d'injecter du code SQL, a été délibérément incluse dans le formulaire d'authentification de l'application. Cette vulnérabilité, équivalente fonctionnellement à (v4), est différente parce que *Insecure* est implémentée en Ruby et les messages d'erreur sont différents de ceux d'Apache.

Damn Vulnerable Web Application (DVWA)[13] Cette application est écrite en PHP et utilise un serveur *MySQL*. Une vulnérabilité (v9), similaire à (v3), a été volontairement introduite dans l'application.

6.2.2. Résultats

Le tableau 1 présente les résultats de détection des vulnérabilités considérées par les différents outils. On peut observer que les performances de *W3af* et *Wapiti* sont similaires en moyenne, même si les vulnérabilités détectées ne sont pas les mêmes (Wapiti détecte avec succès v1 et v2 alors que *W3af* ne les détecte pas; d'autre part, *W3af* détecte v4 et v8 alors que Wapiti ne les détecte pas). Ce résultat est conforme avec le fait que les deux scanners utilisent un algorithme de reconnaissance de messages d'erreurs. Les variations observées sont liées à la génération de différentes requêtes par ces outils. *Wasapy*, quant à lui, permet de détecter toutes ces vulnérabilités. Cela confirme que l'algorithme de classification et de regroupement de pages Web pour la détection de vulnérabilités présente une meilleure couverture que les outils basés sur la reconnaissance de messages d'erreur pour ces classes de vulnérabilité.

En ce qui concerne les vulnérabilités v1 et v2, nous avons vérifié manuellement les injections réalisées par Skipfish (' ", ' " et ' ") et stocké les réponses correspondantes (respectivement A, B et C). Skipfish considère que les pages A et C doivent être différentes de sorte que la vulnérabilité soit présente. Malheureusement, pour ces deux points d'injection, ce n'est pas le cas. Les messages d'erreur SQL renvoyés par le serveur sont très similaires. Par conséquent, Skipfish ne peut pas détecter ces vulnérabilités.

En ce qui concerne les vulnérabilités v5 et v6, elles sont incluses dans des pages PHP pour lesquelles nous avons volontairement désactivé la fonction de notification de message d'erreur dans le fichier de configuration de PHP5. Dans ce cas particulier, aucun des trois scanners (*Skipfish*, *W3af* et *Wapiti*) n'a été capable de détecter les vulnérabilités.

13. http://www.dvwa.co.uk

Table 1. Résultats de détection de vulnérabilités pour les applications modifiées

Vulnérabilités			Scanners			
Type	Application	*ID*	Skipfish	W3af	Wapiti	Wasapy
SQL*i*	phpBB3	v1	✗	✗	✓	✓
	SecurePages	v2	✗	✗	✓	✓
	HardwareStore	v3	✓	✓	✓	✓
		v4	✓	✓	✗	✓
		v5	✗	✗	✗	✓
		v6	✗	✗	✗	✓
		v7	–	–	–	✓
	Insecure	v8	✓	✓	✗	✓
	DVWA	v9	✓	✓	–	✓
XPa	HardwareStore	v10	✗	✗	✗	✓
OsC	HardwareStore	v11	–	–	–	✓
FIn	HardwareStore	v12	–	–	–	✓
Nombre de détections			5	4	3	12

En ce qui concerne la vulnérabilité v7, *Wasapy* est le seul scanner qui est capable de la détecter. Par ailleurs, c'est le seul qui a été capable d'identifier le point d'injection correspondant. En effet, ce point d'injection est inclus dans une page HTML qui n'est accessible qu'après une authentification réussie ou après l'exploitation réussie de la vulnérabilité v4. *Wasapy* est le seul à pouvoir exploiter automatiquement v4, et ainsi accéder à la page comprenant la vulnérabilité v7. Pour les autres scanners, il est nécessaire d'effectuer une exploitation manuelle de v4 afin de pouvoir accéder à cette page.
Les vulnérabilités v11 et v12 ont été identifiées seulement par notre outil pour les mêmes raisons : elles restent masquées jusqu'à ce que l'authentification soit contournée.

Le but de ces premiers tests était l'étalonnage de *Wasapy*. Ils nous ont notamment permis de tester les grammaires que nous avons élaborées pour la génération automatique de requêtes. Bien sûr, les vulnérabilités correspondantes ont été identifiées dans ce but. Ainsi, ces résultats ne sont pas destinés à être utilisés pour faire une comparaison absolue entre les scanners.
Une évaluation plus représentative des différents outils doit être réalisée sur des applications dans lesquelles les vulnérabilités n'ont pas été injectées par nous-mêmes. Ces expériences sont présentées dans la section suivante.

6.3. Expériences avec les applications non modifiées

Cette deuxième série d'expériences nous a permis d'avoir une idée plus précise de la couverture de notre algorithme de détection. À cette fin, nous l'avons comparé aux algorithmes de détection de *Skipfish*, *W3af* et *Wapiti* en considérant des applications Web vulnérables que nous n'avons pas modifiées. Pour certaines de ces applications, nous avons pu comparer notre algorithme avec certains scanners de vulnérabilités commerciaux, en considérant des résultats disponibles dans MOTH. Dans ce docu-

ment, l'auteur présente les résultats de détection des vulnérabilités obtenus avec trois scanners commerciaux : *WebInspect*[14] de HP, *AppScan*[15] d'IBM et *Web Vulnerability Scanner*(WVS)[16] de Acunetix.

6.3.1. Présentation des applications non modifiées

Pour nos expériences, nous avons sélectionné cinq applications Web (la plupart d'entre elles testées dans MOTH, connues pour contenir des vulnérabilités). Ces applications couvrent différentes fonctionnalités et des contextes différents.
Nous avons installé ces applications, et effectué des tests de détection de vulnérabilité sans les modifier.

Cyphor[17] Cette application implémente un forum de discussion, elle se base sur la notion de session dans PHP 4 pour authentifier les utilisateurs et sur une base de données MySQL pour stocker les données de l'utilisateur.
Seuls les utilisateurs enregistrés peuvent poster des messages. Ces utilisateurs sont répartis en utilisateurs normaux, les modérateurs et administrateurs. Les modérateurs et administrateurs peuvent supprimer les discussions, les administrateurs peuvent modifier les paramètres d'administration, créer de nouveaux forums, etc.

Seagull[18] Cette application est un framework orienté objet (OOP : Object-oriented programming) pour développer des applications Web, des sites de commerce en ligne et des interfaces graphiques. Elle permet aux développeurs PHP d'intégrer et de gérer leur code source. Cette application nécessite la configuration suivante : *PHP* 4.3.0 ou plus récent, *MySQL* 4.0.x ou plus récent, `Apache` 1.3.x ou 2.x.

Fttss[19] Cette application a été développée dans le cadre d'un projet de recherche qui porte sur la mise en œuvre d'un système Text-To-Speech permettant de transformer un texte écrit en un texte parlé. *PHP* (4.3 ou plus récent) et *MySQL* (4.1.2 ou plus récent) sont nécessaires pour l'exécution de cette application.

Riotpix[20] Cette application est un forum de discussion en source libre où les internautes ont la possibilité de s'inscrire et poster des message. *PHP* (4.3 ou plus récent) et *MySQL* (4.1.2 ou plus récent) sont nécessaires.

Pligg[21] *Pligg* est une application de gestion de contenu (CMS en anglais pour Content Management System) en source libre. Elle permet de créer un site Internet communautaire où les contenus sont créés et votés par les membres inscrits. Les nouveaux articles soumis par les utilisateurs sont ainsi notés par d'autres utilisateurs et

14. http://www.web-inspect.com
15. http://www-01.ibm.com/software/awdtools/appscan/
16. http://www.acunetix.com/vulnerability-scanner/
17. http://webscripts.softpedia.com/script/Snippets/Cyphor-27985.html
18. http://seagullproject.org/
19. http://fttss.sourceforge.net
20. http://www.riotpix.com/
21. http://www.pligg.com/

affichés en page d'accueil s'ils remportent le succès nécessaire. Le *CMS Pligg* fournit un logiciel d'édition qui permet aux visiteurs de s'inscrire sur le site créé afin qu'ils puissent soumettre des contenus et communiquer avec d'autres utilisateurs. *PHP* (4.3 ou plus récent) et *MySQL* (4.1.2 ou plus récent) sont nécessaires.

6.3.2. Résultats

Nous avons inspecté manuellement toutes les vulnérabilités détectées par chaque scanner, afin d'avoir un contrôle précis sur les résultats de l'expérimentation et obtenir une plus grande confiance sur le nombre de vulnérabilités détectées et les faux positifs.

La figure 2 présente les résultats obtenus pour l'application *Cyphor*. Tous les scanners trouvent toutes les vulnérabilités parce que les messages d'erreur ne sont pas désactivés. Ainsi, il est facile d'identifier l'exploitation réussie des vulnérabilités des messages d'erreur. Nous pouvons remarquer que certains résultats ont été soulignés. Ils correspondent à des détections rendues possibles en fournissant le couple valide (login / mot de passe) afin que les scanners puissent réussir l'authentification à l'application. En d'autres termes, la vulnérabilité correspondante est visible uniquement lorsque l'utilisateur est connecté sur le site. Nous remarquons également qu'il y a un faux positif détecté par *Skipfish*.

Table 2. Résultats de détection de vulnérabilités pour l'application Cyphor

Vulnérabilités			Skipfish	W3af	Wapiti	Wasapy
Type	CVE	Emplacement				
SQL*i*	*NR*	search.php	✓	✓	✓	✓
	2005-3236	lostpwd.php	✓	✓	✓	✓
	2005-3236	newmsg.php	✓	✓	✓	✓
	2005-3575	show.php	✓	✓	✓	✓
Faux positif			1	0	0	0

L'application suivante testée est *Seagull*. Les résultats correspondants sont présentés dans le tableau 3. *Wasapy* est le seul qui signale une vulnérabilité dans cette application. Aucune vulnérabilité n'a été détectée par les autres scanners car l'application désactive l'envoi des messages d'erreurs vers les clients. Concernant les vulnérabilités File Include, les points d'injection qui permettent leur exploitation ne sont pas directement accessibles par le client. Ainsi, le code source est nécessaire pour identifier ces vulnérabilités. C'est ce qui explique l'échec de tous les scanners.

Table 3. Résultats de détection de vulnérabilités pour l'application Seagull

Vulnérabilité			Skipfish	W3af	Wapiti	Wasapy
Type	CVE	Emplacement				
SQL*i*	*2010-3212*	index.php	✗	✗	✗	✓
FIn	*2010-3209*	container.php	✗	✗	✗	✗
	2010-3209	QuickForm.php	✗	✗	✗	✗
	2010-3209	NestedSet.php	✗	✗	✗	✗
	2010-3209	Output.php	✗	✗	✗	✗
Faux positif			0	0	0	0

Fttss est une application qui a été testée dans MOTH. Ainsi, certains résultats associés aux trois scanners commerciaux considérés sont disponibles (cf. tableau 4). Les scanners commerciaux ne détectent pas la vulnérabilité Os Commanding, qui est la seule vulnérabilité connue dans cette application. En revanche, *W3af* et *Wasapy* sont en mesure d'identifier cette vulnérabilité. *Skipfish* et *Wapiti* ne la détectent pas parce que l'application ne signale aucun message d'erreur. Il est à noter qu'aucun des scanners testés ne génère de faux positifs dans ce cas.

Table 4. Résultats de détection de vulnérabilités pour l'application Fttss

Vulnérabilité			Skipfish	W3af	Wapiti	Wasapy	AppScan	WebIns.	Acun.
Type	CVE	Emplacement							
OsC	*NR*	index.php	✗	✓	✗	✓	✗	✗	✗
Faux positif			0	0	0	0	0	0	0

En ce qui concerne *Riotpix* (cf. tableau 5), les résultats sont similaires à ceux de *Cyphor*. Les vulnérabilités ne sont accessibles qu'après l'authentification réussie des utilisateurs. Une vulnérabilité n'a été découverte par aucun scanner. Elle correspond à l'injection de code dans les variables qui ne sont pas accessibles par le client et donc qui ne peuvent pas être découvertes par les scanners (il serait nécessaire d'effectuer une analyse du code source, c'est-à-dire une approche en boîte blanche).

Table 5. Résultats de détection de vulnérabilités pour l'application Riotpix

Vulnérabilité			Skipfish	W3af	Wapiti	Wasapy	AppScan	WebIns.	Acun.
Type	CVE	Emplacement							
SQL*i*	*NR*	edit_post.php	✗	✗	✗	✓	✗	✗	✗
	NR	edit_post_script.php	✗	✗	✗	✗	✗	✗	✗
	NR	index.php	✗	✗	✗	✓	✗	✗	✗
	NR	message.php	✗	✗	✗	✓	✗	✗	✗
	NR	reader.php	✓	✓	✗	✓	✗	✗	✗
Faux positif			0	0	0	0	0	0	0

La dernière application que nous avons testée est *Pligg* (cf. tableau 6). Dans cette application, toutes les vulnérabilités, sauf les deux premières, ne sont pas directement accessibles car les points d'injection correspondants sont cachés. Le scanner doit être averti de la présence de ce point d'injection afin de tester la vulnérabilité. Ceci explique les faux négatifs. Pour les deux premières vulnérabilités, *Wasapy* les a découvertes, alors que les autres scanners trouvent seulement une de ces vulnérabilités. Cela est dû au fait que les messages d'erreur sont désactivés par l'application.

6.4. Synthèse

La synthèse de toutes nos expériences est résumée dans le tableau 7. Ces expériences montrent que :

– *Wasapy* est un scanner efficace, surtout dans des conditions particulières pour lesquelles il a été conçu : 1) il est plus efficace que les autres scanners en source libre

Table 6. Résultats de détection de vulnérabilités pour l'application Pligg

Vulnérabilité			Skipfish	W3af	Wapiti	Wasapy	AppScan	WebIns.	Acun.
Type	CVE	Emplacement							
SQL*i*	*2008-7091*	login.php	✗	✗	✗	✓	✗	✓	✗
	2008-7091	story.php	✓	✗	✓	✓	✓	✓	✓
	NR	userrss.php	✗	✗	✗	✗	✓	✓	✓
	2008-7091	out.php	✗	✗	✗	✗	✓	✗	✓
	2008-7091	trackback.php	✗	✗	✗	✗	✗	✗	✗
	2008-7091	cloud.php	✗	✗	✗	✗	✗	✗	✗
	2008-7091	cvote.php	✗	✗	✗	✗	✗	✗	✗
	2008-7091	recommend.php	✗	✗	✗	✗	✗	✗	✗
	2008-7091	submit.php	✗	✗	✗	✗	✗	✗	✗
	2008-7091	vote.php	✗	✗	✗	✗	✗	✗	✗
	2008-7091	edit.php	✗	✗	✗	✗	✗	✗	✗
False positive			0	0	0	2	1	1	0

testés lorsque les notifications d'erreur sont désactivés, 2) il est plus efficace que les autres scanners pour découvrir et exploiter les vulnérabilités qui sont incluses dans les pages qui ne sont pas directement accessibles (pages qui nécessitent l'exploitation réussie d'une vulnérabilité pour être consultées). En effet, notre scanner est le seul qui est capable d'exploiter la vulnérabilité et fournir les requêtes exactes pour les injections correspondantes.

– *Wasapy* est globalement aussi efficace que les autres scanners de vulnérabilités testés sur les applications web vulnérables non modifiées.

– Notre algorithme de "clustering" peut être facilement adapté à différents types de vulnérabilités. En effet, même s'il a été conçu pour les injections SQL, les résultats des expériences montrent que *Wasapy* détecte également les vulnérabilités XPATH, OS commanding et File Include et qu'il est au moins aussi efficace que les autres scanners de vulnérabilités.

Table 7. Résumé des résultats

	Skipfish	W3af	Wapiti	Wasapy
`Vrais Positifs`	6	6	5	12
`Faux Positifs`	1	0	0	3
`Faux Negatifs`	20	20	21	14

7. Conclusion

Dans cet article, nous avons proposé une nouvelle méthodologie qui permet d'une part d'identifier les vulnérabilités des applications Web et d'autre part d'exhiber des scénarios d'attaque ciblant cette application. Cette méthodologie est basée sur l'analyse dynamique de l'application selon une approche boîte noire. Elle vise également à réduire le nombre de faux positifs des vulnérabilités détectées en fournissant les requêtes qui ont réellement permis d'exploiter chaque vulnérabilité. Cet avantage est double

puisque l'exploitation effective des vulnérabilités nous permet également de découvrir de nouvelles pages de l'application Web que nous ne pouvions atteindre auparavant. Ces nouvelles pages peuvent contenir de nouveaux points d'injection et éventuellement de nouvelles vulnérabilités. Afin de valider et d'évaluer notre approche, nous avons réalisé deux séries d'expériences, sur différents types d'applications.

Nous pouvons envisager différentes perspectives à nos travaux de recherche. Tout d'abord, en ce qui concerne l'approche proposée pour la détection de vulnérabilités et la génération de scénarios d'attaques basée sur l'élaboration du graphe de navigation d'un site Web, des optimisations peuvent s'avérer nécessaires afin de maîtriser la taille de ce graphe, en particulier quand il s'agit d'appliquer cette approche à des sites Web complexes. Une autre perspective intéressante consiste à enrichir les grammaires de génération des requêtes de façon à être capable de générer la plus grande variété possible d'attaques pour les injections déjà étudiées

References

K.Stefan, E. Kirda, C. Kruegel, N. Jovanovic,"SecuBat: a web vulnerability scanner", *Proc. of the 15th Intl. Conf. on World Wide Web (WWW '06)*, Edinburgh, Scotland, 2006

W.G.J.Halfond, J.Viegas, A.Orso. "A Classification of SQL Injection Attacks and Countermeasures", *Proc. of the International Symposium on Secure Software Engineering*, 2006.

Y.-W Huang, S.-K Huang, T.-P. Lin, C.-H.Tsai, "Web Application security assessment by fault injection and behavioral monitoring", *Proc. 12th Int. Conf. on World Wide Web (WWW'03)*, Budapest, Hungary, 2003.

IBM X-Force 2012 Mid-year Trend and Risk Report, September 2012, http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA &subtype=WH&htmlfid=WGL03014USEN

J.Fonseca, M.Vieira, H.Madeira, "Testing and Comparing Web vulnerability scanning tools for SQL injections and XSS attacks", *Proc. IEEE Symposium Pacific Rim Dependable Computing (PRDC'07)*, Victoria, Australia, pp. 330-337, USA, 2007

J. Bau, E. Bursztein, D. Gupta, J. Mitchell, "State of the art: Automated black-box web application vulnerability testing", *Proc. 2010 IEEE Symposium on Security and Privacy*, Oakland, USA, 2010.

A. Doupé, M. Cova, G. Vigna, "Why Johnny can't pentest : An analysis of black-box web vulnerability scanners", *Proc. DIMVA 2010*.

AnantaSec: Web Vulnerability Scanners Evaluation (January 2009), http://anantasec.blogspot.fr/

Dupont, P., "Incremental regular inference", Proc. of the Fourth Intl. Colloquium on Grammatical Inference and Applications (ICGI '96), pp 222-237, 1996.

V.Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals", Soviet Physics Doklady, pp. 707-710, 1966.

J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison", Tech. Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

S. C. Johnson, “Hierarchical Clustering Schemes”, *in Psychometrika Journal*, pp. 241-254, Vol.2, 1967.

A.Kiezun, P. J. Guo, K.Jayaraman and M. D. Ernst, “Automatic creation of SQL Injection and cross-site scripting attacks”, *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on Vancouver*, BC, 2009.

Akrout, R., "Web Applications Vulnerability Analysis and Intrusion Detection Systems Assessment", PhD Thesis, University of Toulouse, October 2012 (in French), http://homepages.laas.fr/rakrout/PhD_Thesis.pdf.