2011 17th IEEE Pacific Rim International Symposium on Dependable Computing

A clustering approach for web vulnerabilities detection

A. Dessiatnikoff R. Akrout E. Alata M. Kaâniche V. Nicomette

CNRS; LAAS ; 7 avenue du Colonel Roche, F-31077 Toulouse, France
Université de Toulouse; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France
Email: {adessiat, rakrout, ealata, kaaniche, nicomett}@laas.fr

***Abstract*—This paper presents a new algorithm aimed at the vulnerability assessment of web applications following a black-box approach. The objective is to improve the detection efficiency of existing vulnerability scanners and to move a step forward toward the automation of this process. Our approach covers various types of vulnerabilities but this paper mainly focuses on SQL injections. The proposed algorithm is based on the automatic classification of the responses returned by the web servers using data clustering techniques and provides especially crafted inputs that lead to successful attacks when vulnerabilities are present. Experimental results on several vulnerable applications and comparative analysis with some existing tools confirm the effectiveness of our approach.**

***Index Terms*—Web security assessment; security scanners; Web vulnerabilities; clustering; dynamic analysis**

I. Introduction

Web application vulnerabilities have become in the recent years a major threat to computer systems security. This is illustrated by the 2010 CWE/SANS top 25 most dangerous software errors report published by MITRE in which the two top positions are taken by web application vulnerabilities (Cross-Site Scripting and SQL injections) [1]. This situation can be explained e.g., by the increase in complexity of web technologies and their frequent evolution, the short development cycles of web applications during which testing and validation activities are limited, and also, in some cases, by the lack of security skills and culture of the developers.

Web applications vulnerabilities can be exploited by the attackers to gain unauthorized access, obtain or modify sensitive data or even perform denial of service attacks. To cope with these threats, several solutions have been developed to prevent, detect or tolerate potential intrusions. Such techniques can be used during the development phase and also during the operation phase. As an example, we can list the following techniques.

- Static analysis of the source code [2], [3], [4], [5], [6], [7] allowing the identification of vulnerabilities before web application deployment.
- Detection and possible sanitization at runtime of malicious requests before they reach the server. The corresponding tools can run on the server [8], [9], or between the client and the server acting as a proxy [10]. Application-level firewalls can be included in the latter category.
- The security testing of web applications based on the use of black-box security scanners [11]. These tools consist in crawling the target application to identify reachable pages and possible input vectors, and generate specially crafted inputs to determine the presence of vulnerabilities.

In this paper, we focus on web vulnerability scanners.

A large number of vulnerability scanners have been developed, including commercial tools such as Acunetix WVS, WebInspect and AppScan [12], open source tools such as W3af, Wapiti, Nikto, and other publicly available tools such as Secubat [11]. Given the high complexity of current web sites that include a large number of pages to be analyzed, these tools provide a useful support for the identification of vulnerabilities in such systems. Recent analysis focused on the assessment of such tools have pointed out the need to improve their detection effectiveness, and to enhance the automation capabilities offered by such tools ([13], [14], [15]). Indeed, depending on the target application, current tools may exhibit a significant number of false positives and false negatives. These tools are designed to provide the request aimed at revealing the possible existence of vulnerabilities. They are not designed to automatically provide the specific requests allowing the exploitation of the identified vulnerabilities.

Clearly there is still room for improving the capabilities of web scanners. The research summarized in this paper is aimed at contributing to fulfilling this goal. The first step in our approach to achieve this objective consists in: i) analyzing the main principles behind the vulnerability detection algorithms implemented in recently developed open-source web scanners (such as Skipfish, W3af, and Wapiti) and then ii) identifying possible rooms for improving the detection ability of these algorithms to address some of the limitations. Such analysis cannot be based on commercial or other tools for which the source code is not available. In the second step, we developed a new algorithm allowing the automated detection of different types of web vulnerabilities including SQL injections, OS commanding, File Include, XPath, etc. The proposed algorithm builds on some of the concepts inherited from existing tools and includes significant extensions. It is based on the automatic classification of the responses returned by the web servers using data clustering techniques and provides specially crafted inputs that should lead to successful attacks when vulnerabilities are present. The automatic generation

978-0-7695-4590-5/11 $26.00 © 2011 IEEE
DOI 10.1109/PRDC.2011.31

of requests allowing the successful exploitation of detected vulnerabilities should be useful to facilitate web applications validation and penetration testing. In order to assess our algorithm, we have run two sets of experiments. In the first experiments, we have injected specific vulnerabilities in five open-source applications and analyzed the detection ability of the new algorithm compared to the algorithms used by Skipfish, W3af and Wapiti. In the second set of experiments, we have considered five other vulnerable applications without modifying them. These experiments allowed us to illustrate the potential benefits of our proposed algorithm.

The paper is structured into 8 sections. Section II briefly describes the principle of vulnerability scanners. Section III discusses related work focussing on the analysis of the vulnerability detection algorithm used by several well-known freeware vulnerability scanners and shows some weaknesses of these tools that we aimed to address by proposing a new algorithm. Section IV presents our clustering algorithm for detecting web application vulnerabilities. The principle of this algorithm is presented as well as examples for SQL injections. The application of this algorithm to other vulnerabilities is discussed in Section V. Section VI briefly presents the scanner we developed to implement this algorithm, called WASAPY. We describe in Section VII the experiments performed in order to validate our approach and assess the efficiency of our scanner. Finally Section VIII concludes this paper and discusses future work.

II. Vulnerability scanners principles

Most frequent attacks on web servers include SQL injection attacks (for web servers connected to an SQL database) and code injection attacks (Flash, Javascript, etc., carried out through so-called *Cross Site Scripting* or XSS attacks). These attacks generally correspond to the exploitation of the same kind of vulnerability related to the lack of sanitization of URL parameters or of HTML form inputs. In the following, we will focus on SQL injection attacks.

To check whether SQL injection attacks are possible, the vulnerability scanners send specially crafted requests and analyze the responses returned by the server. A server may respond with a *rejection* page or with an *execution* page. A *rejection* page corresponds to the detection of syntactically incorrect or invalid inputs. An *execution* page is returned by the server as a consequence of a successful execution of the request. This page may correspond to the "normal" scenario, i.e., in the case of a legitimate use of the web site, but may also result from a successful exploitation of an injection attack. These latter requests are those we consider in this paper, as our objective is to identify the vulnerabilities that can be successfully exploited by the attackers. For instance, the successful exploitation of a SQL injection vulnerability in a login form may lead to bypass an authentication, and the successful exploitation of a file include vulnerability on a search form may lead to display extra data like `/etc/passwd` file content. In order to identify the vulnerabilities of a web site, the scanners generally send specially crafted requests via the identified injection points allowing them to determine whether the input parameters submitted to the target system are sanitized or not. The identification of potential vulnerabilities is generally based on the identification of the *rejection* pages.

The issue is thus the analysis of the responses to determine if they actually correspond to *rejection* or *execution* pages. Two main approaches can be identified based on the analysis of related work on this topic. These approaches are discussed in the next section. In the following, we denote as *false positive* the fact that a vulnerability scanner detects a vulnerability in a web page while this vulnerability does not exist. A *false negative* occurs when the vulnerability scanner does not detect a vulnerability in a web page while it actually exists.

III. Related work

Two main approaches exist to detect the presence of a vulnerability in a web application. The first one relies on an error pattern matching algorithm and is presented in section III-A. The second one relies on the analysis of similarities between the pages returned by the server and is presented in section III-B. The last section III-C proposes a discussion regarding the limits of these approaches.

A. Error pattern matching approach

To identify SQL injections, this approach consists in sending crafted requests to the application and looking for specific patterns in the responses: database error messages. The basic idea is that the presence of an SQL error message in a HTML response page means that the corresponding request has not been sanitized by the application. Therefore, the fact that this request has been sent unchanged to the SQL server reveals the presence of a vulnerability. Scanners such as `W3af`[1] (`sqli` module), `Wapiti`[2] and `Secubat`[11] adopt such approach. As an example, to detect injection vulnerabilities in authentication forms, the `sqli` module of `W3af`, sends three requests based on the SQL injection: `d'z"0` (or `d%2Cz%220` encoded in ASCII). The three corresponding responses are then analyzed. If they include SQL error messages (e.g. `Mysql_` and `supplied argument is not a valid Mysql`), `W3af` informs the user that the application is vulnerable.

The list of keywords adopted by `Secubat` for the error pattern matching approach is presented in [11]. This list, derived by analyzing reponse pages of vulnerable web sites, is aimed to cover a wide range of error reponses and a variety of database servers. A confidence factor that measures the level of confidence that the attacked web form is vulnerable is also assigned to each keyword.

B. Similarity approach

This approach relies on three assumptions: 1) *execution* and *rejection* pages are different, 2) it is easy to build requests that generate *rejection* pages (by generating random or syntactically incorrect requests for instance) and 3) it is

[1] http://w3af.sourceforge.net
[2] http://wapiti.sourceforge.net

difficult to build requests including injection attacks that actually generate *execution* pages (i.e., requests that successfully exploit a vulnerability). The principle of the approach consists in sending different crafted requests to the web application and comparing the similarity of the corresponding responses using a textual distance, in order to identify *execution* pages among the response pages (i.e., pages that correspond to the successful execution of an injection attack).

Let us consider as an example the approach adopted by `Skipfish`[3] for detecting SQL injection vulnerabilities. It sends three requests to the web application (*A*- `'"`, *B*- `\'\"` and *C*- `\\'\\"`). The responses are compared two by two. According to `Skipfish`, a vulnerability is present if both responses associated to B and C are not similar to the response associated to A. The similarity test uses a distance based on the frequency of the words in the response pages.

The algorithm presented in [16] is also based on the similarity approach. It differs from other implementations in the additional use of the error pattern matching approach at a first step to guide the classification. The similarity approach is used to address the uncertainty that arises about the presence or absence of a vulnerability when an injection does not generate an error message.

Another example illustrating the use of the similarity approach in another context is the `formAuthBrute` module of `W3af`, that deals with weak passwords identification. This module first sends two requests including randomly generated usernames and passwords. The responses likely correspond to *rejection* pages. They are used as reference pages. Then, the dictionary attack is carried out in the second step. A large number of requests is generated based on a dictionary. Each response is compared to the reference pages, using the Levenshtein distance [17]. The response is assumed to correspond to an *execution* page if the distances between this page and reference pages are higher than 0.65. Otherwise, this page is considered as a *rejection* page. The threshold value 0.65 is empirically configured by the developer.

C. Discussion

The assumption used by the error pattern matching approach is questionable. Indeed, error messages that are included in HTML response pages do not necessarily come from the database server itself. A database related error message may also has been generated by the application. Moreover, even if the message is actually generated by the database server, this is not sufficient to affirm when receiving this message, that an SQL injection is possible. Indeed, this message means that, for this particular request, the inputs have not been sanitized. But it does not mean that the SQL server does not sanitize all the SQL requests.

Regarding the similarity approach, it is more easy to generate random or syntactically incorrect requests to obtain *rejection* pages than to send valid injection attacks to obtain *execution* pages. Since the main assumption is based on the observation that the content of a *rejection* page is generally different from the content of an *execution* page, it is important to ensure a wide coverage of the different types of rejection pages that could be generated by the application. This can be achieved by generating a large number of requests aimed at activating different types of error pages. However, the existing implementations of this approach, especially in `Skipfish`, generate too few requests. `Skipfish` uses only 3 requests. But, if the responses correspond to different *rejection* pages, it will wrongly conclude that a vulnerability is present.

Also, as in any classification problem, the choice of the distance is very important. The one used in `Skipfish` doesn't take into account the order of the words in a text. However, this order generally defines the semantics of the page. Thus it is important to take it into account to assess the similarity, as performed in [16] with a text similarity distance. As an example, the two following pages use the same words in a different order, but they have a different semantics:

```
- You are authenticated, you have not entered a
wrong login.
- You are not authenticated, you have entered a
wrong login.
```

The algorithm presented in Section IV builds on some of the concepts of the similarity approach and is aimed at addressing the issues raised in the discussion above. In particular, it allows: i) the generation of a large number of requests, that can be tuned by the user, to activate different *rejection* pages returned by the application, ii) the automatic generation of various types of specially crafted requests using a grammar and iii) the automatic clustering of the corresponding HTML pages returned by the web server to distinguish between *rejection* pages and *execution* pages and automatically identify successful injections. This algorithm is also designed to identify and successfully exploit various types of vulnerabilities. Besides SQL injections, the proposed approach can address XPATH, OS Commanding and File Include vulnerabilities. Regarding related work, while our work shares some of the ideas and objectives presented in [16], we follow different approaches. For example, the clustering approach which is the core of our algorithm is not used in [16]. They use a different technique combining error pattern matching and the similarity analysis of application response pages. Also, their approach is firstly based on error pattern matching and hence shares the same concerns raised above.

IV. The proposed algorithm

Let us first define the notion of *injection point*, which is used by our algorithm. An injection point is a piece of a Web page into which a code can be injected: a parameter in the URL or a field of a form, etc. In the following, we consider an authentication form for the sake of illustration and we focus on SQL injections. In subsection IV-A, we present the clustering algorithm used to classify pages. In subsection IV-B, we discuss the distance used to parameterize the algorithm. Finally, we explain in subsection IV-C how we generate the requests sent to the web server.

[3]http://code.google.com/p/skipfish

A. Pages clustering

Our goal is to identify, among several SQL injections, those which allow an attacker to bypass the authentication. The main challenge lies in the automation of this process. Our approach relies on the following assumptions: *a*) the content of an *execution* page is far different from the content of a *rejection* page, *b*) two *rejection* pages may be different from each other and *c*) two *execution* pages may also be different from each other. For example, let us consider a login page. Responses to valid requests include welcome messages and responses to invalid requests include PHP or SQL error messages. The essential point is the existence of differences between *execution* pages and *rejection* pages.

Our approach focuses on the analysis of these differences. The objective is to identify, among several responses, those which correspond to *execution* pages. In other words, we learn the behavior of the application based on the clustering of web server response pages that are similar enough. The entry point of our algorithm is a set of initial requests that have a common property: it is easy to classify the associated responses (either *execution* page or *rejection* page). Obviously, it is easier to generate requests, which generate *rejection* pages than requests that generate *execution* pages. To generate *rejection* pages, we can, for example, use random usernames and passwords to fill the authentication form. We can also easily generate requests which correspond to malformed SQL injections that would lead to error messages.

In our proposal, we distinguish three sets of requests:

R_{r} is the set of requests generated from words randomly chosen from the list [a-zA-Z0-9]+. They are very likely to generate *rejection* pages. For example:

```
http://address/directory/page.php?
        login=ABCDEF&pass=ABCDEF
```

R_{ii} is the set of SQL injection requests inappropriate for the given *injection point*. They are constructed to produce a syntax error in the SQL query sent to the SQL server by the HTTP server. Usually, these requests are composed of an odd number of quotes. They are also very likely to generate *rejection* pages. For example:

```
http://address/directory/page.php?
        login=’’’&pass=’’’
```

R_{vi} is the set of SQL injection requests that are constructed to generate *execution* pages in the presence of vulnerabilities, but they might as well generate *rejection* pages in the absence of vulnerabilities. For example:

```
http://address/directory/page.php?
        login=test&pass=’ or ’1’=’1
```

The main issue is to determine whether the response is a *rejection* page or an *execution* page. To do so, these responses are compared to those associated to sets R_{r} and R_{ii}.

We note S_{r}, S_{ii} and S_{vi} the responses associated to R_{r}, R_{ii} and R_{vi} respectively. The principle of our algorithm is then as follows: R_{vi} requests whose responses are not similar to any of the responses from S_{ii} and S_{r} are considered valid SQL injections. To assess the similarity between the pages returned by different requests, we use a classification technique based on the distance presented in the next subsection.

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

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

Fig. 1. Distance for clustering

B. Distance for response pages similarity assessment

As discussed in Section III-C, it is important to take into account the position of the words in a text when analyzing the similarity between two pages. Thus, to compute the distance between two pages, we use the normalized difference, based on a slightly modified version of the Unix diff operator [18]. Let a and b be two responses of length n and m. We also denote a_i and b_j the i-th character in a and the j-th character in b. The distance is defined in Figure 1.

A threshold is required to determine whether two responses are similar or not. This threshold may vary from one *injection point* to another. Indeed, it depends on the size of responses and the amount of data that changes between two responses. In our algorithm, the threshold is defined empirically by the shortest distance between: *i*) the maximum distance between the responses belonging to S_{r} and *ii*) the maximum distance between the responses belonging to S_{ii}. Using this threshold, we are able to identify clusters of similar responses. For that purpose, we use the hierarchical clustering technique [19].

Using this classification, we are able to identify the type of requests. Note that a request is associated to a response and vice versa. So, in order to identify the type of requests, it is enough to identify the types of corresponding responses. This identification concerns responses in S_{vi}. Indeed, this is the only set containing responses with type not yet defined. The following rule is used:

1) A request, for which the response belongs to a cluster that also contains responses of S_{r} or S_{ii}, is an invalid SQL injection; otherwise the request corresponds to a valid SQL injection.

An illustrative example is presented in Figure 2.

In this example, the third cluster contains only requests from R_{vi}: these correspond to valid SQL injections. The first cluster contains requests from R_{r}, R_{vi} and R_{ii}. However, this cluster is not the only one associated with requests that generate *rejection* pages: there are different error messages for this example. Other tools (like `Skipfish`) may consider that a response included in the second cluster corresponds to a valid SQL injection simply because it is far from the reference

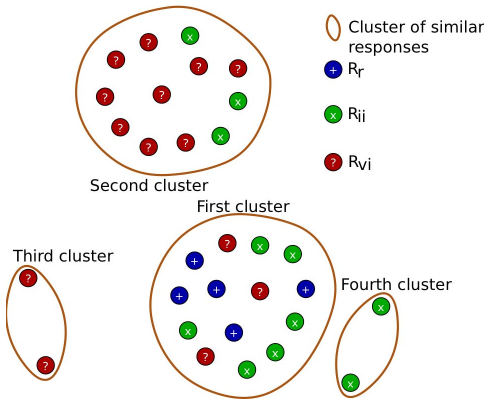

Fig. 2. Example of classification

response page tested by these tools, which would correspond to a request belonging to the first cluster. This case would lead to a false positive.

C. Requests generator

One important aspect of the proposed algorithm is its ability to identify the presence of a vulnerability in an injection point based on multiple responses generated from this injection point. To improve the accuracy of the results, we need to generate a large number of responses, allowing to achieve a high coverage of the response domain. Let us note that other approaches are often based on a small number of responses (for example, 3 for `Skipfish`).

One possible way to generate a significant number of responses (and associated queries) is to record in a static file, queries obtained from security experts (similar e.g., to SQL sheets [20]). A more flexible approach would be to define a grammar to automate this process. Such approach can be compared to some extent to fuzzing techniques [21]. In the following, we outline the grammar that we have defined to automate the generation of R_{r}, R_{ii} and R_{vi} requests.

On the Web server side, most of the time, a SQL query is forged by concatenating SQL terms and parameters sent by the client. For example, the following PHP script deals with the authentication of a user given the username and password sent by the client:

```
$query  = "SELECT id FROM users WHERE
          name='$name' AND pass='$pass'";
```

Given a correct (username, password) couple, the forged SQL query is considered syntactically correct. Also, the forged query associated to a (username, password) couple generated based on a dictionary attack is considered syntactically correct, even if the authentication failed. From this observation, an SQL injection is defined as a string that may change the semantics of the generated SQL query. In many situations, a SQL injection is interesting if it leads to a tautology in the WHERE clause of the forged SQL query. In the previous example, such a SQL injection is:

```
name="' OR 1=1 OR string='"
```

Therefore, the grammar of SQL injections is just a part of the grammar of SQL queries. The advantage of a grammar is that it enables to easily generate as many SQL injections as needed. We can apply the same reasoning to the set of randomly generated words (R_{r}), and to the set of SQL injections that are inappropriate for the given injection point (R_{ii}). A tiny grammar for the set R_{vi} (i.e. the set of SQL injection requests that are constructed in order to generate *execution* pages) can be expressed using [4] notation as follows:

```
INJECTION   |   WORD' POR TAUTAG [' POR TAUTAG]
            |   WORD" POR TAUTAG [" POR TAUTAG]
POR         :=  ' or ' / ')' POR '('
TAUTAG      :=  hex('A')='41
            |    '1'='1
            |    '[f-m]' between '[a-e]' and '[n-z]'
WORD        :=  [0-9a-zA-A]*
...
```

This grammar generates different variations of SQL injection attacks whose principle consists in inserting a tautology inside an expression evaluated by a WHERE clause, in such a way that this expression becomes a tautology itself. To inject the tautology, the initial expression is splitted into several pieces. The `TAUTAG` rules are examples of such tautologies and the `INJECTION` rules express how the tautology is included in an initial expression, i) by closing the expression with delimiter characters (`'`, `"`, or `)`), ii) by inserting the tautology (through a disjonction) and iii) by opening a new expression using the same delimiter characters.

V. Extension to other vulnerabilities

The SQL injection vulnerability detection principle can be generalized. Indeed, many attacks adopt the same behavior: the client sends a string which changes the semantics of the forged query. According to the context, the forged query is sent to a specific web server side component such as the XPATH engine, the OS, etc. The name of the corresponding injection attacks are derived from the name of this component leading to the so-called XPATH injection, OS Commanding, etc. Thus, the clustering algorithm that we have illustrated through SQL injection examples in the previous section can also be used for other kinds of vulnerabilities.

The adaptation of our algorithm to other kinds of vulnerabilities only requires the definition of the three sets of requests R_{r}, R_{ii} and R_{vi} for each kind of vulnerability. Once these sets are established, the algorithm is always the same: sending these requests, storing the corresponding results and obtaining clusters using the distance presented in the previous section. Due to space limits, we briefly present in the following an example that illustrates how our algorithm can be applied for the detection of OS Commanding vulnerabilities. Examples corresponding to XPATH and File Include vulnerabilities are presented in [22].

[4] BNF stands for Backus Normal Form. It is a notation for grammar writing.

In the case of OS commanding vulnerability, the string sent by the client is used to create a command propagated to the OS. This command is then executed by the OS with the privilege of the web server process. The exploitation of this vulnerability allows an attacker to execute arbitrary commands and, for example, to make read or write accesses on the file system. An example of a vulnerable PHP web page on the server side is as follows:

```
<?php
system("cat ".$_GET['cmd']);
?>
```

This code executes the `cat` command with arguments extracted from the `cmd` parameter. As this parameter is not sanitized, the code is vulnerable.

Syntactically invalid and valid requests may respectively look like:

R_{ii}: `.;. LeSS /OFr/BxEBHM . cAt /sCs/wJjqCg`
R_{vi}: `| more /./etc/./passwd`

The first request contains invalid Unix commands and is constructed to be invalid. The second one contains a valid pipe towards a command that is supposed to display the content of the `/etc/passwd` file.

VI. Implementation

Figure 3 presents a high-level view of the tool that we have developed to implement the proposed algorithm following an iterative approach. When the user runs the tool, he provides the initial URL. Most of the time, it corresponds to the main page of the application (`index.html`). Then, the crawling of the application begins, starting from this initial injection point. This crawling identifies all potentially vulnerable injection points. This first stage ends when all accessible injection points have been reached. The second stage begins at this moment. It corresponds to the execution of our clustering algorithm on each injection point, considering the different vulnerabilities presented in the previous section. This second stage may lead to the identification and exploitation of a vulnerability. If so, we obtain a new injection point which was not accessible in the first stage. Consequently, it is possible that a new area of the application becomes accessible from now. Hence, the tool then starts again the first stage using the newly discovered injection point. Figure 3 presents these two stages.

The tool was developed using Python language. The Python libraries make easy the management of HTTP concepts (cookies, parameters, etc.). Our tool is linked to the statistical analysis tool, `R`[5]. The base installation of `R` gives a set of packages useful to achieve hierarchical clustering. They have been used to develop our clustering algorithm. The crawler is developed by ourselves in order to easily integrate it with the vulnerability exploitation part of our algorithm. Our tool is named Wasapy. Wasapy stands for *Web Application Security Assessment in PYthon*.

[5]www.r-project.org

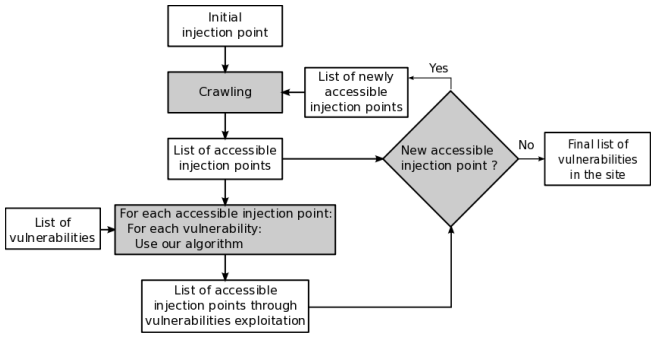

Fig. 3. Principle of the algorithm

VII. Experimental results and discussion

This section presents the experiments that we have carried out to validate and assess our algorithm. We have considered several applications using `Wasapy` and the three open-source vulnerability scanners discussed in this paper: `W3af 1.1`, `Skipfish 1.9.6b` and `Wapiti 2.2.1`. The experiments are run on a Gnu/Linux (2.6 kernel) host running several virtual machines thanks to the `VirtualBox` utility. All the virtual machines run the `Apache web server 1.3.37` or `2.2.8` with `PHP 4.0.0` or `5.0.0` and `MySQL database server 5`.

This section is organised as follows. Subsection VII-A presents notations and abbreviations. Subsection VII-B presents the first experiments realized in order to assess our approach. Five web applications including SQL vulnerabilities are used. We purposely injected these vulnerabilities to calibrate `Wasapy`. Subsection VII-C the second set of experiments with vulnerable off-the-shelf applications, without any modification of these applications. This subsection compares `Wasapy` to other vulnerability scanners on non-purposely injected vulnerabilities. For some of these applications, evaluation reports based on commercial scanners are available in [23]. We reported some of these results in order to compare these scanners with `Wasapy`. Subsection VII-D presents the summary of all these experiments.

A. Notations

The results of our experiments are presented in different tables. We use following notations and abbreviations:

✓ The vulnerability has been detected by the corresponding scanner
✗ The vulnerability has not been detected by the corresponding scanner
– The injection point is not tested by the scanner
SQL*i* stands for `SQL Injection`
XPa stands for `XPath Injection`
OsC stands for `OS Commanding`
FIn stands for `File Include`
CVE reports the CVE reference of the considered vulnerability if it exists
NR The vulnerability does not have a CVE

A vulnerability is considered as detected if the scanner actually sends an alert for this vulnerability, whatever the method used to detect it. A vulnerability is considered as not detected if the scanner actually tested the corresponding injection point without sending any alert. A vulnerability is considered as ignored by the scanner if the corresponding injection point is not tested by the scanner.

B. Experiments with modified applications

The five applications chosen for this first set of experiments are described hereafter:

- `phpBB-3`: This application[6] is a forum manager written in PHP and using a MySQL database. We modified the authentication form of the application so that it includes a vulnerability (`v1`) that can be exploited by a SQL injection. This vulnerability allows an attacker to reach the restricted administration area of the forum.
- `SecurePage`: This application[7] written in PHP, is designed to protect the access of a web site through authentication. Valid couples for this authentication are stored in a MySQL database. A vulnerability (`v2`) was purposely injected, it is similar to `v1`.
- `HardwareStore`: We developed this application, in PHP 5.0. This application allows a user to inventory computer equipments in a database and to interrogate this database. The user needs first to be authenticated. Five SQL vulnerabilities were purposely injected in this application. `v3` allows SQL injection in a search form, and allows an attacker to access the whole database. `v4` allows SQL injection in the authentication form. `v5` allows SQL injection in a parameter of a HTML request. For this vulnerable HTML page, we have purposely disabled the error message reporting, in order to compare the behavior of `W3af` and `Wapiti` in such a situation with the behavior of `Wasapy`. Vulnerability `v6` is similar to `v4` but it is used in a different context: the error message reporting is deactivated. Vulnerability `v7` can only be exploited after the successful exploitation of `v4`. Indeed, this vulnerability is included in a page that can only be accessed after successful authentication on the application or after a successful bypass of the authentication mechanism (through exploitation of `v4`). XPATH, OS Commanding and File Include vulnerabilities were also injected in this application. Vulnerability `v10`, in the authentication page, allows an attacker to bypass the authentication through a XPATH injection. `v11` is an Os Commanding vulnerability that can be exploited only after `v4` is successfully exploited. Indeed, this vulnerability is included in a page that is only accessible after authentication (or bypass of the authentication through successful exploitation of `v4`). Vulnerability `v12` is a File Include vulnerability, it is included in the same page as `v11` and can be exploited in the same conditions as `v11`.
- `Insecure`: This application was developed in Ruby on Rails in the context of the Dali project[8]. It is an e-commerce site, including user sessions through virtual shopping carts. A vulnerability (`v8`), which allows an attacker to inject SQL code, was purposely included in the authentication form of the application. This vulnerability, functionally equivalent to `v4` is anyway different because `Insecure` is implemented in Ruby and the error reporting messages differ from the Apache error reporting messages.
- Damn Vulnerable Web Application (DVWA): This application[9] is written in PHP and uses MySQL server. A vulnerability `v9`, similar to `v3`, was purposely introduced in the application.

Vulnerabilities			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	–	–	–	✓
Number of detections			5	4	3	12

Fig. 4. Vulnerability detection results for modified applications

Figure 4 shows that the performances of `W3af` and `Wapiti` are similar in average, even if the vulnerabilities detected are not the same (`Wapiti` successfully detects `v1` and `v2` whereas `W3af` does not detect them; on the other hand, `W3af` detects `v4` and `v8` whereas `Wapiti` does not detect them). This result is consistent with the fact that both scanners use a pattern matching-based algorithm. The observed variations are related to the generation of different requests by these tools. `Wasapy` allows us to detect all these vulnerabilities. This

[6]http://www.phpbb.com
[7]http://www.01php.com/fiche-scripts-126.html
[8]ANR's program ARPEGE(2009-2011).
[9]http://www.dvwa.co.uk

confirms that the vulnerability detection clustering algorithm presents a better coverage than the pattern matching algorithm for these vulnerability classes.

Regarding vulnerabilities v1 and v2, we manually checked the injections performed by Skipfish (', \' \" and \\' \\") and stored the corresponding responses (respectively A, B et C). As discussed in Section III, Skipfish considers that the pages A and C must be different so that a vulnerability is present. Unfortunately, for these two injection points, this is not the case. The responses correspond to SQL error messages that are very similar.

Regarding vulnerabilities v5 and v6, they are included in PHP pages for which we purposely deactivated the error reporting message feature[10] in the configuration file of PHP5. In this particular case, none of the three scanners (Skipfish, W3af and Wapiti) is able to detect vulnerabilities.

Regarding vulnerability v7, Wasapy is the only scanner that is able to detect it. Moreover, it is the only scanner that is able to test the corresponding injection point. Indeed, this injection point is included in a HTML page that can only be accessed after a successful authentication or after the successful exploitation of vulnerability v4. As Wasapy is the only scanner able to actually exploit v4, it can automatically access the page including vulnerability v7. For the other scanners, it is necessary to manually perform the exploitation of v4 so that it is possible to access the page including v7. Vulnerabilities v11 and v12 were identified only by our tool for the same reasons: they remain masked until the authentication is bypassed.

The purpose of these initial tests was the calibration of Wasapy. The calibration of our tool consists in defining empirically the number of requests to generate for each group and injection point. We set this number to 30 for all the applications tested (i.e., 90 requests per injection point). We have observed that a higher number does not provide significantly higher accuracy, while a lower number generates false negatives.

These initial tests also allowed us to check the grammars that we presented in the previous section. Of course, the corresponding vulnerabilities have been identified for this purpose. So, these results are not aimed to be used to make an absolute comparison between the scanners. A more representative comparative assessment of the different tools should be based on vulnerable applications in which vulnerabilities have not been deliberately injected by ourselves. These experiments are presented in the next subsection.

C. Experiments with non-modified vulnerable applications

This second set of experiments allowed us to have a more precise idea of the coverage of our detection algorithm. For that purpose, we compared it to the detection algorithms of Skipfish, W3af and Wapiti on not purposely modified vulnerable web applications. For some of these applications, we could compare our algorithm with some commercial vulnerability scanners, considering the results available in [23]. In this document, the author presents the vulnerability detection results obtained with three commercial scanners: WebInspect from HP, AppScan from IBM and Web Vulnerability Scanner from Acunetix. These results provide only some preliminary indications to analyse the performance of our tool on the same set of applications and are not meant to be used for a validation purpose.

Vulnerability			Skipfish	W3af	Wapiti	Wasapy
Type	CVE	Location				
SQL*i*	*NR*	search.php	✓	✓	✓	✓
	2005-3236	lostpwd.php	✓	✓	✓	✓
	2005-3236	newmsg.php	✓	✓	✓	✓
	2005-3575	show.php	✓	✓	✓	✓
False positive			1	0	0	0

Fig. 5. Vulnerability detection results for Cyphor application

For our experiments, we selected five web applications (most of them tested in [23]), known to include vulnerabilities. These applications cover different functionalities and execution contexts. We installed these applications, and performed vulnerability detection tests without modifying them.

- Cyphor[11] is a configuration Webforum, which uses PHP 4.0.0 session capabilities to authenticate users and a MySQL database.
- Seagull[12] is an OOP framework for building web, command line and GUI applications. This project allows PHP developers to integrate and manage code resources, and build complex applications. This application requires the following configuration: PHP 4.3.0 or newer, MySQL 4.0.x or newer, Apache 1.3.x or 2.x.
- Fttss is a research project[13] that implements a Text-To-Speech System based on PHP (4.3.0 or newer) and MySQL (4.1.2 or newer).
- Riotpix[14] is an open-source discussion forum for the web based on PHP (4.3.0 or newer) and MySQL (4.1.2 or newer).
- Pligg[15] is a social networking open source CMS (Content Management System) that permits visitors to register on the website, submit content and connect with other users. This software creates websites where stories are created and voted on by members. PHP (4.3.0 or newer) and MySQL (4.1.2 or newer) are required.

We inspected manually all vulnerabilities detected by each scanner to check the experiment results and get more confidence on the number of detected vulnerabilities and false positives.

Figure 5 presents the results for Cyphor application. All

[10]The configuration file of PHP5 includes: *For production web sites, you're strongly encouraged to turn this feature off, and use error logging instead.*

[11]http://webscripts.softpedia.com/script/Snippets/Cyphor-27985.html
[12]http://seagullproject.org/
[13]http://fttss.sourceforge.net
[14]http://www.riotpix.com/
[15]http://www.pligg.com/

Vulnerability			Skipfish	W3af	Wapiti	Wasapy
Type	CVE	Location				
SQL*i*	*2010-3212*	index.php	✗	✗	✗	✓
FIn	*2010-3209*	container.php	✗	✗	✗	✗
	2010-3209	QuickForm.php	✗	✗	✗	✗
	2010-3209	NestedSet.php	✗	✗	✗	✗
	2010-3209	Output.php	✗	✗	✗	✗
False positive			0	0	0	0

Fig. 6. Vulnerability detection results for `Seagull` application

Vulnerability			Skipfish	W3af	Wapiti	Wasapy	AppScan	WebInspect	Acunetix
Type	CVE	Location							
OsC	*NR*	index.php	✗	✓	✗	✓	✗	✗	✗
False positive			0	0	0	0	0	0	0

Fig. 7. Vulnerability detection results for `Fttss` application

the scanners found all the vulnerabilities because error messages are reported to the client. Thus, it is easy to distinguish successful vulnerability exploitation from error messages. The underlined results correspond to detections made possible by supplying a valid (login/password) to the scanners to perform authentication. In other words, the corresponding vulnerability is only visible when logged in the site (the authentication page does not contain any SQL-injection vulnerability, it is the only way for any scanner to access the page including the vulnerability).

The results reported for `Seagull` in Figure 6 show that `Wasapy` is the only one that reports a vulnerability in this application. Others are unable to do so because the application does not report errors to the client. Regarding File Include vulnerabilities, the injection points which allow their exploitation are not directly visible by the client. Hence, the source code is necessary to identify these vulnerabilities. This explains the failure of all scanners.

`Fttss` is an application that has been tested in [23]. Hence, some results associated to the three commercial scanners considered are available (cf. Figure 7). The commercial scanners do not detect the OS commanding vulnerability, which is the only vulnerability known of this application. In contrast, `W3af` and `Wasapy` are able to identify this vulnerability. It is noteworthy that none of tested scanners reports false positives in this case.

Regarding `Riotpix` (cf. figure 8), the results are similar to those of `Cyphor`. The vulnerabilities are only accessible to successfully authenticated users. Therefore we had to provide a valid login/password to all scanners. Two vulnerabilities have not been found by any scanner. They correspond to code injection into variables that are not visible to the client and thus cannot be discovered by scanners (their identification would require a source code analysis). These results also show that `Wasapy` is efficient for this kind of vulnerability.

Regarding the `Pligg` application (cf. figure 9), all vulnerabilities but the first two are available on hidden injection points. The scanner must be aware of the presence of the injection point in order to test the vulnerability. For the first two vulnerabilities, `Wasapy` found them, whereas the other scanners found only one of these vulnerabilities. This is due to the fact that error messages are not forwarded to the client.

Vulnerability			Skipfish	W3af	Wapiti	Wasapy	AppScan	WebInspect	Acunetix
Type	CVE	Location							
SQL*i*	*NR*	edit_post.php	✗	✗	✗	✓	✗	✗	✗
	NR	edit_post_script.php	✗	✗	✗	✗	✗	✗	✗
	NR	index.php	✗	✗	✗	✗	✗	✗	✗
	NR	message.php	✗	✗	✗	✓	✗	✗	✗
	NR	reader.php	✓	✓	✗	✓	✗	✗	✗
False positive			0	0	0	0	0	0	0

Fig. 8. Vulnerability detection results for `Riotpix` application

Vulnerability			Skipfish	W3af	Wapiti	Wasapy	AppScan	WebInspect	Acunetix
Type	CVE	Location							
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

Fig. 9. Vulnerability detection results for `Pligg` application

D. Summary

The main lessons learned form all our experiments are summarized in the following:

- `Wasapy` is an efficient scanner, especially in particular conditions for which it has been designed : 1) it is more efficient than the other freeware scanners tested when the error reporting is disabled, 2) it is more efficient than the other scanners to discover and exploit vulnerabilities that are included in pages not directly accessible (pages that require the successful exploitation of a vulnerability to be accessed). Indeed, our scanner is the only one which is capable of actually exploiting the vulnerability, and supplying the exact corresponding injection requests.
- `Wasapy` is globally as efficient as the other vulnerability scanners tested on not modified vulnerable applications.
- Our clustering algorithm can be easily adapted to different kinds of vulnerabilities. Besides SQL injections, the results of the experiments show that `Wasapy` also detects XPATH, OS Commanding and File Include vulnerabilities and that it is at least as efficient as the other vulnerability scanners.

VIII. Conclusion

In this paper, we presented a vulnerability scanner that implements a new algorithm aimed at detecting web applications vulnerabilities, following a black-box approach. Our algorithm is based on the automatic generation of specially crafted inputs allowing the successful exploitation of detected vulnerabilities, using data clustering techniques and language theory.

We used a large set of applications with and without deliberately injecting vulnerabilities into them in order to validate our approach and analyse its effectiveness compared to three other open source web scanners. The experimental results are promising. In particular, we have shown that our algorithm can contribute to improve the detection capabilities of some existing tools and to pushes one step forward the automation of the vulnerability detection process (by allowing for instance the successful detection of vulnerabilities in pages that are not directly accessible but only accessible after the successful exploitation of a first vulnerability).

In this paper we focussed on SQL injection vulnerabilities. Nevertheless, we have also shown that the proposed algorithm can be successfully applied to other vulnerability types. A more extensive validation case is needed to confirm the promising results obtained so far. The proposed algorithm has been implemented in a new tool `Wasapy`. This algorithm can also be integrated in the open source tools that we have investigated (such as `W3aF` and `Skipfish` to take advantage of the other powerful facilities offered by these tools.

Overall, the experimental results presented in the paper also highlight the usefulness of combining different kinds of vulnerability scanners at the same time to enhance the vulnerability detection capabilities of current scanners. A first kind can be represented by `W3aF` which performs pattern matching on web responses and the second kind can be represented by our scanner which uses a different approach. Such an architecture leads to the problem of correlation of the results provided by diverse tools. It will be interesting to perform a new evaluation considering scanners working together with an overall detection correlation mechanism. The corresponding results may contribute to propose an efficient method to wisely combine different scanners.

References

[1] Mitre, "2010 CWE/SANS Top 25 most dangerous software errors", Document version 1.06, September 2010, http://www.cwe.mitre.org/top25

[2] Y.-W Huang, F. Yu, C. Huang, C.-H.Tsai, D.-T.Lee, and S.-Y Kuo, "Securing Web Application code by static analysis and runtime protection", *Proc. 13th Int. Conf. on World Wide Web (WWW'04)*, NY, USA. ACM. pp. 40-52.

[3] V.B. Livshits and M. S. Lam, "Finding security errors in Java program with static analysis", *Proc. 14th Usenix Security Symposium*, Baltimore, MD, USA, 2005.

[4] N. Jovanovic, C. Kruegel, and E. Kirda, "Static analysis for detecting taint-style vulnerabilities in web applications", *Journal of Computer Security*, 18 (2010), pp. 861-907.

[5] Y. Xie, A. Aiken, "Static detection of vulnerabilities in scripting languages", *Proc. 15th USENIX Security Symposium*, pp. 179-192, 2006

[6] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities", *SIGPLAN Notices*, vol 42, n06, pp. 32-41, 2007.

[7] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing Web Applications with static and dynamic information flow tracking", *Proc . of the 2008 ACM SIGPLAN Symposium on Partial evaluation and semantics-based program manipulation (PEPM'08)*, New York, NY, USA : ACM, 2008, pp. 3-12.

[8] T. Pietraszek, C.V. Berghe, "Defending against injection attacks through context sensitive string evaluation", *Recent Advances in Intrusion Detection (RAID-2005)*, Seattle, WA, USA, 2005.

[9] C. Kruegel, G. Vigna, "Anomaly Detection of Web-based Attacks", *Proc. of the 10th ACM Conference on Computer and Communication Security (CCS'03)*, pp. 251-261, October 2003.

[10] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-side scripting attacks", *21st ACM Symposium on Applied Computing (SAC2006)*, Dijon, France, 2006.

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

[12] Sectools Website, "Top 10 vulnerability scanners", http:// sectools.orf/web-scanners.html

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

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

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

[16] 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.

[17] Levenshtein, V., *Leveinshtein distance*, 1965 http://en.wikipedia.org/wiki/Levenshtein_distance[accessed on 02/22/10]

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

[19] S. C. Johnson, "Hierarchical Clustering Schemes", in *Psychometrika Journal*, pp. 241-254,Volume = 2, 1967.

[20] A.Kiezun, P. J. Guo, K.Jayaraman and M. D. Ernst, "Automatic creation of SQL Injection and cross-site scripting attacks", *IEEE 31st Int. Conf. on Software Engineering(ICSE 2009)* , Vancouver, BC, 2009.

[21] E. Gutesman, "gFuzz: An Instrumented Web Application Fuzzing Environment", *Hack.Lu '08*, Luxembourg, 2008.

[22] A.Dessiatnikoff and R.Akrout and E.Alata and M.Kaâniche and V.Nicomette, "HTML pages clustering algorithm for web security scanners", *Rapport Laas N11053*, 2011.

[23] http://anantasec.blogspot.com/ [accessed on 12/9/10]