An automated black box approach for web vulnerability identification and attack scenario generation

Rim Akrout, Eric Alata, Mohamed Kaâniche, Vincent Nicomette

Noname manuscript No.
(will be inserted by the editor)

An automated black box approach for web vulnerabilities identification and attack scenario generation

Rim Akrout · Eric Alata · Mohamed Kaaniche · Vincent Nicomette

the date of receipt and acceptance should be inserted later

Abstract Web applications have become increasingly vulnerable and exposed to malicious attacks that could affect essential properties of information systems such as confidentiality, integrity or availability. To cope with these threats, it is necessary to develop efficient security protection mechanisms and assessment 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 it 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 methodology led to the development of a new Web vulnerability scanner that has been validated experimentally on several examples of vulnerable applications.

Keywords Web application · Vulnerabilities · Attacks · Evaluation · Web Scanner

1 Introduction

Web application vulnerabilities have become, in the recent years, a major threat to computer systems security. This is illustrated in e.g., the IBM X-force 2012 mid-year trend and risk report which shows that Web application vulnerabilities including SQL injections and Cross-Site Scripting occupy the top highest positions in computer threats [1]. This situation can be explained by the increase in complexity of Web technologies, by the frequent evolution of these technologies, by 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.

In this paper, we propose a novel methodology that allows to automatically identify residual vulnerabilities of a Web application from the analysis of the targeted application, following a black box approach. The proposed approach can automatically identify and exploit vulnerabilities. It is also designed to highlight potential attack scenarios including the exploitation of several successive vulnerabilities that are not necessarily independent. The identification of these scenarios is based on the dynamic crawling of the application, resulting in the creation of a navigation graph that describes the different possibilities for a user to activate the application and associated vulnerabilities. This graph explicitly represents the dependencies between the vulnerabilities of the site and thereafter various attack scenarios. To validate our approach, we developed a new vulnerability scanner, that has been validated on different vulnerable applications and compared experimentally with other existing vulnerability scanners.

This paper extends the results presented in [2] and [3]. It is structured into 7 sections. Section 2 discusses related work focusing on the analysis of the vulnerability detection algorithm used by several well-known freeware vulnerability scanners, and presents some weaknesses of these algorithms. Section 3 presents our clustering algorithm for detecting Web application vulnera-

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse
Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France

E-mail: rakrout, ealata, kaaniche, nicomett@laas.fr

bilities. Section 4 presents an overview of our approach for constructing the graph. The details of the algorithm are outlined in section 5. We present in section 6 the experiments performed in order to validate our approach and assess the efficiency of our scanner. Finally section 7 concludes this paper and discusses future work.

2 Background and Related Work

Most frequent attacks on Web servers include SQL injection attacks (for Web servers connected to a 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, without loss of generality.

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 is returned by the server as a consequence of an error while processing the request. 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. In particular, 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. 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. Overall, the identification of potential vulnerabilities is generally based on the characterization of responses of a Web server to crafted requests sent via the *injection points* and the ability to distinguish *rejection* pages and *execution* 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.

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 2.1. The second one relies on the analysis of similarities between the pages returned by the server and is presented in section 2.2. The last section 2.3 proposes a discussion regarding the limits of these approaches.

2.1 Error pattern matching approach

To identify SQL injections, the error pattern matching approach consists in sending specially crafted requests to the application and looking for specific patterns in the responses, e.g., 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` [4] 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 [4]. This list, derived by analyzing response pages of vulnerable Web sites, is aimed at covering 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.

2.2 Similarity approach

This approach relies on three assumptions: 1) *execution* and *rejection* pages are different, 2) it is easy to build

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

requests that generate *rejection* pages (by generating random or syntactically invalid requests for instance) and 3) it is 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 *rejection* pages among the response pages (i.e., pages that highlight non-sanitized inputs).

Let us consider as an example the approach adopted by `Skipfish`[3] for detecting SQL injection vulnerabilities. Three requests are sent 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 [5] 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.

2.3 Discussion and Contributions

The assumption used by the error pattern matching approach is debatable. 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 conclude 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 neither that the server does not sanitize all the SQL requests nor that the non-sanitized input can be chosen in order to successfully exploit any vulnerability.

Regarding the similarity approach, it is easier to generate random or syntactically invalid 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 generate too few requests. For instance, `Skipfish` uses only 3 requests.

Also, as in any classification problem, the choice of the distance is very important. The one used in `Skipfish` does not 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 [5] with a text similarity distance. As an example, the two following pages use the same words in a different order, but they have 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 3 builds on some of the concepts of the similarity approach and addresses 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. While our work shares some of the ideas and objectives presented in [5], we follow different approaches. For example, the clustering approach which is the core of our algorithm is not used in [5]. 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.

3 Html pages clustering for Web vulnerabilities detection

The approach presented in this section seeks to achieve the automated detection of different types of Web vulnerabilities, corresponding to SQL injection attacks, Os-

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

Commanding, File Include and XPath [4]. It is based on the automatic classification of responses returned by the Web server using data clustering techniques and identifies queries that are able to successfully exploit vulnerabilities.

3.1 Principles

In the following, let us take the example of a SQL injection in an authentication form. Our goal is to identify, among several SQL injections, those which allow an attacker to bypass the authentication. The main challenge is the automation of this process. In the following, we present a method which intends to reduce false-negatives and false-positives in comparison with existing solutions presented in the previous section. Our approach relies on the following assumptions: *a*) the content of an *execution* page is significantly 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 even for the same class of inputs.

In the login page that we consider, responses to valid requests include welcome messages and invalid input error messages. Responses to invalid requests include PHP or SQL error messages. The essential point is the existence of differences between *execution* pages and *rejection* pages, and, more precisely, between welcome messages and invalid input error messages, and between PHP and SQL error messages. Our approach focuses on the analysis of these differences. The objective is to identify, among several responses, those which correspond to *execution* pages generated through syntactically valid requests. 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 as *execution* page or *rejection* page). Obviously, it is easier to generate requests which lead to *rejection* pages or *execution* pages corresponding to invalid input error messages than requests which lead to *execution* pages associated to welcome messages. To generate the former, one can, for example, use random usernames and passwords to fill the authentication form. One can also easily generate requests which lead to *rejection* pages associated to PHP or SQL 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 or *execution* pages associated to invalid input error messages. For example:

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

R_{ii} is the set of syntactically incorrect SQL injection requests that are 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 syntactically correct 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}.

Let us 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.

3.2 Distance

To analyze the similarity between two HTML pages, we need a distance for assessing the difference between two strings. As discussed in section 2.3, the order of words in a text could be relevant. In fact, the same words in a different order can completely change the semantics of the response. Thus, to compute the distance between two pages, we use a normalized version of the Levenshtein distance. 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.

Generally, clustering techniques are based on two different strategies. The first is driven by the number

[4] See [9] for a more detailed description of these vulnerabilities

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

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

Fig. 1 Distance for clustering

of clusters, if it is known a priori. It starts by considering a single cluster containing all requests and divide it progressively, relying on distances, until the desired number of clusters. The second technique is used in case the number of clusters is not known a priori. It consists in grouping in a same cluster the requests whose pairwise distance is below a threshold. In our approach, the number of clusters is not determined a priori, and so we use the second strategy, called hierarchical clustering [11], which requires the choice of a threshold.

The threshold for grouping queries can vary from an injection point to another. Indeed, it depends on the size of the responses and the amount of data that differ between two responses for the same type of requests. Also, this threshold must be adapted to each Web application. In our approach, 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} .

3.3 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. Note that other approaches are often based on a small number of responses (for example, 3 for `Skipfish`).

One possible way to generate different types of responses (and associated queries) is to record in a static file, queries obtained from security experts (similar to e.g., to SQL sheets [12]). A more flexible approach would be to define a grammar to automate this process. Such an approach can be compared to some extent to fuzzing techniques [13]. 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 created 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 semantically valid (user name, password) pair, the created SQL query is considered syntactically valid. Also, the created query associated to a (user name, password) pair generated based on a dictionary attack is considered syntactically valid, even if the authentication failed. From this observation, a SQL injection is defined as a string that leads to a syntactically valid SQL query while changing the semantics of this generated SQL query. In many situations, a SQL injection is relevant if it leads to a tautology in the WHERE clause of the forged SQL query. From the previous example, an example of 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 BNF[5] 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. It 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 disjunction) and iii) by opening a new expression using the same delimiter characters.

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

3.4 Extension to other vulnerability classes

The approach proposed for SQL injection vulnerability detection can be generalized. In fact, many attacks have the same behavior: the client sends a string which changes the semantics of the forged query. Depending on the context, the forged query is sent to a specific component in Web server-side such as the XPath engine, operating system, etc. The names of the corresponding injection attacks are derived from the name of this component leading to XPATH injection, Os Commanding, etc. Thus, the clustering algorithm that we have illustrated using the example of SQL injection in the previous section can be also used for these types of vulnerabilities.

XPATH vulnerabilities consist, like SQL vulnerabilities, in submitting not sanitized input in an HTML form or URL parameters. The difference is that the vulnerability can be exploited to execute XPath queries and not SQL queries.

In the case of OS Commanding vulnerability, the string sent by the client is used to create a command executed by the operating system. This command is executed under the identity of the process corresponding to the Web server. Exploitation of this vulnerability allows an attacker to execute arbitrary commands on the system and can also allow read and/or write access to certain files.

As explained previously, the algorithm uses three sets of queries: R_a (random request), R_{ii} (syntactically invalid injections) and R_{vi} (syntactically valid injections). The adaptation of the algorithm to other types of vulnerabilities requires the definition of these three sets for each type of vulnerability. Once these sets are established, the algorithm proceeds in the same way: send those requests, and store the corresponding results obtained by the clustering distance presented in section 3.2. Further details can be found in [9].

4 Attack scenarios with multiple vulnerabilities

The previous section described a methodology allowing the detection of a single vulnerability by automatic HTML pages clustering. In this section, we present a global approach that aims at exhibiting attack scenarios, resulting from the exploitation of several vulnerabilities, some of which may be causally dependent on each other.

The objective of this approach is to establish these scenarios from the automatic construction of the graph representing all possible navigations on a site taking into account its vulnerabilities. The approach is composed of two steps: 1) a first step aimed at identifying the different possibilities offered to a client to navigate through the web site; and 2) a second step aimed at the identification and exploitation of vulnerabilities in the web pages and injection points identified at step 1, using the methodology presented in section 3. The iteration of steps 1 and 2 allows the elaboration of attack scenarios including the exploitation of multiple vulnerabilities.

We adopt a black box approach since no details of the source code implementation of the Web site is required. The public address of the Web site is used as a starting point for dynamic discovery of the site and its vulnerabilities. We begin this section by introducing some definitions that are useful for the presentation of our approach. Thereafter we describe the principles of our approach.

4.1 Definitions

Our approach aims at automatically building a graph that represents the set of all possible navigations on a Web site, including those that result from exploitations of the Web site vulnerabilities. Let us call a *navigation* a sequence of requests (a request consisting in the activation of an HTML link). A sequence of requests actually sent by a client is called a trace. The set of all the possible navigations by a client during a visit to the web site can be represented by an automata called a *navigation graph*.

A *navigation state* of a client (i.e., a browser) is composed of 1) the HTML page currently displayed by the browser and 2) the current values of the cookies [6] in the browser. A request sent by a browser provokes a change of the current navigation state.

Each node of the navigation graph corresponds to a navigation state. An edge between two navigation states exists if a request whose execution leads to the transition from an initial state to the another one can be sent by the client. An edge may correspond to a "normal" request or to a request that exploits a vulnerability of the Web site. A *vulnerability graph* is a particular case of a navigation graph that includes edges corresponding to the exploitation of vulnerabilities.

It is important to note that a navigation graph is different from a traditional graph of HTML pages describing the structure of a Web site. Each node of an HTML pages graph generally corresponds to an HTML page of the site and an edge between two nodes identifies a link that enables to access the second page from

[6] The cookie stores at the client side a variety of information about the running session (keys, contents, user preferences, etc.)

the first one. The difference is mainly due to the fact that a navigation state does not only depend on the currently accessed HTML page. Indeed, a client can access an HTML page several times, while being in different navigation states. For instance, let us consider an e-business Web site. It is possible to browse the purchasing page after having ordered some products or not. When browsing this page, in the first case, the client is authorized to pay for the products and in the second case, an error message is returned (it makes no sense to purchase for an empty bag). However, in these two situations, the HTML page accessed is the same. The difference is due to the content of the cookies, that indicate that products have been ordered or not, i.e., that reflect the current navigation state.

4.2 Principles

The construction of the navigation graph is progressive and dynamic by identifying different navigations and vulnerabilities. In addition, a vulnerability exploitation opens new possibilities for navigation. The construction is thus done iteratively. Our approach is composed of a navigation step called *crawling* to identify the various possibilities to navigate through the site and the associated navigation states and a step for identifying and exploiting vulnerabilities as presented in section 3.

Figure 2 presents a high-level view of the proposed approach. The crawling starts from the initial URL (which corresponds mostly to the main page of the Web application), after by deleting the cookies at the client side. This initialization is important for having different independent navigations. From this URL, combinatorial site crawling identifies the traces navigation list. Our approach is based on an exhaustive search to obtain all possible navigations of the site. The site is browsed starting with the initial query and storing the requests sent to the site. The choice of the request to send is done by analyzing the content of the page displayed. If this HTML page contains several links, one of these links is chosen to build the following query and the other links are stored for later analysis. As the crawling of the Web site may be infinite, a threshold indicating the maximum exploration depth of the site has to be specified. The corresponding value is an input parameter of the algorithm. If the maximum depth is reached or if no requests can be sent from the current reached state, i.e., the current page no longer contains HTML links, the requests sequence stored from the initial state corresponds to a site navigation. Then, the process restarts from the beginning trying new navigations, based on the stored choices. At the end, the set of sequences of requests representing all site navigations is obtained.

On some complex Web sites, the number of requests sent while crawling may be huge. The potential number of *injection points* is even more important. Hence, it is more appropriate to analyze vulnerabilities on a compact representation rather than directly on the individual navigations. One way to obtain a compact representation is to minimize the navigation graph built using the set of request sequences representing all site navigations. The first version of the navigation graph does not contain any edge corresponding to vulnerabilities.

The construction of the minimal graph from the set of navigations and the associated sequence of requests is similar to a grammatical inference problem whose objective is to find a minimal automaton that represents a language from symbol sequences of this language (so-called *words*). In this analogy, the automaton corresponds to the navigation graph and the symbols correspond to requests. As a language may include an infinite number of words, the algorithm must be able to run based on a subset of the words of a language. Two categories of grammatical inference algorithms exist: i) those that infer the language only from sequences of words that belong to the language and ii) those that consider all the sequences of words. More details can be found in [14]. The RPNI algorithm (**R**egular **P**ositive **N**egative **I**nference) [15] we chose, belongs to the second category. This widely used algorithm presents a polynomial time-based complexity, and is quite simple to implement.

At the end of the crawling step, we note that the only possibility to enrich the graph is to identify vulnerabilities that if successfully exploited could lead to add new edges and nodes to the navigation graph. Such vulnerabilities can be identified using the approach detailed in section 3). This algorithm allows the identification and effective exploitation of existing vulnerabilities. This exploitation leads to the discovery of new pages that, in turn, may contain new *injection points* that were not available at the first step. Therefore, new possibilities for the exploitation of vulnerabilities are available. Subsequently, the approach is re-executed iteratively by including new pages, which leads to the construction of several navigation graphs until satisfying the stopping criterion, which is specified by the maximum navigation depth. The graphs including edges corresponding to the exploitation of a vulnerability are so-called *vulnerability graphs*.

Figure 3 pictures this iterative approach. Red edges identify vulnerabilities whose exploitation reveals new states and edges of the navigation graph that were initially inaccessible.

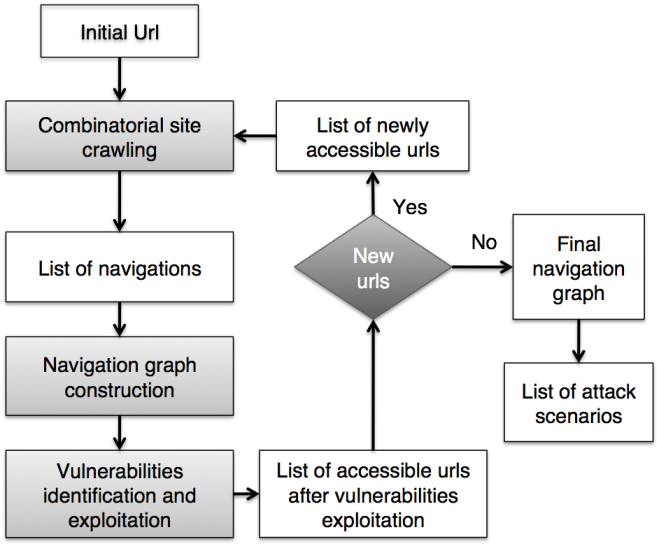

Fig. 2 Injection point extraction algorithm and vulnerabilities identification

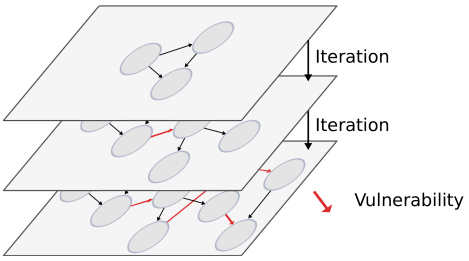

Fig. 3 Iterative construction of the navigation graphs

This algorithm has been implemented in a software tool using the *Python* language, which greatly facilitates the handling of HTTP concepts (cookies, settings, etc.). This tool is interfaced with the statistical analysis software *R* [7] which integrates a set of clustering programs that we detailed in section 3. They have been used to develop our classification algorithm. This tool is called *Wasapy*, which stands for **W**eb **A**pplication **S**ecurity **A**ssessment in **Py**thon.

4.3 Example

In order to illustrate our approach, we developed an e-commerce Web site for buying books, using the `PHP` language and a `MySql` database. This site is a simple proof of concept but it uses technologies and a structure similar to "real" Web sites. Figure 4 presents the HTML pages graph describing the structure of the Web site. A page is represented by an icon. An edge between two pages corresponds to the existence of a HTML link in the source page leading to the second page. Let us note that a particular reflexive link exists for the `display.php` page. This page enables to list the available books and includes a filtering function in a particular form field. A user may then enter a regular expression in this field and submit it to the site, in order to update the list of books.

[7] `http://www.r-project.org/`

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

Fig. 5 List of navigations of the first iteration

This site includes three vulnerabilities. The pages including these vulnerabilities are identified by a star on figure 4. The first vulnerability is associated to the page `login.php`. The exploitation of this vulnerability allows an attacker to bypass the authentication thanks to a SQL injection. The second vulnerability is associated to the page `display.php`. It allows an attacker to download the content of the database. The last vulnerability associated to the page `check.php` allows an attacker to pay the products he ordered without providing any credit card number. This vulnerability cannot be exploited unless some products have been added to the virtual shopping cart.

First, we consider a non malicious user, who does not have any account on the site. The only actions that the user can do are :

- access the `index.html` page
- fill in the form with the authentication information in the `login.php` page
- get information from the `about.html` page

The list of navigations that can result from these actions are summraized in figure 5. All these navigations can be synthesized by the navigation graph in figure 6. In this graph, the edges correspond to links of the site (html page or php, etc.), the nodes correspond to navigation states. The set of edges leaving a node is the set of links accessible from the corresponding navigation state. This set is independent from previously activated links used to achieve this navigation state.

This navigation graph is very simple because the possible actions without valid (`login` / `password`) are limited. However, if we consider an attacker who is able to exploit some vulnerabilities, then he is able to perform more actions than an unregistered benign user. Therefore, the associated navigation graph is a richer version of the graph in figure 6, new edges and new nodes may appear.

This is precisely what the next step serves for. The edges of the graph 6 are tested to identify vulnera-

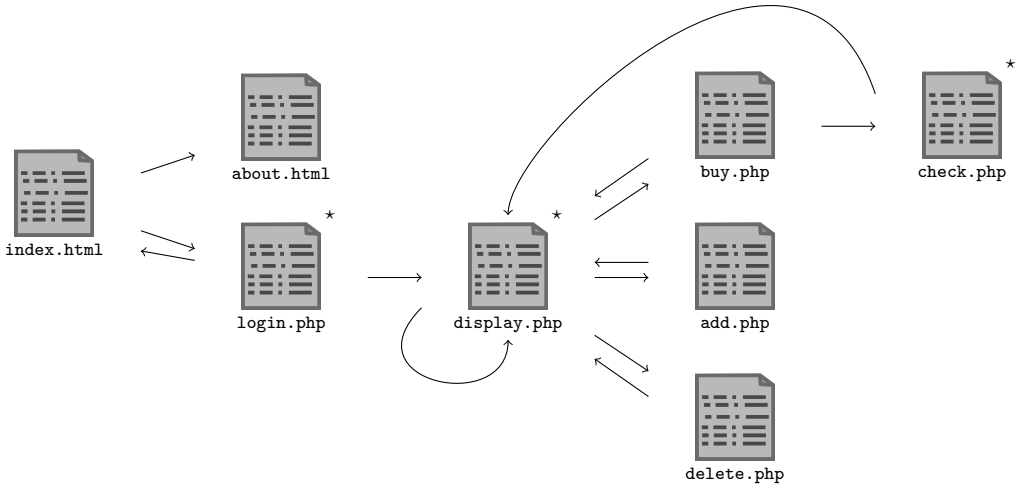

Fig. 4 Structure of the Web site

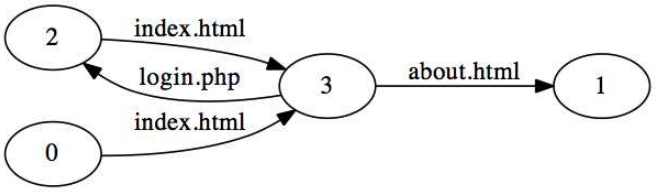

Fig. 6 Navigation graph of a not authentificated user

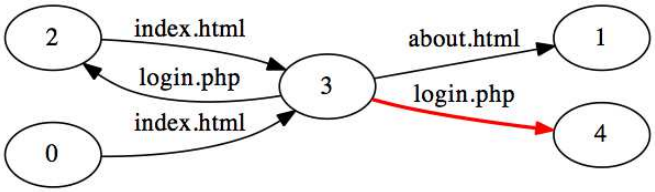

Fig. 7 Vulnerability graph of 1st iteration

bilities[8]. The exploitation of a new vulnerability may change the navigation state. Thus, it can lead to the insertion of a new node in the graph. At this stage, the only available vulnerability corresponds to a SQL injection for the authentication, ie, in the `login.php` page. The result is depicted in figure 7.

During the second iteration, we identify pages that can be accessible after the exploitation of the vulnerability identified during the previous iteration. To reach these pages, it is necessary to cross the edges `index.html` and `login.php`. The set of traces executed for the second iteration includes 65 traces. These traces reach the following files, `display.php`, `add.php`,`delete.php`, `buy.php` or `check.php`. Then the vulnerabilities identification phase is re-executed once for each new edge generated during of the second iteration. We iterate in this way until we get the final graph that covers the entire site as shown in figure 8. In the case of this example, the algorithm stops after 6 iterations considering a maximum depth of navigation set to 7. This means that there are no more vulnerabilities discovered during the sixth iteration.

To automate the process, we have implemented algorithms corresponding to the two main steps of our approach: the crawling and the vulnerabilities discovery. These algorithms are presented in next section.

5 Algorithms

This section presents the algorithms that we have developed to implement the approach described in the previous section.

The *search_vulns* function in figure 9 takes a navigation as input parameter. The latest request of this navigation is analyzed to identify vulnerabilities, considering different vulnerability classes (SQL injections, XPATH injections, OS commanding, etc.). This function returns a list of navigations and, for each of them, one of the identified vulnerabilities. Each new navigation includes one more navigation state that results

[8] For simplicity, only the HTML links activated by the request appear, without explicitly specifying the parameters associated with these requests.

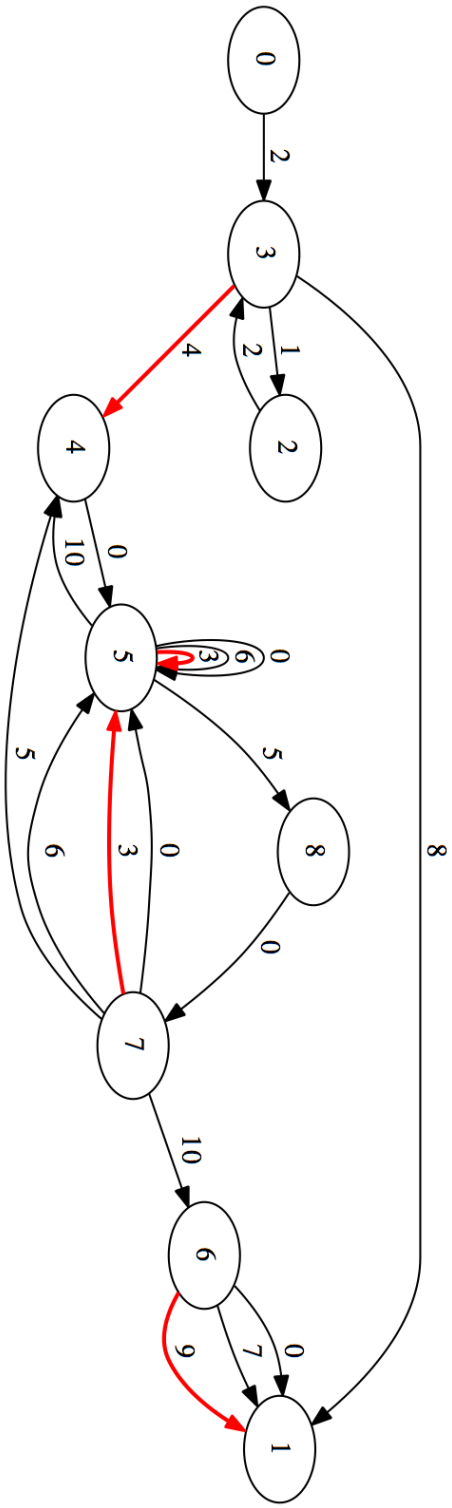

Fig. 8 Final vulnerability graph

from the exploitation of one vulnerability. These new navigations are then analyzed by the *crwl* function.

The *crwl* function is presented in figure 10. This function is used to continue the crawling of the Web site starting from a specified navigation provided as an input parameter. The *remain* variable holds the set of navigations that have just been discovered but not been crawled yet. This algorithm ends when there are no more navigations to crawl, which means that the *remain* set is empty, or when the maximum exploration depth is reached. Before starting the crawling of any navigation, the cookies are removed. Then, each request of the navigation is executed step by step, from the first one to the last one. The content of the response associated to this last request is analysed in order to identify new HTML links. These are provided by the *get_response* function (line 9 of the figure 10). The analysis is only made for this last request because all the requests of the navigation except the last one have already been analyzed during previous iterations of the algorithm. Each HTML link identified in this response is used to build a new navigation of the site (using the concatenation operator $\oplus$). This operation is repeated iteratively until all the HTML links have been discovered or until the maximum exploration depth is reached. The set of the new navigations discovered by *crwl* is saved in the *traces* variable.

The *main* function of figure 11 executes the two previous functions. During the first iteration, the *crwl* function discovers the Web site considering only "normal" requests. Then, thanks to the RPNI algorithm, a graph is built from the navigations obtained. Each state of the graph is then analysed in order to identify vulnerabilities (*search_vulns* function). At the end of the first iteration, all the navigations including at most one vulnerability are identified. The exploitation of these vulnerabilities may enable to discover new parts of the Web site. Then, the next iteration begins. The beginning of each iteration i of the *main* function corresponds to the exploration of a sub-part of the site, more precisely the part that has been discovered thanks to the exploitation of the $(i-1)$th vulnerability of the navigation. At the end of the crawling of each iteration i, the vulnerability identification step is carried out, considering navigations including $i-1$ vulnerabilities and ending with a normal request. At the end of the ith iteration, all the navigations including at most i vulnerabilities are obtained. Finally, the *main* algorithm stops when the maximum exploration depth is reached or when no further vulnerability can be identified.

Require: $path$ = navigation
Ensure: $vulns$ = set of navigations
1: $vulns \leftarrow \emptyset$
2: **for** $class \in vuln\_classes$ **do**
3: $\quad vulns \leftarrow vulns \cup wasapy(path, class)$
4: **end for**
5: **return** $vulns$

Fig. 9 Vulnerability identification algorithm – *search_vulns*

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

Fig. 10 *Crawling* algorithm – *crwl*
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: $\quad$ **for** $nt \in ntraces$ **do**
7: $\quad\quad$ **if** $|nt| < d_m$ **then**
8: $\quad\quad\quad traces \leftarrow traces \cup crwl(nt, d_m)$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad H \leftarrow RPNI(traces)$
12: $\quad new\_nodes \leftarrow H.N \setminus G.N$
13: $\quad ntraces \leftarrow \emptyset$
14: $\quad$ **for** $nn \in new\_nodes$ **do**
15: $\quad\quad ptnn \leftarrow shortest\_path(H.R, H.S, nn)$
16: $\quad\quad$ **if** $|ptnn| < d_m$ **then**
17: $\quad\quad\quad nptv \leftarrow search\_vulns(ptnn)$
18: $\quad\quad\quad$ **for** $np \in nptv$ **do**
19: $\quad\quad\quad\quad new\_vuln \leftarrow np_{|np|}$
20: $\quad\quad\quad\quad vulns \leftarrow vulns \cup \{new\_vuln\}$
21: $\quad\quad\quad$ **end for**
22: $\quad\quad\quad ntraces \leftarrow ntraces \cup nptv$
23: $\quad\quad$ **end if**
24: $\quad$ **end for**
25: $\quad G \leftarrow H$
26: **end while**
27: **return** $(G, vulns)$

Fig. 11 Main algorithm

6 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 6.1 presents conventions and abbreviations. Subsection 6.2 presents the first experiments carried out in order to assess our approach. Five Web applications including SQL vulnerabilities are used. We purposely injected these vulnerabilities to calibrate `Wasapy`. Subsection 6.3 presents 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 [16]. We reported some of these results in order to compare these scanners with `Wasapy`. Subsection 6.4 presents the summary of all these experiments.

6.1 Notations

The results of our experiments are presented in different tables. We use the following conventions 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
SQLi	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.

6.2 Experiments with modified applications

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

- `phpBB-3`: This application[9] is a forum manager written in PHP and using a MySQL database. We modified the authentication form of the application by inserting 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[10] written in PHP, is designed to protect the access of a Web site through authentication. Valid pairs for this authentication are stored in a MySQL database. A vulnerability (`v2`) similar to `v1` was purposely injected.
- `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 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 inserted 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[11]. 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[12] is written in PHP and uses MySQL server. A vulnerability `v9`, similar to `v3`, was 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. 12 Vulnerability detection results for modified applications

Figure 12 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 confirms that the vulnerability detection clustering algorithm presents a better coverage than the pattern matching algorithm for these vulnerability classes.

[9] http://www.phpbb.com
[10] http://www.01php.com/fiche-scripts-126.html
[11] French funded ANR's program ARPEGE(2009-2011).
[12] http://www.dvwa.co.uk

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 2, Skipfish considers that 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[13] 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 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.

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

6.3 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 non 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 [16]. 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.

For our experiments, we selected five Web applications (most of them tested in [16]), 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[14] is a configuration Webforum, which uses PHP 4.0.0 session capabilities to authenticate users and a MySQL database.
- Seagull[15] 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[16] that implements a Text-To-Speech System based on PHP (4.3.0 or newer) and MySQL (4.1.2 or newer).
- Riotpix[17] 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[18] 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.

[14] http://Webscripts.softpedia.com/script/Snippets/Cyphor-27985.html
[15] http://seagullproject.org/
[16] http://fttss.sourceforge.net
[17] http://www.riotpix.com/
[18] http://www.pligg.com/

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. 13 Vulnerability detection results for `Cyphor` application

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. 14 Vulnerability detection results for `Seagull` application

We inspected manually the results provided by each scanner to have more confidence on the number of detected vulnerabilities and false positives.

Figure 13 presents the results for `Cyphor` application. All the scanners detected 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 14 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 accessible from the client interface. 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 [16]. Hence, some results associated to the three commercial scanners considered are available (cf. figure 15). The commercial scanners do not detect the OS commanding vulnerability, which is the only vulnerability known of

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. 15 Vulnerability detection results for `Fttss` application

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. 16 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. 17 Vulnerability detection results for `Pligg` application

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 16), 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 17), 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.

6.4 Summary

The main lessons learned from 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, `Wasapy` 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 non 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.

7 Conclusion

In this paper, we proposed a new methodology that is designed to automatically identify Web applications vulnerabilities and to exhibit attack scenarios targeting these applications. This methodology is based on the dynamic analysis of the application following a black box approach. It is also aimed at reducing the number of false positives by providing the queries that allow the successful exploitation of the detected vulnerabilities. This advantage is twofold since the effective exploitation of vulnerabilities also allows us to discover new pages in the Web application that we could not reach before. These new pages may contain new injection points and possibly new vulnerabilities. To validate and evaluate our approach, we have carried out two sets of experiments on different types of applications. This approach led also to the development of a new vulnerability scanner called `Wasapy`.

Various directions will be considered for extending the results obtained so far. First, regarding the proposed approach for detecting vulnerabilities and generating attack scenarios based on the elaboration of the Website navigation graph, optimisations would be necessary to master the size of the graph, especially when it is to be applied to complex Web sites. Another perspective would be to enrich the grammars implemented in `Wasapy` to allow the generation of a larger variety for injections covering the vulnerabilities included so far, as well as new vulnerabilities.

References

1. 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
2. E.Alata, M.Kaaniche, V.Nicomette and R.Akrout, An automated vulnerability-based approach for web applications attack scenarios generation", LADC-2013: Latin-American Symposium on Dependable Computing, 02-05 Avril 2013, Rio De Janeiro, Brazil. 9p.
3. A.Dessiatnikoff, R.Akrout, E.Alata, M.Kaaniche, V.Nicomette, A clustering approach for web vulnerabilities detection, IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2011), Pasadena (USA), 12-14 Dcembre 2011, 10p.
4. 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.
5. 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.
6. 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.
7. 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.
8. A. Doupé, M. Cova, and G. Vigna, "Why Johnny can't pentest : An analysis of black-box Web vulnerability scanners", *Proc. DIMVA 2010.*
9. 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.
10. Levenshtein, V., *Leveinshtein distance*, 1965 `http://en.wikipedia.org/wiki/Levenshtein_distance`[accessed on 02/22/10]
11. S. C. Johnson, "Hierarchical Clustering Schemes", in *Psychometrika Journal*, pp. 241-254,Volume = 2, 1967.
12. 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.
13. E. Gutesman, "gFuzz: An Instrumented Web Application Fuzzing Environment", *Hack.Lu '08*, Luxembourg, 2008.
14. P. Dupont,"Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG Method", Proc. of the 2nd Intl. Colloquium on Grammatical Inference and Applications (ICGI '94), pp 236-245, London, UK, 1994
15. Dupont, P., "Incremental regular inference", Proc. of the Fourth Intl. Colloquium on Grammatical Inference and Applications (ICGI '96), pp 222-237, 1996.
16. `http://anantasec.blogspot.com/` [accessed on 12/9/10]