# An Automated Approach to Generate Web Applications Attack Scenarios

Eric Alata, Mohamed Kaâniche, Vincent Nicomette, Rim Akrout

# An automated approach to generate Web applications attack scenarios

Eric Alata*‡, Mohamed Kaâniche*† Vincent Nicomette*‡ Rim Akrout*‡
*CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France*
*Université de Toulouse, LAAS*
*F-31400 Toulouse, France*
*Université de Toulouse, INSA, LAAS*
*F-31400 Toulouse, France*
*Email: {firstname.name}@laas.fr*

***Abstract*—Web applications have become one of the most popular targets of attacks during the last years. Therefore it is important to identify the vulnerabilities of such applications and to remove them to prevent potential attacks. This paper presents an approach that is aimed at the vulnerability assessment of Web applications following a black-box approach. The objective is to detect vulnerabilities in Web applications and their dependencies and to generate attack scenarios that reflect such dependencies. Our approach aims to move a step forward toward the automation of this process. The paper presents the main concepts behind the proposed approach and an example that illustrates the main steps of the algorithm leading to the identification of the vulnerabilities of a Web application and their dependencies.**

***Keywords*-Security, vulnerability scanner, vulnerability detection algorithm**

## I. Introduction

Web applications are extremely popular today because they provide a vast variety of services and can be easily maintained and updated. Unfortunately, Web vulnerabilities have also become in the recent years a major threat to computer systems security. This is reflected by various vulnerability statistics and threat reports (see 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 [22]). The recent experimental study published in [21] based on the monitoring of more than 8,000 Web sites also indicates that Web applications include a large number of residual vulnerabilities and a high percentage of such vulnerabilities are generally not fixed. To cope with these threats, several solutions have been developed to prevent, detect or tolerate potential intrusions such as firewalls, Web vulnerability scanners, and intrusion detection system. Such techniques can be used during the development phase and also during the operation phase.

A large number of vulnerability scanners have been developed [3], including commercial products (e.g., Acunetix WVS, WebInspect, AppScan), open source tools (e.g., W3af, Wapiti, Nikto, Skipfish), as well as other publicly available tools such as Secubat [2]. Given the high complexity of current Web sites that include different technologies and a large number of pages to be analyzed, these tools provide a useful support for the identification of vulnerabilities in such systems. Nevertheless, several recent reports have shown that state-of-the art Web application scanners fail to detect a significant number of vulnerabilities and have pointed out the need to improve their detection effectiveness, and to enhance the automation capabilities offered by such tools [4], [5], [6], [7], [8], [9].

Generally, existing vulnerability scanners are not designed to automatically identify causal dependencies between vulnerabilities and to build attack scenarios that include the exploitation of multiple vulnerabilities. Indeed, each vulnerability is generally detected independently. In our context a causal dependency exists between two vulnerabilities A and B if B can be exploited only once A is exploited. The identification of such attack scenarios should allow the detection of some hidden vulnerabilities that can only be revealed once other vulnerabilities are detected and exploited. Such information should be useful to improve the security of complex Web applications and to remove residual vulnerabilities that are not easy to detect with traditional Web scanners.

The automatic identification of attack scenarios including multiple dependent vulnerabilities requires the development of new techniques allowing the automated exploitation of vulnerabilities during the dynamic execution of an application. This is not generally supported by existing Web vulnerability scanners.

This paper proposes a methodology that is aimed at addressing this gap by automatically building complex attack scenarios for Web applications based on the dynamic execution of the application following a black-box approach. Our methodology builds on the `Wasapy` tool presented in [10] and consists of three main steps: 1) the identification of possible navigation traces through a Web application, 2) the creation of a reduced graph called navigation graph including all the identified navigation traces, thus integrating the different possible ways for a client to activate the Web application, and 3) the identification and exploitation of potential vulnerabilities by sending specially crafted requests through the different links available from each node of the

navigation graph. This process has to be run iteratively as the actual exploitation of a vulnerability may reveal new possible browsing paths of the Web application. This iterative process enables to exhibit causal dependencies between vulnerabilities and to identify complex attack scenarios composed of ordered execution of elementary exploitations of these vulnerabilities.

The paper is structured into 6 sections. Section II briefly presents related work connected to this paper. Section III presents the main concepts of our methodology to establish attack scenario for Web applications. Section IV presents a concrete example of a Web application and proposes a manual step-by-step execution of our methodology. This section enables to informally illustrate our approach. Section V briefly introduces some interesting hints regarding the complexity of the algorithms of our methodology. Finally Section VI concludes this paper and discusses future work.

## II. Background

The research work proposed in this paper is related both to the Web vulnerability scanners technologies and to attack graph methodologies.

Two main approaches are generally adopted by Web security scanners in order to detect the presence of a vulnerability in a Web application following a black-box approach[1]. The first one relies on an error pattern matching algorithm (`W3af`[2], `Wapiti`[3] and `Secubat`[2] are examples of tools that adopt this approach) and the second one on the analysis of similarities between the pages returned by the server (e.g., the `skipfish`[4] scanner developed by Google adopts this approach). The error pattern matching approach consists in sending specially crafted requests to the application and looking for specific patterns in the responses (for instance, database error messages for SQL injections). The basic idea is that the presence of an error message in a HTML response page means that the corresponding request has not been sanitized by the application. The similarity approach consists in sending to the Web application various specifically crafted requests corresponding to possible injection attacks and comparing the similarity of the corresponding responses using a textual distance, in order to exhibit the responses that indicate the presence of a vulnerability.

Whatever the approach used, these tools 1) detect each vulnerability independently and 2) do not automatically provide the specific attacks allowing the exploitation of the identified vulnerabilities. As a consequence, they are not designed to provide attack scenarios including the exploitation of multiple vulnerabilities. The methodology proposed in this paper uses a Web vulnerability scanner (so called `Wasapy`, briefly presented in Section III) that is able to detect and actually exploit different types of Web vulnerabilities including SQL injections, OS commanding, File Include and XPath.

Building complex attack scenarios is the objective of many research papers related to *attack graphs*. An attack graph is a formalism that enables to formally represent the combination of vulnerabilities that may be exploited by an intruder to break into a system. An attack graph models security vulnerabilities in a system and the possible attack scenario (through paths in the graph) that may be used by an intruder to achieve a specific goal. Various forms of attack graphs have been proposed in the literature, such as [11], [12], [13], [20]. Attack trees [17], [19] and Attack-Defense trees [18] have also received increasing attention by the security community to model attack scenarios. Nevertheless, to our knowledge, the problem of how to automatically generate attack scenarios from the dynamic execution of Web applications has not been addressed so far. In [14], the authors propose to use a use-case graph for the detection of access-control flows in Web applications but this work is focused on one specific vulnerability class and is not aimed at providing attack scenarios including several classes of vulnerabilities.

To address this gap, we propose in this paper an approach to build automatically an attack navigation graph that defines attack scenarios in order to assess Web application security, based on the dynamic analysis of the Web application following a black-box approach. We consider the point of view of the Web application developers who are interested in analysing the behavior of their application in a context where the attackers do not have access to the source code, which is generally the case. This approach is described in the next section.

## III. Overview of the approach

### A. 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 scenario* a sequence of requests sent by a client (a request consists in the activation of a HTML link). 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 in the browser. A request sent by a browser will provoke a change of the current navigation state. The set of all the possible navigations can be represented by a *navigation graph*. Each node of the graph corresponds to a navigation state. An edge between two navigation states S1 and S2 exists if a request whose execution leads to the transition from S1 to S2 can be sent by the client. An edge may correspond to a non malicious request (called "normal" request) or to a request that exploits a vulnerability of the Web site. A *vulnerability graph* is a particular case of a

[1] White-box approaches are not addressed in this paper
[2] http://w3af.sourceforge.net
[3] http://wapiti.sourceforge.net
[4] http://code.google.com/p/skipfish/

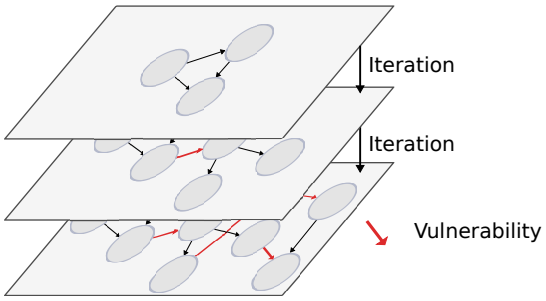


Figure 1. Principle of the algorithm

navigation graph that includes edges corresponding to the exploitation of vulnerabilities.

A navigation graph is different from a traditional URL graph composed of HTML pages describing the structure of a Web site. Each node of such a graph generally corresponds to a HTML page of the site and an edge between two nodes identifies a link that enables to access the second page from the first one. The difference mainly consists in the fact that a navigation graph describes the different possible scenarios for a user or an attacker to navigate through the URL graph. Also, it is important to note that a navigation state does not only depend on the currently accessed HTML page. Indeed, a client can access a HTML page several times while being in different navigation states. For instance, let us consider an e-business Web site. It is possible to activate the payment page after the client 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. 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.

Our algorithm starts from the root URL of a Web site. The navigation graph is built set by step, by identifying the different possible navigations of the site resulting from the execution of "normal" requests but also through the exploitation of vulnerabilities of the Web site. As the exploitation of a vulnerability may reveal new pages, the construction of the navigation graph is thus iterative. Our approach is thus composed of two steps, so-called *crawling* step and *vulnerability identification* step, that are iteratively executed. These two steps are presented in Figure 1. The transition from iteration $i$ to iteration $i+1$ comes from the exploitation of one or more vulnerabilities (detected during iteration $i$) that lead to the creation of a new navigation graph including the new navigation states and edges revealed by the exploitation of the identified vulnerability(ies). The next sections present these two steps as well as the whole algorithm.

### B. Crawling

The first step of the algorithm consists in *crawling* the Web site to identify the different states resulting from the navigation through the site. The algorithm starts from an initial URL by removing all the cookies of the client in order to ensure that the different navigations are independent. Then, it navigates through the site and memorizes all the requests sent to the site. If the currently visited page includes several HTML links, one of these links is chosen and the others are memorized to be analyzed later. 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 of this threshold is an input parameter of the algorithm.

When this threshold is reached or when the crawling does not reveal new navigation states, the whole set of memorized requests represents one navigation of the site. This step is repeated to take into account all the HTML links that were memorized and not used so far, until all the possible navigation traces are obtained. This set of navigation traces is the result of the crawling phase and is used to build a first version of the navigation graph. The main issue when building this graph is to obtain a minimal graph.

The construction of the minimal graph from the navigation traces 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 [15]. The RPNI algorithm [16] we chose, belongs to the second category. This widely used algorithm presents a polynomial time-based complexity, and is quite simple to implement.

Let us note that, at the end of the crawling step, the only possibility to enrich the navigation graph is to consider a vulnerability whose exploitation may add a new edge and a new node to the graph. Thus, the automaton obtained from this crawling phase is an input of the next step of our approach aimed at vulnerability identification.

### C. Vulnerability identification

The second step of our approach aims at identifying vulnerabilities that may be exploited, based on the navigation graph. The identification of vulnerabilities is performed for each edge of the minimal automaton resulting from the first step. As the number of edges is much smaller than the number of requests sent during the crawling phase, the identification is much faster. We assume that the exploitation

of a vulnerability depends on a particular navigation state and is independent of the requests that lead to this state. Thus, we only focus on one path from the initial state that leads to this state, considering the shortest path.

As the algorithm is iterative, we have to pay attention to not test twice the edges of a same state. Thus, for each iteration, we only analyze the output edges of new states, compared to the previous iteration. For each of these edges, the vulnerability identification is performed on each request parameter (called injection point) and is carried out by `Wasapy`.

`Wasapy` allows the automated detection and exploitation of different types of Web vulnerabilities including SQL injections, OS commanding, File Include, XPath. The vulnerability detection and exploitation algorithm implemented in `Wasapy` is briefly presented in the following considering the example of SQL injections (for more details, see [10]). It is based on: i) the automatic generation of inputs based on a grammar that is specific to targeted vulnerabilities, and ii) the classification of the corresponding responses using data clustering techniques.

Three sets of requests are submitted at each injection point:

- $R_{\text{r}}$ corresponds to requests with randomly generated data. These are very likely to generate error pages.
- $R_{\text{ii}}$ corresponds to syntactically invalid SQL injections that are designed to lead to unsuccessful executions.
- $R_{\text{vi}}$ corresponds to syntactically valid SQL injections. The main issue is to automatically determine whether they lead to a *successful execution* page or to a failed execution page. To do so, these responses are compared to those associated to $R_{\text{r}}$ and $R_{\text{ii}}$ using a similarity distance and data clustering.

Let us note $S_{\text{r}}$, $S_{\text{ii}}$ and $S_{\text{vi}}$ the responses associated to $R_{\text{r}}$, $R_{\text{ii}}$ and $R_{\text{vi}}$ respectively. Then, $R_{\text{vi}}$ requests whose responses are not similar to any of the responses from $S_{\text{ii}}$ and $S_{\text{r}}$ are considered valid SQL injections. For more details, see [10].

### D. Algorithm

The algorithm is presented in Figures 2, 3 and 4. A *trace* corresponds to a navigation trace through the Web site.

The $search\_vulns$ function in Figure 2 takes a navigation trace as input parameter. The latest request of this navigation trace is analyzed to identify vulnerabilities, based on `Wasapy`, considering different vulnerability classes (SQL injections, XPATH injections, OS commanding, etc.). This function returns a list of navigation traces and, for each of them, the identified vulnerabilities. Each new navigation trace includes one more navigation state that results from the exploitation of one vulnerability. These new states are then analyzed by the $crwl$ function.

The $crwl$ function of Figure 3 enables to browse the Web site from a navigation provided as input parameter. Before starting the analysis of this navigation, the cookies are removed. Then, the navigation is executed step by step, from its first request to the last request (using the $get\_response$ function). The content of the response associated to this last request is analysed in order to identify new HTML links. The analysis is only made for this last request because it corresponds to the new navigation state discovered by the vulnerability identification step executed before. Each HTML link identified in this response is used to build a new navigation of the site. This operation is repeated iteratively until all the HTML links have been discovered or until the maximum exploration depth $dm$ is reached. The set of the new navigations discovered by $crwl$ is saved in the $traces$ variable.

The $main$ function of Figure 4 executes the two previous functions. In this figure, the symbols $N$ and $N'$ correspond to the set of nodes of the graphs $G$ and $G'$. During the first iteration, the $crwl$ function explores the Web site while only considering "normal" requests. Then, thanks to the RPNI

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

Figure 2. Vulnerability identification algorithm – $search\_vulns$

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

Figure 3. *Crawling* algorithm – $crwl$

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

Figure 4. Main algorithm

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 navigation traces 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 $i$th iteration, all the navigation traces including at most $i$ vulnerabilities are obtained. Finally, the $main$ algorithm stops when the maximum exploration depth is reached or when no additional vulnerabilities can be identified.

## IV. Example

In order to illustrate our approach, we developed as an example an e-commerce Web site for selling books, using the `php` language and a `msysql` database. This site is a simple proof of concept but it uses technologies and a structure similar to "real" Web sites. Figure 5 presents the

```
T1 :  → index.html
T2 :  → index.html  → about.html
T3 :  → index.html  → login.php
T4 :  → index.html  → login.php  → index.html
```

Figure 6. List of navigation traces of the first iteration

URL graph of HTML pages describing the structure of the Web site. A page is represented by an icon. An edge between two pages corresponds to 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.

This site includes three vulnerabilities. The pages including these vulnerabilities are identified by a star on Figure 5. 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 dump the content of the database. The last vulnerability is associated to the page `check.php`. It allows an attacker to pay all the ordered products without providing any credit card number. This vulnerability cannot be exploited unless some products have been added to the virtual shopping cart.

In the following, we illustrate the execution of the algorithm considering a graph exploration depth threshold of 7. During the execution of the crawler for the first iteration of our algorithm, the first trace obtained corresponds to the navigation through HTML pages without exploiting any vulnerability. The corresponding graph is presented in Figure 7. Next, the vulnerability identification phase starts and analyzes each edge of this graph. For each edge under test, a navigation scenario is built, representing the path from the first edge until this edge under test. The set of all these traces is presented in Figure 6.

For each of these scenarios, `Wasapy` is then executed. At this stage, the only accessible vulnerability corresponds to a SQL injection during the authentication, i.e., trace $T_3$. Thus, at the end of the first iteration, the vulnerability graph presented in Figure 8 is obtained. As this graph includes one vulnerability, and thus, a new node and a new edge, the second iteration begins.

During the second iteration, the crawler identifies the pages that can be accessed as a consequence of the exploitation of the vulnerability identified during the previous iteration. In order to reach these pages, it is necessary to cross the edges `index.html` and `login.php`*. The set of the traces executed by the crawler for this second iteration contains 65 traces. These traces reach either `display.php`, `add.php`, `delete.php`, `buy.php` or `check.php`. Once again, the vulnerability identification

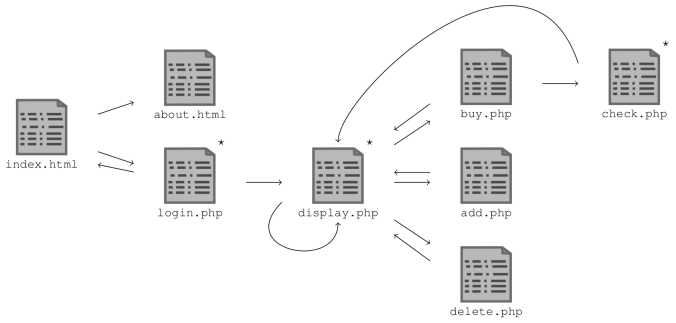



Figure 5. Structure of the Web site

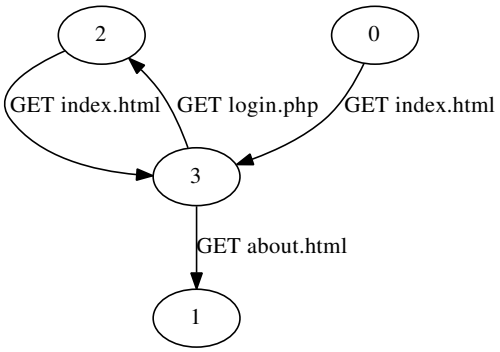



Figure 7. Navigation graph of 1st iteration

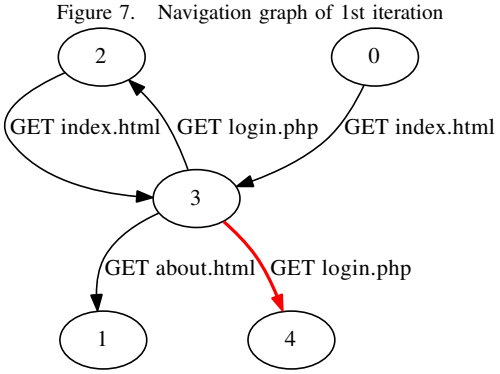



Figure 8. Vulnerability graph of 1st iteration

phase is executed for each new edge of this second iteration. The graph obtained at the end of the algorithm execution is presented in Figure 9. The algorithm stopped after 6 iterations, which means that no more vulnerabilities have been discovered during the sixth iteration.

This graph contains 8 nodes and 19 edges. Nevertheless, it represents 241 traces. The initial graph generated from all the traces without using the RPNI reduction algorithm contains 353 edges. The vulnerability graph is more compact than the initial graph and leads to a more effective vulnerability search. In addition, this graph provides enough information to deduce causal dependencies between vulnerabilities. For instance, the vulnerability associated to edge 3 cannot be identified and exploited before the exploitation of the vulnerability associated to edge 4. Thus, there is a causal dependency between these two vulnerabilities.

The number of executed traces used to obtain this graph is important. Moreover, each of these traces includes several requests. As a consequence, the number of requests is also important. It seems thus important to study and estimate the complexity of our approach.

Based on the graph presented in Figure 9, one can extract different attack scenarios according to specific criteria. For example, if one wants all the attack scenarios containing two vulnerabilities, we need to extract from the graph the set of paths that go through edges 4 and 3, or 4 and 9. An example of such scenarios is given in figure 10.

## V. Complexity of the algorithm

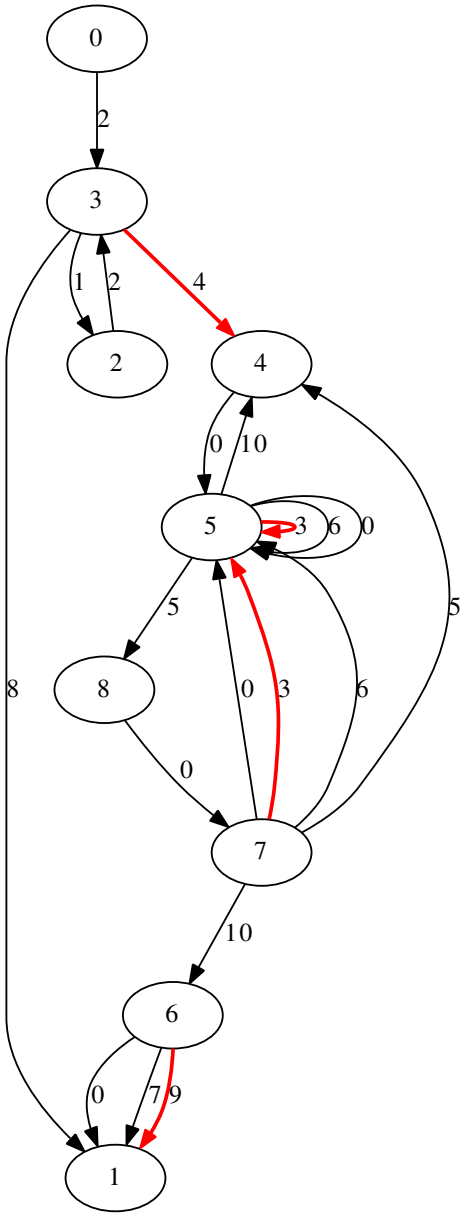
In order to analyze the complexity of the proposed algorithm, figure 11 presents the size of the navigation graph corresponding to different values for the exploration depth threshold. For these results, the algorithm is run on a MacOSX laptop for our algorithm and a Linux desktop (CPU Core2 Duo processor) for the server under test. The execution duration increases with increasing values of the threshold. However, the resulting reduced graph may not change significantly (see, for instance, thresholds 4, 5 and 6). Increasing the threshold to $d + 1$ does not necessarily allow to reach new pages or new vulnerabilities that cannot be reached with threshold $d$. The threshold depends on the site. Empirical experience shows that a threshold around 8 is generally sufficient and enables to access all the pages of most Web sites.

The execution duration of the algorithm increases according to threshold. For our example, the execution time remains reasonable.

To analyse the complexity of the proposed algorithm when applied to other Web sites, let us focus on the most consuming elementary operation: the $get\_response(trace_i)$

```
S1 :   → index.html   → login.php*   → display.php   → display.php*
S2 :   → index.html   → login.php*   → display.php     → add.php      → display.php   → display.php*
S3 :   → index.html   → login.php*   → display.php     → add.php      → display.php     → buy.php      → check.php*
```

Figure 10. Example of attack scenario from the graph




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

Figure 9. Final vulnerability graph

| Depth Threshold | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| Nodes | 2 | 5 | 6 | 6 | 6 | 9 | 9 |
| Edges | 4 | 6 | 11 | 11 | 11 | 19 | 19 |
| Vuln. | 1 | 1 | 2 | 2 | 2 | 3 | 3 |
| Duration | 2s | 2s | 5s | 11s | 20s | 50s | 1m55s |

Figure 11. Experimental results

operation, which requires to send one request on the network and wait for the corresponding response. The worst execution case is considered, in order to provide the most pessimistic estimation.

Let $n$ be the maximum number of links in HTML pages of the Web site. Let $d$ be the maximum exploration depth and $l$ be the number of *Wasapy* executions for each scenario, in order to discover vulnerabilities. The Web site is a tree of arity $n$ and depth $d$. Such a tree includes at most $n^d \times d$ nodes. As a consequence, the number of generated traces is less than $n^d \times d$. The distance between each of the nodes and the root being less than $d$, the number of requests to the Web site is thus less than $n^d \times d \times d \times l$. We can consider that the complexity of the algorithm is $n^d$, which is in adequation with the evolution of the duration presented in the figure 11. Further details about how the complexity of the algorithm is assessed are described in [23]

An improvement of the execution time can be obtained by taking into account the fact that for many Web sites, several URLs implement similar functions and can be considered as equivalent. This is typically the case of e-commerce Web sites where the HTML pages pointing to different products have a similar structure. More precisely, in a Web site that contains various information, URLs used to process this information by the same function often follow a specific format[5]. By considering the equivalence between URLs based on their format, it is possible to dramatically reduce the vulnerability graph of the site by focusing on the unique URLs, and hence the number of requests to be sent to build the navigation graph can be significantly decreased.

## VI. Conclusion

In this paper, we have proposed a novel methodology aiming at identifying potential attack scenarios targeting Web applications based on the dynamic analysis of the application following a black-box approach. Our methodology is able to automatically exhibit causal dependencies between vulnerabilities and to identify attack scenarios exploiting in a ordered way these vulnerabilities. This approach has been

[5]For instance, the two URLs site.com/a?id=2 and site.com/a?id=3 use the same format.

successfully tested and illustrated on a simple but realistic Web application including several vulnerabilities. The first results are promising. They have to be confirmed with experiments on various and more complex Web applications to experimentally assess the scalability of our approach. These experiments are currently carried out. At this stage of our research, the proposed approach is able to addresses the vulnerabilities that are covered by the Wasapy tool, including SQL, Xpath, and LDAP injections, and also OS Commanding and File Include vulnerabilities. Future work will also be focussed on extending our approach to cover other types of vulnerabilities such as Cross-Site Scripting vulnerabilities.

## ACKNOWLEDGMENT

This work was supported by the *Agence Nationale de la Recherche* through project *DALI* and by the french project *Secure Virtual Cloud*.

## REFERENCES

[1] Mitre, "2011 CWE/SANS Top 25 most dangerous software errors", Document version 1.03, September 2011, http::://www.cwe.mitre.org/top25

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

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

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

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

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

[7] AnantaSec: Web Vulnerability Scanners Evaluation (January 2009), http://anantasec.blogspot.com/2009/01/web-vulnerability–scanners-comparison.html

[8] Suto, L.: Analyzing the Accuracy and Time Costs of Web Application Security Scanners, Feb 2010.

[9] E.Fong, V.Okun, "Web application Scanners: Definitions and Functions", In Proc. of HICSS-40Conferee, Hawaii, USA, Jan 2007.

[10] A.Dessiatnikoff, R.Akrout, E.Alata, M.Kaâniche, V.Nicomette, "A Clustering Approach for Web Vulnerabilities Detection", Proc. 17th IEEE Pacific Rim Intl. Symposium on Dependable Computing (PRDC'11),Pasadena, CA, USA, 2011

[11] Ingols, K., Lippmann, R., Piwowarski, K.: Practical attack graph generation for network defense. In: 22nd Annual Computer Security Applications Conference (ACSAC), Miami Beach, Florida (December 2006)

[12] Phillips, C., Swiler, L.P.: A graph-based system for network-vulnerability analysis. In: NSPW 1998: Proc. of the 1998 workshop on New security paradigms, pp. 71–79. ACM Press, New York (1998)

[13] Sheyner, O., Haines, J., Jha, S., Lippmann, R., Wing, J.M.: Automated generation and analysis of attack graphs. In: Proc. of the 2002 IEEE Symposium on Security and Privacy, pp. 254–265 (2002)

[14] G.Noseevich, A.Petukhov, "Detecting Insufficient Access Control in Web Applications", In SysSec Workshop, pp. 11–18, Los Alamitos, CA, USA, 2011

[15] 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

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

[17] B., Schneier, Modeling security threats, Dr Dobb's journal, December 1999.

[18] Kordy, B., Mauw, S., Radomorovic, S., Schweitzer, P.,Foundations of Attack-Defense Trees, Proc. of Formal Aspects of Security and Trust (FAST 2010), LNCS Volume 6561, pp. 80–95.

[19] Mauw, S., Oostdijk, M., Foundations of attack trees, Information Security and Cryptology-ICISC 2005, LNCS Volume 3935, pp. 186–198.

[20] Noel, S., Jajodia, S., and Singhal, A., Measuring Security Risk of Networks Using Attack Graphs, International Journal of Next-Generation Computing, **1 (1)**, pp. 135–147, 2010.

[21] Grossman, J., The State of Website Security, IEEE Security and Privacy, July-August 2012, **10 (4)**, pp. 91–95.

[22] 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

[23] 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.