Automatic Detection of Vulnerabilities in Web Applications using Fuzzing

Miguel Filipe Beatriz
miguel.beatriz@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2014

Abstract

Automatic detection of vulnerabilities is a problem studied in literature and a very important concern in application development with security requirements. Fuzzing is a software testing technique, automated or semi-automated, that involves injecting a massive quantity of semi-random inputs in software in order to find security vulnerabilities. Many vulnerability detection techniques need manual analysis from specialized people in order to confirm if there any vulnerabilities. To solve this problem, it was decided to develop a system that automatically detects vulnerabilities in web applications using fuzzing. Detecting vulnerabilities in web applications is different than detection in other types of software. This happens because web applications contain many back-end components, that causes specific vulnerabilities, therefore, it is convenient monitoring these components. This works presents a framework for fuzzing web applications. In this work, monitoring is made inside each web application component. The framework detects a representative set of web application vulnerabilities: SQL injection; remote and local file inclusion; reflected and stored cross-site scripting. Our SQL injection detection mechanism is able to detect even subtle attacks of this category presented recently. We present an experimental evaluation of the framework, using vulnerable code samples and open source web applications.
Keywords: Security, Discovery of vulnerabilities, Web applications, Fuzzing

1. Introduction

Vulnerability detection is a theme widely studied in the literature and a very important concern in software development with security requirements [1, 2, 3]. Many software vulnerabilities have been found through fuzzing [4].

Fuzzing [5, 6] is a software testing technique, that involves injecting repeatedly semi-random inputs in software in order to be processed by the target application and to check if there is any behavior different than expected. If so, it is possible that a certain input exploited a bug or a vulnerability. Also, fuzzing is a semi-automated or automated process that consists in sending values that are more likely to exploit vulnerabilities. There are two categories of fuzzers: *mutation-based* and *generation-based* [5, 6]. Mutation-based fuzzers create test cases by applying small changes in inputs without changing their structure. Generation-based fuzzers create test cases through target knowledge.

Many web applications have a complex architecture, which a challenge and also an opportunity to detect if some injection exploits a vulnerability. In non-web applications, like network services or command-line commands, fuzzers only try to send inputs but is necessary to the users understand if there is any abnormal behavior in target software [7]. Other mechanisms try to detect abnormal behavior through crash observation, monitoring CPU time or memory consumption [3]. However, users need to understand if these anomalies are vulnerabilities in the software. Many web application vulnerabilities are related with back-end components, for example, database management system (DBMS) or file system, which complicates monitoring but also allows doing it at these points.

This paper presents a framework for fuzzing web applications that meets this challenge and opportunity. The main emphasis of the work is in monitoring injection effects in target applications, so it is made primarily within its components.

It is important that monitoring occurs within web application components because: 1) it is the point where incoming data flow ends, so is a crucial point to a vulnerability; 2) as last point where the input's flow ends, there are less assumptions made about how inputs are processed, for example, if there is any encoding or validation processes; 3) allows to

use components features in order to detect vulnerabilities and to make an effective detection.

The framework was designed to detect input validation vulnerabilities but in the current version only a subset of them was implemented: SQL injection (SQLI); local/remote file inclusion (LFI/RFI); reflected/stored cross-site scripting (XSS). This subset of vulnerabilities was chosen because SQL and XSS were considered for several years the vulnerabilities with highest risk [8], while LFI/RFI have a special impact in web applications written in PHP. The PHP language is currently used in over 77% of web applications [9] and that is why we considered it. This work also shows how the framework can be expanded in order to detect other input validation vulnerabilities.

The framework was implemented to find vulnerabilities in PHP applications executed by the Zend Engine and uses MySQL as DBMS. It was evaluated through small vulnerable code samples and open source applications taking into account number of vulnerabilities found.

The main contributions of this work are: 1) a fuzzing framework with a set of mechanisms to monitor several attacks; 2) implementation of several mechanisms for detecting vulnerabilities based on lack of input validation. These vulnerabilities are: SQL injection; local/remote file inclusion; reflected/stored cross-site scripting 3) the framework's experimental evaluation taking into account vulnerable code and open source applications.

2. Background

2.1. Detecting vulnerabilities exploitation

To determine if a vulnerability has been exploited, it is necessary to exist monitoring mechanisms or resources that can detect the target system state, especially in case of failures, for example [5]: log files where can be recorded any failure or anomaly in execution; debuggers which allow to identify exceptions in system; codes or messages returned by application; check if connection to the target system persists. However, such mechanisms have not been integrated into fuzzers because do not make an accurate detection of attacks that exploit vulnerabilities in web applications.

2.2. Web application vulnerabilities

This section briefly presents the vulnerabilities considered in the current version of the framework. For each type of vulnerability exists a corresponding attack with the same name, for example, SQL injection attack/vulnerability.

A SQL injection vulnerability [10] allows an attacker to use inputs to mislead the target application in order to build unexpected queries and submit them to database. Such attacks occur when user's input is not validated and it is possible to the attacker changes query's structure by inserting SQL keywords. If this type of attacks have an immediate effect on the target's behavior are called first-order SQL injection. In a second-order SQLI attack, in first place the attacker provides an input to the target application which it will be stored in database; after that, the attacker provides a second input that creates a query to extract that last query which was stored and then creates a second modified query.

A RFI vulnerability allows an attacker to include a file from an external web server. This vulnerability happens due to the lack of validation of user's input, allowing to the attacker include remote content through inclusion functions, for example, function *include* in PHP. A LFI vulnerability is similar to the previous but the file that is included must be present in the application server. In order to perform such attack, in first place, it is necessary to load the malicious file or add malicious content in an existing local file (e.g: log file). After that, it is only necessary include this malicious local file.

A reflected XSS vulnerability exists when the web application trusts in inputs from users and reflects them in a response without validation, sanitization or encoding appropriated. If any of these inputs contain a script, it will be executed in victim's browser with several consequences. A stored XSS vulnerability is based on the same principle of previous vulnerability, only in this case the target application, in first place, stores the data and subsequently reflects it.

2.3. Detection of specific web application vulnerabilities

In this subsection will be discussed several detection techniques of the following vulnerabilities: SQL injection, local/remote file inclusion and cross-site scripting.

There are many approaches to detect SQLI vulnerabilities. Halfond and Orso's solution [10, 11] detects this type of attack through static code analysis and monitoring the application at runtime. Also, the solution proposed in [12], CANDID, is based on the principle that a SQL injection changes the structure of queries. In this solution, by sending benign inputs, authors intend to extract a desired model of queries in order to compare with a model generated from future requests. If there is no match between models, indicates the existence of a SQLI vulnerability. This approach differs from SQLI detection mechanism implemented in this work because CANDID needs to change target application and moreover, to extract queries structure is necessary an external tool to the DBMS that may fail in extracting query's structure.

Regarding to mechanisms to detect RFI vulnerability, they pretend to check if insecure inputs be-

fore reaching critical functions (ex: *include*) are target of any validation, sanitization or encoding processes [13]. If so, input becomes safe, otherwise it may exploit a vulnerability. The solution presented in [14] sends an input that refers to a resource in a remote server that is monitored to determine whether it has received any request from target application. If so, web application is vulnerable because attempted to include the target remote file.

To detect LFI vulnerabilities there is a solution proposed in [14] in which the author proposes to include an executable resource in target application and determine whether there any change in application's behavior.

To detect XSS vulnerabilities there are several mechanisms. Mechanisms referred in [15, 16] are based on static analysis. These mechanisms control program's execution flow and checks if user's input is target of any validation or sanitization processes before being sent as response. Beyond this approach, the solution presented in [17] uses a proxy which analyzes requests to the application and its responses. This solution checks if there is any character in requests that correspond to HTML tags. If so, the mechanism checks if respective response contains the same existing tags in order to check if there any vulnerability in the target application.

3. Framework

It was developed a framework for fuzzing web applications and monitoring their effect through mechanisms embedded in back-end components. Next, it will be presented framework's architecture and its components.

3.1. Architecture

Framework's architecture is represented in Figure 1, where is observed its components:

- **Fuzzer**: generates inputs that are injected at certain entry points of the target web application. Also, it contains a monitoring mechanism of reflected XSS attacks.

- **Web application**: as the name suggests, it is the web application tested by framework. It is inserted in a web server with certain operating system and communicates with many back-end components (e.g database, file system);

- **Server-side language interpreter**: where are included several detection mechanisms of vulnerabilities caused by lack of input validation. These vulnerabilities are divided in three categories according if they interact with the file system, DBMS or OS respectively. In first category includes the mechanisms for monitoring the following vulnerabilities: local/remote file inclusion (LFI/RFI); directory/path traversal (DT/PT); source code disclosure (SCD). In second category is included a mechanism to detect stored XSS attacks. At last, in third category, is included a monitoring mechanism of OS command inclusion attacks (OSCI). Also, there is a monitoring mechanism of server-side language code injection attacks.

- **Database management system**: receives queries from the web application and processes it. It contains a monitoring mechanism of SQL injection attacks.

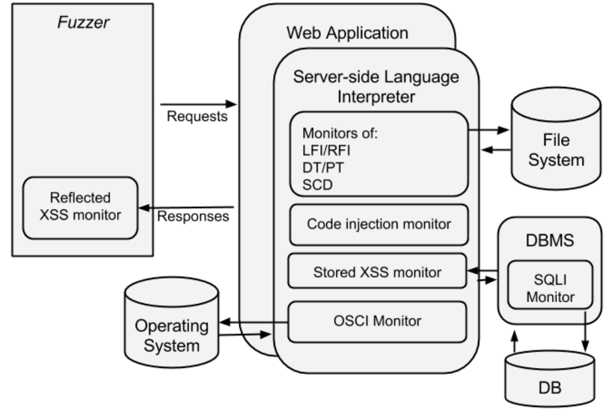

Figure 1: Framework's architecture showing where the monitoring is performed

3.2. Features

Next, it will be presented all features existent in each component of the developed framework.

3.3. Fuzzer

The fuzzer is based on inputs generation through fuzzing vectors, inserting itself in category of iterative fuzzers. These fuzzing vectors, consists in a set of inputs previously selected, containing several existent attacks signatures. Also, the fuzzer applies mutation mechanisms on each element of these vectors. This mechanism, randomly, selects a subset of input's characters to be subject of small changes (e.g: characters substitution), thereby generating a wider variety of inputs that may unexpectedly exploit vulnerabilities.

The fuzzer allows to put target application in a certain state before sending the inputs, inserting itself in category of in-memory fuzzers. So, it is possible, manually, define a sequence of requests to be sent to put the target application in a given state or to reset target application's state after sending a certain input. This sequence of requests consists in HTTP requests of the following types: GET and POST.

Also, the fuzzer takes care of injecting generated inputs to the target application via the HTTP protocol. For this, the fuzzer for each element of fuzzing vector send its original version and their mutations. If state definition mechanism is enabled, in first place, fuzzer sends HTTP requests that corresponds to pre-state process, i.e, put target application in a certain state in order to test it. Placed in a certain state, the fuzzer sends the input to the target application. Later, if it is necessary to reset state of the target application, the fuzzer sends HTTP requests in order to reestablish its state. This process is repeatedly performed until all inputs being sent to the target application.

3.4. Detection of SQL injection

This type of vulnerability is detected within DBMS (Fig. 1) because the mechanism uses its resources, more specifically its parser. In the following explanation we considered the specific case of MySQL because it was the DBMS used in implementation.

The main goal in SQLI attacks detection is identify if a certain query that receives input from users, if its structure changes taking into account a benign model structure. So, in first place, it is necessary to identify what is the expected behavior of each query from target application. The purpose of this identification is to be able to compare different executions of a certain query and checks if its behavior changes. If so, there is a vulnerability in the application.

In specific case of MySQL, query's structure can be interpreted as a stack or post-order tree. However, MySQL stores query's structure in a list. Next, it is presented the structure of query `SELECT * FROM user WHERE Age<23` (Listing 1) by SQLI detection mechanism.

```
1 SELECT_LEX_Nodes_Structure
2 WHERE_Structure
3 FUNC_ITEM      4   <
4 INT_ITEM       23
5 FIELD_ITEM     Age
6 ...
7 SELECT_FIELDS_Structure
8 SELECT_FIELD       *
9 FROM_TABLE     user
```

Listing 1: Structure obtained by query `SELECT * FROM user WHERE Age < 23`

In order to understand query's structure, its items must be observed from bottom to the top. The first element in query's structure, refers to `FROM` element, in this case, table *user*. After, it refers to the elements returned by `SELECT` clause, in this case, all collumns (symbol *). Next, there is a reference to the field `Age` and to the integer `23` and with these two items it is applied `<` operation as can be seen by item above.

In this solution, first execution of certain query that receives input, assuming the input is benign, defines a query's template. So, when occurs the first execution of a query, in runtime, mechanism will store its structure. This template will be compared with obtained structures in the following executions of same query.

For correct operation of mechanism, it is necessary to exist a training phase, where is executed all points in code that executes queries that receives inputs from users with benign content in order to generate the templates.

In the following query's executions, it is made a comparison between obtained structure and respective template generated in training phase. This comparison must be tolerant to certain aspects because otherwise would generate many false positives. For example, a query that pretends to compare a given attribute with a value assigned by the user must tolerate the different values provided. Also, it should tolerate using values different than expected but due to typecasting operations the meaning of these values are the same. Thus, the mechanism considers integer, float, real and string types in a category called primitive type. So, this detection mechanism observes each element from obtained structure and check if type (first element) and parameters (remaining elements) are exactly equals to the corresponding element in the template, except in one situation. When certain element of query's structure refers to a primitive type, is not necessary to analyze its parameters, is only necessary the corresponding element in template be of the primitive type.

If the structure obtained by a given query does not correspond to the respective template means that an input has changed the query's structure and it is exploiting a SQLI vulnerability.

The mechanism can detect two types of SQL injection attacks: structural and mimicry. Also, this mechanism is able to detect first and second order SQLI vulnerabilities because in both cases, when the vulnerability is exploited there is a change in query's structure.

For security purposes, in template, parameters of primitive types are not stored so it is not possible to extract any sensitive information (e.g passwords).

In order to demonstrate this mechanism we suppose the following query: `SELECT info FROM users WHERE password = $input`. This query is executed two times: first one in order to generate its template with benign input `'xpto'`; second one in order to attack with a malicious input `'evil' OR 1=1`. The Figure 2 shows the structures, represented as stacks of both executions, respectively, (a) the *template* and (b) malicious query's structure (*input* in bold). It is possible to observe the difference between structures because it is included

malicious SQL code in query as input, therefore, the mechanism identifies a SQLI vulnerability.

3.5. Detection of local/remote file inclusion

To detect a LFI/RFI attacks is necessary to extract information about included files in each file inclusion function call, for example, in PHP language functions *include* and *require*. This detection is made inside server-side language interpreter (Fig. 1).

When is executed for first time a certain file inclusion function, assuming that uses a benign input, mechanism defines the expected behavior of that call in order to compare with future executions in that place, being defined as template. This template is generic to cover all benign inputs, but also restricted to exclude malicious inputs. It is necessary to exist a training phase where all places in code where exist this type of functions are executed with benign inputs in order to generate respective templates.

The template consists in path and extension of file that is included. If the file is remote, path includes protocol of external address (e.g: http://).

The reason for choosing these elements as part of the template is based on two principles: when an attacker pretends to include a local or remote file, file's path has to change, through path traversal or including protocol of external address, whereas in a normal execution path remains the same; when there is a file inclusion, its extension is the same over the executions, so it is considered that changing file's extension is a dangerous operation.

In following executions, detection mechanism compares the file included in a certain call with respective template. If path or extension of included file are different from those defined in template, it means that there is a vulnerability in target application. Next, the mechanism checks if included file exists or not. If so, the mechanism notifies vulnerability's exploit, otherwise warns that vulnerability would be exploited if the file existed.

Only limitation of this mechanism occurs when an attacker tries to include a malicious file with same path and extension defined in the template.

3.6. Detection of stored cross-site scripting

To detect stored XSS attacks, whenever a query is executed, the mechanism checks if the returned content by query contains any code that can be interpreted by web browsers. This detection is made within server-side language interpreter (Fig. 1) and content analysis is made through a parsing tool, that checks if there is any code in it. The parsing tool analyze the content returned by query and checks if there is any HTML or JavaScript code tags (e.g `<script>`).

Before the content is analyzed by parsing tool, the mechanism performs a pre-check on this because most of the contents returned by queries are harmless, thus causing a performance overhead. This pre-check pretends to verify if there is any dangerous characters (e.g: $<$, $>$) or strings (e.g: href, javascript). If so, the content returned by query is target of parsing tool analysis, otherwise is harmless and it is not necessary to do any further analysis.

As said before, after pre-check phase, the parsing tool analyzes content returned by query and check if there is any code tag that can be interpreted by web browsers. If so, the detection mechanism notifies a stored cross-site scripting attack.

3.7. Detection of reflected cross-site scripting

This mechanism detects if sending a script as input if this is reflected as response by web application. This detection is made inside fuzzer (Fig. 1).

Before the fuzzer sends inputs to the application, these inputs needs to be analyzed taking into account existence of code (e.g: HTML, JavaScript). If so, mechanism sends a HTTP request to the target application with input. Next, it is analyzed application's response.

This analysis checks if the response has a significant portion of input through *Smith-Waterman* algorithm [18]. This algorithm, used in computational biology area, does a local alignment between two characters sequences and works through a score system. In this score system, points are added when there is a match between two characters and points are deducted otherwise. These scores may be different depending on the target characters, for example, characters `<` and `>` have a higher score than other characters because of its importance in this vulnerability. The algorithm returns the sub-sequence of characters with highest similarity, i.e, whose score is highest.

In the purpose of the detection mechanism, it executes a local alignment between input and application's response in order to obtain a sub-sequence of response with higher similarity with input. If score obtained is perfect, i.e, sub-sequence obtained corresponds entirely to the input, we can conclude that target application is vulnerable. If score obtained is not perfect but is higher than a threshold, mechanism examines the sub-sequence obtained from algorithm. This analysis wants to check if there any code in sub-sequence. If so, it is notified a vulnerability, otherwise is considered that application changed significantly the input.

As a limitation, mechanism may fail to identify code in certain input and therefore not identify a vulnerability or there is a significant change in response in comparison to input but still have triggered an attack.

To demonstrate this mechanism consider the fol-

FUNC_ITEM	=
STRING_ITEM	
FIELD_ITEM	password
FIELD_ITEM	info
...	(...)
SELECT_FIELD	info
FROM_TABLE	users

(a) *Template*

COND_ITEM	**OR**
FUNC_ITEM	**=**
INT_ITEM	**1**
INT_ITEM	**1**
FUNC_ITEM	=
STRING_ITEM	**evil**
FIELD_ITEM	password
FIELD_ITEM	info
...	(...)
SELECT_FIELD	info
FROM_TABLE	users

(b) Malicious query's structure

Figure 2: Two executions of query `SELECT info FROM users WHERE password = $input` represented by a stack data structure

lowing code sample (Listing 2), which receives input from user and after reflects it:

```
1 $name = $_GET['name'];
2 echo "Welcome ".$name;
```

Listing 2: Code sample with a reflected XSS vulnerability

In a normal execution will be inserted as input the string `Miguel`, which is not identified any code and therefore input is not sent. Suppose that is inserted as input the malicious content `<script>alert("XSS")</script>`, this is identified as code because there are script tags and therefore input is sent. When response is received, this is analyzed and is identified that existing code in input is contained entirely in response, notifying the existence of a reflected XSS vulnerability.

3.8. Detection of other vulnerabilities

As said before and referred in Figure 1, it is possible extend the framework in order to detect other vulnerabilities existent in web applications, for example, OS command injection vulnerability (OSCI), server-side language code injection, directory/path traversal and source code disclosure.

Framework aims to analyze the sensitive sinks related with these vulnerabilities and check if there is any vulnerability exploit. In order to do that, it is necessary to modify the back-end components related with each vulnerability and gather information about requests made by application from users. This information is focused in structure of requests in order to check if there is any difference in structure depending user's input. In first phase, it is necessary to gather a benign model of each call to the functions related with these vulnerabilities. Next, when is executed a certain call, obtained structure is compared with respective benign model. If there is a difference, mechanism notifies the existence of a vulnerability because request's structure is different that structure obtained from benign inputs, i.e, intended structure by whom designed target application.

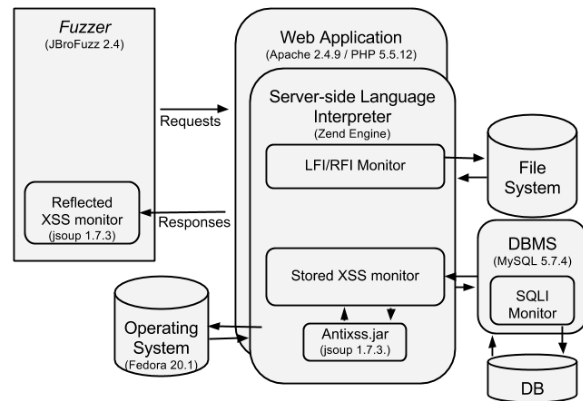

Figure 3: Framework's architecture taking into account monitoring mechanisms implemented

4. Implementation

This section presents framework's architecture taking into account implemented components (Fig. 3). Also, it explains how framework's components were implemented, namely, the fuzzer and detection mechanisms of the following vulnerabilities: SQL injection, local/remote file inclusion and reflected/-stored XSS.

4.1. Fuzzer

In order to implement the fuzzer it was used a tool developed by *OWASP* called *JBroFuzz* (version 2.4) [1]. This tool consists in a fuzzer and also a library that contains several classes that allow implement a fuzzer containing various features: fuzz vectors or fuzzing with multiple parameters. Since JBroFuzz fuzzer does not contain mechanisms for monitoring attacks it was necessary to use this library in order to create a fuzzer with a detection mechanism of

[1] https://www.owasp.org/index.php/JBroFuzz

reflected XSS attacks. It was also necessary to implement other features: input mutation mechanism; sending HTTP requests with generated inputs. Regarding to the fuzz vectors, the fuzzer contains attack signatures for each vulnerability pretended to be detected from different sources and complexities.

A web application goes through several states and only some of them are vulnerable. So, when is used fuzzing to detect vulnerabilities in a target application, it is important to do it in all different states and fuzzing in each state. The current implementation of fuzzer contains a mechanism that allows to define manually a set of HTTP requests to put the target application in a particular state or reset application's state after sending a input. It is possible to define request's type, url, parameters and cookies.

4.2. Detection of SQL injection

To implement this mechanism, it was necessary modify MySQL (version 5.7.4.), more specifically, the parsing function (*mysql_parse* - file *sql_parser.cc*) adding 14 lines of code. Also, it was created a header file with 1098 lines of code in order to implement the detection mechanism. Through these changes it is possible to detect several SQLI attacks.

Also, it is necessary to add to each query an identifier in the form of MySQL comment. This happens because with these identifiers is possible to know the place in application where is executed certain query. So, the programmer is able to understand through code analysis the reason of certain vulnerability. If there is no such identifier in query, MySQL continues to operate normally just is not target of the mechanism.

The current implementation only allows analysis of queries that contains the following commands: `SELECT`, `INSERT INTO`, `UPDATE`, `FROM`, `WHERE`, `HAVING`, `ORDER BY`, `GROUP BY`.

4.3. Detection of local/remote file inclusion

In order to implement this mechanism, it was necessary to modify PHP interpreter (version 5.5.12), namely, Zend Engine. These changes focused on interpreter's functions that implements file inclusion functions in PHP language (e.g: *include*, *include_once*, *require* and *require_once*). With this is possible to extract information related with files target of inclusion and check if the target application is vulnerable.

It was required to change Zend Engine function called *compile_file* (*zend_language_scanner.c*). This function receives as argument the file that will be compiled by PHP, in this case, the file target of inclusion. It is necessary to add an identifier in each inclusion function call in order to know which place in code where is made determined inclusion. If there is no such identifier, PHP continues to operate normally just is not target of the detection mechanism.

In order to extract information about identifiers of the inclusion functions, it was modified the file *zend_vm_execute.h* with a total of 116 lines of code. Changing this function, the detection mechanism is able to make a distinction between a vulnerability warning and when an attack truly exploit a vulnerability.

Regarding to the templates, they are permanently stored through files containing relevant information.

4.4. Detection of stored cross-site scripting

To implement this mechanism, it was modified the Zend Engine (version 5.5.12). These modifications focused on the interpreter's function that implements PHP function *mysql_query*, more specifically, function *php_mysql_do_query_general* in Zend Engine.

To analyze the content returned by queries it was necessary to check if there any code in them, for example, HTML or JavaScript. So, it was used a parsing tool programmed in Java (*antixss.jar*) that contains a parsing library, namely, *jsoup* (version 1.7.3) [2]. This library aims to parse HTML pages or other similar languages (e.g JavaScript, CSS) taking into account the existence of tags. In order to identify code, mechanism provides the content to jsoup analyzes. This parsing tool inserts the content in a dummy HTML page, represented by tags `<html><head><body>` in which will make parsing. With the structure obtained by parsing process, it will be checked if there is at least one code tag in addition to the existing tags in dummy page. If so, it means that content has changed the structure and contains code.

4.5. Detection of reflected cross-site scripting

To implement this mechanism, it was used again jsoup library (version 1.7.3) but in this case is included in fuzzer (Fig. 3).

Regarding to the *Smith-Waterman* algorithm, it was implemented in Java language. It was necessary to configure the mechanism with certain values for agreement/disagreement between characters, gap and define the percentage of similarity between the obtained score and the perfect score to be considered certain input as an attack.

When there is an agreement between two characters (case sensitive) it adds to score 5 points. When there is a disagreement between characters this can be of three types:

- When disagreement concerns to the inclusion of a blank space for sub-sequences being aligned are deducted 2 units from score. This

[2] http://www.jsoup.org

penalty is linear depending on the number of blank spaces added;

- When disagreement refers to two characters that are different from each other but are not characters < or > 5 units are deducted from score. This penalty is linear depending on the number of characters in disagreement;

- When disagreement refers to two characters that are different, in which one of them are characters < or > are deducted 25 times the units deducted in the previous point. This happens because of the importance of these characters in a reflected XSS vulnerability. This penalty is linear depending on the number of characters in disagreement.

At last, it was defined that certain input exploits a reflected XSS vulnerability when score obtained with Smith-Waterman algorithm is greater than 95% of the perfect score, i.e, score obtained if there is an agreement on all characters between the two sequences compared.

5. Results

The framework was evaluated taking into account vulnerable code samples (Subsection 5.1) and open source applications (Subsection 5.2). In both evaluations, it was necessary a training phase in order to exercise all entry points existent in the target application with benign inputs to generate all templates.

In case of vulnerable code samples, it was made a comparison between the number of detected vulnerabilities and existing vulnerabilities. In case of open source applications, it was proceeded to an accounting of vulnerabilities detected by framework. Also, the SQLIA detection mechanism was evaluated taking into account code injection attacks definition made by Ray and Ligatti [19] and compared with other detection mechanisms that also compared their results with this definition.

5.1. Evaluation with vulnerable code samples

In this subsection will be presented the results obtained by framework through code samples with vulnerabilities well identified and documented. One part of these samples was developed at same time than framework in order to test it. It contains a total of 11 files with 14 vulnerabilities identified through manual testing.

Also, will be presented the results obtained by framework through code samples from *NIST Software Assurance Reference Dataset Project* [3]. They jusify existence of a vulnerability through an input that exploits the code sample and also with a vulnerability description using CWE specification [4]. It was observed that a subset of these code samples contained very reduced or incorrect information, that indicates existence of vulnerabilities, when in reality, they did not contain any vulnerability. So, this subset of code samples was excluded from evaluation. In addition, there were valid samples that contained more vulnerabilities than indicated, in which case they were considered as existing vulnerabilities. Altogether, it contains 7 files with 11 vulnerabilities identified.

Vulnerabilities	Framework (11 files)		Samate (7 files)	
	Detected	Existent	Detected	Existent
SQLI	4	4	2	2
LFI	3	3	2	2
RFI	2	2	2	2
Reflected XSS	4	4	5	5
Stored XSS	1	1	0	0
Total	14	14	11	11

Table 1: Results obtained by framework through vulnerable code samples

Table 1 shows obtained results by framework through these code samples, splitting them by source and by each type of detected vulnerability. It was observed that all existing vulnerabilities in these vulnerable code samples were successfully detected.

5.2. Evaluation with open source applications

In this subsection will be presented the results obtained by framework through open source applications taking into account the number of vulnerabilities detected. The applications chosen to evaluate the framework contains vulnerabilities created intentionally.

To identify vulnerabilities it was necessary select all entry points to the fuzzer in order to send inputs to the target web application. Also, it was necessary a training phase in order to generate the required templates.

Table 2 presents all detected vulnerabilities by framework. It was detected a significant number of vulnerabilities in all applications tested as can be seen in Table 3.

During this evaluation, it was observed in the specific case of application *XSSeducation* the existence of false positive detections. This situation happened because some injected inputs, containing malicious scripts, affected other scripts from web application. In this situation, input was reflected but was not executed by web browser.

Also, it was observed in evaluation that certain open source applications (e.g DVWA, Peruggia and Wackopicko) contains different states of execution, so it was necessary to use the state definition mechanism in order to exercise application in all different

[3] http://samate.nist.gov/SARD/

[4] http://cwe.mitre.org/

Web application	Detected vulnerabilities				
	SQLI	RFI	LFI	Reflected XSS	Stored XSS
DVWA 1.0.8	5	1	2	7	2
Mutilidae 1.3	5	1	1	6	5
OWASP Bricks Normada	2	0	0	2	0
OWASP Bricks Tuivai	9	0	0	8	0
Peruggia 1.2	8	1	2	2	2
Wackopicko	3	1	1	2	2
XSSeducation	0	0	0	4 (2 FP)	0
Total	32	4	6	31	11

Table 2: Results obtained by framework through open source applications

Web application	Files	Lines of code	Vulnerable files	Vulnerabilities found
DVWA 1.0.8	316	32880	11	17
Mutilidae 1.3	15	838	7	18
OWASP Bricks Normada	10	654	2	4
OWASP Bricks Tuivai	23	1831	10	17
Peruggia 1.2	10	1015	6	15
Wackopicko	49	3014	6	9
XSSeducation	9	114	4 (2 FP)	4 (2 FP)
Total	432	40346	46	84

Table 3: Overview of results obtained by framework through open source applications

states with inputs and consequently, detects vulnerabilities on these. In all cases, these states focused on users authentication, so there are two states in these applications: *not authenticated* and *authenticated*. It was detected vulnerabilities by framework in state *authenticated* that were not detected in state *not authenticated*.

5.3. Evaluation of SQLIA detection mechanism

Despite SQL injection attacks are popular for nearly a decade, recently, Ray and Ligatti argued that definition of code injection attacks such as SQLI is problematic. So, they introduced the notion of code-injection attacks on output (CIAO) [19] where they used 11 test cases that supports their definition in which defines what is a code injection and what is not. In order to evaluate the SQLI detection mechanism, it was taking into account these test cases.

Table 4 presents results obtained by framework with Ray and Ligatti definition. Also, these results were compared with other tools presented as related work in [19] and with another tool - DIGLOSSIA [20]. Test cases in which it was identified as code are represented with Yes and otherwise are represented with No.

It was observed that developed framework correctly classifies 10 of the 11 cases defined by Ray and Ligatti being on par with DIGLOSSIA while remaining tools had a lower performance in their detections. Despite this, the solution developed in this work can detect first and second order SQLI (structural or mimicry), encoding/space evasion based SQLI whereas DIGLOSSIA only can detect first order SQLI (structural or mimicry).

6. Conclusions

This work presents a framework for fuzzing web applications and detecting vulnerabilities through monitoring mechanisms. The solution developed is different than presented in *state of the art* because, in this work, the monitoring mechanisms are embedded in back-end components of a web application.

This type of monitoring approach has many advantages: as last element of execution flow, does not need make assumptions about previous entities; is not dependent of any sanitization, validation or encoding processes; uses resources from web application's components in order to assist the detection.

The current framework version can detect several vulnerabilities: SQL injection, local/remote file inclusion and also reflected/stored cross-site scripting. Also, this framework enables to put the target application in a certain state in order to test in all states.

The developed framework was tested using intentionally vulnerable code samples and open source applications in order to count how many vulnerabilities were detected. In first group, the framework detected all vulnerabilities. In second group, it was detected a significant amount of vulnerabilities. Also, the SQL injection detection mechanism was tested taking into account Ray and Ligatti code injection definition [19]. We could conclude that developed mechanism performs better than other *state of the art* mechanisms.

Regarding to the future work, we consider the development of remaining detection mechanisms of vulnerabilities: OS command injection; server-side language code injection; directory/path traversal; source code disclosure. Also, the fuzzer can be improved with state-awareness features to be possible, automatically, analyze the target application and infer its execution states.

References

[1] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[2] Adam Doupé, Marco Cova, and Giovanni Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2010.

[3] João Antunes, Nuno Neves, Miguel Pupo Correia, Paulo Verissimo, and Rui Neves. Vulnerability discovery with attack injection. *IEEE Transactions on Software Engineering*, 36(3):357–370, 2010.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Ray and Ligatti's definition [19]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No
DIGLOSSIA [20]	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	No	No
CANDID [12]	Yes	Yes	Yes	No	No	No	Yes	No	No	No	Yes
SQLCHECK [21]	Yes	No	No	Yes	No	No	No	No	No	No	No
Xu et al. [22]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
AMNESIA [11], Nyugen-tuong et al. [23]	Yes	Yes	Yes	No	No	No	No	No	No	No	Yes
Framework	*Yes*	*Yes*	*Yes*	*No*	*Yes*	*Yes*	*No*	*Yes*	*Yes*	*No*	*No*

Table 4: Results obtained by framework and other six similar tools through CIAO definition by Ray and Ligatti [19]

[4] Gregg Keizer. Microsoft runs fuzzing botnet, finds 1,800 office bugs. Computer World, March 2010.

[5] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[6] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[7] LaShanda Dukes, Xiaohong Yuan, and Francis Akowuah. A case study on web application security testing with tools and manual testing. In *Southeastcon, 2013 Proceedings of IEEE*, pages 1–6, 2013.

[8] OWASP. OWASP Top 10 - 2013 - the ten most critical web application security risks. Technical report, OWASP Foundation, 2013.

[9] Imperva. Hacker intelligence initiative, monthly trend report #8. April 2012.

[10] William Halfond and Alessandro Orso. *Malware Detection - Detection and Prevention of SQL Attacks*. Springer, 2006.

[11] William Halfond and Alessandro Orso. Preventing SQL injection attacks using AMNESIA. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[12] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 12–24, 2007.

[13] Johannes Dahse and Jörg Schwenk. Rips-a static source code analyser for vulnerabilities in PHP scripts, 2010. http://www.nds.rub.de/media/nds/attachments/files/2010/09/rips-paper.pdf.

[14] Arpit Bajpai. Securing Apache, Part 9: Attacks that Target PHP-based Instances, 2011. http://www.linuxforu.com/2011/05/securing-apache-part-9-attacks-that-target-php-instances/.

[15] Suman Saha. Consideration points detecting cross-site scripting. *CoRR*, 4, 2009.

[16] Lwin Khin Shar and Hee Beng Kuan Tan. Defending against cross-site scripting attacks. *IEEE Computer*, 45(3):55–62, 2012.

[17] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, volume 2, 2004.

[18] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[19] Donald Ray and Jay Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[20] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1181–1192, 2013.

[21] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 372–382, 2006.

[22] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, volume 15, 2006.

[23] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. *Automatically hardening web applications using precise tainting*. 2005.