Pontifícia Universidade Católica
do Rio de Janeiro

Oslien Mesa Rodríguez

Proactive Mitigation of Vulnerabilities in Plugin-based Web Systems

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Carlos José Pereira de Lucena

Rio de Janeiro
June 2019

Pontifícia Universidade Católica
do Rio de Janeiro

Oslien Mesa Rodríguez

Proactive Mitigation of Vulnerabilities in Plugin-based Web Systems

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Carlos José Pereira de Lucena
Advisor
Departamento de Informática – PUC-Rio

Prof. Marx Leles Viana
Pesquisador Independente – PUC–Rio

Prof. Marcos Kalinowski
Departamento de Informática – PUC–Rio

Rio de Janeiro, June 14th, 2019

Oslien Mesa Rodríguez

Graduou-se em Engenharia em Informática pela Universidade de Ciência da Informática da Havana (Havana, Cuba). Fez mestrado no Departamento de Informática da PUC-Rio, especializando-se na área de Engenharia de Software, na área de Framework e Linhas de Produtos. Desenvolveu um grande número de rotinas para analisar Vulnerabilidades de Segurança em Sistemas Configuráveis.

Bibliographic data

Rodríguez, Oslien Mesa

Proactive Mitigation of Vulnerabilities in Plugin-based Web Systems / Oslien Mesa Rodríguez; advisor: Carlos José Pereira de Lucena. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2019.

v., 73 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. Vulnerabilidade;. 3. Sistemas Configuráveis;. 4. WordPress;. 5. Plugin;. 6. Sistemas de Segurança;. 7. Teste de Software.. I. Lucena, Carlos José Pereira de. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my mother, for your support and encouragement. To my maternal grandmother, for being my first guide in life, educate me and teach me to be who I am.

Acknowledgments

I would like to thank the Department of Information Technology of the PUC-Rio firstly for allowing me to take the Post-Graduation courses and complete the master's degree, an improvement that I did not achieve in my country despite several attempts. To CNPq, for its interest in promoting research in Brazil and allowing me to make use of its study payment system. To Professor Lucena for welcoming me to his team, being my counselor and advisor. To Marx for influencing as a counselor with his advice, desire and motivation for research and hard work. To the union of professors Lucena, Marx and Elder, who supported me and allowed me to attend the conferences in which the articles were published. To Professor Marcos Kalinowski, for be member of the defense court and his constructive criticisms, to the research and thesis document. Chico for his criticisms in the presentations, his help with the recommendation of several articles, his help with the format of the thesis document and his collaboration in the use of Latex. To Vera, who with her kindness and patience, who allowed me to enjoy her company, and teaching moments while waiting for a class shift or a schedule marked with Professor Lucena. I thank the secretarial staff of the Information Technology Department for their patience and guidance.

I can't stop thanking my family for the support they have offered me from a distance. To my brother Osniel, for taking care of the needs of our Mother. To my cousin and sister Geidy, for helping my brother in all circumstances. To my little nephew Jose Julio (JJ) for making my Mother's days happy. To my Mother, for suffering and enduring so much time in the distance, the absence of one of her children and for insisting and influencing her behavior and ideas. The many Cubans that I met here in Brazil, that everyone has somehow offered me useful advice at some point in this journey in my life. I cannot fail to mention Javier Guillot, to whom I owe the thanks of having spent a first semester sharing his time and knowledge with him. To Claudia and Adrian, who always welcomed me and accompanied me in happy and sad moments during this project. To Antonio Iyda, he is not Cuban, but it is as if he were, a Brazilian friend who held out his hand and relieved me on several occasions with my problems at the university. To my girlfriend Darialys, for her quick adaptation to a new family and a new life by my side. For your patience and insistence to conclude this project.

Abstract

Rodríguez, Oslien Mesa; Lucena, Carlos José Pereira de (Advisor). **Proactive Mitigation of Vulnerabilities in Plugin-based Web Systems**. Rio de Janeiro, 2019. 73p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A common software product line strategy involves plug-in-based web systems that support the simple and rapid incorporation of custom behaviors and are widely adopted for building web-based applications. The popularity of ecosystems that support plug-in-based development (such as WordPress) is largely due to the number of customization options available as community-contributed plugins. However, plug-in related vulnerabilities tend to be recurring, exploitable and difficult to detect and can lead to serious consequences for the custom product. Therefore, these vulnerabilities must be understood to enable the prevention of relevant security threats. In this paper, we conduct an exploratory study to characterize plug-in vulnerabilities in web-based systems by examining the WordPress vulnerability bulletins cataloged by the National Vulnerability Database and the associated patches maintained by the WordPress plugin repository. We identify the main types of vulnerabilities, their impact, and the size of the patch to address the vulnerability. We have also identified the most common security-related topics discussed among WordPress developers. We note that while vulnerabilities can have serious consequences and remain unnoticed for a long time, they can often be mitigated with minor changes to source code. Characterization helps provide an understanding of how such vulnerabilities manifest themselves in practice and contributes to new generations of vulnerability testing tools that can anticipate their potential occurrence. This research proposes a support tool to mitigate the occurrence of vulnerabilities in web plugin based systems, facilitating the discovery and anticipation of the possible occurrence of vulnerabilities.

Keywords

Vulnerability; Configurable Ssytems; WordPress; Plugin; Security Systems; Software Testing.

Resumo

Rodríguez, Oslien Mesa; Lucena, Carlos José Pereira de. **Mitigação Proativa de Vulnerabilidades em Sistemas da Web Baseados em Plugin**. Rio de Janeiro, 2019. 73p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Uma estratégia comum de linha de produtos de software envolve sistemas da Web baseados em plug-ins que suportam a incorporação simples e rápida de comportamentos personalizados, sendo amplamente adotados para criar aplicativos baseados na web. A popularidade dos ecossistemas que suportam o desenvolvimento baseado em plug-ins (como o WordPress) é, em grande parte, devido ao número de opções de personalização disponíveis como plug-ins contribuídos pela comunidade. Entretanto, as vulnerabilidades relacionadas a plug-ins tendem a ser recorrentes, exploráveis e difíceis de serem detectadas e podem levar a graves conseqüências para o produto personalizado. Portanto, é necessário entender essas vulnerabilidades para permitir a prevenção de ameaças de segurança relevantes. Neste trabalho, realizamos um estudo exploratório para caracterizar vulnerabilidades causadas por plug-ins em sistemas baseados na web, examinando os boletins de vulnerabilidade do WordPress catalogados pelo National Vulnerability Database e os patches associados, mantidos pelo repositório de plugins do WordPress. Identificamos os principais tipos de vulnerabilidades, o seu impacto e o tamanho do patch para corrigir a vulnerabilidade. Identificamos, também, os tópicos mais comuns relacionados à segurança discutidos entre os desenvolvedores do WordPress. Observamos que, embora as vulnerabilidades possam ter consequências graves e permanecerem despercebidas por muito tempo, elas geralmente podem ser atenuadas com pequenas alterações no código-fonte. A caracterização ajuda a fornecer uma compreensão de como tais vulnerabilidades se manifestam na prática e contribui com as novas gerações de ferramentas de teste de vulnerabilidades capazes de antecipar sua possível ocorrência. Esta pesquisa propõe uma ferramenta de suporte para mitigar a ocorrência de vulnerabilidades em sistemas baseados em plugins web, facilitando a descoberta e antecipação da possível ocorrência de vulnerabilidades.

Palavras-chave

Vulnerabilidade; Sistemas Configuráveis; WordPress; Plugin; Sistemas de Segurança; Teste de Software.

Table of contents

1 Introduction **13**
1.1 Motivation 13
1.2 Problem 14
1.3 Research Questions and Goal 15
1.4 Contributions 16
1.5 Proposal Organization 17

2 Security Vulnerabilities **18**
2.1 Type of Vulnerabilities 18
2.2 Classification according to the CWE Repository 19

3 Security Web Systems **22**
3.1 Definitions of Security Web Systems 22
3.2 Tool for detection of vulnerabilities 22

4 Vulnerabilities in Plugin-based Web System **24**
4.1 Exploratory Research 24
4.2 Answers to Research 25
4.2.1 RQT.1 – What are the main vulnerabilities caused by WordPress plugins? 26
4.2.2 RQT.2 – How critical are the vulnerabilities caused by WordPress plugin? 27
4.2.3 RQT.3 – What is the patch size to fix WordPress plugin vulnerabilities? 29
4.2.4 RQT.4 – How long does a vulnerability survive in the WordPress plugins code? 30
4.2.5 RQT.5 – What are the most common vulnerability related topics discussed among developers? 31
4.3 Mitigation and Detection Methods 32
4.4 Threats to Validity 34
4.4.1 External Validity 34
4.4.2 Internal Validity 34
4.5 Chapter's Conclusions 35

5 A Tool for Supporting the Mitigation of the Occurrence of Vulnerabilities **36**
5.1 Motivation for building the Tool 36
5.2 Functional Requirement 39
5.3 Technologies used for Build the Tool 41
5.4 Tool Design 42
5.5 How to use 45
5.6 Class Diagram 46
5.6.1 Class Main_XSS 47
5.6.2 Class Process 47
5.6.3 Class P_Reader 48
5.6.4 Class Fill_Line 49

5.6.5 Class Line_Word 49
5.6.6 Class Word_Property 50
5.6.7 Class Possible_Vul 50
5.7 Highlight in the Code 51

6 Tool Evaluation **53**
6.1 Partial Results 53
6.2 Results Images 55

7 Conclusions **58**
7.1 Advantages and Weakness of Patterns 58
7.2 Final Recommendations 59
7.3 Future Work 60

Bibliography **61**

A Articles Published at Conferences **67**

B Behavior Patterns **68**

C Solutions to Behavior Patterns **69**

D File CVE-2011-4562-log.php **70**

E File CVE-2011-4562-log_item.php **72**

List of figures

Figure 1.1 Fragment of study made by W3Techs 14
Figure 1.2 Vulnerability CVE-2015-7325 15

Figure 5.1 Case study identified. Plug-in directly connected to WordPress 37
Figure 5.2 Case study rejected. Plug-in connected through another Plug-in with WordPress 37
Figure 5.3 Example of vulnerability produced by use of a plugin 38
Figure 5.4 Architecture proposal for the tool 43
Figure 5.5 Architecture proposal for the tool made 44
Figure 5.6 File Analysis Flow Diagram 45
Figure 5.7 Tool Plugin's Class Diagram 46
Figure 5.8 Class Main_XSS 47
Figure 5.9 Class Process 47
Figure 5.10 Class P_Reader 48
Figure 5.11 Class Fill_Line 49
Figure 5.12 Class Line_Word 49
Figure 5.13 Class Word_Property 50
Figure 5.14 Class Possible_Vul 50

Figure 6.1 Folder Path Request 55
Figure 6.2 Show the folder path inserted in green 55
Figure 6.3 Test image performed showing a possible security vulnerability found in a single file 56
Figure 6.4 Test image performed showing the security vulnerabilities found in with two files 56
Figure 6.5 Test image performed showing the security vulnerabilities found in with three files. 56
Figure 6.6 Fragment of test image made with more than forty files, it shows a sample of the possible security vulnerabilities detected in several files. 57

List of tables

Table 2.1 Reflect the scope of the vulnerability group 21

Table 4.1 Lists the most common vulnerability types 26
Table 4.2 The number of vulnerabilities that do partially affect the web systems 28
Table 4.3 Reflect the number of lines of code and files that were modified to correct vulnerabilities 29
Table 4.4 Reports the survival time of security vulnerability in the plugins code extracted from the 119 patch analyzed 30
Table 4.5 Show the distribution of question (Q) and answers (A) per topic 31

Table 5.1 Characters used for identify words 51

List of Abreviations

(A) --- Int --- Integrity
(B) --- Conf --- Confidentiality
(C) --- Ava --- Availability
(D) --- AsC --- Access Control
(E) --- NoRd --- Non Repudiation
(F) --- Other --- Other
(G) --- AsC / Conf
(H) --- AsC / Conf / Other
(I) ----- Int / Conf / Ava
(J) ---- Conf / Int / Ava / AsC
(K) --- Conf / Int / Ava / AsC / NoRd
Adv --- Advantages
Weak --- Weakness

1
Introduction

The rise of computerization and the wide use of web applications for the personal management of users has led to the need to increase the protection of user data. Web applications are used to manage confidential information; such as bank accounts to mention an example, of each of our users which implies that the need for data protection is even greater. On the other hand, the wide diversity of web applications used to perform similar procedures and that are or may be originated using a common core, generates the applications belonging to the same Product Line or Configurable Management Systems (CMS)(44).

To optimize the functioning of web applications created with CMS, plugins are commonly used (38). The plugins optimize or add functionalities to the CMS cores. Just as there is a wide diversity of web applications, wide diversity of CMS, there is a wide variety of plugins for each CMS. Sometimes plugins can also be grouped as a family.

It happens that the wide diversity of products created with a core in common causes that all result with the same deficiencies or in our case vulnerabilities. In addition to that there are vulnerabilities that are not manifested until a plugin is added as evidenced by the vulnerabilities registered in the National Vulnerability Database (NVD) Repository([1]). Of course, the Repository has registered vulnerabilities that are not related to the plugins, but they are not the vulnerabilities with which we will carry out our work.

1.1
Motivation

We select the WordPress CMS for our study because it is one of the most used by the community of developers and the curious that without being computer programmers dare to develop web applications. In addition, to facilitate, complete and increase its use it has more than 55,000 plugins([2]).

In our research we found a study conducted Web Technology Surveys (W3Techs)([3]), where a part of the CMS more used is tracked. The W3Techs updates its study daily and Figure 1.1 shows a fragment of the upper part of

[1] https://nvd.nist.gov/
[2] https://es.wordpress.org/plugins/
[3] https://w3techs.com/technologies/overview/content_management/all

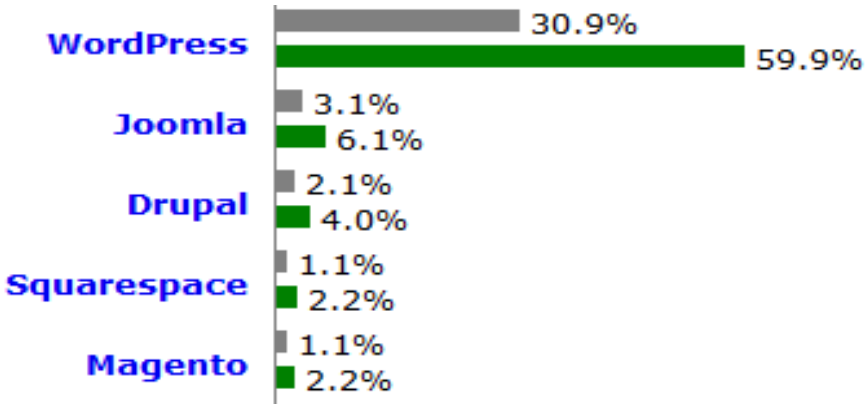

Figure 1.1: Fragment of study made by W3Techs

diagram of May 25, 2018. W3Techs shows that the CMS are used in 48.4% of web applications, followed for it, and only WordPress is used in 30.9%. If limited the study area of site only to CMS, we can be appreciate as WordPress is used in 59.9% of the times.

1.2 Problem

The community of programmers develops useful and practical applications, but its eagerness for the creation and fast obtaining of the results is neglected several aspects. Studies (26)(27)(28) have shown that one of the most neglected aspects are security vulnerabilities; aspects as common as, the sanitization of a variable or the validation of a *GET* method, to prevent an insecure data entry. Another study shows that on many occasions, programmers do not know how to deal with existing security vulnerabilities regardless of the experience of programmers (19).

Figure 1.2([4]) shows an example about how a good code with a slight oversight can cause a serious vulnerability. Figure 2 was derived from SQL Injection vulnerability CVE-2015-7235([5]), it registered in the NVD Repository last September 17, 2015. The vulnerability was evaluated with metric CVSS v2.0 and score was 7.5 (HIGH)([6]), which indicated that the vulnerability is serious. Besides being a serious vulnerability the Figure 1.2 shows how with a small change in the code could be avoided the existence of it. To fix this vulnerability, the group of developers took 1 year, 3 months and 4 days; the solution was announced last December 21, 2016 (36)(37). We understand that

[4]https://plugins.trac.wordpress.org/changeset/1104099/cp-reservation-calendar
[5]https://plugins.trac.wordpress.org/changeset/1104099/cpreservation-calendar/trunk/dex_reservations.php
[6]https://nvd.nist.gov/vuln/detail/CVE-2015-7235

is a very long time to fix a HIGH vulnerability. The vulnerability happened and was inserted by using the CP Reservation Calendar plugin, in the *dex reservations.php* file.

```
if (!defined('CP_CALENDAR_ID'))
    define ('CP_CALENDAR_ID',$_POST["dex_item"]);
    define ('CP_CALENDAR_ID',intval($_POST["dex_item"]));

session_start();

$_SESSION['rand_code'] = '';

$selectedCalendar = $_POST["dex_item"];
$selectedCalendar = intval($_POST["dex_item"]);

$_POST["dateAndTime_s"] =
$_POST["selYear_start".$selectedCalendar]."-".$_POST["selMonth_start".$selectedCalendar]."-".$_POST["selDay_start".$sel
ectedCalendar];

global $wpdb;
if (!defined('CP_CALENDAR_ID'))
    define ('CP_CALENDAR_ID',$_POST["dex_item"]);
    define ('CP_CALENDAR_ID',intval($_POST["dex_item"]));

$data = array(
```

Figure 1.2: Vulnerability CVE-2015-7325

In our research, as a first step, we will focus on understanding the characteristic and behavior about code registered at NVD Repository; and where the programmers neglect small details and how serious is the compromise the security of web applications. As an example we have, the structure of a *GET* method inserted directly as a parameter in another method; when the correct way is to save the *GET* structure in a *variable* and insert this as a parameter. The programmers concentrate their efforts on the correct function of the web application, but an application with a correct operation is not always a safe web application. A Secure Web Application is a web application that hinders access to attackers. As a second step, we will try to reduce the occurrence of security vulnerabilities caused by the addition of plug-ins to configurable applications.

1.3 Research Questions and Goal

Based on the problems and limitations presented previously, we can define the following research questions (33):

(i) RQT.1: What are the main types of security vulnerabilities caused by WordPress plugins? (Sub-section 4.2.1)

(ii) RQT.2: How critical are the security vulnerabilities caused by WordPress plugins? (Sub-section 4.2.2)

(iii) RQT.3: What is the patch size to fix WordPress plugin vulnerabilities? (Sub-section 4.2.3)

(iv) RQT.4: How long does a vulnerability survive in the WordPress plugins code? (Sub-section 4.2.4)

(v) RQT.5: What are the most common security-related topics discussed among WordPress developers? (Sub-section 4.2.5)

The approach proposed in this research deals with such issues, answers to research questions and shows the relevance or our main goal:

(i) Identify Patterns of Behavior in the vulnerabilities registered in the NVD Repository in order to use them for the detection of new vulnerabilities.

(ii) Create a Tool‘s Support for mitigating the occurrence of security vulnerabilities produced by the addition of a plugin to WordPress.

1.4 Contributions

With our research we expect demonstrate the need and posibility of mitigate the security vulnerabilities in web aplications. Initially, we will attack the possible security vulnerabilities in the application-plugin pairs developed with CMS WordPress, and make the following contributions:

(i) Identify the most common vulnerabilities, due to their high frequency of occurrence;

(ii) Demonstrate the need and possibility of decreasing the occurrence of vulnerabilities gradually;

(iii) Start the creation of a plugin for Eclipse IDE, that uses the identified patterns for the identification of new possible security vulnerabilities;

(iv) Use the plugin to show the solution to the possible security vulnerability detected with the identified patterns in our study.

1.5 Proposal Organization

This dissertation proposal is structure as follows:

(i) Chapter 2 - Security Vulnerabilities. We reduce and define the small group of vulnerabilities with we will work.

(ii) Chapter 3 - Security Web Systems. We define security system and show many applications that perform vulnerability tests to web applications

(iii) Chapter 4 - Vulnerabilities in Plugin-based Web System. We describe our research and the questions that guided us.

(iv) Chapter 5 - Tool Support for Mitigating the Occurrence of Vulnerabilities. We describe some detail about our prototype.

(v) Chapter 6 - Tool Evaluation. In this chapter, we analyzed the result of the research and we show images of it.

(vi) Chapter 7 - Conclusions. In this chapter, we analyzed the advantages and weakness of use of our Patterns, and we reveal our conclusions.

2
Security Vulnerabilities

In this chapter, we define the different vulnerabilities with which we will work in this proposal. The group of types of vulnerabilities is very broad but after an exploratory study reflected in Chapter 4, we reduced the group of vulnerabilities with that we will work. In addition, we summarize the classification offered by the Common Weakness Enumeration (CWE) Repository([1]) for the assessment of damages of each vulnerability.

The security vulnerabilities appear when attackers manage to violate one of the aspects identified in the definition of Security Systems.

(i) Security Systems: is "the practice of building software to be secure and function properly under intentional malicious attacks" (20), is an integrative concept that includes four key properties (13): confidentiality, authenticity, integrity, and availability. (3)

These keys properties are useful whenever we talk about system security; without making a difference between the software. The differences are established when in the different types of vulnerabilities identified in the next Section.

2.1
Type of Vulnerabilities

Before starting with our research, we will define the different classifications of existing vulnerabilities and the classes in which they can be grouped according to A Community-Developed List of Software Weakness Types. (51)

(i) Cross-site Scripting (XSS): the code does not neutralize or incorrectly neutralize the user input before it is passed as output to other users;

(ii) Direct Traversal (Dir. Trav.): the code does not neutralize special elements that may cause the requested access path to be returned to another restricted directory;

[1] https://cwe.mitre.org/

(iii) Cross-site Request Forgery (CSRF): the code is not able to verify whether well-formulated and valid requisitions were intentionally or not provided by the user who submitted the requisition;

(iv) SQL Injection: the code does not unduly neutralize or neutralize special elements passed as input and that have the ability to modify commands;

(v) Open Redirect: allows attackers to impersonate or duplicate the identity of the site and thus carry out phishing attacks;

(vi) Bypass: allows an attacker to fool unsuspecting users through pages with malicious content.

2.2 Classification according to the CWE Repository

There are other classifications for the vulnerabilities, but due to their low frequency of occurrence they will be evaluated in future studies. In this study we prefer to focus on the most frequent vulnerabilities.

The CWE (43) Repository offers to Community-Developed a List of Software Weakness Types. Repository where the origin of the vulnerabilities described and explained and possible strategies for the solutions are offered, both from the point of view of the programmer and the network administrator. Because they are another criterion of classification, several classifications of the most known can be combined in some of them.

In addition, several strategies are offered to mitigate the appearance of vulnerabilities in various phases of the software creation process. It also describes the complexity of taking care of the appearance of a vulnerability, because by avoiding the appearance of a vulnerability, we can cause the vulnerability of another vulnerability. After describing vulnerabilities, we visualize a Table 2.1 with the scope of each vulnerability.

The vulnerabilities that we found in the CWE-20([2]) group: Improper Input Validation are originated when the data entry is not validated correctly, an attacker can elaborate the data entry in a manner for which the application is not prepared and misinterprets, causing some parts of the system to receive an involuntary entry resulting in an altered control flow. For the examples that are used in this group, they used code fragments in C, Java and PHP.

The vulnerabilities that appear in the group classified as CWE-22([3]): Improper Limitation of a Path-name to a Restricted Directory [Path Traversal] are dominated by vulnerabilities classified as Dir. Trav., Those that appear

[2]http://cwe.mitre.org/data/definitions/20.html
[3]http://cwe.mitre.org/data/definitions/22.html

when the protection of the address that indicates where the files of the application are saved is neglected.

In the description of these vulnerabilities examples of their existence and correction in languages such as Perl, Java and HTML are shown.

In the group classified as CWE-79([4]): Improper Neutralization of Input During Web Page are those vulnerabilities that do not neutralize or incorrectly neutralize the data inserted by the user. Most of the vulnerabilities found in this group are XSS.

On the page that describes and explains the vulnerabilities of the CWE-79 group, it shows several examples of how to detect and correct these XSS vulnerabilities in languages such as PHP, JSP and ASP.NET.

In the group of classified vulnerabilities such as CWE-89([5]): Improper Neutralization of Special Elements used in SQL Command (SQL Injection). They are vulnerabilities that originate from the lack of validation of the data that have to be inserted by the user, and in turn the application uses them to elaborate SQL queries. For examples of how to mitigate or correct vulnerabilities, use examples in languages such as C#, PHP, SQL and Perl.

The vulnerabilities grouped in CWE-94([6]): Improper Control of Generation of Code (Code Injection). The application is able to build a segment of code from the data inserted by the user and does not neutralize or incorrectly neutralize elements that could modify the syntax or behavior of the desired code segment. In this group of vulnerabilities, they use the codes in PHP and Perl languages.

The vulnerabilities grouped in the classification CWE-200([7]): Information Exposure are produced by the intentional or unintentional disclosure of sensitive information and not accessible to a user not authorized to have access to that information.

In this other group CWE-284([8]): Improper Access Control you can find the vulnerabilities caused by the non-restriction or incorrect restriction of the users' access to resources not allowed. In these cases, the vulnerabilities are not produced by carelessness during the programming, but the carelessness is produced in the configuration of the application taking into account the users and their roles.

Other groups in which our vulnerabilities are listed is CWE-352([9]): Cross-site Request Forgery (CSRF). In this group only vulnerabilities of the CSRF

[4]http://cwe.mitre.org/data/definitions/79.html
[5]http://cwe.mitre.org/data/definitions/89.html
[6]http://cwe.mitre.org/data/definitions/94.html
[7]http://cwe.mitre.org/data/definitions/200.html
[8]http://cwe.mitre.org/data/definitions/284.html
[9]http://cwe.mitre.org/data/definitions/352.html

type are found as their name indicates. In this case the examples were only applied in the PHP and HTML languages.

The group of vulnerabilities classified as CWE-601([10]): URL Redirection to Untrusted Site (Open Network) allows the execution of phishing attacks as they allow the entry of links to external sites or redirecting it to other links. For the examples of how they can be corrected or mitigated the vulnerabilities of this group were used the PHP, Java and HTML languages.

Table 2.1: Reflect the scope of the vulnerability group

CWE/Scope	A	B	C	*D*	*E*	F	G	H	I	J	K
CWE-20		X	X						X		
CWE-22	X	X	X						X		
CWE-79							X		X	X	
CWE-89	X	X		X							
CWE-94				X	X				X		
CWE-200		X									
CWE-284						X					
CWE-352											X
CWE-601				X				X			

In the fields of the Table 2.1, that reflects the scope of the vulnerability group, the classifications of the scope are shown independently and in different combinations, keeping in mind that the information disclosed in the repository does not allow to separate the obtained combinations.

[10]http://cwe.mitre.org/data/definitions/601.html

3
Security Web Systems

In this chapter, we define security system and presents some applications that perform vulnerability tests to web applications.

3.1
Definitions of Security Web Systems

The security vulnerabilities appear when attackers manage to violate one of the aspects identified in the definition of Security Systems (20).

3.2
Tool for detection of vulnerabilities

There are a large number of web application security scanners. The scanners only perform functional tests and do not have access to the application code. The vulnerability detection tests are carried out to the applications shortly before being made available for the use of users. Once published we can add new functionalities with the use of the plugins; these new updates (using plugin) are rarely scanned again.

With the aim of our proposal to anticipate the possible occurrence of vulnerabilities, we will be able to perform the vulnerability detection process during the plugin development process. In this way we want to guarantee that the plugins are the safest possible.

An example of these applications that perform Tests are([1]):

(i) Grabber: execute scans and tell where the vulnerability exists. It can detect vulnerabilities such as: XSS, SQL Injection, and others. It is not a quick application compared to other security scanners, but it is simple and portable, so it should only be used to test small web applications. This tool was developed in Python.

(ii) Vega: this tool is written in Java and offers a GUI-based environment. It can be used to find SQL injection, XSS, file inclusion and other web application vulnerabilities. This tool can also be extended with a powerful API written in JavaScript.

[1] https://resources.infosecinstitute.com/14-popular-web-application-vulnerability-scanners/#gref

(iii) Wapiti: Perform tests by scanning web pages and injecting loads and see if a script is vulnerable. It supports GET and POST HTTP attacks and detects multiple vulnerabilities. It can detect vulnerabilities such as Cross Site Scripting (XSS), Weak configuration .htaccess, and many others. It is a command line application, so it may not be easy for beginners. But for the experts, it will work well. To use this tool, you must learn many commands that can be found in the official documentation.

(iv) WebScarab: is a Java-based security framework for analyzing web applications using the HTTP or HTTPS protocol. It has several add-on available, which can expand its functionality. The available modules can easily detect the most common vulnerabilities, such as SQL Injection, XSS and many other vulnerabilities.

(v) Skipfish: was written in C. It is highly optimized for HTTP handling and uses a minimal CPU. He says he can easily handle 2000 requests per second without adding a load to the CPU. Uses a heuristic approach while tracking and testing web pages. This tool also claims to offer high quality and less false positives.

4
Vulnerabilities in Plugin-based Web System

In this chapter, we describe how the research questions that guide our research were answered. Furthermore, the current work published in (51)(52) presents some results obtained from the analysis of the vulnerabilities registered in the repository.

4.1
Exploratory Research

We base our study on 895 vulnerability bulletins registered in the NVD repository. The information was collected in September 2017 by downloading all the NVD data as JSON Feeds. We then built a parser that went over all bulletin descriptions and collected the ones in which the keywords "WordPress" and "Plugin" occurred together in a given sentence. After reviewing the bulletin descriptions for a second time, our parser extracted the type and severity of the vulnerabilities in each bulletin. Finally, we examined the resulting data to answer RQT.1 (Sub-section 4.2.1) and RQT.2 (Sub-section 4.2.2).

Currently, the CVSS (21) captures the severity of vulnerabilities by considering how they can be accessed and whether or not extra conditions are required to exploit them. In our study we considered the following relevant measures to exam and answer RQT.2: Integrity Impact: the impact to integrity when a vulnerability is exploited successfully; Availability Impact: impact to availability of a successful exploited vulnerability, which evaluates the impact of attacks that consume computational resources (e.g., network bandwidth use, disk space, or processor cycles), and Complexity Impact: the complexity of the attack required to exploit the vulnerability once an attacker has gained access to the target application.

(i) Integrity: None: there is no impact on integrity; Partial: it is possible to make modifications to files or gain access to application information; however, either the attacker has no control over what can be modified or the attacker‘s scope is limited; Completed: there is a total compromise of application integrity;

(ii) Availability: None: no impact on web application availability; Partial: reduction in performance, or partial interruptions in available resources; Complete: stop of affected resources;

(iii) Confidentiality: None: no impact on web application confidentiality; Partial: considerable informational disclosure; Complete: total information disclosure;

(iv) Complexity: Low: does not demand expertise; Medium: requires a little expertise; and High: demands high expertise.

NVD also promotes the incorporation of links to other web sites with complementary information of interest to developers and users. Thus, in order to measure the size of the path to fix plugin-related vulnerabilities and their survivability, in the third inspection of the vulnerabilities bulletins, we extracted information related to patches posted in the WordPress Plugin Repository([1]). This repository supports the development of over 52,000 WordPress-related plugins and provides a set of development tools, a wiki, a version control system, and a bug tracker.

Using the Stack Exchange application programming interface (API) (8), we extracted questions and answers posted between August 2010 and October 2017. To collect questions related to security issues we limited our search to questions in which the keyword "plugin" appeared along with either "vulnerability", "hacked", "security", "insecure" or "safe" (17)(39). We looked for these keywords both in the title and in the body questions. Next, we manually eliminated non-related questions. We extracted information was pre-processed: we removed any code snippet enclosed in <code> ... </code>. In the context of our research, code snippets do not convey much helpful information. Additionally, we removed all HTML tags, all stop words, and applied Porter Stemming to map each word to their base form (54).

4.2 Answers to Research

As mentioned, to answer RQT.1, we analyzed any aspects reported on vulnerability bulletins. First, we probed the vulnerability types and whether their frequency has been increasing or decreasing over the years. Second, we also investigated when they are normally introduced during the development life cycle. Table 4.1 list the most common vulnerability types

[1]https://es.wordpress.org/plugins/

4.2.1 RQT.1 – What are the main vulnerabilities caused by WordPress plugins?

As mentioned, to answer RQT.1, we analyzed sine aspects reported on vulnerability bulletins. First, we probed the vulnerability types and whether their frequency has been increasing or decreasing over the years. Second, we also investigated when they are normally introduced during the development life cycle. Table 4.1 list the most common vulnerability types.

Our results indicate that an extremely limited number of vulnerability types occur and most of them are introduced at implementation time. Out of the 705 software vulnerabilities in CWE, only 9 of them are regularly caused by WordPress plugins. As shown in Table 4.1, the three most common types of vulnerability are XSS with 397 occurrences (43.72%); SQL Injection with 166 occurrences (18.28%), and CSRF with 120 occurrences (13.12%). The least common type is Unrestricted Upload (1.21%), which allows arbitrary code to be uploaded and executed by the web application.

When investigating Table 4.1, we discovered four evolution patterns of vulnerability caused by plugins in web systems: (i) persistent vulnerability, which manifest itself often along the years (P1); (ii) occasional vulnerability, which occurs at irregular intervals over the years and thus has been reappearing from time to time (P2); (iii) once occasional vulnerability but now lays "dormant", which is a vulnerability that manifested itself then disappeared, after which it has not manifested itself again ever since (P3); and (iv) occasional vulnerability that has become prevalent in plugins (P4).

Table 4.1: Lists the most common vulnerability types

Type	CWE	Total	Time of Introduction
XSS	CWE-79	397	Implementation
SQL Injection	CWE-89	166	Implementation/Operation
CSRF	CWE-352	120	Implementation
Path Traversal	CWE-22	48	Implementation
Management of Permissions	CWE-264	28	*De*sign
Improper Input Validation	CWE-20	21	Implementation
Code Injection	CWE-94	20	Implementation
Information Exposure	CWE-200	19	Implementation
UNrestricted Upload	CWE-434	11	Implementation

Notably, the majority of the vulnerability types caused by plugins in WordPress falls into pattern P1: meaning that 76.31% of these vulnerabilities

keep cropping up through the years and that security is still not increasing over time. We conjecture that vulnerabilities that follow pattern P1 are the easiest to be introduced. For instance, XSS and SQL Injection are types of vulnerability in pattern P1. The former, according to (2), is very common in web systems because it requires a great deal of knowledge about program construction and thus it should be avoided altogether. Indeed, whenever developers are negligent during the PHP code writing process, they are prone to lapses that may result in the introduction of XSS vulnerabilities into plugin code. Regarding SQL Injection, we surmise it as a common vulnerability because it is easily detected and exploited. Moreover, as observed in (2), any web system with even a minimal user base is likely to be subject to an attempted attack of SQL Injection.

In general, we also noted only 10.66% of vulnerabilities associated with patterns P2, P3, and P4. Code injection is the only type in P3. We are convinced that code injection was once common and the almost disappeared because developers started avoiding unsafe resources as *eval()*. However, in contrast, other vulnerability types associated with improper input validation as Path Traversal and Unrestricted Upload have become prevalent. We understand that proper input validation is still a major challenge faced by plugin developers, especially because many of the recently developed plugins have been created by not-so-skilled developers (14).

4.2.2 RQT.2 – How critical are the vulnerabilities caused by WordPress plugin?

Out of the 895 vulnerability bulletins analyzed, only 78 (8.71%) indicated that the vulnerability would not lead to a significant compromise of the web system integrity. However, 804 bulletins (89.83%) suggested that the vulnerability had the potential to allow attackers unrestricted access to files or to information maintained by the vulnerable web system, even when the scope the attackers could have had access to was limited. Only 15 reports (1.67%) described vulnerabilities that had the potential to jeopardize the integrity of the web system as a whole. Therefore, our results suggest that the majority of the vulnerabilities have to potential to lead to data breaches while a few vulnerabilities do not expose web systems to serious security risks. Moreover, fewer vulnerabilities (less than 2%) had the potential to render the whole system vulnerable.

We also looked at the impact of the vulnerabilities on the availability and confidentiality of web systems(45)(53). As shown in Table 4.2, the number of vulnerabilities that do not affect, or that only partially affect the web systems

Table 4.2: The number of vulnerabilities that do partially affect the web systems

Type	**L**	**M**	**H**	**N**	**P**	**C**	**N**	**P**	**C**	**N**	**P**	**C**
CWE-19	1	–	–	–	1	–	1	–	–	–	1	–
CWE-20	12	9	–	1	18	2	9	10	2	5	14	2
CWE-22	46	2	–	43	4	1	44	4	–	3	43	2
CWE-59	1	–	–	1	–	–	1	–	–	–	1	–
CWE-77	2	–	–	–	1	1	–	1	1	–	1	1
CWE-78	1	–	–	–	1	–	–	1	–	–	1	–
CWE-79	1	388	8	–	397	–	397	–	–	397	–	–
CWE-89	160	6	–	–	166	–	–	166	–	–	166	–
CWE-94	14	6	–	–	18	2	–	18	2	–	18	2
CWE-200	19	–	–	19	–	–	19	–	–	–	19	–
CWE-254	1	–	–	–	1	–	1	–	–	1	–	–
CWE-264	23	5	–	5	21	2	13	13	2	11	15	2
CWE-284	6	–	–	2	4	–	4	2	–	3	3	–
CWE-285	2	–	–	–	2	–	2	–	–	2	–	–
CWE-287	4	1	–	–	5	–	1	4	–	2	–	–
CWE-352	2	117	1	2	117	1	7	112	1	5	114	1
CWE-434	1	–	–	–	10	1	3	7	1	3	7	1
CWE-552	1	–	–	1	–	–	1	–	–	–	1	–
CWE-601	–	1	–	–	1	–	1	–	–	–	1	–
NO INFO	11	–	–	–	8	3	4	4	3	4	4	3
OTHER	16	16	1	2	29	2	8	23	2	2	29	2

are somewhat similar. The results show that 516 (57.65%) vulnerabilities do not compromise availability. Nevertheless, according to the reports, 365 (40.78%) vulnerabilities have the potential to negatively affect the performance of the web system or partially render resource unavailable. Similar result can also be observed when we consider confidentiality. A considerable number of vulnerabilities, that is, 437 (48.82%) do not affect the confidentiality of the web system while 442 (49.38%) of them enable the attacker to access some system files. Only in 16 (1.78%) reports the vulnerability poses a security breach that allows the attacker to take over the vulnerable site and access sensitive data.

As shown in Table 4.2, 551 (61.56%) vulnerabilities require some special access conditions. However, in 334 (37.31%) cases, the vulnerabilities introduced by plugins do not impose any restriction to attackers nor hamper their access to certain files. That is, any attack can be performed manually and

requires little, to no skill to be carried out. Indeed, SQL Injection (CWE-89) seems to be the most dangerous vulnerability type. Although, the reports indicate that SQL Injection attacks only partially affect the integrity, availability, and confidentiality of the systems, most of the cases suggest that these vulnerabilities can be easily exploited by attackers.

4.2.3 RQT.3 – What is the patch size to fix WordPress plugin vulnerabilities?

We analyzed the impact of maintaining vulnerabilities caused by plugins based on the patches available on the WordPress plugins repository. According to the standard deviation, the size of the patches in terms of the number of lines of code seems to vary considerably. Therefore, the median is a more representative measure of central tendency for our data sample: that is, our results seem to suggest that, on average, 11 lines of code are needed to fix a vulnerability. Considering the 119 patches analyzed, we noted that most changes occur in only a few files. On average, fixing a vulnerability entails changing only one file. According to our analysis, the worst-case scenario involves having to change up to 8 files (Table 4.3).

Table 4.3: Reflect the number of lines of code and files that were modified to correct vulnerabilities

	Llines A	**Lines C**	**Lines R**	**Files**
Min	0	0	0	1
Max	20	12	21	18
Mean	1,96	1,95	1,11	1,7
Median	0	1	0	1
Std. Dev.	3,64	2,51	3,44	2,33

Analyzing the top three patches we observed that they are large because their fix entailed sanitizing a large volume of parameters. For example, the multiple XSS vulnerabilities in wp-login.php in the Genetech Solution Pie-Register plugin (8) was fixed in patch 74024912, which entailed adding and modifying approximately 900 lines of code spread across 2 files to sanitize the input parameter of the Settings and Register Forms. Similarly, to fix two vulnerabilities (9)(10)(46)(50) in the Video Embed & Thumbnail Generator plugin: one allowing attackers to obtain the installation path via unknown vectors and another allowing remote attackers to execute arbitrary commands: the developers in patch 50792415 had to change almost 750 lines of code by implementing the

Media Upload Form and Generator functionalities to basically process input data. This more in-depth analysis, however, indicates that the effort involved in fixing post-release security vulnerabilities is significant only in some very specific cases.

4.2.4 RQT.4 – How long does a vulnerability survive in the WordPress plugins code?

The duration of a vulnerability transits through different states, ranging from the introduction, through disclosure, to the release of a patch that corrects the security threat (4). The introduction or birth of vulnerability occurs unintentionally during the development cycle. Vulnerability is revealed when the discoverer reveals details of the security threat to a wider audience. Finally, a vulnerability is solved when the developer releases a patch that corrects the security threat. We consider the number of days between the introduction of vulnerability (Table 4.4) and its fixation as its survival time.

Table 4.4: Reports the survival time of security vulnerability in the plugins code extracted from the 119 patch analyzed

	Number of Days
Min	1
Max	1.710
Mean	563,41
Median	470
Std. Dev.	1,34

Although disclosure might accelerate patch release, as observer by Arora et. al. (5), the number of days until a vulnerability is fixed after it is introduced in the code of a WordPress plugin is on the other hand, considerable. On average, it takes 653 days until the fixing of a vulnerability after its introduction. This means that a vulnerability could remain unnoticed in the web application for years before being identified and the fixed. For example, the vulnerability CVE-2015-441318 has been fixed in the commit 117873619 made on 6th November 2015. The commit 60349120 performed on 25th September 2012 was identified as the vulnerability-introducing commit. Therefore, a simple to be corrected but very severe vulnerability survived in the code for 989 days (2 years, 8 months, 10 days).

4.2.5 RQT.5 – What are the most common vulnerability related topics discussed among developers?

Table 4.5 shows the distribution of questions (Q) and answers (A) per topic as well as the ratio of answers per questions (A/Q), means of views (Mv), and means of score (Ms). Table 4.5 provides us with interesting result. For example, Server Security is the topic with the least amount of questions and answers and Database Security is the one with most questions and answers. Questions about Server Security usually focus on how server administrator decisions may prevent an attacker from exploiting a vulnerability and from gaining access to sensitive internal information. Although the topic Server Security has the highest Score (5.09), we believe that this category has a low number of questions because plugin developers are not concerned about server-side security. Indeed, we found an outlier question with a Score of 48 and 3,108 views, which alone is responsible for 64.20% of the Server Security topic popularity. Analyzing this question 16, we found that it very popular because it brought up an appealing discussion regarding a collaboratively-edited list of "high-end WordPress webhost" for those "WordPress site that needs really hardened security".

Table 4.5: Show the distribution of question (Q) and answers (A) per topic

Topics Name	Q	A	A/Q	Views	Score
Server Security	11	19	1,72	440,09	5,09
Plugin Update	27	33	1,22	386,77	1,37
File Upload	29	36	1,24	1.835,62	1,93
Permissions	34	51	1,5	1.801,32	1,32
Cryptography	38	54	1,42	991,02	1,84
SSL Certificate	42	45	1,07	1.209,28	0,57
Authentication	43	59	1,37	1.947,55	1,37
Content Security	43	60	1,39	855,04	1,16
Programming	50	63	1,26	817,08	2,4
Database Security	291	410	1,63	794,16	1,63

It is also worth mentioning that the number of views can vary significantly. For instance, Authentication is the topic that has the highest average number of views (1,947.55) and Server Security has the lowest (440.0). Upon analyzing the most visualized questions in Authentication, we noted that they

are very general and recurrent questions about best practices on authentication and authorization. In contrast, we did not observe a considerable discrepancy among question scores. In 2 out of 8 cases, the question scores are close to 1.50, which means that the number of "up-votes" are not more than two times bigger than the number of "down-votes". Therefore, we can conclude that in general Server Security contains few, but relevant, questions and although SSL Certificate is a very popular topic, averaging more than 1200 views, some of the questions are poorly elaborated.

Finally, we investigated how tags are used in StackExchange WordPress, which emphasizes the use of tags to categorize and group questions. We found a total of 325 distinct tags. The five most common tags are: (i) login – 35 occurrences; (ii) updates – 29 occurrences; (iii) ssl – 28 occurrences; (iv) multisite – 26 occurrences; and (v) database – 25 occurrences. These results corroborate to indicate that Authentication, Authorization, SSL Certificate, and Database Security are indeed the most common security-related topics discussed among WordPress developers.

4.3 Mitigation and Detection Methods

Following the typical methods to mitigate vulnerabilities does not guarantee that a given plugin is secure, but it does, however, set a good security baseline. As the results indicate, if plugin developers manage to fully understand, detect, and eliminate XSS, SQL Injection and CSRF, they will be developing plugins with 76.31% less vulnerabilities. To contribute to mitigation of vulnerabilities we summarize some methods that can be used to make plugins code more reliable.

(i) Understand how data is used and the encoding expected by the core parts of the plugin-based web systems (e.g. WordPress, Drupal). This is especially important because plugins have to send input data in specific patterns to many internal components. In this case, developers should assume all input as malicious and reject any input that does not strictly conform to the required encoding pattern, or transform it into a valid input. For example, WordPress provides some helpfully PHP functions such as: *esc html* – escapes HTML to safely output processed inputs; *esc js* – escape Javascript to be used in tag attributes; *sanitize text field* – sanitizes a string checking for invalid characters.

(ii) For instance, to protect database from SQL Injection, developers should use parameterized SQL statements by separating code from data manip-

ulation. Moreover, it is important to check the headers to verify whether the request is from the same origin. Another good practice is avoiding the *GET()* method for any request that triggers a state change.

Unfortunately, there is not method able to detect the most common code weakness with 100% of accuracy and coverage. Even modern techniques that use data flow analysis to detect Cross-site Scripting (25). Indeed, Fonseca and Vieira (16) set out to study the effectiveness of static code analysis tools to detect vulnerabilities in plugins of web systems and understand the potential impact of those vulnerabilities in the security of the core web system. Their results suggest the coverage analysis performed by static analysis tools is inefficient and these tools report false positives often. In contrast, SQL Injection vulnerabilities are usually more effectively detected by dynamic analysis methods. However, it is well-known that even dynamic analysis methods have drawbacks (1). It is also worth mentioning that not even a thorough software testing process can guarantee the detection of all possible vulnerabilities.

(i) Software test: test process designed to ensure that the software complies with the functional requirements for which it was designed. (21)(22).

Creating and maintaining a test suite demands many human resources and executing large test suites might take too long. Therefore, creating secure plugins (devoid of vulnerabilities) poses a complex challenge to developers as testing all possible plugins combination is in general impractical (18)(23)(24).

Manual analysis can be useful for finding SQL Injection, but it might not achieve the desired code coverage within limited time constraints. This becomes difficult for weaknesses that must be considered for all inputs, since the attack surface can be too large. Consequently, some organizations as OWASP stress the importance of a balanced approach that includes both manual reviews and automated analysis that covers all phases of the web system development process. Code recommenders techniques (6)(12) seems to be useful in this context as it can be used in some cases to present a list of all possible sanitization methods, allowing a developer to browse the proposal and to select the appropriate one from the list.

However, as sanitization methods should be recommended with respect to specific contexts (vulnerable codes) we still be missing a set of metrics that could help tools to effectively recommend the places most susceptible to have a vulnerability and the most appropriated sanitation methods (15)(29).

4.4 Threats to Validity

The validation of the exploratory research was divided into two ways to be carried out, an external validation and another internal validation was carried out.

4.4.1 External Validity

The study was limited to the analysis of 895 NVD vulnerability bulletins associated with WordPress plugins. Therefore, the results cannot be generalized to other plugin-based web systems (e.g., Drupal and Joomla). Given that plugin-based web systems projects vary on characteristics as number of participants, community structure, and governance, we cannot draw general conclusions about all projects (i.e., the whole population). To build reliable empirical knowledge, we need a family of experiments (7) that include plugin-based web systems of all types. However, with its 52,000 plugins, WordPress is a relevant and highly popular ecosystem. Moreover, we think the impact of this threat is minimal for three reasons: (i) most plugin-based web system provides the same basic programming model; (ii) the focus of the study was on security vulnerabilities rather than on the specifics of the software systems themselves, and (iii) none of the other plugin based systems focus specifically on promoting secure coding, so there is no clear reason that one would be favored over the other in our study.

4.4.2 Internal Validity

The data analysis method is another aspect that can negatively affect the results: the study was based on a manual verification of both the NVD vulnerability bulletins and the patches collected from the WordPress Plugins repository. We also manually analyzed and eliminated non-related questions extracted from the WordPress Stack Exchange Q&A Website. Independent validation of our results can also be performed once we made all of our data publicly available. Another threat is the validity of our survival time measure (i.e. number of days until a vulnerability was fixed properly). Because most patch include large changes rather than only vulnerability-fixing changes, measuring the extent of the modifications is challenging (40)(41).

Another construct validity threat is that if the keyword set was incomplete, the automatic extraction could have missed some vulnerability bulletins or Stack Exchange questions. To validate the completeness of the set of key-

words, the authors independently inspected randomly chosen NVD bulletins and Stack Exchange questions that did not contain any of the keywords used. The manual review of those assets found any relevant missed case. We applied this analysis to increase our confidence in the soundness of the keyword set.

4.5 Chapter's Conclusions

We could observe that plugin developers still do not know how to write secure code, mainly because they are not-so-skilled developers. Indeed, given the nature of the most common vulnerabilities, it is clear that developers do not understand common attacks, the security functions provided by the language and platform they use, and how certain easily applicable practices can mitigate security problems. We found that the time window of addressing security vulnerabilities post-release can be high is some cases. Overall, it takes less time to address vulnerabilities when they are found earlier in the plugin development lifecycle. Thus, our recommendation is to develop new methodologies tailored to plugin developers, which promote security in the requirements and design phases. In this case, we have to consider that developers are the most important entities in software security. As we mentioned, automated solutions tend to fail to detect some vulnerabilities and inexperienced software developers exhibit a lack of texting skills (49). Worse, strategies as social transparency (48) seems to not influence the testing behavior of plugin development teams as observed in others context (14)(55). As we could observe, the quality of open source plugins does not increase over time as the frequency of disclosed vulnerability does not decrease over time.

The results indicate that we need tools that enable novice developers to reliably build secure plugins. That is, those tools need to be usable by developers without any security background. Unfortunately, the well-established security tools cannot operate at the speed required to achieve a more proactive security. These tools are based on the disclosure of existing vulnerabilities and are incompatible with the development of modern software, since they lack the options that facilitate rapid adaptation to the possible emergence of new security vulnerabilities. Techniques that we can find when combining the methods of the current tools with machine learning techniques.

5
A Tool for Supporting the Mitigation of the Occurrence of Vulnerabilities

Taking into account the information reflected in the previous chapter, we will dedicate this chapter to describe how we carry out the development of our plugin and the tools we use to build it. Directing the chapter from the motivation section, we will present the demo design, and how our demo operates for the detection of vulnerabilities in files with PHP code.

5.1
Motivation for building the Tool

After the study shown at Chapter 4, we use this chapter to describe and analyze our proposed solution. For analysis the tool proposal we will analyze this architecture, the operating process and the current state of the tool. We describe was the tool created, as was the implementation process and the tests performed up to the state in which the application is located.

The goal of this Chapter is to explain our analysis was carried out to help mitigate the possibility of security vulnerabilities occurring in web applications. As we indicated in the study at Chapter 4, the area selected for the study of the behavior of the vulnerabilities were the security vulnerabilities associated with WordPress obtained from the NVD Repository. The study was conducted for the vulnerabilities of websites based in Plugin (created using PHP) and WordPress.

A precondition for the vulnerabilities registered in the NVD Repository must have been generated by the addition of a Plug-in to WordPress. We recognize that the limitation of using only the vulnerabilities recorded in the Repository limits the investigation, but the Repository is used as a starting point for the investigation. We added as a precondition, that we use the Plug-in directly connected with WordPress (Figure 5.1); and we ignore in this first stage the Plug-in connected with WordPress through another Plug-in (Figure 5.2).

The Figure 5.2 shows Plug-in Y, marked with a red cross, representing those Plug-in that we neglect in this first phase of creation of our tool. Rejected for its indirect connection with WordPress, connected with another Plug-in X.

Figure 5.1: Case study identified. Plug-in directly connected to WordPress

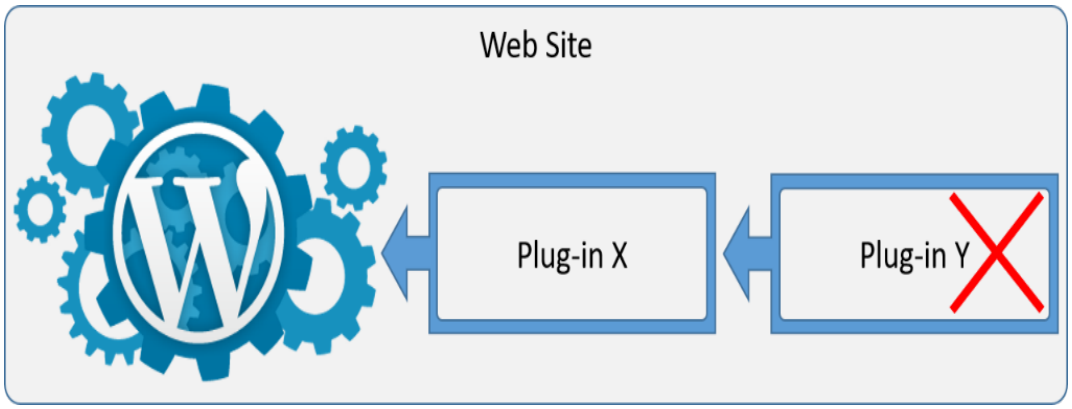

Figure 5.2: Case study rejected. Plug-in connected through another Plug-in with WordPress

Due to their complexity, plugin-based web systems are also plagued by vulnerabilities (28). WordPress provides an infrastructure to which plugins can contribute new features by modifying shared global state or running a function at a specific point in the execution of the core. For example, The WordPress login form works properly in isolation, but the introduction of the aforementioned plugin causes a vulnerability as shown in Figure 5.3. Given that, WordPress allows millions of developers across the world to create their own plugins, it is unrealistic to anticipate all possible plugin installations, interactions, and combinations in a specific web application. As a result, it is clear that security can be violated by unanticipated plugins interactions with the core, creating vulnerabilities that can be potentially exploited by attackers (16)(29)(42). Thus, the security of these web systems hinges on keeping both core and plugin vulnerabilities in check.

Walden et al. (28) carried out a study to observe whether plugins are the source of vulnerabilities present in web systems and they confirmed that most

```
340. 340.  add_action('login_form', 'new_add_fb_login_form');
341. 341.
342. 342.  function new_add_fb_login_form() {
...  ...
361. 361.  if(!window.fb_added){
362. 362.     socialLogins.prepend('<?php echo
               addslashes(preg_replace('/^\s+|\n|\r|\s+$/m',
               '', new_fb_sign_button())); ?>');
...  ...
369. 369.  }
...  ...
429. 429. function new_fb_sign_button() {
430. 430.
431. 431.   global $new_fb_settings;
```

```
432.       return '<a href="' . new_fb_login_url() .
                  (isset($_GET['redirect_to']) ?
                  '&redirect=' . $_GET['redirect_to'] : '') . '"
                   rel="nofollow">' .
                   $new_fb_settings['fb_login_button'] . '</a><br />';
```

```
    432.   return '<a href="' . esc_attr(new_fb_login_url() .
             (isset($_GET['redirect_to']) ? '&redirect=' .
             $_GET['redirect_to'] : '')) . '" rel="nofollow">' .
             $new_fb_settings['fb_login_button'] . '</a><br />';
```

```
433.  }
```

Figure 5.3: Example of vulnerability produced by use of a plugin

of the vulnerabilities (92%) appear in plugin code. In addition, according to their results, plugin code differs from core code in terms of the types of vulnerabilities present in the results. Fonseca and Vieira (29) suggest that many plugins have Cross-site Scripting and SQL injection vulnerabilities that can be easily exploited. Moreover, they found that the coverage analysis performed by static analysis tools is inefficient and these tools often report false positives. Koskinen et al. (42) analyzed 322 randomly selected WordPress plugins. More specifically, they compared the amount of potential vulnerabilities and vulnerability density to the user ratings in hopes to determine if user ratings can be used to select secure plugins. The results suggest that the quality of plugins varies and that there is no correlation between the ratings of plugins and the amount of vulnerabilities found in them.

We conducted a study on the code of the vulnerable files found in the Repository to detect the behavior of security vulnerabilities. We demonstrated in Chapter 4, most vulnerabilities are detected and arranged in a single file. In addition, the variability in the behavior of security vulnerabilities is not very broad, and therefore we decided to make a first definition of Patterns with the detected and concerted vulnerabilities in only one line of code.

A first step for our proposed solution was to divide the files obtained from

the NVD Repository, into two groups. A first group for the creation of Behavior Patterns of vulnerabilities. For create the Behavior Patterns (Annex B), we use a small group of the files collected from the total files collected from the NVD Repository. Behavior Patterns were created respecting the correct code syntax and the frequency of appearance of the proposed structures. The division of the groups of files was performed randomly and the identification of the first Patterns of behaviors was done manually. Most of the patterns detected have the keyword "echo", this is due to the high frequency of use of this word; which showed us that we cannot give importance to the keyword if it is not within the structure we identify.

With the second group, the efficiency tests of the Behavior Patterns created with the first group were carried out. Given the results of the exploratory study shown in Chapter 4, the vulnerabilities of greatest occurrence are vulnerabilities of type XSS, SQL Injection and CSRF, so the first patterns of behavior created are for to anticipate these types of vulnerabilities.

5.2 Functional Requirement

A Functional Requirement, has to express and show the composition of a software (30)(31) to the needs of the user (be understood by a natural person who interacts with the software or software that needs the operation or service offered by other software), we define the following Functional Requirements:

(i) Ability to connect to the IDE Eclipse: because our tool is a plugin to combine with the Eclipse development IDE, we understand that we should not neglect this aspect. Our plugin does not need specific configuration, so it facilitates its connection and only needs the basic configuration for communication with the development IDE.

(ii) Path where the files to be tested are located: in this requirement it presents two variants for the location of the files to be tested. As a first variant, we understood that if the plugin does not receive a location path for the files then, the files that are in the same location of the project being worked on at the time of the test should be scanned. As a second variant and given the possibility of testing several files, communicated with each other or not, belonging to the same project or not; We offer the possibility to write the path of the folder in which the files you wish to test are located.

(iii) The files must contain PHP code: we are talking about a plugin to test files with PHP code regardless of their extension, but it is not objective

to perform the test to detect possible security vulnerabilities if the files that will be tested do not contain PHP code. The absence of PHP code will not cause any error in reading the file, simply the plugin will try to perform the test process without obtaining apparent results due to the absence of PHP code.

(iv) The Knowledge Base must be connected to the plugin and updated: the Knowledge Base must be duly updated, and with a service that allows update yourself of the repositories recommended during the study with a frequency not exceeding 3 months, due to the amount of vulnerabilities that are recorded every year.

(v) Detect the Patterns of Behaviors created in the archives tested: the plugin must be able to use the information of the Knowledge Base to detect the Patterns of Behavior in the archives that are tested for the Detection of Possible Vulnerabilities. In the same way you have to be able to offer the Possible Solution stored in the Knowledge Base.

(vi) Architecture that allows the creation and incorporation of new functionalities for the detection of new Behavior Patterns: For this the plugin presents an architecture based on the Methodology of Multiagent Systems, where each agent can be dedicated to the detection of the Behavior Patterns according to, they want to group. The Patterns of Behavior can be grouped by the Possible Vulnerability in which they are found or in several groups considering the similarity between them.

(vii) Inform the client of the Possible Security Vulnerabilities and Possible Solutions: after detecting the Behavior Pattern in the code line where a Possible Vulnerability may occur, our Plugin must inform the user of the existing solution and stored in the Knowledge Base.

Of the Functional Requirements identified for the realization of the plugin, we apply in this demonstrative version the following requirements:

(i) Path where the files to be tested are located: we apply the second variant, in which we ask for the path of the folder where the files to be tested are located. The plugin requests the address of the folder by console and does not continue the process until some path is informed.

(ii) The files must contain PHP code: we created this first version capable of working in the various formats in which PHP files can be found. Respecting the various formats in which you can find the PHP code.

(iii) Detect the Behavior Patterns created in the tested files: this version detects the Behavior Patterns of the Possible XSS Vulnerabilities, using the Behavior Patterns stored in the Plugin's own code.

(iv) The Knowledge Base: A Knowledge Base stored in the Plugin code was used, which is initialized whenever the execution of the Plugin is invoked.

(v) With the fulfillment of these last Functional Requirements, we understand that they are enough to demonstrate the possibility and the viability of the use of Behavior Patterns for the anticipation of Possible Security Vulnerabilities in the applications made with PHP Code. Demonstrating that there are ways to anticipate and mitigate the occurrence of Security Vulnerabilities.

5.3 Technologies used for Build the Tool

IDE Eclipse([1]): is a free Java IDE for developer. It is mostly written in Java. Eclipse lets you create various cross-platform Java applications for use on mobile, web, desktop and enterprise domains.

Its contains a base workspace with an extensible plug-in system for customizing the IDE to suit your needs. Through plugins you can develop applications in other programming languages. These include C, C++, Java, JavaScript, PHP, and other. Eclipse is available under the Eclipse Public License and is available on Windows, Mac OS X and Linux. This IDE that is very popular among the community of developers of the Java Language([2])([3]) using the JDT Plug-in that is included in the standard IDE distribution. It provides tools for managing workspaces, writing, deploying, executing and debugging applications (34)(35). Both Eclipse and Java enjoy great popularity in the community of programmers.

For the creation of our tool we use Java as programming language, while we are still studying which database NoSQL will be used.

[1]https://blog.idrsolutions.com/2015/03/the-top-11-free-ide-for-java-coding-development-programming/

[2]http://personales.upv.es/rmartin/cursoJava/Java/Introduccion/PrincipalesCaracteristicas.htm

[3]https://www.12caracteristicas.com/java/

5.4 Tool Design

Manually grouping the vulnerabilities by the known classifications and by the CWE classification, combined with the WordPress information reflected on the official CMS site; a group of patterns will identified that will allow us to identify the possible vulnerabilities from the process of creating future plugins in PHP language for WordPress. To strengthen the study of patterns identification of security vulnerabilities, we will combine our study with the different access points that WordPress offers to add a plugin.

Among the various behavior patterns detected, we find that whenever the *echo* syntax appears *htmlspecialchars* (variable); increases the possibility of occurrence of an XSS vulnerability. An example of where the vulnerability can be found according to the proposed pattern is displayed in the *CVE-2011-4562*([4]) (Annexo D) vulnerability refueled in the NVD Repository. For the studies carried out whenever we find the behavior of this Pattern can be solved by changing the *htmlspecialchars* function by the *esc attr* function. This first example of the Vulnerability and Solution Pattern would have the following behavior.

Vulnerability: echo htmlspecialchars (variable)
Solution: echo esc attr (variable)

Another example of Repeated behavior pattern, we find it in the lack of sanitization of the *GET* methods widely used in web applications. In vulnerabilities *CVE-2012-1068*([5]) and *CVE-2011-5106*([6]), the GET method is used to scroll through the various pages of the website. The lack of analysis and conversion of the GET method parameter generated an XSS vulnerability. In this Pattern the String sent as a parameter is the number of the page to which you want to access.

Vulnerability: variable = _GET [String]
Solution: variable = (int) _GET [String]

The patterns were formed after observing the same behavior in the code of several PHP files of the Plug-ins studied. It is valid to remember that the frequency of occurrence of vulnerability by Plug-in is from 1 to 1, each vulnerability registered in the NVD Repository belongs to a single Plug-in. This detected relationship between the vulnerabilities registered in the NVD Repository and the Plug-ins referenced in each vulnerability, validates the created Patterns.

[4]https://nvd.nist.gov/vuln/detail/CVE-2011-4562
[5]https://nvd.nist.gov/vuln/detail/CVE-2012-1068
[6]https://nvd.nist.gov/vuln/detail/CVE-2011-5106

Based on the architecture proposed in Figure Figure 5.4, we developed a Plug-in's tool in the Java language using the Eclipse IDE for its creation. The tool was created to be used in the Eclipse development IDE. It was created to contribute to the security of other Plug-ins created with PHP language. The Eclipse IDE has several Plug-ins that facilitate the creation of efficient and functional codes, for none capable of preventing the existence of possible vulnerabilities in the code.

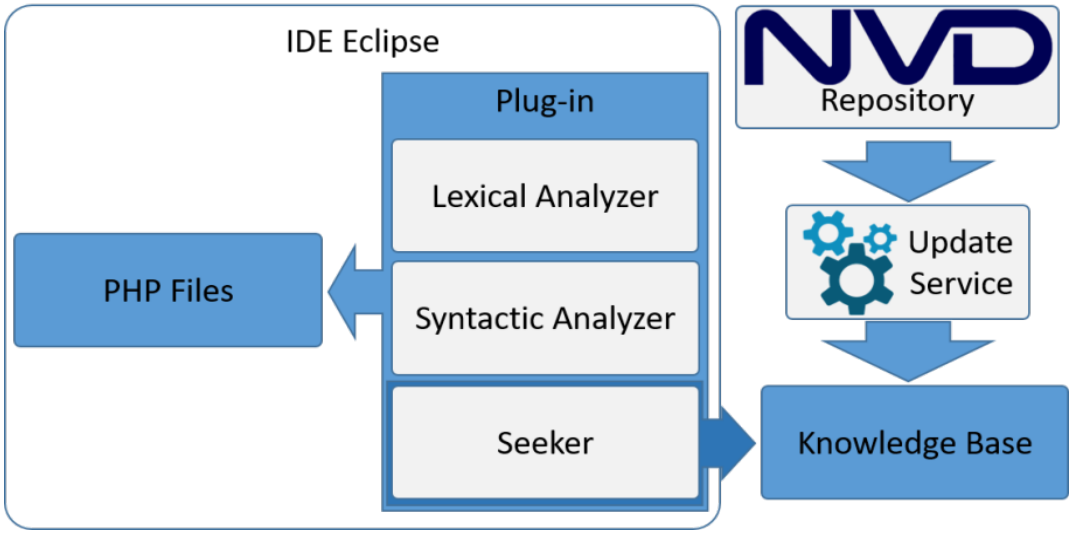

Figure 5.4: Architecture proposal for the tool

We developed the tool in Eclipse for being an open source integrated development platform (IDE open-source)(47), designed to be extended indefinitely through Plug-ins. It is a generic IDE that is very popular among the community of developers of the Java Language([7])([8]) using the JDT Plug-in that is included in the standard IDE distribution. It provides tools for managing workspaces, writing, deploying, executing and debugging applications (34)(35). Both Eclipse and Java enjoy great popularity in the community of programmers.

On the other hand, PHP([9])([10])(32) is an interpreted language and widely used in the construction of Web Applications and the basis of WordPress code. In the same way, the files with PHP are completed with other languages such as Java Script, HTML and JSON, just to mention a few. The variety of languages that complement the PHP language make it difficult to read these files, so we

[7]http://personales.upv.es/rmartin/cursoJava/Java/Introduccion/PrincipalesCaracteristicas.htm

[8]https://www.12caracteristicas.com/java/

[9]https://www.sitesbay.com/php/php-features-of-php

[10]https://www.w3schools.com/php/php_intro.asp

limit our study once again to identify patterns, to read the files between the PHP code.

For the creation of the Plug-in‘s tool and based on the architecture proposed in Figure Figure 5.5, we created a tool able to analyze the files with PHP extension. Because PHP is an interpreted language and does not need to be compiled, we use the principles of a compiler for the construction of the Plug-in. Our tool does not aim to detect errors, so we do not need to create a compiler in its entirety of principles, phases and operations. Of the three phases of the compiler, only the phases of Lexical Analyzer and Syntactic Analyzer were used. It was not necessary to use the Semantic Analysis phase. Our Plug-in is not intended to detect if the code contains errors or operating problems; remember that our goal is to be applied the Plugin in correct and efficient codes, to detect possible security vulnerabilities.

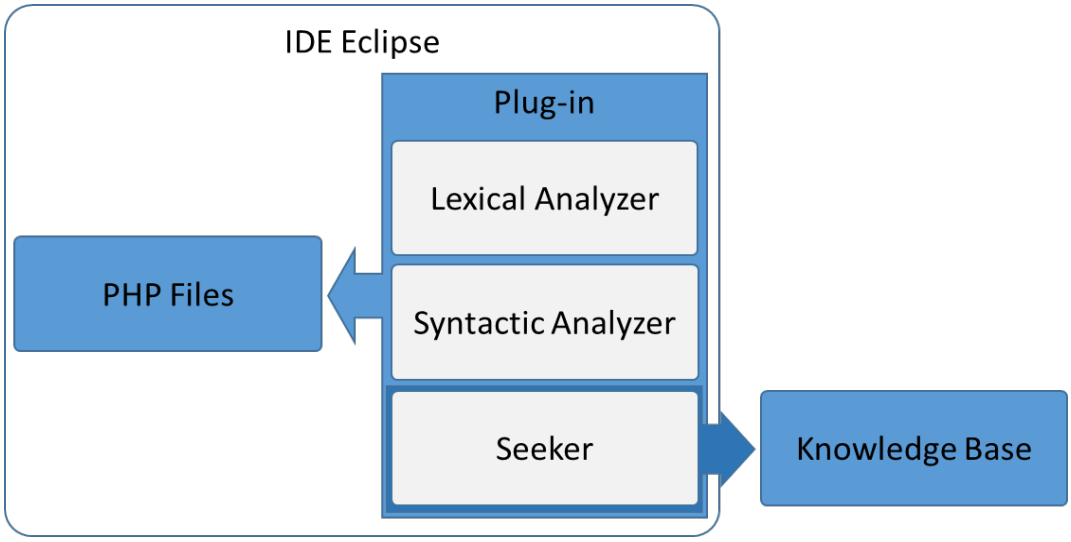

Figure 5.5: Architecture proposal for the tool made

For the creation of the Lexicon Analysis phase, we conducted a manual study on the PHP language and its keywords. The study was conducted in PHP manuals([11])(32) regardless of the version of the language described in the manual. Then, we repeat the same study done on the manuals, in the files collected from the vulnerabilities registered in the NVD repository. The choice of the files of the vulnerabilities chosen for the study was explained previously, and the division of the groups of vulnerabilities was carried out randomly.

In the first group of files we repeat once again the manual study to define the Behavior Patterns and obtain the first data of our knowledge base. With the defined patterns, the next step was to build the code for the Lexical Analysis

[11]https://www.w3schools.com/php/default.asp

and identify the PHP language tokens in the different files. With the tokens identified and using the Syntactic Analysis module, we created the Patterns of the possible vulnerabilities of type XSS, SQL Injection and CSRF.

5.5 How to use

To use the Plug-in you only need to run the Eclipse IDE, in a traditional way; and then append or have attached in advance our Plug-in for the detection of possible vulnerabilities. Established the conditions, it is necessary to load the PHP files; It is not necessary to load the entire Project, just upload the file or the files that you want to test. Remembering that the PHP language is not a compiled code, the Plug-in must interpret each of the lines of the open files with the Eclipse IDE.

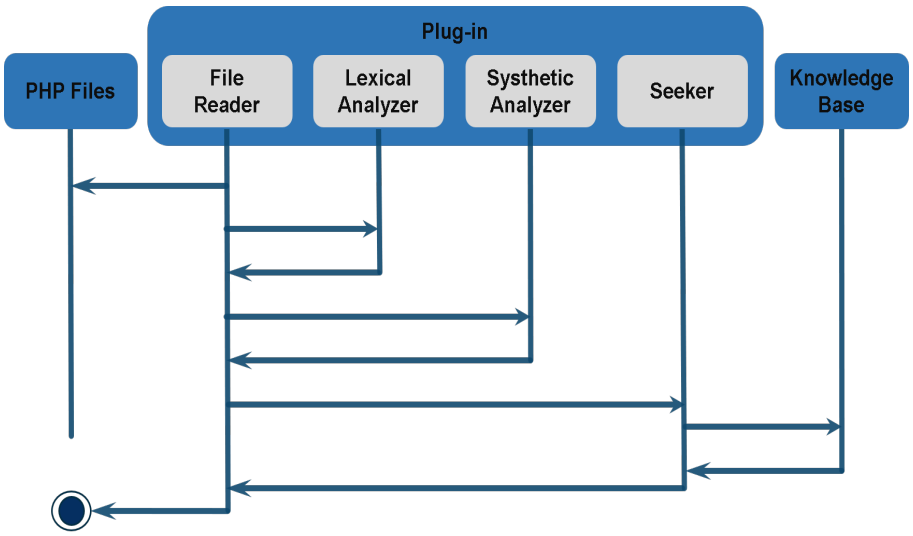

Figure 5.6: File Analysis Flow Diagram

Figure 5.6 shows the Flow of the Reading Process performed by our Plug-in, in Eclipse, to one of the PHP files. The reading is done directly by the module for the Lexicon Analysis. Once the different tokens detected in the file have been formed, the module for the Syntactic Analysis is activated and the sentence will be compared with the patterns found in the study. Then the Seeker module is activated that performs the search and comparison of the sentence elaborated by the Syntactic Analysis module in the Knowledge Base. From the Knowledge Base, the modification proposal to be shown to the developer is obtained.

This version was not developed with all the proposed functions but with the necessary modules to facilitate its easy expansion, growth and incorporation of new functionalities and operations. Example of the functions

we must add in a future version is the Update Service module (Figure 5.4). It is the module in charge of updating the Knowledge Base. Update that will be made with the information registered in the NVD Repository and with the modifications made in PHP files after having made a proposal for our Plug-in.

Continuing with the process of reading the files, this is done line by line, without making a fluid reading of the traceability of each variable or each function. The condition that we do not need to load the whole project and only some file(s) allows us to read the traceability of the variables, functions and operations, with occurrence in several files. It also allows us to isolate the different files, as well as the different variables and procedures created or used in them. We only analyze and give continuity to the variables and functions with occurrence in the same file.

5.6 Class Diagram

In this section we describe the Class Diagram (Figure 5.7) used to create the tool of the Plugin. For the creation of the tool, seven classes were necessary. It was necessary to simulate the principles of Lexicon Analysis and Syntactic Analysis of a compiler to read PHP files, the Knowledge Base was also simulated with one class.

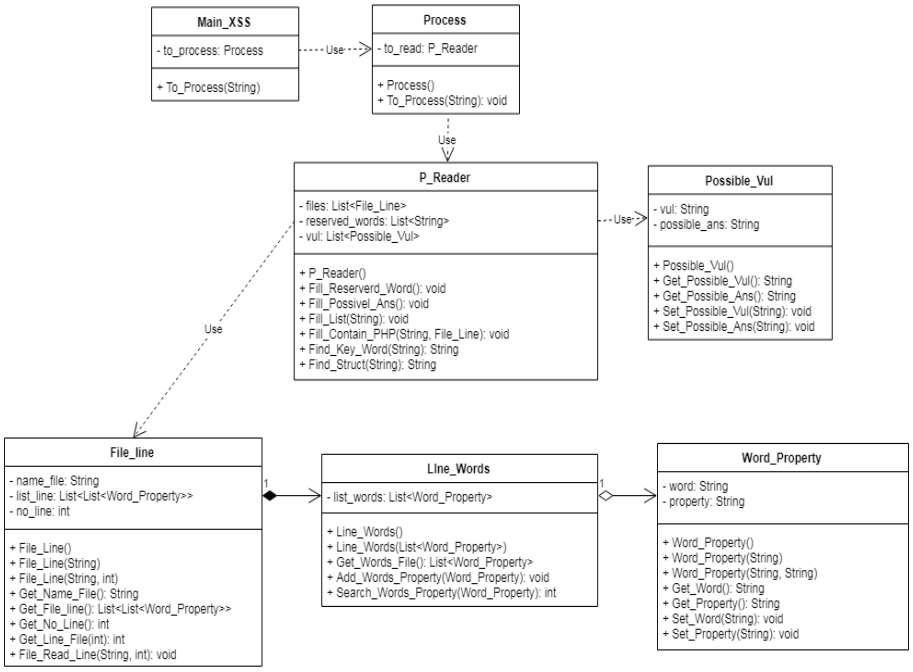

Figure 5.7: Tool Plugin's Class Diagram

The seven classes were divided in: one Class to start the tool of the Plugin and receive the necessary configuration, one class to start the process

of reading the files. Four classes for the construction the Lexicon Analysis, and the Syntactic Analysis. And a last class to simulate the Knowledge Base.

5.6.1 Class Main_XSS

This Class is used to receive the necessary configurations for the correct operation of the tool. It contains a single method that receives the necessary configuration's parameters. In this tool, only the address of folder where the PHP files that are to be tested are found is needed. This Class uses the Process class to deliver the information entered by the keyboard so that the tool can scan the files.

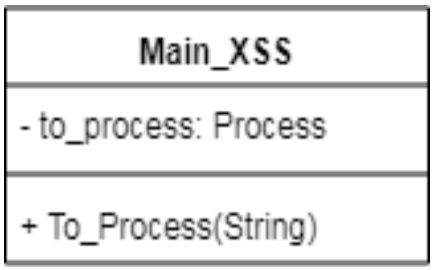

Figure 5.8: Class Main_XSS

5.6.2 Class Process

Created to receive the configurations, in this case the address of the folder where the files to be reviewed are. This class was created thinking about the final version of the Plugin, in this tool it is not very functional. This class uses the Class P_Reader to activate the operation of the plugin.

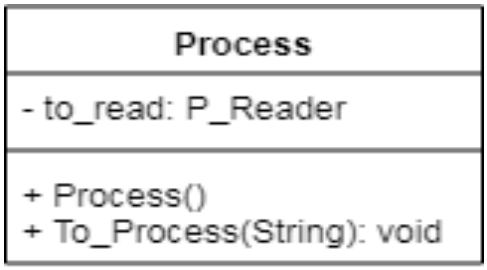

Figure 5.9: Class Process

5.6.3 Class P_Reader

This class has the highest workload. It is who guides the operation of the plugin. It is the class that guides the reading of the file and performs the search of the Patterns of behavior of the Possible Vulnerabilities and the proposals in solution in the Knowledge Base. Its methods guarantee the reading of the totally file, without any lost words.

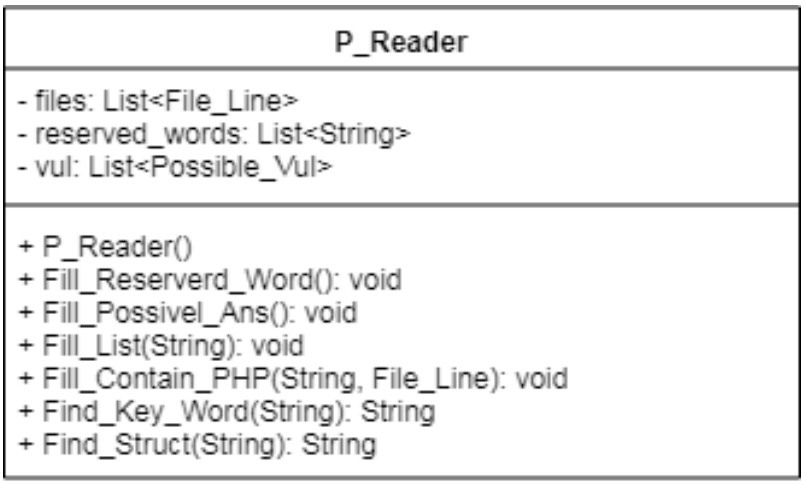

Figure 5.10: Class P_Reader

The **Fill_Contain_PHP**, **Fill_List**, and **Find_Key_Word** methods; are responsible for realizing the process of file's reading. The three (3) methods make up a structure of filtering and reading the code between the tags that identify the beginning and the end of a fragment's file with a PHP code.

Fill_Contain_PHP: is the method responsible for identifying the tags among the PHP code. With this method, the file is read line by line. All lines found between the opening tags of the PHP code **(<?PHP, <?PHP or <?)** are ignored and are not analyzed. Likewise, all lines found after the tags that close the PHP code **(?>)** are ignored. The tool only works with those included in the tags that define the opening and closing of the PHP code.

Fill_List: is the method responsible for storing in a list structure the PHP code lines of the file in which the reading is performed.

Find_Key_Word: this method makes a first contact with the class that simulates the Knowledge Base. In this first contact, each of the words of the lines of code stored in the structure of created list is carefully read. This reading is done by looking for words and symbols of the PHP code, in this way we differentiate the key words of the language from the words created by the programmer.

5.6.4 Class Fill_Line

Class that makes up the objects in the list structure where the different code lines found between the PHP code tags are stored during the reading of the file. The object has the name of the file in which the study is being carried out, the line where the reading is being made and the number that the line represents in the file. This class need the next class.

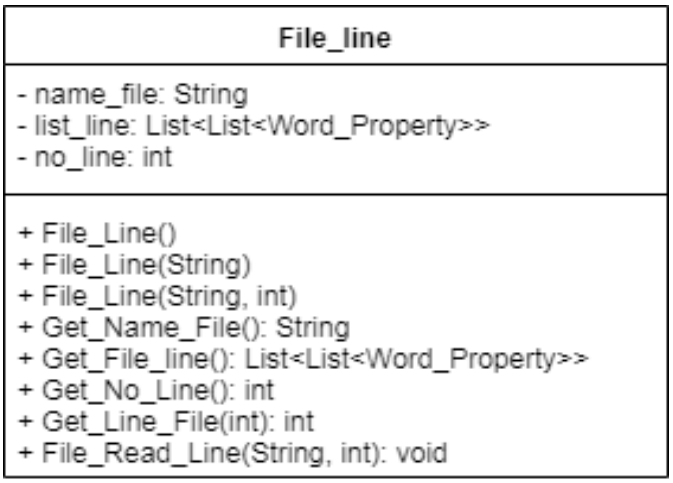

Figure 5.11: Class Fill_Line

5.6.5 Class Line_Word

This Class is used to read the lines of code in the file. The objects created by this class contain each word or symbol contained in the line code that is being studied. This Class provides the **Fill_Line Class** with words, which are specific to the language or not, and symbols found in the file among the tags that identify the PHP code.

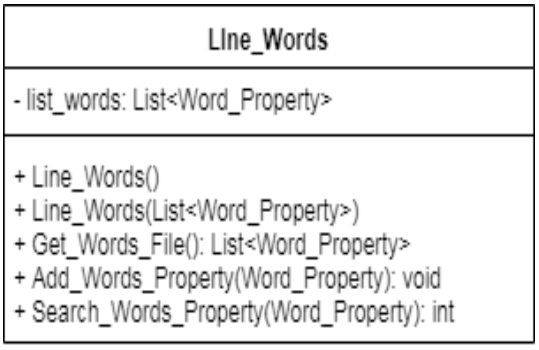

Figure 5.12: Class Line_Word

5.6.6 Class Word_Property

This class is used to create objects with the characteristics of each word and symbol found in the Line_Word Class. These objects are only going to store in the Property attribute the identification that defines whether the words or symbols are proper or not of the language. The key words and symbols of the PHP language must be properly identified for the proper functioning of the tool.

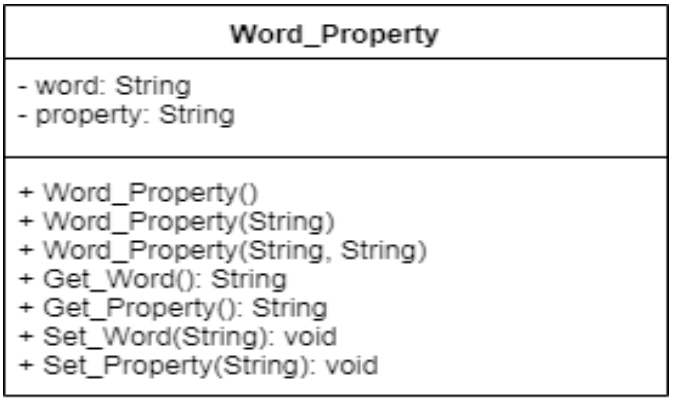

Figure 5.13: Class Word_Property

5.6.7 Class Possible_Vul

It is the class used to simulate the Knowledge Base. The patterns for detecting possible vulnerabilities and the variants of possible solutions for each possible vulnerability are stored in the list structures.

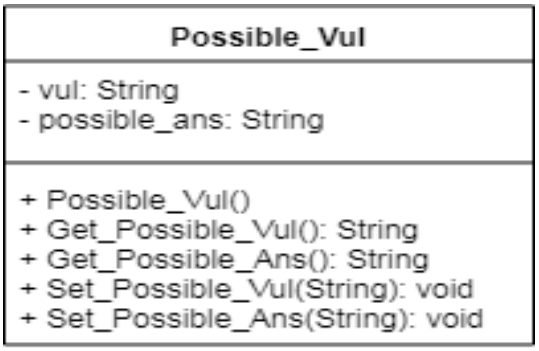

Figure 5.14: Class Possible_Vul

5.7 Highlight in the Code

For the creation of our tool, we developed the code capable of reading PHP files letter by letter. We use this reading system to identify the symbols and spaces that separate the words from the syntax of the PHP code. The identification is performed by reading the characters using ASCII code.

Table 5.1: Characters used for identify words

Character	**ASCII Code**
Space	**ALT + 32**
(	**ALT + 40**
)	**ALT + 41**
[	**ALT + 91**
]	**ALT + 93**
{	**ALT + 123**
}	**ALT + 125**
.	**ALT + 46**
;	**ALT + 59**

In addition to the symbols we identify combinations of characters by their meaning in the PHP language or by the importance of being identified by our study. The combination of characters are:

//, /*, */, ->, <?PHP, <?php, <?, ?>, php?>, PHP?>

It is valid to point out that we do not identify or store each non-key word in the PHP code, neither identify the different variables that were found in the reading of the files. The non-key words and the variables were only differentiated and classified independently of the keywords of the PHP code. With this difference the Patterns of Behavior was created.

During the reading of the code, conditional **if** and **else** were used for the different necessary stops and recognition of the words in the files. Each word found was classified as:

With the exception of the classifications "String" and "Variable", the rest have significant value for the syntax and structure of the PHP code. Variables are all those words preceded by the $ symbol, and String all those that do not belong to another classification.

Key Word:	meaningful word for the code
Non-Key Word:	negative of the meaningful word for the code
Variable:	words preceded by the symbol $
Symbol:	described in the Table 5.1
String:	any word or sentence not classified

Boolean variables were used to segment the files, and indicate which fragments should be studied and which should not. These variables were necessary due to the diversity of structure with which we can find the files with PHP code.

Example of **PHP** code structure. CVE-2011-4562-log.php's fragment.

```
<?php
class RE_Log {
var $id;
var $created;
...
function RE_Log ($values){
...
}
}
?>
```

Example of **PHP** code structure. CVE-2011-4562-log_item.php's fragment.

```
<?php if (!defined ('ABSPATH')) die ('No di-
rect access allowed'); ?>

<td width="16" class="center item">
<input type="checkbox" class="check"
name="checkall[]" value="<?php echo $log-
>id ?>"/>
</td>
```

With these defined lines our tool is able to detect the Patterns of Behavior elaborated and used for our study; as well as proposing a modification of the code mitigating the possible existence of a Vulnerability of type, XSS.

The code of our Demo is ready to be executed with the Eclipse IDE and is available in: https://github.com/omesa1984/Demo-Plugin. The only parameter that our code will request is the path of the folder where the files you want to analyze are located.

6
Tool Evaluation

In this chapter we analyze the results obtained during the tests carried out with our plugin-demo. We highlight the behavior of some Patterns because of their multiple solutions, and we also show images of the results obtained during the tests.

6.1
Partial Results

We developed several tests on the files of the second group of those chosen from those published in the NVD Repository. With the help of the developed code, a quick but efficient reading of the files was made for the detection of the different Patterns of Behavior, developed and shown previously. The developed code tool allows us to perform the isolated reading of each file with PHP extension belonging to the same application without needing the necessary integrity between them.

An example of a Pattern of Behavior detected in the files, we can see it in the file **CVE-2011-4562-log_item.php**. In the line of code number 19, we find the code expression **echo htmlspecialchars ($log->ip)** that responds the Pattern of Behavior (iii) **echo htmlspecialchars (variable -> String)**. It can be found in Annex B.

The proposed solution to the Possible Vulnerability in the Behavior Pattern (iii) we propose the modification (iii) **echo esc_attr (variable -> String)** which can be found in Annex C. It is valid to point out that the existed example is intentional since, we can not argue about the totality of the appearances but without a high percentage of the occurrences of the keyword **htmlspecialchars** there is the possibility of a vulnerability of some kind. In the same way, in a high percentage of the proposed solutions or modifications is the keyword **esc_attr**. So the proposal consisted in maintaining the syntax and only changing the keyword **htmlspecialchars** by **esc_attr**.

Another relevant result found during the creation, reading and analysis of our Behavior Patterns, we can find it in the patterns (vi), (vii), (ix) and (x). It is not difficult to notice that if we group them correctly they are 2 possible solutions for each possible vulnerability detected. We could argue that it is

enough with the second proposal in each case for each possible vulnerability. But we want to incorporate artificial intelligence for our tool and allow you to learn, and we understand that it would be a good exercise to make the first modification and then the second until he understands that he can omit the first proposal and directly propose the second.

(vi)**variable = $_SERVER [String]**
(vi)**variable = htmlspecialchars ($_SERVER [String])**

(vii)**variable = $_SERVER [String]**
(vii)**variable = esc_html ($_SERVER [String])**

(ix)**echo $_SERVER [String]**
(ix)**echo htmlspecialchars ($_SERVER [String])**

(x)**echo $_SERVER [String]**
(x)**echo esc_html ($_SERVER [String])**

It is easy to appreciate that the Behavior Patterns (vi) and (vii) are the same pattern with different solutions proposals. In the same way it happens with patterns (ix) and (x). They were conceived and interpreted as different patterns, because they were located in different files and to contribute with the learner that we want to incorporate into our tool.

The frequency of occurrence of each Pattern is not very high and it varies from an appearance to four appearances, in the totality of files with which we work. Remembering that in this first study we worked only with the files that contained a single vulnerability and that it was corrected in a single line of code. With the exception of the employers (vi), (vii), (ix) and (x); the rest of them always have the same solution proposal. Preliminary studies have shown us that the patterns elaborated until now, can be repeated in the files not analyzed by being corrected with more than one line of code.

It is true that the proposed changes in the solutions will not always be necessary, since we are working with the concept of Possible Vulnerabilities existing in isolated files. This suggests that no matter how many probabilities there are of the occurrence of vulnerability, there is no guarantee of its occurrence. In the same way our study showed that any security in our applications never hurts, remembering that we are working on the anticipation of vulnerabilities of type XSS, CSRF and SQL Injection; which represent more than 70% of the vulnerabilities produced by the addition of a plugin created

with PHP code.

6.2 Results Images

To conclude the tests of our code, several tests were carried out with several files distributed in several groups. In previous chapters we had commented that the files taken from the NVD repository were divided into two groups. The first group was used for the creation of the Behavior Patterns and the second group of files was used for the functional tests of our code.

When we start the process, our demo requests the path of the folder where the files we want to test are located, as shown in Figure 6.1

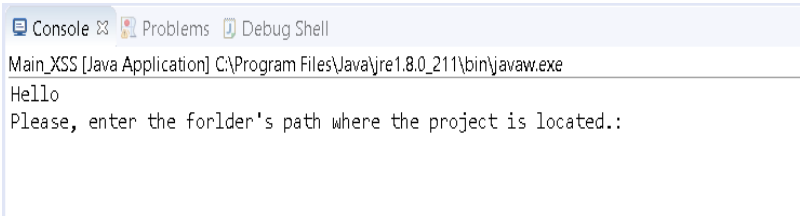

Figure 6.1: Folder Path Request

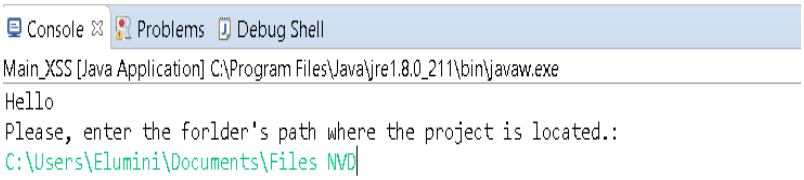

Figure 6.2: Show the folder path inserted in green

The previous figure was captured after inserting the text with the path of the folder where the files to be tested are located.

The files used for the tests were grouped into several groups for the latest tests in our code. The images were taken at different times of the tests showing the different results that were obtained.

The following figures were captured at various times during the performance of the various tests, showing the results at each time. The images show the performance of the tests with several files. At the images is clearly appreciate the name of the files, the behavior pattern of the possible security vulnerability, and the possible proposal for mitigate the vulnerability.

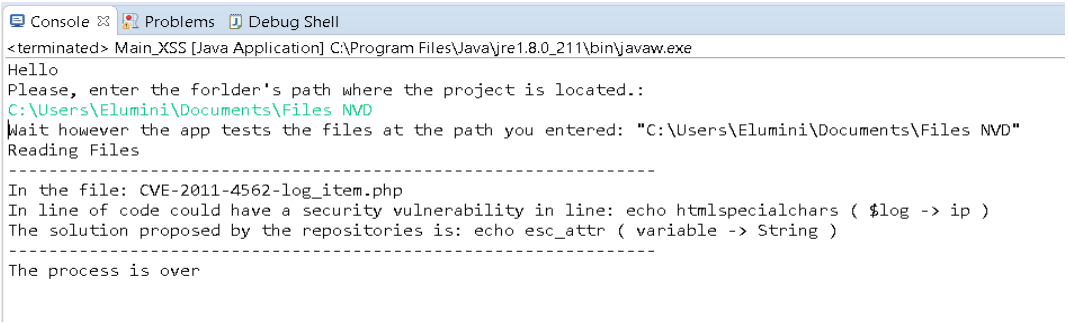

Figure 6.3: Test image performed showing a possible security vulnerability found in a single file

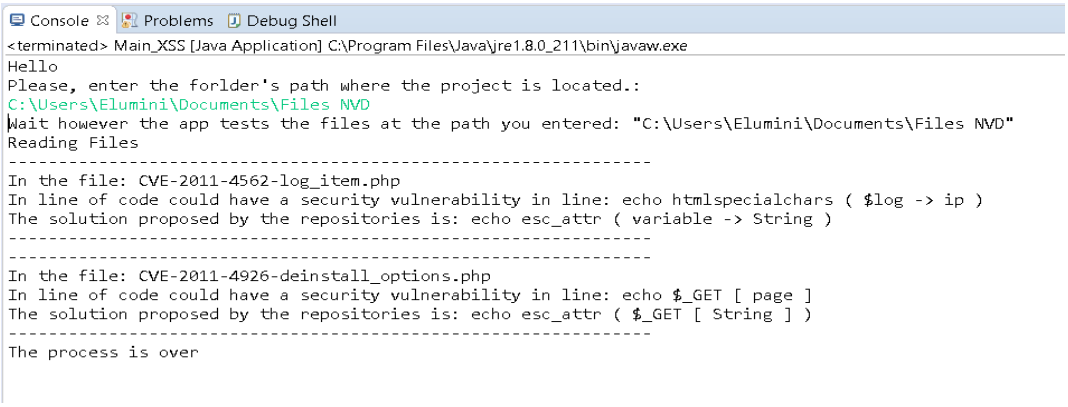

Figure 6.4: Test image performed showing the security vulnerabilities found in with two files

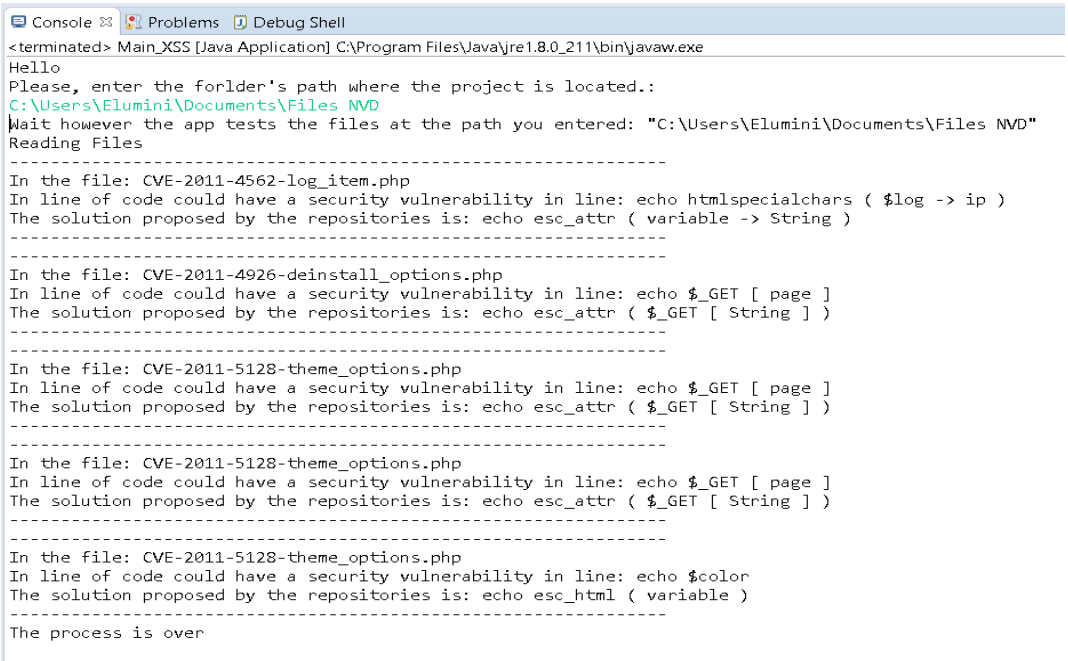

Figure 6.5: Test image performed showing the security vulnerabilities found in with three files.

Figure 6.6: Fragment of test image made with more than forty files, it shows a sample of the possible security vulnerabilities detected in several files.

7
Conclusions

7.1
Advantages and Weakness of Patterns

Among the advantages and weakness of the use of Behavior Patterns, we decided to evaluate the following points:

(Adv) Time to make the vulnerability test: with Behavior Patterns, potential vulnerabilities can be detected at any time during the life cycle of the application, initiating the life cycle during the implementation of the application and ending after the deployment of the application. Without the Behavior Patterns, we are obliged to conclude the applications completely to perform the vulnerability test, being possible to perform the tests only in the final stage of the life cycle of the application.

(Adv) Number of files to be analyzed: with the Behavior Patterns the files can be individually tested in search of Possible Vulnerabilities. If there is any report of the file where the vulnerability is found, you can test that single file and solve with the solution proposals obtained from the vulnerabilities where the studies were carried out.

(Adv) Application test time: As the Behavior Patterns can be used for the detection of vulnerabilities at any time of the life cycle of the applications and the files can be independently tested, it is possible to reduce the test and response time before the vulnerabilities detected.

(Adv) Guarantee of the results: the tool of Plugin works in the detection of Possible Vulnerabilities which does not guarantee that the vulnerability exists in its entirety. But studies have shown that vulnerabilities of the same type have similar behavior, which guarantees the test result.

(Weak) Level of knowledge of the programmer respect to the application or vulnerability detected: if the Behavior Patterns are use in the maintenance of an application, the programmer needs to know the application code; because the patterns detect Possible Vulnerabilities. If you are not working on a reported vulnerability, it may be difficult to detect the vulnerabilities. On the other hand, if you are working on a vulnerability detected and reported, it does not require a high level of knowledge and can

solve the problem with the proposed solution of the Knowledge Base. Ignorance of the application code can cause time losses in the correction of Possible Vulnerabilities, and the tool used to favor the programmer can hinder their work.

(Weak) Guarantee of the existence of the Possible Vulnerability: as the tool of the Plugin detects the existence of Possible Vulnerabilities, we can not guarantee that the existence of the Vulnerability; even when the changes recommended by the tool are not made by the programmer. So we only create a first step in the anticipation of vulnerabilities.

(Weak) Artificial Intelligence and Self-Learning: We understand that the lack of elements of artificial intelligence and self-learning limits the rapid evolution of our tool, so it will be one of the approaches in future work.

7.2 Final Recommendations

The study conducted on WordPress and described in the first chapters demonstrated the need to mitigate the existence of security vulnerabilities. It was shown that resolving or anticipating vulnerabilities of type XSS, SQL Injection or CSRF eliminate more than 70% of vulnerabilities. As a first result from the research we developed the Demo tool to anticipate the most frequent vulnerabilities.

We demonstrate that the use of Behavior Patterns can contribute to detection and anticipation of possible security vulnerabilities. As the location of the possible security vulnerability seems exaggerated or absurd at times, so we recommend deepen the study of Behavior Patterns and combine them with machine learning techniques.

We create a Demo with an easily modifiable and adaptable structure to be combined with the Multiagent Systems methodology. This methodology will facilitate the reading of the files and accelerate the response of future versions.

Our greatest contribution is not in the vulnerability anticipation technique of our Plugin's Demo version, although with the creation and use of Behavior Patterns, we took an important step. Our contribution is in the architecture designed for the use of the Behavior Patterns, and the easy incorporation and adaptability of the methodology of Multiagent Systems and machine learning techniques.

7.3 Future Work

In the next work we will have as a first task, the migration of the our tool to develop with Multiagent Systems methodology (11). Being able to provide greater speed, flexibility and intelligence to the plugin. While we generate the definitive Knowledge Base and the necessary functions so that it is kept updated of the NVD Repository.

We also want to carry out studies of artificial intelligence and machine learning to select the appropriate technique and equip our tool with the ability to learn to create new patterns of behavior. Eliminate existing patterns if they stop appearing for long periods of time, and eliminate intermediate corrections such as those shown at the beginning of this chapter.

Bibliography

[1] AGGARWAL, A.; JALOTE, P.. **Integrating static and dynamic analysis for detecting vulnerabilities**. In: 30TH ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE (COMPSAC'06), volumen 1, p. 343–350. IEEE, 2006.

[2] ALHAZMI, O. H.; MALAIYA, Y. K. ; RAY, I.. **Measuring, analyzing and predicting security vulnerabilities in software systems**. Computers & Security, 26(3):219–228, 2007.

[3] ANTUNES, N. M. D. S.. **Software Vulnerability Detection in Service-Based Infrastructures: Techniques and Tools**. PhD thesis, 2014.

[4] ARBAUGH, W. A.; FITHEN, W. L. ; MCHUGH, J.. **Windows of vulnerability: A case study analysis**. Computer, 33(12):52–59, 2000.

[5] ARORA, A.; KRISHNAN, R.; TELANG, R. ; YANG, Y.. **An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure**. Information Systems Research, 21(1):115–132, 2010.

[6] ASADUZZAMAN, M.; ROY, C. K.; SCHNEIDER, K. A. ; HOU, D.. **Cscc: Simple, efficient, context sensitive code completion**. In: 2014 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION, p. 71–80. IEEE, 2014.

[7] BASILI, V. R.; SHULL, F. ; LANUBILE, F.. **Building knowledge through families of experiments**. IEEE Transactions on Software Engineering, 25(4):456–473, 1999.

[8] BLEI, D. M.; NG, A. Y. ; JORDAN, M. I.. **Latent dirichlet allocation**. Journal of machine Learning research, 3(Jan):993–1022, 2003.

[9] BOSU, A.; CARVER, J. C.; HAFIZ, M.; HILLEY, P. ; JANNI, D.. **Identifying the characteristics of vulnerable code changes: An empirical study**. In: PROCEEDINGS OF THE 22ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 257–268. ACM, 2014.

[10] BOZORGI, M.; SAUL, L. K.; SAVAGE, S. ; VOELKER, G. M.. **Beyond heuristics: learning to classify vulnerabilities and predict exploits**. In: PROCEEDINGS OF THE 16TH ACM SIGKDD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY AND DATA MINING, p. 105–114. ACM, 2010.

[11] GALÁN, E.; ALCAIDE, A.; ORFILA, A. ; BLASCO, J.. **A multi-agent scanner to detect stored-xss vulnerabilities**. In: 2010 INTERNATIONAL CONFERENCE FOR INTERNET TECHNOLOGY AND SECURED TRANSACTIONS, p. 1–6. IEEE, 2010.

[12] BRUCH, M.; MONPERRUS, M. ; MEZINI, M.. **Learning from examples to improve code completion systems**. In: PROCEEDINGS OF THE THE 7TH JOINT MEETING OF THE EUROPEAN SOFTWARE ENGINEERING CONFERENCE AND THE ACM SIGSOFT SYMPOSIUM ON THE FOUNDATIONS OF SOFTWARE ENGINEERING, p. 213–222. ACM, 2009.

[13] CACHIN, C.; CAMENISCH, J.; DESWARTE, Y.; DOBSON, J.; HORNE, D.; KURSAWE, K.; LAPRIE, J.-C.; LEBRAUD, J.-C.; LONG, D.; MCCUTCHEON, T. ; OTHERS. **Maftia: Reference model and use cases**. 2000.

[14] DABBISH, L.; STUART, C.; TSAY, J. ; HERBSLEB, J.. **Social coding in github: transparency and collaboration in an open software repository**. In: PROCEEDINGS OF THE ACM 2012 CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK, p. 1277–1286. ACM, 2012.

[15] FERREIRA, G.; MALIK, M.; KÄSTNER, C.; PFEFFER, J. ; APEL, S.. **Do# ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel**. arXiv preprint arXiv:1605.07032, 2016.

[16] DA FONSECA, J. C. C. M.; VIEIRA, M. P. A.. **A practical experience on the impact of plugins in web security**. In: 2014 IEEE 33RD INTERNATIONAL SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, p. 21–30. IEEE, 2014.

[17] BARUA, A.; THOMAS, S. W. ; HASSAN, A. E.. **What are developers talking about? an analysis of topics and trends in stack overflow**. Empirical Software Engineering, 19(3):619–654, 2014.

[18] GREILER, M.; DEURSEN, A. V. ; STOREY, M.-A.. **Test confessions: A study of testing practices for plug-in systems**. In: PROCEEDINGS OF THE 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 244–254. IEEE Press, 2012.

[19] JERKOVIC, H.; SINKOVIC, B.. **Vulnerability analysis of most popular open source content management systems with focus on wordpress and proposed integration of artificial intelligence cyber security features.** International Journal of Economics and Management Systems, 2, 2017.

[20] MCGRAW, G.. **Software security: building security in**, volumen 1. Addison-Wesley Professional, 2006.

[21] MELL, P.; SCARFONE, K. ; ROMANOSKY, S.. **Common vulnerability scoring system**. IEEE Security & Privacy, 4(6):85–89, 2006.

[22] MYERS, G. J.; BADGETT, T.; THOMAS, T. M. ; SANDLER, C.. **The art of software testing**, volumen 2. Wiley Online Library, 2004.

[23] NGUYEN, H. V.; KÄSTNER, C. ; NGUYEN, T. N.. **Exploring variability-aware execution for testing plugin-based web applications**. In: PROCEEDINGS OF THE 36TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 907–918. ACM, 2014.

[24] NHLABATSI, A.; LANEY, R. ; NUSEIBEH, B.. **Feature interaction: The security threat from within software systems**. Progress in Informatics, 5:75–89, 2008.

[25] SAMPAIO, L.; GARCIA, A.. **Exploring context-sensitive data flow analysis for early vulnerability detection**. Journal of Systems and Software, 113:337–361, 2016.

[26] SHAHZAD, M.; SHAFIQ, M. Z. ; LIU, A. X.. **A large scale exploratory analysis of software vulnerability life cycles**. In: 2012 34TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 771–781. IEEE, 2012.

[27] SMITH, B.; WILLIAMS, L.. **Using sql hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities**. In: 2011 FOURTH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE TESTING, VERIFICATION AND VALIDATION, p. 220–229. IEEE, 2011.

[28] WALDEN, J.; DOYLE, M.; LENHOF, R.; MURRAY, J. ; PLUNKETT, A.. **Impact of plugins on the security of web applications**. In: PROCEEDINGS OF THE 6TH INTERNATIONAL WORKSHOP ON SECURITY MEASUREMENTS AND METRICS, p. 1. ACM, 2010.

[29] WALDEN, J.; DOYLE, M.; WELCH, G. A. ; WHELAN, M.. **Security of open source web applications**. In: 2009 3RD INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT, p. 545–553. IEEE, 2009.

[30] LOUCOPOULOS, P. AND KARAKOSTAS, V.. **System requirements engineering**. McGraw-Hill, Inc., 1995.

[31] VAN LAMSWEERDE, A.. **Requirements engineering: From system goals to UML models to software**. Chichester, UK: John Wiley & Sons, 10th edition, 2009.

[32] BAKKEN, S. S., ALEXANDER AULBACH, A., SCHMID, E., WINSTEAD, J., WILSON, L. T., LERDORF, R., ZMIEVSKI, A., AND AHTO, J.. **PHP Manual**. 7th edition, 2003.

[33] BASILI, V. R.; ROMBACH, H. D.. **The tame project: Towards improvement-oriented software environments**. IEEE Transactions on software engineering, 14(6):758–773, 1988.

[34] GENBETA. **https://www.genbeta.com/desarrollo/eclipse-ide, January, 10, 2014**, 2018.

[35] EDUCACIONIT. **https://blog.educacionit.com/2014/01/16/eclipse-ide-principales-carateristicas/, January, 16, 2014**, 2005–2019.

[36] CANFORA, G.; CECCARELLI, M.; CERULO, L. ; DI PENTA, M.. **How long does a bug survive? an empirical study**. In: 2011 18TH WORKING CONFERENCE ON REVERSE ENGINEERING, p. 191–200. IEEE, 2011.

[37] DI PENTA, M.; CERULO, L. ; AVERSANO, L.. **The life and death of statically detected vulnerabilities: An empirical study**. Information and Software Technology, 51(10):1469–1484, 2009.

[38] GARZARELLI, G.. **Open source software and the economics of organization**. In: MARKETS, INFORMATION AND COMMUNICATION, p. 63–78. Routledge, 2013.

[39] GRIFFITHS, T. L.; STEYVERS, M.. **Finding scientific topics**. Proceedings of the National academy of Sciences, 101(suppl 1):5228–5235, 2004.

[40] JIMENEZ, M.; PAPADAKIS, M.; BISSYANDÉ, T. F. ; KLEIN, J.. **Profiling android vulnerabilities**. In: 2016 IEEE INTERNATIONAL CONFERENCE ON SOFTWARE QUALITY, RELIABILITY AND SECURITY (QRS), p. 222–229. IEEE, 2016.

[41] KIM, S.; ZIMMERMANN, T.; PAN, K.; JAMES JR, E. ; OTHERS. **Automatic identification of bug-introducing changes**. In: 21ST IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING (ASE'06), p. 81–90. IEEE, 2006.

[42] KOSKINEN, T.; IHANTOLA, P. ; KARAVIRTA, V.. **Quality of wordpress plug-ins: an overview of security and user ratings**. In: 2012 INTERNATIONAL CONFERENCE ON PRIVACY, SECURITY, RISK AND TRUST AND 2012 INTERNATIONAL CONFERNECE ON SOCIAL COMPUTING, p. 834–837. IEEE, 2012.

[43] MARTIN, R. A.; CHRISTEY, S. M. ; JARZOMBEK, J.. **The case for common flaw enumeration**. In: PROCEEDINGS OF WORKSHOP ON SOFTWARE SECURITY ASSURANCE TOOLS, TECHNIQUES, AND METRICS, número 500-265, 2005.

[44] MEYER, M. H.; SELIGER, R.. **Product platforms in software development**. MIT Sloan Management Review, 40(1):61, 1998.

[45] MUNSON, J. C.; ELBAUM, S. G.. **Code churn: A measure for estimating the impact of code change**. In: PROCEEDINGS. INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (CAT. NO. 98CB36272), p. 24–31. IEEE, 1998.

[46] NAGAPPAN, N.; BALL, T.. **Use of relative code churn measures to predict system defect density**. In: PROCEEDINGS OF THE 27TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, p. 284–292. ACM, 2005.

[47] ELECTRONICOSONLINE.COM S.A. DE C.V.. **https://www.electronicosonline.com/incrementan-eclipse-y-netbeans-popularidad-como-ides/**, 2018.

[48] PHAM, R.. **Improving the software testing skills of novices during onboarding through social transparency**. In: PROCEEDINGS OF THE 22ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 803–806. ACM, 2014.

[49] PHAM, R.; KIESLING, S.; LISKIN, O.; SINGER, L. ; SCHNEIDER, K.. **Enablers, inhibitors, and perceptions of testing in novice software teams**. In: PROCEEDINGS OF THE 22ND ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 30–40. ACM, 2014.

[50] SHIN, Y.; MENEELY, A.; WILLIAMS, L. ; OSBORNE, J. A.. **Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities**. IEEE transactions on software engineering, 37(6):772–787, 2010.

[51] MESA, O.; VIEIRA, R.; VIANA, M.; DURELLI, V. H.; CIRILO, E.; KALINOWSKI, M. ; LUCENA, C.. **Understanding vulnerabilities in plugin-based web systems: an exploratory study of wordpress**. In: PROCEEEDINGS OF THE 22ND INTERNATIONAL CONFERENCE ON SYSTEMS AND SOFTWARE PRODUCT LINE-VOLUME 1, p. 149–159. ACM, 2018.

[52] MESA, O.; VIANA, M.; CIRILO, E. ; LUCENA, C.. **Vulnerabilidades em sistemas de software web baseados em plug-ins? um estudo exploratório do wordpress**.

[53] TELANG, R.; WATTAL, S.. **An empirical analysis of the impact of software vulnerability announcements on firm stock price**. IEEE Transactions on Software Engineering, 33(8):544–557, 2007.

[54] WILLETT, P.. **The porter stemming algorithm: then and now**. Program, 40(3):219–223, 2006.

[55] XIAO, S.; WITSCHEY, J. ; MURPHY-HILL, E.. **Social influences on secure development tool adoption: why security tools spread**. In: PROCEEDINGS OF THE 17TH ACM CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK & SOCIAL COMPUTING, p. 1095–1106. ACM, 2014.

A
Articles Published at Conferences

The work on the need to anticipate Vulnerabilities generated by the addition of a plugin to a CMS, using WordPress as a research base, resulted in two publications. The first at the CBSOFT conference in 2017. The second at the 22nd International Conference on Systems and Software Product Line in 2018. Both are proof of the need to deepen this research in our area. It is for this reason that both publications will be appended to the following pages.

B
Behavior Patterns

This annex shows the behavior patterns created from the deficiencies published by the NVD repository.

(i) **echo htmlspecialchars (variable)**

(ii) **echo htmlspecialchars (String (variable -> String))**

(iii) **echo htmlspecialchars (variable -> String)**

(iv) **echo $this -> variable**

(v) **echo variable.variable1 -> String**

(vi) **variable = $_SERVER [String]**

(vii) **variable = $_SERVER [String]**

(viii) **echo variable**

(ix) **echo $_SERVER [String]**

(x) **echo $_SERVER [String]**

(xi) **echo gethostbyaddr $_SERVER [String]**

(xii) **echo $_SERVER ['REQUEST_URI']**

(xiii) **echo $_GET [String]**

(xiv) **http://. $_SERVER [String] . $_SERVER ['REQUEST_URI']**

(xv) **variable = $_GET [String]**

(xvi) **variable = (int) $_GET [String]**

(xvii) **variable = $_POST [String]**

(xviii) **variable = $_GET [String]**

(xix) **$this -> delete_post_type ($_GET [String])**

(xx) **$this -> delete_taxonomy ($_GET [String])**

C
Solutions to Behavior Patterns

The annex describes the solutions proposals in the event of the possible existence of any of the vulnerabilities worked on.

(i) **echo esc_attr (variable)**

(ii) **echo esc_attr (String (variable -> String))**

(iii) **echo esc_attr (variable -> String)**

(iv) **echo esc_attr ($this -> variable)**

(v) **echo variable.esc_attr (variable1 -> String)**

(vi) **variable = htmlspecialchars ($_SERVER [String])**

(vii) **variable = esc_html ($_SERVER [String])**

(viii) **echo esc_html (variable)**

(ix) **echo htmlspecialchars ($_SERVER [String])**

(x) **echo esc_html ($_SERVER [String])**

(xi) **echo esc_html gethostbyaddr ($_SERVER [String])**

(xii) **echo esc_url ($_SERVER ['REQUEST_URI'])**

(xiii) **echo esc_attr ($_GET [String])**

(xiv) **http://. $_SERVER [String] . urlencode ($_SERVER ['REQUEST_URI'])**

(xv) **variable = (int) $_GET [String]**

(xvi) **variable = (int) esc_attr $_GET [String]**

(xvii) **variable = sanitize_text_field ($_POST [String])**

(xviii) **variable = esc_html ($_GET [String])**

(xix) **$this -> delete_post_type ($String)**

(xx) **$this -> delete_taxonomy ($String)**

D
File CVE-2011-4562-log.php

Expanded file fragment File **CVE-2011-4562-log.php** used to represent one of the structures of a file with php code.

```
<?php

    class RE_Log {
var $id;
var $created;
var $url;
...

function RE_Log ($values) {
foreach ($values AS $key => $value)
$this->$key = $value;

    $this->created = mysqldate ('U', $this->created);
$this->url = stripslashes ($this->url);
}

function get_by_id ($id) {
global $wpdb;

    $row = $wpdb->get_row ("SELECT * FROM {$wpdb-
>prefix}redirection_logs WHERE id='$id'", ARRAY_A);
if ($row)
return new RE_Log ($row);
return false;
}

function get( &$pager ) {
global $wpdb;
```

```
        $rows = $wpdb->get_results( "SELECT SQL_CALC_FOUND_ROWS
* FROM $wpdb->prefixredirection_logs FORCE INDEX(created) ".$pager-
>to_limits ('redirection_id IS NOT NULL', array ('url', 'sent_to',
'ip')), ARRAY_A ); $pager->set_total ($wpdb->get_var ("SELECT
FOUND_ROWS()"));

        $items = array ();
if (count ($rows) > 0) {
foreach ($rows AS $row)
$items[] = new RE_Log ($row);
}

return $items;
}

...

function referrer () {
return preg_replace ('@https?://(.*?)/.*@', '$1', $this->referrer);
$home = get_bloginfo ('url');
if (substr ($this->referrer, 0, strlen ($home)) == $home)
return substr ($this->referrer, strlen ($home));
return $this->referrer;
}
}

        ?>
```

E
File CVE-2011-4562-log_item.php

File **CVE-2011-4562-log_item.php** used to represent one of the structures of a file with php code.

```
<?php if (!defined ('ABSPATH')) die ('No direct access allowed'); ?>
<td width="16" class="center item">
<input type="checkbox" class="check" name="checkall[]" value="<?php echo
$log->id ?>"/>
</td>
<td style="width:9em">
<a href="<?php echo admin_url( 'admin-ajax.php' ); ?>?ac-
tion=red_log_show&amp;amp;id=<?php echo $log->id ?>&amp;amp;_ajax_nonce=<?php
echo wp_create_nonce( 'redirection-log_'.$log->id )?>" class="show-log">
<?php echo date (str_replace ('F', 'M', get_option ('date_format')), $log-
>created) ?>
</a>
</td>
<td class="info">
<a class="details" href="<?php echo $log->url ?>"><?php echo $log-
>show_url( $log->url ) ?></a>
</td>
<td>
<?php if (strlen ($log->referrer) > 0) : ?>
<a href="<?php echo $this->url ( $log->referrer) ) ?>"><?php echo $log-
>show_url( $log->referrer() ) ?></a>
<?php endif; ?>
</td>
<td style="width:9em" class="center">
<a target="_blank" href="<?php echo $lookup.$log->ip ?>"><?php echo
htmlspecialchars( $log->ip ) ?></a>
</td>
<td style="width: 16px" class="lastcol">
<a href="#add" class="add-log"><img src="<?php echo $this->url ()
```

?>/images/add.png" width="16" height="16" alt="Add"/>
</td>