International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

A New Source Code Auditing Algorithm for Detecting LFI and RFI in PHP Programs

Seyed Ali Mir Heydari, and Mohsen Sayadiharikandeh

***Abstract*—**Static analysis of source code is used for auditing web applications to detect the vulnerabilities. In this paper, we propose a new algorithm to analyze the PHP source code for detecting LFI and RFI potential vulnerabilities. In our approach, we first define some patterns for finding some functions which have potential to be abused because of unhandled user inputs. More precisely, we use regular expression as a fast and simple method to define some patterns for detection of vulnerabilities. As inclusion functions could be also used in a safe way, there could occur many false positives (FP). The first cause of these FP's could be that the function does not use a user-supplied variable as an argument. So, we extract a list of user-supplied variables to be used for detecting vulnerable lines of code. On the other side, as vulnerability could spread among the variables like by multi-level assignment, we also try to extract the hidden user-supplied variables. We use the resulted list to decrease the false positives of our method. Finally, as there exist some ways to prevent the vulnerability of inclusion functions, we define also some patterns to detect them and decrease our false positives.

***Keywords*—**User-supplied Variables, hidden user-supplied variables, PHP vulnerabilities.

I. Introduction

THE World-Wide Web started in the mid 90's as a system to support hypertextual access to static information. Web applications are designed to present to any user with a web browser a system-independent interface to some dynamically generated content. The number and the importance of Web applications have increased rapidly in last decade. At the same time of growing web applications, the quantity and impact of security vulnerabilities in such applications have grown as well.

The application may be designed with the assumption that users will only enter valid input as the programmer intended, in terms of both input values and ways of entering input. However, if the user's input is not handled properly, serious security problems can occur. This has been made possible by the introduction of a number of mechanisms that can be used to trigger the execution of code on both the client and the server side. These mechanisms are the basis to implement web-based applications. And that's why Code reviews and security audits are part of the quality assurance phase. So, security coding must be always considered as an important skill in programming.

The existing approaches for decreasing threats to Web applications can be divided into client-side and server-side solutions. Server-side solutions have the advantage of being able to discover a larger range of vulnerabilities and the benefit of a security flaw fixed by the service provider is instantly propagated by service provider to its all clients. But on the other side, it usually makes some limitation for applications and implicitly developers which is supposed as one of the major disadvantages of this approach.

As mentioned before, another approach is client-side. Client-side techniques can be further classified into dynamic and static approaches. Dynamic tools (e.g., [1, 2, 3], and Perl's taint mode try to detect attacks while executing the audited program, whereas static analyzers ([4, 5, 6, 7]) scan the Web application's source codes for vulnerabilities. From the static point of view, applications could be statically analyzed where it can protect applications before actually running them, so the problem could be eliminated before deploying the code into a sensitive environment. There were only a few great works done in static code analysis because it is time-consuming and complex, in some cases.

In this paper, we present a novel method for detecting LFI and RFI vulnerabilities in PHP source codes. We chose PHP as it is used by most of web developers.

Although, due to the complexity of PHP code, allowing widely used dynamic code generation and multiple levels of indirection in variable and function access, static analysis is unable to achieve comprehensive coverage of the application functionality, the results in many of the same projects show that many existing problems could be eliminated by this approach.

Among the most common of vulnerabilities are LFI and RFI. So we focused on LFI and RFI in this paper. We will discuss these vulnerabilities in depth in next sections.

This paper is organized as follows. In section III, we review the related work in the area of static analysis particularly in RFI and LFI detection on source code in web applications. Section IV briefly describes the regular expression notations in computer science and specially their use in programming languages. In section V we propose our method to improve the security of web applications in client side. Next, we present the results of the experimental evaluation of our algorithm. Finally, in last section there are some notes about

S. A. M. Heydari is with the Islamic Azad University, Taft Branch, Taft, Yazd, Iran (corresponding author to provide phone: 00989133528600; e-mail: Mirheidari@taftiau.ac.ir).

M. Sayadiharikandeh is with Department of Computer Engineering, Sharif University of Technology, Tehran, Iran (e-mail: m_sayyadi@ce.sharif.edu).

the future work.

II. Related Works

There already exist several techniques for source code auditing like FIS [8] which scan the Web application's source code for vulnerabilities. This tool first scan the source code to find all the variables and then test each of them dynamically to find out whether they are source of vulnerability or not. However, the complexity of this technique is high and there are a lot of cases, that the mentioned tool report false vulnerability like below. The following code is a part of microSSys application source code which has a registered vulnerability CVE- 2008-2396.

```
Vulnerable code (index.php@22-25,54-55):
[22] if(isset($_REQUEST["1"])){
[23] $P=$_REQUEST["1"];}else{
[24] $P="main";
[25] }
[..]
[54] if(isset($PAGES[$P])){}else{include("TH.txt");}
[55] @include($PAGES[$P]);
```

FIS disadvantage is that it does not have ability to find out the exploits like below:

```
http://host/index.php?1=lol&PAGES[lol]={Remote shell script }
```

Analogous to the above Noxes [9] is an application-level firewall offering protection in case of suspected cross-site scripting (XSS) attacks that attempt to steal a user's credentials. The mentioned tools pay no attention to LFI and RFI as one of the most dangerous vulnerabilities.

Static approach has been also explored in WebSSARI [10] and by Minamide [11]. WebSSARI has been used to find a number of security vulnerabilities in PHP scripts, but has a large number of false positives and negatives due to its intraprocedural type-based analysis. Minamide's system checks syntactic correctness of HTML output from PHP scripts and does not seem to be effective for finding security vulnerabilities [12].

[13] proposed pixy as the first open source tool for statically detecting taint-style vulnerabilities (in particular, XSS and SQL injection vulnerabilities) in PHP 4 code. Pixy features a high-precision data flow analysis engine that is flow-sensitive, interprocedural, and context-sensitive and performs alias analysis, literal analysis, and taint analysis. Pixy focused on PHP as a popular language but it could be applied only for detecting XSS and SQL injection vulnerabilities.

[14, 15] use the static source code analysis concept with using regular expressions in an interesting way. The major imperfection of the two mentioned approaches is that they rely on only some methods which could have potential vulnerability and they do not pay attention to preventions methods and hidden user-supplied variables. So, in cases which developers were aware of the vulnerabilities and have handled them skillfully, the reports of the mentioned tools have large false positives. Even though the last two mentioned tools are both using pattern matching to detect the PHP weaknesses but the second tool has less false positives because it uses regular expression more precisely than the first one. First tool assume the user-defined functions which have the same patterns of regular expressions like my_inlcude() as a vulnerable function but another one just report the main known vulnerable functions. We extend the idea to use the patterns for prevention methods to enhance the level of accuracy.

III. PHP Vulnerabilities

PHP is a computer scripting language. Originally designed for producing dynamic web pages, it has evolved to include a command line interface capability and can be used in standalone graphical applications. PHP is a widely used general-purpose scripting language that is especially suited for web development. According to NetCraft [16] there has been huge progress in using PHP in web applications. Fig. 1 is the diagram which indicates our point.

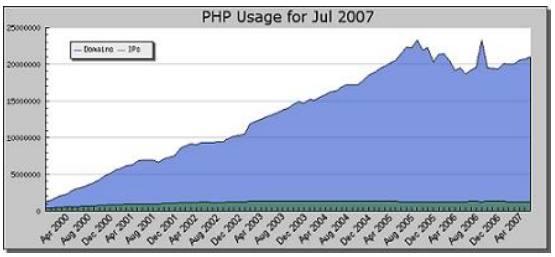

Fig. 1 PHP Usage in July 2007

Several vulnerabilities and a weaknesses have been reported in PHP, where some have unknown impacts and others can be exploited by malicious people to disclose potentially sensitive information, bypass certain security restrictions. Among the most common of them are RFI, LFI, XSS, SQL Injection and RCE. According securityfocus [17] 60% of released exploits are on web applications. We obtained some static from Milw0rm [18] which indicates 27% of released exploits are caused by LFI and RFI vulnerability. Fig. 2 shows a statistical analysis of released exploits from October 2007 to February 2008 years. Our focus is on RFI and LFI because they have wide spread uses in exploits and they are more regular than others and we expect to be simpler for defining patterns of vulnerabilities.

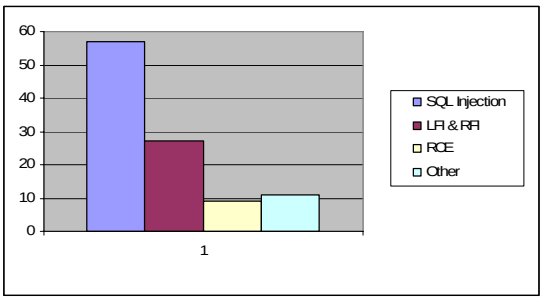

Fig. 2 Percent of Different vulnerabilities' Exploits released during 2007 and 2008

A. Local File Inclusion

This vulnerability is one the most dangerous and most common ones which make it possible to access the local files. The mentioned files could not be accessed and displayed by users through regular access rules. Even it is possible to get permission to CMD.exe and execute desired commands. Followings is a list of some methods that could be misused by giving unhandled arguments:

- Require()
- Require-once()
- Include_once()
- Include()
- Fopen()
- File_get_contents()
- …

For instance, in many situations it would be great to use dynamic includes, where the part of the pathname is stored in a variable. As an illustration example, take the following example:

```
[...]
include("/home/lang/".$language.".php");
[...]
```

The $language is not declared before being used. So, an attacker can put tainted data in this variable and include some other files like /etc/passwd.

For example, a user can easily view another file by modifying the value of the language in the URL. For example:

http://remote_host/bugged.php?language=../../../../etc/passwd%00

The result of this inclusion will be:

```
[...]
include "/users/../../../../etc/passwd%00.php"
[...]
```

So a malicious user can see all contents on passwd file in the server. '%00' means a NULL character that "deletes" the PHP extension. If we omit this NULL Byte, we will be able to display only PHP files because the extension included is PHP.

B. Remote File Inclusion

This is also one of the most dangerous vulnerabilities which is knows as RFI which is also more common than the other ones. Using this security flaw, attackers could get the whole control of one server or one site, upload files, edit or delete files or execute some commands. This vulnerability is also the result of using unhanding arguments. The methods like require, require_once and etc (stated before).

Let's take a look at some code that make the RFI exploits possible.

```
[...]
include($_GET['language']);
[...]
```

As we can see, $page is not validated before being used so a malicious user could include or call (as you prefer to say) his script via the browser and gain access to the machine or view, as before, a file. Example one: (gain access to the machine)

http://remote_host/bugged.php?language=[shell Script - our shell located on our server]

Example two: (view files)

http://remote_host/bugged.php?language=/etc/passwd

C. Prevention Methods

In general, the best way to avoid script injection vulnerabilities is to not pass user-supplied input, or data derived from it, into any dynamic execution or 'Include' functions. If this is considered to be unavoidable for some reason, then the relevant input should be strictly validated to prevent any attack occurring. There are some known methods for preventing attacks using LFI and RFI vulnerabilities. This section will hopefully give you some ideas on how to prevent a file inclusion exploit on your website and most importantly, in your code. Also we will be providing the code examples in PHP format.

One way is to use a white list of known good values (such as a list of all the languages or locations supported by the application), and reject any input that does not appear on this list. After one file has been requested to be included and displayed, the source code checks if the given file (as an argument) is an element of the mentioned array or not. If the result is true, it would be permitted to access to this file.

```
[...]
$lang = array("en", "sp", "fa", "it");
If (!in_array($lang, $language){

   Die('invalid page');
}
```

```
include("/home/lang/".$llang.".php");
[...]
```

Another thing you could do is check that the requested file matches a particular format:

```
$file = str_replace('\\', '/', realpath($page . '.php'));
if (!preg_match('%^/home/someone/public_html/[a-z]+\.php$%',
$file)) {
die('Invalid page');
}
include $file;
```

Basically you need to verify that the entered information is valid and conforms to what you expected.

For other PHP vulnerabilities, there are also many functions that can clean the string Like htmlspecialchars(), htmlentities(), stripslashes() and more which could be used to prevent attacks.

IV. Regular Expressions

A regular expression is a special text string for describing a search pattern. It provides a mechanism to select specific strings from a set of character strings and retrieve the aligning part. Pattern matching is used to test whether some parts of context have a desired structure or not. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as '*.txt' to find all text files in a file manager.

Regular expressions are widely used in Perl, PHP, Java, .Net languages or a multitude of other languages. As a programmer you can save yourself lots of time and effort. You can often accomplish with a single regular expression in one or a few lines of code what would otherwise take dozens or hundreds.

Since efficiency is extremely important when executing an application, patterns should be minimized into their most basic form.

V. Proposed Algorithm

A. Figures and Tables

Our method is a static source code analysis which its focus is on LFI and RFI vulnerabilities and their prevention methods. We assume that the source code has been written in a standard style which is perfect for our package specially in the reporting the malicious codes. As an example there is not any multi-line part in source code. At first step we try to distinguish the comments and not considering them. All styles of PHP comments will be ignored and separated from the main source code while parsing it. It is obviously necessary because maybe comments match with one of the patterns and direct the application to the wrong way as we see in some scripts written in PHP and Perl found on the net. Next step is to extract the variables which could have been supplied by users' input. We also find other variables which are derived by one of the found variables at last step. After that, we try to find the aligning parts of code based on the prepared regular expressions. Aligning lines should have one of the variables which have been extracted before. Finally, we try to find out if the prevention methods were used by developer to prevent exploits or not. The output of running this package is a report of vulnerabilities found on given source code and some preventions methods which were used accurately.

In the remaining of this section, we explain each step in detail. Another supported feature is supporting included files' analyzing. As you know, you could include a file in PHP code and use its contents. When the package reaches the inclusion expression (it is done by regular expression mechanism) while processing, the process stops at the current line and the included file will be analyzed. After it has been finished, processing rest of the origin file starts.

In the remaining of this section we explain each step in details.

B. Finding User-supplied Variables

In this step, we try to find special variables which get their values from users' input. We call these variables user-supplied. On the other hand, if the mentioned variables have been assigned to some others, they could also be misused for performing unauthorized actions (Vulnerability spread). We call these variables hidden user-supplied variables. Generally user-supplied variables are found by special regular expressions, but to find hidden user-supplied variables state machine for finding assignments. We used state machine as they are used in compliers and we had not have novel idea about that.

As an illustration, see the below example:

```
[...]
$file = $_REQUEST['file'];
include($file); // <-- vulnerable code, does not sanitise user
parameter

$file2 = $_REQUEST['file3'];
include($file2); // <-- vulnerable code, does not sanitise
user parameter

 echo "</pre>\n";
[...]
```

In this example, 'file2' variable has a potential to be abused, so it should be also included in the list of user-supplied variables. These variables play vital rule in the report of vulnerabilities in next sections as they decrease the false positives a lot.

C. Detecting Vulnerable Patterns

We tried to grasp the patterns that appear within the vulnerable PHP code. Since efficiency is extremely important when executing an application, we tried to minimize the pattern into their most basic form. A database of vulnerable patterns has been prepared in order to test each of its elements

for finding the aligning parts of PHP code. Here's an example of a pattern for vulnerable code shown in section III.B.

Corresponding Pattern:

((?:include|require)(?:_once)?\s*\(?.*?\$.*?\)?;)

D. Detecting Prevention Patterns

The main contribution of our paper is in this part. We extend the idea of analyzing source code to find the malicious code to find also the prevention methods.

We tried to define patterns for known prevention methods up to now. Like the last part, package processes the code to find the matching parts of PHP code by one of the elements of prevention methods array list. After being found, it checks if the found part was in the report of last step or not. If the answer is true, the mentioned lines will be omitted, but it will be involved in the last report of our package which includes the specification of vulnerable codes and its type of prevention method. As an illustration, see the following example which is a pattern for second part of the prevention method stated previously in section III.C.

If(?: *|\t*)*\((?: *|\t*)\!in_array(?: *|\t*)\((?: *|\t*)\$.*,(?: *|\t*)\$.*(?: *|\t*)\)
{\n*(?: *|\t*)Die(?: *|\t*)\((?: *|\t*)'.*'(?: *|\t*));\n*(?: *|\t*)}

As you could find out, we need to use some techniques more than regular expression in detecting prevention methods like some state graph. The mentioned regular expression is the second part of prevention method shown in section III.C. Before that we should make sure that the first part exists in source code. These kinds of operations are done by using state graph.

VI. Experimental Results

In this section, we summarize the experiments we performed and describe the security violations we found.

To evaluate our method, we developed a PHP application based on our algorithm and it was run on four randomly selected open source PHP programs from 100 application on which LFI and RFI have been reported, and TIKIWIKI (which is one of popular and famous PHP application), to demonstrate the feasibility of the proposed algorithm. After that, the results were compared with the output of DAPHPScan method and put in table which you could see it as Table I.

In the course of our experimental validation, we discovered and reported 4 previously unknown vulnerabilities and detected all registered vulnerabilities (Detection Rate is 100%). Detection Rate is always high and perfect. It means all the registered RFI or LFI vulnerabilities would be found by applying our algorithm. Most of the other algorithms and tools just find the use of include functions (or other vulnerable functions) but as stated previously, our algorithm find them when they have some conditions. The value of this field shows that none of our filtering factors were useless and it would not decrease the detection rate. Correct report field shows how many number of our reported vulnerability were right and correct.

After running the algorithm on about 5 programs, we discovered that Number of RFI and LFI potential is about six times less than another algorithm's. We tried to found the affects of three distinct factors on that, as follows.

- Defining list of user-supplied variables decreased the report and implicitly the false positives up to 36%.
- Defining and using prevention patterns decreased the report about 14%.
- Defining strong regular expressions (which ignores the user defined functions like my_include()) decreased the report about 12%.

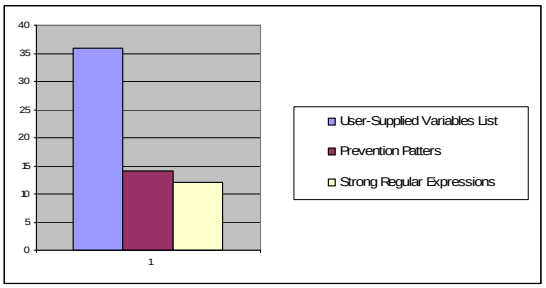

Fig. 3 Effect of Different factors in decreasing FP's

But on the other side, there could exist some non-standard prevention methods in PHP language which affects the number of FP's (make it non-zero). There are also some other reasons for this problem as follows. Consider a variable which has been taken from user input (as a part of URL) and also it was given as an input to one of the vulnerable methods like 'Include' but actually it has not potation to be used in attacks because it was assigned through non-vulnerable manner and then given to the include function as an input.

VII. Conclusion and Future Works

In this paper we proposed a new static analysis algorithm which is able to detect the LFI and RFI vulnerabilities in PHP source codes with higher precision and lower false positives. We implemented our concepts in an application to assure the level of accuracy and compare the results with the other known algorithms.

Future works include the extension of this algorithm for other common vulnerabilities like SQL injection, XSS and RCE.

TABLE I
RESULTS OF APPLYING OUR ALGORITHM TO FIVE PHP PROGRAMS

Application Name&Version	Number of Files & Folders	Number of registered Vulnerabilities	Another algorithm				Our algorithm			
			Number of RFI &LFI Potential	Correct Report	FP	Detection Rate	Number of RFI & LFI Potential	Correct Report	FP	Detection Rate
Tikiwiki 1.9.8	1408 & 329	2	94	2	98%	100%	17	2	88%	100%
Scwiki Beta2	111 & 25	1	47	3	94%	60%	10	5	50%	100%
Php help agent 1.1	49 & 7	1	7	2	71%	100%	2	2	0%	100%
Phportal 1.2	35 & 100	3	190	4	98%	50%	21	8	62%	100%
Pragyan 2.6.2	141 & 163	1	107	5	95%	100%	20	9	55%	100%1

REFERENCES

[1] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In WWW '03: Proceedings of the 12th International Conference on World Wide Web, 2003.
[2] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In IFIP Security 2005, 2005.
[3] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In Recent Advances in Intrusion Detection 2005 (RAID), 2005.
[4] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In Proceedings of the 13th InternationalWorldWideWeb Conference, 2004.
[5] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T.Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In DSN, 2004.
[6] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In Proceedings of the 14th Usenix Security Symposium, Aug. 2005.
[7] Y. Minamide. Static approximation of dynamically generated web pages. In WWW '05: Proceedings of the 14th International Conference on World Wide Web, 2005.
[8] FIS. http://www.segfault.gr
[9] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In The 21st ACM Symposium on Applied Computing (SAC 2006), 2006
[10] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In WWW '04: Proceedings of the 13th International Conference on World Wide Web, 2006.
[11] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In The 21st ACM Symposium on Applied Computing (SAC 2006), 2006.
[12] Yichen Xie, Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In Proceedings of the 15th USENIX Security Symposium, pages 179-192, July 2006.
[13] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In IEEE Symposium on Security and Privacy, 2006.
[14] PHP-Sat. http://PHP-SAT.org
[15] DAPHPScan version 1.0. http://www.acid-root.new.fr
[16] http://www.netcraft.com
[17] http://www.securityfocus.com
[18] http://www.milw0rm.com

1 Our algorithm considers one vulnerable multi-argument function more than one report item. So, it reports 9 items which is much more than report items in another algorithm (DAPHPScan).