RIPS - A static source code analyser for vulnerabilities in PHP scripts

Johannes Dahse

1 Introduction

The amount of websites have increased rapidly during the last years. While websites consisted mostly of static HTML files in the last decade more and more webapplications with dynamic content appear as a result of easy to learn scripting languages such as PHP and other new technologies. In fact, PHP is the most popular scripting language on the world wide web today. Besides a huge amount of new possibilities the new web 2.0 also brings a lot of security risks when data supplied by a user is not handled carefully enough by the application. Different types of vulnerabilities can lead to data leakage, modification or even server compromise. In the last year 30% of all vulnerabilities found in computer software were PHP-related [1].

In order to contain the risks of vulnerable webapplications penetration testers are hired to review the source code. Given the fact that large applications can have thousands of lines of code and time is limited by costs a manual source code review can be incomplete. Tools can help penetration testers, to mitigate time and costs by automating time intense processes while reviewing source code.

In this submission a tool is introduced that can reduce the time a penetration tester needs by automating the process of identifying potential security flaws in PHP source code by using static source code analysis. The finds can then be easily reviewed by the penetration tester in its context without reviewing the whole source code again. Given the limitations of static source code analysis a vulnerability needs to be confirmed by the code reviewer.

2 The concept of taint analysis

By doing source code audits over and over again it is noticed that the same procedure of finding security flaws is done frequently. First potentially vulnerable functions (PVF) like `system()` or `mysql_query()` which can lead to certain vulnerabilities are detected and then their parameters consisting of variables are traced back to their origin. If the parameters with which the PVF has been called can be specified or modified by a user this parameter is marked as tainted and the PVF call is treated as a potential security vulnerability. Sources for userinput in PHP can be the global variables `$_GET`, `$_POST`, `$_COOKIE` and `$_FILES` as well as some `$_SERVER` and `$ENV` variables [2]. Also several functions that read from databases, files or enviroment variables can return userinput and taint other variables. When parameters are traced backwards the conditional program flow and potential securing actions have to be taken into account to avoid false positives.

The following two PHP scripts use the PVF `system()` that executes system commands [3]:

Listing 1: Example 1

```
1   <?php
2       $a = $_GET['a'];
3       $b = $a;
4       system($b, $ret);
5   ?>
```

Listing 2: Example 2

```
1   <?php
2       $a = $_GET['a'];
3       $b = 'date';
4       system($b, $ret);
5   ?>
```

While the first example shows a remote command execution vulnerability where a user can specify any command to be executed given by the GET parameter `a` the second example is not a vulnerability because the command being executed is static and can not be influenced by an attacker.

In order to automate the process of finding security flaws a large list of PVF is build consisting of PHP functions that can lead to a security flaw when called with unsanitized user input. This list includes quite unknown PVF like `preg_replace_callback()` or `highlight_file()` to name a few exotics. RIPS in its current state scans for 139 PVFs.

Once a PVF is detected the next step is to identify its parameters. In the first example this is variable `$b` and `$ret`. Those variables are compared to previously declared variables. This is where line 3 is found which assignes `$a` to `$b`. Again `$a` will be compared to previously declared variables and so on. If a parameter originated from user input the PVF call is treated as a potential vulnerability. The tree of traced parameters is then shown to the user in reversed order who can decide between a correct vulnerability or a false positive.

Listing 3: scan result for example 1

```
    4:  system($b, $ret);
        3:  $b = $a;
            2:  $a = $_GET['a'];
```

It is important to trace only significant parameters to reduce false positives. The second parameter of the function `system()` declares the return value of the command execution to the variable `$ret` in our example. Therefore the second parameter `$ret` should not get traced because a previously defined variable `$ret` with userinput can lead to false positives. Another source for false positives is securing actions taken by the developer. The following example is considered as safe:

Listing 4: Example 3

```php
<?php
    $a = $_GET['a'];
    $b = escapeshellarg($a);
    $c = 'cal ' . $b;
    system($c, $ret);
?>
```

The function `escapeshellarg()` prevents the attacker to inject arbitrary commands to the system call [4]. Also a typecast of `$a` to integer assigned to `$b` would prevent a command execution vulnerability. Therefore a list of securing functions is assigned to each element in the PVF list as well as a global list of securing or eliminating functions (e.g. `md5()`) and actions (e.g. typecasts) is defined. Because securing can be implemented wrongly the user has to have the option to review all found potential vulnerabilities with securing operations.

Here is an example of a PVF entry for the function `system()` with the significant parameter (the first) and securing functions (`escapeshellarg()` and `escapeshellcmd()`):

Listing 5: PVF entry for `system()`

```php
"system" => array(
    array(1), array("escapeshellarg", "escapeshellcmd")
);
```

The significant parameter can also be a list of parameters or `0` if all parameters should be treated as dangerous. Each PVF can be configured precisely like that. The difficult part is to take program flow and code structures into consideration while tracing parameters.

3 The tokenizer

In order to analyse a PHP script correctly the code is split into tokens. For this the PHP function `token_get_all()` [5] is used. Each token is an array with a token identifier which can be turned into a token name by calling `token_name()` [6], the token value and the line number. Single characters which represents the codes semantic appear as string in the token list.

Listing 6: Example 4

```php
<?php
    $a = $_GET['a'];
    system($a, $ret);
?>
```

Listing 7: token list of example 4 generated by `token_get_all()`

```
name: T_OPEN_TAG                        value: <?php      line: 1
name: T_VARIABLE                        value: $a         line: 2
name: T_WHITESPACE                      value:            line: 2
                                               =
name: T_WHITESPACE                      value:            line: 2
name: T_VARIABLE                        value: $_GET,     line: 2
                                               [
name: T_CONSTANT_ENCAPSED_STRING        value: 'a',       line: 2
                                               ]
                                               ;
name: T_WHITESPACE                      value:            line: 2
name: T_STRING                          value: system     line: 3
                                               (
name: T_VARIABLE                        value: $a         line: 3
                                               ,
name: T_WHITESPACE                      value:            line: 3
name: T_VARIABLE                        value: $ret       line: 3
                                               )
                                               ;
name: T_WHITESPACE                      value:            line: 3
name: T_CLOSE_TAG                       value: ?>         line: 4
```

Once the token list of a PHP script is obtained there a several improvements made to analyse the tokens correctly. This includes replacing some special characters with function names (like `` `$a` `` to `backticks($a)` which represents a command execution [7]) or adding curly braces to program flow constructs where no braces have been used (in example `if` or `switch` conditions with only one conditional line following [8]). Also all whitespaces, inline HTML and comments are deleted from the token list to reduce the overhead and to identify connected tokens correctly.

Then the source code can be analysed token by token [9]. The goal of RIPS is to analyse the token list of each file only once to improve the speed. It is looping through the token list and identifies important tokens by name. Several actions are being done when one of the following token is identified.

T_INCLUDE If a file inclusion is found the tokens of the included file will be inserted to the currented token list. Also there is a note about the success of the inclusion added to the output if information gathering is turned on. If the file name consists of variables and strings the file name can be reconstructed dynamically. A internal file pointer keeps track of the current position in the included files. Also each file inclusion is checked for a file inclusion vulnerability.

T_FUNCTION If a new function is declared the name and the parameters are analysed saved for further analysis.

T_RETURN If a user defined function returns a variable, this variable will get traced backwards and is checked for securing actions. If the returned variable is sanitized by a securing or neutralizing function like `md5()` or a securing action like a typecast this function is added to the global securing function list so that user defined sanitizing functions can be identified. If the return value is tainted by userinput the function is added to a list of functions that can taint other variables when assigned to them.

T_VARIABLE If a variable declaration is identified the current scope is checked and the variable declaration is added either to a list of local (if the token is found in a function declaration) or a global variable list together with the according line of the source code. Here are some examples:

Listing 8: examples for variable declarations

```
$a  =>  $a = $_GET['a'];
$b  =>  $b = '';
$b  =>  $b.=$a;
$c['name']  =>  $c['name'] = $b;
$d  =>  while($d = fopen($c['name'], 'r'))
```

This list can be used to trace nested variable declarations backwards to their origin. Also all dependencies are added to each variable declaration to make a trace through different program flows possible.

T_STRING If a function call is detected the tool checks whether the function name is in the defined PVF list and therefore a function call to scan further. A new parent is created and all parameters configured as valuable will get traced backwards by looking up the variable names in the global or local variable list. Finds are added to the PVF tree as a child. All variables in the previously found declarations will also get looked up in the variable list and added to the corresponding parent. If securing actions are detected while analysing the line of a variable declaration the child is marked red. If user input is found the child is marked white and the PVF tree is added to the global output list. Optionally parameters can be marked as tainted if they are tainted by functions that read SQL query results or file contents. Therefore it is possible to identify vulnerabilities with a persistent payload storage.

Also if a traced variable of a PVF in a user defined function declaration depends on a parameter of this function the declaration is added as child and marked yellow. Then this user defined function is added to the PVF list with the according parameter list. The list of securing functions is adapted from the securing functions defined for the PVF found in this user defined function.

At the end all currently needed dependencies in the program flow are addded.

Listing 9: Example 4

```
1  <?php
2      function myexec($a, $b, $c)
3      {
4          exec($b);
5      }
6
7      $aa = "test";
8      $bb = $_GET['cmd'];
9      myexec($aa, $bb, $cc);
10 ?>
```

The PVF call on line 4 is detected and parameter `$b` is traced backwards. It is detected that `$b` depends on a function parameter. Now the function `myexec()` is added to the PVF list with the second parameter defined as valuable and the securing functions defined for `exec()`.

The user defined function `myexec()` is now treated as any other PVF function. If a call with userinput is found the call and the vulnerability is added to the output.

Listing 10: the original PVF is shown

```
4:   exec($b);
        2:   function myexec($a, $b, $c)
```

Listing 11: and the function call that triggers the vulnerability

```
9:   myexec($aa, $bb, $cc);
        7:   $aa = "test";
        8:   $bb = $_GET['cmd'];
```

Additionally variables that are traced and declared in a different code structure than the PVF call was found in will be commented with the dependency for the variable declaration. Dependencies that affect both stay as global dependency for this parent.

T_EXIT, T_THROW Tokens that can lead to a exit of the program flow are also detected and the last found control structure the exits depends on (e.g. a `if` or `switch` statement) is added to the current dependency list. If a program exit is found in a function declaration this function is added to the list of interesting functions with the comment, that a call may cause a exit. With this the user can get an overview which conditions have to be made in order to get to the desired PVF call in the program flow.

Curly braces {} All program flow is detected by curly braces and therefore the tokens have to be prepared in situations where no braces have been used by the programmer. Control structures like if and switch are added to a list of current dependencies. If a PVF is detected in the same block of braces the dependencies will be added to the parent. A closing brace marks the end of the control structure and the dependency is removed from the current dependencies list.

Additionally tokens that identify PHP specialities like `extract()`, `list()` or `define()` are evaluated to improve the correctness of the results. In addition a list of interesting functions which identify a DBMS or session usage is defined and found calls are added to the output with a comment as information gathering if the verbosity level is set to do so.

4 The web interface

RIPS can be completly controlled by a web interface. To start a scan a user has to provide a file or directory name, choose the vulnerability type and click `scan`. Addionally a verbosity level can be chosen to improve the results:

- The default verbosity level 1 scans only for PVF calls which are tainted with user input without any detected securing actions in the trace.

- The second verbosity level also includes files and database content as potentially malicious user input. This level is important to find vulnerabilites with a persistent payload storage but it might increase the false positive rate.
- The third verbosity level will also output secured PVF calls. This is important to detect unsufficient securings which are sometimes hard to detect by a static source code analyser automatically.
- The fourth verbosity level also shows additional informations RIPS collected during the scan. This includes found exits, notes about the success of analysing included files and calls of functions that has been defined in the interesting functions array. On large PHP applications this information gathering can lead to a very large scan result.
- The last verbosity level 5 shows all PVF calls and its traces no matter if tainted by user input or not. This can be useful in scenarios where a list of static input to PVF calls is of interest. However this verbosity level can lead to a lot of false positives.

All found PVF calls and their traces are shown syntax highlighted to the user devided in blocks. The syntax highlighting of the PHP code can be changed on the fly by choosing from 6 different stylesheets. The color schemes are manually adapted from Pastie (pastie.org) and integrated into RIPS own syntax highlighter.

Also a drag and dropable window can be opened to see the original source code by clicking on the file icon. All lines used in the PVF call and its trace are highlighted red in the original code and the code viewer automatically jumps to the PVF call to allow a quick and easy review of the trace.

Another window can be opened for every found vulnerability to quickly create a PHP curl exploit with a few clicks by hitting the target icon. Depending on the found userinput there are prepared code fragments which can be combined to create a exploit in a few seconds by entering the parameter values and a target URL. For multi-stage exploits cookies are supported as well as SSL, HTTP AUTH and a connection timeout.

For further investigation of complicated vulnerabilities a list of user defined functions is given to the user which allows him to directly jump into the functions code by clicking on the name. Also all user defined functions called in the scan result can be analysed by placing the mouse cursor over the function name. Then the code of the function declaration is shown in a mouseover layer. Jumping between finds in a user defined function and the according call of this function is also possible.

5 Results

In order to test RIPS the source code of the current webapplication security internship platform at the Ruhr-University Bochum was scanned. This platform is a virtual online banking webapplication written in PHP and it is designed to have several web application vulnerabilities to test the students abilities learned during the internship.

At first RIPS is run with verbosity level 1 meaning to find PVFs directly tainted with user input and no securing detected. RIPS scanned 17399 lines in 90 files for 142 functions in 2.096 seconds. The following intended vulnerabilities could be found:

- 1/2 reflektive Cross-Site Scripting
- 0/1 persistent Cross-Site Scripting
- 2/2 SQL Injections
- 0/1 Business Logic Flaws
- 1/1 File Inclusion
- 1/1 Remote Code Execution
- 1/1 Remote Command Execution

Two false positives occured. Suprisingly there also was a yet unknown HTTP Response Splitting and another unintended Cross-Site Scripting vulnerability detected.

One false positive was a SQL injection wich has been prevented by a regular expression and therefore could not be evaluated as correct sanitization by RIPS. Another false positive occured with a `fwrite()` call to a logging file. Because of the fact that the file was a text file and the data is sanitized correctly when read by the application again this does not lead to a security flaw. However, it is important to know for the source code reviewer in what files an attacker is able to write because this can lead to other vulnerabilities (e.g. when the attacker can write into a PHP file).

A significant false negative is the missing reflektive XSS vulnerability. This one could only be detected by reviewing the secured PVF calls when setting the verbosity level to 3. The missing argument `ENT_QUOTES` in the securing function `htmlentities()` lead to a false detection of sufficient securing in a scenario where an eventhandler could be injected to an existing HTML element.

To detect the persistent XSS vulnerability RIPS was set to verbosity level 2 and thus allowing to treat file and database content as tainting input for PVFs. The persistent XSS vulnerability was detected successfully, however this verbosity level also lead to 11 false positives. That is because RIPS has no information if an attacker can insert data into the database at all or what kind of table layout is used. Almost all false positives affected a harmless column `id` whith type integer and `auto_increment` set.

As expected the Business Logic Flaw could not be found by taint analysis for PVF because it uses the applications logic without any PVF.

6 Limitations and future work

The main limitation of static source code analysis is the evaluation of dynamic strings. In PHP the name of a included file can be generated dynamically at runtime. Currently RIPS is only capable of reconstructing dynamic file names composed of strings and variables holding strings. However if the file name is constructed by calling functions the name can not be reconstructed. Particularly large PHP projects rely on an interaction of several PHP scripts and a security flaw might depend on several files to work and to get detected correctly. Future work includes adressing this problem. One option could be to combine dynamic and static source code analysis to evaluate dynamic file names. Currently the best workaround is to rewrite complex dynamic file names to static hardcoded file inclusions.

Also it should be obvious that RIPS is only capable of finding security vulnerabilities that are considered as bugs and not as intended obfuscated backdoors which can easily be hidden with dynamic function names:

Listing 12: dynamic backdoor not found by RIPS

```php
$a=base64_decode('c3lzdGVt');$a($_GET['c']);
```

The same limitation appears for a user defined securing function that relies on regular expressions or string replacements which can not be evaluated during a static source code analysis. Therefore it is not possible to determine if securing taken by the developer is safe or not in each scenario. This can lead to false positives or negatives. As a compromise the user has the option to review secured PVF calls.

In the future it is planned to fully support object oriented programming. Vulnerable functions in classes are detected but no interaction with variables assigned to an object is supported by RIPS in its current state as well as classes that implement or extend other classes.

Additionally it is planned to consider automatic typecasts. Currently a typecast by adding an integer to a string is not recognized and may lead to false positives in certain circumstances.

7 Related work

Various techniques such as flow-sensitive, interprocedural, and contextsensitive data flow analysis are described and used by the authors of Pixy [10], the first open source static source code analyser for PHP written in Java [11]. The goal of RIPS is to build a new approach of this written in PHP itself using the build-in tokenizer functions. Unlike Pixy, RIPS runs without any requirements like a database managment system, the Java enviroment or any other programming language than PHP itself. While Pixy is great in finding vulnerabilities with a low rate of false positives, it only supports XSS and SQL injection vulnerabilities. Both vulnerabilities are the most common vulnerabilities in PHP applications but RIPS aims to find a lot more common vulnerabilities including XSS and SQL injection, but also all kinds of header injections, file vulnerabilities and code/command execution vulnerabilities.

A difference in the user interface is that RIPS is designed to easily review and compare the finds with the original source code for a faster and easier confirmation and exploitation and therefore to give a better understanding of how the vulnerabilities work instead of pointing out that the application is vulnerable in a specific line. Often a vulnerability can be found very fast in the depth of the source code and the hard part is to trace back under which conditions this codeblock is called. Since static source code analysis is likely to fail for complicated vulnerabilities RIPS goal is to do its best at finding flaws automatically but also to provide as much information and options to make further analysis as easy and fast as possible.

Compared to Pixy, RIPS is also capable of finding vulnerabilities with persistent payloads stored in files or databases by using different verbosity levels. A disadvantage compared to Pixy is that the lexical analyzation of RIPS assumes some „good coding practices“ in the analysed source code to analyse it correctly. Future work will include to make the lexical analyzation more flexible. Also a lot of research about Aliases in PHP has been done by the authors of Pixy [12] which is not supported by RIPS because of its rareness. Both tools suffer from the limitations of static source code analysis as described in the previous section.

An extended version of Pixy called Saner [13] has been created to address the problem with unknown user defined securing actions and its efficiency. It uses predefined test cases to check whether the filter is efficient enough or not. This approach could also be included to RIPS to avoid a review of secured PVF calls.

Additionally there exists tools like Owasp Swaat [14], which are designed to find security flaws in more than one language but which only detects vulnerable functions by looking for strings. This is sufficient for a first overview of potential unsafe program blocks but without consideration of the application context real vulnerabilities can not be confirmed. However this method with an additional parameter trace can also been forced with RIPS by setting the verbosity level to 5.

8 Summary

In the past there have been a lot of open source webapplication scanners released that aim to find vulnerabilities in a black box scenario by fuzzing. A source code review in a white box scenario can lead to much better results but only a few open source PHP code analyzers are available. RIPS is a new approach using the buildin PHP tokenizer functions. It is specialized for fast source code audits and can save a lot of time. Results have shown, that RIPS is capable of finding known and unknown security flaws in large PHP-based webapplications within seconds. However due to the limitations of static source code analysis and some assumption on the programm code false negatives or false positives can occur and a manual review of the outlined result has to be made or the verbosity level has to be loosend to detect previoulsy missed vulnerabilities that could not be identified correctly. The webinterface assists the reviewer with a lot of useful features like the integrated codeviewer, a list of all user defined functions and a connection between both. Therefore RIPS should be treated as a tool that helps analysing PHP source code for security flaws but not as a ultimate security flaw finder.

References

[1] Fabien Coelho, *PHP-related vulnerabilities on the National Vulnerability Database*
`http://www.coelho.net/php_cve.html`

[2] The PHP Group, *Predefined Variables*
`http://www.php.net/manual/en/reserved.variables.php`

[3] The PHP Group, *system - Execute an external program and display the output*
`http://www.php.net/system`

[4] The PHP Group, *escapeshellarg - Escape a string to be used as a shell argument*
`http://www.php.net/escapeshellarg`

[5] The PHP Group, *token_get_all - Split given source into PHP tokens*
`http://www.php.net/token-get-all`

[6] The PHP Group, *token_name - Get the symbolic name of a given PHP token*
`http://www.php.net/token-name`

[7] The PHP Group, *Execution Operators*
`http://php.net/manual/en/language.operators.execution.php`

[8] The PHP Group, *Control Structures*
`http://php.net/manual/en/language.control-structures.php`

[9] The PHP Group, *List of Parser Tokens*
`http://php.net/manual/en/tokens.php`

[10] Nenad Jovanovic, Christopher Kruegel, Engin Kirda, *Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)*
`http://www.seclab.tuwien.ac.at/papers/pixy.pdf`

[11] Nenad Jovanovic, Christopher Kruegel, Engin Kirda, *Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report)*
`http://www.seclab.tuwien.ac.at/papers/pixy_techreport.pdf`

[12] Nenad Jovanovic, Christopher Kruegel, Engin Kirda, *Precise Alias Analysis for Static Detection of Web Application Vulnerabilities*
`http://www.iseclab.org/papers/pixy2.pdf`

[13] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, Giovanni Vigna, *Saner: Composing Static and Dynamic Analysis to Validate Sanitization inWeb Applications*
`http://www.iseclab.org/papers/oakland-saner.pdf`

[14] OWASP, *OWASP SWAAT Project*
`http://www.owasp.org/index.php/Category:OWASP_SWAAT_Project`