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

Johannes Dahse

Seminar Work

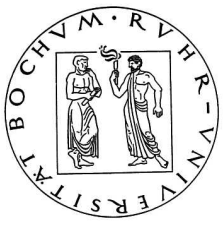
at

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk

advised through Dominik Birk

23.08.2010

Horst Görtz Institute Ruhr-University Bochum



Contents

1 Introduction **1**

2 Motivation **2**

3 Web application security **3**
3.1 Cross-Site Scripting 4
3.2 SQL Injection 5
3.3 Other vulnerabilities 6

4 Static source code analysis **7**
4.1 Configuration 7
4.2 Model construction 7
4.3 Analysis 8
4.3.1 Taint analysis 8
4.3.2 Intraprocedural and interprocedural analysis 9
4.4 Results processing 9

5 RIPS implementation **11**
5.1 Configuration 11
5.2 Model construction 12
5.2.1 Lexical and semantic analysis 12
5.2.2 Parsing 13
5.2.3 Control flow analysis 15
5.3 Analysis 15
5.3.1 Taint analysis 15
5.3.2 Intraprocedural and interprocedural analysis 16
5.4 Web interface 17
5.5 Scan results 19
5.6 Limitations and future work 20

6 Related work **22**

7 Summary **24**

1 Introduction

The amount of websites has increased rapidly during the last years. While websites consisted mostly of static HTML files in the last decade, more and more web applications with dynamic content appeared as a result of easy to learn scripting languages such as PHP and the growing availability and speed of the internet. Almost all web servers support some sort of scripting environment today to deploy dynamic web applications.

Besides a huge amount of new possibilities, the new web 2.0 also brings a lot of new 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. Oftenly one single unfiltered character can have a huge security impact. Because of limited programming skills, lacking security awareness and time constraints vulnerabilities can occur very often and put the whole web server at risk due to the easy accessibility on the internet.

In order to contain the risks of vulnerable webapplications source code has to be reviewed by the developer or by penetration testers. Given the fact that large applications can have thousands of codelines and time is limited by costs, a manual source code review might be incomplete. Tools can help penetration testers to minimize time and costs by automating time intense processes while reviewing a source code.

In this seminar work the concept of web application vulnerabilities is introduced and how they can be detected by static source code analysis automatically. Also a new tool named *RIPS* is introduced which automates the process of identifying potential security flaws in PHP source code. RIPS is open source and freely available at `http://www.sourceforge.net/projects/rips-scanner/`. The result of the analysis can easily be reviewed by the penetration tester in its context without reviewing the whole source code again. This seminar work will describe how RIPS is implemented and which kind of problems are faced when building a static source code analysis tool for PHP.

2 Motivation

The scripting language PHP is the most popular scripting language on the world wide web today. It is very easy to learn and even a programmer with very limited programming skills can build complex web applications in short time. But very often security is only a minor matter or not implemented at all, putting the whole web server at risk. In the last year, 30% of all vulnerabilities found in computer software were PHP-related [1]. The wide distribution of PHP and the many PHP-related vulnerabilities occuring lead to a high interest of finding security vulnerabilities in PHP source code quickly. This interest is shared by source code reviewers with good and bad intents.

My personal intent was to find security vulnerabilities quickly during *Capture The Flag* (CTF) contests. During a CTF contest each participating team sets up a web server with vulnerable web applications and connects to a virtual private network. The source code is then analysed by the teams and security vulnerabilities have to be found, patched and exploited. By exploiting security vulnerabilities it is possible to capture flags from opposing teams webservers that are awarded with points. The team with the most captured flags wins. About 22% of all distributed web applications by CTF organizers have been written in PHP so far thus making PHP the most popular scripting language in CTFs as well.

Building a tool that automates the process of finding security vulnerabilities can help to better secure your own source code, find potentially vulnerabilities in your own or open source PHP applications and to win CTF contests. In all cases it is helpful to find vulnerabilities accurately and in a short amount of time.

Unfortunately there exist only one open source tool called *Pixy* written in Java to analyse PHP source code files for SQL Injection and Cross-Site Scripting vulnerabilities. While these are the most common vulnerabilties there exist a lot more vulnerabilities a source code analysis tool should detect. Pixy has not received any updates since 2007 and is effectively abandoned. My motivation was to build a new tool that detects all known types of vulnerabilities in PHP scripts. Because it is nearly impossible to identify all vulnerabilities correctly 100% the goal was to give the user a powerful interface to review the vulnerable parts as easy as possible. Pixy and RIPS is compared in detail in chapter 6.

3 Web application security

A web application is a computer application that is deployed on a web server and accessable through a web-based user interface by the client. The HTTP server (e.g. Apache or Microsoft IIS) accepts HTTP requests send by the client (normally by a web browser such as Mozilla Firefox or Microsoft Internet Explorer) and hands back a HTTP response with the result of the request.

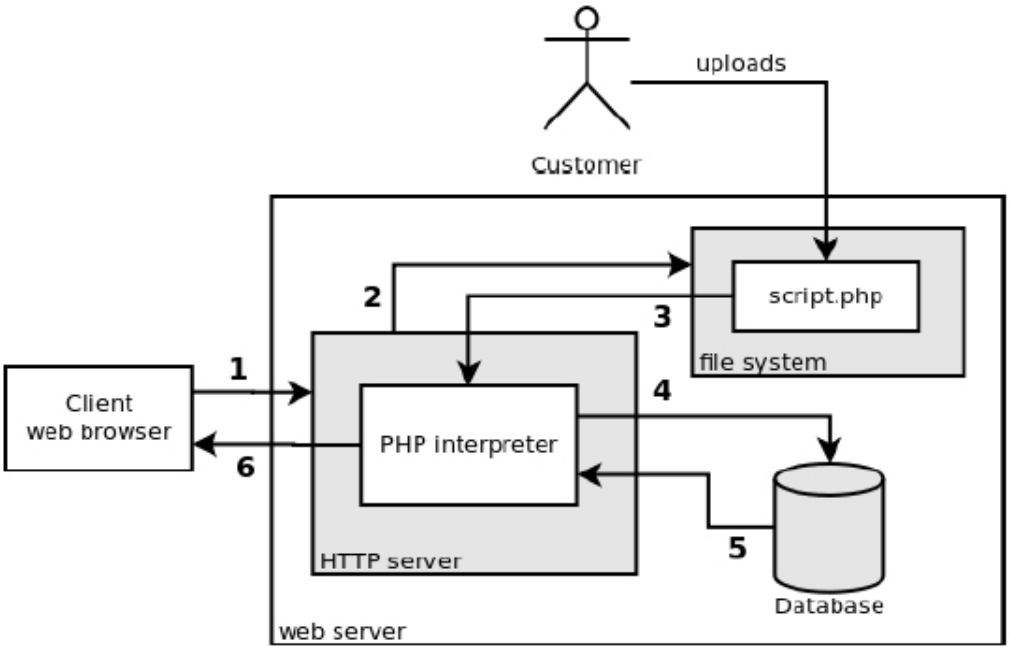


Figure 3.1: Typical web server setup [15]

When a HTTP request is send by the client it is parsed by the HTTP server (step 1 in figure 3.1). The HTTP server extracts the file name that is being requested and the parameters send (for example GET or POST parameters). When it detects a request for a dynamic file (in example a PHP script) it fetches the source code from the file system (step 2 in figure 3.1) and passes it along with the parameters to the scripting language interpreter (step 3 in figure 3.1). The interpreter executes the source code and handles external resources like databases (step 4 and 5 in figure 3.1) or files. It then creates the result (normally a HTML file) and sends it back to the client where the result is displayed in the web browser.

A web application security vulnerability can occur when data supplied by the user (e.g. GET

or POST parameters) is not sanitized correctly and used in critical operations of the dynamic script. Then an attacker might be able to inject code that changes the behavior and result of the operation during the script execution in an unexpected way. These kind of vulnerabilities are called *taint-style vulnerabilities* because untrusted sources such as user supplied data is handled as *tainted* data that reaches vulnerable parts of the program called *sensitive sinks*.

Any kind of data that can be modified by the user such as GET or POST parameters as well as cookie values, the user agent or even database entries or files have to be assumed as *tainted*. For each different vulnerability type exists different sensitive sinks and *sanitization routines*. *Sensitive sinks* are potentially vulnerable functions (PVF) that execute critical operations and should only be called with trusted or sanitized data. Otherwise an attacker may influence the data that is passed to the PVF and read, modify and delete data or attack the underlying web server or a client, depending on the PVF. Figure 3.2 shows the concept of taint-style vulnerabilities in PHP.

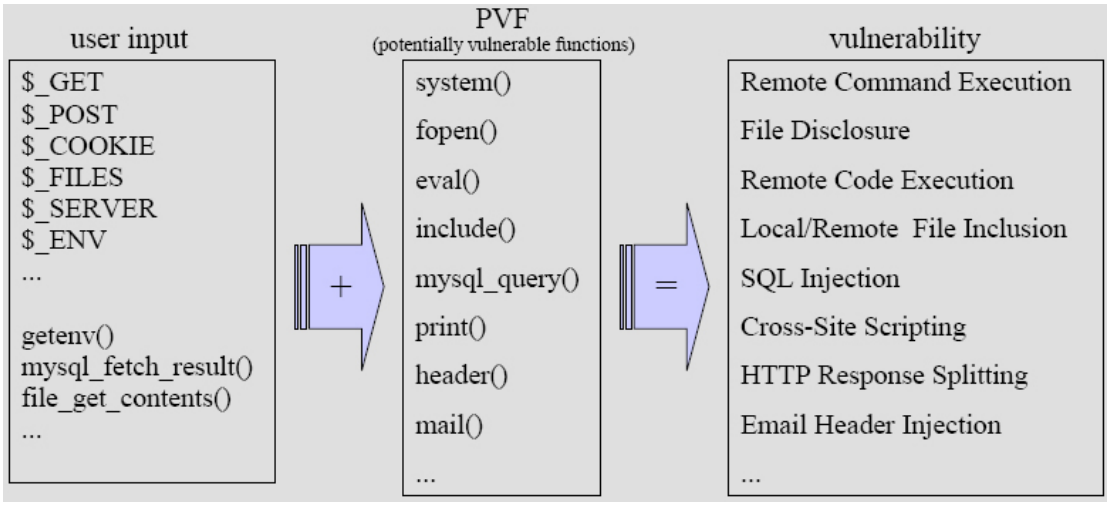


Figure 3.2: Concept of taint-style vulnerabilities in PHP

To give an example for a vulnerability that affects the client and a vulnerability that affects the server the two very common vulnerabiltities *Cross-Site Scripting* and *SQL Injection* are introduced.

3.1 Cross-Site Scripting

Cross-Site Scripting (XSS) attacks the client of the web application and occurs when user input is embedded into the HTML result unsanitized. An attacker can abuse this behavior by embedding malicious HTML or Javascript code to locally deface the web applications result or lunch other attacks. By embedding malicious Javascript it is possible to access the cookies of the clients browser and send them to the attacker to steal session information.

Listing 3.1: Unsanitized output of user input

```
<?php
    print('<h1>Welcome ' . $_GET['name'] . '</h1>');
?>
```

Listing 3.1 shows a typical XSS vulnerability. A user can specify a name by GET parameter and the name is embedded into the HTML result page. Instead of choosing an alphanumerical name he can also choose a name that consists of HTML and Javascript code:

```
http://www.example.com/?name=foo</h1><script>alert(1)</script>
```

The code will be embedded into the output and send back to the clients browser which will parse the HTML and execute the Javascript code. An attacker can abuse this by sending this modified link with his payload in the GET parameter to a victim which will execute the Javascript code unintentially. The crafted HTML result is shown in listing 3.2.

Listing 3.2: Resulting HTML of application in listing 3.1 with injected Javascript

```
<h1>Welcome foo</h1><script>alert(1)</script></h1>
```

To patch this vulnerability the output has to be validated. Characters like < and > as well as quotes can be replaced by their HTML entities. This way the characters will still be displayed by the browser but not rendered as HTML tags. In PHP the buildin function `htmlentities()` and `htmlspecialchars()` can be used for output validation.

3.2 SQL Injection

Web applications are often connected to external resources like databases to handle large data sets. To query the database the *structered query language* (SQL) is used. If a web application creates a SQL query on runtime with unsanitized user input an attacker might inject own SQL syntax to modify the SQL query. This vulnerability is called *SQL Injection*.

Listing 3.3: SQL query build with unsanitized user input

```
<?php
$id = $_POST['id'];
$query = "SELECT id,title,content FROM news WHERE id = $id";
mysql_query($query);
?>
```

Listing 3.3 shows a SQL Injection vulnerability. A user can specify an `id` by POST parameter that is embedded unsanitized into a SQL query and send to the database. The database will respond with a news article from the table `news` corresponding to the specified ID. An attacker can inject own SQL syntax to modify the result of the query and gain sensitive information:

```
id=1+UNION+SELECT+1,username,password+FROM+users
```

Listing 3.4 shows the SQL query that will be build at runtime when posting this `id`. The injected SQL syntax will read sensitive information from the table `users`.

Listing 3.4: SQL query with injected SQL code

```
$query =  "SELECT id,inhalt FROM news WHERE id = 1
        UNION SELECT 1,username,password FROM users";
```

Depending on the database managment system, configuration and privileges even attacks on the web server could follow by abusing SQL functionalities.

To patch this vulnerability user input has to be sanitized before embedding it into the query. Expected integer values should be typecasted to `int` to assure no SQL syntax can be embedded. Expected strings are normally embedded into quotes.

Listing 3.5: Sanitized string embedded in quotes

```
<?php
$name =  mysql_real_escape_string($_POST['name']);
$query = "SELECT id,message FROM users WHERE name = '$name' ";
mysql_query($query);
?>
```

In this case it is sufficient to avoid a break out of these quotes by escaping the quote character so that it is handled as regular character and not as SQL syntax. In PHP the buildin function `mysql_real_escape_string()` can be used for escaping strings before placing them into a MySQL query as shown in listing 3.5.

3.3 Other vulnerabilities

There are many other taint-style vulnerabilties like *File Disclosure* and *File Manipulation* vulnerabilities that allow an attacker to read and write to arbitrary files. *File Inclusion* vulnerabilities enables an attacker to include arbitrary PHP files and execute PHP code. Uncommon vulnerabilities like *Remote Code Execution* or *Remote Command Execution* vulnerabilities even allow an attacker to execute arbitrary system commands on the web server.

However all taint-style vulnerabilities rely on the same principle as XSS and SQL Injection: a PVF is called with untrusted and unsanitized data allowing malicious users to change the behavior and actions of this PVF to their advantage. Depending on the vulnerability their may be PHP buildin functions that can prevent this type of vulnerability.

4 Static source code analysis

During a static source code analysis a computer program's source code is analysed without executing it. Executing a program will make it run through a single path, depending on the input during that execution, and consequently tools that analyze a program while it is running (dynamic analysis) can only examine those parts of the program that are reached during that particular run. Static analysis allows one to examine all the different execution paths through a program at once and make assessments that apply to every possible permutation of inputs. Static analysis is widely used for a variety of goals such as syntax highlighting, type checking, optimization as well as bug and security finding. In general, any tool that examines the source code of a program without executing it, can be categorized as a static analysis tool. [15]

There are four major steps to analyse a source code statically: configuration, model construction, analysis and results processing. During the analysis step configured security knowledge can be used to detect vulnerabilities.

4.1 Configuration

First the tool needs a precise rule set to know what to look for. For a vulnerability analysis it has to be specified which types of vulnerability should be detected and how sensitive sinks can be identified. Also the sources (user input) has to be declared that can taint data. Lastly appropiate securing actions that can be taken by the developer should be configured and detected to prevent secured parts of the program in the analysis result.

4.2 Model construction

A static analysis tool has to transform source code into a program model. This is an abstract internal representation of the source code. The quality of a tool's analysis is largely dependent on the quality of the tool's program model so that it is very important that an analysis tool has a good understanding of the language's semantics. Building a program model takes the following steps:

Lexical analysis The tool has to split the source code into tokens to identify language constructs correctly. Unimportant tokens like whitespaces and comments are normally stripped to identify connected tokens correctly.

Semantic analysis The analysis tool checks the representation of each token. In example it is important to decide between the function call *print* and the same word used in a string. Also variable types can be determined.

Control flow analysis All possible paths that can be taken through the program are determined. This includes casual jumps and function calls. The paths are then combined to several *control flow graphs* (CFG) which represent all possible data flow paths.

Data flow analysis The tool uses data flow analysis on each CFG to determine how data moves throughout the program. In a security vulnerability analyser *taint analysis* is used to determine where vulnerabilities occur. Data flow analysis bases on the results of the semantic and control flow analysis and therefore it is important, that previous results are as correct as possible.

4.3 Analysis

When the model is build the tool can perform taint analysis on every found PVF. It also has to perform *intraprocedural analysis* for analysing a PVF call in an individual function and *interprocedural analysis* to perform analysis through an interaction between several functions.

4.3.1 Taint analysis

During taint analysis it is determined, where tainted data reaches sensitive sinks and therefore a security flaw may exist. The tool has to differentiate between tainted sensitive sinks and untainted sensitive sinks during the data flow analysis step. Listing 4.1 and Listing 4.2 show two PHP scripts that use the PVF `system()` that executes system commands [3].

Listing 4.1: Remote Command Execution vulnerability

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

Listing 4.2: Harmless static command execution

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

While listing 4.1 shows a Remote Command Execution vulnerability where a user can specify any command to be executed given by the GET parameter `a`, listing 4.2 is not a vulnerability because the command being executed is static and cannot be influenced by an attacker. By analysing the data flow of the PVFs parameter the tool can determine if untrusted data reaches the sink or if the PVF is called with harmless static data.

4.3.2 Intraprocedural and interprocedural analysis

During taint analysis the tool also has to perform *intraprocedural analysis*. This analysis involves tracking data within a user defined function that has been called. If a sensitive sink has been found in a function declaration the tool has to decide under which circumstances this function has to be called to trigger a vulnerability in the sensitive sink. This is mostly the case when the sensitive sink depends on parameters of the user defined functions that has to be called with tainted data.

The tool also must be able to decide if tainted data leaves the function untouched or if it is sanitized by running through the function. Also a function can return tainted data although it has not been called with tainted data (in example when new user input is accepted during the function execution).

Interprocedural analysis is about understanding the context outside of the function during a call. Depending on the program state the call might behave differently or external variables in global scope might be changed. Since static source code analysis assumes all possible CFG data can run through, both analysis can be very difficult to achieve without increasing the analysis time too much.

4.4 Results processing

The last important part of static source code analysis is to present the results to the user in such a way that he is able to quickly spot critical flaws. Displaying only relevant parts of the vulnerable code as well as syntax highlighting helps reviewing and understanding the problem. It is also helpful to provide automated information how this flaw can be fixed.

If a problem-free section of code is inappropriately marked as vulnerable, we talk about a *false positive* or false alarm. If a tool fails to identify a vulnerability when in fact there is one, we talk about a *false negative*. Conversely, a correctly identified vulnerability is known as a *true positive*, while an appropriate absence of warnings on a secure section of code is called a *true negative*. False positives are generally seen as intrusive and undesirable and may lead to the rejection of a tool. False negatives are arguably even worse, because they can give the user a false sense of security.[15]

The value of the tool is the better, the more true positives can be identified and the less false positives are displayed. Static source code analysis tools often generate a lot of false positives and warnings. This may occur due to incorrect semantic or flow analysis or circumstances that are needed for a correct identification of a vulnerability but which could not be evaluated by the tool correctly. Therefore it is important to present the results in such a way that the user can

easily decide between a correct vulnerability detection or a false positive himself.

5 RIPS implementation

This chapter describes how the theorethical concepts of static source code analysis are implemented in RIPS. The analyser is written in PHP and the result is a HTML file with a window managment written in Javascript offering several options to review details of the result. RIPS does not need any requirements other than a local web server with a PHP interpreter and a web browser installed.

5.1 Configuration

First several file extensions are configured to analyse only PHP files when scanning directories. To identify PVFs in source code files a huge list of PVFs is created and categorized into vulnerability types to allow the user to scan only for a specific vulnerability. Besides the function name the significant parameters are configured as well. This prevents false positives, where tainted data in the first parameter leads to a vulnerability, but tainted data in the second parameter is not critical. Also, this improves the performance because not related parameters are not traced back. The universal parameter `0` can be selected, when all parameters are potentially vulnerable and thereforce significant. This is important for functions, that can have a variable amount of parameters such as `print()`. RIPS currently scans for 186 PVFs seperated into the following vulnerability types:

- Cross-Site Scripting (5)
- SQL Injection (54)
- File Disclosure (37)
- File Manipulation (20)
- File Inclusion (7)
- Remote Code Execution (17)
- Remote Command Execution (8)
- Connection Handling (28)
- XPath Injection (3)
- Other (7)

Also to each PVF a list of PHP buildin securing functions is configured. This is important to prevent false positives when securing actions are already taken by the developer. A configured PVF entry is implement as simple list of arrays. Listing 5.1 shows an example of a PVF entry for the function `system()` with the significant parameter (the first) and securing functions (`escapeshellarg()` and `escapeshellcmd()`).

Listing 5.1: PVF config entry for `system()`

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

Additionally a global list of securing functions is defined containing functions that will prevent the exploitation of every PVF. Those are usually functions that will return only integers like `strlen()` or sanitized strings like `md5()` that will prevent code injection.

For each vulnerability type there is also a short description, example code, example patch and example exploitation configured to help the user understand the vulnerability that was found.

All tainting sources (user input) has to be configured to identify a vulnerability during a back trace of PVF parameters. User input is commonly passed through GET, POST und Cookie parameters but also through uploaded file names and server enviroment variables such as the user agent or query string. In PHP the global variables `$_GET`, `$_POST`, `$_COOKIE` and `$_FILES` as well as `$_SERVER` and `$ENV` variables are handling these data [2]. Also tainted user input can come from files or databases when written or inserted previously by a user to that external resource. For this case functions that read from files or databases are configured as well.

For information gathering a list of interesting functions (41) is declared that will create a note in the scan result each time a function call of this list is detected. Example are `session_start()` that indicates a session managment or `mysql_connect()` that indicates the usage of the database managment system (DBMS) *MySQL*.

5.2 Model construction

RIPS uses the PHP buildin tokenizer extension that splits PHP source code into tokens. The whole model of the source code however is created during the analysis phase and not before. While this has great performance advantages it is assumed that everything used during the analysis phase has been detected within the previous source code. However in PHP it is allowed to call functions in the source code before they are actually declared later on.

5.2.1 Lexical and semantic analysis

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 the buildin function `token_name()` [6], the token value and the line number. Single characters which represent the codes semantic appear as string in the token list. Listing 5.2 shows a PHP example code and listing 5.3 its representative token list generated by `token_get_all()`.

Listing 5.2: PHP example code

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

Listing 5.3: Token list for listing 5.2 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 represent 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].

5.2.2 Parsing

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. For each scanned file it creates a dependency stack, a file stack, a list of declared variables and several registers that indicate whether it currently scans in a function, class or any other language structure. Several actions are performed when one of the following tokens is identified:

T_INCLUDE If a file inclusion is found, the tokens of the included file will be added to the currented token list as well as an additional token that identifies the end of the included tokens. 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 and saved for further analysis.

T_CLASS If a new class is declared, the name is saved for further analysis and a new list of potentially vulnerable class functions is created. This list is used to save vulnerable user-defined functions depending on their class because several classes can name their functions the same and this could lead to false positives.

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 user input, 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 to a global variable list together with the according line of the source code. Listing 5.4 shows some examples for variable declarations in PHP.

Listing 5.4: Examples for variable declarations in PHP

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

The examples show that it is not sufficient to only parse = and `;` in the token list to identify every variable declaration correctly. RIPS uses this list to trace variables found in PVF calls backwards to their origin during taint analysis. Also all dependencies are added to each variable declaration to make a trace through different program flows possible. RIPS avoids using control flow graphs due to performance reasons, however this might lead to false negatives in very specific scenarios.

Additionally tokens that identify PHP specialities like `extract()`, `list()` or `define()` are evaluated to improve the correctness of the results. Also a list of interesting functions is

defined which identify a DBMS or session usage and detected calls are added to the output with a comment as information gathering if the verbosity level is set to do so.

5.2.3 Control flow analysis

The following tokens are used to perform control flow analysis:

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 current dependencies stack. If a PVF is detected in the same block of braces, the current dependencies will be added to this find. A closing brace marks the end of the control structure, and the last dependency is removed from the dependencies stack.

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 stack. If a program exit is found in a function declaration this function is added to the list of interesting functions with a note about a possible 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 without aborting the program run in advance.

5.3 Analysis

The analysis phase starts every time a PVF call is detected during the analysation of the token list. This has the advantage that the token list only needs to be parsed once. However the previously build model during the parsing step has to be complete in such a way, that the analysis of the current PVF will be correct.

5.3.1 Taint analysis

A function call is detected by the token **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. Findings 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 result. 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.

Listing 5.5: Example code with PVF `system()`

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

Listing 5.5 shows an example code which makes use of the PVF `system()` that executes system commands [3]. Once the PVF is detected the next step is to identify its configured significant parameters. The function `system()` executes only the first parameter (`$b`) while the second handles only the result (`$ret`). Therefore only the first parameter `$b` is traced backwards. Accordingly line 3 is found in the global variable list which assignes variable `$a` to variable `$b`. Again `$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 who can decide between a real vulnerability or a false positive. The output for listing 5.5 is shown in listing 5.6.

Listing 5.6: Scan result for listing 5.5

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

5.3.2 Intraprocedural and interprocedural analysis

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 5.7: PVF `exec()` is called within a user-defined function

```
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 ?>
```

When the PVF call on line 4 in listing 5.7 is detected, the parameter `$b` is traced backwards. It is detected that `$b` depends on a function parameter of the function declaration of `myexec()`.

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 user input is found, the call and the vulnerability is added to the output. This call may occur in another user-defined function so that interprocedural analysis is possible within chained function calls. The scan results for listing 5.7 are shown in listing 5.8 and listing 5.9.

Listing 5.8: Scan result 1 for listing 5.7: the original PVF is shown

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

Listing 5.9: Scan result 2 for listing 5.7: the function call that triggers the vulnerability is shown

```
9:    myexec($aa, $bb, $cc);
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.

For correct intraprocdural analysis also globaled variables have to be considered. They are detected by identifying variables **T_VARIABLE** and a previous token **T_GLOBAL**. The keyword `global` declares a variable to be used in global and not in local variable scope [16]. All variables declared as `global` in a function are traced within the local variable list of the function and the global variable list. Also all changes on a variable declared as `global` within a function stay after the function returns. Therefore the modifications are saved to a extra variable list which is extracted into the global variable list once the function has been called.

5.4 Web interface

RIPS can be completely controlled by a web interface. To start a scan a user simply has to provide a file or directory name, choose the vulnerability type and click scan. RIPS will only scan files with file extensions that have been configured. 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 option is important to detect unsufficient securings which can be hard to detect by static source code analysis.
- 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 and unclear 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 will lead to a lot of false positives.

All found PVF calls and their traces are shown syntax highlighted and devided in blocks to the user. The user can decide to show vulnerabilities in backward direction (as it was scanned from PVF to user input) or in forward direction (as a developer would read it). The syntax highlighting of the PHP code can be changed on the fly by choosing from 7 different stylesheets. The color schemes were manually adapted from Pastie (pastie.org) and integrated into RIPS own syntax highlighter. For each token name a CSS attribute defines a color so that own styles can be created easily. Tainted data is specially highlighted as well as found securing actions taken by the developer. An example scan result of RIPS version 0.32 can be seen in figure 5.1.

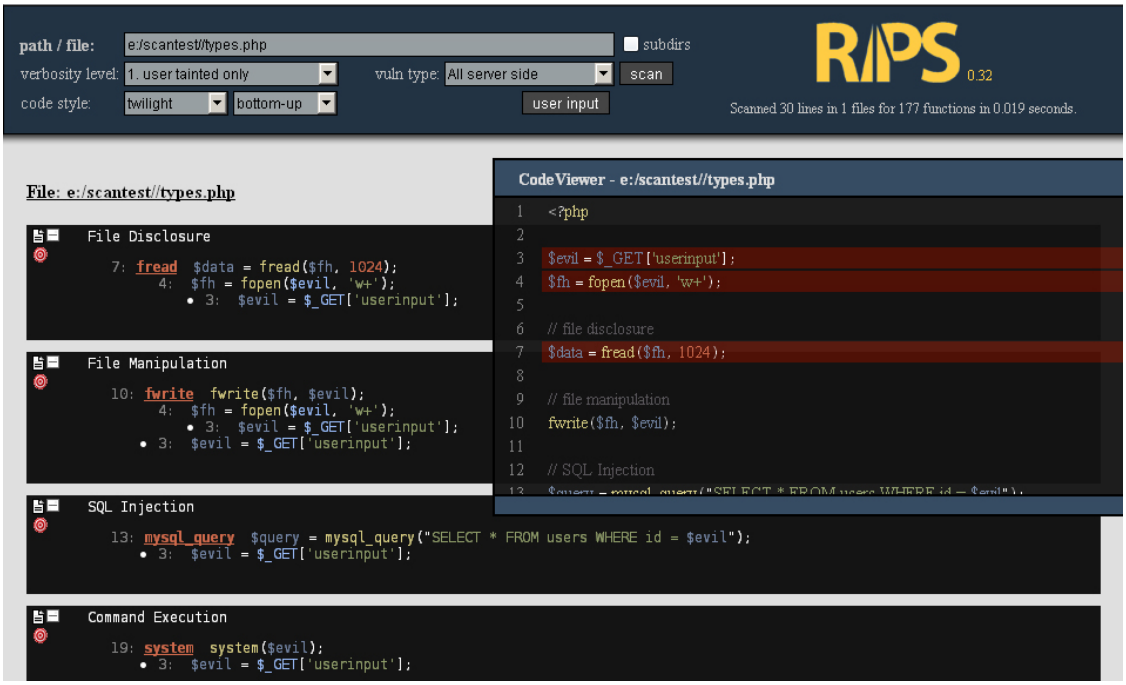


Figure 5.1: Web interface of RIPS 0.32 with scan result and code viewer

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. It also offers the ability to highlight specific variables by click to make an easy understanding of the code flow possible.

Another window can be opened for every detected vulnerability to quickly create a PHP curl exploit with a few clicks by hitting the target icon. Depending on the detected user input 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.

The latest version of RIPS also offers a description, example vulnerability, example exploitation and patch information to the user for every vulnerability found. This help can be opened by clicking on the help button assigned to each block.

For further investigations of complicated vulnerabilities, a list of user-defined functions, a list of program entry points (user input) and a list of all scanned files can be opened by the user which allows him to directly jump into the code by clicking on the list items. 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 findings in a user-defined function and the according call of this function is also possible.

5.5 Scan results

In order to test RIPS, the source code of a internship platform at the Ruhr-University Bochum was scanned. This platform is a virtual online banking webapplication written in PHP and was designed to teach several web application vulnerabilities during the internship. These vulnerabilities include 2 reflective XSS, 1 persistent XSS, 2 SQL Injection, 1 File Disclosure, 1 Code Evaluation, 1 Remote Command Execution vulnerability and 1 Business Logic Flaw.

At first RIPS was run with verbosity level 1 in order to find PVFs directly tainted with user input without detected securing. RIPS scanned 16870 lines in 84 files for 181 functions in 2.3 seconds.

	refl. XSS	pers. XSS	SQL Inj.	File Discl.	Code Eval	RCE	HRS	False Pos.	Time / seconds
1. user input tainted	1/2+1	0/1	2/2	1/1	1/1	1/1	+1	3	2.277
2. File/DB tainted +1	1/2+1	1/1	2/2	1/1	1/1	1/1	+1	19	2.359
3. secured +1,2	2/2+1	1/1	2/2	1/1	1/1	1/1	+1	151	2.707

Figure 5.2: Scan results with verbosity level 1, 2 and 3

Figure 5.2 shows the detected vulnerabilities and false positives (3). Suprisingly, a yet unknown HTTP Response Splitting and another unintended Cross-Site Scripting vulnerability was also detected.

One false positive was a SQL injection wich had been prevented by a regular expression and therefore could not be evaluated as correct sanitization by RIPS. Another two 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 was 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).

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 19 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 that was mistakenly detected as potential tainting source.

A significant false negative is the missing persistent XSS vulnerability. This one could only be detected by reviewing all 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. However this verbosity level generated 151 false positives because all secured PVF calls ended in the result. While this rate of false positives is not acceptable it also shows how many secured PVF calls was detected correctly in verbosity level 1.

As expected, the Business Logic Flaw could not be detected by taint analysis for PVF because it uses the applications logic and is not a taint-style vulnerability.

5.6 Limitations and future work

The main limitation of static source code analysis is the missing evaluation of dynamic strings that are build at runtime. In PHP the name of an 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 or statics. However, if the file name is constructed by calling functions, the name cannot 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 will address 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. Those can be easily hidden from static source code analysis by calling dynamic function names as shown in listing 5.10.

Listing 5.10: dynamic backdoor not found by RIPS

```
$a=base64_decode('c3lzdGVt');$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 RIPS does not fully support all PHP code semantics such as variable variables or aliases and does not consider variable types in all cases. This can lead to false positives and false negatives. Future work will address these problems, although they occur rarely.

6 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]. It uses control flow graphs to scan every possible combination of data flow. While Pixy is great in finding vulnerabilities with a false positive rate of 50%, it only supports XSS and SQL injection vulnerabilities. Both vulnerabilities are the most common vulnerabilities in PHP applications.

The goal of RIPS was to build a new approach of a static source code analyser written in PHP using the built-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. Also 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 findings 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 can fail for very 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. In example RIPS assumes that code structures are written line per line and that user-defined functions are declared before they are called. 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.

Additionally, there exist tools like Owasp Swaat [14] which are designed to find security flaws in more than one language but which only detect 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 cannot be confirmed. However, this method with an additional parameter trace can also been forced with RIPS by setting the verbosity level to 5.

Commercial static source code analysis products have been evaluated in an excellent paper by Nico L. de Poel [15].

7 Summary

In the past, a lot of open source webapplication scanners had been 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 analysers are available. RIPS is a new approach using the built-in PHP tokenizer functions. It is specialized for fast source code audits and can save a lot of time. Tests have shown that RIPS is capable of finding known and unknown security flaws in large PHP-based webapplications within seconds. 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. Found vulnerabilities can be easily tested by creating PHP curl exploits. However, due to the limitations of static source code analysis and some assumption on the programm code made by RIPS, 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. Also most of the wide spread PHP applications today rely on object oriented programming which is not fully supported by RIPS yet. Therefore RIPS should be seen as a tool that helps analysing PHP source code for security flaws but not as a ultimate security flaw finder.

Bibliography

[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

[15] Nico L. de Poel, *Automated Security Review of PHP Web Applications with Static Code Analysis*
http://scripties.fwn.eldoc.ub.rug.nl/FILES/scripties/Informatica/Master/2010/Poel.N.L.de./INF-MA-2010-N.L._de_Poel.pdf

[16] The PHP Group, *Variable scope*
http://php.net/manual/en/language.variables.scope.php