RIPS

A static source code analyser for vulnerabilities in PHP scripts

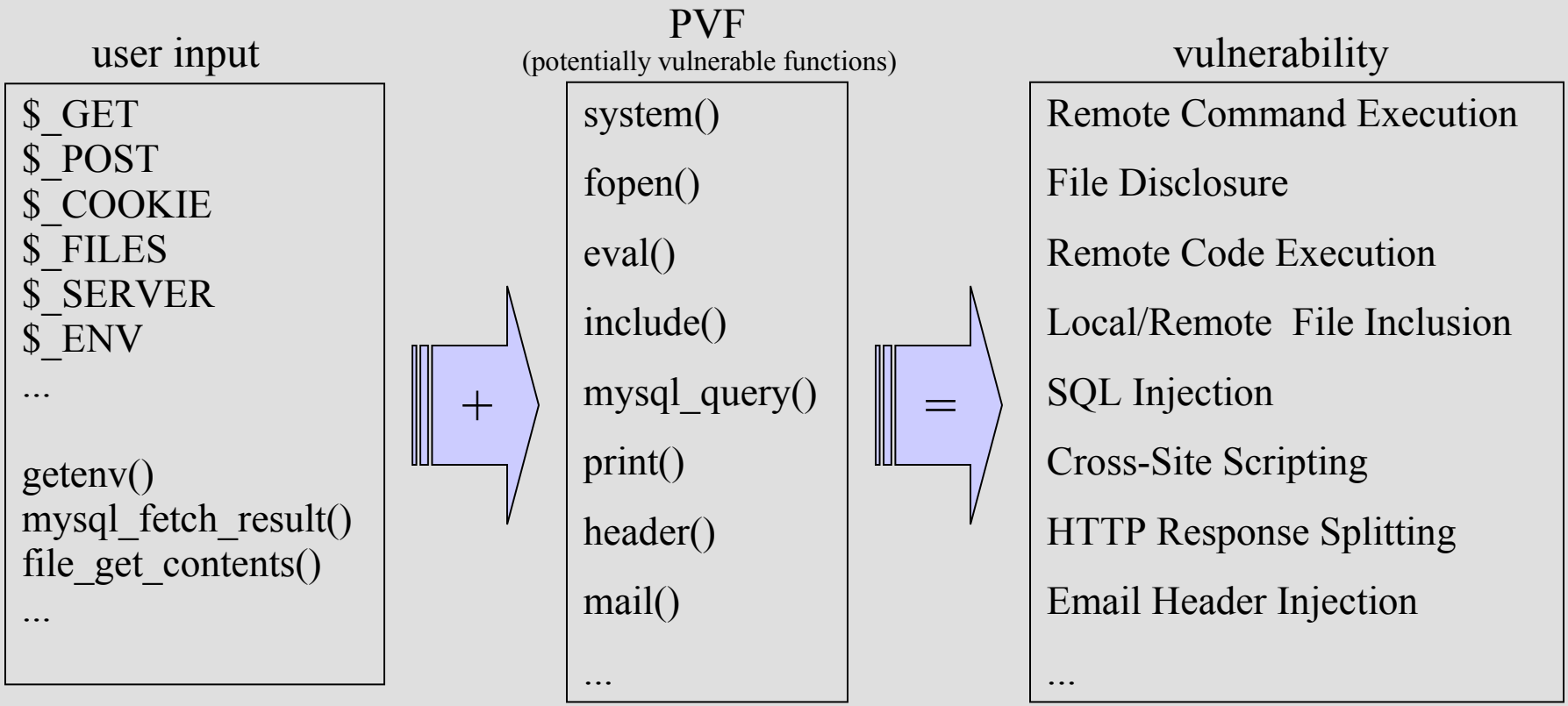
Johannes Dahse

1. Introduction

1.1 Motivation

- vulnerabilities 2.0 with web 2.0
- PHP is the most popular scripting language
- 30% of all vulnerabilities were PHP related in 2009
- finding vulnerabilities can be automated
  (minimizes time and costs)
- lots of free blackbox scanners available
- very few open source whitebox scanners (for PHP)
- Capture The Flag (CTF) contests

1.2 Basic Concept of PHP Vulnerabilities



1.3 The Concept of Taint Analysis

- identify PVF (`file_get_contents()`, `system()`)

```
1  <?php
2  $userfile = "users.txt";
3
4  $user = file_get_contents($userfile);
5  $pass = $_GET['pass'];
6
7  system("htpasswd -mb users/ ".$user." ".$pass);
8  ?>
```

- trace back parameters and check if they are „tainted“

1.3 The Concept of Taint Analysis

Not a vulnerability (file name cannot be influenced by a user):

```
4: file_get_contents  $user = file_get_contents($userfile);
     2:  $userfile = "users.txt";
```

Vulnerability detected (user can execute system commands):

```
7: system  system("htpasswd -mb users/ ".$user." ".$pass);
     4:  $user = file_get_contents($userfile);
   • 5:  $pass = $_GET['pass'];
```

/vuln.php?pass=foobar; nc –l –p 7777 –e /bin/bash

1.4 Static VS Dynamic Code Analysis

Static Source Code Analysis:

- parse source code
- lexical analysis (tokens)
    - interprocedual/flow-sensitive analysis
    - taint analysis

Dynamic Code Analysis:

- compile source code
- parse byte code
- taint analysis

1.4 Static VS Dynamic Code Analysis

Static Source Code Analysis:

- parse source code
- lexical analysis (tokens)
  - interprocedual/flow-sensitive analysis
  - taint analysis

Dynamic Code Analysis:

- compile source code
- parse byte code
- taint analysis

2. Implementation

2.1 Configuration

PVF	parameter	securing functions
system	1	escapeshellarg, escapeshellcmd
file_put_contents	1,2	
printf	0	htmlentities, htmlspecialchars
...		
array_walk_recursive	2	
preg_replace_callback	1,2	preg_quote

RIPS in its current state scans for 167 PVF

2.1 Configuration

global securing functions
intval
count
round
strlen
md5
base64_encode
...

user input
$_GET
$_POST
$_COOKIE
$_FILES
$_SERVER
$_ENV
...

file input
file_get_contents
zip_read
...

database input
mysql_fetch_array
mysql_fetch_row
...

2.2 Most apparent approach

- grep / search by regular expression for PVF:

```
$lines = file($file);
foreach($lines as $line)
{
   if(preg_match(' /exec\(.*\$/ ', $line))
              echo 'vulnerable: ' . $line;
}
```

- fail:

```
exec ($cmd); noexec($cmd); /* exec($cmd); */
$t='exec() and $var';  exec('./transfer $100');
```

2.2 Most apparent approach

- grep / search by regular expression for PVF:

```
$lines = file($file);
foreach($lines as $line)
{
    if(preg_match(' /exec\(.*\$/ ', $line))
                echo 'vulnerable: ' . $line;
}
```

- fail:

```
exec ($cmd); noexec($cmd); /* exec($cmd); */
$t='exec() and $var';  exec('./transfer $100');
```

2.2 The Tokenizer

- splits source code into tokens for correct analysis
- **`token_get_all`**`()` parses the given source string into PHP language tokens (using the Zend engine's lexical scanner)

```
array token_get_all(string $source)
```

- returns three element array or single character for each token

```
array(TOKEN_NAME, STRING, LINENR)
```

2.2 The Tokenizer

```
token_get_all():
```

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

```
array(
   array(T_OPEN_TAG, '<?php', 1),
   array(T_VARIABLE, '$cmd', 2),
   array(T_WHITESPACE, ' ', 2),
   '=',
   array(T_WHITESPACE, ' ', 2),
   array(T_VARIABLE, '$_GET', 2),
   '[',
   array(T_CONSTANT_ENCAPSED_STRING, 'cmd', 2),
   ']',
   ';',
   array(T_STRING, 'system', 3),
   '(',
   array(T_VARIABLE, '$cmd', 3),
   ')',
   ';',
   array(T_CLOSE_TAG, '?>', 4)
);
```

2.2 The Tokenizer

`token_get_all():`

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

delete insignificant tokens for correct analysis

```
array(
    array(T_OPEN_TAG, '<?php', 1),
    array(T_VARIABLE, '$cmd', 2),
    array(T_WHITESPACE, ' ', 2),
    '=',
    array(T_WHITESPACE, ' ', 2),
    array(T_VARIABLE, '$_GET', 2),
    '[',
    array(T_CONSTANT_ENCAPSED_STRING, 'cmd', 2),
    ']',
    ';',
    array(T_STRING, 'system', 3),
    '(',
    array(T_VARIABLE, '$cmd', 3),
    ')',
    ';',
    array(T_CLOSE_TAG, '?>', 4)
);
```

2.2 The Tokenizer

Fix token list:

```
1   if(isset($_GET['cmd']))
2       $cmd = $_GET['cmd'];
3   else
4       $cmd = '2010';
5   system('cal ' . $cmd);
```

2.2 The Tokenizer

Fix token list:

```
1   if(isset($_GET['cmd']))
2   {  $cmd = $_GET['cmd']; }
3   else
4   {  $cmd = '2010'; }
5   system('cal ' . $cmd);
```

Add braces for correct token analysis

2.3 Token Analysis

- loop through all tokens, detect connected language constructs

```
$tokens = fix_tokens( token_get_all($code) );

foreach($tokens as $token)
{
   list($token_name, $token_value, $line_nr) = $token;

   if($token_name === T_VARIABLE && ....
   if($token_name === T_STRING && ....
   if($token_name === T_FUNCTION && ....
   ...
}
```

2.3 Token Analysis (flow-sensitive)

curly braces	`if(condition) {...}`
T_FUNCTION	`function foo($a, $b) {...}`
T_RETURN	`function check($a) {return (int)$a;}`
T_INCLUDE	`include($BASE_DIR.'index.php');`
T_EXIT	`if(empty($a)) exit;`

2.3 Token Analysis

T_VARIABLE `global $text[] = 'hello';`

- identify variable declarations
- add to either local (in function) or global variable list
- add current program flow dependency

Variable	Declaration	Dependency
$m	`$m = $_GET['mode'];`	
$b	`$b+=$a;`	`if($m == 2)`
$c['name']	`$c['name'] = $b;`	
$d	`while($d=fopen($c['name'], 'r'))`	

2.3 Token Analysis (taint-style)

T_STRING `exec($a);`

- check if function in PVF list
- trace parameters with local or global variable list
  - fetch parameter from variable list
  - trace all other variables in variable declaration
  - detect securing
  - loop until declaration not found or tainted by user input
- if tainted and not secured:
  - output tree of traced parameters
  - add dependencies

2.3 Token Analysis (taint-style)

```
1   $default = 'sleep 1';
2   if(isset($_GET['cmd'])) {
3       $cmd = $_GET['cmd'];
4   } else {
5       $cmd = $default;
6   }
7   exec($cmd);
```

Registers			
in_func	0	**braces_open**	0

Variable List	
`$default = 'sleep 1';`	
`$cmd = $_GET['cmd'];`	`if`
`$cmd = $default;`	`else`

Dependency Stack

PVF Config
system, exec, ...

User Input Config
$_GET, $POST, ...

2.3 Token Analysis (interprocedual)

Vulnerability in function declaration detected:

```
<?php
  function myexec($a,$b,$c)
  {
      exec($b);
  }

  $aa = 'test';
  $bb = $_GET['cmd'];
  myexec($aa, $bb, $cc);
?>
```

PVF	param	securing functions
exec	1	escapeshellarg,...
...		
myexec	2	escapeshellarg,...

2.4 Webinterface

- choose verbosity level 1-5
- choose vulnerability type
- integrated code viewer (highlights vulnerable lines)
- mouse-over for user defined functions
- jumping between user defined function declarations and calls
- integrated exploit creator
- show list of entry points (user input)
- show list of user defined functions
- syntax highlighting with 7 different stylesheets

2.5 Results

- source code of virtual online banking internship platform
- 16870 lines in 84 files scanned

	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

- RIPS finds known and unknown security vulnerabilities
- missed flaws can be found with higher verbosity level (FP!)

2.6 Limitations & Future Work

- implementing a control flow graph (CFG)

```
$v = $_GET['v'];
if($a == $b)
   $v = 'hello';
system($v);
```

- automatic type casts

```
$vuln=$_GET['v']; $secure = $vuln + 1; exec($secure);
```

- object oriented programming only partially supported
- dynamic includes, function calls, variables

```
include(str_replace($BASE_DIR, '.', '') . $file);
$a = base64_decode('c3lzdGVt'); $a($_GET['v']);
$$b = $_GET['v'];
```

3. Summary

3. Summary

+ new approach of open source PHP vulnerability scanner written in PHP
+ fast, capable of finding known and unknown security flaws
+ vulnerabilities are easily traceable and exploitable

- RIPS makes assumptions on the program code
- some limitations regarding OOP, data types and data flow
- false positives / false negatives

→ manual review has to be made (verbosity level)

3. Summary

+ RIPS helps analysing PHP source code for security flaws

- RIPS is not (yet ;) an ultimate security flaw finder

RIPS was released during the Month of PHP Security:
http://www.php-security.org

It is open source (BSD License) and freely available at:
http://sourceforge.net/projects/rips-scanner/ Beta

Download! Scan!

Feedback is highly appreciated.

Questions ?

Demo ?

Thank you...

... all for you attention

... Dominik Birk for supervising