A Static Backward Taint Data Analysis Method for Detecting Web Application Vulnerabilities

Xuexiong Yan
Institute of Software Chinese Academy of Sciences
State Key Laboratory of Mathematical Engineering and Advanced Computing
Beijing, China
e-mail: yanxuexiong@sohu.com

Hengtai Ma
Institute of Software Chinese Academy of Sciences
Beijing, China
e-mail: hengtai@iscas.ac.cn

Qingxian Wang
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, China
e-mail: wqx2008@vip.sina.com

Abstract—**This paper addresses detecting taint-style vulnerabilities in PHP code. It extends classical taint-style model with an element called "cleans", which is used to specify sanitation routines. Based on the new model, a static backward taint data analysis method is proposed to detecting taint-style vulnerabilities. This method includes four key steps, first of which is collecting sinks and constructing contexts, the second is backward tracing variables during a basic block, the third is tracing variables between blocks, and the last is tracing variables crossing function call. A tool called POSE implements this method and testing results show that the method is valid for detecting taint-style web application vulnerabilities.**

Keywords-taint data analysis; PHP; web application; vulnerabilities

I. INTRODUCTION

Many web applications have been provided for online services in recent years, such as banking and retailing. But, security vulnerabilities are still the main problems for adopting web applications. According to the report of Risk Based Security [1], 2016 is the new all-time high with VulnDB reaching 15,000 vulnerabilities in which web application vulnerabilities accounted for 53.5%.

A natural alternative for web application vulnerabilities is to detect and then remove them. There are many types of web application vulnerabilities, such as XSS [2, 3], SQL injection [4, 5], CSRF [6]. But there is not a general model of these vulnerabilities. So, it is hard to present a solution to detect web application vulnerabilities of all types. Taint-style vulnerabilities are a special class of web application vulnerabilities [7, 8]. A vulnerability of this style means that tainted data from malicious users can cause security problems at vulnerable points in the program. XSS and SQL injection are the typical cases of taint-style vulnerabilities.

Taint data analysis is a major method for detecting taint-style vulnerabilities [9]. As for PHP code, a static taint data analysis method has been first explored in WebSSARI [10] to find out taint-style vulnerabilities. Since then, many other tools were implemented [8, 11-13] and have been used to find many web application vulnerabilities in PHP scripts.

This paper addresses detecting taint-style vulnerabilities in PHP code. It extends classical taint-style model with an element called "cleans", which is used to specify sanitation routines, such as htmlentities, and introduces a new static backward taint data analysis method, which is a path-sensitive, interprocedural and context-sensitive data flow analysis method, and presents a new framework to implement a tool called POSE to find out taint-style vulnerabilities automatically.

II. THE MODEL OF TAINT-STYLE VULNERABILITIES

There are many types of web application vulnerabilities and yet there is not a general model of them. So it is hard to design a general algorithm to find out web application vulnerabilities of all types.

A. The Classical Taint-Style Vulnerabilities Model

Taint-style vulnerabilities are a special class of web application vulnerabilities. A vulnerability of this type means that tainted data from malicious users can cause security problems at vulnerable points in the program. XSS and SQL injection are the typical cases of taint-style vulnerabilities.

Livshits et al. [7] presented a taint-style vulnerabilities model as a triple <sources, sinks, derivations>, the sources of which specify the ways the user to provide its data for the program, the sinks specify unsafe ways of the program to use user's data, and the derivations specify the ways of data propagating between objects in the program.

When using the classical taint-style vulnerabilities model to detecting vulnerabilities in PHP code, the element of sources specifies the entry points into the program, such as GET, POST and COOKIE etc., the element of sinks specifies the secure functions such as print, mysql_query etc., and the element of derivations is defined by data flow rules.

B. *The Extended Taint-Style Vulnerabilities Model*

The classical taint-style vulnerabilities model includes the basic elements of the taint-style vulnerabilities, but it doesn't include the sanitation routines, such as htmlentities, and that may result in false positives.

Fig. 1 shows an example with a sanitation routine htmlentities. According to the classic taint-style vulnerabilities model, the source is $_GET, the sink is the function print, and the data from the source will reach the sink, so, there is XSS vulnerability in the example of fig.1. But the routine htmllentities will sanitize the data from the source and then clean the XSS vulnerability.

```
1 <?php
2  $name=$_GET['name'];
3  $name=htmlentities($name);
4  $hello="Hello, ".$name;
5  print($hello);
6 ?>
```

Figure 1. An example with a sanitation routine

So, in order to express the taint-style vulnerabilities more precisely, the sanitation routines will be include in the extended model, which is expressed as four elements <sources, sinks, derivations, cleans>, and the first three elements of the extended model are just as the classical model and the fourth element specifies the sanitation routines which may be used to clean some taint-style vulnerabilities. Fig. 2 shows an example of the new model for XSS which is classic taint-style vulnerability. Sources include GET, POST and REQUEST. Sinks include all print functions such as echo and print. Cleans include sanitation functions such as htmlentities and htmlspecialchars.

```
xss_sinks=array(
  sources=>array(
   "GET",
   "POST",
   "REQUEST"),
  sinks=>array(
   "echo"=>array(0),
   "print"=>array(0)),
  cleans=>array(
   "htmlentities",
   "htmlspecialchars")
);
```

Figure 2. The extended model of XSS

The element of derivations is used to describe the rules for variable tracing. Update rules for variables tracing is showed in Fig. 3. There are two types of update rules, the first of which are rules for assignment and the second are rules for single function call. The expression $a|$b means adding the variable $a to the trace list and removing the variable $b from it.

1. Rules for assignment

$$\frac{\text{trace(\$a), \$a=\$b}}{\text{trace(\$b|\$a)}}$$

(1a)

$$\frac{\text{trace(\$a), \$a=\$b op \$c}}{\text{trace(\$b, \$c|\$a)}}$$

(1b)

$$\frac{\text{trace(\$a), \$a=func(\$b, \$c)}}{\text{trace(\$b, \$c|\$a)}}$$

(1c)

2. Rule for single function call

$$\frac{\text{trace(\$a), func(\&\$a, \$b)}}{\text{trace(\$b|\$a)}}$$

Figure 3. Update rules for taint data analysis.

III. The Static Backward Taint Data Analysis Method

The static backward taint data analysis method is used to analyze data flow among variables according to the extended model. The method is based on CFG (Control Flow Graph) and CG (Call Graph) and includes four key steps:

Step1: Collecting sinks and constructing contexts.
Step2: Backward tracing variables during a basic block.
Step3: Tracing variables between blocks.
Step4: Tracing variables crossing function call.

A. *Collecting Sinks and Constructing Contexts*

A sink is a sensitive function with its sensitive parameters which may be relevant to web application vulnerability. The algorithm of collecting sinks and constructing contexts is showed as Fig. 4.

```
function construct_contexts($codes){
  foreach($codes as $s){
   if(istype($s)==function call){
     if($info=is_sink($s)){
       $new_context=new context($s,$info);
       push_context($new_context);
     }
   }
  }
}
```

Figure 4. The algorithm of collecting sinks and constructing contexts

The algorithm checks every statement of PHP code, and if the statement comprises a function call which is among the sinks of a vulnerability model, then there is a sink. For example, when the algorithm checks the PHP code which is showed as fig.1, the statement of line 5 comprises function call print which is a defined in the sinks of the model of XSS, then a XSS sink is find out.

The algorithm creates a context for each sink. The main information of a context includes basic block number, line number, function name, tracing variables, all of which will be used for taint data analysis. The context of the XSS sink in Fig. 1 is showed as following:

- Basic block number: 1;
- Line number: 5;
- Function name: main;
- Tracing variables: $hello.

After collecting all sinks and constructing a context for each sink, the process of backward tracing variables will be started for each context.

B. *Backward Tracing Variables Algorithm*

The backward tracing variables algorithm for each context during a basic block is showed as fig.5. Owing to the algorithm is used for backward variables trace, all PHP codes have been reversed from the end to the start.

The algorithm handles four occasions when it traces variables during a basic block:

- Sanitation routine: If the statement comprises a sanitation routine (line 6), then the function call is_clean will return TRUE, and then the variable $var will be remove from the tracing variables queue.
- Deriving new variables: If a derivation rule can be used to derive some new variables (line 7), the old variable will be remove from and some new variables will be add to the tracing variables queue.
- Sources: If the variable is the source (line 9) ， then there is a vulnerability which will be reported to the user.
- Irrelevant: All variables that are irrelevant with a statement will be retained for subsequent variables tracing.

```
01 function tarce_var($codes,$context){
02  $vars=$context->vars;
03  foreach($codes as $s){
04    $new_vars=array();
05    foreach($vars as $var){
06      if(is_clean($s, $var)) continue;
07      else if($rule=select_rule($s, $var)){
08        $new_var=update_var($var, $rule);
09        if(is_source($new_var))
10         report_vul($new_var, $s, $context);
11        else put_array($new_var, $new_vars);
12     }
13     else put_array($var, $new_vars);
14   }
15  }
16 }
```

Figure 5. Backward tracing variables algorithm

As for the context of the XSS sink in Fig. 1, the algorithm gets the tracing variables which is $hello. When the algorithm reaches line 4 in Fig. 1, then the rule (1b) is selected to update the tracing variable to $name. When it reaches line 3 in Fig. 1, this statement comprises a sanitation routine htmlentities, then the variable $name is cleaned and will be removed from the tracing queue, and all tracing variables are checked and the process of tracing variables will be terminated.

When the backward tracing variables algorithm ends in the point of the start of a basic block, the tracing variables process will be shifted to some new contexts for a new block or a new function, otherwise it will be terminated.

C. *Tracing Variables between Blocks*

When tracing variables process shifts from a block to another block, some new contexts may be created and pushed to the context stack. The algorithm of creating new contexts between blocks is showed as Fig. 6.

The algorithm searches all front blocks according to the CFG (line 2) and then constructs new contexts for each front block according the following information:

- The block number of the front block;
- The last line number of the front block and the statement of this line;
- Tracing variables from the back block.

All new contexts will be pushed to the context stack, and the process of backward tracing variables will be started for these new contexts.

```
01 function shift_block($cfg, $block, $vars) {
02  $new_blocks=get_front($cfg, $block);
03  foreach($new_blocks as $front)
04  {
05    $s=get_last_line($front);
06    $info=context_info($vars, $front);
07    $new_context=new context($s, $info);
08    push_context($new_context);
09  }
10 }
```

Figure 6. The algorithm of creating new contexts between blocks

D. *Tracing Variables Crossing Function Call*

When the process of tracing variables crosses to a new function through a function call, some new contexts may be created and pushed to the context stack. The algorithm of creating new contexts crossing function call is showed as Fig. 7.

```
function shift_call($cg,$func,$vars,$cfg){
 $callees=get_callee($func,$cg);
 foreach($callees as $callee)
 {
   $s=get_point($callee,$func);
   $new_vars=var_map($vars,$func,$callee);
   $front=get_block($callee);
   $info=context_info($new_vars,$front);
   $new_context=new context($s,$info);
   push_context($new_context);
 }
}
```

Figure 7. The algorithm of creating new contexts crossing function call

The algorithm searches front block according to the CFG and the CG, and maps old tracing variables to new tracing variables by the function definition and its arguments.

IV. The Web Application Vulnerabilities Detecting Tool

We implement the static backward taint data analysis method in a tool named POSE (PHP cOde Static rEviewer). The framework of POSE is described as Fig. 8.

The file with PHP code is passed to the PHP preparing module and the outputs are abstract syntax tree (AST), control flow graph (CFG) and call graph (CG). The module of PHP code preparing is implemented based on PHP Parser [14].

The module of sinks searching will collect all sinks of the PHP code according to the CG and sink definition which is stored in a file. The module of path searching implements path searching algorithm and it will search all paths of a context. In order to avoid endless loop of the process of path searching, all circle CFG paths and function call relations will be excluded from the CFG and CG. The module of taint analysis implements backward tracing variables algorithm, and it will trace all variables along all paths from the sink to the start of the PHP code, and judge if there is taint-style vulnerability.

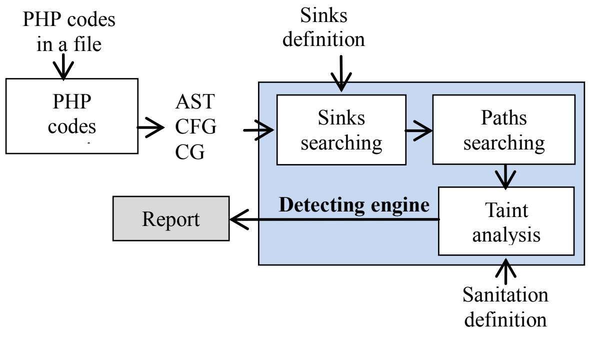

Figure 8. The framework of POSE

The POSE is used to test some open source PHP programs. Table I summarizes the results of our experiments.

TABLE I. Vulnerabilities Discovered by POSE

Programs	Vulnerabilities reported	Vulnerabilities confirmed	False positives
Miniblog-1.0.1	2	1	1
phpicalendar-2.4	6	2	4
OA2012	10	4	6
Students management system 2012	10	6	4
Totals	28	13	15

V. Conclusions

Taint-style vulnerabilities are a special class of web application vulnerabilities, and how to detect them is a research hotspot.

We extend the classical taint-style vulnerabilities model with an element called “cleans” which is used to specify the sanitation routines. Based on the extend model, we propose a static backward taint data analysis method for detecting web application vulnerabilities in PHP code, and this method includes four key steps, first of which is collecting sinks and constructing contexts, the second is backward tracing taint data during a basic block, the third is tracing variables between blocks, and the last is tracing variables crossing function call. A tool called POSE implements this method and testing result shows that the method is valid for detecting taint-style web application vulnerabilities.

There are still high false positives when using static method to detect web application vulnerabilities, and some AI solutions may be proposed to deal with this situation.

References

[1] Risk Based Security. lnerability QuickView 2016 Year End[R]. January 2017. https://pages.riskbasedsecurity.com/2016-ye-nuln-quickview.

[2] Cross Site Scripting (XSS) Flaws[S]. The OWASP Foundation, updated 2013. https://www.owasp.org/index.php/Cross_Site_Scripting_Flaw.

[3] A.Klein. DOM Based Cross Site Scripting or XSS of the Third Kind[R]. Web Application Security Consortium, 2005.

[4] C. Anley. Advanced SQL Injection In SQL Server Applications[R]. White paper, Next Generation Security Software Ltd., 2002.

[5] WGJ Halfond，J Viegas，A Orso. A Classification of SQL Injection Attacks and Countermeasures[C].In Proceeding of International Symposium on Secure Software Engineering , 2006.

[6] B.Adam, J.Collin, C.M.John C. Robust Defenses for Cross-Site Request Forgery[C]. In Proceeding of ACM Conference on Computer and Communications Security(CCS), 2008.

[7] V. B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In Proceedings of the 14th Usenix Security Symposium, 2005.

[8] N.Jovanovic, C.Kruegel, E.Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities(short paper). In Proceeding of IEEE Symposium on Security and Privacy, 2006.

[9] M.S.Lam, M.Martin and V.B.Livshits. Securing web application with static and dynamic information flow tracking. In Proceedings of 2008 ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation, 2008.

[10] Y.W.Huang, F.Yu, C.Hang, C.H.Tsai, D.T.Lee, S,Y,Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 13th International World Wide Web Conference, 2004.

[11] Y.Xie, A.Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In Proceeding of USENIX Security Symposium, 2006.

[12] J.Dahse, T.Holz. Simulation of Build-in PHP Features for Precise Static Code Analysis. In processing of 2014 the network and distributed system security, San Diego, CA, 2014.

[13] P.Nunes, J.Fonseca, M.Vieira. phpSAFE: A Security Analysis Tool for OOP Web Application Plugins. In Proceeding of 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2015.

[14] PHP-Parser. https://github.com/nikic/PHP-Parser.