SSOF - Discovering vulnerabilities in PHP web applications – Project 22/23

1. Introduction

In this project, we tackle the problem of detecting web vulnerabilities statically in PHP programs. These vulnerabilities may occur when user input isn't properly parsed and sanitized, becoming tainted and interfering with the normal flow of the program. This way, the attacker can feed some malicious input to the program in order to compromise it, giving the attacker control to retrieve important information, like user passwords, data, etc. This issue is becoming increasingly important since in 2021 Injection attacks where in OWASP top 3.

2. Usage

Our tool receives as input an AST representation of the slice of the program to analyze and a list of the vulnerability patterns to consider. Then it traverses through the tree and tracks which variables/functions are tainted and/or sanitized. It is written in PHP, and once it as run, it prints the results in the screen. Example:

```
php tool.php <slice>.json <patterns>.json > output/<slice>.output.json
```

3. Implementation

We represented the AST utilizing the data structures of the PHP-Parser provided in the project description. All structures are represented as nodes of the AST. The most important nodes are the ***Variable*** and ***FuncCall***. Both sinks and sources are represented by these nodes (the exception being the ***ArrayDimFetch*** and ***Echo*** which can be sources and sinks, respectively). As we traverse the tree, we keep track of the relevant information regarding the nodes. Henceforth, we will refer to these pieces of information as the following tags: Initialized, Tainted, Source, Sink, Sanitizer, Sanitized. To represent the change of context when we enter an ***If_/ElseIf_/Else_*** or ***While_*** nodes we created a class named "State" that encapsules all the needed information (We will explain more further down the report).

Nodes	Important Information/Security classes
Node\Expr\Variable	Initialized, Tainted, Source, Sink
Node\Expr\FuncCall	Tainted, Source, Sink, Sanitized, Sanitizer
Node\Expr\Assign & Node\Expr\AssignOp	Initialized, Tainted, Source, Sink, Sanitized
Node\Stmt\If_ & Node\Stmt\Else_& Node\Stmt\ElseIf_	Pushes/Pops a new state
Node\Stmt\While_	Pushes/Pops a new state
Node\Stmt\Expression_	Initialized, Tainted, Source, Sink, Sanitized
Node\Stmt\Echo_	Tainted, Sink
Node\Stmt\Break_ & Node\Stmt\Continue_	Alters While state

Table 1 – Nodes and relevant tags (Initialized, Tainted, Source, Sink)

The AST is traversed using a DFS algorithm (the child nodes are visited first). We use the ***NodeTraverser*** class and its method ***traverse*** to traverse the tree.

To distinguish between the different contexts in ***If's*** and ***While's*** we created a class called "State", which contains the nodes within the body of the ***If's*** and ***While's***. This class extends ***NodeVisitorAbstract*** to be used as a ***Visitor***.

To traverse all the nodes, we created another class called "Visitor" that also extends ***NodeVisitorAbstract.*** This class works as a traverser where every time a node that changes the context of the program is encountered, it creates a new object "State" and pushes it to the top of a stack where we keep all the current states. After visiting all the nodes in that state, we pop the stack. When a new "State" is created, all information in previous state is forwarded (E.g., when we enter an ***If***, if a ***Variable*** is tainted, that ***Variable*** remains tainted in the ***If*** state). Traversal

Visitor.php

When we enter a node:

```
If (node is While_ or If_ or Else_ or ElseIf_): Stack.push(new State);
```

When we leave a node:

```
If (node is While_ or If_ or Else_ or ElseIf_): Stack.pop();
```

State.php

When we leave a node:

```
If (node is Variable and is Source): Mark as Tainted;
If (node is FuncCall):
    If (node is Source): Mark as Tainted;
    If (node is Sanitizer):
        Traverse(args):
            If (arg is Variable and is Tainted): Mark as Sanitized;
            If (arg is FuncCall and is Tainted): Mark as Sanitized;
    Traverse(args):
        If (arg is Variable and is (Tainted or not Initialized): Mark as Tainted;
        If (arg is FuncCall and is Tainted): Mark as Tainted;
If (node is Assign/AssignOp):
    Traverse (expr):
        If (node is Variable and is (Tainted or not Initialized): Mark as Tainted;
    If (node is FuncCall and is Tainted): Mark as Tainted;
```

When we enter a node:

```
If (node is Assign): Mark as Initialized;
```

We run this algorithm for every vulnerability, and once it is finished, we check for every ***Sink*** if it is ***Tainted***, and if the ***Sources*** that tainted it are ***Sanitized***.

To check for implicit vulnerabilities, we made just a small adjustment to the previous algorithm (Basically every non initialized Variable in the guard becomes tainted, as well as the variables in the body):

Visitor.php

When we enter a node:

```
If ((node is While_ or If_ or Else_ or ElseIf_) and implicit):
        Forall (args in guard): Mark as Tainted;
        Stack.push(new State);
```

State.php

Before visiting any node:

```
If (node is Variable): Mark as Tainted;
```

4. Test and Evaluate

To test the soundness, precision and scope of our tool, we ran the provided tests by the faculty, the tests in the "Common-Tests" repository as well as some local tests made by us. Here are some of the tests we created and their rationale:

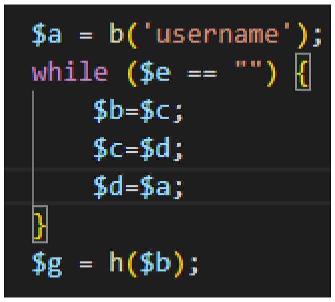
Test 1 – Analyze basic flow of tainted variables.
Test 2 – Analyze flows inside If's and Else's blocks.
Test 3 – Analyze flows inside While loops where it needs more than 2 iterations to get all vulnerabilities.
Test 4 – Analyze flows where a variable is initialized in every state.
Test 5 – Analyze seemingly implicit flows.

5. Critical Analysis

	False positives	False negatives
Information flows	yes	Yes
Input Sanitization	yes	No

Imprecisions in the information flows:

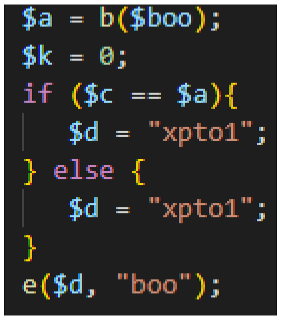
False negatives – In while loops we only traverse their body twice, not recognizing flows where variables become tainted in the n iterations prior (Test 3). Example:



```
$a = b('username');
while ($e == "") {
    $b=$c;
    $c=$d;
    $d=$a;
}
$g = h($b);
```

If the while loop only runs twice and the ***Source*** is ***b***, when ***$b*** is checked in ***Sink h()***, it will not report the vulnerability, leaving it open for exploitation.

False positives – When our tool searches for implicit flows, if the supposed ***Tainted*** variable inside the body of an ***If*** or ***While*** has the same value, it reports the vulnerability when it doesn't exist (Test 5).

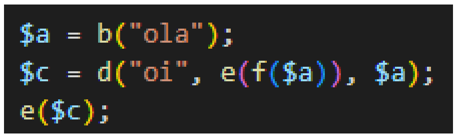


```
$a = b($boo);
$k = 0;
if ($c == $a){
    $d = "xpto1";
} else {
    $d = "xpto1";
}
e($d, "boo");
```

In this example, ***$d*** is not ***Tainted***, since its final value is the same ("xpto1"), however our tool treated it as tainted when it encounters ***Sink e()***. This could be avoided if we checked the final value of ***$d*** for all flows. If it was the same, then no vulnerability is reported.

Imprecise endorsement of input sanitization:

False positives – Although keeping track of sanitized and unsanitized flows, our tool does not output unsanitized flows correctly (test 3a-expr-func-calls of the provided tests).



```
$a = b("ola");
$c = d("oi", e(f($a)), $a);
e($c);
```

In the example, when ***b*** as the ***Source***, ***e()*** as the ***Sink*** and ***f*** and ***d*** as ***Sanitizers***, when analyzing sink e(), it should not report any unsanitized flows. However, our tool reports it as "yes" in the ***unsanitized flows*** field. To avoid this false positive, we could've added a rule in our algorithm that marked as sanitized a ***var*** in an ***Assign*** node if its ***expr*** is sanitized.

We could improve precision in our tool by evaluating the conditions of the guards of If's and While's, making sure it doesn't pursue impossible flows, improving performance as well.

6. Related work

The implementation of our tool has some similarities of *WebSSARi* [1], as we also generate some sort of a control flow graph (CFG) (the "State" class) and a symbol table (ST) (Arrays that track the tags of nodes), however we move through the AST and analyze the results in only one "big" traverse with "small" ones within. [2] and [4] also utilize a similar approach, however, [2] focuses more on SQL injections and [4] works in a more similar way to our project by generating the CFG in a way where the states alter when they "enter new functions and other modules" and doing tainted analysis as well [4],[5]. Relating sanitization, [3] works in a very similar way: A set of specified sanitizers are given to check if the input has been sanitized. Other approach we observed is to analyze the PHP built-in functions, to find "more security vulnerabilities with higher accuracy" [6].

7. Conclusion

Our tool is good for most direct flow analysis, distinguishing between different flows of information by forwarding all important information to the next state. To improve our tool, we could enlarge the scope of it, providing constructs to for loops, for example. To improve soundness, we could traverse the body of a while loop as many times as the number of ***Assign*** nodes it has.

References:

[1] Yao-Wen Huang et. al., Securing web application code by static analysis and runtime protection, WWW'04

[2] Gary Wassermann and Zhendong Su, Sound and Precise Analysis of Web Applications for Injection Vulnerabilities, PLDI'07

[3] Davide Balzarotti et. al., Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications, S&P'08

[4] Ibéria Medeiros et. al., Automatic Detection and Correction of Web Application Vulnerabilities using Data Mining to Predict False Positives, WWW'14

[5] N. Jovanovic, C. Kruegel and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," 2006 IEEE Symposium on Security and Privacy (S&P'06), Berkeley/Oakland, CA, USA, 2006, pp. 6 pp.-263, doi: 10.1109/SP.2006.29.

[6] DAHSE, Johannes; HOLZ, Thorsten. Simulation of Built-in PHP Features for Precise Static Code Analysis. In: *NDSS*. 2014. p. 23-26.