Discovering vulnerabilities in PHP web applications

Software Security

Duarte Bento 92456 Sara Marques 93342 Sara Graça 102100

Introduction

Web applications have become an integral part of modern society, providing various services and functionality to users. Many of these functionalities require user interaction, which usually leads to an increased availability of vulnerabilities. Mis-handled user input can cause security problems such as Cross-Site Scripting (XSS), SQL injection, and General Script Injection.. To better analyze and investigate vulnerabilities in existing PHP code , we have implemented a static analysis tool for taint analysis which identifies illegal information flows.

Usage

The tool is divided into two separate components, a PHP parser, and a PHP analyser which receives as arguments the previously parsed AST (slice.json) and file with the specific vulnerability patterns to consider (patterns.json). To run both tools the only requirement is to have python 3 installed. After meeting the requirements, the tool is called in the command line by:

```
$ python3 php-analyser.py slice.json patterns.json
```

Following, the tool checks for any pattern match and generates an output containing the list of vulnerabilities, with each source, sink, sanitized flows, and if there are any unsanitized flows.

Implementation

We took the json encoded AST given as input and parsed it, creating an updated tree with the corresponding nodes defined in the python class Nodes. Parsing the AST was done top to bottom, the same as the order of statements in the PHP code. We called this updated tree a *decorated AST.*

Each Node was designed to support a visitor design pattern. The Static_Analysis class implements a visitor design pattern and is responsible for applying the taint analysis logic.

Our tool traverses the updated AST by recursion. First, the root node, Program_Node, applies the taint analysis visitor to its child Stmt_Expression_Nodes in a depth-first manner, from a statement expression node down to the most basic children, evaluating the node and progressively returning the value to the parent node in the decorated AST.

```
<?php   $a = "";   ?>
```

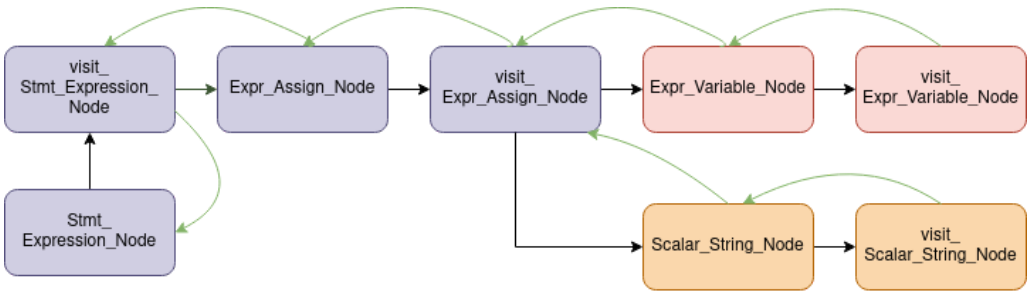

Figure 1. Traversal algorithm for sample PHP code

The Static_Analysis class is instantiated for each given pattern with the respective sources, sinks, and sanitizers. During the tree traversal code units are recorded and evaluated according to the current

pattern, meaning that when a sensitive sink is reached, a possibly relevant information flow, according to the pattern definition, is recorded.

Each Function and Variable present in the PHP code is part of the class Taint_Unit. This class saves the state of the Variables (*pure, sanitized, or tainted*) and the Functions (*sanitizer or source*) and keeps records of the flow of information to those variables or other units involved for the case of functions In Static_Analysis class, we create an instance of a Taint_Unit for every variable or function visited. Each unit is kept in a table called *taint_table* used to later audit the unit state (e.g., variable set to tainted).

When a sink is reached, a flow is added to the list of flows of the Taint_Units involved.

Our tool handles conditions by having a stack of active taint_tables and creating a new taint table for each new block, keeping different versions of possibly the same taint_units - variables. We also have to consider an empty context for the *else* case even if there is none (only the *if* statement) to account for the possibility of not entering the *if* block. We need to have this consideration because at the end of an if/else block we merge the taint tables for the new blocks with the current table (global in case of a single if, previous if's context in case of nested ifs). The merge will consider the variables with the worst taint level and update their value in the current table, along with the flows that led to their current value.

To analyze loops, our tool evaluates the loop node and its statements until there are no changes to the current taint_table. This mitigates cases where variables are tainted at the end of a loop's body and used at the beginning, this way considering all the possible interactions of the loop.

Regarding implicit flows, we have to confirm if the pattern being analyzed is explicitly marked as an implicit flow. Considering this, for tainted (or sanitized) variables that are used in conditional statements (*if* and *while*), we save all the flows that have led to a variable being tainted (implicit_sources stack) and append those sources and sanitizers to other variables assigned inside the condition block, since these variables implicitly depend on the values of the variables checked in the condition.

Test and Evaluate

To test the correctness and robustness of our tool, we used the list of tests provided and our own created test. These tests' objective was to check particular cases at each type of node, gradually increasing the complexity of the code. The first few were related to variable assignments, the following about function calls, next we tested conditions and after that loops. The last few tests defined an implicit flow vulnerability.

Critical Analysis

Imprecisions	Answer	Examples

Are all illegal information flows captured by the adopted technique? (false negatives)	Yes. Our tool is *sound,* detecting all illegal, so we consider all possible branches. In the example ["b"] is a source, ["f"] sanitizer and ["$g"] sink, we will consider both branches of the if (case where condition does not evaluate to true) and detect an possible unsanitized flow, *false positive* in this case.
	
	The soundness of our tool prevents possible vulnerabilities but makes it too restrictive, rejecting possibly safe programs.
		$a = b('username');
		$e = "false"
		if ($e == "false") {
		  $a = "init"
		}
		$g = $a;

Are there flows that are unduly reported? (false	Yes. Considering source:["f"] and sink:["e"], and that during execution, the code does enter at the first and	if ($c > 0) { $a = b();

positives)	the second *if* condition, then our tool returns the supposed vulnerability (f->e). However, it will return two additional vulnerabilities, corresponding to the cases when the code does not enter at the first *if* condition (a->e) and when it does not enter at the *else* condition (c->e). The tool is designed to be *sound (*detect all illegal*)* but not *complete* so false positives will occur..
	
	Dynamic analysis could be employed to correctly detect which conditional flows should be followed and prevent false positives like this one.
		if ($c < 3) {
		$a = f($a);
		}
		else {
		$c = d($a);
		}
		}
		e($a, $c);

Are there sanitization functions that could be ill-used and do not properly sanitize the input? (false negatives)	No. Our tool has no knowledge of what the sanitization function is performing so it will flag every time a tainted variable is used, even if sanitized. This design decision was based on similar work **[1]**, which presented the following example. When using a mis-configured sanitization function it could still allow the attacker to introduce a specially crafted payload. For example the use of *strip_tags()* is not enough to prevent attribute injection, htmlspecialchars() should be used instead.	Code that causes a false positive: $s = "Hello\n"; $x = str_replace("\n", ' ', $s); echo $x; Code that could cause a false negative: $s = 'a'; $x = str_replace('a', $_GET['x'], $s); echo $x;

Are all possible sanitization procedures detected by the tool? (false positives)	Yes but only the ones defined in the pattern. Even the use of multiple sanitizers is recorded, like in the example where sanitizer:["c", "e"], source:["b"], and sink:["z"]. A pattern will be recorded with both sanitizers applied to the $a variable.
	Additionally, procedures that update a variable internally will not be detected as sanitizing a variable, only detected for explicit assignments.
	
	To mitigate this example in particular, there should be a sanitization function definition in the patterns that identify the correct usage of the specific function, for example if function should only be used in conditions to check the value is sanitized (like PHP *filter_input()*) or that the return value is the sanitized string.
		Case with both sanitizers, both detected:
		$a = b("ola");;
		$d = e(c($a));
		z($d);
		
		Case for internal sanitizer, not detected:
		$a = b("ola");
		c($a);
		z($a);

Since considering all branches and possibilities present some scaling issues and become too restrictive, resulting in numerous false positives, it is necessary to apply additional techniques.

One possibility could be the use of an AI machine learning model. Given the source code and a list of error reports, this can be achieved by reducing the source code to a subset of itself to isolate the code locations that are related to the error report. Next, label the error reports by manually examining the code. Lastly, use machine learning techniques on the reduced code to discover code structures correlated with false positive error reports and to learn a classifier that can filter out future false positive error reports. **[2]**

The false positives can be improved by adding dynamic analysis to our tool, mitigating the cases where we consider all the branches and consequently reducing the returned flows.

During the dynamic analysis is tested the effectiveness of the sanitization routines applied between a source and a sink.

One application of this could be executing the corresponding sanitization routines using several attack strings as input. Then, using a decision function to evaluate whether the attack strings were successfully reduced to non-malicious values.

If the testing process confirms that the sanitization is ineffective between a source and a sink, it provides the path along which malicious input can reach the sink and a sample attack string that successfully exploits the vulnerability, providing enough information to fix the problem manually. **[1]**

Related work

Web applications are often targeted for security attacks because they contain many vulnerabilities, for example lack of input validation. Since PHP is one of the most widely-used web scripting languages there are already many techniques for solving problems like the one in this project.

The technique proposed by Huang et al. [3] has a different approach in finding vulnerabilities because it focuses on identifying faulty sanitization processes. The second technique by Wassermann et al. [4] has an initial step similar to our tool doing taint analysis to find potential vulnerabilities and besides that uses machine learning for data mining to predict the existence of false positives. The third technique proposed by Balzarotti et. al. [1] identifies sections of code considered vulnerable and uses runtime guards to secure the application. The last technique by Medeiros et. al. [6] is also more advanced than our tool since it doesn't require written specifications for queries and patterns and considers as attacks queries where the user input changes the syntactic structure of the query to prevent SQL injection.

Besides the 4 papers referred to on the project statement we found other ones that have similarities to our tool. For example, the technique presented by Barth et al. [7] analyzes for each procedure in a program which variables may be modified, used or are possibly preserved by a call which is similar to how we analyze a slice of code saving all variables and respective modifications on a structure. Another similar technique is presented by Jovanovic et al. [8] uses flow-sensitive, interprocedural and context-sensitive dataflow analysis to discover vulnerable points in PHP applications.

Conclusion

To conclude our tool is able to detect most of the vulnerabilities in small slices of a PHP program.

Additionally, our tool can be enhanced to evaluate the actual values of variables during runtime, and to follow the flow of data and execution in the program. This approach is known as dynamic analysis, which allows the tool to only analyze what is actually happening during the execution of the program, rather than just the potential flow of data based on the code structure. This can give a more accurate view of the security risks and vulnerabilities in the program.

References

[1] D. Balzarotti *et al*., "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," *2008 IEEE Symposium on Security and Privacy (sp 2008)*, Oakland, CA, USA, 2008, pp. 387-401, doi: 10.1109/SP.2008.22.

[2] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017). Association for Computing Machinery, New York, NY, USA, 35–42. https://doi.org/10.1145/3088525.3088675

[3] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing web application code by static analysis and runtime protection. In Proceedings of the 13th international conference on World Wide Web (WWW '04). Association for Computing Machinery, New York, NY, USA, 40–52. https://doi.org/10.1145/988672.988679

[4] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). Association for Computing Machinery, New York, NY, USA, 32–41. https://doi.org/10.1145/1250734.1250739

[6] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In Proceedings of the 23rd international conference on World wide web (WWW '14). Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/2566486.2568024

[7] Jeffrey M. Barth. 1978. A practical interprocedural data flow analysis algorithm. Commun. ACM 21, 9 (Sept. 1978), 724–736. https://doi.org/10.1145/359588.359596

[8] 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.