TÉCNICO LISBOA

Software Security 2022/23

Project Report – Group 15

Gonçalo Guerreiro - ist195581
Vasco Correia - ist194188
Vasco Sebastião - ist195686

I. Introduction

Programs that accept input from the users may encode potentially dangerous information flows due to malicious inputs, this means user input (low variables) can interfere with high variables(sinks) of the program, which may affect integrity and confidentiality, by not respecting noninterference, resulting in vulnerabilities. The goal of this project is to statically analyze code and detect these vulnerabilities by means of taint and input sanitization analysis, in the PHP language.

II. Usage

The tool receives as input two JSON files, the first containing an AST representing a program slice that is to be analyzed and the second containing a list of vulnerability patterns to contemplate. The tool has a CLI interface and produces as output, either to stdout or to a file depending on the arguments given, a list of potential vulnerabilities encoded in the program slice that fit the patterns described.

III. Implementation

The AST nodes are represented using the class *'AstNode',* there is no distinction between nodes except for the *'_type'* attribute, which stores an enumeration specifying the type of node. We added an enumeration for every node type involved in the mandatory constructs and a few others, program node, statement elseif node, print node, and unary expressions node. When building the internal AST we make some simplifications that don't affect the correctness of the tool. Loop and if statement conditions are transformed into assignment expressions where the variable being assigned is anonymous. Elseif statements are transformed into an else statement with an if statement inside. Echo and print statements are transformed into statement expressions with a function call inside(of a function named echo and print respectively).

We classify every expression as one of two types, tainted or untainted. All information flows are legal except for untainted to tainted.

We start by visiting the program node we created (*TypeChecker.visit*), it contains all other statements, for each statement we visit it(*TypeChecker.visit_statement*). *visit_statement* for each different type of statement calls a more specific visit, there are three, *TypeChecker.visit_if_statement, TypeChecker.visit_while_statement* and *TypeChecker.visit_expression*. Visit_expression in turn, for each different type of expression also calls a more specific visit.

During visitation we build directed graphs to represent the information flows in the program, where the edges go in the opposite direction of the information flow. We build a different graph for each decision in the program (conditional and loop statements), one for when the condition is true and another one for when it is false. Afterwards we analyze the graphs constructed and if there is a path from a sink to a source, this means the sink has been tainted by the source and a vulnerability is reported. If in that path there is a sanitizer, the flow has been sanitized, and this is also reported.

The transformation of conditional and loop statements into assignment expressions where the variable being assigned is anonymous turns what would be an implicit flow into an explicit one and that is how the tool detects implicit flows. A drawback of this approach is that it results in our program always outputting vulnerabilities caused by implicit flows even when this is not asked in the input.

IV. Test and Evaluate

To test the correctness and robustness of the tool we used the 15 examples provided, of which it detects the correct vulnerabilities for all of them except 5a, where it unduly reports more vulnerabilities due to how we detect implicit flows as mentioned in the previous section.
In addition to these, we also created 5 more tests, which were all correctly analyzed by the tool: the first one tests the detection of sanitized and implicit flows, the second one tests the 'else' statement, the third one tests the 'elseif' statement, the fourth one tests the 'break' statement (the vulnerability occurs after the break so it is never reached and, as such, it should not be reported) and the last one tests the 'continue' statement (in the same way as the 'break' test).

V. Critical Analysis

Question	Answer	Code

Are all illegal information flows captured by the adopted technique? (false negatives)	no	`$x = true;`
		`if ($x) {`
		`  $a = $b;`
		`}`
		`elseif ($x) {`
		`  $a = $d;`
		`}`
		`else {`
		`  $a = $e;`
		`}`

Are there flows that are unduly reported? (false positives)	yes	`$a = b('username');`
		`while ($e == "") {`
		`  $c = d($a);`
		`  $e = f($c);`
		`  $a = $e;`
		`}`
		`$g = h($a);`

Are there sanitization functions that could be ill-used and do not properly sanitize the input? (false negatives)	no	
Are all possible sanitization procedures detected by the tool? (false positives)	yes	

Illegal information flows not being captured can lead to exploits, as shown in the snippet of code in the table, where the 'else' block has a vulnerability from $e to $a, since the $e variable is not

initialized and so should be treated as a source, but the tool does not detect it. Sanitization is detected if during our traversal of the graph we find a sanitizer, since we only report vulnerabilities if during the mentioned traversal we find a source, sanitization is only reported in correct cases.

The false positives given by the tool are implicit flows that get caught even when such behavior is not specified in the input, as we have explained above. This is due to the implementation of the tool and could certainly be improved but it would mean a big code refactoring. As mentioned before, sanitization procedures are collected during traversal of the graph, and reported when sources are found, therefore, if there is a sanitizer in the information flow between a source and a sink, it is reported.

After generating the AST and doing taint analysis, we could implement another phase like the 'data mining' described in *Medeiros, I. et al, 2014* in order to predict false positives and improve the precision of the tool. By using machine learning techniques, the tool can use data collected from humans (label samples of code as vulnerable or not) to analyze the code and classify the vulnerabilities found in the first step as false positives or not.

VI. Related Work

Progpilot is very similar in concept to what we have developed, although obviously much more complete and precise. It also compares programs to known patterns of vulnerabilities containing sources, sinks, sanitizers and validators(which are not included in ours), in a more real and specific way.

(Huang, Y. W. et al., 2004, p. 45) proposes another solution much similar to our own: "We considered all values submitted by a user as tainted, and checked their propagation against a set of predefined trust policies.".

(Medeiros, I. et al, 2014) also have a very similar approach but it goes much further than ours, doing taint analysis on an AST (which is all we do) at the beginning and then moving to data mining to identify false positives and specific code correction.

(Wassermann, G. et al, 2007), propose a sound, automated static analysis algorithm that is grammar-based, modeling string values as context free grammars (CFGs), as such, this is a very different approach to ours.

(Balzarotti, D. et al, 2008) proposes a similar solution where variables that store user input are marked as tainted, once they are sanitized they are set to untainted, and when a tainted variable is used in a sink, a vulnerability is reported. This tool is "sound with respect to the supported language features", but it might produce false positives.

Perl's Taint mode does something very similar for the Perl language, when it is enabled, every variable is monitored by Perl to check whether it is tainted or not and doesn't allow data from outside an application to affect anything external to it, it identifies sources of taint, much like our tool and also provides a way to untaint data, although a drawback is that "Perl's tainted mode tracks information flow at runtime, resulting in expensive overhead." (Huang, Y. W. et al., 2004, p. 49).

VII. Conclusion

The tool is quite reliable in detecting both explicit and implicit vulnerabilities, but it has some limitations, as in some cases it reports false positives. Our tool could be improved by implementing the remaining language specification, using a database to store known vulnerability patterns, use more advanced techniques for conditional statements and loops (e.g. if a loop condition that always evaluates to false we don't need to analyze the inside of the loop), and by using Machine Learning techniques to improve both accuracy and precision.

VIII. References

Huang, Y. W., Yu, F., Hang, C., Tsai, C. H., Lee, D. T., & Kuo, S. Y. (2004, May). Securing web application code by static analysis and runtime protection. In Proceedings of the 13th international conference on World Wide Web (pp. 40-52).

Wassermann, G., & Su, Z. (2007, June). Sound and precise analysis of web applications for injection vulnerabilities. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (pp. 32-41).

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (2008, May). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)* (pp. 387-401). IEEE.

Medeiros, I., Neves, N. F., & Correia, M. (2014, April). 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 (pp. 63-74).

END