Project Report

1 - Introduction

The understanding of how vulnerabilities can be identified and addressed is of great use, not only to professionals working in the security area but also to the common programmer, due to the potential harm that malicious agents can cause by exploiting such vulnerabilities. The usage of tools to help solve these problems can often be very important.

In this report, we will go over our static web vulnerability analyser developed using the Python programming language. The analyser is capable of identifying a set of vulnerabilities given a slice of PHP code that we want to analyse and a pattern file that specifies for a given type of vulnerability its possible sources, sanitizers and sinks. In this report, we will show how the tool can be used, we will provide an overview of the analyser's capabilities as well as discuss the development process, including the challenges faced and solutions implemented.

2 - Usage

The tool can be used by running the following command in a terminal:

```
python ./php-analyser.py slice.json pattern.json
```

A JSON file will then be generated in a folder called 'output'. This file will contain a list of the potential vulnerabilities found in the given slice, according to the patterns specified in the pattern file.

3 - Implementation

- What data structures did you use to internally represent the AST?

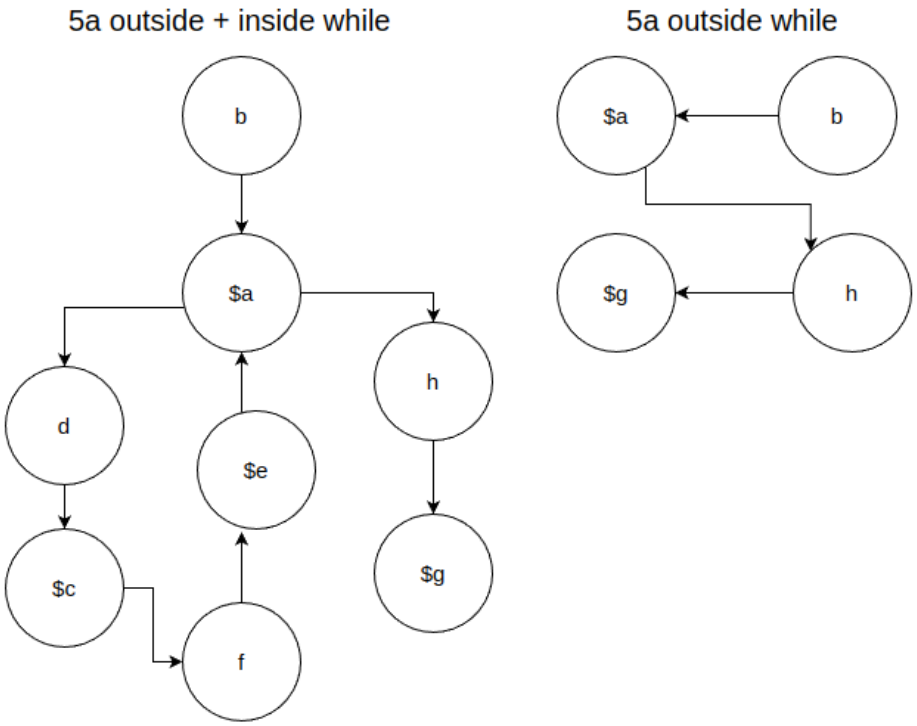
  We interpreted this problem as a graph problem. The data structures used were mainly dictionaries and lists.

- How is the AST traversed?

  We use two ways for AST traverse:

  First way, before conditions, like, if's, while's, we built a recursive function that for each node(nodeType) check what is being done, for example if is a function call, expression assign. We use recursion in order to solve a problem of having functions inside functions a(b(c($d))).

In our second way we treat conditions and for that we needed to cover every possible block of code, for exempla in a while what is inside can or cannot be done and so we need to execute what is inside and out independently.



- How are basic vulnerabilities detected?

  We though in one algorithm (set of instructions)

  1- Traverse ASTs.

  - build graph + ast's initialized + uninitialized nodes list

  2- Process patterns → [['vulnerability'], ['sources'], ['sanitizers], ['sinks'], 'implicit'].

  3- Identify new sources, nodes in uninitialized list for a specific ast.

  4- Make all permutations between source's/new source's, sanitizers, and sinks.

  - for example: [['A'], [b, c], [d], [e, f]]

    [['A'], [b], [d], [e]]
    [['A'], [b], [d], [f]]
    [['A'], [c], [d], [e]]
    [['A'], [c], [d], [f]]
    about empty sanitizers*

  5- For a given source and sink we find every possible path, also with repeated nodes.

6- Check if for a source sink's path a sanitizer node exist, if not exist but the path does (about empty sanitizers*) we add a vulnerability in this format.
[[['A'], 'c', 'e', 'yes', []]]
7- We put in right format to send to json.

- How (and if) are implicit vulnerabilities detected?

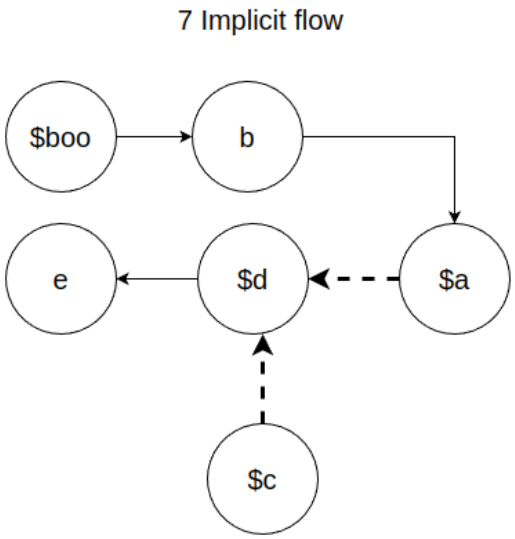
  Due to time-associated issues we could not build a totally implementation for tests 7, 8 and 9. However, we could manage to pass tests 7 and 8 out of delivery time.

  Make a note about the way we are detecting implicit flows it's not correct, because we use our program behaviour to pass only the tests.

  Our solution is not based on above solution of generate new ast's but check for vars/functions before/after and inside conditions.

  The algorithm we thought was:

1. extract vars inside conditions
2. extract vars before condition
3. get vars inside condition's block
4. get vars that are in condition but are not initialized, because they will become new sources
5. build new arches with new sources
6. check if is condition and implicit flow is set as true
7. make all the process for generate output



4 - Test and Evaluate:

We only tested for the 1a-6b tests that were made available.

5 - Critical Analysis

- Identification of imprecisions:

  False positives:

  - The tool was not designed to evaluate the result of conditions or assess the value of variables or functions and therefore it is incapable of knowing whether certain portions of code are executed or not. One example might be a while loop where the condition is always True. This will mean that any lines of code after the while loop are not to be considered in the analysis, but our tool will still analyse them.

- Our tool might not be able to always identify all uninitialized variables. This could happen, for example, because functions might alter the value of a variable without the tool knowing about it.

False negatives:

- Implicit vulnerabilities are not being identified due to the fact that the developed tool did not have that feature implemented.

- Certain PHP functions, expressions and others are not being parsed by the tool which may lead to unidentified vulnerabilities.

- Understanding of imprecisions: Can the identified false negatives lead to exploits?
  - The absence of detection of implicit vulnerabilities in flows can lead to malicious agents gaining access to private information.
  - Because some PHP functions, literals, statements and others are not being considered by the tool, the use of these in the slices can lead to exploits.

- Improving precision: What other program analysis techniques could be incorporated into your tool in order to improve precision?
  - In order to know whether certain blocks of code will be executed or not, the developed tool needs a certain level of knowledge about the value of variables and functions, so that it can know, for example, whether a while loop's code should be considered for analysis.
  - More specification about what might make a variable uninitialized or better parsing of the PHP slice will reduce the number of variables that are currently being incorrectly considered as initialized.
  - For the last example, improving the marking of variables as uninitialized would improve the precision of our tool, but it would inevitably lead to a decrease in performance as more calculations would be needed.

6 - Related work

Several different tools address similar problems in different ways: Saner[1] uses static as well as dynamic analysis techniques; WAP[2] uses global analysis (i.e., the analysis can involve several files) and machine learning; WebSSARI[3] inserts runtime guards in potentially insecure sections of code, securing a piece of code even without programmer intervention; and some tools are more focused on a particular type of vulnerability (i.e., injection vulnerabilities[4]).

Another tool, Pixy[5], though aimed at XSS vulnerabilities, uses similar techniques as the ones used in this project.

7 – Conclusion

Our tool is good at analysing sections of PHP code and detecting vulnerabilities that aren't related to implicit flows. Our tool still struggles with detecting vulnerabilities involving implicit, so further work would be needed on that front.

Beyond that, we think that upgrading our tool to have the capacity to analyse entire files or even multiple files at once could be interesting as we think that could lead to a decrease in the number of false positives.

References

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

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

[3] Yao-Wen Huang et. al., Securing Web Application Code by Static Analysis and Runtime Protection, WWW'04.

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

[5] N. Jovanovic, C. Kruegel and E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities (short paper), S&P'06.

[6] Y. Xie and A. Aiken, Static detection of security vulnerabilities in scripting languages, *Proceedings of the 15th USENIX Security Symposium*.