PHP Vulnerabilities Static Analysis Detector

João FIgueiredo 75741

Miguel Amaral 78865

Tiago Vicente 79620

1. Introduction

Securing on the web is an important topic on today’s web. As such we implemented a tool that does static analysis (tainted analysis) of php code.

Our approach is based on the detecting a set of vulnerabilities detected a-priori by other people, like[3].

It was developed for the class Software Security on IST.

2. Tool’s Architecture

The project consists of two main modules TrueAnalyzer and PatternsIdentifier, developed in PHP. Moreover there is also used an external PHP-Parser produced by github user nikic[1].

TrueAnalyzer is class that takes a file (a php piece of code), navigates the AST tree produced by the parser and processes the nodes with the information needed to send to the PatternsIdentifier module, which will then check for known vulnerability signatures, that were previously loaded from a file (patterns.txt includes the information contain on the table in [3]), being a default file provided along with the tool.

TA=TrueAnalyzer; PId = PatternsIdentifier;

2.1. TrueAnalyzer

To create the parse tree we used a parser created by nikic[1] that takes a piece of code and produces the AST tree. The parser only works if the piece of code is php syntactically correct, otherwise it will generate an error. We process each node accordingly to what it is supposed to know and be.

The two main sources of vulnerabilities are assignments and funcalls (normal funcalls, class funcalls and echo statements which are separated from normal funcalls by the parser).

For the assignments we process the right part of the assignment, process it and create a data structure that will be recognized by the PId. We then attached the left part of the assignment, the variable that we want to attribute being vulnerable or not based on the contents of it is right part. In the format:

($var, array($var1, $var2)),
where $var1/$var2 have the format : array($varName, $varType)

For the funcalls we process the arguments, and generate a data structure for them as well. Here we decided that we don’t need to have a context of all the arguments at once so we separate them and send them one by one. In the format:
($funcallName, $varName, $varType)

2.2. PatternsIdentifier

This module receives information concerned to assignments and function calls.

Regarding assignments there are three types. Variables, result of a function call and an arrayFetch, such as “$_GET[‘input’]”. PId module assumes that vulnerabilities entry points, i.e. user input, are always on the latter format. On the other hand it assumes that sanitizations and sink points are always the call of a function. If any one of this assumptions fails so will the module. Additionally function calls can have three types of arguments, the result of another funcall, a variable and again an arrayFetch.

The module operates in a way that classifies seen variables according to three states, tainted, sanitized and irrelevant. The former being variables that are not directly controlled by the user nor depend on any that is. One simple example would be an index used in a while cycle.

Whenever there is an assign, PId will match the assigned value against a blacklist (known tainted variables and entry points) and against a white list (patched variables and sanitization function calls). If there is a match the variable will then be added to the black or white list respectively.

Whenever there is a function call, PId will search the name of the function against a blacklist of sink points, if there is any match PId will then check if arguments are tainted or not, either by checking the variables and the entryPoint black list. Additionally it also checks the sanitization functions whitelist. Consequently if there is any vulnerability (either patched or not), it will be detected and registered so that later it is shown to the user when the whole slice was examined.

2.3 Examples

Our tool given the following slice of code:

```
<?php
        $nis=$_POST['nis'];
        $query="SELECT *FROM siswa WHERE nis='$nis' '$xd'";
        $q=mysql_query($query,$koneksi);
        $query=mysql_escape_string($query);
        $q=mysql_query($query,$koneksi);
 ?>
```

will produce the following output:

```
There is a vulnerability: SQL Injection
The variable: query has the entryPoint: _POST
The sink point is mysql_query

There is a vulnerability: SQL Injection
The variable: query has the entryPoint: _POST
The sink point would be: mysql_query, fortunately was sanitized with
mysql_escape_string
```

However if there is no identified vulnerability the output of the program will be a message indicating so. “Your code has no vulnerabilities detected”.

3. Tool Limitations

The way that TA is designed has it is problems. If the parser generates a special node for a problematic function or an unknown node, then the TA will not know how to process it. Although it is a problem it is very specific and the program can be easily extended to those nodes given the information of the parse tree.

3.1 False Positives

On the matter of false positives our program is as sound as the patterns that are loaded in runtime. So as long as the patterns are always correct, as in the sink point used with said entry point is always a vulnerability except when the sanitization is applied, all vulnerabilities detected are in fact a real vulnerability.

3.2 False Negatives

On the other hand, and contrarily to the work done in [2] if there is a function call that does not sanitize the input, we simple discard the return value, as if the output of said function breaks the flow of information. This will of course generate false negatives in functions such as sprintf, that copies its arguments directly to the output.

Furthermore if the programmer defines a function that has a sink point inside depending on one of its arguments, if somewhere else in the code that function is called with a tainted variable the tool will not detect it as a vulnerability, however if an entry point is created and used inside said function it will be detected. As illustrated by the following code:

```php
<?php
	function bad_function ( $possible_entry_point )  {
		sink_point( $possible_entry_point );   //Not detected
		$bad = $_EntryPoint[‘door’];
		sink_point( $Bad );                     //Detected
```

```
        }
        bad_function ( $_GET['door']);                //False Negative
?>
```

Finally many nodes are not processed, because we only considered the more common ones, in order to reduce the complexity of the project. For example classes are one node not considered. Another problem not considered is the scope of the variables, the program considers all variables to be at the same scope.

3.3 How to improve tool

First we could create a tree of tainted functions. What this would allow us to do is make a flow of information, which consequently would allow the tool to detect when a user creates a function that uses a sink point with a variable dependent on an argument.

Another way to improve would be to treat all the nodes generated by the parser instead of only the more common ones.

4. State of The Art

Another way to improve our project would be to be able to detect if something is vulnerability besides a text file indicating so. For that we would need, like the state of the art uses [2], an automatic learning algorithm for vulnerability classification.

Even though that is the best approach, we do not think it is adequate for a school project like this one.

We could also improve our project adding a feature to autocorrect the vulnerable sink points by, for example, adding sanitization right before it. For that we would need to alter our project slightly to be able to handle storing the lines where the sink points happen and just use the appropriate sanitize function on the line before.

5. Conclusion

As shown our implementation is able to decide if a piece of code is vulnerable or not and if it has been sanitized.

Also, our PatternsIdentifier module is language independent, so given a parser that would use its API, built for any other language along with a text file stating the vulnerabilities patterns of said language it would still be able to identify said vulnerabilities.

6. References

[1] https://github.com/nikic/PHP-Parser
[2] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.432.7100&rep=rep1&type=pdf
[3] http://awap.sourceforge.net/