# Discovering vulnerabilities in PHP web applications

Group 12

| | |
|---|---|
| António Monteiro | 80990 |
| Bernardo Cordeiro | 78778 |
| João Santos | 81083 |

Instituto Superior Técnico

November 20, 2017

**Abstract**

Security remains one of the majors issues regarding web applications. Our aim is to study how vulnerabilities in PHP code can be detected statically by means of taint and input validation analysis, in order to achieve an in-depth understanding of a security problem.

## 1 Introduction

Efforts for keeping online data and services safe are made every day. One of the major issues regarding web applications is security. In this assignment we propose a tool for identifying data flow integrity violations which are not subjected to proper input validation.

Input validation is a big issue when discussion software security, and a large part of problems related to security issues converge to a single point of exposure, the user input, which may lead to illegal or corrupt information flows, thus compromising data integrity or even confidentiality.

The goal for this work is to build a tool capable of statically detecting vulnerabilities in PHP source code, inspecting data flows and properly validating data input controlled by users, which are very low integrity entities.

## 2 Tool Description

Our tool is written in Java, which we believe to be a very powerful and complete programming language. Besides the default Java libraries we are including open-source libraries in our work, that have proven to be stable and useful, such as Gson, a Java serialization/deserialization library to convert Java Objects into JSON and back.

This analyser is composed by three parts:

- the vulnerabilities patterns parser, that parses files with the structure of Listing 1. The first line is the vulnerability name, the following line represents the entry point (usually the user input), the third line represents

Listing 1: Example of a vulnerability pattern file

```
SQL injection
_GET,_POST,_COOKIE,_REQUEST
mysql_escape_string,mysql_real_escape_string,mysql_real_escape_string
mysql_query,mysql_unbuffered_query,mysql_db_query
```

the validation/sanitizion functions that anull this potential threat, and finally the last line contains the sensitive sinks, the group of functions that if called with improper input will generate illegal/unexpected information flows.

- the PHP Abstract Syntax Tree (AST) parser, that parses JSON tree files that correspond to PHP AST[1].
- the analyser, where the slices analysis is done.

## 2.1 Imprecisions

- The patterns are loaded from a file. If the patterns are incomplete, not enough to detect a vulnerability, the tool will not detect it;
- Node types are not complete, we focus on the slices provided;
- It can produce false negatives and false positives or both.

## 2.2 Examples

The following examples are slices of PHP code, the legend corresponds to the output given by the tool.

Listing 2: This program is vulnerable in function call $mysql_query$

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

Listing 3: This program is vulnerable in function call $mysql_query$

```
<?php
$nis=$_POST['nis'];
$query1="SELECT *FROM siswa WHERE nis='";
$query2="$nis'";
$query=$query1 . $query2;
$q=mysql_query($query,$koneksi);
?>
```

[1]https://github.com/glayzzle/php-parser/blob/master/docs/AST.md

Listing 4: This program is vulnerable in function call echo

```
<?php
echo $_POST['username'];
?>
```

## 3 Discussion

Tools that perform static analysis are complex and would, ideally, find security flaws with a high degree of confidence. However, given the imprecisions refereed and discussed, considerable guarantees about the absence of vulnerabilities can not be provided. The fact that static analyses may produces false positives is also a down side of static analysis.

False negatives can also be often produced, one of the reasons this happens is because information flows are not followed when entering new functions.

### 3.1 Imprecise tracking of information flows

Standard slicing techniques do not preserve the confidentiality and integrity of information. The tool does not follow information flows when entering new functions, so false negatives can be produced because we have no way to tell if the output is tainted. This can be exploited by attackers, a false negative would be reported even if the output of a function was dangerous (See Listing 5).

Given that we analyze program slices by means of a taint analysis, user input is always marked as tainted. The user may also produce his own sanitization functions. So, there will be flows that are unduly reported.

Listing 5: Example of a slice of code that would produce a false negative even if the output of getUid was dangerous.

```
$q = mysql_query("SELECT u FROM db WHERE uid='getUid("pw")'");
```

### 3.2 Imprecise endorsement of input validation

Input data validation is one of the most important steps to guarantee safety in software. So, this must carried out with caution. Sanitization should always be in place, and should be done before any call to a sensitive sink. Also, when building programs, sanitization functions already known should be used instead of creating new sanitization procedures, because the ones that are known are reliable and fully tested. Even so, the programmer may choose to write his own sanitization functions, witch the tool will not detect.

The tool do not detect all possible validation procedures. There are issues, such as string manipulation and validation of user input. Both can sanitize data but the tool will not detect them. As referred, the programmer may also write his own sanitization functions, witch the tool will not detect.

## 3.3 Improvements proposal

To turn this tool into a more precise one, we would need to analyse more than the information flow of the program.

To improve the results given by the tool we could think in changing the approach. Switching from a static to a dynamic analysis. This would give more tracing power to the tool, the tool would be able to follow information flows when entering new functions. The number of false positives and false negatives would be reduced.

We could also try to check the sanitization efficiency using both static and dynamic analyses. When found a sanitization function we would check if it was appropriate given the context and the vulnerability being treated. We could try to predict false positives using machine learning, too.

We would suffer a loss of efficiency with this changes.

## 3.4 State of the art

In the last few years web applications have evolved from simple static pages to very thoughtful dynamic structures. As more complex it gets, more important is to guarantee security and more problems appear.

There are some tools that statically analyze code via taint analysis, such as Pixy[2], SaferPHP[3] and WAP[4]. To do so, an abstract syntax tree (AST) is produced and used to search variables and mark all entry points as tainted. The analyses is done following the tainted variables and check if it passes through a sanitization function or go through a sensitive sink.

In software security to identify vulnerabilities in the code are two types of approaches to take: the static, which is used and discussed in this report, and the dynamic. The static approach consists in the attempt to highlight possible vulnerabilities within the source code by using techniques such as Taint Analysis and Data Flow Analysis. The dynamic approach, done in runtime, is based on following the accepted data as it traverses the source code of the application and analyzes its use in order to identify security problems.

# 4 Conclusion

Static analysis can be unsound (produce false negatives) and/or incomplete (produce false positives). In order to get more guarantees from the tool, improvements, such as follow information flows, should be made. Although static analysis tools can be improved it is important to keep in mind that the tool will still not be full reliable.

[2]https://github.com/oliverklee/pixy
[3]https://github.com/caltechlibrary/safer-php
[4]http://awap.sourceforge.net

## References

[1] Y. Huang et al. *"Securing web application code by static analysis and runtime protection"*, ICWWW 2004.

[2] G. Wassermann and Z. Su. *"Sound and Precise Analysis of Web Applications for Injection Vulnerabilities"*, PLDI 2007.

[3] I. Medeiros et al. *“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, April 2014.

[4] D. Balzarotti et. al. *"Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications"*, S&P 2008.

[5] V. Mendonça, F. Soares, A. Vincenzi and C.Rodrigues *Static Analysis Techniques and Tools: A Systematic Mapping Study*, ICSEA 2013.

[6] The Open Web Application Security Project (OWASP)