Discovering vulnerabilities in PHP web applications

Group 29
Diogo Pereira (58122) — Rodrigo Lourenço (76133) — Bruno Leitão (81785)

Introduction

We built a tool that conservatively analyses information flow in a given slice of a PHP program. Broadly, it works is as follows: we traverse the abstract syntax tree (AST) of the program while keeping track of what variables have *tainted* values, that is, controlled by a possible attacker. If a tainted value reaches a vulnerable point, called *sink*, we report a vulnerability. The initial set of tainted variables are called *entry points*. Tainted values can become *sanitized* or *downgraded* using *sanitization* or *validation* functions.

Design

The tool, written in Python, takes as input an AST in JavaScript object notation (JSON), which is deserialized using the `json` module of the Python standard library, and a list of patterns. A pattern consists of a name, a comma-separated list of entry points, a comma-separated list of validation functions, and a comma-separated list of sensitive sinks.

The order in which the nodes of the AST are visited depends on the semantics of each node, and represents the order in which code is actually executed. For instance, in a for-statement, first the initializer is analysed, then the test, the body, and a loop of increment-test-body.

When assigning, the left-hand side becomes tainted if the right side is also tainted. Binary operations with at least one tainted value produce tainted values. Function calls produce tainted values if any argument is tainted, unless they are validation/downgrading functions, in which case they always return an untainted value.

Control structures require particular care. Consider the if/else structure in **Slice 8**, for example. The if-branch taints a variable while the else-branch untaints it. An analyzer that simply visits both branches sequentially will be unsound: the variable will end up labelled as untainted, while there is clearly a potential vulnerability if the if-branch is taken. The solution to achieve soundness is to taint the least upper bound (LUB) of the variables that may be tainted after the structure is executed. To find this LUB, we visit both branches in turn, while assuming that the other branch was not visited. Then, the results of the two visits are combined by tainting the set union of the variables tainted in either of the branches.

An additional challenge presents itself when considering loop structures. Typically a block of statements can be analyzed simply by analyzing each statement in the same order that they appear. This is reasonable since that is the order in which they are executed. However, a loop introduces a back edge in the control flow graph of the program. Information can flow along this back edge, meaning that later statements may affect preceding statements in the same block. An example of this type of “backward” flow can be seen in **Slice 10**. Our solution involves processing the entire loop body as many times as the number of statements in the body, since if there's a tainted value, it will propagate to at least one other statement, and it can do so at most as many times as there are statements.

Try-catch also requires some consideration, since the control-flow may jump from any statement into one of the catch-clauses. We ignore this fact entirely and assume it runs to the end of the block (as we will see below, this makes the tool unsound in the presence of these exception handling structures). After simulating the try-clause, we simulate each catch-clause independently, and then merge all environments.

One final, but important, design aspect is that whenever an assignment is executed in a block that is conditionally executed based on the value of an expression involving a tainted variable, the variable assigned to is tainted, even if the value being assigned is not, since we consider the whole environment to be low-integrity.

Examples

The tool is called from the console as follows, where `ast.json` is the AST in JSON:

```
./analyzer.py ast.json
```

The output for a program that has no vulnerabilities is:

```
   $q = mysql_query("SELECT * FROM t WHERE x='test'");

Outputs:
   No vulnerabilities found.
```

This slice shows the behavior of the tool when a literal is used on a query.

When a possible vulnerability is found the tool prints a warning string followed by the name of the vulnerability found:

```
  $q =  mysql_query($_GET['x']);

Outputs:
  WARNING: found possible vulnerability: SQL injection (mysql_query)
```

This piece of code shows a vulnerability because it uses the variable `x` without sanitizing its data.

When a possible vulnerability is found but a sanitization function is used to validate tainted data, the tool still prints a message with the vulnerability name and the validation function used like in the example below:

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

Outputs:
   No SQL injection (mysql_query) vulnerability due to endorsers:
mysql_escape_string
```

False negative! This slice shows how the tool handles an entry point where the data is validated by the `mysql_escape_string` function and is assumed to be safe.

Guarantees and limitations

The mechanism we have implemented is neither sound nor complete when considering the entirety of the PHP language and all possible vulnerabilities. However, for the given vulnerability patterns and within the restricted subset of the language that is used in the project slices (variable assignments, if-else structures, while loops, binary operations and function calls), we believe the tool to be sound, in that it will reject any program where information may flow from a sensitive source to a sensitive sink without being downgraded by a sanitization function.

One limitation of the tool is that it assumes that only variables may be sensitive sources and, similarly, that only functions (or language constructs such as *echo*) may be sensitive sinks. This simplification was motivated by the patterns we collected, where it was always the case. While we are not are not aware of any concrete examples (possibly due to our unfamiliarity with the PHP language), it is certainly conceivable that some function may exist that takes high-integrity input and returns low-integrity output, or that some variable could act as a sensitive sink. Our tool would be unsound and frankly inadequate to detect such vulnerabilities.

Continuing on the subject of vulnerability patterns, we opted to follow those given by Medeiros et al. (2014), but it should be noted that these patterns are not complete, which leads to some false positives: for example, the function *strlen* returns a number, thus its output never includes any special characters that could lead to a SQL injection.

For the most part, and again, considering solely the restricted subset of the language, our tool is very conservative, since we chose to eliminate false negatives at the expense of a higher rate of false positives.

Turning now to the full language, there are many PHP features that we either do not handle at all or do not handle soundly. To begin with, we do not support references at all. To do so would require keeping track of aliases for each variable, and when tainting a variable, also tainting all aliases of that variable. Try-catch blocks are supported, however the analysis is not sound. For example, our analyzer would not find a problem with the following slice, because it assumes that the entire try block is executed, and so that *$x* is sanitized:

```
$x = $_GET[‘x'];
try{
    throw Exception();
    $x = mysql_escape_string($x);
} catch { … }
mysql_query($x);
```

One way of solving this problem would be to assume that try blocks are always low-integrity, and taint any assignment inside them. However, this would be extremely conservative and lead to a very high number of false positives to the point where we would risk making the tool practically unusable: it would be impossible, for example, to sanitize a variable inside a try block (and still pass validation). In general, static analysis of exception flows is hard because control flow can jump at almost any time to a different environment.

Related work and possible improvements

There are several methods to statically control information flow in programs. In this project we developed a taint analysis mechanism following Medeiros et al. (2014), but other notable ones include the type-checking approach of Sabelfeld and Myers (2003), or the use of theorem provers as in Darvas et al. (2005).

One interesting taint analysis tool that is in use in the industry is IBM Security AppScan Source. The design of the scanner, ANDROMEDA, is explored in Tripp et al. (2013). One of its advantages is that it is capable of incremental analysis: when the programmer makes a small change to the code, the scanner can determine what subset of the program needs to be re-analyzed, instead of re-scanning the whole program. This allows for much quicker feedback to the developer: in as little as a couple of seconds, he can be warned if the code change may introduce a vulnerability.

As we described in the previous section, our tool is quite conservative. One way of improving it by reducing the prevalence of false positives without sacrificing soundness would be to implement some features of dynamic analysis.

On Huang et al. (2004), WebSSARI extends the current language type checking and inserts runtime guards in sections of code that seem unsound or unsafe.

Balzarotti et al (2008) also uses static analysis and dynamic analysis combined. By characterizing the sanitization process they are able to identify where the sanitization / validation is either incomplete or incorrect. Then by using another tool that is able to reconstruct the code handling the sanitization the programmer can use malicious inputs to test and identify faulty procedures.

The hybrid static/dynamic analysis approach allows better precision than solely static methods, but obviously this comes at the cost of additional runtime overhead.

References

Y. Huang et al. "Securing web application code by static analysis and runtime protection", ICWWW 2004.
G. Wassermann and Z. Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities", PLDI 2007.
D. Balzarotti et. al. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications", S&P 2008.
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.
A. Sabelfeld and A. C. Myers, "Language-based information-flow security," in IEEE Journal on Selected Areas in Communications, vol. 21, no. 1, pp. 5-19, Jan 2003.
Darvas Á., Hähnle R., Sands D. (2005) A Theorem Proving Approach to Analysis of Secure Information Flow. In: SPC 2005. Springer.
O. Tripp, M. Pistoia, P. Cousot, R. Cousot, S. Guarnieri. 2013. ANDROMEDA: Accurate and scalable security analysis of web applications. In FASE'13. Springer, 210-225.