Static Detection of Application Vulnerabilities in PHP

Necdet Can Atesman
c.atesman@gmx.net
937/0004289

December 2007

Abstract

This paper is intended to be a summary of the ideas provided by Yichen Xie & Alex Aiken [1]. Their results will be compred to the conclusions of Nenad Jovanovic, Christopher Kruegel & Engin Kirda in [2]. Both author teams seek to reduce security vulnerabilities in PHP applications through static analysis techniques with two different approaches, each having their own advantages and drawbacks.

Nenad Jovanovic, Christopher Kruegel & Engin Kirda try to find XSS vulnerabilities using taint checking, whereas Yichen Xie & Alex Aiken focus on sql injection attacks using a custom three-tier architecture that could be best described as reverse taint checking, since their approach consists of analysing the control flow of the application and concluding, which initial variables must not be tainted, in contrast to normal taint checking, where variables are traced through a given application.

1 Introduction

Web applications have emerged as the standard for almost any online service provided to end users. As such, dynamic web application programming had a strong lure on inexperienced programmers, who found PHP to be an easy language to work with [3]. Unfortunately, neither the PHP language itself, nor beginner's literature to the language, seldom cover security aspects of web applications, which resulted in a large number of security vulnerabilities during the last few years [4].

There have as well appeared several techniques, static and dynamic, for reducing threats to web applications. In this paper two static techniques will be discussed: one, which uses taint checking [2], and another that uses a unique three-tier architecture to find vulnerabilities [1]. The focus will be on the latter technique to bring up its unique edges.

Static taint checking, as done by Pixy [2], a static analyser developed by Nenad Jovanovic, Christopher Kruegel & Engin Kirda, steps through the source code, storing data on possibly tainted values, namely those received by the client, that could hold arbitrary data. If any of those input values would be used in a

critical function call - like in a call to `mysql_query()`, which could result in a mysql injection - without being checked first, a possible vulnerability is detected.

Yichen Xie and Alex Aiken on the other hand use a custom algorithm to detect programming errors of the same type. Their main idea was to break the source code into annotated parts, which store pre- and postconditions among other data. A possible vulnerability is found if a precondition cannot be met, which can be the case if the value of a variable is arbitrary.

Since both applications were meant to be a proof of concept, only subsets of the PHP language were modeled. Nevertheless, both tools were capable of detecting several vulnerabilities in the latest versions of six PHP applications.

2 Addressed Vulnerabilities

Both techniques are intended to find taint-style vulnerabilities, which basically arise from unvalidated user input. These include Cross-Site-Scripting (XSS) and SQL-Injection attacks. The nature of both attacks is that the attacker tries to provide user data that will taint a statement in another language that is used either by the client or by the server.

With XSS, the attacker provides a javascript in an HTTP Get Request that will be embedded in the page generated by the HTTP response. These scripts mostly force the client to send its domain-specific cookie to the attacker, gathering login data or stealing sessions this way.

Sql injection attacks proceed the same way, sending malicious parts of sql queries, which are then embedded by the server in their own query skeletons. A database query like `select password from user where name='$name'` could be remodeled to `select password from user where name='' or '1'='1'` with a value of `$name=' or '1'='1`, resulting in a fetch of all passwords, instead of the own.

3 Taint Checking

Taint checking (or tainting), refers to the symbolic execution of the application, where tainted (user-provided) values are tracked to the so-called sink functions.

User-supplied input values are stored in the special variables $_GET, $_POST and $_COOKIE in PHP. Any reference to a value within these hashes is well marked as tainted, as well as any references to them, etc. If such an unchecked value is ever used as an argument to a sink function (such as `print` in XSS attacks or `mysql_query()` in sql injection attacks), a security flaw is at hand.

Pixy [2], the analysis tool developed by Nenad Jovanovic, Christopher Kruegel & Engin Kirda, was written for checking vulnerabilities for XSS-attacks but could be modified to include sql injection attacks as well. Some test runs with the analysis tool show that the concept is quite successful, presenting a total of 51 XSS vulnerabilities in 6 applications.

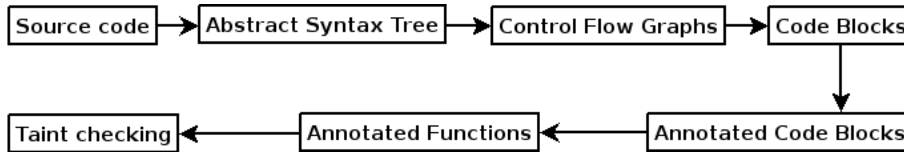

Figure 1: process outline

4 Three-Tier-Analysis

The second approach is a bit more elaborate, since the analysis uses annotations to create a primitive form of semantics of the code. Every PHP Application consists of an entry point and zero or more user-defined functions. If we considered the entry point itself as a unique function, the whole application could be described as a multitude of functions, where one of them will be called on application start. These functions themselves consist of several code blocks.

This break-down is the basic idea behind the three-tier architecture:

- The entire application
- Functions declared in the application
- Code blocks within the functions

4.1 Parsing

As seen in figure 1, the code is first parsed into an abstract syntax tree and every function is then further processed to yield several control flow graphs (CFG). Based on the CFG of a function, the statements therein are bundled into blocks. At this point, the application is reduced to a collection of functions containing code blocks.

4.2 Symbolic Execution

The next step is to simulate all blocks using symbolic execution. An initial symbolic state is generated, where each variable is assigned a start value. Execution of the code block is then simulated for each expression until the code block either has no more expressions or terminates prematurely through a call to `return`, `exit` or a call to another code block that is known to terminate the application.

4.2.1 Block Analysis

During the symbolic execution, a list of variables is stored that have been sanitized through the call of pre-defined functions, casting to safe types (like numeric values) or checking against regular expressions. To guarantee that a regular expression sanitizes an input value, a database of known regular expressions was created that guaranteed to sanitize user input. If a previously unknown regular expression was found, the analyzer would ask the user whether the expression was safe to use and added it to the database accordingly.

After the symbolic execution of a code block, an annotation is attached to it containing the following information:

- **Error set**: A list of input variables that must be sanitized before entering this code block. The values listed here will all be used in a call to a critical function like `mysql_query()`.
- **Definitions**: The set of variables that will be defined in this code block. The code block `$a = $a.$b; $c = 123;` for example will result in {a, c}.
- **Value flow**: A Set of pairs of variables (x, y), where the variable x will be a sub-string of y upon exit. The above code block will yield {(a, a), (b, a)} for example.
- **Termination Predicate**: Indicates whether this code block terminates the application, either through a call to `exit`, or a call to function that has its Termination Predicate set to `true`.
- **Return value**: The representation of the return value, if any.
- **Untaint set**: This set contains values that are sanitized in this block using one of the above mentioned methods. There is one Untaint Set for every successor of this block, since different variables can be sanitized on different paths. The following code block, for example:

  ```
  validate($a);
  $b = (int) $c;
  if (is numeric($d)) ...
  ```

  will result in an untaint set of

  ◇ {a, b, d} for the code block to be evaluated, if the condition is true and

  ◇ {a, b} for the code block to be evaluated, if it is false.

4.2.2 Intraprocedural Analysis

Using the annotations generated in the Block Analysis, the next step generates annotations for functions:

- **Error set**: Again the list of values that must be sanitized before entering this function is stored.
- **Return set**: A list of parameters and global variables that may be part of the returned string. This value is only computed for functions that return string values since this is the only type that can be used in a taint-style attack.
- **Sanitized values**: A list of parameters and global variables that have been sanitized upon function exit. If the function returns a boolean, two different sets of variables is stored, on for the case that the function returns `true`, and one for the case returning `false`. This is merely for catching functions that do taint checking on variables.
- **Program exit**: A boolean indicating whether the current function will terminate program execution in any case.

4.2.3 Interprocedural Analysis

After having annotated the all functions of the application (including the virtual `main()` function), the code blocks calling user-defined functions can be resolved. This is where the real taint checking is done.

First, the pre-conditions are validated. If a function requires one of its parameters to be sanitized, as described in its Error Set, it has to be checked, whether the provided value is really safe. If not, an error is generated, since this could be a security issue.

Second, if the function is marked to unconditionally exit the application, all succeeding code blocks are removed to indicate that the application will stop at this point. The block is further marked as an exit block.

Post-Conditions are checked in the next step. If a function unconditionally marks some values as untainted, these values are added to the list of untainted variables. If taint validation depends on a condition, the function will return multiple sets of variables that are marked as untainted. The intersection of these sets is added to the list of sanitized variables.

5 Results

Both tools successfully identified numerous security flaws, some of which were formerly unknown. A direct comparison of both results would be deceiving, though, since two different attack types were checked against on completely different applications.

The applications tested by Jovanovic, Kruegel & Kirda are:

- PhpNuke 6.9
- PhpMyAdmin 2.6.0-pl2
- Gallery 1.3.3,
- Simple PHP Blog 0.4.5
- Serendipity 0.8.4
- Yapig 0.95b

The applications tested by Xie & Aiken are

- e107
- News Pro
- myBloggie
- DCP Portal
- PHP Webthings

Furthermore, the two techniques have different design philosophies. While the PIXY project guarantees that all security flaws are found, Xie & Aiken observed general PHP practices and tailored their tool to the state-of-the-art programming techniques within the language.

There are some side effects to both approaches. Pixy needs one file, where all file inclusions have been resolved and included source code has been embedded. The architecture of Xie & Aiken on the other hand can handle dynamic file

inclusions.

Furthermore, the Pixy project has difficulties analyzing recursive functions, since it works top-down, i.e. the analysis would re-enter a recursive function. Xie & Aiken do the analysis bottom-up, analyzing the call first, thereafter resolving any calls to itself.

The results of both tests are provided in table 1 for reference nevertheless.

Project	Applications	Found bugs	False positives
Jovanovic, Kruegel & Kirda	6	105	0
Xie & Aiken	5	51	47

Table 1: Test Results

6 Conclusions

Two techniques have been presented to statically analyze dynamically typed scripting languages, which both have a reference implementation for PHP applications. Each approach proves itself successfull by exposing several security flaws in different applications.

References

[1] Yichen Xie and Alex Aiken:
Static Detection of Security Vulnerabilities in Scripting Languages

[2] Nenad Jovanovic, Christopher Kruegel and Engin Kirda:
Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities

[3] Stephen Shankland. Andreessen: PHP succeeding where Java isn't.
http://www.zdnet.com.au, 2005

[4] Symantec Internet Security Threat Report: Vol VII. Technical Report.
Symantec Inc. March 2005