Towards PHP Vulnerability Detection at an Intermediate Language Level

Paulo Antunes, Ibéria Medeiros, Nuno Neves
LASIGE, Faculdade de Ciências da Universidade de Lisboa, Portugal
pdantunes@fc.ul.pt, ivmedeiros@fc.ul.pt, nfneves@fc.ul.pt

***Abstract*—Web applications are a prime target for malicious actors to obtain private user information, such as credit card numbers and other sensitive details. Over the years, the number of vulnerabilities and attacks has increased, demonstrating that current solutions have shortcomings. For example, they can be prone to error or require too much resources/time from developers (or security analysts) to deliver results. This paper presents a new approach to detect vulnerabilities in web applications written in PHP by analyzing their representation in an Intermediate Language (IL) and simulating the execution through a new data structure.**

***Index Terms*—Vulnerability Discovery, Code Analysis, Web Applications**

I. Introduction

The rapid evolution of web technologies and ease of access to computational power has led many services to have an online component. Through these web services, it is possible to acquire various products, contact friends and family, or even access sensitive bank account information. All these functionalities are typically provided through web applications, and although billions of users employ them daily, many inadvertently incorporate latent vulnerabilities. If exploited, these flaws can lead to severe consequences ranging from private data theft and adulteration of information to complete service shutdown. These vulnerabilities are often a result of insecure programming practices allied with the utilization of languages and Content Management Systems (CMS) that rely on insufficient validation procedures [1]. A concrete example would be PHP, which has limited data type validation. Even so, PHP is still the most used language for web application back-end development [2].

There are several approaches to find vulnerabilities, but various of these possess drawbacks. Static analysis can be utilized at any stage of the application's life cycle, but it depends on the vulnerability classes' knowledge base and code modeling technique. Any error or incompleteness in these can result in false positives or false negatives [3]. Fuzzing can create various inputs cyclically to test the application for potential errors and vulnerabilities. However, finding an input that can reliably trigger a vulnerability can take significant time as there is an infinite set of possible input values [4]. Symbolic execution is an approach that associates variables with a group of symbolic values to infer what set of values can result in specific program behavior. While this approach can provide a complete knowledge of program behavior, sometimes there is a significant overhead to solve symbolic expressions, and the exploration of the code can result in the problem of path explosion [5].

```
<?php
$user = $_GET['username'];
echo "Welcome ".$user;
?>
```

```
line     #* E I O op            return  operands
-----------------------------------------------------
2     0  E >   FETCH_R         $2      '_GET'
      1        FETCH_DIM_R     $1      $2, 'username'
      2        ASSIGN                  !0, $1
3     3        CONCAT          ~1      'Welcome+', !0
      4        ECHO                    ~1
4     5      > RETURN                  1
```

Fig. 1. Example of PHP code (left-side) to IL conversion (right-side).

We aim to design and implement a tool to address some of these issues by performing the analysis at an IL level, which is produced while executing a PHP application.

II. Challenges

A. Intermediate Language

During application execution, PHP translates high-level language constructs to a set of IL instructions composed of opcodes, which are then processed by a dedicated virtual machine (VM). This VM interprets the opcodes and executes the corresponding operations on the hardware. The IL has a reduced semantic and syntactic complexity since IL instructions typically comprise an opcode, two operands, and a return value. Therefore, although the IL program has many more instructions than the PHP program, each instruction has significantly less complexity. An example of a simple excerpt of vulnerable PHP code can be found in Figure 1, along with the generated IL code. The PHP code `echo "Welcome".`$user$ is converted into the operations opcodes $CONCAT$ and $ECHO$, where the former concatenates the string `Welcome` with the variable $user$ temporally stored at `!0`, whereas the latter executes the expression returned from the former represented as $\sim$`1`. Our objective is to have a tool that accurately emulates the execution of these IL instructions and the changes in the program state, enabling the detection of vulnerabilities when activated.

B. Code Representation

Several structures exist to represent code and enable analysis, such as the Abstract Syntax Tree (AST), the Control Flow Graph (CFG), the Program Dependence Graph (PDG), and the Code Property Graph (CPG). However, they were not tailored to deal with the PHP IL and support the discovery of web vulnerabilities. In addition, they have a few shortcomings: AST lacks a straightforward way to determine what causes

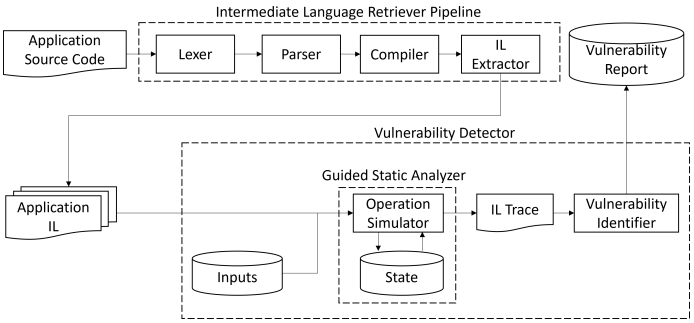

Fig. 2. Proposed vulnerability detection pipeline.

the execution of a given statement and which statements influence the data of other statements; CFG represents the relationship of control between statements but can make the origin of data ambiguous (since it disregards data flow); PDG represents the relation between program statements through two different edges that correspond to data and control dependencies; although this gives a notion of data dependency, we lose the capability of discerning the order in which programs statements are executed. CPG is, in essence, generated by combining the AST, CFG, and PDG into a single structure; it is not designed with IL in mind and does not accurately represent the states of a program's execution. Hence, we need to develop a novel structure to support the analysis.

C. Vulnerability Detection

The tool must output relatively accurate results to be usable in real-life scenarios. False negatives during the analysis will create a false sense of security over the quality of the code. In contrast, false positives can needlessly take developers' time by forcing them to analyze several lines of code for an issue that does not exist, thus eroding trust in the tool and making them skeptical towards future vulnerability reports. It is also necessary to have results in a reasonable time frame, as organizations typically allocate a limited period and/or monetary budget for application testing. Vulnerability detection in IL can benefit from the finer-grain nature of the operations that are being analyzed, which makes operations less ambiguous and require fewer approximations (the usual cause of false positives and false negatives). However, as mentioned in Section II.A, having a larger set of operations to analyze is an added challenge.

III. Current Design and Implementation

The tool implementation, see Figure 2, is being developed in Python and works in tandem with a modified version of VLD [6]. VLD is a plugin that can show low-level PHP structures, namely the IL instructions. We changed VLD slightly to facilitate opcode extraction and obtain additional information regarding certain variable types.

- *IL Extractor*: Intercepts the PHP execution and extracts the IL of the entire application (i.e., the IL of all files and functions). This information is stored by our tool for subsequent use;
- *Guided Static Analyzer*: utilizes the obtained IL jointly with a set of inputs to the application. It simulates the execution of the IL instructions while processing a concrete input, starting from the application's entry point. As the simulation progresses, it saves the changes to the program's state and the relationship between operations in a Program State Graph (PSG) (a structure designed by us). This structure is generated by interpreting the produced IL and simulating their execution with a given input, which can be provided manually or by a fuzzer. The state at each instance of the application is represented through nodes, and the connecting edges are state-altering operations that produce new states;
- *Vulnerability Identifier*: performs a taint analysis over the produced PSG by replacing the concrete values of the simulation with tainted values and propagating them throughout the graph. Whenever a tainted value reaches a critical instruction, this most probably indicates that a vulnerability exists. In this case, a report presenting the affected line of code, the function, and the tainted variable is generated. This report aims to aid developers and security analysts in addressing the issue swiftly.

The tool is partly implemented and already supports over 100 IL opcodes. It can also produce an IL trace (based on the PSG) as intended. Regarding error detection, we have performed initial testing of PHP applications and produced reports that successfully detected a few vulnerabilities.

IV. Conclusions and Future Work

Previous approaches to finding web vulnerabilities include static analysis, fuzzing, and symbolic execution. However, each has drawbacks, and vulnerabilities are still high in web applications. Some research has also focused on bytecode analysis [7], but the proposed methodology differs extensively from the one presented in this paper as it targets a different language (Java) and utilizes a CPG to represent code and conduct an analysis.

About future work to be done, we would like to refine our vulnerability detection further and extend the number of supported opcodes. Additionally, to make the tool more efficient at exploring paths, we are currently developing a methodology that creates new IL traces without full execution. This will be done by forcing state changes at key inflexion points of the code (i.e., conditional jumps), enabling the simulation of only the remaining portion of the code. The benefits are saving resources and exploring new code paths that can be analyzed for vulnerabilities.

Acknowledgments: This work was supported by FCT through a PhD. Scholarship (2020.04482.BD), and the LASIGE research unit (UIDB/00408/2020 & UIDP/00408/2020), and by P2020 through the XIVT project (LISBOA-01-0247,FEDER-039238, an ITEA3 European project (I3C4-17039)).

REFERENCES

[1] The State of Web Application Vulnerabilities in 2019. Imperva, 2020. https://www.imperva.com/blog/the-state-of-vulnerabilities-in-2019/

[2] Usage Statistics for PHP Websites, W3Tech, 2023, https://w3techs.com/technologies/details/pl-php

[3] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, M. Vieira. Benchmarking Static Analysis Tools for Web Security. IEEE Transactions on Reliability, 2018, Vol 67, pp. 1159-1175

[4] G. Klees, A. Ruef, B. Cooper, S. Wei, M. Hicks. Evaluating Fuzz Testing, ACM Conference on Computer and Communications Security, 2018, pp. 2123-2138

[5] C. Cadar and P. Godefroid and S. Khurshid and C.S. Păsăreanu and K. Sen and N. Tillmann and W. Visser. Symbolic Execution for Software Testing in Practice: Preliminary Assessment. Proc. of the International Conference on Software Engineering, 2011, pp. 1066-1071

[6] D. Rethans, Vulcan Logic Dumper, 2023, https://derickrethans.nl/projects.html

[7] W. Keirsgieter, W. Visser. Graft: A Static Analysis Tools for Java Bytecode. Conference of the South African Institute of Computer Scientists and Information Technologists, 2020, pp. 217-226