Multi-Language Web Vulnerability Detection

Alexandra Figueiredo Tatjana Lide Miguel Correia
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
alexandra.figueiredo@tecnico.ulisboa.pt, tatjana.lide@tecnico.ulisboa.pt, miguel.p.correia@tecnico.ulisboa.pt

I. Introduction

Most web applications are compromised due to vulnerable source code [1]. Static code analysis tools that are often used to find security vulnerabilities in code have two main problems: they are language-specific, and they have to be programmed, or at least configured manually, to deal with new types of vulnerabilities.

This paper explores the development of a static analysis tool called MERLIN (Multi-language wEb vulneRabiLity detectIoN) that aims to solve these two problems by *detecting vulnerabilities in code written in different languages* and by *learning how to detect vulnerabilities*.

MERLIN searches for *input validation vulnerabilities* which are the most common web application security weaknesses [1]. Specifically, at the current state we consider the following classes of vulnerabilities: SQL injection, XSS, remote and local file inclusion, path traversal, source code disclosure, operating system and PHP command injection. To detect these vulnerabilities, MERLIN performs *data flow analysis*: it examines how data flows through the code of the program, considering the code semantics. Data flow analysis is a common technique used to perform static code analysis (e.g., RIPS, PHOSPHOR, WAP). Nevertheless, static code analysis tools that use data flow have limitations, such as developers having to code their knowledge about vulnerabilities.

MERLIN supports several programming languages by first translating source code written in a high-level language into intermediate code. Currently the tool supports web code written in Java and PHP and translates it into Jimple, part of the Soot framework [3]. MERLIN uses a machine learning classifier to learn how to detect vulnerabilities from a large set of examples.

During implementation of this tool we faced several challenges. The first challenge was to find an intermediate code language suitable to represent different high-level languages. We ended up selecting Jimple that is obtained from Java bytecode, which is trivially obtained from Java source code. The second challenge was translation from PHP to Java bytecode, since the main goal of the software available for this purpose was not to produce bytecode, but to execute it. Another major challenge was to interpret functions in the intermediate code. Web applications written in languages other than Java do not preserve the original symbols in the intermediate code. Thus, it was necessary to perform an extensive analysis of the intermediate code, so that the tool is able to correctly interpret and, process the symbols, and specifically function names, resulted from the compilation. Yet another challenge is the need to include configuration files with sensitive sinks, sources and sanitization functions for each programming language considered.

II. Approach

MERLIN supports web applications written in different languages. We chose to focus on web applications written in PHP and Java for now, and are considering other programming languages for the next phase. These two programming languages are among most popular programming languages for developing web applications.

Figure 1 shows the architecture of the tool. It consists of two main modules: the *Code Analysis Module* that processes source code and detects candidate vulnerabilities in intermediate code; the *Vulnerability Detection Module* that classifies vulnerabilities using machine learning.

A. Code Analysis Module

Initially, the *Bytecode Converter (1a)(1b)* modules receive a web application as input and compile it to Java bytecode. The bytecode converter used depends on the language in which the web application is written: javac when processing Java code, JPHP when processing PHP code. Thereafter, the bytecode is converted to Jimple, an intermediate representation, by the submodule *Intermediate Language Converter (2)* using the Soot framework. The Soot framework is also used to build *control flow graphs* (CFGs) in the submodule *CFGs Builder (3)*.

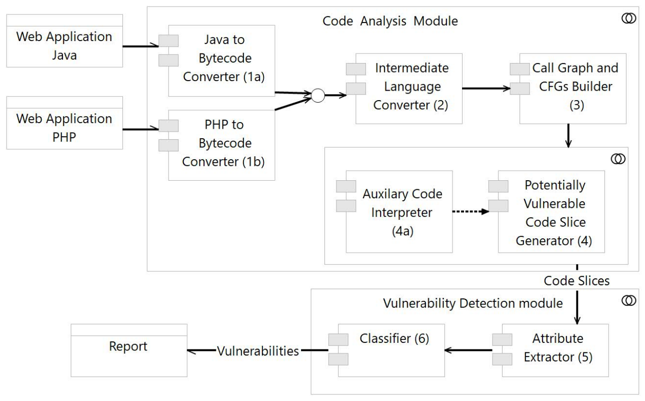
Code Analysis Module
Web Application Java
Web Application PHP
Java to Bytecode Converter (1a)
PHP to Bytecode Converter (1b)
Intermediate Language Converter (2)
Call Graph and CFGs Builder (3)
Auxilary Code Interpreter (4a)
Potentially Vulnerable Code Slice Generator (4)
Code Slices
Vulnerability Detection module
Attribute Extractor (5)
Classifier (6)
Vulnerabilities
Report

Fig. 1. Architecture of the MERLIN tool considering PHP and Java code.

Then, the Jimple code and the generated CFGs are analyzed by the submodule *Potentially Vulnerable Code Slice Generator (4)* along with a configuration file that provides entry points (places where inputs enter the program), sanitization functions (that cleanup dangerous inputs) and sensitive sinks (functions that may be exploited by malicious input). The submodule analyzes all functions and uses the configuration file to collect potentially vulnerable code slices. A code slice contains sensitive sink, the variables passed to the sensitive sink and all statements semantically related in terms of data dependency or control dependency. Whenever there is a call to a function/method of the same module or another module, MERLIN processes the function/method and propagates the data flow through all parameters. Furthermore, it stores the return value of the function if applicable. Thus, MERLIN performs an inter-procedural, global and context sensitive analysis.

When processing web applications written in languages other than Java, an auxiliary submodule may be needed to help interpret compilation of source code to Java bytecode. This happened with web applications written in PHP, and it was necessary to add an extra submodule *(4a)* to interpret functions generated by the JPHP tool that transforms PHP code into Java bytecode instructions.

B. Vulnerability Detection Module

Attribute Extractor (5) receives the processed potentially vulnerable code slices and produces attribute vectors. Based on the set of attributes used in WAP [2], we chose to use 22 attributes which are related with the presence of vulnerabilities. The attributes chosen include the following categories: natures of input sources, types of sensitive sinks, input sanitization methods, string manipulation, validation and encoding functions.

Classifier (6) uses the attribute vector to classify the code as vulnerable or not vulnerable. Initially, we considered 7 classifiers of the Graphical/Symbolic, Random Forest, Probabilistic, and Neural Networks classes, implemented in Python libraries. We trained the classifiers with a set of code samples to extract knowledge about the vulnerabilities and evaluated which machine learning classifier achieved the best results. After these stages, we concluded that SVM has the best performance and it is the one used in MERLIN.

III. Evaluation

So far MERLIN has processed more than 750 thousand lines of code with vulnerabilities from all categories. We ran MERLIN with 17653 files from the SRD database (https://www.nist.gov/srd): 5064 Java; 12589 PHP. Furthermore, we also used a few code samples written by us to train and test the tool. Therefore, the classifiers were trained with a total of 42011 code slices and tested with 18004 code slices. The classifier that obtained the best results was SVM among the seven classifiers used. The SVM achieved a precision of 94.6%, a recall of 98.16% and a f-score of 96.24%.

We also verified MERLIN's ability to correctly process in the same way web applications written in Java and PHP. In order to verify this, we ran the tool with web applications written in Java and PHP with the same types of vulnerabilities

```
$u = $_GET['user'];
$q = "SELECT pass FROM users where user='".$u."'";
$query = mysql_query($q);
```

(a) Code sample in PHP

```
String u = request.getParameter('user');
String q = "SELECT pass FROM  users where user='" + u + "'";
ResultSet query = conn.createStatement().executeQuery(q);
```

(b) Code sample in Java

Fig. 2. Examples of vulnerabilities detected by the tool.

TABLE I
Analysis of SRD and Real World Web Applications

webapp	language	#loc	#files	#vuln
SRD	Java/PHP	604,227	17653	12,126
DVWAP	PHP	14,895	353	38
multidae	PHP	142,515	919	225
bWapp	PHP	24,070	198	192
wackopicko	PHP	1,916	48	65
Java Vulnerable Lab	Java	1,795	60	158
HackMe	Java	824	17	27
Total	Java+PHP	790,242	19,248	12,831

and with similar sanitization. An example is shown on the Figure III where both code samples are vulnerable to SQLi and as a result the tool was able to correctly identify the vulnerabilities in both code samples.

The tool was also tested with real world web applications written in Java and PHP which contained seeded vulnerabilities. The results are summarized in Table I. The columns represent respectively: the web applications analysed, the number of lines of code processed, the number of files processed and the number of vulnerabilities found by the tool.

IV. Conclusion

We developed a tool to improve security in web applications, by detecting vulnerabilities in web applications written in different languages using machine learning. The tool has produced promising results. Next we will compare the results achieved by this tool with other tools that detect vulnerabilities in Java bytecode and in source code. Moreover, we will also test the tool with more real world web applications and with more vulnerability categories.

Acknowledgements Research supported by national funds through Fundação para a Ciência e Tecnologia (FCT) w/references PTDC/CCI-INF/29058/2017, LISBOA-01-0145-FEDER-029058, POCI-01-0145-FEDER-029058 (SEAL) and UIDB/50021/2020 (INESC-ID).

References

[1] M. Meucci and A. Muller, "OWASP testing guide 4.0," OWASP Foundation, Tech. Rep., 2014, pp. 100.
[2] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web ap-plication vulnerabilities using data mining to predict false positives," in Proceedings of the International World Wide Web Conference, Apr. 2014, pp. 63–74.
[3] P. Lam, E. Bodden, O. Lhoták and L. Hendren, "The Soot framework for Java program analysis: a retrospective," in Cetus Users and Compiler Infastructure Workshop (CETUS 2011), 2011.