The Journal of Systems and Software 113 (2016) 337–361

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Exploring context-sensitive data flow analysis for early vulnerability detection

Luciano Sampaio*, Alessandro Garcia

Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Rua Marquês de São Vicente, 225, 22453-900 Rio de Janeiro, Brazil

ARTICLE INFO

Article history:
Received 6 April 2015
Revised 10 October 2015
Accepted 7 December 2015
Available online 18 December 2015

Keywords:
Early detection
Data flow analysis
Secure programming

ABSTRACT

Secure programming is the practice of writing programs that are resistant to attacks by malicious people or programs. Programmers of secure software have to be continuously aware of security vulnerabilities when writing their program statements. In order to improve programmers' awareness, static analysis techniques have been devised to find vulnerabilities in the source code. However, most of these techniques are built to encourage vulnerability detection a posteriori, only when developers have already fully produced (and compiled) one or more modules of a program. Therefore, this approach, also known as late detection, does not support secure programming but rather encourages posterior security analysis. The lateness of vulnerability detection is also influenced by the high rate of false positives yielded by pattern matching, the underlying mechanism used by existing static analysis techniques. The goal of this paper is twofold. First, we propose to perform continuous detection of security vulnerabilities while the developer is editing each program statement, also known as early detection. Early detection can leverage his knowledge on the context of the code being created, contrary to late detection when developers struggle to recall and fix the intricacies of the vulnerable code they produced from hours to weeks ago. Second, we explore context-sensitive data flow analysis (DFA) for improving vulnerability detection and mitigate the limitations of pattern matching. DFA might be suitable for finding if an object has a vulnerable path. To this end, we have implemented a proof-of-concept Eclipse plugin for continuous DFA-based detection of vulnerabilities in Java programs. We also performed two empirical studies based on several industry-strength systems to evaluate if the code security can be improved through DFA and early vulnerability detection. Our studies confirmed that: (i) the use of context-sensitive DFA significantly reduces the rate of false positives when compared to existing techniques, without being detrimental to the detector performance, and (ii) early detection improves the awareness among developers and encourages programmers to fix security vulnerabilities promptly.

1. Introduction

Secure programming is the practice of writing software systems that are resistant to attacks by malicious people or programs (Apple, 2013). In order to promote secure programming, developers have to be continuously aware of security vulnerabilities when writing their program statements. They need to be prepared to continuously perform actions for preventing and removing vulnerabilities from their programs. Security vulnerability (or simply vulnerability) is a flaw within a software system that can be exploited to allow an attacker to reduce the system's information assurance (Organization for Internet Safety, 2004). An attacker is a person or application that intends to cause damage to a software system. By exploiting a security vulnerability, an attacker takes advantage of this vulnerability, typically for malicious purposes, such as stealing information or causing damage to a computer system.

In the context of this paper, we are particularly concerned with security vulnerabilities introduced by programmers when adding or editing code statements. Unfortunately, existing software development environments – such as Eclipse,[1] NetBeans[2] and others – often do not offer the means to make programmers aware they are writing insecure code. Therefore, if a company or a developer wants support for performing secure programming, they have to use additional external tools, such as: IBM Appscan (IBM, 2001), Lapse+ (Livshits, 2006) and others. However, these solutions frequently do not fit properly on the development workflow. They either are not integrated into the development environments or do

* Corresponding author. Tel./fax: +55 21 3527 1500.
E-mail addresses: lsampaio@inf.puc-rio.br, lsampaioweb@gmail.com (L. Sampaio), afgarcia@inf.puc-rio.br (A. Garcia).

[1] https://eclipse.org/home/index.php.
[2] https://netbeans.org/.

http://dx.doi.org/10.1016/j.jss.2015.12.021
0164-1212/

not detect the vulnerabilities exactly when they are added into the source code. Consequently, they only support "a posteriori" security analysis in the source code rather than supporting actual secure programming. Thus, both novice and experienced developers are not encouraged to detect and remove security vulnerabilities in the code they are editing.

Developers should be aware of emerging security vulnerabilities as they write their program statements. If a team of developers wants to perform secure programming on all of its projects, at least the most common security vulnerabilities should be handled by the programmer who is adding or editing the code, leaving only the more complex ones to actual security specialists. In order to achieve secure programming, developers should receive tooling support to continuously detect and remove security vulnerabilities in their code edition context. Otherwise, developers might be unconscious about the security vulnerabilities emerging in their code. Ideally, they should be detected and fixed before the programmer's code is committed into the project's repository. If done afterwards, developers might spend hours, days or weeks to find out and fix vulnerabilities in their code (Baca et al., 2008).

In fact, there is recent trend to investigate solutions that support early detection of some implementation problems, such as modularity problems (Albuquerque et al., 2014) and exception handling flaws (Barbosa et al., 2012). However, there is limited knowledge on how to specifically support early detection of security vulnerabilities in programs. Zhu (2012) was the only author to recently address this problem. They created a prototype solution, called ASIDE (Application Security plugin for Integrated Development Environment), that performs early detection of security vulnerabilities in source code. However, the use of ASIDE might result in a high amount of false positives as it is strictly based on a low-accurate technique called *pattern matching*. False positive is the incorrect indication of the presence of a vulnerability (IBM, 2008). As described by Nadeem et al. (2012), the occurrence of several false positives discourages the use of existing static analysis solutions for security vulnerability detection. Unfortunately, according to Nadeem et al. (2012), other existing solutions (not only ASIDE; Zhu, 2012) result in a high rate of false positives. Most of these solutions are also based on pattern matching and, therefore, would be hard to tailor them to support early vulnerability detection. Unfortunately, the use of early detection without employing a high accuracy technique would not be sufficient for achieving secure programming. Developers would be discouraged to write secure programs if they often become frustrated by continuously treating a high amount of false positives when editing their code.

In order to address the aforementioned problems, we propose the combination of two ideas on this paper. First, we propose to support a change from the default behavior of late detection to early detection. We believe this change improves the support for secure programming. Second, we propose new heuristics to find security vulnerabilities using a technique named *context-sensitive* data flow analysis (Hammer et al., 2006) instead of using low-accurate techniques, such as pattern matching. We expect the use of context-sensitive data flow analysis (DFA) will decrease the rate of false positives yielded by existing solutions. Consequently, the improvement on the accuracy detection will likely to encourage developers to detect and remove vulnerabilities in their source code. We defined a suite of DFA-based heuristics to support detection of vulnerabilities that occur on web applications. These vulnerabilities stem from program inputs and outputs that are not properly validated.

After our detection heuristics were created, we designed and implemented a prototype. This prototype enabled us to verify if and to what extent our detection heuristics could decrease the rate of false positives when compared to other automated techniques. Although our heuristics are can be implemented to the context of several programming languages, the first (and current) version of our prototype only provides support for the Java[3] programming language. This choice was driven by the fact that Java is one of the most popular programming languages (Zeichick, 2012). The prototype is a plugin for the Eclipse[4] IDE (integrated development environment), which is the most popular IDE used for the Java programming language (Geer, 2005). The plugin, called ESVD – Early Security Vulnerability Detector, can be downloaded from the Eclipse Marketplace (Sampaio and Garcia, 2014).

Finally, we design and execute two empirical evaluations. They are aimed to improve our understanding about the use of early vulnerability detection based on data flow analysis. The first evaluation intends to verify if developers receiving continuous detection support (i.e. early detection) could produce more secure code than developers receiving support afterwards, i.e. only at the end (late detection) of their programming session. The second evaluation was specifically targeted at measuring the accuracy of our prototype (using data flow analysis) compared to other existing solutions.

The remainder of this paper is structured as follows. Section 2 presents the theoretical background required to understand the main concepts of this paper. This section also describes the main existing studies about the subject discussed on this paper. Section 3 describes the heuristics created to find security vulnerabilities in the source code. This section also describes the components, which compose our algorithm. It also discusses which vulnerabilities are supported by our heuristics, how these vulnerabilities occur in the source code and what is necessary to remove them. Section 4 presents the software architecture of our implemented solution. Section 5 discusses the empirical evaluations to explicitly address our research questions (Section 5.1). Sections 6 and 7 describe in full detail the experiments that were performed during our study. Finally, Section 8 concludes this paper, by showing the main contributions made and discussing some possibilities and future research directions.

2. Background and related work

This section presents the background required to understand the main concepts of this paper. It also discusses related work.

2.1. Security terminology

We used the commonly referenced taxonomy from Tsipenyuk et al. (2005) and their definitions are presented next. Input - Data provided by the user of a software system or by another application. Malicious input – input that is intended to cause harm to the application or to other users. Untrusted/Unvalidated input – input that has not been compared to a range of expected values or has not removed malicious data from its content to ensure it is safe to use. Trusted/Validated/Sanitized input - Input that has been compared to a range of expected values and has removed (if any) malicious data from its content to ensure it is safe to use. Encode/Escape input – the process of converting some input, e.g. a sequence of characters (letters, numbers, punctuation, and certain symbols) into a specialized digital format, such as HTML tags. Decode/Unescape input – the process of converting an encoded input back into its original sequence of characters. Security vulnerability – security vulnerabilities (or simply vulnerabilities) is a flaw within a software system that can be exploited to allow an attacker to reduce the system's information assurance (Organization for Internet Safety, 2004). In other words, an attacker can exploit a program

[3] https://www.oracle.com/java/.
[4] https://www.eclipse.org.