What Do We Know About Buffer Overflow Detection?
A Survey on Techniques to Detect A Persistent Vulnerability

Marcos Lordello Chaim, School of Arts, Sciences and Humanities, University of Sao Paulo, Sao Paulo, Brazil
Daniel Soares Santos, Institute of Mathematical Sciences and Computing, University of Sao Paulo, São Carlos, Brazil
Daniela Soares Cruzes, SINTEF Digital, Trondheim, Norway

ABSTRACT

Buffer overflow (BO) is a well-known and widely exploited security vulnerability. Despite the extensive body of research, BO is still a threat menacing security-critical applications. The authors present a comprehensive systematic review on techniques intended to detecting BO vulnerabilities before releasing a software to production. They found that most of the studies addresses several vulnerabilities or memory errors, being not specific to BO detection. The authors organized them in seven categories: program analysis, testing, computational intelligence, symbolic execution, models, and code inspection. Program analysis, testing and code inspection techniques are available for use by the practitioner. However, program analysis adoption is hindered by the high number of false alarms; testing is broadly used but in ad hoc manner; and code inspection can be used in practice provided it is added as a task of the software development process. New techniques combining object code analysis with techniques from different categories seem a promising research avenue towards practical BO detection.

KEYWORDS


1. INTRODUCTION

Unintended parties have always tried to access sensitive data stored in software systems. Interconnected computers, though, have multiplied exponentially the harm caused by a breach of security. One of the first largely publicized attacks compromising a great number of interconnected computers was caused by the Morris (or the Internet) worm of 1988. It exploited particulars of the then common DEC VAX machines, the 4 BSD operating system, and the figerd service. However, this exploitation could only be perpetrated by taking advantage of a vulnerability known as buffer overflow or overrun (Erlingsson, 2007).

Buffer overflow (BO) occurs in programs written in languages that do not control the boundaries of arrays during run-time, for instance, FORTRAN, C, and C++ (Spafford, 2003; Viega, Bloch, Kohno, & McGraw, 2000). The goal is to overwrite memory positions to introduce malicious data. The malicious data contain an attack payload, i.e., the malicious code itself, and also a pointer to it. The successful attack does not only copy the malicious data into the memory but also alter the program

DOI: 10.4018/IJSSSP.2018070101

Copyright © 2018, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
execution flow to invoke the attack payload. A BO exploitation allows an attacker to take control of the system. The Morris worm was able to replicate itself because it could attack other computer systems from those whose control were surrendered to it (Spafford, 2003). Another outcome is the crashing of the system, which may lead to Denial of Service (DoS) attacks.

Because BO is associated with the very beginning of the Internet, much research effort has been devoted to address it. One solution is the adoption of memory safe languages such as Java and C# (Horstmann & Cornell, 2005; Drayton, Albahari, & Merrill, 2001), among others. These languages provide mechanisms to control the unintended access to memory positions. For instance, Java and C# check the boundaries of an array when a position beyond the limits are accessed. In these cases, an exception is thrown and the program can take an action to handle it. Another approach is to enforce memory checks in programs written in C/C++ or to add extra code to monitor whether an attack has occurred (Nagarakatte, Zhao, Martin, & Zdancewic, 2009; Ding, He, Wu, Miller, & Criswell, 2012).

With those solutions put in place, BO should be a thing of the past. However, BO exploitations are still common. In July 2012, multiple buffer overflow vulnerabilities were found in the firmware used for VeriFone point-of-sale devices. By exploiting these vulnerabilities, hackers were able to control the terminal and log information input by customers, such as PIN numbers and the magnetic stripe data of a bank card (Constantin, 2012). Among the top security risks for mobile applications (app) tallied in 2016 by the OWASP, “client code quality issues” ranks position #7 (OWASP, 2017). This risk category encompasses vulnerabilities such as buffer overflow, format string, use of insecure or wrong APIs, and insecure language constructs.

Why is an old security issue so persistent? The reason is because C and C++ are still used. Many web and mobile applications rely on application programming interfaces (API) written in “old” languages as C and C++ (Sadeghi, Bagheri, Garcia, & Malek, 2017). If an API is vulnerable, the whole application is as well. Moreover, these languages are still used to develop new applications (Teixeira et al., 2015). C is the second most used language after Java to develop Internet of Things (IoT) applications, according to a recent survey conducted by the Eclipse IoT Working Group and other organizations (Skerret, 2017). The threat to IoT devices is particularly disturbing because they are supposed to take care of people’s intimate activities (Padmanabhuni & Tan, 2015b).

Techniques that provide protection against BO at run-time tend to impose a performance penalty. As result, they might be overlooked when performance is an issue compromising a system. To avoid BO vulnerabilities being neglected, one should detect them during the development of the software. However, the large number and variety of approaches and techniques that have been proposed to deal with buffer overflow vulnerabilities make hard to obtain a good understanding of existing solutions.

Studies such as Erlingsson (2007); Lhee & Chapin (2003); Wilander and Kamkar (2003); Younan, Joosen, & Piessens (2012) provide an overview of dynamic and run-time countermeasures to protect against BO attacks. On the other hand, there are still few studies describing the state-of-the-art, or conducting experimental evaluation of techniques, to detect BO during software development (Pozza, Sisto, Durante, & Valenzano, 2006; Ye, Zhang, Wang, & Li, 2016). In addition, there is evidence that these approaches have being underutilized by security professionals nowadays. Fang and Hafiz (2014) performed an empirical study on 58 reporters of BO vulnerabilities of the securityFocus repository and concluded that almost none of them are in fact used in this context. According to Johnson, Song, Murphy-Hill, & Bowdidge (2013), false positives and developer overload, among other issues, actually may be preventing the utilization of these techniques in practice. Despite that, detection techniques are considered more systematic and scalable, when compared to protection techniques (Pozza, Sisto, Durante, & Valenzano, 2006; Ye, Zhang, Wang, & Li, 2016).

The evidence of the underutilization of BO detection approaches and the lack of studies describing the state-of-the-art motivate the conduction of more comprehensive literature reviews, surveys, comparative analysis, and empirical studies, which are essential to allow a greater recognition and a broader use of these solutions by the industry.
Related Content

Bilateral Histogram Equalization for Contrast Enhancement
Feroz Mahmud Amil, Shanto Rahman, Md. Mostafijur Rahman and Emon Kumar Dey
[www.igi-global.com/article/bilateral-histogram-equalization-for-contrast-enhancement/166541?camid=4v1a](www.igi-global.com/article/bilateral-histogram-equalization-for-contrast-enhancement/166541?camid=4v1a)
Addressing Highly Dynamic Changes in Service-Oriented Systems: Towards Agile Evolution and Adaptation
www.igi-global.com/chapter/addressing-highly-dynamic-changes-service/77704?camid=4v1a

Estimating Interval of the Number of Errors for Embedded Software Development Projects
www.igi-global.com/article/estimating-interval-of-the-number-of-errors-for-embedded-software-development-projects/120089?camid=4v1a

QSE: Service Elicitation with Qualitative Research Procedures
www.igi-global.com/chapter/qse/117924?camid=4v1a