Effectiveness on C Flaws Checking and Removal

João Inácio, Ibéria Medeiros
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
joaommtinacio@gmail.com, imedeiros@di.fc.ul.pt

I. Introduction

The use of software daily has become inevitable nowadays. Almost all everyday tools and the most different areas (e.g., medicine or telecommunications) are dependent on software. The C programming language is one of the most used languages for software development, such as operating systems, drivers, embedded systems, and industrial products. Even with the appearance of new languages, it remains one of the most used [1]. At the same time, C lacks verification mechanisms, like array boundaries, leaving the entire responsibility to the developer for the correct management of memory and resources. These weaknesses are at the root of buffer overflows (BO) vulnerabilities, which range the first place in the CWE's top 25 of the most dangerous weaknesses [2]. The exploitation of BO when existing in critical safety systems, such as railways and autonomous cars, can have catastrophic effects for manufacturers or endanger human lives.

There have had different techniques to detect vulnerabilities, but fuzzing is the most used for its ability to exploit them [3] [4] [5]. However, fuzzing does not give information about them on the code, putting this task on the programmers' side, which can be challenging for those who do not know about security programming. White box fuzzers [6] [7], the combination of them with fuzzing [8] [9], and recently machine learning approaches [10] [11] have been proposed to identify bugs in the code, but they suffer from imprecision, putting once again the effort of checking their output veracity on developer's side. Hence, it is necessary to find ways to automatically detect flaws and remove them by employing more security programming to be helpful for developers. There are some approaches to automatic program repair (APR) for C programs [12] [13]; however, most of them are not for security and the existing ones do not verify the correctness of the fixed code, which can leave the programs syntactically incorrect.

This work introduces an approach for automatically detecting and correcting flaws in C programs. The goal is to provide a pipeline of small modules and tools to discover BOs statically, confirm their presence by fuzzing and remove the vulnerabilities by repairing the code and testing the corrections' effectiveness.

II. Discovering and Fixing BO Approach

We present an approach that identifies and fixes BOs vulnerabilities in the source code of C programs and verifies the effectiveness and correctness of the corrected (fixed) code in an automated manner. The approach has the goal of managing the following challenges: (1) how to find and remove vulnerabilities; (2) what is the right secure code needed to remove them; (3) where to insert this code; (4) how to keep the correct behaviour of the application, after applying the code correction. Our approach addresses these challenges by employing static analysis to find diverse types of BO vulnerabilities, attack injection to confirm the BO found and validate the effectiveness of the code fixed, and fixes to correct the code automatically with fix templates generated dynamically.

Figure 1 illustrates an overview of the approach architecture with the four modules it comprises, which we present next.

1) *BO Finder.* This module is responsible for identifying possible candidate vulnerabilities in the code of the received program. It uses static analysis techniques to collect information about potential vulnerabilities and their location in the program, namely the respective line number in the file. It uses this information to generate slices of the vulnerable code that start at an entry point and end at a sensitive sink.
2) *Executable Generator.* This module generates an executable file for each slice that contains a candidate vulnerability found by the BO Finder. For that, it updates each slice with other instructions needed, from the program files, to obtain a file that can be compiled. Then, the compiled code is instrumented, generating an executable that is forwarded to the Validator.
3) *Validator.* The Validator uses fuzzing techniques for validating the code received from the Executable Generator in two distinct phases. In one phase, it confirms whether the candidate vulnerabilities provided by the finder are real, executing attacks that try to exploit them and generating thus the exploits for them. Those vulnerabilities it cannot exploit are marked as possible false positives (FP). The remaining ones, i.e., the exploitable vulnerabilities, are signalled as such, and their exploits are stored for the second phase. The second

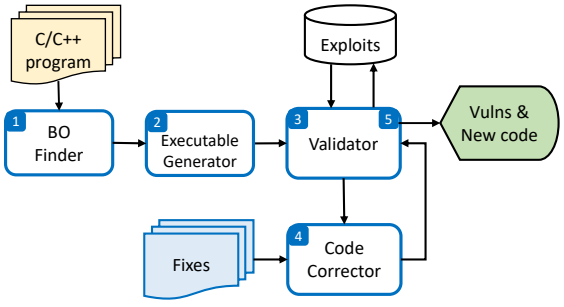

Fig. 1. Overview of the approach architecture.

phase uses the previously generated exploits to verify if the fixes applied are effective and safe. Also, it mutates the exploits to check if there are new exploits that can break the fixes and that the application does not hang.

4) *Code Corrector.* This module analyzes the received code from the Validator (first phase), identifies the existing sensitive sinks (e.g., *strcpy*), and determines the variable sizes of the arguments of the sinks. Next, it checks for the possibility of BOs through the size of the variables used in the sensitive sinks. If it verifies that such vulnerabilities exist, it uses the fix template indicated for that sensitive sink, creates a fix and applies it to the code. Also, it detects whether the code signalled as possible FP or exploitable vulnerabilities were correctly tagged, reporting the former as FP and proceeding with code corrections for the latter. In addition, the corrected code is forwarded to the Executable Generator to produce its executable and then to the Validator to proceed with the second phase of validation. In this validation phase, if the code is found to be correct, a new release of the program is produced, with its files containing the corrected code, i.e., with the vulnerabilities fixed and corrections validated.

We are currently developing a prototype of our approach. The first three modules are already implemented. We resorted to *Flawfinder* [14], and *AFL* [15] to implement the first and third modules. Also, we used the *pycparser* [16] to implement the slice extractor in BO Finder and the third module to parse the program under testing and extract the other code instructions needed to compose a program to be compiled. Also, the interactions between the modules and the correlation of their results were implemented by us in Python. The fourth module is under development, including the fixes templates.

III. Preliminary Results

The objective of the experimental evaluation was to answer the following questions: (1) Can the tool discover potential vulnerabilities and generate their executable slices correctly? (2) Can the tool process the executable slices and exploit the possible vulnerabilities they contain?

We evaluated the tool with 1075 excerpts of C code taken from SARD [17], containing diverse functions related to BOs (e.g., *strcpy, gets, strcat*), and which 560 of them are vulnerable and 515 are not vulnerable. All the excerpts contain more than one data execution flow.

The BO Finder processed all excerpts, and every BO's function they contained was flagged as a possible vulnerability and created 1075 slices for each of them. This means that the 515 not vulnerable excerpts were flagged as possible BOs. We recall that this module is based on Flawfinder, which hits every function it is programmed to discover. Even though this behaviour denotes a high rate of FP, our goal is to check that all sensitive functions are caught and invalidate this result of FPs in the second module. For all slices extracted, we manually verified that they were correctly identified. Note that, Flawfinder only outputs the line of the code of the suspicious function that can cause a BO. So, the slice extractor has the work to extract all lines of code that go from that function until an entry point.

The Executable Generator processed the slices and correctly generated their executables, and the Validator module proved this as it could run all of them. The Validator received 1075 executable files and only exploited 560 of them, i.e., the real vulnerable ones, meaning that it was able to invalidate the FPs.

Besides the Code Corrector not yet completed, it was already capable of correcting more than 50% of the 560 excerpts, and we verified that the corrections made were effective.

The preliminary results showed that it is beneficial in having a pipeline for finding BOs, confirming and correcting them, and testing the new code, which can be helpful for developers.

Acknowledgments. This work was partially supported by the national funds through P2020 with reference to the ITEA3 European through the XIVT project (I3C4-17039/FEDER-039238), and through FCT with reference to SEAL project (PTDC/CCI-INF/29058/2017, LISBOA-01-0145-FEDER-029058, POCI-01-0145-FEDER-029058), and LASIGE Research Unit (UIDB/00408/2020 and UIDP/00408/2020).

References

[1] Tiobe Index, https://www.tiobe.com/tiobe-index/.
[2] 2021 CWE Top 25 Most Dangerous Software Weaknesses, https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.
[3] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Security Symposium*, Aug. 2019, pp. 1967–1983.
[4] P. Oehlert, "Violating assumptions with fuzzing," in *IEEE Security and Privacy*, vol. 3, Number 2, no. 2, March 2005, pp. 58–62.
[5] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling Black-box Mutational Fuzzing," in *Proceedings of the ACM SIGSAC Conference on Computer, Communications Security*, Nov 2013, pp. 511–522.
[6] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, no. 1, pp. 20–27, jan 2012.
[7] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2013, pp. 122–131.
[8] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct 2018, pp. 2095–2108.
[9] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the USENIX Security Symposium (USENIX Security 13)*, Aug 2013, pp. 49–64.
[10] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proceedings of the 33rd Conference on Advances in Neural Information Processing Systems*, Dec. 2019, pp. 10 197–10 207.
[11] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Annual Network and Distributed System Security Symposium*, Feb. 2018.
[12] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.
[13] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys*, vol. 51, no. 1, jan 2018.
[14] "Flawfinder," https://github.com/david-a-wheeler/flawfinder.
[15] "American Fuzzy Lop (AFL)," https://github.com/google/AFL.
[16] "pycparser," https://github.com/eliben/pycparser.

[17] “NIST Software Assurance Reference Dataset Project,” https://samate.nist.gov/SARD/.