ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/priyank-anurag/114

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 2 & Iss: 1

Research Article

Secure Software Development using Fuzzing Techniques

Priyank Rathod* and Anurag

Intel Corporation Folsom, CA, USA

Citation: Rathod P, Anurag. Secure Software Development using Fuzzing Techniques. *J Artif Intell Mach Learn & Data Sci 2024*, 2(1), 411-414. DOI: doi.org/10.51219/JAIMLD/priyank-anurag/114

Received: 01 January, 2024; **Accepted:** 18 January, 2024; **Published:** 20 January, 2024

***Corresponding author:** Priyank Jayantilal Rathod, Intel Corporation Folsom, CA, USA, E-mail: rathodpriyank@gmail.com

ABSTRACT

Fuzzing is a method to try out random input samples to find bugs and vulnerabilities in the software application, where most software testing is limited to unit tests and static and dynamic code analysis. In recent years, after the heartbleed bug was found in a widely used open-source library, it became evident that secure testing is paramount for any software application used anywhere. As people are becoming increasingly reliant on the latest technologies and getting used to working more efficiently than ever, it becomes a necessity rather than a requirement to validate the software in all possible scenarios unimagined by the developer or the validation team. Fuzzing is becoming popular rapidly to secure systems for future attacks and vulnerabilities due to a lack of proper testing and secure code reviews.

Keywords: Secure Software, Fuzzing, Software Testing, Secure Coding, Bug & Vulnerability Detection, System Crash

1. Introduction

The World depends on software now more than ever. As the day goes by, the complexity of software systems and applications is growing exponentially. Growing usage of software would increase the threat and put the systems running this software in harm's way in unimaginable ways. There are many critical systems running these software applications. Fuzzing is a technique where vulnerabilities would be found using various methods, such as malformed or out-of-bound input values. It adds significant value to any software application used in mission-critical systems. Fuzzing generally operates on significant, unpredicted, unexpected inputs to make software applications behave abnormally and either crash or hang the system to identify the weak entry point for any software application. The most common vulnerabilities in any software application are memory-related, which can be exploited to gain unauthorized access to the system and its confidential information.

Borzacchiello et al. [1]propose a novel approximate solver called Fuzzy-Sat for concolic and hybrid fuzzing engines. Traditional consoles and hybrid fuzzing engines rely on Satisfiability Modulo Theories (SMT) solvers to explain the symbolic expressions generated during execution. But, the SMT solvers can be expensive and time-consuming, especially for complex expressions. Fuzzy-Sat addresses this challenge by providing an approximate solution that is faster than SMT solvers while still being able to find bugs effectively. [2]Pham, 2023 describes AFLSmart++ as an extension of the AFLSmart grey box fuzzer that improves its effectiveness and usability. AFLSmart is a structure-aware greybox fuzzer designed for testing programs that take highly structured inputs, such as PNG, PDF, and WAV files. AFLSmart++ improves upon AFLSmart in several ways. First, it introduces new structure-aware mutation operators more effective at generating valid and interesting test inputs. Second, it presents a composite input model that allows AFLSmart++ to handle more complex file formats.

The paper outlines the need for fuzzing methods to ensure the software The development process incorporates these testing methods to mitigate any potential vulnerabilities in software applications in the future. In particular, we will detail the various fuzzing methods, including which method is more effective than others to achieve a higher degree of confidence in any software application.

2. Background

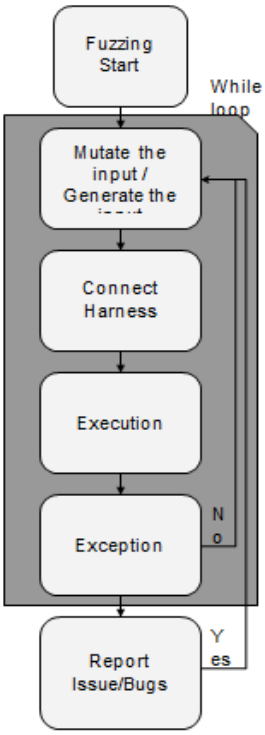
This section briefly explains why secure software development is required for maximum defense against future threats. Many techniques currently used are inadequate to discover these threats beforehand; hence, more advanced techniques, such as fuzzing, are required. The most commonly used techniques are static analysis, dynamic analysis, symbolic execution, and advanced fuzzing, based on various methods and algorithms. Static analysis is one of the fastest methods to scan the source code and sometimes the object code to detect the vulnerability. While it is the quickest method, sometimes it can be a false positive, but the good thing is that all the issues can be addressed without running the code, saving the over-detection time. In contrast, dynamic analysis runs the program and monitors it in real-time to detect the issues. It offers slower speed but higher accuracy and fewer false positives for the code. Often, it includes human interfaces, which require strong technical skills to debug and fix them. Because of human intervention, it is not easily scalable to perform the analysis on larger systems. The symbolic execution method maintains the set of constraints for each execution path used in program inputs. When application execution interacts with components out of The environment, as explained, there would be system calls, APIs, or unreliable signals, and false positives would increase exponentially with the application. It is tricky and time-consuming to scale this type of threat detection. The last technique is the most advanced out of all of the others and has its own merits. It secures the application or system from current and future threats before they can happen.

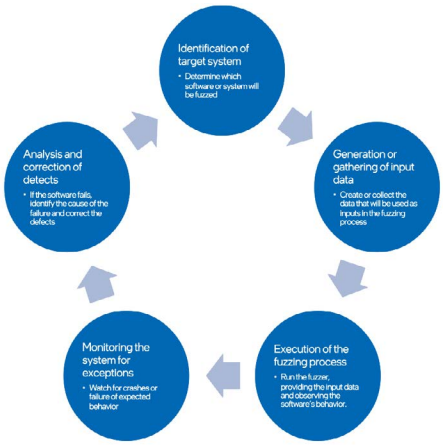
3. Fuzzing

Several fuzzers are used based on their purpose and need. Based on the input, some of them are called generative and mutation fuzzers. Generative fuzzers are there to generate the inputs that rely on the random lines of slightly manipulated data. whereas mutation fuzzers take valid inputs and mutate them to trigger the crash.

Other types of fuzzers are based on the awareness of the different sets of developers or validators. There are three types of fuzzing in this approach. If it is black-box fuzzing in this approach, the software would be viewed as a black box, and testers would not have any idea about the internal workings of the software; still, inputs are generated randomly to make it crash or hang. The second one is white box fuzzing, where the tester or validator is aware of the functionality and inner workings of the software to generate the input to make the software more reliable using the fuzzing. The third type is grey box fuzzing, where the approach is a hybrid, and the tester may have some knowledge about the inner workings, but not all. Hence, it is more like a hybrid.

Fuzzing is a technique where inputs are provided so that the software malfunctions/hangs, or crashes in specific scenarios. The analogy would be throwing whatever input we can to the software entry points and seeing what makes the software behave abnormally or, better, make it crash. The software's input would be unexpected data or symbols, out-of-range values, or something totally out of any expectation input. These inputs would be sent or bombarded to the software to make it malfunction or crash. fundamentally, fuzzing mutates and twists the inputs in some peculiar combinations. It is the most unconventional way to stress-test the software without rules or predefined values.





The aim of running these sorts of stress testing is to expose the issues, bugs, crashes, vulnerabilities, security holes, and other unknown problems otherwise known. It is more like the simulation of real-world attacks before the software is released, so it is more like precautionary testing to build the secure software. While it sounds like a can-do-all tool, it has limitations, such as it won't guarantee that it will be able to find 100% of the bugs in the software. Also is time-consuming, and it would increase based on The complexity of the software is exponential, and last but not least, it would require a secure coding background or expertise to interpret these results and fix them.

4. Literature Survey

D. Kuts [3]presented a method for handling symbolic addresses in dynamic symbolic execution. Symbolic addresses are addresses that are not known at compile time but are instead determined at runtime. This can make it difficult for dynamic

symbolic execution tools to reason about the control flow of a program, as they need to know which memory locations are valid. The method uses a combination of SMT solving and symbolic execution to reason about symbolic addresses. SMT solving is used to determine the bounds of symbolic addresses, and symbolic execution is used to explore the possible values of symbolic addresses. This allows the tool to discover new execution paths that would otherwise be missed. It's a promising new approach to handling symbolic addresses in dynamic symbolic execution. It can improve the tool's accuracy and find new bugs in programs. [2]Pham presented AFLSmart++, an extension of the structure-aware grey box fuzzer AFLSmart. AFLSmart++ improves AFLSmart in two main aspects: structure-aware low-level mutation and composite input model. These tools make low-level mutations to data chunks, fundamental building blocks for input data in any software application. At the same time, a composite input model provides a structured way for inputs to the software application. [4]present Driller, a hybrid vulnerability excavation tool that combines fuzzing and selective symbolic execution to find deeper bugs in software. Traditional fuzzing is good at exploring large portions of the program's state space, but it can struggle to find bugs that require satisfying complex path conditions. Symbolic execution, on the other hand, can be effective at finding such bugs, but it can be expensive and time-consuming to run on large programs. Driller addresses this challenge by using fuzzing to identify exciting paths in the program's state space and then using symbolic execution to generate inputs that satisfy the complex conditions on those paths. This allows Driller to find bugs that would be missed by traditional fuzzing or symbolic execution alone. Generally, a driller is a powerful bug excavation tool that can find bugs rooted deep within software applications.

C. Zhang et al. [5]have presented a method to extract the features of a specific software application and use them to recommend the methods to fuzz in the particular application targeted. The critical thing to remember is before removing the features from any software application, the developer needs to determine which features are required, and based on the requirement gathered from the developer or security researcher, the features are selected to select the appropriate fuzzer for the application[6]. proposed a malicious code image feature extraction method based on entropy filtering. Introducing an entropy filter helps identify the hidden patterns introduced by specific packers and encryptors; therefore, this method performs better than the previous method. Due to the rise of deep learning technology and its wide application in the field of vulnerability detection, more and more feature extraction will choose to use deep neural networks. [7]proposed DL4MD. They introduced the stack autoencoder (SAE) model into malicious code analysis for the first time to realize unsupervised malicious code feature extraction. [8]Popov used convolutional neural networks (CNN) to extract code features. Automatic coding technology based on feature learning and recurrent neural networks based on classifiers is used to realize feature extraction of malicious code. At the same time, there are some automatic tools to extract the characteristics of the target program, and most of them use artificial intelligence technology.

J. Shao et al. [9]proposed a reinforcement learning-based fuzzing approach that can effectively improve the input and achieve higher code coverage. It would divide the fuzzing process into two stages: the stage where bytes causing the crash are identified and the The RL-based approach would apply mutation operators to these bytes to maximize the detection rate[10] studied the BUGOSS, a real-world benchmark for regression bugs found in the OSS-Fuzz. It is designed to be used to evaluate regression fuzzing techniques. Regression fuzzing techniques are fuzzing techniques that are specifically designed to find regressions, which are bugs that are introduced in a new version of a program. The BUGOSS benchmark can evaluate the effectiveness of regression fuzzing techniques by comparing their ability to find the bugs represented by the study artifacts[11] use reinforcement fuzzing, a newer approach that uses the learnings from past inputs to learn how to provide the inputs that would likely find bugs in the software. Reinforcement fuzzing is effective in finding bugs in a variety of software programs. However, there are several challenges to using reinforcement fuzzing, including the fact that it can be difficult to design good rewards and that it can be computationally expensive to learn policies. The benefits would be more efficient and faster than other normal fuzzers.

5. Conclusion

In conclusion, many methods exist to identify and secure vulnerabilities in the code. Fuzzing is one of the most advanced techniques, with its pitfalls, but it still has more coverage than any other method. It is also proven to find any issues skipped during the code reviews or the unit testing of the code base. It also ensures that software works in all possible attack scenarios that would otherwise be catastrophic. It would be more effective if it could be used in any continuous integration environment where all the checked-in code is scanned for potential issues. Along these lines, there are some fuzzers that are highly advanced in detecting bugs using reinforcement-based learning to find bugs by predicting potential inputs that would cause the program to crash faster. So, in final remarks, fuzzing is a way forward, and using novel approaches surveyed in the paper, it is evident that fuzzing would essentially become part of the development process at some point.

6. References

1. Borzacchiello L, Coppa E, Demetrescu C. Fuzzing Symbolic Expressions. 2021 IEEE/ACM 43rd ICSE, 2021; 711-722.
2. Pham V-T. AFLSmart++: Smarter Greybox Fuzzing. 2023 IEEE/ACM International Workshop on SBFT 2023; 76-79.
3. Kuts D. Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution. 2021 IVMEM 2021; 42-49.
4. Stephens N, Grosen J, Salls C, et al. Driller: Augmenting fuzzing through selective symbolic execution. NDSS '16 2016.
5. Zhang C, Chen J. Fuzzing Methods Recommendation Based on Feature Vectors. 2021 36th IEEE/ACM International Conference on Automated Software Engineering 2021; 1079-1081.
6. Dey A, Bhattacharya S, Chaki N. Byte label malware classification using image entropy: volume eight. Proceedings of the 2017 IEEE International Conference on Advanced Computational and Communication Paradigms 2019; 17-29.
7. Hardy W, Chen L, Hou S, Ye Y, Li X. DL4MD: A deep learning framework for intelligent malware detection. Proceedings of the 2016 Int'l Conf. Data Mining 2016; 61-67.
8. Popov I. Malware detection using machine learning based on word2vec embeddings of machine code instructions. Proceedings of the 2017 SSDSE 2017; 1-4.
9. Shao J, Zhou Y, Liu G, Zheng D. Optimized Mutation of Grey-box Fuzzing: A Deep RL-based Approach. 2023 IEEE 12th DDCLS 2023; 1296-1300.

10. Kim J, Hong S. BugOss: A regression bug benchmark for empirical study of regression fuzzing techniques. 2023 IEEE Conference on Software Testing, Verification and Validation 2023; 470-473.

11. Böttinger K, Godefroid P, Singh R. Deep reinforcement fuzzing. 2018 IEEE Security and Privacy Workshops 2018; 116-122.