Utilizing debug information to improve error-finding methods in the context of hybrid fuzzing

Larionov-Trichkine Theodor Arsenij
Information Security
National Research University "Higher School of Economics"
Moscow, Russia
tlarionovtrishkin@edu.hse.ru

***Abstract*—The use of fuzzing as a technique for identifying software vulnerabilities has become increasingly popular in recent years. While traditional fuzzing methods can be effective, they are inherently limited in their ability to find certain types of errors.**

Hybrid fuzzing tries to overcome these limitations by leveraging the power of symbolic execution to generate inputs that can explore different paths in a program. However, the effectiveness of hybrid fuzzing can be hindered by imprecise boundaries calculations for symbolic memory accesses. This research project aims to address this issue by utilizing debug information to improve error-finding methods in the context of hybrid fuzzing.

The core idea of the project is to enhance the calculation of boundaries for symbolic memory accesses by leveraging debug information. With more accurate boundaries, local out-of-bounds bugs can be identified, and overall performance can be improved. The research will explore the feasibility of using debug information for boundaries calculation in the context of hybrid fuzzing and evaluate its effectiveness.

The project's methodology will involve implementing and testing the proposed approach on several open-source software projects. The results of this research will provide insights into the potential of using debug information to enhance hybrid fuzzing and could pave the way for more effective techniques in identifying software vulnerabilities.

***Index Terms*—Hybrid fuzzing, symbolic execution, symbolic pointers, symbolic addresses, symbolic memory model, debug information, error-finding, software security testing.**

I. Introduction

With the rapid development of information technologies, the number and complexity of software systems have increased drastically. This has led to an increase in the number of software vulnerabilities as well as an increasing need for secure software development practices.

One area of concern in software security is memory safety, particularly in the context of programming languages. Memory-unsafe languages, such as C and C++, are known to impose significant vulnerabilities due to their low-level nature and lack of automatic memory management [1], [2]. These vulnerabilities can range from buffer overflows to use-after-frees and beyond, and often lead to full system compromise.

In order to address these issues, a variety of techniques have been developed, including static and dynamic analysis. While static analysis is a powerful tool for identifying vulnerabilities, it has its limitations. To complement static analysis and improve the overall effectiveness of vulnerability detection, dynamic analysis techniques emerged.

Dynamic analysis often referred to as fuzzing, involves an automated generation of input data to test the behavior of a program at runtime.

The field of fuzzing has been actively developing for years, and nowadays, there is a variety of fuzzing techniques and tools available:

- AFL++ [3] is a popular fuzzing tool that uses a genetic algorithm and coverage feedback to generate new inputs.
- Superion [4] and Nautilus [5] are grammar-based fuzzers that use hand-crafted grammar to generate inputs with a specific structure.
- Sydr [6] and Fuzzolic [7] are examples of hybrid fuzzers that combine the power of symbolic execution with the speed of fuzzing.

All approaches are feasible, however, some of them require additional resources and time. That's where hybrid fuzzing shines. By using symbolic execution, it can generate structurally correct inputs that can explore different paths in a program without requiring any additional effort from the user.

Consider the following example:

```
1  void vuln(int key) {
2      if (key * 0x142a2d == 0xdeadbeef) {
3          error();
4      }
5  }
```

Listing 1. Example solvable by hybrid fuzzing

With traditional fuzzing, it is almost impossible to randomly generate an input that will trigger the error in example 1. However, hybrid fuzzing methods allow us to generate a crash input by building a symbolic expression that represents the original input and then solving the constraints. This approach helps to overcome the limitations of traditional fuzzing and can be used to find a variety of bugs.

A hybrid fuzzing tool that I'm going to work with in this research is Sydr [6]. Sydr is a dynamic symbolic execution engine based on Triton [8] that is used to solve the programs' constraints and generate inputs that can explore new program states. Besides the ability to generate new inputs, Sydr also uses a novel technique called "Security Predicates" [9] to purposefully trigger error conditions in a program under test.

Sydr has a lot of features, and one of the most interesting ones is the ability to handle symbolic memory accesses, which in itself is a very complex task [10]. This feature allows Sydr to model symbolic memory accesses and, as such, greatly improve the effectiveness of hybrid fuzzing. However, the current implementation of symbolic pointers is not perfect. Boundaries calculations for symbolic memory accesses are imprecise, which can lead to incorrect results and hinder the effectiveness of hybrid fuzzing.

The goal of this research project is to improve the calculation of boundaries for symbolic memory accesses by leveraging debug information. Debug information allows to find precise boundaries for a lot of different types of memory accesses. For example, it can be used to find the exact size of local and global variables or heap objects. With more accurate boundaries, a few aspects of the Sydr can be improved:

- The effectiveness of Security Predicates because, with more accurate boundaries, more types of bugs can be found (e.g. local out-of-bounds bugs).
- The overall speed and accuracy, because smaller memory regions put less strain on an SMT-solver.

II. Literature review

III. Methods

IV. Anticipated results

V. Conclusion

The software security field is constantly evolving, being in a state of race between security researchers and hackers.

References

[1] "Chromium project memory safety report," https://www.chromium.org/Home/chromium-security/memory-safety/, accessed: 2023-02-12.

[2] "Android project memory safety report," https://source.android.com/docs/security/test/memory-safety, accessed: 2023-02-13.

[3] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[4] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," 2018. [Online]. Available: https://arxiv.org/abs/1812.01197

[5] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Nautilus: Fishing for deep bugs with grammars," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[6] A. V. Vishnyakov, A. Fedotov, D. O. Kuts, A. A. Novikov, D. Parygina, E. Kobrin, V. Logunova, P. Belecky, and S. F. Kurmangaleev, "Sydr: Cutting edge dynamic symbolic execution," *CoRR*, vol. abs/2011.09269, 2020. [Online]. Available: https://arxiv.org/abs/2011.09269

[7] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: Mixing fuzzing and concolic execution," *Comput. Secur.*, vol. 108, no. C, sep 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102368

[8] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications*, ser. SSTIC, 2015, pp. 31–54. [Online]. Available: https://triton.quarkslab.com/files/sstic2015_slide_en_saudel_salwan.pdf

[9] A. V. Vishnyakov, V. Logunova, E. Kobrin, D. O. Kuts, D. Parygina, and A. Fedotov, "Symbolic security predicates: Hunt program weaknesses," *CoRR*, vol. abs/2111.05770, 2021. [Online]. Available: https://arxiv.org/abs/2111.05770

[10] D. O. Kuts, "Towards symbolic pointers reasoning in dynamic symbolic execution," *CoRR*, vol. abs/2109.03698, 2021. [Online]. Available: https://arxiv.org/abs/2109.03698