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, previously undiscovered 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:

```c
void vuln(int key) {
    if (key * 0x142a2d == 0xdeadbeef) {
        error();
    }
}
```

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. This feature allows Sydr to model symbolic accesses to memory and, as such, greatly improve the effectiveness of hybrid fuzzing. Consider the following example:

```
1  uint16_t crc_ibm_table[256] = {
2      0x0000, 0xc0c1, 0xc181, 0x0140, ...
3  };
4
5  uint16_t
6  crc_ibm_byte(uint16_t crc, const uint8_t c)
7  {
8      uint8_t sym_idx = (crc ^ c) & 0xFF;
9      return crc_ibm_table[sym_idx] ^
10         (crc >> 8);
11 }
12
13 if (crc_ibm_byte(0, buf[0]) == 0x1337) {
14     error();
15 }
```

Listing 2. Access to memory via a symbolic pointer

In the example 2, the program accesses memory via a symbolic pointer while calculating the checksum for a user-provided data **buf**. This means that the value of a symbolic pointer/index **sym_idx** depends on the input data. As a result, the program can access any element inside the **crc_ibm_table** array and this relation should be taken into account when building a symbolic expression for the program's constraints. Without correctly describing the memory access, the symbolic execution engine will not be able to generate an input that will lead to the error condition on line 14.

Current implementation of the symbolic memory model in Sydr is described in the paper: [10]. It proposes a wide range of novel techniques to handle symbolic memory accesses. However, it still has a lot of room for improvement. For example, the current implementation of symbolic pointers uses a simple heuristic to determine the boundaries for a symbolic memory access. However, this heuristic is not always accurate and might lead to incorrect results. Which in turn can hinder the effectiveness of hybrid fuzzing.

The aim of this research project is to enhance the computation of symbolic memory access boundaries through the utilization of debug information. Debug information enables the identification of precise boundaries for various memory access types, such as local and global variables or heap objects. By obtaining more accurate boundaries, the Sydr can be improved in several ways:

- The effectiveness of Security Predicates can be enhanced as a result of the ability to detect additional types of bugs, such as local out-of-bounds bugs.
- The overall speed and accuracy of the Sydr can be improved since smaller memory regions exert less pressure on an SMT-solver.

II. Literature review

As was already mentioned, the support for indirect memory dependencies is a crucial feature for any effective hybrid fuzzing tool. There are several approaches to handle symbolic memory accesses, and each of them has its own pros and cons. In this section, I will briefly describe the most popular approaches to model symbolic memory and in particular how to calculate the boundaries for symbolic pointers.

The problem of precise boundaries computation for symbolic memory is not new. First mentions and some approaches were mentioned in the KLEE paper from 2008 [10]. Since then, a lot of research has been done in this area, and the problem has been studied from different perspectives. In this section, I will briefly describe the most relevant approaches to symbolic pointers.

3 MEMORY MODEL [11]

V. INDEX-BASED MEMORY MODELING [12]

A. Boundaries Approximation [10]

[13]

III. Methods

1) Debug info
2) Heap metadata

Experiments:

$$\begin{aligned} \text{Acc}_i = {} & |[\text{Begin}_{\text{default}}, \text{End}_{\text{default}}]| - \\ & |[\text{Begin}_{\text{dbg}}, \text{End}_{\text{dbg}}]| \bigcap \\ & [\text{Begin}_{\text{default}}, \text{End}_{\text{default}}]| \end{aligned}$$

$$\text{Acc} = \frac{\sum_{i=1}^{n} \text{Acc}_i}{n} \tag{1}$$

$$\text{Error}_{\text{left}} = \frac{\sum_{i=1}^{n} |\text{LeftDbg}_i - \text{LeftDefault}_i|}{n} \tag{2}$$

$$\text{Error}_{\text{right}} = \frac{\sum_{i=1}^{n} |\text{RightDbg}_i - \text{RightDefault}_i|}{n} \tag{3}$$

IV. Anticipated results

The results anticipated are based on already implemented techniques and the outcomes of preliminary experiments, as well as analysis of the existing literature. The following improvements are expected:

- Enable more precise computation of symbolic memory access boundaries.
- The effectiveness of Security Predicates will be enhanced as a result of the ability to detect additional types of bugs, such as local out-of-bounds bugs.
- The overall speed and accuracy of the Sydr can be improved since smaller memory regions exert less pressure on an SMT-solver.

It should be noted, that some of the anticipated results are already achieved, but were not thoroughly evaluated or implemented. For example, a fragment of the proposed technique is already implemented as a part of the Sydr's Security Predicates and allows to detect local out-of-bounds bugs.

Eventually, after the final implementation of the proposed approach will be completed, the effectiveness of the Sydr will be evaluated on a set of real-world programs. The results will be presented in the final report.

V. Conclusion

The software security field is constantly evolving, being in a state of race between security researchers and hackers. Therefore, it's important to always stay up-to-date and detect new vulnerabilities before they are exploited.

In this paper we presented an approach to greatly improve capabilities of hybrid fuzzing tool Sydr by utilizing debug information. We not only proposed a new technique to compute symbolic memory access boundaries, but also implemented it and evaluated its effectiveness.

It should be noted that the proposed approach has several notable advantages over the existing techniques:

- It greatly improves the accuracy of symbolic memory access boundaries calculation compared to heuristic and constant offset approaches.
- The performance of the method is remarkably better compared to the SMT-based approaches.

These enhancements promise a significant boost in the efficacy of error-detection methodologies, ultimately resulting in a reduction of exploitable vulnerabilities in the wild.

Further work on this topic can be done in several directions:

- Use debug information to find types of integers (signed/unsigned). With this information, it is possible to improve some aspects of "Security Predicates" feature.
- Add support for recursive structure parsing. This will allow us to detect more bugs, such as out-of-bounds accesses inside structures. Besides, it will refine boundaries for symbolic pointers.
- Add better support for C++. Parse classes and their fields.
- Improve support for optimized builds by expanding the range of locations that can be processed. In addition to single-address or base-pointer offset location descriptions, it is possible to use more complex expressions, such as stack-pointer offsets.
- Use known structure of an object to further optimize formulas for the SMT-solver.

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

[11] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, may 2018. [Online]. Available: https://doi.org/10.1145/3182657

[12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 2012, pp. 380–394. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SP.2012.31

[13] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dblp.uni-trier.de/db/conf/osdi/osdi2008.html#CadarDE08