Kernel Fuzzing

Mahbub Raton Nicholas Baron Benjamin Rutkowski

April 2024

1 Introduction

Kernel fuzzing is a testing strategy where an automated tool generates random input and feeds it to the kernel under the test. The goal is to discover the unspecified behaviors of the kernel. The responsibility of the kernel is to act as an interface between processes and hardware. It is quite inconceivable to find bugs in the kernel by manual inspection because of the sheer complexity. Kernel fuzzing comes to the rescue in this case. In this project, we will review some state-of-the-art techniques for kernel fuzzing. We will mainly focus on the Linux kernel since most of the research efforts are focused on Linux.

2 Motivation

The most important part of any system for testing is its interface, as that is the area where all internal errors will escape and attacks from malicious agents are most frequently made. However, since many system calls allow arbitrary data to be passed in the form of pointers to strings or structs, the attack surface is rather large. Additionally, many operations require multiple system calls. For example, reading a file properly requires at least three (`open`, `read`, and `close`), with the `read` call often being repeated if a buffer is too small or to allow the file to be loaded from disk in smaller chunks.

A common technique to explore large input spaces is fuzzing, which is (at its simplest) using random bytes as input. Many fuzzers use a heuristic of new paths discovered to help determine better inputs, allowing them to explore the control flow under test. For fuzzing a kernel, the best fuzzer would be able to mutate the order of calls and the values (both the identifying number and the other parameters) passed in. The fuzzer would thus simulate a very erratic program and show that a kernel is resilient against such unusual behavior. A more intelligent fuzzer (one with some knowledge of syscalls) would be able to give more "pointed" inputs and act more like an adversary.

3 List of Papers

A contributing factor to the difficulty of effective kernel fuzzing is the problem of dependencies, [1] thoroughly formalizes and quantifies this problem. Another problem that must

be addressed is that of statefulness; i.e., many behaviors within the kernel depend on being is a specific state of global variables. HFL [3] implements kernel fuzzing while handling undetermined behavior caused by varying states and dependencies. The tool MoonShine [6] was developed to create seeds of kernel fuzzers by using syscall traces from real programs. Kyungtae Kim *et al.* [7] use hypervisors and Intel hardware to accelerate kernel fuzzing in an OS-independent method called kAFL.

The tool Razzer [2] searches for race conditions within the kernel and can be used to generate relevant fuzzing. This is implemented using a hypervisor. Seulbae Kim *et al.* [4] uses Hydra to perform fuzzing on the kernel's file system. Dokyung Song *et al.* [10] develop PeriScope, a tool that leverages the kernel's page fault handler, and uses fuzzing to find security and bug risks within the kernel's device drivers.

We list a collection of important tools and methods for fuzzing (that are not necessarily locked to kernel fuzzing). Hyper-Cube [8] is used to test hypervisors. The tools SPFuzz [9] and Steelix [5] are used for stateful fuzzing. Lastly, Vasudev Vikram *et al.* [11] presents a method called Bonsai Fuzzing, which generates meaningful test cases when the system has a well-defined syntax and semantics; the application in the paper was to test compilers.

This paper[12] talked about the race condition bug in the kernel.

References

[1] Yu Hao et al. "Demystifying the dependency challenge in kernel fuzzing". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 659–671.

[2] Dae R Jeong et al. "Razzer: Finding kernel race bugs through fuzzing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 754–768.

[3] Kyungtae Kim et al. "HFL: Hybrid Fuzzing on the Linux Kernel." In: *NDSS*. 2020.

[4] Seulbae Kim et al. "Finding semantic bugs in file systems with an extensible fuzzing framework". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 147–161.

[5] Yuekang Li et al. "Steelix: program-state based binary fuzzing". In: *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 2017, pp. 627–637.

[6] Shankara Pailoor, Andrew Aday, and Suman Jana. "{MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 729–743.

[7] Sergej Schumilo et al. "{kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels". In: *26th USENIX security symposium (USENIX Security 17)*. 2017, pp. 167–182.

[8] Sergej Schumilo et al. "HYPER-CUBE: High-Dimensional Hypervisor Fuzzing." In: *NDSS*. 2020.

[9] Congxi Song et al. "SPFuzz: a hierarchical scheduling framework for stateful network protocol fuzzing". In: *IEEE Access* 7 (2019), pp. 18490–18499.

[10] Dokyung Song et al. “Periscope: An effective probing and fuzzing framework for the hardware-os boundary”. In: *2019 Network and Distributed Systems Security Symposium (NDSS)*. Internet Society. 2019, pp. 1–15.

[11] Vasudev Vikram, Rohan Padhye, and Koushik Sen. “Growing a test corpus with bonsai fuzzing”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 723–735.

[12] Meng Xu et al. “Krace: Data race fuzzing for kernel file systems”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1643–1660.