Abstract

Alice In Kernel Land: Lessons Learned From the eBPF rabbit hole

Extended Berkeley Packet Filter (eBPF) is a technology that provides capabilities to programmers seeking to make use of kernel layer performance and functionality. Fundamentally, eBPF allows users to load programs into kernel space and attach them to hook points. This allows for loading kernel code at runtime without needing to modify the kernel source code itself or develop a kernel module.

eBPF programs are written in a high-level language and then compiled into assembly- like bytecode. At load time, the bytecode is JIT-compiled into the native architecture which allows for the program to be kernel and architecture-independent. The instruction set is minimal and subject to safety constraints but allows programmers to call outside kernel functions, read and store data in various data structures and perform pointer arithmetic and operations.

Programs that run in the kernel must be carefully analyzed to ensure that these programs follow rules to guarantee the integrity and security of the kernel running the program. A single code flaw in any of the components involved in program parsing, analysis, optimization, and compilation may lead to a compromise of the kernel running an eBPF implementation.

As eBPF becomes more prevalent, the goal of our talk is to share the history of eBPF vulnerabilities, bug classes, mitigations and provide an outlook for the future. We will also share our insights into automated vulnerability discovery. It will introduce listeners to advanced concepts of structured fuzzing such as designing and implementing an Intermediate Language. We will also discuss identifying roadblocks such as bug detection and give practical examples on how to overcome them. This will enable anyone to apply these concepts to their own fuzzing campaigns. The source code of our fuzzer will also be made available.

Presentation outline

Introduction to eBPF
- Syscall Interface
- Verifier
    * Security Considerations

JIT compiler

Runtime
- Helper functions
- Hook points

Dissecting vulnerabilities in each component (Roadmap)
- Verifier
  * Memory corruptions in parsing code
    ^ Range calculation bugs
  * CVE-2020-27194
- Branching prediction
  * CVE-2016-2383
- General logic bugs
  * CVE-2021-3490
- JIT compiler
  * Optimization -> Branch miscalculations
    ^ CVE-2021-29154
  * Code generation bugs
  * Translation from classic BPF to eBPF
    ^ CVE-2021-38300
  * Architectural differences
- Runtime
  * Helper functions
    ^ CVE-2021-38166

Automating Vulnerability Discovery
- Why?
  * Code changes
  * eBPF is a standard -> implemented on different platforms
- How?
  * Fuzzer Architecture
    ^ Intermediate Language Design
    ^ Runtime environment (Fuzzing in userland and kernel)
- Instrumentation
- Generation
- Mutation
- Detecting bugs

Presentation allocations

First speaker Chompie (10-15mins)

- Introduction to eBPF
- Context as to why it is still relevant
    - “Frame” talk about history of eBPF
- Bug classes

Second speaker Simon (max 10mins)

- Introduction to fuzzing
    - Why it makes sense
- Talk about first generation / version of eBPF fuzzer
- Talk about shortcomings of it and why it didn’t work

Third speaker JJ (10mins)

- Second version of Fuzzer
- How it addressed shortcomings of first version
- Architecture, Design etc