Seeing the Invisible: Auditing eBPF Programs in Hypervisor with HyperBee

Yutian Wang, Dan Li, Li Chen
Outline

• Motivation
  • Offensive capabilities of eBPF
  • Difficulty in detecting eBPF malware

• HyperBee Design
  • HyperBee-aware guest
  • Verifier and compiler chains

• Evaluation

• Summary
Offensive capabilities of eBPF

• Offensive capabilities
  • Reading & Modifying memory
  • Hijacking Syscall and kfunc
  • Sniffing & Tampering packets
    • XDP (more stealthy)
  • Privilege Escalation
  • Denial-of-service
  • Remote Code Execution

• PoC eBPF malwares
  • BadBPF
  • eBPFKit
  • BoopKit
  • TripleCross

• eBPF malware in wild
  • BPFFDoor (found by PwC)
  • Bvp47 (found by Pangu Lab)
  • Symbiote (found by BlackBerry R&I)
Difficulty in detecting eBPF malware

• Few places in the system can be trusted once the eBPF malware is implanted into the kernel.
  • Monitoring programs can be misled by falsified information.
  • Persistence of system logs can be blocked.
  • Alarms may not be sent through the network.

• The best solution is to audit outside the OS!
  • (Before the eBPF program is actually loaded into the kernel)
Outline

• Motivation
  • Offensive capabilities of eBPF
  • Difficulty in detecting eBPF malware

• HyperBee Design
  • Design Goals
  • HyperBee-aware guest
  • Verifier and compiler chains

• Evaluation
• Summary
Design Goals

• Comprehensiveness
  • All eBPF programs to be loaded into the kernel are audited by HyperBee.

• Transparency
  • eBPF programs and their user space controllers do not require any changes.

• Efficiency
  • HyperBee has no performance impact on the execution of eBPF programs.

• Portability
  • Can be implemented in Linux, Windows and others.
HyperBee-aware guest

• Original section about verification and JIT compilation in the kernel is replaced by calls to HyperBee.
• Compiled bytecode is written into the kernel space after verification.
• The initial state of the kernel is guaranteed through Secure Boot.
• Transparent to user space controllers.
Verifier and compiler chains

- Chain selector selects different verifier and compiler chains for different Guest OSs.
- An eBPF program is compiled and returned to the kernel only after it has been fully verified.
- Any failures are logged and notified to the kernel as well as external auditing systems.
Verifier in chains

• Primary Verifier
  • There must be a primary verifier in the chain, which should usually be the first one.
  • Usually the same verifier as the original kernel implementation of the guest OS.

• Security Policy Verifier
  • Restrictions on instructions, helper function calls, eBPF program types and attach types.

• Signature Verifier
  • For signed eBPF programs. Can be cooperated with Security Policy Verifier.

• Other Verifiers
Outline

• Motivation
  • Offensive capabilities of eBPF
  • Difficulty in detecting eBPF malware

• HyperBee Design
  • HyperBee-aware guest
  • Verifier and compiler chains

• Evaluation

• Summary
Evaluation

• Prototype
  • HyperBee: Based on KVM in Kernel 5.10
    • HyperBee-P: With only primary verifier.
    • HyperBee-SP: With primary and security policy verifier.
  • HyperBee-aware guest: Based on Kernel 5.10

• Dataset
  • Positive samples
    • BCC v0.27 and Kernel 5.10 source tree.
  • Negative samples
    • BadBPF, eBPFKit, BoopKit, and TripleCross.

• Results
  • 9% extra load time when there is no security policy.
  • 17% extra load time when using security policies against known eBPF malicious programs.
Summary

• Motivation:
  • Fighting eBPF malware inside the OS is difficult.

• HyperBee:
  • Auditing eBPF programs in Hypervisor before they are loaded into the guest kernel.
  • Using verifier and compiler chains.
  • Transparent to user space controllers.

• HyperBee improves cloud security.