Linux Virtualization Based Security (LVBS)

Linux Plumbers Conference 2023

Thara Gopinath, Mickaël Salaün, James Morris
Agenda

- Architecture Overview
- Hyper-V Implementation
- KVM Implementation (Heki)
- Q&A
Architecture Overview

James Morris
Use virtualization to provide enhanced security for the guest OS, leveraging the hypervisor security boundary.
Protect the integrity of security-critical guest structures.
Prevent bypass of guest security mechanisms and policies.
Linux Virtualization Based Security (LVBS)

Support a Trusted Execution Environment (TEE) for running security applications

- Executable code integrity
- Key management
- Others
• Bring mainline Linux to state of the art.

• Linux is trailing proprietary solutions across Linux and other OSs.

• Attacks continue to evolve, along with motivation levels.
LVBS Architecture

Open-source architecture for Linux.

Independent of:

- ISA
- Hypervisor
- VMM
- Security monitor
- TEE implementation
Approach

- Mainline acceptance across ecosystem is critical to success

- Reference implementation (HEKI):
  - KVM
  - Linux kernel API
  - Flexible kernel hardening policy

- We are seeking feedback and collaboration.
Hyper-V Implementation

Thara Gopinath
Hyper-V based System

The diagram shows a Hyper-V based system with the following components:

- **Root Partition**
  - User Space
  - Kernel

- **Guest Partition**
  - User Space
  - Kernel

The system is divided into rings, with ring 0 being the highest privilege level and ring -1 at the lowest. The Hyper-V layer is at the bottom, providing virtualization to the hardware.
Virtual Secure Mode (VSM)

Separate privileged execution environment within a partition: Virtual Trust Level (VTL)
VSM Features

- Virtual Processor state isolation
- Memory access hierarchy and protection
- Virtual Interrupt and Intercept handling

LVBS and VSM

- Kernel Hardening
  - Hardening memory permissions (HVCI)
  - Monitoring critical system registers and MSRs
  - Monitoring critical kernel data structures

- Offloading policies (control flow integrity, authentication)

- Offloading secure services (trustlets)

Initial Target: Basic Kernel Hardening.
Threat Model:

Kernel Hardening

- Protect kernel from a user space attacker exploiting a kernel vulnerability
- Assume that the attacker has arbitrary read write access to guest kernel thanks to exploited vulnerability by malicious
  - User space process
  - Network Packet
  - Block Device
- Secure Boot is trusted.
- Defence in Depth; but no extra features!!!
H/W Requirements:

- Second Level Address Translation (SLAT, Two-Dimensional Paging, AMD's RVI/NPT)
  - Enable to manage VM memory and add a secondary complementary layer of permissions only controlled by the hypervisor

- CPU features that allow to differentiate between kernel space and user space memory (MBEC)
Architecture:
(Common Layer)

e.g. Hypervisor Enforced Kernel Integrity (Heki)
Architecture:

Secure Kernel

- Small TCB
- Maintainability
- Ability to support secure interfaces

- Initial choice for secure kernel: Minimal Linux Kernel
Architecture: Control Interfaces

Synchronous: Explicit VTL Call and Return

Asynchronous: Interrupt based entry and exit

Higher VTL gets precedence over lower VTL
Architecture: Boot

- We trust secure boot!
- VTL0 guest kernel boots up VTL1 secure kernel
- Establish kernel hardening and other policies with secure kernel prior to init process.
Architecture:

Boot Sequence

- vCPU0 boots in VTL1
- Enable VTL1 for secondary VPs
- vCPU0 signals vCPUx
- vCPUx boots in VTL1

VTL0 (Guest Kernel)

- ... rest_init
do_initcall
vsm_boot driver
Enable VTL1 on vCPU0

Step 1: vCPU0: Boot vCPU0
vCPU0: vtl1 return

Step 2: vCPU0: Enable VTL1 for secondary VPs
vCPU0: vtl1 return

Step 3: vCPU0: Boot secondary VPs

Step 4: vCPUx: Boot vCPUx
vCPUx: vtl1 return
vCPUx: vtl1 return
Architecture:

The Big Picture (Boot)
Architecture:

The Big Picture
(Late Boot)
Architecture

(Access / Policy Violation)
- https://github.com/heki-linux/lvbs-linux/tree/secure-kernel-lvbs
KVM Implementation (Heki)

Mickaël Salaün
Architecture:
The Big Picture
Sent RFC v2:

• Guest kernel implementation of the common API
• Two new KVM hypercalls: CR-pinning and memory permission
• KVM interface with the VMM: dedicated VM exits and related capabilities
Enforce a bitmask on control registers to guard against locked features (e.g. SMEP)

kvm_hypercall3(KVM_HC_LOCK_CR_UPDATE, 0, // control register
X86_CR0_WP, // flag to pin flags); // options

Can create a VM exit on configuration or policy violation for the VMM to be able to do something.

Generate a GP fault on policy violation.
Configure (a subset of) EPT permissions.

kvm_hypercall1(KVM_HC_PROTECT_MEMORY, pa);  // address of a pagelist

The pagelist atomically maps a set of memory ranges with read, write and execute permissions.

Generate a **synthetic page fault** on policy violation.
Executable permission(s)

Issue: efficiently enforce restriction on kernel executable pages without impacting access to user space pages

Solution: leverage Intel’s Mode Based Execution Control (MBEC)

Split the execution permission into:
- Kernel mode execution
- User mode execution
Kernel memory permissions without MBEC

- __end_rodata (read-only)
- vdso_end (read-only)
- vdso_start (read-only)
- ___start_rodata (read-only)
- _etext (read-execute)
- _text (read-execute)

0x0000... 0xFFFF...
Kernel memory permissions with MBEC

- __end_rodata: read-only
- vdso_end: non-executable
- vdso_start: non-executable
- __start_rodata: read-execute
- _etext: non-executable
- _text: non-executable
https://github.com/heki-linux/linux
branch heki-v2
Wrap up

KVM and Hyper-V supports:
• defense-in-depth mechanism leveraging hardware virtualization
• common API layer across hypervisors

Any feedback?

https://github.com/heki-linux
Q&A
Thank you
Demo: control-register pinning (SMEP)
static void heki_test_cr_disable_smege(struct kunit *test)
{
    unsigned long cr4;

    /* SMEP should be initially enabled. */
    KUNIT_ASSERT_TRUE(test, __read_cr4() & X86_CR4_SMEP);

    kunit_warn(test, "Starting control register pinning tests with SMEP check\n")

    /*
     * Trying to disable SMEP, bypassing kernel self-protection by not
     * using cr4_clear_bits(X86_CR4_SMEP).
     */
    cr4 = __read_cr4() & ~X86_CR4_SMEP;
    asm volatile("mov %0,%%cr4" : "+r"(cr4) : : "memory");

    /* SMEP should still be enabled. */
    KUNIT_ASSERT_TRUE(test, __read_cr4() & X86_CR4_SMEP);
}