Microsoft

Hypervisor-Enforced Kernel Integrity for Linux, powered by KVM – RFC v2

Linux Plumbers Conference

Mickaël Salaün & Madhavan Venkataraman

2023-11-14

Protect Linux users against kernel exploits

Attackers can leverage security vulnerabilities using kernel exploits to get access to all users' data.

Kernel integrity

Security property guaranteeing critical parts of a virtualized Linux kernel to not be tampered by malicious code or data.

Problem

One kernel vulnerability is enough to bypass kernel integrity.

300+ Linux CVEs in 2022

including 8 code executions

Overview

Main threat model

Trust can diminish over time

The attacker has arbitrary read and write access to the guest kernel e.g., thanks to exploited vulnerability by malicious

- User space process
- Network packet
- Block device

E.g., DirtyCOW (CVE-2016-5195)

Extended threat model

We’d like to not trust (too much) the host (VMM)

pKVM should be kept in mind, along with **confidential computing**

Leverage virtualization

The main issue with kernel self-protection is that it is a **self**-protection.

We'd like to rely on a **higher privileged component**: the hypervisor

State of the art

- grsecurity/PaX, OpenBSD
- Windows's Virtualization Based Security (i.e., **VBS**, HVCI, HyperGuard…)
- Samsung RKP, Huawei Hypervisor Execution Environment
- iOS KPP/Watchtower and KTRR/RoRgn
- …and a lot of PoC

Design

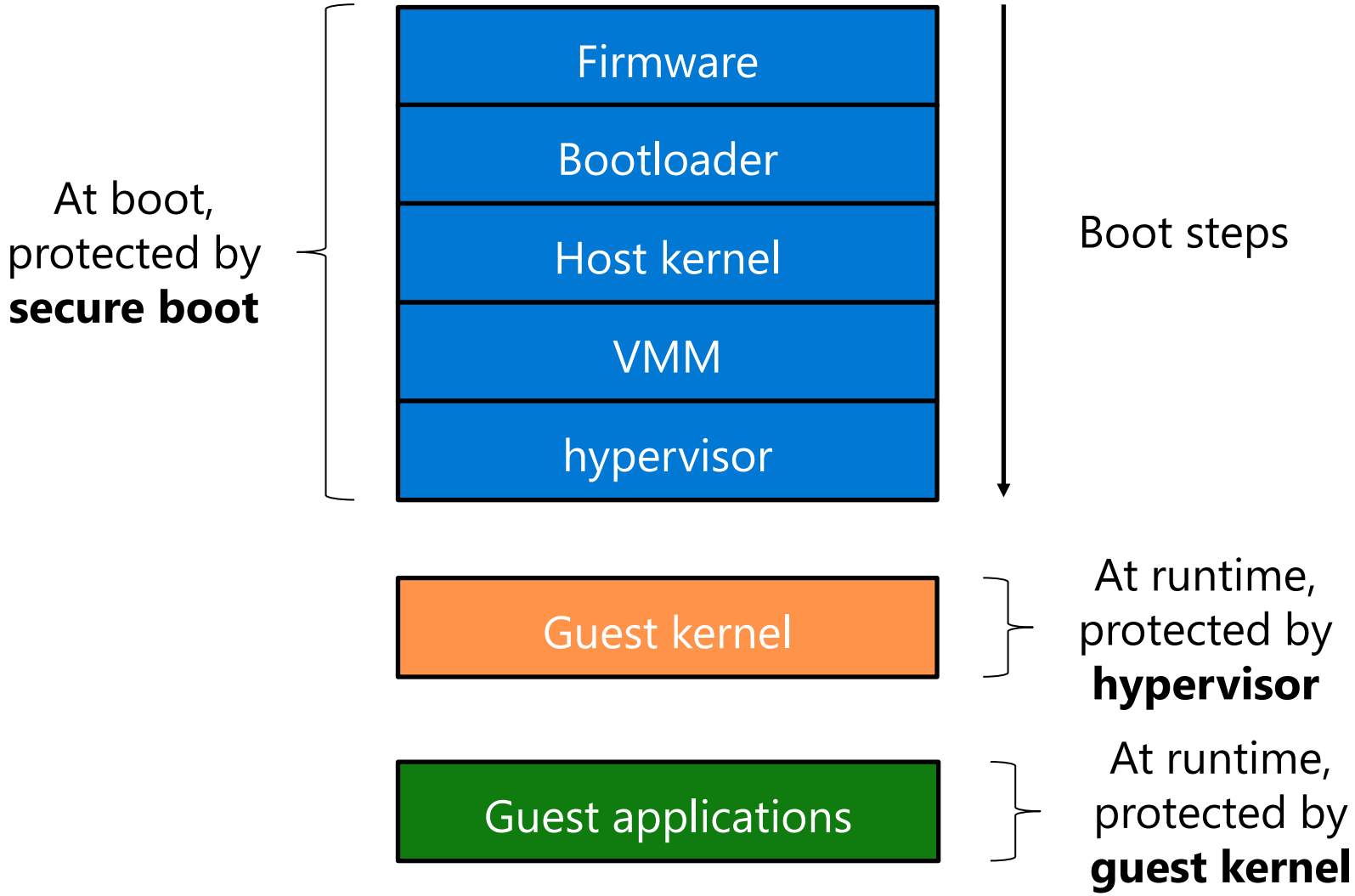
Properties:

- The guest VM configures itself
- The hypervisor manages enforcements
- The VMM protects its resources and get attack signals

Usability:

- Users manage their own kernels
- This feature needs to be simple to use and standalone: almost no configuration

Chain of trust



Security policies

Improve Linux kernel hardening:

- Enforce critical **CPU register pining**: CR0.WP, CR4.SMEP…
- Enforce **read-only** and **non-executable** kernel data (e.g., syscall table, certificates, keys, security configuration)

VM lifetime in a nutshell

VMM:

- Assign memory pages to a new VM
- Run the VM
- Handle VM exits (e.g., emulation)

VM boot time:

- Map memory pages with permissions
- After some variable updates, set them read-only

VM run time:

- Load kernel modules or eBPF programs, use ftrace or kprobes…

RFC's main changes

From RFC v1 to RFC v2:

- From static to dynamic memory permissions thanks to a new memory table: kernel modules, eBPF…
- No more enforced *execute XOR write* for now, we'll get back on that later
- Leveraging the new per-page attributes patch series
- New KVM interface: 2 new types of VM exits and related capabilities
- New hypercall flag to get supported features

KVM implementation

CR-pinning hypercall

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.

Memory permissions

Part of hardware virtualization, the Second Layer Address Translation or Two Dimensional Paging:

- Intel's EPT
- AMD's RVI/NPT

Enable to manage VM memory, and add a second **complementary layer of permissions**, only controlled by the hypervisor.

Memory protection hypercall

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)

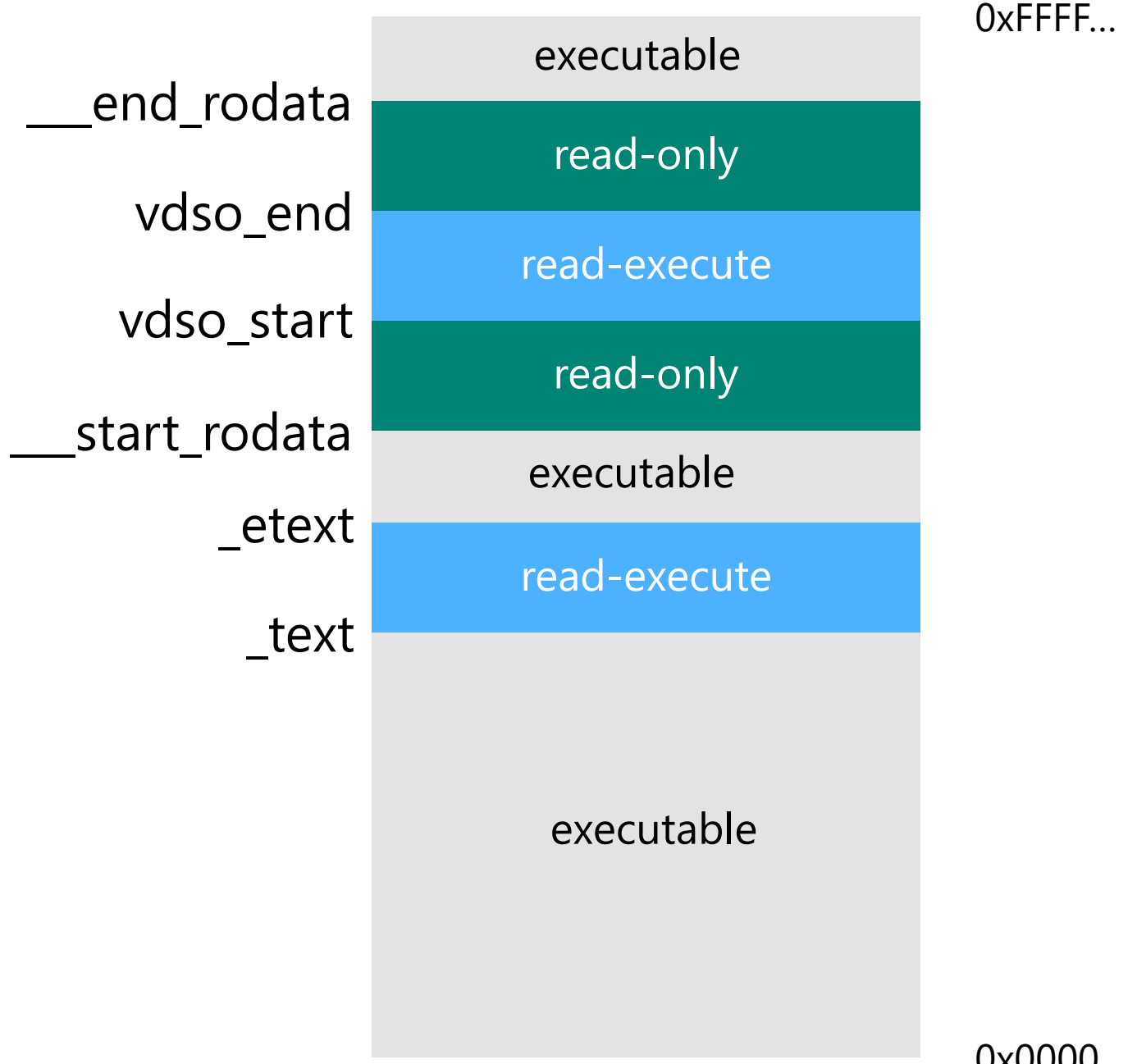
Issue: efficiently enforce restriction on kernel executable pages without impacting access to user space pages

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

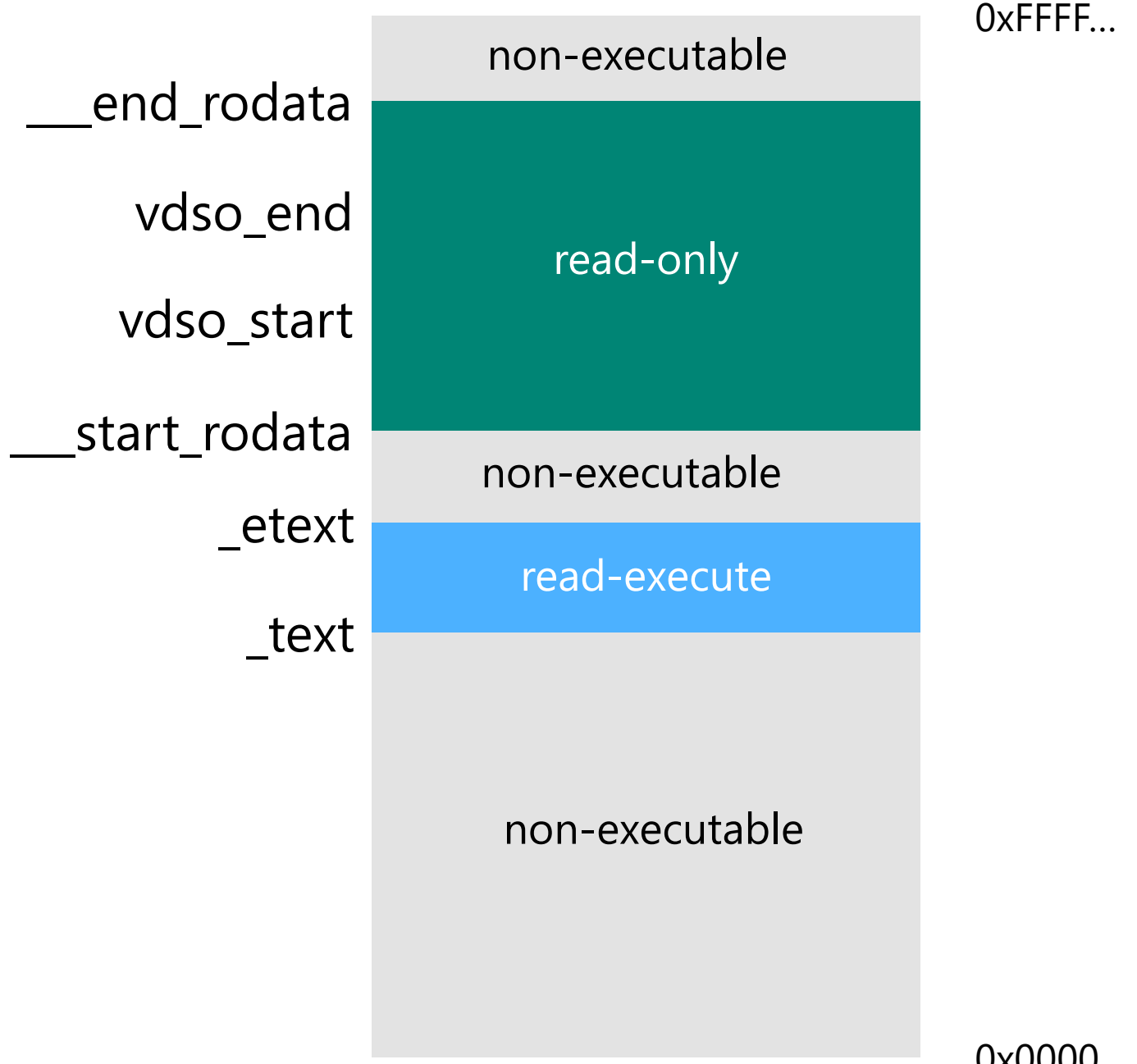
Split the execution permission into:

- Kernel mode execution
- User mode execution

Kernel memory permissions without MBEC



Kernel memory permissions with MBEC



Guest kernel API and implementation

Guest API

Normalized common layer that can be used by any hypervisor to receive guest requests:

- Map kernel memory pages with required permissions or attributes
- Hide hypervisor implementation details (e.g., hypercalls)
- Shared test suites

Memory table and address space walker

Objective: enforce the **union of permissions** for a physical page across **multiple mappings**.

Solution:

- **Generic memory page table** to reflect the hardware page table format
- Walk kernel mappings within a range
- Map permissions counters to each mapped page (read, write, exec) as needed
- Pages never mapped will have no counters and get the default read-write permissions
- Allows for sparse representation and large page entries

Dynamic modification of memory permissions

Kernel features relying on dynamic memory permission update:

- Module loading and unloading
- Static call and jump label optimization
- ftrace/livepatch
- Kprobes/optprobes
- eBPF JIT

Patched helpers to **track permission changes**:

- vmap()/vunmap() et al.
- set_memory_x()
- text_poke()

Kernel code authentication

Requirement for secure systems, but not implemented yet. Some leads:

- Use kernel module signatures: rely on the guest keyring (in a secure way)
- Signed eBPF programs: nothing yet ☹
- Somehow check legitimate kernel code patching. Any idea?

This would need to be delegated to either the **VMM** or a **sidecar VM**.

Demo: control-register pinning (SMEP)

```
static void heki_test_cr_disable_smep(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);
                                                      62,2-9        25%
```

```
user@heki-host$
```

Demo: static kernel memory protections (noexec)

```
long heki_test_exec_data(long);
void _test_exec_data_end(void);

/* Used to test ROP execution against the .rodata section. */
/* clang-format off */
asm(
".pushsection .rodata;" // NOT .text section
".global heki_test_exec_data;"
".type heki_test_exec_data, @function;"
"heki_test_exec_data:"
ASM_ENDBR
"movq %rdi, %rax;"
"inc %rax;"
ASM_RET
".size heki_test_exec_data, .-heki_test_exec_data;"
"_test_exec_data_end:"
".popsection");
/* clang-format on */

static void heki_test_cr_disable_smep(struct kunit *test)
                                                38,1          14%
```

```
user@heki-host$
```

Demo: dynamic kernel memory protections (kernel module)

user@heki-host$

user@heki-host$

Current limitations

- Authentication is not in place: WIP
- ROP protection is out of scope: need to rely on CFI
- Overhead during init: WIP to improve the design
- Should permissions counters be in the guest?

Future work

Securely handle dynamic code execution allowed by an external entity doing the code authentication:

- The VMM, or
- A dedicated sidecar VM (cf. VBS's VTL, or COCONUT Secure VM Service Module)

Figure out how to verify intrinsic features like ftrace and Kprobes.

Going mainline

What should be the next step?

Proposal: stabilize and merge the CR-pinning patches, bringing the foundation for memory protections

Wrap up

Heki is a defense-in-depth mechanism leveraging hardware virtualization.

The Linux RFC defines a common API layer across hypervisors (e.g., Hyper-V): see tomorrow's talk about **LVBS** (Refereed Track)

Test and contribute!

We’re looking for contributions!

- New hypervisors support
- New architecture support
- Improved guest kernels support
- VMM enhancements

https://github.com/heki-linux