LVBS: ADVANCED KERNEL INTEGRITY

Thara Gopinath , Madhavan Venkataraman

LINUX SECURITY SUMMIT EUROPE

Motivation

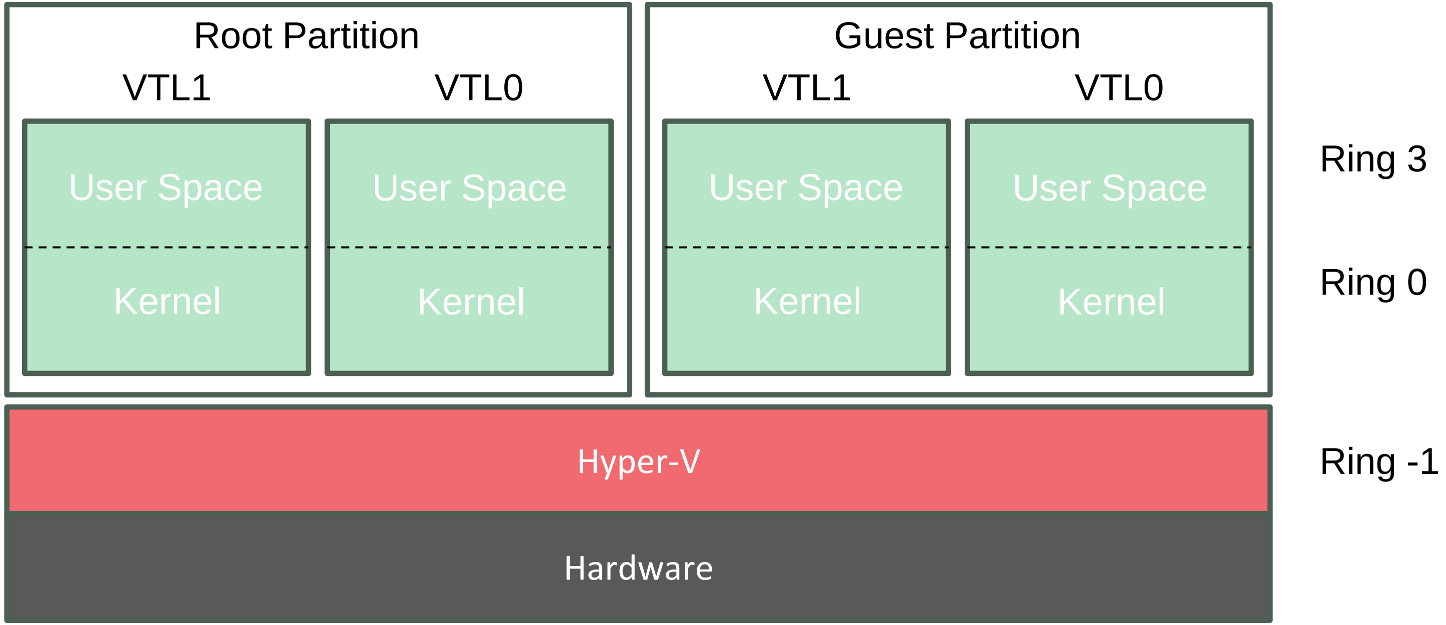
- Linux Kernel vulnerabilities have been steadily rising and getting exploited in the wild
- Our goal is to:
  1. Harden the kernel by enforcing protections, which cannot be turned off by a malicious kernel
  2. Ensure that critical system assets (keys, critical kernel data structures) are inaccessible and/or untampered, even if the kernel gets compromised

LVBS

- Linux Virtualization Based Security (LVBS) is inspired by Windows VBS and uses the hypervisor and HW virtualization support to protect the guest OS.
- Open-source architecture
  - HW agnostic
    - Currently working on AMD & Intel x64
  - Hypervisor agnostic
    - Currently working on Hyper-V & KVM
    - This presentation will focus on Hyper-V implementation

Hyper-V’s Virtual Secure Mode (VSM)

- Hyper-V introduces separate execution environments within a partition, called Virtual Trust Levels (VTLs).
- Higher VTLs have more privilege over lower VTLs. VTL0 is the least privileged.



VSM Features

- Virtual Processor state isolation
  - Virtual Processors maintain separate, per-VTL state. For example, each VTL defines a set of private VP registers.
- Memory access hierarchy and protection
  - Each VTL maintains a set of guest physical memory access protections.
  - Higher level VTLs can impose memory restrictions to lower level VTLs.
  - Access protections are implemented by the hypervisor via Extended Page Table (EPT) or Second Level Page Table (SLT).
- Virtual Interrupt and Intercept handling
  - Each VTL has its own interrupt subsystem (local APIC). This ensures that higher VTLs process interrupts without interference from lower VTLs.
- https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/vsm

LVBS: Relook at Goals

- Kernel Integrity (kernel memory and critical register protection)

- Protecting critical system resources (passwords , secrets etc)

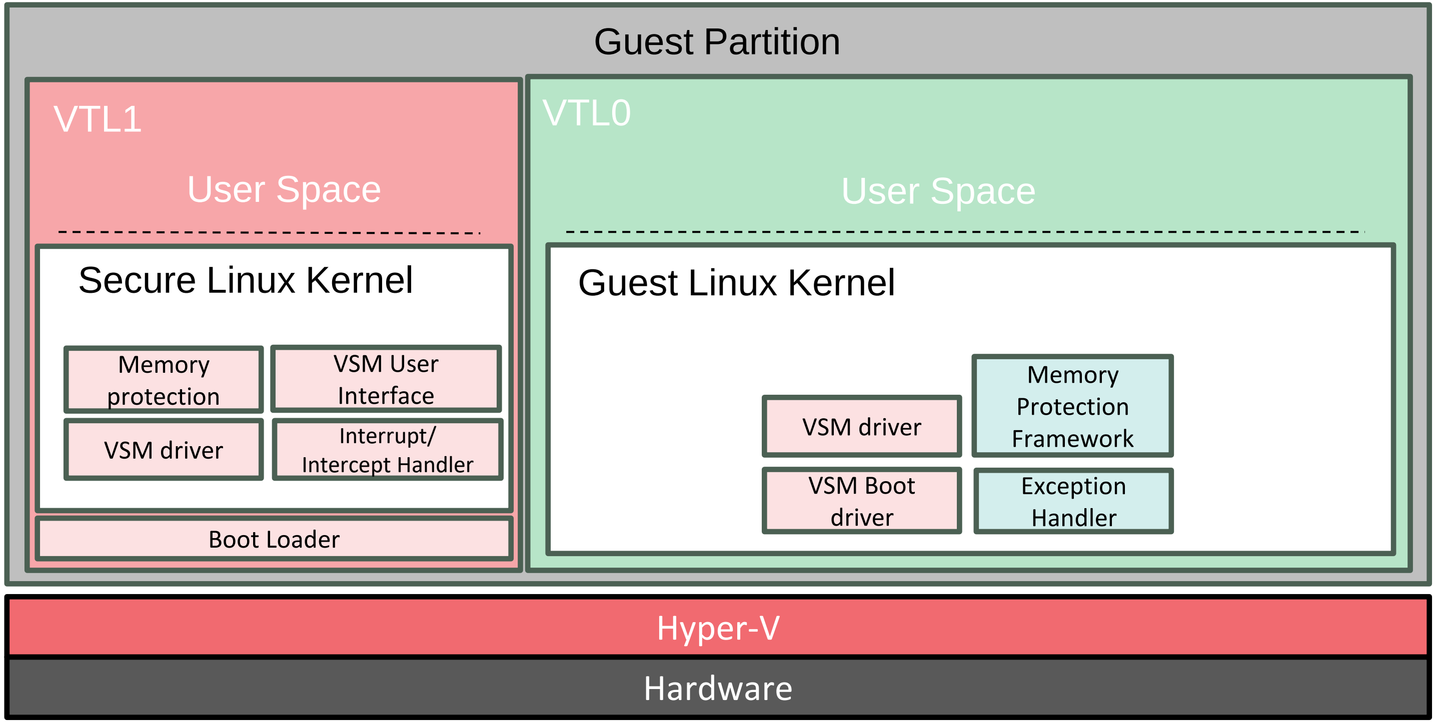
Kernel Integrity : Threat Model

- Security Goal: Protect kernel from a user space attacker exploiting a kernel vulnerability.

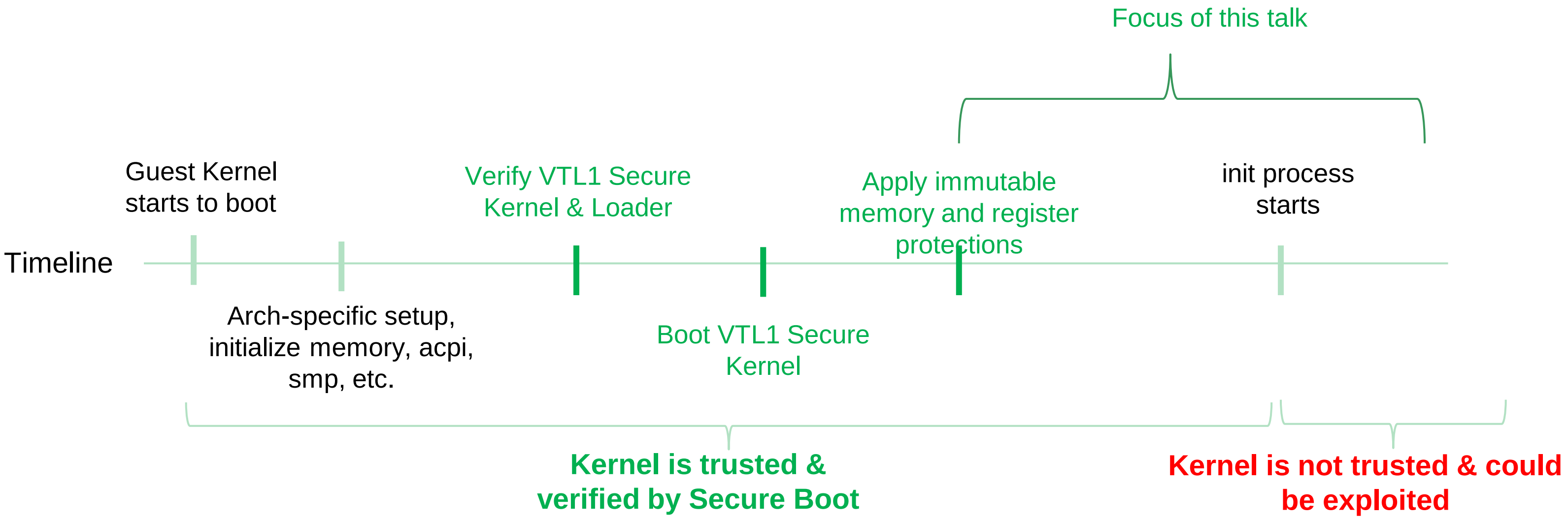
- TCB : Hyper-V and if applicable host withing TCB.

- Defense in depth.

LVBS : High-Level Architecture



LVBS : Quick Look at Boot Sequence



LVBS: VTL0 – VTL1 Interface (1/2)

- Currently, there is no process that runs in VTL1 continuously.

- VTL1 is entered when a VP switches from VTL0 to VTL1. This can happen via:
  1. VTL call: Guest kernel (VTL0) makes an explicit hypercall to invoke VTL1

Category	VTL Call Opcode
VTL1 Boot	VSM_VTL_CALL_FUNC_ID_ENABLE_APS_VTL, VSM_VTL_CALL_FUNC_ID_BOOT_APS
Apply protections	VSM_VTL_CALL_FUNC_ID_PROTECT_MEMORY VSM_VTL_CALL_FUNC_ID_LOCK_CR VSM_VTL_CALL_FUNC_ID_SIGNAL_END_OF_BOOT
Load VTL0 Data	VSM_VTL_CALL_FUNC_ID_LOAD_KDATA
Module Loading	VSM_VTL_CALL_FUNC_ID_VALIDATE_MODULE VSM_VTL_CALL_FUNC_ID_FREE_MODULE_INIT

Kernel integrity functions

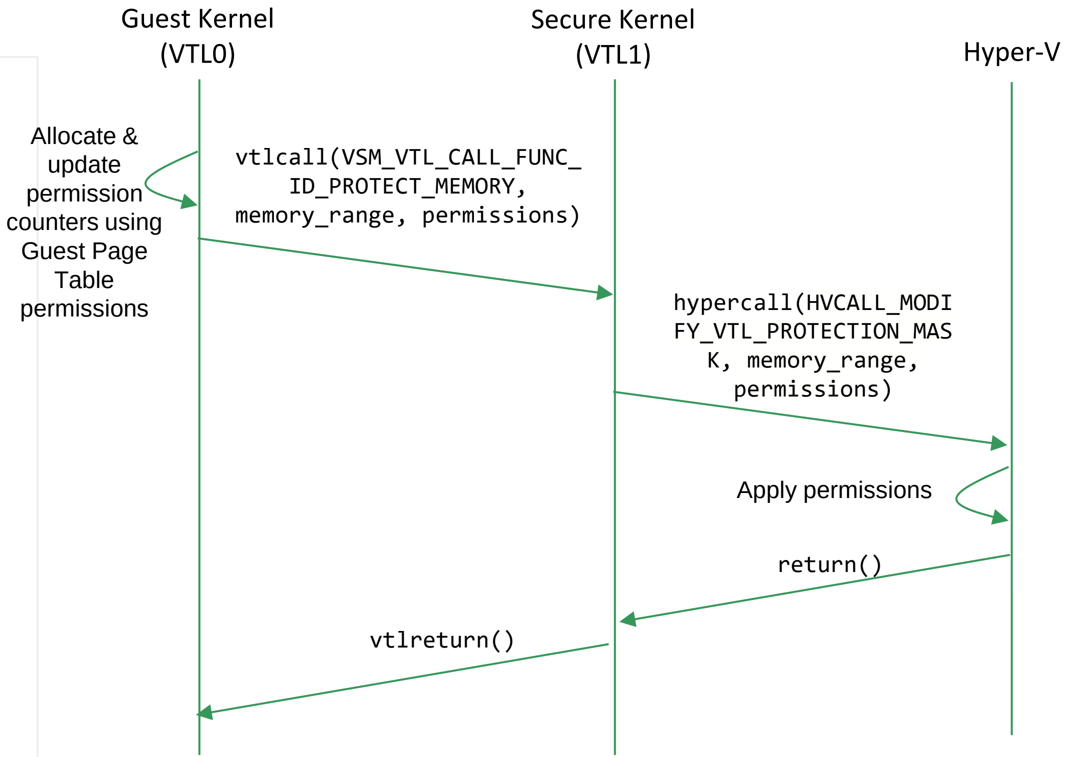
LVBS : VTL0 – VTL1 Interface (2/2)

2. Secure interrupt: If an interrupt is received for a higher VTL, the VP will enter the higher VTL
    - After VTL1 has booted, we disable timer interrupts while the VP is in VTL0. This greatly reduces jitter.

3. Secure intercept: When VTL0 violates VTL1 protections, the VP that triggered the fault enters VTL1

LVBS : Memory Protections

- Guest Kernel uses Memory Protection Framework, introduced by Hypervisor-Enforced Kernel Integrity (HEKI)
    - [RFC PATCH v2 00/19] Hypervisor-Enforced Kernel Integrity
    - Common between Hyper-V & KVM
    - Allocates permission counters for each 4KB page in Guest Kernel
    - Find cumulative permissions using Guest Page Table permissions
        - Read, write, execute (uX / kX using MBEC)
    - Tracks subsequent permission changes



Memory Protections(2/2)

Just before init, a singe VTLCall sets the following immutable permissions:

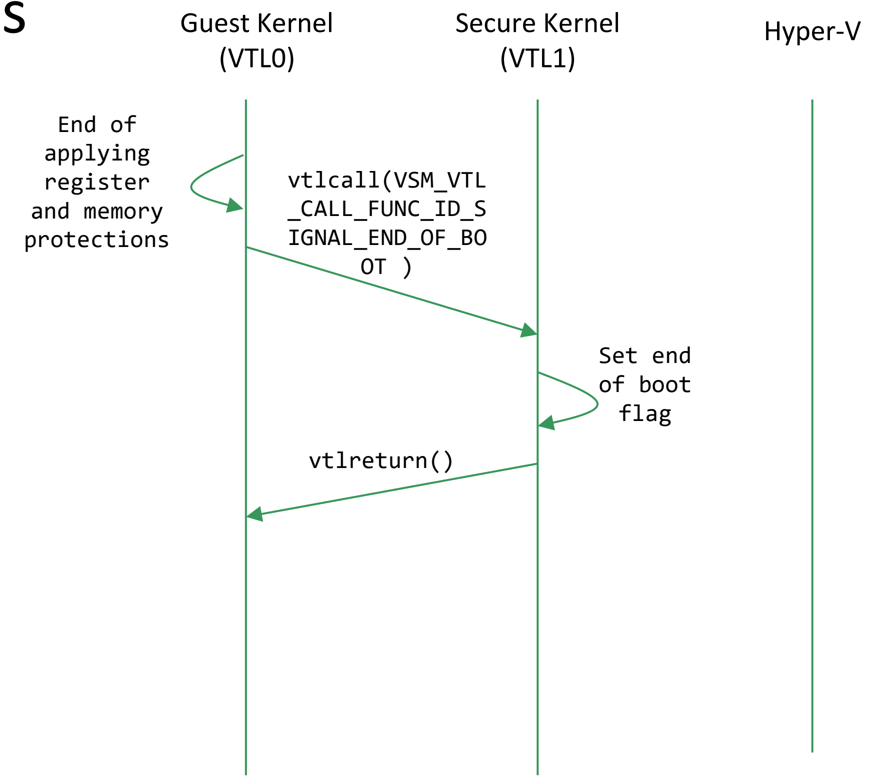
Region	Permission
rodata	Read only
Text	Read Kernel Execute
VTL1 memory space	No access
Rest of the kernel memory	Read Write User Execute

Default permissions, without LVBS, where Read, Write, Kernel Execute, User Execute

- If there is an EPT access violation, a memory intercept is injected to VTL1, which raises a GP fault in VTL0.
- If there is a subsequent request from VTL0 to change permissions of an immutable region, the request is ignored. Future Work: Add bookkeeping and report.

LVBS: Lock Down Protections

- Set end of boot flag in VTL1
- Following vtlcalls are no longer valid
  - VSM_VTL_CALL_FUNC_ID_ENABLE_APS_VTL VSM_VTL_CALL_FUNC_ID_BOOT_APS
  - VSM_VTL_CALL_FUNC_ID_PROTECT_MEMORY
  - VSM_VTL_CALL_FUNC_ID_LOCK_CR
  - VSM_VTL_CALL_FUNC_ID_LOAD_KDATA
- Attempt to invoke them returns error.

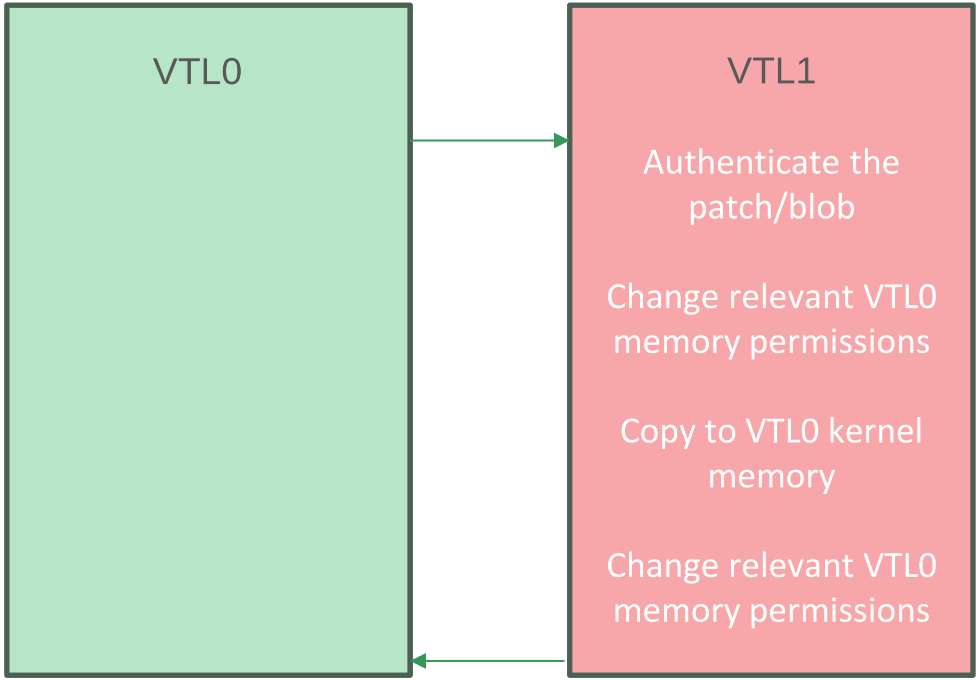


LVBS : A look at kernel memory space

Section	Permission enforced via LVBS	Text patching feature	Operation resulting in EPT violation
text	RX	Ftrace Live patching Jump Label Optimization Static call optimization Kprobes	RX -> RWX -> RX
rodata	R		
Data, .bss	RW		
module loading reserved memory	RW	Module loading	RW->R/RX
crash kernel reserved memory	RW	Kexec	RW->RWX
Rest of kernel memory space	RW	Ebpf-jit	RW->RX

LVBS: Text Patching Design Principle

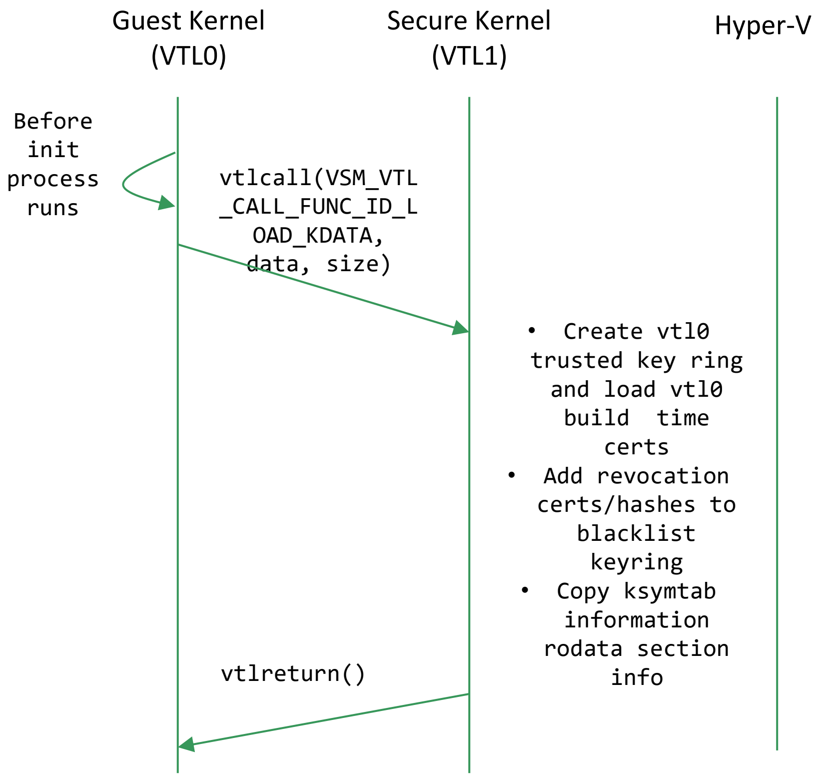
- Authenticate/Verify the text patching
- Copy to memory location
- Change permissions
- One atomic operation



VTL0 Kernel Data

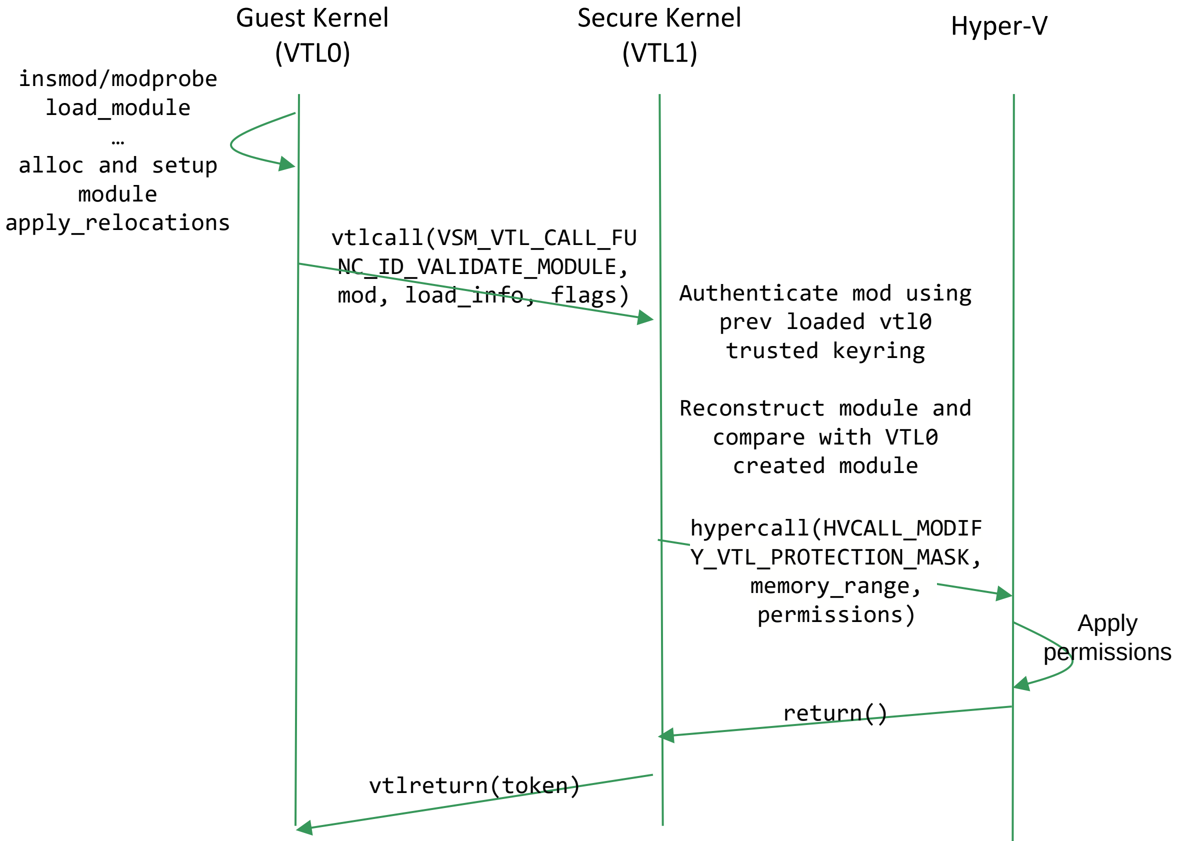
- Build/Boot time certificates/keys
- Revocation(blacklisted) certificates/keys
- Kernel symbol table sections (ksymtab & ksymtab_gpl)
- Read only Data section (rodata)

- Called before VSM_VTL_CALL_FUNC_ID_SIGNAL_END_OF_BOOT



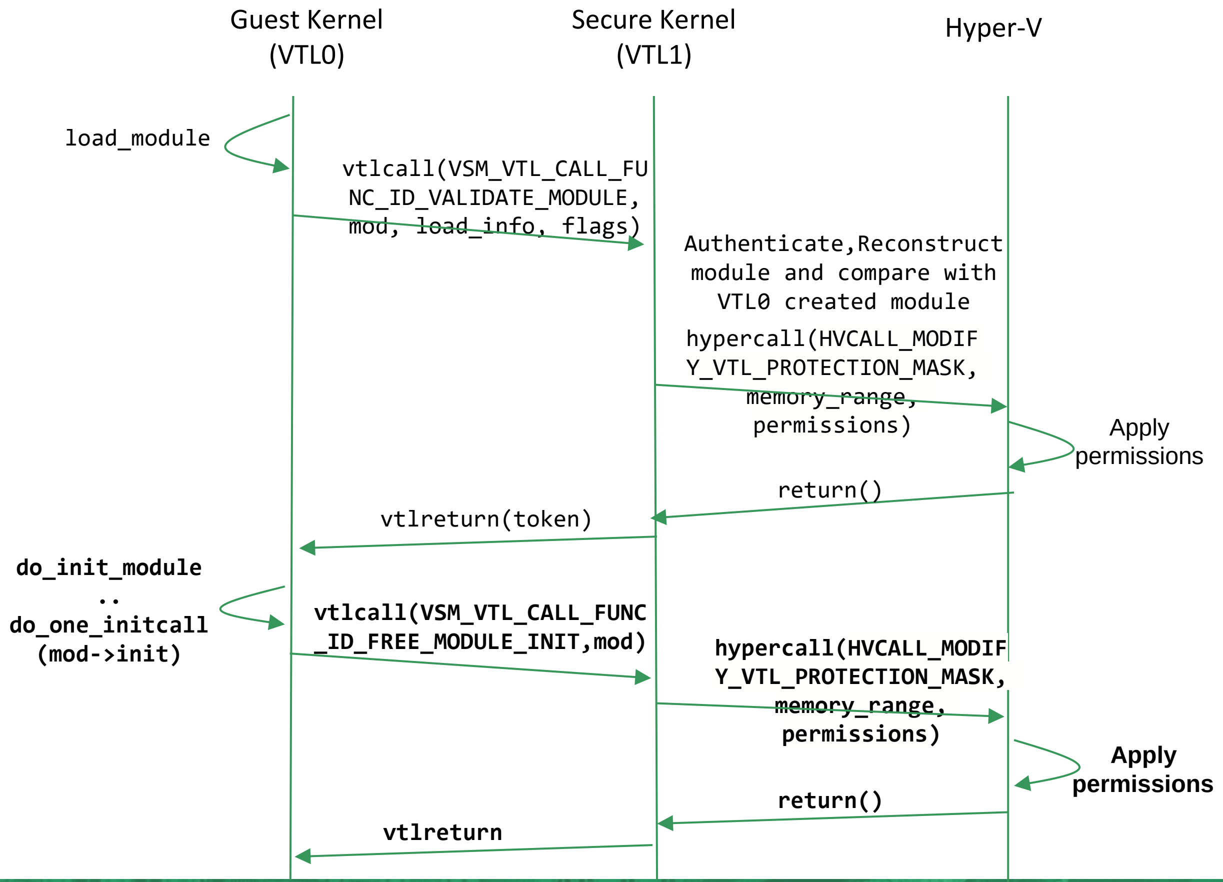
Module Loading(1/3)

- VTL1 independently authenticates module
- VTL1 reconstructs module
- VTL1 compares reconstructed module section with VTL0 module sections
- VTL1 sends request to Hyper-V to change EPT permissions for relevant module sections
- VTL1 returns per module secret token that can be used later



Module Loading(2/3)

- After module init
  - Reset permissions of all module init sections to r-w
  - Set permission of read-only after init memory to ro



Module Loading (3/3)

- Work in progress to support architecture dependent features:
  - retpolines
  - Control flow integrity features
  - Return thunks

Need additional kernel data

DEMO

Lessons Learned

- Text patching and dynamic code injection features are quite nuanced and add extra layer of complexity for basic kernel memory protection
- VTL0 and VTL1 kernel from the same source allows for lot of code reuse

What Next

- Kexec
- Ftrace->Livepatching
- Static call optimization,
- Jump label optimization

Code

- VTL0 : https://github.com/heki-linux/lvbs-linux/tree/ubuntu-6.5-lvbs

- VTL1 : https://github.com/heki-linux/lvbs-linux/tree/secure-kernel-6.6-lvbs

LINUX SECURITY SUMMIT EUROPE

THANK YOU

LVBS : Register Pinning (1/2)

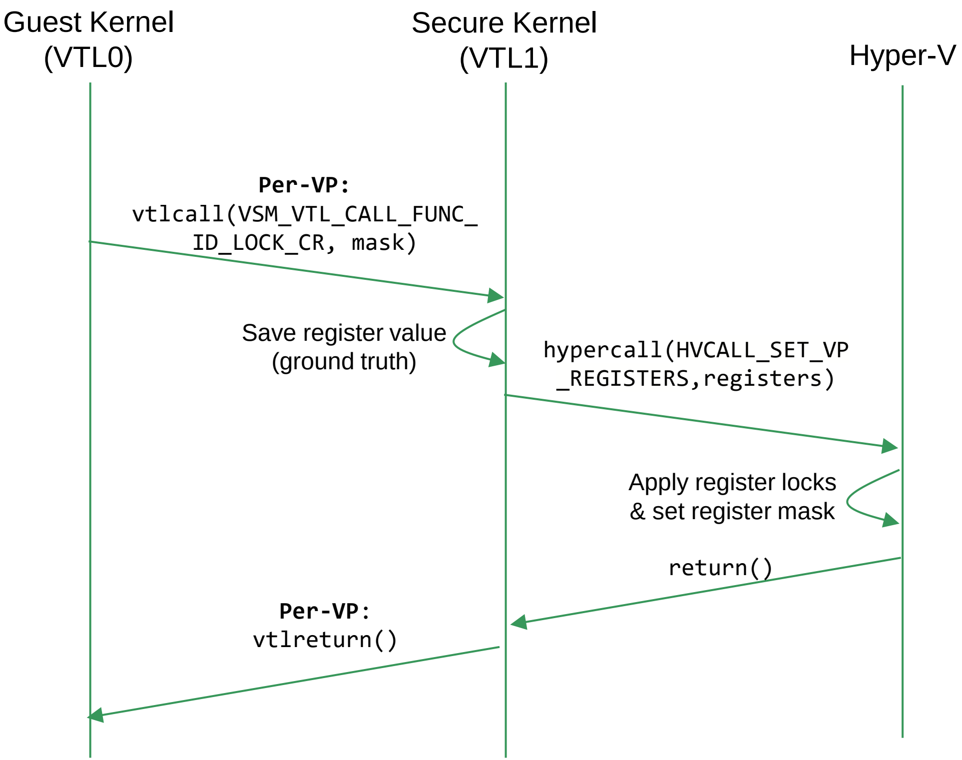
- Hyper-V supports intercepting access to a number of registers
    - 2 sets of registers
        - Block writes altogether
        - Allow writes, if the value is the same as pre-init

Action	Register
Block write & raise GP fault in VTL0	GDTR, IDTR, LDTR, TR
Allow write if new value is the same as pre-init. Otherwise, the write is blocked & raise GP fault in VTL0	LSTAR, STAR, CSTAR, APICBASE, EFER, SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP, SYSCALL_MASK

```c
/*  CR Intercept Control */
union hv_cr_intercept_control {
    u64 as_u64;
    struct {
        u64 cr0_write               : 1;
        u64 cr4_write               : 1;
        u64 xcr0_write              : 1;
        u64 ia32miscenable_read         : 1;
        u64 ia32miscenable_write        : 1;
        u64 msr_lstar_read          : 1;
        u64 msr_lstar_write         : 1;
        u64 msr_star_read           : 1;
        u64 msr_star_write          : 1;
        u64 msr_cstar_read          : 1;
        u64 msr_cstar_write         : 1;
        u64 msr_apic_base_read          : 1;
        u64 msr_apic_base_write         : 1;
        u64 msr_efer_read           : 1;
        u64 msr_efer_write          : 1;
        u64 gdtr_write              : 1;
        u64 idtr_write              : 1;
        u64 ldtr_write              : 1;
        u64 tr_write                : 1;
        u64 msr_sysenter_cs_write       : 1;
        u64 msr_sysenter_eip_write      : 1;
        u64 msr_sysenter_esp_write      : 1;
        u64 msr_sfmask_write            : 1;
        u64 msr_tsc_aux_write           : 1;
        u64 msr_sgx_launch_ctrl_write       : 1;
        u64 reserved                : 39;
    };
} __packed;
```

Register Pinning (2/2)

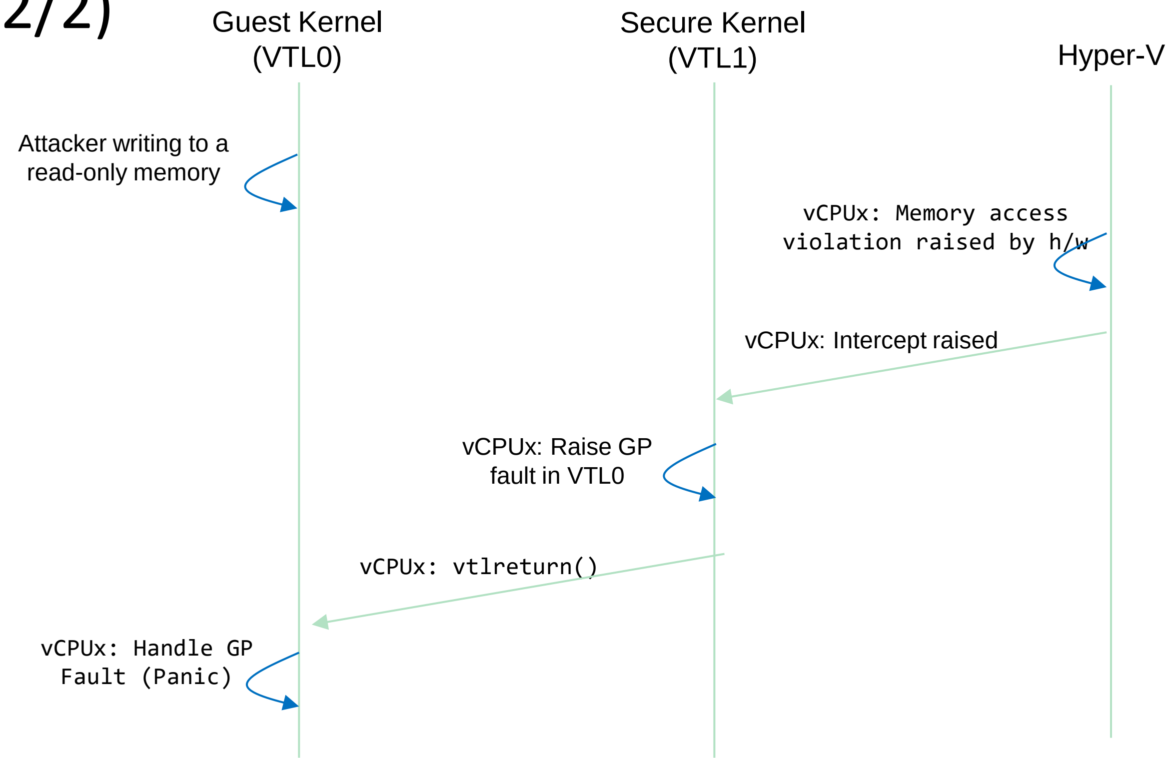
1. VTL0 sets the security policy of CRs to protect
2. For each VP, VTL0 does a LOCK_CR VTLCall
   - CRs are per VP
3. VTL1 sends a hypercall request to lock the CRs
4. Hypervisor applies protection
5. If there is a write request to a monitored register or mask, an intercept is injected to VTL1



LVBS : Exception Handling (1/2)

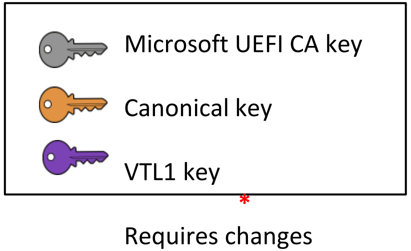
- Exceptions raised for
  - Violating memory access permissions
  - Violating register access
- VTL1 injects a GP fault in VTL0 and returns the control back to VTL0
- VTL0 thread that caused GP fault is killed
  - Depending on configuration, can cause Kernel panic

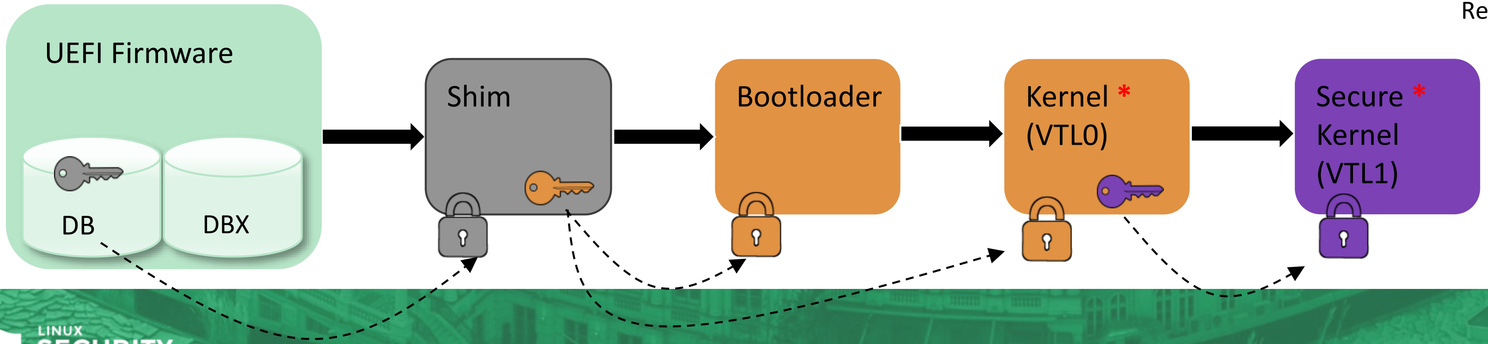
Exception Handling (2/2)



Secure Boot VTL1(1/2)

- If Secure Boot is enabled for Guest Kernel, the kernel extends Secure Boot to the VTL1 Secure Kernel and bootloader. This verifies the authenticity and integrity of VTL1 code.
  - Check if `efi_enabled(EFI_SECURE_BOOT)`
  - Use SHA256 with RSA encryption and PKCS #7 signature
  - X509 Certificate is added in System Trusted Keys
  - If/when accepted in mainline, the Shim canonical key can be used instead





Secure Boot VTL1(2/2)

- Generate signatures of secure kernel and bootloader binaries

- Add signature files to VTL0 initramfs
- Add VTL1 certificate to VTL0 System Trusted Keyring
  - CONFIG_SYSTEM_TRUSTED_KEYS
- VSM Boot driver reads files from initrd and verifies signatures

```c
static int verify_vsm_signature(char *buffer, unsigned int buff_size, char *signature,
                                unsigned int sig_size)
{
    int ret = 0;
    struct pkcs7_message *pkcs7;

    if (!buffer || !signature)
        return -EINVAL;
    pkcs7 = pkcs7_parse_message(signature, sig_size);
    if (IS_ERR(pkcs7)) {
        pr_err("%s: pkcs7_parse_message failed. Error code: %ld", __func__, PTR_ERR(pkcs7));
        return PTR_ERR(pkcs7);
    }
    ret = verify_pkcs7_signature(buffer, buff_size, signature, sig_size, NULL,
                                 VERIFYING_UNSPECIFIED_SIGNATURE, NULL, NULL);
    if (ret) {
        pr_err("%s: verify_pkcs7_signature failed. Error code: %d", __func__, ret);
        return ret;
    }
    return ret;
}
```