Back To The Future: A Radical Insecure Design of KVM on ARM

Abstract

In ARM, there are certain instructions that generate exceptions. Such instructions are typically executed to request a service from the software that runs at a higher privilege level. From OS kernel (EL1), the software can call into hypervisor (EL2) with the HVC instruction.

The KVM Hypervisor is a part of the Linux kernel, and it is enabled on all the supported ARM system by default. In this architecture, KVM is implemented as split-mode and runs across differently privileged CPU modes to execute code. This paper discusses the design, along with a vulnerability in the way Linux kernel initializes the KVM Hypervisor. An attacker having access to host EL1 can execute code in EL2. This vulnerability can be exploited by an attacker to install a hypervisor rootkit on ARM systems.

Introduction

In ARMv8-A, a program executes in one of the four Exception levels. Exception levels determine the level of execution privilege. Execution at EL*n* corresponds to privilege PL*n*. Larger value of *n* mean more privileges. ARMv8-A also provides hardware support for virtualization. The standalone hypervisor runs in EL2 with more privilege than OS kernel running in EL1. The KVM hypervisor is an extension of Linux kernel and is automatically available on devices that are running a recent version of the Linux kernel. Running entire Linux kernel in EL2 has its own problems, so KVM is implemented as split-mode and runs across differently privileged CPU modes. This way, it can take advantage of the functionality offered by each CPU mode. It is divided into two components, the Lowvisor that runs in EL2, and the Highvisor, which runs in EL1. The Lowvisor is designed to take advantage of the hardware virtualization support available in EL2. It provides some key functions like setting up the correct execution context by appropriate configuration of the hardware, and enforces protection and isolation between different execution contexts. The Highvisor runs in kernel mode as part of the host Linux kernel. If the system supports virtualization, Linux starts to boot in EL2 and then installs a hypervisor stub that allows Linux Kernel to further initialize and install KVM Hypervisor. There is a vulnerability in the way Linux kernel initializes the KVM Hypervisor. An attacker having access to host EL1 can exploit this it to execute code in EL2. This paper specifically discusses ARM64v8-A architecture.

Background

Arm privilege layer and exception vector table

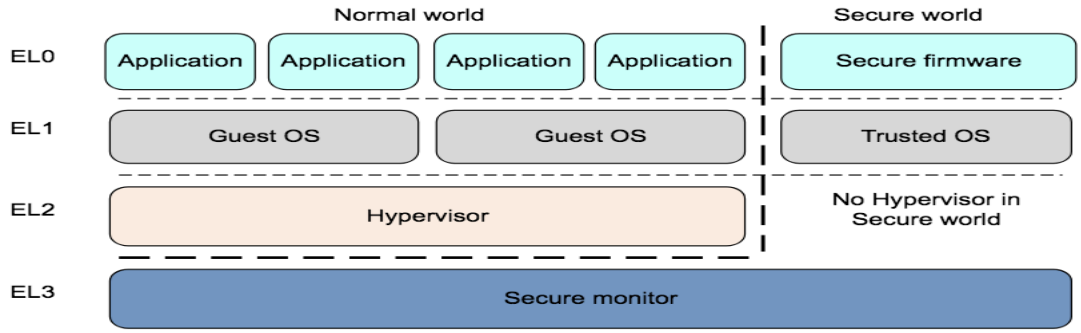
In ARMv8-A, a program executes in one of the four Exception levels. Exception levels determine the level of execution privilege. Following is a typical example of what software runs at each Exception level:

EL0 Normal user applications.

EL1 Operating system kernel typically described as *privileged*.

EL2. Hypervisor.

EL3 Low-level firmware, including the Secure Monitor



Each exception level has its own exception vector table, that is, there is one for each of EL3, EL2 and EL1. When an exception occurs, the processor must execute handler code that corresponds to the exception. The location in memory where a handler is stored is called the exception vector. In the ARM architecture, exception vectors are stored in a table, called the exception vector table. Vectors for individual exceptions are located at fixed offsets from beginning of the table. The virtual address of each table base is set by the Vector Based Address Registers, namely *VBAR_EL3*, *VBAR_EL2* and *VBAR_EL1*.

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

The base address is given by *VBAR_ELn* and then each entry has a defined offset from this base address. Each table has 16 entries, with each entry being 128 bytes (32 instructions) in size. The table effectively consists of 4 sets of 4 entries. Which entry is used depends upon a number of factors:

- The type of exception (SError, FIQ, IRQ or Synchronous)
- If the exception is being taken at the same Exception level, the Stack Pointer to be used (SP0 or SPx)
- If the exception is being taken at a lower Exception level, the execution state of the next lower level (AArch64 or AArch32)

Arm Virtual Extension

Most mainstream operating systems are built on the assumption that a system has a single privileged OS running several unprivileged applications. ARM virtualization, however, enables more than one OS to co-exist and operate on the same system. Implementing these virtual cores requires both, dedicated hardware extensions (to accelerate switching between virtual machines), and hypervisor software. A new mode, EL2, is introduced to support virtualization in the non-secure world. El2 is more privileged than user and kernel modes. Software running in EL2 can configure the hardware to trap from kernel mode into EL2 on various sensitive instructions and hardware interrupts. To run VMs, the hypervisor must at least partially reside in EL2. ARM designed the virtualization support around El2 as they envisioned a standalone hypervisor underneath a more complex rich OS.

Hypervisors can be broadly classified as Type1 and Type 2 hypervisor. Type 1 are the bare-metal hypervisor and each Virtual Machine (VM) contains a guest OS. Type 2 hypervisors are extensions of the host OS with each subsequent guest OS contained in a separate VM. Unlike Type 1 hypervisors, Type 2 do not consider hosts as VMs. There are two major Open Source

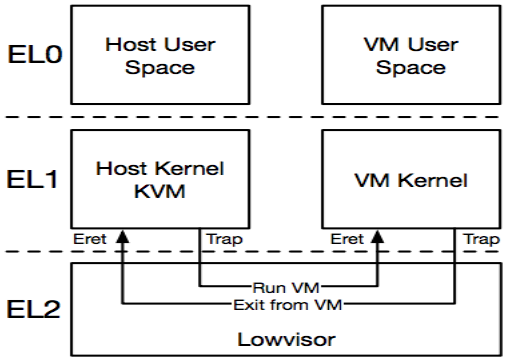
hypervisors, KVM, and Xen. The KVM hypervisor, discussed in this paper, is an extension of Linux and is considered as Type 2 hypervisor.

KVM Design

The KVM hypervisor is an extension of Linux. Although standalone bare metal hypervisor design approach has the potential for better performance and a smaller Trusted Computing Base (TCB), due to diversity in ARM hardware, as compared to x86, this approach is less practical on ARM. Linux, however, is supported across almost all ARM platforms and by integrating KVM with Linux, KVM is automatically available on all devices running a recent version of the Linux kernel. Integration of KVM/ARM in Linux solved the portability and hardware support issues. However, ARM hardware virtualization extensions were designed to support a standalone hypervisor, which is completely separate from any standard kernel. Simply running a hypervisor entirely in EL2 mode is attractive since it is the most privileged level. But, since KVM leverages existing kernel infrastructure such as the scheduler, running KVM in EL2 implies running the entire Linux kernel in EL2. This is considered problematic by the KVM team for various reasons.

As a solution for ARMv8.0, KVM introduced split-mode virtualization, a new approach to hypervisor design that splits the core hypervisor so that it runs across different privileged CPU modes and takes advantage of the specific benefits and functionality offered by each CPU mode. KVM uses split-mode virtualization to leverage the ARM hardware virtualization support enabled in EL2, while at the same time, leveraging existing Linux kernel services running in kernel mode. split-mode virtualization allows KVM to be integrated with the Linux kernel without intrusive modifications to the existing code base.

This is done by splitting the hypervisor into two components, the Lowvisor, and the Highvisor, as shown in picture below. The Lowvisor is designed to take advantage of the hardware virtualization support available in El2, It provides some key functions like setting up the correct execution context by appropriate configuration of the hardware, and enforces protection and isolation between different execution contexts. The Lowvisor performs only the minimal amount of processing required and defers the bulk of the work to the Highvisor, after a world switch to the Highvisor is complete. The Highvisor runs in kernel mode as part of the host Linux kernel. It can therefore directly leverage existing Linux functionality such as the scheduler, and can also make use of standard kernel software data structures and mechanisms to implement its functionality (such as locking mechanisms and memory allocation functions). This makes higher-level functionality easier to implement in the Highvisor.



Note: In ARMv8.1 extension with Virtualization Host Extension, it is possible to run the whole kernel in EL2.

Linux Boot and KVM

In ARM architecture EL2 is more privileged than the kernel modes (EL1) and there is no architecturally defined ABI for entering to EL2 from less privileged modes. So, in order to support KVM on Linux, Linux starts to boot in El2. Once the kernel is booted in EL2, it installs a stub handler that allows other subsystems like KVM to take control of EL2 mode. After installing the stub, the Linux kernel switches back to EL1 for further boot process.

For example, ARM Trusted Firmware which provides a reference implementation of secure world software for ARMv8-A passes controls in EL2 to the normal world software. Following is the code snippet from ARM Trusted Firmware for Resberry Pi 3.
The SPSR register is set to transfer control in EL2.

```
/*******************************************************************
**
* Gets SPSR for BL33 entry
*******************************************************************
/
uint32_t rpi3_get_spsr_for_bl33_entry(void)
{
#if RPI3_BL33_IN_AARCH32
INFO("BL33 will boot in Non-secure AArch32 Hypervisor mode\n");
return SPSR_MODE32(MODE32_hyp, SPSR_T_ARM, SPSR_E_LITTLE,
DISABLE_ALL_EXCEPTIONS);
#else
return SPSR_64(MODE_EL2, MODE_SP_ELX, DISABLE_ALL_EXCEPTIONS);
#endif
}
```

For ARM architecture, as soon as Linux boots, it checks the current CPU mode. In case the current mode is EL2, Linux configures EL2 hardware and then installs a hypervisors stub that allows other subsystems like KVM to take control of EL2 mode.
Following is the code snippet from the file kernel/arch/arm64/kernel/head.s.

```
ENTRY(el2_setup)
        msr     SPsel, #1                       // We want to use SP_EL{1,2}
        mrs     x0, CurrentEL
        cmp     x0, #CurrentEL_EL2
        b.eq    1f
        mov_q   x0, (SCTLR_EL1_RES1 | ENDIAN_SET_EL1)
        msr     sctlr_el1, x0
        mov     w0, #BOOT_CPU_MODE_EL1          // This cpu booted in EL1
        isb
        ret

1:      mov_q   x0, (SCTLR_EL2_RES1 | ENDIAN_SET_EL2)
        msr     sctlr_el2, x0
```

If the CPU is in El2, the control is transferred to install_el2_stub which installs the Hypervisor stub. Following is the code snippet from *kernel/arch/arm64/kernel/head.s* that installs the stub.

```
        /* Hypervisor stub */
 7:     adr_l   x0, __hyp_stub_vectors
        msr     vbar_el2, x0

        /* spsr */
        mov     x0, #(PSR_F_BIT | PSR_I_BIT | PSR_A_BIT | PSR_D_BIT |\
                      PSR_MODE_EL1h)
        msr     spsr_el2, x0
        msr     elr_el2, lr
        mov     w0, #BOOT_CPU_MODE_EL2          // This CPU booted in EL2
        eret
```

The above code snippet updates the register *VBAR_EL2* so that it points to *__hyp_stub_vectors* vector table. The hyp stub *__hyp_stub_vectors* is defined in the file *hyp_stub.s* as follow

```
ENTRY(__hyp_stub_vectors)
        ventry  el2_sync_invalid            // Synchronous EL2t
        ventry  el2_irq_invalid             // IRQ EL2t
        ventry  el2_fiq_invalid             // FIQ EL2t
        ventry  el2_error_invalid           // Error EL2t

        ventry  el2_sync_invalid            // Synchronous EL2h
        ventry  el2_irq_invalid             // IRQ EL2h
        ventry  el2_fiq_invalid             // FIQ EL2h
        ventry  el2_error_invalid           // Error EL2h

        ventry  el1_sync                    // Synchronous 64-bit EL1
        ventry  el1_irq_invalid             // IRQ 64-bit EL1
        ventry  el1_fiq_invalid             // FIQ 64-bit EL1
        ventry  el1_error_invalid           // Error 64-bit EL1

        ventry  el1_sync_invalid            // Synchronous 32-bit EL1
        ventry  el1_irq_invalid             // IRQ 32-bit EL1
        ventry  el1_fiq_invalid             // FIQ 32-bit EL1
        ventry  el1_error_invalid           // Error 32-bit EL1
ENDPROC(__hyp_stub_vectors)
```

Once *VBAR_EL2* is updated with the address of vector table *__hyp_stub_vectors*, the vector table *__hyp_stub_vectors* is installed as an exception vector table for EL2. The function *el1_sync,* defined at offset *0x400* of vector table, is registered as the handler for Synchronous exception from 64-bit EL1 kernel. Function *el1_sync* will be invoked as an exception handler if HVC instruction is executed by 64bit kernel.

Following is the code snippet for *el1_sync* function, defined in the file *hyp_stub.S*

```
el1_sync:
        cmp     x0, #HVC_SET_VECTORS
        b.ne    2f
        msr     vbar_el2, x1
        b       9f

2:      cmp     x0, #HVC_SOFT_RESTART
        b.ne    3f
        mov     x0, x2
        mov     x2, x4
        mov     x4, x1
        mov     x1, x3
        br      x4                              // no return

3:      cmp     x0, #HVC_RESET_VECTORS
        beq     9f                              // Nothing to reset!

        /* Someone called kvm_call_hyp() against the hyp-stub... */
        ldr     x0, =HVC_STUB_ERR
        eret

9:      mov     x0, xzr
        eret
ENDPROC(el1_sync)
```

el1_sync expects *HVC_SET_VECTORS*, *HVC_SOFT_RESTART* and *HVC_RESET_VECTORS* command from EL1. The register value of *X0* determines which command needs to be processed. In case the register *x0* is *HVC_SET_VECTORS,* the *VBAR_EL2* register is reset to the new value passed with register *x1*.

The Linux kernel exposes a function *__hyp_set_vectors*, defined in file "*/kernel/hyp-stub.S*", to install *EL2_VBAR* table. The Linux kernel code running in EL1, which initializes KVM, uses this interface to install KVM Hypervisor. The function *__hyp_set_vectors* is defined as below.

```
ENTRY(__hyp_set_vectors)
        mov     x1, x0
        mov     x0, #HVC_SET_VECTORS
        hvc     #0
        ret
ENDPROC(__hyp_set_vectors)
```

)

Once the EL2 hardware is configured and the stub vector table is installed, Linux kernel switches back to EL1 to perform normal Linux booting.

KVM Initialization

The Linux kernel begins initialization of KVM by invoking "*kvm_init*" function defined in the file *linux\kvm\kvm_main.c*. This function is called by "*arm_init*" function, which is defined in *arm.c*.

The "*kvm_init*" function first checks if the CPU is booted in the EL2 mode. In case the CPU is not booted in El2 mode, it returns the following error and no further KVM initialization is done.

```
if (!is_hyp_mode_available()) {
        kvm_err("HYP mode not available\n");
        return -ENODEV;
}
```

If CPU is booted in EL2, the function "*cpu_init_hyp_mode*" is invoked. This function initializes and installs *__KVM_Initilization_vector* as a new vector table for EL2.The *__hyp_set_vectors* interface is used to install __KVM_Initilization_vector.

__KVM_Initialization_vector is defined as follow in the file *hyp-init.s*

```
ENTRY(__kvm_hyp_init)
        ventry  __invalid               // Synchronous EL2t
        ventry  __invalid               // IRQ EL2t
        ventry  __invalid               // FIQ EL2t
        ventry  __invalid               // Error EL2t

        ventry  __invalid               // Synchronous EL2h
        ventry  __invalid               // IRQ EL2h
        ventry  __invalid               // FIQ EL2h
        ventry  __invalid               // Error EL2h

        ventry  __do_hyp_init           // Synchronous 64-bit EL1
        ventry  __invalid               // IRQ 64-bit EL1
        ventry  __invalid               // FIQ 64-bit EL1
        ventry  __invalid               // Error 64-bit EL1

        ventry  __invalid               // Synchronous 32-bit EL1
        ventry  __invalid               // IRQ 32-bit EL1
        ventry  __invalid               // FIQ 32-bit EL1
        ventry  __invalid               // Error 32-bit EL1
```

After *__KVM_Initialization_vector* is installed as an exception vector table for EL2, the function *__do_hyp_init*, defined at offset *0x400*, is registered as handler for synchronous exception from 64-bit EL1 kernel.

Further, after installing *__KVM_Initialization_vector*, the *cpu_init_hyp_mode* function obtains the pointer for hypervisors page table, hypervisor stack, and the actual hypervisor vector table. Post this, the function *__cpu_init_hyp_mode(pgd_ptr, hyp_stack_ptr, vector_ptr)* is invoked. This function makes HVC call to invoke the exception handler function *__do_hyp_init* and passes the page table, stack, and actual hypervisor vector as parameters.

Following is the implementation of *cpu_init_hyp_mode* function

```
static void cpu_init_hyp_mode(void *dummy)
{
        phys_addr_t pgd_ptr;
        unsigned long hyp_stack_ptr;
        unsigned long stack_page;
        unsigned long vector_ptr;

        /* Switch from the HYP stub to our own HYP init vector */
        __hyp_set_vectors(kvm_get_idmap_vector());

        pgd_ptr = kvm_mmu_get_httbr();
        stack_page = __this_cpu_read(kvm_arm_hyp_stack_page);
        hyp_stack_ptr = stack_page + PAGE_SIZE;
        vector_ptr = (unsigned long)kvm_get_hyp_vector();

        __cpu_init_hyp_mode(pgd_ptr, hyp_stack_ptr, vector_ptr);
        __cpu_init_stage2();

        kvm_arm_init_debug();
}
```

__do_hyp_init function is defined in the file *Arch/arm64/kvm/hyp-init.s.* The implementation is as follows :

```
        /*
         * x0: HYP pgd
         * x1: HYP stack
         * x2: HYP vectors
         */
__do_hyp_init:
        /* Check for a stub HVC call */
        cmp     x0, #HVC_STUB_HCALL_NR
        b.lo    __kvm_handle_stub_hvc

        phys_to_ttbr x4, x0
        msr     ttbr0_el2, x4

        mrs     x4, tcr_el1
        ldr     x5, =TCR_EL2_MASK
        and     x4, x4, x5
        mov     x5, #TCR_EL2_RES1
        orr     x4, x4, x5
```

The function *__do_hyp_init* enables MMU for hypervisor and sets EL2 stack pointer. It then installs actual KVM Hypervisors vector table *__kvm_hyp_vector*. This vector table is defined in the file "*kvm/hyp/hyp-entry.s*" as follow

```
ENTRY(__kvm_hyp_vector)
        invalid_vect    el2t_sync_invalid       // Synchronous EL2t
        invalid_vect    el2t_irq_invalid        // IRQ EL2t
        invalid_vect    el2t_fiq_invalid        // FIQ EL2t
        invalid_vect    el2t_error_invalid      // Error EL2t

        invalid_vect    el2h_sync_invalid       // Synchronous EL2h
        invalid_vect    el2h_irq_invalid        // IRQ EL2h
        invalid_vect    el2h_fiq_invalid        // FIQ EL2h
        valid_vect      el2_error               // Error EL2h

        valid_vect      el1_sync                // Synchronous 64-bit EL1
        valid_vect      el1_irq                 // IRQ 64-bit EL1
        invalid_vect    el1_fiq_invalid         // FIQ 64-bit EL1
        valid_vect      el1_error               // Error 64-bit EL1

        valid_vect      el1_sync                // Synchronous 32-bit EL1
        valid_vect      el1_irq                 // IRQ 32-bit EL1
        invalid_vect    el1_fiq_invalid         // FIQ 32-bit EL1
        valid_vect      el1_error               // Error 32-bit EL1
ENDPROC(__kvm_hyp_vector)
```

Once the actual Hypervisor vector table is installed as an exception vector table for EL2, the function *el1_sync*, defined at offset *0x400* in *__kvm_hyp_vector*, is registered as a handler for synchronous exception originating from 64-bit EL1 kernel.
Following is the code snippet for *el1_sync*, defined in the file *hyp-entry.s*. It invokes the function "*__kvm_handle_stub_hvc*" if the value of *x0* register is less than *HVC_STUB_HCALL_NR*.

```
el1_sync:                               // Guest trapped into EL2

        mrs     x0, esr_el2
        lsr     x0, x0, #ESR_ELx_EC_SHIFT
        cmp     x0, #ESR_ELx_EC_HVC64
        ccmp    x0, #ESR_ELx_EC_HVC32, #4, ne
        b.ne    el1_trap

        mrs     x1, vttbr_el2           // If vttbr is valid, the guest
        cbnz    x1, el1_hvc_guest       // called HVC

        /* Here, we're pretty sure the host called HVC. */
        ldp     x0, x1, [sp], #16

        /* Check for a stub HVC call */
        cmp     x0, #HVC_STUB_HCALL_NR
        b.hs    1f

        /*
         * Compute the idmap address of __kvm_handle_stub_hvc and
         * jump there. Since we use kimage_voffset, do not use the
         * HYP VA for __kvm_handle_stub_hvc, but the kernel VA instead
         * (by loading it from the constant pool).
         *
         * Preserve x0-x4, which may contain stub parameters.
         */
        ldr     x5, =__kvm_handle_stub_hvc
        ldr_l   x6, kimage_voffset

        /* x5 = __pa(x5) */
        sub     x5, x5, x6
        br      x5

1:
        /*
         * Perform the EL2 call
         */
        kern_hyp_va     x0
        do_el2_call

        eret
```

Following is the code snippet for function "*__kvm_handle_stub_hvc*" defined in *hyp-init.s*. If the value in *x0* register is equal to *#HVC_RESET_VECTORS,* the function resets *EL2_VBAR* back to *__hyp_stub_vectors*. This implies that once the actual hypervisor vector table is installed, it provides an interface to reset *VBAR_EL2* back to the initial hypervisor stub. In addition, as discussed earlier, the hypervisor stub provides an interface to update *VBAR_EL2* from EL1. So, combining these two commands provides an opportunity for an attacker to execute code in EL2 from EL1. An attacker can exploit this vulnerability and install a hypervisor rootkit that runs with more privileges than that of host kernel, or any security software running in the host kernel.

```
ENTRY(__kvm_handle_stub_hvc)
        cmp     x0, #HVC_SOFT_RESTART
        b.ne    1f

        /* This is where we're about to jump, staying at EL2 */
        msr     elr_el2, x1
        mov     x0, #(PSR_F_BIT | PSR_I_BIT | PSR_A_BIT | PSR_D_BIT | PSR_MODE_EL2h)
        msr     spsr_el2, x0

        /* Shuffle the arguments, and don't come back */
        mov     x0, x2
        mov     x1, x3
        mov     x2, x4
        b       reset

1:      cmp     x0, #HVC_RESET_VECTORS
        b.ne    1f
reset:
        /*
         * Reset kvm back to the hyp stub. Do not clobber x0-x4 in
         * case we coming via HVC_SOFT_RESTART.
         */
        mrs     x5, sctlr_el2
        ldr     x6, =SCTLR_ELx_FLAGS
        bic     x5, x5, x6              // Clear SCTL_M and etc
        pre_disable_mmu_workaround
        msr     sctlr_el2, x5
        isb

        /* Install stub vectors */
        adr_l   x5, __hyp_stub_vectors
        msr     vbar_el2, x5
        mov     x0, xzr
        eret
```

Exploit

Assuming that an attacker has the execution privilege on host kernel EL1. In order to execute code in EL2, the attacker needs to do the following:

1. Create *HVC_RESET_VECTORS* exception request. This HVC call will invoke *__kvm_handle_stub_hvc* in EL2, which in turn, will disable Hypervisor MMU and reset the *VBAR_EL2* to *__hyp_stub_vectors*.
2. The attacker then allocates a physical continuous memory in the kernel. Here, attacker needs to allocate physical continuous memory because MMU of EL2 is disabled.
3. Attacker embeds the shellcode to be executed in El2 at an offset *0x400* in allocated memory block. The offset *0x400* is set because exception handler for HVC originating from 64bit kernel is at this offset.
4. The attacker then creates *HVC_SET_VECTORS* HVC call and passes that physical address of the memory buffer allocated in step 2.
5. *HVC_SET_VECTORS* request will reset the attacker allocated buffer as new exception vector table for EL2.
6. Finally, the attacker can invoke HVC call, which will execute the attacker's shellcode in EL2.

Note: Given the current design flaw in KVM, this is just one of many ways that might be used to execute code in EL2 form EL1.

Conclusion

This security issue was reported to Red Hat Security, who then escalated this issue to KVM team. It can be concluded from the KVM team's response that their threat model does not consider this as a security issue, and they don't care about this vulnerability that KVM adds into the privilege separation boundary. Their assumption may work in some case, but certainly will not work in all case. The attacker can use this design as a booster once they manage to get into the host kernel. They can gain more privileges and migrate to EL2. This provides attackers the following advantages:

- Attackers can run their code unreferenced by any code running in Linux EL1.
- Can configure EL2 to get code execution from various different places.
- Attackers' code will run with higher privileges than the security software running in host kernel, and thus, will have an upper hand.
- Attackers can use it as a generic way to bypass security mechanisms implemented (like Linux Kernel Runtime Guard) in the kernel by escaping to EL2.
- Attackers can use this to target security monitoring software running in EL2.
- This design flaw gives attackers opportunity for Blue Pill for KVM on ARM.

Mitigation

For robust and secure design, the hypervisor initialization should be done first, and once the hypervisors initialization is complete, the controls should be switched back to EL1 to start kernel initialization. This requires a comprehensive design change in KVM. As an interim security improvement, make sure that Linux starts to boot in EL1 and this will disable KVM in your system.

References:

https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf
https://developer.arm.com/products/architecture/a-profile/docs/100942/latest/hypervisor-software
https://dl.acm.org/citation.cfm?id=2541946
http://www.cs.columbia.edu/~cdall/pubs/atc17-dall.pdf
http://www.cs.columbia.edu/~cdall/pubs/sosp2017-neve.pdf
https://lwn.net/Articles/557132/