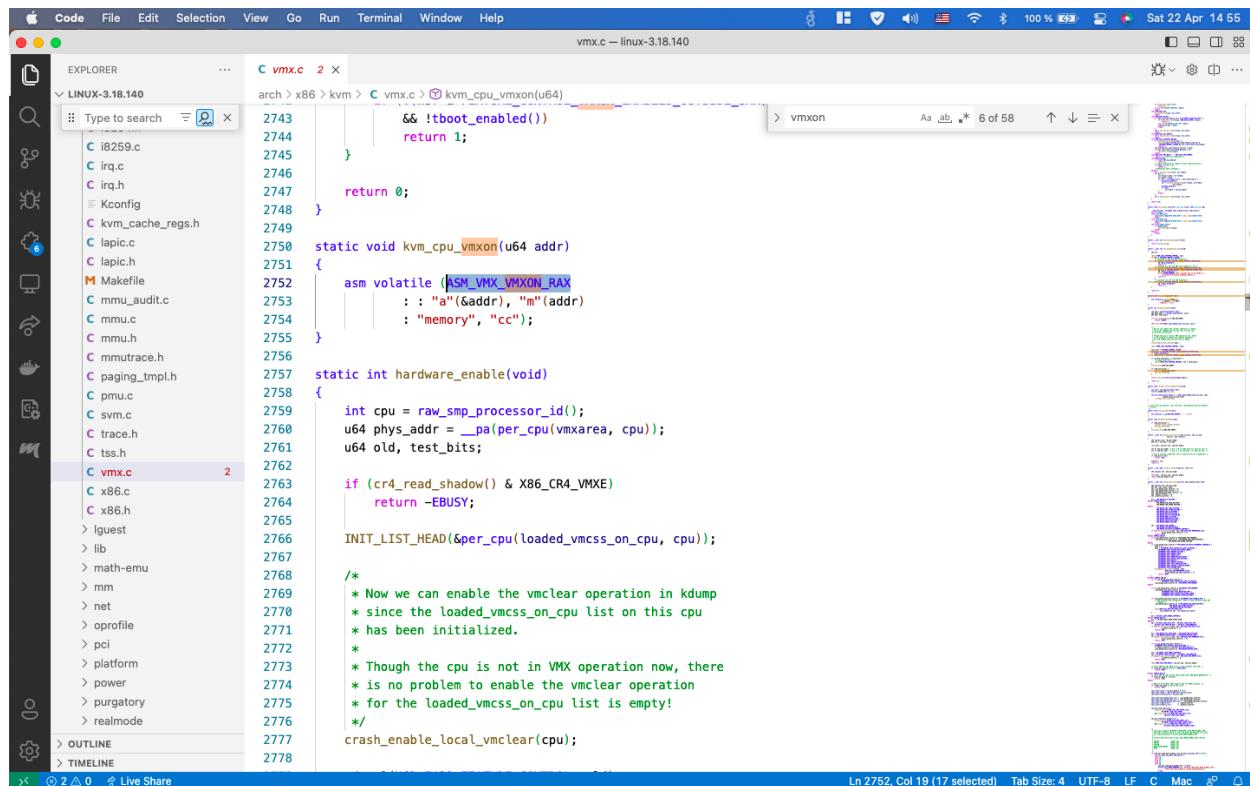


1. Main execution cycle
 - a. infinite loop
 - b. guest enter
 - c. return to VM
2. #PF(page fault) processing

Main execution cycle

Для начала, хочется поискать где вообще есть VMXON, VMXOFF, разные функции, в названии которых фигурирует VMX. Т.к. были даны послабления, например, поиска для конкретной архитектуры, то поиск был начат в папке с логичным названием arch/x86. Там же есть каталог kvm, что напрямую относится к KVM (по собственным предположениям). При дальнейшем рассмотрении в файле vmx.c можно найти следующие строки:



```
Code File Edit Selection View Go Run Terminal Window Help
C vmx.c — linux-3.18.140
EXPLORER
LINUX-3.18.140
Type to search
C vmx.c 2 x
arch > x86 > kvm > C vmx.c > kvm_cpu_vmxon(u64)
2743     && !boot_enabled())
2744     return 1;
2745 }
2746
2747     return 0;
2748 }
2749
2750 static void kvm_cpu_vmxon(u64 addr)
2751 {
2752     asm volatile (ASM_VMX_VMXON_RAX
2753                 : : "(addr)", "(addr"
2754                 : "memory", "cc");
2755 }
2756
2757 static int hardware_enable(void)
2758 {
2759     int cpu = raw_smp_processor_id();
2760     u64 phys_addr = __pa(per_cpu(vmxarea, cpu));
2761     u64 old, test_bits;
2762
2763     if (cr4_read_shadow() & X86_CR4_VME)
2764         return -EBUSY;
2765
2766     INIT_LIST_HEAD(&per_cpu(loaded_vmcss_on_cpu, cpu));
2767
2768     /*
2769     * Now we can enable the vmclear operation in kdump
2770     * since the loaded_vmcss_on_cpu list on this cpu
2771     * has been initialized.
2772     *
2773     * Though the cpu is not in VMX operation now, there
2774     * is no problem to enable the vmclear operation
2775     * for the loaded_vmcss_on_cpu list is empty!
2776     */
2777     crash_enable_local_vmclear(cpu);
2778 }
```

Picture: linux-3.18.140/arch/x86/kvm/vmx.c

При переходе в заголовочный файл vmx.h, можно наблюдать объявление этого имени. Ему соответствует ассемблерный код (инструкция), который выглядит так:

```

422 #define VMX_EPT_WRITABLE_MASK          0x2ull
423 #define VMX_EPT_EXECUTABLE_MASK        0x4ull
424 #define VMX_EPT_IPAT_BIT              (1ull << 6)
425 #define VMX_EPT_ACCESS_BIT            (1ull << 8)
426 #define VMX_EPT_DIRTY_BIT             (1ull << 9)
427
428 #define VMX_EPT_IDENTITY_PAGETABLE_ADDR 0xffffbc000ul
429
430
431 #define ASM_VMX_VMCLEAR_RAX           ".byte 0x66, 0x0f, 0xc7, 0x30"
432 #define ASM_VMX_VMLAUNCH             ".byte 0x0f, 0x01, 0xc2"
433 #define ASM_VMX_VMRESUME             ".byte 0x0f, 0x01, 0xc3"
434 #define ASM_VMX_VMPTRLD_RAX          ".byte 0x0f, 0xc7, 0x30"
435 #define ASM_VMX_VMREAP_RDX_RAX       ".byte 0x0f, 0x78, 0xd0"
436 #define ASM_VMX_VMWRITE_RDX_RDX      ".byte 0x0f, 0x79, 0xd0"
437 #define ASM_VMX_VMWRITE_RSP_RDX      ".byte 0x0f, 0x79, 0xd4"
438 #define ASM_VMX_VMXOFF               ".byte 0x0f, 0x01, 0xc4"
439 #define ASM_VMX_VMXON_RAX             [.byte 0xf3, 0x0f, 0xc7, 0x30]
440 #define ASM_VMX_INVEPT              ".byte 0x66, 0x0f, 0x38, 0x80, 0x08"
441 #define ASM_VMX_INVPID              ".byte 0x66, 0x0f, 0x38, 0x81, 0x08"
442
443 struct vmx_msrs_entry {
444     u32 index;
445     u32 reserved;
446     u64 value;
447 } __aligned(16);
448
449 /*
450  * Exit Qualifications for entry failure during or after loading guest state
451  */
452 #define ENTRY_FAIL_DEFAULT          0
453 #define ENTRY_FAIL_PDPTE            2
454 #define ENTRY_FAIL_NMI              3
455 #define ENTRY_FAIL_VMCS_LINK_PTR    4
456
457 */

```

Picture: linux-3.18.140/arch/x86/include/asm/vmx.h

Он помещает адрес 0xc0f300f в регистр esi, а затем делает ю на адрес 0x37 (<https://defuse.ca/online-x86-assembler.htm#disassembly2>):

```

0:  be 0f 30 0f 0c          mov    esi,0xc0f300f
5:  70 30                  jo     0x37

```

*Remark. jo => jumps to the specified location if the previous instruction sets the overflow flag (OF).

Здесь же, чуть выше, можно наблюдать объявление имени ASM_VMX_VMLAUNCH, что может означать как раз точку входа. Разберём соответствующий ассемблерный код:

```

.text:
vmlaunch

```

Так как у меня не получилось дизассемблировать код в предыдущем редакторе, я воспользовался другим (<https://godbolt.org/>). Соответственно для

ASM_VMX_VMXOFF код будет такой:

.text:

vmxoff

Схема там получается примерно такая:

1. Программное обеспечение переходит в режим работы VMX, выполняя команду VMXON.
 2. VMM выполняет вход в виртуальную машину с помощью инструкций VMLAUNCH и VMRESUME;
 3. Виртуальная машина завершает передачу управления в точку входа, указанную VMM.
 4. В конце концов, VMM может решить завершить работу и прекратить работу VMX. Он делает это, выполняя инструкцию VMXOFF.

Ref: <https://docs.hyperdbg.org/tips-and-tricks/considerations/basic-concepts-in-intel-vt-x>

В итоге, надо поискать где вызываются эти инструкции, вызов может быть найден в одном файле, который мы видели ранее: в файле vmx.c. Здесь же видна ещё одна ассемблерная вставка (в функции `vmx_vcpu_run`):

```
arch > x86_64 kvm > c vmx.c > vmx_vcpu_run(kvm_vcpu *)
    .7657    "mov %c[r10]%, %r10 \n\t"
    .7658    "mov %c[r11]%, %r11 \n\t"
    .7659    "mov %c[r12]%, %r12 \n\t"
    .7660    "mov %c[r13]%, %r13 \n\t"
    .7661    "mov %c[r14]%, %r14 \n\t"
    .7662    "mov %c[r15]%, %r15 \n\t"
#endif
    .7663    "#endif\n"
    .7664    "#define ASM_VMX_VMLAUNCH\n"
    .7665    "#define ASM_VMX_VMRESUME\n"
    .7666    /* Enter guest mode */
    .7667    "jne 1f \n\t"
    .7668    __ex(ASM_VMX_VMLAUNCH) "\n\t"
    .7669    "jmp 2f \n\t"
    .7670    "1: " __ex(ASM_VMX_VMRESUME) "\n\t"
    .7671    "2: "
    .7672    /* Save guest registers, load host registers, keep flags */
    .7673    "mov %0, %c[wordsize](%_ASM_SP) \n\t"
    .7674    "pop %0 \n\t"
    .7675    "setbe %c[fail](%) \n\t"
    .7676    "mov %%_ASM_AX, %c[rax](%) \n\t"
    .7677    "mov %%_ASM_BX, %c[rbx](%) \n\t"
    .7678    __ASM_SIZE(pop) " %c[cx](%) \n\t"
    .7679    "mov %%_ASM_DX, %c[rdx](%) \n\t"
    .7680    "mov %%_ASM_SI, %c[rsi](%) \n\t"
    .7681    "mov %%_ASM_DI, %c[rdi](%) \n\t"
    .7682    "mov %%_ASM_BP, %c[rbp](%) \n\t"
    .7683    #ifdef CONFIG_X86_64
    .7684    "mov %r8, %c[r8](%) \n\t"
    .7685    "mov %r9, %c[r9](%) \n\t"
    .7686    "mov %r10, %c[r10](%) \n\t"
    .7687    "mov %r11, %c[r11](%) \n\t"
    .7688    "mov %r12, %c[r12](%) \n\t"
    .7689    "mov %r13, %c[r13](%) \n\t"
    .7690    "mov %r14, %c[r14](%) \n\t"
    .7691    "mov %r15, %c[r15](%) \n\t"
    .7692    "xor %r8d, %r8d \n\t"
Ln 7668, Col 30 (16 selected) Tab Size: 4 UTF-8 LF C Mac gO
```

```

arch > x86 > kvm > C vmx.c > vmx_vcpu_run(kvm_vcpu *)
7695     "xor %r11d, %r11d \n\t"
7696     "xor %r12d, %r12d \n\t"
7697     "xor %r13d, %r13d \n\t"
7698     "xor %r14d, %r14d \n\t"
7699     "xor %r15d, %r15d \n\t"
7700 #endif
7701     "mov %cr2, %%_ASM_AX " \n\t"
7702     "mov %%_ASM_AX ", %c[cr2](%0) \n\t"
7703
7704     "xor %%eax, %%eax \n\t"
7705     "xor %%ebx, %%ebx \n\t"
7706     "xor %%esi, %%esi \n\t"
7707     "xor %%edi, %%edi \n\t"
7708     "pop %%_ASM_BP " pop %%_ASM_DX " \n\t"
7709 ".pushsection .rodata \n\t"
7710 ".global vmx_return \n\t"
7711 "vmx_return: " _ASM_PTR " 2b \n\t"
7712 ".popsection"
7713 | : *"("vmx, "d"((unsigned long)HOST_RSP),
7714 [launched]"i"(offsetof(struct vcpu_vmx, __launched)),
7715 [fail]"i"(offsetof(struct vcpu_vmx, fail)),
7716 [host_rsp]"i"(offsetof(struct vcpu_vmx, host_rsp)),
7717 [rax]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RAX])),
7718 [rbx]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RXB])),
7719 [rcx]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RCX])),
7720 [rdx]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RDX])),
7721 [rsi]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RSI])),
7722 [rdi]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RDI])),
7723 [rbp]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_RBP])),
7724 #ifdef CONFIG_X86_64
7725     [r8]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R8])),
7726     [r9]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R9])),
7727     [r10]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R10])),
7728     [r11]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R11])),
7729     [r12]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R12])),
7730     [r13]"i"(offsetof(struct vcpu_vmx, vcpu.arch.regs[VCPU_REGS_R13]))

```

Picture: *linux-3.18.140/arch/x86/kvm/vmx.c*

Отсюда мы получаем информацию о структуре `vcpu_vmx`, в частности о структуре `kvm_vcpu`. А ещё находим место использования `vmx_vcpu_run` в объявлении структуры `kvm_x86_ops` `vmx_x86_ops`:

```
arch > x86 > kvm > C vmx.c > vmx_x86_ops
9190     .set_cr4 = vmx_set_cr4,
9191     .set_efer = vmx_set_efer,
9192     .get_idt = vmx_get_idt,
9193     .set_idt = vmx_set_idt,
9194     .get_gdt = vmx_get_gdt,
9195     .set_gdt = vmx_set_gdt,
9196     .get_dr6 = vmx_get_dr6,
9197     .set_dr6 = vmx_set_dr6,
9198     .set_dr7 = vmx_set_dr7,
9199     .sync_dirty_debug_regs = vmx_sync_dirty_debug_regs,
9200     .cache_reg = vmx_cache_reg,
9201     .get_rflags = vmx_get_rflags,
9202     .set_rflags = vmx_set_rflags,
9203     .fpu_activate = vmx_fpu_activate,
9204     .fpu_deactivate = vmx_fpu_deactivate,
9205
9206     .tlb_flush = vmx_flush_tlb,
9207
9208     .run = vmx_vcpu_run,
9209     .handle_exit = vmx_handle_exit,
9210     .skip_emulated_instruction = skip_emulated_instruction,
9211     .set_interrupt_shadow = vmx_set_interrupt_shadow,
9212     .get_interrupt_shadow = vmx_get_interrupt_shadow,
9213     .patch_hypcall = vmx_patch_hypcall,
9214     .set_irq = vmx_inject_irq,
9215     .set_nmi = vmx_inject_nmi,
9216     .queue_exception = vmx_queue_exception,
9217     .cancel_injection = vmx_cancel_injection,
9218     .interrupt_allowed = vmx_interrupt_allowed,
9219     .nmi_allowed = vmx_nmi_allowed,
9220     .get_nmi_mask = vmx_get_nmi_mask,
9221     .set_nmi_mask = vmx_set_nmi_mask,
9222     .enable_nmi_window = enable_nmi_window,
9223     .enable_irq_window = enable_irq_window,
9224     .update_cr8_intercept = update_cr8_intercept,
9225     .set_virtual_x2apic_mode = vmx_set_virtual_x2apic_mode,
9226     .set_apic_access_page_addr = vmx_set_apic_access_page_addr,
```

Ln 9208, Col 24 (12 selected) Tab Size: 4 UTF-8 LF C Mac ⌂

Picture: linux-3.18.140/arch/x86/kvm/vmx.c

Значит где-то будет вызываться vmx_x86_ops->run или kvm_x86_ops->run, поищем это место:

```
x86.c - linux-3.18.140
C vmx.c 2 C kvm_main.c 3 C x86.c 3 x
arch > x86 > kvm > C x86.c > vcpu_enter_guest(kvm_vcpu *)
6299     kvm_load_guest_xcr0(vcpu);
6300
6301     if (req_immediate_exit)
6302         smp_send_reschedule(vcpu->cpu);
6303
6304     kvm_guest_enter();
6305
6306     if (unlikely(vcpu->arch.switch_db_regs)) {
6307         set_debugreg(8, 7);
6308         set_debugreg(vcpu->arch.eff_db[0], 0);
6309         set_debugreg(vcpu->arch.eff_db[1], 1);
6310         set_debugreg(vcpu->arch.eff_db[2], 2);
6311         set_debugreg(vcpu->arch.eff_db[3], 3);
6312         set_debugreg(vcpu->arch.dr6, 6);
6313     }
6314
6315     trace_kvm_entry(vcpu->vcpu_id);
6316     kvm_x86_ops->run(vcpu);
6317
6318
6319 /*
6320 * Do this here before restoring debug registers on the host. And
6321 * since we do this before handling the vmexit, a DR access vmexit
6322 * can (a) read the correct value of the debug registers, (b) set
6323 * KVM_DEBUGREG_WONT_EXIT again.
6324 */
6325 if (unlikely(vcpu->arch.switch_db_regs & KVM_DEBUGREG_WONT_EXIT)) {
6326     int i;
6327
6328     WARN_ON(vcpu->guest_debug & KVM_GUESTDBG_USE_HW_BP);
6329     kvm_x86_ops->sync_dirty_debug_regs(vcpu);
6330     for (i = 0; i < KVM_NR_DB_REGS; i++)
6331         vcpu->arch.eff_db[i] = vcpu->arch.db[i];
6332
6333
6334 /*
6335 * If the guest has used debug registers, at least dr7
6336 */

Ln 6317, Col 21 (16 selected) Tab Size: 4 UTF-8 LF C Mac ⌂
```

Picture: linux-3.18.140/arch/x86/kvm/x86.c

Вызов происходит в функции `vcpu_enter_guest`. Вызов `vcpu_enter_guest` происходит в функции `__vcpu_run` того же файла.

```
x86.c — linux-3.18.140
6396
6397 static int __vcpu_run(struct kvm_vcpu *vcpu)
6398 {
6399     int r;
6400     struct kvm *kvm = vcpu->kvm;
6401
6402     vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
6403
6404     r = 1;
6405     while (r > 0) {
6406         if (vcpu->arch.mp_state == KVM_MP_STATE_RUNNABLE &&
6407             !vcpu->arch.apf.halted)
6408             r = vcpu_enter_guest(vcpu);
6409         else {
6410             srcu_read_unlock(&kvm->srcu, vcpu->srcu_idx);
6411             kvm_vcpu_block(vcpu);
6412             vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
6413             if (kvm_check_request(KVM_REQ_UNHALT, vcpu)) {
6414                 kvm_apic_accept_events(vcpu);
6415                 switch(vcpu->arch.mp_state) {
6416                     case KVM_MP_STATE_HALTED:
6417                         vcpu->arch.pv.pv_unhalted = false;
6418                         vcpu->arch.mp_state =
6419                             KVM_MP_STATE_RUNNABLE;
6420                     case KVM_MP_STATE_RUNNABLE:
6421                         vcpu->arch.apf.halted = false;
6422                         break;
6423                     case KVM_MP_STATE_INIT_RECEIVED:
6424                         break;
6425                     default:
6426                         r = -EINTR;
6427                         break;
6428                 }
6429             }
6430         }
6431     }
}
```

Ln 6408, Col 33 (16 selected) Tab Size: 4 UTF-8 LF C Mac ⌘ ⌘ ⌘

Picture: linux-3.18.140/arch/x86/kvm/x86.c

Вероятно, это и есть main loop. А работу по передаче управления и “входа” в гостя осуществляет vcpu_enter_guest. По поводу выхода постараемся воспользоваться тем, что было и начать, например, с VM_EXIT_REASON. По поиску exit_reason можно найти следующие строки:

Picture: linux-3.18.140/arch/x86/kvm/svm.c

Функция называется `handle_exit`, которая обрабатывает прерывание и ставит причину выхода, возвращает необходимый обработчик. Через неё можно попасть в `vmx_handle_exit` файла `vmx.c`, который мы видели ранее, а `handle_exit` и `kvm_guest_exit()` можно увидеть снова в `vcpu_enter_guest`:

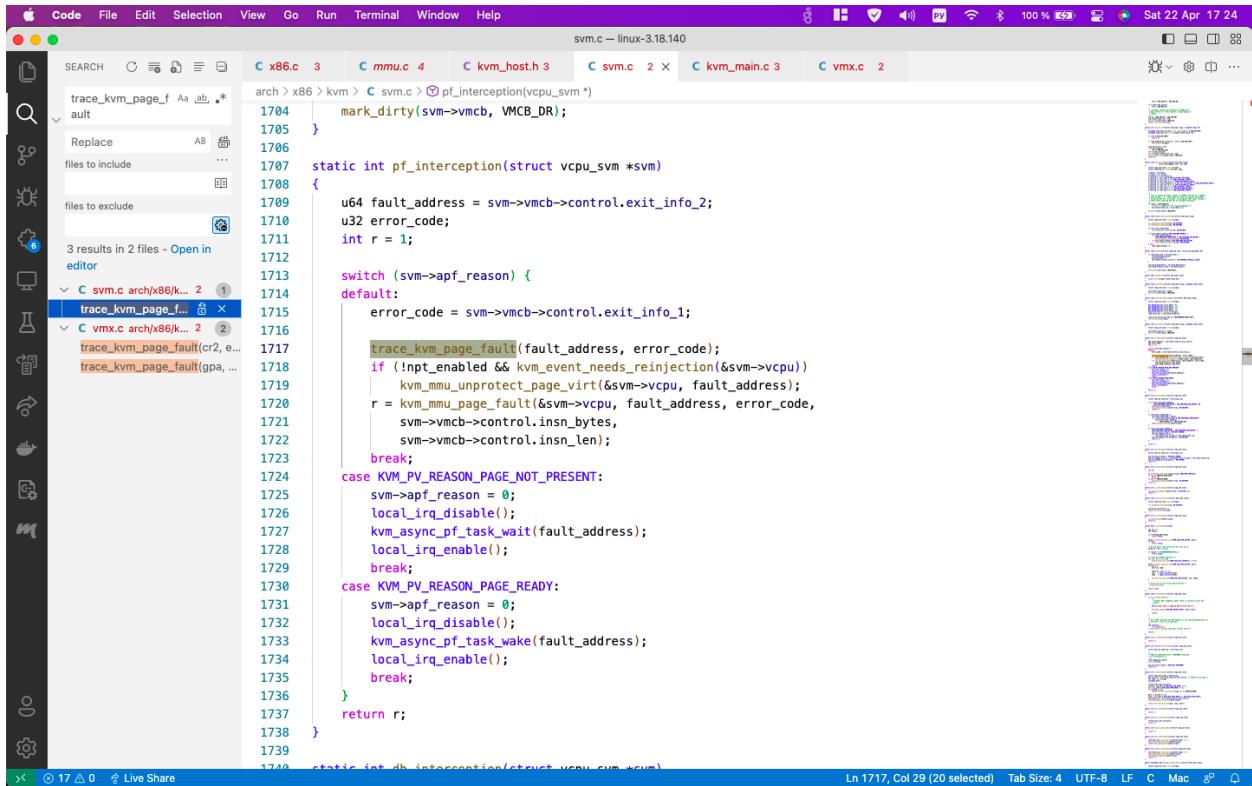
```
x86.c 3 x
arch > x86 > kvm > C x86.c > vcpu_enter_guest(kvm_vcpu *)
6352     /* Intercepted by handle_external_intr() */
6353     kvm_x86_ops->handle_external_intr(vcpu);
6354
6355     ++vcpu->stat.exits;
6356
6357 /*
6358 * We must have an instruction between local_irq_enable() and
6359 * kvm_guest_exit(), so the timer interrupt isn't delayed by
6360 * the interrupt shadow. The stat.exits increment will do nicely.
6361 * But we need to prevent reordering, hence this barrier():
6362 */
6363 barrier();
6364
6365 kvm_guest_exit();
6366
6367 preempt_enable();
6368
6369 vcpu->srcu_idx = srcu_read_lock(&vcpu->kvm->srcu);
6370
6371 /*
6372 * Profile KVM exit RIPs:
6373 */
6374 if (unlikely(prof_on == KVM_PROFILING)) {
6375     unsigned long rip = kvm_rip_read(vcpu);
6376     profile_hit(KVM_PROFILING, (void *)rip);
6377 }
6378
6379 if (unlikely(vcpu->arch.tsc_always_catchup))
6380     kvm_make_request(KVM_REQ_CLOCK_UPDATE, vcpu);
6381
6382 if (vcpu->arch.apic_attention)
6383     kvm_lapic_sync_from_vapic(vcpu);
6384
6385 r = kvm_x86_ops->handle_exit(vcpu);
6386
6387 return r;
6388
cancel_injection:
```

Picture: linux-3.18.140/arch/x86/kvm/x86.c

Picture: [linux-3.18.140/arch/x86/kvm/x86.c](#)

Предположу, что так как и запускали handle_exit будет вызываться от kvm_x86_ops, и видеть мы его можем снова в vcpu_enter_guest.

Page Fault Processing



The screenshot shows a code editor interface with multiple tabs open. The current tab is 'svm.c' from 'linux-3.18.140'. A search bar at the top has 'trace_kvm_page_f' entered. Below the search bar, there are sections for 'files to include' and 'files to exclude'. The main pane displays the code for the 'pf_interception' function. The search results are listed on the left side of the code pane. The code itself is as follows:

```
arch/x86/kvm/svm.c:1704 mark_dirty(svm->vmcb, VMCB_DR);
arch/x86/kvm/svm.c:1705 }
arch/x86/kvm/svm.c:1706 static int pf_interception(struct vcpu_svm *svm)
arch/x86/kvm/svm.c:1707 {
arch/x86/kvm/svm.c:1708     u64 fault_address = svm->vmcb->control.exit_info_2;
arch/x86/kvm/svm.c:1709     u32 error_code;
arch/x86/kvm/svm.c:1710     int r = 1;
arch/x86/kvm/svm.c:1711
arch/x86/kvm/svm.c:1712     switch (svm->apf_reason) {
arch/x86/kvm/svm.c:1713         default:
arch/x86/kvm/svm.c:1714             error_code = svm->vmcb->control.exit_info_1;
arch/x86/kvm/svm.c:1715             ktrace_kvm_page_fault(fault_address, error_code);
arch/x86/kvm/svm.c:1716             if (!npt_enabled && kvm_event_needs_reinjection(&svm->vcpu))
arch/x86/kvm/svm.c:1717                 kvm_mmu_unprotect_page_virt(&svm->vcpu, fault_address);
arch/x86/kvm/svm.c:1718             r = kvm_mmu_page_fault(&svm->vcpu, fault_address, error_code,
arch/x86/kvm/svm.c:1719                         svm->vmcb->control.insn_bytes,
arch/x86/kvm/svm.c:1720                         svm->vmcb->control.insn_len);
arch/x86/kvm/svm.c:1721             break;
arch/x86/kvm/svm.c:1722         case KVM_PV_REASON_PAGE_NOT_PRESENT:
arch/x86/kvm/svm.c:1723             svm->apf_reason = 0;
arch/x86/kvm/svm.c:1724             local_irq_disable();
arch/x86/kvm/svm.c:1725             kvm_async_pf_task_wait(fault_address);
arch/x86/kvm/svm.c:1726             local_irq_enable();
arch/x86/kvm/svm.c:1727             break;
arch/x86/kvm/svm.c:1728         case KVM_PV_REASON_PAGE_READY:
arch/x86/kvm/svm.c:1729             svm->apf_reason = 0;
arch/x86/kvm/svm.c:1730             local_irq_disable();
arch/x86/kvm/svm.c:1731             kvm_async_pf_task_wake(fault_address);
arch/x86/kvm/svm.c:1732             local_irq_enable();
arch/x86/kvm/svm.c:1733             break;
arch/x86/kvm/svm.c:1734     }
arch/x86/kvm/svm.c:1735     return r;
arch/x86/kvm/svm.c:1736 }
arch/x86/kvm/svm.c:1737 static int db_interception(struct vcpu_svm *svm)
```

Picture: linux-3.18.140/arch/x86/kvm/svm.c

Ещё есть kvm_mmu_page_fault в файле kvm/mmu.c. В vmx.c вызов этой функции находится в функции handle_exception:

Code Editor Screenshot (vmx.c - linux-3.18.140)

```

arch > x86 > kvm > C vmx.c > handle_exception(kvm_vcpu *)
4879
4880     static int handle_machine_check(struct kvm_vcpu *vcpu)
4881     {
4882         /* already handled by vcpu_run */
4883         return 1;
4884     }
4885
4886     static int handle_exception(struct kvm_vcpu *vcpu)
4887     {
4888         struct vcpu_vmx *vmx = to_vmx(vcpu);
4889         struct kvm_run *kvm_run = vcpu->run;
4890         u32 intr_info, ex_no, error_code;
4891         unsigned long cr2, rip, dr6;
4892         u32 vect_info;
4893         enum emulation_result er;
4894
4895         vect_info = vmx->idt_vectorizing_info;
4896         intr_info = vmx->exit_intr_info;
4897
4898         if (is_machine_check(intr_info))
4899             return handle_machine_check(vcpu);
4900
4901         if ((intr_info & INTR_INFO_INTR_TYPE_MASK) == INTR_TYPE_NMI_INTR)
4902             return 1; /* already handled by vmx_vcpu_run() */
4903
4904         if (is_no_device(intr_info))
4905             vmx_fpu_activate(vcpu);
4906         return 1;
4907     }
4908
4909     if (is_invalid_opcode(intr_info)) {
4910         er = emulate_instruction(vcpu, EMULTYPE_TRAP_UD);
4911         if (er == EMULATE_USER_EXIT)
4912             return 0;
4913         if (er != EMULATE_DONE)
4914             kvm_queue_exception(vcpu, UD_VECTOR);
4915         return 1;

```

Ln 4945, Col 34 (18 selected) Tab Size: 4 UTF-8 LF C Mac ⌘ ⌘ ⌘

Picture: linux-3.18.140/arch/x86/kvm/vmx.c

Code Editor Screenshot (mmu.c - linux-3.18.140)

```

arch > x86 > kvm > C mmu.c > kvm_mmu_page_fault(kvm_vcpu *, gva_t, u32, void *, int)
4158     {
4159         if (vcpu->arch.mmu.direct_map || mmu_is_nested(vcpu))
4160             return vcpu_match_mmio_gpa(vcpu, addr);
4161
4162         return vcpu_match_mmio_gva(vcpu, addr);
4163     }
4164
4165     int kvm_mmu_page_fault(struct kvm_vcpu *vcpu, gva_t cr2, u32 error_code,
4166                             void *insn, int insn_len)
4167
4168     int r, emulation_type = EMULTYPE_RETRY;
4169     enum emulation_result er;
4170
4171     r = vcpu->arch.mmu.page_fault(vcpu, cr2, error_code, false);
4172     if (r < 0)
4173         goto out;
4174
4175     if (!r) {
4176         r = 1;
4177         goto out;
4178     }
4179
4180     if (is_mmio_page_fault(vcpu, cr2))
4181         emulation_type = 0;
4182
4183     er = x86_emulate_instruction(vcpu, cr2, emulation_type, insn, insn_len);
4184
4185     switch (er) {
4186     case EMULATE_DONE:
4187         return 1;
4188     case EMULATE_USER_EXIT:
4189         ++vcpu->stat.mmio_exits;
4190         /* fall through */
4191     case EMULATE_FAIL:
4192         return 0;
4193     default:

```

Ln 4171, Col 28 Tab Size: 4 UTF-8 LF C Mac ⌘ ⌘ ⌘

Думаю, здесь и должна происходить обработка. А он ссылается на обработчик для этой платформы, который можно искать в trace, mm/fault.c с названием `__do_page_fault`.

The screenshot shows a code editor interface with the following details:

- File Path:** kvm.c — linux-3.18.140
- Search Results:** 171 results in 23 files - Open in editor
- Search Query:** page_fault
- Code Snippet (kvm.c):**

```
250     reason = __this_cpu_read(apf_reason.reason);
251     __this_cpu_write(apf_reason.reason, 0);
252 }
253
254     return reason;
255 }
256 EXPORT_SYMBOL_GPL(kvm_read_and_reset_pf_reason);
257 NOKPROBE_SYMBOL(kvm_read_and_reset_pf_reason);
258
259 dotraplinkage void
260 do_async_page_fault(struct pt_regs *regs, unsigned long error_code)
261 {
262     enum ctx_state prev_state;
263
264     switch (kvm_read_and_reset_pf_reason()) {
265     default:
266         trace_identry_page_fault(do_async_page_fault);
267         identry_async_page_fault(do_async_page_fault);
268         set_intr_gate(14, async_page_fault);
269     case KVM_PV_REASON_PAGE_NOT_PRESENT:
270         /* page is swapped out by the host. */
271         prev_state = exception_enter();
272         exit_idle();
273         kvm_async_pf_task_wait((u32)read_cr2());
274         exception_exit(prev_state);
275     case KVM_PV_REASON_PAGE_READY:
276         rcu_irq_enter();
277         exit_idle();
278         kvm_async_pf_task_wake((u32)read_cr2());
279         rcu_irq_exit();
280         break;
281     }
282 }
283 NOKPROBE_SYMBOL(do_async_page_fault);
284
285 static void __init paravirt_ops_setup(void)
286 {
```

- Right Panel:** Shows the call graph for the current function, with nodes for kvm_mm, kvm_main, and kvm.
- Status Bar:** Ln 261, Col 2 Tab Size: 4 UTF-8 LF C Mac ⌘

Picture: linux-3.18.140/arch/x86/kernel/kvm.c

```

traps.h - linux-3.18.140
C x86.c 3 C mmu.c 4 C traps.h 1 x C pgtable.h C fault.c 4 C kvm.c 3 C kvm_host.h 3 C svm.c 2 C kvm_main.c 3 ... 

SEARCH 🔎 ⌂ ⌂ ⌂ ⌂ 
do_async_page_fa Aa ab.* ult
Replace AB ...
files to include ...
arch/x86/ ...
files to exclude ...
4 results in 3 files - Open in editor
entry_32.S arch/x86/k... 1 pushl_cfi $do_async_page_f...
entry_64.S arch/x86/k... 1 identry async_page_fault d...
C kvmc/arch... 3 do_async_page_fa NOKPROBE_SYMBOL(do_as...
fault.c ~/Developer/s23/tlv/linux-3.18.140/arch/x86/mm - Definitions (3)
1275 } up_t, cod/mm/mmap_sem,
1276 NOKPROBE_SYMBOL(_do_page_fault);
1277
1278 dotraplinkage void notrace
1279 _do_page_fault(struct pt_regs *regs, unsigned long error_code)
1280 {
1281     unsigned long address = read_cr2(); /* Get the faulting address */
1282     enum ctx_state prev_state;
1283
1284 }
1285 #endif

```

Ln 81, Col 15 Tab Size: 4 UTF-8 LF C Mac ⚙

Picture: linux-3.18.140/arch/x86/include/asm/traps.h

```

fault.c - linux-3.18.140
C x86.c 3 C mmu.c 4 C traps.h C pgtable.h C fault.c x C kvm.c 3 C kvm_host.h 3 C svm.c 2 C kvm ... 

SEARCH 🔎 ⌂ ⌂ ⌂ ⌂ 
page_fault Aa ab.* Replace AB ...
files to include ...
arch/x86/ ...
files to exclude ...
171 results in 23 files - Open in editor
C kdebug.h arch/x86/include... 1 DIE_PAGE_FAULT;
C kvm_emulate.h arch/... 1 bool nested_page_fault;
C kvm_host.h arch/x86/... 4 int (*page_fault)(struct kvm_vcpu*, void (*inject_page_fault)..., void kvm_inject_page_fault(..., int kvm_mmuh_page_fault(str...
C traps.h arch/x86/include... 9 asmlinkage void page_fault(..., asmlinkage void async_page_fault(..., asmlinkage void trace_page_fault(..., #define trace_async_page_f...
trace_async_page_fault asy... dotraplinkage void... do_page_fault(void, ..., dotraplinkage void trace_d... static inline void trace_d... do_page_fault(regs, error); C vmm.h arch/x86/include... 2 PAGE_FAULT_ERROR_CODE... PAGE_FAULT_ERROR_CODE... C exceptions.h arch/x8... 8 #ifndef _TRACE_PAGE_FAULT #define _TRACE_PAGE_FAU... #define DEFINE_PAGE_FAU...
NOKPROBE_SYMBOL(_do_page_fault);

dotraplinkage void notrace
do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    unsigned long address = read_cr2(); /* Get the faulting address */
    enum ctx_state prev_state;

/*
 * We must have this function tagged with __kprobes, notrace and call
 * read_cr2() before calling anything else. To avoid calling any kind
 * of tracing machinery before we've observed the CR2 value.
 *
 * exception_enter(), exception_exit() contain all sorts of tracepoints.
*/

    prev_state = exception_enter();
    _do_page_fault(regs, error_code, address);
    exception_exit(prev_state);
}
NOKPROBE_SYMBOL(do_page_fault);

#endif CONFIG_TRACING
static nokprobe_inline void
trace_page_fault_entries(unsigned long address, struct pt_regs *regs,
                        unsigned long error_code)
{

```

Ln 1279, Col 11 Tab Size: 4 UTF-8 LF C Mac ⚙

Picture: linux-3.18.140/arch/x86/mm/fault.c

```
fault.c — linux-3.18.140
fault.c arch/x86/mm/fault.c
1046 * This function must have noinline because both callers
1047 * __{trace_}do_page_fault() have notrace on. Having this an actual function
1048 * guarantees there's a function trace entry.
1049 */
1050
1051 static noinline void
1052 __do_page_fault(struct pt_regs *regs, unsigned long error_code,
1053                 unsigned long address)
1054
1055     struct vm_area_struct *vma;
1056     struct task_struct *tsk;
1057     struct mm_struct *mm;
1058     int fault;
1059     unsigned int flags = FAULT_FLAG_ALLOW_RETRY | FAULT_FLAG_KILLABLE;
1060
1061     tsk = current;
1062     mm = tsk->mm;
1063
1064 /*
1065 * Detect and handle instructions that would cause a page fault for
1066 * both a tracked kernel page and a userspace page.
1067 */
1068 if (kmemcheck_active(regs))
1069     kmemcheck_hide(regs);
1070     prefetchw(&mm->mmap_sem);
1071
1072 if (unlikely(kmmio_fault(regs, address)))
1073     return;
1074
1075 /*
1076 * We fault-in kernel-space virtual memory on-demand. The
1077 * 'reference' page table is init_mm.pgd.
1078 *
1079 * NOTE! We MUST NOT take any locks for this case. We may
1080 * be in an interrupt or a critical region, and should
1081 * only copy the information from the master page table,
```

Picture: linux-3.18.140/arch/x86/mm/fault.c