Play With Capture The Flag

david942j @ 217

2018年10月2日 星期二

[Note] Learning KVM - implement your own kernel

Few weeks ago I solved a great KVM escaping challenge from TWCTF hosted by @TokyoWesterns. I have given a writeup on my blog: [Write-up] TokyoWesterns CTF 2018 - pwn240+300+300 EscapeMe, but it mentions nothing about KVM because there's no bug (at least I didn't find) around it.

Most introduction articles of KVM I found are actually introducing either libvirt or qemu, lack of how to utilize KVM directly, that's why I have this post.

This thread is a good start to implement a simple KVM program. Some projects such as kvm-hello-world and kvmtool are worthy to take a look as well. And OSDev.org has great resources to learn system architecture knowledge.

In this post I will introduce how to use KVM directly and how it works, wish this article can be a quick start for beginners learning KVM.

I've created a public repository for the source code of KVM-based hypervisor and the kernel: david942j/kvm-kernel-example. You can clone and try it after reading this article.
Warning: all code in this post may be simplified to clearly show its function, if you want to write some code, I highly recommend you read examples in the repository instead of copy-paste code from here.

The kernel I implemented is able to execute an ELF in user-space, this is a screenshot of the execution result:

```
david942j at ~/kvm-kernel-example on master
→ hypervisor/hypervisor.elf kernel/kernel.bin user/orw.elf /etc/os-release
NAME="Ubuntu"
VERSION="18.04.1 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.1 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic
+++ exited with 0 +++
```

Introduction

KVM (Kernel-based Virtual Machine) is a virtual machine that implemented native in Linux kernel. As you know, a VM is usually used for creating a separate and independent environment. As the official site described, each virtual machine created by KVM has private virtualized hardware: a network card, disk, graphics adapter, etc.

First of all I'll introduce how to use KVM to execute simple assembled code, and then describe some key points to implement a kernel. The kernel we are going to implement is extremely simple, but more features might be added after this post released.

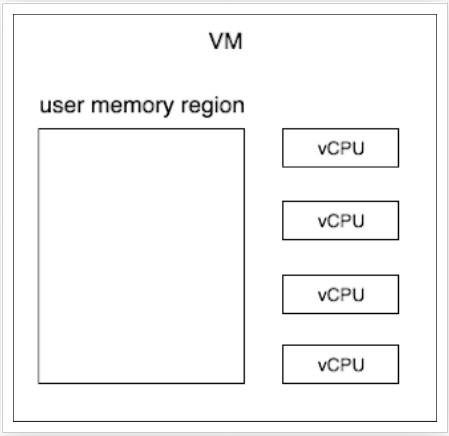
Get Started

All communication with KVM is done by the `ioctl` syscall, which is usually used for getting and setting device status.

Creating a KVM-based VM basically needs 7 steps:

1. Open the KVM device, `kvmfd=open("/dev/kvm", O_RDWR|O_CLOEXEC)`
2. Do create a VM, `vmfd=ioctl(kvmfd, KVM_CREATE_VM, 0)`
3. Set up memory for VM guest, `ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region)`
4. Create a virtual CPU for the VM, `vcpufd=ioctl(vmfd, KVM_CREATE_VCPU, 0)`
5. Set up memory for the vCPU
   - `vcpu_size=ioctl(kvmfd, KVM_GET_VCPU_MMAP_SIZE, NULL)`
   - `run=(struct kvm_run*)mmap(NULL, mmap_size, PROT_READ|PROT_WRITE, MAP_SHARED, vcpufd, 0)`
6. Put assembled code on user memory region, set up vCPU's registers such as `rip`
7. Run and handle exit reason. `while(1) { ioctl(vcpufd, KVM_RUN, 0); ... }`

Too complicated!? See this figure



A VM needs **user memory region** and **virtual CPU(s)**, so all we need is to create VM, set up user memory region, create vCPU(s) and its working space then execute it!

Code is better than plaintext for hackers. *Warning*: code posted here has no error handling**.**

Step 1 - 3, set up a new VM

```c
/* step 1~3, create VM and set up user memory region */
void kvm(uint8_t code[], size_t code_len) {
  // step 1, open /dev/kvm
  int kvmfd = open("/dev/kvm", O_RDWR|O_CLOEXEC);
  if(kvmfd == -1) errx(1, "failed to open /dev/kvm");

  // step 2, create VM
  int vmfd = ioctl(kvmfd, KVM_CREATE_VM, 0);

  // step 3, set up user memory region
  size_t mem_size = 0x40000000; // size of user memory you want to assign
  void *mem = mmap(0, mem_size, PROT_READ|PROT_WRITE,
                   MAP_SHARED|MAP_ANONYMOUS, -1, 0);
  int user_entry = 0x0;
  memcpy((void*)((size_t)mem + user_entry), code, code_len);
  struct kvm_userspace_memory_region region = {
    .slot = 0,
    .flags = 0,
    .guest_phys_addr = 0,
    .memory_size = mem_size,
    .userspace_addr = (size_t)mem
  };
  ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region);
  /* end of step 3 */
  // not finished ...
}
```

In above code fragment I assign 1GB memory (`mem_size`) to the guest, and put assembled code on the first page. Later we will set the instruction pointer to 0x0 (`user_entry`), where the guest should start to execute.

Step 4 - 6, set up a new vCPU

```
/* step 4~6, create and set up vCPU */
void kvm(uint8_t code[], size_t code_len) {
  /* ... step 1~3 omitted */

  // step 4, create vCPU
  int vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, 0);

  // step 5, set up memory for vCPU
  size_t vcpu_mmap_size = ioctl(kvmfd, KVM_GET_VCPU_MMAP_SIZE, NULL);
  struct kvm_run* run = (struct kvm_run*) mmap(0, vcpu_mmap_size,
                                               PROT_READ | PROT_WRITE, MAP_SHARED,
                                               vcpufd, 0);

  // step 6, set up vCPU's registers
  /* standard registers include general-purpose registers and flags */
  struct kvm_regs regs;
  ioctl(vcpufd, KVM_GET_REGS, &regs);
  regs.rip = user_entry;
  regs.rsp = 0x200000; // stack address
  regs.rflags = 0x2; // in x86 the 0x2 bit should always be set
  ioctl(vcpufd, KVM_SET_REGS, &regs); // set registers

  /* special registers include segment registers */
  struct kvm_sregs sregs;
  ioctl(vcpufd, KVM_GET_SREGS, &sregs);
  sregs.cs.base = sregs.cs.selector = 0; // let base of code segment equal to zero
  ioctl(vcpufd, KVM_SET_SREGS, &sregs);
  // not finished ...
}
```

Here we create a vCPU and set up its registers include standard registers and "special" registers. Each `kvm_run` structure corresponds to one vCPU, and we will use it to get the CPU status after execution. Notice that we can create multiple vCPUs under one VM, and with multi-threads we can emulate a VM with multiple CPUs. Note: by default, the vCPU runs in **real mode**, which only executes **16-bit assembled code**. **To run 32 or 64-bit, the page table must be set up**, which we'll describe later.

Step 7, execute!

```
/* last step, run it! */
void kvm(uint8_t code[], size_t code_len) {
  /* ... step 1~6 omitted */
  // step 7, execute vm and handle exit reason
  while (1) {
    ioctl(vcpufd, KVM_RUN, NULL);
    switch (run->exit_reason) {
    case KVM_EXIT_HLT:
      fputs("KVM_EXIT_HLT", stderr);
      return 0;
    case KVM_EXIT_IO:
      /* TODO: check port and direction here */
      putchar(*(((char *)run) + run->io.data_offset));
      break;
    case KVM_EXIT_FAIL_ENTRY:
      errx(1, "KVM_EXIT_FAIL_ENTRY: hardware_entry_failure_reason = 0x%llx",
        run->fail_entry.hardware_entry_failure_reason);
    case KVM_EXIT_INTERNAL_ERROR:
      errx(1, "KVM_EXIT_INTERNAL_ERROR: suberror = 0x%x",
        run->internal.suberror);
    case KVM_EXIT_SHUTDOWN:
      errx(1, "KVM_EXIT_SHUTDOWN");
    default:
      errx(1, "Unhandled reason: %d", run->exit_reason);
    }
  }
}
```

Typically we only care about the first two cases, `KVM_EXIT_HLT` and `KVM_EXIT_IO`. With instruction `hlt`, the `KVM_EXIT_HLT` is triggered. Instructions in and out trigger `KVM_EXIT_IO`. And not only for I/O, we can also use this as hypercall, i.e. to **communicate with the host**. Here we only print one character sent to device.
`ioctl(vcpufd, KVM_RUN, NULL)` will run until an exit-like instruction occurred (such as `hlt`, `out`, or an error). You can also enable the single-step mode (not demonstrated here), then it will stop on every instruction.

Let's try our first KVM-based VM:

```
int main() {
```

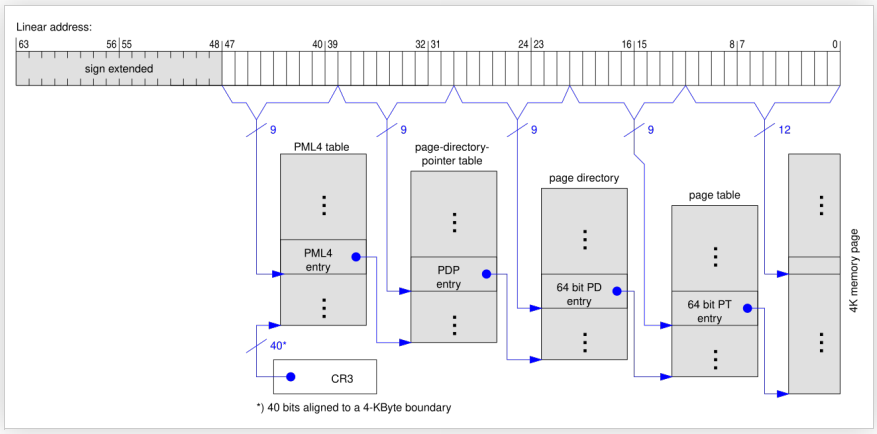
```
  /*
  .code16
  mov al, 0x61
  mov dx, 0x217
  out dx, al
  mov al, 10
  out dx, al
  hlt
  */
  uint8_t code[] = "\xB0\x61\xBA\x17\x02\xEE\xB0\n\xEE\xF4";
  kvm(code, sizeof(code));
}
```

And the execution result is:

```
$ ./kvm
a
KVM_EXIT_HLT
```

64-bit World

To execute 64-bit assembled code, we need to set vCPU into long mode. And this wiki page describes how to switch from real mode to long mode, I highly recommend you read it as well. The most complicated part of switching into long mode is to set up the page tables for mapping virtual address into physical address. x86-64 processor uses a memory management feature named PAE (Physical Address Extension), contains four kinds of tables: PML4T, PDPT, PDT, and PT. The way these tables work is that each entry in the PML4T points to a PDPT, each entry in a PDPT to a PDT and each entry in a PDT to a PT. Each entry in a PT then points to the physical address.



source: https://commons.wikimedia.org

The figure above is called 4K paging. There's another paging method named 2M paging, with the PT (page table) removed. In this method the PDT entries point to physical address.

The control registers (`cr*`) are used for setting paging attributes. For example, `cr3` should point to physical address of `pml4`. More information about control registers can be found in wikipedia.
This code set up the tables, using the **2M paging**.

```
/* Maps: 0 ~ 0x200000 -> 0 ~ 0x200000 */
void setup_page_tables(void *mem, struct kvm_sregs *sregs){
  uint64_t pml4_addr = 0x1000;
  uint64_t *pml4 = (void *)(mem + pml4_addr);

  uint64_t pdpt_addr = 0x2000;
  uint64_t *pdpt = (void *)(mem + pdpt_addr);

  uint64_t pd_addr = 0x3000;
  uint64_t *pd = (void *)(mem + pd_addr);

  pml4[0] = 3 | pdpt_addr; // PDE64_PRESENT | PDE64_RW | pdpt_addr
  pdpt[0] = 3 | pd_addr; // PDE64_PRESENT | PDE64_RW | pd_addr
  pd[0] = 3 | 0x80; // PDE64_PRESENT | PDE64_RW | PDE64_PS

  sregs->cr3 = pml4_addr;
  sregs->cr4 = 1 << 5; // CR4_PAE;
  sregs->cr4 |= 0x600; // CR4_OSFXSR | CR4_OSXMMEXCPT; /* enable SSE instructions */
  sregs->cr0 = 0x80050033; // CR0_PE | CR0_MP | CR0_ET | CR0_NE | CR0_WP | CR0_AM |
CR0_PG
  sregs->efer = 0x500; // EFER_LME | EFER_LMA
}
```

There are some control bits records in the tables, including if the page is mapped, is writable, and can be accessed in user-mode. e.g. 3 (`PDE64_PRESENT|PDE64_RW`) stands for the memory is mapped and writable, and 0x80 (`PDE64_PS`) stands for it's 2M paging instead of 4K. As a result, these page tables can map address below 0x200000 to itself (i.e. virtual address equals to physical address).

Remaining is setting segment registers:

```
void setup_segment_registers(struct kvm_sregs *sregs) {
  struct kvm_segment seg = {
    .base = 0,
    .limit = 0xffffffff,
    .selector = 1 << 3,
    .present = 1,
    .type = 11, /* execute, read, accessed */
    .dpl = 0, /* privilege level 0 */
    .db = 0,
    .s = 1,
    .l = 1,
    .g = 1,
  };
  sregs->cs = seg;
  seg.type = 3; /* read/write, accessed */
  seg.selector = 2 << 3;
  sregs->ds = sregs->es = sregs->fs = sregs->gs = sregs->ss = seg;
}
```

We only need to modify VM setup in step 6 to support 64-bit instructions, change code from

```
  sregs.cs.base = sregs.cs.selector = 0; // let base of code segment equal to zero
```

to

```
  setup_page_tables(mem, &sregs);
  setup_segment_registers(&sregs);
```

Now we can execute 64-bit assembled code.

```
int main() {
  /*
  movabs rax, 0x0a33323144434241
  push 8
  pop rcx
  mov edx, 0x217
OUT:
  out dx, al
  shr rax, 8
  loop OUT
  hlt
  */
  uint8_t code[] =
"H\xB8\x41\x42\x43\x44\x31\x32\x33\nj\bY\xBA\x17\x02\x00\x00\xEEH\xC1\xE8\b\xE2\xF9\xF4";
  kvm(code, sizeof(code));
}
```

And the execution result is:

```
$ ./kvm64
ABCD123
KVM_EXIT_HLT
```

The source code of hypervisor can be found in the repository/hypervisor.

So far you are already able to run x86-64 assembly code under KVM, so our **introduction to KVM is almost finished** (except handling hypercalls). In the next section I will describe how to implement a simple kernel, which contains some OS knowledge. If you are interested in how kernel works, go ahead.

Kernel

Before implementing a kernel, some questions need to be dealt with:

1. How CPU distinguishes between kernel-mode and user-mode?
2. How could CPU transfer control to kernel when user invokes `syscall`?
3. How kernel switches between kernel and user?

kernel-mode v.s. user-mode

An important difference between kernel-mode and user-mode is some instructions can only be executed under kernel-mode, such as `hlt` and `wrmsr`. The two modes are distinguished by the `dpl` (descriptor privilege level) field in segment register `cs`. `dpl=3` in `cs` for user-mode, and zero for kernel-mode (not sure if this "level" equivalent to so-called ring3 and ring0).
In real mode kernel should handle the segment registers carefully, while in x86-64, instructions `syscall` and `sysret` will properly set segment registers automatically, so we don't need to maintain segment registers manually.

And another difference is the permission setting in page tables. In the above example I set all entries as non-user-accessible:

```
pml4[0] = 3 | pdpt_addr; // PDE64_PRESENT | PDE64_RW | pdpt_addr
pdpt[0] = 3 | pd_addr; // PDE64_PRESENT | PDE64_RW | pd_addr
pd[0] = 3 | 0x80; // PDE64_PRESENT | PDE64_RW | PDE64_PS
```

If kernel wants to create virtual memory for user-space, such as handling `mmap` syscall from user, the page tables must set the 3rd bit, i.e. have bit (`1 << 2`) set, then the page can be accessed in user-space. For example,

```
pml4[0] = 7 | pdpt_addr; // PDE64_USER | PDE64_PRESENT | PDE64_RW | pdpt_addr
pdpt[0] = 7 | pd_addr; // PDE64_USER | PDE64_PRESENT | PDE64_RW | pd_addr
pd[0] = 7 | 0x80; // PDE64_USER | PDE64_PRESENT | PDE64_RW | PDE64_PS
```

This is just an example, we should **NOT** set user-accessible pages in hypervisor, user-accessible pages should be handled by our kernel.

Syscall

There's a special register can enable `syscall/sysenter` instruction: EFER (Extended Feature Enable Register). We have used it for entering long mode before:

```
sregs->efer = 0x500; // EFER_LME | EFER_LMA
```

LME and LMA stand for Long Mode Enable and Long Mode Active, respectively.
To enable `syscall` as well, we should do

```
sregs->efer |= 0x1; // EFER_SCE
```

We also need to register syscall handler so that CPU knows where to jump when user invokes syscalls. And of course, this registration should be done in kernel instead of hypervisor. Registration of syscall handler can be achieved via setting special registers named MSR (Model Specific Registers). We can get/set MSR in hypervisor through `ioctl` on `vcpufd`, or in kernel using instructions `rdmsr` and `wrmsr`.
To register a syscall handler:

```
  lea rdi, [rip+syscall_handler]
  call set_handler
syscall_handler:
  // handle syscalls!
set_handler:
  mov eax, edi
  mov rdx, rdi
  shr rdx, 32
  /* input of msr is edx:eax */
  mov ecx, 0xc0000082 /* MSR_LSTAR, Long Syscall TARget */
  wrmsr
  ret
```

The magic number 0xc0000082 is the index for MSR, you can find the definitions in Linux source code.

After setup, we can invoke `syscall` instruction and the program will jump to the handler we registered. `syscall` instruction not only changes `rip`, but also sets `rcx` as return address so that kernel knows where to go back after handling syscall, and sets `r11` as rflags. It will change two segment registers `cs` and `ss` as well, which we will describe in the next section.

Switching between kernel and user

We also need to register the `cs`'s selector for both kernel and user, via the register MSR we have used before.

Here and here describe what does `syscall` and `sysret` do in details, respectively.
From the pseudo code of `sysret` you can see it sets attributes of `cs` and `ss` explicitly:

```
CS.Selector ← IA32_STAR[63:48]+16;
CS.Selector ← CS.Selector OR 3; /* RPL forced to 3 */
/* Set rest of CS to a fixed value */
CS.Base ← 0; /* Flat segment */
CS.Limit ← FFFFFH; /* With 4-KByte granularity, implies a 4-GByte limit */
CS.Type ← 11; /* Execute/read code, accessed */
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
CS.L ← 1;
CS.G ← 1; /* 4-KByte granularity */
CPL ← 3;
SS.Selector ← (IA32_STAR[63:48]+8) OR 3; /* RPL forced to 3 */
/* Set rest of SS to a fixed value */
SS.Base ← 0; /* Flat segment */
SS.Limit ← FFFFFH; /* With 4-KByte granularity, implies a 4-GByte limit */
SS.Type ← 3; /* Read/write data, accessed */
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1; /* 32-bit stack segment*/
SS.G ← 1; /* 4-KByte granularity */
```

We have to register the value of `cs` for both kernel and user through MSR:

```
  xor rax, rax
  mov rdx, 0x00200008
  mov ecx, 0xc0000081 /* MSR_STAR */
  wrmsr
```

The last is set flags mask:

```
  mov eax, 0x3f7fd5
  xor rdx, rdx
  mov ecx, 0xc0000084 /* MSR_SYSCALL_MASK */
  wrmsr
```

The mask 0x3f7fd5 is important, when `syscall` instruction is invoked, CPU will do:

```
rcx = rip;
r11 = rflags;
rflags &= ~SYSCALL_MASK;
```

If the mask is not set properly, kernel will inherit the `rflags` set in user mode, which can cause severe security issues.

The full code of registration is:

```
register_syscall:
  xor rax, rax
  mov rdx, 0x00200008
  mov ecx, 0xc0000081 /* MSR_STAR */
  wrmsr

  mov eax, 0x3f7fd5
  xor rdx, rdx
  mov ecx, 0xc0000084 /* MSR_SYSCALL_MASK */
  wrmsr

  lea rdi, [rip + syscall_handler]
  mov eax, edi
  mov rdx, rdi
  shr rdx, 32
  mov ecx, 0xc0000082 /* MSR_LSTAR */
  wrmsr
```

Then we can safely use the `syscall` instruction in user-mode. Now let's implement the `syscall_handler`:

```
.globl syscall_handler, kernel_stack
.extern do_handle_syscall
.intel_syntax noprefix

kernel_stack: .quad 0 /* initialize it before the first time switching into user-mode */
user_stack: .quad 0

syscall_handler:
  mov [rip + user_stack], rsp
  mov rsp, [rip + kernel_stack]
  /* save non-callee-saved registers */
  push rdi
  push rsi
  push rdx
  push rcx
  push r8
  push r9
  push r10
  push r11

  /* the forth argument */
  mov rcx, r10
  call do_handle_syscall

  pop r11
  pop r10
  pop r9
  pop r8
  pop rcx
  pop rdx
  pop rsi
  pop rdi

  mov rsp, [rip + user_stack]
  .byte 0x48 /* REX.W prefix, to indicate sysret is a 64-bit instruction */
  sysret
```

Notice that we have to properly push-and-pop not callee-saved registers. The `syscall/sysret` will not modify the stack pointer `rsp`, so we have to handle it manually.

Hypercall

Sometimes our kernel needs to communicate with the hypervisor, this can be done in many ways, in my kernel I use the `out/in` instructions for hypercalls. We have used the `out` instruction to simply print a byte to stdout, now we extend it to do more fun things.
An `in/out` instruction contains two arguments, 16-bit `dx` and 32-bit `eax`. I use the value of `dx` for indicating what kind of hypercalls is intended to call, and `eax` as its argument. I defined these hypercalls:

```
#define HP_NR_MARK 0x8000

#define NR_HP_open  (HP_NR_MARK | 0)
#define NR_HP_read  (HP_NR_MARK | 1)
#define NR_HP_write  (HP_NR_MARK | 2)
#define NR_HP_close  (HP_NR_MARK | 3)
#define NR_HP_lseek  (HP_NR_MARK | 4)
#define NR_HP_exit  (HP_NR_MARK | 5)

#define NR_HP_panic (HP_NR_MARK | 0x7fff)
```

Then modify the hypervisor to not only print bytes when encountering `KVM_EXIT_IO`:

```
while (1) {
  ioctl(vm->vcpufd, KVM_RUN, NULL);
  switch (vm->run->exit_reason) {
  /* other cases omitted */
  case KVM_EXIT_IO:
    // putchar(*(((char *)vm->run) + vm->run->io.data_offset));
    if(vm->run->io.port & HP_NR_MARK) {
      switch(vm->run->io.port) {
      case NR_HP_open: hp_handle_open(vm); break;
      /* other cases omitted */
      default: errx(1, "Invalid hypercall");
    }
    else errx(1, "Unhandled I/O port: 0x%x", vm->run->io.port);
    break;
  }
}
```

Take `open` as an example, I implemented the handler of `open` hypercall in hypervisor as: (***warning*: this**

code lacks security checks):

```
/* hypervisor/hypercall.c */
static void hp_handle_open(VM *vm) {
  static int ret = 0;
  if(vm->run->io.direction == KVM_EXIT_IO_OUT) { // out instruction
    uint32_t offset = *(uint32_t*)((uint8_t*)vm->run + vm->run->io.data_offset);
    const char *filename = (char*) vm->mem + offset;

    MAY_INIT_FD_MAP(); // initialize fd_map if it's not initialized
    int min_fd;
    for(min_fd = 0; min_fd <= MAX_FD; min_fd++)
      if(fd_map[min_fd].opening == 0) break;
    if(min_fd > MAX_FD) ret = -ENFILE;
    else {
      int fd = open(filename, O_RDONLY, 0);
      if(fd < 0) ret = -errno;
      else {
        fd_map[min_fd].real_fd = fd;
        fd_map[min_fd].opening = 1;
        ret = min_fd;
      }
    }
  } else { // in instruction
    *(uint32_t*)((uint8_t*)vm->run + vm->run->io.data_offset) = ret;
  }
}
```

In kernel we invoke the `open` hypercall with:

```
/* kernel/hypercalls/hp_open.c */
int hp_open(uint32_t filename_paddr) {
  int ret = 0;
  asm(
    "mov dx, %[port];" /* hypercall number */
    "mov eax, %[data];"
    "out dx, eax;" /* trigger hypervisor to handle the hypercall */
    "in eax, dx;"  /* get return value of the hypercall */
    "mov %[ret], eax;"
    : [ret] "=r"(ret)
    : [port] "r"(NR_HP_open), [data] "r"(filename_paddr)
    : "rax", "rdx"
  );
  return ret;
}
```

Almost done

Now you should know all things to implement a simple kernel running under KVM. Some details are worthy to be mentioned during the implementation.

execve

My kernel is able to execute a simple ELF, to do this you will need knowledge with structure of an ELF file, which is too complicated to introduce here. You can refer to the source code of Linux for details: linux/fs/binfmt_elf.c#load_elf_binary.

memory allocator

You will need `malloc/free` for kernel, try to implement a memory allocator by yourself!

paging

Kernel has to handle the `mmap` request from user mode, so you will need to modify the page tables during runtime. Be careful of NOT mixing kernel-only addresses with user-accessible addresses.

permission checking

All arguments passed from user-mode must be carefully checked. I've implemented checking methods in kernel/mm/uaccess.c. Without properly checking, user-mode may be able to do arbitrary read/write on kernel-space, which is a severe security issue.

Conclusion

This post introduces how to implement a KVM-based hypervisor and a simple Linux kernel, wish it can help you know about KVM and Linux more clearly.
I know I've omitted many details here, especially for the kernel part. Since this post is intended to be an introduction of KVM, I think this arrangement is appropriate.

If you have any questions or find bugs in my code, leave comments here or file an issue on github.
If this post is helpful to you, I'll be very grateful to see 'thanks' on twitter @david942j :D

張貼者： david942j 於 上午9:23

標籤： kvm, linux, note

6 則留言:

Yulin 2019年8月28日 上午11:55

The blog and sample code is awesome! Thank you!

I tried to add exception/interruput handler based on this but fail. The VM exit with "shutdown" reason.

I realize there is not gdt setup in the orignal code. But even after I add it, it still doesn't work.

Could you please let me know what I need to do beside setup gdt and idt?

回覆

回覆

david942j 2019年9月15日 晚上9:37

Seems you have resolved as you mentioned in GitHub ;)

匿名 2022年7月15日 晚上10:52

Such a Mindblowing Post that You have shared here, This is an amazing superb article Keep Sharing this...
Thanks thanks a lottttttttt!!!!
EuropeVPS Hosting

回覆

匿名 2022年5月31日 凌晨2:41

Take your business authority in your hand, and learn the overall web server customization of KVM hypervisor technology. Virtual private server hosting chooses from Onlive Server and know the all over the Singapore VPS Server hosting services plan that is totally managed by Kernel-Based Virtual Machine.

回覆

匿名 2022年5月31日 清晨6:53

This post introduces how to implement a KVM-based hypervisor and a simple Linux kernel. This post is intended to be an introduction to KVM, I think this arrangement is appropriate.
Apart from that, your blog is awesome.
Ukraine VPS Server

回覆

Henry Jenson 2023年9月14日 清晨6:47

Learning KVM and crafting your own kernel is a rewarding journey. If you're considering hosting your virtualized environments, KemuHost's VPS Hosting solutions are designed to support such endeavors. Their services ensure a seamless and reliable experience, allowing you to focus on your kernel development and virtualization projects.

回覆

如要留言，請點按下方的按鈕使用 Google 帳戶登入。

使用 GOOGLE 帳戶登入

較新的文章 首頁 較舊的文章

訂閱： 張貼留言 (Atom)