write-up

how to compile my test program

1. 在根目錄執行 `make` 可以一次編譯所有測試軟體
2. 也可以進入 `userSpaceProgram\*` 各個測試程序手動下 `make` 完成編譯

how to run my test program

請遵循以下步驟

init

1. 執行加入新 API 的 KVM 創建 host VM
2. 在 host VM 中執行任意 KVM 創建 guest VM
3. 編譯 user space program, 把 sheep / mysheep 利用 scp 移入 guest VM
4. 編譯 page-types, 把 page-types 利用 scp 移入 guest VM

Testing new ioctl

1. 執行 guest VM 中的 mysheep
2. mysheep 會自行列印虛擬地址以及文字 (你現在只是一個可愛貝比), ex: 0xaaaad4e60890
3. 利用 `pidof mysheep` 取得 process pid , ex:1885
4. 利用 `./page-types -p <pid> -L | grep <虛擬地址 >> 3 bit>` 取得物理分頁地址 , ex: ./page-types -p 1885 -L | grep aaaad4e60
5. 後面補上原先去掉的 3 bit offset, 即為 gpa (guest physical address) , ex: 0x43e7f890
6. 編輯 host VM 中的 myclient, 把傳入 ioctl 參數 gpa 改成剛剛取得的 gpa 重新編譯後執行
7. 查看 mysheep 列印結果 (恭喜你, 變成酷貝比了)

note: myclient 用 Cpp 寫, 要在 host VM 下載對應 compiler

Code injection

1. 執行 guest VM 中的 sheep
2. 利用 `pidof sheep` 取得 process pid , ex:8473
3. 利用 `./page-types -p <pid> -L` 取得物理分頁地址列表 , ex: ./page-types -p 8473 -L
4. 由於程式很簡單不用思考太多, 取用第一頁物理分頁 ex: 507d4
5. 在 host VM 執行 `objdump -S <path to sheep 執行檔>` 反編譯, 取得會跳轉的目標位置, ex: `71c: 14000000 b 71c <main>` 可以知道主程式會不斷跳轉到 `71c` 邊移
6. 把物理分頁補上偏移, 即為 gpa (guest physical address) , ex: 0x507d471c
7. 編輯 host VM 中的 client, 把傳入 ioctl 參數 gpa 改成剛剛取得的 gpa 重新編譯後執行
8. 查看 sheep 列印結果, 應該要變成 shell

explain my code

代碼說明

我把此部分作業的程式碼拆分成 3 大區塊來做說明

1. ioctl (新增 API 介面)

```
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
    };

     #define KVM_ARM_WRITE_GPA  _IOW(KVMIO, 0xfe, struct
kvm_arm_write_gpa_args)
```

在對應資料夾定義新的 ioctl api 介面

在 `kvm_dev_ioctl` 加入新的 case 處理此新 API 如此一來這個 API 就會是 system api, 可以在 user space 被成功呼叫

接著透過自行撰寫的 `get_vm_by_id` 來取得目標 vm 地址

2. 要求中斷與傳遞參數

```
vm_class_target_vcpu = target_vm->vcpus[0];
vm_class_is_trigger = 1;
vm_class_gpa = gpa_para_struct->gpa;
vm_class_size = gpa_para_struct->size;
memcpy(vm_class_data, gpa_para_struct->data, vm_class_size);
kvm_make_request(KVM_REQ_IRQ_PENDING, vm_class_target_vcpu);
kvm_vcpu_kick(vm_class_target_vcpu);
```

需要利用 extern 在 global 創建共用 buffer 來使不同 ioctl 間傳遞參數, 所以取出參數後要放入 global extern variable 中 值得注意的是, 客戶端的數據要複製到共用 buffer, 因為 ioctl 取得的 data pointer 是客戶端自己的 VA, 另一個 ioctl 的執行者 (qemu) 無法取得數據

`kvm_make_request` 的作用是把中斷請求放入 vcpu

`kvm_vcpu_kick` 的作用是把運行中的 vcpu 退出 guest, 在此處是為了加快該 vcpu 處理中斷請求

3. 處理中斷順便竄改記憶體 `kvm_check_request` 會查看是否有中斷請求, 順便做中斷植入, 因為我在 `kvm_make_request` 傳的請求是 `KVM_REQ_IRQ_PENDING` 所以, 當我查到此中斷且共用參數符合時, 就會執行記憶體竄改

抽象邏輯

1. 利用 qemu 執行 VM
2. 在 VM 中執行要被篡改 MEM 的 process
3. 從 VM 外調用新 ioctl API, 中斷 vcpu 執行, 把數據傳入共用 buffer (ioctl 在 user program 中執行, buffer 在 KVM 實體內存儲)
4. qemu 發現中斷, 額外創建 task 呼叫 ioctl 喚醒 vcpu, 此時在實際啟動前需要處理中斷, 我們在此時進行記憶體竄改
5. 被中斷的 vcpu 使用被篡改的 MEM 被重新啟動

note: ioctl 的執行其實是把呼叫對象代碼拉到同一個 task 由 client 來跑

How my test program work

mysheep / myclient

mysheep: 會定時列印一個字串的內容以及字串的地址

myclient: 調用新的 API 把字串地址換成別的字串

結果: 可以自由改變 mysheep 列印的字串內容, 但長度要控制好

sheep / client

sheep: 不做任何事的無限迴圈

client: 內含跳轉與執行 shell 的程式碼(shellcode), 調用新的 API 把程式碼塞到 sheep 中迴圈的跳轉目的地

結果: sheep 會開始執行程式碼(shellcode), 最終執行一個 shell

answer 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?*

由於 KVM 中 VM 的 MEM 本質上是由一個一個掛載的 memory slot 陣列所組成不是實體記憶體,所以若是在 vcpu 正在被執行時去變更數據會導致記憶體錯誤。

實際上要透過人為的中斷 vcpu, 在 vcpu 停止時竄改 MEM, 再經由完整的內存虛擬化流程啟動記憶體才能順利變更。

我實際實驗了一下直接進行呼叫的結果發現, `kvm_vcpu_write_guest` 回傳 `錯誤地址` 的報錯, 表示在 gpa 和 vha 的轉換甚至無法順利執行。