write-up

code

system ioctl

為了能夠將資料從 userspace on qemu host 複製至 process running on qemu quest；首先需要設定新的 IO number & KVM_ARCH_REQ number，前者包含查詢正確的 `vmid` 與產生 `kvm_make_request()` 兩個 system ioctl 的建立，後者則為 request 的設定。

整個呼叫的流程為：userspace process 透過 ioctl 呼叫 kernel space 的 uapi，在 kernelspace 透過 `vmid` 找出目標 `vcpu` 、複製出 payload、產生 request；等待 qemu (被 context switch) 開始處理 request 時，透過呼叫 `kvm_vcpu_write_guest()` 將資料複製到 guest vm。

test programs

首先開啟 KVM device file descriptors `/dev/kvm` 便能與 kvm API 互動，接者使用 `ioctl()` 就能夠呼叫 kvm API；最後計算 `arm64` instruction size 大小，每個 instruction 佔 4 bytes long，共 11 個 instructions 故佔 44 bytes long；如此填入正確的資訊，便能夠呼叫 kvm API 與 injection the shellcode。

injection attack

preparation

1. compile the `linux/tool/vm` in qemu host, then copy into qemu guest (include `hw2-sheep.c` & `linux/tool/vm/page-types`).
2. copy `my-test.c` & `hw2-test.c` & `Makefile` from ubuntu in to the qemu host.

on guest vm

1. compile the `hw2-sheep.c`
2. 使用 objdump 工具檢查 while loop instruction 所在的位置，找出相對於整個 executable page 的 offset；經過測試找出 offset 為 `71c` 。
3. 運行 `hw2-sheep.out` ，取得 `pid` 。
4. 透過 `cat /proc/<pid>/maps` 可以觀察 virtual address 的分配情況。
5. 最後透過 `./page-types` 取得 virtual address 與 physical address 每個 page 的對應情況。
6. 如此可以計算出目標 while loop 的 gva, gpa，以下圖為例：gva 為 `0xaaaadfa9071c` ，gpa 為 `0x10b27b71c` 。

7. 等待 `hw2-test.out` 呼叫 ioctl 與 request 被處理成功後，便能夠完成 injection attack 且能夠與 shell 互動。

```
root@ubuntu:~# objdump -d ./hw2/hw2-sheep.out --disassemble=main

./hw2/hw2-sheep.out:     file format elf64-littleaarch64


Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

000000000000071c <main>:
 71c:   14000000        b       71c <main>

Disassembly of section .fini:
```

```
root@ubuntu:~# ./hw2/hw2-sheep.out &
[1] 481
root@ubuntu:~# cat /proc/481/maps
aaaadfa90000-aaaadfa91000 r-xp 00000000 fe:00 39982                      /root/hw2/hw2-sheep.out
aaaadfaa0000-aaaadfaa1000 r--p 00000000 fe:00 39982                      /root/hw2/hw2-sheep.out
aaaadfaa1000-aaaadfaa2000 rw-p 00001000 fe:00 39982                      /root/hw2/hw2-sheep.out
ffffb2af6000-ffffb2c51000 r-xp 00000000 fe:00 2935                       /usr/lib/aarch64-linux-gnu/libc-2.31.so
ffffb2c51000-ffffb2c60000 ---p 0015b000 fe:00 2935                       /usr/lib/aarch64-linux-gnu/libc-2.31.so
ffffb2c60000-ffffb2c64000 r--p 0015a000 fe:00 2935                       /usr/lib/aarch64-linux-gnu/libc-2.31.so
ffffb2c64000-ffffb2c66000 rw-p 0015e000 fe:00 2935                       /usr/lib/aarch64-linux-gnu/libc-2.31.so
ffffb2c66000-ffffb2c69000 rw-p 00000000 00:00 0
ffffb2c69000-ffffb2c8a000 r-xp 00000000 fe:00 2856                       /usr/lib/aarch64-linux-gnu/ld-2.31.so
ffffb2c8e000-ffffb2c90000 rw-p 00000000 00:00 0
ffffb2c97000-ffffb2c99000 r--p 00000000 00:00 0                          [vvar]
ffffb2c99000-ffffb2c9a000 r-xp 00000000 00:00 0                          [vdso]
ffffb2c9a000-ffffb2c9b000 r--p 00021000 fe:00 2856                       /usr/lib/aarch64-linux-gnu/ld-2.31.so
ffffb2c9b000-ffffb2c9d000 rw-p 00022000 fe:00 2856                       /usr/lib/aarch64-linux-gnu/ld-2.31.so
ffffdf53f000-ffffdf560000 rw-p 00000000 00:00 0                          [stack]
```

```
root@ubuntu:~# ./page-types -p 481 -L
voffset offset  flags
aaaadfa90       10b27b  __RU_lA____M_____________________________
aaaadfaa0       10afa0  ___U_l_____Ma_b__________________________
aaaadfaa1       10af97  ___U_l_____Ma_b__________________________
ffffb2af6       10153d  __RU_lA____M_____________________________
ffffb2af7       10153e  __RU_lA____M_____________________________
ffffb2af8       10153f  __RU_lA____M_____________________________
ffffb2af9       101540  __RU_lA____M_____________________________
ffffb2afa       101541  __RU_lA____M_____________________________
ffffb2afb       101542  __RU_lA____M_____________________________
ffffb2afc       101543  __RU_lA____M_____________________________
ffffb2afd       101544  __RU_lA____M_____________________________
ffffb2afe       101545  __RU_lA____M_____________________________
```

on host vm

1. compile the test programs。
2. 提供目標 vmid, gpa，以上圖為例：vmid 為 `1` ，gpa 為 `0x10b27b71c` 。
3. 如此便能完成 injection attack。

write to the VM memory

下圖先後執行：

1. **不**透過 `kvm_make_request() -> check_vcpu_requests() -> kvm_check_request()` 流程便直接寫入 VM memory。
2. 透過 `kvm_make_request() -> check_vcpu_requests() -> kvm_check_request()` 流程才寫入 VM memory。

能夠觀察出，若**不**透過流程便直接寫入 VM memory，表示執行的程式仍為 test program，此時所指到的位置並不包含 target VM 的 memory，因此當複製資料至 VM memory 時會產生 `segmentation fault` ；但若透過流程才寫入 VM memory，此時指到的位置就會包含 target VM 的 memory，也就能夠順利地寫入資料。

換句話來說，比較 "若**不**透過流程便直接寫入 VM memory" 與 "若透過流程才寫入 VM memory"， `pid` ， `ttbr` 都指到不同的內容，表示 page table 是不同份，如此一來也變能夠解釋為何會產生 `segmentation fault` 。

```
root@ubuntu:~# dmesg | grep IOCTL
[  425.961720] IOCTL INFO | kvm_arm_write_gpa b: 0
[  425.965444] IOCTL INFO | pid: 700
[  425.965508] IOCTL INFO | ttbr0_el1: 000000010028b000
[  425.965564] IOCTL INFO | ttbr1_el1: 06820000418f2000
[  425.966153] IOCTL ERROR | __copy_to_user return 44
[  425.966290] IOCTL ERROR | kvm_vcpu_write_guest return -14
[  425.966380] IOCTL INFO | kvm_arm_write_gpa b: 1
[  425.966526] IOCTL INFO | pid: 605
[  425.966584] IOCTL INFO | ttbr0_el1: 000000010966a000
[  425.966604] IOCTL INFO | ttbr1_el1: 06760000418f2000
[  425.966716] IOCTL INFO | kvm_vcpu_write_guest success
```