HW0A

survey

Flow

正常的程式流程看起來像：兩次 讀+印東西 後結束, 其實有一個 bof 可以用, 並且長度剛好可以蓋到 return address

Analyze

libc-2.29 會在 main function 中的 old_rbp 殘留 code_section 的 address, 而 old_rbp 前面接著 canary, 所以在第一次讀的時候可以一次將兩個都 leak 出來, 並透過第二次 read 在跳回 read, 達到 code reuse

而這題 `seccomp` 擋掉了 `execve`, 看來只能用 orw, 而 orw 必須要堆 ROP, 所以我們的目標很簡單：

1. leak libc
2. 讓 read 可以讀一堆東西
3. 跳到 rop chain

ATTACK

1

原本要從 stdout 那 leak libc 的, 但是因為當 rbp 遷到 bss 後, 下一次跳回 main function `read()` 時, rsp 會被改成 bss section 的位置, 而 `printf()` 會因為底層實作會做與 rsp 相對位置相關的 operation, 會因為那個 **rsp 相對位置** 是不可存取的所以爛掉, 因此不走從 stdout leak libc, 而是改從 `fflush()` 後, 在 stack 上殘留的 FILE structure 的 member (in libc), 來 leak libc

2 + 3

我們假設 orw ROP 最後會寫到 ROP_CHAIN 內

short ROP

因為 `read()` 前面可以塞一些 gadget, 所以可以先在這輪放一些 GADGET, 之後透過控制 rbp, 讓 rbp+0x10 剛好接在之前輸入的 GADGET, 就可以使用長一點的 ROP chain 來做事

不過因為要接著舊的 GADGET, 所以下一輪的 rbp 勢必會比此輪的還要低, 而 `fflush()` 的會把舊的 GADGET 給洗掉, 猜測應該是 rsp 與 rbp 過近, 讓 `fflush()` 殘留物把 GADGET 洗成 IO 的東西了
解決方法是在下輪時將 rbp 遷到一個遠一點的地方, 並且不做任何事；然後在下下輪才控制 **rbp+0x10 剛好接在之前輸入的 GADGET**

此 ROP 我是先控 rdx, 也就是 `read()` 的大小, 將 rdx 弄成很大的值後, 可以在跳回 `read()` 時寫完整的 orw ROP chain `movsxd_rdx_esi_ret` => `p64(code_base+0x1261)`

P.S. 這邊有一個注意的點，因為此時 rsi 是指向 libc bss 的某個位置, 為 **ROP_CHAIN-0x60**, 那個位置是紀錄 IO 的 data, 預設為 0, 如果更動的話會讓程式 crash, 繞過的方法為寫一堆 b'\x00' 進去 (這邊是 0x60) ===> **`read()` 寫到的位置 + 0x60 個 b'\x00' 後, 會是 ROP_CHAIN 的位置**

rbp + ret

到這邊已經可以寫到 libc bss 的特定位置, 並且在不影響 file operation 的情況下寫 orw ROP chain 了, 那該怎麼跳 ?

我讓執行 short ROP 時, 將 old_rbp 改成 **ROP_CHAIN-0x10**, 並在下一輪的 return addr 改成 `ret` GADGET, 因此在下一輪執行完後, 最後會跳到 **ROP_CHAIN** 上, 執行 ROP chain

EXPLOIT

1. 寫 short ROP (main 裡面的 `read()`)
2. 亂遷 rbp, 避免 `fflush()` 改掉 short ROP
3. 寫 short ROP (movsxd_rdx_esi_ret), 配合 1., 可以在改完 rdx 後跳回 main 裡的 `read()` ; old_rbp 為 rop_addr - 0x10
4. 寫 0x60 個 b'\x00' 避免 file operation 壞掉, 之後寫 orw ROP
5. 隨便跑完後, 配合 3. 的 old_rbp, 透過 ret 會跳到 orw ROP

```
### 1
r.sendafter('What is your name : ', b'A'*0x1) ### garbage
r.sendafter('Leave your message here : ', p64(code_base+0x1261)*3 + p64(canary) +
p64(bss) + p64(printf)) ### migrate stack

### 2
r.sendafter('What is your name : ', b'B'*0x1) ### garbage
r.sendafter('Leave your message here : ', b'F'*0x18 + p64(canary) + p64(bss -
0x200 - 0x20) + p64(printf)) ###

### 3
r.sendafter('What is your name : ', b'C'*0x1) ### garbage
r.sendafter('Leave your message here : ', b'G'*0x18 + p64(canary) + p64(rop_addr -
0x10) + p64(movsxd_rdx_esi_ret)) ###

### 4
r.sendafter('Thanks', b'\x00'*rop_offset + ROP)
### 5
r.sendafter('Leave your message here : ', b'H'*0x18 + p64(canary) + p64(bss) +
p64(ret)) ###
```

Other

這題的 `fflush()` 真的很搞人 XD, 當初在寫的時候有一個版本 local 會過, remote 不會過, 就是因為 buffer 的問題, 後來聽助教說, 因為 local 與 remote 的 buffer type default 不同, 所以可能會造成 `fflush()` 有不同的作為

後來還發現 no ASLR 的情況下, 原本可以跑得 payload 就不能跑了..., 不過因為這題搞 `fflush()` 花太多時間了, 後來也沒有仔細研究

ROBOT

Flow

parent, child 各有一個 memory region, 而 child 可以透過跑 shellcode, 將資料傳給 parent 的 region (fmt), 同時, child 必須 alive, 才能做下一次的 write

parent 會隨機 random 4 個數字, 分別為 val1, 2, 3, 4

fmt 是 char ptr, 指向 parent region, 並且 parent 會藉由第一個字元 S G M 做出不同的動作

- S: fmt 的 0 ~ 4 變成 val1 ~ 4 與 1 個 0
- M: 會看 fmt[1] 是 WASD, 像是上下左右一樣. 他會取 val1 與 val2 作為 x, y 並且看 move 之後的 x 與 y 是否在 0 ~ 99 之間
- G: 把 child process kill 掉
- 都不是的話, 會把 fmt[0] 清為 0

最後有一個判斷 val1 == val3 && val2 == val4, 如果對的話就... ? `NOTFLAG{Super shellcoder}`

之後 val3 與 val4 會隨機往 WASD 方向走一格, 最後 dprinf fmt **Attack**

而 pipe number 分別為

- dpipedes: 3, 4
- read_fd, write_fd: 5, 6

關鍵 breakpoint

0x55555555538f => main addr 0x5555555553f1 => fork 0x5555555555d1 => read

Function 分析

- P_PID: Wait for the child whose process ID matches id
- pipe(O_DIREC T | O_CLOEXEC) # 0x84000
  - O_CLOEXEC == FD_CLOEXEC: child exec 後 close pipe
  - O_DIRECT: "packet" mode, 不會先將資料寫入 buf, 直接寫, 可避免 output 出不來之類的 (?)
- mmap(0LL, 0x100uLL, 3, 0x22, -1, 0LL)
  - 3 = PROT_READ(1) | PROT_WRITE(2)
  - 0x22 = MAP_PRIVATE(2) | MAP_ANONYMOUS(0x20
- mmap(0LL, 0x100uLL, 7, 0x22, -1, 0LL)
- waitid(P_PID, id, &infop, 5);
  - WNOHANG: 1
  - WEXITED: 4

Analyze

CHILD 一開始只知道 vmmap start (in rdx), 所以要先將他好好保管, 放到平常用不到的 register, 在這邊我選 r14

leak: 可以透過 format %p %p 等等撈 code, libc 等等位置 code => 拿 got 位置 libc => 拿 system 位置 rbp => 拿 stack 位置

找到 stack 中是否有 code 段的東東 X2, 由 1 將 2 指到的位置改成 target, 2 就可以寫到 target

Attack

步驟大概如下：

1.1 CHILD: write() 先送 %p %p 等等 format string, leak 出一些 address info 1.2 PARENT: 回傳 libc, code section 的位置, 此時順便觀察 stack 上的情況, 是否有 1->2, 2->X, X->code_section

在此假設:

- %15 為 1
- %17 為 2
- %19 為 3
- *%15 == %17
- target == strcpy 的 got

2.1 CHILD: write("%XX%c%15$hn"), 用 1 改 2 (XX == (&%19) & 0Xffff) 2.2 PARENT: 成功更新 2 裡面的值

3.1 CHILD: write("%XX%c%17$hn"), 用 2 改 3 (XX == (&target) & 0xffff) 3.2 PARENT: 成功更新 3 裡面的值

4.1 CHILD: write("%XX%c%19$hn") => 因為 %19 存著 target address, 所以可以寫東西到裡面 4.2 PARENT: 成功更新 target 裡面的值

CHILD 應該會是： write -> read -> write -> read -> ...

TARGET

原本是想把 strcpy 改成 system, 這樣傳 fmt='MA;/bin/sh' 就會跑到 strcpy 執行 system('MA;/bin/sh')，但是發現 strcpy 不能改, 策略改成

```
kill => mmap(code_base + 0x1957)
close => jmp rax (libc_base + 0x26eb5)
```

這樣在 parent `kill()` child 時, 會往下跳開一個可執行的 memory region, 並透過 `fgets()` 吃 shellcode, 最後在 `close(0)` 時透過 `jmp rax` 跳到 shellcode 上執行