HW3 Writeup

R10922043 黃政瑋

CTF account: cwhuang1937

1. fifo

(1) 靜態分析題目

首先用 IDA 反編譯後，該程式會 fork 一個 child process，而 child 所拿到的 pid 等於 0，故會進入 if 裡面 execve 一個 file。

```
50  pid = fork();
51  if ( pid <= 0 )
52  {
53    if ( !pid )
54    {
55      setsid();
56      argv.st_dev = (__dev_t)file;
57      argv.st_ino = 0LL;
58      execve(file, (char *const *)&argv, 0LL);
59      exit(0);
60    }
61    puts("Bye");
62    exit(0);
63  }
```

(2) 動態分析題目

直接在 0x0000555555555514 設 breakpoint 後，即可知道 child 執行的 file path 為**/tmp/khodsmeogemgoe** 。

```
──────────────────────────────────────────────── arguments (guessed) ──
open@plt (
   $rdi = 0x00007fffffffde70 → "/tmp/khodsmeogemgoe",
   $rsi = 0x0000000000000241,
   $rdx = 0x00000000000001e4
)
──────────────────────────────────────────────────────────────── stack ──
```

接著在 0x0000555555555583 再設一個 breakpoint，並步過該指令後，再**開另外個 gdb 掛上 child process**。此時先繼續 debug parent

的程式，會發現在 0x00005555555555ea 處，會 call mkfifo，上網查之後，parent 會藉由這個檔案來當作與 child 溝通的管道(感覺有點類似 pipe 的概念)。

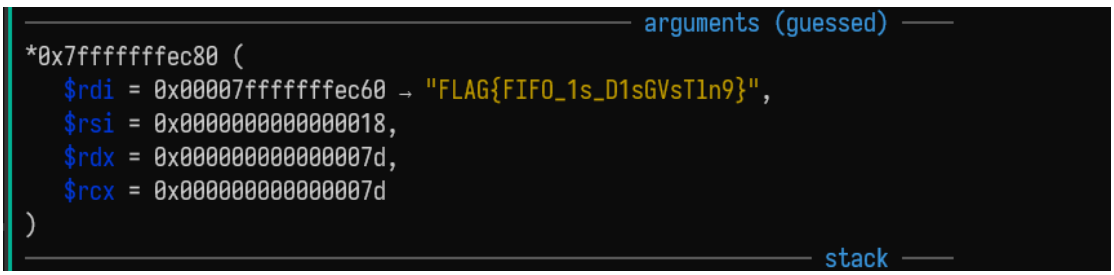
(3) 對 child 執行的檔案做靜態分析

接著再用 IDA 對/tmp/khodsmeogemgoe 做分析，果真發現該程式會**從/tmp/bnpkevsekfpk3/aw3movsdirnqw 這個檔案讀資料到 buf，並試著去執行他**。

(4) 繼續做動態分析

Parent 繼續往後 debug 時，當 open /tmp/khodsmeogemgoe 時，此時會卡住，接著換對 child 按一次步過後，此時 parent 即可繼續執行並寫入東西到/tmp/bnpkevsekfpk3/aw3movsdirnqw。

繼續對 child 做 debug 後，發現他讀 parent 寫進去的 data 到 buf 後，即可成功看到 FLAG 了，此時 child 去執行該 function 就會當掉，這邊只能手動 kill 掉才能終止程式。



速解:

這題我最一開始只是單純 parent 執行下去，這時會看到 child 的 pid，直接用 gdb 掛上去該 pid 即可在 stack 拿到 flag 了。

2. giveUflag

(1) 分析題目

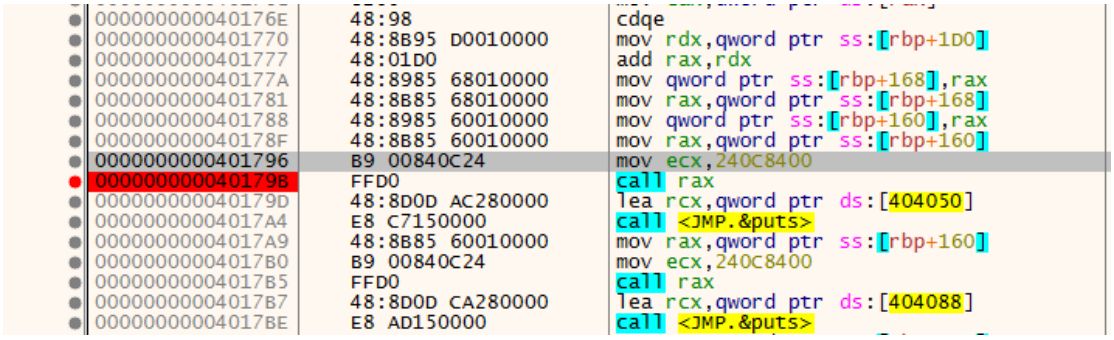
直接點開程式後發現會整個卡住。接著用 IDA 反編譯後，從 main 開始往後點幾個 subroutine，發現在 sub_4015F3 藏有一些不尋常的 code(puts 的網址點開是一些梗圖)，故決定先從這邊開始逆。

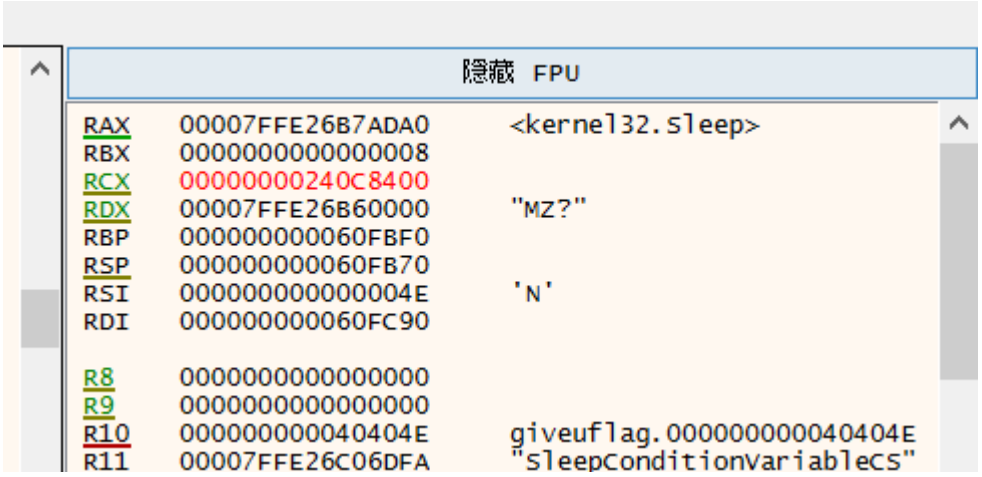
```
v7 = a1 + v11[8];
for ( i = 0; i < v9; ++i )
{
  String1 = (char *)(a1 + *(int *)(v7 + 4 * i));
  if ( !stricmp(String1, "sleep") )
    break;
}
v5 = (void (__fastcall *)(__int64))(a1 + *(int *)(v8 + 4 * i));
v4 = v5;
v5(604800000i64);
puts("https://i.ytimg.com/vi/_T2c8g6Zuq8/maxresdefault.jpg");
v4(604800000i64);
puts("https://i.ytimg.com/vi/MY4sFW83yxg/maxresdefault.jpg");
v4(604800000i64);
puts("https://i.ytimg.com/vi/OVuZ4vGxVKE/maxresdefault.jpg");
for ( j = 0; j <= 44; ++j )
  Buffer[j] = off_403020[j] ^ v3[4 * j];
puts(Buffer);
return v5;
}
```

(2) 動態 debug

接著在 sub_4015F3 處設 breakpoint(00000000004015f3)，接著不

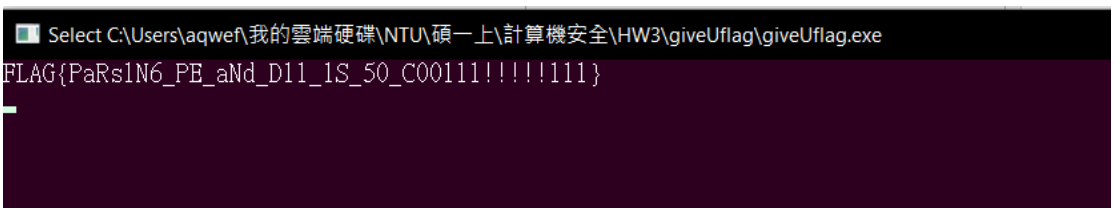
斷 F8 步過，會發現在 000000000040179B 處會 call rax，其中 rax 為 **kernel32.Sleep**。而 sleep 的時間為 604800000ms，恰好符合梗圖的 7 天。





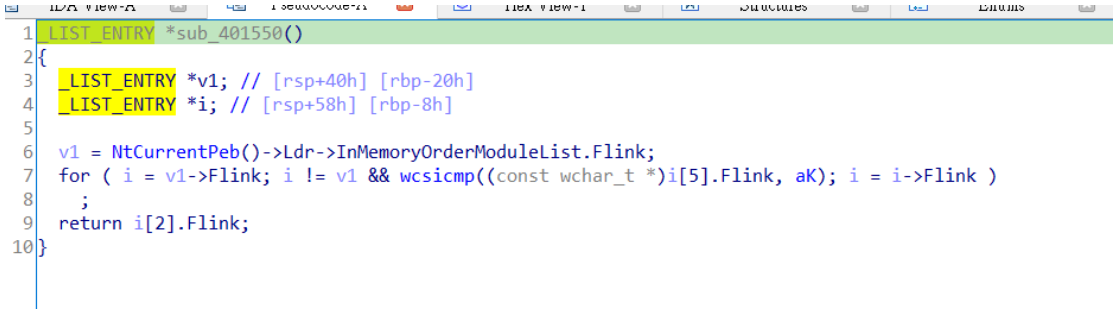
(3) 改 RIP 來跳過 kernel32.Sleep

重新執行後，這次直接將 breakpoint 設在 call rax 的地方 (000000000040179B)，到該 breakpoint 後，搭配 IDA 查看，將 RIP 設成 00000000004017DD 後，即可成功跳過 sleep 的部分，接著直接用 ctrl+F9 跳到 ret，即可成功看到 flag 顯示在視窗上。

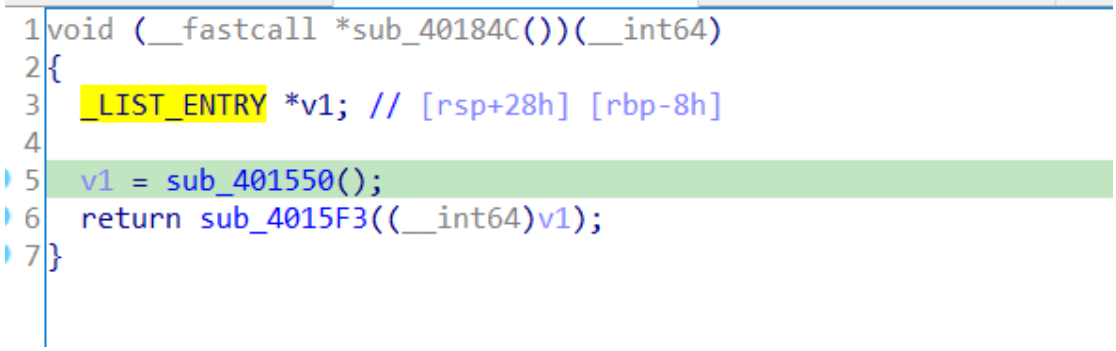


題外話:

在動態 debug 前，從 sub_401550 有發現他藉由爬 PEB 來拿一些 API，並將其回傳出去，然後當作參數傳給上述可能藏有 code 的 sub_4015F3。但動態跑下去後發現 call rax 那邊就直接看到 rax 是什麼了，故這邊並沒有手動去爬 PEB，即可成功知道 rax 為 kernel32.Sleep。



```
1  _LIST_ENTRY *sub_401550()
2  {
3    _LIST_ENTRY *v1; // [rsp+40h] [rbp-20h]
4    _LIST_ENTRY *i; // [rsp+58h] [rbp-8h]
5
6    v1 = NtCurrentPeb()->Ldr->InMemoryOrderModuleList.Flink;
7    for ( i = v1->Flink; i != v1 && wcsicmp((const wchar_t *)i[5].Flink, aK); i = i->Flink )
8      ;
9    return i[2].Flink;
10 }
```

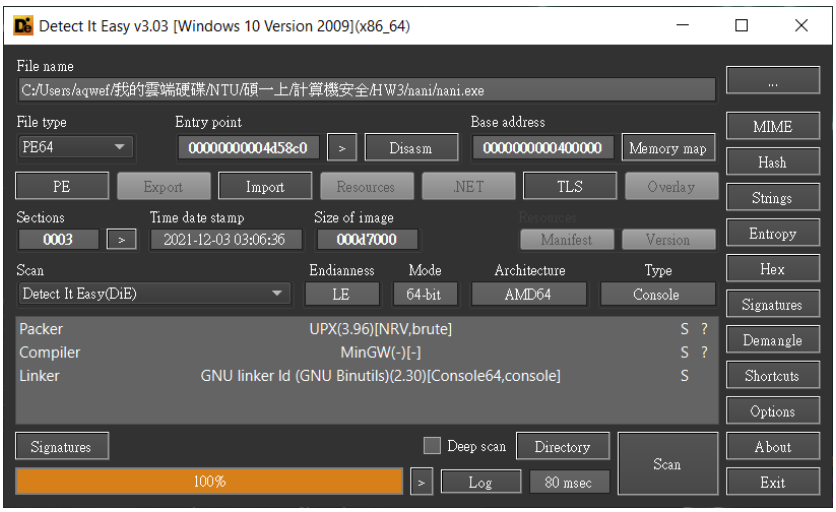


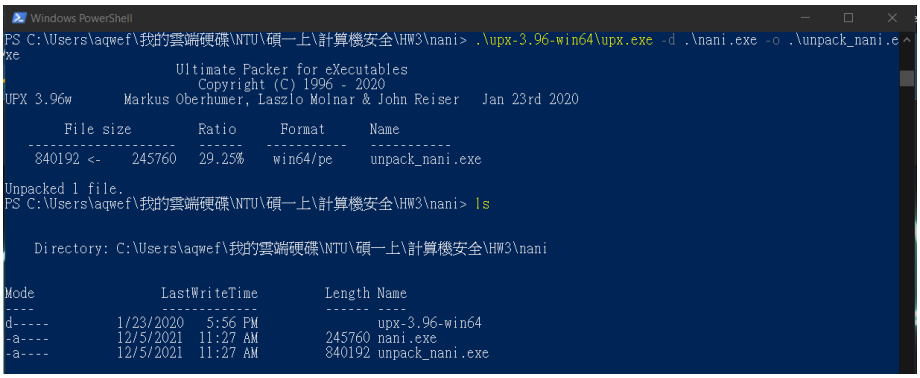
```
1 void (__fastcall *sub_40184C())(__int64)
2 {
3   _LIST_ENTRY *v1; // [rsp+28h] [rbp-8h]
4
5   v1 = sub_401550();
6   return sub_4015F3((__int64)v1);
7 }
```

3. nani

(1) 解 parker

用 DIE 查看，會發現用 **UPX3.96 加殼過**，上網找對應的 upx 做 decompress，即可 unpack。





(2) 解 Anti-Disassembly

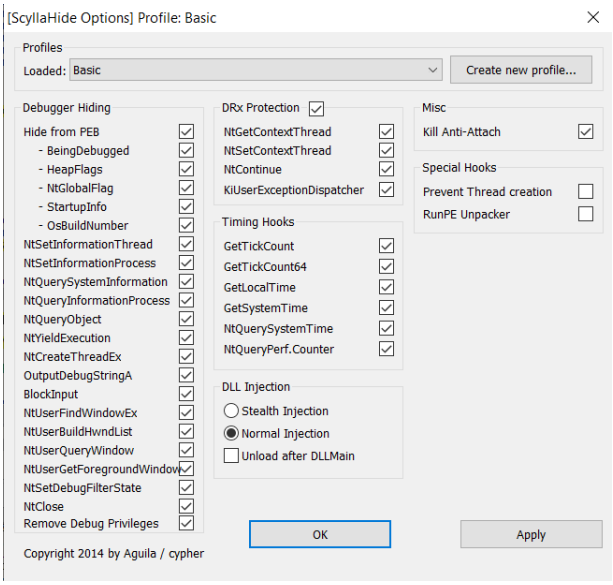
查看原本的 graph-view 發現怪怪的，仔細看 text-view 會發現似乎有被塞 trash code 導致無法成功反編譯，**故這邊手動 undefined 後，把往後的一行也塞進來 trash code** 即可成功反編譯。

```
.text:00000000004019B7         call    $+5
.text:00000000004019BC
.text:00000000004019BC loc_4019BC:                             ; DATA XREF: sub_4019A3+1A↓o
.text:00000000004019BC         pop     rax
.text:00000000004019BD         add     rax, (offset loc_4019C9 - offset loc_4019BC)
.text:00000000004019C1         jmp     rax                     ; goto unk_4019C9
.text:00000000004019C1 ; ---------------------------------------------------------------------------
.text:00000000004019C3         db 0E9h, 80h, 87h, 55h, 66h, 1
.text:00000000004019C9 ; ---------------------------------------------------------------------------
.text:00000000004019C9
.text:00000000004019C9 loc_4019C9:                             ; DATA XREF: sub_4019A3+1A↑o
.text:00000000004019C9         mov     rax, cs:IsDebuggerPresent
.text:00000000004019D0         call    rax ; IsDebuggerPresent
.text:00000000004019D2         test    eax, eax
.text:00000000004019D4         setnz   al
```

(3) 解 Anti-Debug

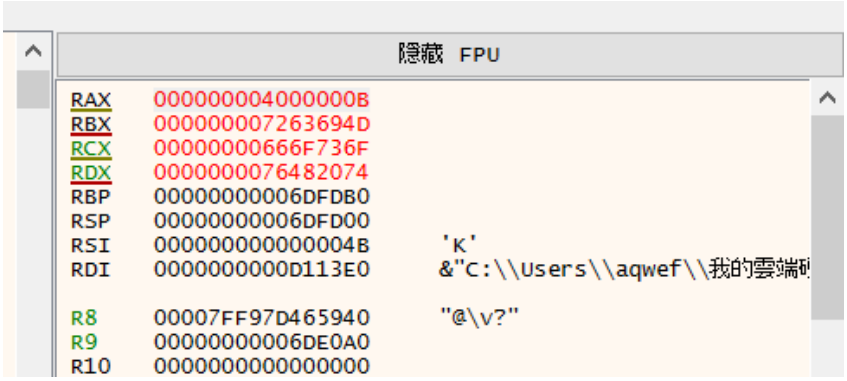
從上面反編譯後，發現它會偵測 Debugger，若被偵測到會直接 exit，

因此這邊**將 x64dbg 裝上 ScyllaHide**，並將 options 全開即可繞過。



(4) 解 Anti-VM

執行完 cpuid 後，會將字串存在 RBX、RCX、RDX 裡面，用 python 解析出來後果真是他的 6 個 blacklist 之一，因此這邊**直接用 RIP 跳到 0x0000000000401990**，即可成功繞過 anti-VM。



(5) 逆 exception handler

繞過前面的 VM 後，從 IDA 會看到有一個 exception handler，因此

x86dbg 在 sub_4016FB 的地方設斷點，接著繼續用 IDA 靜態分析看 handler，會發現該 handler 會對 **byte_4015AF** 做一些操作。

```
; Attributes: noreturn bp-based frame

sub_4017DF proc near

var_4= dword ptr -4

; __unwind { // sub_4016FB
push    rbp
push    rsi
push    rbx
mov     rbp, rsp
sub     rsp, 30h
mov     [rbp+var_4], 0
mov     ecx, 10h
call    sub_4A0560
mov     rbx, rax
lea     rdx, unk_4A5001
mov     rcx, rbx
call    sub_482850
mov     r8, cs:off_4A9B90
lea     rdx, _ZTISt14overflow_error ; `typeinfo for'std::overflow_error
mov     rcx, rbx
call    sub_4A0D40
```

```
 9
10  v6 = &byte_4015AF;
11  v5 = 256;
12  if ( !VirtualProtect(nullsub_1, 0x1000ui64, PAGE_EXECUTE_READWRITE, &v6) )
13    exit(1);
14  for ( i = 0; i < v5; ++i )
15    v6[i] ^= 0x87u;
16  if ( !VirtualProtect(nullsub_1, 0x1000ui64, PAGE_EXECUTE_READ, &flOldProtect) )
17    exit(1);
18  *(v8 + 0xF8) = &byte_4015AF;
19  return 0i64;
20}
```

x64dbg 跟下去後，byte_4015AF 仍無法看到 FLAG，推測 byte_4015AF 可能是一段 code，這邊用 **RIP 直接設成 4015AF 跳過去**，執行下去即可拿到 FLAG。