Homework 0x03

1. fifo

Concept

一開始用 IDA 對 fifo 進行分析，發現此程式會寫兩個檔案，透過 gdb 分析可以得知第一個檔案是`/tmp/khodsmeogemgoe`，第二個檔案是`/tmp/bnpkevsekfpk3/aw3movsdirnqw`，而此檔案為一個 fifo Pipe。執行完程式後第一個檔案會留著，第二個檔案會被刪除 (? 。接著繼續分析，可以發現程式裡用到 fork，且 fork 出來的 child process 去執行了`/tmp/khodsmeogemgoe`這個檔案，

接著使用 IDA 對`/tmp/khodsmeogemgoe`分析，發現檔案會開啟`/tmp/bnpkevsekfpk3/aw3movsdirnqw`fifo 並且從該 fifo 讀取某個數值後，經過某種運算把某個東西算出來。在這邊先猜測這裡可能就是算出 Flag 的地方。

為了要拿到該 child process 的運算結果，上網搜尋到使用 strace 等方法把 process 的 output 印出來，但結果皆失敗。此外也有試著在用 gdb 在寫入 fifo 前設下斷點，並且在此時開啟`/tmp/khodsmeogemgoe`檔案，接著繼續執行，然而得到的結果會是 unlink()。

接著我查詢了 gdb debug child process 的相關資料，找到若在 gdb 輸入

```
set follow-fork-mode child
```

可以在程式 fork 的時候去 debug child process 而不是預設的 parent process。一開始設定後直接使用，會讓 gdb 當掉。接著我試著在 child process 的 execve 這行下斷點。

```
b *0x0000555555554000+0x00000000000016DB (execve)
```

此時 parent process 也被 block 住了，因為 parent process 在寫入 fifo 時，也會 block，直到有其他 process 來讀取後才會繼續執行。
接著繼續執行，發現一樣會卡住，但是在按下 ctrl+c 的時候會跳出去，並且在 gdb 界面拿到 flag。

後來有想到比較好的做法:
既然 child process 跟 parent process 很難同時觀察，我想到可以自己開一個`/tmp/khodsmeogemgoe`來讀 parent process 要傳的東西，並且也同樣用 gdb 進行觀察。因此首先我執行 gdb fifo，並且設斷點在 write fifo 那行，

```
b *0x0000555555554000+0x000000000000165B (write)
```

接著開啟另一個視窗，用 ps -aux | grep khod 找到 fork 出來的 child process，把它刪掉避免該 process 把資料讀走。接著執行`gdb /tmp/khodsmeogemgoe`，並設斷點在 read 那行。

```
b *0x0000555555554000+0x00000000000014B5 (read)
```

最後回到 gdb fifo 執行 write，再切回 gdb `/tmp/khodsmeogemgoe`，成功 read 資料並繼續執行成功拿到 flag。

2. giveUFlag

Concept

使用 IDA 查看 source code，發現程式會卡在某個 function，該 function input 某個數字，且 function 附近有一個網址圖片代表 1 week later，因此猜測這是一個 sleep function，且數字是代表 1 week 的時間，而實際去計算該 function 的參數後發現也相符。這隻程式總共會 sleep 3 week 才會繼續運作。

因此我使用 x64dbg 的" 跳過下一行" 指令功能，在 sleep 函式設下斷點。接著跳過三個 sleep function ，並且繼續執行。最後成功在 x64dbg 的界面拿到 flag。

3. nani

Concept

一開始執行檔案後發現有提示 unpack me，且使用 IDA 觀察檔案後發現只有簡單幾個 function 而已。因此第一步試著先把檔案 decompress。首先使用 DIE 觀察該執行檔使用的 uxp 版本為 3.96，因此下載 uxp 3.96 並且下指令 `.\upx.exe -d nani.exe -o nani_unpack.exe` 成功 decompress 檔案。

unpack 之後使用 IDA 發現解析出的內容變得非常多，無法每個 function 都仔細看過，因此試著先從 main function 並根據程式印出來的提示來 trace。若直接執行檔案會出現 You use VM... 的字串，因此推斷程式裡可能會判斷現在環境是否在 VM 裡面。接著若使用 x64dbg 解析並執行後發現會印出 You are debugger... 的字串，因此推斷程式裡也會判斷是否使用 debugger。

順著 main function 下去 trace code 在sub_4019C9找到了 IsDebuggerPresent()，因此這邊下載了 Scyllahide 外掛程式並勾選相關條件讓這邊不會被判斷為 debugger。

接著在sub_401869找到了cpuid指令，懷疑這裡是檢查執行環境是否為 vm 的地方。在查詢後發現這裡會使用 cpuid 回傳的東西去一一跟黑名單比對，若比對成功則跳出程式。這裡想不到甚麼好解因此就使用 x64dbg 跳過比對 cpuid 回傳字串跟黑名單的地方，讓程式能夠繼續順利執行。

繞過以上判斷後，會進入sub_4017DF()裡面，接著發現最後程式會執行到sub_4aC9D0()，跑到 Raise Exception 就結束了。因為 Raise exception 那邊沒有任何的 if 條件，是無論如何都會執行到的地方，因此一開始使用了 pe-bear 查看 raise exception 的 address，並試著從該 address 找出 handler，但發現找不太到對應的 handler 位置。

後來其實是在不斷的使用 F7(步入指令) 後，意外的在 x64dbg 的變數列表中發現在位置4016C8的附近的迴圈裡面會依序印出 F L A G 的字，因此懷疑這邊是產生 Flag 的地方，結果也確實如此。不過因為這個解法不太有邏輯所以後來又繼續去研究這個程式。

後來在不斷嘗後使用 Pe-bear 查到外層 function: sub_4017DF的 handler 在sub_40169B()，而且 Raise exception 後也確實會跑到sub_40169B()，因此推測在sub_4aC9D0()發生 exception 後會跑到sub_40169B()

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
BD0A4	19	0A	05	05	0A	52	06	03	03	30	02	60	01	50	00	00
BD0B4	FB	16	00	00	FF	9B	21	01	14	28	05	43	03	3E	05	53

图 1: ”sub_4017DF” error handler

接著在sub_4016()裡面發現有 virtualProtect function，參數 protect option 是 0x40，代表的意思為:PAGE_EXECUTE_READWRITE。在改寫權限後程式對 v6，也就是記憶體中的某段 section(byte_4015AF) 不斷進行改寫的動作。推測這邊把原本的指令做了一些修改。

接著使用 F7 繼續執行，發現程式會執行到 40169B 的地方，也就是上述提到找到 flag 的地方，而我也觀察到這個區段本來不是長這樣，而是經過前面的改寫後才變成現在這樣。