

# 2023 NTU Computer Security HW2 Writeup

StudentID : R12922054

## [Lab] AssemblyDev

- FLAG{c0d1Ng\_1n\_a5s3mB1y\_i5\_sO\_fun!}

### 解題流程和思路

本題的解題技巧就是想辦寫出符合題目的 Assembly code, 並且把將其轉成 base64 依序回答即可拿到 Flag。每份 Assembly code 我都有在旁寫註解可以參考!

```
Give me your base64 of your assembly code!
>bW92IGVheCwgZHdvcnQgW3JzcF07IGxldCBLYXggPSBhCmltdWwgZWR4LCBLYXgsIDkKYWRkIGVkeCw
gNwoKbW92IGVieCwgZHdvcnQgW3JzcF0KbW92IGVjeCwgZHdvcnQgW3JzcCs0XQphZGQgZWf4LCBLY3g
Kc3ViIGVieCwgZWN4Cgptb3YgZWN4LCBkd29yZCBbcnNwKzhdCm5lZyBLY3gKCg==
```

Figure 1: arithmetic\_Cmd.

```
Give me your base64 of your assembly code!
>0yBNb2RpZnkcmVnaXN0ZXIgdmFsdWVzCmFkZCByYXgsIDB40DcgICAgiCAgICAgiCAgICAgiCAgICA
gIDsgUkFYICs9IDB40DcKc3VlIHJieCwgMHg2MyAgICAgiCAgICAgiCAgICAgiCAgICAgiCAg0yBSQlggLT0
gMHg2Mwp4Y2hnIHJieCwgcmR4ICAgiCAgICAgiCAgICAgiCAgICA7IFN3YXAgUkNYIGFuZCBSRFg
KCjsgTW9kaWZ5IG1lbW9yeSB2Ywx1ZXMKYWRkIGR3b3JkIFtgc3BdLCAweGRlYWRiZWhVmICAgiCAgICA
g0yBNRU1bUlNQKzB4MDpSU1ArMHg0XSArPSAweGRlYWRiZWhVmCn1YiBkd29yZCBbcnNwKzRdLCAweGZ
hY2ViMDBjICAgICAgiDsgTUVNW1JTUCsweDQ6UlNQKzB40F0gLT0gMHhmYWNLjAwYwoKYWRkIHJzcCw
g0Aptb3YgcjhkLCBkd29yZCBbcnNwXSArICAgiCAgICAgiCAgICA7IEvxYWQgTUVNW1JTUCsweDg6UlN
QKzB4Y10gaW50byBy0GQKbW92IHI5ZCwgZHdvcnQgW3JzcCs0XSArICAgiCAgICAgiCAg0yBMb2FkIE1
FTVtSU1ArMHhj0lJTUCsweDEwXSbpbnRvIHI5Zaptb3YgZHDvcnQgW3JzcF0sIHI5ZCAgICAgiCAgICA
gICA7IFN0b3JlIHI5ZCbpbnRvIE1FTVtSU1ArMHg40lJTUCsweGndCm1vdiBkd29yZCBbcnNwKzR
dLCBy0GQgICAgiCAgICAgiDsgU3RvcnUgcjhkIGludG8gTUVNW1JTUCsweGM6UlNQKzB4MTBdCg=
```

Figure 2: datamovement\_Cmd.

```
Give me your base64 of your assembly code!
>bW92IGVheCwgZHdvcnQgW3JzcF0gICAgiCAgICA7IEvxYWQgYSBmcn9tIFtSU1BdCm1vdiBLYng
sIGR3b3JkIFtgc3AgKyA0XSArICAgiCAg0yBMb2FkIGIgZnJvbSbbUlNQKzRdCm1vdiBLY3gsIGR3b3J
kIFtgc3AgKyA4XSArICAgiAk7IEvxYWQgYBmcn9tIFtSU1Ar0F0KbW92IGVkeCwgZHdvcnQgW3JzcCA
rIDEyXSArICAgiCTsgTG9hZCBkIGZyb20gW1JTUCsxMl0KCMntcCBLYXgsIGVieCAgiCAgICAgiCAgICA
gICAgiCAg0yBDb21wYXJlIGEgYW5kIGIKamdlIGdyZWF0ZXJfb3JfZXFX1YWwgICAgiCAgICAgiCA7IEp
1bXAgawYgYSA+PSB1Cm1vdiBLYXgsIGVieCAgiCAgICAgiCAgICAgiCAg0yBTZQgRUFYIHRvIGI
gaWYgYSA8IGIKCmdyZWF0ZXJfb3JfZXFX1YWw6CiAgICAgiCAgICAgiCAgICAgiCAgICAgiCAgICA
g0yBFQVggaXMgYWxyZWFkeSBhIGlmIGEgPj0gYgogICAgiApjbXAgZWN4LCBLY3gsIDEGICAgiCAgICA
7IENvbXBhcmUgYyBhbmqZAppqYiBsZXNzX3RoYW4gICAgiCAgICA7IEp1bXAgawYgYyA8IGQKbW9
2IGVieCwgZWR4ICAgiCAgICAgiCAg0yBTZQgRUJYIHRvIGQgawYgYyA+PSBkCgpsZXNzX3RoYW46CiA
gICAgiCAgbW92IGVieCwgZWN4ICAgiCAgICAgiCAg0wokdGvzdCBLY3gsIDEGICAgiCAgICAgiCAg0yB
UZXN0IGlmIGMgaXMgb2RkCmpueiBpc19vZGQgICAgiCAgICAgiCAgIDsgSnVtcCpBzjIGlzIG9kZAp
zaGwgZWN4LCAYCmpTCBkb25lCiAgICAgiCAgCmlzX29kZDoKCXNociBLY3gsIDMKZG9uZToK
```

Figure 3: condition\_Cmd.

```
— Your Flag —
Congrats! You passed all challenges! Here is your flag: FLAG{c0d1Ng_1n_a5s3mB1y
_i5_s0_fun!}
```

Figure 4: Assembly\_Flag.

## [Lab] HelloRevWorld

- FLAG{h3110\_revers1ng\_3ngineer5}

## 解題流程和思路

跟著助教上課的內容做, 大概要點就是如何將 data 弄成 Array, 調整 string literal, 完成後就可以拿到 Flag 了, Figure 5 or Figure 6

```
.rodata:0000000000002008 ; int aFlagH311oRever[32]
.rodata:0000000000002008 aFlagH311oRever:          ; DATA XREF: main+8 to
.rodata:0000000000002008             text "UTF-32LE", 'FLAG{h311O_revers1ng_3ngineer5}', 0
.rodata:0000000000002088 unk_2088      db 25h ; %           ; DATA XREF: main+F to
```

Figure 5: helloworld\_Flag.

```
{ __wprintf_chk(1LL, &unk_2088, U"FLAG{h311O_revers1ng_3ngineer5}"); return 0LL;
```

Figure 6: helloworld\_Flag-2.

## [HW2] crackme\_vectorization

```
· FLAG{yOu_kn0w_h0w_to_r3v3r53_4_m47riX!}
```

## 解題流程和思路

- 用 IDA 進行 Decompile, 並將參數進行一些命名並觀察程式邏輯, 其中我有猜測並宣告一個 struct,

```
struct struct_1{
    uint64_t member1;
    int *member2;
}
```

並且將兩個變數轉型成此 struct(由後面可以推知, member1 是 Array\_size, member2 是根據 Array\_size 建出該大小的 Array)

```
_isoc99_scanf(&unk_2004, &input_1, a3);
array_size = input_1;
sqrt_input_1 = sqrt((double)input_1);
ceil_sqrt_input_1 = (int)sqrt_input_1;
if ( sqrt_input_1 > (double)(int)sqrt_input_1 ) // we guess it may find the perfect squares
    ceil_sqrt_input_1 = (int)sqrt_input_1 + 1;
v16 = _mm_shuffle_epi32(_mm_cvtsi32_si128(ceil_sqrt_input_1), 224).m128i_u64[0];
struct1 = (struct1 *)malloc(0x10uLL);
struct1->member1 = v16;
size = 4 * ceil_sqrt_input_1 * (_int64)ceil_sqrt_input_1; // size = N*2
v7 = (int *)malloc(size);
struct1->member2 = v7;
if ( array_size > 0 )
{
    v9 = v7;
    array_index = 0LL;
    do
    {
        _isoc99_scanf(&unk_2004, input_2, v8);
        v9[array_index++] = input_2[0];
    } while ( input_1 > (int)array_index );
}
if ( dword_4038 == ceil_sqrt_input_1
    && (struct2 = (struct2 *)malloc(16uLL),
        struct2->member1 = v16,
        v12 = (int *)malloc(size),
        v13 = src,
        struct2->member2 = v12,
        memcpy(v12, v13, size),
        v14 = sub_1300(
            struct2,
            struct1), // we guess member2 of struct2 and struct1 do some caculation.
        !memcmp(*const void **)(v14 + 8), s2, size)) )
```

Figure 7: crackme\_vectorization\_pseudocodeInMain.

- 觀察上面的程式可以發現, 程式想將第一個輸入變成一個完全平方數, 第二個輸入則是輸入第一輸入的值個 element 到一個陣列中。
- struct1 中的 member1 和 struct2 的 membe1 值都一樣(為第一次輸入的值取根號後取天花板, 然後丟進函式中後得到的數值); struct1 中的 member2 為大小是"第一次輸入的值取根號後取天花板再平方(也就是得到完全方數)"記憶體空間的 Array, 其內容為第二次輸入的值せ, struct2 中的 member2 大小和 struct1 一樣, 不過它的內容是程式裡預設的。

4. 接著是本題的重點 sub\_1300() function, 如下 Figure 8, 我看不懂也不覺得要把它看懂...所以有問 StudentID:r12922146 同學, 他有給我提示“有可能 sub\_1300 是進行矩陣乘法”

```

output[1] = v10;
if ( v84 > 0 )
{
    v11 = v6;
    if ( (int)v6 > 0 )
    {
        v12 = a1[1];
        v93 = v84;
        v94 = output;
        v13 = 16 * v6;
        v85 = v10;
        v88 = v12 - 1;
        v90 = a1;
        v89 = 16LL * ((unsigned int)v12 >> 2);
        v14 = 0;
        while ( 1 )
        {
            v15 = 0LL;
            v16 = v12 * v14;
            v87 = 4LL * v12 * v14;
            v17 = v86 & 7;
            if ( (v86 & 7) == 0 )
                goto LABEL_44;
            switch ( v17 )
            {
                case 1LL:
                    goto LABEL_42;
                case 2LL:
                    goto LABEL_40;
                case 3LL:
                    goto LABEL_38;
                case 4LL:
                    goto LABEL_36;
                case 5LL:
                    goto LABEL_34;
            }
            if ( v17 != 6 )
            {
                if ( v12 > 0 )
                    goto LABEL_12;
            }
        }
    }
}

```

Figure 8: crackme\_vectorization\_sub1300.

5. 因此,首先我先去將 src(struct2->member2)和 s2 的 data 找出來, 如下 Figure 9

rodata:0000000000002018 dword_2018	dd 254, 220, 118, 156, 66, 33, 172, 39, 109, 35, 76, 147
rodata:0000000000002018	; DATA XREF: .data:src+o
rodata:0000000000002048	dd 211, 190, 239, 251, 25, 152, 220, 105, 235, 83, 244
rodata:0000000000002074	dd 101, 56, 103, 202, 151, 40, 33, 149, 21, 102, 248, 136
rodata:00000000000020A4	dd 3, 161, 147, 96, 37, 243, 152, 191, 78, 148, 65, 172
rodata:00000000000020D4	dd 95, 127
rodata:00000000000020DC	db 0
rodata:00000000000020DD	db 0
rodata:00000000000020DE	db 0
rodata:00000000000020DF	db 0
rodata:00000000000020E0 dword_20E0	dd 91221, 86974, 86901, 94427, 80815, 78329, 56322, 74715
rodata:00000000000020E0	; DATA XREF: .data:s2+o
rodata:0000000000002100	dd 67685, 65837, 71267, 65333, 56255, 51122, 117232, 101837
rodata:0000000000002120	dd 99626, 111583, 96436, 90829, 70071, 83947, 73272, 74711
rodata:0000000000002140	dd 87363, 72541, 67796, 59709, 63847, 60753, 61585, 62622
rodata:0000000000002160	dd 63208, 52609, 52454, 68693, 68521, 70379, 78011, 67711
rodata:0000000000002180	dd 58967, 55496, 81829, 71954, 72289, 78617, 76010, 67232
rodata:00000000000021A0	dd 55913

Figure 9: crackme\_vectorization\_S2andSrcData.

6. 接者我們就是要來解矩陣, 也就是我們輸入的 Array 值要為多少(第二次 input 的值)才可以滿足以下

$$\text{struct1-}>\text{member2} * \text{src} = \text{s2}$$

也就是

$$\text{struct1-}>\text{member2} = \text{src}^{-1} * \text{s2}$$

另外忘記提到 dword\_4038 的值為 7, array\_size(第一次輸入的值)就要介於 37~49 之間。那要解決此問題當然就是用 Sage(有附程式碼就不貼了), 最後執行以下指令 Figure 10 ,就可以拿到解。

```
lamb0@Ubuntu-Lamb0:~/Desktop/hw2_r12922054/Reverse0/Crackme_vectorization$ sage
--python solve.py
[[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, 88, 70, 67],
 [122, 50, 49, 105, 104, 69, 75],
 [73, 86, 107, 55, 67, 71, 116],
 [113, 122, 76, 89, 89, 61, 0]]
fgp5FdHX/7z2E1BCJxvPD5crfXFCz21ihEKIVk7CGtqzLYY=
```

Figure 10: crackme\_vectorization\_sageSolve.

## Flag Exchange

Enter RevGuard Session:

fgp5FdHX/7z2E1BCJxvPD5crfXFC

**Exchange Flag**

Information exchanged successfully Flag:

FLAG{yOu\_kn0w\_hOw\_to\_r3v3r53\_4\_m47riX!}

Figure 11: crackme\_vectorization\_Flag.

### [Lab] Clipboard Stealer 1 – sub\_140001C80

- FLAG{T1547.001}

#### 解題流程和思路

跟著助教上課的內容, 找到 Persistence – Boot or Logon Autostart Execution:Registry Run Keys / Startup Folder, 再從 ATT&CK Matrix 中查詢即可得知為 T1547.001, 如下 Figure 12

Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder

Other sub-techniques of Boot or Logon Autostart Execution (14) ▾

ID: T1547.001

Figure 12: Clipboard\_Stealer\_1\_Flag.

### [Lab] Clipboard Stealer 2 – sub\_140001030

- FLAG{T1480}

#### 解題流程和思路

跟著助教上課的內容, 找到 Execution Guardrails, 再從 ATT&CK Matrix 中查詢即可得知為 T1480, 如下 Figure 13

Execution Guardrails

Sub-techniques (1)

ID: T1480

Figure 13: Clipboard\_Stealer\_2\_Flag.

### [Lab] Clipboard Stealer 3 – sub\_140001120

- FLAG{th15\_I4\_4\_mut3x\_k1LL\_SwItcH}

## 解題流程和思路

1. 跟著助教上課所說的用 IDA 去分析, 如下 Figure 14, Name 是由 v3 和 v5 進行 XOR 得出, 將 v5 轉為 bytes, 並查詢 unk\_140003348 此位置共 28 個 byte, 將兩者 XOR 即可(程式碼有附就不貼了)

```

1 int createMutex_1120()
2 {
3     HANDLE MutexA; // rax
4     int i; // [rsp+20h] [rbp-78h]
5     int v3[2]; // [rsp+30h] [rbp-68h]
6     CHAR Name[32]; // [rsp+38h] [rbp-60h] BYREF
7     char v5[32]; // [rsp+58h] [rbp-40h] BYREF
8
9     v3[0] = 1684234874;
10    qmemcpy(v5, &unk_140003348, 0x1Dui64);
11    for ( i = 0; i < 28; ++i )
12        Name[i] = *(_BYTE *)v3 + i % 4) ^ v5[i];
13    MutexA = CreateMutexA(0i64, 1, Name);
14    if ( MutexA )
15    {
16        LODWORD(MutexA) = GetLastError();
17        if ( (_DWORD)MutexA == 183 )
18            exit(0);
19    }
20    return (int)MutexA;
21 }
```

Figure 14: Clipboard\_Stealer3\_IDAcode.

2. 執行程式即拿到 Flag

```

Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Clipboard_Stealer_3$ python Decode_Mu
tex_Name.py

```

Figure 15: Clipboard\_Stealer\_3\_Cmd.

**FLAG{th15\_I4\_4\_mut3x\_k1LL\_SwItcH}**

Figure 16: Clipboard\_Stealer\_3\_Flag.

## [Lab] Clipboard Stealer 4 – Extract Next Stage Payload

- FLAG{462fe0007f86957f59824e113f78947c}

## 解題流程和思路

跟著助教上課的內容跟著做將 pe\_data 抓出來並計算它的 md5 hash, 不過這邊要注意, Ubuntu 的 md5 指令為

md5sum

即可拿到 Flag。

```
Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Clipboard_Stealer_4$ md5sum exported_next-stage.dll
```

Figure 17: Clipboard\_Stealer\_4\_Cmd.

```
462fe0007f86957f59824e113f78947c
```

Figure 18: Clipboard\_Stealer\_4\_Flag.

## [Lab] Clipboard Stealer 5 – Exfiltrate Data

- FLAG{C2\_cU540m\_Pr0t0C01}

### 解題流程和思路

跟著助教上課的內容, 將作業中的 trace 檔丟入 wireshark 中, 由第 8 個封包得知 key, 由第 11 個封包得知 Ct, 接著到 <https://gchq.github.io/CyberChef/> 進行解密即可獲得答案, 如下 Figure 19

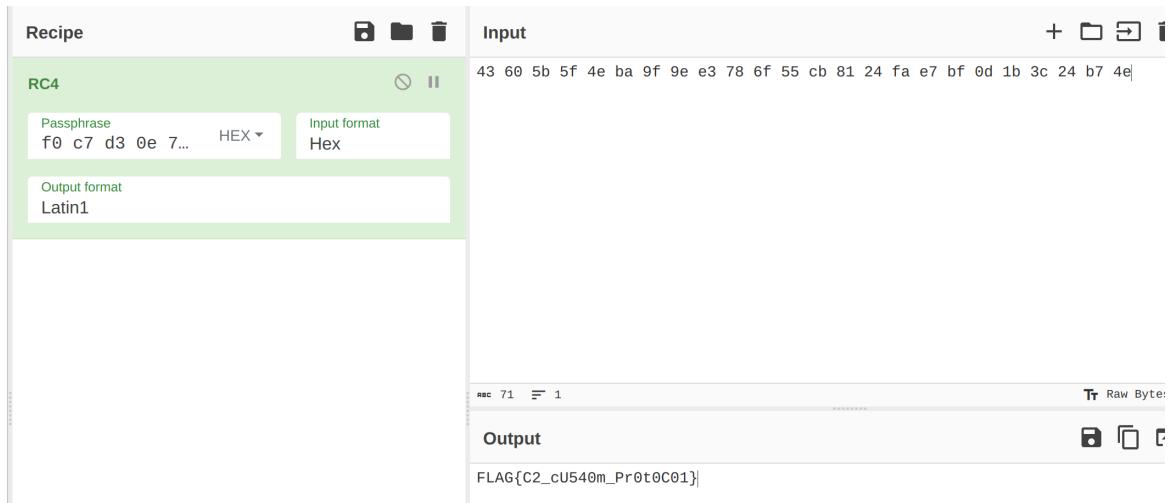


Figure 19: Clipboard\_Stealer\_5\_Flag.

## [Lab] Clipboard Stealer 6 – Dynamic API Resolution

- FLAG{MessageBoxA}

### 解題流程和思路

跟著助教上課的內容, 到 github 上抓取 user32.dll 的 API 名稱並且寫個 python 檔去找該惡意程式是想抓取哪個 API 名稱即可完成。

```
Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Clipboard_Stealer_6$ python find_user32dll_name.py
```

Figure 20: Clipboard\_Stealer\_6\_Cmd.

```
b'MessageBoxA'
```

Figure 21: Clipboard\_Stealer\_6\_Flag.

## [HW2] Banana Donut Verifier

- FLAG{d0\_Y0u\_l1k3\_b4n4Na\_d0Nut?}

## 解題流程和思路

- 首先, 先跑一次執行檔, 發現真的跟 Donut 的實作結果很像, 因此參照 <https://medium.com/analytics-vidhya/the-donuts-code-the-math-behind-6d473eaec61d>, 開始分析 Figure 22

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <unistd.h>

int main() {
    float A = 0, B = 0;
    float i, j;
    int k;
    float z[1760];
    char b[1760];
    printf("\x1b[2J");
    for(;;) {
        memset(b, 32, 1760);
        memset(z, 0, 7040);
        for(j=0; j < 6.28; j += 0.07) {
            for(i=0; i < 6.28; i += 0.02) {
                float c = sin(i);
                float d = cos(j);
                float e = sin(A);
                float f = sin(j);
                float g = cos(A);
                float h = d + 2;
                float D = 1 / (c * h * e + f * g + 5);
                float l = cos(i);
                float m = cos(B);
                float n = sin(B);
                float t = c * h * g - f * e;
                int x = 40 + 30 * D * (l * h * m - t * n);
                int y = 12 + 15 * D * (l * h * n + t * m);
                int o = x + 80 * y;
                int N = 8 * ((f * e - c * d * g) * m - c * d * e - f
                * g - l * d * n);
                if(22 > y && y > 0 && x > 0 && 80 > x && D > z[o]) {
                    z[o] = D;
                    b[o] = ".,-~:;!*$@"[N > 0 ? N : 0];
                }
            }
        }
        printf("\x1b[H");
        for(k = 0; k < 1761; k++) {
            putchar(k % 80 ? b[k] : 10);
            A += 0.00004;
            B += 0.00002;
        }
        usleep(30000);
    }
    return 0;
}
```

Figure 22: Reference of Banana Donut Verify.

- 將靜態分析的 pseudocode 進行參數命名的調整, 如下: Figure 23, Figure 24, Figure 25, 並且會要我們輸入一個記憶體空間有 1024 個 byte 大小的字串

```

48 A = 0.0;
49 B = 0.0;
50 v42 = 0LL;
51 memset(input, 0, 1024);
52 printf("Dount Verifier\nInput: ");
53 _isoc99_scanf("%1023s", input);
54 printf("\x1B[2D");
55 for ( i = 0LL; i <= 499; ++i ) int64 input[129]; // [rsp+2260h] [rbp-480h] BYREF
56 {
57     memset(b, 32, sizeof(b));
58     memset(z, 0, sizeof(z));
59     for ( theta = 0.0; theta < 6.28; theta = next_theta )
60     {
61         iter_count = 0;
62         for ( phi = 0.0; phi < 6.28; phi = next_phi )
63         {
64             tmp_c = _mm_cvtsi32_si128(LDWORD(phi));
65             *tmp_c.m128i_i32 = sub_19EA(*tmp_c.m128i_i32);
66             c = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_c));
67             tmp_d = _mm_cvtsi32_si128(LDWORD(theta));
68             *tmp_d.m128i_i32 = sub_19CF(*tmp_d.m128i_i32);
69             d = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_d));
70             tmp_e = _mm_cvtsi32_si128(LDWORD(A));
71             *tmp_e.m128i_i32 = sub_19EA(*tmp_e.m128i_i32);
72             e = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_e));
73             tmp_f = _mm_cvtsi32_si128(LDWORD(theta));
74             *tmp_f.m128i_i32 = sub_19EA(*tmp_f.m128i_i32);
75             f = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_f));
76             tmp_g = _mm_cvtsi32_si128(LDWORD(A));
77             *tmp_g.m128i_i32 = sub_19CF(*tmp_g.m128i_i32);
78             g = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_g));
79             h = d + 2.0;
80             D = 1.0 / (((c * (d + 2.0)) * e) + (f * g)) + 5.0;
81             tmp_l = _mm_cvtsi32_si128(LDWORD(phi));
82             *tmp_l.m128i_i32 = sub_19CF(*tmp_l.m128i_i32);
83             l = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_l));
84             tmp_m = _mm_cvtsi32_si128(LDWORD(B));
85             *tmp_m.m128i_i32 = sub_19CF(*tmp_m.m128i_i32);

```

Figure 23: Banana\_Donut\_Verify\_IDAPseudocode1.

```

86     m = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_m));
87     tmp_n = _mm_cvtsi32_si128(LDWORD(B));
88     *tmp_n.m128i_i32 = sub_19EA(*tmp_n.m128i_i32);
89     n = COERCE_FLOAT(_mm_cvtsi32_si32(tmp_n));
90     t = ((c * h) * g) - (f * e);
91     x = (((D * 30.0) * ((l * h) * m) - (t * n)) + 40.0);
92     y = (((D * 15.0) * ((t * m) + ((l * h) * n))) + 12.0);
93     o = 80 * y + x;
94     N = (8.0 * (((f * e) - ((c * d) * g)) * m) - ((c * d) * e)) - (f * g)) - ((l * d) * n));
95     if ( y <= 21 && y > 0 && x > 0 && x <= 79 && D > z[o] )
96     {
97         z[o] = D;
98         v11 = N;
99         if ( N < 0 )
100             v11 = 0;
101         b[o] = asc_3F52[v11];
102     }
103     if ( iter_count == 30 && v42 <= 1023 ) // ??? very strange
104     {
105         v22 = o ^ N ^ (x + y); unsigned op; bool
106         variable_alwaysEq0 = v42++;
107         *(input + variable_alwaysEq0) ^= o ^ N ^ (x + y);
108     }
109     ++iter_count;
110     next_phi = phi + 0.02;
111 }
112 next_theta = theta + 0.07000000000000001;
113 }
114 printf("\x1B[H");
115 for ( index = 0; index <= 1760; ++index )
116 {
117     if ( index % 80 )
118         v15 = b[index];
119     else
120         v15 = 10;
121     putchar(v15);
122     v16 = A + 0.00004;

```

Figure 24: Banana\_Donut\_Verify\_IDAPseudocode2.

```

133 v39 = encryption((__int64)input, 1024uLL);
134 v38 = encryption((__int64)off_6050, 1024uLL);
135 if ( v39 == v38 )
136     puts("Donut likes your input!! :D");
137 else
138     puts("Donut Reject You!! :(");
139 puts("No matter donut accept you or not. Here's a bananacat for you");

```

Figure 25: Banana\_Donut\_Verify\_IDAPseudocode3.

3. 可以發現在 Figure 24 中有多出一個奇怪的算式, 邏輯大概是執行到第 30 個迴圈的時候會執行 if 裡面的內容(v42永遠都是 0 沒有動), if 裡的內容是, input 的第一個字元和 donut 中某些參數做 XOR。
4. 最後要能成功的依據是, input 的第一個字元和 donut 中某些參數做 XOR 的新 input(New\_input)丟進 encryption function 後回傳的結果 v39 要等於系統設定的某個值(Compared value)丟進 encryption function 後回傳的結果 v38。
5. 但可以發現其實如果 New\_input = Compared value, 那麼 v38 = v39。
6. 因此我們要解的問題就變成以下:

First element of input  $\oplus$  Some of Donut parameters = Compared value

, where

New\_input = First element of input  $\oplus$  Some of Donut parameters

7. 根據上式我們可以先透過動態分析找出 Compared value 和 New\_input 的值, 只要找到執行 encryption function 前, Compared value 和 New\_input 存放在哪個暫存器並且把他印出來就好。至於 encryption function 的虛擬記憶體位置如下, Figure 26 為(0x1A05)

```
ext.0000000000001A05 ; ===== S U B R O U T I N E =====
ext:0000000000001A05 ; Attributes: bp-based frame
ext:0000000000001A05
ext:0000000000001A05 ; _int64 __fastcall encryption(__int64 input_array, unsigned __int64 num_1024)
ext:0000000000001A05 encryption proc near ; CODE XREF: main+526↑p
ext:0000000000001A05 ; main+53D↑p
ext:0000000000001A05
ext:0000000000001A05 var_30      = qword ptr -30h
ext:0000000000001A05 var_28      = qword ptr -28h
ext:0000000000001A05 var_11      = byte ptr -11h
ext:0000000000001A05 var_10      = qword ptr -10h
ext:0000000000001A05 var_4       = dword ptr -4
ext:0000000000001A05
ext:0000000000001A05 ; __unwind {
ext:0000000000001A05     push    rbp
ext:0000000000001A05     mov     rbp, rsp
ext:0000000000001A06
```

Figure 26: Banana\_Donut\_Verify\_GetFunctionAddr.

8. 開始使用動態分析, 依序執行以下指令即可

```
gdb ./donut-verilfier
```

```
starti
```

```
b *$rebase(0x1A05)
```

```
c
```

```
0
```

```
x/1024b $rdi
```

此時就可以拿到New\_input的值

```
pwndbg> x/1024b $rdi
0x7fffffff880: 13      58      -117     -118     -118     -118     -119     -119
0x7fffffff888: -120    -120    -120     89      86      84      84
0x7fffffff890: 84      86      89      89      39      86      86      87
0x7fffffff898: 75      75      72      72      73      73      78      78
0x7fffffff8a0: 75      -7      -7      -8      7       7       7       5
0x7fffffff8a8: 4       7       7       6       1       1       2       2
0x7fffffff8b0: 1       1       0       -79     -75     -70     -70     -70
0x7fffffff8b8: -70     -70     -89     -89     -88     -88     -88     -85
0x7fffffff8c0: -85     -87     120     120     119     119     119     118
0x7fffffff8c8: 118    117     -60     -53     -52     -52     -51     -51
0x7fffffff8d0: -50     -49     -49     -49     48      48      49      49
0x7fffffff8d8: 50      51      58      56      57      -118    -118    -119
0x7fffffff8e0: -119    -119    -120    -120    -120    -120    89      91
0x7fffffff8e8: 91      91      89      89      89      39      39      87
```

Figure 27: Banana\_Donut\_Verify\_v39\_inputArray.

```
0
```

```
x/1024b $rdi
```

此時就可以拿到Compared value的值

Address	Value							
0x555555556010	71	86	-8	-66	-3	-5	-90	-5
0x555555556018	-89	-1	-14	-14	12	99	51	17
0x555555556020	101	47	24	33	105	99	53	37
0x555555556028	45	19	14	11	51	123	127	39
0x555555556030	35	-68	-78	-79	81	108	48	70
0x555555556038	67	115	118	124	77	88	91	63
0x555555556040	1	1	0	-79	-75	-70	-70	-70
0x555555556048	-70	-70	-89	-89	-88	-88	-88	-85
0x555555556050	-85	-87	120	120	119	119	119	118
0x555555556058	118	117	-60	-53	-52	-52	-51	-51
0x555555556060	-50	-49	-49	-49	48	48	49	49
0x555555556068	50	51	58	56	57	-118	-118	-119
0x555555556070	-119	-119	-120	-120	-120	-120	89	91
0x555555556078	91	91	89	89	89	39	39	87

Figure 28: Banana\_Donut\_Verify\_v38\_inputArray.

9. 然後會發現 New\_input 和 Compared value 的值只有前面 48byte 不一樣所以我們可以只針對這 48bytes 找出可以通過驗證的 Input , 如下: Figure 29

```
1  from sage.all import *
2  from Crypto.Util.number import *
3
4  v39_input = [ 13, 58, -117, -118, -118, -118, -119, -119,
5  |           -120, -120, -120, -120, 89, 86, 84, 84,
6  |           84, 86, 89, 89, 39, 86, 86, 87,
7  |           75, 75, 72, 72, 73, 73, 78, 78,
8  |           75, -7, -7, -8, 7, 7, 7, 5,
9  |           4, 7, 7, 6, 1, 1, 2, 2,
10 |          ]
11
12 v38_input = [ 71, 86, -8, -66, -3, -5, -90, -5,
13 |           -89, -1, -14, -14, 12, 99, 51, 17,
14 |           101, 47, 24, 33, 105, 99, 53, 37,
15 |           45, 19, 14, 11, 51, 123, 127, 39,
16 |           35, -68, -78, -79, 81, 108, 48, 70,
17 |           67, 115, 118, 124, 77, 88, 91, 63,
18 |          ]
19
20 v39_input[0] ^= 48 # 0 ascii
21
22
23 input_string = ''
24 for i in range(48):
25     input_string += (v39_input[i] ^ v38_input[i]).to_bytes(1, 'big').decode()
26
27 print(input_string)
```

Figure 29: Banana\_Donut\_Verify\_SolevCode.

10. 執行以下指令拿到我們要的 Input: Figure 30

```
Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Reverse0/banana_donuts$ sage --python
solve.py
zls4wq/r/wzzU5gE1yAxN5crfXFCz21ihEKIVk7CGtqzLY=
```

Figure 30: Banana\_Donut\_Verify\_Cmd.

11. 獲得 Flag

# Flag Exchange

Enter RevGuard Session:

```
zls4wq/r/wzzU5gE1yAxN5crfXFCi
```

Exchange Flag

Information exchanged successfully Flag:

```
FLAG{d0_Y0u_l1k3_b4n4Na_d0Nut?}
```

Figure 31: Banana\_Donut\_Verify\_Flag.

## [Lab] Super Angry

- FLAG{knowing\_how\_2\_angr!}

### 解題流程和思路

本題跟著助教上課的內容做, 最主要是要讓我們熟悉如何使用 angr 解題, py 檔有附, 以下為執行的 Command, Figure 32, 以及獲得的 Flag, Figure 33

```
Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Reverse2/super_angry/dist$ python super_angry_solve.py
```

Figure 32: super\_angry\_Cmd.

```
b'FLAG{knowing_how_2_angr!}\x00DBUS_S'
```

Figure 33: super\_angry\_Flag.

## [Lab] Scramble

- FLAG{scramble\_and\_using\_solver\_to\_solve\_it}

### 解題流程和思路

本題跟著助教上課的內容做, 最主要是要讓我們熟悉如何使用 z3 解題, py 檔有附, 以下為執行的 Command, Figure 34, 以及獲得的 Flag, Figure 35

```
Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Reverse2/scramble/dist$ python scramble_solve.py
```

Figure 34: scramble\_Cmd.

```
FLAG{scramble_and_using_solver_to_solve_it}
```

Figure 35: scramble\_Flag.

## [Lab] Unpackme

- FLAG{just\_4\_simple\_unpackme\_challenge!}

### 解題流程和思路

- 因為 unpackme 的執行檔有權限的問題, 所以不能用 gdb 分析, 因此先用以下指令, 解決權限問題。

```
chmod -x unpackme
```

2. 用 gdb 開始分析, 找到 upx 進入 memory 位置, 並把它 dump 出來並到 IDA 觀察, Figure 36 Figure 37

```
pwndbg> vmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      Start           End Perm   Size Offset File
0x7ffff7ff2000 0x7ffff7ff6000 r--p 4000 0 [vvar]
0x7ffff7ff6000 0x7ffff7ff8000 r--p 2000 0 [vdso]
0x7ffff7ff8000 0x7ffff7ff9000 rw-p 1000 0 [/home/lambo/Desktop/hw2_r12922054/Reverse2/unpackme/dist/unpackme]
0x7ffff7ff9000 0x7ffff7ffd000 rw-p 4000 0 [anon_7ffff7ff9]
0x7ffff7ffd000 0x7ffff7fff000 r--p 2000 0 [/home/lambo/Desktop/hw2_r12922054/Reverse2/unpackme/dist/unpackme]
0x7ffff7ffd000 0xfffffff7fde000 0xfffffff7f600000 0xf1fffff7f601000 --xp 1000 0 [vsyscall]
pwndbg> b *0x7ffff7ff8000+0x5A5F
Breakpoint 1 at 0x7ffff7ff8005
pwndbg> c
Continuing.
```

Figure 36: unpackme\_gdb\_info1.

```
pwndbg> vmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      Start           End Perm   Size Offset File
0x7ffff7ff1000 0x7ffff7ff2000 r--p 1000 0 [anon_7ffff7ff1]
0x7ffff7ff2000 0x7ffff7ff6000 r--p 4000 0 [vvar]
0x7ffff7ff6000 0x7ffff7ff8000 r--p 2000 0 [vdso]
0x7ffff7ff8000 0x7ffff7ff9000 rw-p 1000 0 [/home/lambo/Desktop/hw2_r12922054/Reverse2/unpackme/dist/unpackme]
0x7ffff7ff9000 0x7ffff7ffd000 rw-p 4000 0 [anon_7ffff7ff9]
0x7ffff7ffd000 0x7ffff7fff000 r--p 2000 0 [/home/lambo/Desktop/hw2_r12922054/Reverse2/unpackme/dist/unpackme]
0x7ffff7ffd000 0xfffffff7fde000 0xfffffff7f600000 0xf1fffff7f601000 --xp 1000 0 [vsyscall]
pwndbg> dump binary memory dump.bin 0x7ffff7ff1000 0x7ffff7ff2000
pwndbg>
```

Figure 37: unpackme\_gdb\_info2.

3. 從第二步觀察到 upx 完之後, 會用 jump 指令跳到 OEP 的 gate, 此時回到 gdb 把剛剛設置的 break point 先 disable, 然後 break point 加入 upx 完後, 即將執行 jump 跳到 OEP 的 gate 的前一個位址, 最後再 dump 原先程式執行內容的記憶體位置, Figure 38

```
pwndbg> vmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      Start           End Perm   Size Offset File
0x7ffff7fb5000 0x7ffff7fb7000 r--p 2000 0 [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2]
0x7ffff7fb5000 0x7ffff7fc1000 r--p 2000 0 [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2]
0x7ffff7fe1000 0x7ffff7fec000 r--p b000 2c000 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fec000 0x7ffff7fed000 r--p 1000 0 [anon_7ffff7fec]
0x7ffff7fed000 0x7ffff7ff1000 rw-p 4000 37000 [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7ff1000 0x7ffff7ff2000 r--p 1000 0 [anon_7ffff7ff1]
0x7ffff7ff2000 0x7ffff7ff6000 r--p 4000 0 [vvar]
0x7ffff7ff6000 0x7ffff7ff8000 r--p 2000 0 [vdso]
0x7ffff7ff8000 0x7ffff7ff9000 r--p 1000 0 [anon_7ffff7ff8]
0x7ffff7ff9000 0x7ffff7ffa000 r--p 1000 0 [anon_7ffff7ff9]
0x7ffff7ffa000 0x7ffff7ffbb000 r--p 1000 0 [anon_7ffff7ffa]
0x7ffff7ffbb000 0x7ffff7ffd000 rw-p 2000 0 [/home/lambo/Desktop/hw2_r12922054/Reverse2/unpackme/dist/unpackme
0x7ffff7ffd000 0x7ffff7fff000 r--p 1000 0x7ffff7ff600000 0xf1fffff7f601000 --xp 1000 0 [vsyscall]
0x7ffff7ffd000 0xfffffff7fde000 0xfffffff7f600000 0xf1fffff7f601000 --xp 1000 0 [vsyscall]
pwndbg> vmap 0x7ffff7fd290
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
      Start           End Perm   Size Offset File
0x7ffff7fb5000 0x7ffff7fb7000 r--p 2000 0 [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
0x7ffff7fb5000 0x7ffff7fd290 r--p 2000 0 [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 +0xe290
0x7ffff7fe1000 0x7ffff7fec000 r--p b000 2c000 [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 +0xe290
pwndbg> dump binary memory glued.bin 0x7ffff7ff8000 0x7ffff7ffd000
```

Figure 38: unpackme\_gdb\_info3.

4. 最後回到 IDA 觀察 glued.bin 可以發現 Enter\_key 在以下地方(我有把它整理過), Figure 39

```
LOAD:00000000000000000002000 ; =====
LOAD:00000000000000000002000 ; Segment type: Pure data
LOAD:00000000000000000002000 ; Segment permissions: Read
LOAD:00000000000000000002000 LOAD segment mempage public 'DATA' use64
LOAD:00000000000000000002000 assume cs:LOAD
LOAD:00000000000000000002000 :org 200h
LOAD:00000000000000000002000 a92323441JJJustA db 1,0,2,0,0,0,0,0,'.923$','0Bh','*',18h,11h,'e^$',0,1Dh,'2',11h,'3',0Eh
LOAD:00000000000000000002000 ; DATA XREF: LOAD:0000000000000130+o
LOAD:0000000000000000000201A db '0Ch,4,5,12h,17h,'4','0Ch,'*:4',6,11h,0Bh,19h,1Fh,11h,'1',6,'J',18h
LOAD:0000000000000000000202F db '0',just_a_key',0
LOAD:0000000000000000000203B aEnterInput db 'Enter input:',0 ; DATA XREF: sub_11C9+1B+o
LOAD:00000000000000000002049 as db 'ts',0 ; DATA XREF: sub_11C9+36+o
LOAD:00000000000000000002049 aIncorrect db 'Incorrect!',0 ; sub_11C9+1F+o
LOAD:0000000000000000000204C aIncorrect db 'Incorrect!',0 ; DATA XREF: sub_11C9+59+o
```

Figure 39: unpackme\_gluedBin\_IDAinfo.

5. 執行 unpackme, 並獲得 Flag

```
Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Reverse2/unpackme/dist$ ./unpackme
Enter input: just_a_key
```

Figure 40: unpackme\_Cmd.

```
:FLAG{just_4_simple_unpackme_challenge!}
```

Figure 41: unpackme\_Flag.

[HW2] Baby Ransom 1 – Next Stage Payload

```
· FLAG{e6b77096375bcff4c8bc765e599fb0c0}
```

## 解題流程和思路

- 首先在程式開始的地方, 到第 76 行的函式 InternetOpenURL1DBB(), 並進入, 前面的部份大概就是再做有關 critcl section 的部份, 看起來沒有什麼可以地方。如圖: Figure 42

```
● 75  _initenv = mainArgs3;
● 76  dword_140017010 = InternetOpenUrlA1DBB((unsigned int)mainArgs1, mainArgs2);
```

Figure 42: BabyRansom1\_info1.

- 進入之後, 我們往 createFolder\_1B0A 前進。

```
1 |__int64 InternetOpenUrlA1DBB()
2 {
3     void *hInternet; // [rsp+38h] [rbp-18h]
4
5     sub_140001F1E();
6     hInternet = InternetOpenA(0i64, 1u, 0i64, 0i64, 0);
7     if ( InternetOpenUrlA(hInternet, URL, 0i64, 0, 0x84000000, 0i64) )// dwFlags -> SECURE
8         return 0i64;
9     else
10        return createFolder_1B0A();
11 }
```

Figure 43: BabyRansom1\_info2.

- 進入之後, 大概可以知道這個部份在對資料夾裡的檔案做一些設定, 並沒有什麼可疑的地方, 於是繼續前往第 61 行的 gotoInject197A(), 如圖 : Figure 44

```
59         free(lpWideCharStr);
60         free(lpFilename);
61         gotoInject197A();
```

Figure 44: BabyRansom1\_info3.

- 戳進去最裡面後會看到如圖 : Figure 45, 但沒啥東西, 繼續前往 CreateProcess1653()

```
1 __int64 sub_1400018DA()
2 {
3     if ( (unsigned int)off_140007088()
4         && *(__QWORD *)(&qword_140017030 + 8)
5         && !strcmp((const char *)(*(__QWORD *)(&qword_140017030 + 8) + 9i64), "start!!"))
6     {
7         return 0i64;
8     }
9     else
10    {
11        return CreateProcess1653();
12    }
13 }
```

Figure 45: BabyRansom1\_info4.

- 進入之後, 我們 Figure 46 和 Figure 47 一起看, 在分析的當下我進入 doPE\_Encryption155A()後 (Figure 47), 在看到 FindResourceA、LoadResourceA、SizeofResourceA 的函式後我就覺得此部份可能是進行 next\_stage\_payload 的地方, 找特定的資源又要拿他的 size, 因此我就將 Figure 46 和 Figure 47 的參數稍微重新命名。

```

1 __int64 CreateProcess1653()
2 {
3     __int64 sizeOfsomething; // [rsp+58h] [rbp-28h] BYREF
4     __int64 ProcessBaseAddress; // [rsp+60h] [rbp-20h] BYREF
5     int v3; // [rsp+6Ch] [rbp-14h]
6     struct _PROCESS_INFORMATION ProcessInformation; // [rsp+70h] [rbp-10h] BYREF
7     struct _STARTUPINFOA StartupInfo; // [rsp+90h] [rbp+10h] BYREF
8     DWORD sizeOf_pe; // [rsp+104h] [rbp+84h] BYREF
9     int *firstByteOf_pe; // [rsp+108h] [rbp+88h] BYREF
10    LPVOID lpContext_baseAddr; // [rsp+110h] [rbp+90h]
11    int j; // [rsp+118h] [rbp+98h]
12    int i; // [rsp+11Ch] [rbp+9Ch]
13
14    doPE_Encryption((LPVOID *)&firstByteOf_pe, &sizeOf_pe);

```

Figure 46: BabyRansom1\_info5.

```

1 HRSRC __fastcall doPE_Encryption155A(LPVOID *ptr_FirstByteofResource, DWORD *sizeOfResource)
2 {
3     HRSRC handle_for_loadingResource; // rax
4     HRSRC handleResourceInfo; // [rsp+30h] [rbp-10h]
5     unsigned int i; // [rsp+3Ch] [rbp-4h]
6
7     handle_for_loadingResource = FindResourceA(0i64, (LPCSTR)0x44, (LPCSTR)0x84);
8     handleResourceInfo = handle_for_loadingResource;
9     if ( handle_for_loadingResource )
10    {
11        handle_for_loadingResource = (HRSRC)LoadResource(0i64, handle_for_loadingResource);
12        if ( handle_for_loadingResource )
13        {
14            *ptr_FirstByteofResource = LockResource(handle_for_loadingResource);
15            handle_for_loadingResource = (HRSRC)*ptr_FirstByteofResource;
16            if ( *ptr_FirstByteofResource )
17            {
18                *sizeOfResource = SizeofResource(0i64, handleResourceInfo);
19                handle_for_loadingResource = (HRSRC)*sizeOfResource;
20                if ( (_DWORD)handle_for_loadingResource )
21                {
22                    for ( i = 0; ; i += 2 )
23                    {
24                        handle_for_loadingResource = (HRSRC)*sizeOfResource;
25                        if ( i >= (unsigned int)handle_for_loadingResource )
26                            break;
27                        *(_WORD *)((char *)*ptr_FirstByteofResource + (int)i) ^= 0x8711u;
28                    }
29                }
30            }
31        }
32    }
33    return handle_for_loadingResource;
34 }

```

Figure 47: BabyRansom1\_info6.

- 接著,我們就要去動態分析一下這個部份到底是不是抓了我們要的東西,因此我們先同步 deassembly code 和 decompiled code( Figure 48 和 Figure 49 ),我們將斷點設在 1674 的下一個位址 1679(因為總得讓函式先找完資源嘛),並查看 rcx -> pe 的起始位址,和 rdx -> pe 的 size

```

14 doPE_Encryption155A((LPVOID *)&firstByteOf_pe, &sizeOf_pe);

```

Figure 48: BabyRansom1\_info7.

.text:0000000140001663	lea rdx, [rbp+0A0h+sizeOf_pe]
.text:000000014000166A	lea rax, [rbp+0A0h+firstByteOf_pe]
.text:0000000140001671	mov rcx, rax
.text:0000000140001674	call doPE_Encryption155A
.text:0000000140001679	lea rax, [rbp+0A0h+StartupInfo]

Figure 49: BabyRansom1\_info8.

- 如圖 Figure 50,我們看到 rcx = 00007FF65D17B058, rdx = 1CA00,接著就是把它 dump 出來。

RAX	0000000000001CA00	baby-rans
RBX	00000000000000001	
RCX	00007FF65D17B058	
RDX	0000000000001CA00	
RBP	000000A6D5BFFC40	
RSP	000000A6D5BFFBC0	
RSI	0000022DF6E517E3	
RDI	0000022DF6E51893	
R8	0000000000000080	
R9	000000A6D5BFFA20	&"PE"
R10	00007FF65D160080	"PE"

Figure 50: BabyRansom1\_info9.

- 把我們要的區段 dump 出來後, 並進行 MD5, 結果真的拿到我們的 Flag 了( Figure 51), 代表我們沒有判斷錯誤 !

```
Lambo@Ubuntu-Lambo:~/VirtualBox VMs/Windows 10/ShareFolder/BabyRansom$ md5sum MEM_00007FF65D17B058_0001CA00.mem
e6b77096375bcff4c8bc765e599fbbcc0  MEM_00007FF65D17B058_0001CA00.mem
```

Figure 51: BabyRansom1\_Flag.

## [HW2] Baby Ransom 2 – Encrypted File

- FLAG{50\_y0u\_p4y\_7h3\_r4n50m?!hmmmmm}

### 解題流程和思路

前言：本題和 Student ID:R12922146 同學 各自通靈嘗試後彼此互相分享交流後, 完成。

- Figure 52 是程式起始的位置

```
1 int64 start()
2 {
3     sub_140003718();
4     return __scrt_common_main_seh();
5 }
```

Figure 52: BabyRansom2\_info1.

- 接著直接進到 WinMain 裡, 如下: Figure 53

```

1 int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPST
2 {
3     HWND hWnd; // [rsp+60h] [rbp-A8h]
4     WNDCLASSW WndClass; // [rsp+70h] [rbp-98h] BYREF
5     struct tagMSG Msg; // [rsp+C0h] [rbp-48h] BYREF
6
● 7     memset(&WndClass, 0, sizeof(WndClass));
● 8     WndClass.lpfnWndProc = store_winword;
● 9     WndClass.hInstance = hInstance;
● 10    WndClass.lpszClassName = Caption;
● 11    WndClass.hbrBackground = CreateSolidBrush(0);
● 12    if ( !RegisterClassW(&WndClass) )
● 13        return 1;
● 14    hWnd = CreateWindowExW(0, Caption, Caption, 0x00000000, 100, 100, 800,
● 15    if ( !hWnd )
● 16        return 2;
● 17    Run2870();
● 18    ShowWindow(hWnd, nShowCmd);
● 19    memset(&Msg, 0, sizeof(Msg));
● 20    while ( GetMessageW(&Msg, 0i64, 0, 0) )
● 21    {
● 22        TranslateMessage(&Msg);
● 23        DispatchMessageW(&Msg);
● 24    }
● 25    return 0;
● 26 }

```

Figure 53: BabyRansom2\_info2.

3. 再來, 進到 Run2870 函式裡, 如下 : Figure 54, 裡面的函式主要有兩個一個是 SearchAPI1B10(), 裡面主要是實作透過 CRC 找尋對應.dll 的 API; 至於 BackUpFile\_EncryptFile1660() 則是本次作業的重點 ! 也就是惡意程式執行的地方 !

```

1 __int64 Run2870()
2 {
3     __int64 result; // rax
4
● 5     LoadLibraryA(LibFileName);
● 6     LoadLibraryA(aWininetDll);
● 7     result = SearchAPI1B10();
● 8     if ( !result )
● 9         return BackUpFile_EncryptFile1660();
● 10    return result;
● 11 }

```

Figure 54: BabyRansom2\_info3.

4. 進到 BackUpFile\_EncryptFile1660() 裡後, 實際一開始很多位定義的變數和函式, 但透過 aSystemfunction 和 aAdvapi32 上網搜尋, 並且比對之後慢慢重新宣告變數成如 Figure 55 的樣子 , ref: <https://osandamalith.com/2022/11/10/encrypting-shellcode-using-systemfunction032-033/> 和 <https://www.cnblogs.com/fdxsec/p/17827348.html>

```

1 __int64 BackUpFile_EncryptFile1660()
2 {
3     __int64 result; // rax
4     HMODULE Library; // rax
5     __int64 v2; // [rsp+20h] [rbp-188h]
6     __int64 Buffer; // [rsp+30h] [rbp-178h] BYREF
7     unsigned int dwSize; // [rsp+38h] [rbp-170h] BYREF
8     char v5[44]; // [rsp+40h] [rbp-168h] BYREF
9     char filename[276]; // [rsp+6Ch] [rbp-13Ch] BYREF
10    char Microsoft_text[24]; // [rsp+180h] [rbp-28h] BYREF
11
12    strcpy(Microsoft_text, "Microsoft Update Backup");
13    if ( qword_140007AA8(Microsoft_text, 0i64) || (result = qword_140007AA0(), result == 0xB7) )
14    {
15        result = File_transportThroughHttp28B0();
16        if ( !result )
17        {
18            Library = LoadLibrary(aAdvapi32);
19            SystemFunction033 = GetProcAddress(Library, aSystemfunction);
20            v2 = qword_140007AB0(asc_1400072E4, v5);
21            if ( v2 == -1 )
22            {
23                return 1i64;
24            }
25            else
26            {
27                do
28                {
29                    Buffer = 0i64;
30                    if ( (v5[0] & 16) == 0 && !File_transportThroughFileRead1000(filename, &Buffer, &dwSize) )
31                    {
32                        if ( BackupFile(filename, Microsoft_text) )
33                            encryptFile1960(filename, Buffer, dwSize);
34                    }
35                }
36            while ( qword_140007AB8(v2, v5) );

```

Figure 55: BabyRansom2\_info4.

- 接著,就是重頭戲了! Figure 56 開始加密我們的檔案,在第 29 行的部份 SystemFunction033(),但其實這個 function 的 Input 參數是(struct ustring\* data, struct ustring\* key) 且 struct ustring 的參數有(Length, MaxLength, data),顯然 IDA 在 Decompile 上出了一點錯誤。不過這一點錯誤不打緊,要緊的是我們還是要趕快找出我們的 key 和 key 的 length 不然解不了。key 的 length 實際上就是經過第 14-25 行來的(邏輯大概是:如果加密的檔名字數小於 19, 則 key 的 length 就等於檔名的字數),至於 key 在哪裡呢通靈,沒錯就是通靈,看看那個第 13 行,很可疑對吧。下一步告訴你它真的很可疑。

```

1 __int64 __fastcall encryptFile1960(const char *file_name, __int64 Buffer, unsigned int dwSize)
2 {
3     unsigned __int64 v4; // [rsp+40h] [rbp-178h]
4     __int64 v5; // [rsp+48h] [rbp-170h]
5     HANDLE hFileNew; // [rsp+50h] [rbp-168h]
6     unsigned int data_size; // [rsp+68h] [rbp-150h] BYREF
7     __int64 v8; // [rsp+70h] [rbp-148h]
8     int key_size; // [rsp+78h] [rbp-140h] BYREF
9     __int64 key; // [rsp+80h] [rbp-138h]
10    char v11[8]; // [rsp+88h] [rbp-130h] BYREF
11    CHAR filename[272]; // [rsp+90h] [rbp-128h] BYREF
12
13    key = qword_140007460;
14    key_size = 19;
15    v4 = -1i64;
16    do
17        ++v4;
18        while ( file_name[v4] );
19        if ( v4 <= 19 )
20        {
21            v5 = -1i64;
22            do
23                ++v5;
24                while ( file_name[v5] );
25            key_size = v5;
26        }
27    v8 = Buffer;
28    data_size = dwSize;
29    SystemFunction033(&data_size, &key_size);
30    create_file(filename, 260i64, "enc_%s", file_name);

```

Figure 56: BabyRansom2\_info5.

- 首先我們在 Figure 56 的第 13 行對應的 Memory address,如 Figure 57,並在 x64dbg 上插入斷點並查看 rax 的值,如 Figure 58,看看它真的很可疑,於是我們拿前面的 8 個字元當作 key (因為 flag.txt 總共 8 個字元)

```

.text:0000000140001960 hFileNew      = qword ptr -168h
.text:0000000140001960 var_160     = dword ptr -160h
.text:0000000140001960 var_158     = dword ptr -158h
.text:0000000140001960 data_size    = dword ptr -150h
.text:0000000140001960 var_148     = dword ptr -148h
.text:0000000140001960 key_size     = dword ptr -140h
.text:0000000140001960 key         = dword ptr -138h
.text:0000000140001960 var_130     = byte ptr -130h
.text:0000000140001960 filename    = byte ptr -128h
.text:0000000140001960 var_18       = dword ptr -120h
.text:0000000140001960 arg_0       = dword ptr 8
.text:0000000140001960 arg_8       = dword ptr 10h
.text:0000000140001960 arg_10      = dword ptr 18h
.text:0000000140001960 ; __unwind { // _GSHandlerCheck
.text:0000000140001960     mov    [rsp+arg_10], r8d
.text:0000000140001965     mov    [rsp+arg_8], rdx
.text:000000014000196A     mov    [rsp+arg_0], rcx
.text:000000014000196F     sub    rsp, 160h
.text:0000000140001976     mov    rax, ss:_security_coc
.text:000000014000197D     xor    rax, rax
.text:0000000140001980     mov    [rsp+180h+var_18], rax
.text:0000000140001985     mov    rax, cs:qword_14000746
.text:000000014000198E     mov    [rsp+180h+key], rax
.text:0000000140001997     mov    [rsp+188h+key_size], 1
.text:000000014000199F     mov    rax, [rsp+188h+arg_0]
.text:00000001400019A7     mov    [rsp+188h+var_160], ra
.text:00000001400019AC     mov    [rsp+188h+var_178], OF
.text:00000001400019B5     ; CODE 30
.text:00000001400019B5 loc_1400019B5: ; CODE 30
                                _int64 __fastcall encrypted_1960(const char *file_name, _int64 Buffer
                                1 {
                                2     unsigned __int64 v4; // [rsp+40h] [rbp-178h]
                                3     __int64 v5; // [rsp+48h] [rbp-170h]
                                4     HANDLE hFileNew; // [rsp+50h] [rbp-168h]
                                5     unsigned int data_size; // [rsp+68h] [rbp-150h] BYREF
                                6     __int64 v8; // [rsp+70h] [rbp-148h]
                                7     int key_size; // [rsp+78h] [rbp-140h] BYREF
                                8     __int64 key; // [rsp+80h] [rbp-138h]
                                9     char v11[8]; // [rsp+88h] [rbp-130h] BYREF
                                10    CHAR filename[272]; // [rsp+90h] [rbp-128h] BYREF
                                11
                                12    key = qword_140007460;
                                13    key_size = 19;
                                14    v4 = -1i64;
                                15    do
                                16        ++v4;
                                17        while ( file_name[v4] );
                                18        if ( v4 <= 19 )
                                19            if ( file_name[v4] == '\0' )
                                20                break;
                                21    v5 = -1i64;
                                22    do
                                23        ++v5;
                                24        while ( file_name[v5] );
                                25        key_size = v5;
                                26    }
                                27    v8 = Buffer;
                                28    data_size = dwSize;
                                29    SystemFunction033(&data_size, &key_size);
                                30    create_file(filename, 260i64, "enc_%s", file_name);

```

Figure 57: BabyRansom2\_info6.

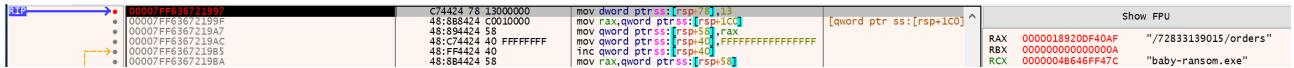


Figure 58: BabyRansom2\_info7.

- 將那 8 字元當作 key 後, 我們真的拿到 flag 了!!!花了我 2 天= = 要哭了, 噢噢忘記說 SystemFunction033() 本身是做 RC4 的加解密。

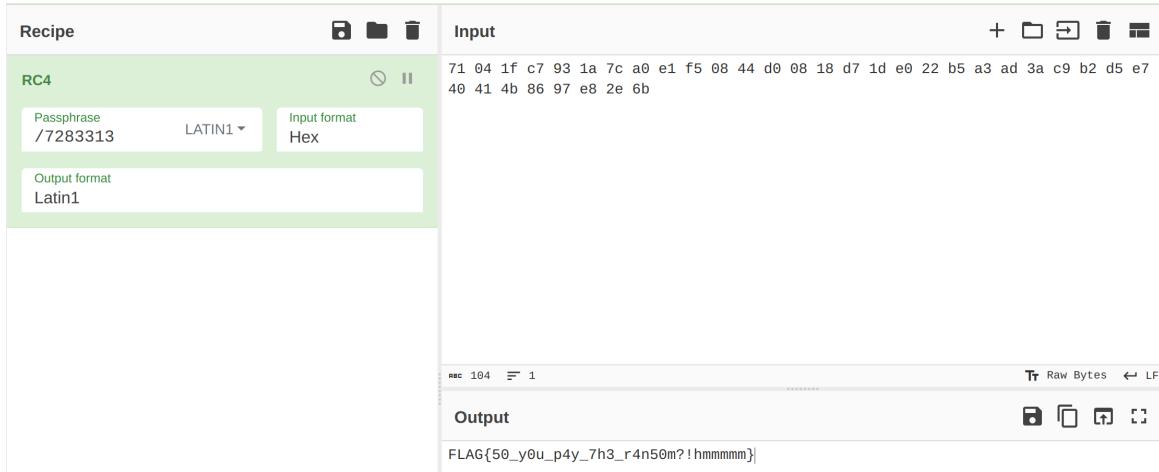


Figure 59: BabyRansom2\_Flag.

## [HW2] Evil FlagChecker

- FLAG{jmp1ng\_a1l\_ar0und}

## 解題流程和思路

- 本題最主要只要看 main\_4013C0() 和 verify\_4012A0(input, strlen(input)), 如 Figure 60 和 Figure 61 裡面有些參數已經被我重新定義和命名了, 在 main\_4013C0() 中可以看到會輸入一個 input 到 verify\_4012A0(input, strlen(input)) 去驗證, 如果輸入是對的就回傳 “Good”, 如果是錯的就回傳 “NO no no...”

```

1 void __noreturn main_4013C0()
2 {
3     bool result; // al
4     const char *v1; // edx
5     char input[1028]; // [esp+0h] [ebp-408h] BYREF
6
7     cout_4015C0((int *)std::cout, "Hello!\n");
8     memset(input, 0, 1024u);
9     cin_4017F0(std::cin, (int)input);
10    result = verify_4012A0(input, strlen(input));
11    v1 = "Good!\n";
12    if ( !result )
13        v1 = "No no no...\\n";
14    cout_4015C0((int *)std::cout, v1);
15    ExitProcess(0);
16 }

```

Figure 60: EvilFlagChecker\_info1.

- 在 Figure 61 中我們可以知道 input 經過一些算式之後最後跟 compare\_var 做比較, 不過這邊可以說一下 output[v5] 其實就是 input 的初始位置, 也就是 output[v5 + v4] = input[v4]

```

15 | memset(output, 0, 1024u);
16 | v4 = 0;
17 | if ( sizeOfInput )
18 |
19 |     v5 = input - output;
20 |     v13 = input - output;
21 |     do
22 |     {
23 |         v6 = v3 ^ output[v4 + v5];
24 |         output[v4] = v6;
25 |         v3 = sizeOfInput + (v6 ^ __ROR4__(v3, 3)) - v4;
26 |         Sleep(1000u);
27 |         cout_4015C0(std::cout, ".");
28 |         v5 = v13;
29 |         ++v4;
30 |     }
31 |     while ( v4 < sizeOfInput );
32 |
33 |     cout_4015C0(std::cout, "\\n");
34 |     v7 = output;
35 |     compare_var = byte_403400;
36 |     v9 = sizeOfInput < 4;
37 |     v10 = sizeOfInput - 4;
38 |     if ( v9 )
39 |     {
40 | LABEL_7:
41 |         if ( v10 == -4 )
42 |             return 1;
43 |     }
44 |     else
45 |     {
46 |         while ( *(_DWORD *)v7 == *(_DWORD *)compare_var )
47 |         {
48 |             v7 += 4;
49 |             compare_var += 4;
50 |             v9 = v10 < 4;
51 |             v10 -= 4;
52 |             if ( v9 )
53 |                 goto LABEL_7;
54 |         }
55 |     }
56 |     return *v7 == *compare_var
57 |     && (v10 == -3
58 |         || v7[1] == compare_var[1] && (v10 == -2 || v7[2] == compare_var[2] && (v10 == -1 || v7[3] == compare_var[3])));

```

Figure 61: EvilFlagChecker\_info2.

- 接著根據上述的算術邏輯寫一個 z3 去解就好, 如 Figure 62, 我這裡有個大問題= =, 為什麼 BitVec 設 64 bits 才會對..., 程式不是是輸入字元嗎...字元的大小是 8bits 阿....我在解的當下卡這個卡超久, 回去看 lab scramble, 看到設 32bits(不過題目很明顯是 4bytes), 然後想說手賤條一下, 改成 32 結果 flag 前面一半都對了, 再改成 64, 答案救出來了。

```

from z3 import *

compare_var = [237, 3, 129, 105, 123, 132, 166, 160, 91, 43, 182, 230, 92, 87, 201, 153, 232, 178, 32, 114

def ROR4(x, r):
    return ((x >> r) | (x << (32 - r))) & 0xFFFFFFFF

def solve():
    input_size = 23
    input_var = [z3.BitVec(f'input_{i}', 64) for i in range(input_size)]
    s = Solver()
    v3 = 0xE0C92EAB
    output = [0]*1024
    for i in range(input_size):
        v6 = (v3 ^ input_var[i]) & 0xFF
        output[i] = v6
        s.add(output[i] == compare_var[i])
        v3 = input_size + (v6 ^ ROR4(v3, 3)) - i

    if s.check() == sat:
        model = s.model()
        flag = ''
        for bv in input_var:
            flag += chr(model[bv].as_long())
        print(flag)
        return True
    else:
        return None

solve()

```

Figure 62: EvilFlagChecker\_info3.

```

Lambo@Ubuntu-Lambo:~/Desktop/hw2_r12922054/Reverse2/Evil FlagChecker$ python solve_z3.py
FLAG{jmp1ng_a1l_ar0und}

```

Figure 63: EvilFlagChecker\_Flag.

## [HW2] Trashcan

- FLAG{mpl3\_tr4S5\_caN}

### 解題流程和思路

1. 本題最主要只要看 main 函式, 和 15B0、13B0, 我有將變數 rename 和轉型, 此題助教應該目的是要我們學會 reverse 繼承的部份。

```

18    loop_index = 0;
19    Block[0] = 0i64;
20    input_length = 0i64;
21    v17 = 15i64;
22    A = operator new(16ui64);
23    *A = 0i64;
24    A->Vtable = &Trashcan::`vtable`;
25    class2_obj = operator new(16ui64);
26    *class2_obj = 0i64;
27    class2_obj->member1 = 0i64;
28    class2_obj->member2 = 0;
29    A->field_8 = class2_obj;
30    Cin_140001BC0(std::cin, Block);
31    v5 = v17;
32    v6 = Block[0];                                // input
33    len = input_length;                           // len
34    if ( input_length )
35    {
36        count = 0i64;
37        do
38        {
39            v9 = Block;
40            if ( v5 >= 16 )                         // 15b0
41                v9 = v6;
42            (A->Vtable->method2)(A, *(v9 + count));
43            ++loop_index;
44            ++count;
45        }
46        while ( loop_index < len );
47    }
48    v10 = (A->Vtable->method1)(A);           // 13b0
49    v11 = "Trashcan accept your input\n";
50    if ( !v10 )
51        v11 = "Trashcan reject your input\n";
52    Cout_1400019F0(std::cout, v11);

```

Figure 64: Trashcan\_info1.

2. 接著進到 15B0, Figure 65 為我有調整過後的樣子。

```

1 class1 * __fastcall sub_1400015B0(class1 *a1, int a2)
2 {
3     class2 *class2_obj; // rdi
4     int a1_member2_member2; // ebp
5     __int64 *a1_member2_member1; // rbx
6     class1 *result; // rax
7
8     class2_obj = a1->field_8;
9     a1_member2_member2 = class2_obj->member2;
10    a1_member2_member1 = class2_obj->member1;
11    class2_obj->member2 = a1_member2_member2 + 1;
12    if ( a1_member2_member1 )
13    {
14        if ( a2 >= *(a1_member2_member1 + 1) )
15        {
16            result = class1::class1(class2_obj, a1_member2_member1[2], a2, a1_member2_member2);
17            a1_member2_member1[2] = result;
18        }
19    else
20    {
21        result = class1::class1(class2_obj, a1_member2_member1[1], a2, a1_member2_member2);
22        a1_member2_member1[1] = result;
23    }
24    class2_obj->member1 = a1_member2_member1;
25 }
26 else
27 {
28     result = operator new(0x18ui64);
29     result->field_0 = a1_member2_member2;
30     result->field_4 = a2;
31     result->field_8 = 0i64;
32     result->field_10 = 0i64;
33     class2_obj->member1 = result;
34 }
35 LOBYTE(result) = 1;
36 return result;
37}

```

Figure 65: Trashcan\_info2.

3. 最後進到 13B0, 然後變數經過整過後, 我用 x64dbg, 直接動態分析並且設斷點在 14E0 的位置, 為什麼是設斷點在 14E0 ? 因為 37-51 行看起來在進行某個操作, 噢噢, 另外, 提一下, 因為助教好像有用 Anti-Debugger, 所以在 x64dbg 裡面要裝 scylla hide, 去防 Anti-Debugger。 (到這邊我其實只是調整變數的型態(class)和 rename, 並還沒有分析程式邏輯...下面告訴你我怎麼就這樣解出來了...)

• xt:00000001400014CD	movdqa [rbp+10h+var_40], xmm0	• 31 HIDWORD(v12) = *v3;
• xt:00000001400014D2	mov [rbp+10h+var_30], 0Ch	• 32 sub_140001F60(Block, 0i64, &v12);
• xt:00000001400014D9	mov [rbp+10h+var_20], 15h	• 33 sub_140001320(v2, *(v3 + 8), Block);
• <b>xt:00000001400014E0</b>	<b>mov r9d, esi</b>	• 34 sub_140001320(v2, *(v3 + 16), Block);
• xt:00000001400014E3	mov rbx, [rbp+10h+Block+8]	• 35 v1 = v20;
• xt:00000001400014E7	mov r11, [rbp+10h+Block]	• 36 }
• xt:00000001400014EE	sub rbx, r11	• 37 v4 = 1;
• xt:00000001400014EB	sar rbx, 3	• 38 flag1[0] = _mm_load_si128(&xmmword_1400044D0);
• xt:00000001400014EF	test rbx, rbx	• 39 flag1[1] = _mm_load_si128(&xmmword_1400044E0);
• xt:00000001400014F2	jz short loc_140001535	• 40 flag1[2] = _mm_load_si128(&xmmword_140004500);
• xt:00000001400014F5	mov rdx, r11	• 41 flag1[3] = _mm_load_si128(&xmmword_1400044F0);
• xt:00000001400014F7	mov r8, rsi	• 42 flag1[4] = _mm_load_si128(&xmmword_140004510);
• xt:00000001400014FA	nop dword ptr [rax]	• 43 v14 = 116;
• xt:00000001400014FD		• 44 v15 = 125;
• xt:0000000140001500		• 45 flag2[0] = _mm_load_si128(&xmmword_1400044A0);
• xt:0000000140001500 loc_140001500:		• 46 flag2[1] = _mm_load_si128(&xmmword_1400044B0);
• xt:0000000140001504	cmp r9d, 16h	• 47 flag2[2] = _mm_load_si128(&xmmword_140004490);
• xt:0000000140001506	jl short loc_14000150A	• 48 flag2[3] = _mm_load_si128(&xmmword_1400044C0);
• xt:0000000140001508	mov ecx, esi	• 49 flag2[4] = _mm_load_si128(&xmmword_1400044B0);
• xt:0000000140001508 loc_14000150A:	jmp short loc_140001522	• 50 v17 = 12;
• xt:000000014000150A ; -----		• 51 v18 = 21;
• xt:000000014000150A		• 52 len = 0;
• xt:000000014000150A loc_14000150A:	mov eax, dword ptr [rsp+r8]	• 53 v6 = Block[0];
• xt:000000014000150A	cmp [rdx], eax	• 54 input_length = (Block[1] - Block[0]) >> 3; ActivateWindbg
• xt:000000014000150F		

Figure 66: Trashcan\_info3.

4. Figure 67 為在 x64dbg 上設置 breakpoint。

● 00007FF767BC142C	48:8B CF	mov rcx,rdi
● 00007FF767BC142F	E8 ECFFFF	call trashcan-dist.7FF767BC1320
● 00007FF767BC1434	4C:8B53 00	mov r10,qword ptr ss:[rbp]
● 00007FF767BC1438	B9 01000000	mov ecx,1
● 00007FF767BC143D	66:0F60 8B 300000	movdq xmm0,xmmword ptr ds:[7FF767BC44D0]
● 00007FF767BC1445	66:0F7F44 24 30	movdq xmmword ptr ss:[rbp+30],xmm0
● 00007FF767BC144B	66:0F60 00 3D300000	movdq xmm1,xmmword ptr ds:[7FF767BC44E0]
● 00007FF767BC1453	66:0F7F4C 24 40	movdq xmmword ptr ss:[rbp+40],xmm1
● 00007FF767BC1459	66:0F60 05 9F300000	movdq xmm0,xmmword ptr ds:[7FF767BC4500]
● 00007FF767BC1461	66:0F7F4C 24 50	movdq xmmword ptr ss:[rbp+50],xmm0
● 00007FF767BC1467	66:0F60D 81300000	movdq xmm1,xmmword ptr ds:[7FF767BC44F0]
● 00007FF767BC146F	66:0F7F4C 24 60	movdq xmmword ptr ss:[rbp+60],xmm1
● 00007FF767BC1475	66:0F60 05 93300000	movdq xmm0,xmmword ptr ds:[7FF767BC4510]
● 00007FF767BC147D	66:0F7F4C 24 70	movdq xmmword ptr ss:[rbp+70],xmm0
● 00007FF767BC1483	C745 80 74000000	mov dword ptr ss:[rbp-80],74
● 00007FF767BC148A	C745 84 7D000000	mov dword ptr ss:[rbp-7C],7D
● 00007FF767BC1491	66:0F60 05 07300000	movdq xmm0,xmmword ptr ds:[7FF767BC44A0]
● 00007FF767BC1499	66:0F7F45 90	movdq xmmword ptr ss:[rbp-90],xmm0
● 00007FF767BC149E	66:0F60D DA2F0000	movdq xmm1,xmmword ptr ds:[7FF767BC4480]
● 00007FF767BC14A6	66:0F7F4D A0	movdq xmmword ptr ss:[rbp-60],xmm1
● 00007FF767BC14AB	66:0F60 05 D2F0000	movdq xmm0,xmmword ptr ds:[7FF767BC4490]
● 00007FF767BC14B3	66:0F7F45 B0	movdq xmmword ptr ss:[rbp-50],xmm0
● 00007FF767BC14B8	66:0F60D 03000000	movdq xmm1,xmmword ptr ds:[7FF767BC44C0]
● 00007FF767BC14C0	66:0F7F4D C0	movdq xmmword ptr ss:[rbp-40],xmm1
● 00007FF767BC14C5	66:0F60 05 E32F0000	movdq xmm0,xmmword ptr ds:[7FF767BC44B0]
● 00007FF767BC14CD	66:0F7F45 D0	movdq xmmword ptr ss:[rbp-30],xmm0
● 00007FF767BC14D2	C745 E0 0C000000	mov dword ptr ss:[rbp-20],C
● 00007FF767BC14D9	C745 E4 15000000	mov dword ptr ss:[rbp-1C],15
RIP → 00007FF767BC14E0	44:8BCE	mov r9d,esi
● 00007FF767BC14E3	48:8B5D F8	mov rbx,qword ptr ss:[rbp-8]
● 00007FF767BC14E7	4C:8B5D F0	mov r11,qword ptr ss:[rbp-10]

Figure 67: Trashcan\_info4

5. 查看記憶體位址(點 ss:[rbp-1C]右鍵), 然後就可以看到 Figure 68。

000000F3C36FFBB0	46 00 00 00	41 00 00 00	31 00 00 00	33 00 00 00	F . A . 1 . 3 .
000000F3C36FFBC0	34 00 00 00	35 00 00 00	4C 00 00 00	47 00 00 00	4 . 5 . L . G .
000000F3C36FFBD0	78 00 00 00	73 00 00 00	6D 00 00 00	6C 00 00 00	{ . s . m . 1 .
000000F3C36FFBE0	5F 00 00 00	53 00 00 00	4E 00 00 00	5F 00 00 00	_ . S . N . _ .
000000F3C36FFBF0	63 00 00 00	61 00 00 00	70 00 00 00	72 00 00 00	c . a . p . r .
000000F3C36FFFC00	74 00 00 00	7D 00 00 00	07 13 BC 67	F7 7F 00 00	t . } . . ¼g ÷ .
000000F3C36FFC10	00 00 00 00	02 00 00 00	06 00 00 00	0A 00 00 00	.....
000000F3C36FFC20	0E 00 00 00	10 00 00 00	01 00 00 00	03 00 00 00	.....
000000F3C36FFC30	04 00 00 00	05 00 00 00	07 00 00 00	09 00 00 00	.....
000000F3C36FFC40	08 00 00 00	0F 00 00 00	14 00 00 00	11 00 00 00	.....
000000F3C36FFC50	12 00 00 00	13 00 00 00	08 00 00 00	0D 00 00 00	.....
000000F3C36FFC60	0C 00 00 00	15 00 00 00	22 16 BC 67	F7 7F 00 00	". ¼g ÷ .

Figure 68: Trashcan\_info5

6. 看一下這些字元, Figure 69, 是不是超像答案的 OAO

000000F3C36FFBBO	46 00 00 00	41 00 00 00	31 00 00 00	33 00 00 00	F . . A . 1 - 3
000000F3C36FFBC0	34 00 00 00	35 00 00 00	4C 00 00 00	47 00 00 00	4 . . 5 . L G
000000F3C36FFBD0	7B 00 00 00	73 00 00 00	6D 00 00 00	6C 00 00 00	{ . . s . m . 1
000000F3C36FFBE0	5F 00 00 00	53 00 00 00	4E 00 00 00	5F 00 00 00	_ . . S . N . _
000000F3C36FFBF0	63 00 00 00	61 00 00 00	70 00 00 00	72 00 00 00	c . . a . p . r
000000F3C36FFC00	74 00 00 00	7D 00 00 00	07 13 BC 67 F7 7F 00 00	t . . } . . . . . .	Yg ÷

Figure 69: Trashcan\_info6.

7. 再往下幾行看, Figure 70, 然後按造順序填寫出來就是答案了 = =。 eg. 00->46(F), 01->4C(L), 02->41(A), 03->47(G)...如此繼續到 15->7D({})。

000000F3C36FFC10	00 00 00 00 02 00 00 00	06 00 00 00 0A 00 00 00	.
000000F3C36FFC20	0E 00 00 00 10 00 00 00	01 00 00 00 03 00 00 00	.
000000F3C36FFC30	04 00 00 00 05 00 00 00	07 00 00 00 09 00 00 00	.
000000F3C36FFC40	0B 00 00 00 0F 00 00 00	14 00 00 00 11 00 00 00	.
000000F3C36FFC50	12 00 00 00 13 00 00 00	08 00 00 00 0D 00 00 00	.
000000F3C36FFC60	0C 00 00 00 15 00 00 00	22 16 BC 67 F7 7F 00 00	" %g %.

Figure 70: Trashcan\_info7