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!
>bW92IGVheCwgZHdvcmQgW3JzcF07IGxldCBlYXggPSBhCmltdWwgZWR4LCBlYXgsIDkKYWRkIGVkeCw
gNwoKbW92IGVieCwgZHdvcmQgW3JzcF0KbW92IGVjeCwgZHdvcmQgW3JzcCs0XQphZGQgZWF4LCBlY3g
Kc3ViIGVieCwgZWN4Cgptb3YgZWN4LCBkd29yZCBbcnNwKzhdCm5lZyBlY3gKCg==
```

Figure 1: arithmetic_Cmd.

```
Give me your base64 of your assembly code!
>OyBNb2RpZnkgcmVnaXN0ZXIgdmFsdWVzCmFkZCByYXgsIDB4ODcgICAgICAgICAgICAgICAgICAgICA
gIDsgUkFYICs9IDB4ODcKc3ViIHJieCwgMHg2MyAgICAgICAgICAgICAgICAgICAgICAgICAgOyBSQlggLT0
gMHg2Mwp4Y2hnIHJjeCwgcmR4ICAgICAgICAgICAgICAgICAgICAgICA7IFN3YXAgUkNYIGFuZCBSRFg
KCjsgTW9kaWZ5IG1lbW9yeSB2YWx1ZXMKYWRkIGR3b3JkIFtyc3BdLCAweGRlYWRiZWVmICAgICAgICA
gOyBNRU1bUlNQKzB4MDpSU1ArMHg0XSArPSAweGRlYWRiZWVmCnN1YiBkd29yZCBbcnNwKzRdLCAweGZ
hY2ViMDBjICAgICAgIDsgTUVNW1JTUCsweDQ6UlNQKzB4OF0gLT0gMHhmYWNlYjAwYwoKYWRkIHJzcCw
gOAptb3YgcjhkLCBkd29yZCBbcnNwXSAgICAgICAgICAgICA7IExvYWQgTUVNW1JTUCsweDg6UlN
QKzB4Y10gaW50byByOGQKbW92IHI5ZCwgZHdvcmQgW3JzcCs0XSAgICAgICAgICAgOyBMb2FkIE1
FTVtSU1ArMHhjOlJTUCsweDEwXSBpbnRvIHI5ZAptb3YgZHdvcmQgW3JzcF0sIHI5ZCAgICAgICAgICA
gICAgICA7IFN0b3JlIHI5ZCBpbnRvIE1FTVtSU1ArMHg4OlJTUCsweGNdCm1vdiBkd29yZCBbcnNwKzR
dLCByOGQgICAgICAgICAgICAgIDsgU3RvcmUgcjhkIGludG8gTUVNW1JTUCsweGM6UlNQKzB4MTBdCg=
```

Figure 2: datamovement_Cmd.

```
Give me your base64 of your assembly code!
>bW92IGVheCwgZHdvcmQgW3JzcF0gICAgICAgICAgICA7IExvYWQgYSBmcm9tIFtSU1BdCm1vdiBlYng
sIGR3b3JkIFtyc3AgKyA0XSAgICAgICAgICAgOyBMb2FkIGIgZnJvbSBbUlNQKzRdCm1vdiBlY3gsIGR3b3J
kIFtyc3AgKyA4XSAgICAgICAgIAk7IExvYWQgYyBmcm9tIFtSU1ArOF0KbW92IGVkeCwgZHdvcmQgW3JzcCA
rIDEyXSAgICAgCTsgTG9hZCBkIGZyb20gW1JTUCsxMl0KCmNtcCBlYXgsIGVieCAgICAgICAgICAgICA
gICAgICAgOyBDb21wYXJlIGEgYW5kIGIKamdlIGdyZWF0ZXJfb3JfZXF1YWwgICAgICAgICA7IEp
1bXAgaWYgYSA+PSBiCm1vdiBlYXgsIGVieCAgICAgICAgICAgICAgICAgICAgOyBTZXQgRUFYIHRvIGI
gaWYgYSA8IGIKCmdyZWF0ZXJfb3JfZXF1YWw6CiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICA
gOyBFQVggaXMgYWxyZWFkeSBhIGlmIGEgPj0gYgogICAgIApjbXAgZWN4LCBlZHggICAgICAgICAgICA
7IENvbXBhcmUgYyBhbmQgZApqYiBsZXNzX3RoYW4gICAgICAgICA7IEp1bXAgaWYgYyA8IGQKbW9
2IGVieCwgZWR4ICAgICAgICAgICAgOyBTZXQgRUJYIHRvIGQgaWYgYyA+PSBkCgpsZXNzX3RoYW46CiA
gICAgICAgbW92IGVieCwgZWN4ICAgICAgICAgICAgOwoKdGVzdCBlY3gsIDEgICAgICAgICAgICAgOyB
UZXN0IGlmIGMgaXMgb2RkCmpueiBpc19vZGQgICAgICAgICAgIDsgSnVtcCBpZiBjIGlzIG9kZAp
zaGwgZWN4LCAyCmptcCBkb25lCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICA7IFN0b3JlIDMKZG9uZToK
```

Figure 3: condition_Cmd.

```
──── Your Flag ──────────────────────────────────────────────────
Congrats! You passed all challenges! Here is your flag:  FLAG{c0d1Ng_1n_a5s3mB1y
_i5_sO_fun!}
```

Figure 4: Assembly_Flag.

[Lab] HelloRevWorld

- FLAG{h311O_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↑o
.rodata:0000000000002008                 text "UTF-32LE", 'FLAG{h311O_revers1ng_3ngineer5}',0
.rodata:0000000000002088 unk_2088        db  25h ; %             ; DATA XREF: main+F↑o
```

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_hOw_to_r3v3r53_4_m47riX!}

解題流程和思路

1. 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,
   ```
   strcut 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 = (struct1 *)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.

2. 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。
3. struct1 中的 member1 和 strcut2 的 memeber1 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ㄝ, 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

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

也就是

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

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。

```
lambo@Ubuntu-Lambo:~/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.

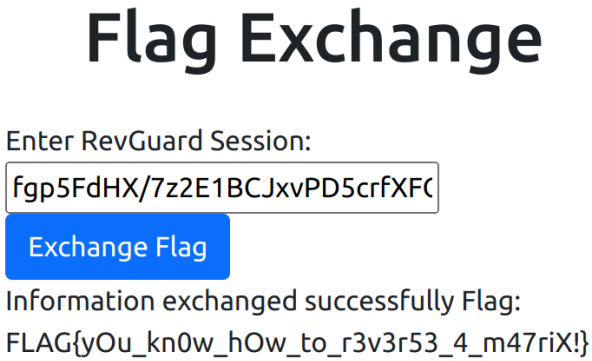


Figure 11: crackme_vectorization_Flag.

[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

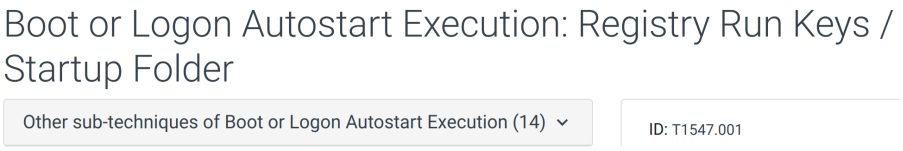


Figure 12: Clipboard_Stealer_1_Flag.

[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 中查詢即可得知為為 T1480, 如下 Figure 13

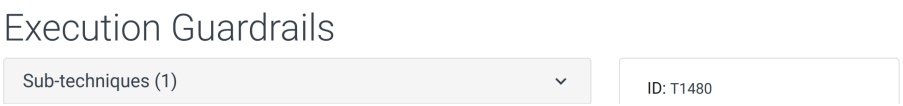


Figure 13: Clipboard_Stealer_2_Flag.

[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)

```
int createMutex_1120()
{
  HANDLE MutexA; // rax
  int i; // [rsp+20h] [rbp-78h]
  int v3[2]; // [rsp+30h] [rbp-68h]
  CHAR Name[32]; // [rsp+38h] [rbp-60h] BYREF
  char v5[32]; // [rsp+58h] [rbp-40h] BYREF

  v3[0] = 1684234874;
  qmemcpy(v5, &unk_140003348, 0x1Dui64);
  for ( i = 0; i < 28; ++i )
    Name[i] = *((_BYTE *)v3 + i % 4) ^ v5[i];
  MutexA = CreateMutexA(0i64, 1, Name);
  if ( MutexA )
  {
    LODWORD(MutexA) = GetLastError();
    if ( (_DWORD)MutexA == 183 )
      exit(0);
  }
  return (int)MutexA;
}
```

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

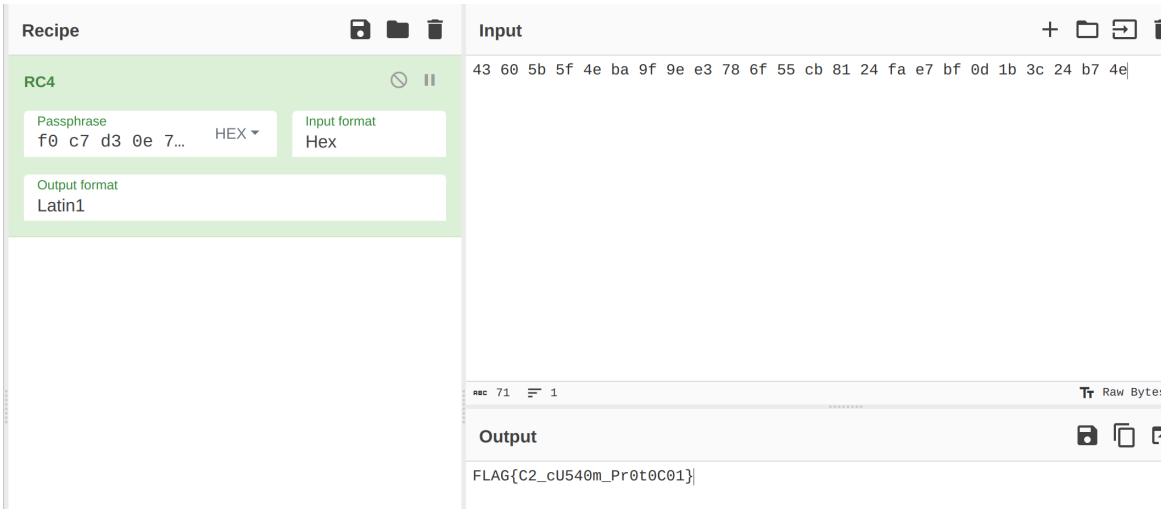


Figure 19: Clipboard_Stealer_5_Flag.

[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_user
32dll_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

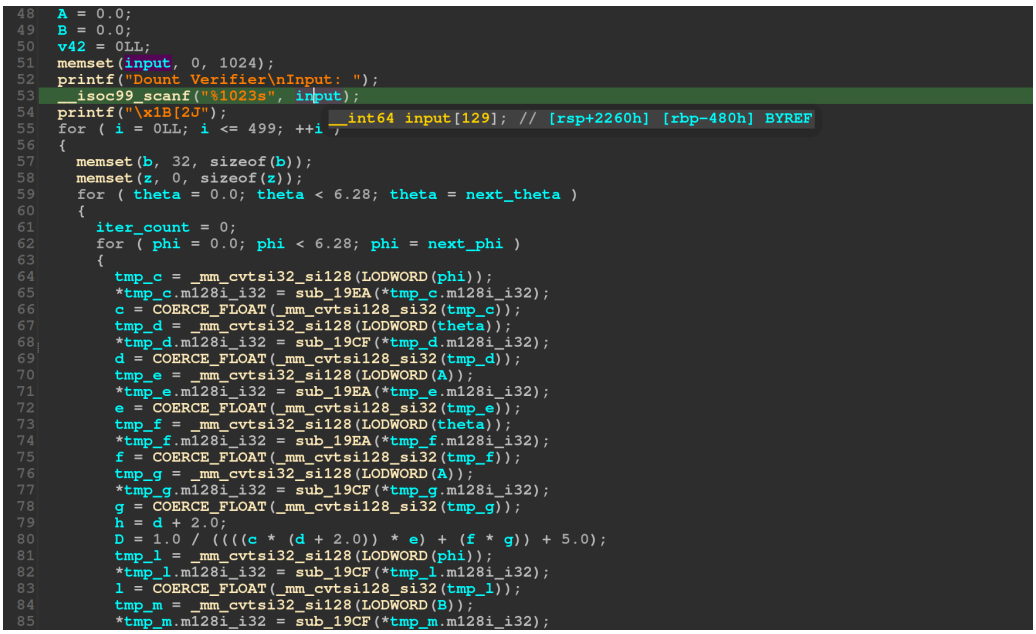
1. 首先, 先跑一次執行檔, 發現真的跟 Donut 的實作結果很像, 因此參照 https://medium.com/analytics-vidhya/the-donuts-code-the-math-behind-6d473eaec61d ,開始分析 Figure 22

```
#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;
}
```

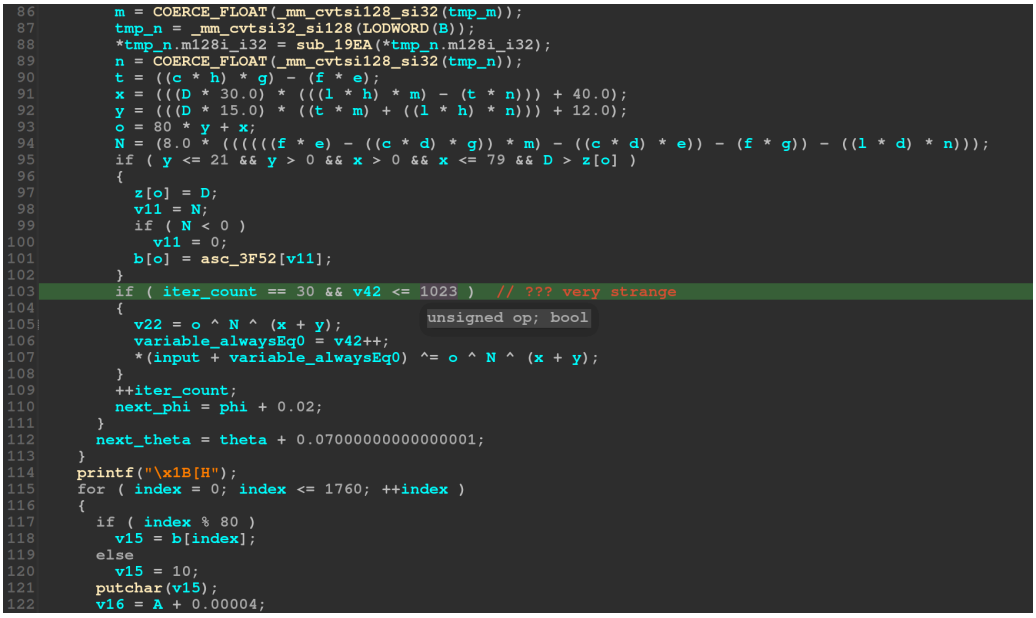
Figure 22: Reference of Banana Donut Verify.

2. 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



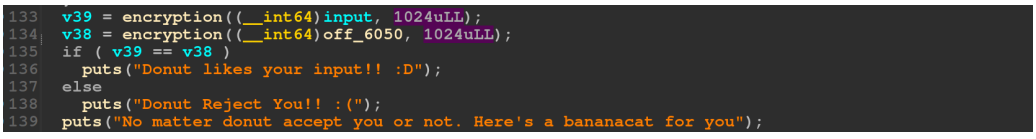
```
A = 0.0;
B = 0.0;
v42 = 0LL;
memset(input, 0, 1024);
printf("Dount Verifier\nInput: ");
__isoc99_scanf("%1023s", input);
printf("\x1B[2J");                              __int64 input[129]; // [rsp+2260h] [rbp-480h] BYREF
for ( i = 0LL; i <= 499; ++i )
{
  memset(b, 32, sizeof(b));
  memset(z, 0, sizeof(z));
  for ( theta = 0.0; theta < 6.28; theta = next_theta )
  {
    iter_count = 0;
    for ( phi = 0.0; phi < 6.28; phi = next_phi )
    {
      tmp_c = _mm_cvtsi32_si128(LODWORD(phi));
      *tmp_c.m128i_i32 = sub_19EA(*tmp_c.m128i_i32);
      c = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_c));
      tmp_d = _mm_cvtsi32_si128(LODWORD(theta));
      *tmp_d.m128i_i32 = sub_19CF(*tmp_d.m128i_i32);
      d = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_d));
      tmp_e = _mm_cvtsi32_si128(LODWORD(A));
      *tmp_e.m128i_i32 = sub_19EA(*tmp_e.m128i_i32);
      e = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_e));
      tmp_f = _mm_cvtsi32_si128(LODWORD(theta));
      *tmp_f.m128i_i32 = sub_19EA(*tmp_f.m128i_i32);
      f = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_f));
      tmp_g = _mm_cvtsi32_si128(LODWORD(A));
      *tmp_g.m128i_i32 = sub_19CF(*tmp_g.m128i_i32);
      g = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_g));
      h = d + 2.0;
      D = 1.0 / ((((c * (d + 2.0)) * e) + (f * g)) + 5.0);
      tmp_l = _mm_cvtsi32_si128(LODWORD(phi));
      *tmp_l.m128i_i32 = sub_19CF(*tmp_l.m128i_i32);
      l = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_l));
      tmp_m = _mm_cvtsi32_si128(LODWORD(B));
      *tmp_m.m128i_i32 = sub_19CF(*tmp_m.m128i_i32);
```

Figure 23: Banana_Donut_Verify_IDAPseudocode1.



```
      m = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_m));
      tmp_n = _mm_cvtsi32_si128(LODWORD(B));
      *tmp_n.m128i_i32 = sub_19EA(*tmp_n.m128i_i32);
      n = COERCE_FLOAT(_mm_cvtsi128_si32(tmp_n));
      t = ((c * h) * g) - (f * e);
      x = (((D * 30.0) * (((l * h) * m) - (t * n))) + 40.0);
      y = (((D * 15.0) * ((t * m) + ((l * h) * n))) + 12.0);
      o = 80 * y + x;
      N = (8.0 * ((((((f * e) - ((c * d) * g)) * m) - ((c * d) * e)) - (f * g)) - ((l * d) * n)));
      if ( y <= 21 && y > 0 && x > 0 && x <= 79 && D > z[o] )
      {
        z[o] = D;
        v11 = N;
        if ( N < 0 )
          v11 = 0;
        b[o] = asc_3F52[v11];
      }
      if ( iter_count == 30 && v42 <= 1023 )  // ??? very strange
      {                                       unsigned op; bool
        v22 = o ^ N ^ (x + y);
        variable_alwaysEq0 = v42++;
        *(input + variable_alwaysEq0) ^= o ^ N ^ (x + y);
      }
      ++iter_count;
      next_phi = phi + 0.02;
    }
    next_theta = theta + 0.07000000000000001;
  }
  printf("\x1B[H");
  for ( index = 0; index <= 1760; ++index )
  {
    if ( index % 80 )
      v15 = b[index];
    else
      v15 = 10;
    putchar(v15);
    v16 = A + 0.00004;
```

Figure 24: Banana_Donut_Verify_IDAPseudocode2.



```
v39 = encryption((__int64)input, 1024uLL);
v38 = encryption((__int64)off_6050, 1024uLL);
if ( v39 == v38 )
  puts("Donut likes your input!! :D");
else
  puts("Donut Reject You!! :(");
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:

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

, where

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

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)

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

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
0x7fffffffd880: 13      58      -117    -118    -118    -118    -119    -119
0x7fffffffd888: -120    -120    -120    -120    89      86      84      84
0x7fffffffd890: 84      86      89      89      39      86      86      87
0x7fffffffd898: 75      75      72      72      73      73      78      78
0x7fffffffd8a0: 75      -7      -7      -8      7       7       7       5
0x7fffffffd8a8: 4       7       7       6       1       1       2       2
0x7fffffffd8b0: 1       1       0       -79     -75     -70     -70     -70
0x7fffffffd8b8: -70     -70     -89     -89     -88     -88     -88     -85
0x7fffffffd8c0: -85     -87     120     120     119     119     119     118
0x7fffffffd8c8: 118     117     -60     -53     -52     -52     -51     -51
0x7fffffffd8d0: -50     -49     -49     -49     48      48      49      49
0x7fffffffd8d8: 50      51      58      56      57      -118    -118    -119
0x7fffffffd8e0: -119    -119    -120    -120    -120    -120    89      91
0x7fffffffd8e8: 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

```
pwndbg> x/1024b $rdi
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

```python
from sage.all import *
from Crypto.Util.number import *

v39_input = [   13, 58, -117, -118, -118, -118, -119, -119,
                -120, -120, -120, -120, 89, 86, 84, 84,
                84, 86, 89, 89, 39, 86, 86, 87,
                75, 75, 72, 72, 73, 73, 78, 78,
                75, -7, -7, -8, 7,  7,  7,  5,
                4,  7,  7,  6,  1,  1,  2,  2,
            ]

v38_input = [   71, 86, -8, -66, -3, -5, -90, -5,
                -89, -1, -14, -14, 12, 99, 51, 17,
                101, 47, 24, 33, 105, 99, 53, 37,
                45, 19, 14, 11, 51, 123, 127, 39,
                35, -68, -78, -79, 81, 108, 48, 70,
                67, 115, 118, 124, 77, 88, 91, 63,
            ]

v39_input[0] ^= 48 # 0 ascii


input_string = ''
for i in range(48):
    input_string += (v39_input[i] ^ v38_input[i]).to_bytes(1, 'big').decode()

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/wzzU5gE1yAxN5crfXFCz21ihEKIVk7CGtqzLYY=
```

Figure 30: Banana_Donut_Verify_Cmd.

11. 獲得 Flag

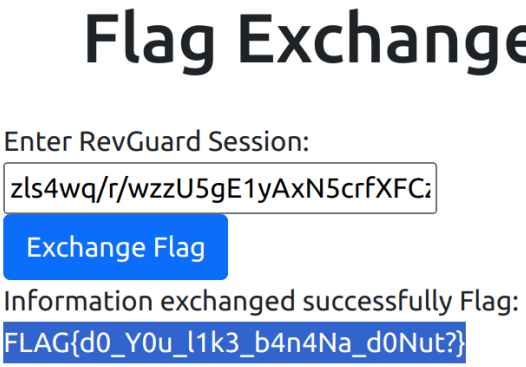


Figure 31: Banana_Donut_Verify_Flag.

[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 sup
er_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 scramb
le_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

1. 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

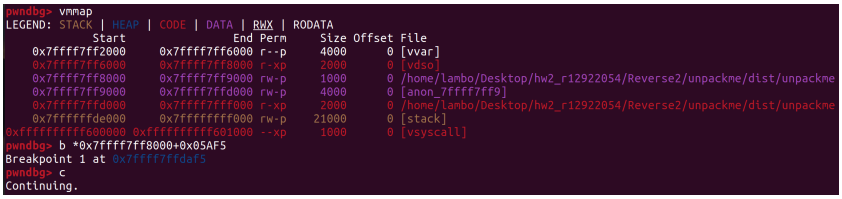


Figure 36: unpackme_gdb_info1.

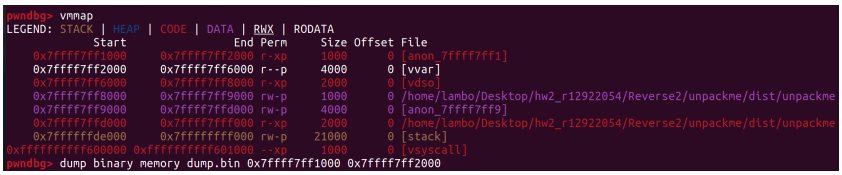


Figure 37: unpackme_gdb_info2.

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

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

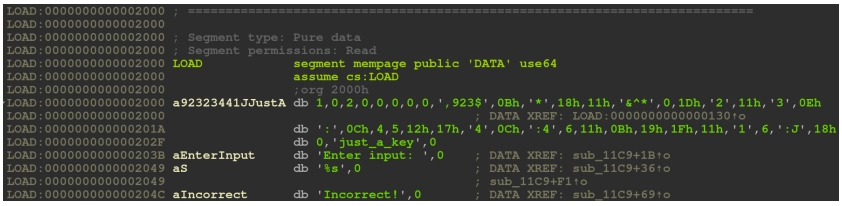


Figure 39: unpackme_gluedBin_IDAinfo.

5. 執行 unpackme, 並獲得 Flag

Figure 40: unpackme_Cmd.

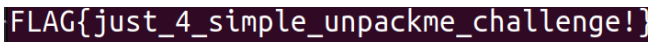


Figure 41: unpackme_Flag.

[HW2] Baby Ransom 1 – Next Stage Payload

· FLAG{e6b77096375bcff4c8bc765e599fbbc0}

解題流程和思路

1. 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 criticl 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.

2. 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.

3. 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 gototInject197A(), 如圖 : Figure 44

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

Figure 44: BabyRansom1_info3.

4. 戳進去最裡面後會看到如圖 : Figure 45, 但沒啥東西, 繼續前往 CreateProcesss1653()

```
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.

5. 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。

```
__int64 CreateProcess1653()
{
  __int64 sizeOfsomething; // [rsp+58h] [rbp-28h] BYREF
  __int64 ProcessBaseAddress; // [rsp+60h] [rbp-20h] BYREF
  int v3; // [rsp+6Ch] [rbp-14h]
  struct _PROCESS_INFORMATION ProcessInformation; // [rsp+70h] [rbp-10h] BYREF
  struct _STARTUPINFOA StartupInfo; // [rsp+90h] [rbp+10h] BYREF
  DWORD sizeOf_pe; // [rsp+104h] [rbp+84h] BYREF
  int *firstByteOf_pe; // [rsp+108h] [rbp+88h] BYREF
  LPCONTEXT lpContext_baseAddr; // [rsp+110h] [rbp+90h]
  int j; // [rsp+118h] [rbp+98h]
  int i; // [rsp+11Ch] [rbp+9Ch]

  doPE_Encryption((LPVOID *)&firstByteOf_pe, &sizeOf_pe);
```

Figure 46: BabyRansom1_info5.

```
HRSRC __fastcall doPE_Encryption155A(LPVOID *ptr_FirstByteofResource, DWORD *sizeOfResource)
{
  HRSRC handle_for_loadingResource; // rax
  HRSRC handleResourceInfo; // [rsp+30h] [rbp-10h]
  unsigned int i; // [rsp+3Ch] [rbp-4h]

  handle_for_loadingResource = FindResourceA(0i64, (LPCSTR)0x44, (LPCSTR)0x84);
  handleResourceInfo = handle_for_loadingResource;
  if ( handle_for_loadingResource )
  {
    handle_for_loadingResource = (HRSRC)LoadResource(0i64, handle_for_loadingResource);
    if ( handle_for_loadingResource )
    {
      *ptr_FirstByteofResource = LockResource(handle_for_loadingResource);
      handle_for_loadingResource = (HRSRC)*ptr_FirstByteofResource;
      if ( *ptr_FirstByteofResource )
      {
        *sizeOfResource = SizeofResource(0i64, handleResourceInfo);
        handle_for_loadingResource = (HRSRC)*sizeOfResource;
        if ( (_DWORD)handle_for_loadingResource )
        {
          for ( i = 0; ; i += 2 )
          {
            handle_for_loadingResource = (HRSRC)*sizeOfResource;
            if ( i >= (unsigned int)handle_for_loadingResource )
              break;
            *(_WORD *)((char *)*ptr_FirstByteofResource + (int)i) ^= 0x8711u;
          }
        }
      }
    }
  }
  return handle_for_loadingResource;
}
```

Figure 47: BabyRansom1_info6.

6. 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

```
  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.

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

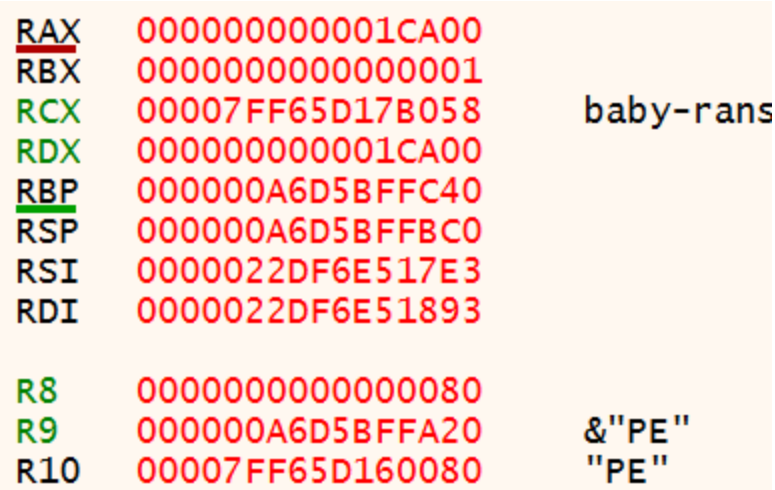


Figure 50: BabyRansom1_info9.

8. 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 ME
M_00007FF65D17B058_0001CA00.mem
e6b77096375bcff4c8bc765e599fbbc0  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。

1. 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.

2. 接著直接進到 WinMain 裡, 如下: Figure 53

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPST
{
  HWND hWnd; // [rsp+60h] [rbp-A8h]
  WNDCLASSW WndClass; // [rsp+70h] [rbp-98h] BYREF
  struct tagMSG Msg; // [rsp+C0h] [rbp-48h] BYREF

  memset(&WndClass, 0, sizeof(WndClass));
  WndClass.lpfnWndProc = store_winword;
  WndClass.hInstance = hInstance;
  WndClass.lpszClassName = Caption;
  WndClass.hbrBackground = CreateSolidBrush(0);
  if ( !RegisterClassW(&WndClass) )
    return 1;
  hWnd = CreateWindowExW(0, Caption, Caption, 0xCF0000u, 100, 100, 800,
  if ( !hWnd )
    return 2;
  Run2870();
  ShowWindow(hWnd, nShowCmd);
  memset(&Msg, 0, sizeof(Msg));
  while ( GetMessageW(&Msg, 0i64, 0, 0) )
  {
    TranslateMessage(&Msg);
    DispatchMessageW(&Msg);
  }
  return 0;
}
```

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 SearchAPIiB10(), 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！

```
__int64 Run2870()
{
  __int64 result; // rax

  LoadLibraryA(LibFileName);
  LoadLibraryA(aWininetDll);
  result = SearchAPI1B10();
  if ( !result )
    return BackUpFile_EncryptFile1660();
  return result;
}
```

Figure 54: BabyRansom2_info3.

4. 進到 BackUpFile_EncryptFile1660() 裡後, 其實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_EncrytFile1660()
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               encrytFile1960(filename, Buffer, dwSize);
34           }
35         }
36         while ( qword_140007AB8(v2, v5) );
```

Figure 55: BabyRansom2_info4.

5. 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 其實就是經過第 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 encrytFile1960(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.

6. 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)

Figure 57: BabyRansom2_info6.

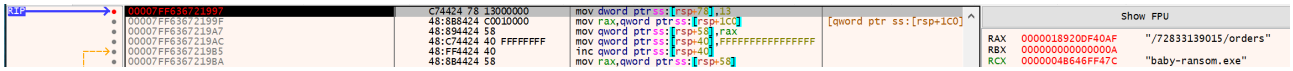


Figure 58: BabyRansom2_info7.

7. 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。

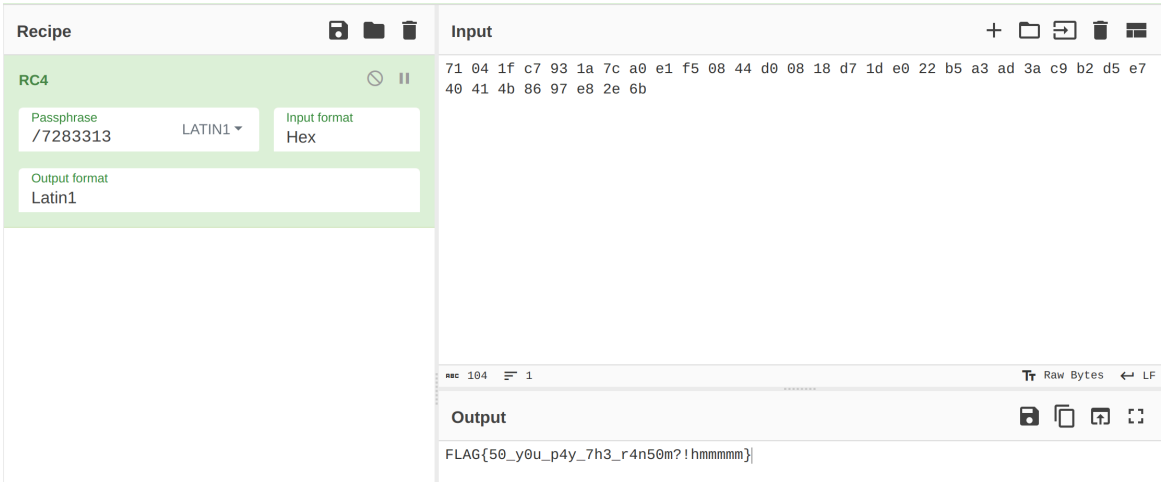


Figure 59: BabyRansom2_Flag.

[HW2] Evil FlagChecker

· FLAG{jmp1ng_a1l_ar0und}

解題流程和思路

1. 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...”

```
void __noreturn main_4013C0()
{
  bool result; // al
  const char *v1; // edx
  char input[1028]; // [esp+0h] [ebp-408h] BYREF

  cout_4015C0((int *)std::cout, "Hello!\n");
  memset(input, 0, 1024u);
  cin_4017F0(std::cin, (int)input);
  result = verify_4012A0(input, strlen(input));
  v1 = "Good!\n";
  if ( !result )
    v1 = "No no no...\n";
  cout_4015C0((int *)std::cout, v1);
  ExitProcess(0);
}
```

Figure 60: EvilFlagChecker_info1.

2. 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]

```
memset(output, 0, 1024u);
v4 = 0;
if ( sizeOfInput )
{
  v5 = input - output;
  v13 = input - output;
  do
  {
    v6 = v3 ^ output[v4 + v5];
    output[v4] = v6;
    v3 = sizeOfInput + (v6 ^ __ROR4__(v3, 3)) - v4;
    Sleep(1000u);
    cout_4015C0(std::cout, ".");
    v5 = v13;
    ++v4;
  }
  while ( v4 < sizeOfInput );
}
cout_4015C0(std::cout, "\n");
v7 = output;
compare_var = byte_403400;
v9 = sizeOfInput < 4;
v10 = sizeOfInput - 4;
if ( v9 )
{
LABEL_7:
  if ( v10 == -4 )
    return 1;
}
else
{
  while ( *(_DWORD *)v7 == *(_DWORD *)compare_var )
  {
    v7 += 4;
    compare_var += 4;
    v9 = v10 < 4;
    v10 -= 4;
    if ( v9 )
      goto LABEL_7;
  }
}
return *v7 == *compare_var
    && (v10 == -3
     || v7[1] == compare_var[1] && (v10 == -2 || v7[2] == compare_var[2] && (v10 == -1 || v7[3] == compare_var[3])));
```

Figure 61: EvilFlagChecker_info2.

3. 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。

```python
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(s.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 sol
ve_z3.py
FLAG{jmp1ng_a1l_ar0und}
```

Figure 63: EvilFlagChecker_Flag.

[HW2] Trashcan

- FLAG{s1mpl3_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::`vftable';
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 )
41        v9 = v6;
42      (A->Vtable->method2)(A, *(v9 + count));     // 15b0
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。

```
class1 *__fastcall sub_1400015B0(class1 *a1, int a2)
{
  class2 *class2_obj; // rdi
  int a1_member2_member2; // ebp
  __int64 *a1_member2_member1; // rbx
  class1 *result; // rax

  class2_obj = a1->field_8;
  a1_member2_member2 = class2_obj->member2;
  a1_member2_member1 = class2_obj->member1;
  class2_obj->member2 = a1_member2_member2 + 1;
  if ( a1_member2_member1 )
  {
    if ( a2 >= *(a1_member2_member1 + 1) )
    {
      result = class1::class1(class2_obj, a1_member2_member1[2], a2, a1_member2_member2);
      a1_member2_member1[2] = result;
    }
    else
    {
      result = class1::class1(class2_obj, a1_member2_member1[1], a2, a1_member2_member2);
      a1_member2_member1[1] = result;
    }
    class2_obj->member1 = a1_member2_member1;
  }
  else
  {
    result = operator new(0x18ui64);
    result->field_0 = a1_member2_member2;
    result->field_4 = a2;
    result->field_8 = 0i64;
    result->field_10 = 0i64;
    class2_obj->member1 = result;
  }
  LOBYTE(result) = 1;
  return result;
}
```

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
xt:00000001400014D2                mov     [rbp+10h+var_30], 0Ch
xt:00000001400014D9                mov     [rbp+10h+var_2C], 15h
xt:00000001400014E0                mov     r9d, esi
xt:00000001400014E3                mov     rbx, [rbp+10h+Block+8]
xt:00000001400014E7                mov     r11, [rbp+10h+Block]
xt:00000001400014EB                sub     rbx, r11
xt:00000001400014EE                sar     rbx, 3
xt:00000001400014F2                test    rbx, rbx
xt:00000001400014F5                jz      short loc_140001535
xt:00000001400014F7                mov     rdx, r11
xt:00000001400014FA                mov     r8, rsi
xt:00000001400014FD                nop     dword ptr [rax]
xt:0000000140001500
xt:0000000140001500 loc_140001500:                         ; CODE
xt:0000000140001500                cmp     r9d, 16h
xt:0000000140001504                jl      short loc_14000150A
xt:0000000140001506                mov     ecx, esi
xt:0000000140001508                jmp     short loc_140001522
xt:000000014000150A ; ---------------------------------------------------------------------------
xt:000000014000150A
xt:000000014000150A loc_14000150A:                         ; CODE
xt:000000014000150A                mov     eax, dword ptr [rsp+r8
xt:000000014000150F                cmp     [rdx], eax
```

```
  HIDWORD(v12) = *v3;
  sub_140001F60(Block, 0i64, &v12);
  sub_140001320(v2, *(v3 + 8), Block);
  sub_140001320(v2, *(v3 + 16), Block);
  v1 = v20;
}
v4 = 1;
flag1[0] = _mm_load_si128(&xmmword_1400044D0);
flag1[1] = _mm_load_si128(&xmmword_1400044E0);
flag1[2] = _mm_load_si128(&xmmword_140004500);
flag1[3] = _mm_load_si128(&xmmword_1400044F0);
flag1[4] = _mm_load_si128(&xmmword_140004510);
v14 = 116;
v15 = 125;
flag2[0] = _mm_load_si128(&xmmword_1400044A0);
flag2[1] = _mm_load_si128(&xmmword_140004480);
flag2[2] = _mm_load_si128(&xmmword_140004490);
flag2[3] = _mm_load_si128(&xmmword_1400044C0);
flag2[4] = _mm_load_si128(&xmmword_1400044B0);
v17 = 12;
v18 = 21;
len = 0;
v6 = Block[0];
input_length = (Block[1] - Block[0]) >> 3;
```

Figure 66: Trashcan_info3.

4. Figure 67 為在 x64dbg 上設置 breakpoint。

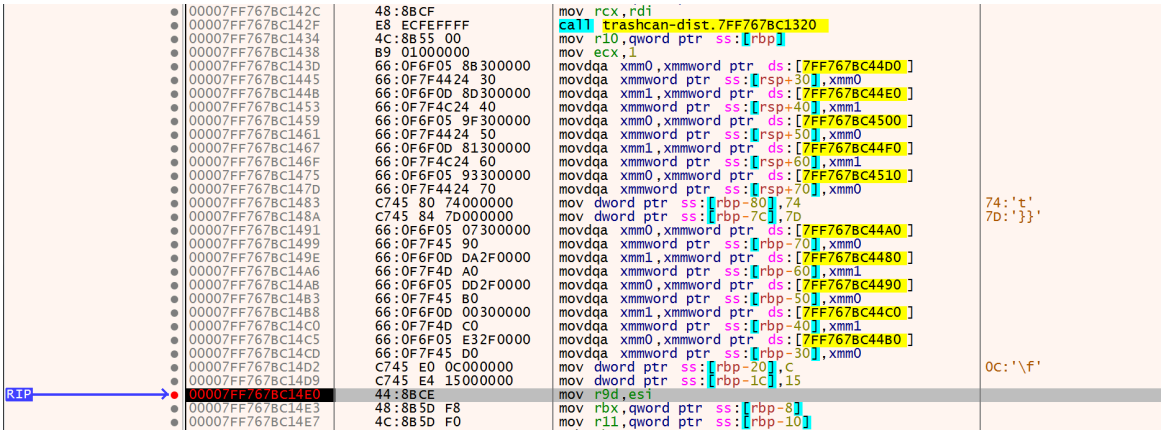


Figure 67: Trashcan_info4.

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

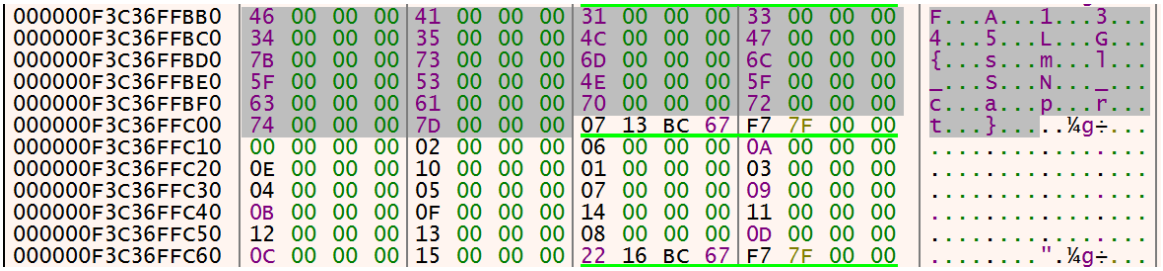


Figure 68: Trashcan_info5.

6. 看一下這些字元, Figure 69, 是不是超像答案的 OAO

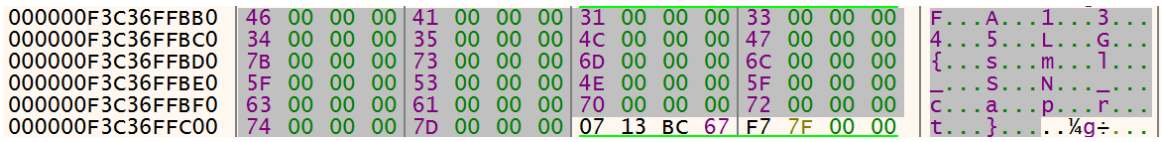


Figure 69: Trashcan_info6.

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(})。

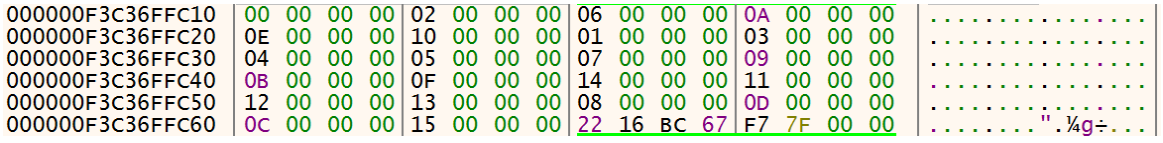


Figure 70: Trashcan_info7.