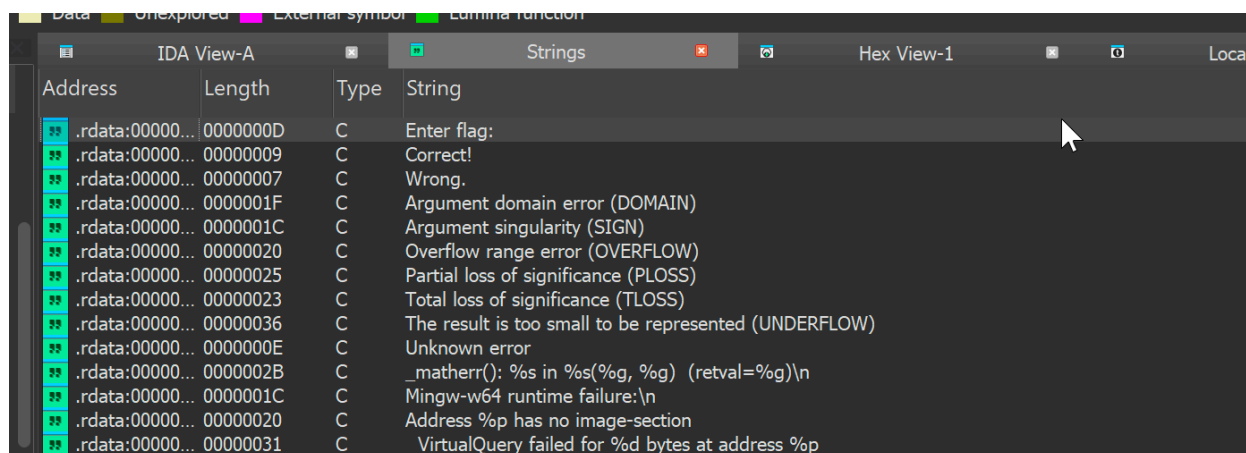


# Reverse tasks writeups

## reverse\_1.exe

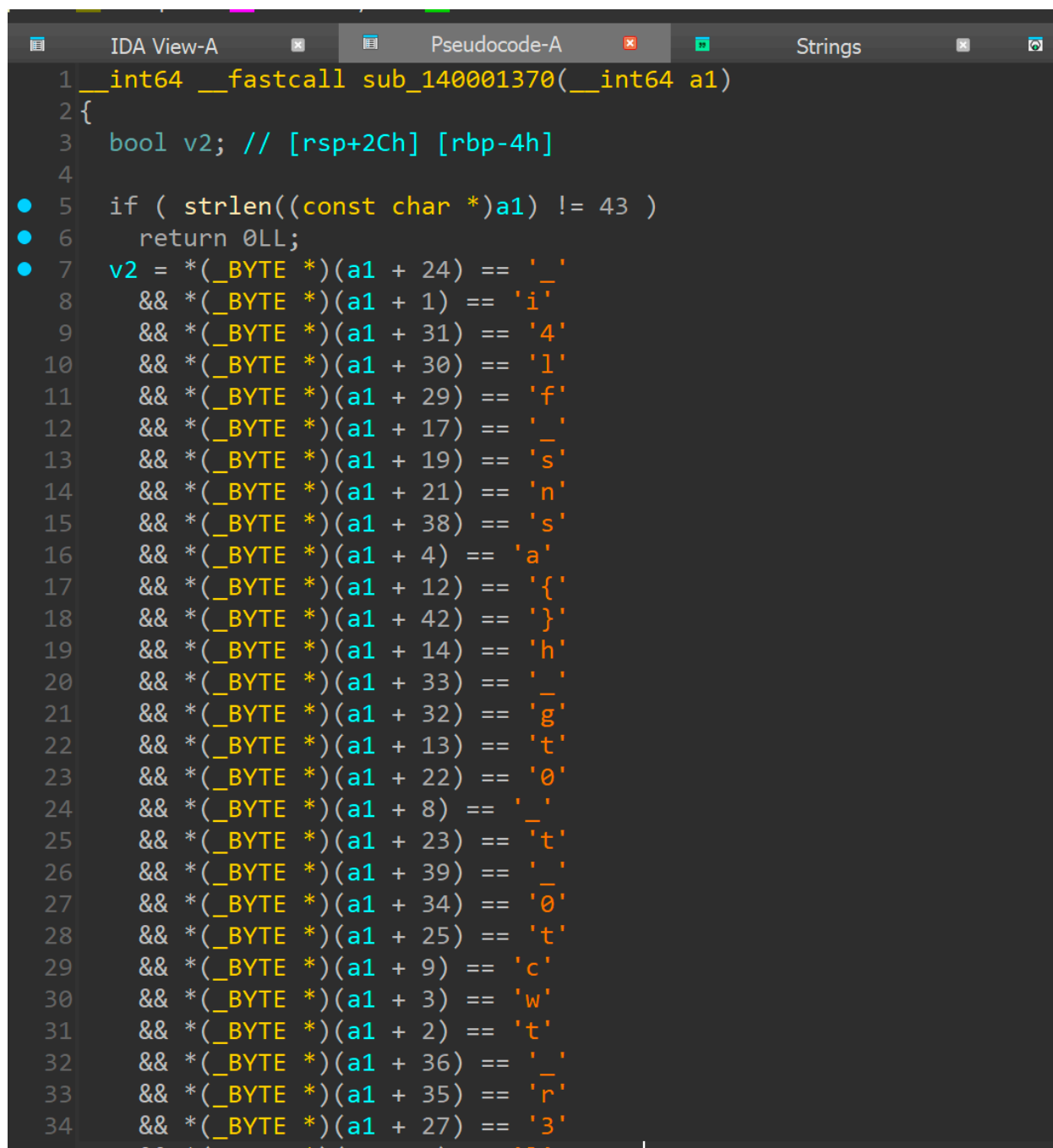
Среди строк находим место где вводится флаг



В декомпилированном виде видно функцию, которая используется для проверки флага

```
2 {
3     FILE *v3; // rax
4     char Buffer[128]; // [rsp+20h] [rbp-80h] BYREF
5
6     sub_14000185E(argc, argv, envp);
7     sub_140002460("Enter flag: ");
8     v3 = __acrt_iob_func(0);
9     fgets(Buffer, 128, v3);
10    Buffer[strcspn(Buffer, "\n")] = 0;
11    if ( (unsigned int)sub_140001370(Buffer) )
12        puts("Correct!");
13    else
14        puts("Wrong.");
15    return 0;
16 }
```

Внутри функции мы видим, что сначала сверяется длина вводимой строки, а потом выполняется посимвольная проверка ascii символов



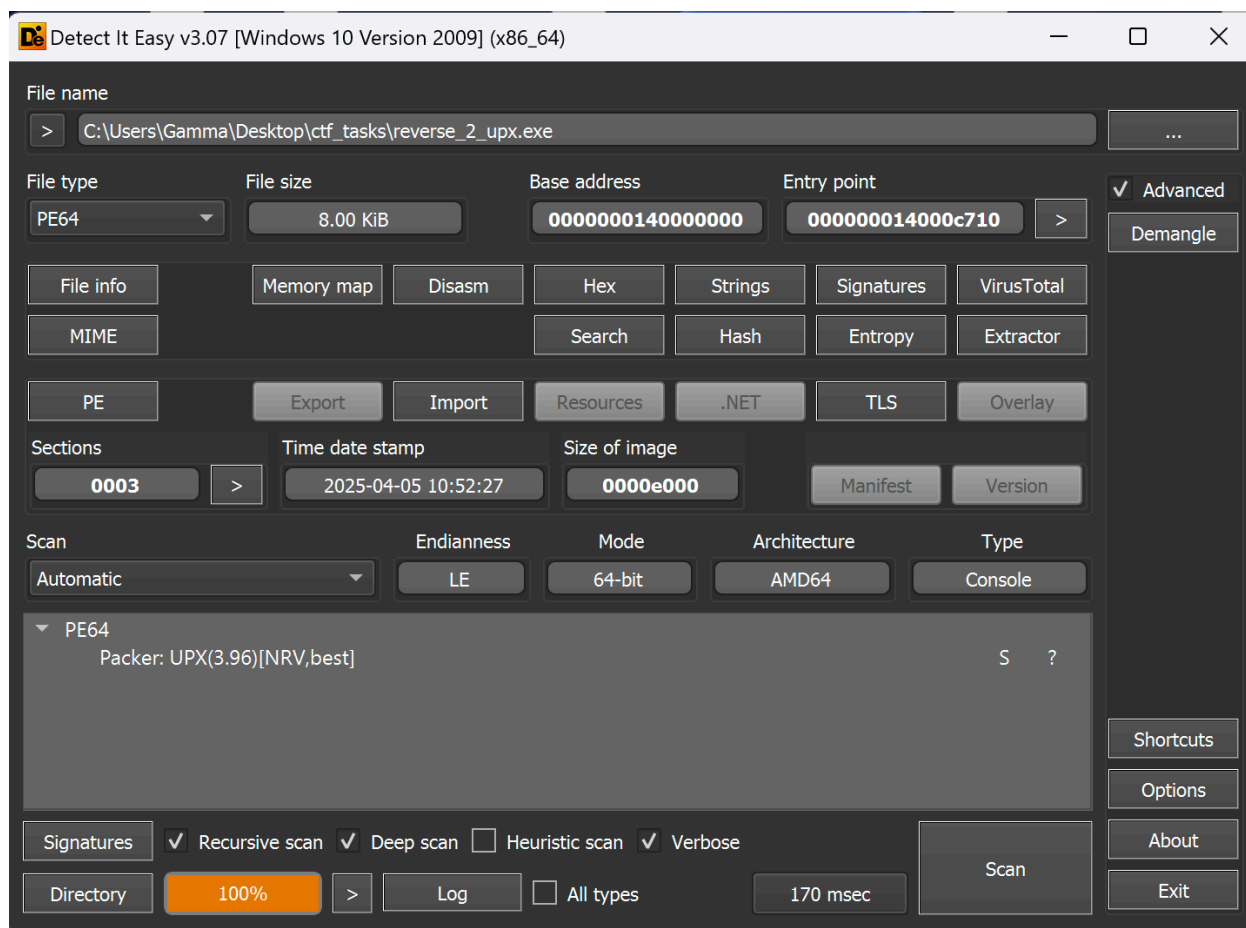
```
1 __int64 __fastcall sub_140001370(__int64 a1)
2 {
3     bool v2; // [rsp+2Ch] [rbp-4h]
4
5     if ( strlen((const char *)a1) != 43 )
6         return 0LL;
7     v2 = *(_BYTE *)(a1 + 24) == '_'
8         && *(_BYTE *)(a1 + 1) == 'i'
9         && *(_BYTE *)(a1 + 31) == '4'
10        && *(_BYTE *)(a1 + 30) == 'l'
11        && *(_BYTE *)(a1 + 29) == 'f'
12        && *(_BYTE *)(a1 + 17) == '_'
13        && *(_BYTE *)(a1 + 19) == 's'
14        && *(_BYTE *)(a1 + 21) == 'n'
15        && *(_BYTE *)(a1 + 38) == 's'
16        && *(_BYTE *)(a1 + 4) == 'a'
17        && *(_BYTE *)(a1 + 12) == '{'
18        && *(_BYTE *)(a1 + 42) == '}'
19        && *(_BYTE *)(a1 + 14) == 'h'
20        && *(_BYTE *)(a1 + 33) == '_'
21        && *(_BYTE *)(a1 + 32) == 'g'
22        && *(_BYTE *)(a1 + 13) == 't'
23        && *(_BYTE *)(a1 + 22) == '0'
24        && *(_BYTE *)(a1 + 8) == '_'
25        && *(_BYTE *)(a1 + 23) == 't'
26        && *(_BYTE *)(a1 + 39) == '_'
27        && *(_BYTE *)(a1 + 34) == '0'
28        && *(_BYTE *)(a1 + 25) == 't'
29        && *(_BYTE *)(a1 + 9) == 'c'
30        && *(_BYTE *)(a1 + 3) == 'w'
31        && *(_BYTE *)(a1 + 2) == 't'
32        && *(_BYTE *)(a1 + 36) == '_'
33        && *(_BYTE *)(a1 + 35) == 'r'
34        && *(_BYTE *)(a1 + 27) == '3'
```

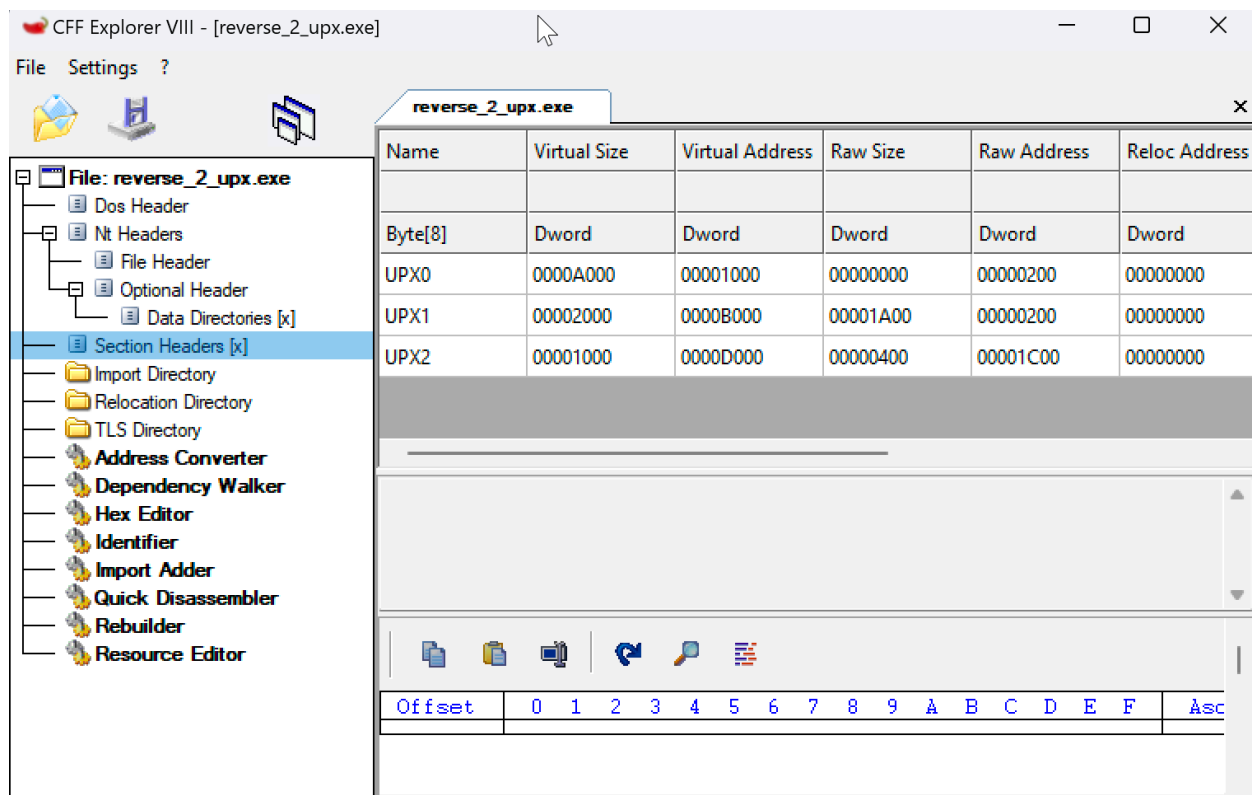
Собрав все символы воедино можно получить флаг -

`bitwalls_ctf{th1s_1s_n0t_th3_fl4g_0r_1s_1t}`

## reverse\_2.exe

Если открыть файл в DetectItEasy или в CFF Explorer, то мы увидим, что файл запакован с помощью upx





Попробуем распаковать upx:

```

Usage: upx [-123456789dlthVL] [-qvfk] [-o file] file..

Commands:
  -1      compress faster                -9      compress better
  -d      decompress                    -l      list compressed file
  -t      test compressed file          -V      display version number
  -h      give more help                -L      display software license

Options:
  -q      be quiet                      -v      be verbose
  -oFILE  write output to 'FILE'
  -f      force compression of suspicious files
  -k      keep backup files
file..   executables to (de)compress

Type 'upx --help' for more detailed help.

UPX comes with ABSOLUTELY NO WARRANTY; for details visit https://upx.github.io

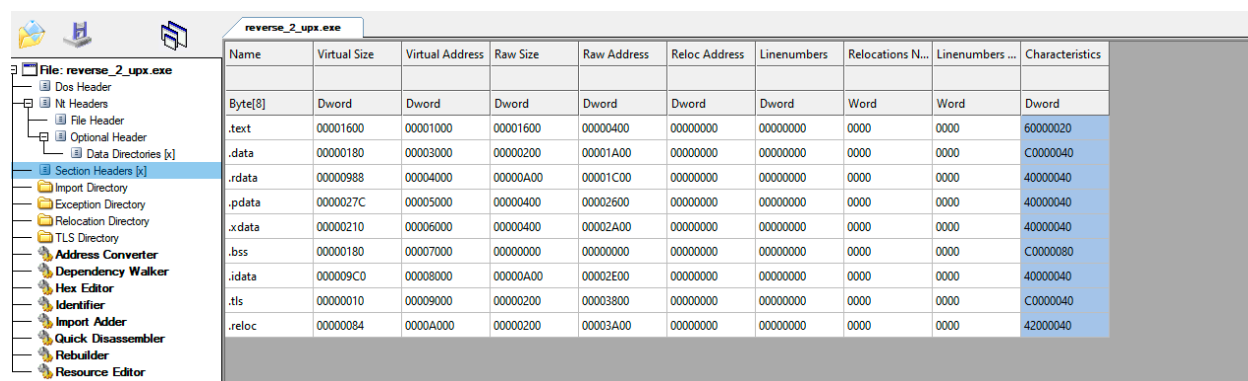
C:\Users\Gamma\Desktop\ctf_tasks>upx -d reverse_2_upx.exe
      Ultimate Packer for eXecutables
      Copyright (C) 1996 - 2023
UPX 4.0.2      Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 30th 2023

-----
File size      Ratio      Format      Name
-----
15360 <-      8192      53.33%     win64/pe     reverse_2_upx.exe

Unpacked 1 file.

```

Файл распаковался успешно, а значит можно продолжать реверс



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00001600	00001000	00001600	00000400	00000000	00000000	0000	0000	60000020
.data	00000180	00003000	00000200	00001A00	00000000	00000000	0000	0000	C0000040
.rdata	00000988	00004000	00000A00	00001C00	00000000	00000000	0000	0000	40000040
.pdata	0000027C	00005000	00000400	00002600	00000000	00000000	0000	0000	40000040
.xdata	00000210	00006000	00000400	00002A00	00000000	00000000	0000	0000	40000040
.bss	00000180	00007000	00000000	00000000	00000000	00000000	0000	0000	C0000080
.idata	000009C0	00008000	00000A00	00002E00	00000000	00000000	0000	0000	40000040
.tls	00000010	00009000	00000200	00003800	00000000	00000000	0000	0000	C0000040
.reloc	00000084	0000A000	00000200	00003A00	00000000	00000000	0000	0000	42000040

В функции ввода флага мы увидим вызов функции, в которую передаётся пока неизвестное нам значение

```
push    rbp
sub     rsp, 138h
lea     rbp, [rsp+80h]
call    sub_14000160E
lea     rax, Format          ; "Enter flag: "
mov     rcx, rax             ; Format
call    sub_140002210
mov     ecx, 0               ; Ix
mov     rax, cs: __imp__acrt_iob_func
call    rax ; __imp__acrt_iob_func
mov     rdx, rax
lea     rax, [rbp+0C0h+Buffer]
mov     r8, rdx              ; Stream
mov     edx, 80h             ; MaxCount
mov     rcx, rax             ; Buffer
call    fgets
lea     rax, [rbp+0C0h+Buffer]
lea     rdx, Control         ; "\n"
mov     rcx, rax             ; Str
call    strcspn
mov     [rbp+rax+0C0h+Buffer], 0
lea     rax, [rbp+0C0h+var_120]
mov     rdx, rax
lea     rax, aQysx88rn1hdbx1 ; "QysX88RNLhDbx1UkGLfLQnIHt8F+dg3g+hI6U/a"...
mov     rcx, rax
call    sub_1400013BA
mov     [rbp+0C0h+var_14], 0
jmp     loc_14000155A

aQysx88rn1hdbx1 db 'QysX88RNLhDbx1UkGLfLQnIHt8F+dg3g+hI6U/awRR0FtNd+JBbq+k8dE/aVR3MX+'
; DATA XREF: sub_140001451+69to
db 'Q==',0
```

Можно предположить что это какое-то декодирование, скорей всего base64 (предположить можно из того, что в функции используется алфавит base64)

```

1 _BYTE *__fastcall sub_1400013BA(char *a1, _BYTE *a2)
2 {
3     char *v2; // rax
4     _BYTE *v3; // rax
5     _BYTE *result; // rax
6     int v5; // [rsp+24h] [rbp-Ch]
7     int v6; // [rsp+28h] [rbp-8h]
8     int v7; // [rsp+2Ch] [rbp-4h]
9
10    v7 = 0;
11    v6 = -8;
12    while ( *a1 )
13    {
14        v2 = a1++;
15        v5 = sub_140001370(*v2);
16        if ( v5 != -1 )
17        {
18            v7 = (v7 << 6) + v5;
19            v6 += 6;
20            if ( v6 >= 0 )
21            {
22                v3 = a2++;
23                *v3 = v7 >> v6;
24                v6 -= 8;
25            }
26        }
27    }
28    result = a2;
29    *a2 = 0;
30    return result;
31 }

```

```

1 _int64 __fastcall sub_140001370(char a1)
2 {
3     char *v2; // [rsp+28h] [rbp-8h]
4
5     v2 = strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+", a1);
6     if ( v2 )
7         return (unsigned int)(v2 - "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+");
8     else
9         return 0xFFFFFFFF;
10 }

```

А потом после этого видим цикл, в котором сверяется введённое с значением v2 (которое передавалось в функцию декодирования base64, так что скорей всего это output\_buffer) + проводится операция XOR с каким-то значением байт

```

12  Buffer[3*5] = 0;
13  sub_1400013BA(aQysx88rnlhdbx1, v2);
14  for ( i = 0; i < strlen(Buffer); ++i )
15  {
16      if ( (byte_140003000[i % 5] ^ Buffer[i]) != (unsigned __int8)v2[i] )// compare input
17      {
18          puts("Wrong.");
19          return 0LL;
20      }
21  }

```

Для декодирования флага можно воспользоваться CyberChef

## 1. Достаём значение base64

```

.data:0000000140003020 aQysx88rnlhdbx1 db 'QysX88RNLhDbx1UkGLfLQnIht8F+dg3g+hI6U/aWRR0FtNd+JBbq+k8dE/aVR3MX+'
.data:0000000140003020 ; DATA XREF: sub_140001451+69↑o
.data:0000000140003061 db 'Q==',0
.data:0000000140003065 align 20h
.data:0000000140003080 off 140003080 dq offset unk_140004920 ; DATA XREF: sub_1400015A0;loc_1400015A4↑r

```

The screenshot shows the CyberChef interface with a 'Recipe' panel on the left and an 'Input' panel on the right. The recipe is 'From Base64' with the alphabet 'A-Za-z0-9+/' and 'Remove non-alphabet chars' checked. The input is 'QysX88RNLhDbx1UkGLfLQnIht8F+dg3g+hI6U/aWRR0FtNd+JBbq+k8dE/aVR3MX+Q=='. The output is 'C+ôÄM.Ô&U\$\$.ÉBr\$.Ä~v`àü\$.S8•E\$`x~\$`éü0\$`ö•Gs`û'.

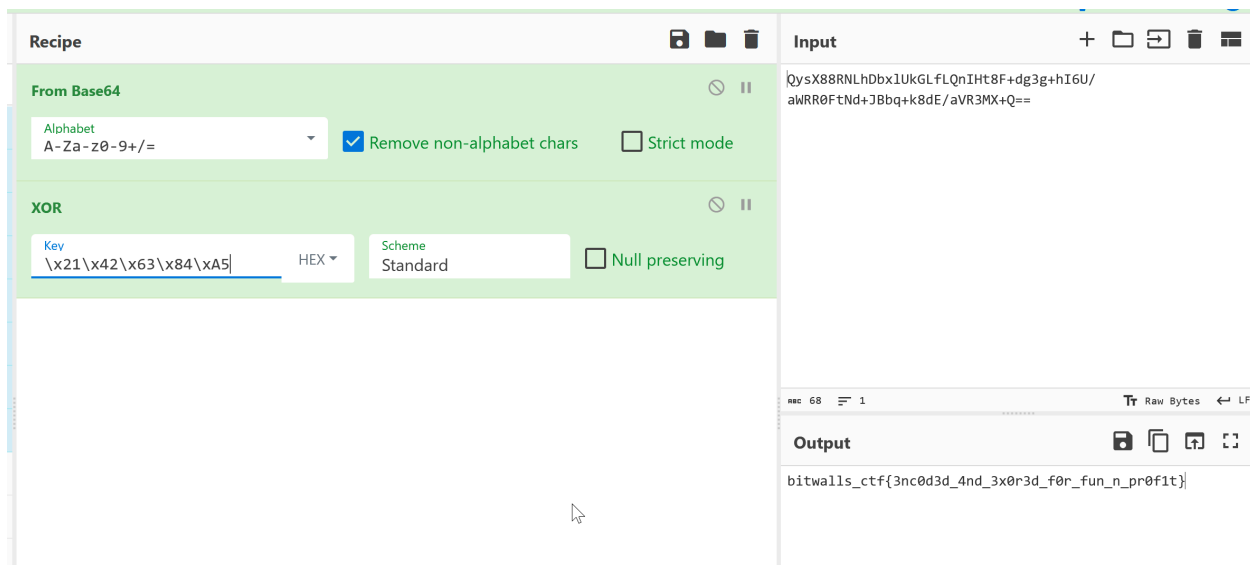
## 2. Достаём байты, которые используются для XOR (XOR ключ)

```

.data:0000000140003000 ; unsigned __int8 byte_140003000[32]
.data:0000000140003000 byte_140003000 db 21h, 42h, 63h, 84h, 0A5h, 1Bh dup(0)
.data:0000000140003000 ; DATA XREF: sub_140001451+C5↑o

```





**Флаг:** `bitwalls_ctf{3nc0d3d_4nd_3x0r3d_f0r_fun_n_pr0f1t}`

## reverse\_3.exe

Первое что мы видим в функции main - проверка на дебаггер - стандартный антиотладочный приём

```

1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     FILE *v4; // rax
4     char Buffer[128]; // [rsp+20h] [rbp-80h] BYREF
5
6     sub_1400017DE(argc, argv, envp);
7     if ( (unsigned int)sub_140001370() )
8     {
9         puts("Debugger detected.");
10        return 1;
11    }
12    else
13    {
14        sub_140001387();
15        sub_1400023E0("Enter flag: ");
16        v4 = __acrt_iob_func(0);
17        fgets(Buffer, 128, v4);
18        Buffer[strcspn(Buffer, "\n")] = 0;
19        if ( (unsigned int)sub_1400013D2(Buffer) )
20            puts("Correct!");
21        else
22            puts("Wrong.");
23        return 0;
24    }
25 }

```

```

IDA View-A
Pseudocode
1 BOOL sub_140001370()
2 {
3     return IsDebuggerPresent();
4 }

```

Позже перед принтом строки `Enter flag:` мы видим функцию, которая используя XOR с значением `0xAA` декодирует байты из `byte_140003000` и складывает их в значение `byte_140007040`

```

1 __int64 sub_140001387()
2 {
3     __int64 result; // rax
4     unsigned int i; // [rsp+Ch] [rbp-4h]
5
6     for ( i = 0; ; ++i )
7     {
8         result = i;
9         if ( i > 0x1B0 )
10             break;
11         byte_140007040[i] = byte_140003000[i] ^ 0xAA;
12     }
13     return result;
14 }

```

Позже судя по всему происходит проверка флага в функции `sub_1400013D2` , но там не всё так просто

```

22 {
23 LABEL_2:
24 while ( 1 )
25 {
26     v1 = v16++;
27     v2 = byte_140007040[v1];
28     if ( (unsigned int)v2 <= 6 )
29         break;
30     if ( v2 == 255 )
31         return 1LL;
32 }
33 }
34 while ( v2 <= 0 );
35 switch ( v2 )
36 {
37     case 1:
38         v10 = byte_140007040[v16];
39         v3 = v16 + 1;
40         v16 += 2;
41         dword_140007240[v10] = byte_140007040[v3];
42         goto LABEL_2;
43     case 2:
44         v11 = byte_140007040[v16];
45         v4 = v16 + 1;
46         v16 += 2;
47         if ( byte_140007040[v4] == dword_140007240[v11] )
48             goto LABEL_2;
49         result = 0LL;
50         break;
51     case 3:
52         v12 = byte_140007040[v16];
53         v6 = v16 + 1;
54         v16 += 2;
55         if ( byte_140007040[v6] == *(_BYTE *) (v12 + a1) )
56             goto LABEL_2;

```

Мы видим, что используется знакомое нам значение

`byte_140007040` , которое считывается и сверяется

Давайте для начала попробуем декодировать значение

`byte_140003000` , которое мы встречали ранее



```

00000000 04 00 00 01 01 62 02 00 62 04 00 01 01 01 69 02 |....b..b....i.|
00000010 00 69 04 00 02 01 01 74 02 00 74 04 00 03 01 01 |.i.....t..t....|
00000020 77 02 00 77 04 00 04 01 01 61 02 00 61 04 00 05 |w..w.....a..a...|
00000030 01 01 6c 02 00 6c 04 00 06 01 01 6c 02 00 6c 04 |..l..l.....l..l.|
00000040 00 07 01 01 73 02 00 73 04 00 08 01 01 5f 02 00 |....s..s....._|
00000050 5f 04 00 09 01 01 63 02 00 63 04 00 0a 01 01 74 |_.....c..c.....t|
00000060 02 00 74 04 00 0b 01 01 66 02 00 66 04 00 0c 01 |..t.....f..f....|
00000070 01 7b 02 00 7b 04 00 0d 01 01 64 02 00 64 04 00 |.{...{.....d..d..|
00000080 0e 01 01 33 02 00 33 04 00 0f 01 01 74 02 00 74 |...3..3.....t..t|
00000090 04 00 10 01 01 33 02 00 33 04 00 11 01 01 63 02 |.....3..3.....c.|
000000a0 00 63 04 00 12 01 01 74 02 00 74 04 00 13 01 01 |.c.....t..t.....|
000000b0 5f 02 00 5f 04 00 14 01 01 74 02 00 74 04 00 15 |_.._.....t..t...|
000000c0 01 01 68 02 00 68 04 00 16 01 01 33 02 00 33 04 |..h..h.....3..3.|
000000d0 00 17 01 01 5f 02 00 5f 04 00 18 01 01 64 02 00 |...._.._.....d..|
000000e0 64 04 00 19 01 01 62 02 00 62 04 00 1a 01 01 67 |d.....b..b.....g|
000000f0 02 00 67 04 00 1b 01 01 5f 02 00 5f 04 00 1c 01 |..g....._.._.....|
00000100 01 34 02 00 34 04 00 1d 01 01 6e 02 00 6e 04 00 |.4..4.....n..n..|
00000110 1e 01 01 64 02 00 64 04 00 1f 01 01 5f 02 00 5f |...d..d....._.._|
00000120 04 00 20 01 01 6d 02 00 6d 04 00 21 01 01 30 02 |.. ..m..m..!..0.|
00000130 00 30 04 00 22 01 01 64 02 00 64 04 00 23 01 01 |.0..".d..d..#..|
00000140 5f 02 00 5f 04 00 24 01 01 74 02 00 74 04 00 25 |_.._..$.t..t..%|
00000150 01 01 68 02 00 68 04 00 26 01 01 33 02 00 33 04 |..h..h..&..3..3.|
00000160 00 27 01 01 5f 02 00 5f 04 00 28 01 01 76 02 00 |.'.._.._..(..v..|
00000170 76 04 00 29 01 01 6d 02 00 6d 04 00 2a 01 01 5f |v..)..m..m..*.._|
00000180 02 00 5f 04 00 2b 01 01 66 02 00 66 04 00 2c 01 |.._..+..f..f...|
00000190 01 6c 02 00 6c 04 00 2d 01 01 30 02 00 30 04 00 |.l..l..-..0..0..|
000001a0 2e 01 01 77 02 00 77 04 00 2f 01 01 7d 02 00 7d |...w..w../..}..}|
000001b0 ff aa aa aa aa aa aa aa aa aa aa aa aa aa aa |ÿaaaaaaaaaaaaaaaa|

```

Можно заметить некоторую закономерность (

`04 00 01` , `01 01 ??` , `02 00 ??` ), выглядят как некие операнды, а значит в теории перед нами виртуальная машина, которая сверяет значения!

Если посмотреть на декомпилированный вид, то видно, что перед нами есть

`switch ( v2 )` и потом парсятся значения, пройдемся по кейсам

case 1:

`v10 = byte_140007040[v16];`

```
v3 = v16 + 1;  
v16 += 2;  
dword_140007240[v10] = byte_140007040[v3];  
goto LABEL_2;
```

Похоже, что здесь происходит некая операция присваивания, будем считать что это некий

`mov` , опираясь на hexdump, предположим, что это `vm_mov 1, <значение>`

```
case 2:  
v11 = byte_140007040[v16];  
v4 = v16 + 1;  
v16 += 2;  
if ( byte_140007040[v4] == dword_140007240[v11] )  
    goto LABEL_2;  
result = 0LL;  
break;
```

case2 очень похож на case1, но здесь уже происходит сравнение регистра и значения, то есть это можно сказать, что эта `vm_cmp <регистр>, <значение>`

```
case 3:  
v12 = byte_140007040[v16];  
v6 = v16 + 1;  
v16 += 2;  
if ( byte_140007040[v6] == *(_BYTE*)(v12 + a1) )  
    goto LABEL_2;  
result = 0LL;  
break;
```

В case3 мы видим, что происходит также сверка значения с тем, что было введено, нам бы это пригодилось, но в нашем дампе это не встречается, всё

равно обозначим это как `vm_cmp_input <введённое_значение>, <значение>`

```
case 4:
v13 = byte_140007040[v16];
v7 = v16 + 1;
v16 += 2;
dword_140007240[v13] = *(char *)(byte_140007040[v7] + a1);
goto LABEL_2;
```

case4 выглядит как присваивание введённого значения, обозначим как

`vm_mov_input <регистр>, <введённое_значение>`

```
case 5:
v14 = byte_140007040[v16];
v8 = v16 + 1;
v16 += 2;
dword_140007240[v14] ^= byte_140007040[v8];
goto LABEL_2;
```

case5 это хор регистра с значением, обозначим `vm_xor <регистр>, <значение>`

```
case 6:
v15 = byte_140007040[v16];
v9 = v16 + 1;
v16 += 2;
dword_140007240[v15] += byte_140007040[v9];
goto LABEL_2;
```

case6 это сложение, обозначим как `vm_add <регистр>, <значение>`

Проанализировав все кейсы, можно понять, что в одном из регистров сверяется значение посимвольно с флагом, можно написать код на python,



который поможет нам распарсить все значения и вытащить флаг

Начнём с расшифровки полезной нагрузки

```
enc = [0xAE, 0xAA, 0xAA, 0xAB, 0xAB, 0xC8, 0xA8, 0xAA, 0xC8, 0xAE, 0xA  
A, 0xAB, 0xAB, 0xAB, 0xC3, 0xA8, 0xAA, 0xC3, 0xAE, 0xAA, 0xA8, 0xAB, 0x  
AB, 0xDE, 0xA8, 0xAA, 0xDE, 0xAE, 0xAA, 0xA9, 0xAB, 0xAB, 0xDD, 0xA8, 0x  
AA, 0xDD, 0xAE, 0xAA, 0xAE, 0xAB, 0xAB, 0xCB, 0xA8, 0xAA, 0xCB, 0xAE, 0  
xAA, 0xAF, 0xAB, 0xAB, 0xC6, 0xA8, 0xAA, 0xC6, 0xAE, 0xAA, 0xAC, 0xAB,  
0xAB, 0xC6, 0xA8, 0xAA, 0xC6, 0xAE, 0xAA, 0xAD, 0xAB, 0xAB, 0xD9, 0xA8,  
0xAA, 0xD9, 0xAE, 0xAA, 0xA2, 0xAB, 0xAB, 0xF5, 0xA8, 0xAA, 0xF5, 0xAE,  
0xAA, 0xA3, 0xAB, 0xAB, 0xC9, 0xA8, 0xAA, 0xC9, 0xAE, 0xAA, 0xA0, 0xAB,  
0xAB, 0xDE, 0xA8, 0xAA, 0xDE, 0xAE, 0xAA, 0xA1, 0xAB, 0xAB, 0xCC, 0xA8,  
0xAA, 0xCC, 0xAE, 0xAA, 0xA6, 0xAB, 0xAB, 0xD1, 0xA8, 0xAA, 0xD1, 0xAE,  
0xAA, 0xA7, 0xAB, 0xAB, 0xCE, 0xA8, 0xAA, 0xCE, 0xAE, 0xAA, 0xA4, 0xAB,  
0xAB, 0x99, 0xA8, 0xAA, 0x99, 0xAE, 0xAA, 0xA5, 0xAB, 0xAB, 0xDE, 0xA8,  
0xAA, 0xDE, 0xAE, 0xAA, 0xBA, 0xAB, 0xAB, 0x99, 0xA8, 0xAA, 0x99, 0xAE,  
0xAA, 0xBB, 0xAB, 0xAB, 0xC9, 0xA8, 0xAA, 0xC9, 0xAE, 0xAA, 0xB8, 0xAB,  
0xAB, 0xDE, 0xA8, 0xAA, 0xDE, 0xAE, 0xAA, 0xB9, 0xAB, 0xAB, 0xF5, 0xA8,  
0xAA, 0xF5, 0xAE, 0xAA, 0xBE, 0xAB, 0xAB, 0xDE, 0xA8, 0xAA, 0xDE, 0xAE,  
0xAA, 0xBF, 0xAB, 0xAB, 0xC2, 0xA8, 0xAA, 0xC2, 0xAE, 0xAA, 0xBC, 0xAB,  
0xAB, 0x99, 0xA8, 0xAA, 0x99, 0xAE, 0xAA, 0xBD, 0xAB, 0xAB, 0xF5, 0xA8,  
0xAA, 0xF5, 0xAE, 0xAA, 0xB2, 0xAB, 0xAB, 0xCE, 0xA8, 0xAA, 0xCE, 0xAE,  
0xAA, 0xB3, 0xAB, 0xAB, 0xC8, 0xA8, 0xAA, 0xC8, 0xAE, 0xAA, 0xB0, 0xAB,  
0xAB, 0xCD, 0xA8, 0xAA, 0xCD, 0xAE, 0xAA, 0xB1, 0xAB, 0xAB, 0xF5, 0xA8,  
0xAA, 0xF5, 0xAE, 0xAA, 0xB6, 0xAB, 0xAB, 0x9E, 0xA8, 0xAA, 0x9E, 0xAE,  
0xAA, 0xB7, 0xAB, 0xAB, 0xC4, 0xA8, 0xAA, 0xC4, 0xAE, 0xAA, 0xB4, 0xAB,  
0xAB, 0xCE, 0xA8, 0xAA, 0xCE, 0xAE, 0xAA, 0xB5, 0xAB, 0xAB, 0xF5, 0xA8,  
0xAA, 0xF5, 0xAE, 0xAA, 0x8A, 0xAB, 0xAB, 0xC7, 0xA8, 0xAA, 0xC7, 0xAE,  
0xAA, 0x8B, 0xAB, 0xAB, 0x9A, 0xA8, 0xAA, 0x9A, 0xAE, 0xAA, 0x88, 0xAB,  
0xAB, 0xCE, 0xA8, 0xAA, 0xCE, 0xAE, 0xAA, 0x89, 0xAB, 0xAB, 0xF5, 0xA8,  
0xAA, 0xF5, 0xAE, 0xAA, 0x8E, 0xAB, 0xAB, 0xDE, 0xA8, 0xAA, 0xDE, 0xAE,  
0xAA, 0x8F, 0xAB, 0xAB, 0xC2, 0xA8, 0xAA, 0xC2, 0xAE, 0xAA, 0x8C, 0xAB,  
0xAB, 0x99, 0xA8, 0xAA, 0x99, 0xAE, 0xAA, 0x8D, 0xAB, 0xAB, 0xF5, 0xA8,
```

```
0xAA, 0xF5, 0xAE, 0xAA, 0x82, 0xAB, 0xAB, 0xDC, 0xA8, 0xAA, 0xDC, 0xAE,
0xAA, 0x83, 0xAB, 0xAB, 0xC7, 0xA8, 0xAA, 0xC7, 0xAE, 0xAA, 0x80, 0xAB,
0xAB, 0xF5, 0xA8, 0xAA, 0xF5, 0xAE, 0xAA, 0x81, 0xAB, 0xAB, 0xCC, 0xA8,
0xAA, 0xCC, 0xAE, 0xAA, 0x86, 0xAB, 0xAB, 0xC6, 0xA8, 0xAA, 0xC6, 0xAE,
0xAA, 0x87, 0xAB, 0xAB, 0x9A, 0xA8, 0xAA, 0x9A, 0xAE, 0xAA, 0x84, 0xAB,
0xAB, 0xDD, 0xA8, 0xAA, 0xDD, 0xAE, 0xAA, 0x85, 0xAB, 0xAB, 0xD7, 0xA8,
0xAA, 0xD7, 0x55]
program = [b ^ 0xAA for b in enc]
```

Проанализировав дампы, можем увидеть, что операций

`vm_xor` и `vm_add` не встречается, а больше всего нам пригодится `vm_cmp`

Также можно заметить, что в конце hexdump есть байт

`0xFF` и также в цикле `while` есть проверка этого значения:

```
if ( v2 == 0xFF )
    return 1LL;
}
```

Напишем код, который распарсит нашу виртуальную машину:

```
i = 0
flag = ""
while i < len(program):
    op = program[i]
    if op == 0x01:
        # print("vm_mov", program[i+1], program[i+2])
        i += 3
    elif op == 0x02:
        print("vm_cmp", program[i+1], chr(program[i+2]))
        flag += chr(program[i+2])
        i += 3
```

```
elif op == 0x03:
    # print("vm_cmp_input", program[i+1], chr(program[i+2]))
    i += 3
elif op == 0x04:
    # print("vm_mov_input", program[i+1], program[i+2])
    i += 3
elif op == 0x05:
    # print("vm_xor", program[i+1], program[i+2])
    i += 3
elif op == 0x06:
    # print("vm_add", program[i+1], program[i+2])
    i += 3
elif op == 0xFF:
    print("HALT")
    break
else:
    print("Unknown:", hex(op))
    break
print(flag)
```

```

~ /Downloads/ctf_reverse_tasks python3 reverse_3_get_flag.py
vm_cmp 0 b
vm_cmp 0 i
vm_cmp 0 t
vm_cmp 0 w
vm_cmp 0 a
vm_cmp 0 l
vm_cmp 0 l
vm_cmp 0 s
vm_cmp 0 _
vm_cmp 0 c
vm_cmp 0 t
vm_cmp 0 f
vm_cmp 0 {
vm_cmp 0 d
vm_cmp 0 3
vm_cmp 0 t
vm_cmp 0 3
vm_cmp 0 c
vm_cmp 0 t
vm_cmp 0 _
vm_cmp 0 t
vm_cmp 0 h
vm_cmp 0 3
vm_cmp 0 _
vm_cmp 0 d
vm_cmp 0 b
vm_cmp 0 g
vm_cmp 0 _
vm_cmp 0 4
vm_cmp 0 n
vm_cmp 0 d
vm_cmp 0 _
vm_cmp 0 m
vm_cmp 0 0
vm_cmp 0 d
vm_cmp 0 _
vm_cmp 0 t
vm_cmp 0 h
vm_cmp 0 3
vm_cmp 0 _
vm_cmp 0 v
vm_cmp 0 m
vm_cmp 0 _
vm_cmp 0 f
vm_cmp 0 l
vm_cmp 0 0
vm_cmp 0 w
vm_cmp 0 }
HALT
bitwalls_ctf{d3t3ct_th3_dbg_4nd_m0d_th3_vm_fl0w}
~ /Downloads/ctf_reverse_tasks

```

**После выполнения кода мы получим флаг -**

```
bitwalls_ctf{d3t3ct_th3_dbg_4nd_m0d_th3_vm_fl0w}
```