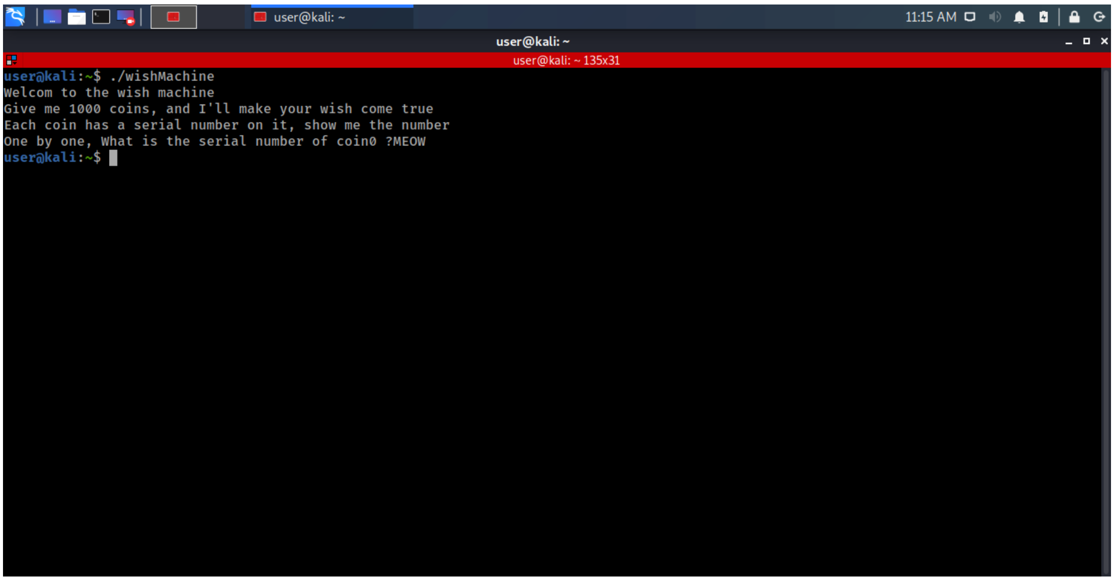
[HW 0x08] Writeup

資工四 B06902031 何承勳 wxrdnx

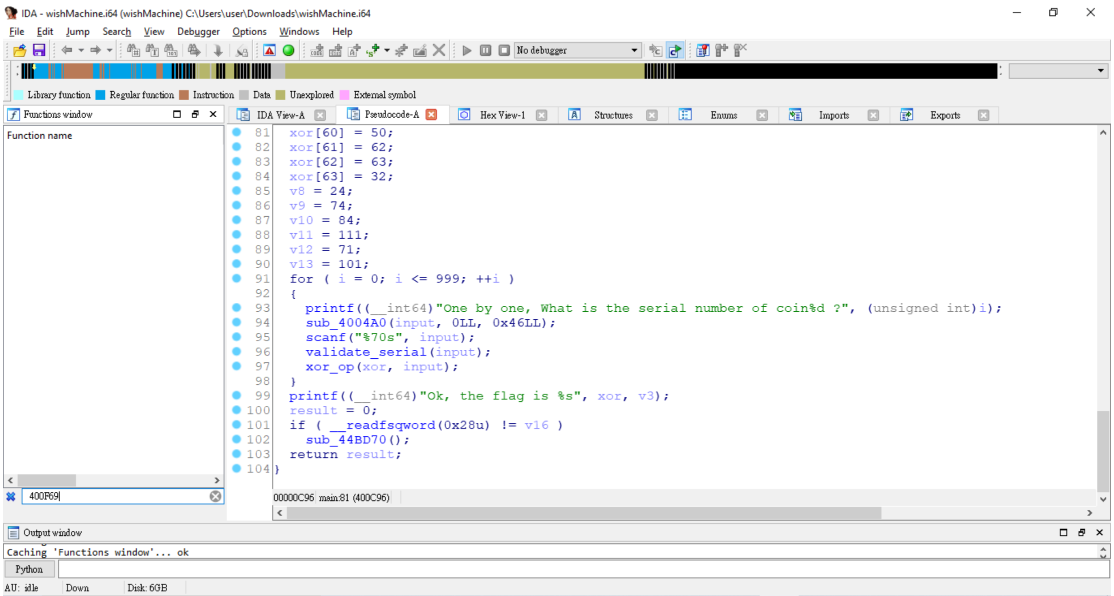
WishMachine

Description:

First run the program in Linux environment:



Let's examine this binary with IDA Pro:



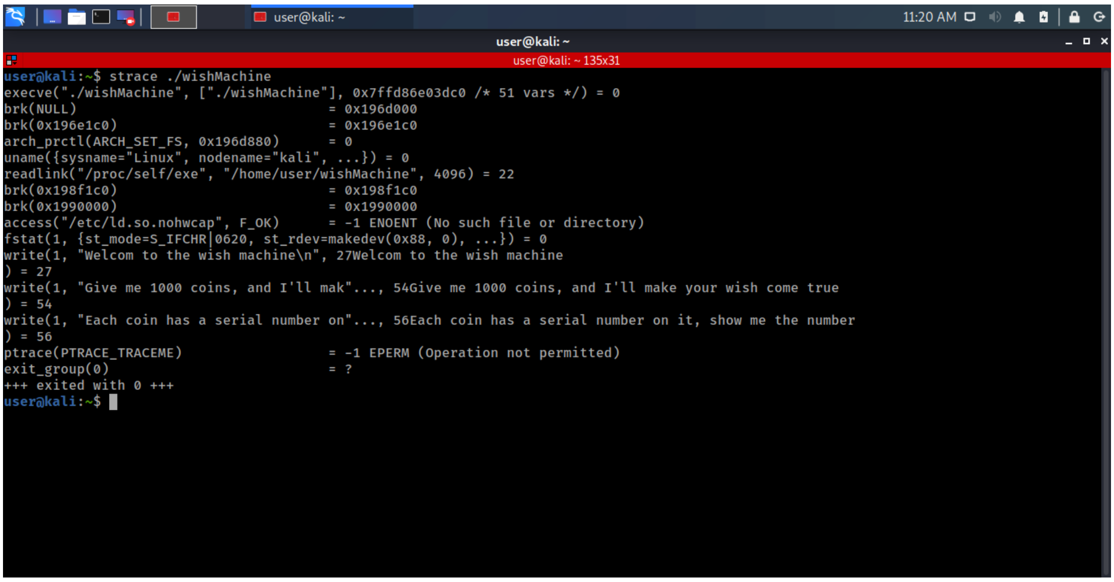
We can see that the program will ask us for 1000 serials, and validate them one by one. The flag is the result of xoring these 1000 serials.

Anti-Anti-Debugging

If we examine this process with GDB, it will exit immediately. It seems that there is an anti-debugging mechanism hidden somewhere in the code. The most common way of anti-debugging in linux is using ptrace ([reference]):

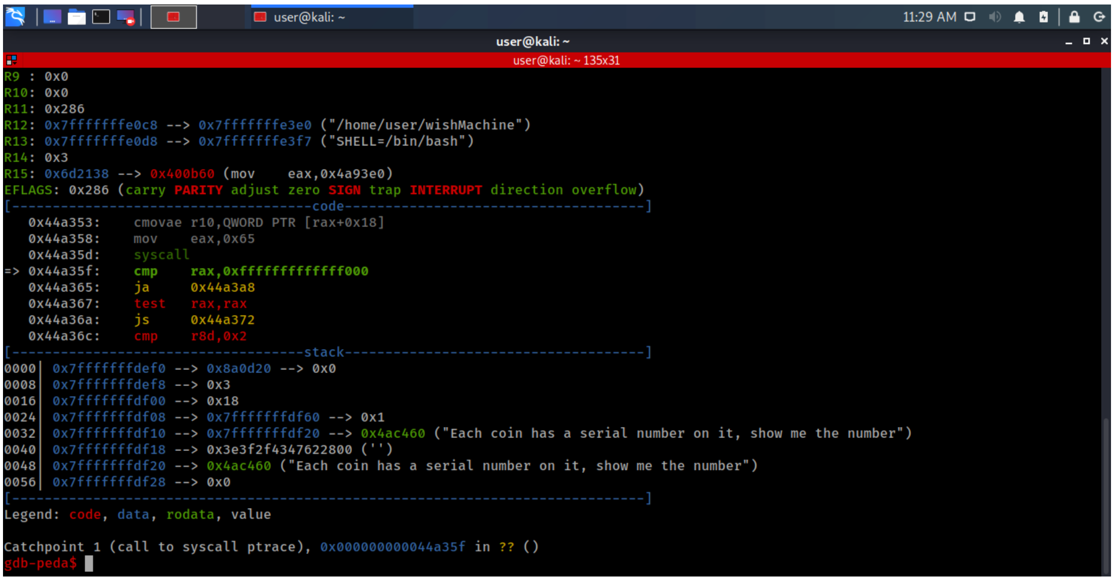
```
#include <sys/ptrace.h>
#include <stdio.h>

int main()
{
        if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0 ) {
                printf("Debugger Found.\n");
                exit(0);
        }
        printf("No debugger, continuing\n");
        return 0;
}
```

We now search for ptrace system call in the binary:



The program calls ptrace(PTRACE_TRACEME), and the return value is -1, which leads to exit_group(0). Indeed, anti-debugging mechanism is included in this binary.

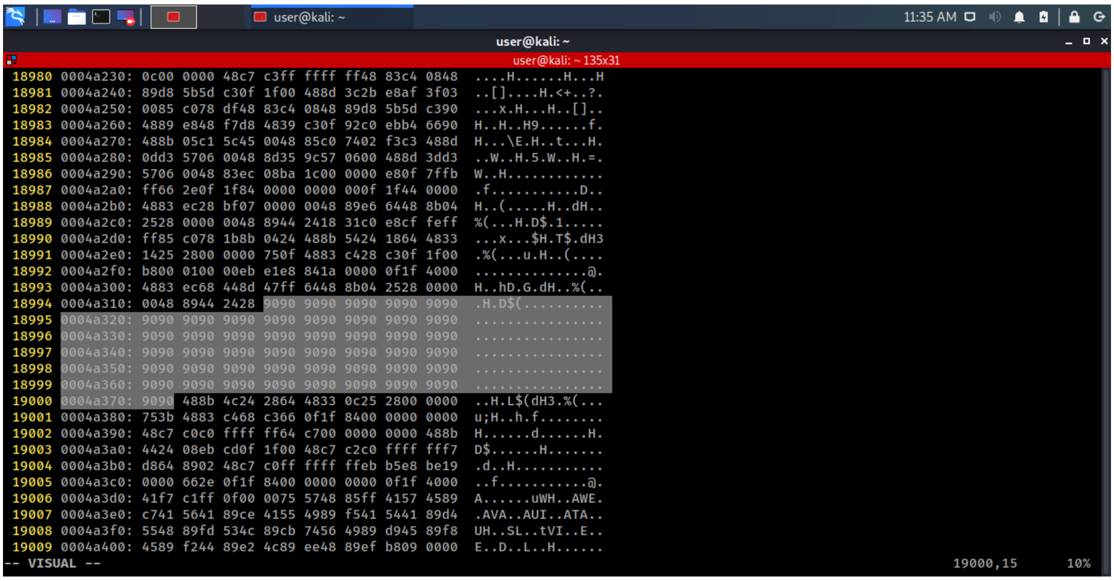
We can use gdb command `catch syscall 101` to catch ptrace system call:



Next, we analyze the function that contains 0x44a35f, which is 0x44a300 :

```
signed __int64 __fastcall sub_44A300(__int64 a1, __int64 a2, __int64 a3, __int64 a4)
{
  signed __int64 result; // rax
  unsigned __int64 v5; // rcx
  unsigned __int64 v6; // rt1
  signed __int64 v7; // [rsp+8h] [rbp-60h]
  unsigned __int64 v8; // [rsp+28h] [rbp-40h]
  __int64 v9; // [rsp+38h] [rbp-30h]
  __int64 v10; // [rsp+40h] [rbp-28h]
  __int64 v11; // [rsp+48h] [rbp-20h]

  v8 = __readfsqword(0x28u);
  v9 = a2;
  v10 = a3;
  v11 = a4;
  a2 = (unsigned int)a2;
  result = 101LL;
  __asm { syscall; LINUX - sys_ptrace }
  if ( (unsigned int)(a1 - 1) <= 2 )
  {
    __writefsdword(0xFFFFFFC0, 0);
    result = v7;
  }
  v6 = __readfsqword(0x28u);
```

In order to conquer anti-debugging, let's patch this function with nops using xxd:



Finally, we can debug this binary normally.

Validation Check

In order to find the 1000 valid serials, we must analyze the validation function in the binary. The validation function is at 0x400e0a:

```
__int64 __fastcall validate(__int64 a1)
{
  __int64 result; // rax
  signed int i; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; i <= 69; i += dword_8A2118 )
  {
    qword_8A2108 = a1;
    dword_8A2114 = *((_DWORD *)&unk_6D5114 + 10 * dword_8A1070);
    dword_8A2118 = *((_DWORD *)&unk_6D5118 + 10 * dword_8A1070);
    dword_8A211C = *((_DWORD *)&unk_6D511C + 10 * dword_8A1070);
    dword_8A2120 = dword_6D5120[10 * dword_8A1070];
    dword_8A2110 = *((_DWORD *)&unk_6D5110 + 10 * dword_8A1070);
    qword_8A2100 = *((_QWORD *)&unk_6D5100 + 5 * dword_8A1070) + dword_8A2110;
    ((void (*)(void))qword_8A2100)();
    ++dword_8A1070;
    result = (unsigned int)dword_8A2118;
  }
  return result;
}
```

The function first retrieves structures at 0x6d5100. The structure contains several fields listed below:

1. Validation function base pointer
2. Validation function offset (function address = base pointer + offset)
3. The character index to check.
4. The number of characters to check.
5. Validation value.

The function will calculate a validation function, which will perform some calculations on some input string characters. If the result matches the validation value, the validation function will return gracefully. Otherwise, the process will exit.

There are five different validation functions. Each of them are listed below:

1. 0x400fbe: check if

```
value = 135 * char
```

```
__int64 sub_400FBE()
{
  __int64 result; // rax
  int i; // [rsp+Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)dword_8A2118;
    if ( i >= dword_8A2118 )
      break;
    if ( *((_DWORD *)&qword_8A2108 + i + 4LL + 1) != 135 * *(char *)(qword_8A2108 + dword_8A2114 + i) )
      sub_40F130(0LL, (unsigned int)dword_8A2114);
  }
  return result;
}
```

2. 0x40102d: check if

```
value = 11 * floor((char + 1) / 2) + 2 * floor(char / 2)
```

```
__int64 __fastcall sub_40102D(__int64 a1, __int64 a2)
{
  __int64 result; // rax
  int v3; // [rsp+4h] [rbp-Ch]
  int i; // [rsp+8h] [rbp-8h]
  signed int j; // [rsp+Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)dword_8A2118;
    if ( i >= dword_8A2118 )
      break;
    v3 = 0;
    for ( j = 0; j < *(char *)(qword_8A2108 + dword_8A2114 + i); ++j )
    {
      if ( j & 1 )
        v3 += 2;
      else
        v3 += 11;
    }
    if ( v3 != *((_DWORD *)&qword_8A2108 + i + 4LL + 1) )
      sub_40F130(0LL, a2);
  }
  return result;
}
```

3. 0x4011d6: check if

```
value = fibonacci[char]
```

```
  int v3; // [rsp+Ch] [rbp-14h]
  int i; // [rsp+10h] [rbp-10h]
  int v5; // [rsp+14h] [rbp-Ch]
  signed int v6; // [rsp+18h] [rbp-8h]
  signed int j; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)dword_8A2118;
    if ( i >= dword_8A2118 )
      break;
    v5 = 0;
    v6 = 1;
    for ( j = 0; j < *(char *)(qword_8A2108 + dword_8A2114 + i); ++j )
    {
      v3 = v5 + v6;
      v5 = v6;
      v6 = v3;
    }
    if ( v3 != *((_DWORD *)&qword_8A2108 + i + 4LL + 1) )
      sub_40F130(0LL, a2);
  }
  return result;
}
```

4. 0x4010c8: check if

```
value = char ^ 0x52756279
```

```
sub_4010C8()

4 result; // rax
 // [rsp+Ch] [rbp-4h]

i = 0; ; ++i )

lt = (unsigned int)dword_8A2118;
 i >= dword_8A2118 )
eak;
 *((_DWORD *)&qword_8A2108 + i + 4LL + 1) != (*(char *)(qword_8A2108 + dword_8A2114 + i) ^ 0x52756279)
b_40F130(0LL, (unsigned int)dword_8A2114);

 result;
```

5. 0x401138: check if

```
value = -88035316 - 30600 * floor((char + 1) / 2) - 120 *
floor(char / 2)
```

```
{
  __int64 result; // rax
  signed int v3; // [rsp+4h] [rbp-Ch]
  int i; // [rsp+8h] [rbp-8h]
  signed int j; // [rsp+Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)dword_8A2118;
    if ( i >= dword_8A2118 )
      break;
    v3 = -88035316;
    for ( j = 0; j < *(char *)(qword_8A2108 + dword_8A2114 + i); ++j )
    {
      if ( j & 1 )
        v3 -= 120;
      else
        v3 -= 30600;
    }
    if ( v3 != *((_DWORD *)&qword_8A2108 + i + 4LL + 1) )
      sub_40F130(0LL, a2);
  }
  return result;
}
```

Since each of them is a 1-1 function, we can calculate the serial by extracting the structure in the binary and performing its reverse function on the character.

The calculation detail can be found in `wishMachine.py` (usage: `python3 wishMachine.py`).

Eventually, after sending the 1000 serials, the binary will calculate the flag and print it out to us.

Flag:

FLAG{7hes3_func710n_ptrs_g1v3_m3_l0t_0o0of_f4n_I_w4n7_m00r3_11!!l1!|!}

Curse

Description:

This binary is a windows 32-bit Portable Executable. If we throw this executable in IDA.

For main function:

```
  if ( !alloc_addr )
  {
    if ( v3 )
    {
      alloc_addr = (char *)VirtualAlloc(0, *((_DWORD *)shift_lpAddress + 20), 0x3000u, 0x40u);
      if ( !alloc_addr )
        return 0;
    }
  }
  *((_DWORD *)shift_lpAddress + 13) = alloc_addr;
  memcpy(alloc_addr, structure, *((_DWORD *)shift_lpAddress + 21));
  for ( i = 0; i < *((unsigned __int16 *)shift_lpAddress + 3); ++i )
    memcpy(
      &alloc_addr[*(_DWORD *)&shift_lpAddress[40 * i + 260]],
      &structure[*(_DWORD *)&shift_lpAddress[40 * i + 268]],
      *(_DWORD *)&shift_lpAddress[40 * i + 264]);
  getAddresses((int)alloc_addr);
  if ( lpAddress != alloc_addr )
  {
    v1 = *((_DWORD *)shift_lpAddress + 20);
    sub_4015C4((int)alloc_addr, 0, (int)lpAddress, 0, (int)alloc_addr);
  }
  return ((int (*)(void))(alloc_addr + 51024))();
}
```

For getAddress:

```
      {
        v7 = (FARPROC *)(v15 + alloc_addr + v8);
        v6 = (FARPROC *)(j + alloc_addr + v16);
        if ( *(_DWORD *)(*v10 + alloc_addr) < 0 )
        {
          v2 = (const CHAR *)(unsigned __int16)*v6;
          v3 = LoadLibraryA(lpLibFileName);
          *v7 = GetProcAddress(v3, v2);
        }
        if ( !*v7 )
          break;
        if ( *v7 == *v6 )
        {
          v5 = (int)*v6 + alloc_addr;
          if ( (signed int)*v6 < 0 )
            return 0;
          v4 = LoadLibraryA(lpLibFileName);
          *v7 = GetProcAddress(v4, (LPCSTR)(v5 + 2));
        }
        v15 += 4;
      }
    }
    return 1;
}
```

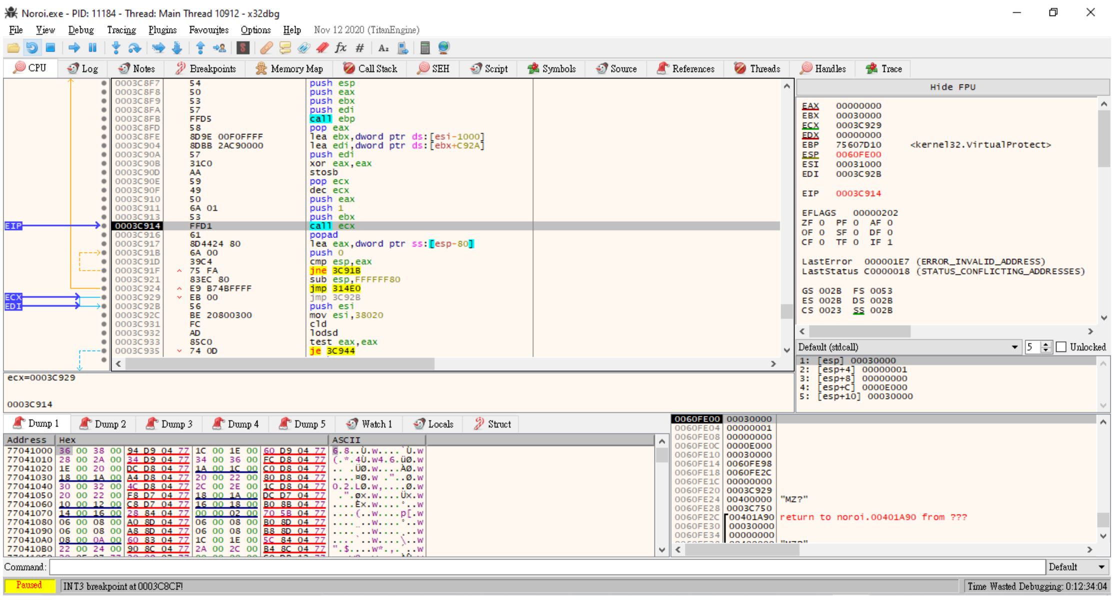
For sub_4015C4:

```
  unsigned int v11; // [esp+44h] [ebp-14h]
  unsigned int j; // [esp+4Ch] [ebp-Ch]
  int v13; // [esp+50h] [ebp-8h]
  unsigned int i; // [esp+54h] [ebp-4h]

  v5 = sub_40155C(a5, 5u);
  v11 = *((_DWORD *)v5 + 1);
  v10 = *(_DWORD *)v5;
  for ( i = 0; i < v11; i += v9[1] )
  {
    v9 = (int *)(i + v10 + a5);
    if ( !*v9 || !v9[1] )
      break;
    v8 = (unsigned int)(v9[1] - 8) >> 1;
    v7 = *v9;
    v13 = (int)(v9 + 2);
    for ( j = 0; j < v8 && v13 && *(_BYTE *)(v13 + 1) >> 4; ++j )
    {
      *(_DWORD *)(a5 + v7 + (*(_WORD *)v13 & 0xFFF)) = a1 + *(_DWORD *)(a5 + v7 + (*(_WORD *)v13 & 0xFF
      v13 += 2;
    }
  }
  return i != 0;
}
```

We can see that the main function calls VirtualAlloc, and calls getAddresses which invokes lots of LoadLibraryA and GetProcAddress. Then, the main function calls sub_4015C4 and performs an intricate calculation. Finally, the program ends… ?

Solution:

According to my experience, the program seems to load some packed data into a virtual allocated memory, resolve the necessary functions, and performs unpacking. So, let's dynamically debug the program using x64dbg:



After several trial and error, I finally figured out the real entry point of the main function. Which is 0x319D0. Dump the codes from 0x319D0 to unpack.exe and analyze it using IDA:

```
  unsigned int i; // [esp+9Ch] [ebp-4h]

  sub_1800();
  fgets(input, 128, MEMORY[0x37180]);
  input[strlen(input) - 1] = '\0';
  for ( i = 0; ; ++i )
  {
    len = strlen(input);
    if ( len <= i )
      break;
    input[i] = (unsigned int)encode((char *)input[i]);
  }
  flag_len = strlen(209696);
  if ( strncmp(209696, input, flag_len) )
  {
    sub_2764(212992);
    sub_2764(213029);
    sub_2764(213052);
    sub_2764(213069);
    sub_2764(213085);
    sub_2764(213102);
    sub_2764(213130);
  }
  return 0;
}
```

Here, we can see that the main function encodes your input and compare it with a fixed string. If the two string matches, then the program performs sub_2764. So basically the input string might be our final flag.

Let's examine the encryption part. The encryption code is as follows:

```
char *__cdecl encode(char *input)
{
  char *result; // eax
  signed int i; // [esp+10h] [ebp-4h]

  result = input;
  for ( i = 0; i <= 0x2F6; i += 2 )
  {
    result = (char *)*(unsigned __int8 *)(i + 0x33020);
    if ( (_BYTE)input == (_BYTE)result )
      return (char *)*(unsigned __int8 *)(i + 0x33021);
  }
  return result;
}
```

The encryption is basically a dictionary lookup ☺

So, we can get the flag by building a reverse dictionary and decoding the fix string with it. The script curse.py contains the detailed implementation.

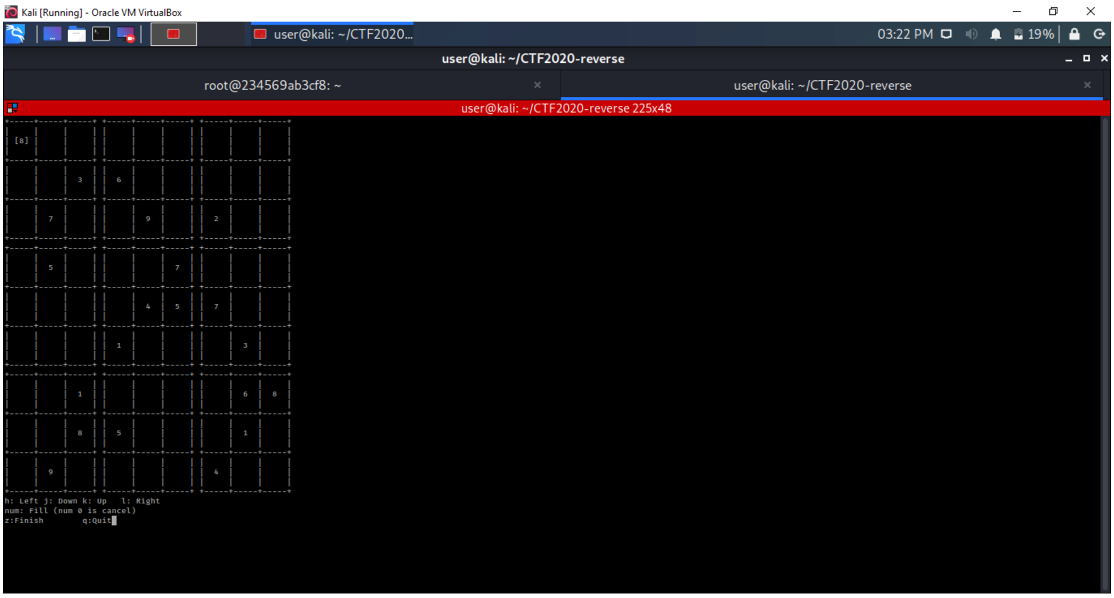
Flag:

FLAG{yes_this_is_your_homework_with_upx_and_no_pe_header_and_this_is_the_flag}

SecureContainProtect

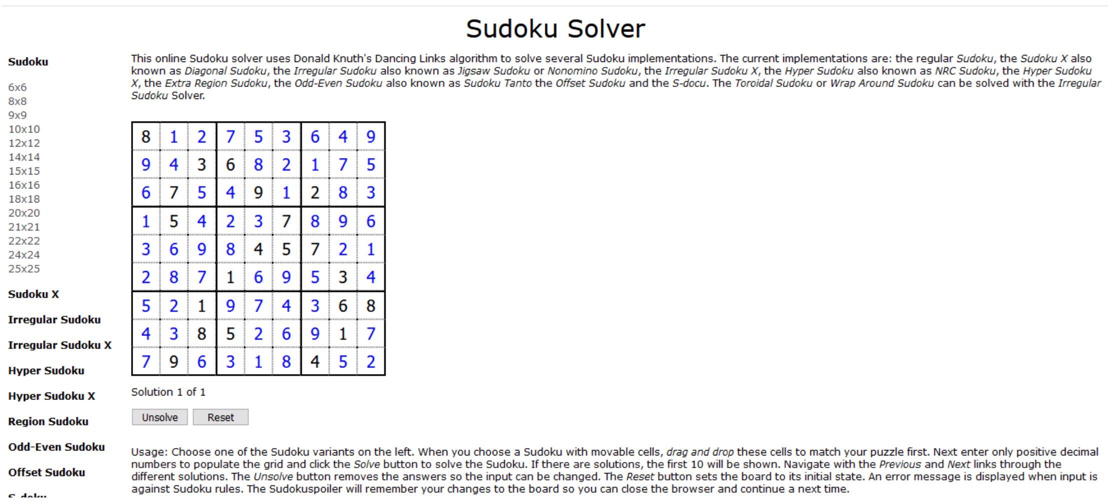
Description

The binary is a 64-bit ELF binary. If we run it in Linux environment:

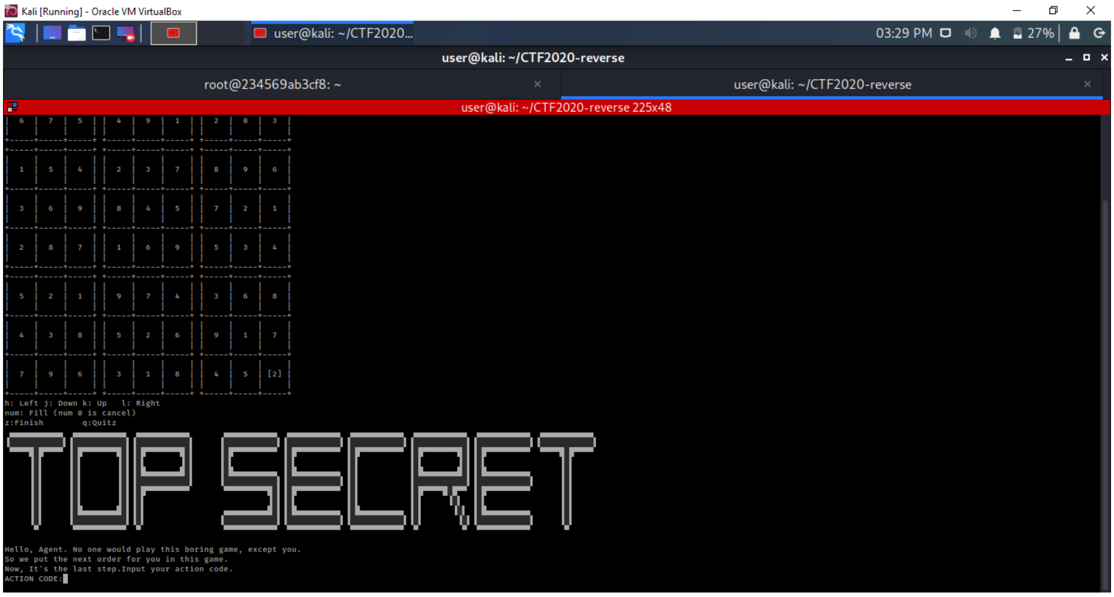


Basically, it is a Sudoku solving challenge (?

I suck at Sudoku, so let's solve it online ☺



Enter it and it will show us a secret panel:



Let's reverse this program with IDA:

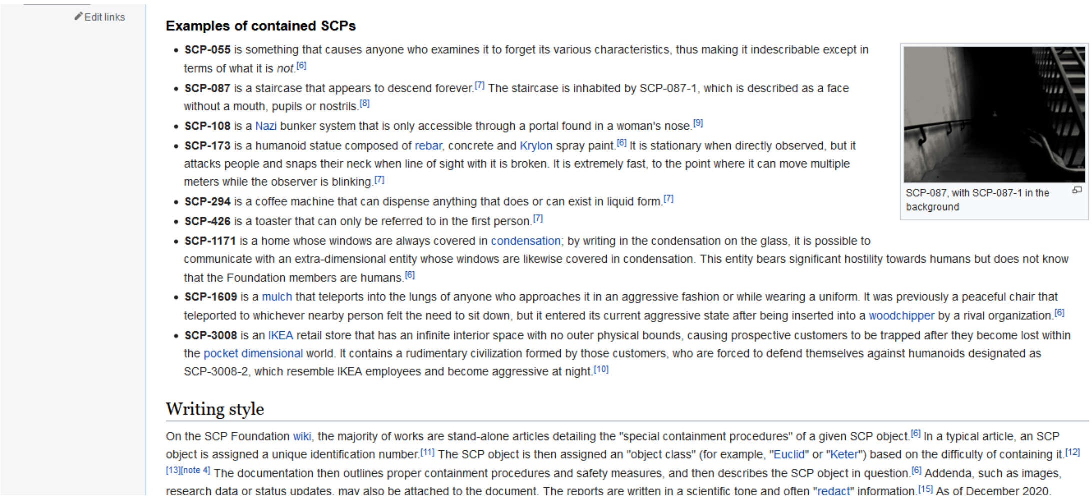
```
for ( i = 0; i <= 3160; ++i )
{
  message1[i] ^= sudoku_table[i % 81];
  v2 += message1[i];
}
for ( j = 0; j <= 168; ++j )
{
  messagw2[j] ^= sudoku_table[j % 81];
  v3 += messagw2[j];
}
if ( v2 == 0xFFFD09F2 && v3 == 0x39AB )
{
  v6 = 0;
  printf("%3161s%159s", message1, messagw2);
  __isoc99_scanf("%39s", s);
  for ( k = 0; k <= 6014; ++k )
  {
    c0 = magic[k];
    c1 = sudoku_table[k % 81];
    magic[k] = c1 ^ s[k % strlen(s)] ^ c0;
    v6 += magic[k];
  }
  if ( v6 == 0x3EDDA )
    puts(magic);
```

So basically, if we provide the correct message, it will print magic which contains the ascii art of the flag. So we have to somehow guess the secret message.

The method I used to guess the secret message is to assume that the flag is b'\0' * 32. The, we can see some spooky message hidden:

```
LHFsah
JECRYPTTHEDOCUMENTOFscpp
<["^PTTHEEDOCUMENTOFscpp
DDECRYPTTHEDOCUCENT0scpp0scp
[YPTTHEDOCUC
        qscp]Z]^

MRYPZ0

;DECRYPTTHEJ[MUENTOFscp[Y$xTHE?=WUMVL=OFscp
TW^gsm?B


YPTq
JCMU
;DECRYPTTHEDO[MUENTW^gaVLgOFscp
;DECRYPTTHEDO[MUENTW^gscp\5DRYVxTFg4HCUMENLgQFscp
U][JYP}-P]=JOCUMELgHFscp
;DECRYPTTHEDO[MUE@TW^gscpLgOFscp
;ECRYPTTHEDOCUMENTOFscp




OAMU]LT0W^g<c?
]
l][JPL}VP]g4HCUME7LOFscp
DECRYPTTHEDOCUMENTOFscpp
```

It seems that the flag contains something like "DECRYPT_THE_DOCUMENT_OF_...". But after some investigation, I began to realize that the flag should be lowercases because if the secret message consists of uppercases, v6 will be too small. Next, the "scp" field might have something to do with "SecureContainProtect". Hence, I looked up what secure container protect is and realize that maybe one of the document name will be the suffix of the secret message



Edit links

Examples of contained SCPs

- **SCP-055** is something that causes anyone who examines it to forget its various characteristics, thus making it indescribable except in terms of what it is *not*.[6]
- **SCP-087** is a staircase that appears to descend forever.[7] The staircase is inhabited by SCP-087-1, which is described as a face without a mouth, pupils or nostrils.[8]
- **SCP-108** is a Nazi bunker system that is only accessible through a portal found in a woman's nose.[9]
- **SCP-173** is a humanoid statue composed of rebar, concrete and Krylon spray paint.[6] It is stationary when directly observed, but it attacks people and snaps their neck when line of sight with it is broken. It is extremely fast, to the point where it can move multiple meters while the observer is blinking.[7]
- **SCP-294** is a coffee machine that can dispense anything that does or can exist in liquid form.[7]
- **SCP-426** is a toaster that can only be referred to in the first person.[7]
- **SCP-1171** is a home whose windows are always covered in condensation; by writing in the condensation on the glass, it is possible to communicate with an extra-dimensional entity whose windows are likewise covered in condensation. This entity bears significant hostility towards humans but does not know that the Foundation members are humans.[6]
- **SCP-1609** is a mulch that teleports into the lungs of anyone who approaches it in an aggressive fashion or while wearing a uniform. It was previously a peaceful chair that teleported to whichever nearby person felt the need to sit down, but it entered its current aggressive state after being inserted into a woodchipper by a rival organization.[6]
- **SCP-3008** is an IKEA retail store that has an infinite interior space with no outer physical bounds, causing prospective customers to be trapped after they become lost within the pocket dimensional world. It contains a rudimentary civilization formed by those customers, who are forced to defend themselves against humanoids designated as SCP-3008-2, which resemble IKEA employees and become aggressive at night.[10]

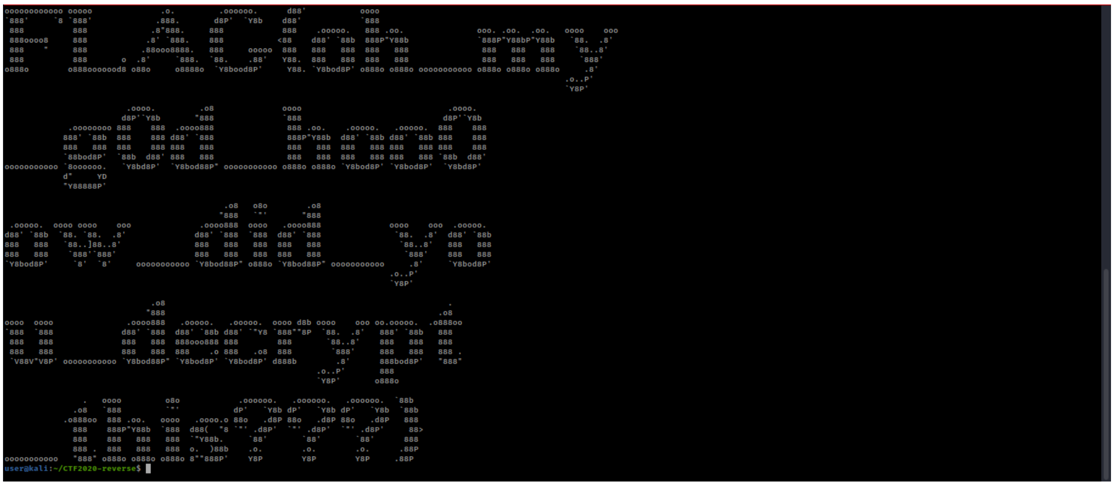
SCP-087, with SCP-087-1 in the background

Writing style

On the SCP Foundation wiki, the majority of works are stand-alone articles detailing the "special containment procedures" of a given SCP object.[6] In a typical article, an SCP object is assigned a unique identification number.[11] The SCP object is then assigned an "object class" (for example, "Euclid" or "Keter") based on the difficulty of containing it.[12][13][note 4] The documentation then outlines proper containment procedures and safety measures, and then describes the SCP object in question.[6] Addenda, such as images, research data or status updates, may also be attached to the document. The reports are written in a scientific tone and often "redact" information.[15] As of December 2020,

After hours of trial and error, I finally got the correct secret message. It is "decrypt_the_document_of_SCP-2521".

After entering the message, the flag appears >_<



The file `SecureContainProtect.py` contains the prove of concept script to get the flag.

Flag:

FLAG{oh_my_g0d_hoo0ow_did_you_decrypt_this???}