Reverse_Lab1

1 Task1

全都是`gcc`，返回效果：

```
gcc -S hello.c -o hellogcc.s
as hellogcc.s -o hellogcc.o
file hellogcc.o
hellogcc.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

全都是`clang`，返回效果：

```
clang -S hello.c -o helloclang.s
llvm-mc -filetype=obj helloclang.s -o helloclang.o
file helloclang.o
helloclang.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

先用`gcc`,后用`clang`

```
llvm-mc -filetype=obj hellogcc.s -o hellogccclang.o
file hellogccclang.o
hellogccclang.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

尝试执行的时候出现了错误：

```
./hellogccclang.o
-bash: ./hellogccclang.o: cannot execute binary file: Exec format error
```

用`clang`后再用`gcc`

```
as helloclang.s -o helloclanggcc.o
helloclang.s: Assembler messages:
helloclang.s:40: Error: unknown pseudo-op: `.addrsig'
helloclang.s:41: Error: unknown pseudo-op: `.addrsig_sym'
```

因此，两种编译器之间无法混用。

2 Task2

首先，将`challenge1`拖进IDA反编译，

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  wh4t_the_h3ll_i5_th1s();
  puts("Where is the flag?");
  return 0;
}
```

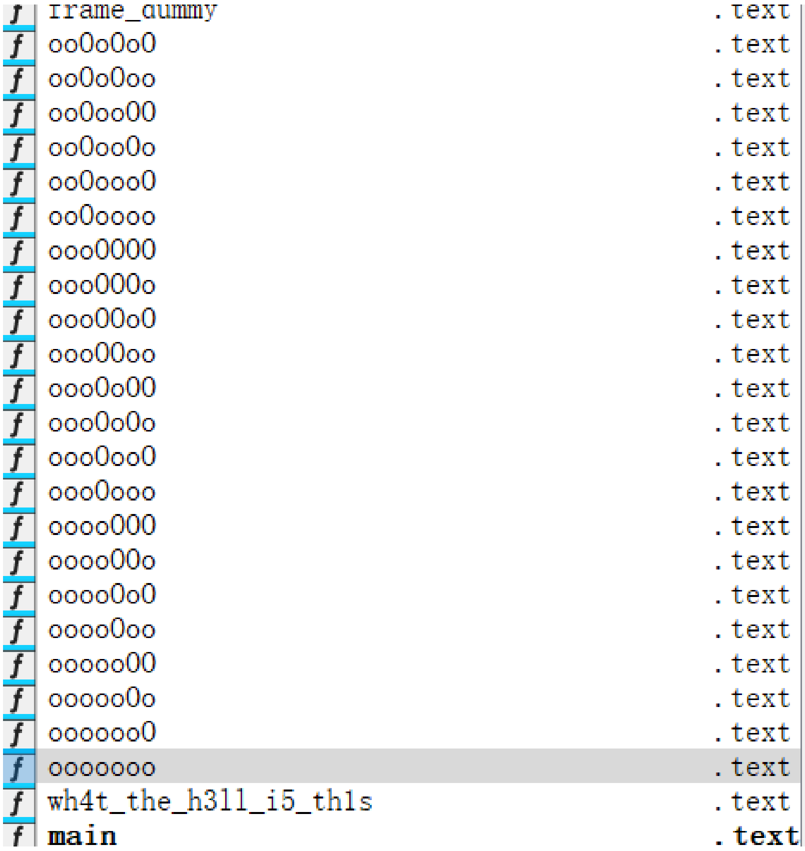
会发现这个函数并没有返回，也就是说，它没法跑，然后我开始考虑利用静态的方式解决它
发现核心的函数主要是这个:

```
__int64 wh4t_the_h3ll_i5_th1s()
{
  return ooooooo(fl4g);
}
```

然后进入`ooooooo`函数:

```
__int64 __fastcall ooooooo(_BYTE *a1)
{
  *a1 = 65;
  return oooooo0(a1 + 1);
}
```

在左边发现还有一系列这样的函数:



```
frame_dummy              .text
oo0o0o0                  .text
oo0o0oo                  .text
oo0oo00                  .text
oo0oo0o                  .text
oo0ooo0                  .text
oo0oooo                  .text
ooo0000                  .text
ooo000o                  .text
ooo00o0                  .text
ooo00oo                  .text
ooo0o00                  .text
ooo0o0o                  .text
ooo0oo0                  .text
ooo0ooo                  .text
oooo000                  .text
oooo00o                  .text
oooo0o0                  .text
oooo0oo                  .text
ooooo00                  .text
ooooo0o                  .text
oooooo0                  .text
ooooooo                  .text
wh4t_the_h3ll_i5_th1s    .text
main                     .text
```

```
_BYTE *__fastcall oo0o0o0(_BYTE *a1)
{
  _BYTE *result; // rax

  result = a1;
  *a1 = 0;
  return result;
}
```

大致理解其中的逻辑，大致是，每次存放一个`*a1`对应的数字的字符，然后地址+1
所以依次点击每个函数，获取所有ascii值：`65 65 65 123 104 111 112 101 95 117 95 104 97 118 101 95 102 117 110 126 125`
转换成字符：`AAA{hope_u_have_fun~}`
即为所求的`flag`

3 Task3

首先，先利用`llc`和`clang`,将`bc`文件变成可执行文件

```
llc challenge2.bc.old -o challenge2.s
clang challenge2.s -o challenge2
```

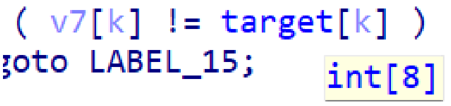
并且拖入IDA进行反汇编主程序如下：

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  size_t v3; // rbx
  size_t v4; // rbx
  int v5; // eax
  int v7[32]; // [rsp+0h] [rbp-F0h] BYREF
  char user_input[72]; // [rsp+80h] [rbp-70h] BYREF
  const char **v9; // [rsp+C8h] [rbp-28h]
  int v10; // [rsp+D4h] [rbp-1Ch]
  int v11; // [rsp+D8h] [rbp-18h]
  int i; // [rsp+DCh] [rbp-14h]
  int j; // [rsp+E0h] [rbp-10h]
  int k; // [rsp+E4h] [rbp-Ch]

  v10 = 0;
  v11 = argc;
  v9 = argv;
  memset(user_input, 0, 0x40uLL);
  memset(v7, 0, sizeof(v7));
  __isoc99_scanf(&unk_402410, user_input);
  for ( i = 0; ; ++i )
  {
    v3 = i;
    if ( v3 >= strlen(user_input) )
      break;
    if ( user_input[i] < 48 || user_input[i] > 57 )
    {
LABEL_15:
      printf("try again\n");
      exit(0);
    }
  }
  for ( j = 0; ; j += 2 )
  {
    v4 = j;
    if ( v4 >= strlen(user_input) )
      break;
    v5 = xcrc32(&user_input[j], 2LL, 4276803LL);
    v7[j / 2] = v5;
  }
  for ( k = 0; (unsigned __int64)k < 8; ++k )
  {
    if ( v7[k] != target[k] )
      goto LABEL_15;
  }
  printf("awesome\n");
  return 0;
}
```

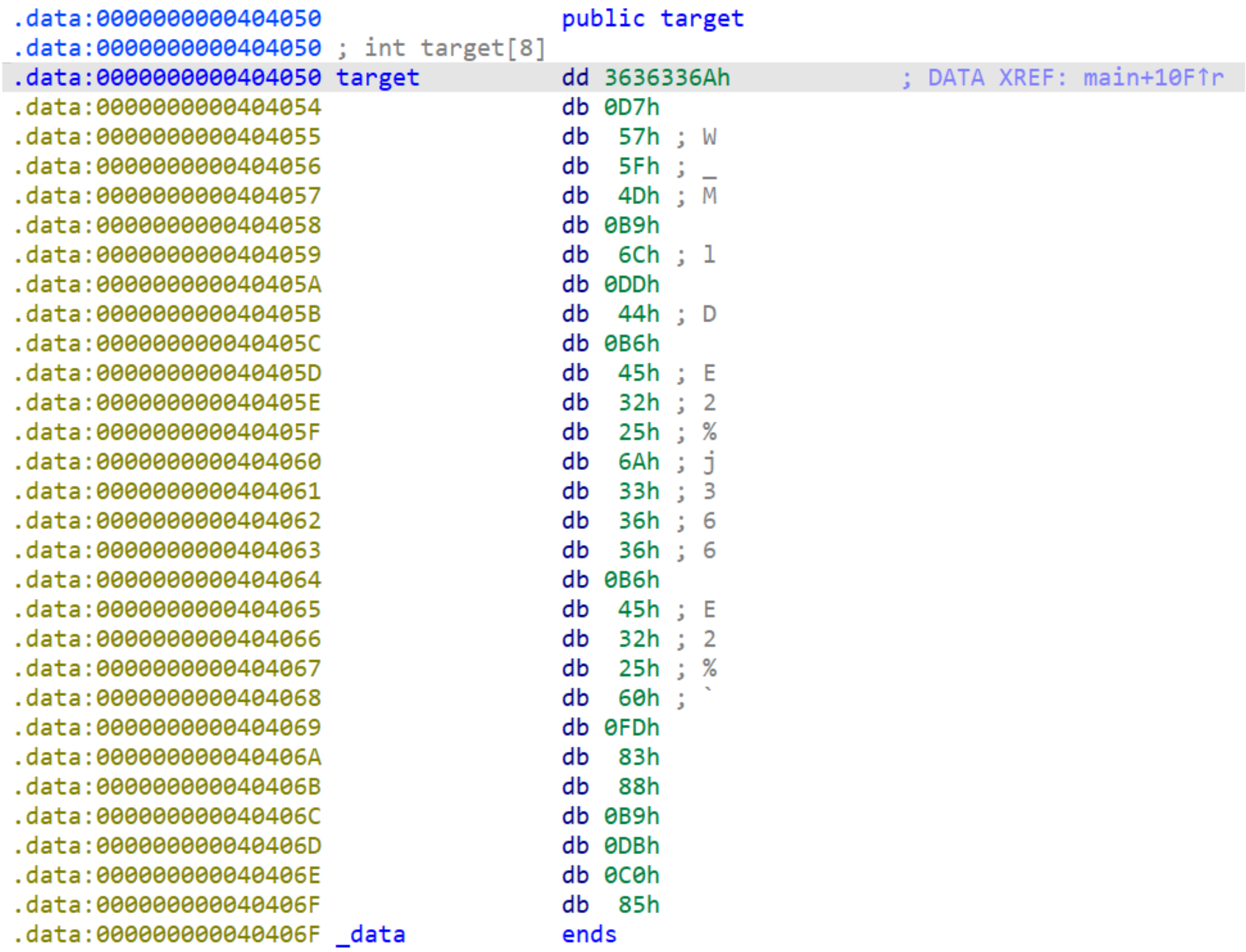
阅读`main`函数，大概了解到程序：

- 仅限数字输入
- 一次取两位数字计算`crc`值，并且和`target`数组中储存的值进行比较

观察到`target`是长度为8的数组，那么自然我们需要输入十六位密码咯x

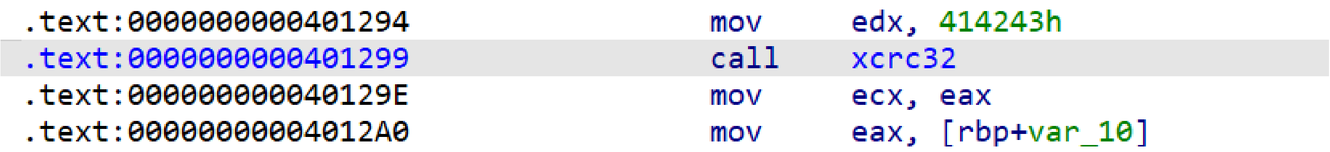


```
( v7[k] != target[k] )
goto LABEL_15;      int[8]
```



```
.data:0000000000404050                     public target
.data:0000000000404050 ; int target[8]
.data:0000000000404050 target              dd 3636336Ah              ; DATA XREF: main+10F↑r
.data:0000000000404054                     db 0D7h
.data:0000000000404055                     db  57h ; W
.data:0000000000404056                     db  5Fh ; _
.data:0000000000404057                     db  4Dh ; M
.data:0000000000404058                     db 0B9h
.data:0000000000404059                     db  6Ch ; l
.data:000000000040405A                     db 0DDh
.data:000000000040405B                     db  44h ; D
.data:000000000040405C                     db 0B6h
.data:000000000040405D                     db  45h ; E
.data:000000000040405E                     db  32h ; 2
.data:000000000040405F                     db  25h ; %
.data:0000000000404060                     db  6Ah ; j
.data:0000000000404061                     db  33h ; 3
.data:0000000000404062                     db  36h ; 6
.data:0000000000404063                     db  36h ; 6
.data:0000000000404064                     db 0B6h
.data:0000000000404065                     db  45h ; E
.data:0000000000404066                     db  32h ; 2
.data:0000000000404067                     db  25h ; %
.data:0000000000404068                     db  60h ; `
.data:0000000000404069                     db 0FDh
.data:000000000040406A                     db  83h
.data:000000000040406B                     db  88h
.data:000000000040406C                     db 0B9h
.data:000000000040406D                     db 0DBh
.data:000000000040406E                     db 0C0h
.data:000000000040406F                     db  85h
.data:000000000040406F _data               ends
```

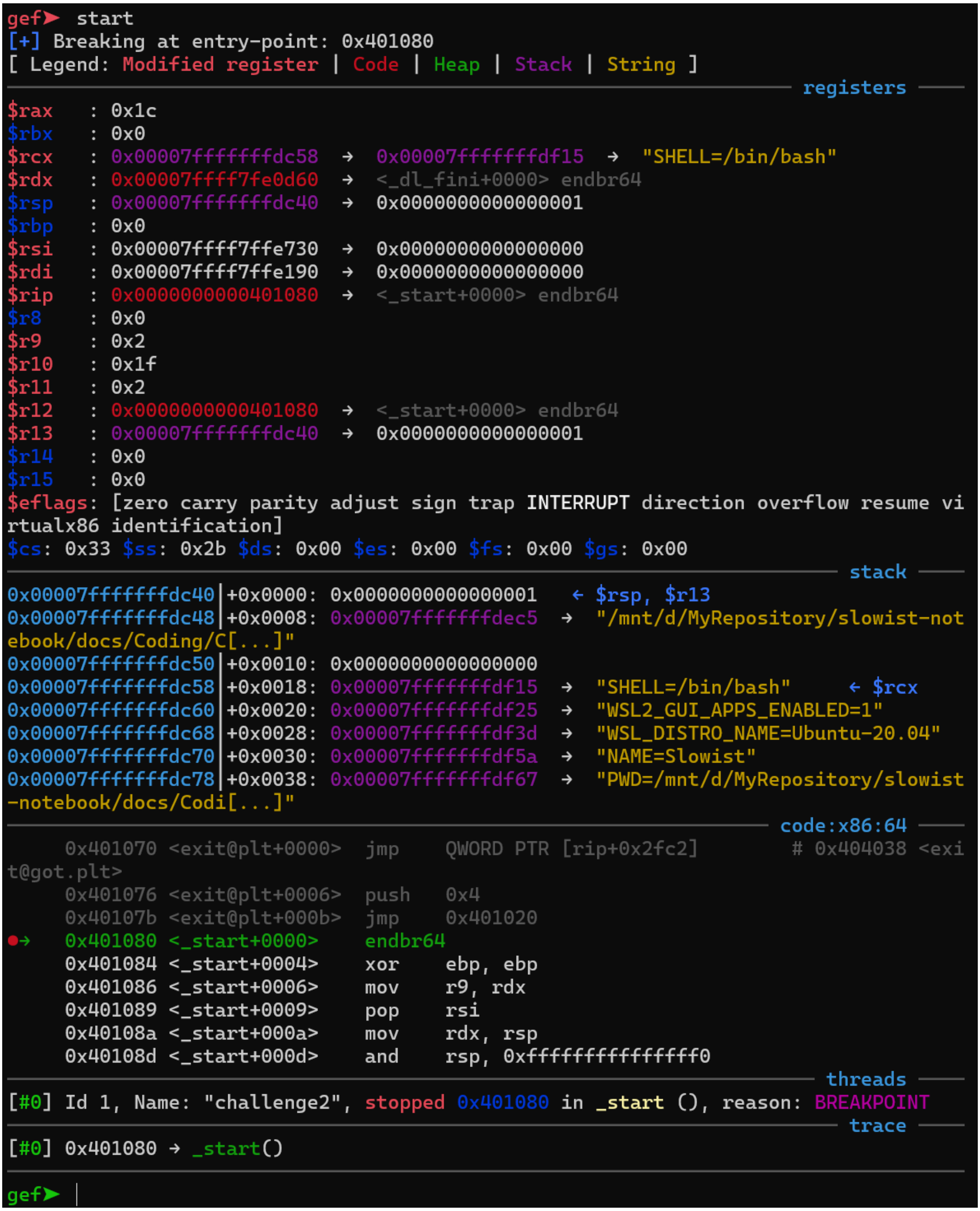
我们观察到主程序中主要进行计算的是 `xcrc32` 函数，观察其地址：



```
.text:0000000000401294                 mov     edx, 414243h
.text:0000000000401299                 call    xcrc32
.text:000000000040129E                 mov     ecx, eax
.text:00000000004012A0                 mov     eax, [rbp+var_10]
```

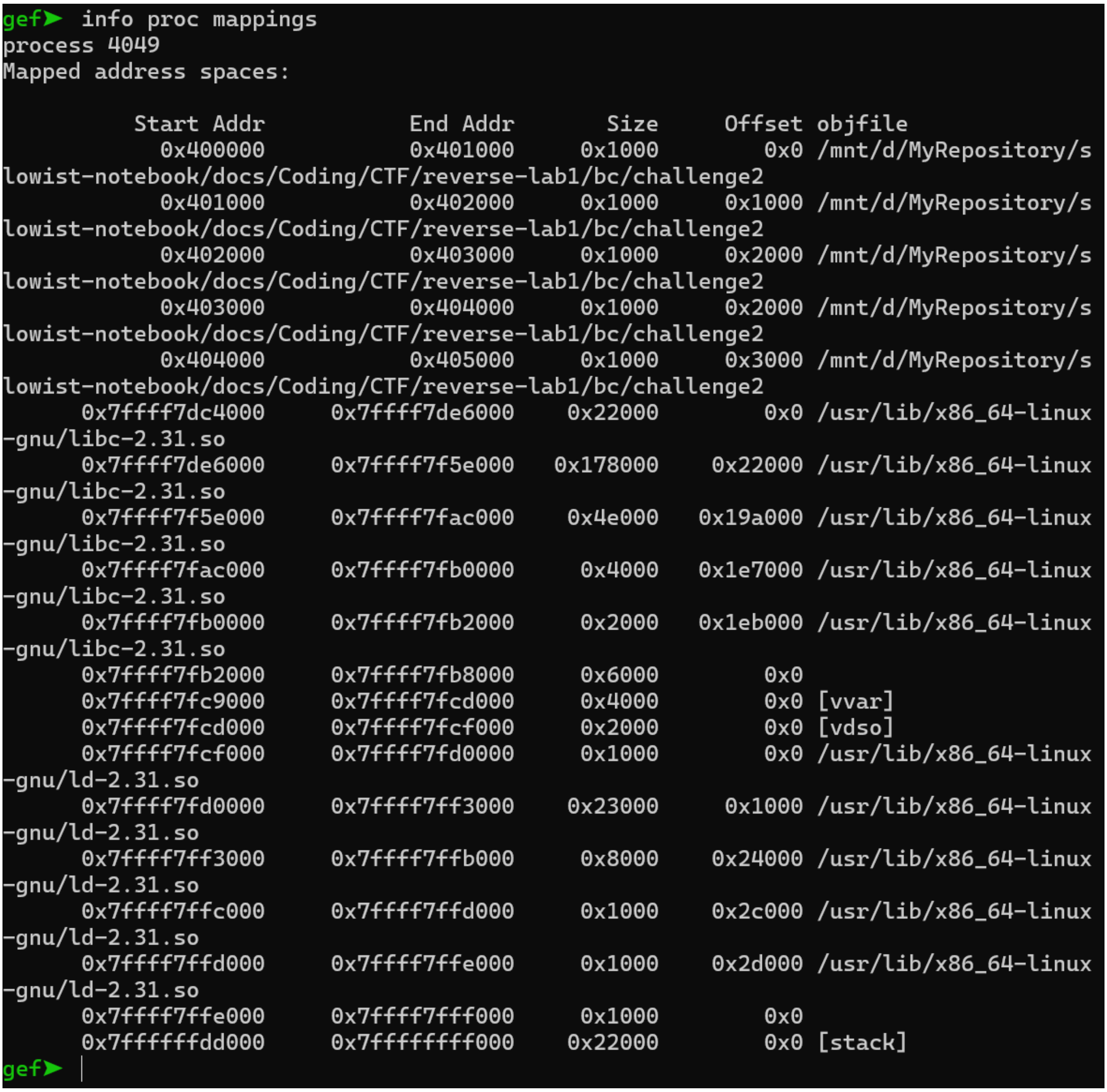
发现函数是 `...01299` 开始 `...0129E` 结束

因此我们使用 `gdb` 调试这段程序：



```
gef➤  start
[+] Breaking at entry-point: 0x401080
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x1c
$rbx   : 0x0
$rcx   : 0x00007fffffffdc58  →  0x00007fffffffdf15  →  "SHELL=/bin/bash"
$rdx   : 0x00007ffff7fe0d60  →  <_dl_fini+0000> endbr64
$rsp   : 0x00007fffffffdc40  →  0x0000000000000001
$rbp   : 0x0
$rsi   : 0x00007ffff7ffe730  →  0x0000000000000000
$rdi   : 0x00007ffff7ffe190  →  0x0000000000000000
$rip   : 0x0000000000401080  →  <_start+0000> endbr64
$r8    : 0x0
$r9    : 0x2
$r10   : 0x1f
$r11   : 0x2
$r12   : 0x0000000000401080  →  <_start+0000> endbr64
$r13   : 0x00007fffffffdc40  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow resume vi
rtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffdc40│+0x0000: 0x0000000000000001   ← $rsp, $r13
0x00007fffffffdc48│+0x0008: 0x00007fffffffdec5  →  "/mnt/d/MyRepository/slowist-not
ebook/docs/Coding/C[...]"
0x00007fffffffdc50│+0x0010: 0x0000000000000000
0x00007fffffffdc58│+0x0018: 0x00007fffffffdf15  →  "SHELL=/bin/bash"    ← $rcx
0x00007fffffffdc60│+0x0020: 0x00007fffffffdf25  →  "WSL2_GUI_APPS_ENABLED=1"
0x00007fffffffdc68│+0x0028: 0x00007fffffffdf3d  →  "WSL_DISTRO_NAME=Ubuntu-20.04"
0x00007fffffffdc70│+0x0030: 0x00007fffffffdf5a  →  "NAME=Slowist"
0x00007fffffffdc78│+0x0038: 0x00007fffffffdf67  →  "PWD=/mnt/d/MyRepository/slowist
-notebook/docs/Codi[...]"
────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401070 <exit@plt+0000>  jmp    QWORD PTR [rip+0x2fc2]        # 0x404038 <exi
t@got.plt>
     0x401076 <exit@plt+0006>  push   0x4
     0x40107b <exit@plt+000b>  jmp    0x401020
●→   0x401080 <_start+0000>    endbr64
     0x401084 <_start+0004>    xor    ebp, ebp
     0x401086 <_start+0006>    mov    r9, rdx
     0x401089 <_start+0009>    pop    rsi
     0x40108a <_start+000a>    mov    rdx, rsp
     0x40108d <_start+000d>    and    rsp, 0xfffffffffffffff0
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "challenge2", stopped 0x401080 in _start (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x401080 → _start()
───────────────────────────────────────────────────────────────────────────────────
gef➤  
```

用 `info proc mappings` 查看进程信息：



```
gef➤  info proc mappings
process 4049
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /mnt/d/MyRepository/s
lowist-notebook/docs/Coding/CTF/reverse-lab1/bc/challenge2
            0x401000           0x402000     0x1000     0x1000 /mnt/d/MyRepository/s
lowist-notebook/docs/Coding/CTF/reverse-lab1/bc/challenge2
            0x402000           0x403000     0x1000     0x2000 /mnt/d/MyRepository/s
lowist-notebook/docs/Coding/CTF/reverse-lab1/bc/challenge2
            0x403000           0x404000     0x1000     0x2000 /mnt/d/MyRepository/s
lowist-notebook/docs/Coding/CTF/reverse-lab1/bc/challenge2
            0x404000           0x405000     0x1000     0x3000 /mnt/d/MyRepository/s
lowist-notebook/docs/Coding/CTF/reverse-lab1/bc/challenge2
      0x7ffff7dc4000     0x7ffff7de6000    0x22000        0x0 /usr/lib/x86_64-linux
-gnu/libc-2.31.so
      0x7ffff7de6000     0x7ffff7f5e000   0x178000    0x22000 /usr/lib/x86_64-linux
-gnu/libc-2.31.so
      0x7ffff7f5e000     0x7ffff7fac000    0x4e000   0x19a000 /usr/lib/x86_64-linux
-gnu/libc-2.31.so
      0x7ffff7fac000     0x7ffff7fb0000     0x4000   0x1e7000 /usr/lib/x86_64-linux
-gnu/libc-2.31.so
      0x7ffff7fb0000     0x7ffff7fb2000     0x2000   0x1eb000 /usr/lib/x86_64-linux
-gnu/libc-2.31.so
      0x7ffff7fb2000     0x7ffff7fb8000     0x6000        0x0
      0x7ffff7fc9000     0x7ffff7fcd000     0x4000        0x0 [vvar]
      0x7ffff7fcd000     0x7ffff7fcf000     0x2000        0x0 [vdso]
      0x7ffff7fcf000     0x7ffff7fd0000     0x1000        0x0 /usr/lib/x86_64-linux
-gnu/ld-2.31.so
      0x7ffff7fd0000     0x7ffff7ff3000    0x23000     0x1000 /usr/lib/x86_64-linux
-gnu/ld-2.31.so
      0x7ffff7ff3000     0x7ffff7ffb000     0x8000    0x24000 /usr/lib/x86_64-linux
-gnu/ld-2.31.so
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x2c000 /usr/lib/x86_64-linux
-gnu/ld-2.31.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x2d000 /usr/lib/x86_64-linux
-gnu/ld-2.31.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffdd000     0x7ffffffff000    0x22000        0x0 [stack]
gef➤  
```

可以发现，程序是从`0x400000`开始加载的
利用`python`计算实际地址：

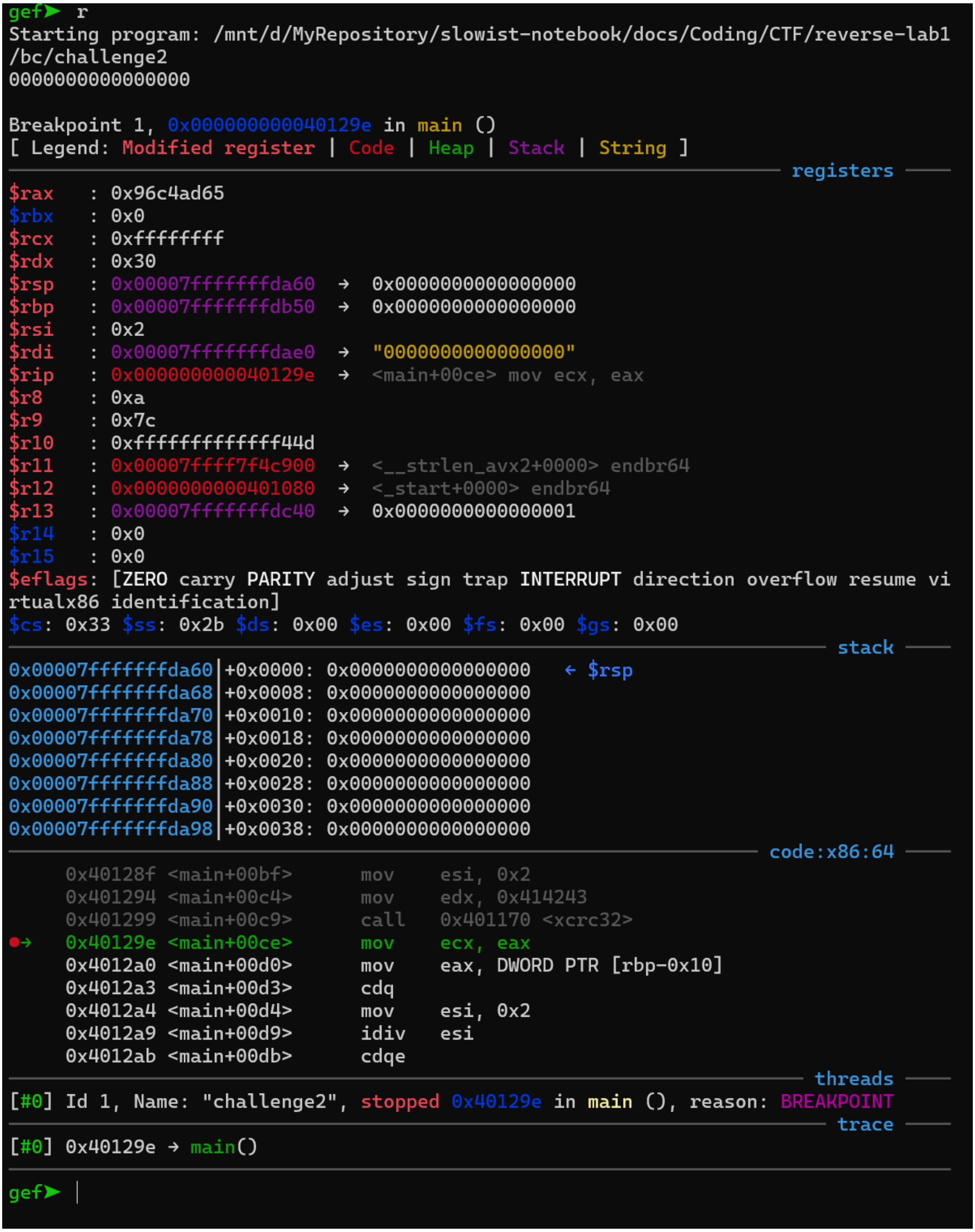
```
gef➤  python print(hex(0x400000+0x129e))
0x40129e
```

为了查看运行之后的结果，我们在上面设置断点：

```
gef➤  b *0x40129e
Breakpoint 1 at 0x555555400b6e
```

下面我们从`0`开始，构造16位输入：

```
gef➤  python print('0'*16)
0000000000000000
```



```
gef➤  r
Starting program: /mnt/d/MyRepository/slowist-notebook/docs/Coding/CTF/reverse-lab1
/bc/challenge2
0000000000000000

Breakpoint 1, 0x000000000040129e in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x96c4ad65
$rbx   : 0x0
$rcx   : 0xffffffff
$rdx   : 0x30
$rsp   : 0x00007fffffffda60  →  0x0000000000000000
$rbp   : 0x00007fffffffdb50  →  0x0000000000000000
$rsi   : 0x2
$rdi   : 0x00007fffffffdae0  →  "0000000000000000"
$rip   : 0x000000000040129e  →  <main+00ce> mov ecx, eax
$r8    : 0xa
$r9    : 0x7c
$r10   : 0xfffffffffffff44d
$r11   : 0x00007ffff7f4c900  →  <__strlen_avx2+0000> endbr64
$r12   : 0x0000000000401080  →  <_start+0000> endbr64
$r13   : 0x00007fffffffdc40  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume vi
rtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────── stack ────
0x00007fffffffda60│+0x0000: 0x0000000000000000   ← $rsp
0x00007fffffffda68│+0x0008: 0x0000000000000000
0x00007fffffffda70│+0x0010: 0x0000000000000000
0x00007fffffffda78│+0x0018: 0x0000000000000000
0x00007fffffffda80│+0x0020: 0x0000000000000000
0x00007fffffffda88│+0x0028: 0x0000000000000000
0x00007fffffffda90│+0x0030: 0x0000000000000000
0x00007fffffffda98│+0x0038: 0x0000000000000000
────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x40128f <main+00bf>       mov    esi, 0x2
     0x401294 <main+00c4>       mov    edx, 0x414243
     0x401299 <main+00c9>       call   0x401170 <xcrc32>
●→   0x40129e <main+00ce>       mov    ecx, eax
     0x4012a0 <main+00d0>       mov    eax, DWORD PTR [rbp-0x10]
     0x4012a3 <main+00d3>       cdq
     0x4012a4 <main+00d4>       mov    esi, 0x2
     0x4012a9 <main+00d9>       idiv   esi
     0x4012ab <main+00db>       cdqe
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "challenge2", stopped 0x40129e in main (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x40129e → main()
───────────────────────────────────────────────────────────────────────────────────
gef➤  
```

由断点位置，知道`eax`存放了位置

```
gef➤  p $eax
$1 = 0x96c4ad65
```

到这里我卡住了 ~~可能是源码没读通~~ 感觉实在没什么思路。我的问题一直卡在`crc32`给我返回的值是32bit、8位十六进制，而`target`中全部都是二位十六进制，我感觉即使我从`00`试到`99`也不可能试出一个值来。

第二个问题是，假使手动打表，从`00`试到`99`，要试`100`个值，真的要手动输入/比对吗？于是在我看懂的意思里面，我尝试写了一个和这个功能类似的程序，用遍历去构造一下输入，然后就有了下面：

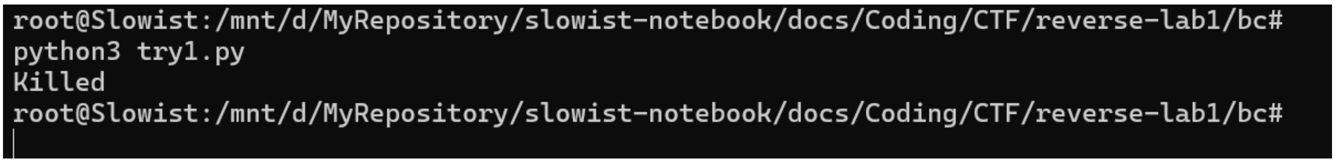
```python
import binascii

crc32_table = [
    # omitted. it's too long!
]
def xcrc32(data, initial_crc):
    crc = initial_crc
    for byte in data:
        table_index = (byte ^ (crc >> 24)) & 0xFF
        crc = crc32_table[table_index] ^ (crc << 8)
        crc &= 0xFF
    return crc

def find_two_byte_combinations(target):
    combinations = []

    for t in target:
        found = False
        for i in range(0, 10):
            for j in range(0, 10):
                candidate = (i << 8) | j
                if binascii.crc32(candidate.to_bytes(2, 'big')) & 0xFFFFFFFF == t:
                    combinations.append(candidate)
                    found = True
                    break
            if found:
                break

    return combinations

target = [
    0xD7, 0x57, 0x5F, 0x4D, 0x89, 0x6C, 0xDD, 0x44, 0x86, 0x45, 0x32, 0x25,
    0x6A, 0x33, 0x36, 0x36, 0x86, 0x45, 0x32, 0x25, 0x60, 0xFD, 0x83, 0x88,
    0x89, 0xDB, 0x85
]

combinations = find_two_byte_combinations(target)
for idx, comb in enumerate(combinations):
    print(f"Target[{idx}]: {target[idx]:04X} corresponds to {comb:04X}")
```

但从最终结果来看，也没有对应的数字，说明我对这个程序还是有一定误解x

我还尝试了一下暴力遍历，毕竟一共16位数字，暴力也许可以遍历出来的，于是：

```python
import subprocess
from multiprocessing import Pool, Manager

def run_elf_program(inputs):
    results = []
    for input_value in inputs:
        input_string = str(input_value)
        process = subprocess.Popen(['/mnt/d/MyRepository/slowist-notebook/docs/Coding/CTF/reverse-lab1/bc/challenge2'], stdin=subprocess.PIPE, stdout=subprocess.PIPE, text=True)
        stdout, _ = process.communicate(input=input_string)
        if "awesome" in stdout.lower():
            results.append(input_value)

    return results

def main():
    manager = Manager()
    found = manager.Value('i', 0)

    def batch_run(batch):
        if found.value:
            return []
        results = run_elf_program(batch)
        if results:
            found.value = 1  # 设置标志
        return results

    batch_size = 1000
    input_values = range(10000000000000000)
    batches = [input_values[i:i + batch_size] for i in range(0, len(input_values), batch_size)]

    with Pool(processes=20) as pool:
        for result_batch in pool.imap_unordered(batch_run, batches):
            if result_batch:
                print(f"找到结果为 'awesome' 的输入值：{result_batch[0]}")
                pool.terminate()
                break
        else:
            print("未找到符合条件的输入值。")

if __name__ == "__main__":
    main()
```

但好像最终由于我一下子用了20个进程，所以最终的结果还是无疾而终x



```
root@Slowist:/mnt/d/MyRepository/slowist-notebook/docs/Coding/CTF/reverse-lab1/bc# python3 try1.py
Killed
root@Slowist:/mnt/d/MyRepository/slowist-notebook/docs/Coding/CTF/reverse-lab1/bc#
```

如果我能想出来的话，我觉得大致思路应该是根据target进行反查表，然后推出对应的数字组合，最后输16位数字，然后得出`awesome`