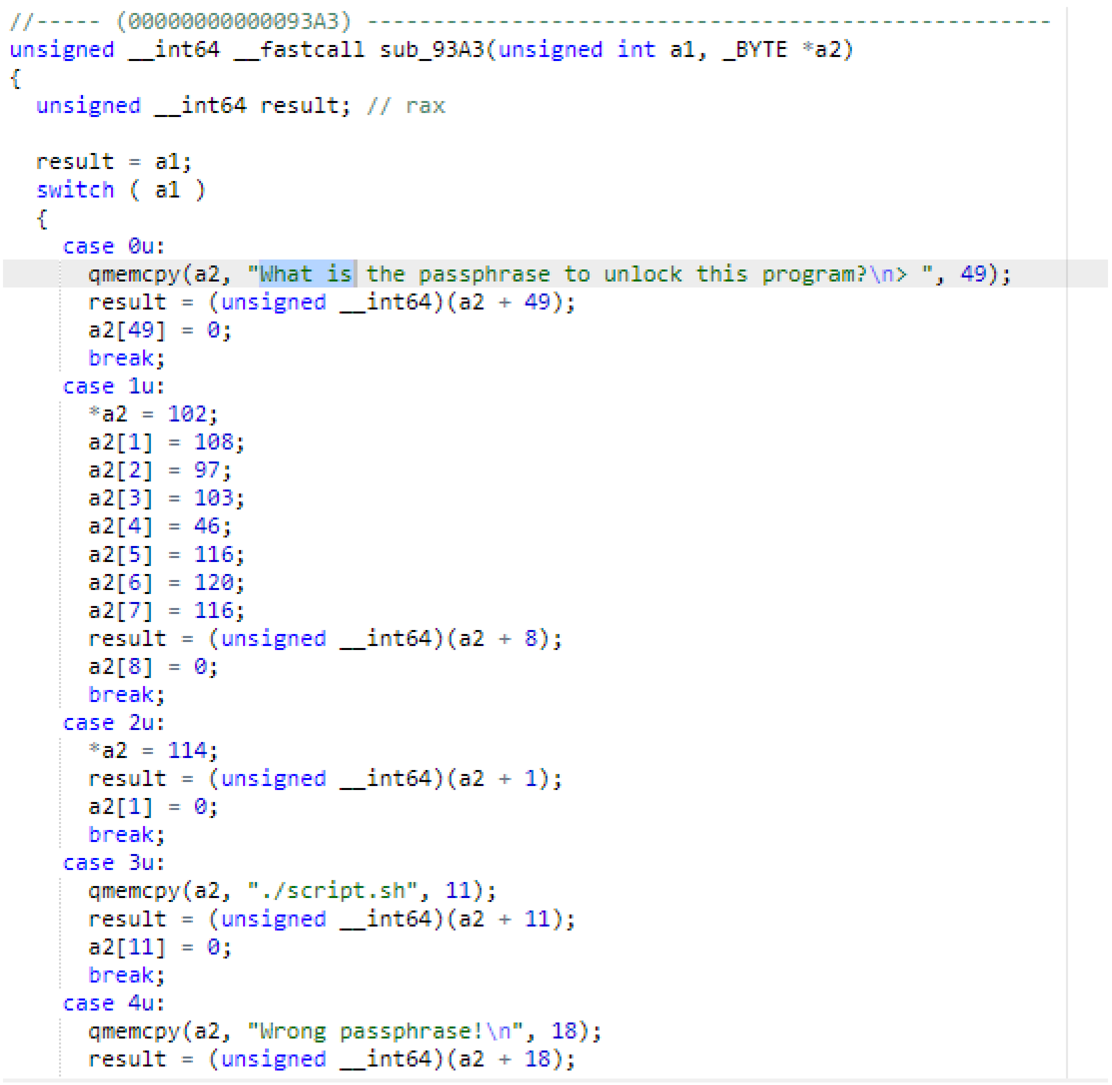
The goal of this challenge is to reverse this highly obfuscated binary and get the correct passphrase

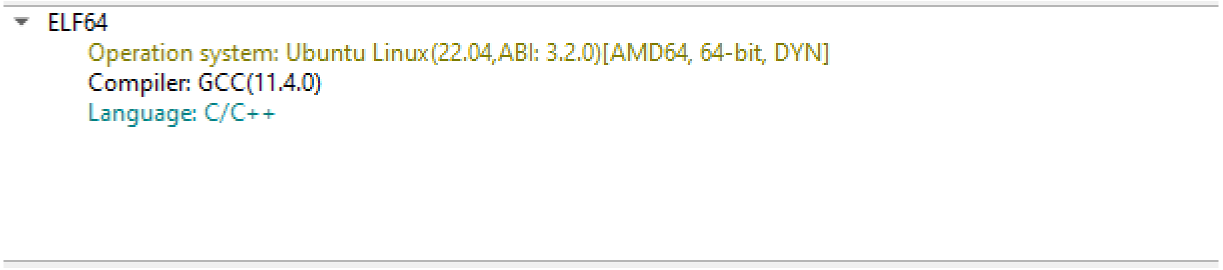
Explanation

The program asks for a passphrase, because of that and the name of the program I assumed there would some passphrase obfuscated, so I decompiled it :



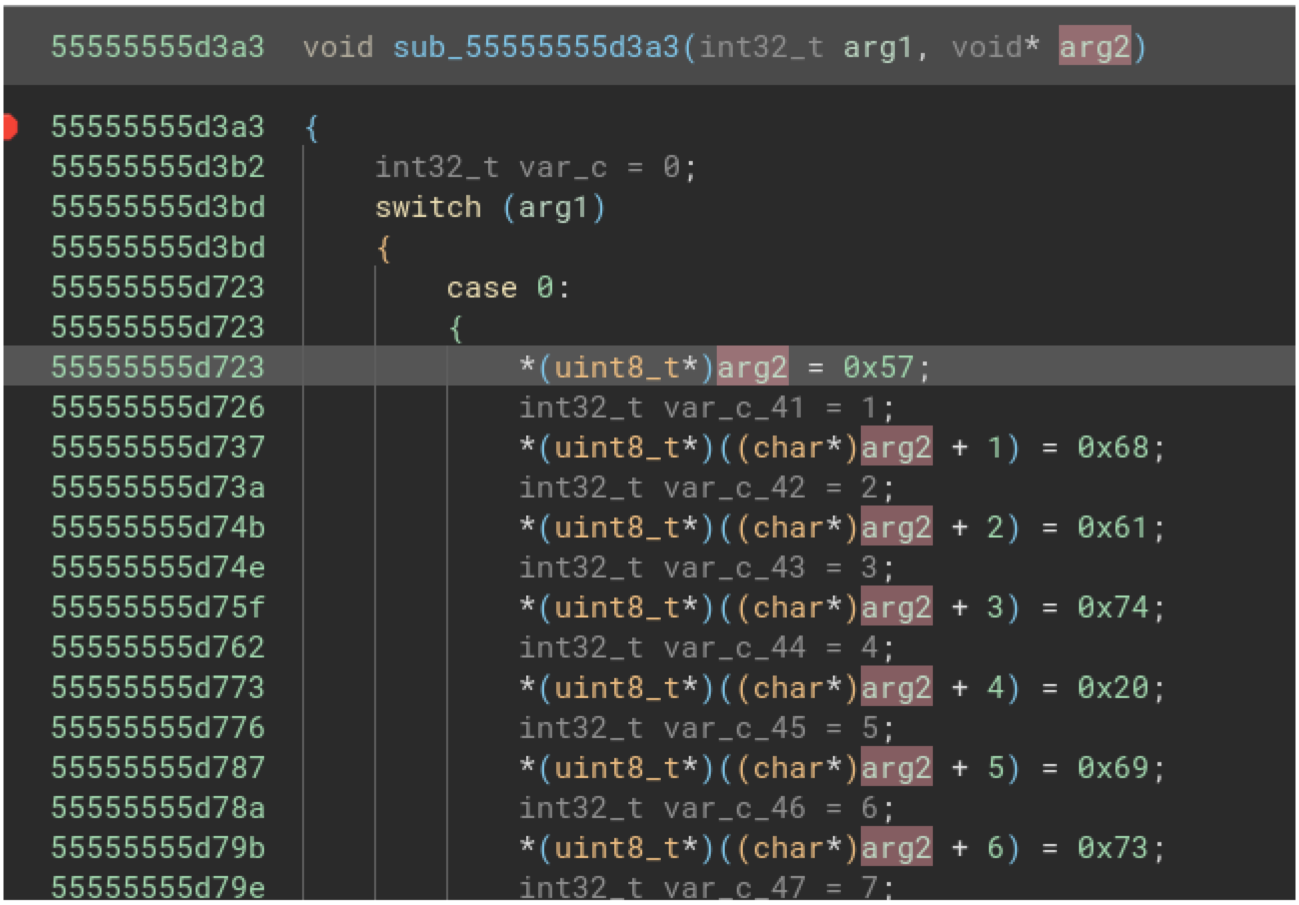
```
//----- (00000000000093A3) ----------------------------------------------------
unsigned __int64 __fastcall sub_93A3(unsigned int a1, _BYTE *a2)
{
  unsigned __int64 result; // rax

  result = a1;
  switch ( a1 )
  {
    case 0u:
      qmemcpy(a2, "What is the passphrase to unlock this program?\n> ", 49);
      result = (unsigned __int64)(a2 + 49);
      a2[49] = 0;
      break;
    case 1u:
      *a2 = 102;
      a2[1] = 108;
      a2[2] = 97;
      a2[3] = 103;
      a2[4] = 46;
      a2[5] = 116;
      a2[6] = 120;
      a2[7] = 116;
      result = (unsigned __int64)(a2 + 8);
      a2[8] = 0;
      break;
    case 2u:
      *a2 = 114;
      result = (unsigned __int64)(a2 + 1);
      a2[1] = 0;
      break;
    case 3u:
      qmemcpy(a2, "./script.sh", 11);
      result = (unsigned __int64)(a2 + 11);
      a2[11] = 0;
      break;
    case 4u:
      qmemcpy(a2, "Wrong passphrase!\n", 18);
      result = (unsigned __int64)(a2 + 18);
```

Couldn't understand anything so I checked if maybe it was packed ?



```
ELF64
    Operation system: Ubuntu Linux(22.04,ABI: 3.2.0)[AMD64, 64-bit, DYN]
    Compiler: GCC(11.4.0)
    Language: C/C++
```

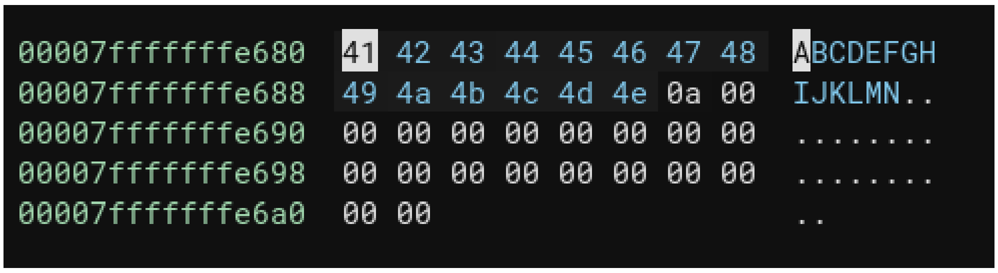
Nope, okay we'll let's use BinaryNinja then :



```
55555555d3a3  void sub_55555555d3a3(int32_t arg1, void* arg2)

55555555d3a3  {
55555555d3b2      int32_t var_c = 0;
55555555d3bd      switch (arg1)
55555555d3bd      {
55555555d723          case 0:
55555555d723          {
55555555d723              *(uint8_t*)arg2 = 0x57;
55555555d726              int32_t var_c_41 = 1;
55555555d737              *(uint8_t*)((char*)arg2 + 1) = 0x68;
55555555d73a              int32_t var_c_42 = 2;
55555555d74b              *(uint8_t*)((char*)arg2 + 2) = 0x61;
55555555d74e              int32_t var_c_43 = 3;
55555555d75f              *(uint8_t*)((char*)arg2 + 3) = 0x74;
55555555d762              int32_t var_c_44 = 4;
55555555d773              *(uint8_t*)((char*)arg2 + 4) = 0x20;
55555555d776              int32_t var_c_45 = 5;
55555555d787              *(uint8_t*)((char*)arg2 + 5) = 0x69;
55555555d78a              int32_t var_c_46 = 6;
55555555d79b              *(uint8_t*)((char*)arg2 + 6) = 0x73;
55555555d79e              int32_t var_c_47 = 7;
```

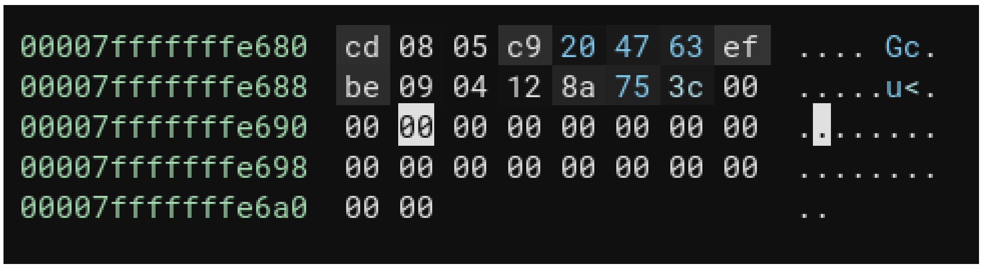
93A3 is our function, so let's set a breakpoint at the start of it and see what happens to our input :

Before :



```
00007fffffffe680  41 42 43 44 45 46 47 48  ABCDEFGH
00007fffffffe688  49 4a 4b 4c 4d 4e 0a 00  IJKLMN..
00007fffffffe690  00 00 00 00 00 00 00 00  ........
00007fffffffe698  00 00 00 00 00 00 00 00  ........
00007fffffffe6a0  00 00                    ..
```

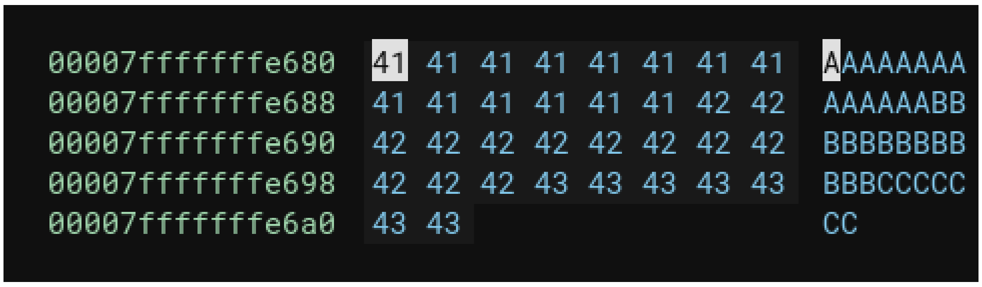
After :



```
00007fffffffe680  cd 08 05 c9 20 47 63 ef  .... Gc.
00007fffffffe688  be 09 04 12 8a 75 3c 00  .....u<.
00007fffffffe690  00 00 00 00 00 00 00 00  ........
00007fffffffe698  00 00 00 00 00 00 00 00  ........
00007fffffffe6a0  00 00                    ..
```

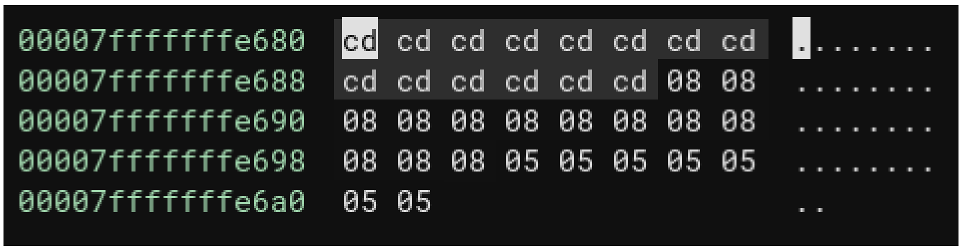
Interesting, looks like there's a mapping function, I sent another input just to make sure and it's probably the case :

Before :



```
00007fffffffe680  41 41 41 41 41 41 41 41  AAAAAAAA
00007fffffffe688  41 41 41 41 41 41 42 42  AAAAAABB
00007fffffffe690  42 42 42 42 42 42 42 42  BBBBBBBB
00007fffffffe698  42 42 42 43 43 43 43 43  BBBCCCCC
00007fffffffe6a0  43 43                    CC
```

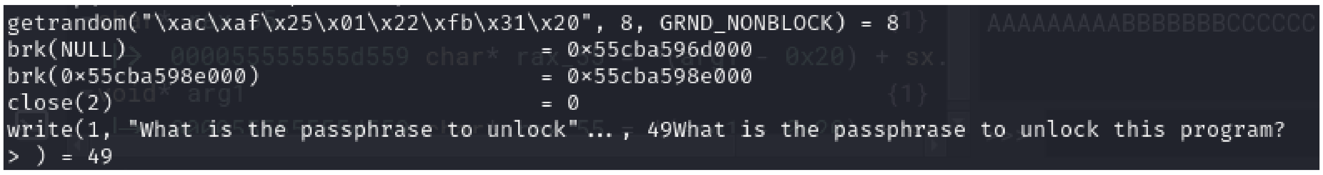
After :



```
00007fffffffe680  cd cd cd cd cd cd cd cd  ........
00007fffffffe688  cd cd cd cd cd cd 08 08  ........
00007fffffffe690  08 08 08 08 08 08 08 08  ........
00007fffffffe698  08 08 08 05 05 05 05 05  ........
00007fffffffe6a0  05 05                    ..
```

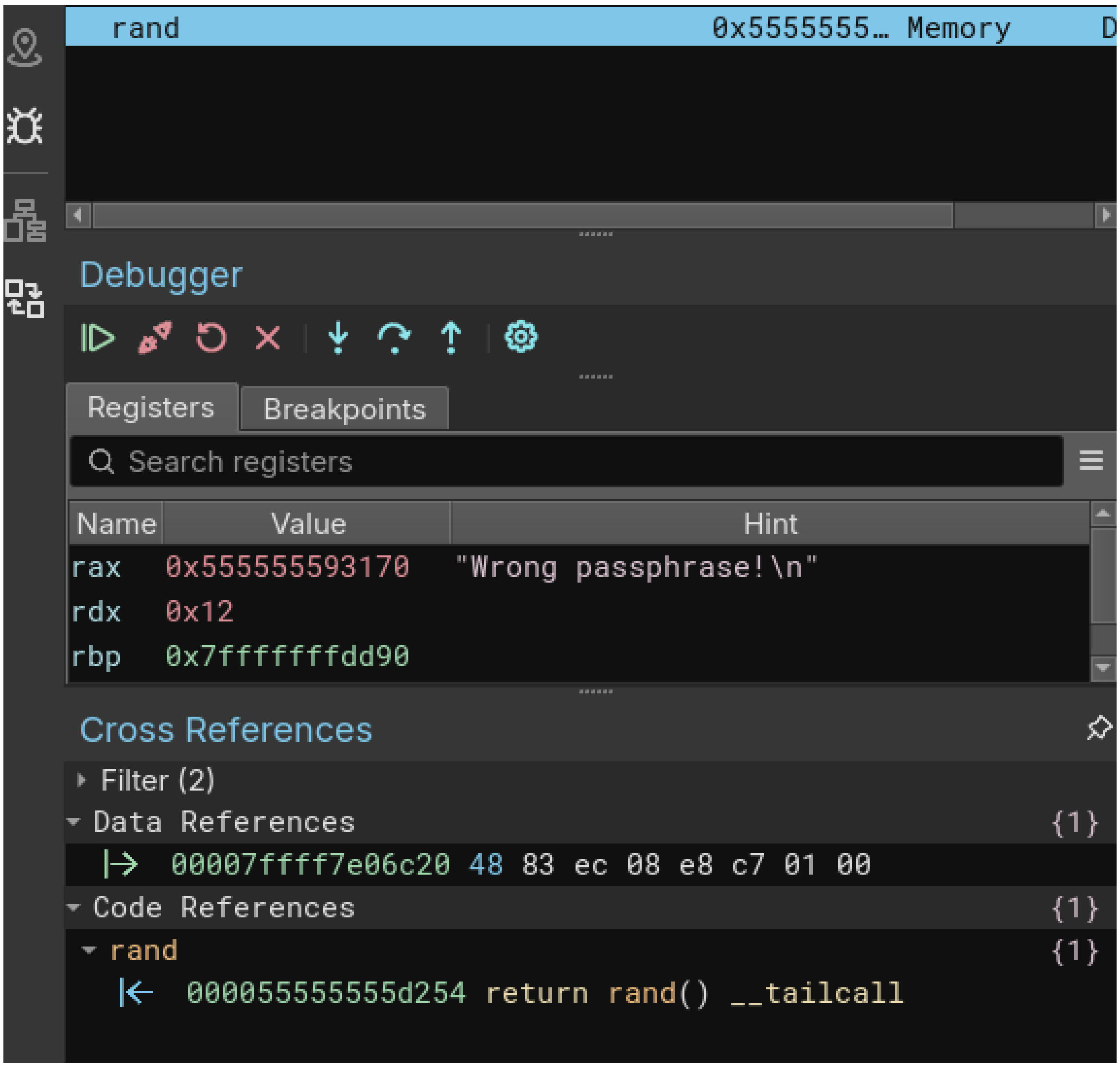
But then I got stuck

So I tried strace and saw this :



```
getrandom("\xac\xaf\x25\x01\x22\xfb\x31\x20", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x55cba596d000
brk(0x55cba598e000)                     = 0x55cba598e000
close(2)                                = 0
write(1, "What is the passphrase to unlock"..., 49What is the passphrase to unlock this program?
> ) = 49
```

I needed to find where this happens, so I went back and checked if rand gets called


```
rand                                    0x5555555… Memory
```

Debugger

Registers | Breakpoints

Search registers

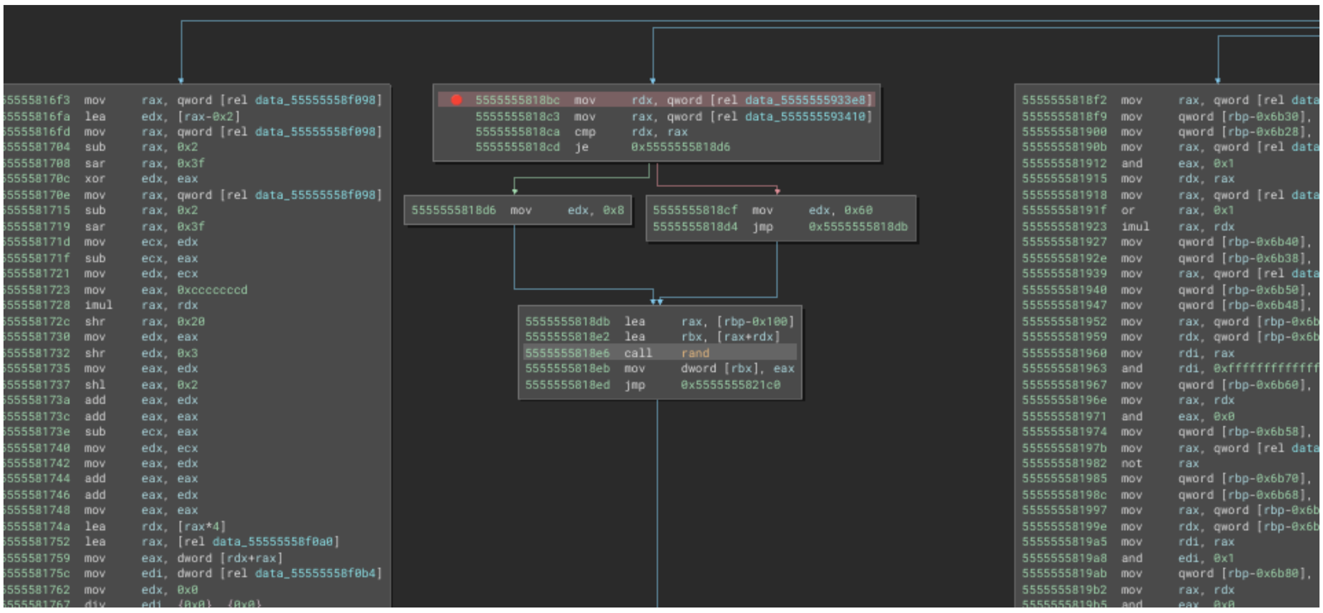
```
Name  Value            Hint
rax   0x555555593170   "Wrong passphrase!\n"
rdx   0x12
rbp   0x7fffffffdd90
```

Cross References

```
Filter (2)
Data References                                          {1}
  |→  00007ffff7e06c20 48 83 ec 08 e8 c7 01 00
Code References                                          {1}
  rand                                                   {1}
    |←  000055555555d254 return rand() __tailcall
```

Let's put a breakpoint on rand and spam continue until I find which function calls it

and there it is : at 818e6


```
5555816f3  mov    rax, qword [rel data_55555558f098]
5555816fa  lea    edx, [rax-0x2]
5555816fd  mov    rax, qword [rel data_55555558f098]
555581704  sub    rax, 0x2
555581708  sar    rax, 0x3f
55558170c  xor    edx, eax
55558170e  mov    rax, qword [rel data_55555558f098]
555581715  sub    rax, 0x2
555581719  sar    rax, 0x3f
55558171d  mov    ecx, edx
55558171f  sub    ecx, eax
555581721  mov    edx, ecx
555581723  mov    eax, 0xcccccccd
555581728  imul   rax, rdx
55558172c  shr    rax, 0x20
555581730  mov    edx, eax
555581732  shr    edx, 0x3
555581735  mov    eax, edx
555581737  shl    eax, 0x2
55558173a  add    eax, edx
55558173c  add    eax, eax
55558173e  sub    ecx, eax
555581740  mov    edx, ecx
555581742  mov    eax, edx
555581744  add    eax, eax
555581746  add    eax, edx
555581748  mov    eax, eax
55558174a  lea    rdx, [rax*4]
555581752  lea    rax, [rel data_55555558f0a0]
555581759  mov    eax, dword [rdx+rax]
55558175c  mov    edi, dword [rel data_55555558f0b4]
555581762  mov    edx, 0x0
555581767  div    edi, {0x0}, {0x0}
```

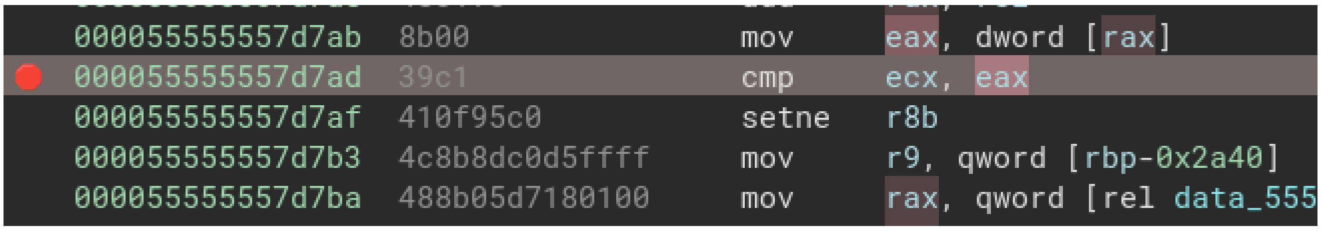
```
● 5555555818bc  mov    rdx, qword [rel data_5555555933e8]
  5555555818c3  mov    rax, qword [rel data_555555593410]
  5555555818ca  cmp    rdx, rax
  5555555818cd  je     0x5555555818d6

5555555818d6  mov    edx, 0x8

5555555818cf  mov    edx, 0x60
5555555818d4  jmp    0x5555555818db

5555555818db  lea    rax, [rbp-0x100]
5555555818e2  lea    rbx, [rax+rdx]
5555555818e6  call   rand
5555555818eb  mov    dword [rbx], eax
5555555818ed  jmp    0x5555555821c0
```

```
5555555818f2  mov    rax, qword [rel data
5555555818f9  mov    qword [rbp-0x6b30],
555555581900  mov    qword [rbp-0x6b28],
55555558190b  mov    rax, qword [rel data
555555581912  and    eax, 0x1
555555581915  mov    rdx, rax
555555581918  mov    rax, qword [rel data
55555558191f  or     rax, 0x1
555555581923  imul   rax, rdx
555555581927  mov    qword [rbp-0x6b40],
55555558192e  mov    qword [rbp-0x6b38],
555555581939  mov    rax, qword [rel data
555555581940  mov    qword [rbp-0x6b50],
555555581947  mov    qword [rbp-0x6b48],
555555581952  mov    rax, qword [rbp-0x6b
555555581959  mov    rdx, qword [rbp-0x6b
555555581960  mov    rdi, rax
555555581963  and    rdi, 0xffffffffffff
555555581967  mov    qword [rbp-0x6b60],
55555558196e  mov    rax, rdx
555555581971  and    eax, 0x0
555555581974  mov    qword [rbp-0x6b58],
55555558197b  mov    rax, qword [rel data
555555581982  not    rax
555555581985  mov    qword [rbp-0x6b70],
55555558198c  mov    qword [rbp-0x6b68],
555555581997  mov    rax, qword [rbp-0x6b
55555558199e  mov    rdx, qword [rbp-0x6b
5555555819a5  mov    rdi, rax
5555555819a8  and    edi, 0x1
5555555819ab  mov    qword [rbp-0x6b80],
5555555819b2  mov    rax, rdx
5555555819b5  and    eax, 0x0
```

now we'll see what happens with this rand call after spamming it after 34 times we finally stop calling rand, this probably means that the flag is 34 characters long.

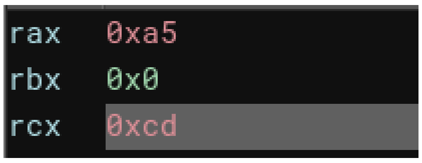
What I did was make rand always return 0 and see what happens.

After trial and error, I found this line that compares our result with the expected result :



```
000055555557d7ab  8b00            mov     eax, dword [rax]
000055555557d7ad  39c1            cmp     ecx, eax
000055555557d7af  410f95c0        setne   r8b
000055555557d7b3  4c8b8dc0d5ffff  mov     r9, qword [rbp-0x2a40]
000055555557d7ba  488b05d7180100  mov     rax, qword [rel data_555
```

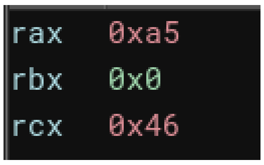
"0xcd" is our input since A turns to 0xcd and we are comparing it with 0xa5



```
rax   0xa5
rbx   0x0
rcx   0xcd
```

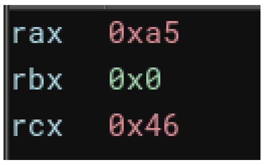
so there's probably a xor with the LSB of the rand value, let's try setting back to generating random values.

let's input "a" :



```
rax   0xa5
rbx   0x0
rcx   0x46
```

Then I wondered : how can I solve this challenge if the values are random each time yet the compared value is always the same ? so I wrote "a" again :



```
rax   0xa5
rbx   0x0
rcx   0x46
```

Same result. So it looks like I'll have to resort to the old reliable : guessing one character at a time.

After hours I finally found it :

`wh47 15 7h3 r1ck 457l3y p4r4d0x?`

Fun.