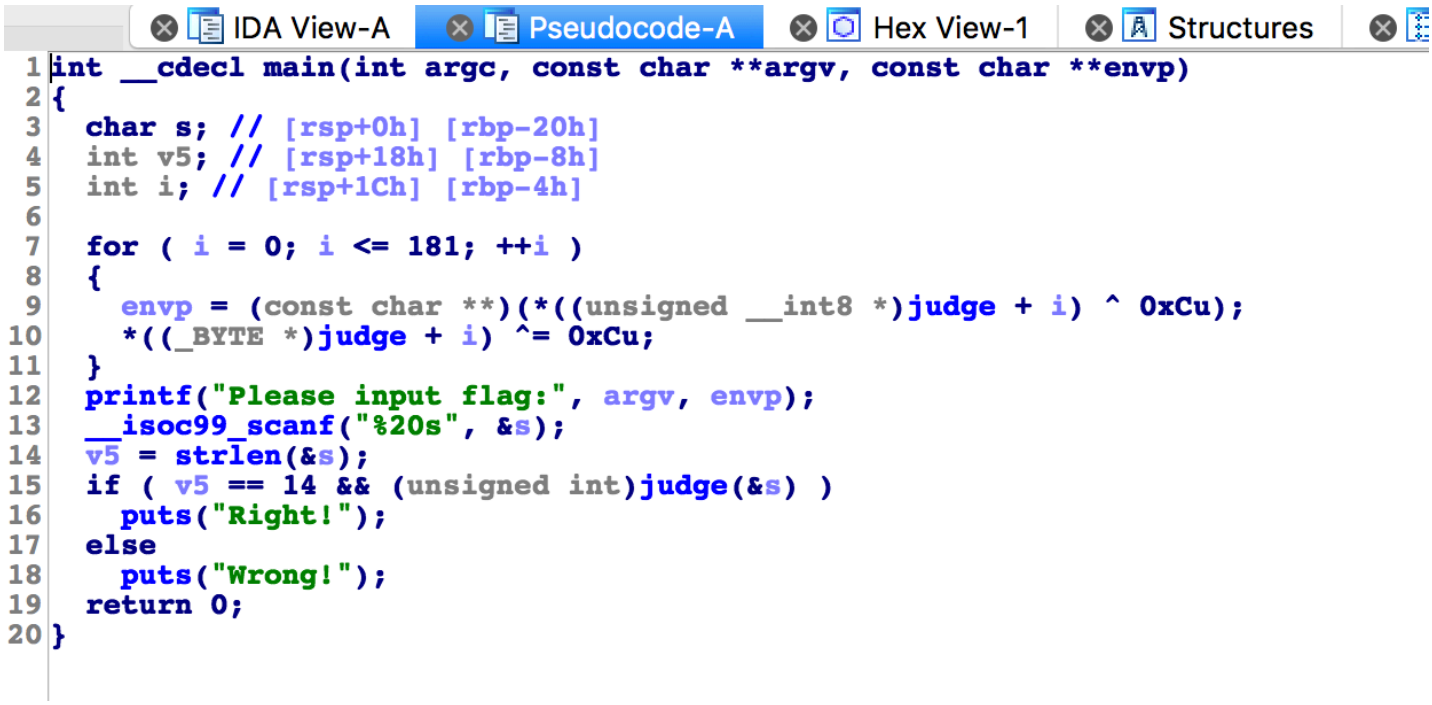
re0

64位的elf

查看main 函数：



我们发现在函数的开头有一个for循环 着一个对judge函数进行加密的 过程。

我们发现 judge函数对我们的逆向关键的函数。

所以 首先要做的就是 解密 judge 函数。

代码如下：

```
import ida_bytes

buf = ida_bytes.get_bytes(0x600B00, 182)

a = ""
for i in xrange(0, len(buf)):
    a += chr(0xC ^ ord(buf[i]))
ida_bytes.patch_bytes(0x600B00, a)
```

将下面的这段代码 复制到 下图的位置中：

然后回车 就可以解密judge函数：

变化如下图所示：

前：

```
ta:0000000000600B00 judge           proc far                ; CODE XREF: main+80↑p
ta:0000000000600B00                                         ; DATA XREF: main+16↑r ...
ta:0000000000600B00                 pop     rcx
ta:0000000000600B01                 test    ecx, r13d
ta:0000000000600B04                 test    [rcx-2Ch], r14d
ta:0000000000600B08                 retf    0EC49h
ta:0000000000600B08 judge           endp ; sp-analysis failed
ta:0000000000600B08
ta:0000000000600B08 ; ---------------------------------------------------------------------------
ta:0000000000600B0B                 db  6Ah ; j
ta:0000000000600B0C                 db 0CAh
ta:0000000000600B0D                 db  49h ; I
ta:0000000000600B0E                 db 0EDh
ta:0000000000600B0F                 db  61h ; a
ta:0000000000600B10                 db 0CAh
ta:0000000000600B11                 db  49h ; I
ta:0000000000600B12                 db 0EEh
ta:0000000000600B13                 db  6Fh ; o
ta:0000000000600B14                 db 0CAh
ta:0000000000600B15                 db  49h ; I
ta:0000000000600B16                 db 0EFh
ta:0000000000600B17                 db  68h ; h
ta:0000000000600B18                 db 0CAh
ta:0000000000600B19                 db  49h ; I
ta:0000000000600B1A                 db 0E8h
ta:0000000000600B1B                 db  73h ; s
ta:0000000000600B1C                 db 0CAh
ta:0000000000600B1D                 db  49h ; I
ta:0000000000600B1E                 db 0E9h
ta:0000000000600B1F                 db  67h ; g
ta:0000000000600B20                 db 0CAh
ta:0000000000600B21                 db  49h ; I
ta:0000000000600B22                 db 0EAh
ta:0000000000600B23                 db  3Bh ; ;
ta:0000000000600B24                 db 0CAh
ta:0000000000600B25                 db  49h ; I
ta:0000000000600B26                 db 0EBh
ta:0000000000600B27                 db  68h ; h
ta:0000000000600B28                 db 0CAh
```

后：

```
.data:0000000000600B00 judge           proc far                ; CODE XREF: main+80↑p
.data:0000000000600B00                                         ; DATA XREF: main+16↑r ...
.data:0000000000600B00                 push    rbp
.data:0000000000600B01                 mov     rbp, rsp
.data:0000000000600B04                 mov     [rbp-28h], rdi
.data:0000000000600B08                 mov     byte ptr [rbp-20h], 66h
.data:0000000000600B08 judge           endp ; sp-analysis failed
.data:0000000000600B08
.data:0000000000600B0C                 mov     byte ptr [rbp-1Fh], 6Dh
.data:0000000000600B10                 mov     byte ptr [rbp-1Eh], 63h
.data:0000000000600B14                 mov     byte ptr [rbp-1Dh], 64h
.data:0000000000600B18                 mov     byte ptr [rbp-1Ch], 7Fh
.data:0000000000600B1C                 mov     byte ptr [rbp-1Bh], 6Bh
.data:0000000000600B20                 mov     byte ptr [rbp-1Ah], 37h
.data:0000000000600B24                 mov     byte ptr [rbp-19h], 64h
.data:0000000000600B28                 mov     byte ptr [rbp-18h], 3Bh
.data:0000000000600B2C                 mov     byte ptr [rbp-17h], 56h
.data:0000000000600B30                 mov     byte ptr [rbp-16h], 60h
.data:0000000000600B34                 mov     byte ptr [rbp-15h], 3Bh
.data:0000000000600B38                 mov     byte ptr [rbp-14h], 6Eh
.data:0000000000600B3C                 mov     byte ptr [rbp-13h], 70h
.data:0000000000600B40                 mov     dword ptr [rbp-4], 0
.data:0000000000600B47                 jmp     short loc_600B71
.data:0000000000600B49 ; ---------------------------------------------------------------------------
.data:0000000000600B49
.data:0000000000600B49 loc_600B49:                             ; CODE XREF: .data:0000000000600B75↓j
.data:0000000000600B49                 mov     eax, [rbp-4]
.data:0000000000600B4C                 movsxd  rdx, eax
.data:0000000000600B4F                 mov     rax, [rbp-28h]
.data:0000000000600B53                 add     rax, rdx
.data:0000000000600B56                 mov     edx, [rbp-4]
.data:0000000000600B59                 movsxd  rcx, edx
.data:0000000000600B5C                 mov     rdx, [rbp-28h]
.data:0000000000600B60                 add     rdx, rcx
.data:0000000000600B63                 movzx   edx, byte ptr [rdx]
.data:0000000000600B66                 mov     ecx, [rbp-4]
.data:0000000000600B69                 xor     edx, ecx
.data:0000000000600B6B                 mov     [rax], dl
```

解密之后， judge函数得到正确的解析

但是我们发现 judge的结束地方并不正确 。

接下来修改judge的结束位置。

在data end的地方 按 `e` yes

然后 选中judge函数的开头 按 `u` 然后再按 `p`

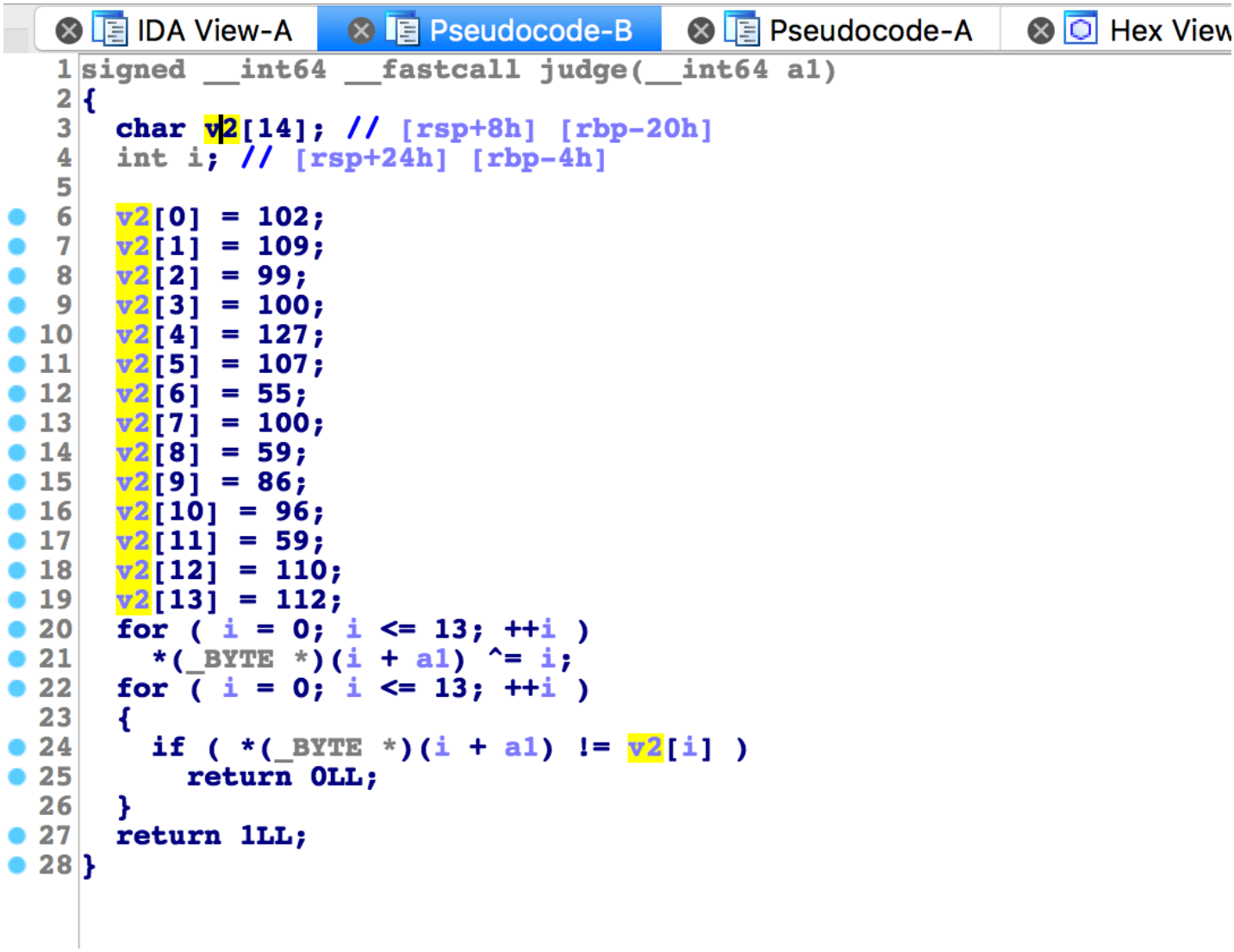
这样就可以反编译 judge函数了。

IDA View-A | Pseudocode-B | Pseudocode-A | Hex V

```
1  signed __int64 __fastcall judge(__int64 a1)
2  {
3    char v2; // [rsp+8h] [rbp-20h]
4    char v3; // [rsp+9h] [rbp-1Fh]
5    char v4; // [rsp+Ah] [rbp-1Eh]
6    char v5; // [rsp+Bh] [rbp-1Dh]
7    char v6; // [rsp+Ch] [rbp-1Ch]
8    char v7; // [rsp+Dh] [rbp-1Bh]
9    char v8; // [rsp+Eh] [rbp-1Ah]
10   char v9; // [rsp+Fh] [rbp-19h]
11   char v10; // [rsp+10h] [rbp-18h]
12   char v11; // [rsp+11h] [rbp-17h]
13   char v12; // [rsp+12h] [rbp-16h]
14   char v13; // [rsp+13h] [rbp-15h]
15   char v14; // [rsp+14h] [rbp-14h]
16   char v15; // [rsp+15h] [rbp-13h]
17   int i; // [rsp+24h] [rbp-4h]
18
19   v2 = 102;
20   v3 = 109;
21   v4 = 99;
22   v5 = 100;
23   v6 = 127;
24   v7 = 107;
25   v8 = 55;
26   v9 = 100;
27   v10 = 59;
28   v11 = 86;
29   v12 = 96;
30   v13 = 59;
31   v14 = 110;
32   v15 = 112;
33   for ( i = 0; i <= 13; ++i )
34     *(_BYTE *)(i + a1) ^= i;
35   for ( i = 0; i <= 13; ++i )
36   {
37     if ( *(_BYTE *)(i + a1) != *(&v2 + i) )
38       return 0LL;
39   }
40   return 1LL;
41 }
```

可以看到 有很多char型的参数 其实是数组 我们进行修改 可以让ida更好的反编译，在 v2的地方按 `y` 在弹出的对话框里 输入v2[14] 因为有14个 变量。

可以看到变成这样：



让数组v2 显示成字符 按 `r`

可以看到 v2 是字符串：fmcd\x7fk7d;V\x60;np

我们看到了变换方式 这个过程可逆：

代码如下：

```
flag_enc="fmcd\x7fk7d;V\x60;np"
flag=""
for i in range(len(flag_enc)):
    c=flag_enc[i]
    flag+=chr(ord(c)^i)

print flag
```