CSAPP实验-二进制炸弹writeup

原创

y4ung 于 2019-07-15 15:44:38 发布

分类专栏： ctf 文章标签： ctf reverse-engineering

本文链接：https://blog.csdn.net/qq_35056292/article/details/95973285

0x01 题目概述

二进制炸弹是《深入理解计算机系统》的一个课程实验。

给定一个二进制文件bomb，及其主程序bomb.c文件，运行二进制文件bomb，一共有6关，用户需要通过6个输入来避免炸弹的爆炸。

我们需要通过对二进制文件进行逆向分析，得到能避开炸弹爆炸的合理的输入。

0x02 涉及知识点

汇编语言的基础，逆向分析工具的使用。

0x03 实验环境

Ubuntu16.04LTS，IDA Pro 7.0，gdb

0x04 解题思路

phase_1

1. 首先读取了用户输入的第一个字符串,保存到rax寄存器中,并通过"mov rdi, rax"将字符串的值赋值给rdi寄存器,作为后面调用函数phase_1()的第一个参数.

```
.text:0000000000400E19                call    initialize_bomb
.text:0000000000400E1E                mov     edi, offset s   ; "Welcome to my fiendish little bomb. You"...
.text:0000000000400E23                call    _puts
.text:0000000000400E28                mov     edi, offset aWhichToBlowYou ; "which to blow yourself up. Have a n
.text:0000000000400E2D                call    _puts
.text:0000000000400E32                call    read_line
.text:0000000000400E37                mov     rdi, rax
.text:0000000000400E3A                call    phase_1
.text:0000000000400E3F                call    phase_defused
.text:0000000000400E44                mov     edi, offset aPhase1Defused_ ; "Phase 1 defused. How about the next
.text:0000000000400E49                call    _puts
.text:0000000000400E4E                call    read_line
.text:0000000000400E53                mov     rdi, rax
.text:0000000000400E56                call    phase_2
.text:0000000000400E5B                call    phase_defused
```

用gdb运行bomb: gdb bomb

为了解出第一个字符串,在phase_1()函数处打断点: b phase_1, 然后运行程序 r. 接下来随便输入一个字符串用于调试, 比如字符串”da”;然后此时进入了phase_1()函数,输入disass查看其汇编代码:

```
(gdb) b phase_1
Breakpoint 1 at 0x400ee0
(gdb) r
Starting program: /home/huangzhenyang/学习/ctf/二进制炸弹/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
da

Breakpoint 1, 0x0000000000400ee0 in phase_1 ()
(gdb) disass
Dump of assembler code for function phase_1:
=> 0x0000000000400ee0 <+0>:     sub    rsp,0x8
   0x0000000000400ee4 <+4>:     mov    esi,0x402400
   0x0000000000400ee9 <+9>:     call   0x401338 <strings_not_equal>
   0x0000000000400eee <+14>:    test   eax,eax
   0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
   0x0000000000400ef2 <+18>:    call   0x40143a <explode_bomb>
   0x0000000000400ef7 <+23>:    add    rsp,0x8
   0x0000000000400efb <+27>:    ret
End of assembler dump.
```


4. 从<strings_not_equal>的名字可知,该函数用于比较两字符串的值,需要两个字符串作为输入. 两个字符串不相等的话则返回1,相等返回0,结果保存在eax中. “test eax,eax”用于检查eax寄存器的值是否为0:如果eax为0,则由于zf标志位为0,因此执行 “je 0x400ef7 <phase_1+23>”,跳过了调用<explode_bomb>的指令代码.

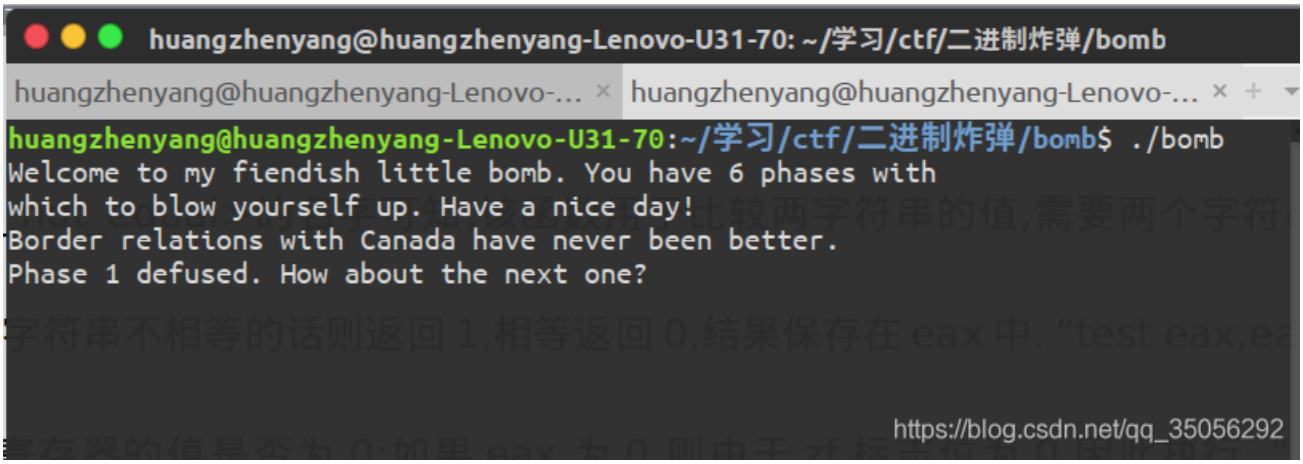
因此, 在这里需要输入的字符串需要与代码中用于比较的字符串相同.

观察phase_1()函数的汇编代码,在调用<strings_not_equal>之前事先通过指令”mov esi,0x402400”, 将内存地址为”0x402400”的值赋值给了esi寄存器,作为函数<strings_not_equal>的参数.

查看内存地址为”0x402400”的值:

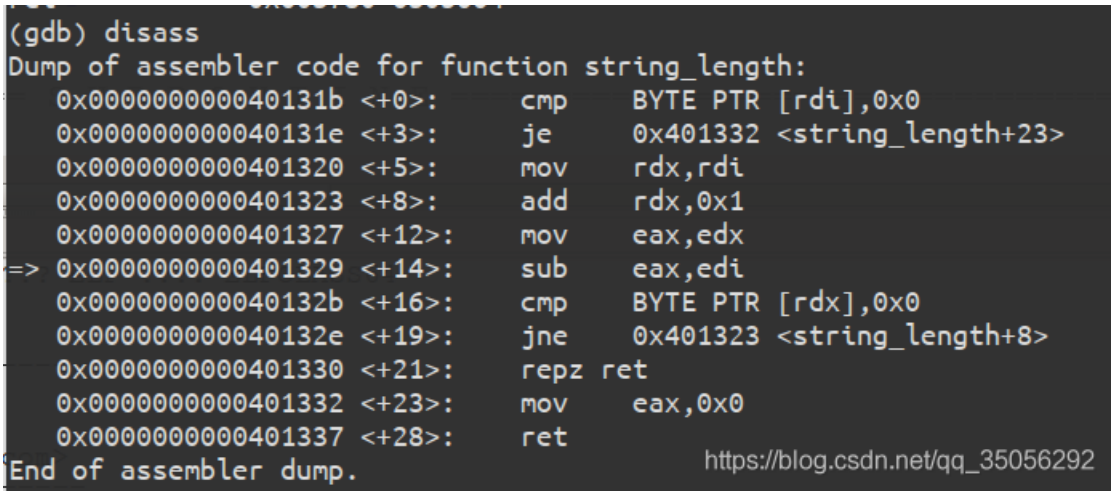
```
(gdb) x/s 0x402400
0x402400:       "Border relations with Canada have never been better."
```

7. 进行测试, 解决phase_1:



Answer1: Border relations with Canada have never been better.

string_length解析



比较rdi存放的字符串地址的内容是不是'\0', 是的话则跳转到0x401332: 将eax赋值为0; 否则的话:

把rdi 也就是用户的输入字符串赋值给rdx, 然后将寄存器rdx中存放的字符串地址+1, 原来是0x6037800,对应的字符串内容为"da"(也就是我一开始输入的测试值),现在变成了0x6037801,对应的字符串内容为"a".

"mov eax, edx" 和 "mov eax, rdx"一样,不过rdx是64位的寄存器

"sub eax, edi" 也就是将eax(地址加了1之后的rdx, 即地址加了1之后的rdi)减去edi, 结果为1并赋值给eax

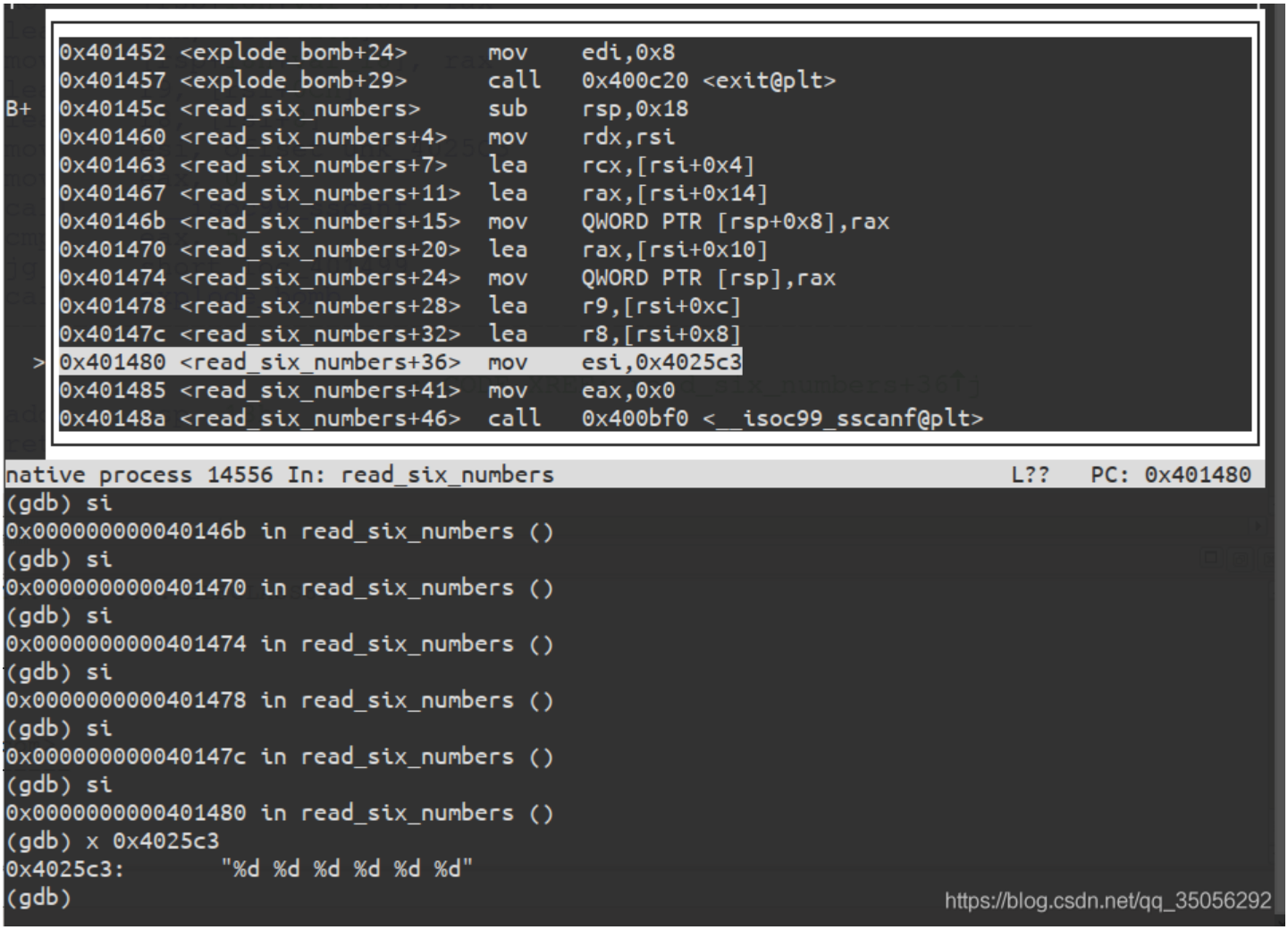
比较此时rdx对应的是不是'\0',是的话结束循环,否则继续循环; 下一次循环中eax就会为2,以此类推

phase_2

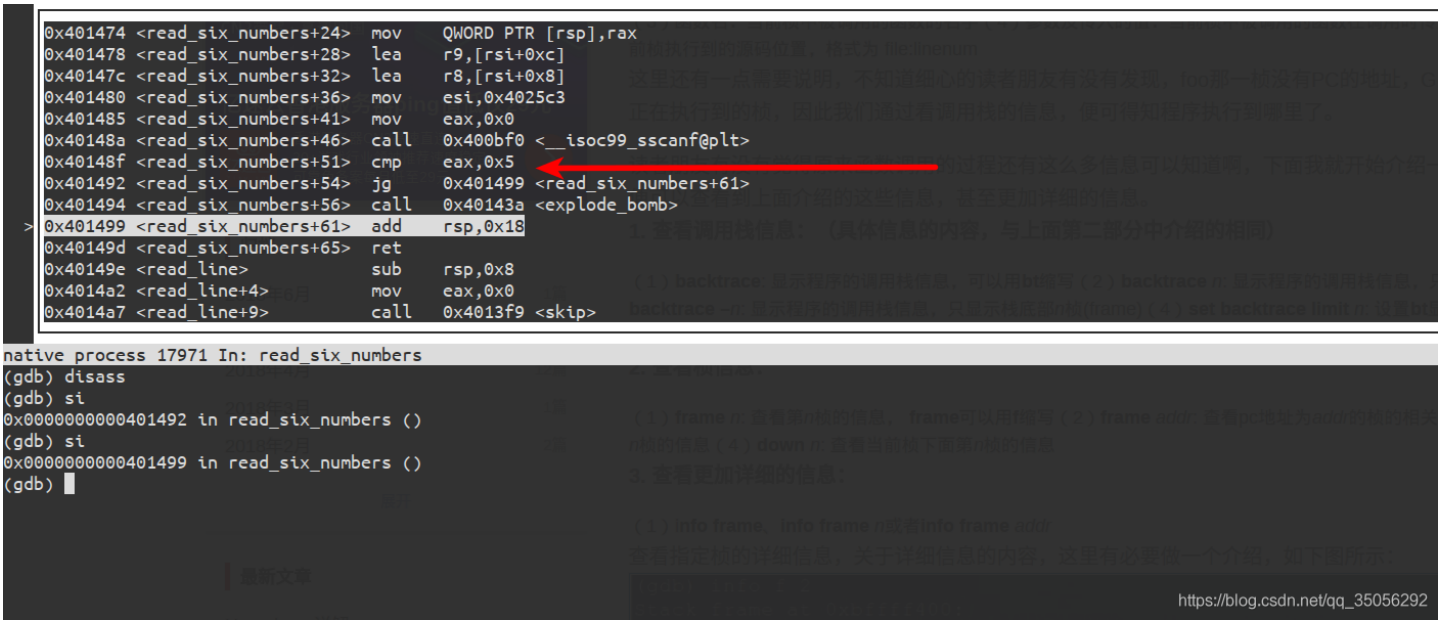
1. 使用gdb进行动态调试, 由phase_2调用的函数<read_six_numbers>可知需要用户输入6个数字. 因此, 先随便输入6个数字, 比如:"1 2 3 4 5 6".

执行到下图的位置时发现有个内存地址, 打印出其中的内容发现是scanf的格式字符串.

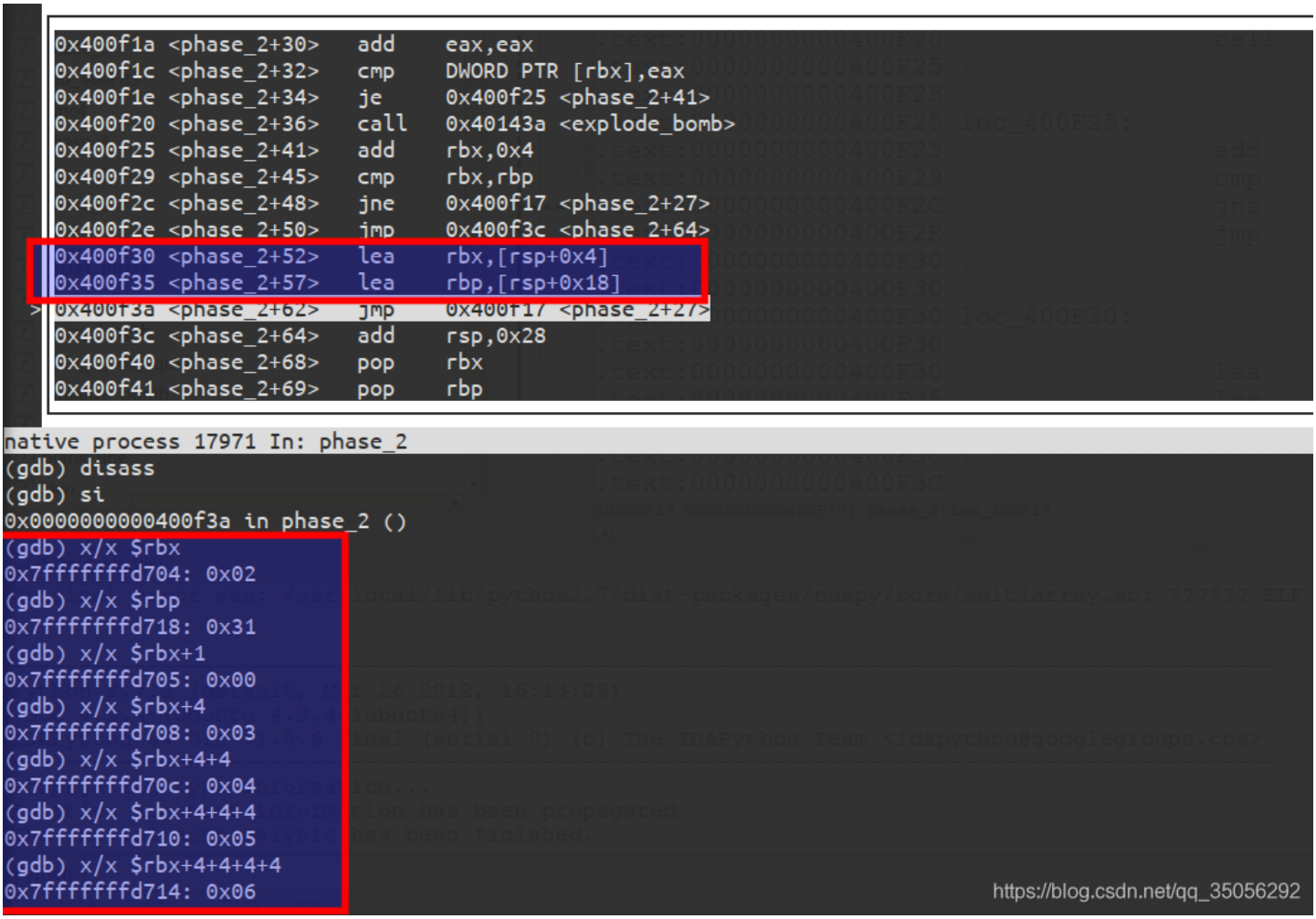
可以知道, 需要输入6个数字, 并且是以空格分隔开的:

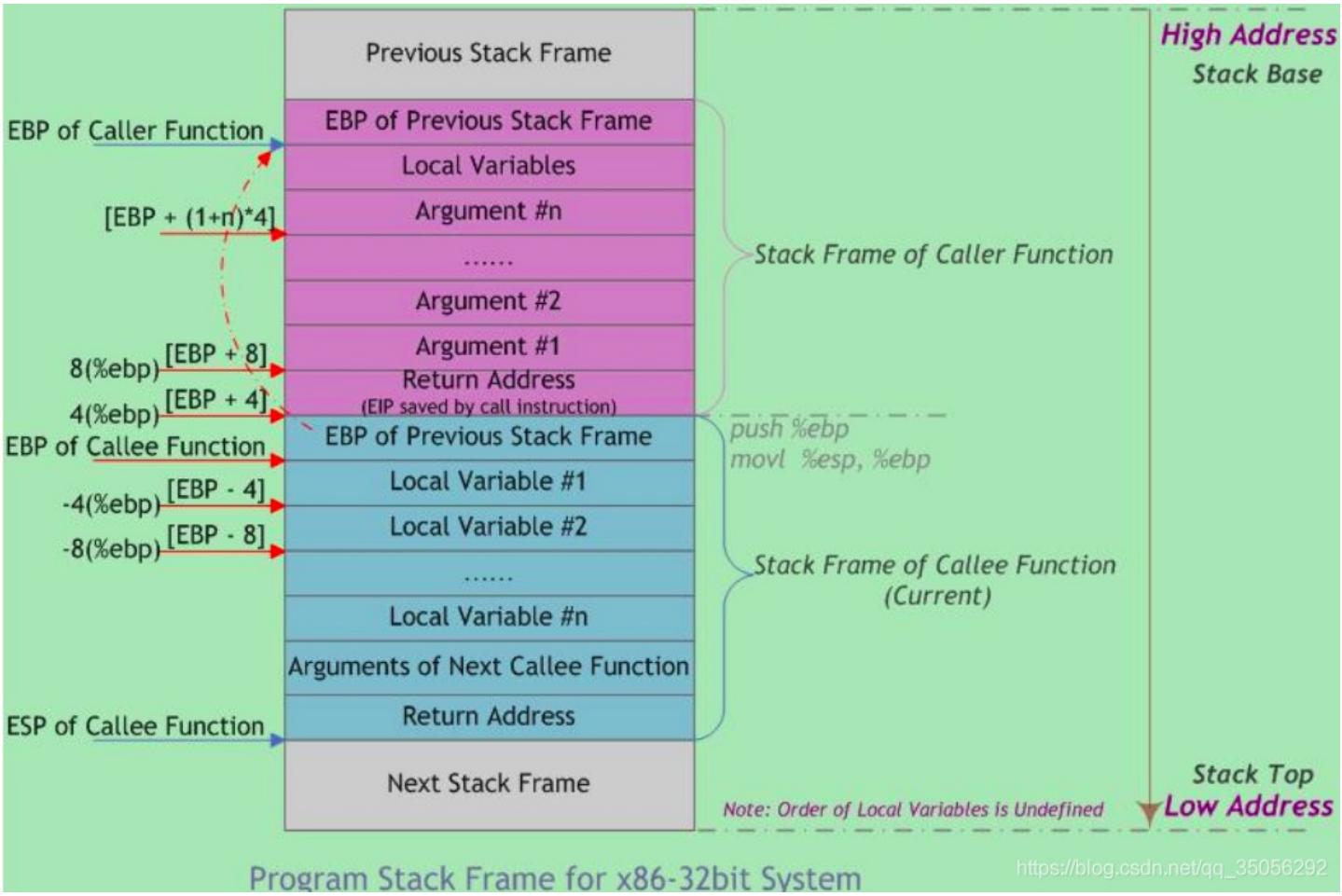


2. 执行完scanf函数以后会有一个判断, 只要输入按要求来就不会引爆炸弹. 这里eax是读取的数据个数, 如果比5大,则跳转到地址0x401499, 即不执行<explode_bomb>函数.



3. 继续执行phase_2()函数. rsp为phase_2() 函数的栈顶指针, 由函数调用栈可知, 指向的是第一个参数.





因此, [rsp] 至 [rsp+0x14] 就是用户传入的数字参数.

一个数字为int=4字节. 0-3为第一个参数, 4-7为第二个参数, 8-11为第三个参数,12-15为第四个参数, 16-20为第五个参数,21-23为第六个参数

由于我输入的是”1 2 3 4 5 6”, 画出来的参数在函数栈中的表示如下:

地址以及存储的值
rbp rsp+24
rsp+20 6
rsp+16 5
rsp+12 4
rsp+8 3
rbx rsp+4 2
rsp 1

4. 接下来在IDA中查看会比较清晰.

rbx-0x4的位置就是rsp的位置,也就是第一个参数的位置; 将[rbx-0x4]指向的数值赋值给了eax后另eax乘以2, 再与第二个参数的值, 也就是[rbx]进行比较, 只有相等才不会触发函数<explode_bomb>. 因此, 第二个参数的值是第一个的两倍.

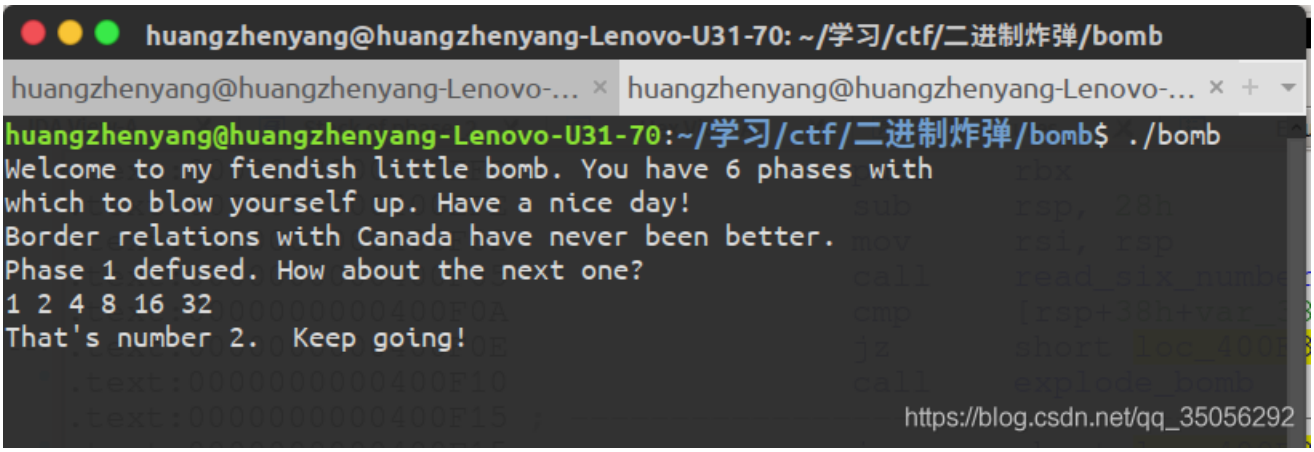
```
.text:0000000000400F17 ; ---------------------------------------------------------------------------
.text:0000000000400F17
.text:0000000000400F17 loc_400F17:                             ; CODE XREF: phase_2+30↓j
.text:0000000000400F17                                         ; phase_2+3E↓j
.text:0000000000400F17                 mov     eax, [rbx-4]    ; rbx-4 就是rsp的位置，把它(当成指针)指
.text:0000000000400F1A                 add     eax, eax        ; 第一个数乘以2
.text:0000000000400F1C                 cmp     [rbx], eax      ; 第二个数的第一个数的两倍
.text:0000000000400F1E                 jz      short loc_400F25 ; 此时rbx为rsp+8
.text:0000000000400F20                 call    explode_bomb
.text:0000000000400F25 ; ---------------------------------------------------------------------------
.text:0000000000400F25
.text:0000000000400F25 loc_400F25:                             ; CODE XREF: phase_2+22↑j
.text:0000000000400F25                 add     rbx, 4          ; 此时rbx为rsp+8
.text:0000000000400F29                 cmp     rbx, rbp        ; rbx=rsp+8, rbp=rsp+24
.text:0000000000400F2C                 jnz     short loc_400F17 ; rbx-4 就是rsp的位置，把它(当成指针)
.text:0000000000400F2E                 jmp     short loc_400F3C
.text:0000000000400F30 ; ---------------------------------------------------------------------------
```

此时rbx为rsp+8，rbx-4为rsp+4.
因此第三个数为第二个数的两倍

在该次判断中, 由于rbx+4以后变成rsp+8, 显然不等于rbp=rsp+24,因此跳到loc_400F17处. 由于此时rbx为rsp+8, rbx-4为rsp+4, 因此第三个数为第二个数的两倍.

以此类推即可知,每个数都是前一个数的两倍,这是一个等比数列.

随便输入一个等比数列, phase_2解决:



Answer2(答案不唯一): 1 2 4 8 16 32

phase_3

1. 同样的方法打断点, 进行调试.
   可以看到, 输入应该为两个数字, 并且以空格隔开:

```
   0x400f3a <phase_2+62>    jmp    0x400f17 <phase_2+27>
   0x400f3c <phase_2+64>    add    rsp,0x28
   0x400f40 <phase_2+68>    pop    rbx
   0x400f41 <phase_2+69>    pop    rbp
   0x400f42 <phase_2+70>    ret
B+ 0x400f43 <phase_3>       sub    rsp,0x18
   0x400f47 <phase_3+4>     lea    rcx,[rsp+0xc]
   0x400f4c <phase_3+9>     lea    rdx,[rsp+0x8]
   0x400f51 <phase_3+14>    mov    esi,0x4025cf
   0x400f56 <phase_3+19>    mov    eax,0x0
   0x400f5b <phase_3+24>    call   0x400bf0 <__isoc99_sscanf@plt>
   0x400f60 <phase_3+29>    cmp    eax,0x1
   0x400f63 <phase_3+32>    jg     0x400f6a <phase_3+39>
   0x400f65 <phase_3+34>    call   0x40143a <explode_bomb>
native process 775 In: phase_3                                  L??   PC: 0x400f51
(gdb) si
0x0000000000400f4c in phase_3 ()
(gdb) si
0x0000000000400f51 in phase_3 ()
(gdb) x/s 0x4025cf
0x4025cf:       "%d %d"
(gdb)
```


在phase_3()函数中, 先将两个参数的值分别赋值给寄存器rdx和rcx, 然后调用scanf函数.

接下来需要注意的地方是, 在箭头处, 先将 [rsp+18h+var_10] 处的值, 也就是rdx处的值, 即参数1与7比较. 如果比7大, 则跳转到地址0x400FAD处, 从地址0x400FAD处的代码可以看到走这个分支的话必然引起炸弹爆炸, 因此第一个参数的值必然小于或等于7.

```
.text:0000000000400F43                 sub     rsp, 18h
.text:0000000000400F47                 lea     rcx, [rsp+18h+var_C] ; 参数2赋值给寄存器rcx
.text:0000000000400F4C                 lea     rdx, [rsp+18h+var_10] ; 参数1赋值给寄存器rdx
.text:0000000000400F51                 mov     esi, offset aDD ; "%d %d"
.text:0000000000400F56                 mov     eax, 0
.text:0000000000400F5B                 call    ___isoc99_sscanf
.text:0000000000400F60                 cmp     eax, 1
.text:0000000000400F63                 jg      short loc_400F6A
.text:0000000000400F65                 call    explode_bomb
.text:0000000000400F6A ; ---------------------------------------------------------------------------
.text:0000000000400F6A
.text:0000000000400F6A loc_400F6A:                             ; CODE XREF: phase_3+20↑j
.text:0000000000400F6A                 cmp     [rsp+18h+var_10], 7
.text:0000000000400F6F                 ja      short loc_400FAD
.text:0000000000400F71                 mov     eax, [rsp+18h+var_10]
.text:0000000000400F75                 jmp     ds:off_402470[rax*8]
.text:0000000000400F7C ; ---------------------------------------------------------------------------
.text:0000000000400F7C
.text:0000000000400F7C loc_400F7C:                             ; DATA XREF: .rodata:off_402470↓o
.text:0000000000400F7C                 mov     eax, 0CFh
.text:0000000000400F81                 jmp     short loc_400FBE
.text:0000000000400F83 ; ---------------------------------------------------------------------------
.text:0000000000400F83
.text:0000000000400F83 loc_400F83:                             ; DATA XREF: .rodata:0000000000402480↓o
.text:0000000000400F83                 mov     eax, 2C3h
```


地址0x400FAD处的代码:

```
.text:0000000000400FAD ; ---------------------------------------------------------------------------
.text:0000000000400FAD
.text:0000000000400FAD loc_400FAD:                             ; CODE XREF: phase_3+2C↑j
.text:0000000000400FAD                 call    explode_bomb
.text:0000000000400FAD phase_3         endp
.text:0000000000400FAD
```

4. 接下来, 将[rsp+18h+var_10] 处的值, 即参数1的值赋值给eax, 并跳转到 “jmp QWORD PTR [rax*8+0x402470] ”, 先假设输入的参数1值为2, 那么跳转的地址就是0x402480, 接下来是跳转到0x400F83处:

```
.rodata:0000000000402470 off_402470      dq offset loc_400F7C    ; DATA XREF: phase_3+32↑r
.rodata:0000000000402478                 dq offset sub_400FB9
.rodata:0000000000402480                 dq offset loc_400F83
.rodata:0000000000402488                 dq offset loc_400F8A
.rodata:0000000000402490                 dq offset loc_400F91
.rodata:0000000000402498                 dq offset loc_400F98
.rodata:00000000004024A0                 dq offset loc_400F9F
.rodata:00000000004024A8                 dq offset loc_400FA6
```

对eax赋值为 `0x2C3` , 跳转到0x400FBE处

```
.text:0000000000400F83 ; ---------------------------------------------------------------------------
.text:0000000000400F83
.text:0000000000400F83 loc_400F83:                             ; DATA XREF: .rodata:0000000000402480↓o
.text:0000000000400F83                 mov     eax, 2C3h
.text:0000000000400F88                 jmp     short loc_400FBE
.text:0000000000400F8A ; ---------------------------------------------------------------------------
```

可以看出,参数2的值必须等于eax的值

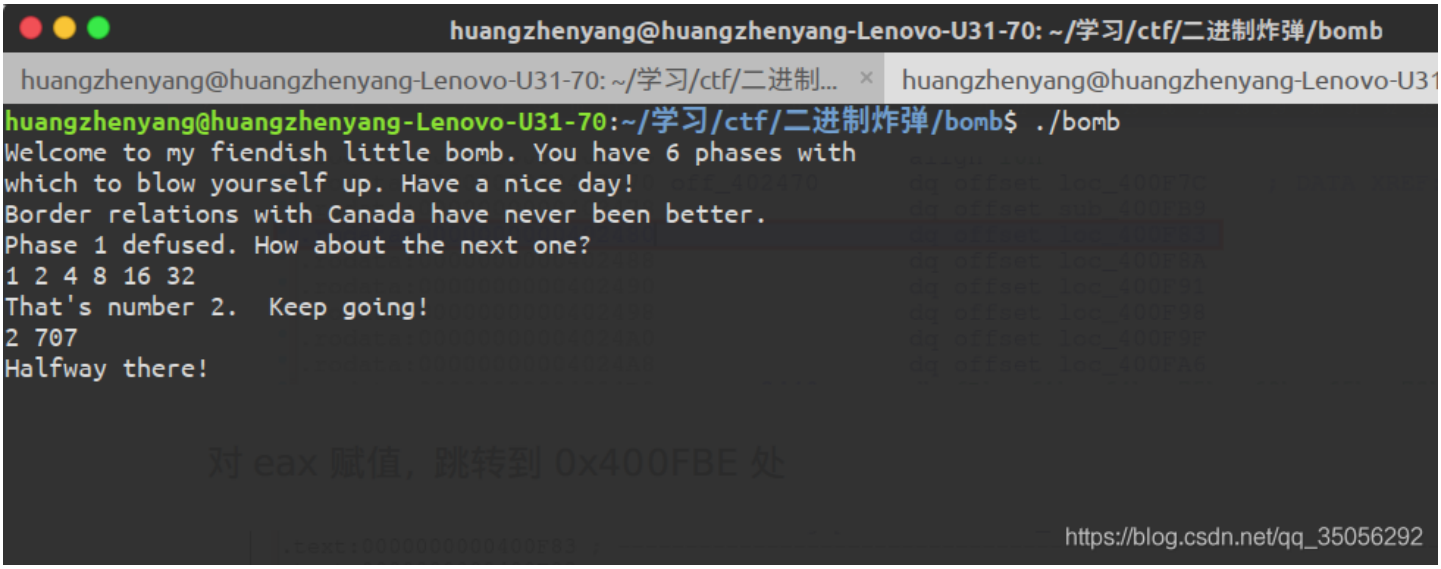
```
.text:0000000000400FBE loc_400FBE:                             ; CODE XREF: phase_3+3E↑j
.text:0000000000400FBE                                         ; phase_3+45↑j ...
.text:0000000000400FBE                 cmp     eax, [rsp+18h+var_C]
.text:0000000000400FC2                 jz      short loc_400FC9
.text:0000000000400FC4                 call    explode_bomb
```

[rsp+18h+var_C] 即 [rsp+0xC] 也就是参数2

可以观察到, 对于参数1的不同取值, 参数2的值也应该与程序中最终对eax赋的值相同才行.

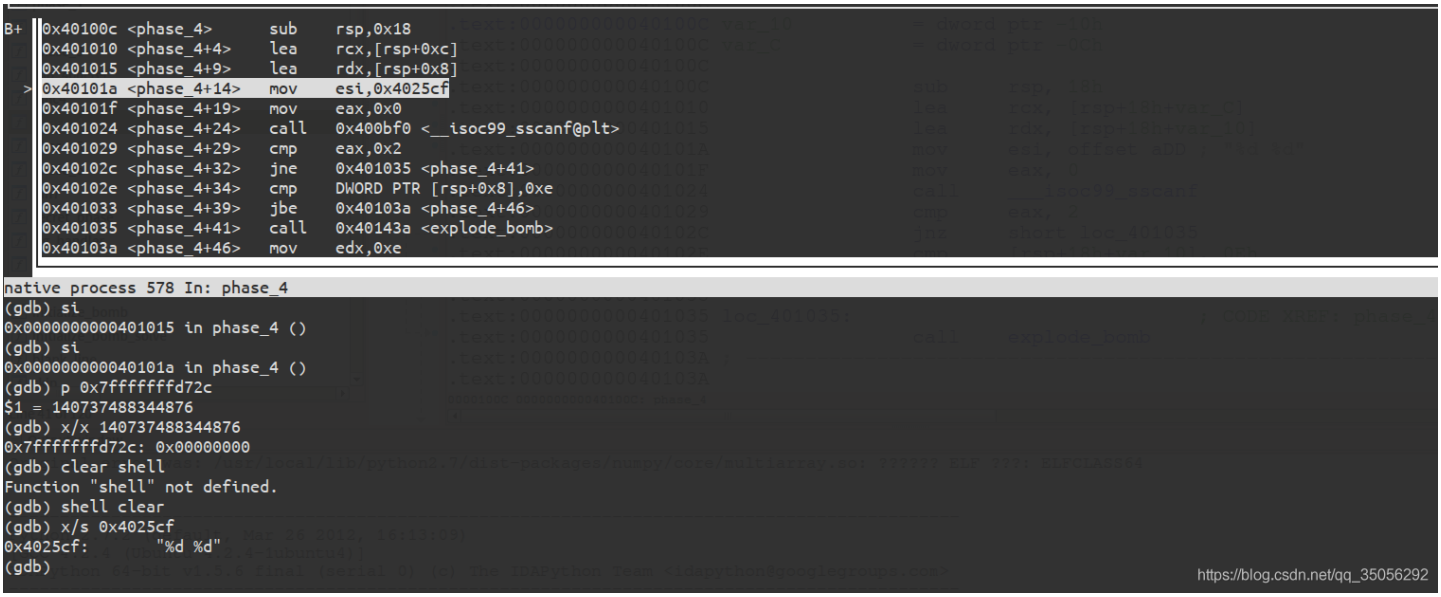
因此, phase_3 的答案可以测试小于7非负数作为参数1, 并走通程序来判断参数2的值.

这里选择输入数据为: 2 707 , 其中 707 即 0x2C3 可以看到测试通过

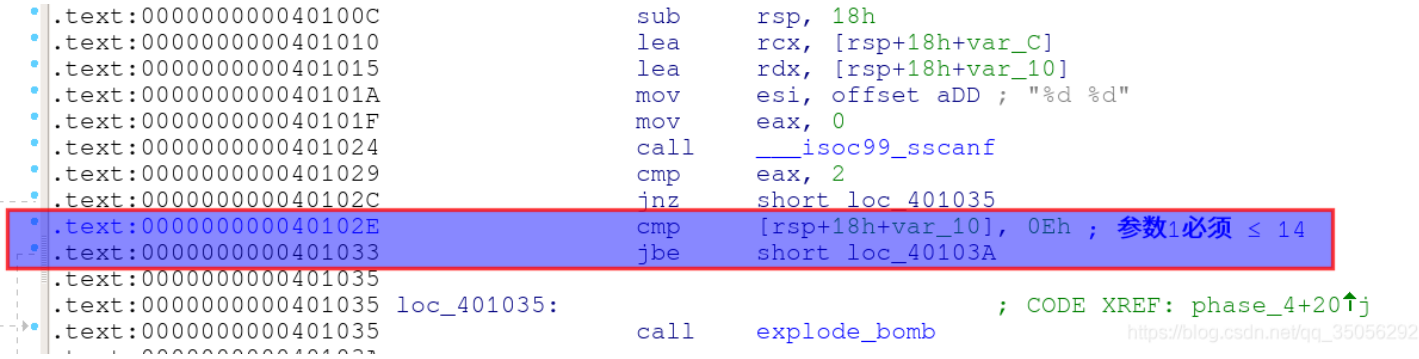


phase_4

1. 同样的方法打断点, 进行调试. 和phase_3一样, 输入也为两个数字, 空格隔开



2. 可以看到, 参数1必须 ≤ 14才不会引爆炸弹



3. 再往下看, 先将参数1的值保存在寄存器edi中.

然后phase_4()函数先调用了func4()函数, 然后判断func4()函数的返回值(保存在寄存器eax中), 如果eax不是0的话, 则会跳转到引爆炸弹的地方. 因此函数func4()返回值eax必须是0.

接下来则判断第二个参数是否为0, 如果不是0的话则引爆炸弹, 因此第二个参数已经可以确定为0.

```
.text:000000000040103A ; ---------------------------------------------------------------------------
.text:000000000040103A
.text:000000000040103A loc_40103A:                             ; CODE XREF: phase_4+27↑j
.text:000000000040103A                 mov     edx, 0Eh
.text:000000000040103F                 mov     esi, 0
.text:0000000000401044                 mov     edi, [rsp+18h+var_10]
.text:0000000000401048                 call    func4
.text:000000000040104D                 test    eax, eax
.text:000000000040104F                 jnz     short loc_401058 ; eax不是0的话,则会跳转到引爆炸弹的
.text:0000000000401051                 cmp     [rsp+18h+var_C], 0 ; 第二个参数[rsp+18h+var_C] 必须为0
.text:0000000000401056                 jz      short loc_40105D
.text:0000000000401058
.text:0000000000401058 loc_401058:                             ; CODE XREF: phase_4+43↑j
.text:0000000000401058                 call    explode_bomb
.text:000000000040105D ; ---------------------------------------------------------------------------
```


4. 接下来要做的就是分析func4()函数

```
.text:0000000000400FCE func4           proc near               ; CODE XREF: func4+1B↓p
.text:0000000000400FCE                                         ; func4+30↓p ...
.text:0000000000400FCE                 sub     rsp, 8
.text:0000000000400FD2                 mov     eax, edx        ; eax<-edx = 14
.text:0000000000400FD4                 sub     eax, esi        ; eax=eax-esi=14-0=14
.text:0000000000400FD6                 mov     ecx, eax        ; ecx<-eax=14
.text:0000000000400FD8                 shr     ecx, 1Fh        ; ecx逻辑右移31位 = 0
.text:0000000000400FDB                 add     eax, ecx        ; eax= eax+ecx = 14+0 = 14
.text:0000000000400FDD                 sar     eax, 1          ; eax算术右移1位=7
.text:0000000000400FDF                 lea     ecx, [rax+rsi]  ; ecx=[7+0*1]=[7]
.text:0000000000400FE2                 cmp     ecx, edi        ; cmp ecx=7, 参数1
.text:0000000000400FE4                 jle     short loc_400FF2 ; ecx=7 ≤ 参数1 则跳转
.text:0000000000400FE6                 lea     edx, [rcx-1]    ; edx<-[rcx-1] = 6
.text:0000000000400FE9                 call    func4
.text:0000000000400FEE                 add     eax, eax
.text:0000000000400FF0                 jmp     short loc_401007
.text:0000000000400FF2 ; ---------------------------------------------------------------------------
.text:0000000000400FF2
.text:0000000000400FF2 loc_400FF2:                             ; CODE XREF: func4+16↑j
.text:0000000000400FF2                 mov     eax, 0
.text:0000000000400FF7                 cmp     ecx, edi
.text:0000000000400FF9                 jge     short loc_401007
.text:0000000000400FFB                 lea     esi, [rcx+1]
.text:0000000000400FFE                 call    func4
.text:0000000000401003                 lea     eax, [rax+rax+1]
.text:0000000000401007
.text:0000000000401007 loc_401007:                             ; CODE XREF: func4+22↑j
.text:0000000000401007                                         ; func4+2B↑j
.text:0000000000401007                 add     rsp, 8
.text:000000000040100B                 retn
.text:000000000040100B func4           endp
.text:000000000040100B
```


框框框住的为两个关键的条件判断. ecx<=edi则跳转, 以及ecx>=edi则跳转.
接下来有两种方法

a) 第一种是将汇编代码写成高级语言,直接执行. 因为参数1的范围已知. 最终结果为, 参数1的取值可以有:0, 1, 3, 7

```python
# -*- coding:utf-8 -*-

edx = 14
esi = 0
edi = 8  # param 1
eax = 0
ecx = 0


def func4():
    global edx, esi, edi, eax, ecx
    eax = edx
    eax = eax - esi
    ecx = eax
    ecx = ecx >> 31
    eax = eax + ecx
    eax = eax >> 1
    ecx = eax + esi

    if ecx <= edi:
        eax = 0
        if ecx >= edi:  # ecx == edi
            return eax
        else:           # ecx < edi
            esi = ecx + 1
            func4()
            eax = eax*2 + 1
            return eax
    else:               # ecx > edi
        edx = ecx - 1
        func4()
        eax = eax*2
        return eax


if __name__ == "__main__":
    # for edi in range(7, 15):
    edi = 7
    res = func4()
    print(edi, ": ", res)
```

b) 第二种方法, 分析程序的逻辑.

可以看到, func4()中有三个条件判断很关键:

从地址0x400fd2到0x 400fdf:

从高地址往低地址逆着推:

> ecx = eax + esi
> = eax>>1 + esi = eax / 2 + esi
> = (eax + ecx) / 2 + esi = eax / 2 + esi # ecx是eax的符号, 这里都是正数ecx为0
> = (eax - esi) / 2 + esi = (eax + esi) / 2
> = (edx + esi) / 2

a) 情况1: 如果ecx > edi，则 400fe6 处代码将 ecx-1 赋给edx,接着递归调用func4函数。eax = eax + eax

b) 情况2: 如果ecx == edi，则将eax赋值为0并返回。

c) 情况3: 如果ecx < edi，则 400ffb 处代码将 ecx+1 赋给esi,接着递归调用func4函数。 eax = eax + eax + 1

这样一分析的话, 可以看到这个过程像二分查找. esi初始为0, 为左边界, edx为右边界, ecx为区间的中间值, edi为参数1.

情况2能保证最后eax为0;

情况1中会将eax= eax+ eax. 在情况3中, 会将eax = eax + eax + 1. 因此在递归过程中不能出现ecx<edi的情况, 如果出现了, 那么eax =eax * 2 + 1, eax必不等于0.

由于ecx为区间的中间值, 那么, 为了能到达情况2另eax=0, 则参数1也就是edi的值必须是ecx在区间变化过程中的值.

按照程序的逻辑来走的话, 区间变化如下:

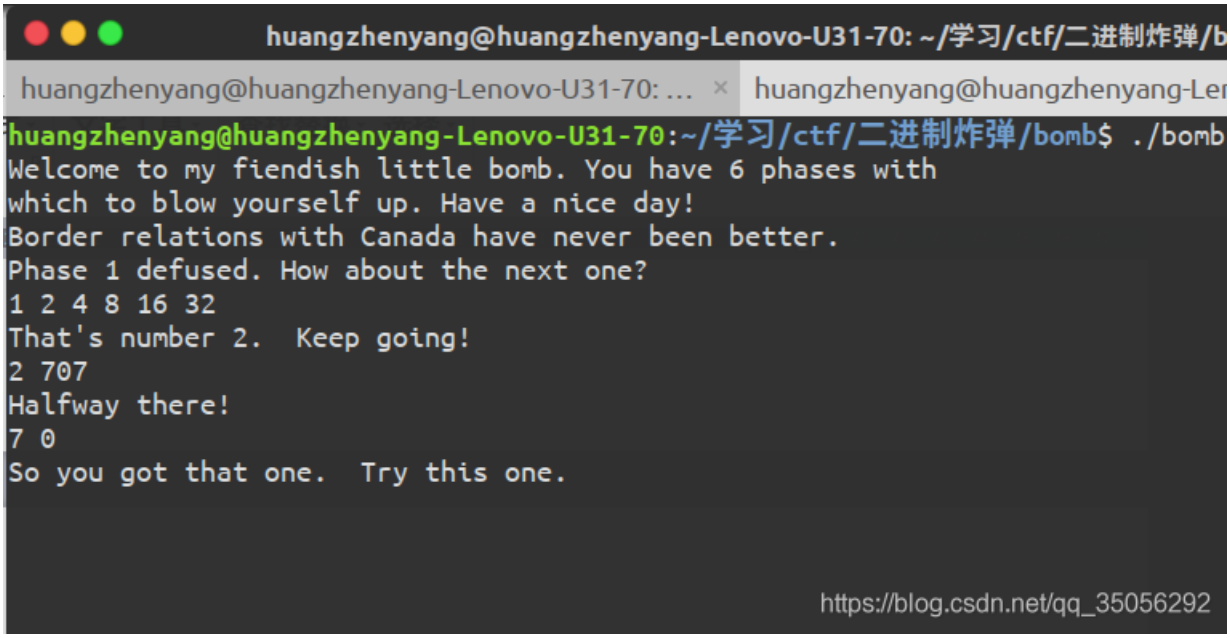
[esi, edx] = [0, 14], ecx=7; edx = ecx-1=6; 如果edi==7, 则此时已经满足情况2的条件;

[esi, edx] = [0, 6], ecx=3; edx = ecx-1=2; 如果edi==3, 则此时已经满足情况2的条件;

[esi, edx] = [0, 2], ecx=1; edx = ecx-1=0; 如果edi==1, 则此时已经满足情况2的条件;

[esi, edx] = [0, 0], ecx=0; edx = ecx-1=-1; 如果edi==0, 则此时已经满足情况2的条件;

因此, 参数1的取值为: 0, 1, 3, 7, 参数2的取值为0. 测试通过

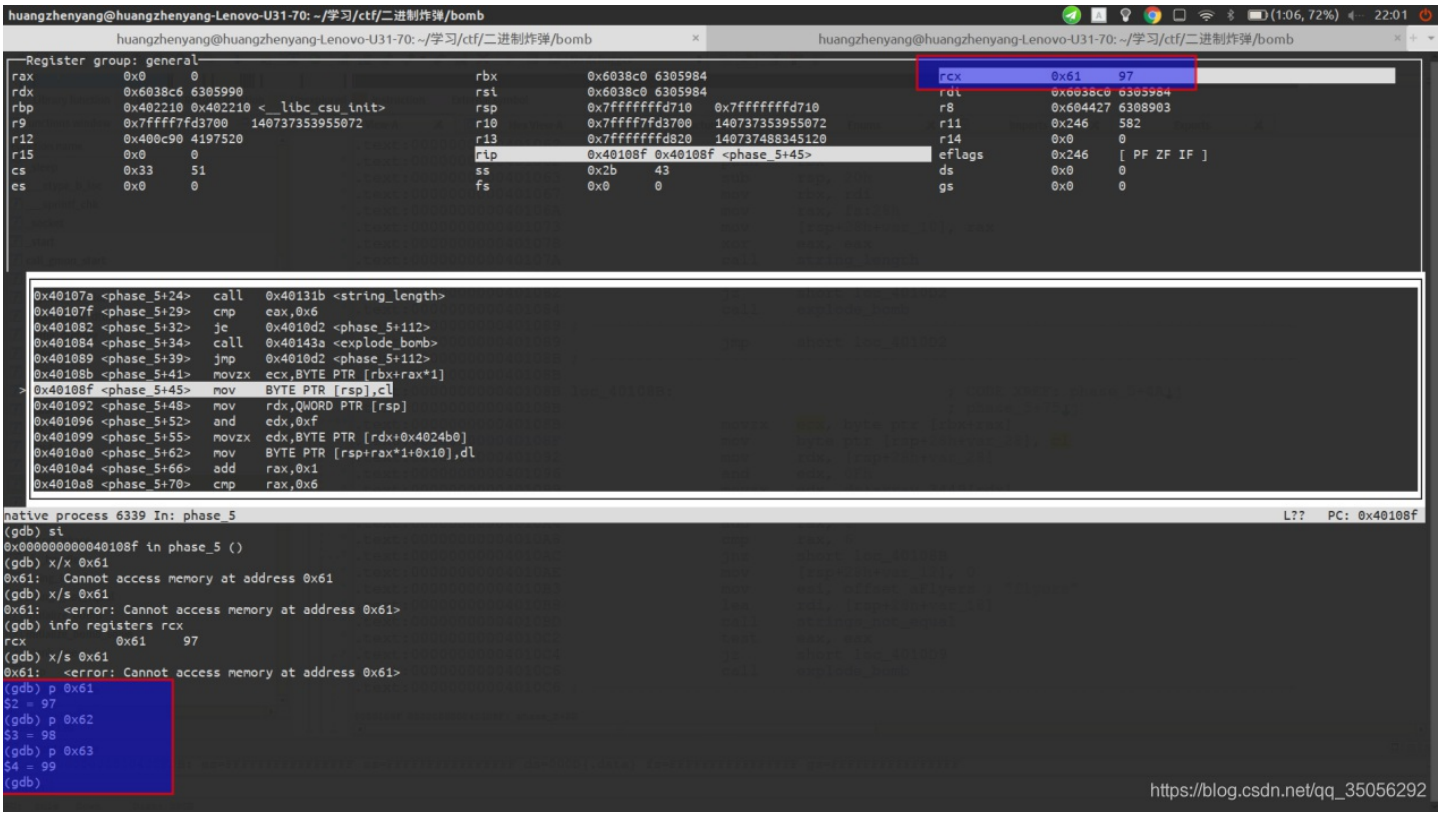


phase_5

1. 由string_length()函数对输入判断的返回值可知, 应该输入长度为6的字符串

```
.text:0000000000401062                 public phase_5
.text:0000000000401062 phase_5         proc near               ; CODE XREF: main+10A↑p
.text:0000000000401062
.text:0000000000401062 var_28          = qword ptr -28h
.text:0000000000401062 var_18          = byte ptr -18h
.text:0000000000401062 var_12          = byte ptr -12h
.text:0000000000401062 var_10          = qword ptr -10h
.text:0000000000401062
.text:0000000000401062                 push    rbx
.text:0000000000401063                 sub     rsp, 20h
.text:0000000000401067                 mov     rbx, rdi
.text:000000000040106A                 mov     rax, fs:28h
.text:0000000000401073                 mov     [rsp+28h+var_10], rax
.text:0000000000401078                 xor     eax, eax
.text:000000000040107A                 call    string_length
.text:000000000040107F                 cmp     eax, 6
.text:0000000000401082                 jz      short loc_4010D2
.text:0000000000401084                 call    explode_bomb
```


2. 输入数据”abcdef”进行测试. 此时地址0x61处也就是寄存器ecx存储的是97, 也就是”a”, 依次打印0x62为”b”, 0x63为”c”.



从地址0x40108B开始到0x4010AC是一个循环, 当寄存器rax的值不等于6 的时候, 会在地址0x4010AC处跳转到0x40108B.

执行完循环以后, 将0x40245e地址处的内容放入esi寄存器中, 打印出来发现是字符串”flyers”. 然后将用户输入的字符串放入rdi寄存器, 打印发现此时已经跟我一开始输入的”abcdef”不是同一个了. 因此, phase_5应该是要构造一个字符串, 使得经过循环以后rdi的值为”flyers”. 最终, 在函数strings_not_equal()中对esi的内容和rdi的内容进行比较

```
0x40108f <phase_5+45>    mov    BYTE PTR [rsp],cl
0x401092 <phase_5+48>    mov    rdx,QWORD PTR [rsp]
0x401096 <phase_5+52>    and    edx,0xf
0x401099 <phase_5+55>    movzx  edx,BYTE PTR [rdx+0x4024b0]
0x4010a0 <phase_5+62>    mov    BYTE PTR [rsp+rax*1+0x10],dl
0x4010a4 <phase_5+66>    add    rax,0x1
0x4010a8 <phase_5+70>    cmp    rax,0x6
0x4010ac <phase_5+74>    jne    0x40108b <phase_5+41>
0x4010ae <phase_5+76>    mov    BYTE PTR [rsp+0x16],0x0
0x4010b3 <phase_5+81>    mov    esi,0x40245e
> 0x4010b8 <phase_5+86>    lea    rdi,[rsp+0x10]
0x4010bd <phase_5+91>    call   0x401338 <strings_not_equal>
0x4010c2 <phase_5+96>    test   eax,eax
native process 6339 In: phase_5
(gdb) si
0x00000000004010ac in phase_5 ()
(gdb) si
0x00000000004010ae in phase_5 ()
(gdb) si
0x00000000004010b3 in phase_5 ()
(gdb) x/s 0x40245e
0x40245e:       "flyers"
(gdb) si
0x00000000004010b8 in phase_5 ()
(gdb) info registers rsp
rsp            0x7fffffffd710   0x7fffffffd710
(gdb) p 0x7fffffffd720
$7 = 140737488344864
(gdb) x/s 0x7fffffffd720
0x7fffffffd720: "aduier"
(gdb)
```


5. 接下来来具体看循环里面的内容. 地址0x4024b0打印出来, 发现是一个字符串, 通过rdx的低四位值来对字符串中的数据进行读取, 最后存放到edx寄存器中.

那么rdx值怎么来的? 溯源上去可以看到是用户输入的字符

```
0x401084 <phase_5+34>    call   0x40143a <explode_bomb>
0x401089 <phase_5+39>    jmp    0x4010d2 <phase_5+112>
0x40108b <phase_5+41>    movzx  ecx,BYTE PTR [rbx+rax*1]
0x40108f <phase_5+45>    mov    BYTE PTR [rsp],cl
0x401092 <phase_5+48>    mov    rdx,QWORD PTR [rsp]
> 0x401096 <phase_5+52>    and    edx,0xf
0x401099 <phase_5+55>    movzx  edx,BYTE PTR [rdx+0x4024b0]
0x4010a0 <phase_5+62>    mov    BYTE PTR [rsp+rax*1+0x10],dl
0x4010a4 <phase_5+66>    add    rax,0x1
0x4010a8 <phase_5+70>    cmp    rax,0x6
0x4010ac <phase_5+74>    jne    0x40108b <phase_5+41>
0x4010ae <phase_5+76>    mov    BYTE PTR [rsp+0x16],0x0
0x4010b3 <phase_5+81>    mov    esi,0x40245e
0x4010b8 <phase_5+86>    lea    rdi,[rsp+0x10]
(gdb) x/s 0x4024b0
0x4024b0 <array.3449>:  "maduiersnfotvbylSo you think you can stop the bomb with
 ctrl-c, do you?"
(gdb)
```

然后再保存到[rsp+rax*1+0x10], 因为rax是从0–5, 因此存放地址就是[rsp+0x10]–[rsp+0x15]

地址0x4010ae处: 退出循环以后,将[rsp+0x16] 置为0, 作为循环生成后字符串的结束标志:'\0'.

然后地址0x4010b3处将字符串"flyers"放入esi寄存器中,用于后续strings_not_equal()函数的字符串比较

8. 然后地址0x4010b8处将[rsp+0x10]放入寄存器rdi中, 用于后续strings_not_equal()函数的字符串比较

所以phase_5的解决方案就是:

从字符串"maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"中, 根据ASCII码与0xF逻辑与操作得到的后四位, 取出"flayers".

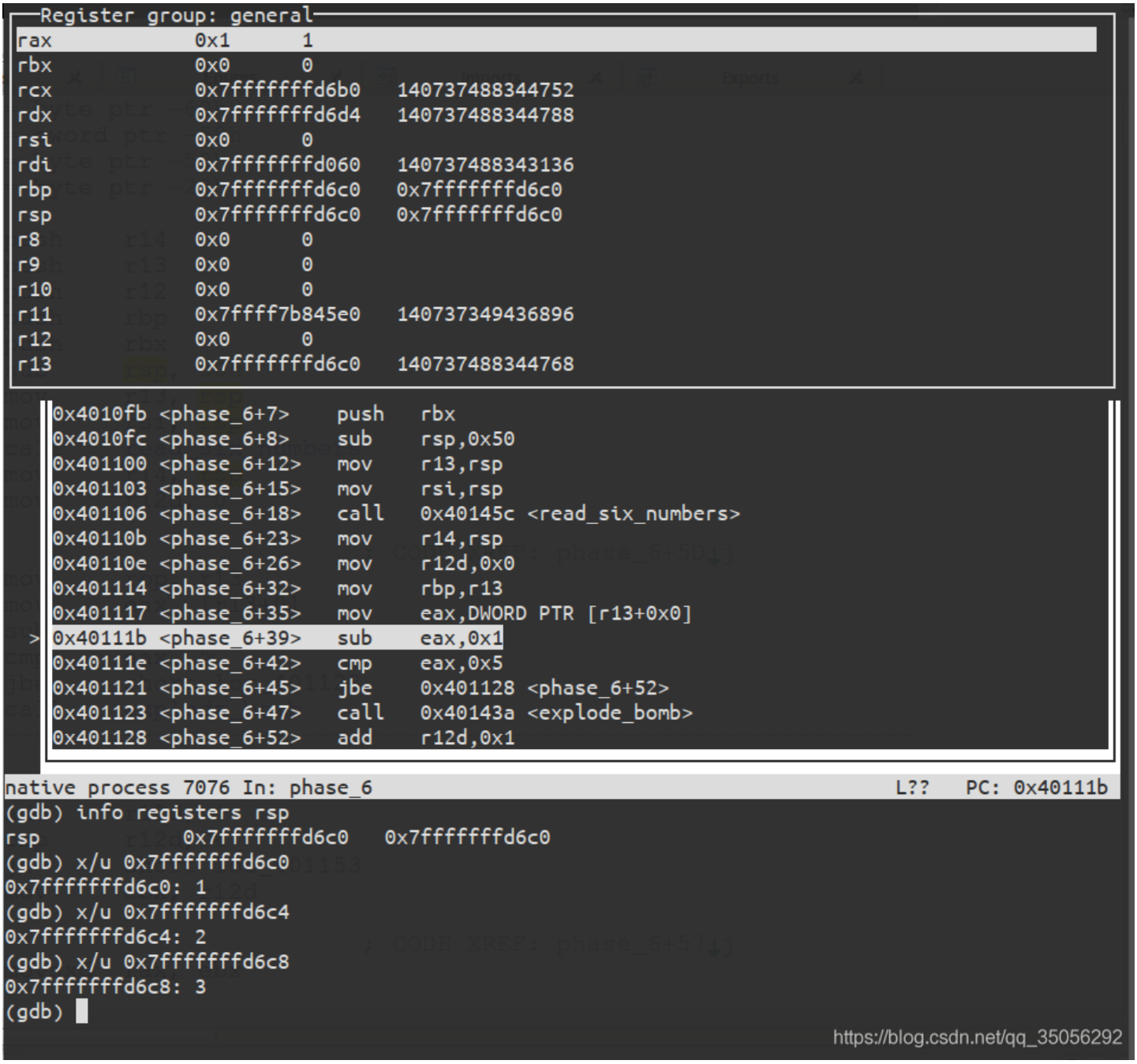
参考: http://ascii.911cha.com/

目标字符	字符串中的索引	索引对应的二进制	可能的取值	取值对应的二进制
f	9	1001	i	0110
l	15	1111	o	0110
y	14	1110	n	0110
e	5	0101	e	0110
r	6	0110	f	0110
s	7	0111	g	0110

所以, 最终可能的一种答案为: ionefg

phase_6

1. 从地址0x401106可以看到, phase_6的输入为6个数字, 输入"1 2 3 4 5 6" 进行测试



则数据从rsp开始存放

地址以及存储的值
rsp+0x14 6
rsp+0x10 5
rsp+0xc 4
rsp+0x8 3
rsp+0x4 2
rsp 1

走完第一遍循环, 发现输入的数字满足两个条件: a. 参数1≤6; b.参数1和参数2,3,4,5,6都不相等

在该循环结束之后, 地址0x40114D处, 对r13中保存的地址+4, 即此时r13保存的为参数2的地址, 在地址0x401151处跳回地址0x401114.

则可以推断出:

a) 输入的6个数字都要≤6

b) 每个数字和后面的数字均不相等

```
.text:0000000000401114 loc_401114:                             ; CODE XREF: phase_6+5D↓j
.text:0000000000401114                 mov     rbp, r13        ; rsp赋值给rbp, rbp=r13=1=1+4=5
.text:0000000000401117                 mov     eax, [r13+0]    ; [r13] 赋值给eax, 第一个参数的值给eax
.text:000000000040111B                 sub     eax, 1          ; !!! 参数1 - 1 ≤ 5, 即参数1≤6
.text:000000000040111E                 cmp     eax, 5          ; Compare Two Operands
.text:0000000000401121                 jbe     short loc_401128 ; r12=r12+1=0+1=1   r12用于控制循环,一共循环6次
.text:0000000000401123                 call    explode_bomb    ; Call Procedure
.text:0000000000401128 ; ---------------------------------------------------------------------------
.text:0000000000401128
.text:0000000000401128 loc_401128:                             ; CODE XREF: phase_6+2D↑j
.text:0000000000401128                 add     r12d, 1         ; r12=r12+1=0+1=1   r12用于控制循环,一共循环6次
.text:000000000040112C                 cmp     r12d, 6         ; Compare Two Operands
.text:0000000000401130                 jz      short loc_401153 ; [rsp+18h]
.text:0000000000401132                 mov     ebx, r12d       ; ebx = r12 = 1    r12赋值给ebx,
.text:0000000000401135
.text:0000000000401135 loc_401135:                             ; CODE XREF: phase_6+57↓j
.text:0000000000401135                 movsxd  rax, ebx        ; rax = ebx = 1 = 2 = 3=4=5   ebx给rax用于获取输入的数字
.text:0000000000401138                 mov     eax, [rsp+rax*4+78h+var_78] ; [rsp+1*4+0]=[rsp+0x4]=参数2 = [rsp+2*4]= 参数3 =
.text:000000000040113B                 cmp     [rbp+0], eax    ; !!! 参数1和参数2,3,4,5,6 不能相等
.text:000000000040113E                 jnz     short loc_401145 ; ebx = ebx + 1 = 2 =3=4=5=6
.text:0000000000401140                 call    explode_bomb    ; Call Procedure
.text:0000000000401145 ; ---------------------------------------------------------------------------
.text:0000000000401145
.text:0000000000401145 loc_401145:                             ; CODE XREF: phase_6+4A↑j
.text:0000000000401145                 add     ebx, 1          ; ebx = ebx + 1 = 2 =3=4=5=6
.text:0000000000401148                 cmp     ebx, 5          ; ebx=2,3,4,5<=5   6>5
.text:000000000040114B                 jle     short loc_401135 ; rax = ebx = 1 = 2 = 3=4=5   ebx给rax用于获取输入的数字
.text:000000000040114D                 add     r13, 4          ; r13=r13+4=[rsp+4] = 参数2
.text:0000000000401151                 jmp     short loc_401114 ; rsp赋值给rbp, rbp=r13=1=1+4=5
.text:0000000000401153 ; ---------------------------------------------------------------------------
.text:0000000000401153
.text:0000000000401153 loc_401153:                             ; CODE XREF: phase_6+3C↑j
00001130 0000000000401130: phase_6+3C
```

r12寄存器用于控制循环, 当r12寄存器中的值为6时,跳转到地址0x401153处

4. 地址0x40115-0x401174

```
.text:0000000000401153
.text:0000000000401153 loc_401153:                             ; CODE XREF: phase_6+3C↑j
.text:0000000000401153                 lea     rsi, [rsp+78h+var_60] ; [rsp+18h], 参数6的上一个地址, 控制循环的退出
.text:0000000000401158                 mov     rax, r14        ; 参数1的地址给rax
.text:000000000040115B                 mov     ecx, 7
.text:0000000000401160
.text:0000000000401160 loc_401160:                             ; CODE XREF: phase_6+79↓j
.text:0000000000401160                 mov     edx, ecx        ; ecx不变,因此每次循环都将edx重新赋值为7
.text:0000000000401162                 sub     edx, [rax]      ; 用7减去当前的参数值
.text:0000000000401164                 mov     [rax], edx      ; 结果重新赋值给参数的地址进行覆盖
.text:0000000000401166                 add     rax, 4          ; rax存放的地址+4, 取下一个参数
.text:000000000040116A                 cmp     rax, rsi        ; Compare Two Operands
.text:000000000040116D                 jnz     short loc_401160 ; ecx不变,因此每次循环都将edx重新赋值为7
.text:000000000040116F                 mov     esi, 0
.text:0000000000401174                 jmp     short loc_401197 ; Jump
```

0x401153处将rsp+18h的地址给rsi, 也就是参数6的地址再加4.

0x401158处将r14中存放的地址, 也就是参数1的地址(往上溯源发现是rsp中存放的值)赋值给rax寄存器. rax存放的地址在0x401166处自增4,也就是存放下一个参数的地址. 然后rsi寄存器在地址0x40116A处与rax进行比较, 控制循环的次数.

接下来, 在0x40115B处将7赋值给ecx寄存器, 在每一层循环中都用ecx寄存器对edx寄存器重新赋值. 接下来用edx减去当前参数, 并对当前参数重新覆盖.

参数1 = 7 - 参数1; 参数2=7-参数2; 参数3=7-参数3… 以此类推

再令esi为0, 跳转到0x401197.

5. 接下来, 注意到gdb调试中的0x401183和0x4011A4的指令中, 都有个地址: 0x6032d0. 在IDA中, 显示为node1. 猜测为链表.

可以看到是在数据段, 因此这个链表应该是个全局变量. 注意到node1中, 0x6032D8-0x6032DF为0x6032E0(高位放在高地址,低位放在低地址), 也就是node2的地址. 因此, 该节点应该是个结构体, 最后一个成员为指针, 指向下一个节点, 并且占了八个字节. 一个节点占了16个字节.

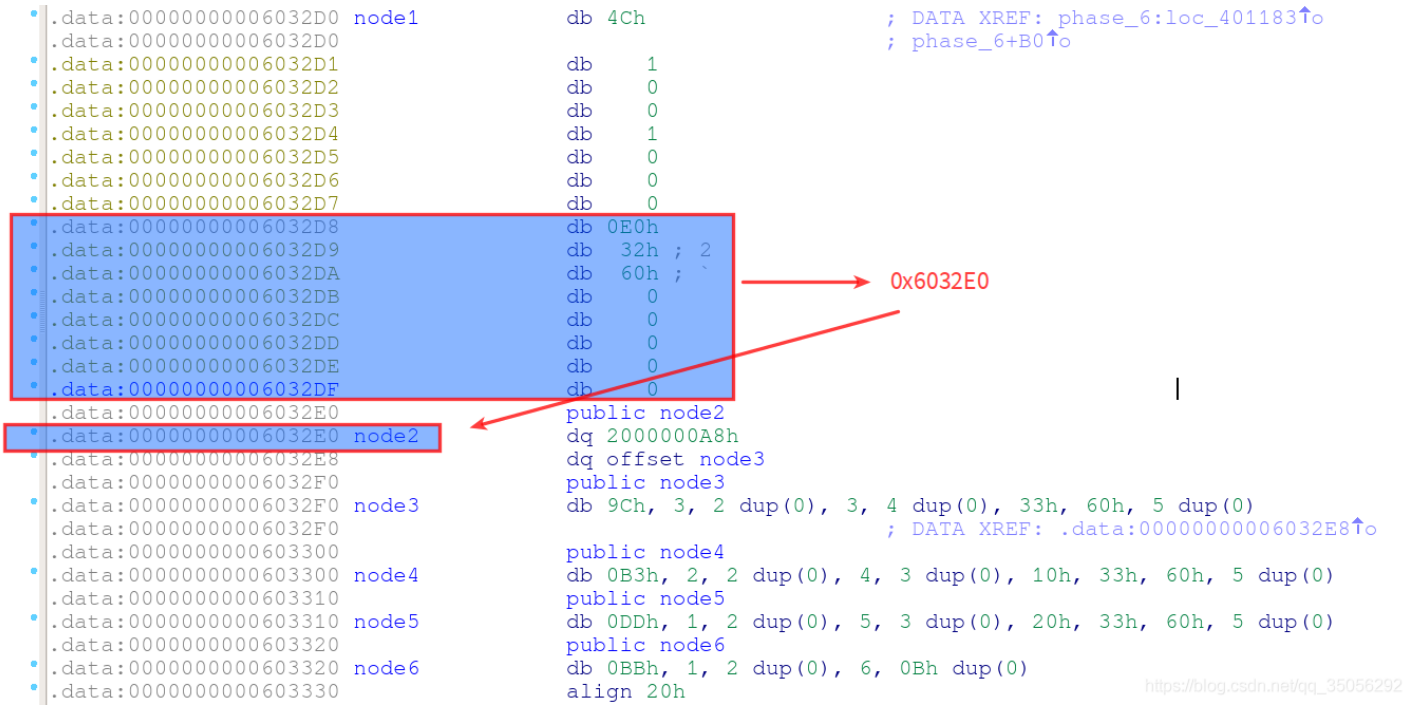
因此可以猜测结构体为:

struct node{

…,

node *next;

};



6. 地址0x401197开始, 有两个循环, 外层循环由rsi寄存器控制, 当rsi寄存器值为0x18时退出, 内层循环由eax控制, 当eax值和ecx寄存器值一样时退出.

a) 如果当前的参数值(也就是被7减过的)大于1, 则进入一层子循环0x401176-0x40117F, 用寄存器eax的值来控制, 只有当当前的参数与eax的值相同时才会退出.

并且将rdx寄存器中的值赋值为每个节点的起始地址 :

0x401176处: mov rdx, QWORD ptr [rdx+8] 这里比较奇怪的是IDA中没有显示QWORD ptr

从rdx+8的地址开始, 取8个字节放进rdx寄存器, rdx是node1的 起始地址, +8是刚好到指向下个节点的指针的起始地址, 再取8字节, 刚好就是下一个节点的起始地址

当eax的值与当前参数相同时, 跳出循环,并且跳转至0x401188处, 把node1的地址赋值给[rsp+rsi*2+20h], 然后rsi自增4, 当rsi值为24时退出循环, 刚好赋值了6次. 当rsi值不为24时, 继续回到0x401197, 处理下一个参数.

b) 如果当前参数≤1, 则跳转到0x401183处, 先将node1地址给edx寄存器, 然后在0x401188处对地址[rsp+rsi*2+20h]进行赋值.

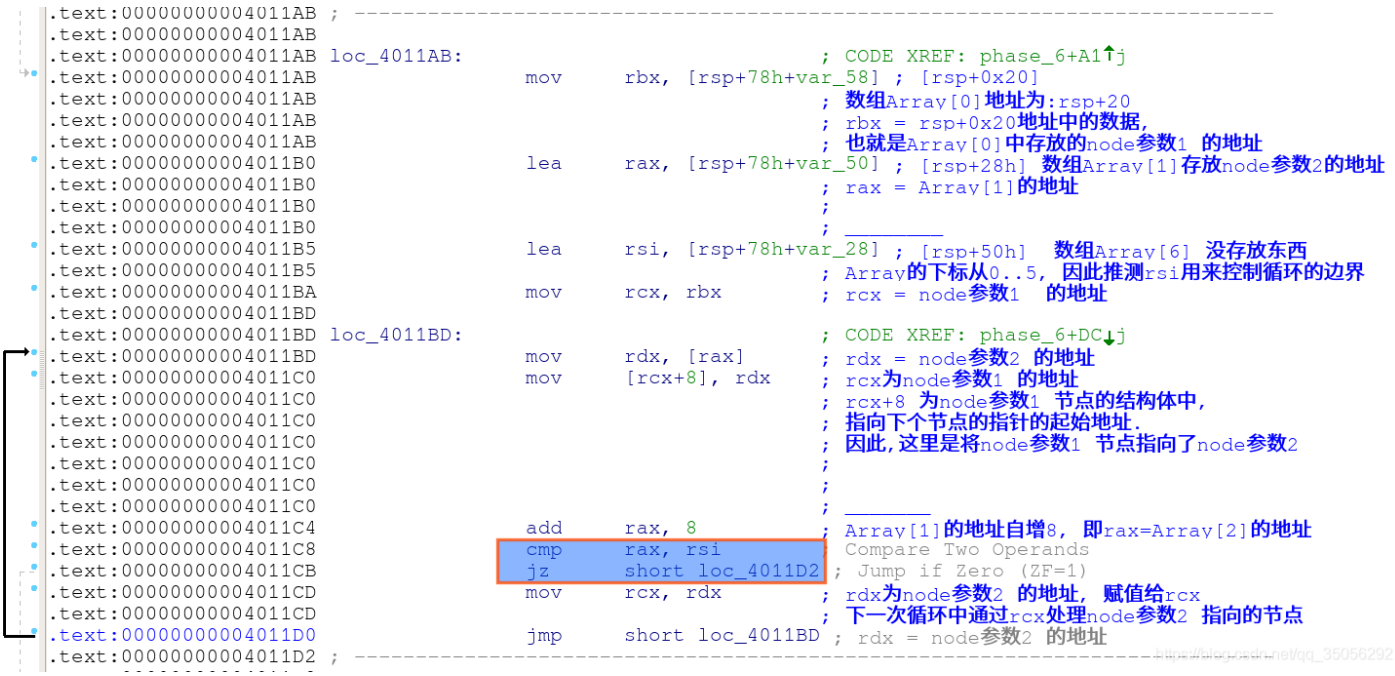
c) 因此, 从0x401176-0x401197的意义是, 根据被7减过的参数, 对地址[rsp+rsi*2+0x20]进行赋值.
*比如当前参数1的值为2, 此时rsi值为0, 那么会将地址[rsp+0*2*+0x20]的内容赋值为 node参数1 也就是node2 的地址.* 可以猜想[rsp+0+20h]开始分配了一个数组, 数组起始地址从[rsp+0x20]开始, 最后一个元素起始地址是[rsp+0x48]:

struct node *Array[6];

其中, Arrayi是一个指针, 指向了节点p(nodep, p=1,2…,6)的地址, p为参数的值. 并且, 由于指针是八个字节, 因此地址[rsp+rsi*2+20h]中rsi需要乘以2.

哭了,这部分终于理清楚了.

7. 从地址0x4011AB开始, 这里需要注意的是, 数组本身的地址以及数组中元素存储的节点的起始地址的区别:



分成两部分来看:

a) 首先是初始化的部分: 0x4011AB至0x4011BA

0x4011AB mov rbx, [rsp+78h+var_58]

[rsp+0x20] 是Array[0]的起始地址, mov指令将该起始地址保存的地址, 也就是 node参数1 的地址赋值给了rbx寄存器.

0x4011B0 lea rax, [rsp+78h+var_50]

[rsp+0x28] 是Array[1]的起始地址, lea指令将Array[1]的起始地址赋值给rax寄存器.

4011B5 lea rsi, [rsp+78h+var_28]

[rsp+0x50]可以看作是Array[6]的起始地址, 把该地址直接赋值给rsi寄存器. 但是Array数组元素下标从0到5, 所以猜测这个地址应该是后续用来判断循环边界的. 这点在地址0x4011C8处得到验证.

4011BA mov rcx, rbx

这里将rbx保存的值, 也就是 node参数1 的起始地址赋值给rcx寄存器. rcx寄存器在这里的作用是保存当前的 node的起始地址.

b) 地址0x4011BD至0x4011D0 为循环, 由rax寄存器控制, rsi寄存器为边界:

这里就不在一行一行解读, 详细步骤看图片中的注释即可.

rcx寄存器保存了当前遍历到的Array元素Array[i] 保存的地址, 也就是node参数i+1 的地址, 其中i∈[0,5] . 因为数组元素Array[i]保存的是节点 nodei+1 的起始地址.

rax寄存器保存了当前Array元素的下一个元素 Array[i+1] 的地址, 再通过 [rax]就能得到该元素保存的 node参数i+2 的地址.

0x4011C0 mov [rcx+8], rdx

rcx为 node参数i+1 的地址 rcx+8 为node参数i 节点的结构体中, 指向下个节点的指针的起始地址. 因此,这里是将 node参数i 节点指向了node参数i+1.

然后在0x4011C4处让rax寄存器的值自增8, 也就是将Array[i+1] 的地址自增8, 来到Array[i+2] 的地址. 依次类推.

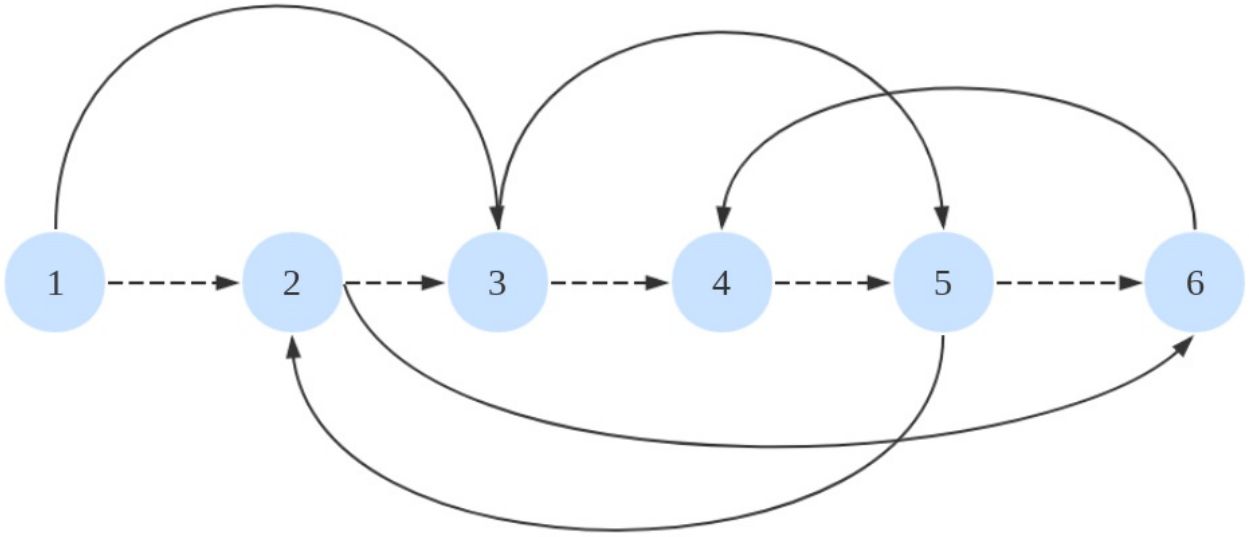
当rax地址与rsi相同, 即rax从[rsp+0x28]变成[rsp+0x50]时, 一共经过5个循环, 修改了5个节点指向的节点地址. 然后跳转到0x4011D2.

总结一下, 6)中是将 Array[i]保存了 node参数i+1 的地址, i∈[0, 5].

7)中是遍历数组Array, 对于每个元素Array[i]保存的节点地址, 让保存的节点地址 node参数i+1 指向 node参数i+2.

比如,

Array	0	1	2	3	4	5
node	1	3	5	2	6	4



8. 地址0x4011D2:

地址0x4011D2处, rdx此时保存的值为 node参数6 的地址.

地址0x4011DF处, 由0x4011AB处可知, rbx为[rsp+0x20], 是 node参数1 的地址. 则这里是将[rbx+8]的值, 也就是node1结构体中指向下个节点的指针, 即 node参数2 的地址给了rax寄存器.

地址0x4011E3处将[rax]的值, 也就是node参数2的值赋值给eax寄存器.

结合地址0x4011E5和0x4011E7可知, [rbx]必须大于 eax寄存器的值, 也就是node参数1 的值 必须≥node参数2的值. 不满足该条件就会引爆炸弹! 也就是说, 当前节点的值必须 > 它指向的节点的值. 也就是说, 用户的输入能对这个链表进行从大到小的排序.

同时注意到0x4011E5是将eax的值赋值给dword大小的内存空间[rbx], 也就是4个字节. 因此, 可以判断结构体第一个成员是int类型. 还记得之前分析的0x4011C0(第7)点分析)中, 为了将地址偏移到当前节点的结构体中指向下个节点的指针, 需要将rcx+8.回想起之前老师说的对齐机制, 这里应该是在节点的值–4字节 与 指针–8字节之间填充了4个字节. 结构体可以初步判断为:

struct node{

int val;

int padding;

node* next;

};

满足这个条件判断, 跳转到0x4011EE.

0x4011EE mov rbx, [rbx+8]

[rbx+8] 为node1结构体中指向下个节点的指针, 即 node参数2 的地址, 将该地址赋值给rbx. 也就是遍历到下一个节点. 然后跳回地址0x4011DF.

9. 那么, 现在的问题就变成了需要知道链表中节点存储的值. 然后对其从大到小排序, 再反推出输入的数值.

在IDA中查看或是在gdb中打印, 这里我直接在gdb中打印, 以16进制打印12个 8个字节的值:

```
(gdb) x/12xg 0x6032d0
0x6032d0 <node1>:       0x000000010000014c      0x00000000006032e0
0x6032e0 <node2>:       0x00000002000000a8      0x00000000006032f0
0x6032f0 <node3>:       0x000000030000039c      0x0000000000603300
0x603300 <node4>:       0x00000004000002b3      0x0000000000603310
0x603310 <node5>:       0x00000005000001dd      0x0000000000603320
0x603320 <node6>:       0x00000006000001bb      0x0000000000000000
```

根据高位放在高地址, 低位放在低地址的原则, 结合结构体的结构:

struct node{

int val; // 低4字节

int padding; // 高4字节

node* next; // 高8字节

};

画出节点的初始关系图:

用python将其转为10进制:

```
>>> hex_arr = ['0x14c', '0xa8', '0x39c', '0x2b3', '0x1dd', '0x1bb']
>>> dec_arr = [int(i, 16) for i in hex_arr]
>>> dec_arr
[332, 168, 924, 691, 477, 443]
>>>
```

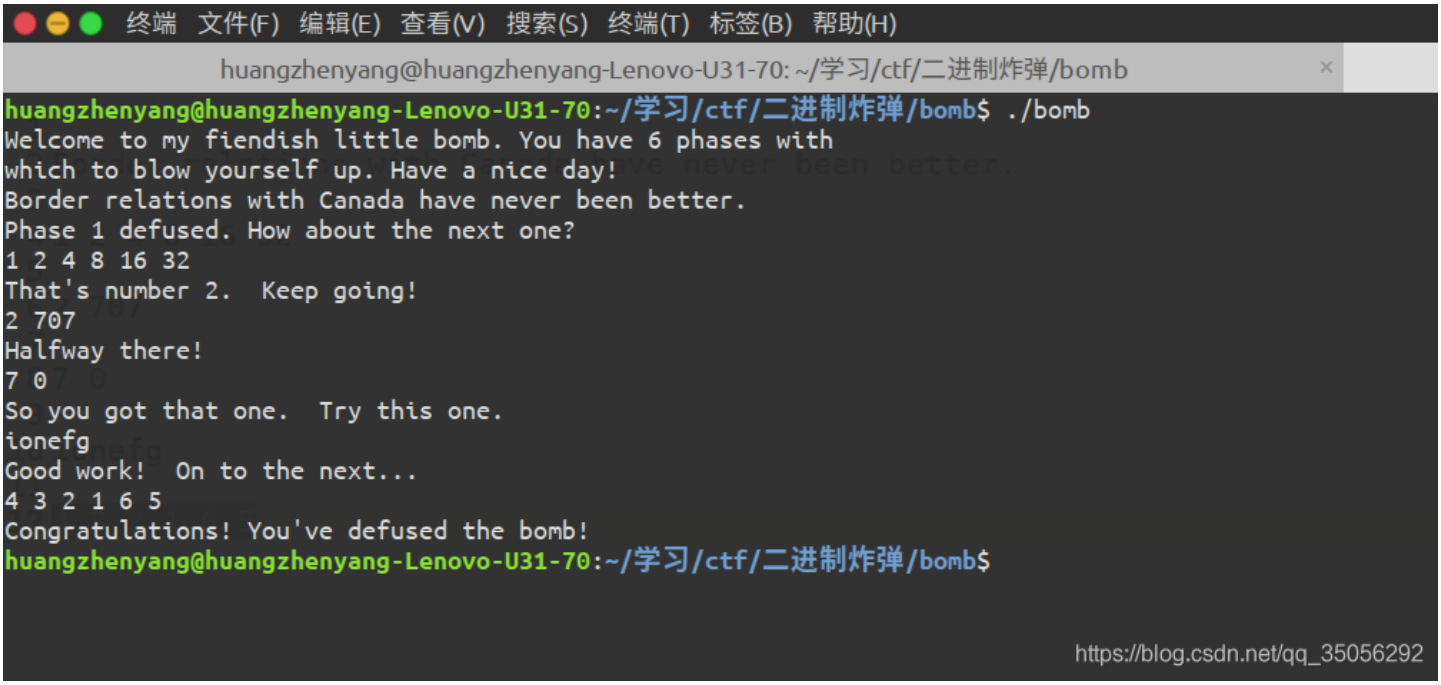
对节点的值进行从大到小排序:

节点val	0x39c	0x2b3	0x1dd	0x1bb	0x14c	0xa8
节点编号	node 3	node 4	node 5	node 6	node 1	node 2

也就是说, 用户的输入在经由 7 减去每个数之后得到的是 3, 4, 5, 6, 1, 2. 这样一来才能保证重新链接后的节点的值是从大到小排序的.

因此, 用户的输入为4 3 2 1 6 5

测试成功!~



0x05 总结收获

通过这次二进制炸弹的实验，算是对汇编的基础有了一点了解。并且对IDA的使用和gdb的使用也有了初步的了解。

不过这篇被diss惹，不能写得太细，应该把整个题目的框架揪出来就好。 下次好好照着师兄的要求来！

0x06 参考资料

1.关于汇编跳转指令的说明