

磨刀不误砍柴工

提高汇编读写能力，大致分三个阶段吧：

(1) 先要了解熟悉指令，特别是常用的指令（**通用指令**）。而其它的的 system 指令、x87 float 指令、SIMD 指令**大致能认得，大致知道是做什么的。**（另外：**现在 x64 平台浮点数处理基本都使用 SIMD 指令处理了，包括 SSE 系列指令**）

(2) 了解汇编语表达 C/C++ 语言代码以及它的各种结构，比如：各种数据结构（如：struct 等）、for()，while()，switch() 等。**这里有捷径可走的：对照看编译生成的汇编代码。**多看多思考就行了。

(3) 能将汇编语言反过来，用 c/c++ 来表达。

=====

实际上，**这里最重要就是第二阶段，多看编译器生成的代码。**第一阶段的熟悉指令不用花太多时间的（除非你想写汇编代码）。

基本经受过 bomblab 折磨后觉得很有道理

试图闷声发大财，炸弹该爆就爆，但是不要上报上去！

(kill <send_msg>

```
=> 0x000055555555d11 <+0>:    sub    $0x8,%rsp
0x000055555555d15 <+4>:    lea    0x153c(%rip),%rdi      # 0x555555557258
0x000055555555d1c <+11>:   call   0x555555555070 <puts@plt>
0x000055555555d21 <+16>:   lea    0x172d(%rip),%rdi      # 0x555555557455
0x000055555555d28 <+23>:   call   0x555555555070 <puts@plt>
0x000055555555d2d <+28>:   lea    0x173e(%rip),%rdi      # 0x555555557472
0x000055555555d34 <+35>:   call   0x555555555070 <puts@plt>
0x000055555555d39 <+40>:   lea    0x174f(%rip),%rdi      # 0x55555555748f
0x000055555555d40 <+47>:   call   0x555555555070 <puts@plt>
0x000055555555d45 <+52>:   lea    0x1760(%rip),%rdi      # 0x5555555574ac
0x000055555555d4c <+59>:   call   0x555555555070 <puts@plt>
0x000055555555d51 <+64>:   lea    0x1771(%rip),%rdi      # 0x5555555574c9
0x000055555555d58 <+71>:   call   0x555555555070 <puts@plt>
0x000055555555d5d <+76>:   lea    0x1781(%rip),%rdi      # 0x5555555574e5
0x000055555555d64 <+83>:   call   0x555555555070 <puts@plt>
0x000055555555d69 <+88>:   lea    0x1792(%rip),%rdi      # 0x555555557502
0x000055555555d70 <+95>:   call   0x555555555070 <puts@plt>
0x000055555555d75 <+100>:  lea    0x17a3(%rip),%rdi      # 0x55555555751f
0x000055555555d7c <+107>:  call   0x555555555070 <puts@plt>
0x000055555555d81 <+112>:  lea    0x17b4(%rip),%rdi      # 0x55555555753c
0x000055555555d88 <+119>:  call   0x555555555070 <puts@plt>
0x000055555555d8d <+124>:  lea    0x17c5(%rip),%rdi      # 0x555555557559
0x000055555555d94 <+131>:  call   0x555555555070 <puts@plt>
0x000055555555d99 <+136>:  lea    0x17d6(%rip),%rdi      # 0x555555557576
0x000055555555da0 <+143>:  call   0x555555555070 <puts@plt>
0x000055555555da5 <+148>:  lea    0x17e7(%rip),%rdi      # 0x555555557593
0x000055555555dac <+155>:  call   0x555555555070 <puts@plt>
0x000055555555db1 <+160>:  mov    $0x0,%edi
0x000055555555db6 <+165>:  call   0x555555555c44 <send_msg>
0x000055555555dbb <+170>:  lea    0x14de(%rip),%rdi      # 0x5555555572a0
0x000055555555dc2 <+177>:  call   0x555555555070 <puts@plt>
0x000055555555dc7 <+182>:  mov    $0x8,%edi
0x000055555555dcc <+187>:  call   0x555555555190 <exit@plt>
End of assembler dump.
```

Explode_bomb 函数地址:

0x000055555555d11

Send_msg 调用地址:

0x000055555555db6

Exit@plt 调用地址:

0x000055555555dcc

指令偏移量: (作差)

0x000055555555db6-0x000055555555d11=0x a5

0x000055555555dcc-0x000055555555d11=0x bb

Phase_1 —— 体验

```
Dump of assembler code for function phase_1:  
0x000055555555429 <+0>:    sub    $0x8,%rsp  
=> 0x00005555555542d <+4>:    lea    0x1d14(%rip),%rsi          # 0x555555557148  
0x000055555555434 <+11>:   call   0x55555555aac <strings_not_equal>  
0x000055555555439 <+16>:   test   %eax,%eax  
0x00005555555543b <+18>:   jne    0x55555555442 <phase_1+25>  
0x00005555555543d <+20>:   add    $0x8,%rsp  
0x000055555555441 <+24>:   ret  
0x000055555555442 <+25>:   call   0x55555555d11 <explode_bomb>  
0x000055555555447 <+30>:   jmp    0x5555555543d <phase_1+20>  
End of assembler dump.
```

%rsp - 8 => %rsp

%rsp:

0x7fffffff dc18 => 0x7fffffff dc10

0x55555339 0x00000001

0x1d14(%rip) => %rsi 莫名其妙的赋值，很可疑

0x1d14(%rip): %rsi:

0x5555555713d 0x5555555b6b1

0x6e6f2073 0x0000000a

Call <strings_not_equal> //对比字符串

/*以下是一些无用的研究，在汇编的海洋里迷失自己，走了死胡同呀~ 也许经过天昏地暗的阅读理解后，会发现最开始的 rsi rdi 之可疑性，于是来到最开始，终于注意到第二句话的蹊跷*/

输入字符存入了 %rdi

strings_not_equal:

rdi 存入 rbx

rsi 存入 rbp 似乎是在传参，那么反推 rdi & rsi 一定存着好东西

```
Dump of assembler code for function strings_not_equal:  
=> 0x000055555555aac <+0>:    push    %r12  
0x000055555555aae <+2>:    push    %rbp  
0x000055555555aaf <+3>:    push    %rbx  
0x000055555555ab0 <+4>:    mov     %rdi,%rbx  
0x000055555555ab3 <+7>:    mov     %rsi,%rbp  
0x000055555555ab6 <+10>:   call    0x55555555a8f <string_length>  
0x000055555555abb <+15>:   mov     %eax,%r12d  
0x000055555555abe <+18>:   mov     %rbp,%rdi  
0x000055555555ac1 <+21>:   call    0x55555555a8f <string_length>  
0x000055555555ac6 <+26>:   mov     %eax,%edx  
0x000055555555ac8 <+28>:   mov     $0x1,%eax  
0x000055555555acd <+33>:   cmp     %edx,%r12d  
0x000055555555ad0 <+36>:   jne    0x55555555b03 <strings_not_equal+87>  
0x000055555555ad2 <+38>:   movzbl (%rbx),%edx  
0x000055555555ad5 <+41>:   test   %dl,%dl  
0x000055555555ad7 <+43>:   je     0x55555555af7 <strings_not_equal+75>  
0x000055555555ad9 <+45>:   mov     $0x0,%eax  
0x000055555555ade <+50>:   cmp     %dl,0x0(%rbp,%rax,1)  
0x000055555555ae2 <+54>:   jne    0x55555555afe <strings_not_equal+82>  
0x000055555555ae4 <+56>:   add    $0x1,%rax  
0x000055555555ae8 <+60>:   movzbl (%rbx,%rax,1),%edx  
0x000055555555aec <+64>:   test   %dl,%dl  
0x000055555555aee <+66>:   jne    0x55555555ade <strings_not_equal+50>  
0x000055555555af0 <+68>:   mov     $0x0,%eax  
0x000055555555af5 <+73>:   jmp    0x55555555b03 <strings_not_equal+87>  
0x000055555555af7 <+75>:   mov     $0x0,%eax  
0x000055555555afc <+80>:   jmp    0x55555555b03 <strings_not_equal+87>  
0x000055555555afe <+82>:   mov     $0x1,%eax  
0x000055555555b03 <+87>:   pop    %rbx  
0x000055555555b04 <+88>:   pop    %rbp  
0x000055555555b05 <+89>:   pop    %r12  
0x000055555555b07 <+91>:   ret  
End of assembler dump.
```

```
(gdb) x $r12  
0x7fffffffdd38: 0xfffffd8: 0xffffdfa8  
          (gdb) x $rbp  
0x1: Cannot access memory at address 0x1  
          (gdb) x $rbx  
0xa: Cannot access memory at address 0xa
```

```
(gdb) x $rdi  
0x555555559700 <input_strings>: 0x66616573
```

```
(gdb) x $rsi  
0x555555557148: 0x64206f53  
          (gdb) print $rsi  
$6 = 93824992244040
```

发现 rdi 原来存的是 输入字符串 那 rsi 就更加可疑了，
直接一探究竟 x/s \$rsi 得到可爱的答案！

string_length: 测输入字符串(input_strings)长度

测的是 rdi 长度

```
Dump of assembler code for function string_length:  
=> 0x000055555555a8f <+0>:    cmpb    $0x0,(%rdi)  
 0x000055555555a92 <+3>:    je      0x55555555aa6 <string_length+23>  
 0x000055555555a94 <+5>:    mov     $0x0,%eax  
 0x000055555555a99 <+10>:   add     $0x1,%rdi  
 0x000055555555a9d <+14>:   add     $0x1,%eax  
 0x000055555555aa0 <+17>:   cmpb    $0x0,(%rdi)  
 0x000055555555aa3 <+20>:   jne     0x55555555a99 <string_length+10>  
 0x000055555555aa5 <+22>:   ret  
 0x000055555555aa6 <+23>:   mov     $0x0,%eax  
 0x000055555555aab <+28>:   ret  
  
End of assembler dump.
```

x/<n/f/u> <addr>

n、f、u是可选的参数。

n 是一个正整数，表示显示内存的长

f 表示显示的格式，参见上面。如果那么格式可以是i。

x 按十六进制□ 格式显示变量。

d 按十进制格式显示变量。

u 按十六进制格式显示无符号整型。

o 按八进制格式显示变量。

t 按二进制格式显示变量。

a 按十六进制格式显示变量。

c 按字符格式显示变量。

f 按浮点数格式显示变量。

Phase_2 — 计数器 array

```
Dump of assembler code for function phase_2:  
=> 0x0000555555555449 <+0>:    push  %rbp  
0x000055555555544a <+1>:    push  %rbx  
0x000055555555544b <+2>:    sub   $0x68,%rsp  
0x000055555555544f <+6>:    mov   %fs:0x28,%rax  
0x0000555555555458 <+15>:   mov   %rax,0x58(%rsp)  
0x000055555555545d <+20>:   xor   %eax,%eax  
0x000055555555545f <+22>:   movl  $0x0,0x20(%rsp)  
0x0000555555555467 <+30>:   movl  $0x0,0x24(%rsp)  
0x000055555555546f <+38>:   movl  $0x0,0x28(%rsp)  
0x0000555555555477 <+46>:   movl  $0x0,0x2c(%rsp)  
0x000055555555547f <+54>:   movl  $0x0,0x30(%rsp)  
0x0000555555555487 <+62>:   movl  $0x0,0x34(%rsp)  
0x000055555555548f <+70>:   movabs $0x4246434642414441,%rax  
0x0000555555555499 <+80>:   movabs $0x46434245454341,%rdx  
0x00005555555554a3 <+90>:   mov   %rax,0x40(%rsp)  
0x00005555555554a8 <+95>:   mov   %rdx,0x48(%rsp)  
0x00005555555554ad <+100>:  mov   %rsp,%rsi  
0x00005555555554b0 <+103>:  call  0x555555555dd1 <read_six_numbers>  
0x00005555555554b5 <+108>:  cmpl  $0x0,(%rsp)  
0x00005555555554b9 <+112>:  js   0x5555555554c7 <phase_2+126>  
0x00005555555554bb <+114>:  mov   $0x0,%ebx  
0x00005555555554c0 <+119>:  lea   0x40(%rsp),%rbp  
0x00005555555554c5 <+124>:  jmp  0x5555555554e0 <phase_2+151>  
0x00005555555554c7 <+126>:  call  0x555555555d11 <explode_bomb>  
0x00005555555554cc <+131>:  jmp  0x5555555554bb <phase_2+114>  
0x00005555555554ce <+133>:  movsbl (%rbx,%rbp,1),%eax  
0x00005555555554d2 <+137>:  sub   $0x41,%eax  
0x00005555555554d5 <+140>:  cltq  
0x00005555555554d7 <+142>:  addl  $0x1,0x20(%rsp,%rax,4)  
0x00005555555554dc <+147>:  add   $0x1,%rbx  
0x00005555555554e0 <+151>:  mov   %rbp,%rdi  
0x00005555555554e3 <+154>:  call  0x555555555a8f <string_length>  
0x00005555555554e8 <+159>:  cmp   %ebx,%eax  
0x00005555555554ea <+161>:  jg   0x5555555554ce <phase_2+133>  
0x00005555555554ec <+163>:  mov   $0x0,%ebx  
0x00005555555554f1 <+168>:  lea   0x20(%rsp),%rbp  
0x00005555555554f6 <+173>:  jmp  0x555555555502 <phase_2+185>  
0x00005555555554f8 <+175>:  add   $0x4,%rbx  
0x00005555555554fc <+179>:  cmp   $0x18,%rbx  
0x0000555555555500 <+183>:  je   0x555555555512 <phase_2+201>  
0x0000555555555502 <+185>:  mov   0x0(%rbp,%rbx,1),%eax  
0x0000555555555506 <+189>:  cmp   %eax,(%rsp,%rbx,1)  
0x0000555555555509 <+192>:  je   0x5555555554f8 <phase_2+175>  
0x000055555555550b <+194>:  call  0x555555555d11 <explode_bomb>  
0x0000555555555510 <+199>:  jmp  0x5555555554f8 <phase_2+175>  
0x0000555555555512 <+201>:  mov   0x58(%rsp),%rax  
0x0000555555555517 <+206>:  sub   %fs:0x28,%rax  
0x0000555555555520 <+215>:  jne  0x555555555529 <phase_2+224>
```

汇编中 `mov %fs:0x28,%rax` 的作用：

栈保护功能，将一个特殊值（fs:0x28）存在栈的底部，函数运行结束后再取出这个值和fs:0x28做比较，如果有改变就说明栈被破坏，调用`_stack_chk_fail@plt`。
fs寄存器的值本身指向当前线程结构

事情开始复杂起来了，寄存器看的头昏脑胀，磨刀不误砍柴工。

我希望多了解一些他们的习性：

%rax寄存器

该寄存器最常用的就是作为函数调用的返回值，一个函数如果有返回值，该返回值一定是通过rax寄存器或者他的低位寄存器返回的，当然，在函数执行过程中也可以存储别的数据。

%rdi %rsi %rdx %rcx %r8 %r9

这六个寄存器是作为函数传递参数使用的，并且使用规则很简单，如果一个参数能够保存到寄存器，一定是在按照标题寄存器的顺序列表依次保存的，当然参数可能无法使用寄存器保存，比如参数超过了寄存器能保存的数量

%rbx %rbp %r12 %r13 %r14 %r15

这六个寄存器被称为被调用者保存寄存器，意思就是被调用函数必须保证调用该函数之前和从该函数返回之前这几个寄存器的值是不变的。

一般情况下，如果函数中用到哪个寄存器的话，会在函数开始的时候先把该寄存器的值入栈，函数返回之前在把寄存器出栈，这样，函数返回后就能保证寄存器的值没有发生变化。

%rsp

运行时栈顶指针，与其说rsp保存的是运行时栈顶的位置还不如说，运行时栈顶的位置随着rsp寄存器的值的改变而改变

%r10 %r11

这两个寄存器是被称为调用者保存的寄存器，意思就是比如函数A调用函数B，函数A中使用了这两个寄存器，但是函数B可以随便修改这两个寄存器的值而不用考虑任何问题，函数A调用函数B之后，需要确保这两个寄存器的值依然可用。

哦！原来那么多乱七八糟的入栈是这个意图！

通用寄存器

64位寄存器名称	用途	32位寄存器名称	32位下用途	16位	8位
rax	保存系统调用号、函数调用的返回值、乘法运算结果的低64位、除法运算被除数的低64位、除法运算结果的商	eax	—	ax	al
rbx	—	ebx	函数调用的第一个参数	bx	bl
rcx	—	ecx	函数调用的第二个参数	cl	
rdx	函数调用的第3个参数、乘法运算结果的高64位、除法运算被除数的高64位、除法运算结果的商	edx	用作函数调用的第三个参数	dx	dl
rdi	函数调用的第一个参数	edi	用作函数调用的第五个参数	di	dil
rsi	函数调用的第二个参数	esi	函数调用的第四个参数	si	sil
rbp	base pointer，标识栈帧的起始位置；在函数调用的时候更改以分配栈空间（个人理解）	rbp	用作函数调用的第六个参数	bp	bpl
rsp	永远指向栈顶，函数调用的时候需要先压栈保存	esp	—	sp	spl
r8	函数调用的第五个参数	r8d	—	r8w	r8b
r9	函数调用的第六个参数	r9d	—	r9w	r9b
r10	函数调用的第四个参数	r10d	—	r10w	r10b
r11	随便用（个人理解）	r11d	—	r11w	r11b
r12	随便用（个人理解）	r12d	—	r12w	r12b
r13	随便用（个人理解）	r13d	—	r13w	r13b
r14	随便用（个人理解）	r14d	—	r14w	r14b
r15	随便用（个人理解）	r15d	—	r15w	r15b

两个栈指针
栈底 rbp
栈顶 rsp

```
(gdb) x/s $rsi  
0x55555555575ab: "%d %d %d %d %d %d"
```

输入格式 ↑

```
(gdb) x/s $rbx+$rbp  
0x7fffffffdb0: "ADABFCFBACEEBCF" 65 68 65 66 70 67 70 66 65 67 69 69 66 67 70
```

x/s \$rbx+\$rbp

3

x/d \$rsp+\$rbx

x/s \$rsp+0x40

\$rsp+\$rbx==\$rbx+\$rbp 进入胜利循环

Phase_3 —— switch

JE ; 等于则跳转
JNE ; 不等于则跳转

JZ ; 为 0 则跳转
JNZ ; 不为 0 则跳转

JS ; 为负则跳转
JNS ; 不为负则跳转

JC ; 进位则跳转
JNC ; 不进位则跳转

JO ; 溢出则跳转
JNO ; 不溢出则跳转

JA ; 无符号大于则跳转
JNA ; 无符号不大于则跳转
JAE ; 无符号大于等于则跳转
JNAE ; 无符号不大于等于则跳转

JG ; 有符号大于则跳转
JNG ; 有符号不大于则跳转
JGE ; 有符号大于等于则跳转
JNGE ; 有符号不大于等于则跳转

JB ; 无符号小于则跳转
JNB ; 无符号不小于则跳转
JBE ; 无符号小于等于则跳转
JNBE ; 无符号不小于等于则跳转

JL ; 有符号小于则跳转
JNL ; 有符号不小于则跳转
JLE ; 有符号小于等于则跳转
JNLE ; 有符号不小于等于则跳转

JP ; 奇偶位置位则跳转
JNP ; 奇偶位清除则跳转
JPE ; 奇偶位相等则跳转
JPO ; 奇偶位不等则跳转

有了 phase_2 长久迷惘的沉淀，到 phase_3

终于可以流畅舒适的读懂每一行代码

一步步读就完事了，一步步收集输入线索

特别快搞定

记不住花里胡哨的各类 jmp, 放张图方便查

```
15ab: 0f 85 83 00 00 00 jne 1634 <phase_3+0x106> //指向<_stack_chk_fail@plt>
//金丝雀
15b1: 48 83 c4 18 add $0x18,%rsp
15b5: c3 ret
15b6: b8 15 54 3b 00 00 mov 0x3b54(%rip),%edx # 5110 <delta.1>
15bc: b8 fb 00 00 00 mov $0xfb,%eax
15c1: 29 d0 sub %edx,%eax
15c3: eb c7 jmp 158c <phase_3+0x5e>
15c5: b8 15 45 3b 00 00 mov 0x3b45(%rip),%edx # 5110 <delta.1>
15cb: b8 4c 02 00 00 mov $0x24c,%eax
15d0: 29 d0 sub %edx,%eax
15d2: eb b8 jmp 158c <phase_3+0x5e>
15d4: b8 15 36 3b 00 00 mov 0x3b36(%rip),%edx # 5110 <delta.1>
15da: b8 3c 00 00 00 mov $0x3c,%eax
15df: 29 d0 sub %edx,%eax
15e1: eb a9 jmp 158c <phase_3+0x5e>
15e3: b8 15 27 3b 00 00 mov 0x3b27(%rip),%edx # 5110 <delta.1>
15e9: b8 e2 02 00 00 mov $0x2e2,%eax
15ee: 29 d0 sub %edx,%eax
15f0: eb 9a jmp 158c <phase_3+0x5e>
15f2: b8 15 18 3b 00 00 mov 0x3b18(%rip),%edx # 5110 <delta.1>
15f8: b8 2f 02 00 00 mov $0x22f,%eax
15fd: 29 d0 sub %edx,%eax
15ff: eb 8b jmp 158c <phase_3+0x5e>
1601: b8 15 09 3b 00 00 mov 0x3b09(%rip),%edx # 5110 <delta.1>
1607: b8 10 03 00 00 mov $0x310,%eax
160c: 29 d0 sub %edx,%eax
160e: e9 79 ff ff ff jmp 158c <phase_3+0x5e> //中间好大一段没用上
```

此题出现大段没走的代码，结构高度相似，推测是分支语句。仔细检查后没发现什么奖励关卡入口。

无伤大雅，走吧

Phase_4 —— func4

忍不住调查一下每次开头一段神秘的“无用”代码：

文档里直接用“金丝雀”代称标出，现在不必深究

这一段代码是一个很典型的函数开头，使用`endbr64`来防止ROP攻击，然后压栈`%rbx`。

回忆一下，`%rbx`是一个被调用者保存的寄存器，除了`%rbx`之外还有`%rbp`和`%r12`到`%r15`，你可以通过`%rbx`和`%rbp`中的b是Backup的首字母来记忆，另外再强记一下`%r12`到`%r15`就行了。

接着再将`%rsp`栈针减去`0x20`（注意是十六进制，也就是32个字节）扩大栈，然后使用`mov %fs:0x28,%rax`将`%fs`段寄存器中的`0x28`处的值（也就是`0x28 + %fs`处的值）赋值给`%rax`，再复制到栈指针往上24个字节处（也就是`%rsp + 0x18`处）存储起来。

这代表了一个你在后续Attack lab中会遇到的东西，叫做“金丝雀值（Canary Value）”，用于防止缓冲区溢出攻击（Buffer Overflow Attack）。这个值会在函数的结尾处进行校验，如果发现被修改了，就会抛出异常，阻止程序继续执行。

我们也可以看到，在函数的结尾处，会将`%fs`段寄存器中的`0x28`处的值（也就是`0x28 + %fs`处的值）赋值给`%rax`，然后与之前保存在栈中的金丝雀值进行异或运算，如果结果不为0，则说明金丝雀值被修改了，就会抛出异常，阻止程序继续执行（`call 1290 <__stack_chk_fail@plt>`一句）。

为什么要使用`fs`段寄存器？

这是因为`fs`段寄存器是一个特殊的寄存器，它的值是由操作系统决定的，而不是由程序决定的，所以它的值是不会被修改的，这样就可以防止被恶意修改。

遇到繁琐的func4 鉴于输入范围很小，考虑高效的暴力尝试法绕过对func的解读

```
int func4(int edi) {
    int ebx; // 存储中间结果
    int eax; // 存储计算结果
    int esi = 0; // 第二个输入参数初始化为0
    int edx = 14; // 第三个输入参数初始化为14

    // 计算eax
    eax = edx - esi; // eax = 14 - 0
    ebx = eax >> 1; // 将eax逻辑右移1位
    ebx += (eax & 1) ? 1 : 0; // 加上可能的符号位

    // 比较并根据条件递归
    if (edi < ebx) {
        edx = ebx - 1; // 更新edx
        return func4(edsi) + ebx; // 递归调用
    } else if (edi > ebx) {
        esi = ebx + 1; // 更新esi
        return func4(edsi) + ebx; // 递归调用
    } else {
        return ebx; // 返回ebx
    }
}
```

Phase_5

65	101	41	0100001	A	A	大写字母 A
66	102	42	0100010	B	B	大写字母 B
67	103	43	0100011	C	C	大写字母 C
68	104	44	01000100	D	D	大写字母 D
69	105	45	01000101	E	E	大写字母 E
70	106	46	01000110	F	F	大写字母 F
71	107	47	01000111	G	G	大写字母 G
72	110	48	01001000	H	H	大写字母 H
73	111	49	01001001	I	I	大写字母 I
74	112	4A	01001010	J	J	大写字母 J
75	113	4B	01001011	K	K	大写字母 K
76	114	4C	01001100	L	L	大写字母 L
77	115	4D	01001101	M	M	大写字母 M
78	116	4E	01001110	N	N	大写字母 N
79	117	4F	01001111	O	O	大写字母 O
80	120	50	01010000	P	P	大写字母 P
81	121	51	01010001	Q	Q	大写字母 Q
82	122	52	01010010	R	R	大写字母 R
83	123	53	01010011	S	S	大写字母 S
84	124	54	01010100	T	T	大写字母 T
85	125	55	01010101	U	U	大写字母 U
86	126	56	01010110	V	V	大写字母 V
87	127	57	01010111	W	W	大写字母 W
88	130	58	01011000	X	X	大写字母 X
89	131	59	01011001	Y	Y	大写字母 Y
90	132	5A	01011010	Z	Z	大写字母 Z
91	133	5B	01011011	[[左中括号
92	134	5C	01011100	\	\	反斜杠
93	135	5D	01011101]]	右中括号
94	136	5E	01011110	^	^	音调符号
95	137	5F	01011111	-	_	下划线
96	140	60	01100000	'	`	重音符
97	141	61	01100001	a	a	小写字母 a
98	142	62	01100010	b	b	小写字母 b
99	143	63	01100011	c	c	小写字母 c
100	144	64	01100100	d	d	小写字母 d
101	145	65	01100101	e	e	小写字母 e
102	146	66	01100110	f	f	小写字母 f
103	147	67	01100111	g	g	小写字母 g
104	150	68	01101000	h	h	小写字母 h
105	151	69	01101001	i	i	小写字母 i
106	152	6A	01101010	j	j	小写字母 j
107	153	6B	01101011	k	k	小写字母 k
108	154	6C	01101100	l	l	小写字母 l
109	155	6D	01101101	m	m	小写字母 m
110	156	6E	01101110	n	n	小写字母 n
111	157	6F	01101111	o	o	小写字母 o
112	160	70	01110000	p	p	小写字母 p
113	161	71	01110001	q	q	小写字母 q
114	162	72	01110010	r	r	小写字母 r
115	163	73	01110011	s	s	小写字母 s
116	164	74	01110100	t	t	小写字母 t
117	165	75	01110101	u	u	小写字母 u
118	166	76	01110110	v	v	小写字母 v
119	167	77	01110111	w	w	小写字母 w
120	170	78	01111000	x	x	小写字母 x
121	171	79	01111001	y	y	小写字母 y
122	172	7A	01111010	z	z	小写字母 z

短小的一串代码，不知道为什么放在第五题的位置，感觉只要读的懂基本汇编值比较，把数组打印出来就能过关了。

Phase_6

5 4 2 1 6 3

链表的一大串操作读不懂到底干了啥，于是带几组值总结共性，试图反推出本质含义

17da:	48 8b 5c 24 20	mov 0x20(%rsp),%rbx//rbx=>1nodeA
17df:	48 8b 44 24 28	mov 0x28(%rsp),%rax//rax=> 1nodeB
17e4:	48 89 43 08	mov %rax,0x8(%rbx) //2node5+8=1431671312
17e8:	48 8b 54 24 30	mov 0x30(%rsp),%rdx//rdx=> 2nodeC =-93
17ed:	48 89 50 08	mov %rdx,0x8(%rax) //1nodeB+8 =>(32=>-16)
17f1:	48 8b 44 24 38	mov 0x38(%rsp),%rax//rax:1node4=>2node1=34
17f6:	48 89 42 08	mov %rax,0x8(%rdx) //2node2+8=-32
17fa:	48 8b 54 24 40	mov 0x40(%rsp),%rdx//rdx=> 2nodeE =-23
17ff:	48 89 50 08	mov %rdx,0x8(%rax) //2node1+8=48
1803:	48 8b 44 24 48	mov 0x48(%rsp),%rax//2nodeF =90
1808:	48 89 42 08	mov %rax,0x8(%rdx) //2node6+8=0
180c:	48 c7 40 08 00 00 00	movq \$0x0,0x8(%rax) //2node3+8=0
1814:	bd 05 00 00 00	mov \$0x5,%ebp
1819:	eb 09	jmp 1824 <phase_6+0xf1>
181b:	48 8b 5b 08	mov 0x8(%rbx),%rbx
181f:	83 ed 01	sub \$0x1,%ebp
1822:	74 11	je 1835 <phase_6+0x102>
1824:	48 8b 43 08	mov 0x8(%rbx),%rax //rax:1nodeB
1828:	8b 00	mov (%rax),%eax //eax=68
182a:	39 03	cmp %eax,(%rbx) //2nodeA<= 1nodeB
182c:	7e ed	jle 181b <phase_6+0xe8>
182e:	e8 de 04 00 00	call 1d11 <explode_bomb>
1833:	eb e6	jmp 181b <phase_6+0xe8>

5 1 2 4 6 3

17da:	48 8b 5c 24 20	mov 0x20(%rsp),%rbx//rbx=>1nodeA
17df:	48 8b 44 24 28	mov 0x28(%rsp),%rax//rax=> 1nodeB
17e4:	48 89 43 08	mov %rax,0x8(%rbx) //1node5+8=1431671264
17e8:	48 8b 54 24 30	mov 0x30(%rsp),%rdx//2node3 => 2nodeC =-93
17ed:	48 89 50 08	mov %rdx,0x8(%rax) //2node1+8 => -16 没变化
17f1:	48 8b 44 24 38	mov 0x38(%rsp),%rax// 2nodeB=>2nodeD
17f6:	48 89 42 08	mov %rax,0x8(%rdx) //2nodeC+8 值更改 0=>16
17fa:	48 8b 54 24 40	mov 0x40(%rsp),%rdx//2node2 => 2nodeE
17ff:	48 89 50 08	mov %rdx,0x8(%rax) //2node4+8 值更改 32=>48
1803:	48 8b 44 24 48	mov 0x48(%rsp),%rax//2nodeE
1808:	48 89 42 08	mov %rax,0x8(%rdx) //2node6+8=0 没变化
180c:	48 c7 40 08 00 00 00	movq \$0x0,0x8(%rax) //2node3+8 值更改 16=>0
1813:	00	
1814:	bd 05 00 00 00	mov \$0x5,%ebp
1819:	eb 09	jmp 1824 <phase_6+0xf1>
181b:	48 8b 5b 08	mov 0x8(%rbx),%rbx //rbx=2node1
181f:	83 ed 01	sub \$0x1,%ebp
1822:	74 11	je 1835 <phase_6+0x102>
1824:	48 8b 43 08	mov 0x8(%rbx),%rax //rax: 2node3 => 2nodeB //=>2node2
1828:	8b 00	mov (%rax),%eax //eax=546 (1node1) //eax=1node2=419
182a:	39 03	cmp %eax,(%rbx) //2nodeA<= 1nodeB //2nodeB<=1nodeC
182c:	7e ed	jle 181b <phase_6+0xe8>
182e:	e8 de 04 00 00	call 1d11 <explode_bomb>
1833:	eb e6	jmp 181b <phase_6+0xe8>

1 2 3 5 6 4

17da:	48 8b 5c 24 20	mov	0x20(%rsp),%rbx//rbx=>1nodeA
17df:	48 8b 44 24 28	mov	0x28(%rsp),%rax//rax=> 1nodeB
17e4:	48 89 43 08	mov	%rax,0x8(%rbx) //1node4+8=1431671280
17e8:	48 8b 54 24 30	mov	0x30(%rsp),%rdx//1node4 => 1nodeC
17ed:	48 89 50 08	mov	%rdx,0x8(%rax) //1nodeB+8=1431671296 没变化
17f1:	48 8b 44 24 38	mov	0x38(%rsp),%rax// 1node2=>1nodeD
17f6:	48 89 42 08	mov	%rax,0x8(%rdx) //1nodeC+8 值+16
17fa:	48 8b 54 24 40	mov	0x40(%rsp),%rdx//1nodeC => 1nodeE
17ff:	48 89 50 08	mov	%rdx,0x8(%rax) //1node4+8 没变化
1803:	48 8b 44 24 48	mov	0x48(%rsp),%rax//1node5 => 1nodeF
1808:	48 89 42 08	mov	%rax,0x8(%rdx) //1node6+8 0 => 1431671312
180c:	48 c7 40 08 00 00 00	movq	\$0x0,0x8(%rax) //1nodeF+8 清零
1813:	00		
1814:	bd 05 00 00 00	mov	\$0x5,%ebp //原来是4
1819:	eb 09	jmp	1824 <phase_6+0xf1>
181b:	48 8b 5b 08	mov	0x8(%rbx),%rbx //rbx=2node1
181f:	83 ed 01	sub	\$0x1,%ebp
1822:	74 11	je	1835 <phase_6+0x102>
1824:	48 8b 43 08	mov	0x8(%rbx),%rax //rax: 1nodeF => 1nodeB
1828:	8b 00	mov	(%rax),%eax //eax=546 (1nodeB)
182a:	39 03	cmp	%eax,(%rbx) //1nodeA<= 1nodeB
182c:	7e ed	jle	181b <phase_6+0xe8>
182e:	e8 de 04 00 00	call	1d11 <explode_bomb>
1833:	eb e6	jmp	181b <phase_6+0xe8>

4 5 6 3 1 2

17da:	48 8b 5c 24 20	mov	0x20(%rsp),%rbx//rbx=>1nodeA
17df:	48 8b 44 24 28	mov	0x28(%rsp),%rax//rax=> 1nodeB
17e4:	48 89 43 08	mov	%rax,0x8(%rbx) //1node4+8=1431671328 没变化
17e8:	48 8b 54 24 30	mov	0x30(%rsp),%rdx//1nodeA => 1nodeC
17ed:	48 89 50 08	mov	%rdx,0x8(%rax) //1nodeB+8= 1431671088 没变化
17f1:	48 8b 44 24 38	mov	0x38(%rsp),%rax// 1nodeB=>1nodeD
17f6:	48 89 42 08	mov	%rax,0x8(%rdx) //1nodeC+8 值+16
17fa:	48 8b 54 24 40	mov	0x40(%rsp),%rdx//1nodeC => 1nodeE
17ff:	48 89 50 08	mov	%rdx,0x8(%rax) //1node4+8 没变化
1803:	48 8b 44 24 48	mov	0x48(%rsp),%rax//1nodeB => 1nodeF 根据输入调整链表顺序
1808:	48 89 42 08	mov	%rax,0x8(%rdx) //1node6+8 0 => 1431671312
180c:	48 c7 40 08 00 00 00	movq	\$0x0,0x8(%rax) //1nodeF+8 清零
1813:	00		
1814:	bd 05 00 00 00	mov	\$0x5,%ebp //原来是4
1819:	eb 09	jmp	1824 <phase_6+0xf1>
181b:	48 8b 5b 08	mov	0x8(%rbx),%rbx //rbx=2node1
181f:	83 ed 01	sub	\$0x1,%ebp
1822:	74 11	je	1835 <phase_6+0x102>
1824:	48 8b 43 08	mov	0x8(%rbx),%rax //rax: 1nodeF => 1nodeB
1828:	8b 00	mov	(%rax),%eax //eax=546 (1nodeB)
182a:	39 03	cmp	%eax,(%rbx) //1nodeA<= 1nodeB 链表递增
182c:	7e ed	jle	181b <phase_6+0xe8>
182e:	e8 de 04 00 00	call	1d11 <explode_bomb>
1833:	eb e6	jmp	181b <phase_6+0xe8>

其实这么作反而陷进去了，只见树木不见森林。归根结底在于我阅读汇编能力不足，无法敏锐精准的看出其真实含义，致使有时候不得不靠 gdb 一步一步观察值的更改来猜测其意思，相当于半个抓瞎。读代码当然不分主次，分不清哪些是结构性语句，哪些只是传值（谁叫汇编代码表面看上去全是天花乱坠的传值）。可谓陷入泥潭，越陷越深，心力交瘁，只好暴力尝试以求痕迹。遇到这种情况，我想应该及时止损。暴力尝试时间开销极大，更对提升解读汇编能力效果甚微，

```
(gdb) x/20d 0x555555555591e0
0x555555555591e0 <node1>: 546      1      1431671280      21845
0x555555555591f0 <node2>: 419      2      1431671296      21845
0x55555555559200 <node3>: 858      3      1431671312      21845
0x55555555559210 <node4>: 68       4      1431671328      21845
0x55555555559220 <node5>: 497      5      1431671088      21845
```

继续读汇编试着理

```
(gdb) x/20x 0x555555555591e0
0x555555555591e0 <node1>: 0x00000222      0x00000001      0x555591f0      0x00005555
0x555555555591f0 <node2>: 0x000001a3      0x00000002      0x55559200      0x00005555
0x55555555559200 <node3>: 0x0000035a      0x00000003      0x55559210      0x00005555
0x55555555559210 <node4>: 0x00000044      0x00000004      0x55559220      0x00005555
0x55555555559220 <node5>: 0x000001f1      0x00000005      0x55559130      0x00005555
(gdb) x/4d 0x55555555559130
0x55555555559130 <node6>: 745      6      0      0
```

结合我们对链表的基本理解，试着打印 node 所在地址的内容。

<code>node :</code>	<code>value</code>	<code>(int)</code>
	<code>index</code>	<code>(mpified)</code>
	<code>ptn</code>	<code>(node*)</code>

10 进制方便看值

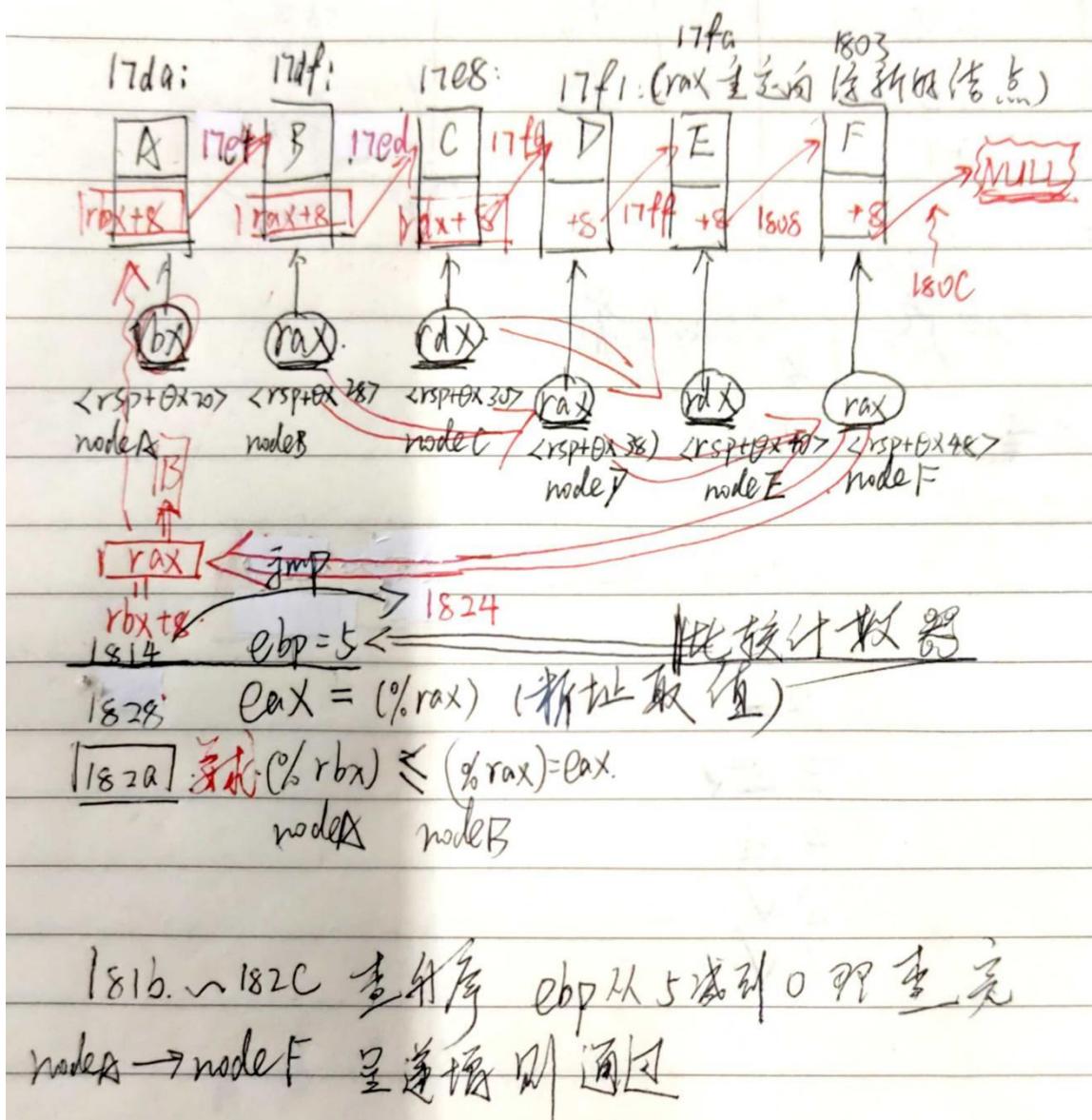
16 进制揭示了本质——第三列数字是下一个结点的地址！

← 我们如此形象的画出其含义

一下子就好理解了！带着这个顿悟再回看操作 node 的那段汇编

17da:	48 8b 5c 24 20	mov 0x20(%rsp),%rbx //rbx=>1nodeA	rbx 前驱指针=>1 st Node: A
17df:	48 8b 44 24 28	mov 0x28(%rsp),%rax //rax=> 1nodeB	rax 后继指针=>2 nd Node: B
17e4:	rsp 存 input		
17e8:		mov %rax,0x8(%rbx) //后继指向前驱的下一位 (确实没变)	
17ed:		mov 0x30(%rsp),%rdx//rdxC	
17f1:	48 8b 44 24 38	mov %rdx,0x8(%rax) //rdx 内容传给 rax 后继指针的下一位 rax 后继指针=>3 rd Node: C	
17f6:	48 89 42 08	mov 0x38(%rsp),%rax// 1nodeB=>1nodeD	
17fa:	48 8b 54 24 40	mov %rax,0x8(%rdx) //1nodeC+8 值+16	
17ff:	48 89 50 08	mov 0x40(%rsp),%rdx//1nodeC => 1nodeE	
1803:	48 8b 44 24 48	mov %rdx,0x8(%rax) //1node4+8 没变化	
1808:	48 89 42 08	mov 0x48(%rsp),%rax//1nodeB => 1nodeF	根据输入调整链表顺序
180c:	48 c7 40 08 00 00 00	mov %rax,0x8(%rdx) //1node6+8 0 => 1431671312	
1813: 00		movq \$0x0,0x8(%rax) //1nodeF+8 清零	
1814:	bd 05 00 00 00	mov \$0x5,%ebp //原来是4	
1819:	eb 09	jmp 1824 <phase_6+0xf1>	
181b:	48 8b 5b 08	mov 0x8(%rbx),%rbx //rbx=2node1	
181f:	83 ed 01	sub \$0x1,%ebp	
1822:	74 11	je 1835 <phase_6+0x102>	
1824:	48 8b 43 08	mov 0x8(%rbx),%rax //rax: 1nodeF => 1nodeB	
1828:	8b 00	mov (%rax),%eax //eax=546 (1nodeB)	
182a:	39 03	cmp %eax,(%rbx) //1nodeA<= 1nodeB	要求链表值递增
182c:	7e ed	jle 181b <phase_6+0xe8>	
182e:	e8 de 04 00 00	call 1d11 <explode_bomb>	
1833:	eb e6	jmp 181b <phase_6+0xe8>	

手绘图解：



Secret_Phase---地狱难度走迷宫

最开始浏览整个汇编文件的时候就发现还有 Secret_phase, Invalid_phase 两个多余关卡。而经过试验发现后者无入口，强行进入会直接终结函数。那我们看怎么进 Secret_Phase。

利用文档搜索功能发现只有 Defused 函数里有调用他的地方，而之前正常流程没出现隐藏关想必是有些条件没满足。

我们看看他前文，兴许有些提示。

```
0x55555555fdd <phase_defused+145> jne    0x55555555fa7 <phase_defused+91>
0x55555555fdf <phase_defused+147> lea    0x1322(%rip),%rdi      # 0x555555557308
0x55555555fe6 <phase_defused+154> call   0x555555555070 <puts@plt>
0x55555555feb <phase_defused+159> lea    0x133e(%rip),%rdi      # 0x555555557330
0x55555555ff2 <phase_defused+166> call   0x555555555070 <puts@plt>
0x55555555ff7 <phase_defused+171> mov    $0x0,%eax
0x55555555ffc <phase_defused+176> call   0x55555555597a <secret_phase>
0x555555556001 <phase_defused+181> jmp   0x55555555fa7 <phase_defused+91>
```

把调用前的一些奇怪传参（把一些神秘的话传入 rdi 然后输出）打印出来看看。

```
(gdb) x/s $rip+0x130c
0x555555557258: "\nBy my efflux of deep crimson, topple this white world! EXPLOSION!!!"
```

Ps: 需要进行一些地址微调才能打印出完整内容（下面展示一些微调过程）

```
(gdb) x/s $rip+0x1302
0x55555555724e: ", do you?"
(gdb) x/s $rip+0x1308
0x555555557254: "ou?"
(gdb) x/s $rip+0x130c
0x555555557258: "\nBy my efflux of deep crimson, topple this white world! EXPLOSION!!!"
(gdb) x/s $rip+0x130a
0x555555557258: "Your instructor has been notified."
```

By my efflux of deep crimson, topple this white world! EXPLOSION!!!

这是我党恐怖袭击老蒋的口号吗？

再往上看：

```
5555519c <phase_defused+80> jg    0x55555555fa2 <phase_defused+91>
55555f9e <phase_defused+82> test  %cl,%cl
55555fa0 <phase_defused+84> jne   0x55555555f84 <phase_defused+54>
55555fa2 <phase_defused+86> cmp   $0x2,%edx
55555fa5 <phase_defused+89> je    0x55555555fc1 <phase_defused+117>
55555fa7 <phase_defused+91> lea    0x13ba(%rip),%rdi      # 0x555555557258
55555fae <phase_defused+98> call  0x555555555070 <puts@plt>
55555fb3 <phase_defused+103> lea    0x13de(%rip),%rdi      # 0x555555557258
55555fba <phase_defused+110> call  0x555555555070 <puts@plt>
55555fbf <phase_defused+115> jmp   0x555555555f63 <phase_defused+214>
55555fc1 <phase_defused+117> movslq %esi,%rsi
55555fc4 <phase_defused+120> lea    0x389d(%rip),%rax      # 0x555555557258
55555fcb <phase_defused+127> lea    (%rsi,%rax,1),%rdi
55555fcf <phase_defused+131> lea    0x12f2(%rip),%rsi      # 0x555555557258
55555fd6 <phase_defused+138> call  0x555555555aac <strings_not_equal>
55555fdb <phase_defused+143> test  %eax,%eax
55555fdd <phase_defused+145> jne   0x555555555fa7 <phase_defused+91>
55555fdf <phase defused+147> lea    0x1322(%rip),%rdi      # 0x555555557258
```

摸索到了进入奖励关的提示词!!

还一不小心看到了自以为很有趣的幼稚信息，
非常无语的断定这恐怕就是密钥了。

把他打印全：

```
(gdb) x/s $rip+0x12f2
0x555555555723e: "bomb with ctrl-c, do you?"
(gdb) x/s $rip+0x389d
0x555555557e9 <input_strings+233>:      ""
(gdb) x/s $rip+0x13de
0x55555555732a: "ase!""
(gdb) x/s $rip+0x13de
0x55555555732a: "ase!""
(gdb) x/s $rip+0x13ce
0x55555555731a: "nd the secret phase!""
(gdb) x/s $rip+0x13ae
0x5555555573fa: "ed by miHoYo"
(gdb) x/s $rip+0x13be
0x55555555730a: "rses, you've found the secret phase!""
(gdb) x/s $rip+0x13b8
0x555555557304: "Yo"
(gdb) x/s $rip+0x13ba
0x555555557306: ""
(gdb) x/s $rip+0x13bb
0x555555557307: ""
(gdb) x/s $rip+0x13bc
0x555555557308: "Curses, you've found the secret phase!"
```

Genshin Impact is an open-world action RPG developed by miHoYo

常见跳转：

test eax eax

je

查资料，等价于

If(eax==0 则跳转)

```
(gdb) x/s $rip+0x139e
0x5555555572ea: "tion RPG developed by miHoYo"
(gdb) Quit
(gdb) x/s $rip+0x137e
0x5555555572ca: "nshin Impact is an open-world action RPG developed by miHoYo"
(gdb) x/s $rip+0x135e
0x5555555572aa: "uctor has been notified."
(gdb) $rip+0x136e
Undefined command: "$rip+0x136e". Try "help".
(gdb) x/s $rip+0x136e
0x5555555572ba: "otified."
(gdb) x/s $rip+0x1368
0x5555555572b4: "been notified."
(gdb) x/s $rip+0x1378
0x5555555572c4: ""
(gdb) x/s $rip+0x137a
0x5555555572c6: ""
(gdb) x/s $rip+0x137b
0x5555555572c7: ""
(gdb) x/s $rip+0x137c
0x5555555572c8: "Genshin Impact is an open-world action RPG developed by miHoYo"
(gdb) █
```

密钥到手，在哪输入呢？

发现 defuse 函数只有在第六次调用的时候才会执行第一个 je 指令，跳过眼前的 ret，转到神秘区域：

```
0x55555555f5a <phase_defused+14>    cmpl   $0x6,0x3797(%rip)      # 0x5555555596f8
0x55555555f61 <phase_defused+21>    je     0x55555555f68 <phase_defused+28>
0x55555555f63 <phase_defused+23>    add    $0x8,%rsp
0x55555555f67 <phase_defused+27>    ret
0x55555555f68 <phase_defused+28>    movzbl 0x38f9(%rip),%ecx      # 0x555555559868
0x55555555f6f <phase_defused+35>    test   %cl,%cl
0x55555555f71 <phase_defused+37>    je     0x55555555fa7 <phase_defused+91>
0x55555555f73 <phase_defused+39>    mov    $0x1,%eax
0x55555555f78 <phase_defused+44>    mov    $0x0,%edx
0x55555555f7d <phase_defused+49>    lea    0x38e4(%rip),%rdi      # 0x555555559868
> 0x55555555f84 <phase_defused+56>    cmp    $0x20,%cl
```

繁琐，理论上能找到一个地址存着输入格式，根据此格式找到对应输入位置。可是这个密钥一共就没几个可能的位置（每题答案之后 or 最后一题之后）简单尝试后发现是第四题后面。人生苦短，还有一个整个 phase 等着你呢，先把他干了吧。

简单概括下进门要点：

1. 找 Secret_Phase 入口 (call Secret_Phase)

发现只在 defused 函数里出现 且 大前提条件是做完第六题

2. 翻找 call Secret 前的一些传参，大量的打印，最终找到密钥和密钥输入格式

与 `cmp` 指令和 跳转 指令组合的区别是：这个组合比较的是 `cmp A,B` 中， `A` 与 `B` 的关系。

而 `test A,A` 则比较的是 `A` 与 `0` 的关系。

```
test    edx,edx  
jle    某地址
```

：如果 `edx <= 0`，就跳到 `某地址`，否则继续往下执行

成 `jg` 的话，就是 `edx > 0` 跳转。

接下来就是，扎扎实实的漫长解读....数不清花了多少时间....

我把内容先初步解读在 word 文档 dump 文字稿中，接着又翻译了在 A4 纸上（更贴近 c 语言），这时候程序在干什么已经初见端倪了。经过交流研讨，我恍然大悟，醍醐灌顶。发现自己太拘泥于汇编语言本身，而缺乏“想象力”，以致于出现“只见表象之运算，不见抽象之功能”的可怕现象，真可谓一叶障目，不见泰山于前。代价难以想象...
不想赘述一遍程序流程了，只简要概括一下大概思路吧。

Secret_phase 函数主要做的事情有 4 件：

1. 录入一个神秘数组 我们称之为 Key[4]: 4 3 2 5
2. 读取 input 并验证长度(<70)
3. 调用 fun7....
4. if (返回值==4) 解开！

那 fun7 干了什么呢？我的妈，他可真是城府极深啊...

我猜相比于汇编细节，助教和老师们想必更关注我们是否从茫茫汇编的解读里升华领悟到了程序的真正意思（简单到能像讲一个小故事一样说出来），于是不必赘述细节（我想，语言描述起来这报告恐怕要浩如烟海了，不如直接上文档证据、上演草图。dump 文档有我反复的翻译记录 后面附图的照片 A4 显示了我尝试概括归纳汇编程序形成理解的过程，包含很多细节）

input 字符串%4 转化为 0 1 2 3，

0 1 2 3 分别对应 x+1 (右行) x-1 (左) y+1 (下) y-1 (上)

这是用来走迷宫的。。。迷宫当然蕴藏在链表 line 里 (1 3 5 7 2 4 6 8)

(注意小端存储)

这是一个走迷宫游戏，每走一步，都是一层 fun7 递归....

从 (0,0) 沿着非 1 路径 (1 视为墙壁)，以此经过 key 数组元素

4 3 2 5，累加分数，最后走到终点 (7,7)。游戏成功结束

(如图)



