Phase_1:

```
kurumi@kurumi:/mnt/d/another_C/bomblab-handout$ gdb bomb
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(No debugging symbols found in bomb)
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000401545 <+0>:     endbr64
   0x0000000000401549 <+4>:     push   %rbp
   0x000000000040154a <+5>:     mov    %rsp,%rbp
   0x000000000040154d <+8>:     sub    $0x20,%rsp
   0x0000000000401551 <+12>:    mov    %rdi,-0x18(%rbp)
   0x0000000000401555 <+16>:    mov    0x4df1(%rip),%eax        # 0x40634c <phase_1_offset>
   0x000000000040155b <+22>:    cltq
   0x000000000040155d <+24>:    lea    0x4b5c(%rip),%rdx        # 0x4060c0 <phase_1_str>
   0x0000000000401564 <+31>:    add    %rdx,%rax
   0x0000000000401567 <+34>:    mov    %rax,-0x8(%rbp)
   0x000000000040156b <+38>:    mov    -0x8(%rbp),%rdx
   0x000000000040156f <+42>:    mov    -0x18(%rbp),%rax
   0x0000000000401573 <+46>:    mov    %rdx,%rsi
   0x0000000000401576 <+49>:    mov    %rax,%rdi
   0x0000000000401579 <+52>:    call   0x4014e2 <string_not_equal>
   0x000000000040157e <+57>:    xor    $0x1,%eax
   0x0000000000401581 <+60>:    test   %al,%al
   0x0000000000401583 <+62>:    je     0x40158a <phase_1+69>
   0x0000000000401585 <+64>:    call   0x402048 <explode_bomb>
   0x000000000040158a <+69>:    nop
   0x000000000040158b <+70>:    leave
   0x000000000040158c <+71>:    ret
End of assembler dump.
(gdb)
```

分析 phase_1 调用函数 string_not_equal 的功能：将%rdi 和%rsi 的值写入栈中，之后获得第一个字符串中当前字符的地址，并将其加载到%al 寄存器中，获得第二个字符串中当前字符的地址，并将其加载到%dl 寄存器中。Cmp 比较这俩寄存器中值是否相等，相等则 add1，也就是比较下一个，不相等直接返回 0，最终若字符串相等，将返回 1（存在%eax 中）。回到 phase_1：将%exa 的值与 1 进行异或操作，也就是，如果是相等，返回 1，经过异或为 0。接下来测试%al 寄存器的值（即%eax 的低 8 位），是 0，跳过爆炸，得到答案。

```
(gdb) disas string_not_equal
Dump of assembler code for function string_not_equal:
   0x00000000004014e2 <+0>:     endbr64
   0x00000000004014e6 <+4>:     push   %rbp
   0x00000000004014e7 <+5>:     mov    %rsp,%rbp
   0x00000000004014ea <+8>:     mov    %rdi,-0x18(%rbp)
   0x00000000004014ee <+12>:    mov    %rsi,-0x20(%rbp)
   0x00000000004014f2 <+16>:    movl   $0x0,-0x4(%rbp)
   0x00000000004014f9 <+23>:    mov    -0x4(%rbp),%eax
   0x00000000004014fc <+26>:    movslq %eax,%rdx
   0x00000000004014ff <+29>:    mov    -0x20(%rbp),%rax
   0x0000000000401503 <+33>:    add    %rdx,%rax
   0x0000000000401506 <+36>:    movzbl (%rax),%eax
   0x0000000000401509 <+39>:    test   %al,%al
   0x000000000040150b <+41>:    je     0x40153e <string_not_equal+92>
   0x000000000040150d <+43>:    mov    -0x4(%rbp),%eax
   0x0000000000401510 <+46>:    movslq %eax,%rdx
   0x0000000000401513 <+49>:    mov    -0x18(%rbp),%rax
   0x0000000000401517 <+53>:    add    %rdx,%rax
   0x000000000040151a <+56>:    movzbl (%rax),%edx
   0x000000000040151d <+59>:    mov    -0x4(%rbp),%eax
   0x0000000000401520 <+62>:    movslq %eax,%rcx
   0x0000000000401523 <+65>:    mov    -0x20(%rbp),%rax
   0x0000000000401527 <+69>:    add    %rcx,%rax
   0x000000000040152a <+72>:    movzbl (%rax),%eax
   0x000000000040152d <+75>:    cmp    %al,%dl
   0x000000000040152f <+77>:    je     0x401538 <string_not_equal+86>
   0x0000000000401531 <+79>:    mov    $0x0,%eax
   0x0000000000401536 <+84>:    jmp    0x401543 <string_not_equal+97>
   0x0000000000401538 <+86>:    addl   $0x1,-0x4(%rbp)
--Type <RET> for more, q to quit, c to continue without paging--c
   0x000000000040153c <+90>:    jmp    0x4014f9 <string_not_equal+23>
   0x000000000040153e <+92>:    mov    $0x1,%eax
   0x0000000000401543 <+97>:    pop    %rbp
   0x0000000000401544 <+98>:    ret
End of assembler dump.
```

Phase_1_offset 就存储了偏移值，我们用 x /x 语句以 16 进制查看它存储的偏移值

```
(gdb) x /x  0x40634c
0x40634c <phase_1_offset>:      0x000000ee
```

偏移量是 0xee，再和 phase_1_str 的地址（4060c0）相加后得到最后地址后四位是 1ae，则查询 0x4061ae 存储的内容

```
(gdb) x /s 0x4061ae
0x4061ae <phase_1_str+238>:     "Computer science is not a boring subject"
```

得到答案。

Phase_2

```
   0x00000000004021aa <+4>:     push   %rbp
   0x00000000004021ab <+5>:     mov    %rsp,%rbp
   0x00000000004021ae <+8>:     sub    $0x20,%rsp
   0x00000000004021b2 <+12>:    mov    %rdi,-0x18(%rbp)
   0x00000000004021b6 <+16>:    mov    %rsi,-0x20(%rbp)
   0x00000000004021ba <+20>:    mov    -0x20(%rbp),%rax
   0x00000000004021be <+24>:    lea    0x14(%rax),%rdi
   0x00000000004021c2 <+28>:    mov    -0x20(%rbp),%rax
   0x00000000004021c6 <+32>:    lea    0x10(%rax),%rsi
   0x00000000004021ca <+36>:    mov    -0x20(%rbp),%rax
   0x00000000004021ce <+40>:    lea    0xc(%rax),%r9
   0x00000000004021d2 <+44>:    mov    -0x20(%rbp),%rax
   0x00000000004021d6 <+48>:    lea    0x8(%rax),%r8
   0x00000000004021da <+52>:    mov    -0x20(%rbp),%rax
   0x00000000004021de <+56>:    lea    0x4(%rax),%rcx
   0x00000000004021e2 <+60>:    mov    -0x20(%rbp),%rdx
   0x00000000004021e6 <+64>:    mov    -0x18(%rbp),%rax
   0x00000000004021ea <+68>:    push   %rdi
   0x00000000004021eb <+69>:    push   %rsi
   0x00000000004021ec <+70>:    lea    0x1180(%rip),%rsi        # 0x403373
   0x00000000004021f3 <+77>:    mov    %rax,%rdi
   0x00000000004021f6 <+80>:    mov    $0x0,%eax
   0x00000000004021fb <+85>:    call   0x401160 <__isoc99_sscanf@plt>
   0x0000000000402200 <+90>:    add    $0x10,%rsp
   0x0000000000402204 <+94>:    mov    %eax,-0x4(%rbp)
   0x0000000000402207 <+97>:    cmpl   $0x6,-0x4(%rbp)
   0x000000000040220b <+101>:   jne    0x402217 <read_six_numbers+113>
--Type <RET> for more, q to quit, c to continue without paging--
   0x000000000040220d <+103>:   mov    -0x20(%rbp),%rax
   0x0000000000402211 <+107>:   mov    (%rax),%eax
   0x0000000000402213 <+109>:   test   %eax,%eax
   0x0000000000402215 <+111>:   jne    0x40221c <read_six_numbers+118>
   0x0000000000402217 <+113>:   call   0x402048 <explode_bomb>
   0x000000000040221c <+118>:   nop
   0x000000000040221d <+119>:   leave
   0x000000000040221e <+120>:   ret
```

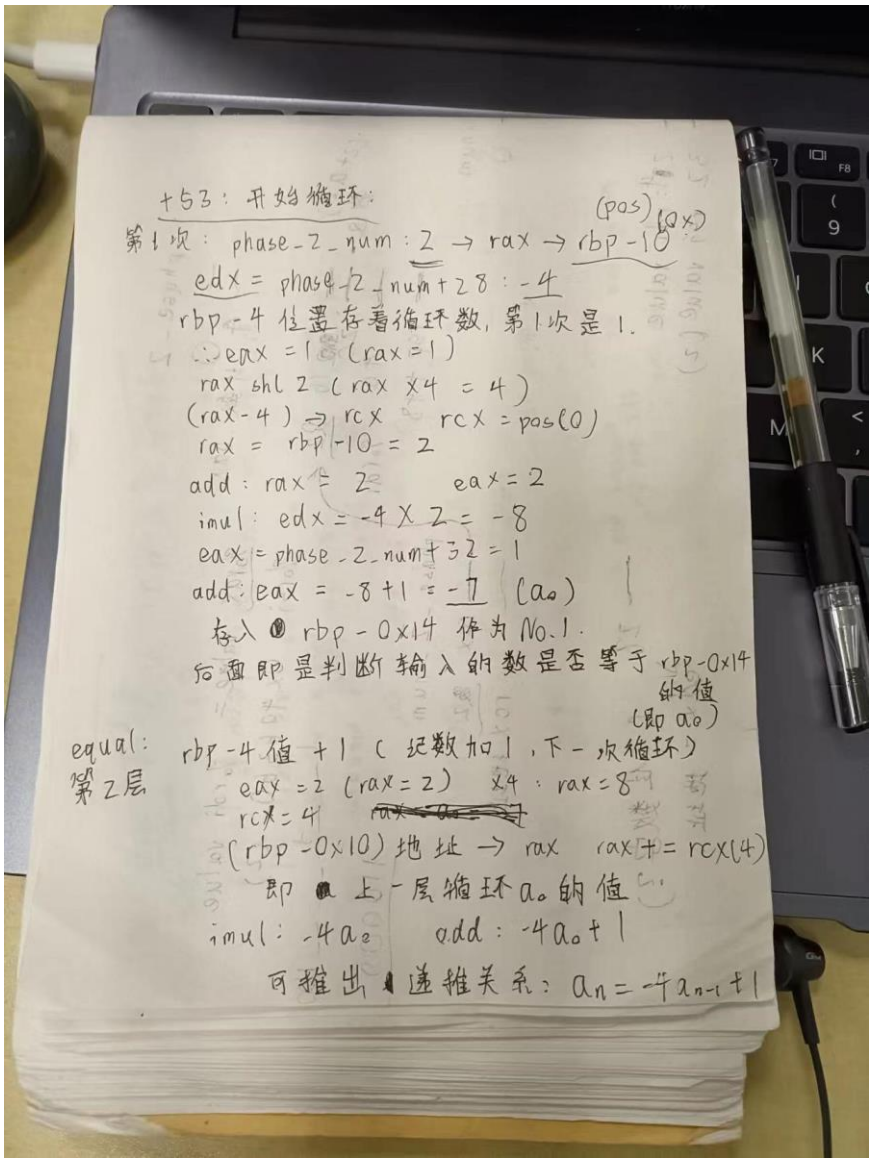
题解步骤如下，是一个循环

```
Invalid character '#' in expression.
(gdb) x/s  0x403373
0x403373:       "%d %d %d %d %d %d"
```

发现要输入六个整数。从+53 那里开始循环，栈里面找了个地方记录循环次数

```
(gdb) x/gx 0x406360
0x406360 <phase_2_nums>:        0x0000000400000002
(gdb) x/gx  0x40637c
0x40637c <phase_2_nums+28>:     0x00000001fffffffc
(gdb) x/gx  0x406380
0x406380 <phase_2_nums+32>:     0x0000000000000001
```

三个关键数据大小。



Phase_3

```
(gdb) x/s 0x403248
0x403248:       "%d %d %c"
```

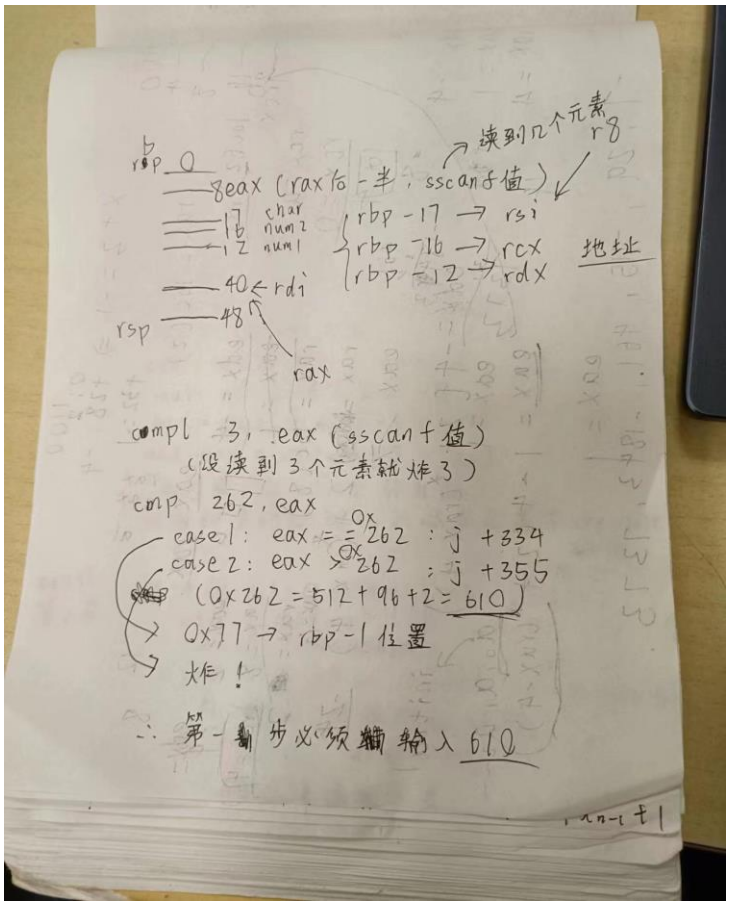
要输入“整数 整数 字符”的形式存放到 rsi 中。根据前面 rbp-17 的地址赋给 rsi，rbp-16 的地址赋给 rcs，rbp-12 的地址赋给 rdx，刚好是 4 4 1byte 对应起来了。

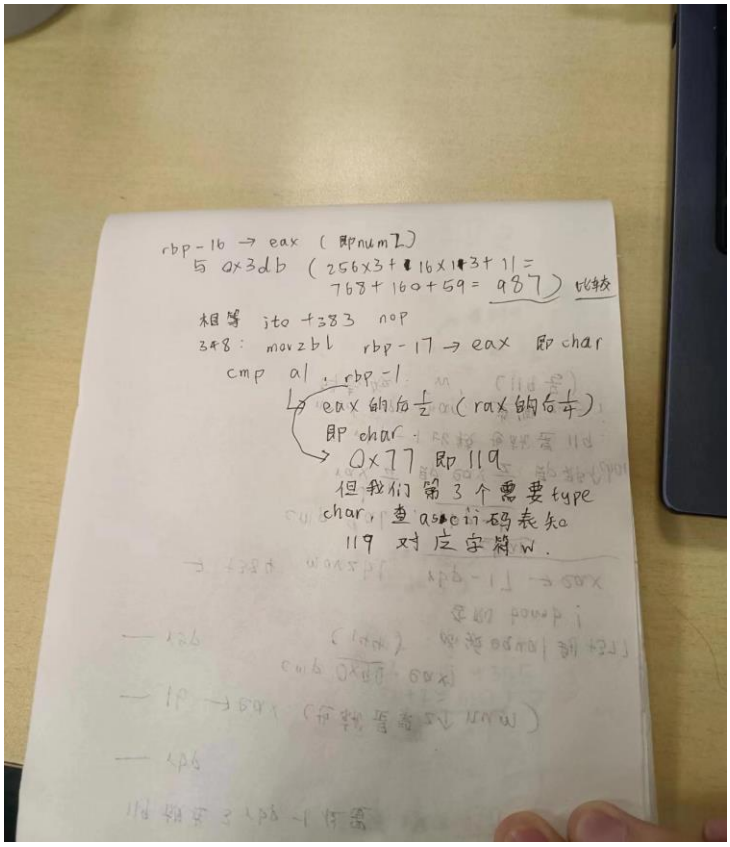
（疑惑的点在第+88 行往后：但是这里先不解决，secret 了再回来看

```
668 <+77>:    je     0x401769 <phase_3+334>
66e <+83>:    cmp    $0x262,%eax
673 <+88>:    jg     0x40177e <phase_3+355>
679 <+94>:    cmp    $0xe9,%eax
67e <+99>:    je     0x401754 <phase_3+313>
684 <+105>:   cmp    $0xe9,%eax
689 <+110>:   jg     0x40177e <phase_3+355>
```

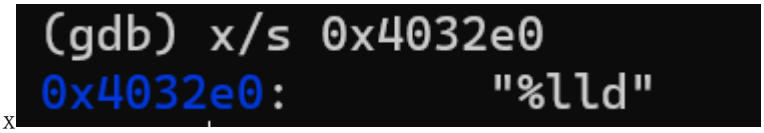
若 num1 小于 610？没有跳转往后走了，或许是不只一个答案，也有可能是 secret 入口？）

分析思路如下：逐条分析跳转。





Phase_4

x 

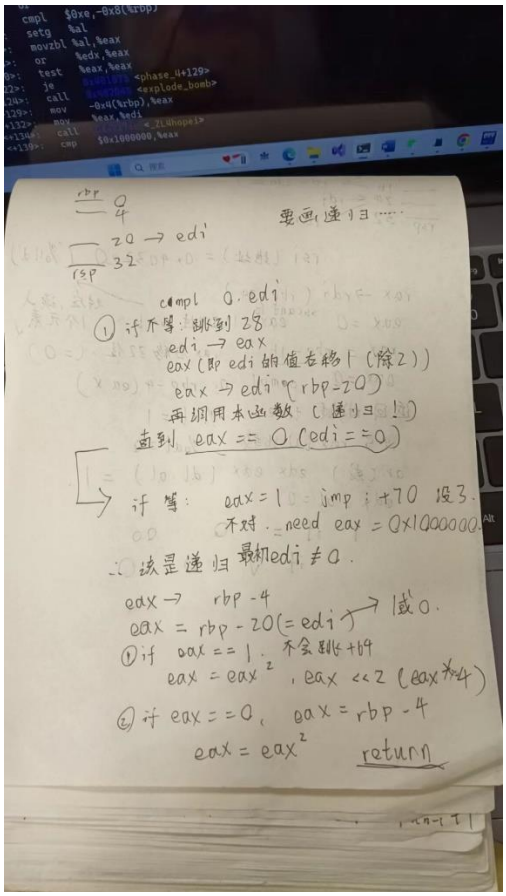
要输入一 long long int 整数

前面都是些开栈存储的操作。我先从调用函数之后逆向分析：

```
   0x000000000040187a <+134>:   call   0x4017ac <_ZL4hopei>
   0x000000000040187f <+139>:   cmp    $0x1000000,%eax
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401884 <+144>:   setne  %al
   0x0000000000401887 <+147>:   test   %al,%al
   0x0000000000401889 <+149>:   je     0x401890 <phase_4+156>
   0x000000000040188b <+151>:   call   0x402048 <explode_bomb>
   0x0000000000401890 <+156>:   nop
   0x0000000000401891 <+157>:   leave
   0x0000000000401892 <+158>:   ret
End of assembler dump.
```

如果：比较很容易想到需要 eax 和 0x1000000 相等，就这样往下推一下：如果 eax（rax，函数返回值是 0x1000000，则 setne（不相等则设置） al 被置为 0，则 test 0，0（与操作）后 ZF 被置为 1，je 会跳转（跳过爆炸），这才是我们需要的。于是就可以推出，eax（函数返回值）必须是 0x1000000。

接下来我们再来看这个递归函数：分析如下

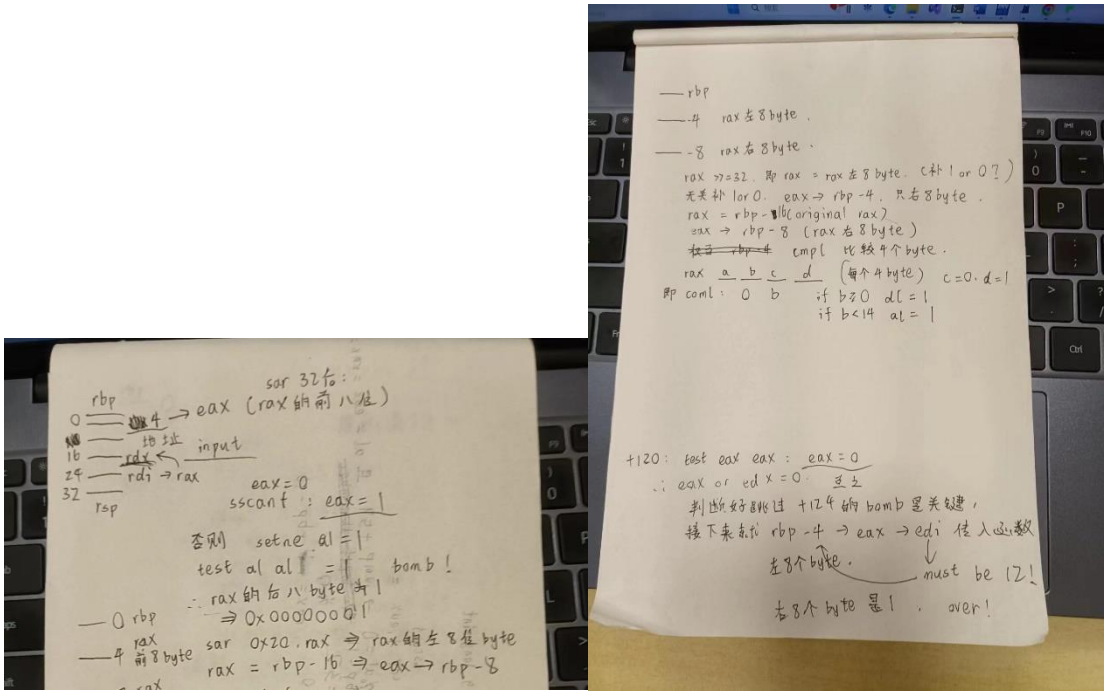


这段递归过程画起来属实麻烦，但是考虑将其转成 c++语言就很好调试了。需要 16 的 6 次

方，即 2 的 24 次方。模拟等价代码如下：（这是关键）

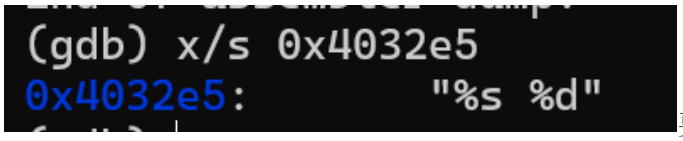
```cpp
#include<iostream>
using namespace std;

int hope(int value)
{
    if (value == 0) return 1;
    else
    {
        int result = hope(value >> 1); // 右移一位相当于除以2
        if (value & 1) return result * result * 4;
        else return result * result;
    }
}


int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    cout<<hope(4);
    return 0;
}
```

我发现输入 0 得到 1，输入 1 得到 4，输入 4 得到 256，推测功能就是 4 的 n 次方，我们需要 4 的 12 次方。即，输入数该是 12，即，原来寄存器 edi 传来的值应该是 12。
再来从 phase_4 主体开始看：

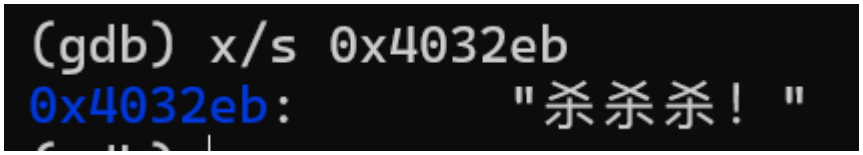


最后结果：rax 左八个 byte 是 12，右八个 byte 是 1。

Phase_5



要输入一个字符串和一个整数



？？？乐

往下看，发现总共有三个地方是可以判断输入的字符串到底是啥的。先知道了在标准的 C 库中，strcmp 函数用于比较两个字符串的内容，**如果两个字符串相等，则返回 0（这一个非常重要，而不是我想当然的相等返回 1）**，如果不相等则返回非零值。

1. 如果是“杀杀杀!”返回 0，不跳到 114 行，一直做跳到 219 行过了 bomb，可行
2. 如果是“退退退。”78 行返回 1，跳到 114 行，128 行返回 0，跳到 219 行，可行
3. 如果是“冲冲冲~”，78—》114—》164，178 行返回 0，到 219 行，可行

都不会炸。可以断定%s 是这三个中的一个。

接下来判断整数是个啥。逆向倒推：

```
   0x0000000000401995 <+258>:   call   0x401ea6 <_ZN9worldline18is_phase5_passableEv>
   0x000000000040199a <+263>:   test   %eax,%eax
   0x000000000040199c <+265>:   jne    0x4019a5 <phase_5+274>
   0x000000000040199e <+267>:   mov    $0x1,%eax
   0x00000000004019a3 <+272>:   jmp    0x4019aa <phase_5+279>
   0x00000000004019a5 <+274>:   mov    $0x0,%eax
   0x00000000004019aa <+279>:   test   %al,%al
   0x00000000004019ac <+281>:   je     0x4019b3 <phase_5+288>
   0x00000000004019ae <+283>:   call   0x402048 <explode_bomb>
   0x00000000004019b3 <+288>:   nop
   0x00000000004019b4 <+289>:   add    $0x48,%rsp
   0x00000000004019b8 <+293>:   pop    %rbx
   0x00000000004019b9 <+294>:   pop    %rbp
```

在调用 0x401ea6 <_ZN9worldline18is_phase5_passableEv> 之后，返回的 eax 要是是 0，eax 被赋值为 1，跳到 279 行，test 是 1，不跳，bomb！　所以：函数返回的 eax 一定要是非 0。

```
Dump of assembler code for function _ZN9worldline18is_phase5_passableEv:
   0x0000000000401ea6 <+0>:     endbr64
   0x0000000000401eaa <+4>:     push   %rbp
   0x0000000000401eab <+5>:     mov    %rsp,%rbp
   0x0000000000401eae <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000000000401eb2 <+12>:    mov    -0x8(%rbp),%rax
   0x0000000000401eb6 <+16>:    mov    0x8(%rax),%rax
   0x0000000000401eba <+20>:    cmp    $0xf423f,%rax
   0x0000000000401ec0 <+26>:    setg   %al
   0x0000000000401ec3 <+29>:    movzbl %al,%eax
   0x0000000000401ec6 <+32>:    pop    %rbp
   0x0000000000401ec7 <+33>:    ret
```

重点是+20 那里，也就是 rax（即传入参数 rdi）一定要大于 0xf423f，al（eax）才被设为 1。

```
or function _ZN10worldline1C2Ev:
>:      endbr64
4>:     push   %rbp
5>:     mov    %rsp,%rbp
8>:     sub    $0x10,%rsp
12>:    mov    %rdi,-0x8(%rbp)
16>:    mov    -0x8(%rbp),%rax
20>:    mov    %rax,%rdi
23>:    call   0x401e88 <_ZN9worldl
28>:    lea    0x3e85(%rip),%rdx
35>:    mov    -0x8(%rbp),%rax
39>:    mov    %rdx,(%rax)
42>:    mov    -0x8(%rbp),%rax
46>:    movq   $0x8b690,0x8(%rax)
54>:    nop
55>:    leave
56>:    ret
```

```
or function _ZN10worldline2C2Ev:
0>:     endbr64
4>:     push   %rbp
5>:     mov    %rsp,%rbp
8>:     sub    $0x10,%rsp
12>:    mov    %rdi,-0x8(%rbp)
16>:    mov    -0x8(%rbp),%rax
20>:    mov    %rax,%rdi
23>:    call   0x401e88 <_ZN9worldline
28>:    lea    0x3ddd(%rip),%rdx
35>:    mov    -0x8(%rbp),%rax
39>:    mov    %rdx,(%rax)
42>:    mov    -0x8(%rbp),%rax
46>:    movq   $0x6f8d2,0x8(%rax)
54>:    nop
55>:    leave
56>:    ret
```

```
tion _ZN10worldline3C2Ev:
 endbr64
 push   %rbp
 mov    %rsp,%rbp
 sub    $0x10,%rsp
 mov    %rdi,-0x8(%rbp)
 mov    -0x8(%rbp),%rax
 mov    %rax,%rdi
 call   0x401e88 <_ZN9worldlin
 lea    0x3d35(%rip),%rdx
 mov    -0x8(%rbp),%rax
 mov    %rdx,(%rax)
 mov    -0x8(%rbp),%rax
 movq   $0x1124fd,0x8(%rax)
 nop
 leave
```

验证后发现仅有三满足条件。（“冲冲冲~”）

```
0x0000000000401935 <+162>:   jmp    0x40196e <phase_5+219>
0x0000000000401937 <+164>:   lea    -0x30(%rbp),%rax
0x000000000040193b <+168>:   lea    0x19c3(%rip),%rsi        # 0x403305
0x0000000000401942 <+175>:   mov    %rax,%rdi
0x0000000000401945 <+178>:   call   0x4011e0 <strcmp@plt>
0x000000000040194a <+183>:   test   %eax,%eax
0x000000000040194c <+185>:   jne    0x401969 <phase_5+214>
0x000000000040194e <+187>:   mov    $0x10,%edi
0x0000000000401953 <+192>:   call   0x401180 <_Znwm@plt>
0x0000000000401958 <+197>:   mov    %rax,%rbx
0x000000000040195b <+200>:   mov    %rbx,%rdi
0x000000000040195e <+203>:   call   0x401fc8 <_ZN10worldline3C2Ev>
0x0000000000401963 <+208>:   mov    %rbx,-0x18(%rbp)
0x0000000000401967 <+212>:   jmp    0x40196e <phase_5+219>
0x0000000000401969 <+214>:   call   0x402048 <explode_bomb>
0x000000000040196e <+219>:   mov    -0x18(%rbp),%rax
0x0000000000401972 <+223>:   mov    (%rax),%rax
0x0000000000401975 <+226>:   add    $0x10,%rax
0x0000000000401979 <+230>:   mov    (%rax),%rcx
0x000000000040197c <+233>:   mov    -0x34(%rbp),%edx
0x000000000040197f <+236>:   mov    -0x18(%rbp),%rax
0x0000000000401983 <+240>:   mov    %edx,%esi
0x0000000000401985 <+242>:   mov    %rax,%rdi
0x0000000000401988 <+245>:   call   *%rcx
0x000000000040198a <+247>:   test   %eax,%eax
0x000000000040198c <+249>:   je     0x40199e <phase_5+267>
```

来看 3 的部分：比较字符串相等返回 eax 为 0，edi 赋为 16，开空间（这个开空间的函数没看太懂，好像就是 c++的 class 那种操作？反之没有对 rax 的值啊啥的做出任何修改，单纯只是开了个空间，不理会它）rdi=rbx=rax，调用函数 ZN10，ZN10 又调用了 ZN9，ZN9 相当于是个初始化（类似）的功能，总之 ZN9 修改的值传回来后 ZN10 又会把它修改掉。
ZN10 将 0x405d20 这个地址的值传到（rax）里面返回了（相当于 rax 指向这个地址）。再往下看+223 开始，这个地址指向的值赋给 rax（0x405d20）并加 10 赋给 rcx。接下来是关键：到了用 rbp-0x34 的值的时候了（之前完全没出现过，这地方存的就是输入的整数，也就是说终于到告诉我整数该输入什么的时候了），调用了*rcx（rcx 存的值为地址的函数）（也就是 0x405d30 这个地址的存的值为地址的函数（有个*，解引用）。找到答案。

```
(gdb) x/gx 0x405d30
0x405d30 <_ZTV10worldline3+32>: 0x000000000040202a
(gdb) disas 40202a
Invalid number "40202a".
(gdb) disas 0x40202a
Dump of assembler code for function _ZN10worldline35dmailEi:
   0x000000000040202a <+0>:     endbr64
   0x000000000040202e <+4>:     push   %rbp
   0x000000000040202f <+5>:     mov    %rsp,%rbp
   0x0000000000402032 <+8>:     mov    %rdi,-0x8(%rbp)
   0x0000000000402036 <+12>:    mov    %esi,-0xc(%rbp)
   0x0000000000402039 <+15>:    cmpl   $0x7e7,-0xc(%rbp)
   0x0000000000402040 <+22>:    sete   %al
   0x0000000000402043 <+25>:    movzbl %al,%eax
   0x0000000000402046 <+28>:    pop    %rbp
   0x0000000000402047 <+29>:    ret
End of assembler dump.
```

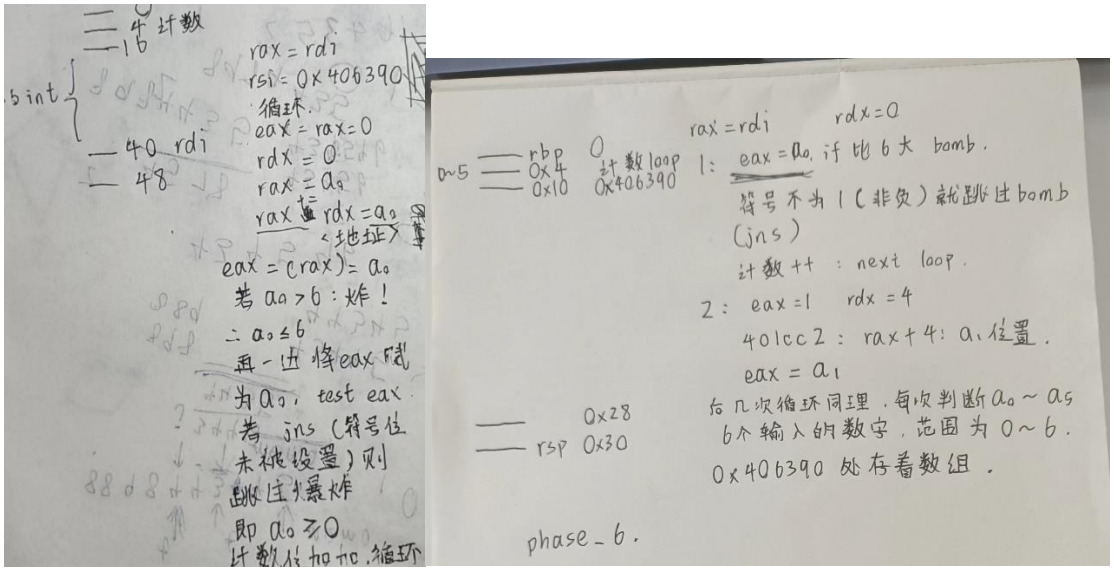
Phase_6
前半部分是读入六个数字，好像进行了许多判断如何不会炸，稍后再看。我还是喜欢从后往前看起：

```
<+130>:    call   0x401a96 <build_target>
<+135>:    mov    %al,-0x11(%rbp)
<+138>:    movzbl -0x11(%rbp),%eax
<+142>:    xor    $0x1,%eax
<+145>:    test   %al,%al
<+147>:    je     0x401d10 <phase_6+154>
, q to quit, c to continue without paging--c
<+149>:    call   0x402048 <explode_bomb>
```

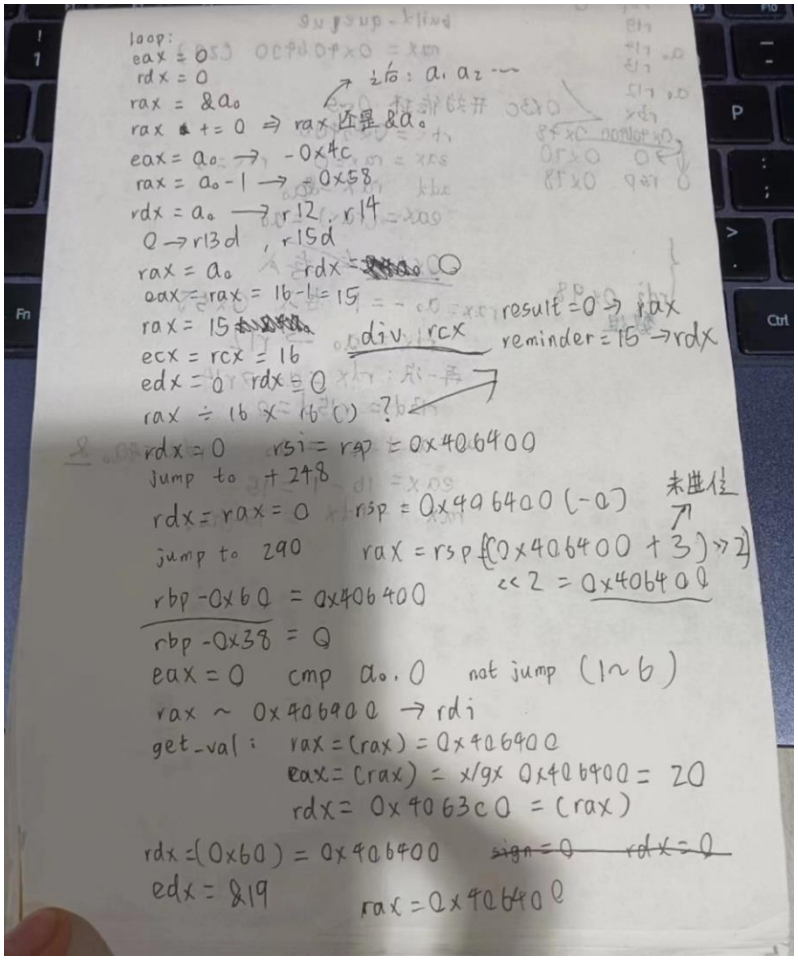
Build_target 函数返回值如果是 1，eax=1，与 1xor 异或为 0，跳过 bomb，即需要该函数返回 1。接下来看了看 build_targret 函数：（巨长无比……）我选择再从头开始（）

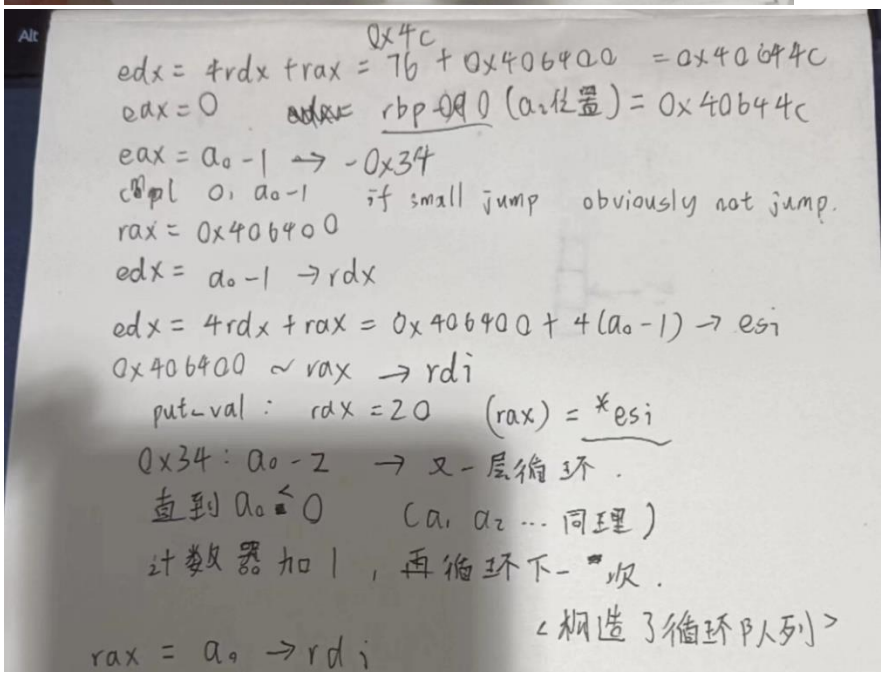
发现前面又是像第二题一样的读入六个数字，然后一个循环判断每一个数的大小范围。注意 rax 接受的是 lea，在栈-0x10 到-0x28 的六个整数位置上存的是读入的六个数的地址，所以 rdx 每次是 0，4，8……加到 rax 上，相当于是移到数组的下一个数字来判断，分析结果如下图所示：

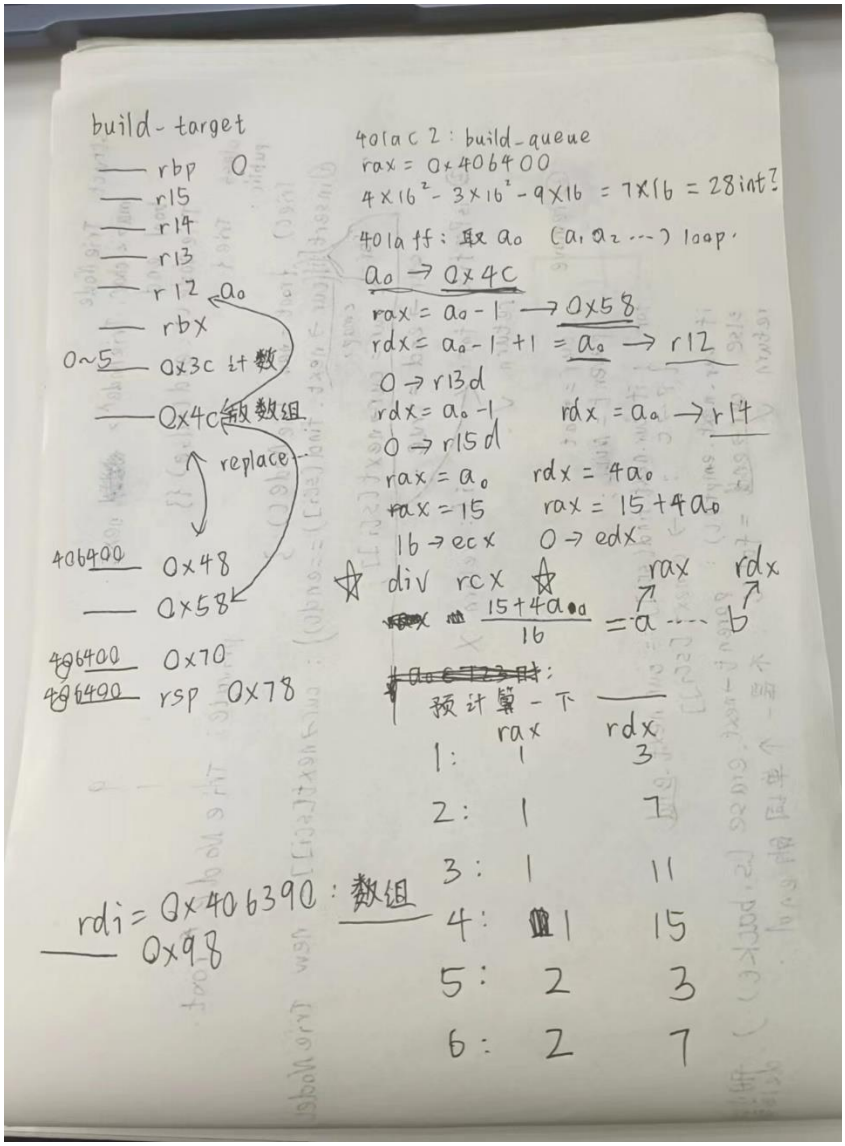
得到输入的数字必须为 1~6 这六个数字。（key）

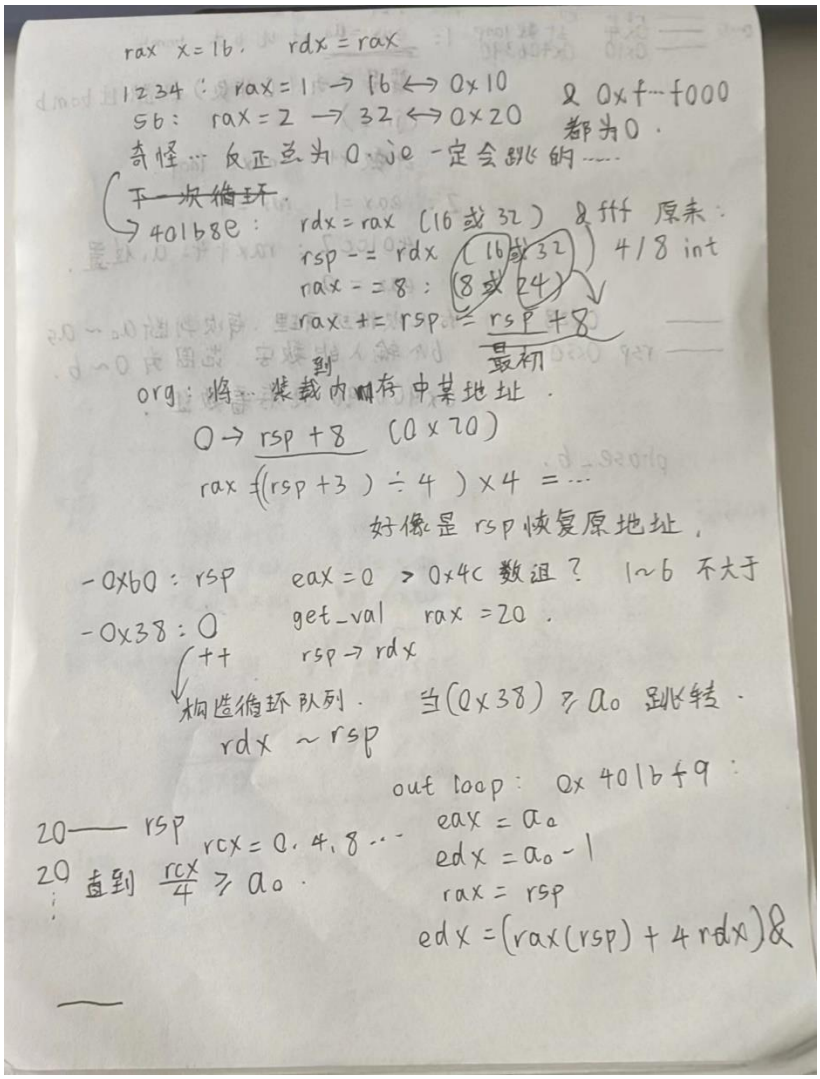


接下来分析 build_target：（基本只过了一遍，过了第一次循环）


loop:
eax = 0
rdx = 0
rax = &a0
rax += 0 ⇒ rax 还是 &a0
eax = a0 → -0x4c
rax = a0 - 1 → -0x58
rdx = a0 → r12, r14
0 → r13d, r15d
rax = a0 rdx = 0
eax = rax = 16-1 = 15
rax = 15
ecx = rcx = 16
edx = 0 rdx = 0
rax ÷ 16 (16()) ?
div rcx
result = 0 → rax
reminder = 15 → rdx
rdx = 0 rsi = rsp = 0x406400
jump to +248
rdx = rax = 0 rsp = 0x406400 (-0)
jump to 290
rax = rsp[(0x406400 + 3) >> 2] << 2 = 0x406400
rbp - 0x60 = 0x406400
rbp - 0x38 = 0
eax = 0 cmp a0, 0 not jump (1~6)
rax ~ 0x406400 → rdi
get_val: rax = (rax) = 0x406400
eax = (rax) = x/gx 0x406400 = 20
rdx = 0x4063c0 = (rax)
rdx = (0x60) = 0x406400
edx = 819
rax = 0x406400


edx = 4rdx + rax = 76 + 0x406400 = 0x40644c
0x4c
eax = 0
rbp - 0x40 (a1位置) = 0x40644c
eax = a0 - 1 → -0x34
cmpl 0, a0 - 1 if small jump obviously not jump.
rax = 0x406400
edx = a0 - 1 → rdx
edx = 4rdx + rax = 0x406400 + 4(a0 - 1) → esi
0x406400 ~ rax → rdi
put_val: rax = 20 (rax) = *esi
0x34: a0 - 2 → 又一层循环.
直到 a0 ≤ 0 (a1, a2 … 同理)
计数器加1，再循环下一次.
<构造了循环队列>
rax = a0 → rdi


build_target
rbp 0
r15
r14
r13
r12 a0
rbx
0~5 0x3c 计数
0x4c 数组
replace
406400 0x48
0x58
406400 0x70
406490 rsp 0x78
401ac2: build_queue
rax = 0x406400
4×16²-3×16²-9×16 = 7×16 = 28int?
401aff: 取 a0 (a1 a2 ...) loop.
a0 → 0x4c
rax = a0 - 1 → 0x58
rdx = a0 - 1 + 1 = a0 → r12
0 → r13d
rdx = a0 - 1
0 → r15d
rdx = a0 → r14
rax = a0
rax = 15
16 → ecx
rdx = 4a0
rax = 15 + 4a0
0 → edx
div rcx
rax
rdx
15+4a0 / 16 = a ... b
预计算一下
rax rdx
1: 1 3
2: 1 7
3: 1 11
4: 1 15
5: 2 3
6: 2 7
rdi = 0x406390: 数组
0x98



在 build_target 中有一个非常重要的是 div rcx，这句话是把 rax/rcx，计算得到的值存入 rax 并且余数存入 rdx。

r12 和 r14 在每一次循环中都储存着数组 a0—a6，rcx 则存储着 a0-1, a1-1……a6-1

```
Breakpoint 9, 0x0000000000401a33 in check_answer ()
(gdb) i r
rax            0x7fffffffdc00      140737488346112
rbx            0x7fffffffdbf0      140737488346096
rcx            0x1                 1
rdx            0x4063e0            4219872
rsi            0x28                40
rdi            0x7fffffffdc00      140737488346112
rbp            0x7fffffffdbe0      0x7fffffffdbe0
rsp            0x7fffffffdbe0      0x7fffffffdbe0
r8             0x1999999999999999  1844674407370955161
r9             0x0                 0
r10            0x7ffff7d20ac0      140737351125696
r11            0x7ffff7d213c0      140737351128000
r12            0x2                 2
r13            0x0                 0
```

最后 check_answer 是将 dc00 处的首地址传进来的。
对应前面在 4063c0—406410 处的六个值即可最终找到答案：

```
(gdb) i e
Ambiguous info command "e": exceptions, extensions.
(gdb) i r
rax            0x1                 1
rbx            0x7fffffffdbf0      140737488346096
rcx            0x0                 0
rdx            0x4063c0            4219840
rsi            0x14                20
rdi            0x7fffffffdc18      140737488346136
rbp            0x7fffffffdc90      0x7fffffffdc90
rsp            0x7fffffffdbf0      0x7fffffffdbf0
r8             0x1999999999999999  1844674407370955161
r9             0x0                 0
r10            0x7ffff7d20ac0      140737351125696
r11            0x7ffff7d213c0      140737351128000
r12            0x1                 1
r13            0x0                 0
r14            0x1                 1
r15            0x0                 0
rip            0x401af7            0x401af7 <build_target+97>
eflags         0x297               [ CF PF AF SF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) ni
```

```
(gdb) ni
0x0000000000401c24 in build_target ()
(gdb) i r
rax            0x7fffffffdc18      140737488346136
rbx            0x7fffffffdbf0      140737488346096
rcx            0x5                 5
rdx            0x4063d0            4219856
rsi            0xa                 10
rdi            0x7fffffffdc18      140737488346136
rbp            0x7fffffffdc90      0x7fffffffdc90
rsp            0x7fffffffdbd0      0x7fffffffdbd0
r8             0x1999999999999999  1844674407370955161
r9             0x0                 0
r10            0x7ffff7d20ac0      140737351125696
r11            0x7ffff7d213c0      140737351128000
r12            0x6                 6
r13            0x0                 0
r14            0x6                 6
r15            0x0                 0
rip            0x401c24            0x401c24 <build_target+398>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) next
```

```
Breakpoint 6, 0x0000000000401c1e in build_target ()
(gdb) i r
rax            0x7fffffffdc18      140737488346136
rbx            0x7fffffffdbf0      140737488346096
rcx            0x3                 3
rdx            0x4063e0            4219872
rsi            0xa                 10
rdi            0x7fffffffdc18      140737488346136
rbp            0x7fffffffdc90      0x7fffffffdc90
rsp            0x7fffffffdbe0      0x7fffffffdbe0
r8             0x1999999999999999  1844674407370955161
r9             0x0                 0
r10            0x7ffff7d20ac0      140737351125696
r11            0x7ffff7d213c0      140737351128000
r12            0x4                 4
r13            0x0                 0
r14            0x4                 4
r15            0x0                 0
rip            0x401c1e            0x401c1e <build_target+392>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) next
```

Secret_phase:

在 main 函数中，我们看到这一段：在 je 这里它跳过了 secret

```
401463:   48 8d 45 d0             lea    -0x30(%rbp),%rax
401467:   48 89 c7                mov    %rax,%rdi
40146a:   e8 a7 fe ff ff          call   401316 <read_line>
40146f:   48 8d 45 d0             lea    -0x30(%rbp),%rax
401473:   48 89 c7                mov    %rax,%rdi
401476:   e8 fb 07 00 00          call   401c76 <phase_6>
40147b:   48 8d 3d 66 1c 00 00    lea    0x1c66(%rip),%rdi        # 4030e8 <_IO_
401482:   e8 79 fd ff ff          call   401200 <puts@plt>
401487:   48 83 7d f8 00          cmpq   $0x0,-0x8(%rbp)
40148c:   74 4d                   je     4014db <main+0x14c>
40148e:   48 8d 3d 8b 1c 00 00    lea    0x1c8b(%rip),%rdi        # 403120 <_IO_
401495:   e8 40 0c 00 00          call   4020da <slow_put>
40149a:   48 8d 3d a6 1c 00 00    lea    0x1ca6(%rip),%rdi        # 403147 <_IO_
4014a1:   e8 34 0c 00 00          call   4020da <slow_put>
4014a6:   48 8d 3d a7 1c 00 00    lea    0x1ca7(%rip),%rdi        # 403154 <_IO_
4014ad:   e8 8e 0c 00 00          call   402140 <slow_slow_put>
4014b2:   48 8d 45 d0             lea    -0x30(%rbp),%rax
4014b6:   48 89 c7                mov    %rax,%rdi
4014b9:   e8 58 fe ff ff          call   401316 <read_line>
4014be:   48 8d 45 d0             lea    -0x30(%rbp),%rax
4014c2:   48 89 c7                mov    %rax,%rdi
4014c5:   e8 eb 08 00 00          call   401db5 <secret_phase>
4014ca:   48 8d 3d 9f 1c 00 00    lea    0x1c9f(%rip),%rdi        # 403170 <_IO_
4014d1:   e8 2a fd ff ff          call   401200 <puts@plt>
4014d6:   e8 44 0d 00 00          call   40221f <true_ending>
4014db:   b8 00 00 00 00          mov    $0x0,%eax
4014e0:   c9                      leave
4014e1:   c3                      ret
```

也就是说，想要进入这里，得让 rbp-8 的位置存储的数不等于 0 即可。

“你或许能从 main.cpp 中发现某些违和之处……”

```
void read_line(char* input)
{
    char ch;
    int i;
    for (i = 0; i <= 40; i++) {
        ch = getchar();
        if (ch == '\n') {
            input[i] = '\0';
            return;
        } else {
            input[i] = ch;
        }
    }
    input[i] = '\0';

    while (getchar() != '\n'); // clear the input buffer
}
```

在 read_line 中，字符结束后只要不是换行都还能继续读入，而且注意到在 bomb 中只要不是有新的%s%d%c 啥的也不会被切割。然后，在最后做完六题后我打了个换行，粘贴到 run 里面就炸了，于是就可以想到，读入的字符，结尾某些元素是不能加的某些元素是可以加的。新元素不能加,但空格不会被 sscanf 切割,于是在答案后打空格,果然没炸,进入了 secret。

```
(gdb) x/s 0x403319
0x403319:       "%ud"
```

要求输入一个无符号整数。

```
401ded:   b8 00 00 00 00          mov    $0x0,%eax
401df2:   e8 69 f3 ff ff          call   401160 <__isoc99_sscanf@plt>
401df7:   83 f8 01                cmp    $0x1,%eax
401dfa:   0f 95 c0                setne  %al
401dfd:   84 c0                   test   %al,%al
401dff:   74 05                   je     401e06 <secret_phase+0x51>
401e01:   e8 42 02 00 00          call   402048 <explode_bomb>
401e06:   8b 45 f8                mov    -0x8(%rbp),%eax
401e09:   33 45 fc                xor    -0x4(%rbp),%eax
401e0c:   89 45 f8                mov    %eax,-0x8(%rbp)
401e0f:   8b 45 f8                mov    -0x8(%rbp),%eax
401e12:   3d 0d f0 ad ba          cmp    $0xbaadf00d,%eax
401e17:   74 05                   je     401e1e <secret_phase+0x69>
401e19:   e8 2a 02 00 00          call   402048 <explode_bomb>
401e1e:   90                      nop
401e1f:   c9                      leave
```

后半部分：cmp 那里 eax 必须为 1，al 才为 0，je 跳过爆炸，也就是要输入一个东西，对应前面，就是要输入一个无符号整数。先随便输个数去 gdb 调试：

Eax 就是我随便输入的数字 1 在 rbp-8 的位置，和 rbp-4 位置的元素异或后值为

```
(gdb) ni
0x0000000000401e0c in secret_phase ()
(gdb) i r
rax            0xdeadc0df          3735929055
rbx            0x0                 0
rcx            0x18                24
rdx            0x0                 0
rsi            0x7ffff7d7c580      140737351501184
rdi            0x7fffffffd660      140737488344672
rbp            0x7fffffffdcd0      0x7fffffffdcd0
rsp            0x7fffffffdcb0      0x7fffffffdcb0
r8             0x1                 1
r9             0x0                 0
r10            0x7ffff7d20ac0      140737351125696
r11            0x0                 0
```

也就是，rbp-8 位置的数是：0xdeadcde。输入与这个数异或后的值得是这

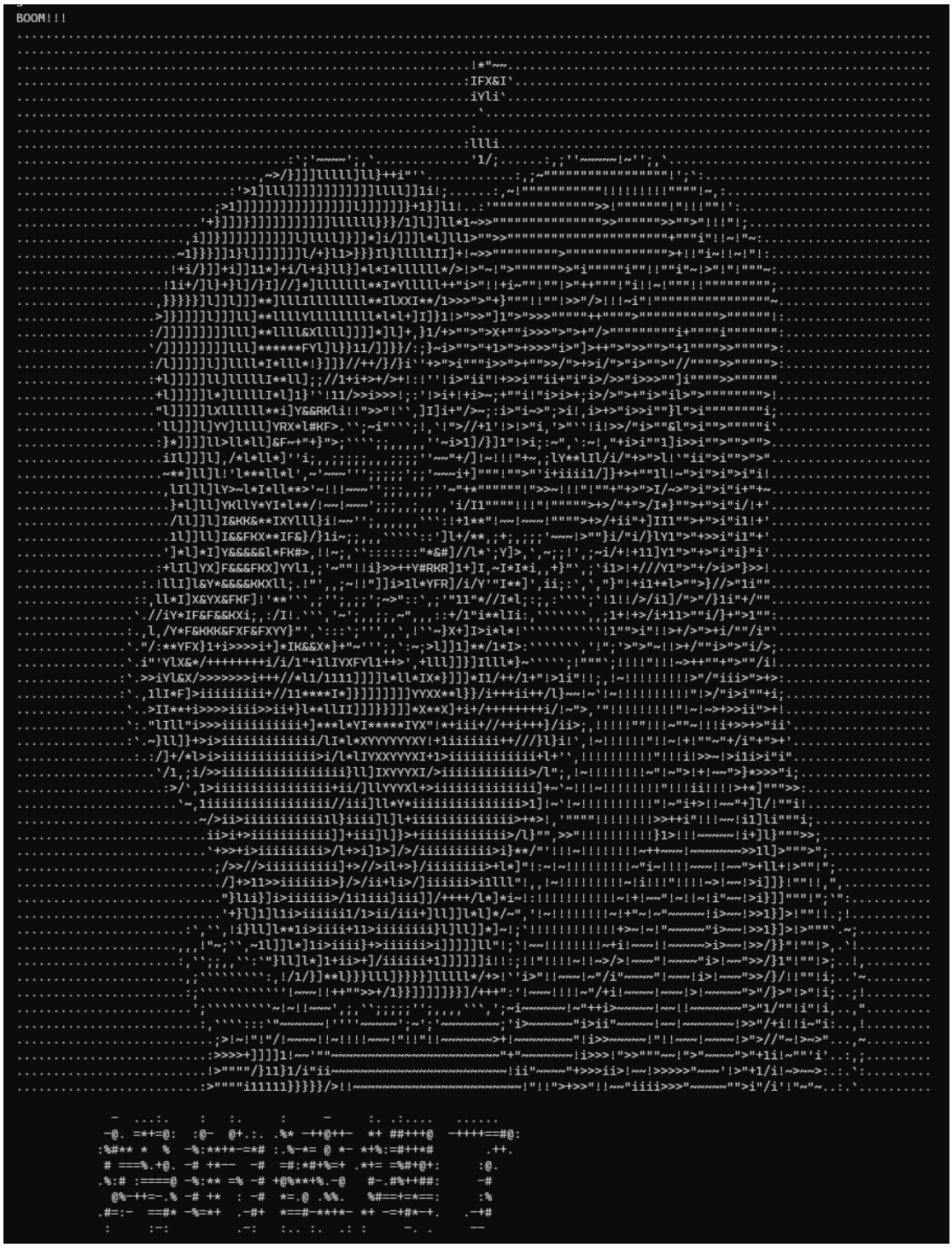
```
1107    401e0c:   89 45 f8                mov    %eax,-0x8(%rbp)
1108    401e0f:   8b 45 f8                mov    -0x8(%rbp),%eax
1109    401e12:   3d 0d f0 ad ba          cmp    $0xbaadf00d,%eax
1110    401e17:   74 05                   je     401e1e <secret_phase+0x69>
1111    401e19:   e8 2a 02 00 00          call   402048 <explode_bomb>
1112    401e1e:   90                      nop
```

x—deadc0de

y—baadf00d

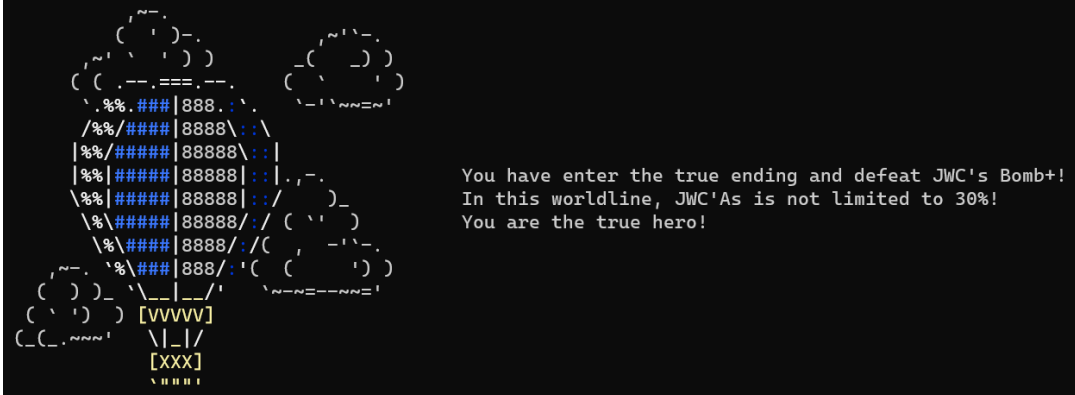
input—640030d3

转为十进制，即 1677734099。完成!!!!!!!!!!!!!



可爱捏波喜多

```
[Inferior 1 (process 694) exited normally]
(gdb) run
Starting program: /mnt/d/another_C/bomblab-handout/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
You have 6 phases with which to blow yourself up. Have a nice day!
Computer science is not a boring subject
Phase 1 defused. How about the next one?
-7 29 -115 461 -1843 7373
That's number 2. Keep going!
610 987 w
Halfway there!
51539607553
So you got that one. Try this one.
冲冲冲~ 2023
Good work! On to the next...
1 6 4 3 5 2
Cool! your skill on Reverse Engineering is great.
Welcome to the secret phase of Bomb+!
It is called... THE STEINS;GATE
1677734099
test
You are really a Master of Reverse Engineering!
```




```
[Inferior 1 (process 695) exited normally]
(gdb)
```