

计算机系统

LAB3实验报告

班级：人工智能2103

学号：202107030125

姓名：姚丁钰

实验目的

实验准备

实验过程

phase_1

phase_2

phase_3

phase_4

phase_5

phase_6

secret_phase

实验结果

实验总结

实验目的

程序运行在linux环境中。程序运行中有6个关卡（6个phase），每个phase需要用户在终端上输入特定的字符或者数字才能通关，否则将会引爆炸弹。需要通过分析汇编代码，使用gdb调试等方式找到正确的字符。

实验准备

首先拿到代码我们先看 `README` 文件，好吧，什么都没有，我们继续看其他文件。

发现 `bomb.c` 文件，但是没有头文件，所以不能进行运行和编译。但可以看出该程序要求从命令行或者文件以“行”为单位读入字符串，每行字符串对应一个phase的输入。如果phase执行完毕，会调用`phase_defused` 函数表明该 phase 成功搞定。

最后剩一个可执行文件，我们通过gdb调试，反汇编bomb文件，尝试得到该可执行文件的汇编代码。具体操作如下：`objdump -d bomb > bomb.asm` 这样就把反汇编得到的汇编代码存在一个文件 `bomb.asm` 中，便于查看。

然后通过打 `breakpoint` 的方法，以及查看寄存器和内存里存的值的情况，结合汇编语句，推算出应该输入的语句。

实验过程

phase_1

- phase_1的汇编代码

```
1 08048b60 <phase_1>:
2 8048b60: 83 ec 1c          sub     $0x1c,%esp#将0x1c字
   节的空间分配给堆栈
3 8048b63: c7 44 24 04 dc a1 04 movl    $0x804a1dc,0x4(%esp)#将0x804a1dc存储到%esp+4的地址中
4 8048b6a: 08
5 8048b6b: 8b 44 24 20      mov     0x20(%esp),%eax#
   将%esp+20的值存储到寄存器%eax中
6 8048b6f: 89 04 24         mov     %eax, (%esp)#将%eax
   中的值存储到%esp的地址中
7 8048b72: e8 9d 04 00 00   call    8049014
   <strings_not_equal>#调用strings_not_equal函数
8 8048b77: 85 c0            test    %eax,%eax#测试%eax
   中的值是否为零
9 8048b79: 74 05           je      8048b80
   <phase_1+0x20>#如果%eax中的值为零，则跳转到phase_1+0x20标签处
10 8048b7b: e8 a6 05 00 00   call    8049126
   <explode_bomb>#调用explode_bomb函数
11 8048b80: 83 c4 1c          add     $0x1c,%esp#回收堆栈
   空间
12 8048b83: c3              ret     #返回
```

1. 由于我们不能执行 `explode_bomb` 函数，所以需要在该函数之前执行跳转指令跳过 `explode_bomb` 函数
2. `%eax` 作为上一个函数的返回值，若 `%eax` 为0，才可以执行跳转。继续往上推发现了函数 `strings_not_equal`，通过阅读代码可以发现这个函数是判断输入

的两个字符串是否相等，该函数具有两个参数——一个是我们输入的字符串的首地址，另一个是待比较的字符串的首地址。

3. 输入字符串首地址的保存

返回main函数保存的文件中，查看调用 `phase1` 函数之前的几句代码。栈顶地址 `(%esp)` 被 `<read_line>` 函数的返回值赋值。因此，猜测 `<read_line>` 函数用于读取一行字符串，将返回值保存于 `%eax`，将 `eax` 寄存器中的值转移到 `esp` 寄存器保存的地址当中，这个值应该为输入字符串的地址

```
1  8048aa6:  e8 a2 06 00 00      call    804914d
    <read_line>#调用函数 read_line, 函数地址为 804914d
2  8048aab:  89 04 24            mov     %eax, (%esp) #将函数
    read_line 返回值（在 eax 寄存器中）存储到栈顶地址（%esp）中
3  8048aae:  e8 ad 00 00 00      call    8048b60 <phase_1>#调
    用函数 phase_1, 函数地址为 8048b60
```

4. 在调用 `<string_no_equal>` 函数之前的地址也存储了一个待比较的字符串首地址。

```
1  movl    $0x804a1dc, 0x4(%esp)
```

5. 先进入bomb

```
1  gdb bomb
```

6. 查看该地址内容

```
(gdb) x/s 0x804a1dc
0x804a1dc:  "When a problem comes along, you must zip it!"
```

7. 因此， `phase_1` 函数的作用只是单纯的让我们输入一个字符串。再将我们输入的字符串和存储的字符串进行比较而已

```
(gdb) run
Starting program: /home/cs18/Desktop/csapp/lab3/bomb202107030125/bomb103/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
When a problem comes along, you must zip it!
Phase 1 defused. How about the next one?
```

- 伪代码

```

1 void phase_1(char* output)
2 {
3     if( string_not_equal(output, "xxxxx") == 0)
4         explode_bomb();
5     return;
6 }

```

phase_2

- phase_2的汇编代码

```

1 08048b84 <phase_2>:
2 8048b84: 56                                push    %esi#将%esi的值压入
   栈中
3 8048b85: 53                                push    %ebx#将%ebx的值压入
   栈中
4 8048b86: 83 ec 34                          sub     $0x34,%esp#将%esp减
   去52(0x34)个字节,为局部变量腾出空间
5 8048b89: 8d 44 24 18                       lea     0x18(%esp),%eax#计
   算地址0x18(%esp)的值,将结果存储到%eax寄存器中
6 8048b8d: 89 44 24 04                       mov     %eax,0x4(%esp)#
   将%eax寄存器的值存储到地址0x4(%esp)处
7 8048b91: 8b 44 24 40                       mov     0x40(%esp),%eax#将
   地址0x40(%esp)处的值存储到%eax寄存器中
8 8048b95: 89 04 24                          mov     %eax,(%esp)#将%eax
   寄存器的值存储到%esp寄存器的值所指向的地址处
9 8048b98: e8 be 06 00 00                   call    804925b
   <read_six_numbers>#调用read_six_numbers函数
10 8048b9d: 83 7c 24 18 01                   cmpl    $0x1,0x18(%esp)#比
   较地址0x18(%esp)处的值和整数1
11 8048ba2: 74 05                            je      8048ba9#如果相等,跳
   转到地址8048ba9处 <phase_2+0x25>
12 8048ba4: e8 7d 05 00 00                   call    8049126
   <explode_bomb>#果不相等,调用explode_bomb函数
13 8048ba9: 8d 5c 24 1c                       lea     0x1c(%esp),%ebx#计
   算地址0x1c(%esp)的值,将结果存储到%ebx寄存器中
14 8048bad: 8d 74 24 30                       lea     0x30(%esp),%esi#计
   算地址0x30(%esp)的值,将结果存储到%esi寄存器中
15 8048bb1: 8b 43 fc                          mov     -0x4(%ebx),%eax#将
   地址%ebx-4处的值存储到%eax寄存器中
16 8048bb4: 01 c0                            add     %eax,%eax#将%eax的
   值加倍
17 8048bb6: 39 03                            cmp     %eax,(%ebx)#将%eax
   中的值与%ebx寄存器中的值进行比较

```

```

18  8048bb8:  74 05                      je      8048bbf
    <phase_2+0x3b>#如果上一条指令的比较结果为相等，则跳转到地址8048bbf处。
19  8048bba:  e8 67 05 00 00            call    8049126
    <explode_bomb>#如果上一条指令的比较结果不相等，则调用函数
    explode_bomb，炸弹爆炸
20  8048bbf:  83 c3 04                  add     $0x4,%ebx#将%ebx寄存器+0x4
21  8048bc2:  39 f3                      cmp     %esi,%ebx#将%ebx寄存器与%esi寄存器中的值进行比较
22  8048bc4:  75 eb                      jne     8048bb1
    <phase_2+0x2d>#如果两个寄存器中的值不相等，则跳转到地址 8048bb1 处执行
23  8048bc6:  83 c4 34                  add     $0x34,%esp#这条指令将栈指针 %esp 的值加上 0x34，释放掉一些栈空间，回收函数中用于局部变量和参数的内存空间
24  8048bc9:  5b                          pop     %ebx
25  8048bca:  5e                          pop     %esi#这两条指令分别将%ebx 和 %esi 寄存器中的值出栈，回收函数中保存的寄存器状态
26  8048bcb:  c3                          ret     #将程序控制流返回到调用该函数的地址处

```

1. phase_2中调用了 `read_six_number`，提示答案应该为6个数字
2. 首先将`esp+24`所指向的位置保存的数据与1进行比较，不相同则炸弹爆炸

```

1  8048b86:  83 ec 34                  sub     $0x34,%esp#将%esp减去52（0x34）个字节，为局部变量腾出空间
2  8048b89:  8d 44 24 18              lea     0x18(%esp),%eax#%eax=array[0]
3  8048b8d:  89 44 24 04              mov     %eax,0x4(%esp)#
4  8048b91:  8b 44 24 40              mov     0x40(%esp),%eax#%eax=
5  8048b95:  89 04 24                  mov     %eax,(%esp)#将%eax寄存器的值存储到%esp寄存器的值所指向的地址处
6  8048b98:  e8 be 06 00 00          call    804925b
    <read_six_numbers>#调用read_six_numbers函数
7  8048b9d:  83 7c 24 18 01          cmpl    $0x1,0x18(%esp)#if (array[0] != 1)

```

```

1  call    8049126 <explode_bomb>#果不相等，调用explode_bomb函数

```

3. 相同将 `esp+28`赋值给 `ebx`，`esp+48`赋值给 `esi`

```

1  lea     0x1c(%esp),%ebx#%ebx=0x1c+%esp初始化 init
2  lea     0x30(%esp),%esi#数组结束

```

4. 接下来将 `ebx - 4` 所指向位置的数乘2与 `ebx` 比较

```
1  mov    -0x4(%ebx),%eax#将地址%ebx-4处的值存储到%eax寄存器中  
    array[i-1]  
2  add    %eax,%eax#将%eax的值加倍 array[i-1] += array[i-1]  
3  cmp    %eax,(%ebx)#将%eax中的值与%ebx寄存器中的值进行比较 array[i]  
    != array[i-1]
```

5. 比较成功后将 `ebx + 4`, 并将 `ebx` 与 `esi` 比较, 作为一个循环

```
1  add    $0x4,%ebx#ebx += 4  
2  cmp    %esi,%ebx#ebx != esi判断循环结束  
3  jne    8048bb1 <phase_2+0x2d>#如果两个寄存器中的值不相等, 则跳转到地址  
    8048bb1 处执行
```

6. 开始比较时 `ebx` 与 `esi` 相差20, 5次循环刚好判断后5个数是否正确, 而每个数正确与否又是与上一个数乘2进行比较。

7. 根据这部分逻辑可以得出, 这段代码先判断数字第一位是否是1, 接下来判断每一位是否是上一位的2倍, 即答案的数字应该为1 2 4 8 16 32, 输入这6个数字, `phase_2` 通过

```
1 2 4 8 16 32  
That's number 2.  Keep going!
```

过程

```

369 08048b84 <phase_2>:
370 8048b84: 56          push %esi
371 8048b85: 53          push %ebx
372 8048b86: 83 ec 34    sub $0x34,%esp
373 8048b89: 8d 44 24 18  lea 0x18(%esp),%eax
374 8048b8d: 89 44 24 04  mov %eax,0x4(%esp)
375 8048b91: 8b 44 24 40  mov 0x40(%esp),%eax
376 8048b95: 89 04 24    mov %eax,(%esp)
377 8048b98: e8 be 06 00 00 call 804925b <read_six_numbers>
378 8048b9d: 83 7c 24 18 01 cmpl $0x1,0x18(%esp)
379 8048ba2: 74 05       je 8048ba9 <phase_2+0x25>
380 8048ba4: e8 7d 05 00 00 call 8049126 <explode_bomb>
381 8048ba9: 8d 5c 24 1c  lea 0x1c(%esp),%ebx
382 8048bad: 8d 74 24 30  lea 0x30(%esp),%esi
383 8048bb1: 8b 43 fc    mov -0x4(%ebx),%eax
384 8048bb4: 01 c0      add %eax,%eax
385 8048bb6: 39 03      cmp %eax,(%ebx)
386 8048bb8: 74 05       je 8048bbf <phase_2+0x3b>
387 8048bba: e8 67 05 00 00 call 8049126 <explode_bomb>
388 8048bbf: 83 c3 04    add $0x4,%ebx
389 8048bc2: 39 f3      cmp %esi,%ebx
390 8048bc4: 75 eb      jne 8048bb1 <phase_2+0x2d>
391 8048bc6: 83 c4 34    add $0x34,%esp
392 8048bc9: 5b         pop %ebx
393 8048bca: 5e         pop %esi
394 8048bcb: c3         ret

```

对应的c语言代码

```

1 void phase_2(char* input) {
2     int nums[6]; // 在栈上分配了0x34 = 52字节的空间用于存储六个整数，
    // nums[0]在0x18(%esp)处，nums[5]在0x30(%esp)处
3     read_six_numbers(input, nums); // 调用read_six_numbers函数
    // 读入六个数字到nums数组中
4
5     // 检查nums[0]是否等于1，如果不是，则引爆炸弹
6     if (nums[0] != 1) {
7         explode_bomb();
8     }
9
10    int* ebx = &nums[1]; // ebx指向nums[1]
11    int* esi = &nums[0]; // esi指向nums[0]
12
13    // 循环6次，每次检查当前数字是否是前一个数字的两倍，如果不是，则引爆炸弹
14    for (int i = 0; i < 6; i++) {
15        if (*ebx != *esi * 2) {
16            explode_bomb();
17        }
18        ebx++; // ebx指向下一个数字
19        esi++; // esi指向下一个数字

```



```
20     }
21 }
```

phase_3

- phase_3的汇编代码

```
1  08048bcc <phase_3>:
2      8048bcc:  83 ec 2c          sub     $0x2c,%esp#给栈上的
      空间分配 44 字节
3      8048bcf:  8d 44 24 1c       lea     0x1c(%esp),%eax#将
      0x1c(%esp) 的地址存储在 eax 中
4      8048bd3:  89 44 24 0c       mov     %eax,0xc(%esp)#将
      eax 中的值存储到 0xc(%esp) 中
5      8048bd7:  8d 44 24 18       lea     0x18(%esp),%eax#将
      0x18(%esp) 的地址存储在 eax 中
6      8048bdb:  89 44 24 08       mov     %eax,0x8(%esp)#将
      eax 中的值存储到 0x8(%esp) 中
7      8048bdf:  c7 44 24 04 cf a3 04  movl    $0x804a3cf,0x4(%esp)#将 0x804a3cf 的值存储到 0x4(%esp) 中
8      8048be6:  08
9      8048be7:  8b 44 24 30       mov     0x30(%esp),%eax#将
      0x30(%esp) 的值存储到 eax 中
10     8048beb:  89 04 24          mov     %eax,(%esp)#将 eax
      中的值存储到 (%esp) 中,即存储在栈顶
11     8048bee:  e8 7d fc ff ff    call    8048870
      <__isoc99_sscanf@plt>#调用 sscanf 函数
12     8048bf3:  83 f8 01          cmp     $0x1,%eax#比较 eax
      中的值与 1
13     8048bf6:  7f 05            jg      8048bfd
      <phase_3+0x31>#如果 eax > 1,则跳转到 0x8048bfd 处
14     8048bf8:  e8 29 05 00 00    call    8049126
      <explode_bomb>#如果 eax <= 1,则调用 explode_bomb 函数
15     8048bfd:  83 7c 24 18 07    cmpl    $0x7,0x18(%esp)#比
      较 0x18(%esp) 的值与 7
16     8048c02:  77 64            ja      8048c68
      <phase_3+0x9c>#如果 0x18(%esp) 的值 > 7,则跳转到 0x8048c68 处
17     8048c04:  8b 44 24 18       mov     0x18(%esp),%eax#将
      0x18(%esp) 的值存入寄存器 eax
18     8048c08:  ff 24 85 3c a2 04 08  jmp     *0x804a23c(,%eax,4)# 跳转到 (%eax4+0x804a23c) 处
19     8048c0f:  b8 00 00 00 00    mov     $0x0,%eax#将 0 存入
      寄存器 eax
20     8048c14:  eb 05            jmp     8048c1b
      <phase_3+0x4f>#跳转到 0x8048c1b 处
```


21	8048c16:	b8 3a 02 00 00	mov	\$0x23a,%eax#将 0x23a 存入寄存器 eax
22	8048c1b:	2d 20 03 00 00	sub	\$0x320,%eax#将 0x320 减去寄存器 eax 的值, 并将结果存入 eax
23	8048c20:	eb 05	jmp	8048c27 <phase_3+0x5b>#跳转到 0x8048c27 处
24	8048c22:	b8 00 00 00 00	mov	\$0x0,%eax#将 0 存入 寄存器 eax
25	8048c27:	83 c0 66	add	\$0x66,%eax#将寄存器 eax 的值加上 0x66
26	8048c2a:	eb 05	jmp	8048c31 <phase_3+0x65>#跳转到 0x8048c31 处
27	8048c2c:	b8 00 00 00 00	mov	\$0x0,%eax#将 0 存入 寄存器 eax
28	8048c31:	2d 78 02 00 00	sub	\$0x278,%eax#将 0x278 减去寄存器 eax 的值, 并将结果存入 eax
29	8048c36:	eb 05	jmp	8048c3d <phase_3+0x71># 跳转到 0x8048c3d 处
30	8048c38:	b8 00 00 00 00	mov	\$0x0,%eax#将 0 存入 寄存器 eax
31	8048c3d:	05 78 02 00 00	add	\$0x278,%eax#将 0x278 加到 eax 中
32	8048c42:	eb 05	jmp	8048c49 <phase_3+0x7d>#跳转到 0x8048c49 处
33	8048c44:	b8 00 00 00 00	mov	\$0x0,%eax#将 0 存入 寄存器 eax
34	8048c49:	2d 78 02 00 00	sub	\$0x278,%eax#将 0x278 减去 eax 中的值
35	8048c4e:	eb 05	jmp	8048c55 <phase_3+0x89>#跳转到 0x8048c55 处
36	8048c50:	b8 00 00 00 00	mov	\$0x0,%eax# 将 0 存 入寄存器 eax
37	8048c55:	05 78 02 00 00	add	\$0x278,%eax#将 0x278 加到 eax 中
38	8048c5a:	eb 05	jmp	8048c61 <phase_3+0x95>#跳转到 0x8048c61 处
39	8048c5c:	b8 00 00 00 00	mov	\$0x0,%eax#将 0 存入 寄存器 eax
40	8048c61:	2d 78 02 00 00	sub	\$0x278,%eax#将 0x278 减去 eax 中的值
41	8048c66:	eb 0a	jmp	8048c72 <phase_3+0xa6># 跳转到 0x8048c72 处
42	8048c68:	e8 b9 04 00 00	call	8049126 <explode_bomb>#调用函数 explode_bomb

```

43  8048c6d:  b8 00 00 00 00      mov     $0x0,%eax#将 0 存入
    寄存器 eax
44  8048c72:  83 7c 24 18 05      cmpl    $0x5,0x18(%esp)# 将
    0x5 与栈中偏移量为 0x18 的值进行比较
45  8048c77:  7f 06              jg      8048c7f
    <phase_3+0xb3>#如果大于则跳转到 0x8048c7f 处
46  8048c79:  3b 44 24 1c      cmp     0x1c(%esp),%eax#将
    栈顶偏移 0x1c 处的值与 eax 比较
47  8048c7d:  74 05              je      8048c84
    <phase_3+0xb8>#如果相等则跳转到 0x8048c84 处
48  8048c7f:  e8 a2 04 00 00      call    8049126
    <explode_bomb>#如果不相等则调用函数 0x8049126 <explode_bomb>
49  8048c84:  83 c4 2c      add     $0x2c,%esp#将栈指针
    增加 0x2c
50  8048c87:  c3              ret     #返回到调用者处

```

1. 分析phase_3的汇编代码，phase_3中是通过调用 `sscanf()` 读取格式化字符串的。注意到在调用 `sscanf()` 前有一条操作将立即数0x804a3cf指向的数据保存到esp+4的位置，gdb调试将断点设置在phase_3，查看该立即数指向的数据，得到一个字符串，推断出该字符串是 `sscanf()` 的参数，由此可知phase_3的答案为两个整数。

```
1 | call    8048870 <__isoc99_sscanf@plt>#调用 sscanf 函数
```

```
1 | movl    $0x804a3cf,0x4(%esp)#将 0x804a3cf 的值存储到 0x4(%esp) 中
```

```
(gdb) x/s 0x804a3cf
0x804a3cf:      "%d %d"
```

2. 调用 `sscanf()` 后将eax寄存器保存的值与1比较，小于等于1则爆炸。eax中保存的值为`sscanf`返回的成功匹配的个数，说明输入的数个数小于等于1将会直接爆炸

```

1 | cmp     $0x1,%eax#比较 eax 中的值与 1
2 | jg      8048bfd <phase_3+0x31>#如果 eax > 1, 则跳转到 0x8048bfd 处
3 | call    8049126 <explode_bomb>#如果 eax <= 1, 则调用 explode_bomb
    函数

```

3. 接下来将esp+24处的值与7进行比较，如果大于7将爆炸。可以判断esp+24处的值为第一个输入的数字，这个数字必须小于7（且大于等于0）。

```

1 | cmpl    $0x7,0x18(%esp)#比较 0x18(%esp) 的值与 7
2 | ja      8048c68 <phase_3+0x9c>#如果 0x18(%esp) 的值 > 7, 则跳转到
   | 0x8048c68 处

```

4. 这是典型的 switch 跳转语句，即跳转到以地址 `*0x804a23c` 为基址的跳转表中。输入 `p/x *0x804a23c`，得到地址 `0x8048c16`，在代码中找到该处指令

```

1 | jmp      *0x804a23c(,%eax,4)# 跳转到 (%eax*4+0x804a23c) 处

```

```

(gdb) p/x *0x804a23c
$1 = 0x8048c16

```

5. 个数字为0进行跳转，分析汇编代码进行的操作。主要是将第一个数字进行一些加减操作，再与第二个数比较，相等即可通过。计算后得到当第一个数为0时，第二个数应该为-760。

8048c16:	b8 3a 02 00 00	mov	<code>\$0x23a,%eax</code>	#0x23a赋值给eax
8048c1b:	2d 20 03 00 00	sub	<code>\$0x320,%eax</code>	#eax-0x320
8048c20:	eb 05	jmp	<code>8048c27 <phase_3+0x5b></code>	
8048c22:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c27:	83 c0 66	add	<code>\$0x66,%eax</code>	#eax+0x66
8048c2a:	eb 05	jmp	<code>8048c31 <phase_3+0x65></code>	
8048c2c:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c31:	2d 78 02 00 00	sub	<code>\$0x278,%eax</code>	#eax-0x278
8048c36:	eb 05	jmp	<code>8048c3d <phase_3+0x71></code>	
8048c38:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c3d:	05 78 02 00 00	add	<code>\$0x278,%eax</code>	#eax+0x278
8048c42:	eb 05	jmp	<code>8048c49 <phase_3+0x7d></code>	
8048c44:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c49:	2d 78 02 00 00	sub	<code>\$0x278,%eax</code>	#eax-0x278
8048c4e:	eb 05	jmp	<code>8048c55 <phase_3+0x89></code>	
8048c50:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c55:	05 78 02 00 00	add	<code>\$0x278,%eax</code>	#eax+0x278
8048c5a:	eb 05	jmp	<code>8048c61 <phase_3+0x95></code>	
8048c5c:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c61:	2d 78 02 00 00	sub	<code>\$0x278,%eax</code>	#eax-0x278
8048c66:	eb 0a	jmp	<code>8048c72 <phase_3+0xa6></code>	
8048c68:	e8 b9 04 00 00	call	<code>8049126 <explode_bomb></code>	
8048c6d:	b8 00 00 00 00	mov	<code>\$0x0,%eax</code>	
8048c72:	83 7c 24 18 05	cmpl	<code>\$0x5,0x18(%esp)</code>	#将原值与5比较
8048c77:	7f 06	jg	<code>8048c7f <phase_3+0xb3></code>	#跳转到explode_bomb
8048c79:	3b 44 24 1c	cmp	<code>0x1c(%esp),%eax</code>	#eax的值与esp+28处比较
8048c7d:	74 05	je	<code>8048c84 <phase_3+0xb8></code>	
8048c7f:	e8 a2 04 00 00	call	<code>8049126 <explode_bomb></code>	#不相等调用explode_bomb

```

0 -760
Halfway there!

```

6. 注意到在执行完对第一个数的加减操作后将原来的第一个数与5比较，若大于5则爆炸，故第一个数的实际范围是0-5。

```
(gdb) x/16a 0x804a23c
0x804a23c: 0x8048c16 <phase_3+74> 0x8048c0f <phase_3+67> 0x8048c22 <phase_3+86> 0x8048c2c <phase_3+96>
0x804a24c: 0x8048c38 <phase_3+108> 0x8048c44 <phase_3+120> 0x8048c50 <phase_3+132> 0x8048c5c <phase_3+144>
0x804a25c <array.2999>: 0x7564616d 0x73726569 0x746f666e 0x6c796276
0x804a26c: 0x79206f53 0x7420756f 0x6b6e6968 0x756f7920
```

16进制	10进制
0x23a	570
0x320	800
0x66	102
0x278	632

第一个数字	对应的case语句下取出的值
0	$570-800+102-632=-760$
1	$-800+102-632=-1330$
2	$102-632=-530$
3	-632
4	0
5	-632

分析跳转表中的其他情况，当第一个数字为1,2,3,4,5时，对应的第二个数字分别为-1330，-530，-632，0，-632，经测试这些答案同样可通过phase_3。

```
1 -1330
Halfway there!
```

```
2 -530
Halfway there!
```

```
3 -632
Halfway there!
```

4 0
Halfway there!

5 -632
Halfway there!

phase_4

- phase_4的汇编代码

```
1 08048cf5 <phase_4>:
2 8048cf5: 83 ec 2c          sub    $0x2c,%esp#开辟堆栈
   空间，大小为0x2c（44个字节）
3 8048cf8: 8d 44 24 1c      lea    0x1c(%esp),%eax#将
   堆栈顶端地址+0x1c的地址赋值给eax
4 8048cfc: 89 44 24 0c      mov    %eax,0xc(%esp)#将
   eax中的地址存入堆栈顶端地址+0xc的位置
5 8048d00: 8d 44 24 18      lea    0x18(%esp),%eax#将
   堆栈顶端地址+0x18的地址赋值给eax
6 8048d04: 89 44 24 08      mov    %eax,0x8(%esp)#将
   eax中的地址存入堆栈顶端地址+0x8的位置
7 8048d08: c7 44 24 04 cf a3 04 movl   $0x804a3cf,0x4(%esp)#将立即数 0x804a3cf 移动到偏移地址4(%esp)处
8 8048d0f: 08
9 8048d10: 8b 44 24 30      mov    0x30(%esp),%eax#将
   堆栈顶端地址+0x30的地址中存储的字符串转换为整数并存储在eax中
10 8048d14: 89 04 24         mov    %eax,(%esp)#将eax中
   的值存入堆栈顶端地址的位置中，作为第一个参数
11 8048d17: e8 54 fb ff ff   call   8048870
   <__isoc99_sscanf@plt>#调用scanf函数，将读入的整数存入eax，并返回读入
   的参数数量
12 8048d1c: 83 f8 02         cmp    $0x2,%eax#比较eax寄
   存器中的值是否为2
13 8048d1f: 75 0d           jne    8048d2e
   <phase_4+0x39>#如果不等于2，则跳转到0x8048d2e处（若数字不为两个 则引爆
   bomb）
14 8048d21: 8b 44 24 18      mov    0x18(%esp),%eax#将
   堆栈顶端地址+0x18中的值存入eax
15 8048d25: 85 c0           test   %eax,%eax# 将eax寄
   存器与自身进行AND操作
16 8048d27: 78 05           js     8048d2e
   <phase_4+0x39>#如果结果为负，则跳转到0x8048d2e处
```

```

17  8048d29:  83 f8 0e                cmp     $0xe,%eax#比较eax寄存器
    中的值是否小于或等于14
18  8048d2c:  7e 05                   jle     8048d33
    <phase_4+0x3e>#如果比较结果小于或等于0xe，则跳转到地址0x8048d33处，否
    则继续执行
19  8048d2e:  e8 f3 03 00 00          call    8049126
    <explode_bomb>#调用地址0x8049126处的函数 explode_bomb
20  8048d33:  c7 44 24 08 0e 00 00    movl    $0xe,0x8(%esp)#把常
    量0xe存储到栈中相对于esp偏移量为0x8的位置
21  8048d3a:  00
22  8048d3b:  c7 44 24 04 00 00 00    movl    $0x0,0x4(%esp)#把常
    量0存储到栈中相对于esp偏移量为0x4的位置
23  8048d42:  00
24  8048d43:  8b 44 24 18             mov     0x18(%esp),%eax#把
    栈中相对于esp偏移量为0x18的位置的值存储到eax寄存器中
25  8048d47:  89 04 24                mov     %eax,(%esp)#把eax寄
    存器中的值存储到栈顶
26  8048d4a:  e8 39 ff ff ff          call    8048c88 <func4>#调
    用地址0x8048c88处的函数func4
27  8048d4f:  83 f8 07                cmp     $0x7,%eax#将寄存器
    eax中的值与常量0x7进行比较
28  8048d52:  75 07                   jne     8048d5b
    <phase_4+0x66>#如果比较结果不等于0，则跳转到地址 0x8048d5b处，否则继续
    执行
29  8048d54:  83 7c 24 1c 07          cmpl    $0x7,0x1c(%esp)#将
    栈中相对于esp偏移量为0x1c的位置的值与常量0x7进行比较
30  8048d59:  74 05                   je      8048d60
    <phase_4+0x6b>
31  8048d5b:  e8 c6 03 00 00          call    8049126
    <explode_bomb>
32  8048d60:  83 c4 2c                add     $0x2c,%esp
33  8048d63:  c3                      ret

```

- func4的汇编代码

```

1  08048c88 <func4>:
2  8048c88:  83 ec 1c                sub     $0x1c,%esp#为局部变
    量分配空间
3  8048c8b:  89 5c 24 14             mov     %ebx,0x14(%esp)#将
    ebx 寄存器的值存入 esp-0x14 的内存地址，保留该寄存器
4  8048c8f:  89 74 24 18             mov     %esi,0x18(%esp)#将
    esi 寄存器的值存入 esp-0x18 的内存地址，保留该寄存器
5  8048c93:  8b 54 24 20             mov     0x20(%esp),%edx#将
    esp+0x20 的内存地址的值存入 edx 寄存器

```


6	8048c97:	8b 44 24 24	mov	0x24(%esp),%eax#将 esp+0x24 的内存地址的值存入 eax 寄存器
7	8048c9b:	8b 5c 24 28	mov	0x28(%esp),%ebx#将 esp+0x28 的内存地址的值存入 ebx 寄存器
8	8048c9f:	89 d9	mov	%ebx,%ecx#将 ebx 寄存器的值存入 ecx 寄存器
9	8048ca1:	29 c1	sub	%eax,%ecx#计算 ecx = ecx - eax
10	8048ca3:	89 ce	mov	%ecx,%esi#将 ecx 寄存器的值存入 esi 寄存器
11	8048ca5:	c1 ee 1f	shr	\$0x1f,%esi#计算 esi = esi >> 31, 即 esi 的符号位
12	8048ca8:	01 f1	add	%esi,%ecx#计算 ecx = ecx + esi, 相当于对 ecx 进行无符号的向下取整
13	8048caa:	d1 f9	sar	%ecx#计算 ecx = ecx / 2, 相当于对 ecx 进行有符号的向下取整
14	8048cac:	01 c1	add	%eax,%ecx#计算 ecx = ecx + eax
15	8048cae:	39 d1	cmp	%edx,%ecx#比较 edx 和 ecx 的值, 设置相应的条件码
16	8048cb0:	7e 17	jle	8048cc9 <func4+0x41>#如果 ecx <= edx, 则跳转到 8048cc9 处
17	8048cb2:	83 e9 01	sub	\$0x1,%ecx#计算 ecx = ecx - 1
18	8048cb5:	89 4c 24 08	mov	%ecx,0x8(%esp)#将 ecx 的值存入 esp+0x8 的内存地址, 作为第二个参数
19	8048cb9:	89 44 24 04	mov	%eax,0x4(%esp)#把 EAX的值存储到堆栈指针(ESP)的4字节偏移处
20	8048cbd:	89 14 24	mov	%edx,(%esp)#把EDX的 值存储到堆栈指针(ESP)的0字节偏移处
21	8048cc0:	e8 c3 ff ff ff	call	8048c88 <func4>#调用函数func4
22	8048cc5:	01 c0	add	%eax,%eax#将eax中的 值乘以2, 结果仍保存在eax中。
23	8048cc7:	eb 20	jmp	8048ce9 <func4+0x61>#跳转到func4+0x61地址
24	8048cc9:	b8 00 00 00 00	mov	\$0x0,%eax#把0存储到 EAX
25	8048cce:	39 d1	cmp	%edx,%ecx#比较EDX和 ECX的值
26	8048cd0:	7d 17	jge	8048ce9 <func4+0x61>#如果ECX大于等于EDX, 则跳转到func4+0x61地址
27	8048cd2:	89 5c 24 08	mov	%ebx,0x8(%esp)#把 EBX的值存储到堆栈指针(ESP)的8字节偏移处
28	8048cd6:	83 c1 01	add	\$0x1,%ecx#把ECX加1


```

29  8048cd9:  89 4c 24 04          mov    %ecx,0x4(%esp)#把
    ECX的值存储到堆栈指针(ESP)的4字节偏移处
30  8048cdd:  89 14 24             mov    %edx,(%esp)#把EDX的
    值存储到堆栈指针(ESP)的0字节偏移处
31  8048ce0:  e8 a3 ff ff ff      call   8048c88 <func4>#调
    用func4函数
32  8048ce5:  8d 44 00 01          lea    0x1(%eax,%eax,1),%eax#把EAX的值乘以2并加1
33  8048ce9:  8b 5c 24 14          mov    0x14(%esp),%ebx#把
    esp+20 的值存入 ebx
34  8048ced:  8b 74 24 18          mov    0x18(%esp),%esi#把
    esp+24 的值存入 esi
35  8048cf1:  83 c4 1c             add    $0x1c,%esp#回收栈上
    的空间
36  8048cf4:  c3                  ret

```

1. 发现phase_4的前面几行和phase_3没有区别，说明也是读入两个整数
2. x 为第一个参数，储存于 0x18(%esp) 和 0x8(%esp), y 为第二个参数，储存于 0x1c(%esp) 和 0xc(%esp) 中。

```

1  lea    0x1c(%esp),%eax#y
2  mov    %eax,0xc(%esp)#y
3  lea    0x18(%esp),%eax#x
4  mov    %eax,0x8(%esp)#x

```

3. 接下来便是准备参数，调用 func4，从汇编中可以看出，func4 的第一个参数为我们输入的第一个数，第二个参数为0x0，即十进制的0，第三个参数为0xe，即十进制的14

```

1  8048d33:  c7 44 24 08 0e 00 00  movl    $0xe,0x8(%esp)#参数3
2  8048d3a:  00
3  8048d3b:  c7 44 24 04 00 00 00  movl    $0x0,0x4(%esp)#参数2
4  8048d42:  00
5  8048d43:  8b 44 24 18          mov    0x18(%esp),%eax#参数
    1 (x)
6  8048d47:  89 04 24             mov    %eax,(%esp)

```

4. 再往下可以发现程序将 func4 的返回值与7进行了对比，且将输入的第二个数与7进行了对比，如果有一个不相等则引爆炸弹，所以输入的第二个数可以确定是7，输入的第一个数x要使 func4(x, 0, 14) 的返回值为7。

```

1  8048d4a:  e8 39 ff ff ff      call    8048c88 <func4>
2  8048d4f:  83 f8 07            cmp     $0x7,%eax#对比返回值
3  8048d52:  75 07              jne     8048d5b
    <phase_4+0x66>
4  8048d54:  83 7c 24 1c 07      cmpl    $0x7,0x1c(%esp)#对比
    第二个数
5  8048d59:  74 05              je      8048d60
    <phase_4+0x6b>

```

- 伪代码

```

1  void phase_4(char* input)
2  {
3      int x, y;
4      int ret = sscanf(input, "%d %d", &x, &y);
5      if (ret != 2 || x > 14) {
6          explode_bomb();
7      }
8      else {
9          int ret = func4(x, 0, 14);
10         if (ret != 0 || y != 0) {
11             explode_bomb();
12         }
13     }
14     return;
15 }

```

5. func4 的汇编代码:将三个参数分别记为x,y,z,程序首先对参数进行了一系列算术操作得到(z-y)

```

1  8048c93:  8b 54 24 20      mov     0x20(%esp),%edx#x
2  8048c97:  8b 44 24 24      mov     0x24(%esp),%eax#y
3  8048c9b:  8b 5c 24 28      mov     0x28(%esp),%ebx#z
4  8048c9f:  89 d9            mov     %ebx,%ecx#ecx:z
5  8048ca1:  29 c1            sub     %eax,%ecx#ecx:z-y
6  8048ca3:  89 ce            mov     %ecx,%esi#esi:z-y

```

6. 然后又将其右移了31位，这个**右移的目的是获得它的符号位**，然后将该符号位 sign 与(z-y)相加再右移一位，不难猜出这个右移的目的是做除法操作除以2(**右移向负无穷大取整，而除法是向0取整，加上符号位就可以对负数除法进行修正，让其向0取整。**说这其实就是一个除法操作，得到(z-y)/2，然后进行一个加法操作得到y+(z-y)/2。)

```

1  8048ca5:  c1 ee 1f          shr     $0x1f,%esi#计算 esi
    = esi >> 31, 即 esi 的符号位
2  8048ca8:  01 f1          add     %esi,%ecx#计算 ecx =
    ecx + esi, 相当于对 ecx 进行无符号的向下取整
3  8048caa:  d1 f9          sar     %ecx#计算 ecx = ecx
    / 2, 相当于对 ecx 进行有符号的向下取整
4  8048cac:  01 c1          add     %eax,%ecx#计算 ecx =
    ecx + eax

```

7. 再往下可以发现 func4 是一个递归函数当 $x < y + (z - y) / 2$ 时返回 $2 * \text{func}(x, y, y + (z - y) / 2 - 1)$

```

1  8048cae:  39 d1          cmp     %edx,%ecx#比较
    edx(x) 和 ecx(y+(z-y)/2) 的值, 设置相应的条件码
2  8048cb0:  7e 17          jle     8048cc9
    <func4+0x41>#如果 ecx <= edx, 则跳转到 8048cc9 处

```

```

1  8048cb2:  83 e9 01          sub     $0x1,%ecx#ecx = y+
    (z-y)/2 - 1
2  8048cb5:  89 4c 24 08      mov     %ecx,0x8(%esp)#第三个
    参数: y+(z-y)/2 - 1
3  8048cb9:  89 44 24 04      mov     %eax,0x4(%esp)#第二个
    参数: y
4  8048cbd:  89 14 24          mov     %edx,(%esp)#第一个参
    数: x
5  8048cc0:  e8 c3 ff ff ff   call    8048c88 <func4>#调用
    函数func4
6  8048cc5:  01 c0          add     %eax,%eax#返回值*2

```

8. 当 $x > y + (z - y) / 2$ 时返回 $2 * \text{func4}(x, y + (z - y) / 2 + 1, z) + 1$

```

1  8048cce:  39 d1          cmp     %edx,%ecx#比较x和y+
    (z-y)/2
2  8048cd0:  7d 17          jge     8048ce9
    <func4+0x61>#如果y+(z-y)/2>=x, 则跳转到func4+0x61地址

```

```

1  8048cd2:  89 5c 24 08          mov    %ebx,0x8(%esp)#第三个
    参数: z
2  8048cd6:  83 c1 01             add    $0x1,%ecx#y+(z-
    y)/2+1
3  8048cd9:  89 4c 24 04          mov    %ecx,0x4(%esp)#第二个
    参数: y+(z-y)/2+1
4  8048cdd:  89 14 24             mov    %edx,(%esp)#第一个参
    数: x
5  8048ce0:  e8 a3 ff ff ff      call   8048c88 <func4>#调用
    func4函数
6  8048ce5:  8d 44 00 01          lea    0x1(%eax,%eax,1),%eax#返回值*2+1

```

9. 当 $x=y+(z-y)/2$ 时返回0

10. func4的c代码

这段代码实现了一个递归函数 `func4`，该函数在区间 `[y, z]` 中查找一个数 `x`，如果 `x` 存在则返回 0，否则返回某个整数值。函数 `cal` 的作用是计算 `[y, z]` 中点的值，即 $(y+z)/2$ ，用于判断 `x` 是否在左半部分还是右半部分。

函数 `func4` 的实现是一个经典的二分查找算法，递归地判断 `x` 是否在当前区间 `[y, z]` 中。如果 `x` 比当前区间的中点小，那么继续在左半部分 `[y, cal(y, z)-1]` 中查找，否则在右半部分 `[cal(y, z)+1, z]` 中查找。最终如果找到了 `x`，返回 0，否则返回某个整数值，这个值可以根据具体需求进行设置。

```

1  int cal(int y,int z){
2      return y + (z-y)/2;
3  }
4  int func4(int x, int y, int z){
5      if(cal(y, z) < x)
6          return 2 * func4(x, cal(y, z)+1, z) + 1;
7      else if(cal(y, z) > x)
8          return 2 * func4(x, y, cal(y, z)-1);
9      else
10         return 0;
11 }

```

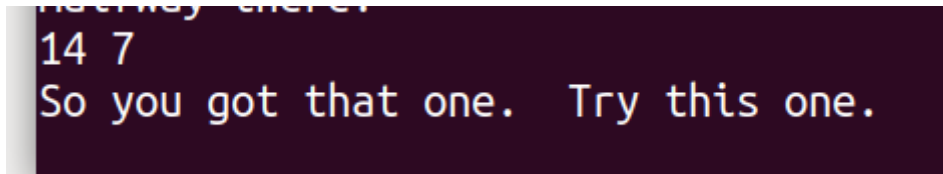
11. 现在的任务就是 $\text{func4}(x, 0, 14) = 7$ ，要求出 `x` 的值。

$\text{func4}(x, 0, 14) = 2 * \text{func4}(x, 7+1, 14) + 1$, $\text{func}(x, 8, 14)$ 需等于3

$\text{func4}(x, 8, 14) = 2 * \text{func4}(x, 11+1, 14) + 1$, $\text{func}(x, 12, 14)$ 需等于1

$\text{func4}(x, 12, 14) = 2 * \text{func4}(x, 13+1, 14) + 1$, $\text{func4}(x, 14, 14)$ 需等于0

$x = 14 + (14-14)/2 = 14$,至此, 推出第一个数为14, 第二个数为7



phase_5

- phase_5的汇编代码

```
1 08048d64 <phase_5>:
2 8048d64: 53                                push    %ebx#保存ebx寄存器到
   栈中
3 8048d65: 83 ec 28                        sub     $0x28,%esp#分配40字
   节的栈空间
4 8048d68: 8b 5c 24 30                    mov     0x30(%esp),%ebx#将
   第3个参数赋值给ebx寄存器
5 8048d6c: 65 a1 14 00 00 00             mov     %gs:0x14,%eax#取出
   GS段寄存器中0x14偏移处的值到eax中
6 8048d72: 89 44 24 1c                    mov     %eax,0x1c(%esp)#将
   eax寄存器的值存储到栈中, 用于stack check
7 8048d76: 31 c0                          xor     %eax,%eax#eax清零
8 8048d78: 89 1c 24                        mov     %ebx, (%esp)#将ebx的
   值存储到栈中
9 8048d7b: e8 7b 02 00 00                call    8048ffb
   <string_length>#调用string_length函数, 返回值存储在eax中
10 8048d80: 83 f8 06                       cmp     $0x6,%eax# 比较eax
   中的值是否为6
11 8048d83: 74 05                          je      8048d8a
   <phase_5+0x26>#如果是, 则跳转到0x8048d8a处
12 8048d85: e8 9c 03 00 00                call    8049126
   <explode_bomb>#如果不是, 调用explode_bomb函数
13 8048d8a: b8 00 00 00 00                mov     $0x0,%eax#将eax赋值
   为0
14 8048d8f: 0f be 14 03                    movsbl  (%ebx,%eax,1),%edx#将ebx寄存器地址偏移eax的值中的一个字节符号扩展到
   edx寄存器中
15 8048d93: 83 e2 0f                       and     $0xf,%edx#将edx的值
   和0xf进行按位与运算
16 8048d96: 0f b6 92 5c a2 04 08          movzbl  0x804a25c(%edx),%edx#将0x804a25c + edx寄存器的地址中的一个字节无符
   号扩展到edx寄存器中
17 8048d9d: 88 54 04 15                    mov     %dl,0x15(%esp,%eax,1)#将edx寄存器的值存储到地址为0x15 + eax * 1的内
   存中
```

```

18  8048da1:  83 c0 01          add    $0x1,%eax# eax寄存
    器自增1
19  8048da4:  83 f8 06          cmp    $0x6,%eax#比较EAX寄
    存器和6是否相等
20  8048da7:  75 e6            jne    8048d8f
    <phase_5+0x2b>#如果不相等则跳转到0x8048d8f
21  8048da9:  c6 44 24 1b 00    movb   $0x0,0x1b(%esp)#将
    立即数0x0存储到地址0x1b(%esp)中，即将0x1b(%esp)清零
22  8048dae:  c7 44 24 04 32 a2 04  movl   $0x804a232,0x4(%esp)#将立即数0x804a232存储到地址0x4(%esp)中
23  8048db5:  08
24  8048db6:  8d 44 24 15      lea     0x15(%esp),%eax#将
    0x15(%esp)的地址赋给EAX寄存器
25  8048dba:  89 04 24          mov     %eax,(%esp)# 将EAX
    寄存器的值存储到(%esp)中，即将0x15(%esp)的地址存入栈中
26  8048dbd:  e8 52 02 00 00    call   8049014
    <strings_not_equal>#调用strings_not_equal函数
27  8048dc2:  85 c0            test    %eax,%eax#将EAX寄存
    器的值和0进行比较
28  8048dc4:  74 05            je      8048dcb
    <phase_5+0x67>#如果两者相等则跳转到0x8048dcb
29  8048dc6:  e8 5b 03 00 00    call   8049126
    <explode_bomb>#否则调用explode_bomb函数
30  8048dcb:  8b 44 24 1c      mov     0x1c(%esp),%eax#将
    地址0x1c(%esp)中的值存储到EAX寄存器中
31  8048dcf:  65 33 05 14 00 00 00 xor     %gs:0x14,%eax#0x14
    的值与EAX寄存器的值进行异或
32  8048dd6:  74 05            je      8048ddd
    <phase_5+0x79>#如果结果为0，则跳转到0x8048ddd
33  8048dd8:  e8 f3 f9 ff ff    call   80487d0
    <__stack_chk_fail@plt>#否则调用__stack_chk_fail函数
34  8048ddd:  83 c4 28          add     $0x28,%esp# 释放40
    个字节的栈空间
35  8048de0:  5b              pop     %ebx#弹出EBX寄存器的
    值
36  8048de1:  c3              ret     # 返回

```

1. 调用string_length测长度，若不为6则爆炸

```

1  8048d68:  8b 5c 24 30          mov     0x30(%esp),%ebx#%ebx=输入字符首地址
2  8048d6c:  65 a1 14 00 00 00    mov     %gs:0x14,%eax#取出GS
    段寄存器中0x14偏移处的值到eax中，gs是段寄存器
3  8048d72:  89 44 24 1c          mov     %eax,0x1c(%esp)
4  8048d76:  31 c0                xor     %eax,%eax#清零
5  8048d78:  89 1c 24              mov     %ebx, (%esp)#传参
6  8048d7b:  e8 7b 02 00 00      call    8048ffb
    <string_length>#调用string_length函数
7  8048d80:  83 f8 06              cmp     $0x6,%eax# 比较eax中
    的值是否为6

```

- 伪代码

```

1  if (string_length(output) != 6) {
2      explode_bomb();
3  }

```

2. 循环6次，取出字符，截取低四位

与 0xf “与运算” 操作意味着能取到 0x4024b0 处字符串的范围是 0-15

```

1  8048d8a:  b8 00 00 00 00      mov     $0x0,%eax#初始化i=0
2  8048d8f:  0f be 14 03          movsbl  (%ebx,%eax,1),%edx#
    将字符依次赋值
3  8048d93:  83 e2 0f              and     $0xf,%edx#edx取后四位
    (取输入字符串的字符，然后逐次将每个字符与0xf“与”操作，得到的值作为0x4024b0
    处字符串的下标。)
4  8048d96:  0f b6 92 5c a2 04 08  movzbl  0x804a25c(%edx),%edx#字符串首地址0x804a25c
5  8048d9d:  88 54 04 15          mov     %dl,0x15(%esp,%eax,1)#*(esp+eax+0x15) = dl; 数组操作
6  8048da1:  83 c0 01              add     $0x1,%eax# i++
7  8048da4:  83 f8 06              cmp     $0x6,%eax#和6比
8  8048da7:  75 e6                jne     8048d8f
    <phase_5+0x2b>#循环

```

- 伪代码

```

1  for (int i = 0; i != 6; i++) {
2      str[i] = g_str[output[i] & 0xf];
3  }

```


3. `x/s 0x804a25c`, 查看字符串, 通过for循环能够生成一个字符串, 该字符串由输入的每个字符和 `0xf` "与运算" 得到的值, 作为 `maduiersnfotvbyl` 的下标, 来选择字符

```
(gdb) x/s 0x804a25c
0x804a25c <array.2999>: "maduiersnfotvbylSo yo
ctrl-c, do you?"
```

4. `x/s 0x804a232`

```
(gdb) x/s 0x804a232
0x804a232:      "oilers"
(gdb)
```

然后 `*(esp + 0x15) = 0` 是字符串结尾符, 最后将该字符串与 `0x804a232` 处的字符串进行比较。

```
1 8048da9: c6 44 24 1b 00      movb    $0x0,0x1b(%esp)
2 8048dae: c7 44 24 04 32 a2 04  movl
  $0x804a232,0x4(%esp)
3 8048db5: 08
4 8048db6: 8d 44 24 15          lea     0x15(%esp),%eax
5 8048dba: 89 04 24             mov     %eax, (%esp)
6 8048dbd: e8 52 02 00 00       call   8049014
  <strings_not_equal>#调用strings_not_equal函数
7 8048dc2: 85 c0               test    %eax,%eax#与*(esp +
  0x15) = 0;
8 8048dc4: 74 05              je      8048dcb
  <phase_5+0x67>#如果两者相等则跳转到0x8048dcb
9 8048dc6: e8 5b 03 00 00       call   8049126
  <explode_bomb>#否则调用explode_bomb函数
```

- 伪代码

```
1 str[7] = '\0';
2 if(string_not_equal(str, "oilers") != 0) {
3     explode_bomb();
4 }
```

5. 根据输入字符ASCII码低四位与索引比较, 不同则爆炸

我们得到的作为参考用，也就是索引表功能的字符串为 `maduiersnfotvbyl`，而我们的目的字符串为 `oilers`，分别为10位，4位，15位，5位，6位和7位，所以只要我们的六个字符的相应的低4位的二进制表示为 `1010 0100 1111 0101 0110 0111`，ASCII 码末尾四位为上述值的字符：`zdoefw` (答案不唯一)

```
zdoefw
Good work! On to the next...
```

- 伪代码

```
1  const char g_str[16] = "maduiersnfotvbyl";
2  void phase_5(char* output)
3  {
4      char str[7];
5      if (string_length(output) != 6) {
6          explode_bomb();
7      }
8      for (int i = 0; i != 6; i++) {
9          str[i] = g_str[output[i] & 0xf];
10     }
11     str[7] = '\0';
12     if(string_not_equal(str, "oilers") != 0) {
13         explode_bomb();
14     }
15 }
```

phase_6

- phase_6的汇编代码

```
1  08048de2 <phase_6>:
2      8048de2:  56                    push    %esi#将 esi 入栈
3      8048de3:  53                    push    %ebx#将 ebx 入栈
4      8048de4:  83 ec 44              sub     $0x44,%esp#为局部变量分配空间
5      8048de7:  8d 44 24 10          lea     0x10(%esp),%eax#将 esp+0x10 的地址存入 eax(输入的6个数)
6      8048deb:  89 44 24 04          mov     %eax,0x4(%esp)#将 eax 的值存入 esp+0x4 的地址中
7      8048def:  8b 44 24 50          mov     0x50(%esp),%eax#将 esp+0x50 的值存入 eax
8      8048df3:  89 04 24              mov     %eax,(%esp)#将 eax 的值存入 esp 的地址中
```

```

9      8048df6:  e8 60 04 00 00          call    804925b
      <read_six_numbers>#调用 read_six_numbers 函数
10     8048dfb:  be 00 00 00 00          mov     $0x0,%esi#将 0 存入
      esi 寄存器
11     8048e00:  8b 44 b4 10             mov     0x10(%esp,%esi,4),%eax#将 esp+esi*4+0x10 的值存入 eax
12     8048e04:  83 e8 01                sub     $0x1,%eax#将 eax 的
      值减去 1
13     8048e07:  83 f8 05                cmp     $0x5,%eax#将 eax 的
      值和 5 进行比较
14
15     8048e0a:  76 05                  jbe     8048e11
      <phase_6+0x2f>#如果 eax 的值小于等于 5, 跳转到 phase_6+0x2f
16     8048e0c:  e8 15 03 00 00          call    8049126
      <explode_bomb>#如果比较结果为大于 5, 调用 explode_bomb 函数
17     8048e11:  83 c6 01                add     $0x1,%esi#将 esi 的
      值加上 1
18     8048e14:  83 fe 06                cmp     $0x6,%esi#将 esi 的
      值和 6 进行比较
19     8048e17:  74 33                  je      8048e4c
      <phase_6+0x6a>#如果比较结果为等于 6, 跳转到 phase_6+0x6a
20     8048e19:  89 f3                  mov     %esi,%ebx#将 esi 的
      值存入 ebx
21     8048e1b:  8b 44 9c 10             mov     0x10(%esp,%ebx,4),%eax#将 esp+ebx*4+0x10 的值存入 eax
22     8048e1f:  39 44 b4 0c             cmp     %eax,0xc(%esp,%esi,4)
23     8048e23:  75 05                  jne     8048e2a
      <phase_6+0x48>#比较esp+esi4+0xc的值与eax的值, 跳转到0x8048e2a
      <phase_6+0x48>位置
24     8048e25:  e8 fc 02 00 00          call    8049126
      <explode_bomb>
25     8048e2a:  83 c3 01                add     $0x1,%ebx#将ebx加1
26     8048e2d:  83 fb 05                cmp     $0x5,%ebx# 比较ebx
      和5的值, 跳转到0x8048e1b <phase_6+0x39>
27     8048e30:  7e e9                  jle     8048e1b
      <phase_6+0x39>
28     8048e32:  eb cc                  jmp     8048e00
      <phase_6+0x1e>#无条件跳转到8048e00 <phase_6+0x1e>处执行
29     8048e34:  8b 52 08             mov     0x8(%edx),%edx#从
      edx + 8地址中读取数据, 存入edx寄存器中
30     8048e37:  83 c0 01                add     $0x1,%eax#eax寄存器
      的值加1
31     8048e3a:  39 c8                  cmp     %ecx,%eax#比较eax和
      ecx寄存器中的值

```

```

32  8048e3c:  75 f6                                jne    8048e34
    <phase_6+0x52>#如果它们不相等，则跳转到8048e34
33  8048e3e:  89 54 b4 28                          mov     %edx,0x28(%esp,%esi,4)# 将edx寄存器中的值存入%esp + %esi * 4 +
    0x28的地址中
34  8048e42:  83 c3 01                              add     $0x1,%ebx# eax寄存
    器的值加1
35  8048e45:  83 fb 06                              cmp     $0x6,%ebx#比较eax寄
    存器和0x6的值
36  8048e48:  75 07                                jne     8048e51
    <phase_6+0x6f>#如果它们不相等，则跳转到8048e51
37  8048e4a:  eb 1c                                jmp     8048e68
    <phase_6+0x86>#否则跳转到8048e68 <phase_6+0x86>处执行
38  8048e4c:  bb 00 00 00 00                      mov     $0x0,%ebx#将0赋值给
    ebx寄存器
39  8048e51:  89 de                                mov     %ebx,%esi#将ebx寄存
    器的值存入esi寄存器中
40  8048e53:  8b 4c 9c 10                          mov     0x10(%esp,%ebx,4),%ecx#从esp + ebx * 4 + 0x10地址中读取数据，存入
    ecx寄存器中
41  8048e57:  b8 01 00 00 00                      mov     $0x1,%eax# 将0x1赋
    值给eax寄存器
42  8048e5c:  ba 3c c1 04 08                      mov     $0x804c13c,%edx#将
    0x804c13c赋值给edx寄存器
43  8048e61:  83 f9 01                              cmp     $0x1,%ecx# 比较ecx
    寄存器和0x1的值
44  8048e64:  7f ce                                jg      8048e34
    <phase_6+0x52>#如果ecx寄存器的值大于1，则跳转到8048e34
    <phase_6+0x52>处执行
45  8048e66:  eb d6                                jmp     8048e3e
    <phase_6+0x5c>#否则跳转到8048e3e <phase_6+0x5c>处执行
46  8048e68:  8b 5c 24 28                          mov     0x28(%esp),%ebx#从
    esp + 0x28地址中读取数据，存入ebx寄存器中
47  8048e6c:  8b 44 24 2c                          mov     0x2c(%esp),%eax#从
    esp + 0x2c地址中读取数据，存入eax
48  8048e70:  89 43 08                              mov     %eax,0x8(%ebx)#将
    eax寄存器的值移动到ebx寄存器中的地址偏移8的位置
49  8048e73:  8b 54 24 30                          mov     0x30(%esp),%edx# 将
    栈顶偏移30字节的位置中的值移动到edx寄存器中
50  8048e77:  89 50 08                              mov     %edx,0x8(%eax)#将
    edx寄存器中的值移动到eax寄存器中的地址偏移8的位置
51  8048e7a:  8b 44 24 34                          mov     0x34(%esp),%eax#将
    栈顶偏移34字节的位置中的值移动到eax寄存器中
52  8048e7e:  89 42 08                              mov     %eax,0x8(%edx)# 将
    eax寄存器中的值移动到edx寄存器中的地址偏移8的位置

```

53	8048e81:	8b 54 24 38	mov	0x38(%esp),%edx#将 栈顶偏移38字节的位置中的值移动到edx寄存器中
54	8048e85:	89 50 08	mov	%edx,0x8(%eax)#将 edx寄存器中的值移动到eax寄存器中的地址偏移8的位置
55	8048e88:	8b 44 24 3c	mov	0x3c(%esp),%eax#将 栈顶偏移3c字节的位置中的值移动到eax寄存器中
56	8048e8c:	89 42 08	mov	%eax,0x8(%edx)#将 eax寄存器中的值移动到edx寄存器中的地址偏移8的位置
57	8048e8f:	c7 40 08 00 00 00 00	movl	\$0x0,0x8(%eax)#将0 移动到eax寄存器中地址偏移8的位置
58	8048e96:	be 05 00 00 00	mov	\$0x5,%esi#将5移动到 esi寄存器中
59	8048e9b:	8b 43 08	mov	0x8(%ebx),%eax#将 ebx寄存器中地址偏移8的值移动到eax寄存器中
60	8048e9e:	8b 10	mov	(%eax),%edx#将eax寄 存器中的值所指向的地址中的值移动到edx寄存器中
61	8048ea0:	39 13	cmp	%edx, (%ebx)
62	8048ea2:	7d 05	jge	8048ea9 <phase_6+0xc7>#如果%edx大于等于%ebx指向的内存地址的值, 跳转到 8048ea9, 否则继续执行
63	8048ea4:	e8 7d 02 00 00	call	8049126 <explode_bomb>
64	8048ea9:	8b 5b 08	mov	0x8(%ebx),%ebx# 将%ebx寄存器的值设置为%ebx指向的内存地址的偏移量为8的位置的值
65	8048eac:	83 ee 01	sub	\$0x1,%esi#将%esi寄 存器的值减1
66	8048eaf:	75 ea	jne	8048e9b <phase_6+0xb9>#如果%esi不等于0, 跳转到8048e9b
67	8048eb1:	83 c4 44	add	\$0x44,%esp#弹出%ebx 寄存器的值
68	8048eb4:	5b	pop	%ebx#弹出%ebx寄存器 的值
69	8048eb5:	5e	pop	%esi#弹出%esi寄存器 的值
70	8048eb6:	c3	ret	

1. 可以看见前面还是调用了read_six_numbers, 所以本关还是输入六个数字
2. 判断这6个数字是否都小于等于6并且互相不重复, 如果不满足则引爆炸弹

1	8048de7:	8d 44 24 10	lea	0x10(%esp),%eax#(输 入的6个数)
2	8048deb:	89 44 24 04	mov	%eax,0x4(%esp)
3	8048def:	8b 44 24 50	mov	0x50(%esp),%eax
4	8048df3:	89 04 24	mov	%eax, (%esp)#另外一个 参数

```

5  8048df6:  e8 60 04 00 00      call  804925b
   <read_six_numbers>#判断输入的确实是6个数
6  8048dfb:  be 00 00 00 00      mov    $0x0,%esi#i=0
7  8048e00:  8b 44 b4 10          mov    0x10(%esp,%esi,4),%eax#%eax=a[i]
8  8048e04:  83 e8 01             sub    $0x1,%eax#%eax=a[i]-1
9  8048e07:  83 f8 05             cmp    $0x5,%eax#5
10
11 8048e0a:  76 05               jbe    8048e11
   <phase_6+0x2f>#a[i]-1<=5跳转，即输入的数字小于等于6
12 8048e11:  83 c6 01             add    $0x1,%esi#i++
13 8048e14:  83 fe 06             cmp    $0x6,%esi#循环6次
14 8048e17:  74 33               je     8048e4c
   <phase_6+0x6a>#i=6跳转 跳出循环(外层循环)

```

```

1  8048e19:  89 f3               mov    %esi,%ebx#ebx=j=i(内
   层循环 初始化)
2  8048e1b:  8b 44 9c 10          mov    0x10(%esp,%ebx,4),%eax#%eax=a[j]
3  8048e1f:  39 44 b4 0c          cmp    %eax,0xc(%esp,%esi,4)#a[i] a[j]
4  8048e23:  75 05               jne    8048e2a
   <phase_6+0x48>#不相等就跳转 相等就bomb
5  8048e25:  e8 fc 02 00 00      call  8049126
   <explode_bomb>
6  8048e2a:  83 c3 01             add    $0x1,%ebx#j++
7  8048e2d:  83 fb 05             cmp    $0x5,%ebx
8  8048e30:  7e e9               jle    8048e1b
   <phase_6+0x39>#j<=5继续内层循环
9  8048e32:  eb cc               jmp    8048e00
   <phase_6+0x1e>#外层循环继续

```

- 伪代码

```

1  for(int i=0;i<=6;i++){
2      if(nums[i]>6){
3          bomb();
4      }
5      for(int j=i;j<=5;j++){
6          if(nums[i]==nums[j]){
7              bomb();
8          }
9      }
10 }

```

4. 生成 node_array, 程序从i=1开始, 将结构体数组中的第a[i]个元素的首地址存放进node_array[i]中, 循环6次, 直到i=6。

```

1  8048e34:  8b 52 08                mov
0x8(%edx),%edx#node=node->next
2  8048e37:  83 c0 01                add    $0x1,%eax#%eax+1
j++
3  8048e3a:  39 c8                  cmp    %ecx,%eax#j<cur
4  8048e3c:  75 f6                  jne    8048e34
<phase_6+0x52>#如果它们不相等, 则跳转(内循环)
5
6  8048e3e:  89 54 b4 28            mov
%edx,0x28(%esp,%esi,4)#node_array[i] = node
7  8048e42:  83 c3 01                add    $0x1,%ebx#ebx+1
8  8048e45:  83 fb 06                cmp    $0x6,%ebx#和6比较大
小 猜测在一个循环里(外循环)
9  8048e48:  75 07                  jne    8048e51
<phase_6+0x6f>#如果它们不相等, 则跳转
10 8048e4a:  eb 1c                  jmp    8048e68
<phase_6+0x86>#跳出循环

```



```

1  8048e4c:  bb 00 00 00 00      mov     $0x0,%ebx
2  8048e51:  89 de               mov     %ebx,%esi#%esi=0
3  8048e53:  8b 4c 9c 10         mov     0x10(%esp,%ebx,4),%ecx#%ecx=a[i]
4  8048e57:  b8 01 00 00 00      mov     $0x1,%eax# %eax=1
5  8048e5c:  ba 3c c1 04 08      mov     $0x804c13c,%edx#将
                        0x804c13c赋值给edx寄存器
6  8048e61:  83 f9 01            cmp     $0x1,%ecx# 比较%ecx
                        0x1(if(cur>1))
7  8048e64:  7f ce              jg      8048e34
                        <phase_6+0x52>#如果%ecx大于1, 则跳转到8048e34(内循环)
8  8048e66:  eb d6              jmp     8048e3e
                        <phase_6+0x5c>#否则跳转到8048e3e(外循环)

```

- 伪代码

```

1      for (int i = 0; i < 6; i ++) {
2          int cur = a[i];
3          ListNode* node = 0x804c13c;      // 链表head
4          if (cur > 1) {
5              for (int j = 1; j < cur; j++) {
6                  node = node->next;
7              }
8          }
9          node_array[i] = node;
10     }

```

```

1  8048e68:  8b 5c 24 28         mov     0x28(%esp),%ebx#%ebx=node_array[0]
2  8048e6c:  8b 44 24 2c         mov     0x2c(%esp),%eax#%ecx=node_array[1]
3  8048e70:  89 43 08             mov     %eax,0x8(%ebx)#node_array[0]->next = node_array[1];
4  8048e73:  8b 54 24 30         mov     0x30(%esp),%edx#%edx=node_array[2]
5  8048e77:  89 50 08             mov     %edx,0x8(%eax)#node_array[1]->next = node_array[2]
6  8048e7a:  8b 44 24 34         mov     0x34(%esp),%eax#%eax=node_array[3]
7  8048e7e:  89 42 08             mov     %eax,0x8(%edx)#node_array[2]->next = node_array[3]
8  8048e81:  8b 54 24 38         mov     0x38(%esp),%edx#%edx=node_array[4]

```

```

9   8048e85:  89 50 08                mov
   %edx,0x8(%eax)#node_array[3]->next = node_array[4]
10  8048e88:  8b 44 24 3c            mov
   0x3c(%esp),%eax#%edx=node_array[5]
11  8048e8c:  89 42 08                mov
   %eax,0x8(%edx)#node_array[4]->next = node_array[5]
12  8048e8f:  c7 40 08 00 00 00 00   movl
   $0x0,0x8(%eax)#node_array[4]->next = 0
13  8048e96:  be 05 00 00 00        mov    $0x5,%esi#%esi=5
14  8048e9b:  8b 43 08                mov
   0x8(%ebx),%eax#%eax=node_array[0]
15  8048e9e:  8b 10                  mov    (%eax),%edx
16  8048ea0:  39 13                  cmp    %edx, (%ebx)#%edx=
   (%eax) (%ebx)
17  8048ea2:  7d 05                  jge    8048ea9
   <phase_6+0xc7>#如果%edx大于等于%ebx指向的内存地址的值，跳转到
   8048ea9，否则继续执行
18  8048ea4:  e8 7d 02 00 00        call   8049126
   <explode_bomb>

```

- 伪代码

```

1   for (int i = 0; i < 5; i++) {
2       node_array[i]->next = node_array[i+1];
3   }

```

4. 0x804c13c 是链表地址，p/x *0x804c13c@18 输出获得链表，phase_6中引用了来自0x804c13c的结构体数组，该结构体共16个字节，其中前8个字节存放了一个int型的数字（结构体对齐），后8个字节存放了一个指针，该指针在指向结构体数组中的下一个元素。这实际上是一个有6个元素的单向链表。

```

1   8048e5c:  ba 3c c1 04 08        mov    $0x804c13c,%edx#; 链
   表地址

```

```

(gdb) p/x *0x804c13c@18
$3 = {0x364, 0x1, 0x804c148, 0x2fa, 0x2, 0x804c154, 0x34a, 0x3, 0x804c160,
      0x1bd, 0x4, 0x804c16c, 0x293, 0x5, 0x804c178, 0x138, 0x6, 0x0}
(gdb)

```

三个数一组，第一个数是一个权值，第二个数按照 123456 排序下来，第三个数是下一个节点的首地址

```

1   typedef struct {
2       int val;
3       ListNode* next;
4   } ListNode;

```

5. 链表的权值需要按着递减的顺序排列，否则就 bomb

```
1  8048e96:  be 05 00 00 00      mov     $0x5,%esi#遍历链表
2  8048e9b:  8b 43 08             mov     0x8(%ebx),%eax#下一个
    结点的首地址 %eax=node_array[i]
3  8048e9e:  8b 10               mov     (%eax),%edx#%edx=下
    一个结点的权值
4  8048ea0:  39 13               cmp     %edx,(%ebx)#比较两个
    结点的权值
5  8048ea2:  7d 05               jge     8048ea9
    <phase_6+0xc7>#如果>=就不会bomb
6  8048ea4:  e8 7d 02 00 00      call    8049126
    <explode_bomb>
7  8048ea9:  8b 5b 08             mov     0x8(%ebx),%ebx#移到下
    一个结点
8  8048eac:  83 ee 01             sub     $0x1,%esi#将%esi寄存
    器的值减1(i--)
9  8048eaf:  75 ea               jne     8048e9b
    <phase_6+0xb9>#如果%esi不等于0，跳转到8048e9b(循环)
```

```
1  ListNode* ptr = node_array[0];
2  for (int i = 5; i > 0; i--) {
3      if (ptr->val < ptr->next->val)
4          explode_bomb();
5      ptr = ptr->next;
6  }
```

6. 排序，按权值递减

• 伪代码

```
1  typedef struct {
2      int val;
3      ListNode* next;
4  } ListNode;
5
6  void phase_6(char* output)
7  {
8      int array[6];
9      ListNode* node_array[6];
10     read_six_numbers(output, array);
11     // 数字范围必须为1-6且互不重复
12     for (int i = 0; i != 6; i++) {
13         int num = array[i];
14         num--;
```

```
15         if ((unsigned int)num > 5)           // 最大为6
16             explode_bomb();
17         for (int j = i+1; j <= 5; j++) {
18             if (array[i] == array[j])        // 每个元素都不重复
19                 explode_bomb();
20         }
21     }
22     // 修改 array
23     for (int i = 0; i < 6; i++) {
24         array[i] = (7 - array[i]);
25     }
26     // 生成 node_array
27     for (int i = 0; i < 6; i++) {
28         int cur = array[i];
29         ListNode* node = 0x6032d0;          // 链表head
30         if (cur > 1) {
31             for (int j = 1; j < cur; j++) {
32                 node = node->next;
33             }
34         }
35         node_array[i] = node;
36     }
37     for (int i = 0; i < 5; i++) {
38         node_array[i]->next = node_array[i+1];
39     }
40
41     ListNode* ptr = node_array[0];
42     for (int i = 5; i > 0; i--) {
43         if (ptr->val < ptr->next->val)
44             explode_bomb();
45         ptr = ptr->next;
46     }
47 }
```

权值	第二个数
364	1
34a	3
2fa	2
293	5
1bd	4
138	6

```

good work. on to the next...
1 3 2 5 4 6
Congratulations! You've defused the bomb!

```

secret_phase

在文件中找到有 secret_phase 的地方，发现在 phase_defused 中有跳转到 secret_phase 函数的地方

```

1 | 8049327:    e8 dc fb ff ff        call    8048f08
    <secret_phase>

```

而 phase_defused 是每次拆除炸弹都会走的地方，看来就是从 phase_defused 中进入隐藏炸弹。调试观察 phase_defused 函数，这里是判断你是不是六个炸弹都已经拆除了，拆除了才可以进入隐藏炸

```

1 | 80492bd:    83 3d cc c3 04 08 06    cmp1    $0x6,0x804c3cc

```

```

1 | 804931b:    c7 04 24 cc a2 04 08    movl    $0x804a2cc, (%esp)

```

```

(gdb) x/s 0x804a3d5
0x804a3d5:    "%d %d %s"

```

要两个整数，一个字符，后面判断如果不是 3 个就进入不了隐藏关，看来进入隐藏关的关键是这个字符

查看内存为 0x804a3de 处的值，这个就是进入隐藏关卡的字符串密码值

```

(gdb) x/s 0x804a3de
0x804a3de:    "DrEvil"

```

上面的“%d %d %s”也提示我们第四关可以输入一个字符串，而后面的cmp \$0x3,%eax更验证了这一点：如果第四关输入了三个参数则触发隐藏关，否则直接跳转至0x804932c

```
1 80492f2: 83 f8 03          cmp     $0x3,%eax
2 80492f5: 75 35             jne     804932c
   <phase_defused+0x81>
```

```
cs18@games101vm:~/Desktop/csapp/lab3/bomb202107030125/bomb103$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
When a problem comes along, you must zip it!
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
0 -760
Halfway there!
14 7 DrEvil
So you got that one. Try this one.
zdoefw
Good work! On to the next...
1 3 2 5 4 6
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

- secret_phase的汇编代码

```
1 08048f08 <secret_phase>:
2 8048f08: 53                push    %ebx
3 8048f09: 83 ec 18          sub     $0x18,%esp
4 8048f0c: e8 3c 02 00 00    call    804914d
   <read_line>#输入一个数字
5 8048f11: c7 44 24 08 0a 00 00 movl    $0xa,0x8(%esp)
6 8048f18: 00
7 8048f19: c7 44 24 04 00 00 00 movl    $0x0,0x4(%esp)
8 8048f20: 00
9 8048f21: 89 04 24          mov     %eax,(%esp)# 准备参
   数
10 8048f24: e8 b7 f9 ff ff    call    80488e0
   <strtol@plt>
11 8048f29: 89 c3            mov     %eax,%ebx
12 8048f2b: 8d 40 ff         lea     -0x1(%eax),%eax
13 8048f2e: 3d e8 03 00 00    cmp     $0x3e8,%eax# 数字不
   能大于1001
14 8048f33: 76 05            jbe     8048f3a
   <secret_phase+0x32>#输入小于等于0x3e8
15 8048f35: e8 ec 01 00 00    call    8049126
   <explode_bomb>
```

```

16  8048f3a:  89 5c 24 04      mov    %ebx,0x4(%esp)#参数
    2
17  8048f3e:  c7 04 24 88 c0 04 08  movl   $0x804c088, (%esp)#
    参数1
18  8048f45:  e8 6d ff ff ff   call   8048eb7 <fun7>#调用
    func7 返回值需是6
19  8048f4a:  83 f8 06         cmp    $0x6,%eax#检查返回值
20  8048f4d:  74 05           je     8048f54
    <secret_phase+0x4c>
21  8048f4f:  e8 d2 01 00 00   call   8049126
    <explode_bomb>
22  8048f54:  c7 04 24 0c a2 04 08  movl   $0x804a20c, (%esp)
23  8048f5b:  e8 a0 f8 ff ff   call   8048800 <puts@plt>
24  8048f60:  e8 46 03 00 00   call   80492ab
    <phase_defused>
25  8048f65:  83 c4 18         add    $0x18,%esp
26  8048f68:  5b              pop    %ebx
27  8048f69:  c3              ret
28  8048f6a:  90              nop
29  8048f6b:  90              nop
30  8048f6c:  90              nop
31  8048f6d:  90              nop
32  8048f6e:  90              nop
33  8048f6f:  90              nop

```

- func7的汇编代码

```

1  08048eb7 <fun7>:
2  8048eb7:  53              push   %ebx
3  8048eb8:  83 ec 18        sub    $0x18,%esp#准备工作
4  8048ebb:  8b 54 24 20      mov    0x20(%esp),%edx#%edx=root
5  8048ebf:  8b 4c 24 24      mov    0x24(%esp),%ecx#%ecx=x
6  8048ec3:  85 d2           test   %edx,%edx#相与
7  8048ec5:  74 37           je     8048efe
    <fun7+0x47>#如果root==NULL 跳转return很大的数
8  8048ec7:  8b 1a           mov    (%edx),%ebx#%ebx=root.v
9  8048ec9:  39 cb           cmp    %ecx,%ebx#比较
    root.v-x<=0
10 8048ecb:  7e 13           jle    8048ee0
    <fun7+0x29>#小于等于就跳转
11 8048ecd:  89 4c 24 04      mov    %ecx,0x4(%esp)#x

```



```

12  8048ed1:  8b 42 04          mov     0x4(%edx),%eax#取
    root->l
13  8048ed4:  89 04 24          mov     %eax, (%esp)
14  8048ed7:  e8 db ff ff ff    call    8048eb7
    <fun7>#func7(root->l,x)
15  8048edc:  01 c0            add     %eax,%eax#递归返回后
    将其返回值加倍
16  8048ede:  eb 23            jmp     8048f03
    <fun7+0x4c>#如果root.v>x return 2*fun(root->l,x)
17  8048ee0:  b8 00 00 00 00    mov     $0x0,%eax#%eax=0
18  8048ee5:  39 cb            cmp     %ecx,%ebx#重复比较
19  8048ee7:  74 1a            je      8048f03
    <fun7+0x4c>#如果x==root.v 函数结束 return 0
20  8048ee9:  89 4c 24 04       mov     %ecx,0x4(%esp)
21  8048eed:  8b 42 08          mov     0x8(%edx),%eax#root->r
22  8048ef0:  89 04 24          mov     %eax, (%esp)
23  8048ef3:  e8 bf ff ff ff    call    8048eb7
    <fun7>#func7(root->r,x)
24  8048ef8:  8d 44 00 01       lea     0x1(%eax,%eax,1),%eax# 递归结束后将返回值加倍在加1
25  8048efc:  eb 05            jmp     8048f03
    <fun7+0x4c>
26  8048efe:  b8 ff ff ff ff    mov     $0xffffffff,%eax#%eax很大的数
27  8048f03:  83 c4 18          add     $0x18,%esp#函数结束
28  8048f06:  5b               pop     %ebx
29  8048f07:  c3               ret

```

1. 在前面可以看到调用的strtol函数，该函数将输入的字符串转化成数字，再往下有cmp指令，要求输入的数字小于等于1001

```

1  8048f0c:  e8 3c 02 00 00    call    804914d
    <read_line>#输入一个数字

```

```

1  8048f33:  76 05            jbe     8048f3a
    <secret_phase+0x32>#输入小于等于0x3e8

```

2. 再接下来便是准备参数的环节，准备调用fun7函数，可以看到在调用完fun7函数后，对fun7函数的返回值进行了检查，只有返回值等于6时才不会引爆炸弹，因此本题的关键在函数fun7上，只要fun7的返回值等于6即可。

1	8048f3a:	89 5c 24 04	mov	%ebx, 0x4(%esp) #参数2
2	8048f3e:	c7 04 24 88 c0 04 08	movl	\$0x804c088, (%esp) #参数1
3	8048f45:	e8 6d ff ff ff	call	8048eb7 <fun7> #调用func7 返回值需是6
4	8048f4a:	83 f8 06	cmp	\$0x6, %eax #检查返回值

3. fun7是一个递归函数，secret_phase在调用fun7时传入的两个初始参数是0x804c088和我们输入的那个数，下面用gdb调试查看参数1储存的内容：

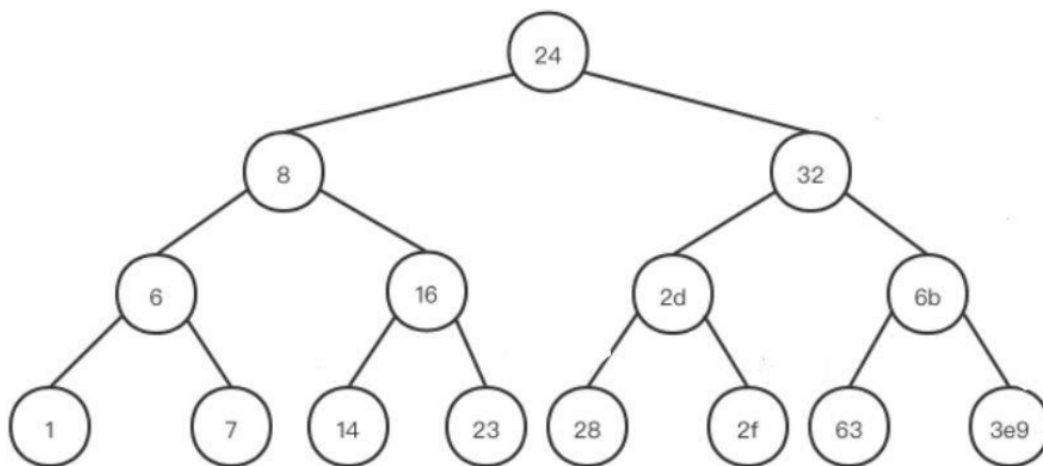
```
(gdb) x/256bx 0x804c088
0x804c088 <n1>: 0x24 0x00 0x00 0x00 0x94 0xc0 0x04 0x08
0x804c090 <n1+8>: 0xa0 0xc0 0x04 0x08 0x08 0x00 0x00 0x00
0x804c098 <n21+4>: 0xc4 0xc0 0x04 0x08 0xac 0xc0 0x04 0x08
0x804c0a0 <n22>: 0x32 0x00 0x00 0x00 0xb8 0xc0 0x04 0x08
0x804c0a8 <n22+8>: 0xd0 0xc0 0x04 0x08 0x16 0x00 0x00 0x00
0x804c0b0 <n32+4>: 0x18 0xc1 0x04 0x08 0x00 0xc1 0x04 0x08
0x804c0b8 <n33>: 0x2d 0x00 0x00 0x00 0xdc 0xc0 0x04 0x08
0x804c0c0 <n33+8>: 0x24 0xc1 0x04 0x08 0x06 0x00 0x00 0x00
0x804c0c8 <n31+4>: 0xe8 0xc0 0x04 0x08 0x0c 0xc1 0x04 0x08
0x804c0d0 <n34>: 0x6b 0x00 0x00 0x00 0xf4 0xc0 0x04 0x08
0x804c0d8 <n34+8>: 0x30 0xc1 0x04 0x08 0x28 0x00 0x00 0x00
0x804c0e0 <n45+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c0e8 <n41>: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c0f0 <n41+8>: 0x00 0x00 0x00 0x00 0x00 0x63 0x00 0x00
0x804c0f8 <n47+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c100 <n44>: 0x23 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c108 <n44+8>: 0x00 0x00 0x00 0x00 0x07 0x00 0x00 0x00
0x804c110 <n42+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c118 <n43>: 0x14 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c120 <n43+8>: 0x00 0x00 0x00 0x00 0x2f 0x00 0x00 0x00
0x804c128 <n46+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c130 <n48>: 0xe9 0x03 0x00 0x00 0x00 0x00 0x00 0x00
0x804c138 <n48+8>: 0x00 0x00 0x00 0x00 0x64 0x03 0x00 0x00
```

0x804c088 是一个结点

可以看出该地址是一种特殊数据结构的首地址，该数据结构第一个字节存放数据，第二和第三个字节存放两个指针，到这里已经可以猜出这个数据结构是什么了，就是二叉树，并且通过变量的命名也能验证这一结论，n1应该是根节点，n21是第二层第一个节点，以此类推...

```
(gdb) p/x *0x804c088@100
$1 = {0x24, 0x804c094, 0x804c0a0, 0x8, 0x804c0c4, 0x804c0ac, 0x32, 0x804c0b8,
0x804c0d0, 0x16, 0x804c118, 0x804c100, 0x2d, 0x804c0dc, 0x804c124, 0x6, 0x804c0e8,
0x804c10c, 0x6b, 0x804c0f4, 0x804c130, 0x28, 0x0, 0x0, 0x1, 0x0, 0x0, 0x63, 0x0, 0x0,
0x23, 0x0, 0x0, 0x7, 0x0, 0x0, 0x14, 0x0, 0x0, 0x2f, 0x0, 0x0, 0x3e9, 0x0, 0x0,
0x364, 0x1, 0x804c148, 0x2fa, 0x2, 0x804c154, 0x34a, 0x3, 0x804c160, 0x1bd, 0x4,
0x804c16c, 0x293, 0x5, 0x804c178, 0x138, 0x6, 0x0, 0x67, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x804a3e5, 0x0 <repeats 29 times>}
```

即是一颗二叉搜索树



4. 接下来看汇编的内容就比较好理解了，edx储存的是当前节点的首地址，edx+8储存的是它的右子节点的首地址，edx+4储存的是它的左子节点的首地址，如果当前节点数据域等于我们输入的数则返回0，大于则返回2*fun7(左子节点首地址, y),小于则返回2*fun7(右子节点首地址, y)+1，下面给出函数原型：

```

1 int fun7(int *btree, int y){
2     if(btree->data == y)
3         return 0;
4     else if(btree->data < y)
5         return 2 * fun7(btree->rchild, y) + 1;
6     else
7         return 2 * fun7(btree->lchild, y);
8 }

```

y的值即为我们要输入的内容，接下来反推y的值：

$\text{fun7}(0x804c088, y) = 6 = 2 * \text{fun7}(0x804c094, y) + 1, \text{fun7}(0x804c094, y)$ 应等于3

$\text{fun7}(0x804c094, y) = 3 = 2 * \text{fun7}(0x804c0ac, y) + 1, \text{fun7}(0x804c0ac, y)$ 应等于1

$\text{fun7}(0x804c0ac, y) = 1 = 2 * \text{fun7}(0x804c100, y) + 1, \text{fun7}(0x804c100, y)$ 应等于0

5. 因此路径应该为 左右右，即0x23

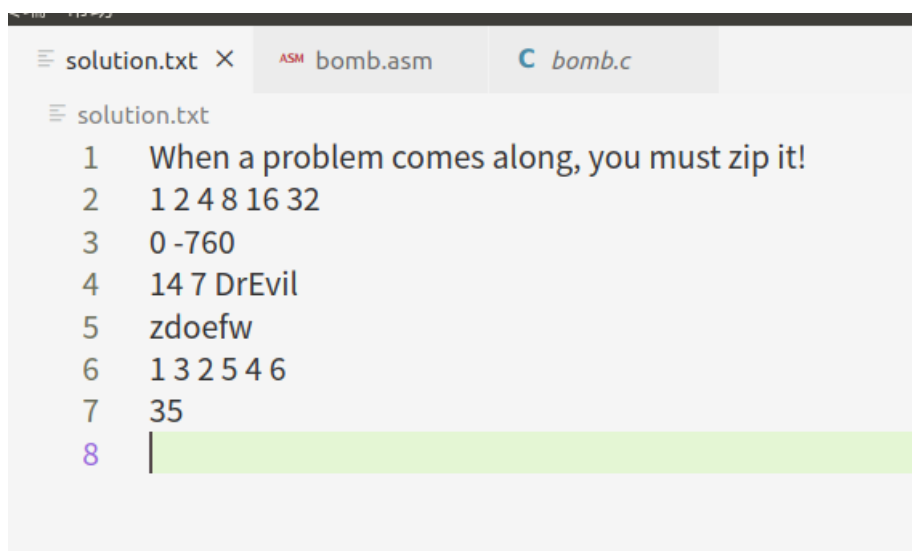
```

Curses, you've found the secret phase!
But finding it and solving it are quite different...
35
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!

```

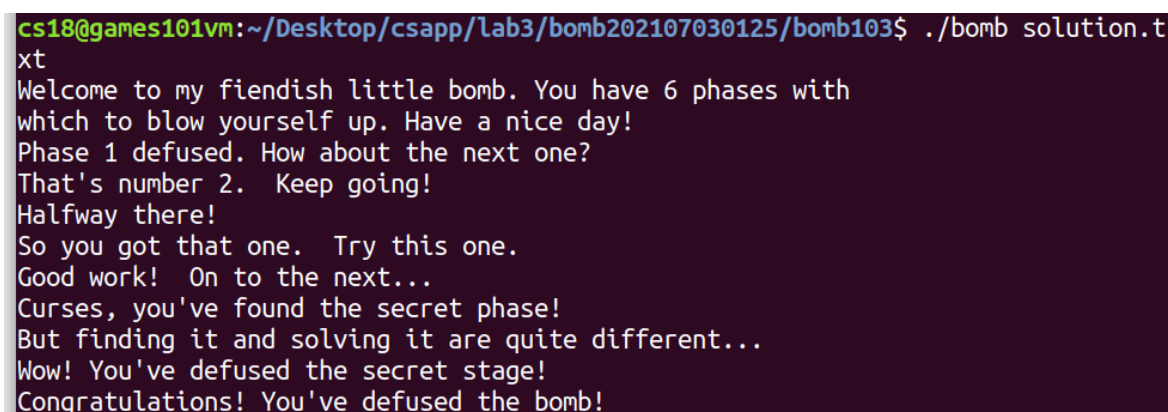
实验结果

solution.txt文件：



```
1  When a problem comes along, you must zip it!
2  1 2 4 8 16 32
3  0 -760
4  14 7 DrEvil
5  zdoefw
6  1 3 2 5 4 6
7  35
8  
```

至此，全部七关已全部通过



```
cs18@games101vm:~/Desktop/csapp/lab3/bomb202107030125/bomb103$ ./bomb solution.t
xt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

实验总结

这一次的bomb实验，包含了计算机系统中第三章汇编语言的几乎所有知识点。通过本次的练习，我的汇编语言能力获得了很好的锻炼，对于一些重要知识点（如跳转表，循环）的知识点，掌握的更加牢靠。而本实验中包含的许多有趣实用的汇编语言技巧（如一些精巧的中间变量的使用、灵活的jump跳转指令的运用）使我更加注意编程技巧的学习。

六关+隐藏关对应了以下内容：

- 常量字符串存储
- 二叉树在汇编代码中的表示
- 链表在汇编中代码的表示
- 字符的ASCII码表示
- 递归调用的过程
- 跳转表

- 循环