

# CSAPP-BombLab 实验报告

## 1 评分

6 bomb65	Thu Nov 2 17:32	7	0	70	valid
----------	-----------------	---	---	----	-------

## 2 解题思路

### 2.1 Phase 1

观察反汇编代码的phase\_1部分，发现主要调用了strings\_not\_equal这个函数，而调用前把0x1801(%rip)加载到了%rsi中作为函数的参数，推断为用我们输入的字符串和这个地址的对比。我们用gdb进行调试，在phase\_1设置断点，如图所示：

```
B+> 0x555555401208 <phase_1>      sub    $0x8,%rsp
0x555555401208 <phase_1+4>      lea    0x1801(%rip),%rsi          # 0x555555402a10
0x55555540120f <phase_1+11>     call   0x555555401721 <strings_not_equal>
0x555555401214 <phase_1+16>     test   %eax,%eax
0x555555401216 <phase_1+18>     jne    0x55555540121d <phase_1+25>
0x555555401218 <phase_1+20>     add    $0x8,%rsp
0x55555540121c <phase_1+24>     ret
0x55555540121d <phase_1+25>     call   0x5555554019dc <explode_bomb>
0x555555401222 <phase_1+30>     jmp   0x555555401218 <phase_1+20>
0x555555401224 <phase_2>       push   %rbp
0x555555401225 <phase_2+1>     push   %rbx
0x555555401226 <phase_2+2>     sub    $0x28,%rsp
0x55555540122a <phase_2+6>     mov    %fs:0x28,%rax
0x555555401233 <phase_2+15>    mov    %rax,0x18(%rsp)
0x555555401238 <phase_2+20>    xor    %eax,%eax
0x55555540123a <phase_2+22>    mov    %rsp,%rsi
multi-thread Thread 0xffff7d8a7 In: phase_1
(gdb) r
Starting program: /home/avalon/bomb65/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
Breakpoint 2, 0x0000555555401204 in phase_1 ()
(gdb)
```

用x/s命令显示地址0x555555402a10的字符串，即可得到答案：

```
(gdb) x/s 0x555555402a10
0x555555402a10: "You can Russia from land here in Alaska."
```

### 2.2 Phase 2

观察phase\_2的反汇编，发现了read\_six\_numbers函数，用si指令进入这个函数

```
B+> 0x555555401224 <phase_2>      push   %rbp
0x555555401225 <phase_2+1>     push   %rbx
0x555555401226 <phase_2+2>     sub    $0x28,%rsp
0x55555540122a <phase_2+6>     mov    %fs:0x28,%rax
0x555555401233 <phase_2+15>    mov    %rax,0x18(%rsp)
0x555555401238 <phase_2+20>    xor    %eax,%eax
0x55555540123a <phase_2+22>    mov    %rsp,%rsi
0x55555540123d <phase_2+25>    call   0x555555401a18 <read_six_numbers>
0x555555401242 <phase_2+30>    cmpl   $0x0,(%rsp)
0x555555401246 <phase_2+34>    js    0x555555401252 <phase_2+46>
0x555555401248 <phase_2+36>    mov    $0x1,%ebx
0x55555540124d <phase_2+41>    mov    %rsi,%rbp
0x555555401250 <phase_2+44>    jmp   0x555555401263 <phase_2+63>
0x555555401252 <phase_2+46>    call   0x5555554019dc <explode_bomb>
0x555555401257 <phase_2+51>    jmp   0x555555401248 <phase_2+36>
0x555555401259 <phase_2+53>    add    $0x1,%rbx
0x55555540125d <phase_2+57>    cmp    $0x6,%rbx
0x555555401261 <phase_2+61>    je    0x555555401276 <phase_2+82>
0x555555401263 <phase_2+63>    mov    %ebx,%eax
0x555555401265 <phase_2+65>    add    -0x4(%rbp,%rbx,4),%eax
0x555555401269 <phase_2+69>    cmp    %eax,0x0(%rbp,%rbx,4)
0x55555540126d <phase_2+73>    je    0x555555401259 <phase_2+53>
0x55555540126f <phase_2+75>    call   0x5555554019dc <explode_bomb>
0x555555401274 <phase_2+80>    jmp   0x555555401259 <phase_2+53>
0x555555401276 <phase_2+82>    mov    $0x18(%rsp),%rax
0x55555540127b <phase_2+87>    xor    %fs:0x28,%rax
```

发现加载了地址0x555555402ca9, 用x/s指令显示

```
0x555555401a18 <read_six_numbers>    sub    $0x8,%rsp
0x555555401a1c <read_six_numbers+4>   mov    %rsi,%rdx
0x555555401a1f <read_six_numbers+7>   lea    0x4(%rsi),%rcx
0x555555401a23 <read_six_numbers+11>  lea    0x14(%rsi),%rax
0x555555401a27 <read_six_numbers+15>  push   %rax
0x555555401a28 <read_six_numbers+16>  lea    0x10(%rsi),%rax
0x555555401a2c <read_six_numbers+20>  push   %rax
0x555555401a2d <read_six_numbers+21>  lea    0xc(%rsi),%r9
0x555555401a31 <read_six_numbers+25>  lea    0x8(%rsi),%r8
0x555555401a35 <read_six_numbers+29>  lea    0x126(%rip),%rsi      # 0x555555402ca9
0x555555401a3c <read_six_numbers+36>  mov    $0x0,%eax
0x555555401a41 <read_six_numbers+41>  call   0x555555400ee0 <__isoc99_sscanf@plt>
0x555555401a46 <read_six_numbers+46>  add    $0x10,%rsp
0x555555401a4a <read_six_numbers+50>  cmp    $0x5,%eax
0x555555401a4d <read_six_numbers+53>  jle    0x555555401a54 <read_six_numbers+60>
0x555555401a4f <read_six_numbers+55>  add    $0x8,%rsp
0x555555401a53 <read_six_numbers+59>  ret
0x555555401a54 <read_six_numbers+60>  call   0x5555554019dc <explode_bomb>
```

发现需要输入六个数字, 用空格分隔

```
(gdb) x/s 0x555555402ca9
0x555555402ca9: "%d %d %d %d %d %d"
$0x0
```

我们发现在`read_six_numbers`后, 首先把内存(%rsp)中的数字和0做了比较, 如果为负数则爆炸, 用指令`x $rsp`可知道, 这是我们输入的第一个数字, 所以一个要求是第一个数字非负。阅读后面的汇编代码, 我们发现它是在做一个循环, 用%rbx作为计数器, 一开始为1, 之后每次+1直到6, 把%rsp赋给了%rbp, 之后用(%rbp,%rbx,4)索引读取我们输入的数字, 每次循环进行检查, 不符合要求就爆炸, 其检查模式为, 每一轮中, 把%rax赋值为当前的%rbx (也就是当前的循环轮数), 然后再加上某一个我们输入的数, 判断此时%rax是否和我们输入的下一个数相等, 比如说, 第一轮中就是判断我们输入的第一个数+1是否等于第二个数, 之后的以此类推, 若不相等就爆炸。到这里就可以得到我们的答案, 我们不妨让第一个数为0 (事实上可以是任何一个非负整数), 第二个数为第一个数+1, 第三个数为第二个数+2, 以此类推, 可以得到一组答案0 1 3 6 10 15, 当然这并不是唯一的, 例如1 2 4 7 11 16这样的答案 (即给我们先前的答案中的每一个数都加上一个相同的正值) 同样可以, 我们可以给出通式 $k k + 1 k + 3 k + 6 k + 10 k + 15, k \geq 0$

## 2.3 Phase 3

在`phase_3`设置断点, 观察汇编如下:

```
B+> 0x555555401292 <phase_3>    sub    $0x18,%rsp
0x555555401296 <phase_3+4>    mov    %fs:0x28,%rax
0x55555540129f <phase_3+13>   mov    %rax,0x8(%rsp)
0x5555554012a4 <phase_3+18>   xor    %eax,%eax
0x5555554012a6 <phase_3+20>   lea    0x4(%rsp),%rcx
0x5555554012ab <phase_3+25>   mov    %rsp,%rdx
0x5555554012ae <phase_3+28>   lea    0x1a0(%rip),%rsi      # 0x555555402cb5
0x5555554012b5 <phase_3+35>   call   0x555555400ee0 <__isoc99_sscanf@plt>
0x5555554012ba <phase_3+40>   cmp    $0x1,%eax
0x5555554012bd <phase_3+43>   jle    0x5555554012dc <phase_3+74>
0x5555554012bf <phase_3+45>   cmpl   $0x7,(%rsp)
0x5555554012c3 <phase_3+49>   ja    0x555555401362 <phase_3+208>
0x5555554012c9 <phase_3+55>   mov    (%rsp),%eax
0x5555554012cc <phase_3+58>   lea    0x179d(%rip),%rdx      # 0x555555402a70
0x5555554012d3 <phase_3+65>   movslq (%rdx,%rax,4),%rax
0x5555554012d7 <phase_3+69>   add    %rdx,%rax
0x5555554012da <phase_3+72>   jmp    *%rax
0x5555554012dc <phase_3+74>   call   0x5555554019dc <explode_bomb>
0x5555554012e1 <phase_3+79>   jmp    0x5555554012bf <phase_3+45>
0x5555554012e3 <phase_3+81>   mov    $0x1d5,%eax
0x5555554012e8 <phase_3+86>   jmp    0x5555554012ef <phase_3+93>
0x5555554012ea <phase_3+88>   mov    $0x0,%eax
0x5555554012ef <phase_3+93>   sub    $0x1fa,%eax
0x5555554012f4 <phase_3+98>   add    $0x1c3,%eax
0x5555554012f9 <phase_3+103>  sub    $0x249,%eax
```

```

0x5555554012fe <phase_3+108>    add    $0x249,%eax
0x555555401303 <phase_3+113>    sub    $0x249,%eax
0x555555401308 <phase_3+118>    add    $0x249,%eax
0x55555540130d <phase_3+123>    sub    $0x249,%eax
> 0x555555401312 <phase_3+128>    cmpl   $0x5,(%rsp)
0x555555401316 <phase_3+132>    jg     0x55555540131e <phase_3+140>
0x555555401318 <phase_3+134>    cmp    %eax,0x4(%rsp)
0x55555540131c <phase_3+138>    je     0x555555401323 <phase_3+145>
0x55555540131e <phase_3+140>    call   0x5555554019dc <explode_bomb>
0x555555401323 <phase_3+145>    mov    0x8(%rsp),%rax
0x555555401328 <phase_3+150>    xor    %fs:0x28,%rax
0x555555401331 <phase_3+159>    jne   0x55555540136e <phase_3+220>
0x555555401333 <phase_3+161>    add    $0x18,%rsp
0x555555401337 <phase_3+165>    ret
0x555555401338 <phase_3+166>    mov    $0x0,%eax
0x55555540133d <phase_3+171>    jmp   0x5555554012f4 <phase_3+98>
0x55555540133f <phase_3+173>    mov    $0x0,%eax
0x555555401344 <phase_3+178>    jmp   0x5555554012f9 <phase_3+103>
0x555555401346 <phase_3+180>    mov    $0x0,%eax
0x55555540134b <phase_3+185>    jmp   0x5555554012fe <phase_3+108>
0x55555540134d <phase_3+187>    mov    $0x0,%eax
0x555555401352 <phase_3+192>    jmp   0x555555401303 <phase_3+113>
0x555555401354 <phase_3+194>    mov    $0x0,%eax
0x555555401359 <phase_3+199>    jmp   0x555555401308 <phase_3+118>

```

和先前相同，我们通过 *x/s 0x555555402cb5* 观察输入格式，发现是两个整数，我们不妨输入 1 2，我们发现它把(%rsp)（检测可得是我们输入的第一个数）和 7 做了比较，如果 > 7，就爆炸，并且用的是无符号指令 ja，也就是说第一个数应为 0 到 7 之间的整数。然后进行了一系列操作，在此我们转换思维，我们可以不像上一题那样搞清楚每一行做了什么，而是用 ni 不断推进程，观察它为什么会爆炸，而核心就是观察 cmp, test 这些指令。我们发现 <phase\_3 + 128> 处进行了一个比较，如果第一个数 > 5 就会爆炸。而之后进行了 %eax 和 0x4(%rsp)（也就是我们输入的第二个数）的比较，不相等就爆炸。我们发现此时 %eax 的值是 -640，而在此之前，我们没用到过第二个数，也就是说，此时 %eax 的值只和第一个数有关（我们也可以输入 1 3 来验证），那么我们已经知道了对于 1 来说，正确的第二个数应该是 -640，至此，我们已经得到了一个答案 1 - 640，当然这个题目的答案显然也不是唯一的，我们可以用同样的方法去找到和第一个数匹配的第二个数，就可以得到其他答案。

当然，上面这种方法听起来有一些取巧，我们可以继续详细研究汇编代码。

```

12c9: 8b 04 24        mov    (%rsp),%eax
12cc: 48 8d 15 9d 17 00 00  lea    0x179d(%rip),%rdx      # 2a70 <_IO_stdin_used+0x1b0>
12d3: 48 63 04 82        movslq (%rdx,%rax,4),%rax
12d7: 48 01 d0        add    %rdx,%rax
12da: ff e0        jmp    *%rax

```

我们先来看一下这个内存处放了什么：

```
(gdb) x/8dw 0x555555402a70
0x555555402a70: -6029    -6022    -5944    -5937
0x555555402a80: -5930    -5923    -5916    -5909
```

我们发现，这段代码是用我们输入的第一个数作为索引在这八个数里面取一个，然后和 %rdx 中的 0x555555402a70 相加放入 %rax，然后跳至这个地址。可以做如下的表格：

我们输入的第一个数	%rax 中的地址
0	0x5555554012e3
1	0x5555554012ea

2	0x555555401338
3	0x55555540133f
4	0x555555401346
5	0x55555540134d
6	0x555555401354
7	0x55555540135b

我们发现，0和1是一组，后面五个数是另外一组，我们先看2 – 7的：

```

1338: b8 00 00 00 00      mov    $0x0,%eax
133d: eb b5                jmp    12f4 <phase_3+0x62>
133f: b8 00 00 00 00      mov    $0x0,%eax
1344: eb b3                jmp    12f9 <phase_3+0x67>
1346: b8 00 00 00 00      mov    $0x0,%eax
134b: eb b1                jmp    12fe <phase_3+0x6c>
134d: b8 00 00 00 00      mov    $0x0,%eax
1352: eb af                jmp    1303 <phase_3+0x71>
1354: b8 00 00 00 00      mov    $0x0,%eax
1359: eb ad                jmp    1308 <phase_3+0x76>
135b: b8 00 00 00 00      mov    $0x0,%eax
1360: eb ab                jmp    130d <phase_3+0x7b>

```

我们发现，都是把%eax清零，然后跳回上面的某个位置，我们向上看：

```

12e3: b8 d5 01 00 00      mov    $0x1d5,%eax-
12e8: eb 05                jmp    12ef <phase_3+0x5d>
12ea: b8 00 00 00 00      mov    $0x0,%eax
12ef: 2d fa 01 00 00      sub    $0x1fa,%eax
12f4: 05 c3 01 00 00      add    $0x1c3,%eax
12f9: 2d 49 02 00 00      sub    $0x249,%eax
12fe: 05 49 02 00 00      add    $0x249,%eax
1303: 2d 49 02 00 00      sub    $0x249,%eax
1308: 05 49 02 00 00      add    $0x249,%eax
130d: 2d 49 02 00 00      sub    $0x249,%eax
1312: 83 3c 24 05      cmpl   $0x5,(%rsp)
1316: 7f 06                jg    131e <phase_3+0x8c>
1318: 39 44 24 04      cmp    %eax,0x4(%rsp)
131c: 74 05                je    1323 <phase_3+0x91>
131e: e8 b9 06 00 00      call   19dc <explode_bomb>

```

他们最终跳去的地方，也是0和1跳去的地方，尽管具体在哪一行不同，但都是对%eax进行了一些操作，然后和我们输入的第二个数比较。这里一个有趣的事是，他们最后都跳在1312前面，也就是说无论第一个数是什么，都要执行这句cmpl，这就导致了最后我们输入的第一个数必须小于等于5。最后我们只要看0 – 5分别执行了什么样的对%eax的运算（仅仅只是加加减减而已），就可以得到全部答案，如下：

第一个数	第二个数
0	-171
1	-640
2	-134
3	-585
4	0
5	-585

## 2.4 Phase 4

首先通过和先前类似的方法，明确需要输入两个整数。我们不妨还是输入1 2，值得注意的是，我们发现在这里，(%rsp)是我们输入的第二个数，观察反汇编如图：

```

0x5555554013d9 <phase_4+45>    mov    (%rsp),%eax
0x5555554013dc <phase_4+48>    sub    $0x2,%eax
> 0x5555554013df <phase_4+51>    cmp    $0x2,%eax
0x5555554013e2 <phase_4+54>    jbe    0x5555554013e9 <phase_4+61>
0x5555554013e4 <phase_4+56>    call   0x5555554019dc <explode_bomb>
0x5555554013e9 <phase_4+61>    mov    (%rsp),%esi
0x5555554013ec <phase_4+64>    mov    $0x7,%edi
0x5555554013f1 <phase_4+69>    call   0x555555401373 <func4>
0x5555554013f6 <phase_4+74>    cmp    %eax,0x4(%rsp)
0x5555554013fa <phase_4+78>    je    0x555555401401 <phase_4+85>
0x5555554013fc <phase_4+80>    call   0x5555554019dc <explode_bomb>
0x555555401401 <phase_4+85>    mov    0x8(%rsp),%rax

```

它要求第二个数只能是2,3,4中的一个，之后把7和第二个数作为参数传递给了`func4`，把返回值和第一个数（0x4(%rsp)）比较。我们再次发现，`func4`和第一个数没关系，只和第二个数有关，我们发现在我们的这次输入中，%eax是66，这说明和2匹配的数字是66，我们通过同样的方法得到了一组正确答案66 2。那么下面我们再来研究一下`func4`到底是怎么回事，反汇编如下：

```

00000000000001373 <func4>:
1373: b8 00 00 00 00      mov    $0x0,%eax
1378: 85 ff               test   %edi,%edi
137a: 7e 07               jle    1383 <func4+0x10>
137c: 89 f0               mov    %esi,%eax
137e: 83 ff 01             cmp    $0x1,%edi
1381: 75 02               jne    1385 <func4+0x12>
1383: f3 c3               repz   ret
1385: 41 54               push   %r12
1387: 55                  push   %rbp
1388: 53                  push   %rbx
1389: 41 89 f4             mov    %esi,%r12d
138c: 89 fb               mov    %edi,%ebx
138e: 8d 7f ff             lea    -0x1(%rdi),%edi
1391: e8 dd ff ff ff     call   1373 <func4>
1396: 42 8d 2c 20             lea    (%rax,%r12,1),%ebp
139a: 8d 7b fe             lea    -0x2(%rbx),%edi
139d: 44 89 e6             mov    %r12d,%esi
13a0: e8 ce ff ff ff     call   1373 <func4>
13a5: 01 e8               add    %ebp,%eax
13a7: 5b                  pop    %rbx
13a8: 5d                  pop    %rbp
13a9: 41 5c               pop    %r12
13ab: c3                  ret

```

`func4`是一个递归函数，一开始，%rdi是7，而%rsi是我们输入的第二个数。我们可以把这个函数总结如下：

$$func4(n, x) = \begin{cases} 0, n \leq 0 \\ x, n = 1 \\ func4(n - 1, x) + x + func4(n - 2, x), n > 1 \end{cases}$$

这是Fibonacci数列的一个变体，他把标准Fibonacci数列的每一项都×x。也就是说这个数列是{ $a_n$ } = x, x, 2x, 3x, 5x, 8x, ...，或者说

$$a_n = \begin{cases} x, n \leq 2 \\ a_{n-1} + a_{n-2}, n > 2 \end{cases}$$

而`func4`最后求的是前n项的和，用数学归纳法给出如下证明：

记 $a_n$ 的前 $n$ 项和为 $S_n$

归纳奠基:  $n = 1$ 时,  $func4(1, x) = x = S_1$

归纳递推: 假设 $n = k$ 时,  $func4(k, x) = S_k$ , then,  $func4(k + 1, x) = func4(k, x) + x + func4(k - 1, x) = S_k + S_{k-1} + x = (a_1 + a_2 + \dots + a_k) + (a_1 + a_2 + \dots + a_{k+1}) + x = (a_k + a_{k-1}) + (a_{k-1} + a_{k-2}) + \dots + (a_2 + a_1) + a_1 + x = a_{k+1} + a_k + \dots + a_2 + a_1 = S_{k+1}$ , 得证

回到这个题目,  $n$ 为7, 而 $x$ 为2或3或4, 我们给出所有可能答案如下:

第一个数	第二个数
66	2
99	3
132	4

## 2.5 Phase 5

观察反汇编代码, 我们再次看到了`strings_not_equal`, 说明我们需要输入字符串。`string_length`的返回值提示我们, 这个字符串应该有六个字符。

```
0x55555540141b <phase_5>    push  %rbx
0x55555540141c <phase_5+1>   sub   $0x10,%rsp
0x555555401420 <phase_5+5>   mov   %rdi,%rbx
0x555555401423 <phase_5+8>   mov   %fs:0x28,%rax
0x55555540142c <phase_5+17>  mov   %rax,0x8(%rsp)
0x555555401431 <phase_5+22>  xor   %eax,%eax
0x555555401433 <phase_5+24>  call  0x555555401704 <string_length>
0x555555401438 <phase_5+29>  cmp   $0x6,%eax
0x55555540143b <phase_5+32>  jne   0x555555401492 <phase_5+119>
0x55555540143d <phase_5+34>  mov   $0x0,%eax
0x555555401442 <phase_5+39>  lea   0x1647(%rip),%rcx      # 0x555555402a90 <array.3415>
0x555555401449 <phase_5+46>  movzbl (%rbx,%rax,1),%edx
0x55555540144d <phase_5+50>  and   $0xf,%edx
0x555555401450 <phase_5+53>  movzbl (%rcx,%rdx,1),%edx
0x555555401454 <phase_5+57>  mov   %dl,0x1(%rsp,%rax,1)
0x555555401458 <phase_5+61>  add   $0x1,%rax
0x55555540145c <phase_5+65>  cmp   $0x6,%rax
0x555555401460 <phase_5+69>  jne   0x555555401449 <phase_5+46>
0x555555401462 <phase_5+71>  movb  $0x0,0x7(%rsp)
0x555555401467 <phase_5+76>  lea   0x1(%rsp),%rdi
0x55555540146c <phase_5+81>  lea   0x15f3(%rip),%rsi      # 0x555555402a66
0x555555401473 <phase_5+88>  call  0x555555401721 <strings_not_equal>
0x555555401478 <phase_5+93>  test  %eax,%eax
0x55555540147a <phase_5+95>  jne   0x555555401499 <phase_5+126>
0x55555540147c <phase_5+97>  mov   0x8(%rsp),%rax
0x555555401481 <phase_5+102> xor   %fs:0x28,%rax
```

我们查看两个地址的字符串:

```
(gdb) x/s 0x555555402a90
0x555555402a90 <array.3415>:    "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
(gdb) x/s 0x555555402a66
0x555555402a66: "flyers"
```

我们发现, 我们需要的字符串应该是`flyers`, 然而并不像Phase 1那么直接。在调用`strings_not_equal`之前, 它对我们输入的字符串进行了一些处理, 我们需要研究清楚。观察可以发现, 这又是一个循环结构, 循环计数器是`%eax`, 每次循环+1直到6。而循环内操作如下: 第一步, 每次循环依次取出我们输入的一个字符, 从8位零扩展为32位后放入`%edx`, 然后和`0xf`取并, 也就是只保留最低四位的值; 第二步, 我们现在有了输入的一个字符的低四位二进制对应的值, 现在我们以这个值为索引, 去取先前展示的`0x555555402a90`处的字符串中的字符, 然后放到`0x1(%rsp)`开始的一段内存中的相应位置。之后将`0x7(%rsp)`设为0作为字符串结尾, 把`0x1(%rsp)`作为参数传给`strings_not_equal`。我们可以看一个具体的例子, 比如我们输入第一个字符为*i*, 它的ASCII码为`0x69`, 那我们的索引是9, 取得的字符为*f*。也就是说, 我们需要ASCII码低四位分别是`0x9, 0xf, 0xe, 0x5, 0x6, 0x7`的六个字符, 这样就可以依次取得`flyers`。显然答案也并不唯一, 一个可行的答案是`ionuvwxyz`。

## 2.6 Phase 6

我们再次看到了`read_six_numbers`, 答案是六个数字。这段汇编代码非常复杂, 我们用*ni*推进, 逐步看。我们先看这个片段:

```

14c7: e8 4c 05 00 00          call  1a18 <read_six_numbers>
14cc: 4d 89 ec                mov   %r13,%r12
14cf: 41 be 00 00 00 00        mov   $0x0,%r14d
14d5: eb 25                  jmp   14fc <phase_6+0x57>
14d7: e8 00 05 00 00          call  19dc <explode_bomb>
14dc: eb 2d                  jmp   150b <phase_6+0x66>
14de: 83 c3 01                add   $0x1,%ebx
14e1: 83 fb 05                cmp   $0x5,%ebx
14e4: 7f 12                  jg   14f8 <phase_6+0x53>
14e6: 48 63 c3                movslq %ebx,%rax
14e9: 8b 04 84                mov   (%rsp,%rax,4),%eax
14ec: 39 45 00                cmp   %eax,0x0(%rbp)
14ef: 75 ed                  jne   14de <phase_6+0x39>
14f1: e8 e6 04 00 00          call  19dc <explode_bomb>
14f6: eb e6                  jmp   14de <phase_6+0x39>
14f8: 49 83 c5 04          add   $0x4,%r13
14fc: 4c 89 ed                mov   %r13,%rbp
14ff: 41 8b 45 00          mov   0x0(%r13),%eax
1503: 83 e8 01                sub   $0x1,%eax
1506: 83 f8 05                cmp   $0x5,%eax
1509: 77 cc                  ja    14d7 <phase_6+0x32>
150b: 41 83 c6 01          add   $0x1,%r14d
150f: 41 83 fe 06          cmp   $0x6,%r14d
1513: 74 05                  je    151a <phase_6+0x75>
1515: 44 89 f3                mov   %r14d,%ebx
1518: eb cc                  jmp   14e6 <phase_6+0x41>

```

我们发现, 这个片段是和先前类似的循环操作, 它对我们输入的数字有两个要求: 首先, 每个数字必须是正整数且 $\leq 6$ (1503,1506,1509); 其次, 所有数字不相等, 实现方式是用第一个数和第二至六个数比较, 再用第二个数和第三至六个数比较, 以此类推( $14e6 - 14ef$ )。这两个要求告诉我们, 我们输入的数字是1 – 6的一个排列。继续往下看:

```

151a: 49 8d 4c 24 18          lea   0x18(%r12),%rcx
151f: ba 07 00 00 00          mov   $0x7,%edx
1524: 89 d0                  mov   %edx,%eax
1526: 41 2b 04 24          sub   (%r12),%eax
152a: 41 89 04 24          mov   %eax,(%r12)
152e: 49 83 c4 04          add   $0x4,%r12
1532: 4c 39 e1                cmp   %r12,%rcx
1535: 75 ed                  jne   1524 <phase_6+0x7f>

```

这一段的操作是把我们输入的每一个数*i*变成了 $7 - i$ 。继续向下看:

```

1537: be 00 00 00 00          mov   $0x0,%esi
153c: eb 1a                  jmp   1558 <phase_6+0xb3>
153e: 48 8b 52 08          mov   0x8(%rdx),%rdx
1542: 83 c0 01                add   $0x1,%eax
1545: 39 c8                  cmp   %ecx,%eax
1547: 75 f5                  jne   153e <phase_6+0x99>
1549: 48 89 54 f4 20          mov   %rdx,0x20(%rsp,%rsi,8)
154e: 48 83 c6 01          add   $0x1,%rsi
1552: 48 83 fe 06          cmp   $0x6,%rsi
1556: 74 16                  je    156e <phase_6+0xc9>
1558: 8b 0c b4                mov   (%rsp,%rsi,4),%ecx
155b: b8 01 00 00 00          mov   $0x1,%eax
1560: 48 8d 15 c9 2c 20 00      lea   0x202cc9(%rip),%rdx      # 204230 <node1>
1567: 83 f9 01                cmp   $0x1,%ecx
156a: 7f d2                  jg   153e <phase_6+0x99>
156c: eb db                  jmp   1549 <phase_6+0xa4>

```

我们发现, 这段代码在往`0x20(%rsp)`开始的一段内存里面写指针(1549), 指针由1560的`leaq`而来, 我们用*gdb*调试, 可以看到`node1`是`0x555555604230`, 一个`node`

里面有一个值，这个`node`的编号，以及下一个`node`的地址。`node1`到`node5`是在内存连续排列的，而`node6`在`0x55555604110`。而我们发现，它第一个写的指针就是`node < 我们输入的第一个数 >`的地址。

我们做一个小总结：如果我们输入`1 2 3 4 5 6`，程序会把我们的输入变成`6 5 4 3 2 1`，之后在`0x20(%rsp)`开始的一段内存里面写入`node6, node5, node4, node3, node2, node1`的地址。我们继续看：

```

156e: 48 8b 5c 24 20    mov    0x20(%rsp), %rbx
1573: 48 8b 44 24 28    mov    0x28(%rsp), %rax
1578: 48 89 43 08      mov    %rax, 0x8(%rbx)
157c: 48 8b 54 24 30    mov    0x30(%rsp), %rdx
1581: 48 89 50 08      mov    %rdx, 0x8(%rax)
1585: 48 8b 44 24 38    mov    0x38(%rsp), %rax
158a: 48 89 42 08      mov    %rax, 0x8(%rdx)
158e: 48 8b 54 24 40    mov    0x40(%rsp), %rdx
1593: 48 89 50 08      mov    %rdx, 0x8(%rax)
1597: 48 8b 44 24 48    mov    0x48(%rsp), %rax
159c: 48 89 42 08      mov    %rax, 0x8(%rdx)
15a0: 48 c7 40 08 00 00 00  movq   $0x0, 0x8(%rax)
15a7: 00
15a8: bd 05 00 00 00 00  mov    $0x5, %ebp
15ad: eb 09              jmp    15b8 <phase_6+0x113>
15af: 48 8b 5b 08      mov    0x8(%rbx), %rbx
15b3: 83 ed 01          sub    $0x1, %ebp
15b6: 74 11              je     15c9 <phase_6+0x124>
15b8: 48 8b 43 08      mov    0x8(%rbx), %rax
15bc: 8b 00              mov    (%rax), %eax
15be: 39 03              cmp    %eax, (%rbx)
15c0: 7d ed              jge    15af <phase_6+0x10a>
15c2: e8 15 04 00 00 00  call   19dc <explode_bomb>
15c7: eb e6              jmp    15af <phase_6+0x10a>
15c9: 48 8b 44 24 58    mov    0x58(%rsp), %rax

```

这段代码中，首先把先前在`0x20(%rsp)`开始的内存写的那些节点连成了一个新的链表（也就是对原链表按我们写的顺序对节点重新排列）。而之后遍历我们重新排列的链表，要求前一个节点的值大于后面一个节点的，否则爆炸。我们查看各个节点的值：

```

0x55555604110 <node6>: 0x00000217
(gdb) x/24x 0x55555604230
0x555555604230 <node1>: 0x0000028b
0x555555604240 <node2>: 0x0000012a
0x555555604250 <node3>: 0x00000033
0x555555604260 <node4>: 0x00000137
0x555555604270 <node5>: 0x000003d6

```

我们发现，降序排列为`5 1 6 4 2 3`。别忘了，我们输入的`i`会变成`7 - i`，我们最后的答案是`2 6 1 3 5 4`。

## 2.7 Secret Phase

`bomb.c`最后的注释：

```

/* Wow, they got it! But isn't something... missing? Perhaps
 * something they overlooked? Mua ha ha ha! */

```

我们可以在phase\_6后面看到secret\_phase，不妨在完整的汇编代码文件中检索secret\_phase，可以看到phase\_defused调用了secret\_phase。研究其汇编代码：

```

1bb6: e8 fd fc ff ff    call  18b8 <send_msg>
1bbb: 83 3d ea 2a 20 00 06  cmpl $0x6,0x202aea(%rip) # 2046ac <num_input_strings>
1bc2: 74 19             je   1bdd <phase_defused+0x40>
1bc4: 48 8b 44 24 68     mov  %r8(%rsp),%rax
1bc9: 64 48 33 04 25 28 00 xor  %fs:0x28,%rax
1bd0: 00 00
1bd2: 0f 85 84 00 00 00  jne   1c5c <phase_defused+0xbff>
1bd8: 48 83 c4 78       add  $0x78,%rsp
1bdc: c3                ret
1bdd: 48 8d 4c 24 0c     lea   0xc(%rsp),%rcx
1be2: 48 8d 54 24 08     lea   0x8(%rsp),%rdx
1be7: 4c 8d 44 24 10     lea   0x10(%rsp),%r8
1bec: 48 8d 35 0c 11 00 00 lea   0x110c(%rip),%rsi      # 2cff <array.3415+0x26f>
1bf3: 48 8d 3d b6 2b 20 00 lea   0x202bb6(%rip),%rdi      # 2047b0 <input_strings+0xf0>
1bfa: b8 00 00 00 00 00  mov  $0x0,%eax
1bff: e8 dc f2 ff ff    call  ee0 <__isoc99_sscanf@plt>
1c04: 83 f8 03           cmp  $0x3,%eax
1c07: 74 1a             je   1c23 <phase_defused+0x86>
1c09: 48 8d 3d b0 0f 00 00 lea   0xfb0(%rip),%rdi      # 2bc0 <array.3415+0x130>
1c10: e8 0b f2 ff ff    call  e20 <puts@plt>
1c15: 48 8d 3d d4 0f 00 00 lea   0xfd4(%rip),%rdi      # 2bf0 <array.3415+0x160>
1c1c: e8 ff f1 ff ff    call  e20 <puts@plt>
1c21: eb a1             jmp  1bc4 <phase_defused+0x27>
1c23: 48 8d 7c 24 10     lea   0x10(%rsp),%rdi
1c28: 48 8d 35 d9 10 00 00 lea   0x10d9(%rip),%rsi      # 2d08 <array.3415+0x278>
1c2f: e8 ed fa ff ff    call  1721 <strings_not_equal>
1c34: 85 c0             test $eax,%eax
1c36: 75 d1             jne  1c09 <phase_defused+0x6c>
1c38: 48 8d 3d 21 0f 00 00 lea   0xf21(%rip),%rdi      # 2b60 <array.3415+0xd0>
1c3f: e8 dc f1 ff ff    call  e20 <puts@plt>
1c44: 48 8d 3d 0f 00 00  lea   0xf3d(%rip),%rdi      # 2b88 <array.3415+0xf8>
1c4b: e8 d0 f1 ff ff    call  e20 <puts@plt>
1c50: b8 00 00 00 00 00  mov  $0x0,%eax
1c55: e8 d0 f9 ff ff    call  162a <secret_phase>
1c5a: eb ad             jmp  1c09 <phase_defused+0x6c>
1c5c: e8 df f1 ff ff    call  e40 < stack_chk_fail@plt>

```

我们又一次看到了strings\_not\_equal，观察前面一个地址的字符串：

```
(gdb) x/s 0x555555402d08
0x555555402d08: "DrEvil"
```

看来我们要在某个地方输入DrEvil，我们再向前看看：

```
(gdb) x/s 0x555555402cff
0x555555402cff: "%d %d %s"
```

再去看下一行的，我们发现是我们在phase\_4输入的答案。它进行了这样一个判断(1c04)，如果我们输入的只有两个元素，就正常进行，如果有三个元素，判断第三个元素是不是DrEvil，如果是，进入secret\_phase。

也就是说，我们需要在phase\_4的答案后面加上DrEvil来进入secret\_phase。

观察secret\_phase的反汇编，我们发现，它把我们输入的数和一个地址n1作为参数传递给了fun7，若返回值不为3就爆炸，也就是说我们要研究输入什么能让fun7返回3。

```

000000000000000162a <secret_phase:>
162a: 53                 push  %rbx
162b: e8 29 04 00 00     call  1a59 <read_line>
1630: ba 0a 00 00 00     mov   $0xa,%edx
1635: be 00 00 00 00     mov   $0x0,%esi
163a: 48 89 c7           mov   %rax,%rdi
163d: e8 7e f8 ff ff    call  e01 <strtoll@plt>
1642: 48 89 c3           mov   %rax,%rbx
1645: 8d 40 ff           lea   -0x1(%rax),%eax
1648: 3d e8 03 00 00     cmp   $0x3e8,%eax
164d: 77 2b               ja   167a <secret_phase+0x50>
164f: 89 de               mov   %ebx,%esi
1650: ba 00 00 00 00     mov   $0x0,%rdi      # 204150 <n1>
1651: 48 8d 3d f8 2a 20 00 lea   0x202af8(%rip),%rdi
1658: e8 8e ff ff ff    call  15eb <fun7>
165d: 83 f8 03           cmp   $0x3,%eax
1660: 74 05               je   1667 <secret_phase+0x3d>
1662: e8 75 03 00 00     call  19dc <explode_bomb>
1667: 48 8d 3d d2 13 00 00 lea   0x13d2(%rip),%rdi      # 2a40 <_IO_stdin_used+0x180>
166e: e8 ad f7 ff ff    call  e20 <puts@plt>
1673: e8 25 05 00 00     call  1b9d <phase_defused>
1678: 5b                 pop   %rbx
1679: c3                 ret
167a: e8 5d 03 00 00     call  19dc <explode_bomb>
167f: eb ce             jmp  164f <secret_phase+0x25>

```

我们来研究一下n1对应的内存中存放了什么：

```
0x555555604150 <n1>: 0x00000024 0x00000000 0x55604170 0x00005555
0x555555604160 <n1+16>: 0x55604190 0x00005555 0x00000000 0x00000000
0x555555604170 <n2>: 0x00000008 0x00000000 0x556041f0 0x00005555
0x555555604180 <n21+16>: 0x556041b0 0x00005555 0x00000000 0x00000000
0x555555604190 <n22>: 0x00000032 0x00000000 0x556041d0 0x00005555
0x5555556041a0 <n22+16>: 0x55604210 0x00005555 0x00000000 0x00000000
0x5555556041b0 <n32>: 0x00000016 0x00000000 0x556040b0 0x00005555
0x5555556041c0 <n32+16>: 0x55604070 0x00005555 0x00000000 0x00000000
0x5555556041d0 <n33>: 0x0000002d 0x00000000 0x55604010 0x00005555
0x5555556041e0 <n33+16>: 0x556040d0 0x00005555 0x00000000 0x00000000
0x5555556041f0 <n31>: 0x00000006 0x00000000 0x55604030 0x00005555
0x555555604200 <n31+16>: 0x55604090 0x00005555 0x00000000 0x00000000
0x555555604210 <n34>: 0x0000006b 0x00000000 0x55604050 0x00005555
```

我们发现，这是一个二叉树，一个节点里面是一个值，左子节点的地址和右子节点的地址。

我们继续去看fun7：

```
0000000000000015eb <fun7>:
15eb: 48 85 ff test %rdi,%rdi
15ee: 74 34 je 1624 <fun7+0x39>
15f0: 48 83 ec 08 sub $0x8,%rsp
15f4: 8b 17 mov (%rdi),%edx
15f6: 39 f2 cmp %esi,%edx
15f8: 7f 0e jg 1608 <fun7+0x1d>
15fa: b8 00 00 00 00 mov $0x0,%eax
15ff: 39 f2 cmp %esi,%edx
1601: 75 12 jne 1615 <fun7+0x2a>
1603: 48 83 c4 08 add $0x8,%rsp
1607: c3 ret
1608: 48 8b 7f 08 mov 0x8(%rdi),%rdi
160c: e8 da ff ff ff call 15eb <fun7>
1611: 01 c0 add %eax,%eax
1613: eb ee jmp 1603 <fun7+0x18>
1615: 48 8b 7f 10 mov 0x10(%rdi),%rdi
1619: e8 cd ff ff ff call 15eb <fun7>
161e: 8d 44 00 01 lea 0x1(%rax,%rax,1),%eax
1622: eb df jmp 1603 <fun7+0x18>
1624: b8 ff ff ff ff mov $0xffffffff,%eax
1629: c3 ret
```

fun7的两个参数分别是节点地址(%rdi)和我们输入的数字(%rsi)，且%rdi一开始为n1，它是一个递归函数，执行逻辑如下：

首先，如果地址是NULL，返回-1；

其次，如果地址对应节点的值等于我们输入的数字，返回0；

然后，如果地址对应节点的值小于我们输入的数字，返回fun7(%rdi->r,%rsi) × 2 + 1，在这里，我们用%rdi->r表示右子节点的地址；

否则，返回fun7(%rdi->l,%rsi) × 2，在这里，我们用%rdi->l表示左子节点的地址。

而我们要fun7返回3，那么需要返回 $1 \times 2 + 1$ ，也就是要递归返回1，就是要返回 $0 \times 2 + 1$ ，要递归返回0，是地址对应节点的值等于我们输入的数字的情况。也就是说，我们输入的这个数字，大于n1中的值，大于n1的右子节点中的值，等于n1的右子节点的右子节点中的值。我们观察发现，这三个节点的值分别是0x24 0x32 0x6b，我们要输入的数和最后一个相等，也就是107，如此就得到了最终的答案。