

## 实验总结：

总体来说还是挺好玩的，教会了我许多gdb的操作。但是 phase\_6 太多循环了。做的心烦，没做好。日后再重做一遍吧。

最后的 phase\_6 与 secret\_phase 由于涉及到数据结构，所以做的效果都不是很理想，都参考了答案。

## 实验任务：

This is an x86-64 bomb for self-study students.

bomb 每个 phase 都等待一个输入。我们需要利用 gdb 等调试工具，分析 bomb 的反汇编代码，得到一个合理的输入。

## 实验思路：

本实验主要考察 gdb 的使用，与对于 x86 的基本汇编代码语义的理解。在本实验的前几个小时，应该都是在与gdb搏斗。

如果很熟练 gdb，哪怕不理解本文的汇编代码，应该也可以完成lab。

## 实验内容：

### phase\_1

```
1 (gdb) disassemble phase_1
2 Dump of assembler code for function phase_1:
3 => 0x0000000000400ee0 <+0>:      sub    $0x8,%rsp
4     0x0000000000400ee4 <+4>:      mov    $0x402400,%esi
5     0x0000000000400ee9 <+9>:      callq 0x401338 <strings_not_equal>
6     0x0000000000400eee <+14>:     test   %eax,%eax
7     0x0000000000400ef0 <+16>:     je     0x400ef7 <phase_1+23>
8     0x0000000000400ef2 <+18>:     callq 0x40143a <explode_bomb>
9     0x0000000000400ef7 <+23>:     add    $0x8,%rsp
10    0x0000000000400efb <+27>:     retq
11 End of assembler dump.
```

#### phase\_1的汇编代码做了什么呢？

首先，将一个立即数 0x402400 放入寄存器 %esi。之后调用一个函数 strings\_not\_equal()，看名字发现这个函数可能是比较两个字符串是否相等。

如果 test %eax,%eax 的结果是一个1，就满足了 je 指令的跳转条件，可以跳过 explode\_bomb，到 phase\_1+23，进而我们解决了这个phase。

以上如果对于 je test 的语义不清楚，需要自行查阅课本。

#### 我们的输入去哪里了？

string\_not\_equal,很明显需要两个参数，我们不可能拿一个字符串跟自己比较。

根据x86的默认规则，%esi 在函数调用时会充当第二个参数。我们可以猜测，第一个参数是我们的输入。

根据x86的规定，第一个参数使用的寄存器是 `%edi`，我们可以在GDB中验证下。这个规律对其他所有 `phase` 都适用，即我们的输入被放在 `%edi` 中。

```
(gdb) info registers $edi
edi                0x603780                6305664
(gdb) x/s 0x603780
0x603780 <input_strings>:                "hello,world!"
(gdb) |
```

我们猜对了！。那么一个很自然的想法就是，`0x40200` 是要与我们输入进行比较的那个字符串的地址。找到它，我们就解决了第一个 `phase`。

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

```
1 | Border relations with Canada have never been better.
```

本文提及到的gdb指令，请自行阅读课本，与善用搜索引擎。你做bomb lab前几个小时都会与它搏斗。

## phase\_2

首先，照例随便输入一个值，来让程序运行到断点处。

```
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
hello,world!

Breakpoint 2, 0x0000000000400efc in phase_2 ()
(gdb) |
```

之后查看 `phase_2` 的反汇编代码。

```
1 (gdb) disassemble phase_2
2 Dump of assembler code for function phase_2:
3 => 0x0000000000400efc <+0>:      push    %rbp
4   0x0000000000400efd <+1>:      push    %rbx
5   # caller saves.
6
7   0x0000000000400efe <+2>:      sub     $0x28,%rsp
8   0x0000000000400f02 <+6>:      mov     %rsp,%rsi
9   0x0000000000400f05 <+9>:      callq  0x40145c <read_six_numbers>
10  # get parameters. %rsi = %rsp %rdi = my input
11  # call arr = read_six_numbers(%rdi,%rsi).
12
13  0x0000000000400f0a <+14>:     cmpb    $0x1,(%rsp)
14
15  # if arr[0] == 1 then jump to +52
16  0x0000000000400f0e <+18>:     je      0x400f30 <phase_2+52>
```

```

17      0x0000000000400f10 <+20>:    callq  0x40143a <explode_bomb>
18      0x0000000000400f15 <+25>:    jmp     0x400f30 <phase_2+52>
19
20
21      0x0000000000400f17 <+27>:    mov     -0x4(%rbx),%eax # eax = arr[0]
22      # eax = arr[1]
23      0x0000000000400f1a <+30>:    add     %eax,%eax # eax *= 2
24      0x0000000000400f1c <+32>:    cmp     %eax,(%rbx) # 2 * arr[0] == arr[1]
first loop
25                                     # 2 * arr[1] == arr[2]
second loop
26      # so arr[1] = 2 => 1 2 4 8 16 32
27      0x0000000000400f1e <+34>:    je      0x400f25 <phase_2+41>
28      0x0000000000400f20 <+36>:    callq   0x40143a <explode_bomb>
29
30      0x0000000000400f25 <+41>:    add     $0x4,%rbx # rbx = arr[2]
31      0x0000000000400f29 <+45>:    cmp     %rbp,%rbx # rbx is last element?
32      0x0000000000400f2c <+48>:    jne     0x400f17 <phase_2+27>
33      0x0000000000400f2e <+50>:    jmp     0x400f3c <phase_2+64>
34
35      0x0000000000400f30 <+52>:    lea     0x4(%rsp),%rbx # rbx = rsp + 4
(arr[1]) (second element)
36      0x0000000000400f35 <+57>:    lea     0x18(%rsp),%rbp # rbp = rsp + 24
(arr[5])(last element)
37      0x0000000000400f3a <+62>:    jmp     0x400f17 <phase_2+27>
38
39      # end , if we pass ph2.
40      0x0000000000400f3c <+64>:    add     $0x28,%rsp
41      0x0000000000400f40 <+68>:    pop     %rbx
42      0x0000000000400f41 <+69>:    pop     %rbp
43      0x0000000000400f42 <+70>:    retq
44      End of assembler dump.
45      (gdb)

```

```

(gdb) r
Starting program: /home/sugar/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32

Breakpoint 1, 0x0000000000400efc in phase_2 ()
(gdb) c
Continuing.
That's number 2. Keep going!

```

成功过关。

写下这个函数的c形式。似乎是一个不错的想法？留待以后再说吧。

## phase\_3

照理随便输入一个值，之后查看反汇编代码。

```

1 | Dump of assembler code for function phase_3:
2 | => 0x000000000400f43 <+0>:      sub    $0x18,%rsp
3 |    0x000000000400f47 <+4>:      lea     0xc(%rsp),%rcx
4 |    0x000000000400f4c <+9>:      lea     0x8(%rsp),%rdx

```

```

1 | # 分配了下栈空间，同时将两个地址放入%rcx,%rdx。目前看不出这两个地址的作用，先放着
2 |
3 |    0x000000000400f51 <+14>:     mov     $0x4025cf,%esi
4 |    0x000000000400f56 <+19>:     mov     $0x0,%eax
5 |    #一路ni 到这里，查看下 这个地址的值

```

```

(gdb) x/s 0x4025cf
0x4025cf:      "%d %d"
(gdb) |

```

```

1 |    0x000000000400f5b <+24>:     callq  0x400bf0 <__isoc99_sscanf@plt>

```

百度下这个库函数的原型：

```

1 | int sscanf (char *str, char * format [, argument, ...]);

```

很明显 第一个参数是我们的输入，第二个参数是 %d %d，至此，我们知道了我们这个题需要输入两个整数。

```

1 |    0x000000000400f60 <+29>:     cmp     $0x1,%eax
2 |    0x000000000400f63 <+32>:     jg      0x400f6a <phase_3+39>
3 |    0x000000000400f65 <+34>:     callq  0x40143a <explode_bomb>
4 |    # %eax > 1

```

返回值要大于1，sscanf的返回值是什么呢？

sscanf 的返回值是成功转换的个数。

```

1 |    0x000000000400f6a <+39>:     cmpl    $0x7,0x8(%rsp)
2 |    0x000000000400f6f <+44>:     ja      0x400fad <phase_3+106>

```

这里+106 跳到了一个bomb，这告诉我们 rsp+8地址处的值要小于等于7。通过gdb，我们知道这个地方的值是我们的第一个输入。

我随便输入的值是 6 12

```

(gdb) x/x $rsp + 8
0x7fffffffdf88: 0x06
(gdb) |

```

```

1 | 0x000000000400f71 <+46>:    mov     0x8(%rsp),%eax
2 | 0x000000000400f75 <+50>:    jmpq    *0x402470(,%rax,8)
3 | # *运算符的作用类似于解引用，意思是实际上要跳转的地址在0x402470+offset处。

```

后面是一个跳转表(switch case)，如果你输入6 那么就跳转到对应的6号位。

```

(gdb) x/x 0x402470
0x402470:      0x00400f7c
(gdb) x/x 0x402470 + 8
0x402478:      0x00400fb9
(gdb) x/x 0x402470 + 16
0x402480:      0x00400f83
(gdb) x/x 0x402470 + 24
0x402488:      0x00400f8a
(gdb) x/x 0x402470 + 32
0x402490:      0x00400f91
(gdb) x/x 0x402470 + 40
0x402498:      0x00400f98
(gdb) x/x 0x402470 + 48
0x4024a0:      0x00400f9f
(gdb) x/x 0x402470 + 56
0x4024a8:      0x00400fa6
(gdb) |

```

```

1 | # case 0:
2 | 0x000000000400f7c <+57>:    mov     $0xcf,%eax
3 | 0x000000000400f81 <+62>:    jmp     0x400fbe <phase_3+123>
4 | # case 1:
5 | 0x000000000400f83 <+64>:    mov     $0x2c3,%eax
6 | 0x000000000400f88 <+69>:    jmp     0x400fbe <phase_3+123>
7 | # case 2:
8 | 0x000000000400f8a <+71>:    mov     $0x100,%eax
9 | 0x000000000400f8f <+76>:    jmp     0x400fbe <phase_3+123>
10 | # case 3:
11 | 0x000000000400f91 <+78>:    mov     $0x185,%eax
12 | 0x000000000400f96 <+83>:    jmp     0x400fbe <phase_3+123>
13 | # case 4:
14 | 0x000000000400f98 <+85>:    mov     $0xce,%eax
15 | 0x000000000400f9d <+90>:    jmp     0x400fbe <phase_3+123>
16 | # case 5:
17 | 0x000000000400f9f <+92>:    mov     $0x2aa,%eax
18 | 0x000000000400fa4 <+97>:    jmp     0x400fbe <phase_3+123>
19 | # case 6:
20 | 0x000000000400fa6 <+99>:    mov     $0x147,%eax

```

```

21      0x0000000000400fab <+104>:  jmp     0x400fbe <phase_3+123>
22
23      0x0000000000400fad <+106>:  callq   0x40143a <explode_bomb>
24  # case 7:
25      0x0000000000400fb2 <+111>:  mov     $0x0,%eax
26      0x0000000000400fb7 <+116>:  jmp     0x400fbe <phase_3+123>

```

最后，比对你的第二个输入是否与跳转表对应表项相同。如果相同的话，我们就通过了这个phase。

```

1      0x0000000000400fb9 <+118>:  mov     $0x137,%eax
2      0x0000000000400fbe <+123>:  cmp     0xc(%rsp),%eax
3      0x0000000000400fc2 <+127>:  je      0x400fc9 <phase_3+134>
4      0x0000000000400fc4 <+129>:  callq   0x40143a <explode_bomb>
5      0x0000000000400fc9 <+134>:  add     $0x18,%rsp
6      0x0000000000400fcd <+138>:  retq
7  End of assembler dump.

```

```

--Type <RET> for more, q to quit, c to continue without paging--c
0x0000000000400fb7 <+116>:  jmp     0x400fbe <phase_3+123>
0x0000000000400fb9 <+118>:  mov     $0x137,%eax
=> 0x0000000000400fbe <+123>:  cmp     0xc(%rsp),%eax
0x0000000000400fc2 <+127>:  je      0x400fc9 <phase_3+134>
0x0000000000400fc4 <+129>:  callq   0x40143a <explode_bomb>
0x0000000000400fc9 <+134>:  add     $0x18,%rsp
0x0000000000400fcd <+138>:  retq
End of assembler dump.

```

通过GDB，当程序执行到这里的时候，发现rsp + 0xc的值是我们的第二个输入，而eax的值是由上述的case语句决定的。所以我们的第二个输入只需要跟对应表项一致即可。

即我第一个输入为6的话，那么第二个输入就要是682。

```

(gdb) info registers eax
eax                0x2aa                682
(gdb) x/x $rsp+12
0x7fffffffdf8c: 0x0c
(gdb) x/d $rsp+12
0x7fffffffdf8c: 12
(gdb) |

```

## phase\_4

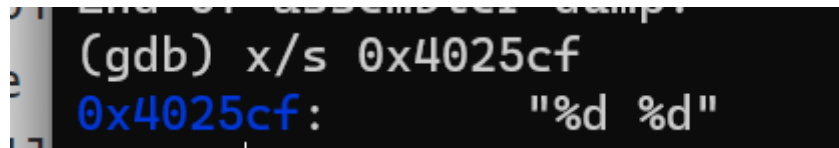
首先，在phase\_4处断点，之后我们随便输入点东西，让我们可以停在ph4的位置。然后查看ph4的反汇编代码。

```

1  Dump of assembler code for function phase_4:
2  => 0x000000000040100c <+0>:      sub     $0x18,%rsp
3      0x0000000000401010 <+4>:      lea     0xc(%rsp),%rcx
4      0x0000000000401015 <+9>:      lea     0x8(%rsp),%rdx
5      0x000000000040101a <+14>:     mov     $0x4025cf,%esi
6      0x000000000040101f <+19>:     mov     $0x0,%eax
7      0x0000000000401024 <+24>:     callq   0x400bf0 <__isoc99_sscanf@plt>
8      0x0000000000401029 <+29>:     cmp     $0x2,%eax
9      0x000000000040102c <+32>:     jne     0x401035 <phase_4+41>

```

做完上一个ph，这段代码就很熟悉了。照例我们看下 0x4025cf 处的值。



```

(gdb) x/s 0x4025cf
0x4025cf:      "%d %d"

```

与第8，9行的内容做印证，可知本题让我们输入两个整数。暂且输入 7 77吧。

```

1      0x000000000040102e <+34>:     cmpl     $0xe,0x8(%rsp)
2      0x0000000000401033 <+39>:     jbe     0x40103a <phase_4+46>
3      0x0000000000401035 <+41>:     callq   0x40143a <explode_bomb>

```

这段告诉我们，我们的第一个输入要小于等于14。

```

1      0x000000000040103a <+46>:     mov     $0xe,%edx
2      0x000000000040103f <+51>:     mov     $0x0,%esi
3      0x0000000000401044 <+56>:     mov     0x8(%rsp),%edi
4      0x0000000000401048 <+60>:     callq   0x400fce <func4>

```

这里给 func 准备了三个参数，分别是我们的第一个输入，0和14。我们先不去看 func4()，继续往下看。

```

1      0x000000000040104d <+65>:     test     %eax,%eax
2      0x000000000040104f <+67>:     jne     0x401058 <phase_4+76>
3      0x0000000000401058 <+76>:     callq   0x40143a <explode_bomb>

```

这段大家应该很熟悉了，跟ph1一样，这告诉我们 func4 的返回值必须是0。

```

1      0x0000000000401051 <+69>:     cmpl     $0x0,0xc(%rsp)
2      0x0000000000401056 <+74>:     je      0x40105d <phase_4+81>
3      0x0000000000401058 <+76>:     callq   0x40143a <explode_bomb>
4      0x000000000040105d <+81>:     add     $0x18,%rsp
5      0x0000000000401061 <+85>:     retq
6  End of assembler dump.

```

最后这段要求我们第二个输入必须是0。

```
(gdb) x/d $rsp + 8
0x7fffffffdf78: 7
(gdb) x/d $rsp + 0xc
0x7fffffffdf7c: 77
(gdb) |
```

现在我们知道四件事。

- 我们需要两个输入。
- 第二个输入是0。
- 第一个输入必须小于等于14。
- `func4()` 的返回值是0。

由于 `func4()` 的返回值是已知的，去研究它或许可以帮助我们确定第一个输入的具体值，进而解决这个 ph。

于是我们在 `func4` 设置断点，并且运行到这里。查看其反汇编代码。

现在我们要记住：

```
1 func4(
2     arg1_myinput1(%edi) = 7,
3     arg2(%esi) = 0,
4     arg3(%edx) = 14
5 )
```

对于以下这段....我速通了，因为我运气好，选择的输入是7。

```
1 Dump of assembler code for function func4:
2 0x000000000400fce <+0>: sub $0x8,%rsp
3 0x000000000400fd2 <+4>: mov %edx,%eax
4 0x000000000400fd4 <+6>: sub %esi,%eax
5 0x000000000400fd6 <+8>: mov %eax,%ecx
6 0x000000000400fd8 <+10>: shr $0x1f,%ecx
7 0x000000000400fdb <+13>: add %ecx,%eax
8 0x000000000400fdd <+15>: sar %eax
9 0x000000000400fdf <+17>: lea (%rax,%rsi,1),%ecx
```

以上这段，请善用 `ni` 与 `info registers`。

```
1 0x000000000400fe2 <+20>: cmp %edi,%ecx
2 0x000000000400fe4 <+22>: jle 0x400ff2 <func4+36>
```

这段是说，如果我们的第一个输入，**小于等于7**的话，就跳转到+36的位置。我们的输入是7，也就是直接跳转。

```
1 0x000000000400fe6 <+24>: lea -0x1(%rcx),%edx
2 0x000000000400fe9 <+27>: callq 0x400fce <func4>
3 0x000000000400fee <+32>: add %eax,%eax
4 0x000000000400ff0 <+34>: jmp 0x401007 <func4+57>
```



这段先不看了，直接看跳转后的部分了。

```
1 0x0000000000400ff2 <+36>: mov    $0x0,%eax
2 0x0000000000400ff7 <+41>: cmp    %edi,%ecx
3 0x0000000000400ff9 <+43>: jge    0x401007 <func4+57>
4 0x0000000000400ffb <+45>: lea    0x1(%rcx),%esi
5 0x0000000000400ffe <+48>: callq  0x400fce <func4>
6 0x0000000000401003 <+53>: lea    0x1(%rax,%rax,1),%eax
7 0x0000000000401007 <+57>: add    $0x8,%rsp
8 0x000000000040100b <+61>: retq
9 End of assembler dump.
```

在跳转之后，我们将返回值设置为0(注意，这正是我们需要的!!!)。并且比较我们的输入与ecx的值，如果我们的输入大于等于7，则func4结束了。满足既大于等于7，又小于等于7的数字只有一个，那就是7本身。

我们也得到了我们phase\_4的输入: 7 与 0。

实际上本题答案应该还有 3与0, 1与0, 0与0, 四个个答案的ecx分别是  $14/2$ ,  $7/2$ ,  $3/2$ ,  $1/2$ 。这样的ecx会让我们的输入既大于等于又小于等于，得到唯一解，进而让最后的eax等于0。感兴趣可以分析下。

我写了一个伪代码描述，也许对理解有帮助。

```
1 def func4(arg1,begin = 0,end = 14):
2     '''
3         arg1 = my_input:%rdi must <= 14
4         begin : %rsi
5         end   : %rdx
6         mid   : %rcx
7     '''
8     result = end - begin
9
10    mid = result >> 31
11    result += mid
12    result /= 2
13    mid = result + begin
14
15    if arg1 <= mid:
16        result = 0
17        if arg1 >= mid:
18            return result
19        else:
20            begin = mid + 1
21            result = func4(arg1,begin,end)
22            return result * 2 + 1
23    else:
24        end = mid - 1
25        result = func4(arg1,begin,end)
26        return result * 2
```

## phase\_5

```

1 (gdb) disassemble phase_5
2 Dump of assembler code for function phase_5:
3 => 0x000000000401062 <+0>:      push    %rbx
4     0x000000000401063 <+1>:      sub     $0x20,%rsp
5     0x000000000401067 <+5>:      mov     %rdi,%rbx
6     # rbx -> myinput
7     0x00000000040106a <+8>:      mov     %fs:0x28,%rax
8     0x000000000401073 <+17>:     mov     %rax,0x18(%rsp)
9     0x000000000401078 <+22>:     xor     %eax,%eax

```

以上三条指令，前两条是在验证是否有缓冲区溢出，在3.10的金丝雀值中提到过。`xor reg, reg` 是一种将某寄存器快速置零的方法，这种方式比 `mov $0, reg` 更快一点。这个也是课本上的题目。

```

1     0x00000000040107a <+24>:     callq   0x40131b <string_length>
2     0x00000000040107f <+29>:     cmp     $0x6,%eax
3     0x000000000401082 <+32>:     je      0x4010d2 <phase_5+112>
4     0x000000000401084 <+34>:     callq   0x40143a <explode_bomb>
5     0x000000000401089 <+39>:     jmp     0x4010d2 <phase_5+112>

```

这段似乎在说，我们的输入需要是一个字符串，且这个字符串长度为6。

```

1     0x0000000004010d2 <+112>:    mov     $0x0,%eax
2     0x0000000004010d7 <+117>:    jmp     0x40108b <phase_5+41>

```

为了文章的可读性，我将112处的代码复制到了上方。那么接下来的执行流是从+41处开始。

```

1     0x00000000040108b <+41>:     movzbl  (%rbx,%rax,1),%ecx
2     0x00000000040108f <+45>:     mov     %c1, (%rsp)

```

这两条指令，由于 `rbx` 是我们的输入，即一个字符串，这两条指令是将字符串中第一个字符取出，并且放在栈上。

```

1     0x000000000401092 <+48>:     mov     (%rsp),%rdx
2     0x000000000401096 <+52>:     and     $0xf,%edx

```

将我们的第一个字符，暂称 `c1`，让 `c1 & 0xF`，即让 `C1` 的范围在4bit之内。

```

1     0x000000000401099 <+55>:     movzbl  0x4024b0(%rdx),%edx

```

```

0x4024b0 <array.3449>: "maduiersnfotvbylSo you think you can stop the bomb with
ctrl-c, do you?"

```

试着打印了下，发现是个莫名其妙的句子。这条指令会让寄存器 `d1` 变成一个字母，这个字母取决于 `C1` 的值。

```

1     0x0000000004010a0 <+62>:     mov     %d1,0x10(%rsp,%rax,1)

```

`rax` 此时等于0，这句话表明我们将 `d1` 的值放在栈上。假设这个值叫 `D1` 吧。

```

1     0x0000000004010a4 <+66>:     add     $0x1,%rax
2     0x0000000004010a8 <+70>:     cmp     $0x6,%rax
3     0x0000000004010ac <+74>:     jne     0x40108b <phase_5+41>

```

通过这句我们看出，这是个循环，我们将我们的字符串中的字符 c1c2c3c4c5c6 变成了 D1D2D3D4D5D6。

```
1 0x0000000004010ae <+76>: movb $0x0,0x16(%rsp)
```

这个是在字符D1....D6的末尾添加0，使其变为一个字符串。

```
1 0x0000000004010b3 <+81>: mov $0x40245e,%esi
2 0x0000000004010b8 <+86>: lea 0x10(%rsp),%rdi
```

```
(gdb) x/s 0x40245e
0x40245e: "flyers"
```

```
1 0x0000000004010bd <+91>: callq 0x401338 <strings_not_equal>
2 0x0000000004010c2 <+96>: test %eax,%eax
3 0x0000000004010c4 <+98>: je 0x4010d9 <phase_5+119>
```

这段都是老朋友了，可以说照抄了phase1。也就是说，我们需要我们的输入&0xF 被一个映射一个神秘句子上，会得到一个新的字符串。这个字符串需要与 `flyers` 相同。现在照着那个神秘句子对照一下。我们就可以知道我们的输入是什么了。

```
(gdb) x/s $rdi
0x7fffffffdf70: "aduiar"
(gdb) x/s $rsi
0x40245e: "flyers"
```

输入 `ione fg` 与 `IONEFG` 或者大小写混杂都可以，因为他们的低四位都是相同的。至于其他的非字母表示，也可以查询ascii码表去获取。只需要 `string[C&0xF] == 'flyers'` C是我们的输入的字符。

```
1 0x0000000004010c6 <+100>: callq 0x40143a <explode_bomb>
2 0x0000000004010cb <+105>: nopl 0x0(%rax,%rax,1)
3 0x0000000004010d0 <+110>: jmp 0x4010d9 <phase_5+119>
4 0x0000000004010d2 <+112>: mov $0x0,%eax
5 0x0000000004010d7 <+117>: jmp 0x40108b <phase_5+41>
6 0x0000000004010d9 <+119>: mov 0x18(%rsp),%rax
7 0x0000000004010de <+124>: xor %fs:0x28,%rax
8 0x0000000004010e7 <+133>: je 0x4010ee <phase_5+140>
9 0x0000000004010e9 <+135>: callq 0x400b30 <__stack_chk_fail@plt>
10 0x0000000004010ee <+140>: add $0x20,%rsp
11 0x0000000004010f2 <+144>: pop %rbx
12 0x0000000004010f3 <+145>: retq
13 End of assembler dump.
14 (gdb)
```

最后是处理金丝雀值部分，先不研究了。

## phase\_6

```

1 (gdb) disassemble phase_6
2 Dump of assembler code for function phase_6:
3     0x0000000004010f4 <+0>:      push    %r14
4     0x0000000004010f6 <+2>:      push    %r13
5     0x0000000004010f8 <+4>:      push    %r12
6     0x0000000004010fa <+6>:      push    %rbp
7 => 0x0000000004010fb <+7>:      push    %rbx
8     0x0000000004010fc <+8>:      sub     $0x50,%rsp

```

保存寄存器，并且分配栈空间，对我们理解程序没啥帮助。

```

1     0x000000000401100 <+12>:     mov     %rsp,%r13
2     0x000000000401103 <+15>:     mov     %rsp,%rsi
3     0x000000000401106 <+18>:     callq  0x40145c <read_six_numbers>
4     0x00000000040110b <+23>:     mov     %rsp,%r14
5 # r13,r14,rsi,rbp,rsp 五个寄存器保存了相同的值。
6     0x00000000040110e <+26>:     mov     $0x0,%r12d
7 # 将%r12d置零
8     0x000000000401114 <+32>:     mov     %r13,%rbp
9     0x000000000401117 <+35>:     mov     0x0(%r13),%eax
10 # eax中保存了6个数字中的第一个。
11    0x00000000040111b <+39>:     sub     $0x1,%eax
12    0x00000000040111e <+42>:     cmp     $0x5,%eax
13    0x000000000401121 <+45>:     jbe     0x401128 <phase_6+52> # eax <= 5 jump
14    0x000000000401123 <+47>:     callq  0x40143a <explode_bomb>
15    0x000000000401128 <+52>:     add     $0x1,%r12d
16    0x00000000040112c <+56>:     cmp     $0x6,%r12d
17 # 循环6次，也就是所有的元素都要小于等于6。
18    0x000000000401130 <+60>:     je      0x401153 <phase_6+95>
19    0x000000000401132 <+62>:     mov     %r12d,%ebx
20    0x000000000401135 <+65>:     movslq  %ebx,%rax
21    0x000000000401138 <+68>:     mov     (%rsp,%rax,4),%eax
22    0x00000000040113b <+71>:     cmp     %eax,0x0(%rbp)
23    0x00000000040113e <+74>:     jne     0x401145 <phase_6+81>
24    0x000000000401140 <+76>:     callq  0x40143a <explode_bomb>
25    0x000000000401145 <+81>:     add     $0x1,%ebx
26    0x000000000401148 <+84>:     cmp     $0x5,%ebx
27    0x00000000040114b <+87>:     jle     0x401135 <phase_6+65>
28    0x00000000040114d <+89>:     add     $0x4,%r13
29    0x000000000401151 <+93>:     jmp     0x401114 <phase_6+32>
30 # 这是内层的循环。
31 # 两个循环的结果是：所有元素的值互不相同，且均小于等于6。

```

这段代码表示，所有元素的值要小于等于6，且互不相同。

```

1     0x000000000401153 <+95>:     lea     0x18(%rsp),%rsi
2 # rsi 是 第六个数字。
3     0x000000000401158 <+100>:    mov     %r14,%rax
4 # rax 指向第六个元素。
5     0x00000000040115b <+103>:    mov     $0x7,%ecx
6 # ecx = 7
7     0x000000000401160 <+108>:    mov     %ecx,%edx
8 # edx = 7
9     0x000000000401162 <+110>:    sub     (%rax),%edx
10 # edx = 7 - num6
11    0x000000000401164 <+112>:    mov     %edx,(%rax)

```

```

12 # num6 = 7 - num6
13 0x0000000000401166 <+114>: add    $0x4,%rax
14 # rax指向第五个元素。
15 0x000000000040116a <+118>: cmp    %rsi,%rax
16 0x000000000040116d <+121>: jne    0x401160 <phase_6+108>
17 # 又是一个循环
18 # for (int i = 0; i < 7; i++)
19 #     num[i] = 7 - num[i]
20 # 实际上也就是反转了下6个值

```

```

1 0x000000000040116f <+123>: mov    $0x0,%esi
2 0x0000000000401174 <+128>: jmp    0x401197 <phase_6+163>
3
4 # LOOP:
5 0x0000000000401176 <+130>: mov    0x8(%rdx),%rdx
6 0x000000000040117a <+134>: add    $0x1,%eax
7 0x000000000040117d <+137>: cmp    %ecx,%eax
8 0x000000000040117f <+139>: jne    0x401176 <phase_6+130>
9 # 使eax = ecx
10 0x0000000000401181 <+141>: jmp    0x401188 <phase_6+148>
11 0x0000000000401183 <+143>: mov    $0x6032d0,%edx
12
13 0x0000000000401188 <+148>: mov    %rdx,0x20(%rsp,%rsi,2)
14 0x000000000040118d <+153>: add    $0x4,%rsi
15 0x0000000000401191 <+157>: cmp    $0x18,%rsi
16 0x0000000000401195 <+161>: je     0x4011ab <phase_6+183>
17 # 将每个节点的地址都放在栈上，顺序目前还看不出来。
18
19
20 # 跳转到此处，
21 0x0000000000401197 <+163>: mov    (%rsp,%rsi,1),%ecx
22 0x000000000040119a <+166>: cmp    $0x1,%ecx
23 0x000000000040119d <+169>: jle    0x401183 <phase_6+143>
24
25 0x000000000040119f <+171>: mov    $0x1,%eax
26 0x00000000004011a4 <+176>: mov    $0x6032d0,%edx
27 0x00000000004011a9 <+181>: jmp    0x401176 <phase_6+130>
28 # 将edx设置为0x6032d0。
29 # LOOP END

```

由于 `edx` 被设置成了一个神秘的立即数，我们猜它可能是某个地址。查看了下从 `0x6032d0` 处开始的地址，我们发现他们构成了一个链表结构。然后 `node1` 的 `val` 被设置成 1，`next` 指向了 `node2`。

我偷看答案了，这个 `x/32xw` 是展示从 `0x6032d0` 开始的 32 个 word 的内容。

```

(gdb) x/32xw 0x6032d0
0x6032d0 <node1>: 0x0000014c 0x00000001 0x006032e0 0x00000000
0x6032e0 <node2>: 0x000000a8 0x00000002 0x006032f0 0x00000000
0x6032f0 <node3>: 0x0000039c 0x00000003 0x00603300 0x00000000
0x603300 <node4>: 0x000002b3 0x00000004 0x00603310 0x00000000
0x603310 <node5>: 0x000001dd 0x00000005 0x00603320 0x00000000
0x603320 <node6>: 0x000001bb 0x00000006 0x00000000 0x00000000
0x603330: 0x00000000 0x00000000 0x00000000 0x00000000

```

```

1 struct node{
2     int val;
3     int next;
4 }
5 // 0x6032d0是链表的开始。

```

上面的代码，最终会将链表所有节点的地址按照某种顺序排放在栈上，之后跳转到此处。如果我们能确定排放的顺序，我们就可以知道phase\_6的输入了。

```

1 0x0000000004011ab <+183>: mov 0x20(%rsp),%rbx
2 0x0000000004011b0 <+188>: lea 0x28(%rsp),%rax
3 0x0000000004011b5 <+193>: lea 0x50(%rsp),%rsi
4 0x0000000004011ba <+198>: mov %rbx,%rcx
5 0x0000000004011bd <+201>: mov (%rax),%rdx
6 0x0000000004011c0 <+204>: mov %rdx,0x8(%rcx)
7 0x0000000004011c4 <+208>: add $0x8,%rax
8 0x0000000004011c8 <+212>: cmp %rsi,%rax
9 0x0000000004011cb <+215>: je 0x4011d2 <phase_6+222>
10 0x0000000004011cd <+217>: mov %rdx,%rcx
11 0x0000000004011d0 <+220>: jmp 0x4011bd <phase_6+201>
12 0x0000000004011d2 <+222>: movq $0x0,0x8(%rdx)

```

以上这段代码，是将链表按照stack中排列的顺序重新连接起来。

```

1 0x0000000004011da <+230>: mov $0x5,%ebp
2 0x0000000004011df <+235>: mov 0x8(%rbx),%rax
3 0x0000000004011e3 <+239>: mov (%rax),%eax
4 0x0000000004011e5 <+241>: cmp %eax,(%rbx)
5 0x0000000004011e7 <+243>: jge 0x4011ee <phase_6+250>
6 0x0000000004011e9 <+245>: callq 0x40143a <explode_bomb>
7 0x0000000004011ee <+250>: mov 0x8(%rbx),%rbx
8 0x0000000004011f2 <+254>: sub $0x1,%ebp
9 0x0000000004011f5 <+257>: jne 0x4011df <phase_6+235>

```

这段代码表示，只有节点大小按照从小到大排列时，才能避开炸弹。

摆烂了，不看这个了。答案是：4 3 2 1 6 5

```

1 0x0000000004011f7 <+259>: add $0x50,%rsp
2 0x0000000004011fb <+263>: pop %rbx
3 0x0000000004011fc <+264>: pop %rbp
4 0x0000000004011fd <+265>: pop %r12
5 0x0000000004011ff <+267>: pop %r13
6 0x000000000401201 <+269>: pop %r14
7 0x000000000401203 <+271>: retq
8 End of assembler dump.
9 (gdb)

```

最后这部分是无用的部分，具体想了解可以去关注下caller与callee。

```

→ bomb ./bomb input
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...
Congratulations! You've defused the bomb!
→ bomb cat input
Border relations with Canada have never been better.
1 2 4 8 16 32
6 682
7 0
ioneftg
4 3 2 1 6 5

```

结束了。疲惫，phase\_6 做的很累，全是循环。

## Missing? Overlooked?

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

Mua ha ha ha ha!

```

/* Oh yeah? Well, how good is your math? Try on this saucy problem! */
input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one. Try this one.\n");

/* Round and 'round in memory we go, where we stop, the bomb blows! */
input = read_line();
phase_5(input);
phase_defused();
printf("Good work! On to the next...\n");

/* This phase will never be used, since no one will get past the
 * earlier ones. But just in case, make this one extra hard. */
input = read_line();
phase_6(input);
phase_defused();

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

return 0;

```

排除法，`read_line()` 是我们 bomblab 的六个输入。phase\_{1,2,3,4,5,6} 我们也一个一个的 disassemble 过了。

就剩下一个可疑的 `phase_defused()`。

```

1 (gdb) disassemble phase_defused
2 Dump of assembler code for function phase_defused:
3 => 0x00000000004015c4 <+0>:      sub     $0x78,%rsp
4     0x00000000004015c8 <+4>:      mov     %fs:0x28,%rax
5     0x00000000004015d1 <+13>:     mov     %rax,0x68(%rsp)

```

关于金丝雀值的部分，先不谈了。

```
1 0x00000000004015d6 <+18>: xor    %eax,%eax
2 # eax = 0
3 0x00000000004015d8 <+20>: cmpl   $0x6,0x202181(%rip)      # 0x603760
  <num_input_strings>
4 0x00000000004015df <+27>: jne    0x40163f <phase_defused+123>
```

这里的注释提示我们，我们要累计输入六个字符串，也就是在phase\_6之后才能开启隐藏关。可以通过gdb查看 0x603760 处得知。

```
gdb
(gdb) x/d 0x603760
0x603760 <num_input_strings>: 1
(gdb) c
Continuing.
Phase 1 defused. How about the next one?

Breakpoint 1, 0x00000000004015c4 in phase_defused ()
(gdb) x/d 0x603760
0x603760 <num_input_strings>: 2
(gdb) c
Continuing.
That's number 2. Keep going!

Breakpoint 1, 0x00000000004015c4 in phase_defused ()
(gdb) x/d 0x603760
0x603760 <num_input_strings>: 3
(gdb)
```

```
1 0x00000000004015e1 <+29>: lea    0x10(%rsp),%r8
2 0x00000000004015e6 <+34>: lea    0xc(%rsp),%rcx
3 0x00000000004015eb <+39>: lea    0x8(%rsp),%rdx
4 # `r8 rcx rdx` 分别保存了一个数字，现在还不知道什么用处。
5 0x00000000004015f0 <+44>: mov    $0x402619,%esi
6 0x00000000004015f5 <+49>: mov    $0x603870,%edi
7 0x00000000004015fa <+54>: callq  0x400bf0 <__isoc99_sscanf@plt>
```

```
(gdb) x/8wd $rsp
0x7fffffffdf40: 3      4      5      6
0x7fffffffdf50: 1      2      0      0
(gdb) x/s 0x402619
0x402619:      "%d %d %s"
(gdb) x/s 0x603870
0x603870 <input_strings+240>:  "7 0 "
(gdb) |
```



```
Breakpoint 4, __GI___isoc99_sscanf (s=0x603870 <input_strings+240> "7 0 ",
    format=0x402619 "%d %d %s") at isoc99_sscanf.c:24
24     isoc99_sscanf.c: No such file or directory.
```

```
1      0x00000000004015ff <+59>:    cmp     $0x3,%eax
2      0x0000000000401602 <+62>:    jne     0x401635 <phase_defused+113>
```

被转换的输入字符串要是两个数字加一个字符串。也就是我们第三个输入要是 7 0 string?。下面看看如何确定这个 string。

```
1      0x0000000000401604 <+64>:    mov     $0x402622,%esi
2      0x0000000000401609 <+69>:    lea     0x10(%rsp),%rdi
3      0x000000000040160e <+74>:    callq   0x401338 <strings_not_equal>
4      0x0000000000401613 <+79>:    test    %eax,%eax
5      0x0000000000401615 <+81>:    jne     0x401635 <phase_defused+113>
```

这段我们太熟了，bomb至少出现了三次了。直接查看 0x402622 处的字符串，就得到我们需要的字符串了。Ok，是作者的名字 -.- 这彩蛋真冷。

```
(gdb) x/s 0x402622
0x402622:      "DrEvil"
(gdb) |
```

```
1      0x0000000000401617 <+83>:    mov     $0x4024f8,%edi
2      0x000000000040161c <+88>:    callq   0x400b10 <puts@plt>
3      0x0000000000401621 <+93>:    mov     $0x402520,%edi
4      0x0000000000401626 <+98>:    callq   0x400b10 <puts@plt>
5      0x000000000040162b <+103>:   mov     $0x0,%eax
6      0x0000000000401630 <+108>:   callq   0x401242 <secret_phase>
7      0x0000000000401635 <+113>:   mov     $0x402558,%edi
8      0x000000000040163a <+118>:   callq   0x400b10 <puts@plt>
```

```
(gdb) c
Continuing.
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

这样我们就进入了 secret phase

关于金丝雀值的部分，不谈了。

```
1      0x000000000040163f <+123>:   mov     0x68(%rsp),%rax
2      0x0000000000401644 <+128>:   xor     %fs:0x28,%rax
3      0x000000000040164d <+137>:   je      0x401654 <phase_defused+144>
4      0x000000000040164f <+139>:   callq   0x400b30 <__stack_chk_fail@plt>
5      0x0000000000401654 <+144>:   add     $0x78,%rsp
6      0x0000000000401658 <+148>:   retq
7      End of assembler dump.
8      (gdb)
```

# secret\_phase

彩蛋关的代码并不是很长，这很好。

```
1 Dump of assembler code for function secret_phase:
2 => 0x0000000000401242 <+0>:      push    %rbx
3     0x0000000000401243 <+1>:      callq   0x40149e <read_line>
4     0x0000000000401248 <+6>:      mov     $0xa,%edx
5     0x000000000040124d <+11>:     mov     $0x0,%esi
6     0x0000000000401252 <+16>:     mov     %rax,%rdi
7     0x0000000000401255 <+19>:     callq   0x400bd0 <strtol@plt>
8     0x000000000040125a <+24>:     mov     %rax,%rbx
```

首先读取一行输入，结果会被保存在rax中。之后通过：

```
1 long int strtol(const char *str, char **endptr, int base)
2 // strtol(read_line,NULL,10)
3 // 10表示转换成10进制表示，endptr表示第一个不能转换的字符。
```

rax中存放着这个被转换的数字。

```
1     0x000000000040125d <+27>:     lea     -0x1(%rax),%eax
2     0x0000000000401260 <+30>:     cmp     $0x3e8,%eax
3     0x0000000000401265 <+35>:     jbe     0x40126c <secret_phase+42>
4     0x0000000000401267 <+37>:     callq   0x40143a <explode_bomb>
5     # eax <= 0x3e8
```

```
1     0x000000000040126c <+42>:     mov     %ebx,%esi
2     0x000000000040126e <+44>:     mov     $0x6030f0,%edi
3     0x0000000000401273 <+49>:     callq   0x401204 <fun7>
```

这里调用了 `fun7` 我们先跳过，看看后面的内容。后面的内容表明，我们 `fun7` 的返回值必须是2。之后，他会打印一句话。表示我们成功解决了这个彩蛋关卡。

```
1     0x0000000000401278 <+54>:     cmp     $0x2,%eax
2     0x000000000040127b <+57>:     je      0x401282 <secret_phase+64>
3     0x000000000040127d <+59>:     callq   0x40143a <explode_bomb>
4     0x0000000000401282 <+64>:     mov     $0x402438,%edi
5     0x0000000000401287 <+69>:     callq   0x400b10 <puts@plt>
6     0x000000000040128c <+74>:     callq   0x4015c4 <phase_defused>
7     0x0000000000401291 <+79>:     pop     %rbx
8     0x0000000000401292 <+80>:     retq
9 End of assembler dump.
10 (gdb)
```

## fun7

对于 `fun7` ,我们现在知道：

- arg1: `edi = $0x6030f0`
- arg2: 是 `read_line`，即我们的输入。我们的输入必须小于等于1001
- `return == 2 is True`

现在，跳转到 `fun7`：

```

1 (gdb) disassemble fun7
2 Dump of assembler code for function fun7:
3 => 0x000000000401204 <+0>:      sub    $0x8,%rsp
4      0x000000000401208 <+4>:      test   %rdi,%rdi
5      0x00000000040120b <+7>:      je     0x401238 <fun7+52>
6

```

```

1      0x00000000040120d <+9>:      mov     (%rdi),%edx
2      0x00000000040120f <+11>:     cmp     %esi,%edx
3      0x000000000401211 <+13>:     jle     0x401220 <fun7+28>
4      # 如果我们的输入 >= 36,就跳转到了+28处。

```

先假设不跳转，继续向下看。如果不跳转，会修改  $rdi = rdi + 0x8$ ，之后再次调用 `fun7`

```

1 # my_input < 36  rdi = rdi + 0x8 rsi不变 调用fun7
2      0x000000000401213 <+15>:     mov     0x8(%rdi),%rdi
3      0x000000000401217 <+19>:     callq  0x401204 <fun7>
4      0x00000000040121c <+24>:     add     %eax,%eax
5      0x00000000040121e <+26>:     jmp     0x40123d <fun7+57>
6
7 # my_input == 36 ret 0
8      0x000000000401220 <+28>:     mov     $0x0,%eax
9      0x000000000401225 <+33>:     cmp     %esi,%edx
10     0x000000000401227 <+35>:     je      0x40123d <fun7+57>
11     0x000000000401229 <+37>:     mov     0x10(%rdi),%rdi
12
13 # my_input > 36  rdi = $0x6030f0 + 0x10 rsi不变, 调用fun7
14     0x00000000040122d <+41>:     callq  0x401204 <fun7>
15     0x000000000401232 <+46>:     lea     0x1(%rax,%rax,1),%eax
16     0x000000000401236 <+50>:     jmp     0x40123d <fun7+57>
17     0x000000000401238 <+52>:     mov     $0xffffffff,%eax
18     0x00000000040123d <+57>:     add     $0x8,%rsp
19     0x000000000401241 <+61>:     retq
20 End of assembler dump.
21 (gdb)

```

第一次执行完成后，存在三种情况，分别是 `my_input` 大于等于或者小于三种。我们可以创建一颗决策树。通过访问存于 `rdi` 中地址所指向的地址处的值，可以绘制出来发现是一颗 AVL 树。

```
(gdb) x/d 0x603110
0x603110 <n21>: 8
(gdb) x/d 0x603190
0x603190 <n31>: 6
(gdb) x/d 0x6031f0
0x6031f0 <n41>: 1
(gdb) x/d 0x603250
0x603250 <n42>: 7
(gdb) x/d 0x603150
0x603150 <n32>: 22
(gdb) x/d 0x603270
0x603270 <n43>: 20
(gdb) x/d 0x603230
0x603230 <n44>: 35
(gdb) x/d 0x603130
0x603130 <n22>: 50
(gdb) x/d 0x603170
0x603170 <n33>: 45
(gdb) x/d 0x6031d0
0x6031d0 <n45>: 40
(gdb) |
```



```
~ /bomb
→ bomb ./bomb input
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!
→ bomb cat input
Border relations with Canada have never been better.
1 2 4 8 16 32
6 682
7 0 DrEvil
ione fg
4 3 2 1 6 5
20
→ bomb |
```