

实验要求
相关命令
Phase_1
Phase_2
Phase_3
Phase_4
Phase_5
Phase_6
secret_phase
总结

实验要求

- 第三章：程序的机器级表示
 - C语言、汇编代码以及机器代码之间的关系
 - 如何实现C语言中的控制结构
 - 过程的实现、数据和控制的传递、局部变量的存储
 - 如何实现像数组、结构和联合这样的数据结构
 - 内存访问越界，缓冲区溢出攻击
- 实验要求

有6个 phase，对于每个 phase，需要输入一段字符串，然后让代码中的 `explode_bomb()` 函数不执行，这样就不会爆炸。

实验给了个 `bomb.c` 的代码文件，但是没有头文件，所以不能运行和编译，还给了个 `bomb` 可执行目标程序。

主要方法就是要用 `gdb` 调试，反汇编 `bomb` 文件，然后通过打断点的方法，阅读汇编语句，以及查看寄存器和内存里存的值的情况，推算出应该输入的语句

相关命令

解压命令 `tar -xvf bomb.tar`

`bomb`是炸弹程序，直接运行就行

`.d` 反汇编代码

反汇编指令 `objdump -d sum > sum.d`

调试 `gdb sum`

在 `gdb` 中实现反汇编 `disassemble sumstore`

同时显示三个窗口 (`src, asm, gdb`) `layout split`

`kill` 结束调试

`set args [inputfile]` 输入重定向到指定的文件

或者直接 `run answer.txt`

si 单步调试（会进入函数，后面可以加跳过的步数）

ni 单步调试（会跳过函数）

continue 继续执行到下一个断点

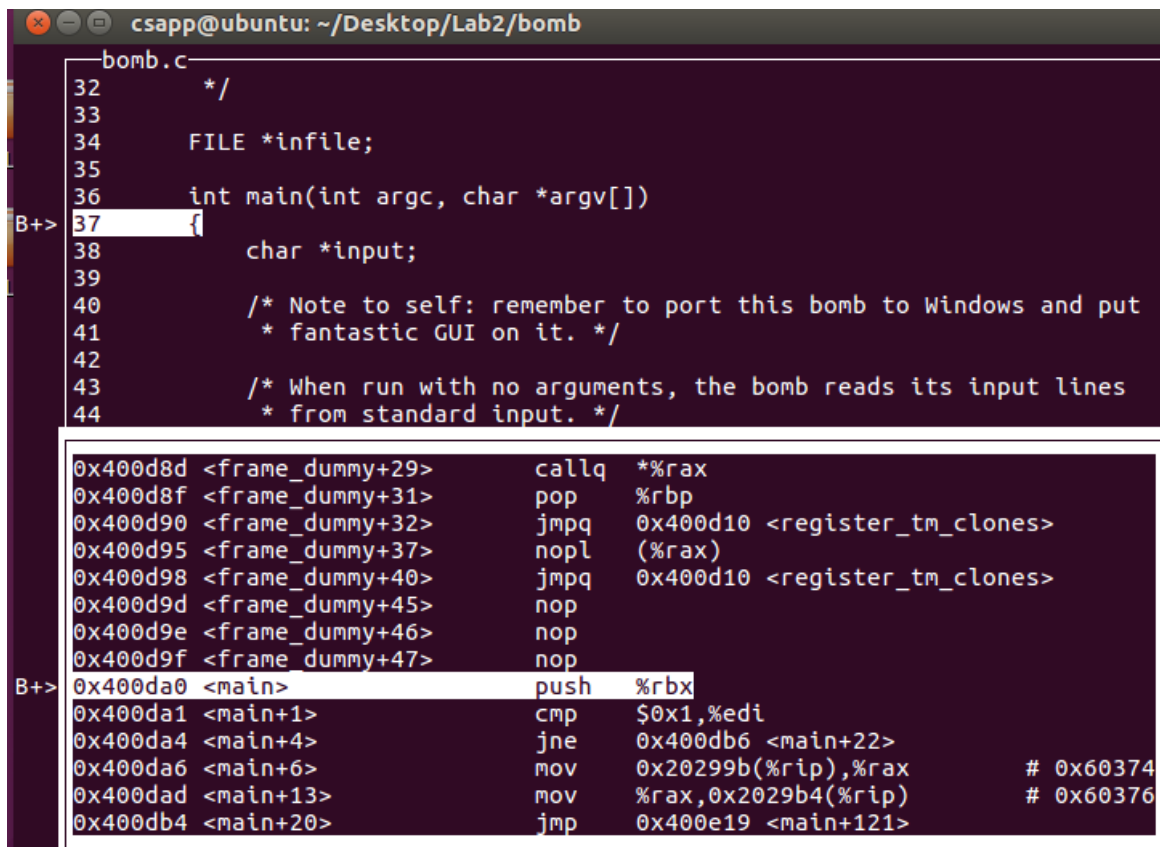
finish 继续执行到函数结束

refresh 显示有重复

Phase_1

在main()开始处添加断点，查看汇编文件的main()开始处

```
b main
run
```



The screenshot shows a debugger window with the title bar "csapp@ubuntu: ~/Desktop/Lab2/bomb". The main window displays the source code of "bomb.c". The code is as follows:

```
32  */
33
34  FILE *infile;
35
36  int main(int argc, char *argv[])
37  {
38      char *input;
39
40      /* Note to self: remember to port this bomb to Windows and put
41       * fantastic GUI on it. */
42
43      /* When run with no arguments, the bomb reads its input lines
44       * from standard input. */
```

The left margin shows a "B+>" symbol next to line 37. Below the source code, the assembly code for the "main" function is displayed. The assembly code is as follows:

```
0x400d8d <frame_dummy+29>    callq  *%rax
0x400d8f <frame_dummy+31>    pop    %rbp
0x400d90 <frame_dummy+32>    jmpq   0x400d10 <register_tm_clones>
0x400d95 <frame_dummy+37>    nopl   (%rax)
0x400d98 <frame_dummy+40>    jmpq   0x400d10 <register_tm_clones>
0x400d9d <frame_dummy+45>    nop
0x400d9e <frame_dummy+46>    nop
0x400d9f <frame_dummy+47>    nop
B+> 0x400da0 <main>        push   %rbx
0x400da1 <main+1>        cmp    $0x1,%edi
0x400da4 <main+4>        jne    0x400db6 <main+22>
0x400da6 <main+6>        mov    0x20299b(%rip),%rax    # 0x60374
0x400dad <main+13>       mov    %rax,0x2029b4(%rip)    # 0x60376
0x400db4 <main+20>       jmp    0x400e19 <main+121>
```

main()开始于264行

phase_1()开始于346行

利用 `strings_not_equal()` 函数检查输入字符串是否正确

%rdi 第一个参数： 输入字符串起始地址

%rsi 第二个参数

```

0000000000400ee0 <phase_1>:
  400ee0:  48 83 ec 08          sub    $0x8,%rsp      # 分配栈帧
  400ee4:  be 00 24 40 00      mov    $0x402400,%esi  # %rdi保存输入参数，
                        %rsi赋予新值
  400ee9:  e8 4a 04 00 00      callq 401338 <strings_not_equal>
  400eee:  85 c0              test   %eax,%eax      # 测试前面调用的函数返回值是
                        否为0
  400ef0:  74 05              je     400ef7 <phase_1+0x17>  #如果为0则跳转
                        退出
  400ef2:  e8 43 05 00 00      callq 40143a <explode_bomb>
  400ef7:  48 83 c4 08          add    $0x8,%rsp      # 回收栈帧
  400efb:  c3                retq

```

```
test    %eax,%eax
```

将两个参数进行&操作，因此只有当 %eax 值为0时，0&0结果为0，而非零值和自身相与必为1（如果为0会将ZF标志位置为1）

je 根据ZF位是否被置为1，决定是否跳转（如果是1则跳转）

strings_not_equal 判断输入密码是否正确

输入参数值为 **字符串起始地址** 和 0x402400

猜测地址 0x402400 处存放着正确密码！

%rbx **字符串起始地址** %rbp 0x402400

先比较两个参数的长度（输入字符串起始地址和 0x402400）

然后逐位进行比较

```

0000000000401338 <strings_not_equal>:
  401338:  41 54              push   %r12
  40133a:  55              push   %rbp
  40133b:  53              push   %rbx      # 类似保存上下文，当前函数需要使用
                        这几个寄存器，函数返回时再恢复
  40133c:  48 89 fb          mov    %rdi,%rbx  # 输入字符串的起始地址，第一
                        个参数
  40133f:  48 89 f5          mov    %rsi,%rbp  # 0x402400，第二个参数
  401342:  e8 d4 ff ff ff    callq 40131b <string_length> # 密码长度
  401347:  41 89 c4          mov    %eax,%r12d # 密码长度存在%r12d
  40134a:  48 89 ef          mov    %rbp,%rdi
  40134d:  e8 c9 ff ff ff    callq 40131b <string_length> # 第二个起始地址
                        为0x402400的字符串长度
  401352:  ba 01 00 00 00    mov    $0x1,%edx
  401357:  41 39 c4          cmp    %eax,%r12d  # 比较两个参数的长度（输入字
                        符串起始地址和0x402400）
  40135a:  75 3f              jne    40139b <strings_not_equal+0x63> #长度
                        相同就继续
  40135c:  0f b6 03          movzbl (%rbx),%eax # 将字符串当前地址的值赋值
                        给%eax，间接寻址
  40135f:  84 c0              test   %al,%al    # 判断字符串是否为空
  401361:  74 25              je     401388 <strings_not_equal+0x50>  # 不
                        为空就继续
  401363:  3a 45 00          cmp    0x0(%rbp),%al # 将当前位和正确答案比较

```

```

401366: 74 0a      je      401372 <strings_not_equal+0x3a> # 相等
401368: eb 25      jmp     40138f <strings_not_equal+0x57>
40136a: 3a 45 00   cmp     0x0(%rbp),%al # 循环检查开始点
40136d: 0f 1f 00   nopl    (%rax) # No Operation 空指令
401370: 75 24      jne     401396 <strings_not_equal+0x5e> # 相等就继续
401372: 48 83 c3 01 add     $0x1,%rbx # 第一个参数移一位
401376: 48 83 c5 01 add     $0x1,%rbp # 第二个参数移一位
40137a: 0f b6 03   movzbl  (%rbx),%eax
40137d: 84 c0      test    %al,%al # 检查是否结束（当前字符为0x0）
40137f: 75 e9      jne     40136a <strings_not_equal+0x32> # 未结束继续循环
401381: ba 00 00 00 00 mov     $0x0,%edx
401386: eb 13      jmp     40139b <strings_not_equal+0x63>
401388: ba 00 00 00 00 mov     $0x0,%edx
40138d: eb 0c      jmp     40139b <strings_not_equal+0x63>
40138f: ba 01 00 00 00 mov     $0x1,%edx
401394: eb 05      jmp     40139b <strings_not_equal+0x63>
401396: ba 01 00 00 00 mov     $0x1,%edx
40139b: 89 d0      mov     %edx,%eax
40139d: 5b        pop     %rbx
40139e: 5d        pop     %rbp
40139f: 41 5c      pop     %r12
4013a1: c3        retq

```

string_length() 获得字符串长度

从字符串首地址开始，地址递增，直到找到第一个地址指向的值为 0x0 的地址，返回当前遍历地址减去初始地址值（即字符串长度）

逐位检查字符是否为空，遇到 0x0 就结束

每次循环都会得到当前字符位置与初始位置的差值（即当前长度），结束循环时刚好是完整字符串的长度

```

00000000040131b <string_length>:
40131b: 80 3f 00   cmpb    $0x0,(%rdi) # 比较两个值 b-a
40131e: 74 12      je      401332 <string_length+0x17> # 相等则跳转结束（空字符串）
401320: 48 89 fa   mov     %rdi,%rdx # 字符串初始地址存在 %rdi
401323: 48 83 c2 01 add     $0x1,%rdx # 加1得到字符串新地址%rdx
401327: 89 d0      mov     %edx,%eax # 字符串新地址赋值给返回参数
401329: 29 f8      sub     %edi,%eax # 新字符串地址 减 初始字符串地址（此时%rax存放当前长度）
40132b: 80 3a 00   cmpb    $0x0,(%rdx) # 新字符串地址所指向的值是否为0
40132e: 75 f3      jne     401323 <string_length+0x8> # 不为0就循环add $0x1,%rdx
401330: f3 c3      repz retq
401332: b8 00 00 00 00 mov     $0x0,%eax # 返回字符串长度为0
401337: c3        retq

```

查看存放在 0x402400 的字符串

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

最终密码

Border relations with Canada have never been better.

```
csapp@ubuntu:~/Desktop/Lab2/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?
█
```

Phase_2

phase_2() 在356行

```
000000000400efc <phase_2>:
400efc: 55                push    %rbp          # 被调用者保存
400efd: 53                push    %rbx
400efe: 48 83 ec 28       sub     $0x28,%rsp     # 分配栈帧 40字节
400f02: 48 89 e6          mov     %rsp,%rsi      # 当前栈指针传递给第二个参
数%rsi(第一个参数%rdi存字符串地址)
400f05: e8 52 05 00 00    callq   40145c <read_six_numbers>
400f0a: 83 3c 24 01       cmpl    $0x1,(%rsp)    # 此时的%rsp相当于
read_six_numbers中的%rsi, 保存第一个数
400f0e: 74 20            je      400f30 <phase_2+0x34>    # 第一个数字等
于1, 正确
400f10: e8 25 05 00 00    callq   40143a <explode_bomb>
400f15: eb 19            jmp     400f30 <phase_2+0x34>
400f17: 8b 43 fc          mov     -0x4(%rbx),%eax # %rax保存上一个正确的值
400f1a: 01 c0            add     %eax,%eax      # %rax乘以2
400f1c: 39 03            cmp     %eax,(%rbx)    # 将%rax和当前待检测值相比较
400f1e: 74 05            je      400f25 <phase_2+0x29>    # 相等就跳过爆
炸
400f20: e8 15 05 00 00    callq   40143a <explode_bomb>    # 不相等就爆炸
400f25: 48 83 c3 04       add     $0x4,%rbx      # %rbx保存下一个待检测值
400f29: 48 39 eb          cmp     %rbp,%rbx      # 是否结束(6个数字都检测完
了)
400f2c: 75 e9            jne     400f17 <phase_2+0x1b>    # 未结束就继续
400f2e: eb 0c            jmp     400f3c <phase_2+0x40>    # 结束就成功退
出
400f30: 48 8d 5c 24 04    lea     0x4(%rsp),%rbx  # %rbx存放下一个数字
400f35: 48 8d 6c 24 18    lea     0x18(%rsp),%rbp # %rbp存放结束位置(第6
个数字的下一个位置)
400f3a: eb db            jmp     400f17 <phase_2+0x1b>    # 返回继续判断
400f3c: 48 83 c4 28       add     $0x28,%rsp
400f40: 5b                pop     %rbx
400f41: 5d                pop     %rbp
400f42: c3                retq
```

`read_six_numbers()` 可以得知我们需要输入6个数字，中间空格隔开。回到 `phase_2()` 中，`(%rsp)` 中的值和1比较，可以看到此时的 `%rsp` 就是 `read_six_numbers()` 的 `%rsi`，所以取得就是6个值里的第一个。如果不相等，就会 `explode_bomb`，所以第一个必须为1，然后 `%rbx` 为下一个待检测值，然后拿前一个值乘2和当前待检测值相比，不相等就爆炸，所以必须相等，因此这个序列应该是公比为2、首项为1的等比数列：1,2,4,8,16,32

`read_six_numbers()`

```
00000000040145c <read_six_numbers>:
40145c:  48 83 ec 18          sub     $0x18,%rsp    # 分配栈帧: %rsi栈底 %rsp
栈顶
401460:  48 89 f2             mov     %rsi,%rdx    # 栈帧基地址%rsi作为第三个参
数, 放在%rdx
401463:  48 8d 4e 04          lea     0x4(%rsi),%rcx # %rsi+4作为第四个参
数, 放在%rcx
401467:  48 8d 46 14          lea     0x14(%rsi),%rax
40146b:  48 89 44 24 08       mov     %rax,0x8(%rsp) # %rsi+20作为第八个参
数, 放在栈中%rsp+8
401470:  48 8d 46 10          lea     0x10(%rsi),%rax
401474:  48 89 04 24          mov     %rax,(%rsp)   # %rsi+16作为第七个参
数, 放在栈中%rsp
401478:  4c 8d 4e 0c          lea     0xc(%rsi),%r9  # %rsi+12作为第六个参
数, 放在%r9
40147c:  4c 8d 46 08          lea     0x8(%rsi),%r8  # %rsi+8作为第五个参
数, 放在%r8
401480:  be c3 25 40 00       mov     $0x4025c3,%esi # sscanf()的第二个参
数, 指定格式
401485:  b8 00 00 00 00       mov     $0x0,%eax     # sscanf()的返回值先默认设为
0
40148a:  e8 61 f7 ff ff       callq   400bf0 <__isoc99_sscanf@plt>
40148f:  83 f8 05             cmp     $0x5,%eax     # 检查输入数字的个数是否大于5
401492:  7f 05               jg      401499 <read_six_numbers+0x3d>
401494:  e8 a1 ff ff ff       callq   40143a <explode_bomb>
401499:  48 83 c4 18          add     $0x18,%rsp    # 回收栈帧
40149d:  c3                 retq
```

从 `phase_2()` 的栈顶，每隔4字节，取6个地址，其中前4个存放在寄存器中，后2个存放在当前 `read_six_numbers()` 栈中

因为 `sscanf()` 的第一个参数指定字符串，第二个参数指定格式，后面的参数为存放字符串处理后结果的地址，本题共需6个地址

所以本题共需8个参数，而最多只有6个寄存器可以保存参数，所以后两个参数只能通过栈进行保存（使用的是调用者的栈）

结论：当函数参数的数量超过6个时，会使用栈来保存多余的参数

这6个寄存器分别是 `%rdi`，`%rsi`，`%rdx`，`%rcx`，`%r8`，`%r9`

`%rdi` 保存input字符串地址，`%rsi` 指定格式，其次剩余的寄存器保存前4个地址，后2个地址保存在当前 `read_six_numbers()` 栈帧的 `(%rsp)` 和 `8(%rsp)`

也即这6个地址分别为 `0x0(%rsi)`，`0x4(%rsi)`，`0x8(%rsi)`，`0xc(%rsi)`，`0x10(%rsi)`，`0x14(%rsi)`

当前 `read_six_numbers()` 栈帧的 `%rsi`，即为 `phase_2()` 栈帧的栈顶 `%rsp`，即从 `phase_2()` 的栈顶，每隔4字节，取的6个地址

注: 40148a: e8 61 f7 ff ff callq 400bf0 <__isoc99_sscanf@plt>

- 调用 `sscanf()`

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

`sscanf()` 会将参数 `str` 的字符串根据参数 `format` (格式化字符串) 来转换并格式化数据, 转换后的结果存于对应的变量中

- 参数
 - 参数 `str` 为要读取的整个字符串
 - `format` 为指定的格式
 - `argument` (可选)为地址, 用来保存处理完成后读取到的数据
- 返回值
 - 成功则返回成功匹配和赋值的个数(附加参数`argument`数目), 失败则返回-1
- 在本题中, `sscanf()` 的第二个参数的指定格式存放在 `0x4025c3`, 可以看到要求输入**六个整数**

```
(gdb) x/s 0x4025c3
0x4025c3: "%d %d %d %d %d %d"
```

- 可以看到 `sscanf()` 返回之后, 将返回值与5进行比较, 如果大于5则成功退出, 否则爆炸

比较a和b的大小: `a>b`

```
cmp b a
```

即 `a-b`

```
root@ubuntu:/home/csapp/Desktop/Lab2/bomb# ./bomb answer.txt
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!
```

Phase_3

`phase_3()`

此问题要求输入两个数, 且第一个数`x`在整数0到7的范围内, 每个`x`都对应一个特定的第二个数`y`, 因此共有8种情况

问题中提供了一个基地址 `0x402470`, 对于符合要求的`x`, 执行 `0x402470+8*x` 得到一个新地址, 再利用 `jmp` 的间接跳转

间接跳转: 以 `0x402470+8*x` 计算所得的值作为读地址, 从该地址中读出跳转目标

每种情况都提供给`y`一个特定值, 再将输入的`y`与其进行比较, 如果匹配则成功退出

```
000000000400f43 <phase_3>:
400f43: 48 83 ec 18          sub    $0x18,%rsp
```


400f47: 48 8d 4c 24 0c	lea 0xc(%rsp),%rcx # 设第二个数为y
400f4c: 48 8d 54 24 08	lea 0x8(%rsp),%rdx # 设第一个数为x
400f51: be cf 25 40 00	mov \$0x4025cf,%esi # 格式为两个数
400f56: b8 00 00 00 00	mov \$0x0,%eax
400f5b: e8 90 fc ff ff	callq 400bf0 <__isoc99_sscanf@plt>
400f60: 83 f8 01	cmp \$0x1,%eax
400f63: 7f 05	jg 400f6a <phase_3+0x27> # 少于两个数就
爆炸	
400f65: e8 d0 04 00 00	callq 40143a <explode_bomb>
400f6a: 83 7c 24 08 07	cmpl \$0x7,0x8(%rsp) # 第一个数和7比较（无符
号比较）	
400f6f: 77 3c	ja 400fad <phase_3+0x6a> # 第一个数大于7
或小于0就爆炸就爆炸，只能取0到7	
400f71: 8b 44 24 08	mov 0x8(%rsp),%eax
400f75: ff 24 c5 70 24 40 00	jmpq *0x402470(,%rax,8) # 跳转到
0x402470+8*x的地址处	
400f7c: b8 cf 00 00 00	mov \$0xcf,%eax # x=0, y=207
400f81: eb 3b	jmp 400fbe <phase_3+0x7b>
400f83: b8 c3 02 00 00	mov \$0x2c3,%eax # x=2, y=707
400f88: eb 34	jmp 400fbe <phase_3+0x7b>
400f8a: b8 00 01 00 00	mov \$0x100,%eax # x=3, y=256
400f8f: eb 2d	jmp 400fbe <phase_3+0x7b>
400f91: b8 85 01 00 00	mov \$0x185,%eax # x=4, y=389
400f96: eb 26	jmp 400fbe <phase_3+0x7b>
400f98: b8 ce 00 00 00	mov \$0xce,%eax # x=5, y=206
400f9d: eb 1f	jmp 400fbe <phase_3+0x7b>
400f9f: b8 aa 02 00 00	mov \$0x2aa,%eax # x=6, y=682
400fa4: eb 18	jmp 400fbe <phase_3+0x7b>
400fa6: b8 47 01 00 00	mov \$0x147,%eax # x=7, y=327
400fab: eb 11	jmp 400fbe <phase_3+0x7b>
400fad: e8 88 04 00 00	callq 40143a <explode_bomb>
400fb2: b8 00 00 00 00	mov \$0x0,%eax
400fb7: eb 05	jmp 400fbe <phase_3+0x7b>
400fb9: b8 37 01 00 00	mov \$0x137,%eax # x=1, y=311
400fbe: 3b 44 24 0c	cmp 0xc(%rsp),%eax # 比较当前x取值情况
下，y与给定值是否相等	
400fc2: 74 05	je 400fc9 <phase_3+0x86>
400fc4: e8 71 04 00 00	callq 40143a <explode_bomb>
400fc9: 48 83 c4 18	add \$0x18,%rsp
400fcd: c3	retq

读取基地址 0x402470 附近的值，此处保存的值分别是 phase3() 中的地址值

```
objdump -s -j .rodata bomb # -s表示段segment，-j指明具体段（ELF格式：test,data,rodata等）
```

```
402470 7c0f4000 00000000 b90f4000 00000000
402480 830f4000 00000000 8a0f4000 00000000
402490 910f4000 00000000 980f4000 00000000
4024a0 9f0f4000 00000000 a60f4000 00000000
```

- 小端序（更常见）

第一个字节是最低有效字节（数字最低位所在的字节）

00400f7c 的最低有效字节为 7c，将其放在最前面；剩余依次为 0f，40 和 00，因此表示为 7c0f4000

测试结果 *0x402470(,%rax,8) -1 0x00000000 8 0x7564616d (x超出范围0到7)

0 400f7c 1 400fb9 2 400f83 3 400f8a
4 400f91 5 400f98 6 400f9f 7 400fa6

Phase_4

phase_4()

输入两个数字x和y，要求x小于等于14且y=0

func4() 会检查x是否符合该函数的要求

func4() 的返回值 %rax 必须为0才不会发生爆炸

(将测试, x只可以取: 7, 3, 1, 0)

```
000000000040100c <phase_4>:
40100c: 48 83 ec 18      sub    $0x18,%rsp
401010: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx
401015: 48 8d 54 24 08    lea    0x8(%rsp),%rdx
40101a: be cf 25 40 00    mov    $0x4025cf,%esi # 输入两个整数x和y
40101f: b8 00 00 00 00    mov    $0x0,%eax
401024: e8 c7 fb ff ff    callq 400bf0 <__isoc99_sscanf@plt>
401029: 83 f8 02          cmp    $0x2,%eax # %rax=2
40102c: 75 07             jne    401035 <phase_4+0x29>
40102e: 83 7c 24 08 0e    cmpl   $0xe,0x8(%rsp) # 将x和14比较
401033: 76 05             jbe    40103a <phase_4+0x2e> # 要求x<=14
401035: e8 00 04 00 00    callq 40143a <explode_bomb>
40103a: ba 0e 00 00 00    mov    $0xe,%edx # c=14
40103f: be 00 00 00 00    mov    $0x0,%esi # b=0
401044: 8b 7c 24 08       mov    0x8(%rsp),%edi # a=x
401048: e8 81 ff ff ff    callq 400fce <func4> # func4的3个参数a,b,c
40104d: 85 c0             test   %eax,%eax
40104f: 75 07             jne    401058 <phase_4+0x4c> # 非0就爆炸
401051: 83 7c 24 0c 00    cmpl   $0x0,0xc(%rsp) # 将y和0比较, 要求y=0
401056: 74 05             je     40105d <phase_4+0x51>
401058: e8 dd 03 00 00    callq 40143a <explode_bomb>
40105d: 48 83 c4 18       add    $0x18,%rsp
401061: c3               retq
```

func4()

该函数是一个递归调用，在第一次执行时，会先判断x>=7，再判断x<=7，因此很容易想到令x=7

但如果这么简单又为什么要设计这么复杂的函数呢

首先，与x进行比较的这个值的选取，与二分查找选取mid值的过程是相同的，但为什么当x等于7,3,1,0可以，但与之对称的(7),11,13,14就不可以呢？

```
0000000000400fce <func4>:
400fce: 48 83 ec 08      sub    $0x8,%rsp
400fd2: 89 d0            mov    %edx,%eax # %rax=%rdx=c=14
400fd4: 29 f0            sub    %esi,%eax # %rax=c-b=14-0=14
400fd6: 89 c1            mov    %eax,%ecx # %rcx=14(phase_4已经实现
将地址传给sscanf()了, 不再需要)
```

400fd8:	c1 e9 1f	shr	\$0x1f,%ecx	# 将c-b逻辑右移31位=0
400fdb:	01 c8	add	%ecx,%eax	# %rax=(c-b)>>>31 + (c-b)=0+14=14
400fdd:	d1 f8	sar	%eax	# 算术右移1位%rax=7(相当于14/2=7)
400fdf:	8d 0c 30	lea	(%rax,%rsi,1),%ecx	# %rcx=%rax+b*1=7+0*1=7, 即0和14的中间值7
400fe2:	39 f9	cmp	%edi,%ecx	# 将a与%rcx比较(即x和%rcx)
400fe4:	7e 0c	jle	400ff2 <func4+0x24>	# 小于等于 (a>=%rcx)
400fe6:	8d 51 ff	lea	-0x1(%rcx),%edx	# %rdx=7-1=6
400fe9:	e8 e0 ff ff ff	callq	400fce <func4>	
400fee:	01 c0	add	%eax,%eax	# (%rax*2)
400ff0:	eb 15	jmp	401007 <func4+0x39>	
400ff2:	b8 00 00 00 00	mov	\$0x0,%eax	# 这一步将%rax置为0, 也即不会爆炸
400ff7:	39 f9	cmp	%edi,%ecx	# 再将a与%rcx比较(即x和%rcx)
400ff9:	7d 0c	jge	401007 <func4+0x39>	# 大于等于 (a<=%rcx)
400ffb:	8d 71 01	lea	0x1(%rcx),%esi	
400ffe:	e8 cb ff ff ff	callq	400fce <func4>	
401003:	8d 44 00 01	lea	0x1(%rax,%rax,1),%eax	# 这一步会将置为0的%rax重新变为1(%rax*2+1)
401007:	48 83 c4 08	add	\$0x8,%rsp	
40100b:	c3	retq		

- x=7

先判断 $x \geq 7$, 正确, 通过 400fe4: `jle 400ff2 <func4+0x24>` 进行跳转; 将%rax置为0, 再判断 $x \leq 7$, 正确, 跳转结束

- x=3

先判断 $x \geq 7$, 错误, 缩小范围使区间为0到6, 即mid=3, 通过 400fe9: `callq 400fce <func4>` 进行递归(入口)回到函数起始点(记住下一条指令为递归返回后要执行的第一条指令); 再判断 $x \geq 3$, 正确; %rax置为0, 判断 $x \leq 3$, 正确, 当前递归调用跳转结束; 返回到递归入口的下一条指令 400fee: `add %eax,%eax`, 而%rax=0*2=0, 接着跳转结束整个过程, 返回值还是0

- x=11

先判断 $x \geq 7$, 正确; %rax置为0, 判断 $x \leq 7$, 错误, 缩小范围使区间为8到14, 即mid=11, 通过 400ffe: `callq 400fce <func4>` 进行递归(入口)回到函数起始点; 再判断 $x \geq 11$, 正确; %rax置为0, 判断 $x \leq 11$, 正确, 当前递归调用跳转结束; 返回到递归入口的下一条指令 401003: `lea 0x1(%rax,%rax,1),%eax`, 而%rax=0*2+1=1, 因此这一步导致之前被置为0的%rax又变成了1, 接着结束整个过程, 返回值是1

- x=2

先判断 $x \geq 7$, 错误, 缩小范围使区间为0到6, 即mid=3, 通过 400fe9: `callq 400fce <func4>` 进行递归(入口1)回到函数起始点; 再判断 $x \geq 3$, 错误, 缩小范围使区间为0到2, 即mid=1, 通过 400fe9: `callq 400fce <func4>` 进行递归(入口2)回到函数起始点; 再判断 $x \geq 1$, 正确; %rax置为0, 判断 $x \leq 1$, 错误, 缩小范围使区间为2到2, 即mid=2, 通过 400ffe: `callq 400fce <func4>` 进行递归(入口3)回到函数起始点; 再判断 $x \geq 2$, 正确; %rax置为0, 判断 $x \leq 2$, 正确, 当前递归调用跳转结束; 返回到递归入口3的下一条指令 401003: `lea 0x1(%rax,%rax,1),%eax`, 而%rax=0*2+1=1, 因此这一步导致之前被置为0的%rax又变成了1; 当前递归调用跳转结束, 返回到递归入口2的下一条指令 400fee: `add %eax,%eax`, 而%rax=1*2=2; 当前递归调用跳转结束, 返回

到递归入口1的下一条指令 400fee: add %eax,%eax , 而 %rax = 2 * 2 = 4; 接着结束整个过程, 返回值是4

设递归1为选择左区间, 递归2为选择右区间

问题就在于调用一次递归2结束后的下一条指令为 401003: lea 0x1(%rax,%rax,1),%eax , 导致即使之前 %rax 已经被置为0, 但依然使得 %rax = 0*2+1=1 导致 %rax = 1

而正确的4种情况(7, 3, 1, 0), x=7时未调用递归, 剩余3种情况都只调用了递归1, 之后在进行第二次比较(x<=mid)之前将 %rax 置为了0, 接着第二次比较结果正确, 不会执行 401003: lea 0x1(%rax,%rax,1),%eax 这一条指令, 因此递归返回以后 %rax 依旧为0, 即使执行了多少次 400fee: add %eax,%eax 这一条指令, %rax 也依旧为0

而其余错误的情况, 都是因为调用了一次或多次递归2, 导致在递归返回时将 %rax 置为了非0, 因此结果也必然不为0

Phase_5

phase_5()

```
0000000000401062 <phase_5>:
401062: 53                push    %rbx
401063: 48 83 ec 20       sub     $0x20,%rsp
401067: 48 89 fb          mov     %rdi,%rbx      # 字符串地址存放在%rbx
40106a: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
401071: 00 00
401073: 48 89 44 24 18     mov     %rax,0x18(%rsp)
401078: 31 c0             xor     %eax,%eax
40107a: e8 9c 02 00 00     callq  40131b <string_length>  # 字符串长度
40107f: 83 f8 06          cmp     $0x6,%eax      # 要求长度等于6
401082: 74 4e             je      4010d2 <phase_5+0x70>
401084: e8 b1 03 00 00     callq  40143a <explode_bomb>
401089: eb 47            jmp     4010d2 <phase_5+0x70>
40108b: 0f b6 0c 03       movzbl  (%rbx,%rax,1),%ecx  # %rcx保存当前字符
的地址,遍历每个字符
40108f: 88 0c 24          mov     %cl, (%rsp)      # 将字符的低8位传
给%rdx
401092: 48 8b 14 24       mov     (%rsp),%rdx      # %rdx保存偏移值
(当前字符的ASCII码)
401096: 83 e2 0f          and     $0xf,%edx        # 只取%rdx的低4位
(偏移值0~15)
401099: 0f b6 92 b0 24 40 00 movzbl  0x4024b0(%rdx),%edx  # 基地址+偏移值出的
字符,传递给%rdx
4010a0: 88 54 04 10       mov     %dl,0x10(%rsp,%rax,1)  # 将%rdx的低8
位放在栈中,正好占1字节
4010a4: 48 83 c0 01       add     $0x1,%rax        # 选择下一个字符
4010a8: 48 83 f8 06       cmp     $0x6,%rax        # 判断是否结束
4010ac: 75 dd            jne     40108b <phase_5+0x29>
4010ae: c6 44 24 16 00     movb    $0x0,0x16(%rsp)    # 如果结束,将栈中的
字符串末尾填上一个空字节
4010b3: be 5e 24 40 00     mov     $0x40245e,%esi    # 第二个参数
(flyers)
```

4010b8:	48 8d 7c 24 10	lea	0x10(%rsp),%rdi	# 第一个参数 (转换完的字符串)
4010bd:	e8 76 02 00 00	callq	401338 <strings_not_equal>	
4010c2:	85 c0	test	%eax,%eax	# 匹配则结束
4010c4:	74 13	je	4010d9 <phase_5+0x77>	
4010c6:	e8 6f 03 00 00	callq	40143a <explode_bomb>	
4010cb:	0f 1f 44 00 00	nopl	0x0(%rax,%rax,1)	
4010d0:	eb 07	jmp	4010d9 <phase_5+0x77>	
4010d2:	b8 00 00 00 00	mov	\$0x0,%eax	# %rax保存字符
标号				
4010d7:	eb b2	jmp	40108b <phase_5+0x29>	
4010d9:	48 8b 44 24 18	mov	0x18(%rsp),%rax	
4010de:	64 48 33 04 25 28 00	xor	%fs:0x28,%rax	
4010e5:	00 00			
4010e7:	74 05	je	4010ee <phase_5+0x8c>	
4010e9:	e8 42 fa ff ff	callq	400b30 <__stack_chk_fail@plt>	
4010ee:	48 83 c4 20	add	\$0x20,%rsp	
4010f2:	5b	pop	%rbx	
4010f3:	c3	retq		

地址 40106a 处 %fs:0x28 是栈保护者的canary值, 下面有个函数 `__stack_chk_fail` 检查是否破坏这个值。(和做题关系不大)

将 `string_length()` 的返回值与6进行比较, 所以得知输入长度为6

对输入的每个字符进行**处理**, 将每个结果逐个放入栈中, 起始地址为 `0x10(%rsp)`, 最后再将末尾放置一个空字符

接着进行字符串比较 `strings_not_equal()`, `%rdi=0x10(%rsp)`, `%rsi=$0x40245e`, 查看待比较的字符串为 `flyers`

- 那么如何对输入的字符串进行处理呢?

将当前字符保存在 `%rdx`, 与 `0xf` 进行与操作, 即只保留低4位, 其余全部置为0, 所以只能取0~15

然后执行 `movzbl 0x4024b0(%rdx),%edx`, 先**间接寻址**, **内存地址**为 `0x4024b0+%rdx` 存放的字符, 保存到 `%rdx` 中, 再将其保存在栈中

```
(gdb) x/s 0x4024b0
0x4024b0 <array.3449>: "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
(gdb)
```

而基地址 `0x4024b0` 处存放的字符串为 `maduiersnfotvbyl`, 刚好包含 `flyers`

因此可以将输入字符的低4位看作偏移量, 使其加上基地址得到的地址, 存放的字符与 `flyers` 进行——对应即可, 即9,15,14,5,6,7

40108f:	88 0c 24	mov	%cl, (%rsp)
401092:	48 8b 14 24	mov	(%rsp), %rdx
401096:	83 e2 0f	and	\$0xf, %edx

虽然取的是字符低8位, 但再和 `0xf` 执行与操作之后, 只保留了低4位, 也即前4位可以为任意值

所以将6个数字整体加上64 (`100xxxx`), 得到73,79,78,69,70,71, 对应字符I,O,N,E,F,G

即输入字符串为 `IONEFG`

Phase_6

phase_6()

```
0000000004010f4 <phase_6>:
4010f4: 41 56                push    %r14
4010f6: 41 55                push    %r13
4010f8: 41 54                push    %r12
4010fa: 55                  push    %rbp
4010fb: 53                  push    %rbx
4010fc: 48 83 ec 50         sub     $0x50,%rsp
401100: 49 89 e5            mov     %rsp,%r13
401103: 48 89 e6            mov     %rsp,%rsi
401106: e8 51 03 00 00      callq   40145c <read_six_numbers> #读6个数字，
存放地址从%rsp开始，每个占4字节
40110b: 49 89 e6            mov     %rsp,%r14
40110e: 41 bc 00 00 00 00    mov     $0x0,%r12d
# 检查输入的6个数字在1到6的范围内
401114: 4c 89 ed            mov     %r13,%rbp
401117: 41 8b 45 00         mov     0x0(%r13),%eax # %r13保存着栈顶地址%rsp
40111b: 83 e8 01            sub     $0x1,%eax # 用第一个数减去1
40111e: 83 f8 05            cmp     $0x5,%eax # 是否小于等于5且大于等于0
401121: 76 05              jbe     401128 <phase_6+0x34>
401123: e8 12 03 00 00      callq   40143a <explode_bomb>
401128: 41 83 c4 01         add     $0x1,%r12d # %r12d=1+%r12d
40112c: 41 83 fc 06         cmp     $0x6,%r12d # 与6相比（检查取值范围的循环边界）
401130: 74 21              je      401153 <phase_6+0x5f>
401132: 44 89 e3            mov     %r12d,%ebx # 1
# 逐个检查数字在取值范围内的同时，检查它后面的所有数字都与其不重复
401135: 48 63 c3            movslq  %ebx,%rax # 1 后面的数不能与前面的数相等
401138: 8b 04 84            mov     (%rsp,%rax,4),%eax # %rax=%rsp+1*4
40113b: 39 45 00            cmp     %eax,0x0(%rbp) # %rbp保存栈顶指针
40113e: 75 05              jne     401145 <phase_6+0x51>
401140: e8 f5 02 00 00      callq   40143a <explode_bomb> # 爆炸（如果%rax指向栈顶指针）
401145: 83 c3 01            add     $0x1,%ebx # 2
401148: 83 fb 05            cmp     $0x5,%ebx
40114b: 7e e8              jle     401135 <phase_6+0x41> # 到6结束，
%rax=%rsp+5*4
40114d: 49 83 c5 04         add     $0x4,%r13 # %r13=%rsp+4
401151: eb c1              jmp     401114 <phase_6+0x20>
# 将输入的每个数（用x代替），原变成7-x
401153: 48 8d 74 24 18      lea     0x18(%rsp),%rsi # %rsi存放数字结束位置
401158: 4c 89 f0            mov     %r14,%rax # %rax=%r14=%rsp
40115b: b9 07 00 00 00      mov     $0x7,%ecx
401160: 89 ca              mov     %ecx,%edx # 将a变为7-a，其他同理
401162: 2b 10              sub     (%rax),%edx
401164: 89 10              mov     %edx,(%rax)
401166: 48 83 c0 04         add     $0x4,%rax
40116a: 48 39 f0            cmp     %rsi,%rax
40116d: 75 f1              jne     401160 <phase_6+0x6c>
# 输入的数字作为标号i，对应node_i，按照输入数字的排序，将node_i也排序，放在起始地址
0x20(%rsp)处，每个node占8个字节
```

```

40116f:  be 00 00 00 00      mov     $0x0,%esi
401174:  eb 21               jmp     401197 <phase_6+0xa3>
401176:  48 8b 52 08         mov     0x8(%rdx),%rdx
40117a:  83 c0 01            add     $0x1,%eax
40117d:  39 c8               cmp     %ecx,%eax
40117f:  75 f5               jne     401176 <phase_6+0x82>
401181:  eb 05               jmp     401188 <phase_6+0x94>
401183:  ba d0 32 60 00      mov     $0x6032d0,%edx      # $0x6032d0 =
0x0000014c
401188:  48 89 54 74 20      mov     %rdx,0x20(%rsp,%rsi,2)
40118d:  48 83 c6 04         add     $0x4,%rsi
401191:  48 83 fe 18         cmp     $0x18,%rsi
401195:  74 14               je      4011ab <phase_6+0xb7>
401197:  8b 0c 34             mov     (%rsp,%rsi,1),%ecx
40119a:  83 f9 01            cmp     $0x1,%ecx          # 两种情况 (1) 7-
x<=1 (2) 7-x>1
40119d:  7e e4               jle     401183 <phase_6+0x8f>
40119f:  b8 01 00 00 00      mov     $0x1,%eax
4011a4:  ba d0 32 60 00      mov     $0x6032d0,%edx
4011a9:  eb cb               jmp     401176 <phase_6+0x82>
# 根据node的顺序, 将其连接起来, 类似链表
4011ab:  48 8b 5c 24 20      mov     0x20(%rsp),%rbx
4011b0:  48 8d 44 24 28      lea     0x28(%rsp),%rax
4011b5:  48 8d 74 24 50      lea     0x50(%rsp),%rsi
4011ba:  48 89 d9             mov     %rbx,%rcx          # 结点p地址
4011bd:  48 8b 10             mov     (%rax),%rdx        # 结点q地址
4011c0:  48 89 51 08         mov     %rdx,0x8(%rcx)    # 将结点q地址放到结点p中
存放指针的位置p->next=q
4011c4:  48 83 c0 08         add     $0x8,%rax
4011c8:  48 39 f0             cmp     %rsi,%rax          # 判断是否结束
4011cb:  74 05               je      4011d2 <phase_6+0xde>
4011cd:  48 89 d1             mov     %rdx,%rcx
4011d0:  eb eb               jmp     4011bd <phase_6+0xc9>
4011d2:  48 c7 42 08 00 00 00 movq     $0x0,0x8(%rdx)    # 尾指针后面为空
4011d9:  00
# 从头遍历链表, 保证每个结点保存的值是按序减小的
4011da:  bd 05 00 00 00      mov     $0x5,%ebp
4011df:  48 8b 43 08         mov     0x8(%rbx),%rax    #取q=p->next (结点p地址
保存在%rbx中, 起始0x20(%rsp))
4011e3:  8b 00               mov     (%rax),%eax        # 结点q的值赋给%eax
4011e5:  39 03               cmp     %eax,(%rbx)        # 将p和q的值进行比较
4011e7:  7d 05               jge     4011ee <phase_6+0xfa> # 保证p>=q
4011e9:  e8 4c 02 00 00      callq   40143a <explode_bomb>
4011ee:  48 8b 5b 08         mov     0x8(%rbx),%rbx    # p=p->next
4011f2:  83 ed 01            sub     $0x1,%ebp          # 计数器
4011f5:  75 e8               jne     4011df <phase_6+0xeb>#
# 结束
4011f7:  48 83 c4 50         add     $0x50,%rsp
4011fb:  5b                 pop     %rbx
4011fc:  5d                 pop     %rbp
4011fd:  41 5c               pop     %r12
4011ff:  41 5d               pop     %r13
401201:  41 5e               pop     %r14
401203:  c3                 retq

```

- 首先读取6个数字, 利用一个双重循环判断每个数字是否符合题目要求
外层6次计算, 内层6-i次

外层判断第*i*个数字是否在1到6这个范围内，内层循环确保当前第*i*个数字后面的数字都与其不相等
就是判断6个数字范围为1-6，并且都不相等（因为跳转 `jbe` 是无符号，所以都是**正数**）

- 接着对每个数字进行原地操作，设某个数字为 `x`，将其变为 `7-x`
- 在栈中利用一段空间存放链表结点（**左边一列为结点值，右边一列为下一个结点的地址**）

```
(gdb) x/12gx 0x6032d0
0x6032d0 <node1>:      0x000000010000014c      0x0000000000006032e0
0x6032e0 <node2>:      0x00000002000000a8      0x0000000000006032f0
0x6032f0 <node3>:      0x0000000300000039c     0x000000000000603300
0x603300 <node4>:      0x000000040000002b3     0x000000000000603310
0x603310 <node5>:      0x000000050000001dd     0x000000000000603320
0x603320 <node6>:      0x000000060000001bb     0x000000000000000000
(gdb)
```

按顺序遍历每个输入的数字（经过 `7-x` 处理后的结果），如果为1的话，则保存 `0x6032d0` 这个地址；如果大于1的话，通过这个操作 `mov 0x8(%rdx),%rdx`，将得到它的下一个结点的地址，相当于 `p=p->next` 操作，而这个操作将执行 `i-1` 次，即数字为 `i` 就对应第 `i` 个结点 `node_i`。存放这一系列结点的地址从 `0x20(%rsp)` 开始，每个结点占据8个字节。

```
(gdb) x/10gx $rsp
0x7fffffffde00: 0x0000000400000003      0x0000000060000005
0x7fffffffde10: 0x0000000200000001      0x0000000000000000
0x7fffffffde20: 0x0000000000006032f0    0x000000000000603300
0x7fffffffde30: 0x000000000000603310    0x000000000000603320
0x7fffffffde40: 0x0000000000006032d0    0x0000000000006032e0
```

上图可以看到，从栈顶 `%rsp` 开始，每4个字节存放一个输入的数字，以此为3,4,5,6,1,2（经过 `7-x` 处理后的）；从 `0x20(%rsp)` 处开始，每8个字节存放一个结点地址。

- 将这些结点依次连接起来

```
mov    %rbx,%rcx      # %rcx为结点p地址
mov    (%rax),%rdx     # %rdx为结点q地址
mov    %rdx,0x8(%rcx)  # 将结点q地址放到结点p中存放指针的位置p->next=q
```

- 从头遍历链表，判断降序排列

```
mov    0x8(%rbx),%rax  # 利用%rax取q=p->next（结点p地址保存在%rbx中，起始0x20(%rsp)）
mov    (%rax),%eax     # 结点q的值赋给%eax
cmp    %eax,(%rbx)     # 将p和q的值进行比较（%eax为q（%rbx）为p）
jge    4011ee <phase_6+0xfa> # 保证p>=q
```

这里 `cmp` 有点不太明白，一个32位，一个64位，暂时将它们都看作32位进行比较


```
(gdb) x *0x6032f0
0x39c: Cannot access memory at address 0x39c
(gdb) x *0x603300
0x2b3: Cannot access memory at address 0x2b3
(gdb) x *0x603310
0x1dd: Cannot access memory at address 0x1dd
(gdb) x *0x603320
0x1bb: Cannot access memory at address 0x1bb
(gdb) x *0x6032d0
0x14c: Cannot access memory at address 0x14c
(gdb) x *0x6032e0
0xa8: Cannot access memory at address 0xa8
(gdb)
```

可以看到结点值为降序排列

- 最终答案

0x6032f0	0x39c	node3	4
0x603300	0x2b3	node4	3
0x603310	0x1dd	node5	2
0x603320	0x1bb	node6	1
0x6032d0	0x14c	node1	6
0x6032e0	0xa8	node2	5

secret_phase

`phase_defused()`

在拆除炸弹的函数中，先检查输入数字的个数是否为6，如果不为6则正常退出，如果为6才进入下面的阶段

接着检查输入的第四行，即第4个炸弹的输入值，原本 `phase_4()` 只检查两个数字，但在这个阶段会这两个数字后面检查一个字符串，即先检查这一行的输入有3个，格式为 `%d %d %s`；接着对第3个字符串进行检查

利用 `strings_not_equal()` 函数将其与地址为 `0x402622` 的字符串进行比较，查看可知该字符串为 `DrEvil`

如果输入字符串不为 `DrEvil` 也没关系，会正常退出，6个炸弹全部拆除，不过不会进入秘密阶段；而如果输入字符串为 `DrEvil`，接下来就会调用 `secret_phase()` 进入秘密阶段！

```
0000000004015c4 <phase_defused>:
4015c4:  48 83 ec 78          sub     $0x78,%rsp
4015c8:  64 48 8b 04 25 28 00  mov     %fs:0x28,%rax
4015cf:  00 00
4015d1:  48 89 44 24 68        mov     %rax,0x68(%rsp)
4015d6:  31 c0                xor     %eax,%eax
4015d8:  83 3d 81 21 20 00 06  cmp     $0x6,0x202181(%rip)      # 603760
<num_input_strings>
                                # 此时%rip=0x4015df, 0x202181(%rip) = *
(0x4015df+0x202181) = *0x603760 = 6
4015df:  75 5e                jne     40163f <phase_defused+0x7b> # 输入个数
不等于6直接结束
4015e1:  4c 8d 44 24 10        lea     0x10(%rsp),%r8
```

```

4015e6:  48 8d 4c 24 0c      lea    0xc(%rsp),%rcx
4015eb:  48 8d 54 24 08      lea    0x8(%rsp),%rdx
4015f0:  be 19 26 40 00      mov     $0x402619,%esi    # %d %d %s
4015f5:  bf 70 38 60 00      mov     $0x603870,%edi    # 7 0 DrEvil
<input_strings+240> (推测为输入的某一行)
4015fa:  e8 f1 f5 ff ff      callq  400bf0 <__isoc99_sscanf@plt>
4015ff:  83 f8 03            cmp     $0x3,%eax        # 要求输入3个数
401602:  75 31              jne     401635 <phase_defused+0x71>
401604:  be 22 26 40 00      mov     $0x402622,%esi    # 此处字符为 DrEvil
401609:  48 8d 7c 24 10      lea    0x10(%rsp),%rdi
40160e:  e8 25 fd ff ff      callq  401338 <strings_not_equal>    # 检查输入的
秘密字符是否为DrEvil
401613:  85 c0              test    %eax,%eax
401615:  75 1e              jne     401635 <phase_defused+0x71>    # 如果为
DrEvil则进入, 否则直接正常结束
401617:  bf f8 24 40 00      mov     $0x4024f8,%edi    # Curses, you've
found the secret phase!
40161c:  e8 ef f4 ff ff      callq  400b10 <puts@plt>
401621:  bf 20 25 40 00      mov     $0x402520,%edi    # But finding it and
solving it are quite different...
401626:  e8 e5 f4 ff ff      callq  400b10 <puts@plt>
40162b:  b8 00 00 00 00      mov     $0x0,%eax
401630:  e8 0d fc ff ff      callq  401242 <secret_phase>
401635:  bf 58 25 40 00      mov     $0x402558,%edi    # Congratulations!
You've defused the bomb!
40163a:  e8 d1 f4 ff ff      callq  400b10 <puts@plt>
40163f:  48 8b 44 24 68      mov     0x68(%rsp),%rax
401644:  64 48 33 04 25 28 00 xor     %fs:0x28,%rax
40164b:  00 00
40164d:  74 05              je      401654 <phase_defused+0x90>
40164f:  e8 dc f4 ff ff      callq  400b30 <__stack_chk_fail@plt>
401654:  48 83 c4 78        add     $0x78,%rsp
401658:  c3                retq

```

此处为输入字符串的第4行, 前两个为炸弹4的密码

```

(gdb) x/s 0x603870
0x603870 <input_strings+240>:  "7 0 DrEvil"
(gdb)

```

可以看到 input_strings 从地址 0x603780 开始, 每80个字节存放一个字符串

```

(gdb) x/s 0x603780
0x603780 <input_strings>:  "Border relations with Canada have never been be
tter."
(gdb) x/s 0x6037d0
0x6037d0 <input_strings+80>:  "1 2 4 8 16 32"
(gdb) x/s 0x603820
0x603820 <input_strings+160>:  "0 207"
(gdb) x/s 0x603870
0x603870 <input_strings+240>:  "7 0 DrEvil"
(gdb) x/s 0x6038c0
0x6038c0 <input_strings+320>:  "IONEFG"
(gdb) x/s 0x603910
0x603910 <input_strings+400>:  "4 3 2 1 6 5"
(gdb) x/s 0x603960
0x603960 <input_strings+480>:  "22"
(gdb) █

```

secret_phase()

首先获得秘密阶段读入的字符串，检查它的值，保证是1到1001之间的整数，否则就爆炸

接着见第一个参数%rdi 设置为地址 0x6030f0，第二个参数%rsi 设置为输入值x，调用 fun7()

只有当 fun7() 的返回值%rax 为2时，才可以解除秘密阶段的炸弹

```
000000000401242 <secret_phase>:
401242: 53                push    %rbx
401243: e8 56 02 00 00    callq   40149e <read_line>    # 读入一个字符串
401248: ba 0a 00 00 00    mov     $0xa,%edx
40124d: be 00 00 00 00    mov     $0x0,%esi
401252: 48 89 c7          mov     %rax,%rdi
401255: e8 76 f9 ff ff    callq   400bd0 <strtol@plt>    # 返回值%rax为输入
的
第一个数字
40125a: 48 89 c3          mov     %rax,%rbx
40125d: 8d 40 ff          lea     -0x1(%rax),%eax
401260: 3d e8 03 00 00    cmp     $0x3e8,%eax            # 0x3e8=1000,即输
入的数字要小于等于1001,大于等于1
401265: 76 05            jbe     40126c <secret_phase+0x2a>
401267: e8 ce 01 00 00    callq   40143a <explode_bomb>
40126c: 89 de            mov     %ebx,%esi              # 输入数字x作为第二
个参数
40126e: bf f0 30 60 00    mov     $0x6030f0,%edi        # 将地址0x6030f0作
为第一个参数
401273: e8 8c ff ff ff    callq   401204 <fun7>          # 调用fun7()
401278: 83 f8 02          cmp     $0x2,%eax              # 返回值为2时，解开
秘密炸弹，否则爆炸
40127b: 74 05            je      401282 <secret_phase+0x40>
40127d: e8 b8 01 00 00    callq   40143a <explode_bomb>
401282: bf 38 24 40 00    mov     $0x402438,%edi        # Wow! You've
defused the secret stage!
401287: e8 84 f8 ff ff    callq   400b10 <puts@plt>
40128c: e8 33 03 00 00    callq   4015c4 <phase_defused>
401291: 5b                pop     %rbx
401292: c3                retq
```

fun7()

地址 0x6030f0 作为 fun7() 的第一个参数，保存着一个二叉搜索树，每个结点占32个字节

每个结点中以8个字节为1个单位，依次为结点值、左叶子结点地址、右叶子结点地址 和 0x0

要求返回值%rax 为2才可以将炸弹拆除

```
000000000401204 <fun7>:
401204: 48 83 ec 08       sub     $0x8,%rsp
401208: 48 85 ff          test    %rdi,%rdi            # 当前结点不为空（结点中
的值不为0）
40120b: 74 2b            je      401238 <fun7+0x34>    # 否则直接将%rax置为
0xffffffff
40120d: 8b 17            mov     (%rdi),%edx           # 读取结点中的值给%rdx
40120f: 39 f2            cmp     %esi,%edx             # 将%rdx和输入的秘密数字
x作比较
```

401211: 7e 0d	jle	401220 <fun7+0x1c> # 如果当前结点值小于等于x, 跳转401220
401213: 48 8b 7f 08	mov	0x8(%rdi),%rdi # 当前结点值大于x, 向左搜索 (左叶子结点)
401217: e8 e8 ff ff ff	callq	401204 <fun7> # 递归(左)入口
40121c: 01 c0	add	%eax,%eax # 递归(左)出口,
将%rax*2		
40121e: eb 1d	jmp	40123d <fun7+0x39> # 结束本次递归
401220: b8 00 00 00 00	mov	\$0x0,%eax # 接上面未进入递归情况时的跳转
401225: 39 f2	cmp	%esi,%edx # 将当前结点值和x比较
401227: 74 14	je	40123d <fun7+0x39> # 如果等于则结束本轮过程
401229: 48 8b 7f 10	mov	0x10(%rdi),%rdi # 不等于, 当前结点值小于x, 向右搜索 (右叶子结点)
40122d: e8 d2 ff ff ff	callq	401204 <fun7> # 递归(右)入口
401232: 8d 44 00 01	lea	0x1(%rax,%rax,1),%eax # 递归(右)出口
401236: eb 05	jmp	40123d <fun7+0x39> # 结束本次递归
401238: b8 ff ff ff ff	mov	\$0xffffffff,%eax
40123d: 48 83 c4 08	add	\$0x8,%rsp
401241: c3	retq	

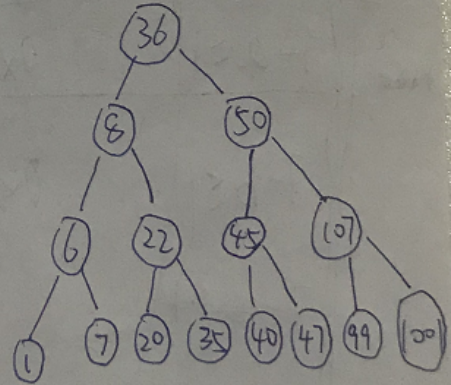
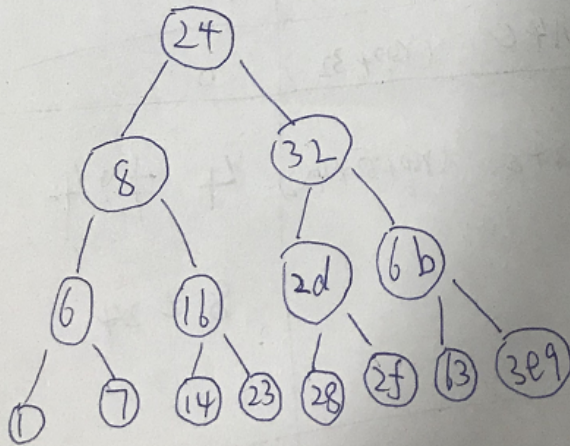
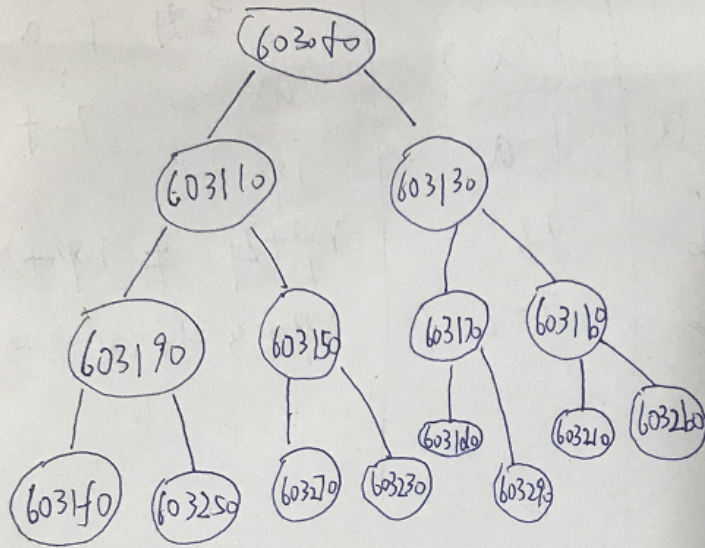
递归过程

- x=22

%rdx 保存当前结点内的值, 初始为36

- 36>22, 进入左子树, 通过**入口1** 401217: callq 401204 <fun7>, 进入**递归1(左)**
- 8<=22, 不进入左子树, 跳转, 将%rax置为0, 但8!=22, 进入右子树, 通过**入口2** 401229: mov 0x10(%rdi),%rdi, 进入**递归2(右)**
- 22<=22, 不进入左子树, 跳转, 将%rax置为0, 22==22, 结束本轮**递归2**
- 返回到**入口2**的下一条指令, 将%rax*2+1=1, 结束本轮**递归1**
- 返回到**入口1**的下一条指令, 将%rax*2=2, 结束整个调用, 返回%rax=2

```
(gdb) x/60xg 0x6030f0
0x6030f0 <n1>: 0x0000000000000024      0x000000000000603110
0x603100 <n1+16>: 0x0000000000603130      0x0000000000000000
0x603110 <n21>: 0x0000000000000008      0x000000000000603190
0x603120 <n21+16>: 0x0000000000603150      0x0000000000000000
0x603130 <n22>: 0x0000000000000032      0x000000000000603170
0x603140 <n22+16>: 0x00000000006031b0      0x0000000000000000
0x603150 <n32>: 0x0000000000000016      0x000000000000603270
0x603160 <n32+16>: 0x0000000000603230      0x0000000000000000
0x603170 <n33>: 0x000000000000002d      0x0000000000006031d0
0x603180 <n33+16>: 0x0000000000603290      0x0000000000000000
0x603190 <n31>: 0x0000000000000006      0x0000000000006031f0
0x6031a0 <n31+16>: 0x0000000000603250      0x0000000000000000
0x6031b0 <n34>: 0x000000000000006b      0x000000000000603210
0x6031c0 <n34+16>: 0x00000000006032b0      0x0000000000000000
0x6031d0 <n45>: 0x0000000000000028      0x0000000000000000
0x6031e0 <n45+16>: 0x0000000000000000      0x0000000000000000
0x6031f0 <n41>: 0x0000000000000001      0x0000000000000000
0x603200 <n41+16>: 0x0000000000000000      0x0000000000000000
0x603210 <n47>: 0x0000000000000063      0x0000000000000000
0x603220 <n47+16>: 0x0000000000000000      0x0000000000000000
0x603230 <n44>: 0x0000000000000023      0x0000000000000000
0x603240 <n44+16>: 0x0000000000000000      0x0000000000000000
0x603250 <n42>: 0x0000000000000007      0x0000000000000000
0x603260 <n42+16>: 0x0000000000000000      0x0000000000000000
0x603270 <n43>: 0x0000000000000014      0x0000000000000000
0x603280 <n43+16>: 0x0000000000000000      0x0000000000000000
0x603290 <n46>: 0x000000000000002f      0x0000000000000000
0x6032a0 <n46+16>: 0x0000000000000000      0x0000000000000000
0x6032b0 <n48>: 0x000000000000003e9      0x0000000000000000
0x6032c0 <n48+16>: 0x0000000000000000      0x0000000000000000
(gdb) █
```

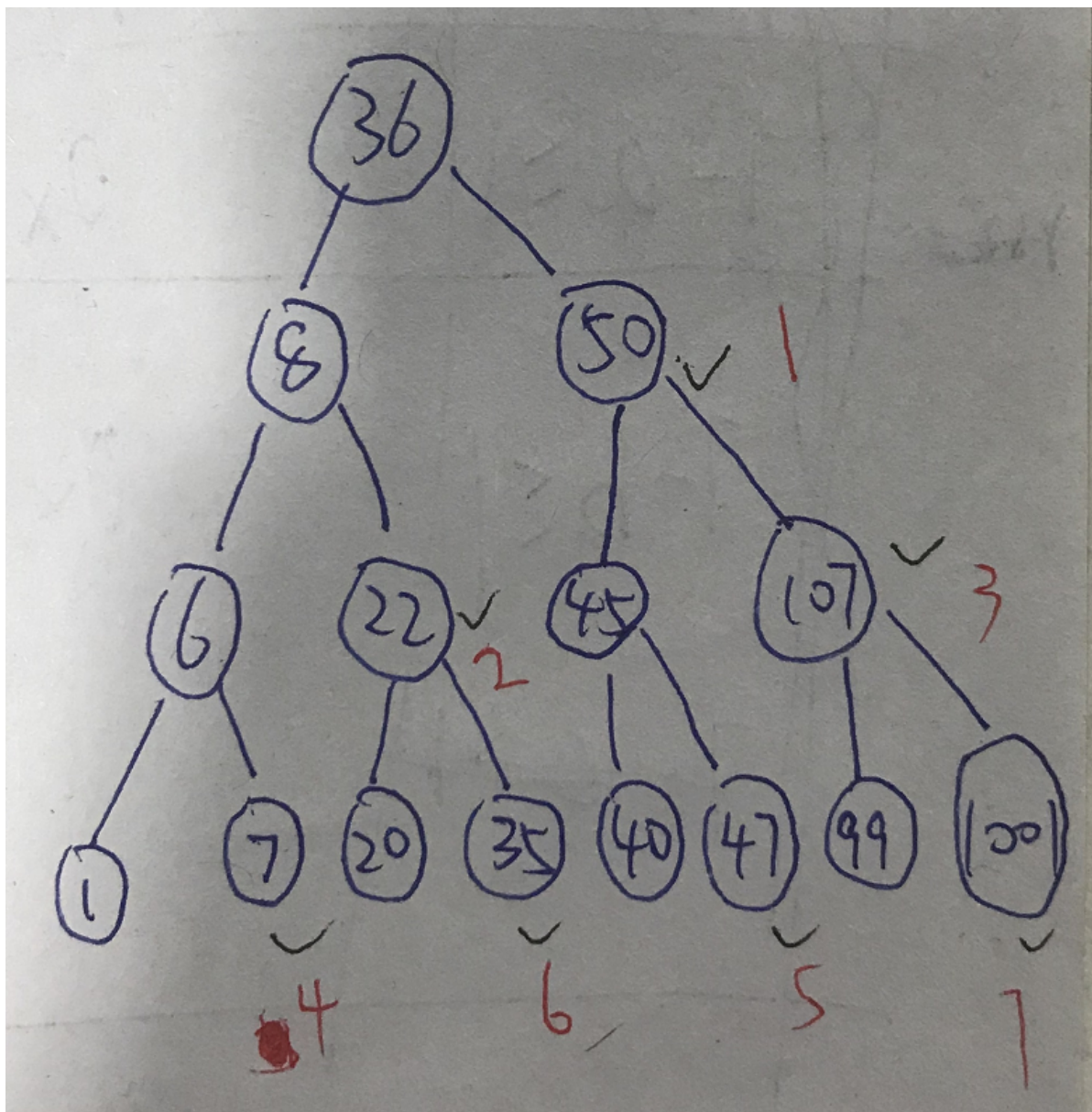
```

Contents of section .data:
6030e0 00000000 00000000 00000000 00000000
6030f0 24000000 00000000 10316000 00000000
603100 30316000 00000000 00000000 00000000
603110 08000000 00000000 90316000 00000000
603120 50316000 00000000 00000000 00000000
603130 32000000 00000000 70316000 00000000
603140 b0316000 00000000 00000000 00000000
603150 16000000 00000000 70326000 00000000
603160 30326000 00000000 00000000 00000000
603170 2d000000 00000000 d0316000 00000000
603180 90326000 00000000 00000000 00000000
603190 06000000 00000000 f0316000 00000000
6031a0 50326000 00000000 00000000 00000000
6031b0 6b000000 00000000 10326000 00000000
6031c0 b0326000 00000000 00000000 00000000
6031d0 28000000 00000000 00000000 00000000
6031e0 00000000 00000000 00000000 00000000
6031f0 01000000 00000000 00000000 00000000
603200 00000000 00000000 00000000 00000000
603210 63000000 00000000 00000000 00000000
603220 00000000 00000000 00000000 00000000
603230 23000000 00000000 00000000 00000000
603240 00000000 00000000 00000000 00000000
603250 07000000 00000000 00000000 00000000
603260 00000000 00000000 00000000 00000000
603270 14000000 00000000 00000000 00000000
603280 00000000 00000000 00000000 00000000
603290 2f000000 00000000 00000000 00000000
6032a0 00000000 00000000 00000000 00000000
6032b0 e9030000 00000000 00000000 00000000
6032c0 00000000 00000000 00000000 00000000

```

如果输入值不在二叉搜索树中，返回 %rax 为负数；

为了使 %rax 不为0，该结点应该为其父节点的右叶子结点



如上图，未标号的都是返回值 `%rax=0`，其余情况只有结点22才会返回 `%rax=2`

总结上面的过程：`%rdi` 指向一个树的节点，令 `%rdi` 节点的值与我们读入的值进行比较。

- 如果两者相等：返回0
- 如果前者大于后者：`%rdi` 移至左子树，返回 `2 * %rax`
- 如果后者大于前者：`%rdi` 移至右子树，返回 `2 * %rax + 1`

那么我们需要返回2，应该在最后一次调用返回0，倒数第二次调用返回 `2 * %rax + 1`，第一次调用返回 `2 * %rax`。因此，这个数应该在第三层，比父节点大且比根节点小。观察上图，唯一的答案是：

0x16 (22)

总结

```
root@ubuntu:/home/csapp/Desktop/Lab2/bomb# ./bomb answer.txt
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!
root@ubuntu:/home/csapp/Desktop/Lab2/bomb#
```

这个lab对于我来说真的收获很大

首先学会了 `gdb` 调试的基本操作，对程序的执行过程更加清晰明了，尤其是寄存器值的变化，也真正理解了栈的强大作用

其次，对于递归、链表和二叉树，在汇编角度有了更深的认识，尤其是递归和栈的联系，递归的入口和出口

在内存中，间接地址和地址偏移对于数组所起到的作用

总之，虽然花费了我非常多的时间，但还是觉的很值得！