

# 中国海洋大学 计算机科学与技术系

## 实验报告

姓名：岳宇轩      学号：19020011038      专业：慧与

科目：计算机系统基础      题目：Lab2 Binary Bombs

实验时间：      12.25

实验成绩：      实验教师：孙鑫

### 一、实验目的：

- 1 增强对程序机器级表示、汇编语言、调试器和逆向工程的理解。
2. 熟悉 linux 基本操作命令，其中常用工具和程序开发环境
3. 炸弹运行各阶段要求输入一个字符串，若输入符合程序预期，该阶段炸弹被“拆除”，

否则“爆炸”。

4. 熟悉阶段 1:字符串比较    2:循环    3:条件/分支:含 switch 语句    4:递归调用和  
栈

5:指针

6:链表/指针/结构

### 二、实验要求

1. 熟悉 GDB 调试基本操作命令，还有其中常用工具和程序开发环境
2. 熟练使用 objdummp
3. 单步跟踪调试每一阶段的机器代码
4. 理解汇编语言代码的行为或作用
5. “推断”拆除炸弹所需的目标字符串
6. 设置断点进行调试

# 实验内容

-----先附的图，图下面是说明-----

## 1.

```
00000d30 <phase_1>:
    d30:      55                push    %ebp
    d31:    89 e5              mov     %esp,%ebp
    d33:      53                push    %ebx
    d34:    83 ec 0c            sub     $0xc,%esp
    d37:    e8 34 fd ff ff      call    a70 <_x86.get_pc_thunk.bx>
    d3c:    81 c3 c4 32 00 00    add     $0x32c4,%ebx
    d42:    8d 83 84 e3 ff ff    lea     -0x1c7c(%ebx),%eax
    d48:      50                push    %eax
    d49:    ff 75 08            pushl   0x8(%ebp)
    d4c:    e8 42 05 00 00      call    1293 <strings_not_equal>
    d51:    83 c4 10            add     $0x10,%esp
    d54:    85 c0                test    %eax,%eax
    d56:    75 05                jne     d5d <phase_1+0x2d>
    d58:    8b 5d fc            mov     -0x4(%ebp),%ebx
    d5b:    c9                  leave   %ebx
    d5c:    c3                  ret
    d5d:    e8 74 06 00 00      call    13d6 <explode_bomb>
    d62:    eb f4                jmp     d58 <phase_1+0x28>
```

分析可知，d49 命令 push 了第一个参数，也就是用户输入的值，d48push 的值应该就是匹配字符串的位置，所以用 layout regs 命令查看运行至 d48 时 eax 寄存器的值，发现是

```
ics@debian: ~/Downloads/bomb15
File Edit View Search Terminal Help
Register group: general
eax      0x402384 4203396
ecx      0x5      5
edx      0x1      1
ebx      0x404000 4210688
esp      0xbffff2f8 0xbffff2f8
ebp      0xbffff308 0xbffff308

> 0x400d3c <phase_1+12> add $0x32c4,%ebx
> 0x400d42 <phase_1+18> lea -0x1c7c(%ebx),%eax
> 0x400d48 <phase_1+24> push %eax
0x400d49 <phase_1+25> pushl 0x8(%ebp)
0x400d4c <phase_1+28> call 0x401293 <strings_not_equal>
0x400d51 <phase_1+33> add $0x10,%esp

native process 1857 In: phase 1 L?? PC: 0x400d48
0x00400d37 in phase_1 ()
(gdb) ni
0x00400d3c in phase_1 ()
(gdb) ni
0x00400d42 in phase_1 ()
(gdb) ni
0x00400d48 in phase_1 ()
(gdb)
```

0x402384, 然后用 x/2s 0x402384 找到字符串

```
(gdb) x/2s 0x402384
0x402384: "There are rumors on the internets."
```

即为答案:There are rumors on the internets.

## 2.

```
d7f: e8 8a 06 00 00      call 140e <read_six_numbers>
```

d7f call 了一个函数是 read\_six\_number, 分析这个函数可知它的功能是读取了六个数并且存放在一个字符串里了。

```
d87: 83 7d d0 00      cmpl $0x0, -0x30(%ebp)
```

找到 d87 行, 将 ebp-30 与 0 比较, 可知 ebp-30 是第一个数的地址, 立即数 0 是第一个数

```
d8b: 78 0a      js d97 <phase_2+0x33>
d8d: be 01 00 00 00      mov $0x1, %esi
d92: 8d 7d d0      lea -0x30(%ebp), %edi
```

接下来看到 d8b 行有个 js d97<phase\_2+0x33>, 意思是如果 SF 标志位为 1, 则跳转至 phase\_2+0x33 处的位置, 找到这个位置:

```
d97: e8 3a 06 00 00      call 13d6 <explode_bomb>
```

一看, 哦, 原来是爆炸了。结合 d87 可以得出: 如果第一个数是负数则爆炸。

然后 d8d 行给 esi 赋值 1, 用作计数; 给 edi 赋值第一个数的位置作为基址。

```
da6: 89 f0      mov %esi, %eax
da8: 03 44 b7 fc      add -0x4(%edi, %esi, 4), %eax
dac: 39 04 b7      cmp %eax, (%edi, %esi, 4)
```

代码的核心是这两行。分析可知, 第 n 个数需要是第 n-1 个数+n-1

```
daf: 74 ed      je d9e <phase_2+0x3a>
db1: e8 20 06 00 00      call 13d6 <explode_bomb>
```

相同则跳转至 phase\_2+0x3a, 不同则爆炸

```
d9e: 83 c6 01      add $0x1, %esi
da1: 83 fe 06      cmp $0x6, %esi
da4: 74 12      je db8 <phase_2+0x54>
```

phase\_2+0x3a 处执行的操作是: 循环计数器+1, 并与 6 进行比较, 从而作为循环是否结束的依据。

综上可知, 答案为: 第一个数不是负数, 第 n 个数的值第 n-1 个数的值+n-1 (2<=n<=6)。

### 3.

```
dd2:  8d 45 f0          lea    -0x10(%ebp), %eax
dd5:  50                push   %eax
dd6:  8d 45 ef          lea    -0x11(%ebp), %eax
dd9:  50                push   %eax
dda:  8d 45 f4          lea    -0xc(%ebp), %eax
ddd:  50                push   %eax
```

压入了 3 个参数

The screenshot shows a GDB terminal window titled "ics@debian: ~/Downloads/bomb15". The "Register group: general" section displays the following values:   
eax: 0x4023ce 4203470   
ecx: 0x3 3   
edx: 0x3 3   
ebx: 0x404000 4210688   
esp: 0xbffff2d8 0xbffff2d8   
ebp: 0xbffff308 0xbffff308   
Below the registers, a list of instructions is shown with their addresses and disassembled code:   
0x400ddd <phase\_3+29> push %eax   
0x400dde <phase\_3+30> lea -0x1c32(%ebx), %eax   
0x400de4 <phase\_3+36> push %eax   
0x400de5 <phase\_3+37> pushl 0x8(%ebp)   
0x400de8 <phase\_3+40> call 0x400980 <\_\_isoc99\_sscanf@plt>   
0x400ded <phase\_3+45> add \$0x20, %esp   
At the bottom, the GDB prompt shows the current state: "native process 2623 In: phase\_3 L?? PC: 0x400de4". The user has entered several commands: (gdb) ni, (gdb) ni, (gdb) x/2s 0x4023ce, and (gdb) ni, with the output showing the memory contents at those addresses.

查看格式字符串，是一个整型，一个字符型，一个整型

```
df5:  83 7d f4 07          cmpl   $0x7, -0xc(%ebp)
df9:  0f 87 f7 00 00 00      ja     ef6 <.L24+0x1a>
```

接着是比较了第一个参数和 7，如果第一个参数-7>0 则跳转至 ef6

```
ef6:  e8 db 04 00 00          call   13d6 <explode_bomb>
```

ef6 爆炸了，说明第一个参数要小于等于 7

```
e0b:  ff e2                  jmp     *%edx
```

继续看，输入第一个参数是 3 时，这里跨段跳转到了 edx 处，通过 gdb 单步调试找到 edx 中的地址为 0x400e74

```

ics@debian: ~/Downloads/bomb15
File Edit View Search Terminal Help
Register group: general
eax      0x3      3
ecx      0x40     64
edx      0x400e74 4198004
ebx      0x404000 4210688
esp      0xbffff2f0 0xbffff2f0
ebp      0xbffff308 0xbffff308
> 0x400e02 <phase_3+66> mov %ebx,%edx
0x400e04 <phase_3+68> add -0x1c20(%ebx,%eax,4),%edx
> 0x400e0b <phase_3+75> jmp *%edx
0x400e0d <phase_3+77> call 0x4013d6 <explode_bomb>
0x400e12 <phase_3+82> jmp 0x400df5 <phase_3+53>
0x400e14 <phase_3+84> mov $0x75,%eax
native process 1816 In: phase 3 L?? PC: 0x400e0b
0x00400dff in phase_3 ()
(gdb) ni
0x00400e02 in phase_3 ()
(gdb) ni
0x00400e04 in phase_3 ()
(gdb) ni
0x00400e0b in phase_3 ()
(gdb)

```

e74 处为. L20

```

e74:  b8 68 00 00 00      mov    $0x68,%eax
e79:  81 7d f0 4a 02 00 00  cmpl   $0x24a,-0x10(%ebp)
e80:  74 7e                je     f00 <.L24+0x24>
e82:  e8 4f 05 00 00      call  13d6 <explode_bomb>

```

此时 eax 中存的是第一个参数，e74: eax=eax+0x68

e79: 比较第三个参数与 0x24a，不同则爆炸，所以第三个参数应该是 0x24a = 586

```

e87:  b8 68 00 00 00      mov    $0x68,%eax
e8c:  eb 72                jmp    f00 <.L24+0x24>

```

接着又把 0x68 放到了 eax 中，跳转到 f00

```

f00:  3a 45 ef            cmp    -0x11(%ebp),%al
f03:  74 05                je     f0a <.L24+0x2e>
f05:  e8 cc 04 00 00      call  13d6 <explode_bomb>
f0a:  8b 5d fc            mov    -0x4(%ebp),%ebx
f0d:  c9                  leave
f0e:  c3                  ret

```

把 al 中的值和第二个参数比较（0x68 都在 al 里了），不同则爆炸，所以第二个参数应该是 0x68，即为 0110 1000，对照 ASCII 码表，找到是字符 'h'。所以三个数都找到了，输入 **3 h 586**，结果正确。

下面是输入第一个值为 0,1,2,4,5,6,7 时的情形，过程相似，不再赘述，只给出最终结果：

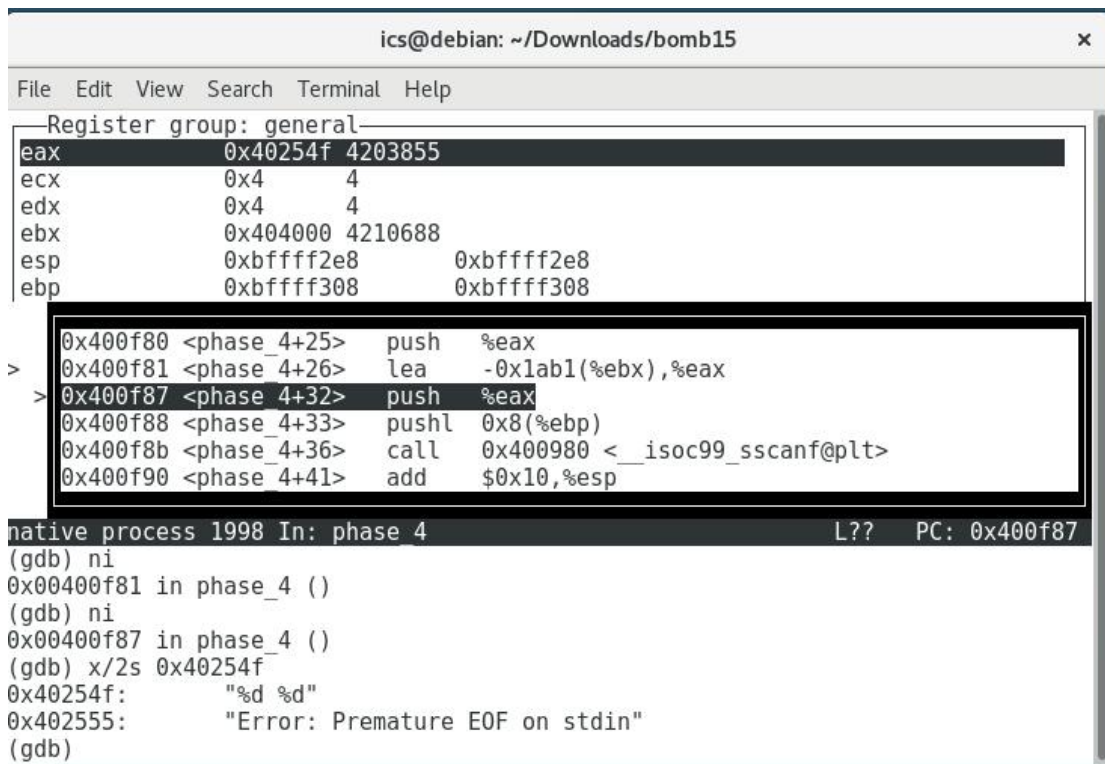
**0 u 332   1 x 116   2 v 621   4 u 543   5 n 163   6 p 660   7 k 962**



## 4.

```
lea    -0x10(%ebp), %eax
push   %eax
lea    -0xc(%ebp), %eax
push   %eax
```

分析上面代码可知，压入了两个参数



```
ics@debian: ~/Downloads/bomb15
File Edit View Search Terminal Help
Register group: general
eax      0x40254f 4203855
ecx      0x4      4
edx      0x4      4
ebx      0x404000 4210688
esp      0xbffff2e8 0xbffff2e8
ebp      0xbffff308 0xbffff308
> 0x400f80 <phase_4+25> push %eax
> 0x400f81 <phase_4+26> lea -0x1ab1(%ebx), %eax
> 0x400f87 <phase_4+32> push %eax
0x400f88 <phase_4+33> pushl 0x8(%ebp)
0x400f8b <phase_4+36> call 0x400980 <__isoc99_sscanf@plt>
0x400f90 <phase_4+41> add $0x10, %esp
native process 1998 In: phase_4 L?? PC: 0x400f87
(gdb) ni
0x00400f81 in phase_4 ()
(gdb) ni
0x00400f87 in phase_4 ()
(gdb) x/2s 0x40254f
0x40254f: "%d %d"
0x402555: "Error: Premature EOF on stdin"
(gdb)
```

找到输入格式为两个整型

```
cmp     $0x2, %eax
je      fbe <phase_4+0x57>
call    13d6 <explode_bomb>
```

将参数个数与 2 进行比较，不同则爆炸，说明只能输入两个参数  
相同则跳转至 fbe:

```
fbe:    83 7d f4 0e          cmpl    $0xe, -0xc(%ebp)
fc2:    76 d9              jbe     f9d <phase_4+0x36>
fc4:    eb d2              jmp     f98 <phase_4+0x31>
```

比较第一个参数和 0xe, 若第一个参数大于 e 则跳转至 f98, 查看 f98 是爆炸，所以第一个参数要  $\leq 0xe$ ，也就是小于等于 14，这样就来到了 f9d:

f9d:	83 ec 04	sub	\$0x4, %esp
fa0:	6a 0e	push	\$0xe
fa2:	6a 00	push	\$0x0
fa4:	ff 75 f4	pushl	-0xc(%ebp)
fa7:	e8 63 ff ff ff	call	f0f <func4>

可以看到，在 f9d 处压入了三个参数，分别是 0xe,0x0,和原参数 1,然后 call 了一个 func4，所以去看 fun4 的调用执行了什么

f14:	8b 45 08	mov	0x8(%ebp), %eax
f17:	8b 55 0c	mov	0xc(%ebp), %edx
f1a:	8b 4d 10	mov	0x10(%ebp), %ecx

设 func4 的三个参数分别为 x,y,z，则执行完上面的语句后把 x 放到了 eax 寄存器中，把 y 放到了 edx 寄存器中，把 z 放到 ecx 寄存器中

f1d:	89 ce	mov	%ecx, %esi
f1f:	29 d6	sub	%edx, %esi
f21:	89 f3	mov	%esi, %ebx
f23:	c1 eb 1f	shr	\$0x1f, %ebx
f26:	01 f3	add	%esi, %ebx
f28:	d1 fb	sar	%ebx
f2a:	01 d3	add	%edx, %ebx

接下来进行一系列运算：先 z-y,结果放入 ebx，ebx 逻辑右移 31 位后再加上 z-y，然后 ebx 算数右移 1 位，相当于 ebx 中的值除以 2，最终结果再+y。

f2c:	39 c3	cmp	%eax, %ebx
f2e:	7f 0d	jg	f3d <func4+0x2e>
f30:	39 c3	cmp	%eax, %ebx
f32:	7c 1e	jl	f52 <func4+0x43>
f34:	89 d8	mov	%ebx, %eax

如果 ebx > eax（第一个参数），则跳转至 f3d，如果 ebx < eax，则跳转至 f52.如果 ebx = eax，则将 ebx 值放入 eax 中返回。

f3d 中：

f3d:	83 ec 04	sub	\$0x4, %esp
f40:	8d 4b ff	lea	-0x1(%ebx), %ecx
f43:	51	push	%ecx
f44:	52	push	%edx
f45:	50	push	%eax
f46:	e8 c4 ff ff ff	call	f0f <func4>

可以看到这里在 func4 中又调用了 func4，属于递归调用，三个参数分别是 x,y,ebx-1

```

f52:  83 ec 04          sub    $0x4,%esp
f55:  51               push   %ecx
f56:  8d 53 01         lea    0x1(%ebx),%edx
f59:  52             push   %edx
f5a:  50             push   %eax
f5b:  e8 af ff ff ff   call   f0f <func4>

```

f52 中，也是递归调用，三个参数分别为 x,ebx+1,z

```

fa7:  e8 63 ff ff ff   call   f0f <func4>
fac:  83 c4 10         add    $0x10,%esp
faf:  83 f8 0d         cmp    $0xd,%eax
fb2:  74 12           je     fc6 <phase_4+0x5f>
fb4:  e8 1d 04 00 00   call   13d6 <explode_bomb>

```

phase\_4 中，将 phase\_4 调用的 func4 的返回值与 0xd 比较，不同则爆炸，相同跳转至 fc6

```

fc6:  83 7d f0 0d      cmpl   $0xd,-0x10(%ebp)
fca:  75 e8           jne    fb4 <phase_4+0x4d>

```

fc6 中将输入的第二个参数与 0xd 比较，不相同则跳转至 fb4，fb4 是爆炸，所以第二个参数应该是 0xd，即 13，第一个参数与 0,0xe 作为三个参数调用 func4 的返回值应该是 0xd。在上面已经分析过 func4 的代码了，所以容易得到当第一个输入为 2 时递归调用的最终返回值为 0xd，所以答案为 **2 13**

## 5.

```

fde:  8b 75 08        mov    0x8(%ebp),%esi
fel:  83 ec 0c        sub    $0xc,%esp
fe4:  56             push   %esi
fe5:  e8 87 02 00 00   call   1271 <string_length>

```

将输入的字符串放入 esi 中并压栈，调用字符串长度函数，字符串长度返回值放在 eax 中

```

fea:  83 c4 10        add    $0x10,%esp
fed:  83 f8 06        cmp    $0x6,%eax
ff0:  74 05          je     ff7 <phase_5+0x29>
ff2:  e8 df 03 00 00   call   13d6 <explode_bomb>

```

esp+0x10 是开栈，比较 eax 中的值（即输入字符串的长度）和 0x6，如果长度不等于 6，就要爆炸，说明输入的字符串要是长度为 6 的。

```

ff7:  89 f0          mov    %esi,%eax
ff9:  83 c6 06        add    $0x6,%esi
ffc:  b9 00 00 00 00   mov    $0x0,%ecx

```



将 esi 中的值（输入字符串的地址）赋给 eax，然后使 esi 中的值加 6，给 eax 初始值为 0x0

```
1001: 0f b6 10 movzbl (%eax), %edx
1004: 83 e2 0f and $0xf, %edx
1007: 03 8c 93 00 e4 ff ff add -0x1c00(%ebx, %edx, 4), %ecx
```

取 eax 寄存器中地址处的值无符号扩展至双字，放入 edx，edx 与 0xf 进行与操作，相当于取后四位

然后 1007 进行了一次基址+比例变址+偏移 寻址，其中基址+偏移就是比对数组的首地址，该行的含义是：根据“输入字符串的某个字符，其对应 ASCII 码表二进制表示的后四位”，在一个已知数组中找到相应位置的数，使 ecx+=这个数。

```
100e: 83 c0 01 add $0x1, %eax
1011: 39 f0 cmp %esi, %eax
1013: 75 ec jne 1001 <phase_5+0x33>
```

循环体执行结束,ecx+=1，与 esi 比较，不相同则回到 1001 重新执行循环。这三句是循环计数器自增和循环条件的判断。

```
1015: 83 f9 2a cmp $0x2a, %ecx
1018: 74 05 je 101f <phase_5+0x51>
101a: e8 b7 03 00 00 call 13d6 <explode_bomb>
```

最终结果要使 ecx 中的值等于 0x2a(十进制 42)。

通过 gdb 调试，找到刚才基址+比例变址+偏移 寻址中的基址+偏移处的数组

```
ics@debian: ~/Downloads/bomb15
File Edit View Search Terminal Help
Register group: general
eax 0x404500 4211968
ecx 0xa 10
edx 0x1 1
ebx 0x404000 4210688
esp 0xbffff300 0xbffff300
ebp 0xbffff308 0xbffff308
> 0x401004 <phase_5+54> and $0xf,%edx
0x401007 <phase_5+57> add -0x1c00(%ebx,%edx,4),%ecx
> 0x40100e <phase_5+64> add $0x1,%eax
0x401011 <phase_5+67> cmp %esi,%eax
0x401013 <phase_5+69> jne 0x401001 <phase_5+51>
0x401015 <phase_5+71> cmp $0x2a,%ecx
native process 2797 In: phase 5 L?? PC: 0x40
0x00401004 in phase_5 ()
(gdb) ni
0x00401007 in phase_5 ()
(gdb) ni
0x0040100e in phase_5 ()
(gdb) p *(0x402400)@16
$1 = {2, 10, 6, 1, 12, 16, 9, 3, 4, 7, 14, 5, 11, 8, 15, 13}
(gdb)
```

a[16]={2,10,6,1,12, 16,9,3,4,7,14,5,11,8,15,13}

选取对应的组合即可，比如在这里我选了 p(对应 a[0]=2),a(对应 a[1]=10),c(对应 a[3]=1),c(对应 a[3]=1),d(对应 a[4]=12),e(对应 a[5]=16), 2+10+1+1+12+16=42,所以本关的一个解为 **paccde**  
当然还有很多其他组合

## 6.

1026:	55	push	%ebp
1027:	89 e5	mov	%esp,%ebp
1029:	57	push	%edi
102a:	56	push	%esi
102b:	53	push	%ebx
102c:	83 ec 54	sub	\$0x54,%esp
102f:	e8 3c fa ff ff	call	a70 <__x86.get_pc_thunk.bx>
1034:	81 c3 cc 2f 00 00	add	\$0x2fcc,%ebx
103a:	8d 45 d0	lea	-0x30(%ebp),%eax
103d:	50	push	%eax
103e:	ff 75 08	pushl	0x8(%ebp)
1041:	e8 c8 03 00 00	call	140e <read_six_numbers>

读六个数，存到 `ebp-30` 处，设为 `a[6]`

1046:	83 c4 10	add	\$0x10,%esp
1049:	c7 45 b4 00 00 00 00	movl	\$0x0,-0x4c(%ebp)
1050:	eb 25	jmp	1077 <phase_6+0x51>

0 放 `ebp-4c`，跳转至 1077

1077:	8b 45 b4	mov	-0x4c(%ebp),%eax
107a:	8b 44 85 d0	mov	-0x30(%ebp,%eax,4),%eax
107e:	89 45 b0	mov	%eax,-0x50(%ebp)
1081:	83 e8 01	sub	\$0x1,%eax
1084:	83 f8 05	cmp	\$0x5,%eax
1087:	76 c9	jbe	1052 <phase_6+0x2c>
1089:	e8 48 03 00 00	call	13d6 <explode bomb>

`ebp-4c` 放 `eax`，`a[0]`放 `eax`，送到 `ebp-50` 处，然后 `eax` 自减 1。比较 `eax` 和 5，如果 `eax` 中的值小于等于 5，则跳转至 1052，否则爆炸，说明 `a[0]-1` 要小于等于 5，即 `a[0]<=6`。

1052:	83 45 b4 01	addl	\$0x1,-0x4c(%ebp)
1056:	8b 45 b4	mov	-0x4c(%ebp),%eax
1059:	83 f8 06	cmp	\$0x6,%eax
105c:	74 39	je	1097 <phase_6+0x71>
105e:	89 c6	mov	%eax,%esi
1060:	8d 7d d0	lea	-0x30(%ebp),%edi

来到 1052，`ebp-4c` 自增 1，后放入 `eax`，此时 `eax=1`，比较 `eax` 中的值和 6，如果相同则跳转至 1097，推断这是循环 6 次计数的标志。因为 `1<6`，所以这里我们继续执行。把 `eax` 中的值放入 `esi`，此时 `esi=1`，加载 `ebp-30` 有效地址进 `edi`，此时 `edi` 作为对数组进行寻址的基址

1063:	8b 45 b4	mov	-0x4c(%ebp), %eax
1066:	8b 0c b7	mov	(%edi, %esi, 4), %ecx
1069:	39 4c 87 fc	cmp	%ecx, -0x4(%edi, %eax, 4)
106d:	74 21	je	1090 <phase_6+0x6a>
106f:	83 c6 01	add	\$0x1, %esi
1072:	83 fe 05	cmp	\$0x5, %esi
1075:	7e ec	jle	1063 <phase_6+0x3d>

ebp-4c 送 eax,此时 eax=1,a[esi]送 ecx(即 a[1]),a[1]和 a[0]比较, 相等则跳转至 1090

1090:	e8 41 03 00 00	call	13d6 <explode_bomb>
-------	----------------	------	---------------------

爆炸, 所以 a[1]和 a[0]不能相等。esi+=1, 小于等于 5 则跳转至 1063, 可知这这也是一个循环计数的判断。执行完五次后, 得到 a[1]!=a[0],a[2]!=a[0],a[3]!=a[0],a[4]!=a[0],a[5]!=a[0]。

1077:	8b 45 b4	mov	-0x4c(%ebp), %eax
107a:	8b 44 85 d0	mov	-0x30(%ebp, %eax, 4), %eax
107e:	89 45 b0	mov	%eax, -0x50(%ebp)
1081:	83 e8 01	sub	\$0x1, %eax
1084:	83 f8 05	cmp	\$0x5, %eax
1087:	76 c9	jbe	1052 <phase_6+0x2c>
1089:	e8 48 03 00 00	call	13d6 <explode_bomb>

内层循环完成后, 程序继续执行, ebp-4c 送 eax,a[esi]送 eax,此时 eax 存的是 a[1], 再把 eax 送 ebp-50。a[1]-1 与 5 比, 小于等于 5 则跳转 1052, 大于 5 则爆炸。所以 a[1]的值应该<=6。跳转至 1052 后, 有重复了上述的操作。对于上述过程, 可以写出如下伪代码:

```
for(int i = 0; i < 6; i++){
    if( a[i]>6)
        call bomb;
    for(int j = i+1; j <6; j++)
        if(a[i] == a[j])
            call bomb;
}
```

作用就是: 输入的数不能超过 6, 而且各不相同。所以我们可以输入 1 2 3 4 5 6 来调试程序。

1059:	83 f8 06	cmp	\$0x6, %eax
105c:	74 39	je	1097 <phase_6+0x71>

可以看到, 当外层循环执行完后, 跳转到了 1097 行

1097:	be 00 00 00 00	mov	\$0x0, %esi
109c:	89 f7	mov	%esi, %edi
109e:	8b 4c b5 d0	mov	-0x30(%ebp, %esi, 4), %ecx
10a2:	b8 01 00 00 00	mov	\$0x1, %eax
10a7:	8d 93 4c 01 00 00	lea	0x14c(%ebx), %edx
10ad:	83 f9 01	cmp	\$0x1, %ecx
10b0:	7e 0a	jle	10bc <phase_6+0x96>

0 放 esi,esi 放 edi, a[esi]放 ecx, 1 放 eax, 加载 ebp+14c 有效地址送 edx。此时 ecx 中存 a[esi] 的值, 如果 a[esi]<=1 则跳转至 10bc。在这里把一个地址加载进了 edx, 通过 gdb 调试查看该处存了什么。通过打印该处的信息, 我得到了一个链表:



```

(gdb) p/x *0x40414c@3      $17 = {0x27c, 0x3, 0x404170}
$15 = {0xee, 0x1, 0x404158} (gdb) p/x *0x404170@3
(gdb) p/x *0x404158@3      $18 = {0xc0, 0x4, 0x40417c}
$16 = {0x10c, 0x2, 0x404164} (gdb) p/x *0x40417c@3
(gdb) p/x *0x404164@3      $19 = {0x27b, 0x5, 0x4040e8}
$17 = {0x27c, 0x3, 0x404170} (gdb) p/x *0x4040e8@3
(gdb)                       $20 = {0x2d8, 0x6, 0x0}

```

链表中第一个元素是节点值，第二个是链表中的顺序，第三个是 next 指向的地址。通过打印下一个地址处的信息，我得到了一个由六个节点构成的链表。继续看它想干什么。

```

10ad: 83 f9 01          cmp     $0x1,%ecx
10b0: 7e 0a              jle     10bc <phase_6+0x96>

```

回到刚才，看看 `a[esi]<=1` (也就是 `a[esi]==1` 时) 会发生什么。

```

10bc: 89 54 bd b8          mov     %edx,-0x48(%ebp,%edi,4)
10c0: 83 c6 01            add     $0x1,%esi
10c3: 83 fe 06            cmp     $0x6,%esi
10c6: 75 d4              jne     109c <phase_6+0x76>

```

把链表的首地址放到了 `ebp+4edi-48` 的位置，此时 `edi` 中是 `esi`，其实就是把第 `esi+1` 个节点放到了 `ebp-48[esi]` 的位置。`esi` 自增 1，不等于 6 的话回到 109c, 进行对 `a[esi+1]` 的判断  
再看，如果 `a[esi]>1` 会做什么

```

10b2: 8b 52 08          mov     0x8(%edx),%edx
10b5: 83 c0 01          add     $0x1,%eax
10b8: 39 c8             cmp     %ecx,%eax
10ba: 75 f6              jne     10b2 <phase_6+0x8c>

```

把 `edx+8` (下一个节点) 送 `edx`, `eax` 自增 1，如果不=`ecx` (此时 `ecx` 中存 `a[esi]`)，那么继续执行。

```

10bc: 89 54 bd b8          mov     %edx,-0x48(%ebp,%edi,4)
10c0: 83 c6 01            add     $0x1,%esi
10c3: 83 fe 06            cmp     $0x6,%esi
10c6: 75 d4              jne     109c <phase_6+0x76>

```

把这个节点放到从 `ebp-48` 开始的第 `edi` 个位置处，此时 `edi` 中存的是 `esi`。重复 6 次上述循环体。分析上述代码，其实就是将 `ebp+14c` 处的链表以一定顺序重新放入了以 `ebp-48` 为起始地址处。这个重新排放的规则是：如果 `a[i]==1`，那就放第一个；如果 `a[i]>1`，那就把第 `a[i]` 个放到第 `i+1` 个。

```

10c8: 8b 75 b8          mov     -0x48(%ebp),%esi
10cb: 89 f1             mov     %esi,%ecx
10cd: b8 01 00 00 00    mov     $0x1,%eax
10d2: 8b 54 85 b8          mov     -0x48(%ebp,%eax,4),%edx
10d6: 89 51 08          mov     %edx,0x8(%ecx)
10d9: 83 c0 01          add     $0x1,%eax
10dc: 89 d1             mov     %edx,%ecx
10de: 83 f8 06          cmp     $0x6,%eax
10e1: 75 ef              jne     10d2 <phase_6+0xac>
10e3: c7 42 08 00 00 00 00 movl    $0x0,0x8(%edx)

```



程序继续执行，上述代码的执行作用是将 `ebp-48` 开始的 6 个节点重新连接，用一个循环，循环 6 次，使当前节点的 `next` 指向下一个节点，最后一个节点 `next` 指 `0x0`。

```

10ea:  bf 05 00 00 00      mov     $0x5,%edi
10ef:  eb 08              jmp     10f9 <phase_6+0xd3>
10f1:  8b 76 08          mov     0x8(%esi),%esi
10f4:  83 ef 01          sub     $0x1,%edi
10f7:  74 10             je      1109 <phase_6+0xe3>
10f9:  8b 46 08          mov     0x8(%esi),%eax
10fc:  8b 00             mov     (%eax),%eax
10fe:  39 06             cmp     %eax,(%esi)
1100:  7e ef             jle     10f1 <phase_6+0xcb>
1102:  e8 cf 02 00 00     call    13d6 <explode_bomb>

```

设置 `esi` 为 5（循环 5 次），跳转至 `10f9`，比较当前节点和下一节点，如果当前节点大于下一节点，则爆炸。说明 `ebp-48` 开始的节点值要按照从小到大排列。

重新梳理一下，输入 6 个数，小于等于 6 且各不相同，锁定输入 1-6

有一个链表，各节点值分别为 (16 进制) `ee 10c 27c c0 27b 2d8`

根据输入的六个数，将这 6 个值重排，规则是：这 6 个数第 `m` 个数是 `n`，则把原来第 `n` 个节点放到第 `m` 处，使得重排后个节点值按照从小到大排列。

因为 `c0 < ee < 10c < 27b < 27c < 2d8`

所以，得到答案为：**4 1 2 5 3 6**

## 7.

在 `phase_defused` 中找到加载了 `ebp-50` 有效地址进入 `eax` 进行字符串比较

```

15ef:  8d 83 b2 e5 ff ff  lea     -0x1a4e(%ebx),%eax
15f5:  50                push    %eax
15f6:  8d 45 a8          lea     -0x58(%ebp),%eax
15f9:  50                push    %eax
15fa:  e8 94 fc ff ff    call    1293 <strings_not_equal>

```

```

oc Breakpoint 1, 0x0040102c in phase_6 ()
(gdb) x/s 0x4025b2
0x4025b2:  "DrEvil"
(gdb) q

```

打印 `0x4025b2`,

得到一个字符串 `DrEvil`

```

15be:  8d 83 c0 03 00 00  lea     0x3c0(%ebx),%eax
15c4:  8d 80 f0 00 00 00  lea     0xf0(%eax),%eax
15ca:  50                push    %eax
15cb:  e8 b0 f3 ff ff    call    980 <__isoc99_sscanf@plt>

```

在 `sscanf` 前发现加载了一个有效地址，打印发现输入格式为 `%d %d %s`

其中 `%d%d` 的格式与第四题相同，因此推测在第四题后附加这个字符串进入隐藏关卡

```
ics@debian:~/Downloads/bomb15$ ./bomb ans.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...
```

进入 `secret_phase`:

```
1163: 55                push    %ebp
1164: 89 e5             mov     %esp,%ebp
1166: 56                push    %esi
1167: 53                push    %ebx
1168: e8 03 f9 ff ff    call    a70 <__x86.get_pc_thunk.bx>
116d: 81 c3 93 2e 00 00 add     $0x2e93,%ebx
1173: e8 e1 02 00 00    call    1459 <read_line>
1178: 83 ec 04          sub     $0x4,%esp
117b: 6a 0a             push    $0xa
117d: 6a 00             push    $0x0
117f: 50                push    %eax
1180: e8 5b f8 ff ff    call    9e0 <strtol@plt>
```

调用了 `read_line` 函数，读入一个字符串转化为整型

```
1185: 89 c6             mov     %eax,%esi
1187: 8d 40 ff          lea     -0x1(%eax),%eax
118a: 83 c4 10          add     $0x10,%esp
118d: 3d e8 03 00 00    cmp     $0x3e8,%eax
1192: 77 3b             ja      11cf <secret_phase+0x6c>
```

继续看，将 `eax` 中的返回值-1 后与 `0x3e8` 进行比较，如果大于 `0x3e8` 则跳转至 `11cf`

```
1194: 83 ec 08          sub     $0x8,%esp
1197: 56                push    %esi
1198: 8d 83 f8 00 00 00 lea     0xf8(%ebx),%eax
119e: 50                push    %eax
119f: e8 6d ff ff ff    call    1111 <fun7>
```

爆炸了，说明输入的数不能高于 `0x3e8+1H`，即要小于等于 `1001`

小于等于 `1001` 时程序继续执行，发现调用了 `fun7` 函数，并压入了两个参数，一个是 `ebx-f8`，另一个是 `esi`

```
1180: e8 5b f8 ff ff    call    9e0 <strtol@plt>
1185: 89 c6             mov     %eax,%esi
```

在之前的代码中看到 `esi` 中存的值为输入的数，所以第二个参数是输入的数

```
register group: general
eax 0x4040f8 4210936
```

```
(gdb) x 0x4040f8
0x4040f8 <n1>: 0x00000024
```

通过 gdb 调试查看寄存器，找到地址 0x4040f8 处的值为 0x24.

综上，在调用 fun7 是传入了两个参数，第一个参数是 0x24，第二个参数是输入的数

11a4:	83 c4 10	add	\$0x10,%esp
11a7:	83 f8 03	cmp	\$0x3,%eax
11aa:	74 05	je	11b1 <secret_phase+0x4e>
11ac:	e8 25 02 00 00	call	13d6 <explode_bomb>

调用完 fun7 后，将返回值与 0x3 进行了比较，不同则爆炸，说明 fun7 的返回值要是 3

进入 fun7 看执行了什么操作：

分析 fun7，循环递归调用。再看是一个二叉树。

1111:	55	push	%ebp
1112:	89 e5	mov	%esp,%ebp
1114:	53	push	%ebx
1115:	83 ec 04	sub	\$0x4,%esp
1118:	8b 55 08	mov	0x8(%ebp),%edx
111b:	8b 4d 0c	mov	0xc(%ebp),%ecx
111e:	85 d2	test	%edx,%edx
1120:	74 3a	je	115c <fun7+0x4b>

首先把第一个参数存入了 edx,如果 edx 是 0, tset 指令执行后 ZF=1, 跳转至 115c

115c:	b8 ff ff ff ff	mov	\$0xffffffff,%eax
1161:	eb e1	jmp	1144 <fun7+0x33>

返回 0

1122:	8b 1a	mov	(%edx),%ebx
1124:	39 cb	cmp	%ecx,%ebx
1126:	7f 21	jg	1149 <fun7+0x38>

edx 是当前节点，ecx 是用户输入值，如果当前节点>用户输入，跳转至 1149

1149:	83 ec 08	sub	\$0x8,%esp
114c:	51	push	%ecx
114d:	ff 72 04	pushl	0x4(%edx)
1150:	e8 bc ff ff ff	call	1111 <fun7>
1155:	83 c4 10	add	\$0x10,%esp
1158:	01 c0	add	%eax,%eax
115a:	eb e8	jmp	1144 <fun7+0x33>

当前节点<用户输入时

```

1128:  b8 00 00 00 00      mov     $0x0,%eax
112d:  39 cb               cmp     %ecx,%ebx
112f:  74 13              je      1144 <fun7+0x33>
1131:  83 ec 08          sub     $0x8,%esp
1134:  51                push    %ecx
1135:  ff 72 08          pushl   0x8(%edx)
1138:  e8 d4 ff ff ff    call    1111 <fun7>

```

根据此上三种情况，可以尝试写出 c 语言代码：

```

int fun7(Node *root,int x){
    if(root->value==x)
        return 0;
    else if(root->next>x)
        return 2*fun7(root->left,x);
    else
        return 2*fun7(root->right,x)+1;
}

```

这个函数的功能是，当当前节点少于输入时，查询右子数；当前节点大于输入，查左子树；相同返回 0.

3 是奇数，一定从右侧返回，上一个返回值是  $(3-1)/2=1$

1 是奇数，从右侧返回，上一个返回值是  $(1-1)/2=0$

0 代表当前节点的值等于查询值，即为我们要输入的数

最初的地址为 0x4040f8，所以要从该地址 右→右，访问对应侧的地址，从而找到要输入的值，访问过程如下

```

(gdb) p/x *0x4040f8@3
$3 = {0x24, 0x404104, 0x404110}
(gdb) p/x *0x404110@3
$4 = {0x32, 0x404128, 0x404140}
(gdb) p/x *0x404140@3
$5 = {0x6b, 0x4040a0, 0x4040dc}
(gdb)

```

找到 0x6b，即 107，输入后结果正确

太感动了，终于解完了。最后来一张全解开的图纪念一下我被炸烂的日子。

下面是自己的一些心得体会：



```
ics@debian:~/Downloads/bomb15$ ./bomb ans.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!
ics@debian:~/Downloads/bomb15$
```

### 个人总结感受:

1. 学会 `gdb` 调试, 比如我要从 `phase_2` 开始, 可以在这里打一个端点: `b phase_2`
2. 学会查看寄存器, 用 `(gdb) layout regs` 指令
3. `x/2s` 可以查看字符串, `print (char*)` 也可以
4. 在看汇编代码时, 要学会抓住核心, 比如比较语句和跳转语句往往比较重要
5. 像是 `je f0a <.L24+0x2e>` 这样的跳转指令, 不需要自己去算跳转地址, 在指令中已经给出 `f0a`, 就是跳转的地址
6. 要注意区分 `jmp`, `jle`, `je`, `js` 等。如果细节处理不好整个程序过程都无法分析到位
7. `cmp` 的地方往往是得出答案的关键
8. 分支结构的特点是: 有很多处语句都会跳转到同一处, 比如第三个炸弹中都跳转到了 `f00`
9. 递归调用的特点是在一个函数里又调用自身
10. `SHR` 是逻辑右移, 右移 31 位相当于取最高位
11. `SAR` 是算数右移, 最高位补符号位, 右移一位相当于除以 2
12. 多个参数的情况下, 先压栈的是后面的参数。
13. 有些函数调用需要吧参数压入 `esi` 中, 返回值一般都放在 `eax` 中, 比如第五题中的 `string_length` 函数
14. 见到基址+比例变址+偏移的寻址方式, 一般变址寄存器中存的都是相当于 `a[i]` 的 `i`, 基址+偏移是数组的首地址, 打印这个地址可以看数组
15. 用 `mov` 指令访问数组元素的值需要用 `()` 加载地址的值
16. 当有很多循环嵌套时, 要仔细分析跳转的条件
17. 一下子看不出程序要干什么时, 可以先在汇编代码旁边写出直白的意思, 比如“送 0 给 `eax`”这样的语句, 全部写出后, 在结合 `gdb` 调试过程中寄存器的存值, 就比较好理解了