

CSAPP lab2 bomblab 实验报告

实验日期：2025.Mar.19-2025.Apr.2

1 实验解题思路

1.1 phase_1

答案：You can Russia from land here in Alaska.

解题思路：

- 在汇编代码中发现了名为 `<strings_not_equal>` 函数，猜测此题应输入一段**字符串**。
- 根据 `main` 函数汇编代码的以下部分，判断调用 `phase_1` 函数前，**输入的字符串已在 `%rdi` 。**

```
call    1c78 <read_line>      # 读取字符串
mov     %rax,%rdi             # 读取结果放入%rdi
call    15e7 <phase_1>        # 调用phase_1
```

- 接下来进入 `phase_1` 函数**，在调用函数 `<strings_not_equal>` 前，向 `%rsi` 中载入了一个地址，猜测其为目标字符串的地址，利用 `gdb` 进行验证：

```
(gdb) info r rsi
rsi                0x555555557150          93824992244048
(gdb) x/s 0x555555557150
0x555555557150: "You can Russia from land here in Alaska."
```

发现 `%rsi` 中地址果然指向字符串 "You can Russia from land here in Alaska."

- 然后进入 `<strings_not_equal>` 函数**，此函数比较 `%rdi` 指向的字符串（输入的字符串）与 `%rsi` 指向的字符串（目标字符串）是否不相等，**如果不相等返回1，如果相等返回0。**
- 之后通过 `test %eax,%eax` 检查返回值是否为0，若不是0（字符串不相等）就会爆炸。因此要拆炸弹应输入目标字符串 "You can Russia from land here in Alaska."

1.2 phase_2

答案：6 7 9 12 16 21

答案不唯一，满足**第一个数大于0且下一个数=前一个数+前一个数的序号**即可。

解题思路

- 首先根据函数 `<read_six_numbers>`，及 `%rsi` 中的格式化信息，判断应输入**6个十进制整数**。（6位的整数数组）

```

0x000055555555c5b in read_six_numbers ()
(gdb) info r rsi
rsi                0x555555557323                93824992244515
(gdb) x/s 0x555555557323
0x555555557323: "%d %d %d %d %d %d"

```

- 接下来是检查**第一个数不能小于0**，并对之后的循环进行一些**初始化**。`%rbp` 存放当前访问的数的地址，初始是第一个数的地址。`%ebx` 表示当前访问的**数的序号**，初始为1。

```

cmpl    $0x0, (%rsp)           # 将第一个数与0比较
js      163d <phase_2+0x32>     # 如果小于0会爆炸

mov     %rsp, %rbp             # %rbp=%rsp= &nums[0] 第一个数地址
mov     $0x1, %ebx             # n = 1 (数的序号)

```

- 之后是一个**do-while**循环，上一步完成初始化后先跳转到 `0x1650` 执行（do-while是先执行再判断），执行的内容是：判断**下一个数是否=前一个数+前一个数的序号**，若是不等于就爆炸。若是没有爆炸就继续**更新 n**，更新后判断 `n` 是否达到6，若是达到了就终止循环，否则继续执行。

```

163b:  eb 13          jmp     1650 <phase_2+0x45>

1644:  83 c3 01      add     $0x1, %ebx          # n++
1647:  48 83 c5 04   add     $0x4, %rbp          # 地址指向下一个整数
164b:  83 fb 06      cmp     $0x6, %ebx          # 若n==6 就结束
164e:  74 11         je      1661 <phase_2+0x56>
1650:  89 d8         mov     %ebx, %eax          # %eax = n
1652:  03 45 00      add     0x0(%rbp), %eax     # %eax = n+nums[n-1]
1655:  39 45 04      cmp     %eax, 0x4(%rbp)     # 判断nums[n]是否等于%eax
1658:  74 ea         je      1644 <phase_2+0x39> #等于继续循环, 否则爆炸

```

对应的c的do-while代码如下：

```

int n=1;
do{
    int t=n+nums[n-1];
    if(t!=nums[n]){
        bomb();
    }
    n++;
}while(n<6)

```

- 由此得出输入的6位数组要求**第一个数大于0且下一个数=前一个数+前一个数的序号**。
- 若第一个数为6，则数组可为6 7 9 12 16 21。

1.3 phase_3

答案：0 458/1 222/3 917/4 404/5 703/6 282/7 419

(第二个整数取决于第一个整数)

解题思路

- 首先，通过 `%rsi` 中存的格式化信息判断应输入两个十进制整数(`%d %d`)。

```
(gdb) x/s 0x55555555732f
0x55555555732f: "%d %d"
```

- 然后需要判断第一个输入的整数小于等于7

```
cmpl    $0x7, (%rsp)          # rsp装第一个值的地址
ja      1719 <phase_3+0x9c>    # 要小于等于7
```

- 根据第一个输入的整数（跳转序号），进行跳转表的跳转。跳转表中装的是32位的地址相对偏移量（大小为 4bytes），然后再将地址相对偏移量加到跳转基地址，得到要跳转的绝对地址。

```
mov      (%rsp), %eax          # eax = 输入的第一个整数(跳转序号)
lea      0x1b02(%rip), %rdx     # rdx = 跳转基地址
movslq   (%rdx, %rax, 4), %rax  # 取跳转表对应位置存的偏移量(32位)传给%rax, 并拓展为64位
add      %rdx, %rax            # 基地址加上地址偏移量, 得到需跳转的绝对地址, 存到%rax
notrack  jmp  *%rax            # 读取%rax存的指令的地址 间接跳转
```

- 根据基地址 `%rdx` 显示出不同跳转序号对应的偏移量（4字节的原始字节内容）

```
(gdb) x/32xb 0x5555555571c0
0x5555555571c0: 0x65 0xe5 0xff 0xff 0 0x0f 0xe5 0xff 0xff 1
0x5555555571c8: 0x2f 0xe5 0xff 0xff 0x36 0xe5 0xff 0xff
0x5555555571d0: 0x3d 0xe5 0xff 0xff 0x44 0xe5 0xff 0xff
0x5555555571d8: 0x4b 0xe5 0xff 0xff 6 0x52 0xe5 0xff 0xff
```

比如序号为6（第七个偏移量）时，偏移量的原始字节是 0x4b 0xe5 0xff 0xff，表示 0xfffffe54b（小端序），再拓展为64位为 0xfffffffffffffe54b，然后再将偏移量与 `%rbx` 的基地址（0x5555555571c0）相加，得到目标指令地址 0x5555555570b。

```
(gdb)
0x0000555555556c2 in phase_3 ()
(gdb) info r rax
rax                0xfffffffffffffe54b    -6837    movslq后%rax
(gdb) si
0x0000555555556c5 in phase_3 ()
(gdb) info r rax
rax                0x5555555570b          93824992237323    add %rdx %rax后
                                                              的%rax
```

- 然后对照汇编代码，找到要跳转的目标指令地址，对应的指令是给 `%eax` 赋值

```
170b: b8 1a 01 00 00      mov $0x11a,%eax    # 跳转到170b eax = 0x11a(282)
```

- 第二个输入整数应与 %eax 相等，因此**第一个整数为6时，第二个整数应为282。**

```
cmp    %eax,0x4(%rsp)      # 第二个数和eax比较 不相等就会爆炸
jne    172c <phase_3+0xaf>
```

- 第一个整数为其他小于等于7的正整数时也可用类似方法查找，不同的跳转指令给 %eax 的赋值不同，如下：

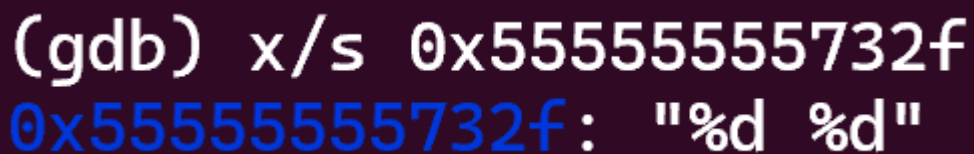
```
16cf: b8 de 00 00 00      mov $0xde,%eax      # 输入1跳到这 %eax = 222
16ef: b8 5c 02 00 00      mov $0x25c,%eax     # 输入2跳到这 %eax = 604
16f6: b8 95 03 00 00      mov $0x395,%eax     # 输入3跳到这 %eax = 917
16fd: b8 94 01 00 00      mov $0x194,%eax     # 输入4跳到这 %eax = 404
1704: b8 bf 02 00 00      mov $0x2bf,%eax     # 输入5跳到这 %eax = 703
170b: b8 1a 01 00 00      mov $0x11a,%eax     # 输入6跳到这 %eax = 282
1712: b8 a3 01 00 00      mov $0x1a3,%eax     # 输入7跳到这 %eax = 419
1725: b8 ca 01 00 00      mov $0x1ca,%eax     # 输入0跳到这 %eax = 458
```

1.4 phase_4

答案：13 31

解题思路

- 首先，通过 %rsi 中存的**格式化信息**判断应输入**两个十进制整数**。



```
(gdb) x/s 0x55555555732f
0x55555555732f: "%d %d"
```

- 由以下代码判断出**第一个输入的整数应小于等于14**

```
cmpl    $0xe, (%rsp)      # rsp是第一个参数 应小于等于0xe
jbe     17aa <phase_4+0x3c> # jbe:无符号数小于等于
call    1c07 <explode_bomb> # 不小于等于将会爆炸
```

- **调用函数 func4()**，并初始化传入的参数

```
mov     $0xe,%edx          # edx = 0xe = right(右端点)
mov     $0x0,%esi          # esi = 0 = left(左端点)
```

```
mov    (%rsp),%edi                # edi = 输入的第一个整数(目标值target)
call   1738 <func4>
```

- 进入 func4()，相当于一个**递归二分查找**的函数【从 0x0 到 0xe 的连续整数中找到所要找到 target，target 在前面的判断时已保证在范围之内】

```
00000000000001738 <func4>:    # 递归二分查找（以下仅为部分汇编代码）
173f:  29 f0          sub    %esi,%eax    # eax = r-l
1743:  c1 eb 1f      shr    $0x1f,%ebx    # 取t符号位(l-r为负时也能下取整)
1746:  01 c3          add    %eax,%ebx    # ebx=r-l（若x>=y:+0;否则+1）
1748:  d1 fb          sar    %ebx          # ebx = (r-l)//2(向下取整)
174a:  01 f3          add    %esi,%ebx    # ebx = mid = l+(r-l)//2
174c:  39 fb          cmp    %edi,%ebx    # mid和目标值(w)比较
174e:  7f 06          jg     1756 <func4+0x1e>    # mid>target 跳到1756
1750:  7c 10          jl     1762 <func4+0x2a>    # mid<target 跳到1762
1752:  89 d8          mov    %ebx,%eax    # 找到 return mid

### mid>w:
1756:  8d 53 ff      lea    -0x1(%rbx),%edx    # r = mid-1
1759:  e8 da ff ff ff call   1738 <func4>
175e:  01 c3          add    %eax,%ebx    # mid += eax（返回值）
1760:  eb f0          jmp    1752 <func4+0x1a>    # return

### mid<w:
1762:  8d 73 01      lea    0x1(%rbx),%esi    # l = mid+1
1765:  e8 ce ff ff ff call   1738 <func4>
176a:  01 c3          add    %eax,%ebx    # mid +=eax
176c:  eb e4          jmp    1752 <func4+0x1a>    # return
```

对应的大致代码如下：

```
int binary_search(int l,int r,int target){
    int mid=l+(r-l)/2;
    if(mid>target){
        return mid+binary_search(nums,l,mid-1,target); //注意返回值要加上mid!
    }else if(mid<target){
        return mid+binary_search(nums,mid+1,r,target);
    }
    return mid;
}
```

- 函数调用结束后，判断**函数的返回值**和**第二个整数**的情况。**第二个整数应该为31。**

```
cmp     $0x1f,%eax                # func4返回值要等于0x1f(31)
cmpl    $0x1f,0x4(%rsp)          # 第二个输入整数要等于0x1f(31)
```

func4() 的返回值为所有 mid 值的累加（包括最后找到的那次，即包括搜索值自身），取决于第一个输入的整数（即要查找的值）。通过尝试，凑出来**查找13**时，返回值为31。

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14
mid_1=(1+14)/2=7          res=7
mid_2=((7+1)+14)/2=11     res=7+11=18
mid_3=((11+1)+14)/2=13(target) res=18+13=31
∴ 第一个输入整数为13
```

1.5 phase_5

答案：5（对16取模为5均可） 115

解题思路

- 首先，通过 %rsi 中存的**格式化信息**判断应输入**两个十进制整数**

```
(gdb)
0x000055555555304 in __isoc99_sscanf@plt ()
(gdb)
GI __isoc99_sscanf (s=0x555555559840 <input_strings+320> "5 115",
format=0x55555555732f "%d %d") at ./stdio-common/isoc99_sscanf.c:24
24 ./stdio-common/isoc99_sscanf.c: No such file or directory.
```

- 之后将第一个整数**对16取模**，并判断取模后不能等于15

```
and    $0xf,%eax          # 将第一个整数x提取后四位 即x%16
cmp    $0xf,%eax          # 和0xf(15)进行比较，模16不能等于15
je     1858 <phase_5+0x71> # 否则会炸
```

- 之后是一个**遍历数组**的循环，由**对第一个输入整数对16取模后的值**作为**起始访问索引**，每次遍历到的值累加到 %ecx 上，循环次数累加到 %edx 上，并且**上一次访问到的值**会作为**下一次访问的索引**。

```
1826:  b9 00 00 00 00      mov    $0x0,%ecx # 累加器
182b:  ba 00 00 00 00      mov    $0x0,%edx # 计数器
1830:  48 8d 35 a9 19 00 00 lea     0x19a9(%rip),%rsi # %rsi是数组基地址
1837:  83 c2 01 <----      add    $0x1,%edx # 累加
183c:  8b 04 86             |      mov    (%rsi,%rax,4),%eax # %eax=nums[%rax]
183f:  01 c1             |循环  add    %eax,%ecx # %ecx+=%eax
1841:  83 f8 0f             |      cmp    $0xf,%eax # 遍历到15停止
1844:  75 f1             ---   jne     1837 <phase_5+0x50> # 没到15就继续循环
```

- 循环结束，判断循环次数和累加值，**循环次数要等于15**，**累加值要等于第二个输入整数**，否则会爆炸

```

cmp    $0xf,%edx
jne    1858 <phase_5+0x71> # 一定要循环15次
cmp    %ecx,0x4(%rsp)      # 访问值的总和应等于第二个整数
je     185d <phase_5+0x76>
call   1c07 <explode_bomb>

```

- 借助gdb查看需要遍历的数组的内容

```

(gdb) p *0x5555555571e0@16
$4 = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}

```

然后根据第15次访问到15进行逆推：第15次访问到15(对应索引为6)，第14次访问到6（索引值为14），第13次访问到14（索引值为2），第12次访问到2（索引值为1）.....第1次应访问12（索引值为5）。

```

arr={10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}
index=0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```

- 因此第一个输入整数应为对16取模为5的数。
- 访问值累加的结果为：1到15的数总和减去索引值为15所对应的值5（ $1+2+3+...+15-5=115$ ），因此第二个输入整数为115。

1.6 phase_6

答案：1 2 3 6 4 5

解题思路

- 首先根据 <read_six_numbers> 可判断，需要输入6个整数
- 该phase可分为四部分来看

part1 输入数据检查

- 由 0x18b2 开始进入一个通过循环遍历检查输入数字的双层循环。外循环从前到后遍历输入的6个数字，先检查该数-1后是否大于5（减1后大于5会爆炸），然后进入内循环检查该


```

18a9: 41 be 01 00 00 00    mov     $0x1,%r14d → 循环计数器 # r14 = 1
18af: 49 89 e4             mov     %rsp,%r12 → 循环变量地址 # r12 = rsp
-----part1
18b2: eb 28              jmp     18dc <phase_6+0x5e> # 开始进入双层循环
18b4: e8 4e 03 00 00    call   1c07 <explode_bomb>
18b9: eb 30              jmp     18eb <phase_6+0x6d>
18bb: 48 83 c3 01        add     $0x1,%rbx ← 内循环update.
18bf: 83 fb 05           cmp     $0x5,%ebx
18c2: 7f 10             jg      18d4 <phase_6+0x56>
18c4: 41 8b 04 9c        mov     (%r12,%rbx,4),%eax # %eax 地址取下一个数.
18c8: 39 45 00           cmp     %eax,0x0(%rbp) # (%rbp) 是当前位置.
18cb: 75 ee             jne     18bb <phase_6+0x3d>
18cd: e8 35 03 00 00    call   1c07 <explode_bomb>
18d2: eb e7             jmp     18bb <phase_6+0x3d>
18d4: 49 83 c6 01        add     $0x1,%r14 # r14+=1
18d8: 49 83 c5 04        add     $0x4,%r13 # 取下一个数. 存前数在内存中的地址.
18dc: 4c 89 ed           mov     %r13,%rbp # rbp=r13 →
18df: 41 8b 45 00        mov     0x0(%r13),%eax # %eax 为对应序号的数.
18e3: 83 e8 01           sub     $0x1,%eax # %eax - 1
18e6: 83 f8 05           cmp     $0x5,%eax # 数组中的每个数都要减1后与5比较. 若大于6 会爆炸.
18e9: 77 c9             ja      18b4 <phase_6+0x36> # -1后>5会爆炸
18eb: 41 83 fe 05        cmp     $0x5,%r14d # r14是否大于5
18ef: 7f 05             jg      18f6 <phase_6+0x78>
18f1: 4c 89 f3           mov     %r14,%rbx # rbx=r14
18f4: eb ce             jmp     18c4 <phase_6+0x46>

```

内循环的 第一次循环

内循环退出

地址超出外循环

```
for(int i=0;i<6;i++){
    if(nums[i]-1>5){
        bomb();
    }
    for(int j=i+1;j<6;j) //内循环
        if(nums[i]==nums[j]){
            bomb();
        }
}
}
```

- 继续看汇编代码，看到这一行不知道是什么意思

于是用 gdb 查看 %rdx 内容，查看后发现 %rdx 放的应该是地址，再查看该地址周围的字节情况。

```
(gdb) x/20x ($rdx)
0x55555559210 <node1>: 0x000000ec 0x00000001 0x55559220 0x00005555
0x55555559220 <node2>: 0x00000285 0x00000002 0x55559230 0x00005555
0x55555559230 <node3>: 0x000002ed 0x00000003 0x55559240 0x00005555
0x55555559240 <node4>: 0x00000333 0x00000004 0x55559250 0x00005555
0x55555559250 <node5>: 0x000003b7 0x00000005 0x55559110 0x00005555
(gdb) x/4x 0x55555559110
0x55555559110 <node6>: 0x0000031f 0x00000006 0x00000000 0x00000000
```

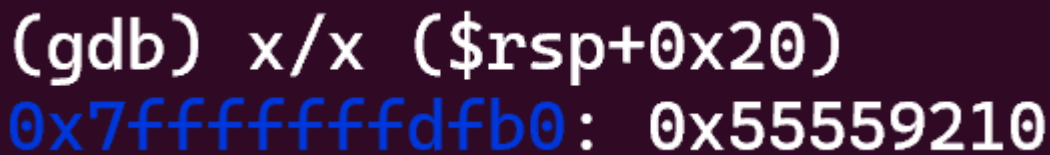

发现是5个 node，每一个 node 存放三个数据，其中前一个 node 的第三个数据和后一个 node 的地址相同（最后一个 node 指向空地址），于是推测 %rdx 是一个**链表**的首地址。

- 两个节点之间的地址差是**16字节**，第一个数据一时难以看出，第二个数据是节点的序号（类型为 int，大小**4字节**），第三个数据是下一个节点的地址（类型是 pointer，大小**8字节**），推得第一个数据的大小应为**4字节**，推测应该也是 int。该**链表数据结构**如下：

```
typedef struct node{
    int nodeValue;
    int num;
    node* next;
}
```

- 运行这一行汇编后，查看 %rsp+0x20，发现存入了**当前序号对应的节点的地址**。步长为8，代表这组成一个**由节点地址组成的指针数组**。

```
191a: 48 89 54 f4 20      mov     %rdx,0x20(%rsp,%rsi,8)
```



```
(gdb) x/x ($rsp+0x20)
0x7fffffffdfb0: 0x55559210
```

- 由此推出part2是**根据输入的数字将链表节点地址放入数组**的操作。对应汇编代码分析如下：

```
18f6: be 00 00 00 00      mov     $0x0,%esi          # rsi = 0 (初始化)
18fb: 8b 0c b4             mov     (%rsp,%rsi,4),%ecx  # ecx = 第n个数
18fe: b8 01 00 00 00      mov     $0x1,%eax          # eax = 1
1903: 48 8d 15 06 39 00 00 lea     0x3906(%rip),%rdx    # rdx为链表首地址
190a: 83 f9 01             cmp     $0x1,%ecx          # 比较ecx和1
190d: 7e 0b               jle     191a <phase_6+0x9c> # 若是序号小于等于1，将该节点直接放入
190f: 48 8b 52 08          mov     0x8(%rdx),%rdx     # %rdx = node.next 取下一个地址
1913: 83 c0 01             add     $0x1,%eax          # eax += 1
1916: 39 c8               cmp     %ecx,%eax          # 比较eax和ecx
1918: 75 f5               jne     190f <phase_6+0x91>
191a: 48 89 54 f4 20      mov     %rdx,0x20(%rsp,%rsi,8) # rsp+0x20是当前序号对应node的地址，排成一个有序指针数组
191f: 48 83 c6 01          add     $0x1,%rsi          # rsi +=1 (update) rsi表示目标数组下标序号(逆序递增)
1923: 48 83 fe 06          cmp     $0x6,%rsi
1927: 75 d2               jne     18fb <phase_6+0x7d>  # 继续循环
```

通过一个小循环，
取到对应序号的节点。
node<n>

```
//此部分对应c代码
//nums为输入的6个整数构成的数组
node** node_Addrs_Arr=(node**)malloc(6*sizeof(node*));
for(int i=0;i<6;i++){
    node* targetNode=head;//head为链表首地址
    for(int j=nums[i];j>1;j--){
        targetNode=targetNode.next;
    }//得到当前数字对应的节点的地址
    node_Addrs_Arr[i]=targetNode; //放入数组
}
```

part3 链表重连

- part3是将排进数组中的节点按其数组中的顺序进行**重连**。方便之后遍历。

```

1929:  48 8b 5c 24 20      mov     0x20(%rsp),%rbx   # %rbx = node<i>
192e:  48 8b 44 24 28      mov     0x28(%rsp),%rax   # %rax = node<j>
1933:  48 89 43 08         mov     %rax,0x8(%rbx)    # node<i>.next=node<j>

```

part4 遍历链表 判断前后节点值的大小

遍历重连后的链表，若是当前node中的第一个数据大于后一个node中的第一个数据，就会爆炸。

```

mov     0x8(%rbx),%rax      # (%rax) = node.next.nodeValude
mov     (%rax),%eax         # %eax = (%rax)
cmp     %eax,(%rbx)         # 比较 node.nodeValue和node.next.nodeValue
jle     196a <phase_6+0xec> # 若是前一个的值大于后一个,就会爆炸
call    1c07 <explode_bomb>

```

通过part2中放 %rdx 中存放的地址周围的字节情况，查看链表中字节的内容。（以十进制方式显示，方便比较大小）

```

(gdb) x/20d ($rdx)
0x55555559210 <node1>: 236      1      1431671328      21845
0x55555559220 <node2>: 645      2      1431671344      21845
0x55555559230 <node3>: 749      3      1431671360      21845
0x55555559240 <node4>: 819      4      1431671376      21845
0x55555559250 <node5>: 951      5      1431671056      21845
(gdb) x/4d 0x55555559110
0x55555559110 <node6>: 799      6      0      0

```

第一个数据从小到大排序是234 645 749 799 819 951，其对应的节点序号是1 2 3 6 4 5
因此答案为1 2 3 6 4 5。

secret_phase

答案：7（或者所含有效数字为7的字符串，如"7abc"）

同时要在 phase_4 的输入时加上字符串"DrEvil"以触发 secret_phase

解题思路

1.找到secret_phase

- 在汇编代码中搜索 secret_phase，发现除了 secret_phase 本身的函数，在 phase_defused 中会在满足条件下调用了 secret_phase。
- 查看 phase_defused 的代码，发现有一个与6比较的跳转，将断点打在 phase_defused 后运行查看后发现，只有当之前6个phase全都defused后，才会跳转。

```

1dc8:  83 3d 21 39 00 00 06      cmpl    $0x6,0x3921(%rip)

```

```
1dcf: 74 15 je 1de6 <phase_defused+0x36>
```

- 跳转后，有一个对 scanf 返回值与3的比较判断，检查 %rdi 发现是 phase_4 输入的内容，因此在 **在 phase_4 输入阶段应输入三个数据**才能触发 secret_phase。

```
1dfc: 48 8d 3d ed 39 00 00 lea 0x39ed(%rip),%rdi
1e03: e8 f8 f4 ff ff call 1300 <__isoc99_sscanf@plt>
1e08: 83 f8 03 cmp $0x3,%eax
1e0b: 74 0e je 1e1b <phase_defused+0x6b>
```

输入的格式为

```
format=0x555555557379 "%d %d %s"
```

- 然后是一个类似 phase_1 中的比较字符串是否相等的操作，第三个输入的**字符串应**为"DrEvil"

```
(gdb) x/s $rsi
0x555555557382: "DrEvil"
```

- 之后会输出两句提示字符，成功进入 secret_phase

2.破解secret_phase

答案：7（或者所含有有效数字为7的字符串，如"7abc"）

解题思路

- 首先程序调用 <strtol@plt> 将输入的**字符串转为数字**(若是输入的字符串中全是字母或开头非有效数字，就会返回-1，否则将有效数字转换为数字)，并将返回值(转换后的十进制整数)减1后与1000比较大小，若是大于1000则会爆炸，即**输入中的有效数字不能大于1001**。

```
long strtol(const char *nptr, char **endptr, int base);
//nptr: 指向要转换的字符串。
//endptr: 可选参数，用于存储解析结束的位置（通常传入 `NULL` 表示不关心）。
//base: 指定数字的进制（例如 10 表示十进制，16 表示十六进制）。
```

```
(gdb)
__strtol (nptr=0x5555555598e0 <input_strings+480> "7", endptr=0x0, base=10)
```

```
19fe: e8 dd f8 ff ff call 12e0 <strtol@plt>
1a05: 83 e8 01 sub $0x1,%eax # %eax -=1
1a08: 3d e8 03 00 00 cmp $0x3e8,%eax # 和0x3e8(1000)比较大小
```

- 然后将转换后的**有效整数**及一个**全局变量的地址**传入 fun7，并且其**返回值等于4**时**炸弹解除**。

```

1a0f:  89 de                mov    %ebx,%esi        # %esi=%ebx=%eax
1a11:  48 8d 3d 18 37 00 00  lea     0x3718(%rip),%rdi # rip-relative全局
变量
1a18:  e8 89 ff ff ff      call   19a6 <fun7>      # 调用fun7
1a1d:  83 f8 04            cmp     $0x4,%eax       # 将fun7的返回值和4
比较

```

- fun7 是一个**遍历二叉树的递归函数**，并根据输入值返回一个编码结果。
- 若是**节点值为0**，则返回-1。若是节点值不为0：若**匹配节点**，则返回0；若是小于当前节点，就进入**左子树**，返回值乘2；若是大于当前节点，就进入**右子树**，返回值乘2加1。

```

19aa:  48 85 ff            test   %rdi,%rdi        ** 检查二叉树是否结束
19ad:  74 32              je     19e1 <fun7+0x3b> ** 二叉树结束--跳转
19b3:  8b 17              mov     (%rdi),%edx
19b5:  39 f2              cmp     %esi,%edx       # 节点值-输入值 比较
19b7:  7f 0c              jg     19c5 <fun7+0x1f> # 小于节点--跳转
19b9:  b8 00 00 00 00     mov     $0x0,%eax       ## 匹配节点 返回0
19be:  75 12              jne     19d2 <fun7+0x2c> ### 大于节点--跳转
19c5:  48 8b 7f 08       mov     0x8(%rdi),%rdi  # 小于节点--进入左子
树
19c9:  e8 d8 ff ff ff     call   19a6 <fun7>      # 小于节点--递归
19ce:  01 c0              add     %eax,%eax       # 小于节点--返回值乘2
19d0:  eb ee              jmp     19c0 <fun7+0x1a> # return
19d2:  48 8b 7f 10       mov     0x10(%rdi),%rdi ### 大于节点--进入右
子树
19d6:  e8 cb ff ff ff     call   19a6 <fun7>      ### 大于节点--递归
19db:  8d 44 00 01       lea     0x1(%rax,%rax,1),%eax ### 大于节点--乘
2加1
19df:  eb df              jmp     19c0 <fun7+0x1a> ### return
19e1:  b8 ff ff ff ff     mov     $0xffffffff,%eax ** 二叉树结束--返回-1
19e6:  c3                ret

```

根据代码猜测二叉树数据结构如下

```

struct TreeNode {
    int value;           // 当前节点的值
    struct TreeNode *right; // 右子树地址
    struct TreeNode *left;  // 左子树地址
};

```

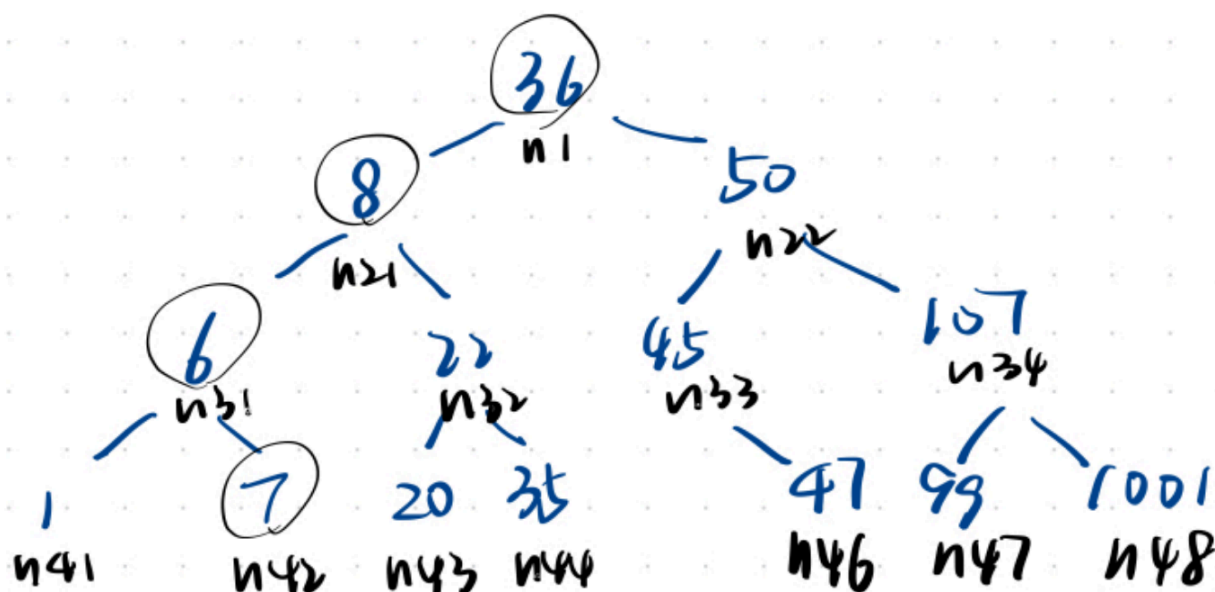
- 然后通过 gdb 查看进入 fun7 前 %rdi 所指的周围地址的内容，查看二叉树的内容

```
(gdb) x/28xg $rdi 当前节点值
0x555555559130 <n1>: 0x00000000000000024 0x0000555555559150 右节点地址
0x555555559140 <n1+16>: 0x0000555555559170 0x00000000000000000 左节点地址
0x555555559150 <n21>: 0x00000000000000008 0x00005555555591d0
0x555555559160 <n21+16>: 0x0000555555559190 0x00000000000000000
0x555555559170 <n22>: 0x00000000000000032 0x00005555555591b0
0x555555559180 <n22+16>: 0x00005555555591f0 0x00000000000000000
```

继续查看更多数据，发现该二叉树共有4层

```
(gdb) x/32d 0x555555559030
0x555555559030 <n41>: 1 0
0x555555559040 <n41+16>: 0 0
0x555555559050 <n47>: 99 0
0x555555559060 <n47+16>: 0 0
0x555555559070 <n44>: 35 0
0x555555559080 <n44+16>: 0 0
0x555555559090 <n42>: 7 0
0x5555555590a0 <n42+16>: 0 0
0x5555555590b0 <n43>: 20 0
0x5555555590c0 <n43+16>: 0 0
0x5555555590d0 <n46>: 47 0
0x5555555590e0 <n46+16>: 0 0
0x5555555590f0 <n48>: 1001 0
0x555555559100 <n48+16>: 0 0
0x555555559110 <node6>: 25769804575 93824992252480
0x555555559120 <bomb_id>: 8 0
```

二叉树结构图如下:



- 凑出返回值为4，可以 $((0*2+1)*2*2=4$ ，于是由从递归从后往前推得，输入值要小于 $n1(36)$ (进入左子树，返回值乘2)，小于 $n21(8)$ (进入左子树，返回值乘2)，大于 $n31(6)$ (进入右子树，返回值乘2 加1)，等于 $n42(7)$ 返回值0，递归到底部，再回上去从0开始进行运算。所以 secret_phase 的答案为7 (或者所含有有效数字为7的字符串，如"7abc")。

2实验中遇到和解决的问题

2.1关于字节的显示格式

问题：

在 phase_3 中，将跳转表的对应偏移量显示成了十进制数字的形式 (x/8d)

```
(gdb) x/8d 0x5555555571c0
0x5555555571c0: 101      -27      -1       -1       15       -27      -1       -1
```

并将其作为 `movslq (%rdx,%rax,4),%rax` 中 `%rax` 实际存储的偏移量进行计算

```
0: 0x5555555571c0+0d101=0x555555557326
1 5: 0x5555555571c0-0d27=0x5555555571a5
2 3 6 7: 0x5555555571c0-1=0x5555555571bf
4: 0x5555555571c0+0d15=0x5555555571cf
```

结果发现这样手动计算的结果并不是 `add %rdx,%rax` 后 `%rax` 的值。

- 在第一个输入整数为6时，实际运行 `movslq (%rdx,%rax,4),%rax` 和 `add %rdx,%rax` 后 `%rax` 的值：

```
(gdb)
0x00005555555556c2 in phase_3 ()
(gdb) info r rax
rax                                0xfffffffffffffe54b
(gdb) si
0x00005555555556c5 in phase_3 ()
(gdb) info r rax
rax                                0x55555555570b
```

- 偏移量和偏移后的值并不是预期的 `0xffffffffffff (-1)` 和 `0x5555555571bf`。

解决：

利用 `x/32xb` 显示内存中的原始字节内容 (小端序)

```
(gdb) x/32xb 0x5555555571c0
0x5555555571c0: 0x65  0xe5  0xff  0xff  0x0f  0xe5  0xff  0xff
0x5555555571c8: 0x2f  0xe5  0xff  0xff  0x36  0xe5  0xff  0xff
0x5555555571d0: 0x3d  0xe5  0xff  0xff  0x44  0xe5  0xff  0xff
0x5555555571d8: 0x4b  0xe5  0xff  0xff  0x52  0xe5  0xff  0xff
```

每四个字节 (32位) 对应一个序号的偏移量的原始字节 (小端序表示)

比如第七个偏移量 (序号为6)，原始字节是 `0x4b 0xe5 0xff 0xff`，表示 `0xffffffffffffe54b`，再

拓展为64位为 `0xffffffffffffe54b`，与用 `gdb` 显示出来的一致，然后再将偏移量与 `%rbx` 的基地址（`0x5555555571c0`）相加，得到目标地址 `0x5555555570b`。

两种显示结果数值不同的可能原因（猜测）

（以下解释来自ai）

(1) GDB 的解释方式

直接将原始字节解释为十进制值可能会忽略以下问题：

- **动态初始化或修改：**
 - 跳转表的内容可能在运行时被**动态初始化或修改**。
 - 例如，程序可能将某些**占位值（如 -1）**，替换为实际的偏移量（如 `0xFFFFE54B`）。
- **逻辑值与存储值的差异：**
 - 程序可能使用**某种逻辑计算**偏移量，而不是直接存储静态值。

(2) 实际加载的值

`movslq (%rdx,%rax,4), %rax` 指令直接从内存中读取 4 字节的内容，并将其扩展为 64 位有符号整数。因此，它加载的是**内存中的原始值（如 `0xFFFFE54B`）**，而不是 **GDB 解释的逻辑值（如 -1）**。

2.2递归程序的调用顺序

问题：

在 `secret_phase` 中推导如何凑出通过 $((0*2+1) * 2*2=4$ 得到返回值为4时，以为应该从上往下，从 `n1` 开始，按照所需运算操作，先进入右节点 `n22` (乘2加1)，再进入左节点 `n33` (乘2)再进入左节点(乘2)，因此输入应为小于45的数，结果错误。

解决：

用 `gdb` 进行一步一步的调试，发现函数调用的顺序是先把被调用的函数运行完后再进行调用函数的程序的后续步骤。所以**树从上往下的遍历应该对应从后到前的运算**，将进行的运算反过来倒推树的遍历，找到正确结果应为7。

3实验最终结果截图

```
wcx@LAPTOP-OMTCB5PG:~/lab2/bomb$ cat ans.txt
```

You can Russia from land here in Alaska.

6 7 9 12 16 21

6 282

13 31 DrEvil

5 115

1 2 3 6 4 5

7

```
wcx@LAPTOP-OMTCB5PG:~/lab2/bomb$ ./bomb8 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!