哈爾濱工業大學

实验报告

实 验（三）

题　　目 Binary Bomb

二进制炸弹

专　　业 计算机类

学　　号 1170500913

班　　级 1703002

学　　生 熊健羽

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2018.10.16

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

熟练掌握计算机系统的 ISA 指令系统与寻址方式
熟练掌握 Linux 下调试器的反汇编调试跟踪分析机器语言的方法
增强对程序机器级表示、汇编语言、调试器和逆向工程等的理解

1.2 实验环境与工具

1.2.1 硬件环境

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64 位)
GPU：Intel(R) HD Graphics 620
Nvidia GeForce 940MX
物理内存：8.00GB
磁盘：1TB HDD
128GB SSD

1.2.2 软件环境

Windows10 64 位；
Vmware 14.11；
Ubuntu 18.04 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位；
Code::Blocks；
gedit，gcc，notepad++；

1.3 实验预习

sample.c 代码如下

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*递归函数定义*/
long fact(long n)
{
    if(n > 0)
        return (n * fact(n - 1));
    else
        return 1;
}

int main()
{
/*字符串比较*/
    char s1[100];
    char s2[100];
    printf("please input string 1:");
    scanf("%s", s1);
    printf("please input string 2:");
    scanf("%s", s2);

    int len1 = strlen(s1);
    int len2 = strlen(s2);
    if(strcmp(s1, s2))
        printf("it is different\n");
    else
        printf("it is same\n");

/*循环*/
    for(int i = 0; i < len1; i++)
        printf("%c\n", s1[i]);

/*分支*/
    if(len1 > 5)            //if
        printf("s1 len is above 5\n");
    else
        printf("s1 len is not above 5\n");

    switch(len2)            //switch
    {
        case 0: printf("len2 = 0\n");break;
        case 1: printf("len2 = 1\n");break;
        case 2: printf("len2 = 2\n");break;
        case 3: printf("len2 = 3\n");break;
        case 4: printf("len2 = 4\n");break;
        case 5: printf("len2 = 5\n");break;
        default: printf("s2 len is above 5\n");
    }
/*调用递归函数*/
    printf("9! = %ld\n", fact(9));

/*指针(数组)操作*/
    int a[3] = {1, 4, 9};
    int *p = a;
    printf("*p + 1 = %d\n", *p + 1);
```

```
59.     printf("*(p + 1) = %d\n", *(p + 1));
60. 
61. /*结构体*/
62.     struct stu
63.     {
64.         char id[16];
65.         short age;
66.         int score[3];
67.     };
68.     struct stu stu1;
69.     strcpy(stu1.id, "1170500913");
70.     stu1.age = 18;
71.     stu1.score[0] = 98;
72.     stu1.score[1] = 99;
73.     stu1.score[2] = 100;
74. 
75. /*链表*/
76.     /*链表数据结构定义*/
77.     typedef struct LINK
78.     {
79.         int data;
80.         struct LINK *next;
81.     }link;
82. 
83.     /*创建一个长度为 3,数据分别为 0, 1, 2 的链表*/
84.     link *head1 = (link *)malloc(sizeof(link));
85.     link *lp = head1, *lq;
86.     for(int i = 0; i < 3; i++)
87.     {
88.         lp -> data = i;
89.         lp -> next = (link *)malloc(sizeof(link));
90.         lq = lp;
91.         lp = lp -> next;
92.     }
93.     free(lp);
94.     lq ->next = NULL;
95. 
96.     /*输出链表元素*/
97.     lp = head1;
98.     while(lp != NULL)
99.     {
100.         printf("%d\n", lp -> data);
101.         lp = lp -> next;
102.      }
103. 
104.      /*删除链表并释放空间*/
105.      lp = head1;
106.      while(lp != NULL)
107.      {
108.          lq = lp;
109.          lp = lp -> next;
110.          free(lq);
111.      }
112. 
113.      return 0;
114. }
```

各部分的汇编代码：

字符串比较部分：

```
1.  0x401583    push   %r12
2.  0x401585    push   %rbp
3.  0x401586    push   %rdi
4.  0x401587    push   %rsi
5.  0x401588    push   %rbx
6.  0x401589    sub    $0x120,%rsp
7.  0x401590    callq  0x401970 <__main>
8.  0x401595    lea    0x2a64(%rip),%rcx        # 0x404000
9.  0x40159c    callq  0x402e08 <printf>
10. 0x4015a1    lea    0xb0(%rsp),%rbx
11. 0x4015a9    mov    %rbx,%rdx
12. 0x4015ac    lea    0x2a64(%rip),%rcx        # 0x404017
13. 0x4015b3    callq  0x402df8 <scanf>
14. 0x4015b8    lea    0x2a5b(%rip),%rcx        # 0x40401a
15. 0x4015bf    callq  0x402e08 <printf>
16. 0x4015c4    lea    0x40(%rsp),%r12
17. 0x4015c9    mov    %r12,%rdx
18. 0x4015cc    lea    0x2a44(%rip),%rcx        # 0x404017
19. 0x4015d3    callq  0x402df8 <scanf>
20. 0x4015d8    mov    $0xffffffffffffffff,%rcx
21. 0x4015df    mov    $0x0,%eax
22. 0x4015e4    mov    %rbx,%rdi
23. 0x4015e7    repnz scas %es:(%rdi),%al
24. 0x4015e9    mov    %rcx,%rdx
25. 0x4015ec    not    %rdx
26. 0x4015ef    lea    -0x1(%rdx),%rsi
27. 0x4015f3    mov    %esi,%edi
28. 0x4015f5    mov    %r12,%rdx
29. 0x4015f8    mov    %rbx,%rcx
30. 0x4015fb    callq  0x402de8 <strcmp>
31. 0x401600    test   %eax,%eax
32. 0x401602    je     0x401617 <main+148>
33. 0x401604    lea    0x2a26(%rip),%rcx        # 0x404031
34. 0x40160b    callq  0x402e00 <puts>
35. 0x401610    mov    $0x0,%ebx
36. 0x401615    jmp    0x40163f <main+188>
37. 0x401617    lea    0x2a23(%rip),%rcx        # 0x404041
38. 0x40161e    callq  0x402e00 <puts>
39. 0x401623    jmp    0x401610 <main+141>
```

循环部分：

```
1.  0x401625    movslq %ebx,%rax
2.  0x401628    movsbl 0xb0(%rsp,%rax,1),%edx
3.  0x401630    lea    0x2a15(%rip),%rcx        # 0x40404c
4.  0x401637    callq  0x402e08 <printf>
5.  0x40163c    add    $0x1,%ebx
6.  0x40163f    cmp    %edi,%ebx
7.  0x401641    jl     0x401625 <main+162>
```

分支部分：

a) if-else 语句

```
0x401658    cmp    $0x5,%r12d
0x40165c    jle    0x401685 <main+258>
0x40165e    lea    0x29eb(%rip),%rcx        # 0x404050
0x401665    callq  0x402e20 <puts>
0x401685    lea    0x29d6(%rip),%rcx        # 0x404062
0x40168c    callq  0x402e20 <puts>
0x401691    jmp    0x40166a <main+231>
```

b) switch 语句

```
   0x40166a    cmp    $0x5,%edi
0x40166d    ja     0x40177f <main+508>
0x401673    mov    %edi,%edi
0x401675    lea    0x2a64(%rip),%rdx        # 0x4040e0
0x40167c    movslq (%rdx,%rdi,4),%rax
0x401680    add    %rdx,%rax
0x401683    jmpq   *%rax
0x401693    lea    0x29de(%rip),%rcx        # 0x404078
0x40169a    callq  0x402e20 <puts>
0x40172a    lea    0x2950(%rip),%rcx        # 0x404081
0x401731    callq  0x402e20 <puts>
0x401736    jmpq   0x40169f <main+284>
0x40173b    lea    0x2948(%rip),%rcx        # 0x40408a
0x401742    callq  0x402e20 <puts>
0x401747    jmpq   0x40169f <main+284>
0x40174c    lea    0x2940(%rip),%rcx        # 0x404093
0x401753    callq  0x402e20 <puts>
0x401758    jmpq   0x40169f <main+284>
0x40175d    lea    0x2938(%rip),%rcx        # 0x40409c
0x401764    callq  0x402e20 <puts>
0x401769    jmpq   0x40169f <main+284>
0x40176e    lea    0x2930(%rip),%rcx        # 0x4040a5
0x401775    callq  0x402e20 <puts>
0x40177a    jmpq   0x40169f <main+284>
0x40177f    lea    0x2928(%rip),%rcx        # 0x4040ae
0x401786    callq  0x402e20 <puts>
0x40178b    jmpq   0x40169f <main+284>
```

调用递归函数：

```
0x401560    push   %rbx
0x401561    sub    $0x20,%rsp
0x401565    mov    %ecx,%ebx
0x401567    test   %ecx,%ecx
0x401569    jg     0x401576 <fact+22>
0x40156b    mov    $0x1,%eax
0x401570    add    $0x20,%rsp
0x401574    pop    %rbx
0x401575    retq
0x401576    lea    -0x1(%rcx),%ecx
0x401579    callq  0x401560 <fact>
0x40157e    imul   %ebx,%eax
0x401581    jmp    0x401570 <fact+16>
```

指针操作：

```
0x4016b7    mov    $0x2,%edx
```

```
2.  0x4016bc    lea    0x29fd(%rip),%rcx        # 0x4040c0
3.  0x4016c3    callq  0x402e28 <printf>
4.  0x4016c8    mov    $0x4,%edx
5.  0x4016cd    lea    0x29f9(%rip),%rcx        # 0x4040cd
6.  0x4016d4    callq  0x402e28 <printf>
```

结构体操作：

```
1.  0x4016d9    movabs $0x3930303530373131,%rax
2.  0x4016e3    mov    %rax,0x20(%rsp)
3.  0x4016e8    movw   $0x3331,0x28(%rsp)
4.  0x4016ef    movb   $0x0,0x2a(%rsp)
5.  0x4016f4    movw   $0x12,0x30(%rsp)
6.  0x4016fb    movl   $0x62,0x34(%rsp)
7.  0x401703    movl   $0x63,0x38(%rsp)
8.  0x40170b    movl   $0x64,0x3c(%rsp)
```

链表操作：

```
1.  0x401713    mov    $0x10,%ecx
2.  0x401718    callq  0x402e38 <malloc>
3.  0x40171d    mov    %rax,%rbx
4.  0x401720    mov    %rax,%rsi
5.  0x401723    mov    $0x0,%edi
6.  0x401728    jmp    0x4017a9 <main+550>
7.  0x4017a0    add    $0x1,%edi
8.  0x4017a9    cmp    $0x2,%edi
9.  0x4017ac    jle    0x401790 <main+525>
10. 0x401790    mov    %edi,(%rsi)
11. 0x401792    mov    $0x10,%ecx
12. 0x401797    callq  0x402e38 <malloc>
13. 0x40179c    mov    %rax,0x8(%rsi)
14. 0x4017a3    mov    %rsi,%rbp
15. 0x4017a6    mov    %rax,%rsi
16. 0x4017ae    mov    %rsi,%rcx
17. 0x4017b1    callq  0x402e48 <free>
18. 0x4017b6    movq   $0x0,0x8(%rbp)
19. 0x4017be    mov    %rbx,%rsi
20. 0x4017c1    jmp    0x4017d5 <main+594>
21. 0x4017d5    test   %rsi,%rsi
22. 0x4017d8    jne    0x4017c3 <main+576>
23. 0x4017da    jmp    0x4017eb <main+616>
24. 0x4017c3    mov    (%rsi),%edx
25. 0x4017c5    lea    0x2910(%rip),%rcx        # 0x4040dc
26. 0x4017cc    callq  0x402e28 <printf>
27. 0x4017d1    mov    0x8(%rsi),%rsi
28. 0x4017eb    test   %rbx,%rbx
29. 0x4017ee    jne    0x4017dc <main+601>
30. 0x4017dc    mov    0x8(%rbx),%rsi
31. 0x4017e8    mov    %rsi,%rbx
32. 0x4017e0    mov    %rbx,%rcx
33. 0x4017e3    callq  0x402e48 <free>
```

第 2 章 实验环境建立

2.1 Ubuntu 下 CodeBlocks 反汇编（10 分）

CodeBlocks 运行 hellolinux.c。反汇编查看 printf 函数的实现。

要求：C、ASM、内存(显示 hello 等内容)、堆栈（call printf 前）、寄存器同时在一个窗口。

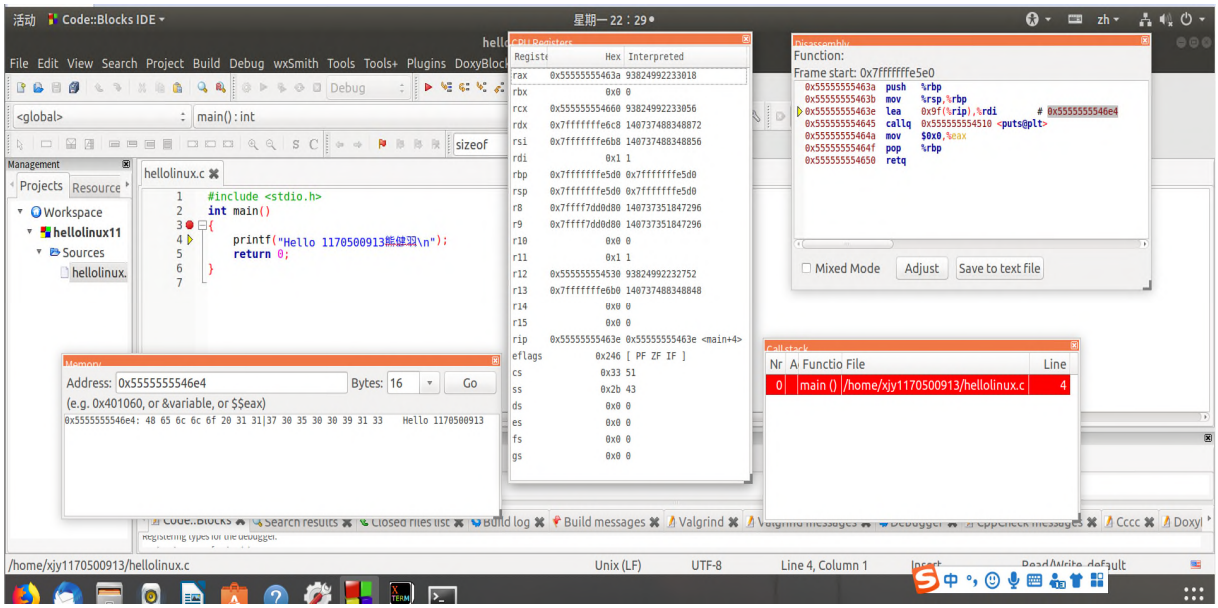


图 2-1 Ubuntu 下 CodeBlocks 反汇编截图

2.2 Ubuntu 下 EDB 运行环境建立（10 分）

用 EDB 调试 hellolinux.c 的执行文件，截图，要求同 2.1

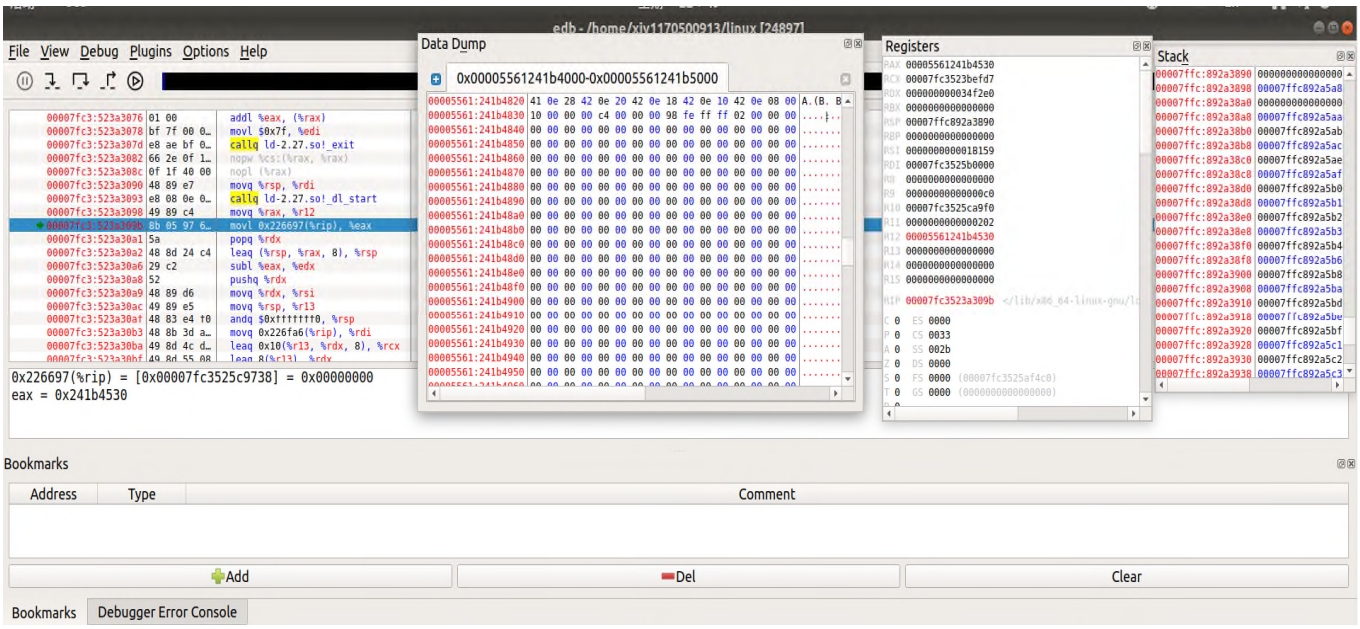


图 2-2　Ubuntu 下 EDB 截图

第 3 章 各阶段炸弹破解与分析

每阶段 15 分，密码 10 分，分析 5 分，总分不超过 80 分

3.1 阶段 1 的破解与分析

密码如下：Why make trillions when we could make... billions?

破解过程：

使用 GDB 调试，给 phase_1 函数打上断点，使用 x/10i $rip 命令，展示当前执行语句及后续 10 条汇编语句：

```
Breakpoint 1, 0x0000555555555174 in phase_1 ()
(gdb) x/10i $rip
=> 0x555555555174 <phase_1>:    sub    $0x8,%rsp
   0x555555555178 <phase_1+4>:
    lea    0x14e1(%rip),%rsi        # 0x555555556660
   0x55555555517f <phase_1+11>: callq  0x5555555555c5 <strings_not_equal>
   0x555555555184 <phase_1+16>: test   %eax,%eax
   0x555555555186 <phase_1+18>: jne    0x55555555518d <phase_1+25>
   0x555555555188 <phase_1+20>: add    $0x8,%rsp
   0x55555555518c <phase_1+24>: retq
   0x55555555518d <phase_1+25>: callq  0x5555555556d1 <explode_bomb>
   0x555555555192 <phase_1+30>: jmp    0x555555555188 <phase_1+20>
   0x555555555194 <phase_2>:    push   %rbp
```

其中调用了函数 strings_not_equal，如果返回值（%eax）不为 0，则跳转到 explode_bomb 函数，引爆炸弹。由函数名猜想，这个函数的作用是比较两个字符串是否相等，若不相等，则引爆炸弹。前一条语句中，某一全局变量的地址 0x555555556660 传送给寄存器%rsi，即作为 strings_not_equal 函数的第二个参数。但并没有给存储第一个参数的寄存器传送数值，因此推断，第一个参数，即是当前 phase_1 函数的第一个参数。查看 bomb.c 文件发现，phase_1 函数的参数即为输入字符串的地址。

```
/* Hmm...  Six phases must be more secure than one phase! */
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
phase_defused();                 /* Drat!  They figured it out!
```

故 strings_not_equal 是比较输入字符串与目标字符串，只有当他们相同，才不

会引爆炸弹。使用 x/s 0x555555556660 命令，查看目标字符串（如下图）。

```
(gdb) x/s 0x555555556660
0x555555556660: "Why make trillions when we could make... billions?"
```

得出阶段 1 拆弹密码为 Why make trillions when we could make... billions?

3.2 阶段 2 的破解与分析

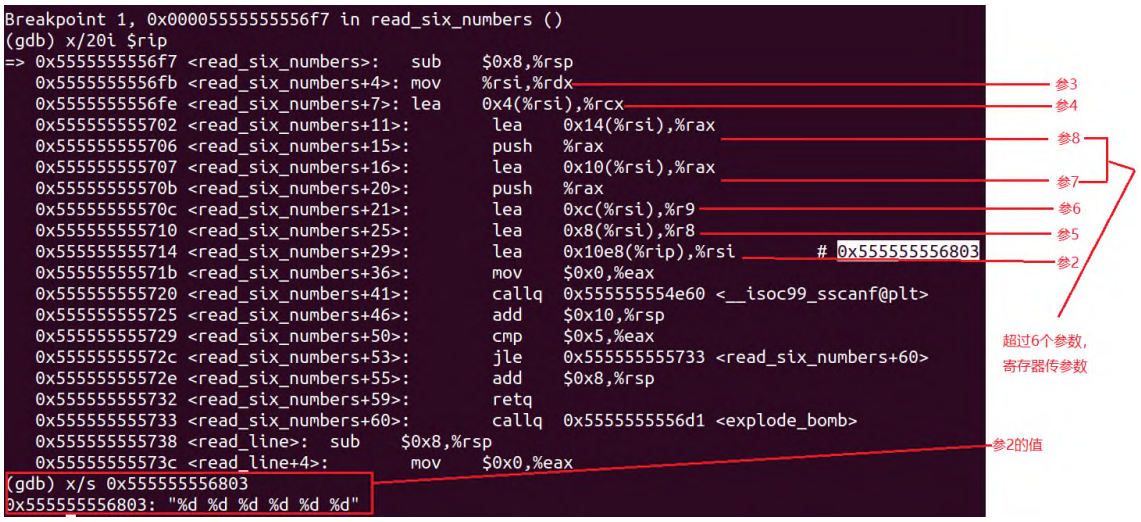
密码如下：1 2 4 8 16 32

破解过程：

同上述阶段，查看 phase_2 函数的反汇编代码：

```
0x555555555194 <phase_2>:    push   %rbp
0x555555555195 <phase_2+1>:  push   %rbx
0x555555555196 <phase_2+2>:  sub    $0x28,%rsp
0x55555555519a <phase_2+6>:  mov    %rsp,%rsi
0x55555555519d <phase_2+9>:
 callq  0x5555555556f7 <read_six_numbers>
0x5555555551a2 <phase_2+14>: cmpl   $0x1,(%rsp)
0x5555555551a6 <phase_2+18>: jne    0x5555555551b1 <phase_2+29>
0x5555555551a8 <phase_2+20>: mov    %rsp,%rbx
0x5555555551ab <phase_2+23>: lea    0x14(%rbx),%rbp
0x5555555551af <phase_2+27>: jmp    0x5555555551c1 <phase_2+45>
0x5555555551b1 <phase_2+29>: callq  0x5555555556d1 <explode_bomb>
0x5555555551b6 <phase_2+34>: jmp    0x5555555551a8 <phase_2+20>
0x5555555551b8 <phase_2+36>: add    $0x4,%rbx
0x5555555551bc <phase_2+40>: cmp    %rbp,%rbx
0x5555555551bf <phase_2+43>: je     0x5555555551d1 <phase_2+61>
0x5555555551c1 <phase_2+45>: mov    (%rbx),%eax
0x5555555551c3 <phase_2+47>: add    %eax,%eax
0x5555555551c5 <phase_2+49>: cmp    %eax,0x4(%rbx)
0x5555555551c8 <phase_2+52>: je     0x5555555551b8 <phase_2+36>
0x5555555551ca <phase_2+54>: callq  0x5555555556d1 <explode_bomb>
0x5555555551cf <phase_2+59>: jmp    0x5555555551b8 <phase_2+36>
0x5555555551d1 <phase_2+61>: add    $0x28,%rsp
-Type <return> to continue, or q <return> to quit---
0x5555555551d5 <phase_2+65>: pop    %rbx
0x5555555551d6 <phase_2+66>: pop    %rbp
0x5555555551d7 <phase_2+67>: retq
```

发现<phase_2+9>处调用函数 read_six_numbers，且第一个参数为 input（输入的字符串的首地址），第二个参数为某指针（局部变量）（以下记为 a）。于是使用 si 命令，进入该函数，查看反汇编代码：



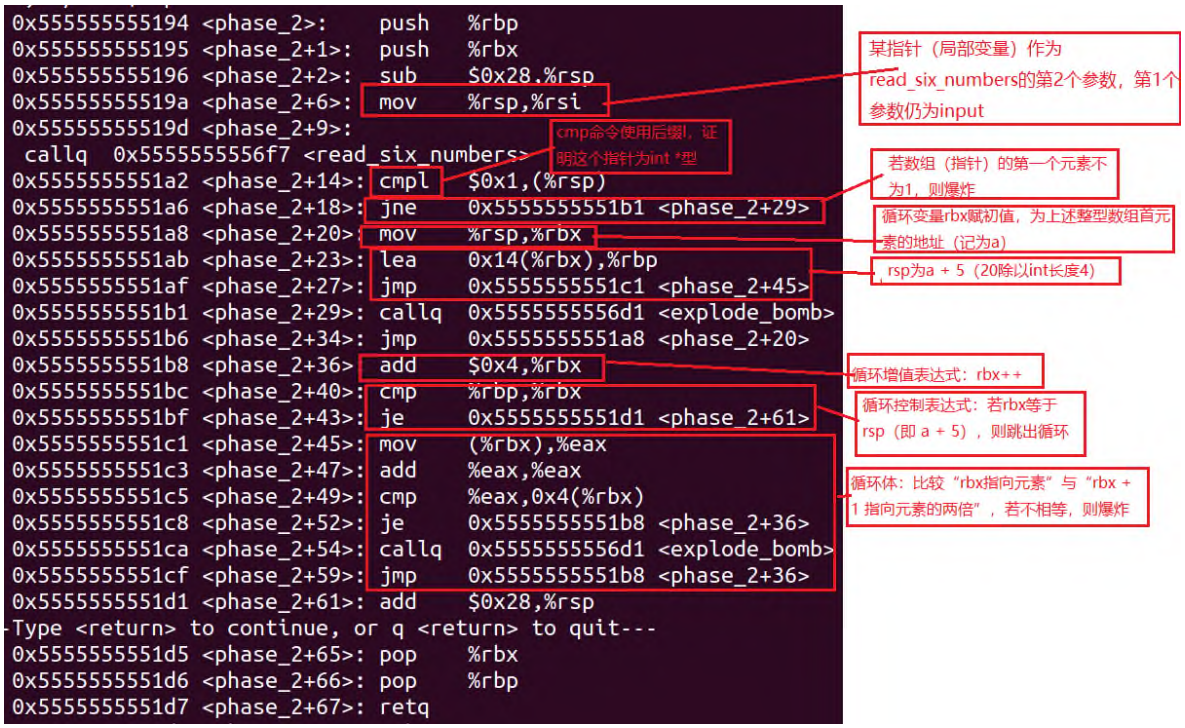
其中调用了库函数 sscanf，第一个参数为 input（输入的字符串的首地址）。

第二个参数如图 x/s 命令所示，为”%d %d %d %d %d %d”，第三个为 a，第四个为 a + 1，第五个为 a + 2。。。。。以此类推，第八个为 a + 5。

由此可以看出，read_six_numbers 函数的作用为：按格式读入用户输入字符串 input 的 6 个整型数，并将其存储在一个长度为 6 的整型数组 a[]中。

回到 phase_2 函数：

具体分析思路如图中所示：



据上述分析，写成 c 代码如下：

```
1.  void read_six_numbers(char *input, int *a)
2.  {
3.      if(sscanf(input, "%d %d %d %d %d %d", a, a + 1, a + 2, a + 3, a + 4, a + 5
    ) <= 5)
4.          explode_bomb();
5.      return;
6.  }
7.
8.  void phase_2(char* input)
9.  {
10.     int a[6];
11.     read_six_numbers(input, a);
12.     if(a[0] != 1)
13.         explode_bomb();
14.     int *rbx = a;
15.     int *rbp = a + 5;  //5 = 20 / 4
16.     for(rbx = a; rbx < rbp; rbx++)
17.     {
18.         if(*rbx != *(rbx + 1) * 2)
19.             explode_bomb();
20.     }
21. }
```

可知要使炸弹不爆炸，要按格式输入 6 个数。第一个数必须为 1，后面的数，每一个必须是前一个的两倍，即：1 2 4 8 16 32

3.3 阶段 3 的破解与分析

密码如下：

有 6 组可用的密码，任意输入一组即可：

a) 0 -886

b) 1 -936

c) 2 -526

d) 3 -835

e) 4 0

f) 5 -835

破解过程：

反汇编如下：

```
(gdb) x/50i $rip
=> 0x5555555551d8 <phase_3>:    sub    $0x18,%rsp
   0x5555555551dc <phase_3+4>:  lea    0x8(%rsp),%rcx
   0x5555555551e1 <phase_3+9>:  lea    0xc(%rsp),%rdx
   0x5555555551e6 <phase_3+14>: lea    0x1622(%rip),%rsi        # 0x55555555680f
   0x5555555551ed <phase_3+21>: mov    $0x0,%eax
   0x5555555551f2 <phase_3+26>: callq  0x555555554e60 <__isoc99_sscanf@plt>
   0x5555555551f7 <phase_3+31>: cmp    $0x1,%eax
   0x5555555551fa <phase_3+34>: jle    0x55555555521b <phase_3+67>
   0x5555555551fc <phase_3+36>: cmpl   $0x7,0xc(%rsp)
   0x555555555201 <phase_3+41>: ja     0x555555555292 <phase_3+186>
   0x555555555207 <phase_3+47>: mov    0xc(%rsp),%eax
   0x55555555520b <phase_3+51>: lea    0x14be(%rip),%rdx        # 0x5555555566d0
   0x555555555212 <phase_3+58>: movslq (%rdx,%rax,4),%rax
   0x555555555216 <phase_3+62>: add    %rdx,%rax
   0x555555555219 <phase_3+65>: jmpq   *%rax
   0x55555555521b <phase_3+67>: callq  0x5555555556d1 <explode_bomb>
   0x555555555220 <phase_3+72>: jmp    0x5555555551fc <phase_3+36>
   0x555555555222 <phase_3+74>: mov    $0x32,%eax
   0x555555555227 <phase_3+79>: jmp    0x55555555522e <phase_3+86>
   0x555555555229 <phase_3+81>: mov    $0x0,%eax
   0x55555555522e <phase_3+86>: sub    $0x19a,%eax
   0x555555555233 <phase_3+91>: add    $0x135,%eax
   0x555555555238 <phase_3+96>: sub    $0x343,%eax
   0x55555555523d <phase_3+101>:        add    $0x343,%eax
   0x555555555242 <phase_3+106>:        sub    $0x343,%eax
   0x555555555247 <phase_3+111>:        add    $0x343,%eax
   0x55555555524c <phase_3+116>:        sub    $0x343,%eax
   0x555555555251 <phase_3+121>:        cmpl   $0x5,0xc(%rsp)
   0x555555555256 <phase_3+126>:        jg     0x55555555525e <phase_3+134>
   0x555555555258 <phase_3+128>:        cmp    %eax,0x8(%rsp)
   0x55555555525c <phase_3+132>:        je     0x555555555263 <phase_3+139>
   0x55555555525e <phase_3+134>:        callq  0x5555555556d1 <explode_bomb>
   0x555555555263 <phase_3+139>:        add    $0x18,%rsp
   0x555555555267 <phase_3+143>:        retq
   0x555555555268 <phase_3+144>:        mov    $0x0,%eax
   0x55555555526d <phase_3+149>:        jmp    0x555555555233 <phase_3+91>
   0x55555555526f <phase_3+151>:        mov    $0x0,%eax
   0x555555555274 <phase_3+156>:        jmp    0x555555555238 <phase_3+96>
   0x555555555276 <phase_3+158>:        mov    $0x0,%eax
   0x55555555527b <phase_3+163>:        jmp    0x55555555523d <phase_3+101>
   0x55555555527d <phase_3+165>:        mov    $0x0,%eax
   0x555555555282 <phase_3+170>:        jmp    0x555555555242 <phase_3+106>
   0x555555555284 <phase_3+172>:        mov    $0x0,%eax
   0x555555555289 <phase_3+177>:        jmp    0x555555555247 <phase_3+111>
   0x55555555528b <phase_3+179>:        mov    $0x0,%eax
   0x555555555290 <phase_3+184>:        jmp    0x55555555524c <phase_3+116>
   0x555555555292 <phase_3+186>:        callq  0x5555555556d1 <explode_bomb>
   0x555555555297 <phase_3+191>:        mov    $0x0,%eax
   0x55555555529c <phase_3+196>:        jmp    0x555555555251 <phase_3+121>
```

首先，函数调用 sscanf 函数，格式化读入两个整型数。

显然接下来是一个 switch 语句。num1 大于 7 时或者小于 0，炸弹爆炸，说明 case 语句最大为 case7，最小为 case 0。接下来，关键在于找出跳转表表项对应的指令地址。

由<phase_3+51>处指令可知，跳转表的地址为 0x5555555566d0，于是查看跳转表的内容，如下图所示：（上面为 10 进制，下面为 16 进制）

```
(gdb) x/9dw 0x5555555566d0
0x5555555566d0: -5294   -5287   -5224   -5217
0x5555555566e0: -5210   -5203   -5196   -5189
0x5555555566f0 <array.3415>:    1969512813
(gdb) Quit
(gdb) x/9xw 0x5555555566d0
0x5555555566d0: 0xffffeb52      0xffffeb59      0xffffeb98      0xffffeb9f
0x5555555566e0: 0xffffeba6      0xffffebad      0xffffebb4      0xffffebbb
```

跳转地址 = 对应跳转表项（一个 int 型数）+跳转表首地址，故计算如下：

jump = 0x5555555566d0

jump[0] = 0xffffeb52　jump[0] + jump = 0x555555555222

jump[1] = 0xffffeb59　jump[1] + jump = 0x555555555229

jump[2] = 0xffffeb98　jump[2] + jump = 0x555555555268

jump[3] = 0xffffeb9f　jump[3] + jump = 0x55555555526F

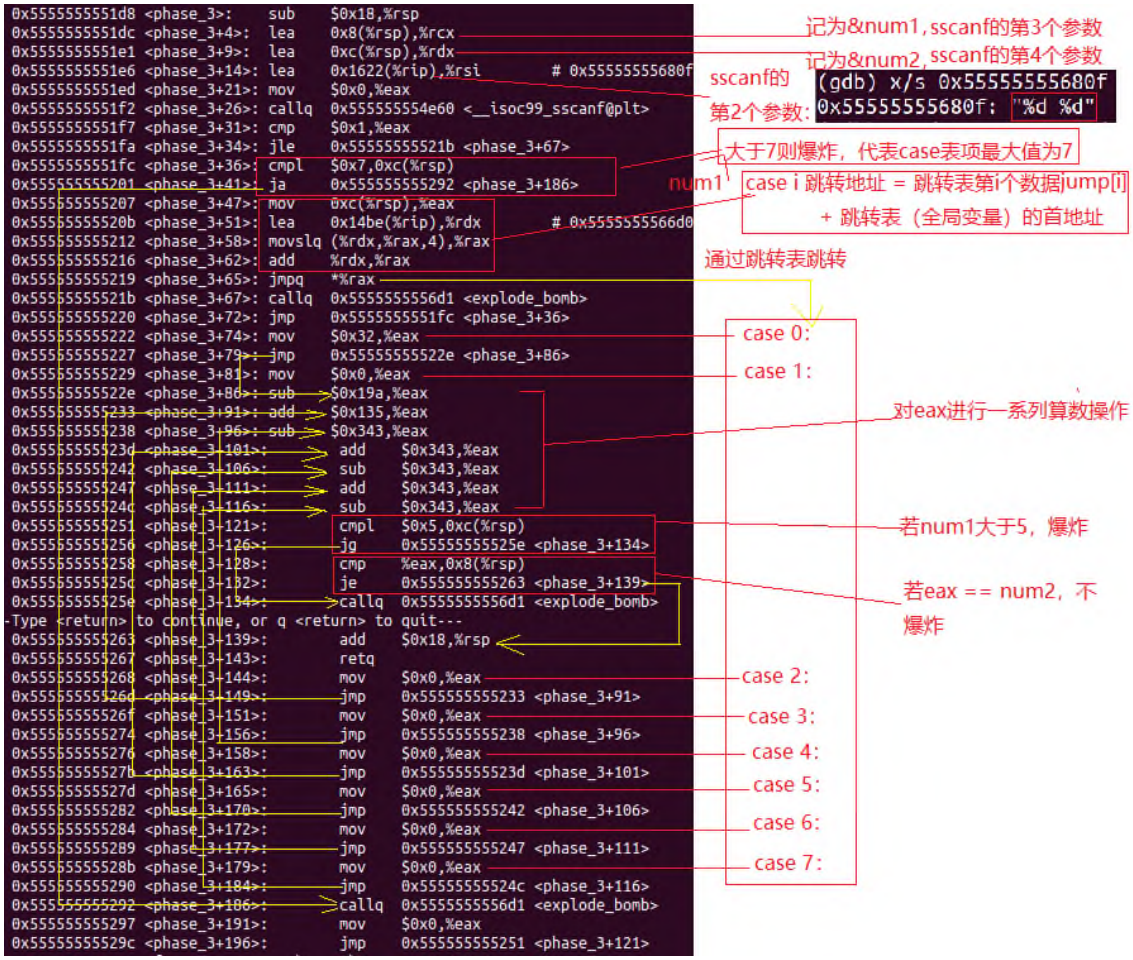
jump[4] = 0xffffeba6　jump[4] + jump = 0x555555555276

jump[5] = 0xffffebad　jump[5] + jump = 0x55555555527D

jump[6] = 0xffffebb4　jump[6] + jump = 0x555555555284

jump[7] = 0xffffebbb　jump[7] + jump = 0x55555555528B

即可对应跳转地址，具体分析如下图：



据上述分析，写成 c 代码如下：

```
1.  void phase_3(char *input)
2.  {
3.      int num1, num2, eax;  //num1 in 0xc(%rsp),num2 in 0x8(%rsp)
4.      if(sscanf(input, "%d %d", &num1, &num2) <= 1)
5.          explode_bomb();
6.      switch(num1)
7.      {
8.          case 0: eax = 50;
9.                  eax = eax - 936;
10.                 break;
11.         case 1: eax = 0;
12.                 eax = eax - 936;
13.                 break;
14.         case 2: eax = 0;
15.                 eax = eax - 526;
16.                 break;
17.         case 3: eax = 0;
18.                 eax = eax - 835;
19.                 break;
20.         case 4: eax = 0;
21.                 break;
22.         case 5: eax = 0;
23.                 eax = eax - 835;
```

```
24.                 break;
25.         case 6:
26.         case 7:
27.         default: explode_bomb();
28.     }
29.     if(eax == num2)
30.         return;
31.     else
32.         explode_bomb();
33. }
```

num1 = 0 时，执行 case 1:eax = 50 – 936 = -886，num2 应为-886

同理，根据以上 c 代码：

num1 = 1，num2 应为-936；num1 = 2，num2 应为-526；

num1 = 3，num2 应为-835；num1 = 4，num2 应为 0

num1 = 5，num2 应为-835；num > 5 或 num < 0 时，炸弹爆炸

故，密码有 6 组：

a) 0 -886

b) 1 -936

c) 2 -526

d) 3 -835

e) 4 0

f) 5 -835

3.4 阶段 4 的破解与分析

密码如下：4 19 DrEvil

（DrEvil 为进入隐藏关卡的附加字符串）

破解过程：将 phase_4 函数反汇编如下：

```
=> 0x5555555552d2 <phase_4>:    sub    $0x18,%rsp
   0x5555555552d6 <phase_4+4>:  lea    0x8(%rsp),%rcx
   0x5555555552db <phase_4+9>:  lea    0xc(%rsp),%rdx
   0x5555555552e0 <phase_4+14>: lea    0x1528(%rip),%rsi        # 0x55555555680f
   0x5555555552e7 <phase_4+21>: mov    $0x0,%eax
   0x5555555552ec <phase_4+26>: callq  0x555555554e60 <__isoc99_sscanf@plt>
   0x5555555552f1 <phase_4+31>: cmp    $0x2,%eax
   0x5555555552f4 <phase_4+34>: jne    0x5555555552fd <phase_4+43>
   0x5555555552f6 <phase_4+36>: cmpl   $0xe,0xc(%rsp)
   0x5555555552fb <phase_4+41>: jbe    0x555555555302 <phase_4+48>
   0x5555555552fd <phase_4+43>: callq  0x5555555556d1 <explode_bomb>
   0x555555555302 <phase_4+48>: mov    $0xe,%edx
   0x555555555307 <phase_4+53>: mov    $0x0,%esi
   0x55555555530c <phase_4+58>: mov    0xc(%rsp),%edi
   0x555555555310 <phase_4+62>: callq  0x55555555529e <func4>
   0x555555555315 <phase_4+67>: cmp    $0x13,%eax
   0x555555555318 <phase_4+70>: jne    0x555555555321 <phase_4+79>
   0x55555555531a <phase_4+72>: cmpl   $0x13,0x8(%rsp)
   0x55555555531f <phase_4+77>: je     0x555555555326 <phase_4+84>
   0x555555555321 <phase_4+79>: callq  0x5555555556d1 <explode_bomb>
   0x555555555326 <phase_4+84>: add    $0x18,%rsp
   0x55555555532a <phase_4+88>: retq
```

其中先调用了函数 sscanf(),同上一关的方法，确定了第一个参数为 input，第二个参数为”%d %d”，第三个参数为一个指针（0xc(%rip)）（局部变量）（记为&num1），第四个参数为另一个指针（0x8(%rip)）（局部变量）（记为&num2）。即按格式读取了输入字符串的两个整型数。

接着是一系列分支语句，其中调用了函数 func4()，第一个参数为 0xc(%rip) 地址中的数，即 num1，第二个参数为 0， 第三个参数为 0xe，即 14.

很容易写出对应的 c 语言代码：

```
1.  void phase_4(char *input)
2.  {
3.      int num1, num2;  //num1 in 0xc(%rsp),num2 in 0x8(%rip)
4.      if(sscanf(input, "%d %d", &num1, &num2) != 2)
5.          explode_bomb();
6.      if((unsigned)num1 > 14)
7.          explode_bomb();
8.      if(func4(num1, 0, 14) != 19)
9.          explode_bomb();
10.     if(num2 != 19)
11.         explode_bomb();
12.     return;
13. }
```

由第 10，11 行可知，**num2 为 19**，炸弹才不会爆炸。

关键在于分析 func4 函数，使最终返回值为 19，从而得出 num1。

进入该函数后，反汇编得：

```
0x555555555529e <func4>:      push   %rbx
0x555555555529f <func4+1>:    mov    %edx,%eax
0x5555555552a1 <func4+3>:     sub    %esi,%eax
0x5555555552a3 <func4+5>:     mov    %eax,%ebx
0x5555555552a5 <func4+7>:     shr    $0x1f,%ebx
0x5555555552a8 <func4+10>:    add    %eax,%ebx
0x5555555552aa <func4+12>:    sar    %ebx
0x5555555552ac <func4+14>:    add    %esi,%ebx
0x5555555552ae <func4+16>:    cmp    %edi,%ebx
0x5555555552b0 <func4+18>:    jg     0x5555555552ba <func4+28>
0x5555555552b2 <func4+20>:    cmp    %edi,%ebx
0x5555555552b4 <func4+22>:    jl     0x5555555552c6 <func4+40>
0x5555555552b6 <func4+24>:    mov    %ebx,%eax
0x5555555552b8 <func4+26>:    pop    %rbx
0x5555555552b9 <func4+27>:    retq
0x5555555552ba <func4+28>:    lea    -0x1(%rbx),%edx
0x5555555552bd <func4+31>:    callq  0x555555555529e <func4>
0x5555555552c2 <func4+36>:    add    %eax,%ebx
0x5555555552c4 <func4+38>:    jmp    0x5555555552b6 <func4+24>
0x5555555552c6 <func4+40>:    lea    0x1(%rbx),%esi
0x5555555552c9 <func4+43>:    callq  0x555555555529e <func4>
0x5555555552ce <func4+48>:    add    %eax,%ebx
0x5555555552d0 <func4+50>:    jmp    0x5555555552b6 <func4+24>
```

函数里面自身调用自身，说明是递归函数。函数首先对第 2,3 个参数进行了一系列运算，这部分翻译成 c 语言如下：

```
1.  int ans, rbx;
2.      ans = z - y;
3.      rbx = ans + ((unsigned)ans >> 31);//if (ans < 0) rbx = ans + 1;
4.                                        //else rbx = ans
5.      rbx = rbx >> 1;
6.      rbx = y + rbx;      //rbx = (z - y) / 2 + y;
```

第 2 行首先计算 z – y。在第三行中，利用无符号数的逻辑右移：若 z – y 小于零，则加上 bias = 1；若 z – y 大于零，则不加 bias。第 5 行对结果除以 2.至此，这段代码实现了(z – y) / 2，且结果向零舍入。（为了使负数向上（向零）舍入，除以 2^k 时，被除数加上偏置量 2^k – 1，这里 k = 1，故 bias = 1）。接着第 6 行再加上 y。总的实现了 rbx = (z - y) / 2 + y。

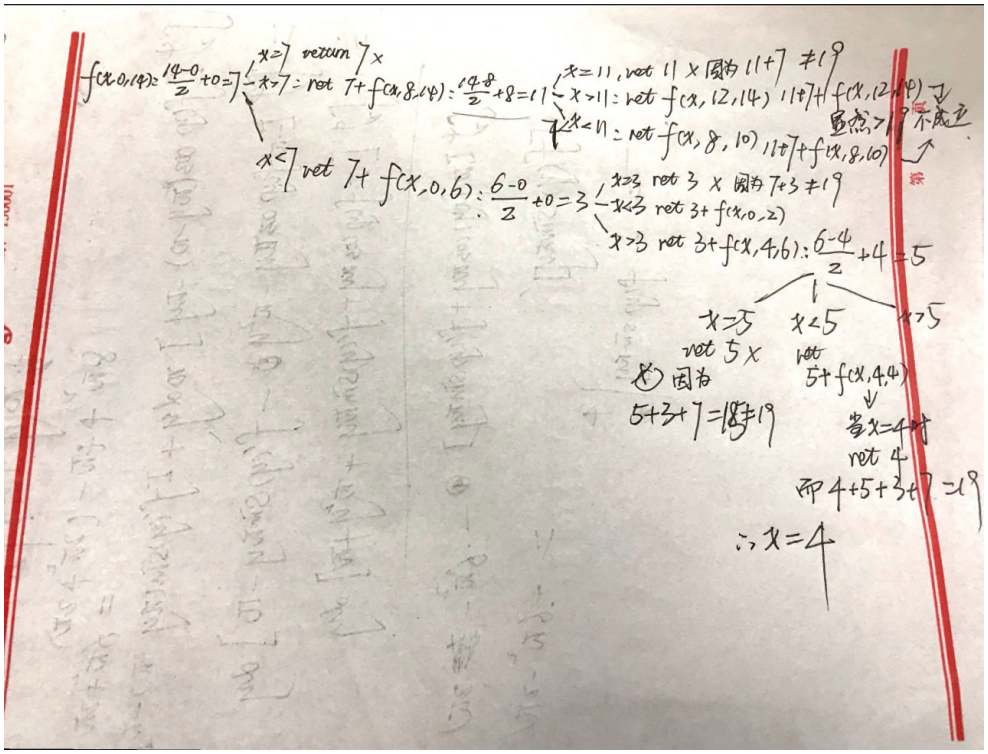
接着是返回环节，是一系列分支语句、

分析可知，这部分与上部分合起来的 c 代码如下：

```
1.  void phase_4(char *input)
2.  {
3.      int num1, num2;  //num1 in 0xc(%rsp),num2 in 0x8(%rip)
4.      if(sscanf(input, "%d %d", &num1, &num2) != 2)
5.          explode_bomb();
6.      if((unsigned)num1 > 14)
```

```
7.          explode_bomb();
8.      if(func4(num1, 0, 14) != 19)
9.          explode_bomb();
10.     if(num2 != 19)
11.         explode_bomb();
12.     return;
13. }
14. int func4(int x, int y, int z) //x->di y->si z->dx
15. {
16.     int ans, rbx;
17.     ans = z - y;
18.     rbx = ans + ((unsigned)ans >> 31);//if (ans < 0) rbx = ans + 1;
19.                                       //else rbx = ans
20.     rbx = rbx >> 1;
21.     rbx = y + rbx;      //rbx = (z - y) / 2 + y;
22.     if(rbx == x)
23.     {
24.         return rbx;
25.     }
26.     else if(rbx < x)
27.     {
28.         return rbx + fun(x,rbx + 1, z);
29.     }
30.     else
31.     {
32.          return rbx + fun(x, y, rbx - 1);
33.     }
34. }
```

对递归函数进行分析：



由上图分析可得 num1 等于 4，func4 函数才会最终返回 19，炸弹才不会爆炸。

故密码为 4 19

接着寻找进入隐藏关的密码，发现主函数框架中，每次解决一关，都存在一个 phase_defused 函数，猜想进入的方法可能在其中。

```
input = read_line();
phase_4(input);
phase_defused();
```

加断点、反汇编如下：

```
Breakpoint 8, 0x000055555555587c in phase_defused ()
(gdb) x/40i $rip                                    用户已输入6个字符串，才执行接下来的程序，意在让隐藏关出现在bomb的最后
=> 0x55555555587c <phase_defused>:      cmpl   $0x6,0x202e09(%rip)        # 0x55555575868c <num_input_strings>
   0x555555555883 <phase_defused+7>:    je     0x555555555887 <phase_defused+11>
   0x555555555885 <phase_defused+9>:    repz retq
   0x555555555887 <phase_defused+11>:   sub    $0x68,%rsp
   0x55555555588b <phase_defused+15>:   lea    0x8(%rsp),%rcx
   0x555555555890 <phase_defused+20>:   lea    0xc(%rsp),%rdx      调用sscanf，第一个参数为某全局字符串，第二个为另一全局字符串，第3、4、5个分别是两个局部的字符指针
   0x555555555895 <phase_defused+25>:   lea    0x10(%rsp),%r8
   0x55555555589a <phase_defused+30>:   lea    0xfb8(%rip),%rsi        # 0x555555556859
   0x5555555558a1 <phase_defused+37>:   lea    0x202ee8(%rip),%rdi        # 0x555555758790 <input_strings+240>
   0x5555555558a8 <phase_defused+44>:   mov    $0x0,%eax
   0x5555555558ad <phase_defused+49>:   callq  0x555555554e60 <__isoc99_sscanf@plt>
   0x5555555558b2 <phase_defused+54>:   cmp    $0x3,%eax
   0x5555555558b5 <phase_defused+57>:   je     0x5555555558c8 <phase_defused+76>
   0x5555555558b7 <phase_defused+59>:   lea    0xeda(%rip),%rdi        # 0x555555556798
   0x5555555558be <phase_defused+66>:   callq  0x555555554db0 <puts@plt>
   0x5555555558c3 <phase_defused+71>:   add    $0x68,%rsp
   0x5555555558c7 <phase_defused+75>:   retq
   0x5555555558c8 <phase_defused+76>:   lea    0x10(%rsp),%rdi
   0x5555555558cd <phase_defused+81>:   lea    0xf8e(%rip),%rsi        # 0x555555556862
   0x5555555558d4 <phase_defused+88>:   callq  0x5555555555c5 <strings_not_equal>      将上面读取的第5个字符指针（即：按格式读取的第3个变量）指向的字符串与某字符串比较，若相等，则进入secret_phase（隐藏关）
   0x5555555558d9 <phase_defused+93>:   test   %eax,%eax
   0x5555555558db <phase_defused+95>:   jne    0x5555555558b7 <phase_defused+59>
   0x5555555558dd <phase_defused+97>:   lea    0xe54(%rip),%rdi        # 0x555555556738
   0x5555555558e4 <phase_defused+104>:  callq  0x555555554db0 <puts@plt>
   0x5555555558e9 <phase_defused+109>:  lea    0xe70(%rip),%rdi        # 0x555555556760
   0x5555555558f0 <phase_defused+116>:  callq  0x555555554db0 <puts@plt>
   0x5555555558f5 <phase_defused+121>:  mov    $0x0,%eax
   0x5555555558fa <phase_defused+126>:  callq  0x5555555554ce <secret_phase>
   0x5555555558ff <phase_defused+131>:  jmp    0x5555555558b7 <phase_defused+59>
```

于是分别查看 sscanf 的第 1、2 个参数字符串：

```
(gdb) x/s 0x555555758790
0x555555758790 <input_strings+240>:     "4 19 DrEvil"
(gdb) x/s 0x555555556859
0x555555556859: "%d %d %s"
```

发现，按格式读取的字符串，正是第 4 关输入的字符串。

于是继续查看下面 string_not_equal 的标准字符串：

```
(gdb) x/s 0x555555556862
0x555555556862: "DrEvil"
```

得到进入隐藏关的附加字符串”DrEvil”。

根据图中对 phase_defused 的前两行分析，隐藏关将在 bomb 最后出现。

3.5 阶段 5 的破解与分析

密码如下：

密码有 5^6 组：（任意一组均可）

第一个字符可以是’ 2 B R b 五个中任意一个

第二个字符可以是% 5 E U e 五个中任意一个

第三个字符可以是, < L \ l 五个中任意一个

第四个字符可以是$ 4 D T d 五个中任意一个

第五个字符可以是/ ? O _ o 五个中任意一个

第六个字符可以是’ 7 G W g 五个中任意一个

下面是其中 5 个密码的例子：

a) "%,$/'

b) 25<4?7

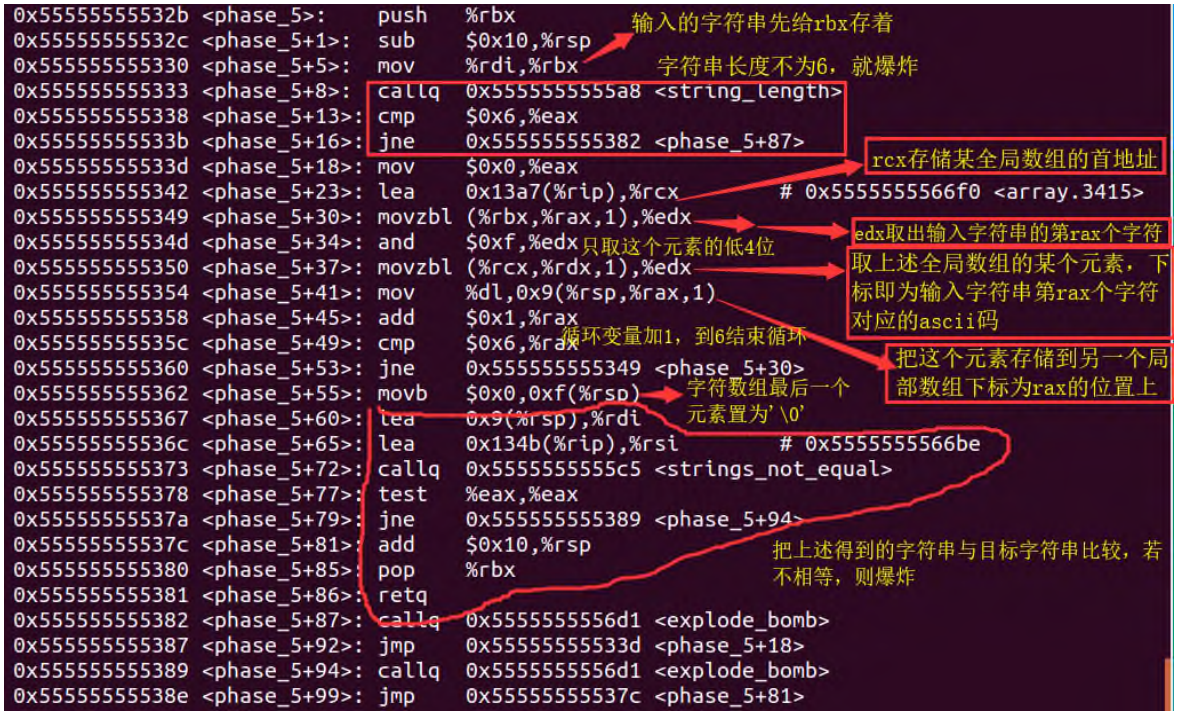
c) BELDOG

d) RU\T_W

e) beldog

破解过程：

反汇编及分析如下：



首先查看后面 string_not_equal 的标准字符串：

```
(gdb) x/s 0x5555555566be
0x5555555566be: "devils"
```

写成 c 语言代码形式：

```c
char array[];

void phase_5(char *input)
{
    // char *rbx = input;
    char *p;
    char s[7];   //s = 0x9(%rsp)
    int dx;
    char bt;
    if(string_length(input) != 6)
        explode_bomb();
    for(int i = 0, p = array; i < 6; i++)
    {
        bt = 0xf & input[i];
        s[i] = array[(int)bt];
    }
    s[6] = '\0';
    if(string_not_equal(s, "devils") != 0)
        explode_bomb();
    return;
}
```

关键在于：查看 array 字符串的值，寻找相应的字母'd','e','v','i','l','s'的对应位置。于是查看 array 的值：

```
0x5555555566f0 <array.3415>:    "maduiersnfotvbylSo you think you can stop the bomb with ctrl
-c, do you?"
```

分别数出’d’,’e’,’v’,’i’,’l’,’s’的对应位置：2, 5, 12, 4, 15, 7。

对应的 16 进制：0x2, 0x5, 0xc, 0x4, 0xf ,0x7。

这是高 4 位置零之后的结果，结合 ascii 码表，它的高四位可以为 0x2,0x3,0x4,0x5,0x6 任意一个。运算及查表结果如下：

(a)当高四位为 2：0x22, 0x25, 0x2c, 0x24, 0x2f, 0x27

'\"' , '%', ',' , '$', '/', '\' -> "%,$/'

(b)当高四位为 3：0x32, 0x35, 0x3c, 0x34, 0x3f, 0x37

'2' , '5', '<', '4', '?', '7' -> 25<4?7

(c)当高四位为 4：0x42, 0x45, 0x4c, 0x44, 0x4f, 0x47

'B' , 'E' , 'L' , 'D' , 'O' , 'G' -> BELDOG

(d)当高四位为 5：0x52, 0x55, 0x5c, 0x54, 0x5f, 0x57

'R' , 'U' , '\\' , 'T', '_' , 'W' -> RU\T_W

(e)当高四位为 6：0x62, 0x65, 0x6c, 0x64, 0x6f, 0x67

'b' , 'e' , 'l' , 'd' , 'o' , 'g' -> beldog

即得到最后的密码：（共 5^6 种）

第一个字符可以是’ 2 B R b 五个中任意一个

第二个字符可以是% 5 E U e 五个中任意一个

第三个字符可以是, < L \ l 五个中任意一个

第四个字符可以是$ 4 D T d 五个中任意一个

第五个字符可以是/ ? O _ o 五个中任意一个

第六个字符可以是’ 7 G W g 五个中任意一个

（注：我在 ans.txt 文件中选用的是 beldog）

3.6 阶段 6 的破解与分析

密码如下：1 5 3 6 4 2

破解过程：

把 phase_6 反汇编及分析过程如下（附注释）：

```
0x555555555390 <phase_6>: push   %r13
   0x555555555392 <phase_6+2>:    push   %r12
   0x555555555394 <phase_6+4>:    push   %rbp
   0x555555555395 <phase_6+5>:    push   %rbx
   0x555555555396 <phase_6+6>:    sub    $0x58,%rsp

     int num[6];
     int *p = num;
     read_six_numbers(input, num);
     int i = 0;
     int ax;
   0x55555555539a <phase_6+10>:   lea    0x30(%rsp),%r12  //p = num;

   0x55555555539f <phase_6+15>:   mov    %r12,%rsi
   0x5555555553a2 <phase_6+18>:   callq  0x5555555556f7 <read_six_numbers>


   0x5555555553a7 <phase_6+23>:   mov    $0x0,%r13d  //i = 0;
   0x5555555553ad <phase_6+29>:   jmp    0x5555555553d5 <phase_6+69>
   0x5555555553af <phase_6+31>:   callq  0x5555555556d1 <explode_bomb>
   0x5555555553b4 <phase_6+36>:   jmp    0x5555555553e4 <phase_6+84>
    for(bx = i; bx <= 5; bx++)
        {
            ax = num[bx];
            if(*p == ax)
                explode_bomb();
            bx++;
        }
   0x5555555553b6 <phase_6+38>:   add    $0x1,%ebx
   0x5555555553b9 <phase_6+41>:   cmp    $0x5,%ebx
   0x5555555553bc <phase_6+44>:   jg     0x5555555553d1 <phase_6+65>

   0x5555555553be <phase_6+46>:   movslq %ebx,%rax
   0x5555555553c1 <phase_6+49>:   mov    0x30(%rsp,%rax,4),%eax
   0x5555555553c5 <phase_6+53>:   cmp    %eax,0x0(%rbp)
   0x5555555553c8 <phase_6+56>:   jne    0x5555555553b6 <phase_6+38>
   0x5555555553ca <phase_6+58>:   callq  0x5555555556d1 <explode_bomb>
   0x5555555553cf <phase_6+63>:   jmp    0x5555555553b6 <phase_6+38>
   p++;
   0x5555555553d1 <phase_6+65>:   add    $0x4,%r12     //p++
   if((unsigned)(*p - 1) > 5)
            explode_bomb();
   0x5555555553d5 <phase_6+69>:   mov    %r12,%rbp
   0x5555555553d8 <phase_6+72>:   mov    (%r12),%eax
   0x5555555553dc <phase_6+76>:   sub    $0x1,%eax
   0x5555555553df <phase_6+79>:   cmp    $0x5,%eax
   0x5555555553e2 <phase_6+82>:   ja     0x5555555553af <phase_6+31>
   i++;
   0x5555555553e4 <phase_6+84>:   add    $0x1,%r13d  //i++

```

```
    if(i == 6)
              break;
    0x5555555553e8 <phase_6+88>:   cmp    $0x6,%r13d   //while (i < 6)
    0x5555555553ec <phase_6+92>:   je     0x555555555423 <phase_6+147>//circle

    0x5555555553ee <phase_6+94>:   mov    %r13d,%ebx    # bx = i
    0x5555555553f1 <phase_6+97>:   jmp    0x5555555553be <phase_6+46>


for(j = 0; j < 6; j++)
{
     cx = num[j];
     dx = node1;
     if(cx > 1)
     {
         for(k = 1; k < cx; k++)
             dx = dx -> next;
     }
     array[j] = dx;
}

    0x5555555553f3 <phase_6+99>:   mov    0x8(%rdx),%rdx    //dx = dx -> next;
    0x5555555553f7 <phase_6+103>:  add    $0x1,%eax         //k++;
    0x5555555553fa <phase_6+106>:  cmp    %ecx,%eax         //if (k < cx)
    0x5555555553fc <phase_6+108>:  jne    0x5555555553f3 <phase_6+99> //circle;

    0x5555555553fe <phase_6+110>:  mov    %rdx,(%rsp,%rsi,8)   array[j] = dx;
    0x555555555402 <phase_6+114>:  add    $0x1,%rsi   // j++;
    0x555555555406 <phase_6+118>:  cmp    $0x6,%rsi  //if (j == 6)
    0x55555555540a <phase_6+122>:  je  0x55555555542a <phase_6+154>  //break;
    0x55555555540c <phase_6+124>:  mov    0x30(%rsp,%rsi,4),%ecx  //cx = num[j];
    0x555555555410 <phase_6+128>:  mov    $0x1,%eax  //ax = 1
    0x555555555415 <phase_6+133>:
     lea    0x202df4(%rip),%rdx        # 0x555555758210 <node1>  //dx = node1;
    0x55555555541c <phase_6+140>:  cmp    $0x1,%ecx       // if (cx > 1)
    0x55555555541f <phase_6+143>:  jg     0x5555555553f3 <phase_6+99>  //jump
    0x555555555421 <phase_6+145>:  jmp    0x5555555553fe <phase_6+110>  //else

    0x555555555423 <phase_6+147>:  mov    $0x0,%esi
    0x555555555428 <phase_6+152>:  jmp    0x55555555540c <phase_6+124>

     bx = array[0];
     ax = array[1];
     bx -> next = ax;
     dx = array[2];
     ax -> next = dx;
     ax = array[3];
     dx -> next = ax;
     dx = array[4];
     ax -> next = dx;
     ax = array[5];
     dx -> next = ax;
     ax -> next = NULL;
    0x55555555542a <phase_6+154>:  mov    (%rsp),%rbx   //bx = array[0];
    0x55555555542e <phase_6+158>:  mov    0x8(%rsp),%rax   // ax = array[1]
    0x555555555433 <phase_6+163>:  mov    %rax,0x8(%rbx)  // bx -> next = ax;
    0x555555555437 <phase_6+167>:  mov    0x10(%rsp),%rdx // dx = array[2];
    0x55555555543c <phase_6+172>:  mov    %rdx,0x8(%rax)  // ax -> next = dx;
    0x555555555440 <phase_6+176>:  mov    0x18(%rsp),%rax // ax = array[3];
    0x555555555445 <phase_6+181>:  mov    %rax,0x8(%rdx)  // dx -> next = ax;
    0x555555555449 <phase_6+185>:  mov    0x20(%rsp),%rdx  //dx = array[4];
```

```
112.    0x55555555544e <phase_6+190>:  mov    %rdx,0x8(%rax)   //ax -> next = dx;
113.    0x555555555452 <phase_6+194>:  mov    0x28(%rsp),%rax  // ax = array[5];
114.    0x555555555457 <phase_6+199>:  mov    %rax,0x8(%rdx)   // dx -> next = ax;
115.    0x55555555545b <phase_6+203>:  movq   $0x0,0x8(%rax)   // ax -> next = NULL
116.
117.
118.
119.     for(k = 5; k > 0; k--)
120.     {
121.         ax = bx -> next;
122.         if(bx -> data > ax -> data)
123.             explode_bomb();
124.         bx = bx ->next;
125.     }
126.    0x555555555463 <phase_6+211>:  mov    $0x5,%ebp
127.    0x555555555468 <phase_6+216>:  jmp    0x555555555473 <phase_6+227>
128.    0x55555555546a <phase_6+218>:  mov    0x8(%rbx),%rbx
129.    0x55555555546e <phase_6+222>:  sub    $0x1,%ebp
130.    0x555555555471 <phase_6+225>:  je     0x555555555484 <phase_6+244>
131.    0x555555555473 <phase_6+227>:  mov    0x8(%rbx),%rax
132.    0x555555555477 <phase_6+231>:  mov    (%rax),%eax
133.    0x555555555479 <phase_6+233>:  cmp    %eax,(%rbx)
134.    0x55555555547b <phase_6+235>:  jle    0x55555555546a <phase_6+218>
135.    0x55555555547d <phase_6+237>:  callq  0x5555555556d1 <explode_bomb>
136.    0x555555555482 <phase_6+242>:  jmp    0x55555555546a <phase_6+218>
137.    0x555555555484 <phase_6+244>:  add    $0x58,%rsp
138.    0x555555555488 <phase_6+248>:  pop    %rbx
139.    0x555555555489 <phase_6+249>:  pop    %rbp
140.    0x55555555548a <phase_6+250>:  pop    %r12
141.    0x55555555548c <phase_6+252>:  pop    %r13
142.    0x55555555548e <phase_6+254>:  retq
143.
```

整理上述注释中等效的 c 代码可得（**分析见注释！**）：

```
1.  typedef struct NODE
2.  {
3.      long data;
4.      struct NODE* next;
5.  } node;
6.
7.  node *node1;
8.
9.  void read_six_numbers(char *input, int *a)
10. {
11.     if(sscanf(input, "%d %d %d %d %d %d", a, a + 1, a + 2, a + 3, a + 4, a + 5
    ) <= 5)
12.         explode_bomb();
13.     return ;
14. }
15.
16. void phase(char *input)
17. {
```

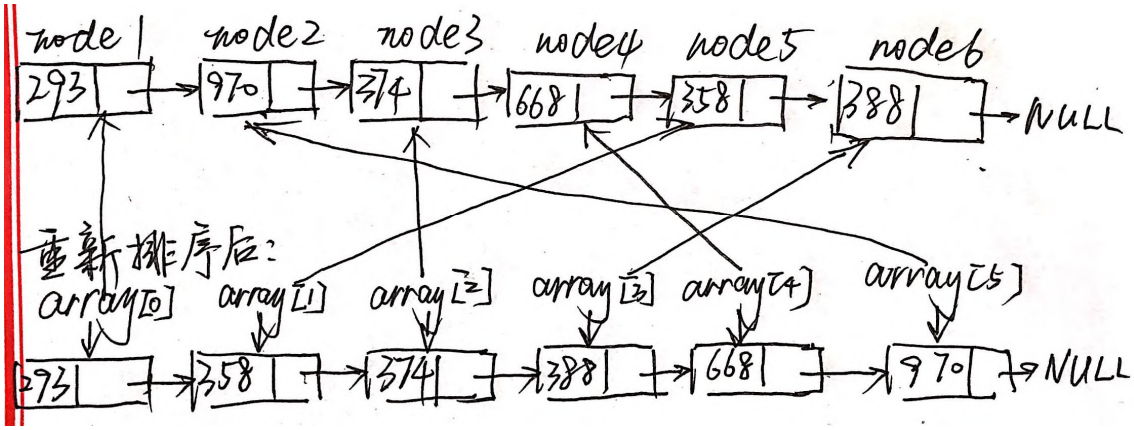
```
    int num[6];  //num in 0x30(rsp)
    int *p = num; //p in %r12
    read_six_numbers(input, num);
    int i = 0;  // i in %r13d
    int ax;
    build(node1);  //以 node1 为表头，构建了一个单链表
    while(i < 6)
    {
        if((unsigned)(*p - 1) > 5) /*若 p 指向的数大于 6，
                                    或小于 1，则爆炸*/
            explode_bomb();
        i++;
        if(i == 6)
            break; //如果 i 到了 6，提前结束，不执行下面的 for 循环
        for(bx = i; bx <= 5; bx++)
        {
            // ax = i;
            ax = num[bx];  //从第 i 个数（即 p 指向的数）开始取
            if(*p == ax)   //若取得这个数等于 p 指向的数，则爆炸
                explode_bomb();
            bx++;          //每次向后取一个
        }
        p++;      //p 向后指一位
    }
    /****************************************
    以上一段的总结：num 中的数，只能在 1 ~ 6 之间，且不能重复
    结论：num[]中的数为 1,2,3,4,5,6,（顺序待定）
    ****************************************/


    node *array[];
    int j, k;  //j in %si
    int cx;
    node *bx, *ax;
    node *dx;
    for(j = 0; j < 6; j++)
    {
        cx = num[j];    //cx 为 num 的第 j 项
        dx = node1;  //dx 指向链表第一项

        if(cx > 1)
        {
            for(k = 1; k < cx; k++)  //指针 dx 向后移 cx 次
                dx = dx -> next;
        }      //此时，dx 指向第 cx 个元素
        array[j] = dx;    /*链表的第 “num[j]” 个节点地址，
                            给了指针数组 array[j]*/
    }
    /****************************************
    以上一段的总结：链表的第 “num[j]” 个节点地址，给了 array[j]
    结论：只要找到 array[]各个项，是链表的第几个节点，
          就能找出相应 num[]的值
    ****************************************/


    bx = array[0];
    ax = array[1];
    bx -> next = ax; /*把 array[1]指向的节点
```

```
                        连在 array[0]指向的节点的后面*/
    dx = array[2];
    ax -> next = dx; /*把 array[2]指向的节点
                        连在 array[1]指向的节点的后面*/
    ax = array[3];
    dx -> next = ax; /*把 array[3]指向的节点
                        连在 array[2]指向的节点的后面*/
    dx = array[4];
    ax -> next = dx; /*把 array[4]指向的节点
                        连在 array[3]指向的节点的后面*/
    ax = array[5];
    dx -> next = ax; /*把 array[5]指向的节点
                        连在 array[4]指向的节点的后面*/
    ax -> next = NULL; //最后几个节点 next 节点赋为 NULL
    /**************************************
    以上一段的总结：把原来的链表，按照 array
                    数组的顺序重新排序
    **************************************/


   //这个时候，bx 指向链表的第一个节点
    int k;
     for(k = 5; k > 0; k--)
     {
         ax = bx -> next; //ax 指向 bx 的后趋节点
         if(bx -> data > ax -> data)   /*如果 bx 节点的值
                                         大于 ax 节点的值,则爆炸*/
             explode_bomb();
         bx = bx -> next;   //bx 向后移动一次
     }
}
     /**************************************
     以上一段的总结：新的链表，必须是从小到大排列，否则会爆炸

     结论：要想不爆炸，array[]依次指向的节点，值必须是从小到大的
     **************************************/
```

查看原链表的值即顺序：

```
(gdb) x/1dw 0x555555758210
0x555555758210 <node1>: 293
(gdb) x/1xg 0x555555758210+8
0x555555758218 <node1+8>:       0x0000555555758220
(gdb) x/1dw 0x0000555555758220
0x555555758220 <node2>: 970
(gdb) x/1xg 0x0000555555758220+8
0x555555758228 <node2+8>:       0x0000555555758230
(gdb) x/1dw 0x0000555555758230
0x555555758230 <node3>: 374
(gdb) x/1xg 0x0000555555758230+8
0x555555758238 <node3+8>:       0x0000555555758240
(gdb) x/1dw 0x0000555555758240
0x555555758240 <node4>: 668
(gdb) x/1xg 0x0000555555758240+8
0x555555758248 <node4+8>:       0x0000555555758250
(gdb) x/1dw0x0000555555758250
0x555555758250 <node5>: 358
(gdb) x/1xg 0x0000555555758250+8
0x555555758258 <node5+8>:       0x0000555555758110
(gdb) x/1dw 0x0000555555758110
0x555555758110 <node6>: 388
(gdb) x/1xg 0x0000555555758110+8
0x555555758118 <node6+8>:       0x0000000000000000
```

得到如图所示链表：



通过重新从大到小排序，可以看出：

array[0]指向 node1；

array[1]指向 node5；

array[2]指向 node3；

array[3]指向 node6；

array[4]指向 node4；

array[5]指向 node2；

根据我上面注释中的分析：

```
/
以上一段的总结：链表的第“num[j]”个节点地址，给了 array[j]
结论：只要找到 array[]各个项，是链表的第几个节点，
      就能找出相应 num[]的值
```

num[] = {1, 5, 3, 6, 4, 2}.

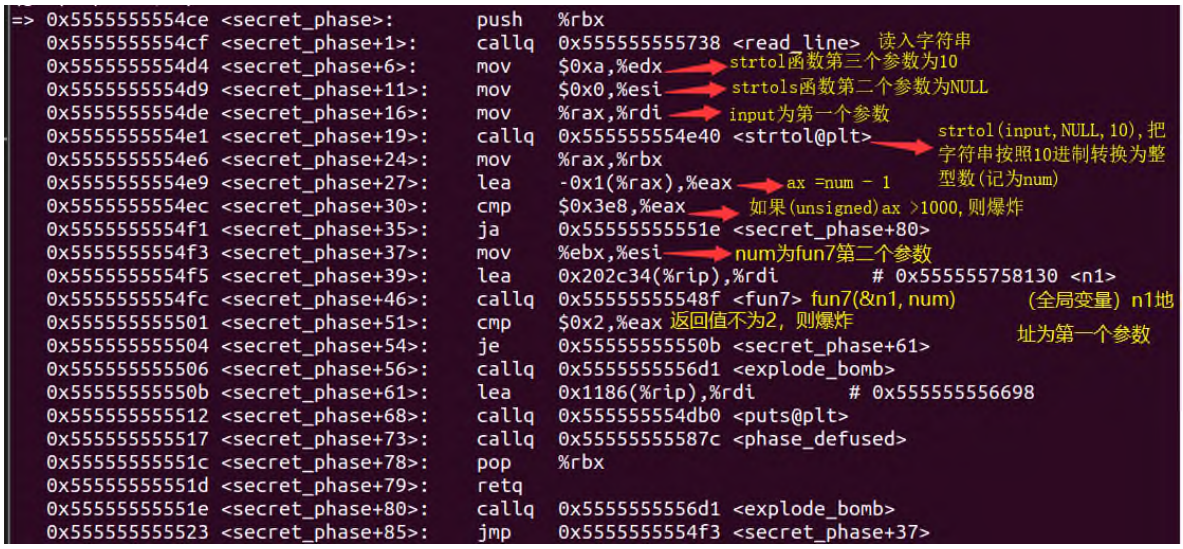
即，1 5 3 6 4 2 是最终密码

3.7 阶段 7 的破解与分析(隐藏阶段)

密码如下：20

破解过程：

果不其然，阶段 4 的分析是正确的。出现了隐藏关卡（进入方法见阶段 4 末尾），

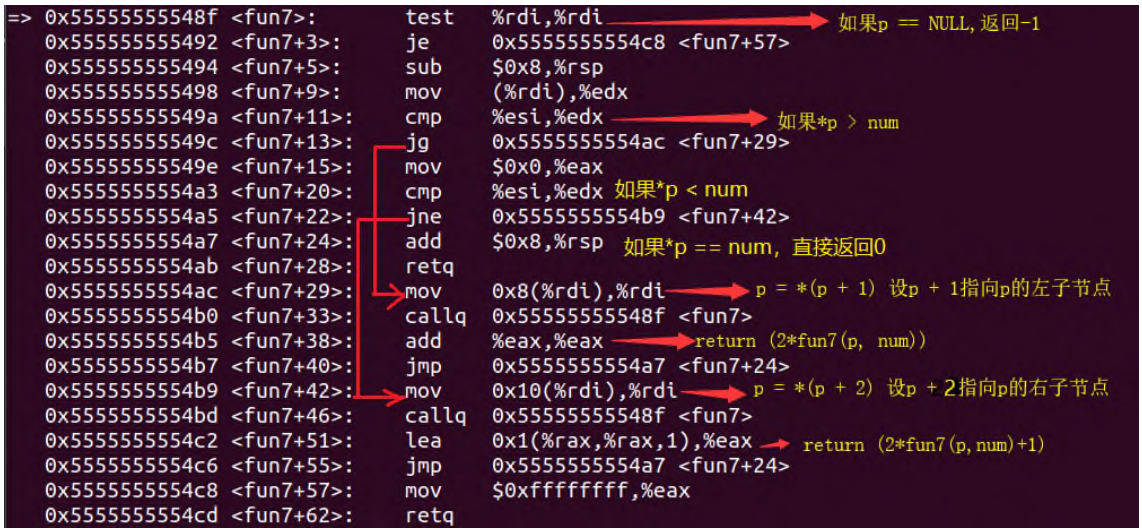



```
(gdb) x/s 0x555555556698
0x555555556698: "Wow! You've defused the secret stage!"
```

写成 c 语言形式：

```c
void phase_secret(void)
{
    char *input = read_line();
    long num = strtol(input, NULL, 10);
    long bx = num;
    long ax = num - 1;
    if(ax > 1000)
        explode_bomb();
    long n1 = 36;
    if(fun7(&n1, num) != 2)
        explode_bomb();
    puts("Wow! You've defused the secret stage!");
    phase_defused();
    return;
}
```

下面研究 fun7 函数：



由于用的是 rdi，且使用 p 的地址加 8 来访问结构体的第二个元素，因此 p 指向的节点的数据段是 long 型

整理为 c 代码：

```c
long fun7(long *p, long num)
{
    long ax;
    if(p == NULL)
        return -1;

    // ax = 0;
    if(*p == num)
        return 0;//ax;

    else if(*p < num)
    {
        p = *(p + 2); //right
        return (2 * fun7(p, num) + 1);
    }

    else
    {
        p = *(p + 1); //left
        return (2 * fun7(p,num));
    }
}
```

可以推测，p 指向的数据结构，是一个二叉树。下面来查看这个二叉树的值：

查看命令过程如下：

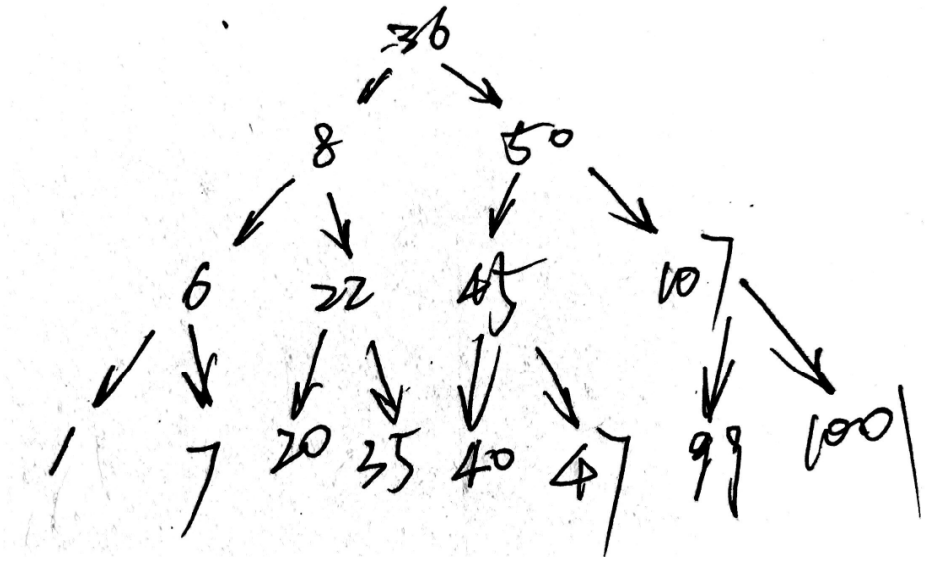
```
(gdb) x/1dg 0x555555758130
```

```
0x555555758130 <n1>: 36
(gdb) x/1xg 0x555555758130+8
0x555555758138 <n1+8>:  0x0000555555758150
(gdb) x/1dg 0x0000555555758150
0x555555758150 <n21>:   8
(gdb) x/1xg 0x555555758130+16
0x555555758140 <n1+16>: 0x0000555555758170
(gdb) x/1dg 0x0000555555758170
0x555555758170 <n22>:   50
(gdb) x/1xg 0x0000555555758150+8
0x555555758158 <n21+8>: 0x00005555557581d0
(gdb) x/1dg 0x00005555557581d0
0x5555557581d0 <n31>:   6
(gdb) x/1xg 0x0000555555758150+16
0x555555758160 <n21+16>:0x0000555555758190
(gdb) x/1dg 0x0000555555758190
0x555555758190 <n32>:   22
(gdb) x/1xg 0x0000555555758170+8
0x555555758178 <n22+8>: 0x00005555557581b0
(gdb) x/1dg 0x00005555557581b0
0x5555557581b0 <n33>:   45
(gdb) x/1xg 0x0000555555758170+16
0x555555758180 <n22+16>:0x00005555557581f0
(gdb) x/1dg 0x00005555557581f0
0x5555557581f0 <n34>:   107
(gdb) x/1xg 0x00005555557581d0+8
0x5555557581d8 <n31+8>: 0x0000555555758030
(gdb) x/1dg 0x0000555555758030
0x555555758030 <n41>:   1
(gdb) x/1xg 0x00005555557581d0+0x10
0x5555557581e0 <n31+16>:0x0000555555758090
(gdb) x/1xg 0x00005555557581d0+16
0x5555557581e0 <n31+16>:0x0000555555758090
(gdb) x/1dg 0x0000555555758090
0x555555758090 <n42>:   7
(gdb) x/1xg 0x0000555555758190+8
0x555555758198 <n32+8>: 0x00005555557580b0
```

```
(gdb) x/1dg 0x00005555557580b0
0x5555557580b0 <n43>:   20
(gdb) x/1xg 0x0000555555758190+16
0x5555557581a0 <n32+16>:0x0000555555758070
(gdb) x/1dg 0x0000555555758070
0x555555758070 <n44>:   35
(gdb) x/1xg 0x00005555557581b0+8
0x5555557581b8 <n33+8>: 0x0000555555758010
(gdb) x/1dg 0x0000555555758010
0x555555758010 <n45>:   40
(gdb) x/1xg 0x00005555557581b0+16
0x5555557581c0 <n33+16>:0x00005555557580d0
(gdb) x/1dg 0x00005555557580d0
0x5555557580d0 <n46>:   47
(gdb) x/1xg 0x5555557581f0+8
0x5555557581f8 <n34+8>: 0x0000555555758050
(gdb) x/1dg 0x0000555555758050
0x555555758050 <n47>:   99
(gdb) x/1xg 0x5555557581f0+16
0x555555758200 <n34+16>:0x00005555557580f0
(gdb) x/1dg 0x00005555557580f0
0x5555557580f0 <n48>:   1001
(gdb) x/1xg 0x0000555555758030+8
0x555555758038 <n41+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x0000555555758030+16
0x555555758040 <n41+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758090+8
0x555555758098 <n42+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758090+16
0x5555557580a0 <n42+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x00005555557580b0+8
0x5555557580b8 <n43+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x00005555557580b0+16
0x5555557580c0 <n43+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758070+8
0x555555758078 <n44+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758070+16
```

```
0x555555758080 <n44+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758010+8
0x555555758018 <n45+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758010+16
0x555555758020 <n45+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x5555557580d0+8
0x5555557580d8 <n46+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x5555557580d0+16
0x5555557580e0 <n46+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758050+8
0x555555758058 <n47+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x555555758050+16
0x555555758060 <n47+16>:0x0000000000000000  //NULL
(gdb) x/1xg 0x5555557580f0+8
0x5555557580f8 <n48+8>: 0x0000000000000000  //NULL
(gdb) x/1xg 0x5555557580f0+16
0x555555758100 <n48+16>: 0x0000000000000000  //NULL
```

据上述结果，画出这个树：



显然这是一个二叉排序树，左子树节点的值 < 右子树节点的值。下面分析，如何让递归函数 fun7(&n1, num)的返回值为 2：

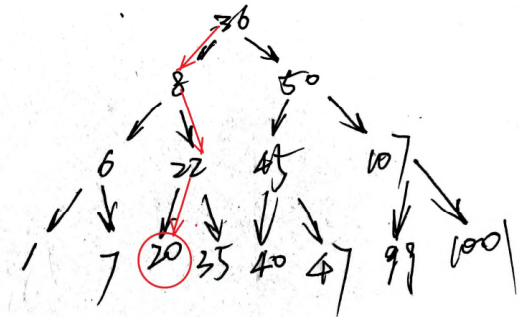
```
23. long fun7(long *p, long num)
```

```
{
    long ax;
    if(p == NULL)
        return -1;

    // ax = 0;
    if(*p == num)
        return 0;//ax;

    else if(*p < num)
    {
        p = *(p + 2); //right
        return (2 * fun7(p, num) + 1);
    }

    else
    {
        p = *(p + 1); //left
        return (2 * fun7(p,num));
    }
}
```

遍历所有情况显然工作量过大。

每当指针直到空节点的时候，则返回-1，递归函数进入回归阶段。
如果中途碰到某节点等于 num，则直接返回 0，递归函数直接进入回归阶段。
若以上情况都不是，则继续递推。
观察可知，每次递推，返回值都会有一个 “乘以 2”的操作，若最后遇到空节点，返回值多次乘 2，最后显然不会等于 2，所以得出结论：**指针中途一定碰到了某节点等于 num。**
所以，最深层的递推阶段，返回值一定是 0；若这个节点为左子节点，则回归到上一层时，返回 2 * fun7，上层返回依然是 0；若这个节点为右子节点，则回归上层时，返回 2 * fun7 + 1，上层返回 1。这个时候发现，如果这个时候，再乘一个 2（即这个节点是左子节点），我们想要的“返回值为 2”就出来了。

因此，我们发现一种可行的形式已经出来了：
从头结点开始：左（返回 1 * 2 = 2）
右（返回 0 + 1 = 1）
左（最底层返回 0）



因此，num = 20，就能保证 p 的移动按照上图红箭头所示。
即 20 即为最终答案。

完结！撒花！

```
xjy1170500913@ubuntu:~/桌面/hitics/lab3$ ./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...
20
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

第 4 章 总结

4.1 请总结本次实验的收获

1. 熟悉了 gdb 调试命令；
2. 分析汇编代码更加熟练；
3. 训练了逻辑思维能力；
4. 拆 bomb 有种打游戏的感觉，拆完 bomb 的我很快乐。

4.2 请给出对本次实验内容的建议

无，拆 bomb 好玩又有收获。

注：本章为酌情加分项。

参考文献

[1] 大卫 R.奥哈拉伦，兰德尔 E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社.2018.4