

华东师范大学软件学院实验报告

实验课程：计算机系统

年级：23 级本科

实验成绩：

实验名称：Labs (2) Binary Bomb

姓名：张梓卫

实验编号：（2）

学号：10235101526 实验日期：2024/04/02

指导教师：肖波

组号：

一、实验目的

1、初步掌握 gdb 调试工具，再通过本实验提高对汇编语言的理解，更加熟练 Linux 系统的使用。

2、通过对汇编语言的分析，理解运算和逻辑语句在汇编语言中的体现形式，为后续学习打基础。

二、实验前置准备

1、将 bomb.tar 置于 Ubuntu 20.06 之中解压，安装 gdb（之前已安装）

```
deralive@10235101526: ~/Binary Bomb/bomb
deralive@10235101526:~/Binary Bomb/bomb$ sudo apt-get install gdb
[sudo] deralive 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
gdb 已经是最新版 (12.1-0ubuntu1~22.04)。
gdb 已设置为手动安装。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 20 个软件包未被升级。
```

2、运行命令，获取 bomb 的汇编语言以供后续阅读。



```
deralive@10235101526: ~/Binary Bomb/bomb
deralive@10235101526:~/Binary Bomb/bomb$ objdump -d bomb > bomb_assembly.S
deralive@10235101526:~/Binary Bomb/bomb$
```

3、Unix > gdb bomb, 试运行查看炸弹情况：

键入 run, 以运行程序,

随意键入一句话, 如: “Don’t Bomb, Please!”

出现炸弹爆炸语句, 说明实验前置工作准备完成。

```

deralive@10235101526:~/Binary Bomb/bomb$ objdump -d bomb > bomb_assembly.S
deralive@10235101526:~/Binary Bomb/bomb$ gdb bomb
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) run
Starting program: /home/deralive/Binary Bomb/bomb/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Don't Bomb, Please!

BOOM!!!
The bomb has blown up.
[Inferior 1 (process 2669) exited with code 010]
(gdb)

```

三、实验过程分析

1、Phase 1:

由于在 Ubuntu 中不熟练编辑.S, 故使用 VSCode 打开进行查看与编辑, 更容易编辑注释。

346	000000000400ee0 <phase_1>:			
347	400ee0: 48 83 ec 08	sub	\$0x8,%rsp	# 栈指针向下8位,
348	400ee4: be 00 24 40 00	mov	\$0x402400,%esi	# 将内存地址为 0x402400 的数据移入 esi 寄存器
349	400ee9: e8 4a 04 00 00	call	401338 <strings_not_equal>	# 调用“字符串不相等”函数
350	400eee: 85 c0	test	%eax,%eax	# 进行自身按位与运算, 猜测用于判断输入的字符串是否相等
351	400ef0: 74 05	je	400ef7 <phase_1+0x17>	# 条件跳转指令, 若上述条件符合, 则跳转到0x400ef7处
352	400ef2: e8 43 05 00 00	call	40143a <explode_bomb>	# 猜测字符串不相等时, 调用explode_bomb函数, 导致炸弹爆炸
353	400ef7: 48 83 c4 08	add	\$0x8,%rsp	# 对应 sub 命令, 使得栈指针上移
354	400efb: c3	ret		

我们要避开炸弹.则需要第 351 行 (400ef0) 跳到 353 行 (400ef7), 自身按位与运算, 猜测用于判断输入的字符串是否相等, 故只需找到内存 0x402400 存放的数据即可。

C 语言的伪代码如下所示:

```

void phase_1 (char* rdi) {
    char* esi = (char*)0x402400;
    int eax = string_no_equal(rdi, esi);
    if (eax == 0) {
        explode_bomb();
    }
    return;
}

```

使用 gdb print (char*) 0x402400 查看该字符串

```

deralive@10235101526: ~/Binary Bomb/bomb
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) print (char*) 0x402400
$1 = 0x402400 "Border relations with Canada have never been better."

```

使用(gdb) run 指令进入第一阶段的拆弹环节，并输入对应的字符串，拆弹完成
对应字符串答案为 Border relations with Canada have never been better.

```

(gdb) run
Starting program: /home/deralive/Binary Bomb/bomb/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?

```

2、Phase 2:

首先要理解基础知识：第一个参数到第六个参数分别存储在：

%rdi,%rsi,%rdx,%rcx,%r8,%r9 寄存器中，%rsp 是此时栈指针指向的位置

我们先关注一些条件语句的汇编指令：

Cmp 指令 Cmp a,b：如果 a-b=0，就设置零标志为 1（只进行比较，不改变值），两种跳转指令如下，一种是无条件跳转，另一种是根据标志位跳转：

无条件跳转：JMP

根据标志位跳转：

JE 等于则跳转	JA 无符号大于则跳转	JBE 无符号小于等于则跳转
JNE 不等于则跳转	JNA 无符号不大于则跳转	JNBE 无符号不小于等于则跳转
JZ 为 0 则跳转	JAE 无符号大于等于则跳转	JL 有符号小于则跳转
JNZ 不为 0 则跳转	JNAE 无符号不大于等于则跳转	JNL 有符号不小于则跳转
JS 为负则跳转（符号位 1）	JG 有符号大于则跳转	JLE 有符号小于等于则跳转
JNS 不为负则跳转	JNG 有符号不大于则跳转	JNLE 有符号不小于等于则跳转
JC 进位则跳转	JGE 有符号大于等于则跳转	JP 奇偶位置位则跳转
JNC 不进位则跳转	JNGE 有符号不大于等于则跳转	JNP 奇偶位清除则跳转
JO 溢出则跳转	JB 无符号小于则跳转	JPE 奇偶位相等则跳转
JNO 不溢出则跳转	JNB 无符号不小于则跳转	JPO 奇偶位不等则跳转

Lea 指令不进行数据传送，它只把一个内存地址的计算结果存入寄存器中，而不是将内存中的“数据”传送到寄存器。

使用(gdb) disas phase_2 指令可以在 Linux 终端中查看汇编代码，为了排版美观，我们继续使用 Vscode 来进行每一行的编辑。

```
000000000400efc <phase_2>:
400efc: 55                push    %rbp

400efd: 53                push    %rbx
400efe: 48 83 ec 28       sub     $0x28,%rsp          # 栈指针向下40位，为本函数变量提供空间
400f02: 48 89 e6          mov     %rsp,%rsi          # 将栈指针移入寄存器%rsi (第二个变量)
400f05: e8 52 05 00 00    call   40145c <read_six_numbers> # 读取六个数字

400f0a: 83 3c 24 01       cmpl    $0x1,(%rsp)        # %rsp是栈指针,与立即数1作比较
400f0e: 74 20            je      400f30 <phase_2+0x34> # 比较结果若相等，则跳至(400efc + 0x34) = 400f30

400f10: e8 25 05 00 00    call   40143a <explode_bomb> # 若不相等，则炸弹爆炸，相等则跳过该行调用
400f15: eb 19            jmp     400f30 <phase_2+0x34> # 无条件跳转至 (400efc + 0x34) = 400f30

400f17: 8b 43 fc          mov     -0x4(%rbx),%eax     # 将 (%rbx - 4) 移动到 %eax 寄存器
400f1a: 01 c0            add     %eax,%eax          # 将 %eax * 2
400f1c: 39 03            cmp     %eax,(%rbx)        # 比较 %eax 和 %rbx 指向的值
400f1e: 74 05            je      400f25 <phase_2+0x29> # 若相等，则跳转至 400f25
400f20: e8 15 05 00 00    call   40143a <explode_bomb> # 否则调用炸弹爆炸函数
400f25: 48 83 c3 04       add     $0x4,%rbx          # 不相等，继续将 %rbx + 4，即指向下一个元素
400f29: 48 39 eb          cmp     %rbp,%rbx          # %rbp - %rbx != 0,
400f2c: 75 e9            jne     400f17 <phase_2+0x1b> # 则跳转到 400f17,继续执行以上过程

400f2e: eb 0c            jmp     400f3c <phase_2+0x40> # 若上述指令都执行完，没出现问题，就跳转至400f3c, 函数结束
400f30: 48 8d 5c 24 04     lea     0x4(%rsp),%rbx      # 将 (栈指针指向的地址 + 4) 后的结果
# 移动进入 %rbx 暂存 (其实就是压入栈内的下一个元素)

400f35: 48 8d 6c 24 18     lea     0x18(%rsp),%rbp     # 将 %rsp 栈指针 + 24，移动进入 %rbp 暂存 (栈的最后一个元素)
400f3a: eb db            jmp     400f17 <phase_2+0x1b> # 无条件跳转至 (400efc + 0x1b) = 400f17

400f3c: 48 83 c4 28       add     $0x28,%rsp          # 回收栈指针和返回等操作
400f40: 5b                pop     %rbx
400f41: 5d                pop     %rbp
400f42: c3                ret
```

根据上述分析：我们可以得知，栈指针下移 40 位，相当于首先开辟了十个元素的数组，进入一个循环体，从 1 开始，到 7 结束。C 语言的伪代码如下所示：

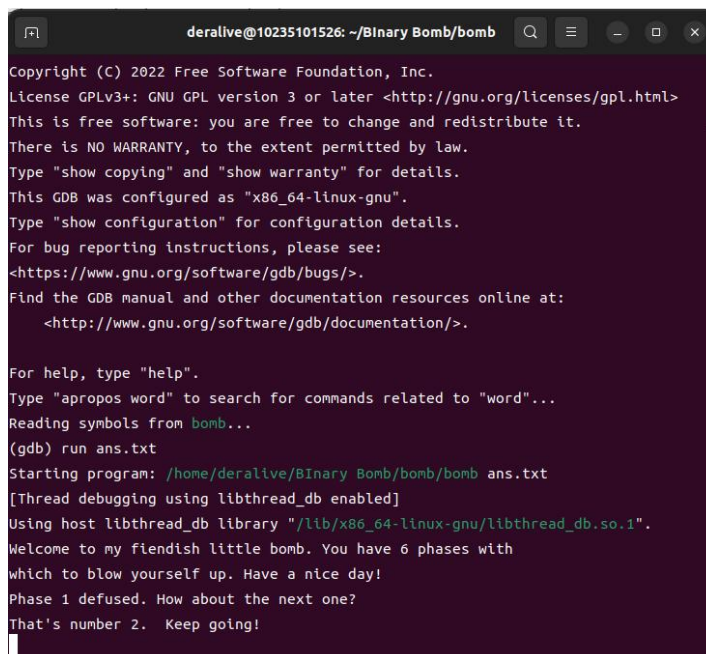
```
void phase_2(char* get) {

    int arr[10];
    read_six_numbers(get, arr[]);
    if (arr[1] != 1) {
        explode_bomb();
    }
    for (int i = 2; i < 6; i++) {
        if (arr[i] != 2 * arr[i - 1]) {
            explode_bomb();
        }
    }
}
```


根据分析，已可以得到，Phase2 的拆弹答案为一个以 2 为公比的等比数列，数据个数为 6 个，答案为：

1 2 4 8 16 32

将答案写入 ans.txt，结果正确。



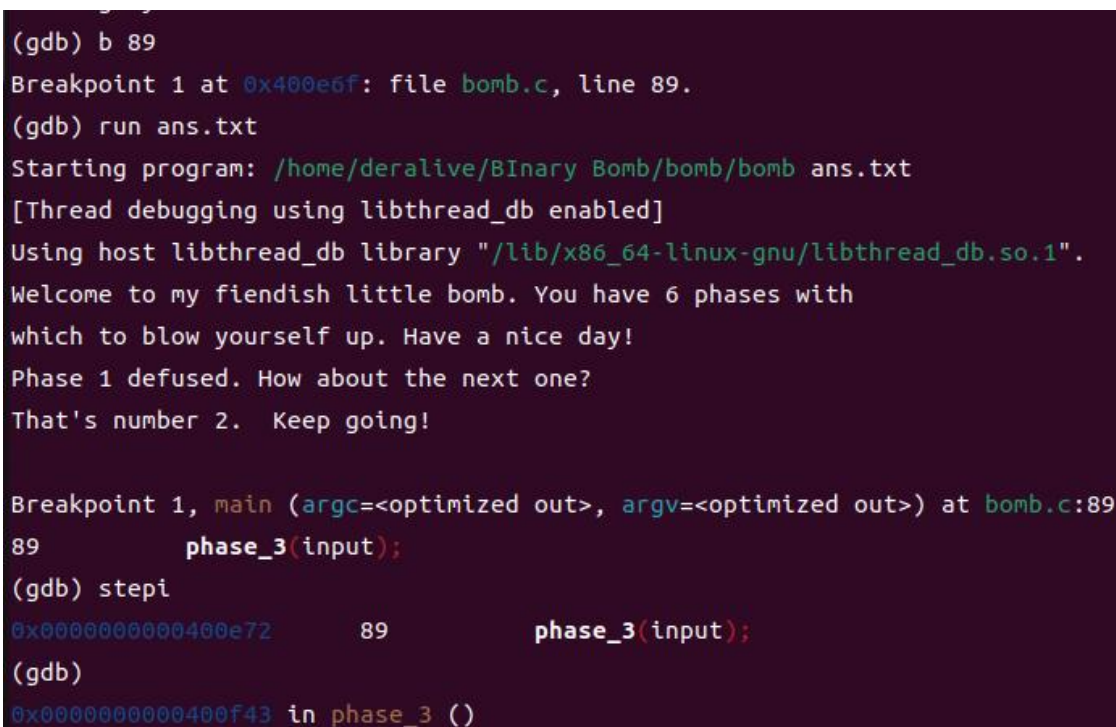
```
deralve@10235101526: ~/Binary Bomb/bomb
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) run ans.txt
Starting program: /home/deralve/Binary Bomb/bomb/bomb ans.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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!
```

运行结果截图，Bomb Phase 2 已拆除

3、Phase 3:

在 bomb.c 文件中，phase_3 在第 89 行，所以先使用 gdb bomb 命令，再使用 b 89 在调试环境中打上一个断点，执行命令 stepi（单步执行）以查看函数运行的过程：



```
deralve@10235101526: ~/Binary Bomb/bomb
(gdb) b 89
Breakpoint 1 at 0x400e6f: file bomb.c, line 89.
(gdb) run ans.txt
Starting program: /home/deralve/Binary Bomb/bomb/bomb ans.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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!

Breakpoint 1, main (argc=<optimized out>, argv=<optimized out>) at bomb.c:89
89      phase_3(input);
(gdb) stepi
0x0000000000400e72      89      phase_3(input);
(gdb)
0x0000000000400f43 in phase_3 ()
```

```
0x00000000400f5b in phase_3 ()
(gdb)
0x00000000400bf0 in __isoc99_sscanf@plt ()
(gdb)
__GI___isoc99_sscanf (s=0x603820 <input_strings+160> "123 456" format=0x4025cf "%d %d" at ./stdio-common/isoc99_sscanf.c:24
24      ./stdio-common/isoc99_sscanf.c: 没有那个文件或目录.
(gdb)
```

由上图可知，不难发现，题目要求我们输入两个十进制正整数（Decimal）。

直到函数由 <phase_3> 进入 <__isoc99_sscanf@plt>，这是一个 C99 标准的 sscanf 函数，为了保证题解的完整和精准，我查看了 sscanf 函数的原型：

描述

C 库函数 `int sscanf(const char *str, const char *format, ...)` 从字符串读取格式化输入。

声明

下面是 sscanf() 函数的声明。

```
int sscanf(const char *str, const char *format, ...)
```

接下来，我们继续分析 phase_3 的汇编代码：

```
00000000400f43 <phase_3>:
400f43: 48 83 ec 18      sub    $0x18,%rsp          # 栈指针下移 24，为变量提供空间
400f47: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx      # %rcx = (%rsp + 0xc)，存放第二个数
400f4c: 48 8d 54 24 08    lea    0x8(%rsp),%rdx      # %rdx = (%rsp + 0x8)，存放第一个数

// 为什么是 + 0x8：因为 + 0x4 的位置存放的是输入字符串的首地址，是为了后续调用sscanf做准备

400f51: be cf 25 40 00    mov    $0x4025cf,%esi      # %esi = $0x4025cf
400f56: b8 00 00 00 00    mov    $0x0,%eax           # %eax = 0
```

通过查询上表（见 Phase_2 的前提知识），Ja 命令是严格大于，而不参与等于的判断，所以 Switch 对应的跳转表实际上可取值的范围是[0,7]。我们继续看下方的汇编代码：

```
400f5b: e8 90 fc ff ff    call   400bf0 <__isoc99_sscanf@plt>
400f60: 83 f8 01          cmp    $0x1,%eax           # 函数调用后有返回值
400f63: 7f 05            jg     400f6a <phase_3+0x27> # 如果输入个数大于1，则跳转到 400f60a
400f65: e8 d0 04 00 00    call   40143a <explode_bomb> # 否则炸弹就会引爆

400f6a: 83 7c 24 08 07    cmpl   $0x7,0x8(%rsp)      # 比较第一个参数与7的大小关系
400f6f: 77 3c            ja     400fad <phase_3+0x6a> # 如果参数大于7，则跳转至 400fad
400f71: 8b 44 24 08       mov    0x8(%rsp),%eax      # 不大于，那么将参数放至 % eax 中
400f75: ff 24 c5 70 24 40 00 jmp     *0x402470(,%rax,8)   # 跳转表格式，跳转至 *(0x402470) = 0x400f7c
```

jmpq *0x402470(,%rax,8)中的 0x402470 为跳转表的首地址，这是 switch 跳转语句，即跳转到以地址*0x402470 为基址的跳转表中。我们可以查看这个跳转表中的地址元素。

(为了确保显示完整, 使用 x/32x 命令而不是 x/16x 或 x/8x)

```
(gdb) x/32x 0x402470
0x402470:      0x00400f7c      0x00000000      0x00400fb9      0x00000000
0x402480:      0x00400f83      0x00000000      0x00400f8a      0x00000000
0x402490:      0x00400f91      0x00000000      0x00400f98      0x00000000
0x4024a0:      0x00400f9f      0x00000000      0x00400fa6      0x00000000
0x4024b0 <array.3449>: 0x7564616d      0x73726569      0x746f666e      0x6c796276
0x4024c0:      0x79206f53      0x7420756f      0x6b6e6968      0x756f7920
0x4024d0:      0x6e616320      0x6f747320      0x68742070      0x6f622065
0x4024e0:      0x7720626d      0x20687469      0x6c727463      0x202c632d

400f7c: b8 cf 00 00 00      mov     $0xcf,%eax      # %eax = $0xcf
400f81: eb 3b                jmp     400fbe <phase_3+0x7b> # 跳转至 400fbe
400f83: b8 c3 02 00 00      mov     $0x2c3,%eax     # %eax = $0x2c3
400f88: eb 34                jmp     400fbe <phase_3+0x7b> # 跳到后面判断第二个参数和 (%eax) 是否相同
400f8a: b8 00 01 00 00      mov     $0x100,%eax     # %eax = $0x100
400f8f: eb 2d                jmp     400fbe <phase_3+0x7b>
400f91: b8 85 01 00 00      mov     $0x185,%eax     # %eax = $0x185
400f96: eb 26                jmp     400fbe <phase_3+0x7b>
400f98: b8 ce 00 00 00      mov     $0xce,%eax     # %eax = $0xce
400f9d: eb 1f                jmp     400fbe <phase_3+0x7b>
400f9f: b8 aa 02 00 00      mov     $0x2aa,%eax     # %eax = $0x2aa
400fa4: eb 18                jmp     400fbe <phase_3+0x7b>
400fa6: b8 47 01 00 00      mov     $0x147,%eax     # %eax = $0x147
400fab: eb 11                jmp     400fbe <phase_3+0x7b>

400fad: e8 88 04 00 00      call    40143a <explode_bomb> # 炸弹爆炸

400fb2: b8 00 00 00 00      mov     $0x0,%eax       # %eax = 0
400fb7: eb 05                jmp     400fbe <phase_3+0x7b>
400fb9: b8 37 01 00 00      mov     $0x137,%eax     # %eax = 0x137
// 这里可以补上一句,方便理解:
400fb9: jmp     400fbe <phase_3+0x7b>

400fbe: 3b 44 24 0c          cmp     0xc(%rsp),%eax   # 比较第二个参数 与 目前 %eax 的值
400fc2: 74 05                je      400fc9 <phase_3+0x86> # 相等, 则跳转至 400fc9 拆弹成功
400fc4: e8 71 04 00 00      call    40143a <explode_bomb> # 否则炸弹爆炸
400fc9: 48 83 c4 18          add     $0x18,%rsp       # 将栈回收, 函数结束
400fcd: c3                  ret
```

对照上述汇编代码, 输入的数字相当于下标以对应表项, 进行进一步的跳转:

可以得到: case 0: 0x00400f7c, 此时 %eax = [0xcf]H = 207[D]
 同理可得: case 1: 0x00400fb9, 此时 %eax = [0x137]H = 311[D];
 同理可得: case 2: 0x00400f83, 此时 %eax = [0x2c3]H = 707[D];
 同理可得: case 3: 0x00400f8a, 此时 %eax = [0x100]H = 256[D];
 同理可得: case 4: 0x00400f91, 此时 %eax = [0x185]H = 389[D];
 同理可得: case 5: 0x00400f98, 此时 %eax = [0xce]H = 206[D];
 同理可得: case 6: 0x00400f9f, 此时 %eax = [0x2aa]H = 682[D];
 同理可得: case 7: 0x00400fa6, 此时 %eax = [0x147]H = 327[D];

继续分析汇编代码：

```

400fbe: 3b 44 24 0c      cmp     0xc(%rsp),%eax      # 比较第二个参数 与 目前 %eax 的值
400fc2: 74 05            je      400fc9 <phase_3+0x86> # 相等，则跳转至 400fc9 拆弹成功
400fc4: e8 71 04 00 00    call   40143a <explode_bomb> # 否则炸弹爆炸
400fc9: 48 83 c4 18      add     $0x18,%rsp         # 将栈回收，函数结束
400fcd: c3              ret

```

可知，答案是含两个正整数的七组输入。

| 0 207 | 1 311 | 2 707 | 3 256 | 4 389 | 5 206 | 6 682 | 7 327 |

随意选取一个答案，我选择 7 327，使用 vim 编辑器写入 ans.txt，运行，答案正确。

```

(gdb) b 95
Breakpoint 1 at 0x400e8b: file bomb.c, line 95.
(gdb) run ans.txt
Starting program: /home/deralive/Binary Bomb/bomb/bomb ans.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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!

```

4、Phase 4:

我们优先研究 Func4 在干什么，首先根据各种指令的知识，与 Phase 3 的分析过程相似，先列出基本的逻辑，标注在汇编代码右侧。

```

00000000400fce <func4>:
400fce: 48 83 ec 08      sub     $0x8,%rsp
400fd2: 89 d0            mov     %edx,%eax          # %eax = %edx, 此时应为0xe
400fd4: 29 f0            sub     %esi,%eax          # %eax = %eax - %esi, 即 0xe - $0x4025cf
400fd6: 89 c1            mov     %eax,%ecx          # %ecx = %eax
400fd8: c1 e9 1f         shr     $0x1f,%ecx         # 逻辑右移, %ecx >> 0x1f;
400fdb: 01 c8            add     %ecx,%eax          # %eax = %eax + %ecx;
400fdd: d1 f8            sar     %eax               # 算术右移, 不带操作数, 理解为算术右移 1 位
400fdf: 8d 0c 30         lea     (%rax,%rsi,1),%ecx  # 让 %ecx 存储 R[%rsi] + R[%rax] 的数据
400fe2: 39 f9            cmp     %edi,%ecx          # 比较 %ecx 和 %edi 的大小
400fe4: 7e 0c            jle     400ff2 <func4+0x24> # 有符号小于等于则跳转, 即 %edi <= %ecx 则跳转至 400ff2
400fe6: 8d 51 ff         lea     -0x1(%rcx),%edx
400fe9: e8 e0 ff ff ff   call   400fce <func4>      # 函数内部有自我调用, 是递归函数原型
400fee: 01 c0            add     %eax,%eax          # %eax = %eax * 2
400ff0: eb 15            jmp     401007 <func4+0x39> # 无条件跳转至 401007, 函数结束
400ff2: b8 00 00 00 00   mov     $0x0,%eax          # 重置命令: %eax = 0
400ff7: 39 f9            cmp     %edi,%ecx          # 比较 %edi 和 %ecx 的大小
400ff9: 7d 0c            jge     401007 <func4+0x39> # 若 %edi >= %ecx 则 跳转至 401007
400ffb: 8d 71 01         lea     0x1(%rcx),%esi     # %esi = (%rcx + 0x1)
400ffe: e8 cb ff ff ff   call   400fce <func4>      # 重新调用函数 func4, 形成递归
401003: 8d 44 00 01      lea     0x1(%rax,%rax,1),%eax # %eax = 2 * %rax + 0x1;
401007: 48 83 c4 08      add     $0x8,%rsp         # 回收栈, 函数结束
40100b: c3              ret

```


显而易见，这个函数是一个递归函数（因为在函数内部调用了自身）

令%rax 为返回值，那么函数还需要三个参数，分别为：

%edi,%esi,%edx

%ecx 作为中间变量 Temp 存在

根据上述分析，可将 Func4 的 C 语言伪代码如下所示：

```
int func4(int First, int Second, int Third) {  
    // First in %edi, Second in %esi, Third in %rdx  
  
    int result = Third;  
    result = result - Second;  
    int temp = result;  
    temp = temp >> 31;  
    result = result + temp;  
    result = result >> 1;  
    temp = result + Second;  
  
    if (First - temp < 0) {  
        Third = temp - 1;  
        func4(First, Second, Third);  
        result = result + result;  
        return result;  
    } else {  
        result = 0;  
        if(First - temp > 0) {  
            return result;  
        }  
        else {  
            Second = temp + 1;  
            func4(First, Second, Third);  
            result = result + result + 1;  
            return result;  
        }  
    }  
    return result;  
}
```

Func4 的功能：

这是一个递归式的二分查找函数。

使用二分法找出元素 (%rdi) 的值，如果目标在左半部分，则返回奇数，在右半部分，则返回偶数。找到元素后，返回值设置为 0.

如果查找过程中，目标值一直在左半部分，则返回 0.

下面，我们将目光转至 Phase_4 函数：

Phase_4 函数的汇编代码如下所示，同样地，可以使用(gdb) disas phase_4 查看，为了方便编辑注释，我使用 VScode 展开。

```

0000000040100c <phase_4>:
40100c: 48 83 ec 18      sub    $0x18,%rsp          # 栈指针下移 24 位, 为变量提供空间
401010: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx      # %rcx = %rsp + 0xc, 第二个参数
401015: 48 8d 54 24 08    lea    0x8(%rsp),%rdx      # %rdx = %rsp + 0x8, 第一个参数
40101a: be cf 25 40 00    mov    $0x4025cf,%esi      # %esi = $0x4025cf
40101f: b8 00 00 00 00    mov    $0x0,%eax          # %eax = 0
401024: e8 c7 fb ff ff    call   400bf0 <__isoc99_sscanf@plt> # 标准调用, 返回读取的个数
401029: 83 f8 02         cmp    $0x2,%eax          # 若读取个数不为2
40102c: 75 07           jne    401035 <phase_4+0x29>    # 炸弹爆炸
40102e: 83 7c 24 08 0e    cmpl   $0xe,0x8(%rsp)      # 若为2, 再比较第一个参数 和 $0xe
401033: 76 05           jbe    40103a <phase_4+0x2e>    # 如果: 第一个参数 <= 14, 则跳转到 40103a
401035: e8 00 04 00 00    call   40143a <explode_bomb>    # 否则, 炸弹爆炸
40103a: ba 0e 00 00 00    mov    $0xe,%edx          # %edx = 0xe
40103f: be 00 00 00 00    mov    $0x0,%esi          # %esi = 0
401044: 8b 7c 24 08      mov    0x8(%rsp),%edi      # %edi = 第一个参数
401048: e8 81 ff ff ff    call   400fce <func4>        # 调用func4
40104d: 85 c0           test   %eax,%eax          # %进行与运算, 判断是否相等
40104f: 75 07           jne    401058 <phase_4+0x4c>    # 不相等, 则跳转到 401058
401051: 83 7c 24 0c 00    cmpl   $0x0,0xc(%rsp)      # 比较第二个参数和0的大小
401056: 74 05           je     40105d <phase_4+0x51>    # 如果等于0, 则跳转至 40105d
401058: e8 dd 03 00 00    call   40143a <explode_bomb>    # 炸弹爆炸
40105d: 48 83 c4 18      add    $0x18,%rsp          # 回收栈指针, 函数结束
401061: c3             ret

```

其 C 语言伪代码可以写作:

```

void phase_4 (int rdx, int rcx, char* input) {
    int ret = 0;
    ret = sscanf(input, (char*)0x4025cf, &rcx, &rdx);
    if (ret != 2 || rcx > 14) {
        explode_bomb();
    } else {
        int ret = func4(rdx, 0, 14);
        if (ret != 0 || rcx != 0) {
            explode_bomb();
        }
    }
}

```

通过分析, 第二个传入的参数一定为 0, 根据 Func4 函数, 现在我们需要找到第一个参数的值 ($\%rdi \leq 14$), Func4 是一个使用了递归形式的二分查找 (Binary Search)。

其中, $\text{Third} = (\text{Second} - \text{First}) / 2 + \text{Second}$;

根据汇编代码中的信息: $\text{First} = 0$, $\text{Second} = 14$ (初始值)

故 c 的值为 $(\text{First} + \text{Second}) = 7$

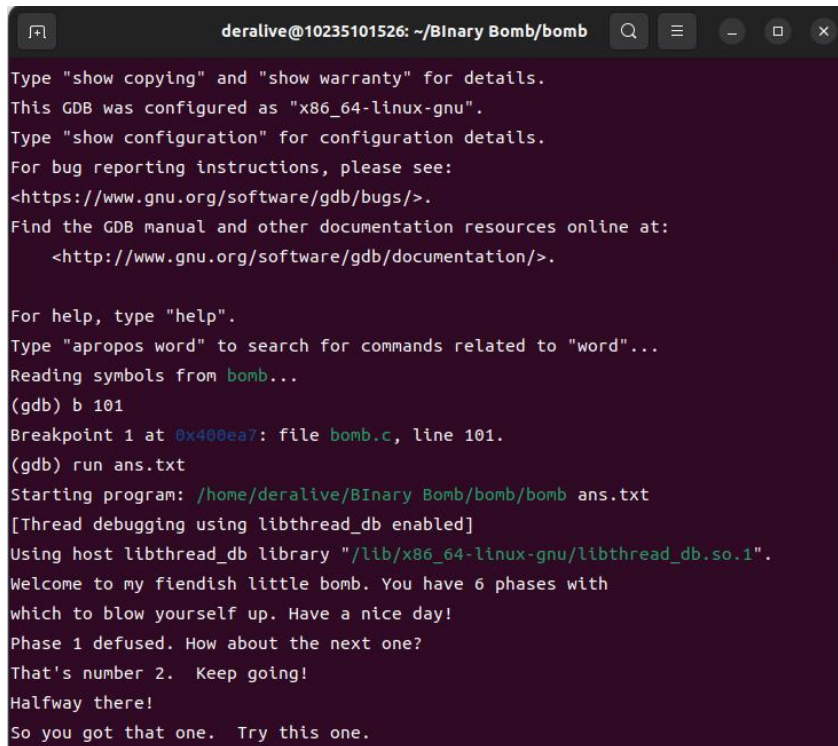
根据二分查找的算法可知, 存在 $x \in \{0, 1, 3, 7 \mid x \leq 14\}$

故本关卡的答案为:

00|10|30|70

随意选择一个, 输入 ans.txt 中:

顺便在 Phase 5 的入口处（第 101 行打上断点），运行 `run ans.txt`



```

deralive@10235101526: ~/Binary Bomb/bomb
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) b 101
Breakpoint 1 at 0x400ea7: file bomb.c, line 101.
(gdb) run ans.txt
Starting program: /home/deralive/Binary Bomb/bomb/bomb ans.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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.

```

结果正确。

5、Phase 5:

Phase 5 中出现了一个新的命令：`movzbl`，我们首先对这个命令进行辨析与了解。

`Mov` 传送指令共有五个子指令：

`movl` 传送双字；`movb` 传送一个字节；`movw` 传送两个字节；

`movsbl`，`movzbl` 指令负责拷贝一个字节，并设置目的操作数其余的位。

区别在于：

`movsbl` 源操作数是单字节，将 24 位设置位源字节的最高位，然后拷贝到双字目的中

`movzbl` 源操作数单字节，前面加 24 个 0 扩展到 32 位。然后拷贝到 32 位中。

假设：`%dh=8D,%eax=98765432`，那么有：

`movb %dh %al` `%eax=9876548D`

`movsbl %dh %eax` `%eax=FFFFFF8D`

`movzbl %dh %eax` `%eax=0000008D`

这里出现的 `Xor` 运算，即 `Xor %eax, %eax` 命令，实际是进行 `%eax` 清零的操作（因为是位运算，效率很高，比 `mov $0x0, %eax` 这一指令的效率要高）

Part 1: 我们先分析第一段代码:

```

401062: 53          push    %rbx          # 将 %rbx 的值压入栈中
401063: 48 83 ec 20  sub    $0x20,%rsp    # 分配 32 字节的空间
401067: 48 89 fb     mov    %rdi,%rbx     # %rbx = %rdi
40106a: 64 48 8b 04 25 28 00 mov    %fs:0x28,%rax  # %rax = %fs:0x28
401071: 00 00
401073: 48 89 44 24 18 mov    %rax,0x18(%rsp) # (%rsp + 0x18) = %rax
401078: 31 c0       xor    %eax,%eax     # 异或操作 (%rax的低四位字节), 清零
40107a: e8 9c 02 00 00 call   40131b <string_length> # 调用函数, 查看字符串的长度
40107f: 83 f8 06    cmp    $0x6,%eax     # 将字符串长度与 6 做比较
401082: 74 4e       je     4010d2 <phase_5+0x70> # 若字符串长度为 6
401084: e8 b1 03 00 00 call   40143a <explode_bomb>  # 则炸弹爆炸
401089: eb 47       jmp    4010d2 <phase_5+0x70> # 否则跳转至 4010d2
40108b: 0f b6 0c 03 movzbl (%rbx,%rax,1),%ecx # %ecx = %rax + %rbx      跳转 <--
40108f: 88 0c 24    mov    %cl,(%rsp)    # *(%rsp) = %cl
401092: 48 8b 14 24 mov    (%rsp),%rdx   # %rdx = *(rsp)
401096: 83 e2 0f    and    $0xf,%edx     # 将 %edx 与 0xf 相与, 存入 %edx
401099: 0f b6 92 b0 24 40 00 movzbl 0x4024b0(%rdx),%edx #
4010a0: 88 54 04 10 mov    %dl,0x10(%rsp,%rax,1) # (%rax + %rsp + 0x16) = %dl
4010a4: 48 83 c0 01 add    $0x1,%rax     # %rax++
4010a8: 48 83 f8 06 cmp    $0x6,%rax     # %rax == 6 ?
4010ac: 75 dd       jne    40108b <phase_5+0x29> # 若 %rax != 6, 跳转到 40108b 跳转 -->

```

显然, 我们输入的字符串的长度要为 6, 否则炸弹会直接爆炸。%rax 在此应该作为一个循环因子, 循环 6 次。

我们知道, %rax 和 %eax 的区别是: %rax 是一个 64 位寄存器, 而 %eax 是 %rax 的低 32 位部分, 用于 32 位操作。

我们输入的字符串的起始地址保存在 %rdi 中, 然后通过 mov 指令传输到 %rbx 中, 根据循环指令, 寄存器 %ecx 的值不断更新, 原因是 %rax 作为地址的偏移量, 在循环体中不断自增, 导致 %ecx 的值也不断改变。

注意到出现了一个 movzbl 指令, 有 0x4024b0 的字符串存在, 我们先读取:

```

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

```

之后, 我们查看后续的汇编代码:

```

4010ae: c6 44 24 16 00 movb    $0x0,0x16(%rsp) # 循环结束后, (%rap + 22) = 0 (一个字节)
4010b3: be 5e 24 40 00 mov     $0x40245e,%esi # %esi = $0x40245e
4010b8: 48 8d 7c 24 10 lea     0x10(%rsp),%rdi # rdi = %rsp + 16
4010bd: e8 76 02 00 00 call    401338 <strings_not_equal> # 调用函数查看字符串是否相等
4010c2: 85 c0       test    %eax,%eax     #
4010c4: 74 13       je     4010d9 <phase_5+0x77> # 若相等, 则跳转至 4010d9      跳转 ---->
4010c6: e8 6f 03 00 00 call    40143a <explode_bomb> # 否则爆炸
4010cb: 0f 1f 44 00 00 nopl    0x0(%rax,%rax,1) # %rax = 2 * %rax
4010d0: eb 07       jmp     4010d9 <phase_5+0x77> # 无条件跳转至 4010d9      跳转 -->
4010d2: b8 00 00 00 00 mov     $0x0,%eax     # 将 %eax = 0 置零
4010d7: eb b2       jmp     40108b <phase_5+0x29> # 跳转至 40108b            跳转 -->
4010d9: 48 8b 44 24 18 mov     0x18(%rsp),%rax # %rax = %rsp + 24      跳转<--
4010de: 64 48 33 04 25 28 00 xor     %fs:0x28,%rax  # %rax ^ %fs:0x28
4010e5: 00 00
4010e7: 74 05       je     4010ee <phase_5+0x8c> # 如果相等, 函数结束
4010e9: e8 42 fa ff ff call    400b30 <__stack_chk_fail@plt> # 如果不相等, 炸弹爆炸      跳转 <----
4010ee: 48 83 c4 20 add     $0x20,%rsp    # 回收栈指针
4010f2: 5b        pop     %rbx         # Pop 出 %rbx
4010f3: c3        ret                # 函数结束

```

在这一张含有汇编代码的图片里,我们可以发现,在 Phase_5 函数调用 <strings_not_equal> 函数之前, mov 指令将 0x40245e 移入了 %esi, 然后 %rdi = %rsp + 16, 猜测 0x40245e 中保存的正是正确答案的字符串, 我们使用 x/s 命令查看该字符串存储的信息:

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

总体而言: 这段代码揭示了:

- * 在地址 0x40108b 处, 函数进入一个循环, 循环 6 次。
- * 在每次迭代中, 它将地址 (%rbx + %rax) 处的字节移动到 %ecx 中。
- * 然后将 %ecx 的值存储在栈指针指向的地址处。
- * 然后函数检索栈指针指向的地址处的值, 并将其存储在 %rdx 中。
- * %rdx 的低 4 位被提取出来并存储在 %edx 中。
- * 然后将地址 0x4024b0 + %rdx 处的字节移动到地址 (%rsp + %rax + 0x10) 处。
- * %rax 的值增加 1。
- * 然后函数将 %rax 的值与 6 进行比较。
- * 如果它们不相等, 函数跳回到地址 0x40108b 处。
- * 循环结束后, 函数将值 0x0 移动到地址 (%rsp + 0x16) 处的字节。
- * 然后将值 0x40245e 移动到 %esi 中, 并将地址 (%rsp + 0x10) 移动到 %rdi 中。
- * 函数调用带有 %esi 和 %rdi 作为参数的函数 strings_not_equal。
- * 然后测试 strings_not_equal 的返回值。
- * 如果为 0, 则函数跳转到地址 0x4010d9。
- * 否则, 它调用函数 explode_bomb, 炸弹爆炸。

我们输入的六个字符, 函数会将每一个字符的 ASCII 码的低四位作为索引值。并查找 maduiersnfotvbyl 的字符, 最后返回的字符应该是 “flyers”

其中, f 在第 9 位, l 在第 15 位, y 在第 14 位, e 在第 5 位, r 在第 6 位, s 在第 7 位
(模拟字符数组, 以 0 作为索引)

所以我们输入的六个字符, 它们 ASCII 码低四位分别是:
1001, 1111, 1110, 0101, 0110, 0111.

根据汇编代码, 我们可以推断:

Phase 5 的 伪代码 (C Language) 形式如下所示:

```
void phase_5 (char* s) {
    const char con[16] = "maduiersnfotvbyl";
    char str[6];
    if (string_length(s) != 6) {
        explode_bomb();
    }

    for (int i = 0; i < 6; i++) {
        char temp = s[i] & 0xf;
        str[i] = con[temp];
    }

    if(string_no_equal(str, "flyers") != 0) {
        explode_bomb();
    }
}
```

下面，我们展示二进制形式的 ASCII 码表：

字 符	ASCII码			字 符	ASCII码		
	十进制K	二进制B	十六进制H		十进制K	二进制B	十六进制H
NUL (空)	0	0	0	M	77	1001101	4D
换行	10	1010	A	N	78	1001110	4E
空格	32	100000	20	O	79	1001111	4F
! (感叹号)	33	100001	21	P	80	1010000	50
"	34	100010	22	Q	81	1010001	51
#	35	100011	23	R	82	1010010	52
\$	36	100100	24	S	83	1010011	53
%	37	100101	25	T	84	1010100	54
&	38	100110	26	U	85	1010101	55
' (引号)	39	100111	27	V	86	1010110	56
(40	101000	28	W	87	1010111	57
)	41	101001	29	X	88	1011000	58
*	42	101010	2A	Y	89	1011001	59
+	43	101011	2B	Z	90	1011010	5A
,	44	101100	2C	[91	1011011	5B
-(减号)	45	101101	2D	\	92	1011100	5C
.	46	101110	2E]	93	1011101	5D
/ (除号)	47	101111	2F	^	94	1011110	5E
0	48	110000	30	-	95	1011111	5F
1	49	110001	31	a	97	1100001	61
2	50	110010	32	b	98	1100010	62
3	51	110011	33	c	99	1100011	63
4	52	110100	34	d	100	1100100	64
5	53	110101	35	e	101	1100101	65
6	54	110110	36	f	102	1100110	66
7	55	110111	37	g	103	1100111	67
8	56	111000	38	h	104	1101000	68
9	57	111001	39	i	105	1101001	69
:	58	111010	3A	j	106	1101010	6A
:	59	111011	3B	k	107	1101011	6B
<	60	111100	3C	l	108	1101100	6C
=	61	111101	3D	m	109	1101101	6D
>	62	111110	3E	n	110	1101110	6E
?	63	111111	3F	o	111	1101111	6F
@	64	1000000	40	p	112	1110000	70
A	65	1000001	41	q	113	1110001	71
B	66	1000010	42	r	114	1110010	72
C	67	1000011	43	s	115	1110011	73
D	68	1000100	44	t	116	1110100	74
E	69	1000101	45	u	117	1110101	75
F	70	1000110	46	v	118	1110110	76
G	71	1000111	47	w	119	1110111	77
H	72	1001000	48	x	120	1111000	78
I	73	1001001	49	y	121	1111001	79
J	74	1001010	4A	z	122	1111010	7A

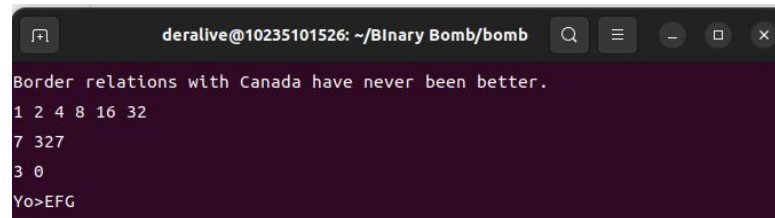
以 1001 作为低 4 位的字符有:)9IYiy
 以 1111 作为低 4 位的字符有: /?O-o
 以 1110 作为低 4 位的字符有: .>N^n
 以 0101 作为低 4 位的字符有: %5EUeu
 以 0110 作为低 4 位的字符有: &6FVfv
 以 0111 作为低 4 位的字符有: '7GWgw

任取上述字符进行组合。

(在表内所能查到的字符中) 该 Phase 5 答案共有:

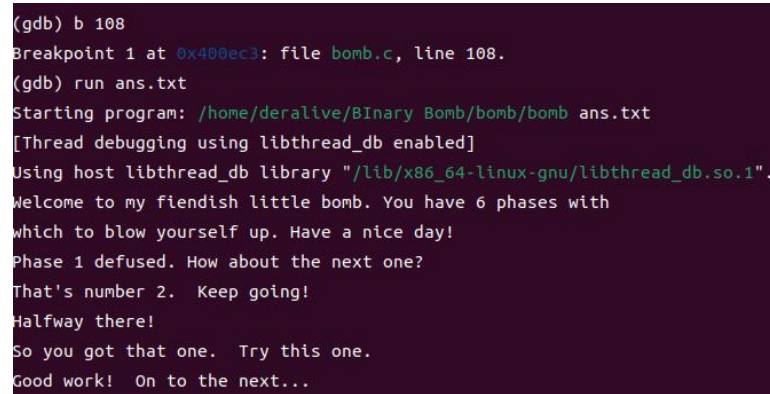
$$6 \times 5 \times 5 \times 6 \times 6 \times 6 = 32400 \text{ (种)}$$

我随机选取了 Yo^EFG 作为答案, 使用 vim 将答案写入 ans.txt 中。



```
deralive@10235101526: ~/Binary Bomb/bomb
Border relations with Canada have never been better.
1 2 4 8 16 32
7 327
3 0
Yo>EFG
```

将 phase 6 设置断点, run ans.txt, 答案正确。



```
(gdb) b 108
Breakpoint 1 at 0x400ec3: file bomb.c, line 108.
(gdb) run ans.txt
Starting program: /home/deralive/Binary Bomb/bomb/bomb ans.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
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...
```

6、Phase 6:

作为最后一个阶段, 它的工程量非常大, 我拿到代码的第一刻是无法将它们完整地捋出来的。但是通过观察, 我发现了几个循环体 (有许多的跳转指令), 于是, 我猜想是否能把这些代码分割出来。

显然, 函数的开始压入了许多个参数至栈中, 函数退出时又将这些参数弹出栈, 这一部分太过于冗余, 在前面的函数中已经有了多次分析, 所以我们省略这一部分的内容。

第一部分：喜闻乐见的读取六个整数，和之前的 Phase 一样的套路，%eax 作为读取的返回值，如果没有成功读取六个整数，调用炸弹爆炸的函数。

```
000000004010f4 <phase_6>:
4010f4: 41 56          push    %r14
4010f6: 41 55          push    %r13
4010f8: 41 54          push    %r12
4010fa: 55            push    %rbp
4010fb: 53            push    %rbx
4010fc: 48 83 ec 50    sub     $0x50,%rsp
401100: 49 89 e5       mov     %rsp,%r13      # %r13 = %rsp (栈指针)
401103: 48 89 e6       mov     %rsp,%rsi      # %rsi = %ssp
401106: e8 51 03 00 00 call    40145c <read_six_numbers> # 调用函数，读取六个数字
40110b: 49 89 e6       mov     %rsp,%r14      # %r14 = %rsp
40110e: 41 bc 00 00 00 00 mov     $0x0,%r12d     # %r12d = 0
401114: 4c 89 ed       mov     %r13,%rbp     # %rbp = %r13
401117: 41 8b 45 00    mov     0x0(%r13),%eax # %eax = (%r13)
40111b: 83 e8 01      sub     $0x1,%eax     # %eax--
40111e: 83 f8 05      cmp     $0x5,%eax     # %eax <= 5 ? (无符号小于等于)
401121: 76 05         jbe     401128 <phase_6+0x34> # 若判断为真，则跳转至 401128
401123: e8 12 03 00 00 call    40143a <explode_bomb> # 否则炸弹爆炸
```

第二部分：看到前面的两个循环条件，我们可以推测这里是一个循环，循环终止的条件是 %r12d == 6

我们同时还注意到，这里除了 %r12d 在自增之外，在这个循环体内还存在另一个 %ebx 在自增，所以我们推测这里面有嵌套的双重循环，则推测有两重 for 循环。

```
401128: 41 83 c4 01    add     $0x1,%r12d     # %r12d++
40112c: 41 83 fc 06    cmp     $0x6,%r12d     # %r12d == 6 ?
401130: 74 21         je      401153 <phase_6+0x5f> # 若相等，则跳转至 401153 跳转 -->
401132: 44 89 e3       mov     %r12d,%ebx     # 不相等，则 %ebx = %r12d
401135: 48 63 c3       movslq  %ebx,%rax       #
401138: 8b 04 84       mov     (%rsp,%rax,4),%eax # %eax = (4*%rax + %rsp)
40113b: 39 45 00       cmp     %eax,0x0(%rbp)  # (%rbp) != %eax
40113e: 75 05         jne     401145 <phase_6+0x51> # 若不相等，则跳转至 401145

401140: e8 f5 02 00 00 call    40143a <explode_bomb> # 若相等，则炸弹爆炸

401145: 83 c3 01      add     $0x1,%ebx     # %ebx++
401148: 83 fb 05      cmp     $0x5,%ebx     # 5 - %ebx <= 0 ? (有符号小于等于)
40114b: 7e e8         jle     401135 <phase_6+0x41> # 若小于等于，则跳转至 401135
40114d: 49 83 c5 04    add     $0x4,%r13     # 若大于，则 %r13 = %r13 + 4
401151: eb c1         jmp     401114 <phase_6+0x20> # 无条件跳转至 401114

401153: 48 8d 74 24 18 lea     0x18(%rsp),%rsi  # %rsi = (%rsp + 24) 跳转 <--
401158: 4c 89 f0       mov     %r14,%rax     # %rax = %r14
40115b: b9 07 00 00 00 mov     $0x7,%ecx     # %ecx = 7
401160: 89 ca         mov     %ecx,%edx     # %edx = %ecx 跳转 <---
401162: 2b 10         sub     (%rax),%edx    # %edx = %edx - (%rax)
401164: 89 10         mov     %edx,(%rax)    # (%rax) = %edx
401166: 48 83 c0 04    add     $0x4,%rax     # %rax = %rax + 4
40116a: 48 39 f0       cmp     %rsi,%rax     # %rax != %rsi ?
40116d: 75 f1         jne     401160 <phase_6+0x6c> # 若不相等，跳转至 401160 跳转 --->
40116f: be 00 00 00 00 mov     $0x0,%esi     # %esi = 0
```

在这两个部分当中，我们注意到 %ecx 被赋予了立即数 7，再使用 %edx = %ecx = 7 来做减法，根据汇编代码，我们可以将上述的代码转化成伪代码（注意，只是片段，非函数）。

```

// char* read;
int array[6];
read_six_numbers(read, array);

for (int i = 0; i != 6; i++) {
    int number = array[i];
    number--;

    if ((unsigned) num > 6) {
        explode_bomb();
    }

    for (int j = i+1; j <= 5; j++) {
        if (array[i] == array[j]) {
            explode_bomb();
        }
    }
}

// 将输入的 6 个数字 x 进行 7 - x 的映射
for (int i = 0; i < 6; i++) {
    array[i] = (7 - array[i]);
}

```

第三部分：

接下来，我们继续查看汇编代码：

注意到汇编代码中有一个很特别的值 0x6032d0，先通过 gdb 查看：

```

108      phase_6(input);
(gdb) x/32 0x6032d0
0x6032d0 <node1>:      332      1      6304480 0
0x6032e0 <node2>:      168      2      6304496 0
0x6032f0 <node3>:      924      3      6304512 0
0x603300 <node4>:      691      4      6304528 0
0x603310 <node5>:      477      5      6304544 0
0x603320 <node6>:      443      6      0        0
0x603330:              0        0        0        0
0x603340 <host_table>: 4204073 0      4204099 0

```

这里面的名词为 Node，是节点，说明这里的循环使用了链表。

根据链表的定义：

```

Struct ListNode {
    int val;
    ListNode* next;
};

```

我们可以推测，Node 1 中的数据域是 332，指向下一个节点的指针是 6304480。

根据推测，我们试着画出一个链表：

332 168 924 691 477 443
(0x6032d0) -> (0x6032e0) -> (0x6032f0) -> (0x603300) -> (0x603310) -> (0x603320)

接下来，我们继续查看汇编代码，看看后续进行了怎样的判断，炸弹才会爆炸，这里有六个节点，我们输入的数是 6 个 %d (Decimal)，所以猜测估计是要找到 1 2 3 4 5 6 的顺序

后面的代码有些复杂，所以我使用排序的方法猜测了一下，先是降序：

924 (3) -> 691 (4) -> 477 (5) -> 443 (6) -> 332 (1) -> 168 (2)

后是升序，即倒置 2 -> 1 -> 6 -> 5 -> 4 -> 3

发现答案都错了，但是面对汇编代码，并没有进行大量的运算，而是使用了多个 cmp 命令，我试图调转链表的方向为数据重新排链表下标。

降序： 4, 3, 2, 1, 6, 5

升序： 6, 5, 1, 2, 3, 4

在四种情况都尝试后，得到了结果 4 3 2 1 6 5 是正确答案：

使用 vim 将答案输入 ans.txt 中，获得了正确的结果。

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) run ans.txt
Starting program: /home/deralive/Binary Bomb/bomb/bomb ans.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Congratulations! You've defused the bomb!
[Inferior 1 (process 3471) exited normally]
(gdb)
```

但是仍未结束，代码的分析尚未完成。

401174: eb 21	jmp	401197 <phase_6+0xa3>	# 无条件跳转至 401197 跳转---->
401176: 48 8b 52 08	mov	0x8(%rdx),%rdx	
40117a: 83 c0 01	add	\$0x1,%eax	
40117d: 39 c8	cmp	%ecx,%eax	
40117f: 75 f5	jne	401176 <phase_6+0x82>	
401181: eb 05	jmp	401188 <phase_6+0x94>	
401183: ba d0 32 60 00	mov	\$0x6032d0,%edx	# %edx = \$0x6032d0
401188: 48 89 54 74 20	mov	%rdx,0x20(%rsp,%rsi,2)	
40118d: 48 83 c6 04	add	\$0x4,%rsi	
401191: 48 83 fe 18	cmp	\$0x18,%rsi	
401195: 74 14	je	4011ab <phase_6+0xb7>	
401197: 8b 0c 34	mov	(%rsp,%rsi,1),%ecx	# %ecx = (%rsi + %rsp) 跳转<---
40119a: 83 f9 01	cmp	\$0x1,%ecx	# %ecx - 1 <= 0 ? (有符号小于等于)
40119d: 7e e4	jle	401183 <phase_6+0x8f>	# 若条件成立, 则跳转至 401183
40119f: b8 01 00 00 00	mov	\$0x1,%eax	
4011a4: ba d0 32 60 00	mov	\$0x6032d0,%edx	
4011a9: eb cb	jmp	401176 <phase_6+0x82>	
4011ab: 48 8b 5c 24 20	mov	0x20(%rsp),%rbx	
4011b0: 48 8d 44 24 28	lea	0x28(%rsp),%rax	
4011b5: 48 8d 74 24 50	lea	0x50(%rsp),%rsi	
4011ba: 48 89 d9	mov	%rbx,%rcx	
4011bd: 48 8b 10	mov	(%rax),%rdx	
4011c0: 48 89 51 08	mov	%rdx,0x8(%rcx)	
4011c4: 48 83 c0 08	add	\$0x8,%rax	
4011c8: 48 39 f0	cmp	%rsi,%rax	
4011cb: 74 05	je	4011d2 <phase_6+0xde>	
4011cd: 48 89 d1	mov	%rdx,%rcx	
4011d0: eb eb	jmp	4011bd <phase_6+0xc9>	
4011d2: 48 c7 42 08 00 00 00	movq	\$0x0,0x8(%rdx)	
4011d9: 00			
4011da: bd 05 00 00 00	mov	\$0x5,%ebp	
4011df: 48 8b 43 08	mov	0x8(%rbx),%rax	
4011e3: 8b 00	mov	(%rax),%eax	
4011e5: 39 03	cmp	%eax,(%rbx)	
4011e7: 7d 05	jge	4011ee <phase_6+0xfa>	
4011e9: e8 4c 02 00 00	call	40143a <explode_bomb>	
4011ee: 48 8b 5b 08	mov	0x8(%rbx),%rbx	
4011f2: 83 ed 01	sub	\$0x1,%ebp	

我们回到汇编代码当中,

%edx 存放的是链表首结点地址, 根据汇编代码分析, 注释如下图所示:

4011ab: 48 8b 5c 24 20	mov	0x20(%rsp),%rbx	# %rbx = (%rsp + 32) 链表的节点地址
4011b0: 48 8d 44 24 28	lea	0x28(%rsp),%rax	# 将 %rax 指向下一个节点的地址
4011b5: 48 8d 74 24 50	lea	0x50(%rsp),%rsi	# %rsi 指向链表最后一个节点的地址 (+0x50)
4011ba: 48 89 d9	mov	%rbx,%rcx	
4011bd: 48 8b 10	mov	(%rax),%rdx	
4011c0: 48 89 51 08	mov	%rdx,0x8(%rcx)	# 栈中后一个节点的地址变成前一个节点的next指针
4011c4: 48 83 c0 08	add	\$0x8,%rax	# 指针移动到下一个节点
4011c8: 48 39 f0	cmp	%rsi,%rax	# 是否循环了六次?
4011cb: 74 05	je	4011d2 <phase_6+0xde>	
4011cd: 48 89 d1	mov	%rdx,%rcx	# 如果没有, 继续遍历
4011d0: eb eb	jmp	4011bd <phase_6+0xc9>	
4011d2: 48 c7 42 08 00 00 00	movq	\$0x0,0x8(%rdx)	# 最后一个节点的指针接地 (%rdx + 0x8) = nullptr
4011d9: 00			

注意到还有一个循环，这里是用于判断链表重新调整后，所有节点是否已经降序排列。

```
4011da: bd 05 00 00 00      mov     $0x5,%ebp
4011df: 48 8b 43 08         mov     0x8(%rbx),%rax      # 跳转 <--
4011e3: 8b 00              mov     (%rax),%eax
4011e5: 39 03             cmp     %eax, (%rbx)
4011e7: 7d 05            jge     4011ee <phase_6+0xfa>
4011e9: e8 4c 02 00 00     call    40143a <explode_bomb>
4011ee: 48 8b 5b 08         mov     0x8(%rbx),%rbx
4011f2: 83 ed 01          sub     $0x1,%ebp
4011f5: 75 e8            jne     4011df <phase_6+0xeb>  # 跳转 -->
```

这一串代码就很好地解释了刚刚的猜测，就是因为链表逆序了，所以才导致我最初的判断出现了差错。

参考 C 语言的伪代码为（非自己所写）：

```
// 生成 node_array
for (int i = 0; i < 6; i++) {
    int cur = array[i];
    ListNode* node = 0x6032d0;      // 链表头指针的地址
    if (cur > 1) {
        for (int j = 1; j < cur; j++) {
            node = node->next;
        }
    }
    node_array[i] = node;
}

for (int i = 0; i < 5; i++) {
    node_array[i]->next = node_array[i+1];
}

ListNode* ptr = node_array[0];
for (int i = 5; i > 0; i--) {
    if (ptr->val < ptr->next->val)
        explode_bomb();
    ptr = ptr->next;
}
```

拆弹完成。

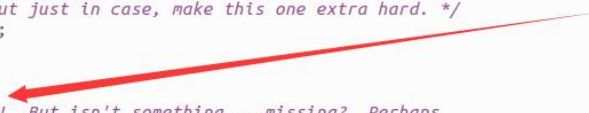
7、Secret Phase

在炸弹的六个阶段拆除完成后，我们通过 bomb.c 的函数，可以发现：

```

99  /* Round and 'round in memory we go, where we stop, the bomb blows! */
100 input = read_line();
101 phase_5(input);
102 phase_defused();
103 printf("Good work! On to the next...\n");
104
105 /* This phase will never be used, since no one will get past the
106    * earlier ones. But just in case, make this one extra hard. */
107 input = read_line();
108 phase_6(input);
109 phase_defused();
110
111 /* Wow, they got it! But isn't something... missing? Perhaps
112    * something they overlooked? Mua ha ha ha ha! */
113
114 return 0;

```



先寻找入口：在.S 汇编文件中使用 Ctrl + F 查看每次炸弹的一个阶段拆除后调用的 phase_defused() 函数，看看是否藏了什么东西：

```

000000004015c4 <phase_defused>:
4015c4: 48 83 ec 78      sub    $0x78,%rsp
4015c8: 64 48 8b 04 25 28 00 mov    %fs:0x28,%rax
4015cf: 00 00
4015d1: 48 89 44 24 68    mov    %rax,0x68(%rsp)
4015d6: 31 c0            xor    %eax,%eax
4015d8: 83 3d 81 21 20 00 06 cmpl   $0x6,0x202181(%rip)      # 603760 <num_input_strings>
4015df: 75 5e            jne    40163f <phase_defused+0x7b>
4015e1: 4c 8d 44 24 10    lea    0x10(%rsp),%r8
4015e6: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx
4015eb: 48 8d 54 24 08    lea    0x8(%rsp),%rdx
4015f0: be 19 26 40 00    mov    $0x402619,%esi
4015f5: bf 70 38 60 00    mov    $0x603870,%edi
4015fa: e8 f1 f5 ff ff    call   400bf0 <__isoc99_sscanf@plt>
4015ff: 83 f8 03          cmp    $0x3,%eax
401602: 75 31            jne    401635 <phase_defused+0x71>
401604: be 22 26 40 00    mov    $0x402622,%esi
401609: 48 8d 7c 24 10    lea    0x10(%rsp),%rdi
40160e: e8 25 fd ff ff    call   401338 <strings_not_equal>
401613: 85 c0            test   %eax,%eax
401615: 75 1e            jne    401635 <phase_defused+0x71>
401617: bf f8 24 40 00    mov    $0x4024f8,%edi
40161c: e8 ef f4 ff ff    call   400b10 <puts@plt>
401621: bf 20 25 40 00    mov    $0x402520,%edi
401626: e8 e5 f4 ff ff    call   400b10 <puts@plt>
40162b: b8 00 00 00 00    mov    $0x0,%eax
401630: e8 0d fc ff ff    call   401242 <secret_phase>
401635: bf 58 25 40 00    mov    $0x402558,%edi
40163a: e8 d1 f4 ff ff    call   400b10 <puts@plt>
40163f: 48 8b 44 24 68    mov    0x68(%rsp),%rax
401644: 64 48 33 04 25 28 00 xor    %fs:0x28,%rax
40164b: 00 00
40164d: 74 05            je     401654 <phase_defused+0x90>
40164f: e8 dc f4 ff ff    call   400b30 <__stack_chk_fail@plt>
401654: 48 83 c4 78      add    $0x78,%rsp
401658: c3              ret

```

在 401630 内存中，显然有一个 `strings_not_equal` 的判断，和前面的分析是一致的，在 `phase_defuse()` 执行时，如果再输入一个字符串与之匹配，就能进入 Secret Phase。

```

4015f0: be 19 26 40 00      mov     $0x402619,%esi
4015f5: bf 70 38 60 00      mov     $0x603870,%edi
4015fa: e8 f1 f5 ff ff      call    400bf0 <__isoc99_sscanf@plt>
4015ff: 83 f8 03            cmp     $0x3,%eax
401602: 75 31              jne     401635 <phase_defused+0x71>

```

这里和前面的方法一样，我们先试图进入 0x402619 和 0x603870。
查看字符串到底是什么。

```

(gdb) x/s 0x402619
0x402619:      "%d %d %s"

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

```

显然，我们看到了，我们的输入格式应该是两个正整数和一个字符。在互联网查了一下，这里用到了二叉树的知识，因为数据结构还没学到，刚好到了实验报告的提交时间，所以暂时先留着 Secret Phase，等之后学完了之后再回来把这一个大魔王给解决掉。

四、心得体会

在本次 CSAPP: Lab2 Bomb Lab 的实验中，我充分感受到了自己知识框架的不完整，面对汇编代码却无法严谨地书写我的推导步骤，有一部分只能连蒙带猜，最后草草结束，做这个实验报告的时间并不长，长的是慢慢在网上查找各种指令，如 `mov` 和 `lea` 的区别，`movq` 和 `movzbl` 的区别等等，让我感受到了计算机底层的美。

最重要的是，我还在这次实验中体悟到了计算机是如何通过各种变化的指令操作让计算更高效，如 `Xor` 命令代替 `mov $0x0` 等等，还从中学习到了链表、双重循环、二分查找等等的古典算法与数据结构的思想，甚至还将以前从不入眼的 `sscanf` 函数的原型找了出来，发现了许多自己知识体系的不完善和漏洞的欠缺，让我觉得道阻且长。

五、附录：

汇编代码及注释分析详情如下所示：

```

0000000000400ee0 <phase_1>:
400ee0: 48 83 ec 08          sub     $0x8,%rsp          # 栈指针向下 8 位,
400ee4: be 00 24 40 00        mov     $0x402400,%esi      # 将内存地址为 0x402400 的数据移入 esi 寄存器
400ee9: e8 4a 04 00 00        call    401338 <strings_not_equal> # 调用“字符串不相等”函数
400eee: 85 c0                test    %eax,%eax          # 进行自身按位与运算，猜测用于判断输入的字符串是否相等
400ef0: 74 05                je      400ef7 <phase_1+0x17> # 条件跳转指令，若上述条件符合，则跳转到 0x400ef7 处

```

```

400ef2:  e8 43 05 00 00      call 40143a <explode_bomb>      # 猜测字符串不相等时, 调用 explode_bomb 函数, 导致炸弹爆炸
400ef7:  48 83 c4 08         add $0x8,%rsp                  # 对应 sub 命令, 使得栈指针上移
400efb:  c3                 ret

000000000400efc <phase_2>:
400efc:  55                 push %rbp

400efd:  53                 push %rbx
400efe:  48 83 ec 28         sub $0x28,%rsp                # 栈指针向下 40 位, 为本函数变量提供空间
400f02:  48 89 e6           mov %rsp,%rsi                 # 将栈指针移入寄存器%rsi (第二个变量)
400f05:  e8 52 05 00 00      call 40145c <read_six_numbers>  # 读取六个数字

400f0a:  83 3c 24 01         cmpl $0x1,(%rsp)              # %rsp 是栈指针,与立即数 1 作比较
400f0e:  74 20              je 400f30 <phase_2+0x34>       # 比较结果若相等, 则跳至(400efc + 0x34) = 400f30

400f10:  e8 25 05 00 00      call 40143a <explode_bomb>      # 若不相等, 则炸弹爆炸, 相等则跳过该行调用
400f15:  eb 19              jmp 400f30 <phase_2+0x34>       # 无条件跳转至 (400efc + 0x34) = 400f30

400f17:  8b 43 fc           mov -0x4(%rbx),%eax           # 将 (%rbx - 4) 移动到 %eax 寄存器
400f1a:  01 c0              add %eax,%eax                 # 将 %eax * 2
400f1c:  39 03              cmp %eax,(%rbx)               # 比较 %eax 和 %rbx 指向的值
400f1e:  74 05              je 400f25 <phase_2+0x29>       # 若相等, 则跳转至 400f25
400f20:  e8 15 05 00 00      call 40143a <explode_bomb>      # 否则调用炸弹爆炸函数
400f25:  48 83 c3 04         add $0x4,%rbx                 # 不相等, 继续将 %rbx + 4, 即指向下一个元素
400f29:  48 39 eb           cmp %rbp,%rbx                 # %rbp - %rbx != 0,
400f2c:  75 e9              jne 400f17 <phase_2+0x1b>       # 则跳转到 400f17.继续执行以上过程

400f2e:  eb 0c              jmp 400f3c <phase_2+0x40>       # 若上述指令都执行完, 没出现问题, 就跳转至 400f3c, 函数结束
400f30:  48 8d 5c 24 04      lea 0x4(%rsp),%rbx            # 将 (栈指针指向的地址 + 4) 后的结果
# 移动进入 %rbx 暂存 (其实就是压入栈内的下一个元素)

400f35:  48 8d 6c 24 18      lea 0x18(%rsp),%rbp           # 将 %rsp 栈指针 + 24, 移动进入 %rbp 暂存 (栈的最后一个元素)
400f3a:  eb db              jmp 400f17 <phase_2+0x1b>       # 无条件跳转至 (400efc + 0x1b) = 400f17

400f3c:  48 83 c4 28         add $0x28,%rsp                # 回收栈指针和返回等操作
400f40:  5b                 pop %rbx
400f41:  5d                 pop %rbp
400f42:  c3                 ret

000000000400f43 <phase_3>:
400f43:  48 83 ec 18         sub $0x18,%rsp                # 栈指针下移 24, 为变量提供空间
400f47:  48 8d 4c 24 0c      lea 0xc(%rsp),%rcx            # %rcx = (%rsp + 0xc), 存放第二个数
400f4c:  48 8d 54 24 08      lea 0x8(%rsp),%rdx            # %rdx = (%rsp + 0x8), 存放第一个数

// 为什么是 +0x8 : 因为 +0x4 的位置存放的是输入字符串的首地址, 是为了后续调用 sscanf 做准备

400f51:  be cf 25 40 00      mov $0x4025cf,%esi            # %esi = $0x4025cf
400f56:  b8 00 00 00 00      mov $0x0,%eax                 # %eax = 0

400f5b:  e8 90 fc ff ff      call 400bf0 <__isoc99_sscanf@plt>
400f60:  83 f8 01           cmp $0x1,%eax                 # 函数调用后有返回值
400f63:  7f 05              jg 400f6a <phase_3+0x27>       # 如果输入个数大于 1, 则跳转到 400f60a
400f65:  e8 d0 04 00 00      call 40143a <explode_bomb>      # 否则炸弹就会引爆

400f6a:  83 7c 24 08 07      cmpl $0x7,0x8(%rsp)           # 比较第一个参数与 7 的大小关系
400f6f:  77 3c              ja 400fad <phase_3+0x6a>       # 如果参数大于 7, 则跳转至 400fad
400f71:  8b 44 24 08         mov 0x8(%rsp),%eax            # 不大于, 那么将参数放至 %eax 中

```

```

400f75: ff 24 c5 70 24 40 00    jmp     *0x402470(%rax,8)      # 跳转表格式, 跳转至 *(0x402470) = 0x400f7c

400f7c: b8 cf 00 00 00          mov     $0xcf,%eax             # %eax = $0xcf
400f81: eb 3b                   jmp     400f8e <phase_3+0x7b>  # 跳转至 400f8e
400f83: b8 c3 02 00 00          mov     $0x2c3,%eax            # %eax = $0x2c3
400f88: eb 34                   jmp     400f8e <phase_3+0x7b>  # 跳到后面判断第二个参数和 (%eax) 是否相同
400f8a: b8 00 01 00 00          mov     $0x100,%eax            # %eax = $0x100
400f8f: eb 2d                   jmp     400f8e <phase_3+0x7b>
400f91: b8 85 01 00 00          mov     $0x185,%eax            # %eax = $0x185
400f96: eb 26                   jmp     400f8e <phase_3+0x7b>
400f98: b8 ce 00 00 00          mov     $0xce,%eax             # %eax = $0xce
400f9d: eb 1f                   jmp     400f8e <phase_3+0x7b>
400f9f: b8 aa 02 00 00          mov     $0x2aa,%eax            # %eax = $0x2aa
400fa4: eb 18                   jmp     400f8e <phase_3+0x7b>
400fa6: b8 47 01 00 00          mov     $0x147,%eax            # %eax = $0x147
400fab: eb 11                   jmp     400f8e <phase_3+0x7b>

400fad: e8 88 04 00 00          call    40143a <explode_bomb>  # 炸弹爆炸

400fb2: b8 00 00 00 00          mov     $0x0,%eax              # %eax = 0
400fb7: eb 05                   jmp     400f8e <phase_3+0x7b>
400fb9: b8 37 01 00 00          mov     $0x137,%eax            # %eax = 0x137
// 这里可以补上一句,方便理解:
// jmp     400f8e <phase_3+0x7b>

400fbe: 3b 44 24 0c             cmp     0xc(%rsp),%eax          # 比较第二个参数 与 目前 %eax 的值
400fc2: 74 05                   je      400fc9 <phase_3+0x86>  # 相等, 则跳转至 400fc9 炸弹成功
400fc4: e8 71 04 00 00          call    40143a <explode_bomb>  # 否则炸弹爆炸
400fc9: 48 83 c4 18             add     $0x18,%rsp              # 将栈回收, 函数结束
400fcd: c3                       ret

0000000000400fce <func4>:
400fce: 48 83 ec 08             sub     $0x8,%rsp
400fd2: 89 d0                   mov     %edx,%eax              # %eax = %edx, 此时应为 0xc
400fd4: 29 f0                   sub     %esi,%eax              # %eax = %eax - %esi, 即 0xc - $0x4025cf
400fd6: 89 c1                   mov     %eax,%ecx              # %ecx = %eax
400fd8: c1 e9 1f               shr     $0x1f,%ecx             # 逻辑右移, %ecx >> 0x1f;
400fdb: 01 c8                   add     %ecx,%eax              # %eax = %eax + %ecx;
400fdd: d1 f8                   sar     %eax                   # 算术右移, 不带操作数, 理解为算术右移 1 位
400fdf: 8d 0c 30               lea     (%rax,%rsi,1),%ecx      # 让 %ecx 存储 R[%rsi] + R[%rax] 的数据
400fe2: 39 f9                   cmp     %edi,%ecx              # 比较 %ecx 和 %edi 的大小
400fe4: 7e 0c                   jle     400ff2 <func4+0x24>     # 有符号小于等于则跳转, 即 %edi <= %ecx 则跳转至 400ff2
400fe6: 8d 51 ff               lea     -0x1(%rcx),%edx
400fe9: e8 e0 ff ff ff         call    400fce <func4>         # 函数内部有自我调用, 是递归函数原型
400fee: 01 c0                   add     %eax,%eax              # %eax = %eax * 2
400ff0: eb 15                   jmp     401007 <func4+0x39>     # 无条件跳转至 401007, 函数结束
400ff2: b8 00 00 00 00          mov     $0x0,%eax              # 重置命令: %eax = 0
400ff7: 39 f9                   cmp     %edi,%ecx              # 比较 %edi 和 %ecx 的大小
400ff9: 7d 0c                   jge     401007 <func4+0x39>     # 若 %edi >= %ecx 则 跳转至 401007
400ffb: 8d 71 01               lea     0x1(%rcx),%esi         # %esi = (%rcx + 0x1)
400ffe: e8 cb ff ff ff         call    400fce <func4>         # 重新调用函数 func4, 形成递归
401003: 8d 44 00 01             lea     0x1(%rax,%rax,1),%eax  # %eax = 2 * %rax + 0x1;
401007: 48 83 c4 08             add     $0x8,%rsp              # 回收栈, 函数结束
40100b: c3                       ret

000000000040100c <phase_4>:
40100c: 48 83 ec 18             sub     $0x18,%rsp            # 栈指针下移 24 位, 为变量提供空间
401010: 48 8d 4c 24 0c          lea     0xc(%rsp),%rcx         # %rcx = %rsp + 0xc, 第二个参数

```



```

401015: 48 8d 54 24 08      lea     0x8(%rsp),%rdx      # %rdx = %rsp + 0x8, 第一个参数
40101a: be cf 25 40 00      mov     $0x4025cf,%esi     # %esi = $0x4025cf
40101f: b8 00 00 00 00      mov     $0x0,%eax          # %eax = 0
401024: e8 c7 fb ff ff      call    400bf0 <__isoc99_sscanf@plt> # 标准调用, 返回读取的个数
401029: 83 f8 02            cmp     $0x2,%eax          # 若读取个数不为 2
40102c: 75 07              jne     401035 <phase_4+0x29> # 炸弹爆炸
40102e: 83 7c 24 08 0e      cmpl    $0xc,0x8(%rsp)     # 若为 2, 再比较第一个参数 和 $0xc
401033: 76 05              jbe     40103a <phase_4+0x2e> # 如果: 第一个参数 <= 14, 则跳转到 40103a
401035: e8 00 04 00 00      call    40143a <explode_bomb> # 否则, 炸弹爆炸
40103a: ba 0e 00 00 00      mov     $0xc,%edx          # %edx = 0xc
40103f: be 00 00 00 00      mov     $0x0,%esi          # %esi = 0
401044: 8b 7c 24 08         mov     0x8(%rsp),%edi     # %edi = 第一个参数
401048: e8 81 ff ff ff      call    400fce <func4>     # 调用 func4
40104d: 85 c0              test    %eax,%eax          # %进行与运算, 判断是否相等
40104f: 75 07              jne     401058 <phase_4+0x4c> # 不相等, 则跳转到 401058
401051: 83 7c 24 0c 00      cmpl    $0x0,0xc(%rsp)     # 比较第二个参数和 0 的大小
401056: 74 05              je      40105d <phase_4+0x51> # 如果等于 0, 则跳转至 40105d
401058: e8 dd 03 00 00      call    40143a <explode_bomb> # 炸弹爆炸
40105d: 48 83 c4 18         add     $0x18,%rsp         # 回收栈指针, 函数结束
401061: c3                ret

000000000401062 <phase_5>:
401062: 53                push    %rbx              # 将 %rbx 的值压入栈中
401063: 48 83 ec 20        sub     $0x20,%rsp        # 分配 32 字节的空间
401067: 48 89 fb          mov     %rdi,%rbx         # %rbx = %rdi
40106a: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax      # %rax = %fs:0x28
401071: 00 00
401073: 48 89 44 24 18     mov     %rax,0x18(%rsp)    # (%rsp + 0x18) = %rax
401078: 31 c0             xor     %eax,%eax          # 异或操作 (%rax 的低四位字节), 清零
40107a: e8 9c 02 00 00     call    40131b <string_length> # 调用函数, 查看字符串的长度
40107f: 83 f8 06          cmp     $0x6,%eax          # 将字符串长度与 6 做比较
401082: 74 4e             je      4010d2 <phase_5+0x70> # 若字符串长度为 6
401084: e8 b1 03 00 00     call    40143a <explode_bomb> # 则炸弹爆炸
401089: eb 47            jmp     4010d2 <phase_5+0x70> # 否则跳转至 4010d2
40108b: 0f b6 0c 03        movzbl (%rbx,%rax,1),%ecx  # %ecx = [%rax] + %rbx      跳转 <--
40108f: 88 0c 24          mov     %cl,(%rsp)         # *(%rsp) = %cl
401092: 48 8b 14 24        mov     (%rsp),%rdx        # %rdx = *(rsp)
401096: 83 e2 0f          and     $0xf,%edx          # 将 %edx 与 0xf 相与, 存入 %edx
401099: 0f b6 92 b0 24 40 00 movzbl 0x4024b0(%rdx),%edx  # 将 %edx = %rdx + 0x4024b0
4010a0: 88 54 04 10        mov     %dl,0x10(%rsp,%rax,1) # (%rax + %rsp + 0x16) = %dl
4010a4: 48 83 c0 01        add     $0x1,%rax          # %rax++
4010a8: 48 83 f8 06        cmp     $0x6,%rax          # %rax == 6 ?
4010ac: 75 dd            jne     40108b <phase_5+0x29> # 若 %rax != 6, 跳转到 40108b 跳转 -->

4010ae: c6 44 24 16 00     movb    $0x0,0x16(%rsp)    # 循环结束后, (%rap + 22) = 0 (一个字节)
4010b3: be 5e 24 40 00     mov     $0x40245e,%esi     # %esi = $0x40245e
4010b8: 48 8d 7c 24 10     lea     0x10(%rsp),%rdi     # rdi = %rsp + 16
4010bd: e8 76 02 00 00     call    401338 <strings_not_equal> # 调用函数查看字符串是否相等
4010c2: 85 c0             test    %eax,%eax          #
4010c4: 74 13            je      4010d9 <phase_5+0x77> # 若相等, 则跳转至 4010d9      跳转 ---->
4010c6: e8 6f 03 00 00     call    40143a <explode_bomb> # 否则爆炸
4010cb: 0f 1f 44 00 00     nopl    0x0(%rax,%rax,1)    # %rax = 2 * %rax
4010d0: eb 07            jmp     4010d9 <phase_5+0x77> # 无条件跳转至 4010d9      跳转-->
4010d2: b8 00 00 00 00     mov     $0x0,%eax          # 将 %eax = 0 置零
4010d7: eb b2            jmp     40108b <phase_5+0x29> # 跳转至 40108b              跳转 -->
4010d9: 48 8b 44 24 18     mov     0x18(%rsp),%rax     # %rax = %rsp + 24      跳转<--

```

```

4010de: 64 48 33 04 25 28 00    xor    %fs:0x28,%rax          # %rax ^ %fs:0x28
4010e5: 00 00
4010e7: 74 05                    je     4010ee <phase_5+0x8c>   # 如果相等，函数结束
4010e9: e8 42 fa ff ff    call  400b30 <__stack_chk_fail@plt> # 如果不相等，炸弹爆炸      跳转 <----
4010ee: 48 83 c4 20        add    $0x20,%rsp            # 回收栈指针
4010f2: 5b                pop     %rbx                  # Pop 出 %rbx
4010f3: c3                ret                          # 函数结束

00000000004010f4 <phase_6>:
4010f4: 41 56                push   %r14
4010f6: 41 55                push   %r13
4010f8: 41 54                push   %r12
4010fa: 55                push   %rbp
4010fb: 53                push   %rbx
4010fc: 48 83 ec 50        sub    $0x50,%rsp
401100: 49 89 e5            mov     %rsp,%r13             # %r13 = %rsp (栈指针)
401103: 48 89 e6            mov     %rsp,%rsi             # %rsi = %ssp
401106: e8 51 03 00 00    call  40145c <read_six_numbers> # 调用函数，读取六个数字
40110b: 49 89 e6            mov     %rsp,%r14             # %r14 = %rsp
40110e: 41 bc 00 00 00 00    mov     $0x0,%r12d            # %r12d = 0
401114: 4c 89 ed            mov     %r13,%rbp             # %rbp = %r13
401117: 41 8b 45 00        mov     0x0(%r13),%eax         # %eax = (%r13)
40111b: 83 ec 01            sub     $0x1,%eax              # %eax--
40111e: 83 f8 05            cmp     $0x5,%eax              # %eax <= 5 ? (无符号小于等于)
401121: 76 05              jbe     401128 <phase_6+0x34>   # 若判断为真，则跳转至 401128
401123: e8 12 03 00 00    call  40143a <explode_bomb>    # 否则炸弹爆炸
401128: 41 83 c4 01        add     $0x1,%r12d             # %r12d++
40112c: 41 83 fc 06        cmp     $0x6,%r12d             # %r12d == 6 ?
401130: 74 21              je      401153 <phase_6+0x5f>   # 若相等，则跳转至 401153      跳转 -->
401132: 44 89 e3            mov     %r12d,%ebx             # 不相等，则 %ebx = %r12d
401135: 48 63 c3            movslq  %ebx,%rax              #
401138: 8b 04 84            mov     (%rsp,%rax,4),%eax      # %eax = (4*%rax + %rsp)
40113b: 39 45 00            cmp     %eax,0x0(%rbp)          # (%rbp) != %eax
40113e: 75 05              jne     401145 <phase_6+0x51>   # 若不相等，则跳转至 401145

401140: e8 f5 02 00 00    call  40143a <explode_bomb>    # 若相等，则炸弹爆炸

401145: 83 c3 01            add     $0x1,%ebx              # %ebx++
401148: 83 fb 05            cmp     $0x5,%ebx              # 5 - %ebx <= 0 ? (有符号小于等于)
40114b: 7e e8              jle     401135 <phase_6+0x41>   # 若小于等于，则跳转至 401135
40114d: 49 83 c5 04        add     $0x4,%r13              # 若大于，则 %r13 = %r13 + 4
401151: eb c1              jmp     401114 <phase_6+0x20>   # 无条件跳转至 401114

401153: 48 8d 74 24 18    lea     0x18(%rsp),%rsi         # %rsi = (%rsp + 24)      跳转 <--
401158: 4c 89 f0            mov     %r14,%rax              # %rax = %r14
40115b: b9 07 00 00 00    mov     $0x7,%ecx              # %ecx = 7
401160: 89 ca            mov     %ecx,%edx              # %edx = %ecx      跳转 <---
401162: 2b 10            sub     (%rax),%edx             # %edx = %edx - (%rax)
401164: 89 10            mov     %edx,(%rax)             # (%rax) = %edx
401166: 48 83 c0 04        add     $0x4,%rax              # %rax = %rax + 4
40116a: 48 39 f0            cmp     %rsi,%rax              # %rax != %rsi ?
40116d: 75 f1              jne     401160 <phase_6+0x6c>   # 若不相等，跳转至 401160      跳转 ---->
40116f: be 00 00 00 00    mov     $0x0,%esi              # %esi = 0
401174: eb 21              jmp     401197 <phase_6+0xa3>   # 无条件跳转至 401197      跳转 ---->
401176: 48 8b 52 08        mov     0x8(%rdx),%rdx
40117a: 83 c0 01            add     $0x1,%eax
40117d: 39 c8            cmp     %ecx,%eax
40117f: 75 f5              jne     401176 <phase_6+0x82>

```

```

401181:  cb 05                jmp     401188 <phase_6+0x94>
401183:  ba d0 32 60 00      mov     $0x6032d0,%edx          # %edx = $0x6032d0
401188:  48 89 54 74 20      mov     %rdx,0x20(%rsp,%rsi,2)
40118d:  48 83 c6 04         add     $0x4,%rsi
401191:  48 83 fe 18         cmp     $0x18,%rsi
401195:  74 14               je      4011ab <phase_6+0xb7>
401197:  8b 0c 34           mov     (%rsp,%rsi,1),%ecx      # %ecx = (%rsi + %rsp) 跳转<----
40119a:  83 f9 01           cmp     $0x1,%ecx              # %ecx - 1 <= 0 ? (有符号小于等于)
40119d:  7e e4             jle     401183 <phase_6+0x8f>    # 若条件成立, 则跳转至 401183
40119f:  b8 01 00 00 00     mov     $0x1,%eax
4011a4:  ba d0 32 60 00     mov     $0x6032d0,%edx
4011a9:  cb cb             jmp     401176 <phase_6+0x82>

4011ab:  48 8b 5c 24 20     mov     0x20(%rsp),%rbx        # %rbx = (%rsp + 32) 链表的节点地址
4011b0:  48 8d 44 24 28     lea     0x28(%rsp),%rax        # 将 %rax 指向下一个节点的地址
4011b5:  48 8d 74 24 50     lea     0x50(%rsp),%rsi        # %rsi 指向链表最后一个节点的地址 (+0x50)
4011ba:  48 89 d9           mov     %rbx,%rcx
4011bd:  48 8b 10           mov     (%rax),%rdx
4011c0:  48 89 51 08       mov     %rdx,0x8(%rcx)        # 栈中后一个节点的地址变成前一个节点的 next 指针
4011c4:  48 83 c0 08       add     $0x8,%rax              # 指针移动到下一个节点
4011c8:  48 39 f0           cmp     %rsi,%rax              # 是否循环了六次?
4011cb:  74 05             je      4011d2 <phase_6+0xde>
4011cd:  48 89 d1           mov     %rdx,%rcx              # 如果没有, 继续遍历
4011d0:  eb eb             jmp     4011bd <phase_6+0xc9>
4011d2:  48 c7 42 08 00 00 00 movq     $0x0,0x8(%rdx)        # 最后一个节点的指针接地 (%rdx + 0x8) = nullptr
4011d9:  00

4011da:  bd 05 00 00 00     mov     $0x5,%ebp

4011df:  48 8b 43 08       mov     0x8(%rbx),%rax        # 跳转 <--
4011e3:  8b 00           mov     (%rax),%eax
4011e5:  39 03           cmp     %eax,(%rbx)
4011e7:  7d 05           jge     4011ee <phase_6+0xfa>
4011e9:  e8 4c 02 00 00     call    40143a <explode_bomb>
4011ee:  48 8b 5b 08       mov     0x8(%rbx),%rbx
4011f2:  83 ed 01         sub     $0x1,%ebp
4011f5:  75 e8           jne     4011df <phase_6+0xeb>    # 跳转 -->
4011f7:  48 83 c4 50       add     $0x50,%rsp
4011fb:  5b             pop     %rbx
4011fc:  5d             pop     %rbp
4011fd:  41 5c           pop     %r12
4011ff:  41 5d           pop     %r13
401201:  41 5e           pop     %r14
401203:  c3             ret

0000000000401204 <fun7>:
401204:  48 83 ec 08       sub     $0x8,%rsp
401208:  48 85 ff         test    %rdi,%rdi
40120b:  74 2b           je      401238 <fun7+0x34>
40120d:  8b 17           mov     (%rdi),%edx
40120f:  39 f2           cmp     %esi,%edx
401211:  7e 0d           jle     401220 <fun7+0x1c>
401213:  48 8b 7f 08       mov     0x8(%rdi),%rdi
401217:  e8 e8 ff ff      call    401204 <fun7>
40121c:  01 c0           add     %eax,%eax
40121e:  eb 1d           jmp     40123d <fun7+0x39>
401220:  b8 00 00 00 00     mov     $0x0,%eax
401225:  39 f2           cmp     %esi,%edx

```

```

401227: 74 14          je      40123d <fun7+0x39>
401229: 48 8b 7f 10    mov     0x10(%rdi),%rdi
40122d: e8 d2 ff ff    call    401204 <fun7>
401232: 8d 44 00 01    lea     0x1(%rax,%rax,1),%eax
401236: eb 05          jmp     40123d <fun7+0x39>
401238: b8 ff ff ff    mov     $0xffffffff,%eax
40123d: 48 83 c4 08    add     $0x8,%rsp
401241: c3            ret

```

000000000401242 <secret_phase>:

```

401242: 53            push    %rbx
401243: e8 56 02 00 00 call    40149e <read_line>
401248: ba 0a 00 00 00 mov     $0xa,%edx
40124d: be 00 00 00 00 mov     $0x0,%esi
401252: 48 89 c7      mov     %rax,%rdi
401255: e8 76 f9 ff ff call    400bd0 <strtol@plt>
40125a: 48 89 c3      mov     %rax,%rbx
40125d: 8d 40 ff      lea     -0x1(%rax),%eax
401260: 3d e8 03 00 00 cmp     $0x3e8,%eax
401265: 76 05        jbe     40126c <secret_phase+0x2a>
401267: e8 ce 01 00 00 call    40143a <explode_bomb>
40126c: 89 de        mov     %ebx,%esi
40126e: bf f0 30 60 00 mov     $0x6030f0,%edi
401273: e8 8c ff ff ff call    401204 <fun7>
401278: 83 f8 02      cmp     $0x2,%eax
40127b: 74 05        je      401282 <secret_phase+0x40>
40127d: e8 b8 01 00 00 call    40143a <explode_bomb>
401282: bf 38 24 40 00 mov     $0x402438,%edi
401287: e8 84 f8 ff ff call    400b10 <puts@plt>
40128c: e8 33 03 00 00 call    4015c4 <phase_defused>
401291: 5b           pop     %rbx
401292: c3           ret
401293: 90           nop
401294: 90           nop
401295: 90           nop
401296: 90           nop
401297: 90           nop
401298: 90           nop
401299: 90           nop
40129a: 90           nop
40129b: 90           nop
40129c: 90           nop
40129d: 90           nop
40129e: 90           nop
40129f: 90           nop

```