**汇编实践实验报告**



|  |  |
| --- | --- |
| 实验名称： | 二进制炸弹 |
| 姓 名： | 张绍磊 |
| 学 号： | 2016211392 |
| 班 级： | 2016211310 |
| 专 业： | 计算机科学与技术 |
| 指导教师： | 周峰 |

2018年 7月 25日

目录

[一、实验目的 3](#_Toc520586094)

[二、实验环境 3](#_Toc520586095)

[三、实验内容 3](#_Toc520586096)

[四、实验步骤 4](#_Toc520586097)

[五、实验原理 4](#_Toc520586098)

[六、破译过程 7](#_Toc520586099)

[主函数分析 7](#_Toc520586100)

[反汇编 9](#_Toc520586101)

[拆除炸弹 10](#_Toc520586102)

[1. Phase\_1 10](#_Toc520586103)

[2. Phase\_2: 11](#_Toc520586104)

[3. Phase\_3 14](#_Toc520586105)

[4. Phase\_4 18](#_Toc520586106)

[5. Phase\_5 22](#_Toc520586107)

[6. Phase\_6 26](#_Toc520586108)

[7. Secret\_Phase 29](#_Toc520586109)

[七、实验结果 34](#_Toc520586110)

[八、收获与体验 35](#_Toc520586111)

[九、参考文献 36](#_Toc520586112)

# 一、实验目的

1.理解C语言程序的机器级表示

2.初步掌握GDB调试器的用法

3.阅读C编译器生成的x86-64机器代码，理解不同控制结构生成的基本指令模式，过程的实现。

# 二、实验环境

Window 10；

UltraEdit 二进制文件编辑器；

Visual Studio 2017；

Linux服务器；

GDB调试工具；

# 三、实验内容

输入学号和密码下载为你定制的可运行的bomb，该程序是由C语言代码编译生成的，运行该程序要求你输入6个字符串，若输入错误则炸弹爆炸；要求阅读汇编代码，从而推测出所要求的字符串。

# 四、实验步骤

1. 输入学号和密码下载定制的可运行的bomb。

2. 将.tar文件上传至服务器<http://10.105.222.108:16310/>。

3. 输入命令tar -xvf bomb22.tar将bomb22.tar中的文件减压到当前文件夹。

4. 输入命令objdump -d bomb22 -d bomb22 > bomb22。

5. 将bomb22.s下载到本地。

6. 阅读.s文件，分析推测程序要求输入的字符串格式。

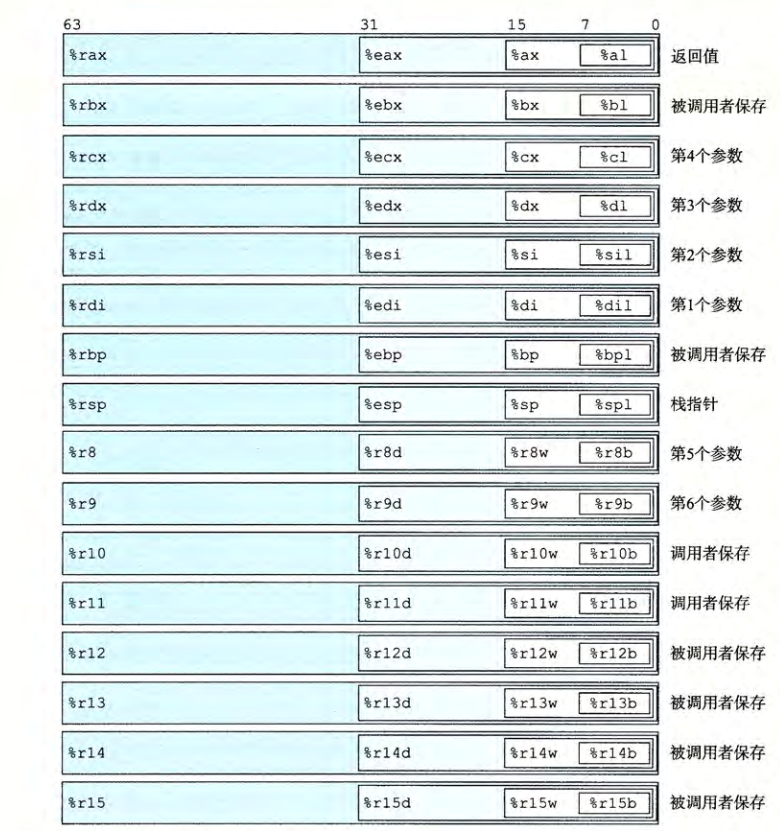
7. 在服务器上，运行bomb22，输入相关字符。

8. 判断测试结果是否正确。

9. 撰写实验报告。

# 五、实验原理

**X86-84寄存器含义：**

、

**GDB调试方法：**

gcc -g main.c //在目标文件加入源代码的信息

(gdb) start //开始调试

(gdb) n //一条一条执行

(gdb) step/s //执行下一条，如果函数进入函数

(gdb) backtrace/bt //查看函数调用栈帧

(gdb) info/i locals //查看当前栈帧局部变量

(gdb) frame/f //选择栈帧，再查看局部变量

(gdb) print/p //打印变量的值

(gdb) finish //运行到当前函数返回

(gdb) set var sum=0 //修改变量值

(gdb) list/l 行号或函数名 //列出源码

(gdb) display/undisplay sum //每次停下显示变量的值/取消跟踪

(gdb) break/b 行号或函数名 //设置断点

(gdb) continue/c //连续运行

(gdb) info/i breakpoints //查看已经设置的断点

(gdb) delete breakpoints 2 //删除某个断点

(gdb) disable/enable breakpoints 3 //禁用/启用某个断点

(gdb) break 9 if sum != 0 //满足条件才激活断点

(gdb) run/r //重新从程序开头连续执行

(gdb) watch input[4] //设置观察点

(gdb) info/i watchpoints //查看设置的观察点

(gdb) x/7b input //打印存储器内容，b--每个字节一组，7--7组

(gdb) disassemble //反汇编当前函数或指定函数

(gdb) si // 一条指令一条指令调试 而 s 是一行一行代码

(gdb) info registers // 显示所有寄存器的当前值

(gdb) x/20 $esp //查看内存中开始的20个数

**破解流程：**

（1）反汇编成.s文件；

（2）阅读源码，逐句翻译；

（3）对源码进行功能分析和逻辑分析；

（4）找到防止爆炸的密码；

（5）设置断点，开始测试。

# 六、破译过程

## 主函数分析

从主函数，得知程序一共分为6关，每关包含read\_line()、phase\_1(input)和phase\_defused()三部分。

通过调用库函数readline读入一行数据，字符串地址通过eax返回，然后将字符串传给phase\_1函数。同样，其后几个阶段遵循这种处理方式：读取一行字符串，调用phase\_n，传入字符串参数。

具体主函数如下所示：

#include *<stdio.h>*

#include *<stdlib.h>*

#include *"support.h"*

#include *"phases.h"*

**FILE** \*infile;

**int** **main**(**int** argc, **char** \*argv[])

{

**char** \*input;

*/\* Note to self: remember to port this bomb to Windows and put a*

*\* fantastic GUI on it. \*/*

*/\* When run with no arguments, the bomb reads its input lines*

*\* from standard input. \*/*

**if** (argc == **1**) {

infile = stdin;

}

**else** **if** (argc == **2**) {

**if** (!(infile = fopen(argv[**1**], "r"))) {

printf("%s: Error: Couldn't open %s**\n**", argv[**0**], argv[**1**]);

exit(**8**);

}

}

*/\* You can't call the bomb with more than 1 command line argument. \*/*

**else** {

printf("Usage: %s [<input\_file>]**\n**", argv[**0**]);

exit(**8**);

}

*/\* Do all sorts of secret stuff that makes the bomb harder to defuse. \*/*

initialize\_bomb();

printf("Welcome to my fiendish little bomb. You have 6 phases with**\n**");

printf("which to blow yourself up. Have a nice day!**\n**");

*/\* 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!*

*\* Let me know how they did it. \*/*

printf("Phase 1 defused. How about the next one?**\n**");

input = read\_line();

phase\_2(input);

phase\_defused();

printf("That's number 2. Keep going!**\n**");

input = read\_line();

phase\_3(input);

phase\_defused();

printf("Halfway there!**\n**");

*/\* Oh yeah? Well, how good is your math? Try on this saucy problem! \*/*

input = read\_line();

phase\_4(input);

phase\_defused();

printf("So you got that one. Try this one.**\n**");

*/\* Round and 'round in memory we go, where we stop, the bomb blows! \*/*

input = read\_line();

phase\_5(input);

phase\_defused();

printf("Good work! On to the next...**\n**");

input = read\_line();

phase\_6(input);

phase\_defused();

**return** **0**;

}

## 反汇编

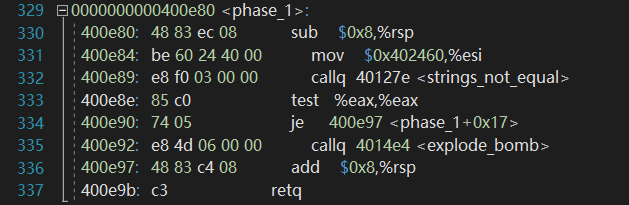
在服务器，输入命令objdump -d bomb22 -d bomb22 > bomb22，使二进制文件转化成.s汇编文件，方便阅读理解代码。

## 拆除炸弹

### 1. Phase\_1

#### （1）源代码

Phase\_1汇编源码如下：



#### （2）汇编语言翻译：

|  |  |  |
| --- | --- | --- |
| 内存地址 | 汇编代码 | 含义 |
| 400e80: | sub $0x8,%rsp | 对 rsp-0x8，保存回rsp。 |
| 400e84: | mov $0x402460,%esi | 将$0x402460的值存入esi中。 |
| 400e89: | callq 40127e <strings\_not\_equal> | 入口函数的地址 0x40127e，该函数要调用的参数为esi（即 $0x402460 处的值），和 rsp（即我们的输入）。该函数的作用是 ： 判断字符串是否相等。 |
| 400e8e: | test %eax,%eax | 函数返回值存在eax中，是判断字符串是否相等的结果。  这三句，如果 eax 不为 0，则引爆炸弹， 如果 eax 为 0，则跳转到 leave 语句函数结束。 说明 eax 为 0 时，可以进入下一关 |
| 400e90: | je 400e97 <phase\_1+0x17> |
| 400e92: | callq 4014e4 <explode\_bomb> |
| 400e97: | add $0x8,%rsp | 对 rsp+0x8，保存回rsp。 |
| 400e9b: | retq | 返回。 |

#### （3）源码分析

根据以上分析，phase\_1是从键盘输入一个值，放到rsp-0x8中，并将$0x402460的值存入esi中。接着，程序对这两个参数进行函数调用，调用判断字符串是否相等的函数 <string not equal> 进行判断，如果二者相等，则返回值为0，不引爆炸弹，反之，只要二者不相等，则炸弹爆炸。

故此处的密码存在地址$0x402460中，我们只要查看该地址的值，即可找到密码。

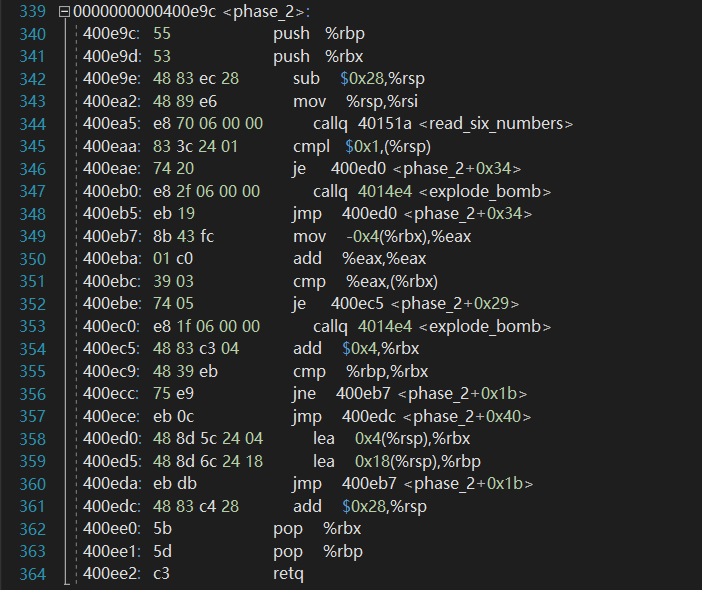
通过GDB，查找到$0x402460中的值为“When a problem comes along, you must zip it!”。

答案：When a problem comes along, you must zip it!

### 2. Phase\_2:

#### （1）源代码

Phase\_2汇编源码如下：



#### （2）汇编语言翻译：

|  |  |  |
| --- | --- | --- |
| 内存地址 | 汇编代码 | 含义 |
| 400e9c: | push %rbp | 压栈%rbx、压栈%rbp，设置%rsp。 |
| 400e9d: | push %rbx |
| 400e9e: | sub $0x28,%rsp |
| 400ea2: | mov %rsp,%rsi |
| 400ea5: | callq 40151a <read\_six\_numbers> | 入口函数的地址 0x40151a。该函数的作用是 ：输入六个数字，以上部分结束后，%rsp即为输入的第一个数字的地址。 |
| 400eaa: | cmpl $0x1,(%rsp) | 比较与1的大小。 |
| 400eae: | je 400ed0 <phase\_2+0x34> | 若等于1，跳转。 |
| 400eb0: | callq 4014e4 <explode\_bomb> | 爆炸。 |
| 400eb5: | jmp 400ed0 <phase\_2+0x34> | 跳出。 |
| 400eb7: | mov -0x4(%rbx),%eax | %eax=%rbx-0x4。 |
| 400eba: | add %eax,%eax | 当前数字乘2。 |
| 400ebc: | cmp %eax,(%rbx) | 判断是否等比 |
| 400ebe: | je 400ec5 <phase\_2+0x29> | 若相等，跳转。 |
| 400ec0: | callq 4014e4 <explode\_bomb> | 爆炸。 |
| 400ec5: | add $0x4,%rbx | %rbx=%rbx+4。 |
| 400ec9: | cmp %rbp,%rbx | 判断是否跳出循环。 |
| 400ecc: | jne 400eb7 <phase\_2+0x1b> |  |
| 400ece: | jmp 400edc <phase\_2+0x40> | 跳转至400edc。 |
| 400ed0: | lea 0x4(%rsp),%rbx | 指针后移 移到第二个数字。 |
| 400ed5: | lea 0x18(%rsp),%rbp | 这里指向最后一个数。 |
| 400eda: | jmp 400eb7 <phase\_2+0x1b> | 跳转至400eb7。 |
| 400edc: | add $0x28,%rsp | 恢复%rsp，弹出%rbx、%rbp。 |
| 400ee0: | pop %rbx |
| 400ee1: | pop %rbp |
| 400ee2: | retq | 返回。 |

#### （3）源码分析

根据汇编代码，得知需要输入六个数字，内存中输入的六个数字分布如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| %rsp | %rsp+0x4 | %rsp+0x8 | %rsp+0xC | %rsp+0x10 | %rsp+0x14 |
| Number[0] | Number[1] | Number[2] | Number[3] | Number[4] | Number[5] |

还原成C语言，phase\_2()大致如下所示：

**void** **phase\_2**()

{*//Number in %rsp,Edge in %rbp,(%register)表示寻址得到的值*

**if**((%rsp)==**1**) *//保证第一个数是1*

{

**goto** Label\_400f30;

}

**Label\_400f17**:

%eax=(%rbx-**0x4**);

%eax=**2**\*%eax;

**if**((%rbx)!=%eax) *//保证后一个数为前一个数的两倍*

{

explode\_bomb();

}

%rbx=%rbx+**0x4**;

**if**(%rbx==%rbp)

{

**return**;

}

**else**

{

**goto** Label\_400f17;

}

**Label\_400f30**:

%rbx=%rsp+**0x4**;

%rbp=%rsp+**0x18**;

**goto** Label\_400f17;

}

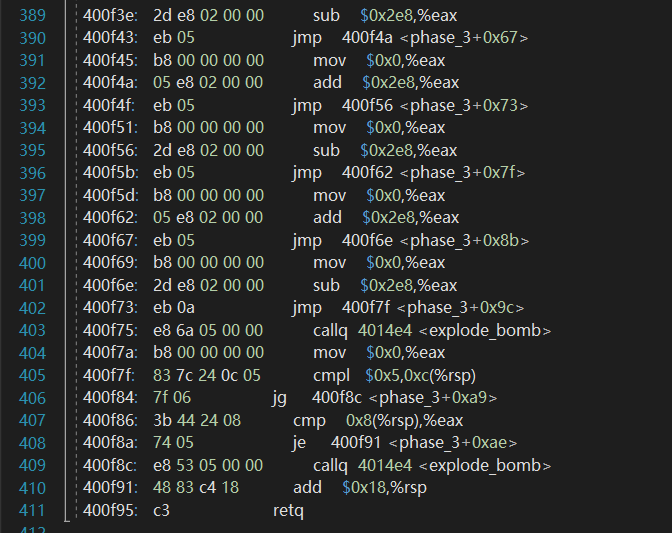
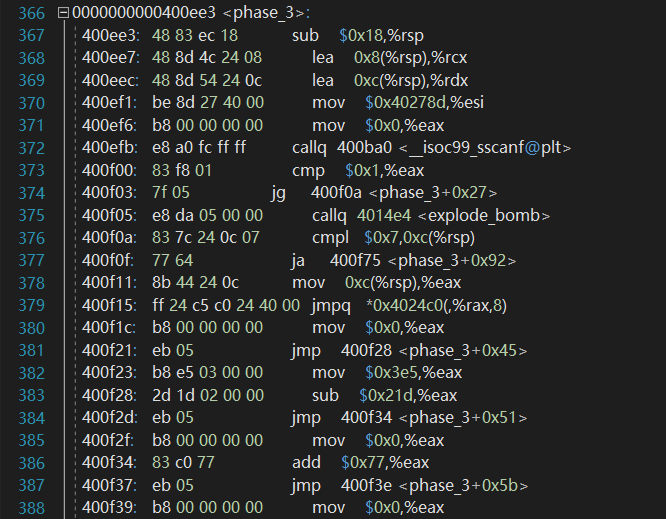
故输入为首项为1，公比为2的等比数列。

答案：1 2 4 8 16 32

### 3. Phase\_3

#### （1）源代码

Phase\_3汇编源码如下：



#### （2）汇编语言翻译：

|  |  |  |
| --- | --- | --- |
| 内存地址 | 汇编代码 | 含义 |
| 400ee3: | sub $0x18,%rsp | 设置%rsp，设置两个输入的地址。 |
| 400ee7: | lea 0x8(%rsp),%rcx |
| 400eec: | lea 0xc(%rsp),%rdx |
| 400ef1: | mov $0x40278d,%esi |
| 400ef6: | mov $0x0,%eax |
| 400efb: | callq 400ba0 <\_\_isoc99\_sscanf@plt> | 输入两个整数。返回输入个数。 |
| 400f00: | cmp $0x1,%eax | 若输入的个数小于等于1，则爆炸。 |
| 400f03: | jg 400f0a <phase\_3+0x27> |
| 400f05: | callq 4014e4 <explode\_bomb> |
| 400f0a: | cmpl $0x7,0xc(%rsp) | 将输入的第一个数和7比较，大于7则爆炸。 |
| 400f0f: | ja 400f75 <phase\_3+0x92> |
| 400f11: | mov 0xc(%rsp),%eax | Switch-Case选择跳转 |
| 400f15: | jmpq \*0x4024c0(,%rax,8) |
| 400f1c: | mov $0x0,%eax | 当a=1时，跳转至此。 |
| 400f21: | jmp 400f28 <phase\_3+0x45> |  |
| 400f23: | mov $0x3e5,%eax | 当a=0时，跳转至此。 |
| 400f28: | sub $0x21d,%eax |  |
| 400f2d: | jmp 400f34 <phase\_3+0x51> |  |
| 400f2f: | mov $0x0,%eax | 当a=2时，跳转至此。 |
| 400f34: | add $0x77,%eax |  |
| 400f37: | jmp 400f3e <phase\_3+0x5b> |  |
| 400f39: | mov $0x0,%eax | 当a=3时，跳转至此。 |
| 400f3e: | sub $0x2e8,%eax |  |
| 400f43: | jmp 400f4a <phase\_3+0x67> |  |
| 400f45: | mov $0x0,%eax | 当a=4时，跳转至此。 |
| 400f4a: | add $0x2e8,%eax |  |
| 400f4f: | jmp 400f56 <phase\_3+0x73> |  |
| 400f51: | mov $0x0,%eax |  |
| 400f56: | sub $0x2e8,%eax | 当a=5时，跳转至此。 |
| 400f5b: | jmp 400f62 <phase\_3+0x7f> |  |
| 400f5d: | mov $0x0,%eax | 当a=6时，跳转至此。 |
| 400f62: | add $0x2e8,%eax |  |
| 400f67: | jmp 400f6e <phase\_3+0x8b> |  |
| 400f69: | mov $0x0,%eax | 当a=7时，跳转至此。 |
| 400f6e: | sub $0x2e8,%eax |  |
| 400f73: | jmp 400f7f <phase\_3+0x9c> | 跳转至400f7f。 |
| 400f75: | callq 4014e4 <explode\_bomb> | 爆炸。 |
| 400f7a: | mov $0x0,%eax |  |
| 400f7f: | cmpl $0x5,0xc(%rsp) | 将输入的第一个数和5比较，大于5则爆炸。 |
| 400f84: | jg 400f8c <phase\_3+0xa9> |
| 400f86: | cmp 0x8(%rsp),%eax | 比较输入的第二个数和%eax，若不相等则爆炸，相等则成功。 |
| 400f8a: | je 400f91 <phase\_3+0xae> |
| 400f8c: | callq 4014e4 <explode\_bomb> |
| 400f91: | add $0x18,%rsp | 恢复%rsp。 |
| 400f95: | retq | 返回。 |

#### （3）源码分析

可以看出sscanf的具体作用，用x/s $rsi查看格式串，得出我们需要输入2个整数a、b，通过sscanf分别读到%rsp + 0x8，%rsp + 0xc中。



然后将输入的第一个数和7比较，大于7炸弹爆炸，大于5时炸弹爆炸。因此第一个数为1-5之间。之后通过GDB查看1-5之间的数字对于跳转位置的映射。

|  |  |
| --- | --- |
| 第一个数 | 条件跳转地址 |
| 0 | 0x 400F23 |
| 1 | 0x 400F1C |
| 2 | 0x 400F2F |
| 3 | 0x 400F39 |
| 4 | 0x 400F45 |
| 5 | 0x 400F51 |
| 6 | 0x 400F5D |
| 7 | 0x 400F69 |

通过加减运算，若最终%eax等于b则成功，否则爆炸。

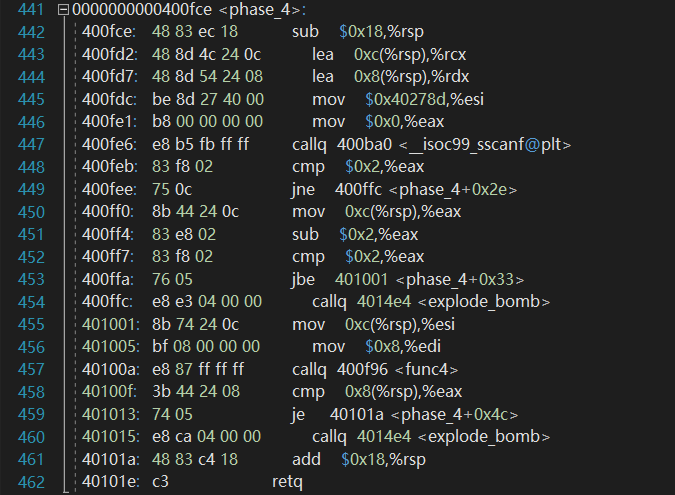
以a=5为例，程序跳转至400f51处：0 - 0x2e8 + 0x2e8 - 0x2e8 = b。故b=-0x2e8=-744。此题答案不唯一。

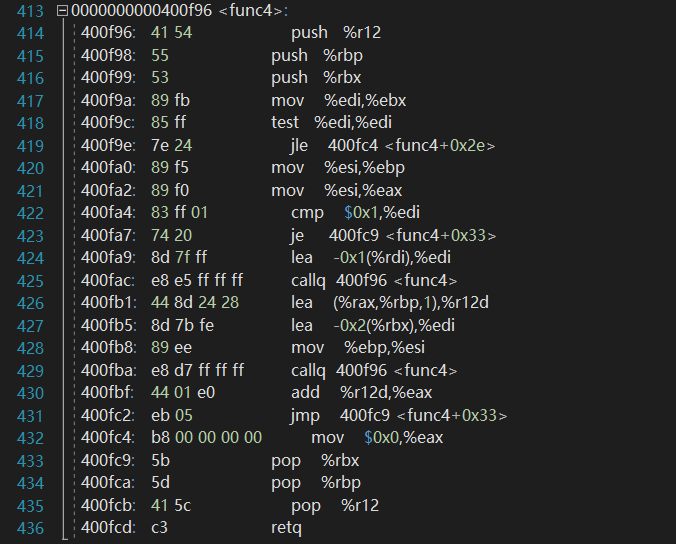
答案：5 -744

### 4. Phase\_4

#### （1）源代码

Phase\_4汇编源码如下：





#### （2）汇编语言翻译：

Phase\_4：

|  |  |  |
| --- | --- | --- |
| 内存地址 | 汇编代码 | 含义 |
| 400fce: | sub $0x18,%rsp | 设置%rsp，设置两个输入的地址。 |
| 400fd2: | lea 0xc(%rsp),%rcx |
| 400fd7: | lea 0x8(%rsp),%rdx |
| 400fdc: | mov $0x40278d,%esi |
| 400fe1: | mov $0x0,%eax |
| 400fe6: | callq 400ba0 <\_\_isoc99\_sscanf@plt> | 输入两个整数a、b。返回输入个数。 |
| 400feb: | cmp $0x2,%eax | 若输入个数不等于2，则爆炸。 |
| 400fee: | jne 400ffc <phase\_4+0x2e> |
| 400ff0: | mov 0xc(%rsp),%eax | 若输入的第二个数大于4，则爆炸。 |
| 400ff4: | sub $0x2,%eax |
| 400ff7: | cmp $0x2,%eax |
| 400ffa: | jbe 401001 <phase\_4+0x33> |
| 400ffc: | callq 4014e4 <explode\_bomb> |
| 401001: | mov 0xc(%rsp),%esi | 调用func(8，b)，并返回结果。 |
| 401005: | mov $0x8,%edi |
| 40100a: | callq 400f96 <func4> |
| 40100f: | cmp 0x8(%rsp),%eax | 比较输入的第一个数和func(8，b)，若相等则成功，否则爆炸。 |
| 401013: | je 40101a <phase\_4+0x4c> |
| 401015: | callq 4014e4 <explode\_bomb> |
| 40101a: | add $0x18,%rsp | 恢复%rsp |
| 40101e: | retq | 返回。 |

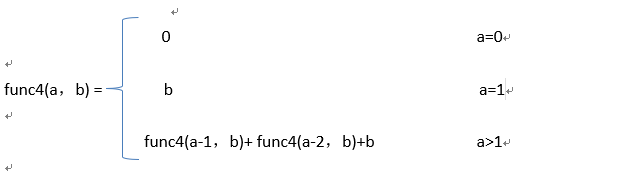
func4：

|  |  |  |
| --- | --- | --- |
| 内存地址 | 汇编代码 | 含义 |
| 400f96: | push %r12 | 压栈%r12。 |
| 400f98: | push %rbp | 压栈%rbp。 |
| 400f99: | push %rbx | 压栈%rbx。 |
| 400f9a: | mov %edi,%ebx | %ebx = %edi。 |
| 400f9c: | test %edi,%edi | 若%edi=0，返回0。 |
| 400f9e: | jle 400fc4 <func4+0x2e> |
| 400fa0: | mov %esi,%ebp | 若%edi=1，返回%rbp。 |
| 400fa2: | mov %esi,%eax |
| 400fa4: | cmp $0x1,%edi |
| 400fa7: | je 400fc9 <func4+0x33> |
| 400fa9: | lea -0x1(%rdi),%edi | 递归调用func(a-1，b)。 |
| 400fac: | callq 400f96 <func4> |
| 400fb1: | lea (%rax,%rbp,1),%r12d | 递归调用func(a-2，b)。 |
| 400fb5: | lea -0x2(%rbx),%edi |
| 400fb8: | mov %ebp,%esi |
| 400fba: | callq 400f96 <func4> |
| 400fbf: | add %r12d,%eax | 返回func4(a-1，b)+ func4(a-2，b)+b。 |
| 400fc2: | jmp 400fc9 <func4+0x33> | 跳转至400fc9 |
| 400fc4: | mov $0x0,%eax | 返回0。 |
| 400fc9: | pop %rbx | 弹栈%rbx。 |
| 400fca: | pop %rbp | 弹栈%rbp。 |
| 400fcb: | pop %r12 | 弹栈%r12。 |
| 400fcd: | retq | 返回。 |

#### （3）源码分析

可以看出sscanf的具体作用，用x/s $rsi查看格式串，得出我们需要输入2个整数a、b，通过sscanf分别读到%rsp + 0x8，%rsp + 0xc中。当a=func4(8，b)时，破解成功，否则爆炸。

其中func4(a，b)为递归程序，递归方式为：



递归计算结果如下所示：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Fun4(0,n) | Fun4(1,n) | Fun4(2,n) | Fun4(3,n) | Fun4(4,n) | Fun4(5,n) | Fun4(6,n) | Fun4(7,n) | Fun4(8,n) |
| 0 | n | 2n | 4n | 7n | 12n | 20n | 33n | 54n |

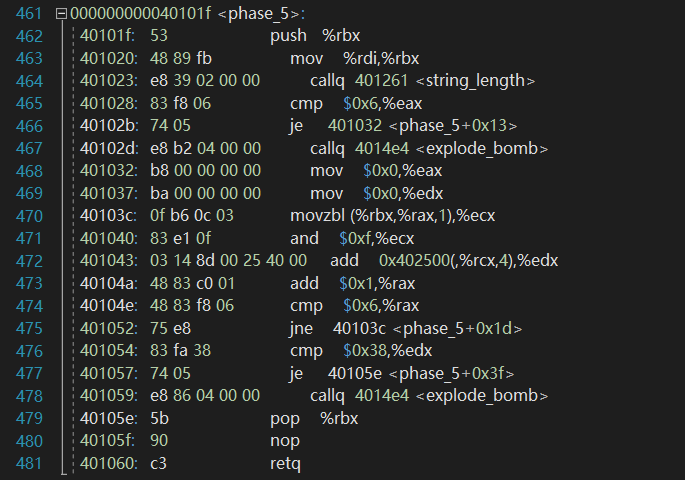
a=func4(8，b)，故a=54b。

答案：162 3

### 5. Phase\_5

#### （1）源代码

Phase\_5汇编源码如下：



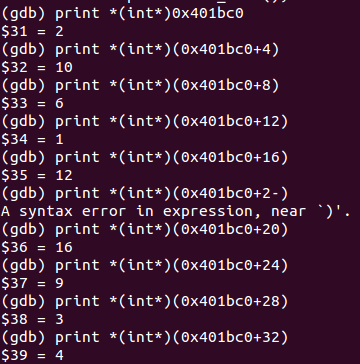
#### （2）汇编语言翻译：

|  |  |  |
| --- | --- | --- |
| 内存地址 | 汇编代码 | 含义 |
| 40101f: | push %rbx | 压栈%rbx。 |
| 401020: | mov %rdi,%rbx | %rbx=%rdi。 |
| 401023: | callq 401261 <string\_length> | 计算字符串长度。 |
| 401028: | cmp $0x6,%eax | 比较字符串长度和的大小，若不等于6，则爆炸。 |
| 40102b: | je 401032 <phase\_5+0x13> |
| 40102d: | callq 4014e4 <explode\_bomb> |
| 401032: | mov $0x0,%eax | %eax=0。 |
| 401037: | mov $0x0,%edx | %edx=0。 |
| 40103c: | movzbl (%rbx,%rax,1),%ecx | 字符的低4位ascii码值对应一个数组的index，然后数组中对应数字累加进%edx。 |
| 401040: | and $0xf,%ecx |
| 401043: | add 0x402500(,%rcx,4),%edx |
| 40104a: | add $0x1,%rax | %rax计数，%rax++。 |
| 40104e: | cmp $0x6,%rax | 若%rax小于6，跳转回40103c，循环。 |
| 401052: | jne 40103c <phase\_5+0x1d> |
| 401054: | cmp $0x38,%edx | 比较累加结果与0x38的大小，若等于则通过，否则爆炸。 |
| 401057: | je 40105e <phase\_5+0x3f> |
| 401059: | callq 4014e4 <explode\_bomb> |
| 40105e: | pop %rbx | 弹出%rbx。 |
| 40105f: | nop | 空。 |
| 401060: | retq | 返回。 |

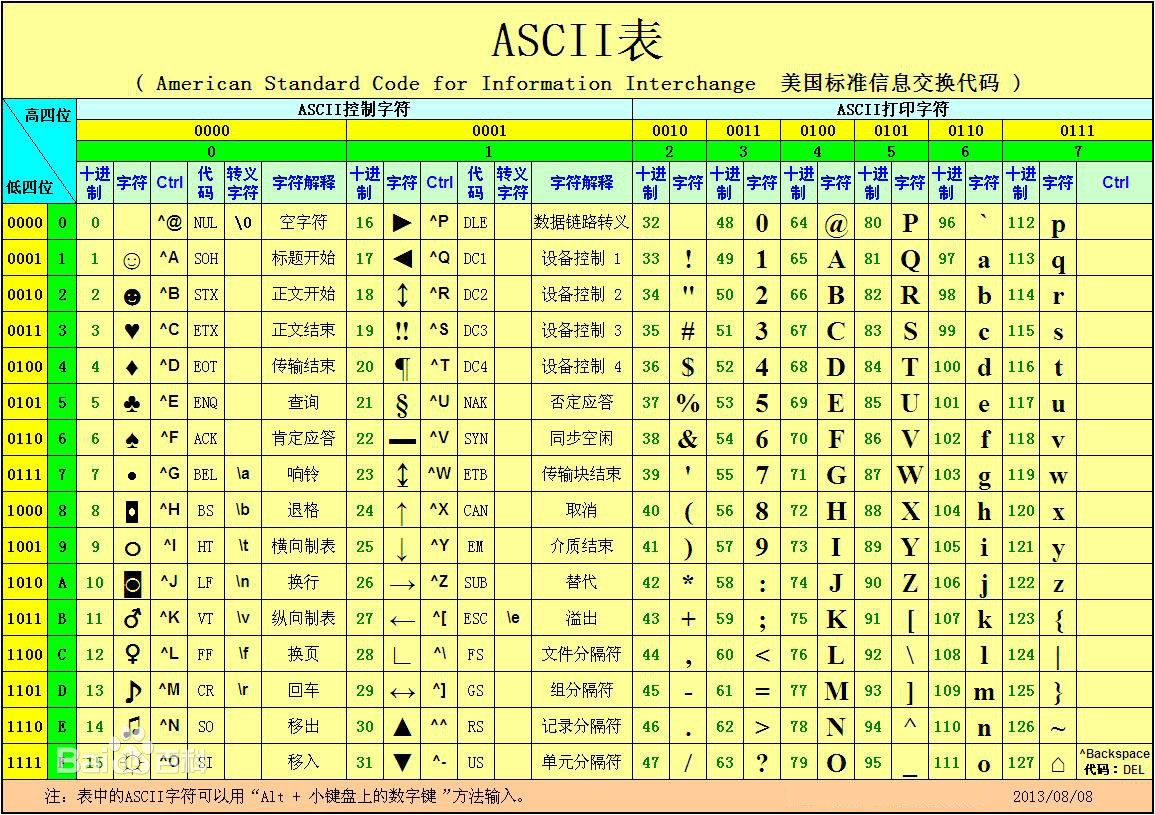
#### （3）源码分析

对于phase\_5,概括性理解，就是输入6个ascii字符，然后这6个字符的低4位对应一个数组的index，然后对应数字累加要等于0x38,即56。

通过GDB查看该数组的内容，如下所示：



56可拆分为2+2+10+10+16+16。



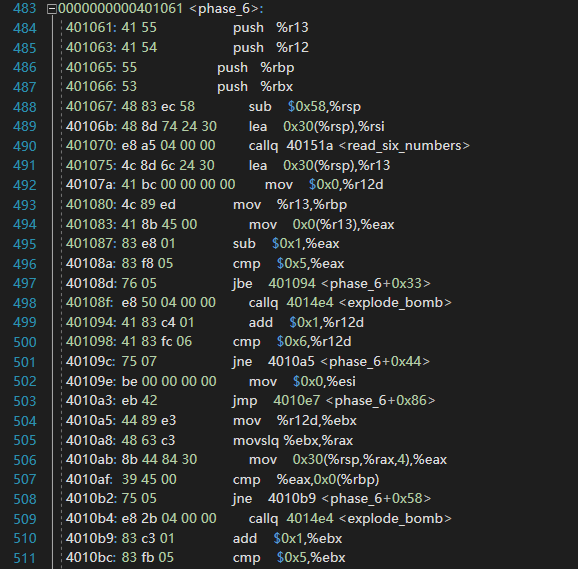
按上表中的低四位，查询出“2+2+10+10+16+16”对应的字符，应是001155。

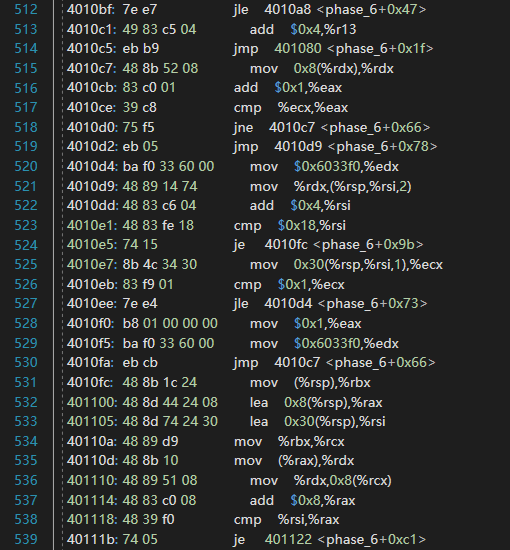
答案：001155

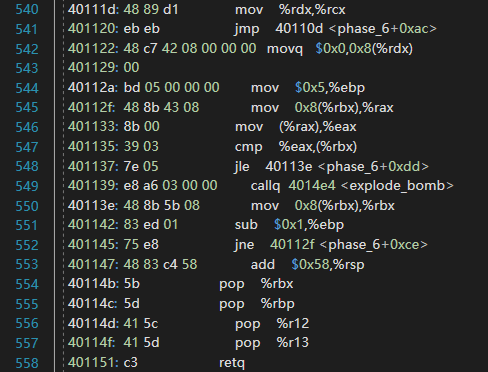
### 6. Phase\_6

#### （1）源代码

Phase\_6汇编源码如下：







#### （2）源码分析

对phase\_6（）函数反汇编。可以通过函数看出，我们输入的6位数字，且都小于6。接下去是一个while循环，r12d和ebx是一个循环计数器。该循环用于判断这6个数是否存在等于0的。可以推测得到每个数字都不为0。接下来查看0x6033F0位置处的内容，按照我们输入的数字的标号在指定位置排序。之后将node之间建立链表连接指针。最后判断该链表是否递减，如果递增，则不爆，否则爆炸。

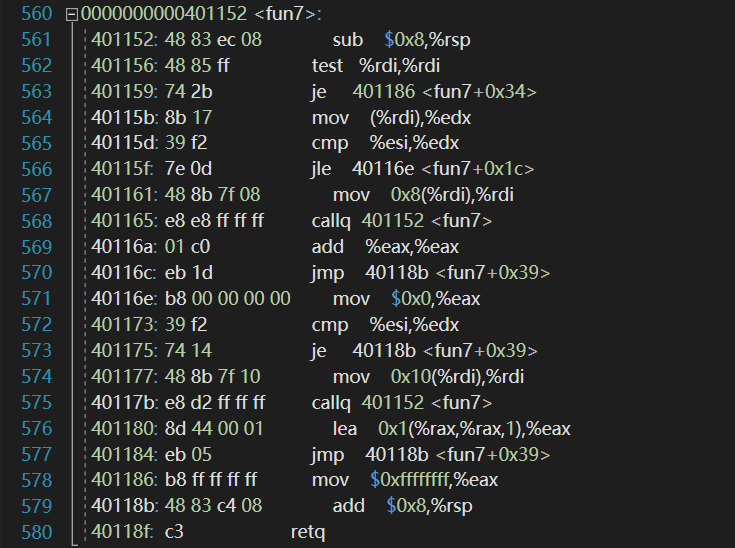
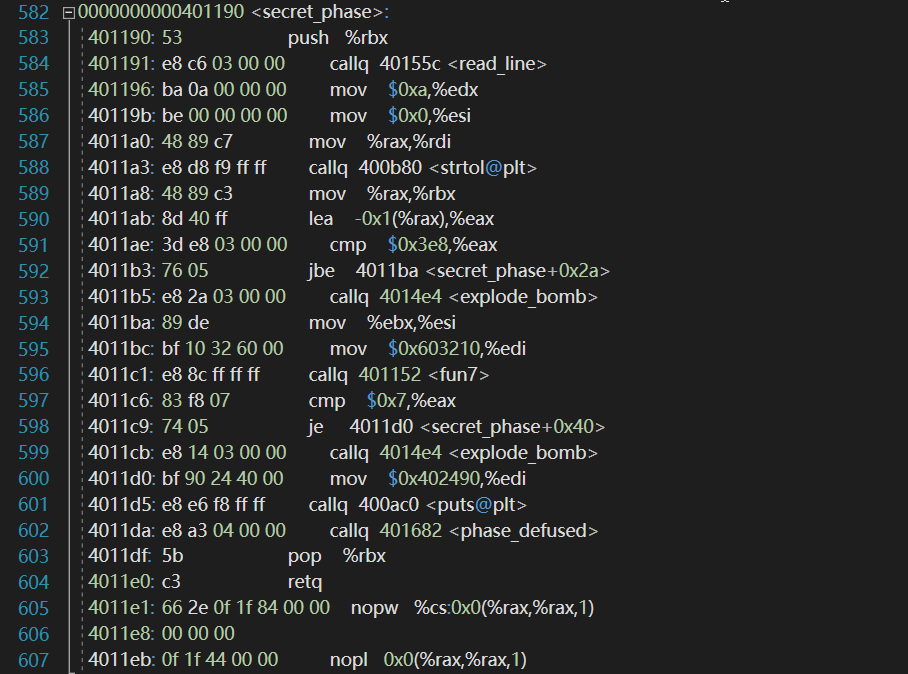
|  |  |
| --- | --- |
| Node | 数据 |
| 1 | 25d |
| 2 | 65 |
| 3 | 15D |
| 4 | 19C |
| 5 | 155 |
| 6 | 3C7 |

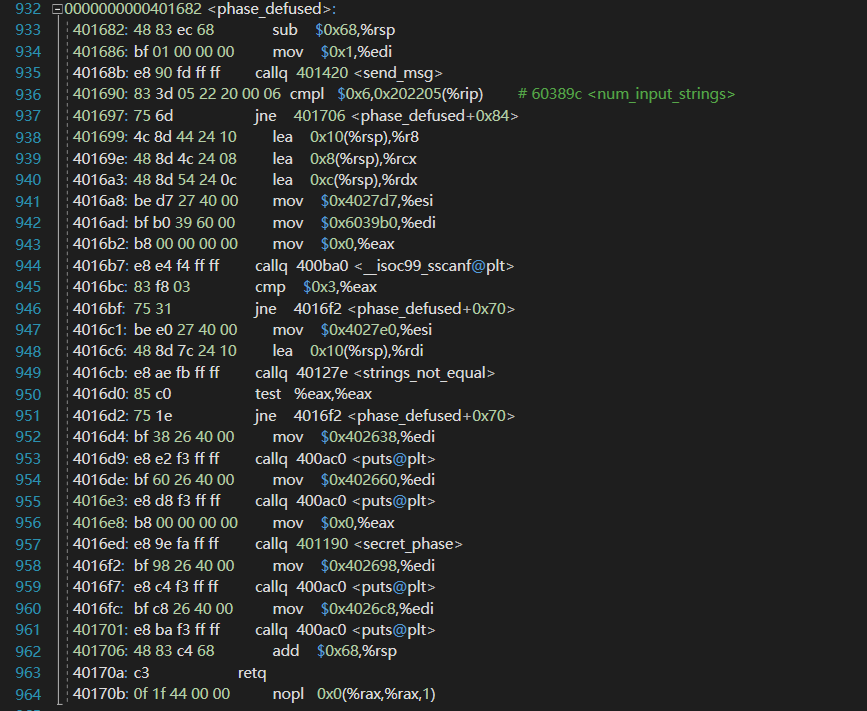
答案：2 5 3 4 1 6

### 7. Secret\_Phase

#### （1）源代码

secret\_Phase汇编源码如下：





phase\_defused

可以看到第7行将函数num\_input\_strings的返回值与6进行比较，如果不等于6则的直接跳过中间代码到达最后的结束部分。

从函数名我们可以推测这个函数的作用的是检测读取的字符串的数量，当读取了6个字符串时，就不会跳过中间的代码。第9到14行又是熟悉的sscanf调用过程，我们已经知道esi指向的是格式化字符串的首地址，查看它的内容：

https://www.viseator.com/images/bomblab9.png

读取两个整数和一个字符串。有所不同的是在12行之后又有一行给edi赋上了一个地址值，我们之前所有阶段中edi的值都是来自于我们read\_line的地址，想到sscanf 参数中确实存在一个输入，我们可以推测这个edi中存放的是我们读取位置的首地址。

那么就可以在运行时查看这个地址的内容，看是从哪里进行读取的：

https://www.viseator.com/images/bomblab10.png

第15、16行对成功输入的数据个数进行了一个判断，如果不为3个则跳过调用secret\_phase的代码。

第17-19行是对strings\_not\_equal的调用，我们已经知道它的两个参数分别是esi与edi，esi被赋上了一个地址值，edi被赋上了esp+0x10，我们可以推测出edi的地址就是指向我们读入的第三个字符串的，那么需要比较的对象是什么呢？我们在运行时查看内存的内容：

https://www.viseator.com/images/bomblab11.png

这就是需要的第三个参数。

可以看到如果第三个参数与上面这个字符串相同的话就会调用两次puts输出提示信息，然后进入secret\_phase阶段。

secret\_phase：

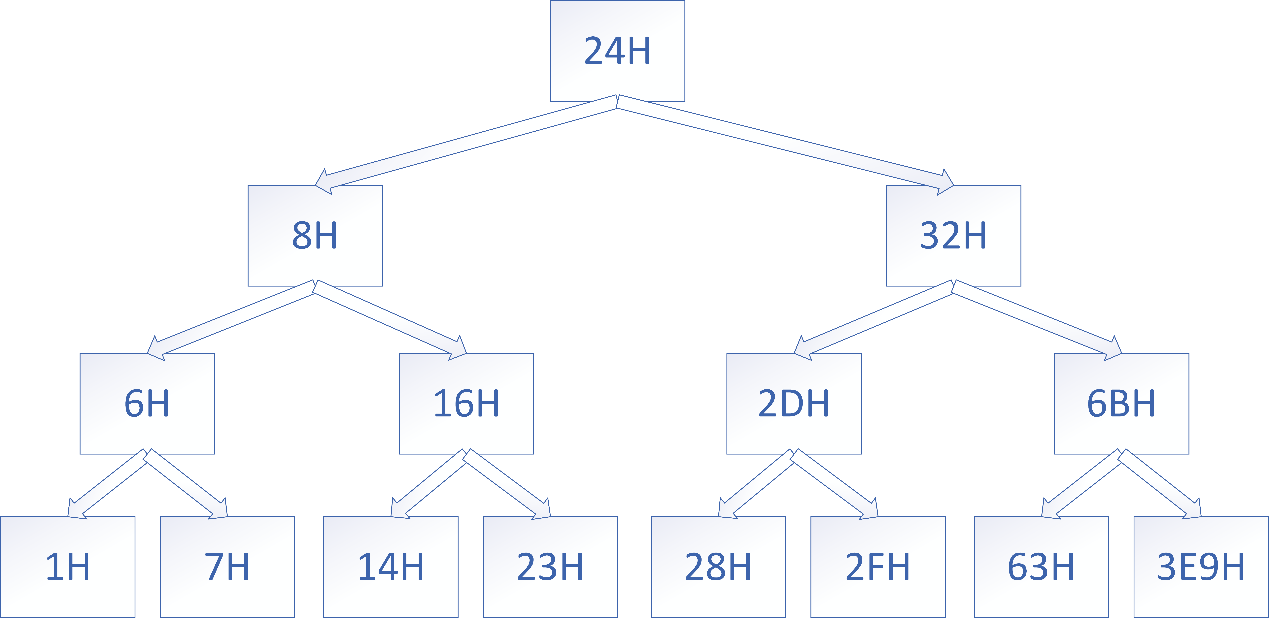
可以看到第3行调用了read\_line函数，接着把read\_line的返回值赋给了rdi，并调用了strtol函数，这个标准库函数的作用是把一个字符串转换成对应的长整型数值。返回值还是存放在rax中，第8行将rax复制给了rbx，第9行将rax减1赋给eax，第十行与0x3e8进行比较，如果这个值小于等于0x3e8就跳过引爆代码。看到这里我们可以知道我们需要再加入一行数据，它应该是一个小于等于1001的数值。

接下来将ebx赋给了esi，也就是我们一开始输入的rax值。第14行将一个地址值赋给了edi，15行调用了fun7函数。我们还是先往下了解一下我们需要得到的结果。函数返回后令返回值eax与0x7做了一个比较，如果相等则跳过引爆代码。

fun7：

第3、4两行先对我们输入的这个数作一个判断，如果等于0直接跳到第19行，返回-1，这显然不是我们想要的结果。第5行将rdi的值读入到了edx中，第6行则将这个数与我们读入的数进行比较，如果这个数小于等于我们读入的数就跳至第12行，第12行将eax置0，再进行一次相同的比较，如果相等则跳至第20行返回。

这是一个二叉树的结构，每个节点第1个8字节存放数据，第2个8字节存放左子树地址，第3个8字节存放右子树位置。并且命令也有规律，nab，a代表层数，b代表从左至右第b个节点。二叉树如下图所示：



fun7()输入的值对二叉树进行操作，并计算返回值。根据递归反写出C语言。

void fun7(Node\* node,int value)

{*//node in %rdi,value in %rsi,return\_value in %eax*

*//require %eax to be 2(Very important)*

int t=node->val;

**if**(t>value)

{

node=node->left;

fun7(node,value);

return\_value=2\*return\_value;

**return**;

}

**else**

{

**if**(value==t)

{

**return**;

}

node=node->right;

fun7(node,value);

return\_value=0x1+2\*return\_value;

**return**;

}

}

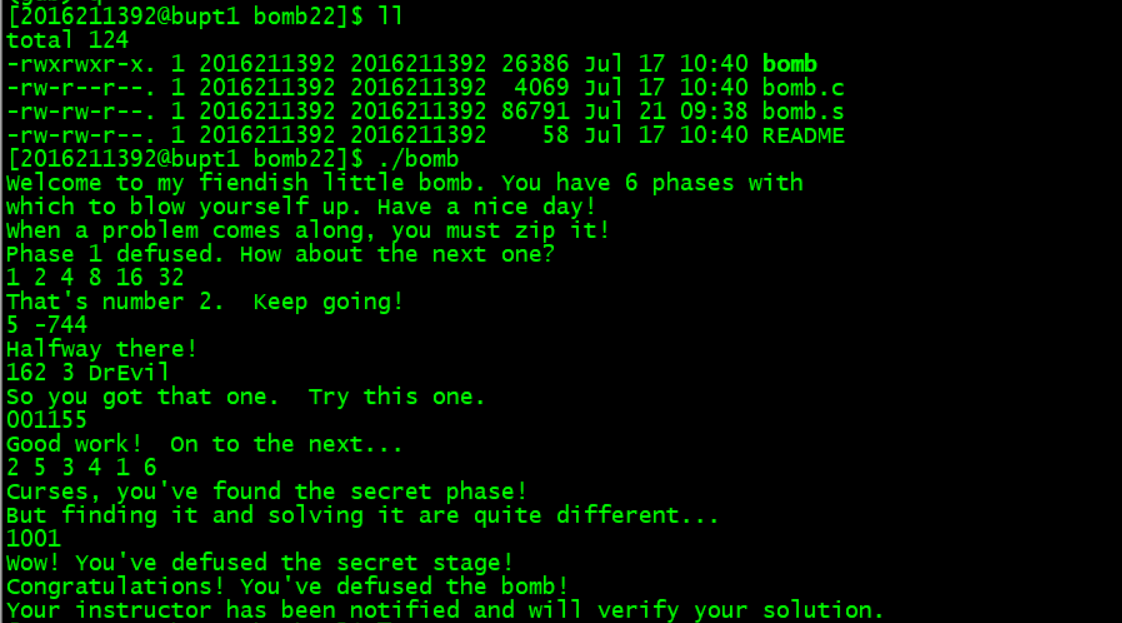
有返回值要等于0x7，故通过计算得输入1001.

答案：1001

# 七、实验结果

全部关卡全部通过，实验过程中共爆炸6次，全部出现在附加关卡中。由于没有任何提示信息，我下意识以为输入完“DrEvil”之后就紧接着附加关卡，直接就输入了附加关卡的答案，并且没意识到这个问题，一直以为是计算错误，最后才发现附加关卡是在6个关卡结束之后才显示，很遗憾在此处产生这么多次爆炸。

实验全部炸弹拆除成果如下图所示：



# 八、收获与体验

**本次实验过程中主要卡在了隐藏关卡的查找上，错误爆炸都是在尝试进入隐藏关卡时产生。****由于没有任何提示信息，我下意识以为输入完“DrEvil”之后就紧接着附加关卡，直接就输入了附加关卡的答案，并且没意识到这个问题，一直以为是计算错误，最后才发现附加关卡是在6个关卡结束之后才显示，很遗憾在此处产生这么多次爆炸，希望老师谅解。**

整个实验包括秘密部分用时8个小时，引爆了6次炸弹（全是在尝试进入Secret\_Phase时产生）。

一开始拿到题目的时候会比较蒙，需要先去学习工具的使用与一些编译的基础知道（符号表、定址表等等）花费了一些时间。前几个阶段过于关注函数的具体实现而没有根据常识去推测一些明显函数的作用花费了一些时间。

前4个阶段都算比较简单，考查了一些常用结构在汇编中的出现形式。第5、6阶段、秘密关卡分别考察了堆、链表、二叉树等数据结构在内存中的结构与汇编级的使用，受益良多。

整个破解思想基本为先翻译解释源码，找到能避开爆炸的输入，然后符合要求的输入。

从破解过程来看，需要较高的水平的汇编代码阅读能力，还要不怕麻烦的对许多地址进行递归的查询，很多时候对于 “查询哪个地址可以看到需要的东西” 的问题只有在大致的范围上不断寻找和计算才能准确找到能完美表现变化的地址。所以还是需要很大的耐心的。由于不需要大量的指令，只要懂得gdb的基本使用和查看内存的指令并且耐心调试就可以。

这个实验需要细致的分析与大胆的猜测与实验验证，还需要小心操作，最重要的是耐心，面对非常晦涩的汇编代码如何一步步地弄清代码的作用很需要毅力。当然通过这次实验，我对汇编语言有了更加透彻的了解，掌握了GDB调试方法。也更加热爱阅读汇编源码的过程。同时，也感谢老师设置这么有意思的实验，不仅提高了我的汇编能力，还增进了我对汇编语言的兴趣。

# 九、参考文献

深入理解计算机系统（原书第三版）