《计算机系统》

实验二报告

班级：计科 2302

学号：202308010208

姓名：成烨

目录

1 实验目的

本次实验为熟悉汇编程序及其调试方法的实验。

实验内容包含 2 个文件 bomb（可执行文件）和 bomb.c（c 源文件）。

实验主题内容为：

程序运行在 linux 环境中。程序运行中有 6 个关卡（6 个 phase），每个 phase 需要用户在终端上输入特定的字符或者数字才能通关，否则会引爆炸弹！那么如何才能知道输入什么内容呢？这需要你使用 gdb 工具反汇编出汇编代码，结合 c 语言文件找到每个关卡的入口函数。然后分析汇编代码，找到在每个 phase 程序段中，引导程序跳转到“explode_bomb”程序段的地方，并分析其成功跳转的条件，以此为突破口寻找应该在命令行输入何种字符通关。

2 实验准备

gdb 工具、objdump 工具

3 实验过程

(1) <phase_1>

```
08048b33 <phase_1>:
 8048b33:   83 ec 14              sub    $0x14,%esp
 8048b36:   68 24 a0 04 08        push   $0x804a024
 8048b3b:   ff 74 24 1c           push   0x1c(%esp)
 8048b3f:   e8 c9 04 00 00        call   804900d <strings_not_equal>
 8048b44:   83 c4 10              add    $0x10,%esp
 8048b47:   85 c0                 test   %eax,%eax
 8048b49:   74 05                 je     8048b50 <phase_1+0x1d>
 8048b4b:   e8 b4 05 00 00        call   8049104 <explode_bomb>
 8048b50:   83 c4 0c              add    $0xc,%esp
 8048b53:   c3                    ret
```

知识：

test %eax, %eax：通常用于条件分支，检查 %eax 是否为零

je label：如果 %eax 为零，跳转到 label。

函数的参数从右到左依次压入堆栈（参数是栈顶指针指向的值），返回值储存在 eax 寄存器中。

分析：

第 369 行的 call 指令，调用<explode_bomb>。为了不引爆炸弹，程序必须在第 368 行跳转。即 je=1，从而倒推 eax 中的值=0。发现第 365 行调用了一个<string_not_equal>的函数。字符串相等，返回 0。返回值保存在 eax 中。可以猜测，要求输入的字符是传入<string_not_equal>函数的参数。而函数的参数在栈顶。第 363 行 push 了一个立即数。不难猜测，该立即数是传入<string_not_equal>函数的参数，保存的是待比较的字符串的地址。

检验：

（1）通过 gdb 调试检验

```
(gdb) disassemble
Dump of assembler code for function strings_not_equal:
=> 0x0804900d <+0>:     push   %edi
   0x0804900e <+1>:     push   %esi
   0x0804900f <+2>:     push   %ebx
   0x08049010 <+3>:     mov    0x10(%esp),%ebx
   0x08049014 <+7>:     mov    0x14(%esp),%esi
   0x08049018 <+11>:    push   %ebx
   0x08049019 <+12>:    call   0x8048fee <string_length>
   0x0804901e <+17>:    mov    %eax,%edi
   0x08049020 <+19>:    mov    %esi,(%esp)
   0x08049023 <+22>:    call   0x8048fee <string_length>
   0x08049028 <+27>:    add    $0x4,%esp
   0x0804902b <+30>:    mov    $0x1,%edx
   0x08049030 <+35>:    cmp    %eax,%edi
--Type <RET> for more, q to quit, c to continue without paging--c
```

```
362   8048b33:   83 ec 14              sub    $0x14,%esp #esp-20
363   8048b36:   68 24 a0 04 08        push   $0x804a024 #esp-24 立即数压栈
364   8048b3b:   ff 74 24 1c           push   0x1c(%esp) #esp-28 esp地址中的数压栈
365   8048b3f:   e8 c9 04 00 00        call   804900d <strings_not_equal>
```

从函数入口 0x0804900d 开始，连续三次压栈，esp 的值变为 esp-40。

0x08049010<+3>:将 esp 的偏移地址中的值传递给 ebx。该地址是 esp-40+16=esp-24;从汇编代码可知，这是调用<string_not_equal>时第一次压栈的栈顶指针的位置。

0x08049010<+12>:将 ebx 中的值压栈，作为<string_length>的参数。可见，esp-24 栈顶指针指向的内容为传入函数<strinh_not_equal>的一个参数，应该为待比较字符串的地址。

(2)**验证**

```
xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
```

(2) <phase_2>

```
385   8048b76:   83 7c 24 04 00          cmpl   $0x0,0x4(%esp)
386   8048b7b:   75 07                   jne    8048b84 <phase_2+0x30> #ZF=1,不跳
387   8048b7d:   83 7c 24 08 01          cmpl   $0x1,0x8(%esp)
388   8048b82:   74 05                   je     8048b89 <phase_2+0x35> #ZF=1,跳
389   8048b84:   e8 7b 05 00 00          call   8049104 <explode_bomb>
```

为了拆除炸弹，注意到第 385-389 行。合适的输出应该是 esp 向上偏移 4 个字节处的内存的内容等于 0，ZF=1,jne 不跳转，继续执行下一条 cmpl 指令，esp 向上偏移 8 个字节处的内存的内容等于 1，ZF=1，je 跳转，跳过下一条<explode_bomb>函数。

由此，可以猜测，传入<read_six_numbers>的第一个参数和第二个参数分别为 0 和 1。

```
390   8048b89:   8d 5c 24 04             lea    0x4(%esp),%ebx
391   8048b8d:   8d 74 24 14             lea    0x14(%esp),%esi
392   8048b91:   8b 43 04                mov    0x4(%ebx),%eax
393   8048b94:   03 03                   add    (%ebx),%eax
394   8048b96:   39 43 08                cmp    %eax,0x8(%ebx)
395   8048b99:   74 05                   je     8048ba0 <phase_2+0x4c>
396   8048b9b:   e8 64 05 00 00          call   8049104 <explode_bomb>
397   8048ba0:   83 c3 04                add    $0x4,%ebx          ebx+4+4+4+4+4
398   8048ba3:   39 f3                   cmp    %esi,%ebx          #esi=esp+20,ebx
399   8048ba5:   75 ea                   jne    8048b91 <phase_2+0x3d> #5次 jne=1,不跳转
```

解答这道题目，**最核心的部分是汇编代码中的循环部分。**【实际是：斐波那契数列的逻辑】

初始化：

第 390-391 行：lea 指令，用于计算地址。利用 esp 向上偏移 4 个字节的地址初始化 ebx；利用 esp 向上偏移 20 个字节的地址初始化 esi；

循环内部：

第 392-393 行：将 ebx 向上偏移 4 个字节地址空间的值保存到 eax；将 ebx 地址的值累加到 eax 的值上；

第 394-396 行：cmp 指令，将累加的结果和 ebx 向上偏移 8 个字节地址空间的值比较。两者相等，则 ZF=1。je 跳转，跳过<explode_bomb>函数；【实际作用：相邻四个字节空间的值相加等于紧随其后的四个字节空间的值。】

第 397-398 行：更新 ebx 的值，向上加 4。给出循环终止条件：将 esi 中的值和 ebx 中的值比较，两者不等 ZF=0，jne 跳转到第 392 行。两者相等 ZF=1，jne 不跳转，结束循环。

bomb7 > C phase_2.c > phase_2()

```c
void phase_2(){
    int m[6];
    scanf("%d %d %d %d %d %d",&m[0],&m[1],&m[2],&m[3],&m[4],&m[5]);
    if((m[0]!=0)||(m[1]!=1)){
        bomb();
    }
    for(int i=2;i<6;i++){
        if(m[i]!=(m[i-1]+m[i-2])){
            bomb();
        }
    }
}
```

Phase_2()函数 c 代码

综上，累加的结果为：0，1，1，2，3，5

```
xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2$ cd bomb7
xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
```

（3）<phase_3>

```
8048bcc:   31 c0                xor    %eax,%eax      #eax清零
8048bce:   8d 44 24 08          lea    0x8(%esp),%eax
8048bd2:   50                   push   %eax           #esp-32指向esp-20的地址(即传入的第一个参数)
8048bd3:   8d 44 24 08          lea    0x8(%esp),%eax
8048bd7:   50                   push   %eax           #esp-36指向esp-24的地址（即传入的第二个参数）
8048bd8:   68 0f a2 04 08       push   $0x804a20f     #esp-40(即传入的第三个参数)
8048bdd:   ff 74 24 2c          push   0x2c(%esp)     #esp-44指向esp的值(即传入的第四个参数)
8048be1:   e8 2a fc ff ff       call   8048810 <__isoc99_sscanf@plt> #输入符合格式的字符，返回的是有效数据项的个数
```

分析函数__isoc99_sscanf：

（1）基本知识：

__isoc99_sscanf 是 C 标准库函数 sscanf 的一个变体，用于从字符串中读取格式化数据。它的函数签名如下：

int __isoc99_sscanf(const char *str, const char *format, ...);

参数如下：

const char *str：输入字符串，从中读取数据。

const char *format：格式化字符串，用于指定如何解析输入数据。

...：可变参数列表，用于存储解析后的数据。

返回值如下：

返回值 ＞0：表示成功读取的输入项数量。例如，返回值为 2 表示成功解析了两个输入项；

返回值 ＝0：表示没有成功解析任何输入项。这可能是因为输入为空；

返回值 ＝EOF（通常是 -1）：表示读取失败。这通常发生在格式字符串无效的情况下。

（2）注意到 call 指令之前的四个 push 指令。

已知参数是从右到左压入堆栈，上述四次 push 分别传入四个参数。其中第三个参数是立即数 0x804a20f 地址空间上的内容。

```
(gdb) x/s 0x804a20f
0x804a20f:      "%d %d"
```

上图表明，将字符串解析为两个整数。如果解析成功，该函数返回 2。

分析第一个<explode_bomb>：

```
420   8048be1:   e8 2a fc ff ff          call   8048810 <__isoc99_sscanf@plt> #输入符合格式的字符，返回的是有效数据项的个数
421   8048be6:   83 c4 10                add    $0x10,%esp
422   8048be9:   83 f8 01                cmp    $0x1,%eax          #eax>1，有效数据项个数至少=2
423   8048bec:   7f 05                   jg     8048bf3 <phase_3+0x34>
424   8048bee:   e8 11 05 00 00          call   8049104 <explode_bomb> #跳过
```

函数__isoc99_sscanf 返回值存储在 eax 中。为了 jg 跳转，跳过<explode_bomb>，eax 中值大于 1，使得 eflag 寄存器组 sf==of，zf=0，由此可以推知，**函数__isoc99_sscanf 必须正确解析字符串，且解析后的数据个数至少为 2。**

分析第二个<explode_bomb>：

```
425   8048bf3:   83 7c 24 04 07          cmpl   $0x7,0x4(%esp)          #esp-24地址的值<=7 不跳转
426   8048bf8:   77 3c                   ja     8048c36 <phase_3+0x77>
427   8048bfa:   8b 44 24 04             mov    0x4(%esp),%eax
428   8048bfe:   ff 24 85 a0 a0 04 08    jmp    *0x804a0a0(,%eax,4)       【(1,429);(2,431);(3,433);(4,435);(5,437);(6,439);(7,441)】
429   8048c05:   b8 3a 00 00 00          mov    $0x3a,%eax
430   8048c0a:   eb 3b                   jmp    8048c47 <phase_3+0x88>
431   8048c0c:   b8 58 03 00 00          mov    $0x358,%eax
432   8048c11:   eb 34                   jmp    8048c47 <phase_3+0x88>
433   8048c13:   b8 43 01 00 00          mov    $0x143,%eax
434   8048c18:   eb 2d                   jmp    8048c47 <phase_3+0x88> #第447条
435   8048c1a:   b8 3f 02 00 00          mov    $0x23f,%eax
436   8048c1f:   eb 26                   jmp    8048c47 <phase_3+0x88>
437   8048c21:   b8 57 01 00 00          mov    $0x157,%eax
438   8048c26:   eb 1f                   jmp    8048c47 <phase_3+0x88>
439   8048c28:   b8 7b 02 00 00          mov    $0x27b,%eax
440   8048c2d:   eb 18                   jmp    8048c47 <phase_3+0x88>
441   8048c2f:   b8 ad 03 00 00          mov    $0x3ad,%eax
442   8048c34:   eb 11                   jmp    8048c47 <phase_3+0x88>#无条件跳转
443   8048c36:   e8 c9 04 00 00          call   8049104 <explode_bomb>
```

第 425-426 行：由于 0x8048c36 有炸弹，所以 ja 不可跳转。由此推得，cmpl 指令执行完成后 eflags 寄存器中（zf=1||cf=0),即 esp-24 的地址处的值小于等于 7。而这是 push 进函数__isoc99_sscanf 的第二个参数。**由此可得，待输入的第一个整数大于 0，小于等于 7。**

第 427-428 行：输入的第一个整数值存在 esp-24 的地址中，而它又保存在 eax 中，eax 乘以比例因子作为偏移地址，加上基地址 0x804a0a0，得到 jmp 要跳转的地址所保存在的内存空间，

第 428-442 行：条件分支指令。

```
(gdb) x/7wx 0x804a0a4
0x804a0a4:      0x08048c05      0x08048c0c      0x08048c13      0x08048c1a
0x804a0b4:      0x08048c21      0x08048c28      0x08048c2f
(gdb)
```

eax 中值为 1，eax 被赋值，jmp 跳转到 0x804a04 地址空间储存的值处（0x08048c05，即汇编代码的第 429 行）；

eax 中值为 2，eax 被赋值，jmp 跳转到 0x804a0a8 地址空间储存的值处（0x08048c0c，即汇编代码的第 431 行）；

依次类推。

分析程序的最后一部分：

```
443    8048c36:   e8 c9 04 00 00          call   8049104 <explode_bomb>
444    8048c3b:   b8 00 00 00 00          mov    $0x0,%eax
445    8048c40:   eb 05                   jmp    8048c47 <phase_3+0x88>
446    8048c42:   b8 23 02 00 00          mov    $0x223,%eax
447    8048c47:   3b 44 24 08             cmp    0x8(%esp),%eax
448    8048c4b:   74 05                   je     8048c52 <phase_3+0x93>
449    8048c4d:   e8 b2 04 00 00          call   8049104 <explode_bomb>
450    8048c52:   8b 44 24 0c             mov    0xc(%esp),%eax  #esp-28+12=esp-16
451    8048c56:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax   #清零
452    8048c5d:   74 05                   je     8048c64 <phase_3+0xa5> #跳转
453    8048c5f:   e8 2c fb ff ff          call   8048790 <__stack_chk_fail@plt>
454    8048c64:   83 c4 1c                add    $0x1c,%esp  #栈清空
455    8048c67:   c3                      ret
```

当条件分支正确执行后，程序会跳过第三个<explode_bomb>,直接执行 0x8048c47 处的指令。比较 eax 的值和 push 进的第一个参数，当两者相等，eflags 寄存器中 zf=1，je 会跳转到 0x8048c52 处执行指令。这样，程序就跳过了在 0x8048c4d 处的最后一个<explode_bomb>。第 450-455 行，将 eax 清零，je 跳转，栈清空，phase_3 完成。

综上，本题需传入两个整数。第一个整数的大小在 0~7 之间，第二个整数的大小取决于条件分支中 eax 的值。

```
1 58
Halfway there!
```

```
2 856
Halfway there!
```

```
3 323
Halfway there!
```

```
4 575
Halfway there!
```

```
5 343
Halfway there!
```

```
6 635
Halfway there!
```

```
7 941
Halfway there!
```

(4)<phase_4>

首先分析__isoc99_sscanf@plt 函数，

```
Breakpoint 1, 0x08048cc6 in phase_4 ()
(gdb) x/s 0x804a20f
0x804a20f:      "%d %d"
```

和第三关一样，查看格式字符串的方式，“%d %d”。

```
513   8048cf5:   83 7c 24 04 0e          cmpl   $0xe,0x4(%esp)
514   8048cfa:   76 05                    jbe    8048d01 <phase_4+0x3b>
515   8048cfc:   e8 03 04 00 00           call   8049104 <explode_bomb>
```

0x4(%esp)指向的地址是 push 给__isoc99_sscanf@plt 函数的最后一个参数的地址。为了不引爆炸弹，终端输入的数字从右往左第二个数应该小于等于 14。

```
524   8048d19:   83 7c 24 08 05          cmpl   $0x5,0x8(%esp)
525   8048d1e:   74 05                    je     8048d25 <phase_4+0x5f>
526   8048d20:   e8 df 03 00 00           call   8049104 <explode_bomb>
```

0x8（%esp)指向的地址是 push 给__isoc99_sscanf@plt 函数的倒数第二个参数的地址。**为了不引爆炸弹，终端输入的数字从右往左第一个数应该等于 5。**

其次，分析 fun4 函数，

```
520   8048d0c:   e8 57 ff ff ff          call   8048c68 <func4>     #调用<fun4>
521   8048d11:   83 c4 10                add    $0x10,%esp
522   8048d14:   83 f8 05                cmp    $0x5,%eax           #<fun4>返回5
523   8048d17:   75 07                   jne    8048d20 <phase_4+0x5a>
```

```
526   8048d20:   e8 df 03 00 00           call   8049104 <explode_bomb>
```

注意到，调用 fun4 函数之后，返回值不等于 5 时，指令会从第 523 行跳转到第 526 行，引爆炸弹，**所以调用 fun4 函数的返回值应该等于 5。**

本关闯关成功的核心在于递归调用函数 fun4。

而正确理解 fun4 的核心在于理解 ret，即汇编中实现回调的指令。

fun4 函数的指令解释如下：

```
457  08048c68 <func4>:
458   8048c68:   56                    push   %esi
459   8048c69:   53                    push   %ebx
460   8048c6a:   83 ec 04              sub    $0x4,%esp
461   8048c6d:   8b 4c 24 10           mov    0x10(%esp),%ecx
462   8048c71:   8b 5c 24 14           mov    0x14(%esp),%ebx
463   8048c75:   8b 74 24 18           mov    0x18(%esp),%esi
464   8048c79:   89 f0                 mov    %esi,%eax
465   8048c7b:   29 d8                 sub    %ebx,%eax
466   8048c7d:   89 c2                 mov    %eax,%edx
467   8048c7f:   c1 ea 1f              shr    $0x1f,%edx
468   8048c82:   01 d0                 add    %edx,%eax
469   8048c84:   d1 f8                 sar    $1,%eax
470   8048c86:   8d 14 18              lea    (%eax,%ebx,1),%edx
471   8048c89:   39 ca                 cmp    %ecx,%edx              #分支
472   8048c8b:   7e 15                 jle    8048ca2 <func4+0x3a>
473   8048c8d:   83 ec 04              sub    $0x4,%esp
474   8048c90:   83 ea 01              sub    $0x1,%edx
475   8048c93:   52                    push   %edx
476   8048c94:   53                    push   %ebx
477   8048c95:   51                    push   %ecx
478   8048c96:   e8 cd ff ff ff        call   8048c68 <func4>        #递归
```

第 457-470 行：预备栈空间，变量的赋值和运算；

第 471-472 行：如果 edx 中的值小于等于 ecx 中的值，那么跳转到第 482 行；

第 473-478 行：在 edx 中的值大于 ecx 中的值时，递归调用的参数压栈和 call 指令；

```
479   8048c9b:   83 c4 10              add    $0x10,%esp
480   8048c9e:   01 c0                 add    %eax,%eax
481   8048ca0:   eb 1e                 jmp    8048cc0 <func4+0x58>
482   8048ca2:   b8 00 00 00 00        mov    $0x0,%eax
483   8048ca7:   39 ca                 cmp    %ecx,%edx              #分支
484   8048ca9:   7d 15                 jge    8048cc0 <func4+0x58>
485   8048cab:   83 ec 04              sub    $0x4,%esp
486   8048cae:   56                    push   %esi
487   8048caf:   83 c2 01              add    $0x1,%edx
488   8048cb2:   52                    push   %edx
489   8048cb3:   51                    push   %ecx
490   8048cb4:   e8 af ff ff ff        call   8048c68 <func4>        #递归
491   8048cb9:   83 c4 10              add    $0x10,%esp
492   8048cbc:   8d 44 00 01           lea    0x1(%eax,%eax,1),%eax
493   8048cc0:   83 c4 04              add    $0x4,%esp
494   8048cc3:   5b                    pop    %ebx
495   8048cc4:   5e                    pop    %esi
496   8048cc5:   c3                    ret                           #回调
```

第 479 行-481 行：当上一个 call 指令成功后，栈保存 call 指令调用函数的返回地址，即 call 指令的下一条指令的地址。ret 指令执行时，恢复指令指针，eip 寄存器会被设置为这个地址。

第 482-484 行：另一个分支；

第 485-490 行：递归参数压栈和 call 指令；

第 491-492 行：回调执行的指令；

第 493-495 行：清理栈空间；

第 496 行：ret 指令回调。

ret 指令的作用：从栈（Stack）中弹出之前保存的 call 指令调用函数的返回地址，并将程序的控制权跳转到该地址继续执行。

func4()函数的 c 语言代码：

hnu > 大二课程 > 计算机系统 > 实验 > 参考 > lab2 > bomb7 > phase_4.cpp > ..

```
int func(int x,int y,int z){
    int res,tmp;
    res=(z-y);
    if(z>=y){
        res=res/2;
    }else{
        res=(res+1)/2;
    }
    tmp=res+y;
    if(tmp>x){
        z=tmp-1;
        res=func(x,y,z);
        res=2*res;
        return res;
    }
    else{
        res=0;
        if(tmp==x){
            return res;
        }else{
            y=tmp+1;
            res=func(x,y,z);
            return res*2+1;
        }
    }
}
```

为了使得函数返回 5，终端输入的第一个数为 10。递归回调过程如下：

func(10,0,14)，res=tmp=7,x=10,tmp<x,递归调用 func(10,8,14);

func(10,8,14),res=3,tmp=11,x=10,tmp>x，递归调用 func(10,8,10);

func(10,8,10),res=1,tmp=9,x=10,tmp<x,递归调用 func(10,10,10);

func(10,10,10),res=0,tmp=10,x=10,tmp=x,返回 res=0。

回调（重点）：

res=2*0+1 =1

res=2*1=2

res=2*2+1

算术执行逻辑分别对应压栈的 call 指令调用函数的返回地址的汇编代码的逻辑。

综上，本关答案为 10 5。

```
(base) xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
0 1 1 2 3 5
3 323
10 5
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
```

(5)<phase_5>

首先分析__isoc99_sscanf@plt 函数，和前面两关一样，这里不再赘述。

```
545   8048d5d:   e8 ae fa ff ff          call   8048810 <__isoc99_sscanf@plt>
546   8048d62:   83 c4 10                add    $0x10,%esp
547   8048d65:   83 f8 01                cmp    $0x1,%eax
548   8048d68:   7f 05                   jg     8048d6f <phase_5+0x34>
549   8048d6a:   e8 95 03 00 00          call   8049104 <explode_bomb>
```

```
(gdb) x/s 0x804a20f
0x804a20f:      "%d %d"
```

按正确格式解析字符串后，__isoc99_sscanf@plt 函数的返回值为 2，保存在 eax 中。eax 中的值大于 1，jg 跳转，避免引爆炸弹。按照要求，输入应为两个整数。不妨假设先入栈地址上存放的值为 A，后入栈地址上存放的值为 B。

<phase_5>只有两处调用了<explode_bomb>函数，故，接着分析<explode_bomb>函数。

```
568   8048dad:   e8 52 03 00 00          call   8049104 <explode_bomb>
```

<explode_bomb>函数的地址在 0x8048dad。

```
550   8048d6f:   8b 44 24 04             mov    0x4(%esp),%eax
551   8048d73:   83 e0 0f                and    $0xf,%eax
552   8048d76:   89 44 24 04             mov    %eax,0x4(%esp)
553   8048d7a:   83 f8 0f                cmp    $0xf,%eax
554   8048d7d:   74 2e                   je     8048dad <phase_5+0x72>
```

0x4(%esp)的地址是后入栈的地址。其上保存 B。上述代码将 B 从内存取出，与 0xf 按位与，结果保存在 eax 中，再将答案写回原内存。如果 eax 中的值等于 0xf，则跳转到<explode_bomb>函数的地址，而 eax 是 B 和 0xf 按位与的结果，**所以为了不引爆，炸弹 B 不等于 0xf。**更准确地说 B 的低四位不等于 0xffff。

```
564   8048da2:   83 fa 0f              cmp    $0xf,%edx             #edx等于0xf
565   8048da5:   75 06                 jne    8048dad <phase_5+0x72>
566   8048da7:   3b 4c 24 08           cmp    0x8(%esp),%ecx
567   8048dab:   74 05                 je     8048db2 <phase_5+0x77>
568   8048dad:   e8 52 03 00 00        call   8049104 <explode_bomb>
```

如果 edx 中的值不等于 0xf,则跳转到<explode_bomb>函数的地址，而 edx 是循环中的计数器。每次自增 1。**为了不引爆炸弹，循环次数为 15 次。**

0x4(%esp)的地址是先入栈的地址。其上保存 A。而 ecx 是循环相加的结果，**当循环相加的结果等于 A 时，则可以跳转，成功避免引爆炸弹。**

由此，得到关于循环的两个条件。

接下来，分析循环过程。

```
555   8048d7f:   b9 00 00 00 00        mov    $0x0,%ecx
556   8048d84:   ba 00 00 00 00        mov    $0x0,%edx
557   8048d89:   83 c2 01              add    $0x1,%edx               #累加
558   8048d8c:   8b 04 85 c0 a0 04 08  mov    0x804a0c0(,%eax,4),%eax
559   8048d93:   01 c1                 add    %eax,%ecx
560   8048d95:   83 f8 0f              cmp    $0xf,%eax
561   8048d98:   75 ef                 jne    8048d89 <phase_5+0x4e> #循环结束
```

第 555-556 行：初始化。ecx，edx 初始化为 0；

第 557 行：计数器+1；

第 558 行：取出内存中的值。由前面的分析可知，eax 保存 B（在 B 小于 15 的情况下）。而这里偏移地址等于 B*4，并且每次更新 eax 为前一次内存地址上的值；

第 559 行：把取出的值累加到 ecx；

第 560-561 行：循环结束条件，最后一次更新的 eax 值等于 0xf，不执行 jne，不跳转到第 557 行，循环结束。

为了不引爆炸弹且正确退出循环，要在循环 15 次时，ecx 累加的结果等于 A，最后一次更新 eax 的值为 0xf。查看基地址 0x804a0c0 后 64 个字节的地址中的值：

```
(gdb) x/16wd 0x804a0c0
0x804a0c0 <array.3249>: 10      2       14      7
0x804a0d0 <array.3249+16>:      8       12      15      11
0x804a0e0 <array.3249+32>:      0       4       1       13
0x804a0f0 <array.3249+48>:      3       9       6       5
```

B=5 时，即 eax 初始值为 5。

偏移地址	20	48	12	28	44	52	36	16	32	0	40	4	8	56	24
内存中值	12	3	7	11	13	9	4	8	0	10	1	2	14	6	15

刚好符合条件，计算累加结果为 115。

验证：

```
(base) xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
0 1 1 2 3 5
3 323
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
10 5
So you got that one.  Try this one.
5 115
Good work!  On to the next...
```

(6)<phase_6>

按顺序分析，

首先分析第一个函数<read_six_numbers>，

```
576  08048dc8 <phase_6>:
577   8048dc8:   56                    push   %esi
578   8048dc9:   53                    push   %ebx
579   8048dca:   83 ec 4c              sub    $0x4c,%esp
580   8048dcd:   65 a1 14 00 00 00     mov    %gs:0x14,%eax
581   8048dd3:   89 44 24 44           mov    %eax,0x44(%esp)
582   8048dd7:   31 c0                 xor    %eax,%eax
583   8048dd9:   8d 44 24 14           lea    0x14(%esp),%eax
584   8048ddd:   50                    push   %eax
585   8048dde:   ff 74 24 5c           push   0x5c(%esp)
586   8048de2:   e8 42 03 00 00        call   8049129 <read_six_numbers>
```

读入六个数字，%eax 中存放的是读入数据的首地址，压栈，作为参数传入即将调用的函数；

其次，分析数据的检验部分，核心是双层循环；

```
587   8048de7:   83 c4 10              add    $0x10,%esp
588   8048dea:   be 00 00 00 00        mov    $0x0,%esi
589   8048def:   8b 44 b4 0c           mov    0xc(%esp,%esi,4),%eax
590   8048df3:   83 e8 01              sub    $0x1,%eax
591   8048df6:   83 f8 05              cmp    $0x5,%eax
592   8048df9:   76 05                 jbe    8048e00 <phase_6+0x38>
593   8048dfb:   e8 04 03 00 00        call   8049104 <explode_bomb>
594   8048e00:   83 c6 01              add    $0x1,%esi
595   8048e03:   83 fe 06              cmp    $0x6,%esi
596   8048e06:   74 1b                 je     8048e23 <phase_6+0x5b>     #
597   8048e08:   89 f3                 mov    %esi,%ebx
598   8048e0a:   8b 44 9c 0c           mov    0xc(%esp,%ebx,4),%eax
599   8048e0e:   39 44 b4 08           cmp    %eax,0x8(%esp,%esi,4)
600   8048e12:   75 05                 jne    8048e19 <phase_6+0x51>
601   8048e14:   e8 eb 02 00 00        call   8049104 <explode_bomb>
```

```
602   8048e19:   83 c3 01              add    $0x1,%ebx
603   8048e1c:   83 fb 05              cmp    $0x5,%ebx
604   8048e1f:   7e e9                 jle    8048e0a <phase_6+0x42>
605   8048e21:   eb cc                 jmp    8048def <phase_6+0x27>  #
```

第 587 行：移动栈顶指针，为找到读入数据的首地址作准备；

第 588 行：初始化 esi 为 0；

第 589-592 行：第一层循环。首先通过比例变址基址寻址得到读入数据的首地址，从中读取数据，将其减 1，如果小于等于 5 则不引爆炸弹，跳过它执行。

第 594-596 行：第一次循环的计数器。esi 递增 1，当 esi 等于 6 时，跳出循环；

第 597-605 行：第二层循环。ebx 储存变址。将其初始化为 esi。每次循环 ebx+1，一直加到到。将由 ebx 变址乘以比例因子 4 加上基地址加上偏移量得到的内存地址的值与由 esi 变址乘以比例因子 4 加上基地址加上偏移量得到的内存地址的值比较，只要不相等，就跳过炸弹执行。**实际上是验证 x1,x2,x3,x4,x5,x6(x5!=6,x4!=x6,・・・,x1!=x6)。结合外层循环，就能保证没有重复的数字。**

接着，分析第 606 行-614 行：前三行是初始化，eax 中保存读入数据的起始位置，ebx 保存读入数据的末尾位置向上偏移 4 个字节的位置。由 cmp 和 jne 指令可知，当 eax 中保存的值和 ebx 中的值相等时，跳出循环。

```
606   8048e23:   8d 44 24 0c          lea    0xc(%esp),%eax
607   8048e27:   8d 5c 24 24          lea    0x24(%esp),%ebx
608   8048e2b:   b9 07 00 00 00       mov    $0x7,%ecx
609   8048e30:   89 ca                mov    %ecx,%edx
610   8048e32:   2b 10                sub    (%eax),%edx
611   8048e34:   89 10                mov    %edx,(%eax)
612   8048e36:   83 c0 04             add    $0x4,%eax
613   8048e39:   39 c3                cmp    %eax,%ebx
614   8048e3b:   75 f3                jne    8048e30 <phase_6+0x68>
```

最开始，我粗浅地认为只有循环的结果重要，中间的部分不重要。导致最后验证结果的时候出现了很大的问题。那循环过程的分析如下：

第 608-611 行：设 x 为输入数据，将输入的数据变成 7-x;**这就导致最终得到的节点下标是 7-输入数据的结果。**

第 612 行：循环递增；

第 613-614 行：循环退出条件。

其次，分析排序（按照输入数据的顺序给节点排序）。

在分析之前，首先分析节点 node

```
(gdb) x/3x 0x804c13c
0x804c13c <node1>:      0x000002d9      0x00000001      0x0804c148
(gdb) x/3x 0x804c148
0x804c148 <node2>:      0x0000023b      0x00000002      0x0804c154
(gdb) x/3x 0x0804c154
0x804c154 <node3>:      0x0000009c      0x00000003      0x0804c160
(gdb) x/3x 0x0804c160
0x804c160 <node4>:      0x00000314      0x00000004      0x0804c16c
(gdb) x/3x 0x0804c16c
0x804c16c <node5>:      0x0000015d      0x00000005      0x0804c178
(gdb) x/3x 0x0804c178
0x804c178 <node6>:      0x00000275      0x00000006      0x00000000
```

首先，注意到 0x804c13c 放入到 edx 中，每次循环 edx 的值加 8，作为内存地址。结合汇编结果，这个地址是存储下一个节点的地址。**由此分析，每个节点占 12 个字节。第一部分是数据域，第二部分是节点编号，第三个部分是 next 域（存储下一个节点的地址）。**

```
615   8048e3d:   bb 00 00 00 00          mov    $0x0,%ebx
616   8048e42:   eb 16                   jmp    8048e5a <phase_6+0x92>
617   8048e44:   8b 52 08                mov    0x8(%edx),%edx
618   8048e47:   83 c0 01                add    $0x1,%eax
619   8048e4a:   39 c8                   cmp    %ecx,%eax
620   8048e4c:   75 f6                   jne    8048e44 <phase_6+0x7c>
621   8048e4e:   89 54 b4 24             mov    %edx,0x24(%esp,%esi,4)
622   8048e52:   83 c3 01                add    $0x1,%ebx
623   8048e55:   83 fb 06                cmp    $0x6,%ebx
624   8048e58:   74 17                   je     8048e71 <phase_6+0xa9>
625   8048e5a:   89 de                   mov    %ebx,%esi
626   8048e5c:   8b 4c 9c 0c             mov    0xc(%esp,%ebx,4),%ecx
627   8048e60:   b8 01 00 00 00          mov    $0x1,%eax
628   8048e65:   ba 3c c1 04 08          mov    $0x804c13c,%edx
629   8048e6a:   83 f9 01                cmp    $0x1,%ecx
630   8048e6d:   7f d5                   jg     8048e44 <phase_6+0x7c>
631   8048e6f:   eb dd                   jmp    8048e4e <phase_6+0x86>
```

第 615-616 行：初始化 ebx=0，并跳到第 625 行。

第 625-628 行：寄存器值的更新。我们发现，ebx 和 esi 中存储变址。ecx 保存读入的数据。eax 作为计数器，也指明读取的是第几个数。edx 从内存中读取节点。

第 629-631 行：**if-else。**如果读取的数据大于 1 则跳转到第 617 行执行；如果读取的数据小于等于 1 则跳转到第 621 行执行。

重点分析，if-else 内部的逻辑。

（1）第 617-620 行，edx 取值地址每次递增 8 个字节，eax 的值加 1，一直加到和读入数据相等为止。edx 的值也随之一次次更新。

（2）第 621-624 行，把 edx 的值保存到内存。该地址恰好是读入第六个数据往上递增的地址。更新变址 ebx+1，当 ebx 等于 6 时，跳出循环。【这一段，当 if 段执行完之后，也会执行】

相关 c 代码：

```
3  //存序号的数组设为a,存结点的数组设为b
4  for(int i=0;i<6;i++){
5      if(a[i]>1){
6          for(int j=1;j<a[i];j++){
7              head=head->next;
8          }
9          b[i]=&head;//分析节点发现，每次偏移8字节，得到的是节点的指针域。
10     }
11 }
```

接着分析链表重组

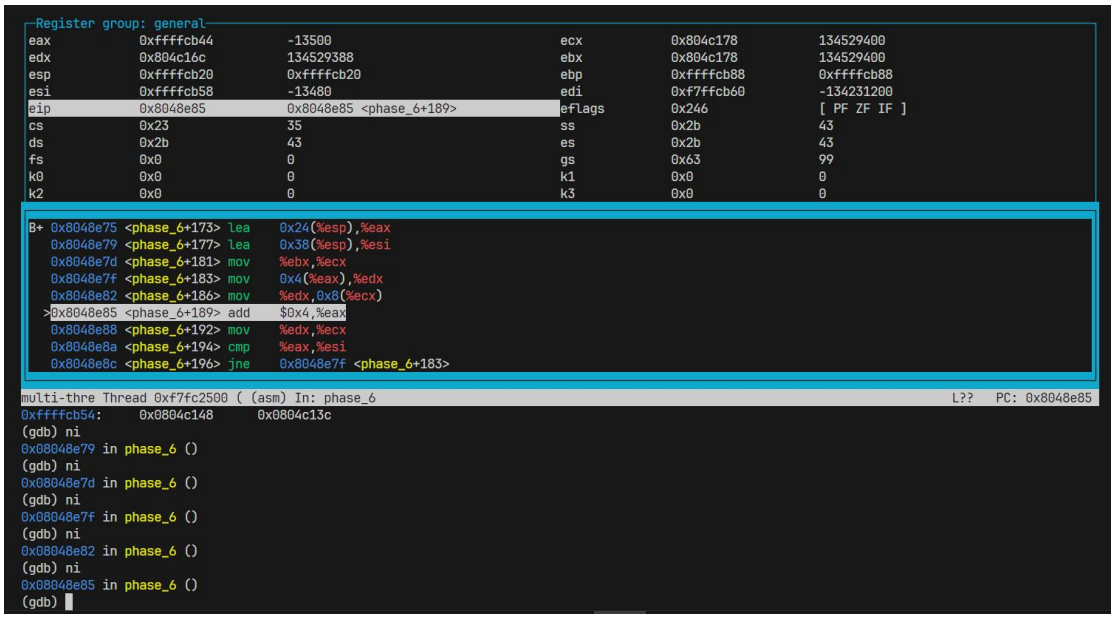
```
632   8048e71:   8b 5c 24 24          mov    0x24(%esp),%ebx
633   8048e75:   8d 44 24 24          lea    0x24(%esp),%eax
634   8048e79:   8d 74 24 38          lea    0x38(%esp),%esi
635   8048e7d:   89 d9                mov    %ebx,%ecx
636   8048e7f:   8b 50 04             mov    0x4(%eax),%edx
637   8048e82:   89 51 08             mov    %edx,0x8(%ecx)
638   8048e85:   83 c0 04             add    $0x4,%eax
639   8048e88:   89 d1                mov    %edx,%ecx
640   8048e8a:   39 c6                cmp    %eax,%esi
641   8048e8c:   75 f1                jne    8048e7f <phase_6+0xb7>
```

```
Register group: general
eax            0xffffcb44         -13500          ecx            0x1            1
edx            0x804c13c          134529340       ebx            0x804c178      134529400
esp            0xffffcb20         0xffffcb20      ebp            0xffffcb88     0xffffcb88
esi            0xffffcb58         -13480          edi            0xf7ffcb60     -134231200
eip            0x8048e7d          0x8048e7d <phase_6+181>  eflags   0x246          [ PF ZF IF ]
cs             0x23               35              ss             0x2b           43
ds             0x2b               43              es             0x2b           43
fs             0x0                0               gs             0x63           99
k0             0x0                0               k1             0x0            0
k2             0x0                0               k3             0x0            0

B+ 0x8048e75 <phase_6+173> lea    0x24(%esp),%eax
   0x8048e79 <phase_6+177> lea    0x38(%esp),%esi
  >0x8048e7d <phase_6+181> mov    %ebx,%ecx
   0x8048e7f <phase_6+183> mov    0x4(%eax),%edx
   0x8048e82 <phase_6+186> mov    %edx,0x8(%ecx)
   0x8048e85 <phase_6+189> add    $0x4,%eax
   0x8048e88 <phase_6+192> mov    %edx,%ecx
   0x8048e8a <phase_6+194> cmp    %eax,%esi
   0x8048e8c <phase_6+196> jne    0x8048e7f <phase_6+183>

multi-thre Thread 0xf7fc2500 ( (asm) In: phase_6                          L??   PC: 0x8048e7d
(gdb) x/6wx 0xffffcb20+0x24
0xffffcb44:     0x0804c178      0x0804c16c      0x0804c160      0x0804c154
0xffffcb54:     0x0804c148      0x0804c13c
(gdb) ni
0x08048e79 in phase_6 ()
(gdb) ni
0x08048e7d in phase_6 ()
(gdb)
```

这段调试：eax 存储地址上的值是链表的 next 域。eax 初始化为首地址，esi 初始化为尾地址。



这段调试：对节点的 next 域进行重写。按照之前代码的顺序，把节点的下一地址赋值给节点的 next 域。

之后检验数据域的递增

```
642   8048e8e:   c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
643   8048e95:   be 05 00 00 00          mov    $0x5,%esi
644   8048e9a:   8b 43 08                mov    0x8(%ebx),%eax
645   8048e9d:   8b 00                   mov    (%eax),%eax
646   8048e9f:   39 03                   cmp    %eax,(%ebx)
647   8048ea1:   7d 05                   jge    8048ea8 <phase_6+0xe0>
648   8048ea3:   e8 5c 02 00 00          call   8049104 <explode_bomb>
649   8048ea8:   8b 5b 08                mov    0x8(%ebx),%ebx
650   8048eab:   83 ee 01                sub    $0x1,%esi
651   8048eae:   75 ea                   jne    8048e9a <phase_6+0xd2>
```

esi 是控制循环的变量，比较前面的节点的数据域和后面节点的数据域的大小，**如果大于或等于则不引爆炸弹。遍历完列表，退出循环。**

相应 c 语言代码：

```
struct node *head;
for(int i=5;i>0;i--){
    if(head->next->data<head->data){
        bomb();
    }
}
```

分析结果，为了保证节点的数据域递减，我们输入的节点编号顺序应该为 4 1 6 2 5 3，**经过 7-x 映射之后，正确结果为 3 6 1 5 2 4。**

```
0x804c13c <node1>:      729
(gdb) x/1wd 0x0804c148
0x804c148 <node2>:      571
(gdb) x/1wd 0x0804c154
0x804c154 <node3>:      156
(gdb) x/1wd 0x0804c160
0x804c160 <node4>:      788
(gdb) x/12d 0x0804c16c
0x804c16c <node5>:      349
```

```
(gdb) x/1wd 0x0804c178
0x804c178 <node6>:      629
```

验证：

hnu > 大二课程 > 计算机系统 > 实验 > 参考 > lab2 > bomb7 > ans.txt

```
1  He is evil and fits easily into most overhead storage bins.
2  0 1 1 2 3 5
3  3 323
4  10 5
5  5 115
6  3 6 1 5 2 4
7
```

问题 2　输出　调试控制台　终端　端口

```
(base) xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./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...
Congratulations! You've defused the bomb!
(base) xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$
```

(7)<secret_phase>

首先寻找隐藏关入口。在 bomb.c 文件中，每一关都会调用 phase_defused()。推测隐藏关和 phase_defused()有关。

分析 phase_defused()代码：

```
sub    $0x6c,%esp
mov    %gs:0x14,%eax          #%gs:0x14 存储线程局部变量、线程或进程控制块等
mov    %eax,0x5c(%esp)
xor    %eax,%eax
cmpl   $0x6,0x804c3cc         #当前六关都通关，则进入隐藏关，否则直接退出
jne    80492e8 <phase_defused+0x8b>
sub    $0xc,%esp
lea    0x18(%esp),%eax
push   %eax
lea    0x18(%esp),%eax
push   %eax
lea    0x18(%esp),%eax
push   %eax
push   $0x804a269             #查看发现格式字符串的方式为"%d %d %s"
push   $0x804c4d0
call   8048810 <__isoc99_sscanf@plt>  #第四关都出现了相同的调用
add    $0x20,%esp
cmp    $0x3,%eax              #如果只解析两个整数，则跳过隐藏关
jne    80492d8 <phase_defused+0x7b>
```

由上图代码可知，进入隐藏关，第四关需要解析两个整数和一个字符串；但通过四关，只需要两个整数。**可以推测输入的整数后面应加入一个正确的字符串**，使得它在四关解析时可以有两个整数，在隐藏关又可以解析为两个整数+一个字符串。

```
sub    $0x8,%esp
push   $0x804a272
lea    0x18(%esp),%eax
push   %eax
call   804900d <strings_not_equal>    #通关的字符串要正确
add    $0x10,%esp
test   %eax,%eax              #<strings_not_equal>返回1，则跳过隐藏关
jne    80492d8 <phase_defused+0x7b>
```

查看要添加的字符串

```
(gdb) x/s 0x804a272
0x804a272:      "DrEvil"
```

验证答案：

```
hnu > 大二课程 > 计算机系统 > 实验 > 参考 > lab2 > bomb7 > ≡ ans.txt
1  He is evil and fits easily into most overhead storage bins.
2  0 1 1 2 3 5
3  3 323
4  10 5DrEvil
5  5 115
6  3 6 1 5 2 4
7

问题 3   输出   调试控制台   终端   端口

(base) xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./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...
```

接下来，分析隐藏关：

分析第一个<explode_bomb>：

```
705  8048f37:   3d e8 03 00 00          cmp    $0x3e8,%eax
706  8048f3c:   76 05                   jbe    8048f43 <secret_phase+0x2a>
707  8048f3e:   e8 c1 01 00 00          call   8049104 <explode_bomb>
```

eax 中的值小于等于 0x3e8，则成功跳过炸弹。而 eax 中值来自读入数据解析后的转换。

```
694  8048f19:   53                      push   %ebx
695  8048f1a:   83 ec 08                sub    $0x8,%esp
696  8048f1d:   e8 42 02 00 00          call   8049164 <read_line>
697  8048f22:   83 ec 04                sub    $0x4,%esp
698  8048f25:   6a 0a                   push   $0xa
699  8048f27:   6a 00                   push   $0x0
700  8048f29:   50                      push   %eax
701  8048f2a:   e8 51 f9 ff ff          call   8048880 <strtol@plt>
702  8048f2f:   89 c3                   mov    %eax,%ebx
703  8048f31:   8d 40 ff                lea    -0x1(%eax),%eax
```

第 694-696 行：从终端读入字符串；

第 698-700 行：参数准备；long strtol(const char *nptr, char **endptr, int base);从右往左的第一个参数是 0xa，表示按十进制转换，第二个参数是转换后字符串的首地址；第三个参数是待转换的字符串的首地址。

第 701 行：调用<strtol@plt>函数，函数返回转换后的值，保存在 eax 中；

第 702-703 行：变量赋值，把 eax 的值保存在 ebx 中，eax 中的值减一。

综上，输入的字符串解析后的整数<=1000(十进制）。

分析第二个<explode_bomb>：

```
713   8048f54:   83 f8 05              cmp    $0x5,%eax
714   8048f57:   74 05                 je     8048f5e <secret_phase+0x45>
715   8048f59:   e8 a6 01 00 00        call   8049104 <explode_bomb>
```

eax 中的值等于 5，则成功跳过炸弹。

```
708   8048f43:   83 ec 08              sub    $0x8,%esp
709   8048f46:   53                    push   %ebx
710   8048f47:   68 88 c0 04 08        push   $0x804c088
711   8048f4c:   e8 77 ff ff ff        call   8048ec8 <fun7>
712   8048f51:   83 c4 10              add    $0x10,%esp
713   8048f54:   83 f8 05              cmp    $0x5,%eax
714   8048f57:   74 05                 je     8048f5e <secret_phase+0x45>
```

那么，fun7 的返回值应该为 5。由上述代码知，fun7 的第二个参数是 ebx（保存解析出的整数值。第一个参数推测应该是某地址。

接下来分析，fun7。

```
661   08048ec8 <fun7>:
662    8048ec8:   53                    push   %ebx
663    8048ec9:   83 ec 08              sub    $0x8,%esp
664    8048ecc:   8b 54 24 10           mov    0x10(%esp),%edx
665    8048ed0:   8b 4c 24 14           mov    0x14(%esp),%ecx
666    8048ed4:   85 d2                 test   %edx,%edx
667    8048ed6:   74 37                 je     8048f0f <fun7+0x47>    #内存地址为0，跳出
668    8048ed8:   8b 1a                 mov    (%edx),%ebx
669    8048eda:   39 cb                 cmp    %ecx,%ebx              #条件分支
670    8048edc:   7e 13                 jle    8048ef1 <fun7+0x29>
671    8048ede:   83 ec 08              sub    $0x8,%esp
672    8048ee1:   51                    push   %ecx
673    8048ee2:   ff 72 04              push   0x4(%edx)
674    8048ee5:   e8 de ff ff ff        call   8048ec8 <fun7>         #递归调用res=fun7(address+1,input)
675    8048eea:   83 c4 10              add    $0x10,%esp
676    8048eed:   01 c0                 add    %eax,%eax              #res*2
677    8048eef:   eb 23                 jmp    8048f14 <fun7+0x4c>
678    8048ef1:   b8 00 00 00 00        mov    $0x0,%eax              #res=0
679    8048ef6:   39 cb                 cmp    %ecx,%ebx              #递归终止 num[address+1]==input
680    8048ef8:   74 1a                 je     8048f14 <fun7+0x4c>
681    8048efa:   83 ec 08              sub    $0x8,%esp
682    8048efd:   51                    push   %ecx
683    8048efe:   ff 72 08              push   0x8(%edx)
684    8048f01:   e8 c2 ff ff ff        call   8048ec8 <fun7>          #递归调用res=fun7(address+2,input)
685    8048f06:   83 c4 10              add    $0x10,%esp
686    8048f09:   8d 44 00 01           lea    0x1(%eax,%eax,1),%eax    #res*2+1
687    8048f0d:   eb 05                 jmp    8048f14 <fun7+0x4c>
688    8048f0f:   b8 ff ff ff ff        mov    $0xffffffff,%eax
689    8048f14:   83 c4 08              add    $0x8,%esp
690    8048f17:   5b                    pop    %ebx
691    8048f18:   c3                    ret
```

递归调用的逻辑：结合上述注释，可以将其翻译成 c 代码：

```
3   int fun7(int index,int x){
4       int res;
5       if(index==0){
6           exit(1);
7       }
8       if(num[index]>=x){
9           res=0;
10          if(num[index]=x){
11              return res;
12          }
13          else{
14              res=fun7(index+2,x);
15              return 2*res+1;       //右孩子
16          }
17      }
18      else{
19          res=fun7(index+1,x);
20          return 2*res;             //左孩子
21      }
22  }
23
```

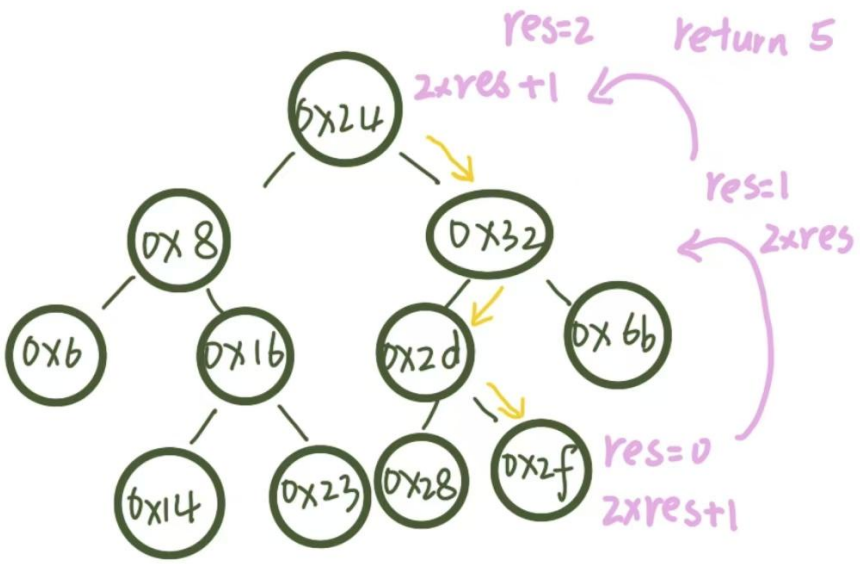
对传入的地址进行查看，

```
(gdb) x/3x 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0
(gdb) x/3x 0x0804c094
0x804c094 <n21>:        0x00000008      0x0804c0c4      0x0804c0ac
(gdb) x/3x 0x0804c0a0
0x804c0a0 <n22>:        0x00000032      0x0804c0b8      0x0804c0d0
(gdb) x/3x 0x0804c0b8
0x804c0b8 <n33>:        0x0000002d      0x0804c0dc      0x0804c124
(gdb)
```

会发现这和第六关类似。**这里的首地址是根节点的地址。地址上存储的是根节点的值，地址+4，是左孩子的地址，地址+8，是右孩子的地址。结合上述 c 代码可知，根节点地址为 0，退出函数；如果根节点的值小于 input，找左孩子；如果相等则找到；如果根节点的值大于 input，找右孩子。这实际是一棵二叉检索树。**

那么为了使得结果输出 5，我们需要关注找左孩子时，返回 2*res；找右孩子时，返回 2*res+1。先找右孩子，再找左孩子，再找右孩子。分析如下：

```
(gdb) x/3x 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0
(gdb) x/3x 0x804c094
0x804c094 <n21>:        0x00000008      0x0804c0c4      0x0804c0ac
(gdb) x/3x 0x804c0a0
0x804c0a0 <n22>:        0x00000032      0x0804c0b8      0x0804c0d0
(gdb) x/3x 0x804c0ac
0x804c0ac <n32>:        0x00000016      0x0804c118      0x0804c100
(gdb) x/3x 0x804c0ac
0x804c0ac <n32>:        0x00000016      0x0804c118      0x0804c100
(gdb)
0x804c0b8 <n33>:        0x0000002d      0x0804c0dc      0x0804c124
(gdb) x/3x 0x804c0b8
0x804c0b8 <n33>:        0x0000002d      0x0804c0dc      0x0804c124
(gdb) x/3x 0x804c0dc
0x804c0dc <n45>:        0x00000028      0x00000000      0x00000000
(gdb) x/3x 0x804c124
0x804c124 <n46>:        0x0000002f      0x00000000      0x00000000
(gdb)
```



验证：


hnu > 大二课程 > 计算机系统 > 实验 > 参考 > lab2 > bomb7 > ans.txt
1 He is evil and fits easily into most overhead storage bins.
2 0 1 1 2 3 5
3 3 323
4 10 5DrEvil
5 5 115
6 3 6 1 5 2 4
7
问题 7
输出
调试控制台
终端
端口
(base) xiaoye@长乐:/mnt/d/hnu/大二课程/计算机系统/实验/参考/lab2/bomb7$ ./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...
47
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!

4 总结

问题：

（1）一开始不习惯用 gdb，会下意识手算；

（2）汇编代码不习惯，一开始不太看得明白。尤其是看递归的时候，看了很久才弄清逻辑；

（3）对于非数组的结构的模拟，链表、二叉树，很有意思，但一开始很难发现。

总结：

1. 汇编和 c 的映射：

- 汇编代码中的 lea,add 操作==c 语言中算术运算；
- 汇编代码中的 shr,sar 操作==c 语言中位移运算符；
- 汇编代码中 cmp，jle，jge 等指令==c 语言中条件语句、控制循环；
- 递归：call 地址<函数> 后面对 eax 的指令进行操作==int func（）{ans=func（）;return ans+2;}；ret 执行递归的回调
- eax 一般都是存储结果，ebx，edx，ecx 是几个变量存储的地方，一般用栈作为内存，可以模拟数组、链表、二叉树等数据结构，可以根据内存中的保存的数据的规律结合代码，去判断它是在模拟什么数据结构。一般链表一个节点至少 8 位，保存数据和指针；一颗树的节点至少 12 位，保存数据和左右孩子的地址。汇编通常通过不同的地址偏移量去实现取数据或者取地址的操作。

2. 堆栈的变化

（1）数据的压栈和入栈；

（2）函数参数的压栈和入栈，从右往左压栈；

（3）函数调用过程中，函数返回地址压栈入栈。

调用函数时，当从 phase_4 调用 fun_4 时，会执行 call 指令。call 指令会将当前的指令指针 eip 压入堆栈（eip 是指向当前正在执行的指令的地址。在函数调用过程中，eip 会指向 call 指令的下一条指令），以便函数返回时能够跳回原来的位置。

然后 esp 会指向新的栈顶，即 phase_4 的返回地址。

3. gdb 断点调试，objdump 反汇编

x/s <地址>:以字符串的格式查看内存内容 ni：单步执行下一条指令，不进入函数。

si：单步执行下一条指令，会进入函数内部。

设置断点在函数入口处，然后运行。可以使得函数在入口的第一个断点停下。之后就可以单步运行。【这一条很好用。可以在繁复的代码快速定位到需要调试的部分】

disassemble：显示当前函数的汇编代码。next： 会跳过函数调用，直接执行到函数返回。

nexti ：会逐条执行汇编指令，跳过函数调用。

objdump -d bomb >1.txt 由二进制文件生成汇编文件

x/4bt 0x23456789：

查看内存内容 4 内存单元有 4 个，

b-单字节，h-双字节，w-四字节，g-八字节

x 按照十六进制格式显示变量，d 按照十进制格式显示变量，u 按照十六进制格式显示无符号整型，o 按照八进制格式显示变量，t 二进制格式，a 十六进制格式，c 字符格式。f 浮点数格式

0x23456789 这是要查看的内存地址。

4.指令和 c 标准库函数的变体。

（1)__isoc99_sscanf@plt 是 C 标准库函数 sscanf 的一个变体，通常在编译器支持 ISO C99 标准时使用。

参数：输入字符串；格式字符串；

输出参数的地址__isoc99_sscanf 返回成功解析的参数数量。如果成功解析了两个整数，那么返回值为 2.

(2)call strtol@plt

这表示调用 strtol，但实际上会跳转到 PLT 入口，再由 PLT 跳转到实际的 strtol 实现。

<strtol@plt> 代表的是 strtol 函数在 PLT（Procedure Linkage Table，过程链接表） 中的入口。strtol 是一个标准的 C 库函数，它用于将字符串转换为 long 型整数。long strtol(const char *nptr, char **endptr, int base);

各种指令的知识不再赘述。在分析过程，不要混淆是移动地址还是地址中的值。mov 指令是将地址中的值移动；lea 才是移动地址。

感受：

对学习汇编和使用 gdb 的帮助很大，解题过程也很有意思。