Bomb lab实验

湖南大学信息科学与工程学院

计科2102 梅炳寅 （学号202108010206）

目录

[前言 2](#_Toc13889)

[<Phase\_1>字符串比较 3](#_Toc17387)

[<Phase\_2>循环 3](#_Toc1502)

[<Phase\_3>分支语句与跳转表 4](#_Toc20197)

[<Phase\_4>递归调用 7](#_Toc6177)

[<func\_4>递归函数体分析 8](#_Toc7334)

[<phase\_5>字符串末尾累加 10](#_Toc18133)

[<phase\_6>链表 12](#_Toc22464)

[<secret\_phase>二叉检索树（BinarySearchTree） 16](#_Toc4749)

[<phase\_defused>寻找隐藏关入口 16](#_Toc1452)

[<secret\_phase>隐藏关 18](#_Toc15655)

[<fun7>检索目标元素 19](#_Toc12769)

[深度拓展与思考： 19](#_Toc22824)

[进一步思考树结构的实现： 20](#_Toc3619)

[<撒花完结> 21](#_Toc32048)

[附录：自己写的各部分phase的c代码 22](#_Toc6847)

[void phase\_1() 22](#_Toc986)

[void phase\_2() 22](#_Toc15712)

[void phase\_3() 23](#_Toc17342)

[int func4(int n, int x) 24](#_Toc9094)

[void phase\_4() 24](#_Toc18549)

[void phase\_6() 25](#_Toc30207)

[int fun7(int \*x, int input) 26](#_Toc7376)

[void secret\_phase() 26](#_Toc19653)

**前言**

bomb实验，感觉十分有趣。它非常考验对于汇编代码的理解，很考验读汇编代码的能力与写汇编代码的能力。这个实验十分推荐自己独立完成，可以独立完成之后再去CSDN上校对自己的思路与寻求更新的思路，会有一种茅塞顿开的领悟之感。下面总结了几点感悟。

1、总的来说，给的汇编代码有很多，甚至有很多汇编代码段。但对于解题来说，不是每一个汇编代码段我们都要细看（当然如果是为了加深对于汇编的理解，也可以认真研究）。比如说一些把名字能够很好表示含义的函数，如phase1中的<strings\_not\_equal>函数和phase2中的<read\_six\_numbers>函数等，光是看名字就可以比较有效地理解其意义，就没必要细看了（当然有时还是看几眼防止逻辑上正反的问题）。

2、单是针对解题而言，一个方法是先关注<explode\_bomb>段。既然我们的目标是不要爆炸，那么我可以在所有引向爆炸的路上避免，即可。

3、题目中往往藏着提示，根据这些提示可以很快找到方向，比如phase1根据strings可以马上发现我们要研究的对象是字符串，而phase2根据six numbers可以很快发现我们要研究的是6个数字。这样一来我们就可以很快地找到思路，避免对着冗长的汇编代码发呆。

4、感到纸笔演算比较困难？不妨借助强大的gdb功能进行推算。

5、熟悉循环结构，熟悉循环控制变量，熟悉函数调用时的堆栈空间变化，这将为解题带来很大帮助。

6、熟悉一些常见的数据结构的汇编级实现，如二叉检索树、链表等，这是必要的基础理解。

7、对于递归函数，弄清楚其某一层的意思，总结出递归表达式，写出c代码，这样可以加深我们的理解。

8、有条件应对每一段汇编代码写出相对应的c代码。

**<Phase\_1>字符串比较**

（1）首先关注导致爆炸的关键代码段

 8048b77:   85 c0                   test   %eax,%eax

 8048b79:   74 05                   je     8048b80 <phase\_1+0x20>

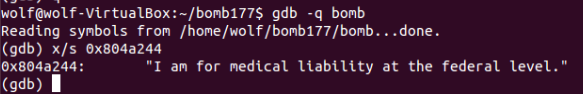
 8048b7b:   e8 f6 05 00 00          call   8049176 <explode\_bomb>

不难发现，导致爆炸的原因是没有je，如果je了，就会跳转到安全的地方。若%eax值为0，test将其相与，若结果为0，则成功跳转。这个%eax是<strings\_not\_equal>函数的返回值，故其实我们要比较两个字符串是否相等，其中一个应该是我们输入的，另外一个是题目保存在某个地址的。我们的目标就是找到题目把这个答案保存在哪里。

既然这个函数是比较两个字符串的相等与否，那必然会把两个字符串所在地址传到函数中。关注到前面的这句话

 8048b63:   c7 44 24 04 44 a2 04    movl   $0x804a244,0x4(%esp)

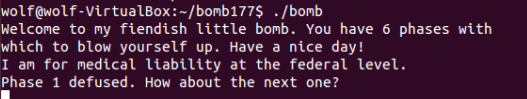
发现将一个小可爱放到了0x4(%esp)，如果比较敏感的话，这里恰好是调用函数传参的地方。那就没得跑了，这个0x804a244如无意外就应该是答案所在的地址。我们以s类型查看该地址，发现确实就是答案。



至此我们得到phase1的答案

I am for medical liability at the federal level.

我们输入答案，验证通过。



**<Phase\_2>循环**

这道题一开始我花了太多时间去研究<read\_six\_numbers>函数体了，后来发现其实不是很有这个必要。包括还犯了一个错误就是没有借助Linux强大的gdb功能进行推算，试图纯用纸笔和人脑进行计算，这真的是愚蠢至极。

这道题的提示实际上就藏在<read\_six\_numbers>这个函数的名字中，顾名思义，就是我们输入6个数字，程序对这6个数字进行验证。那我们的目标就是找出这6个数字。再结合之前总结的“找到什么导致爆炸”，我们很容易关注到这个代码段。

 8048b9d:   83 7c 24 18 00          cmpl   $0x0,0x18(%esp)

 8048ba2:   75 07                   jne    8048bab <phase\_2+0x27>

 8048ba4:   83 7c 24 1c 01          cmpl   $0x1,0x1c(%esp)

 8048ba9:   74 05                   je     8048bb0 <phase\_2+0x2c>

 8048bab:   e8 c6 05 00 00          call   8049176 <explode\_bomb>

M[0x18(%esp)]在和0比较，如果不相等，会导致跳转并爆炸，那0x18(%esp)肯定为0；

M[0x1c(%esp)]在和1比较，如果相等，就会跳转到爆炸后的安全代码段，否则爆炸，所以0x1c(%esp)为1；

那么既然用来作为比较，肯定这两个值都是我们输入的关键值，假设我们输入的数值为数组m的数。我们猜测M[0x18(%esp)]和M[0x1c(%esp)]为m[1]和m[2]（即我们输入的第一个和第二个数值），即我们尝试输入0,1,2,3,4,5



发现我们在 0x8048bab这个点没有爆炸，那么我们的猜测是正确的，前两个数分别是0,1。

然后关注到这个jne指令，如果不满足%esi里的值等于%ebx里的值，那么会回到mov指令的地方。这说明了两点。第一，这里有一个循环；第二，mov前的两个lea应该进行了一个初始化。由前面M[0x18(%esp)]和M[0x1c(%esp)]为m[1]和m[2]，这里M[0x20(%esp)]应该是m[3]（即第三个数），mov    -0x8(%ebx),%eax和add    -0x4(%ebx),%eax这两步很好理解，就是在把前两个进行累加并放在%eax中，和M[%ebx]进行比较，需要相等。

 8048bb0:   8d 5c 24 20             lea    0x20(%esp),%ebx

 8048bb4:   8d 74 24 30             lea    0x30(%esp),%esi

 8048bb8:   8b 43 f8                mov    -0x8(%ebx),%eax

 8048bbb:   03 43 fc                add    -0x4(%ebx),%eax

 8048bbe:   39 03                   cmp    %eax,(%ebx)

 8048bc0:   74 05                   je     8048bc7 <phase\_2+0x43>

 8048bc2:   e8 af 05 00 00          call   8049176 <explode\_bomb>

 8048bc7:   83 c3 04                add    $0x4,%ebx

 8048bca:   39 f3                   cmp    %esi,%ebx

 8048bcc:   75 ea                   jne    8048bb8 <phase\_2+0x34>

 8048bce:   83 c4 34                add    $0x34,%esp

 8048bd1:   5b                      pop    %ebx

 8048bd2:   5e                      pop    %esi

 8048bd3:   c3                      ret

也就是说，第三个数是前两个数的加和。再关注到add    $0x4,%ebx这一步，这是在将处理地址增加，也就是现在%ebx中保存m[4]的地址了。从而实现了循环的传递。

这里的 cmp    %esi,%ebx是循环的跳出判定，其实可以看到%esi内存放的0x30(%esp)正好是六个数的数组的结束地址，这就很好地解释了这个循环。

根据上述分析，随着循环进行，答案依次为0,1,1,2,3,5。是一个斐波那契数列。

我们输入答案，验证通过。



**<Phase\_3>分支语句与跳转表**

首先关注这一段调用scanf函数前的前置工作，实际上干的事情就是把scanf(“%d”,&a)中的几个地址参数&a给确定好，我们看到关键的有这三个0x28(%esp)、0x2f(%esp)、0x24(%esp)。而这三个在向调用函数传递的位置也是连续的0x10(%esp)、0xc(%esp)、0x8(%esp)。也就是说，把读入进来的数分别存放在这三个位置上。

再根据我们对scanf的了解，应该还要传递读取的方式。movl   $0x804a29e,0x4(%esp)应该干的就是这个事情，我们可以用字符的形式查看这个地址0x804a29e



可以发现我们需要输入的是一个整数，一个字符和一个整数，中间以空格隔开。

 8048bd7:   8d 44 24 28             lea    0x28(%esp),%eax

 8048bdb:   89 44 24 10             mov    %eax,0x10(%esp)

 8048bdf:   8d 44 24 2f             lea    0x2f(%esp),%eax

 8048be3:   89 44 24 0c             mov    %eax,0xc(%esp)

 8048be7:   8d 44 24 24             lea    0x24(%esp),%eax

 8048beb:   89 44 24 08             mov    %eax,0x8(%esp)

 8048bef:   c7 44 24 04 9e a2 04    movl   $0x804a29e,0x4(%esp)

 8048bf6:   08

 8048bf7:   8b 44 24 40             mov    0x40(%esp),%eax

 8048bfb:   89 04 24                mov    %eax,(%esp)

 8048bfe:   e8 6d fc ff ff          call   8048870 <\_\_isoc99\_sscanf@plt>

上面关注了我们输入的形式，接下来继续阅读代码，尝试解出这三个值。

 8048bfe:   e8 6d fc ff ff          call   8048870 <\_\_isoc99\_sscanf@plt>

 8048c03:   83 f8 02                cmp    $0x2,%eax

 8048c06:   7f 05                   jg     8048c0d <phase\_3+0x39>

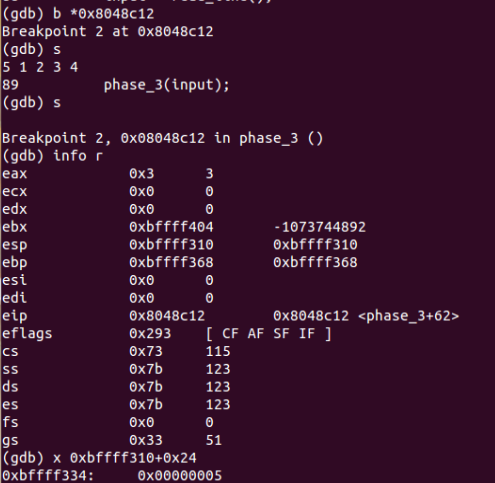
 8048c08:   e8 69 05 00 00          call   8049176 <explode\_bomb>

 8048c0d:   83 7c 24 24 07          cmpl   $0x7,0x24(%esp)

 8048c12:   0f 87 fc 00 00 00       ja     8048d14 <phase\_3+0x140>

根据这一段逻辑，这里要避开两个炸弹，要求的条件是call函数返回后%eax内的值>2且0x24(%esp)<=7，这里0x24(%esp)内存放的应该是跳转的数n，所以从这里可以看出这个case语句应该有0~7这8个选项，跳转表有8个终点。

我们假定这个0x24(%esp)里存放的是我们第一个读入的数，我们尝试输入一个5，发现它真的被存在%eax内，可以确定第一个输入的数决定跳转地址。

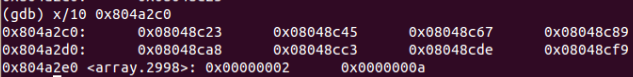


看到这一句话，关注到这应该是一个跳转表。

 8048c1c:   ff 24 85 c0 a2 04 08    jmp    \*0x804a2c0(,%eax,4)

而跳转表的首地址应该是0x804a2c0。

我们看一看跳转表（其实可以根据上面推断出应该有8个，但为了不缺漏，我们可以直接查看10个，更加方便）



这样我们就拿到了跳转表的终点地址。

接下来的8个这样的结构，实际上是多个case分支的下要干的事情以及判定。

 8048c23:   b8 64 00 00 00          mov    $0x64,%eax

 8048c28:   81 7c 24 28 80 00 00    cmpl   $0x80,0x28(%esp)

 8048c2f:   00

 8048c30:   0f 84 e8 00 00 00       je     8048d1e <phase\_3+0x14a>

 8048c36:   e8 3b 05 00 00          call   8049176 <explode\_bomb>

 8048c3b:   b8 64 00 00 00          mov    $0x64,%eax

 8048c40:   e9 d9 00 00 00          jmp    8048d1e <phase\_3+0x14a>

我们以第一个举例，首先0x28(%esp)要等于0x80，然后再关注所有case分支最终跳转的汇合点。不难发现这里还要比较%eax的值和0x2f(%esp)保存的值是否相等。

总结来说，对于我们输入的一组数据n，c，m。这里分别判定了在n分支下c和m是否为给定值。对于这个分支，是c=0x64，m=0x80，也就是字符为d，数为128。

 8048d1e:   3a 44 24 2f             cmp    0x2f(%esp),%al

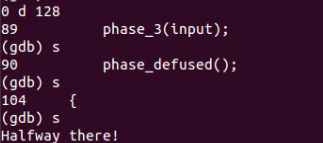
 8048d22:   74 05                   je     8048d29 <phase\_3+0x155>

 8048d24:   e8 4d 04 00 00          call   8049176 <explode\_bomb>

 8048d29:   83 c4 3c                add    $0x3c,%esp

 8048d2c:   c3                      ret

我们输入0 d 128进行验证，发现验证正确。



同理，另外7组跳转表也可以一一得出相应的答案。本题一共8组答案，任意一组输入都为正确。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | C(hex) | M(hex) | c | m |
| 0 | 0x64 | 0x80 | d | 128 |
| 1 | 0x6f | 0x135 | o | 309 |
| 2 | 0x67 | 0x348 | g | 840 |
| 3 | 0x6b | 0x16d | k | 365 |
| 4 | 0x75 | 0x1d7 | u | 471 |
| 5 | 0x72 | 0x31b | r | 795 |
| 6 | 0x6e | 0x204 | n | 516 |
| 7 | 0x6f | 0xb0 | o | 176 |

最终答案即为

|  |  |  |
| --- | --- | --- |
| n | c | m |
| 0 | d | 128 |
| 1 | o | 309 |
| 2 | g | 840 |
| 3 | k | 365 |
| 4 | u | 471 |
| 5 | r | 795 |
| 6 | n | 516 |
| 7 | o | 176 |

**<Phase\_4>递归调用**

这道题与上一道题的代码有很大的相似之处。

像是这一段

 8048d8d:   8d 44 24 18             lea    0x18(%esp),%eax

 8048d91:   89 44 24 0c             mov    %eax,0xc(%esp)

 8048d95:   8d 44 24 1c             lea    0x1c(%esp),%eax

 8048d99:   89 44 24 08             mov    %eax,0x8(%esp)

 8048d9d:   c7 44 24 04 83 a4 04    movl   $0x804a483,0x4(%esp)

和这一段

 8048dac:   e8 bf fa ff ff          call   8048870 <\_\_isoc99\_sscanf@plt>

分别是传递用scanf所读取数的存放位置，和scanf函数本身。这个思路我们在上一道题就已经看到过了。第一件事情肯定是查看这个地址，获取读入的类型。



可见这次是读入两个整数。

这一点也可以从下面看出来，这里%eax是scanf函数的返回值，也就是读入数的个数，若这个个数不为2，就会爆炸。

 8048db1:   83 f8 02                cmp    $0x2,%eax

 8048db4:   75 0e                   jne    8048dc4 <phase\_4+0x3a>

接着由以下两句，%eax存的是我们输入的第二个值，我们称它为x。

 8048db6:   8b 44 24 18             mov    0x18(%esp),%eax

 8048dba:   83 f8 01                cmp    $0x1,%eax

再关注到这里对x的两个限制。有1<x<=4，所以x有3个可能的取值，分别是1,2,3。

 8048dba:   83 f8 01                cmp    $0x1,%eax

 8048dbd:   7e 05                   jle    8048dc4 <phase\_4+0x3a>

 8048dbf:   83 f8 04                cmp    $0x4,%eax

 8048dc2:   7e 05                   jle    8048dc9 <phase\_4+0x3f>

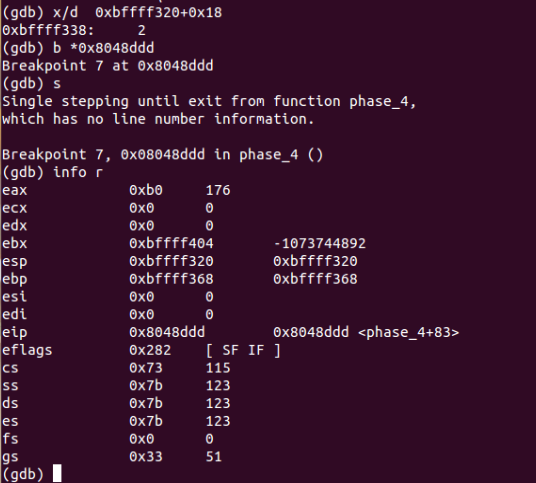
 8048dc4:   e8 ad 03 00 00          call   8049176 <explode\_bomb>

接下来有一个取巧的方法，就是让电脑帮我们计算出结果，我们再直接拿。

关注到这一句话，这里将我们输入的第一个数（我们称它为check）和%eax比较，若相等则正确否则爆炸。那么如果我们直接在这一步break，再使用info r查看%eax的值，不就可以轻轻松松取得答案了嘛。

 8048ddd:   3b 44 24 1c             cmp    0x1c(%esp),%eax

下面以x=2为例，按照这个思路操作一下。



可以得出结果为176，代入之后成功，说明我们的想法是正确的。

**<func\_4>递归函数体分析**

当然，出于对自己的要求以及对func4函数的尊敬，我还是看了一遍func4函数。

下面是一些分析。

（1）主函数

Phase\_4中这一段，很详细地描述了func\_4函数的两个参数，(%esp)也就是第一个参数，是9，也就是递归层数，我们称作n；0x4(%esp)也就是第二个参数，是x。

所以func\_4(n,x)

 8048dc9:   8b 44 24 18             mov    0x18(%esp),%eax

 8048dcd:   89 44 24 04             mov    %eax,0x4(%esp)

 8048dd1:   c7 04 24 09 00 00 00    movl   $0x9,(%esp)

 8048dd8:   e8 50 ff ff ff          call   8048d2d <func4>

接下来真正进入func\_4函数

（2）寄存器保护

首先，开头的这三步

 8048d30:   89 5c 24 10             mov    %ebx,0x10(%esp)

 8048d34:   89 74 24 14             mov    %esi,0x14(%esp)

 8048d38:   89 7c 24 18             mov    %edi,0x18(%esp)

和结尾的这三步

 8048d7a:   8b 5c 24 10             mov    0x10(%esp),%ebx

 8048d7e:   8b 74 24 14             mov    0x14(%esp),%esi

 8048d82:   8b 7c 24 18             mov    0x18(%esp),%edi

将三个寄存器里的值先放在主存中保存，在函数结束之后再放回寄存器内，这是因为在这个函数中，这几个寄存器要被用到，这是一种保护的措施。这是一种在函数调用间很常见的步骤，在本题的具体的原因还有待研究。

（3）传参的保存

接下来关注这一句（建立）

 8048d2d:   83 ec 1c                sub    $0x1c,%esp

和这两句（传参保存）

 8048d3c:   8b 74 24 20             mov    0x20(%esp),%esi

 8048d40:   8b 5c 24 24             mov    0x24(%esp),%ebx

可以发现，%ebx存的是x，%esi存的是n。

（4）递归边界条件

接下来是递归的边界条件。

第一个条件，test是判0，若n=0，返回0。

 8048d44:   85 f6                   test   %esi,%esi

 8048d46:   7e 2b                   jle    8048d73 <func4+0x46>

第二个条件，若n=1，返回%ebx内的值，也就是x。

 8048d48:   83 fe 01                cmp    $0x1,%esi

 8048d4b:   74 2b                   je     8048d78 <func4+0x4b>

（5）递归函数执行部分

再接下来是递归函数的执行部分。

这是第一段，计算func4(n-1,x)+x并保存在%edi内

 8048d4d:   89 5c 24 04             mov    %ebx,0x4(%esp)

 8048d51:   8d 46 ff                lea    -0x1(%esi),%eax

 8048d54:   89 04 24                mov    %eax,(%esp)

 8048d57:   e8 d1 ff ff ff          call   8048d2d <func4>

 8048d5c:   8d 3c 18                lea    (%eax,%ebx,1),%edi

下面是第二段，计算func4(n-2,x)，再和%edi相加，保存在%ebx内。

 8048d5f:   89 5c 24 04             mov    %ebx,0x4(%esp)

 8048d63:   83 ee 02                sub    $0x2,%esi

 8048d66:   89 34 24                mov    %esi,(%esp)

 8048d69:   e8 bf ff ff ff          call   8048d2d <func4>

 8048d6e:   8d 1c 07                lea    (%edi,%eax,1),%ebx

最后若n不是0或1，返回%ebx的值。

（5）总结

总的来看，就是计算了func4(n-1,x)+func4(n-2,x)+x的值并返回；若n=0，返回0；若n=1，返回x。知道了这个，我们就可以写一段c程序来模拟这件事，并靠自己得出结果。

#include <iostream>

using namespace std;

int result(int n, int x)

{

  if (n == 0)

    return 0;

  if (n == 1)

    return x;

  return (result(n - 1, x) + result(n - 2, x) + x);

}

int main()

{

  int x;

  x = 4;

  cout << result(9, x) << endl;

  return 0;

}

最后输出的结果就是check的值。证明我们推断正确。

所以本题的答案如下。

|  |  |
| --- | --- |
| Check | x |
| 176 | 2 |
| 264 | 3 |
| 352 | 4 |

**<phase\_5>字符串末尾累加**

首先分析题目。

（1）比较字符个数

首先关注到第一个炸弹点，字符个数，这里的string\_length猜测是返回了我们输入字符的个数，且必须要等于6，也就是说限定了我们必须输入6位字符。

 8048df7:   e8 4f 02 00 00          call   804904b <string\_length>

 8048dfc:   83 f8 06                cmp    $0x6,%eax

 8048dff:   74 05                   je     8048e06 <phase\_5+0x1a>

 8048e01:   e8 70 03 00 00          call   8049176 <explode\_bomb>

1. 赋初值、各寄存器含义、以及初探循环

这里给寄存器%eax赋了初值0，给寄存器%edx赋了初值0。

 8048e06:   ba 00 00 00 00          mov    $0x0,%edx

 8048e0b:   b8 00 00 00 00          mov    $0x0,%eax

结合后面%eax寄存器自增以及与6比较并跳转（如下），不难发现%eax应该承担着一个循环控制变量i的作用。而与6相比较，不难发现这其实是一个遍历我们输入字符串各个位置的循环，总共执行6次。

 8048e1e:   83 c0 01                add    $0x1,%eax

 8048e21:   83 f8 06                cmp    $0x6,%eax

 8048e24:   75 ea                   jne    8048e10 <phase\_5+0x24>

再结合跳出循环后%edx与固定值相比较，我们不难发现，实际上这个%edx是一个累加器。而第i重循环的作用也正是不断地给这个累加器加上一个特定的值，这个特定的值与我们输入的字符串的第i位有关。

 8048e26:   83 fa 38                cmp    $0x38,%edx

 8048e29:   74 05                   je     8048e30 <phase\_5+0x44>

 8048e2b:   e8 46 03 00 00          call   8049176 <explode\_bomb>

1. 循环执行体

其实循环执行体很简单，就只有如下几句，它想干的事情是这样的。s[i]的末4位与0xf相与，结果以整数形式存储，记作r。（s[i]本应该是8位，这里实际上取了它的末4位，结果共有16种，即0~15）。访问（基地址0x804a2e0+4\*r）这个地址，记作target\_address，把这个地址上的值累加到寄存器%edx上。这就结束了。

 8048e10:   0f be 0c 03             movsbl (%ebx,%eax,1),%ecx

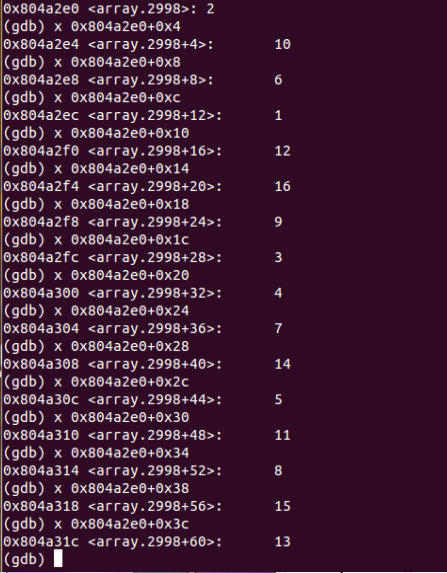
 8048e14:   83 e1 0f                and    $0xf,%ecx

 8048e17:   03 14 8d e0 a2 04 08    add    0x804a2e0(,%ecx,4),%edx

1. 累加结果比较

最后%edx与0x38，也就是十进制下的56比较，要求%ebx内的值等于它。

分析完题目之后，我们要来解题了。要知道输入什么字符能最终得到正确的累加结果。我们可以先查看所有（基地址0x804a2e0+4\*r）关于r从0到15的结果。如下。



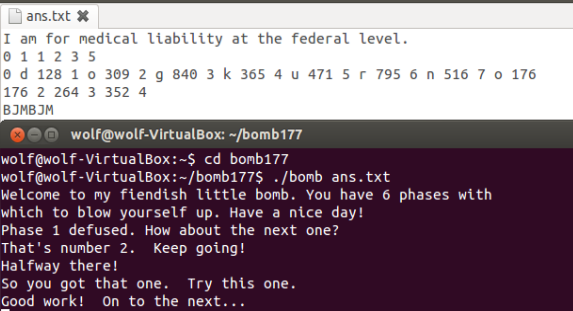
查阅ASCII码表后，我们可以总结出下表。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 可能字符 | | | | | | 字符ascii码末4位 | | Target\_address | 单次累加值 |
| Bin | Hex |
| (space) | 0 | @ | P | 、 | p | 0000 | 0 | Bias+0 | 2 |
| ! | 1 | A | Q | a | q | 0001 | 1 | Bias+0x4 | 10 |
| “ | 2 | B | R | b | r | 0010 | 2 | Bias+0x8 | 6 |
| # | 3 | C | S | c | s | 0011 | 3 | Bias+0xc | 1 |
| $ | 4 | D | T | d | t | 0100 | 4 | Bias+10 | 12 |
| % | 5 | E | U | e | u | 0101 | 5 | Bias+0x14 | 16 |
| & | 6 | F | V | f | v | 0110 | 6 | Bias+0x18 | 9 |
| . | 7 | G | W | g | w | 0111 | 7 | Bias+0x1c | 3 |
| ( | 8 | H | X | h | x | 1000 | 8 | Bias+20 | 4 |
| ) | 9 | I | Y | i | y | 1001 | 9 | Bias+0x24 | 7 |
| \* | : | J | Z | j | z | 1010 | a | Bias+0x28 | 14 |
| + | ; | K | [ | k | { | 1011 | b | Bias+0x2c | 5 |
|  | < | L | / | l | | | 1100 | c | Bias+30 | 11 |
|  | = | M | ] | m | } | 1101 | d | Bias+0x34 | 8 |
|  | > | N | ^ | n |  | 1110 | e | Bias+0x38 | 15 |
|  | ? | O |  | o |  | 1111 | f | Bias+0x3c | 13 |

以本题要求最后6个字符的累加值为56，我们可以取BJMBJM这个字符串，因为B对应6，而J对应14，M对应8，加和正好为56。

由此我们可以看到，这道题的答案应该有很多组，只要按照上表其最终累加结果为56即可。我们不再一一列出。

附上使用BJMBJM验证的结果。



可见我们的推论是正确的。

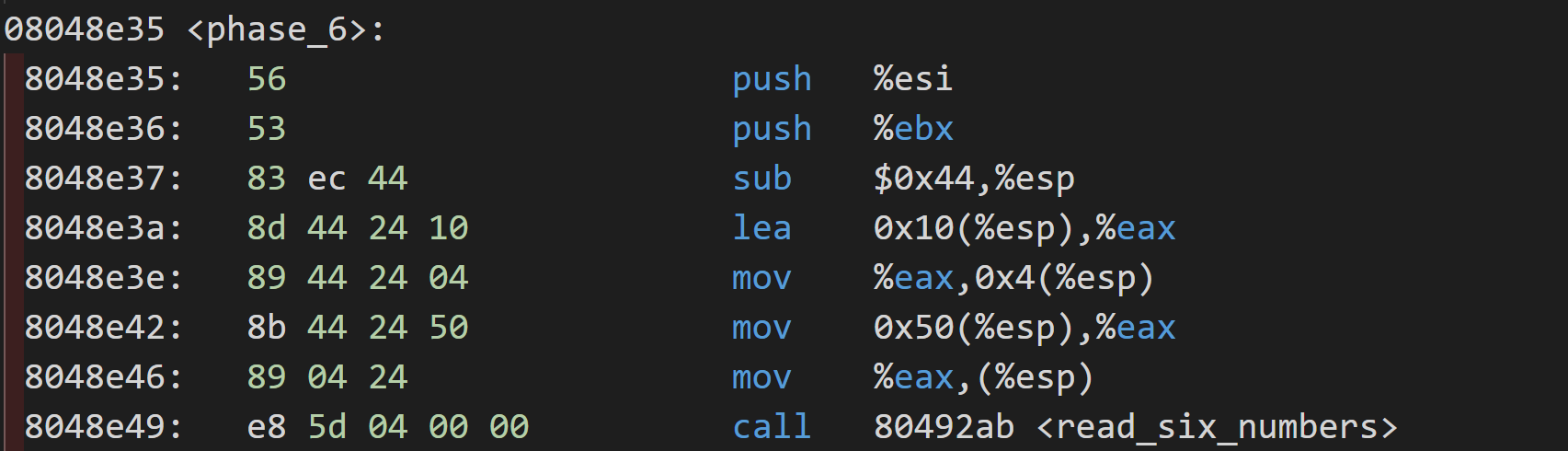
**<phase\_6>链表**

总述：

有六个节点，每个节点中已预先存储了一个值。要求我们按照顺序输入节点编号，来保证其中结点的值依次递增。

首先，我们来对代码进行静态的分析。

（1）读入节点编号



这里没有什么可说的，就是读入节点编号。

1. 第一阶段：合规性检验

确保六个数字符合规定。规定是指，不超过6，不重复，即必须是1、2、3、4、5、6的排列。

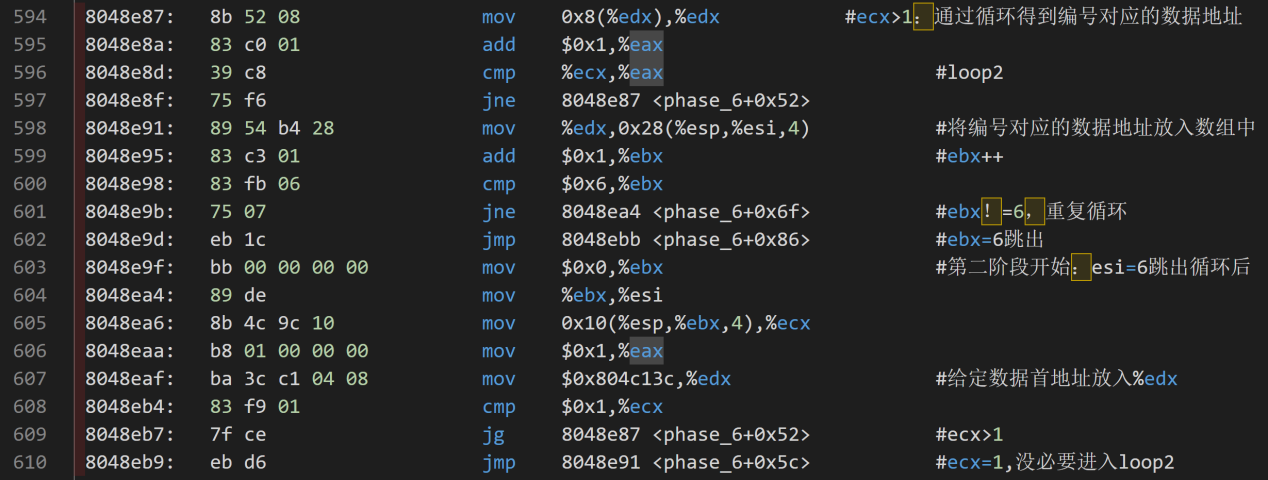


这部分代码，使用c++可以写成如下形式。

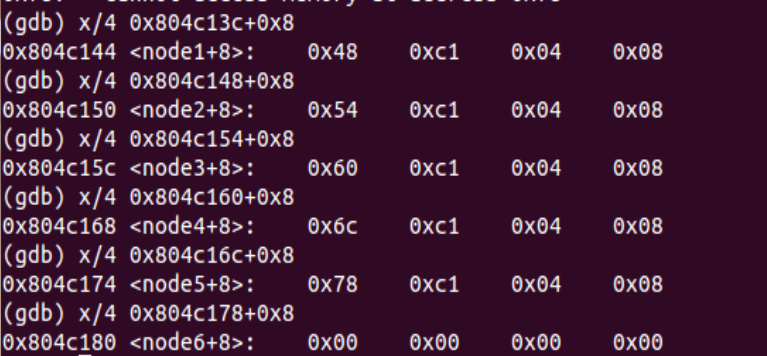
有一个循环的嵌套，其中，外层循环的控制变量为%esi，内层循环的控制变量为%ebx



（3）第二阶段：排序



注意到有给定一个地址，我们发现循环loop2在不断访问(%edx+8)，遂查看这些地址。



注意到给定地址0x804c13c储存在%edx内，而在%ecx>1时，会通过循环不断将(%edx+8)所对应的值赋值给%edx。实际上，我们查看每一个地址的(%edx+8)会发现，这实际上是一个链表型的结构，每两个结构之间相邻12个字节，而这12个字节被用来存3样东西。(%edx+8)也就是储存的第三项数据，实际上就是下一结构的地址。这一段是循环loop2的解释。

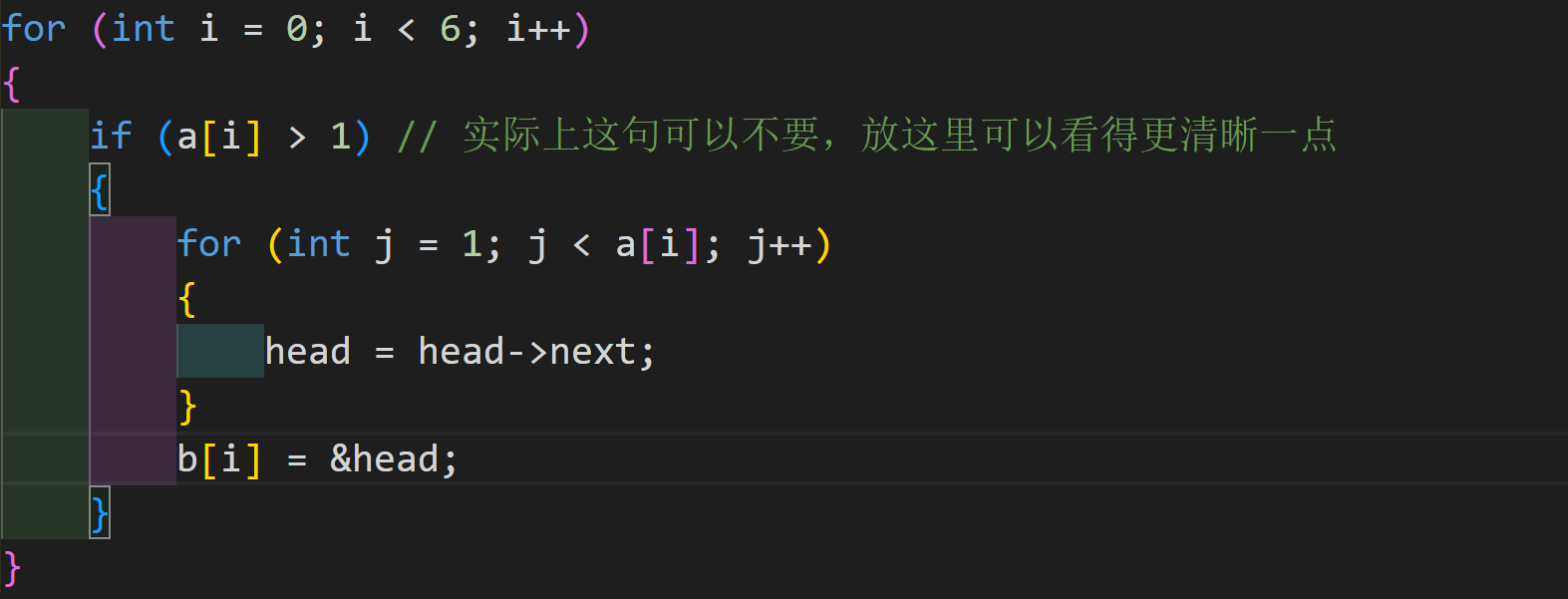
为方便说明，我们约定如下：

0x10(%esp,%ebx,4)开始的，为我们输入的六个元素按顺序排列，约定为数组a[]

0x28(%esp,%esi,4)为新数组，约定为b[]

所以这里我们可以看出来是通过循环访问（循环控制变量为%ebx）输入的编号a[i].对于特定的一次编号i，若其为1，则就将链表结构的第一个节点的地址0x804c13c移至一个新的数组b[i]；若不为1，则通过循环loop2（上面解释过）找到这个编号i对对应的节点，将其地址（0x804c148、0x804c154、0x804c160、0x804c16c、0x804c178中的一个）移至b[i]。

使用c来更直观地表述如下。



这实际上是一个排序的过程。

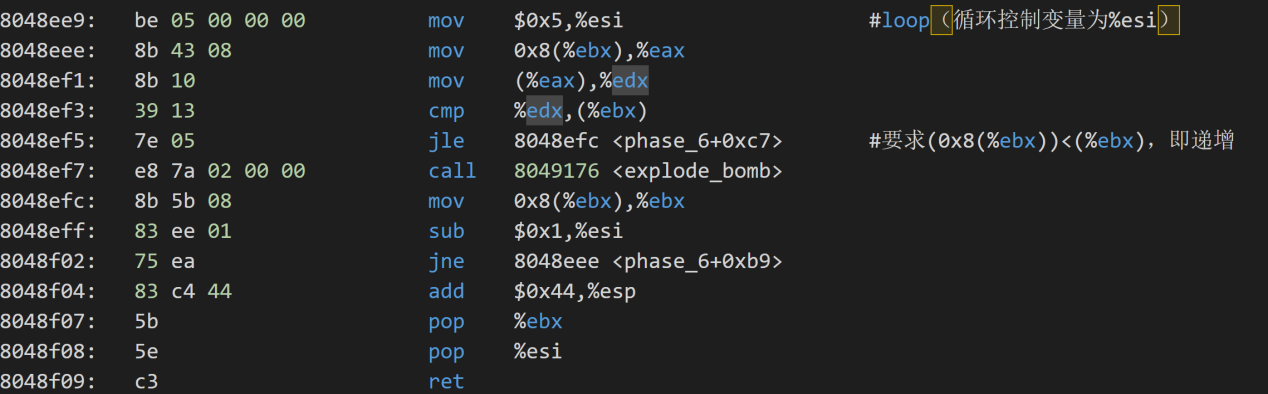
（4）第三阶段：复写下一状态（链表重组）



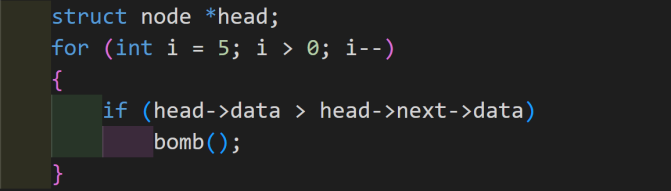
这一部分根据上一阶段的排序结果来对链表节点进行重组，按照我们输入的数组a[]的顺序来重新排列链表节点的顺序。

（5）检验数据域递增性

最后，查看输入顺序是否正确，即在按照我们输入的数组a[]进行重新排列后，链表内节点的data域是否递增。

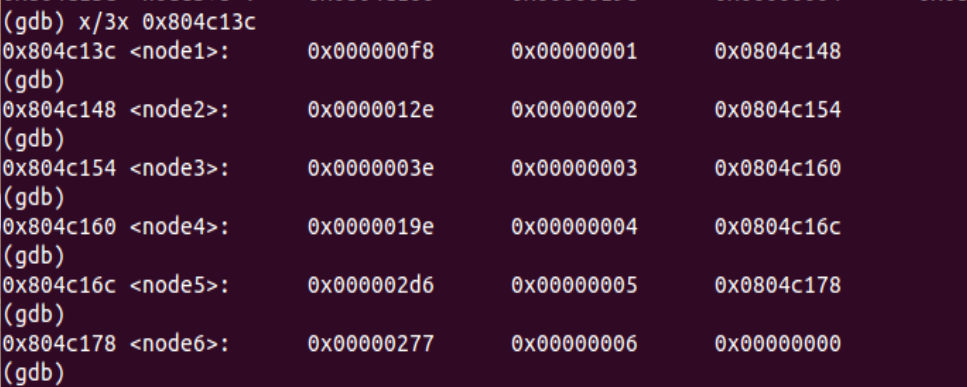


用c来更清晰地表示：

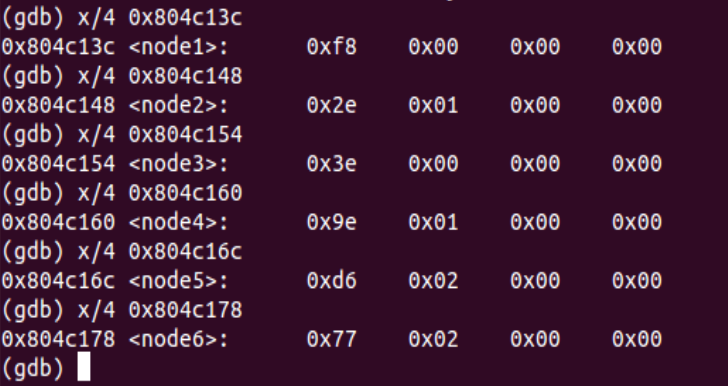


（6）node类型分析

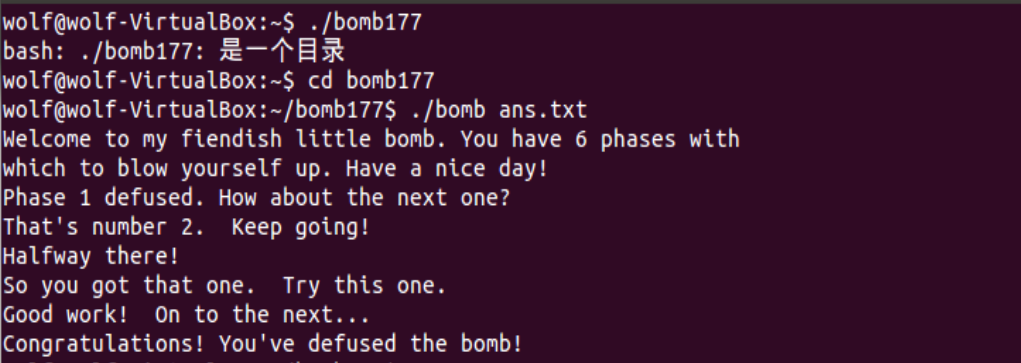
事实上，如果我们对这个节点进行一个查看，截图如下。



不难发现其实节点有三个域，第一个是数据域，存储的是值的大小；第二个是编号域，反应的是节点的编号（实际上用处不大）；第三个是next域，是指针类型，指向下一个节点的地址。这是一个典型的链表结构。

根据上述分析，若要输入正确的顺序，我们必须得知道每个节点的数据域是多少。

查看每个节点的数据域，我们发现它们按顺序分别是0xf8、0x12e、0x3e、0x19e、0x2d6、0x277。排序过后的编码应该为3、1、2、4、6、5.输入验证，发现正确。



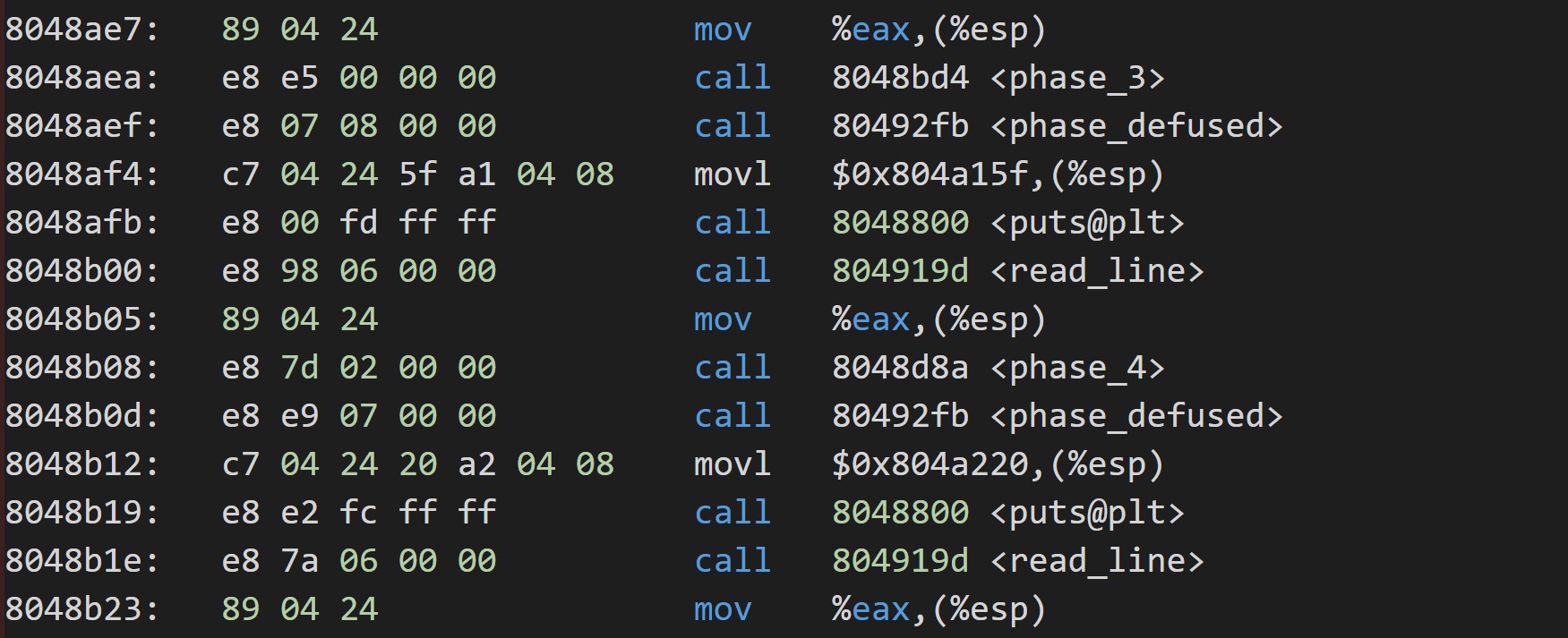
**<secret\_phase>二叉检索树（BinarySearchTree）**

前言：这道题比较难看出来，是一个二叉检索树并且寻找目标节点的编号。最终实现的目标是我们输入一个数input，它所在的位置与题目中所规定的相符合。

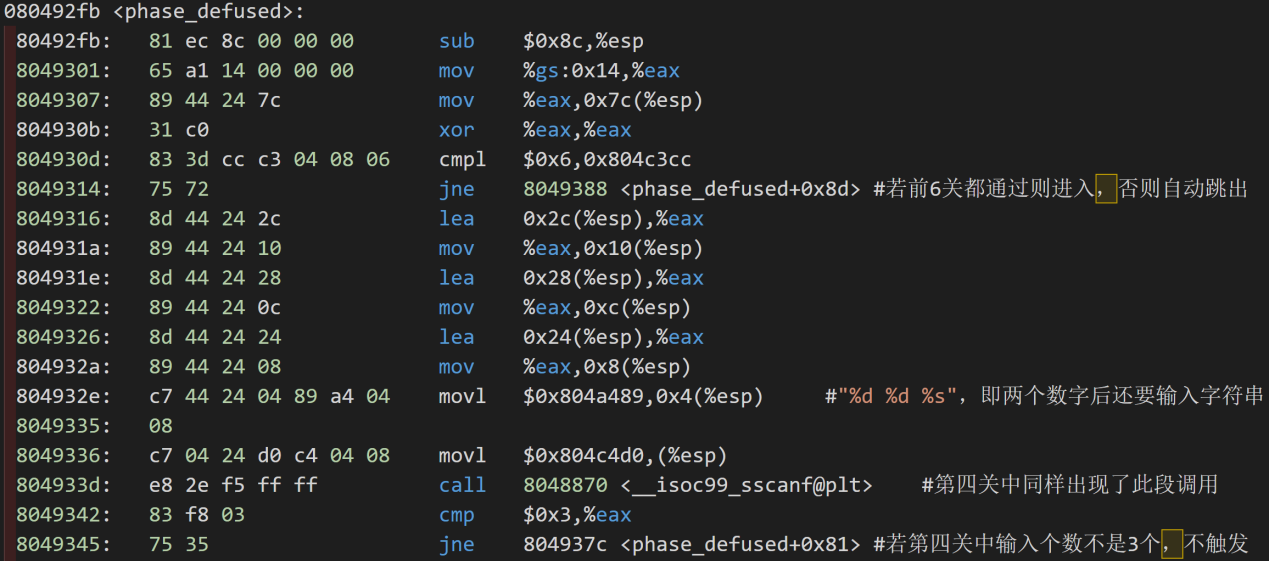
**<phase\_defused>寻找隐藏关入口**

关注到main函数部分，每次在调用相应的phase之后都会调用<phase\_defused>，猜测可能跟隐藏关卡相关，所以必须要对这一段函数进行研究。

（下图为main函数的phase\_3、4在完成档次验证后都调用了<phase\_defused>）



下面查看<phase\_defused>代码

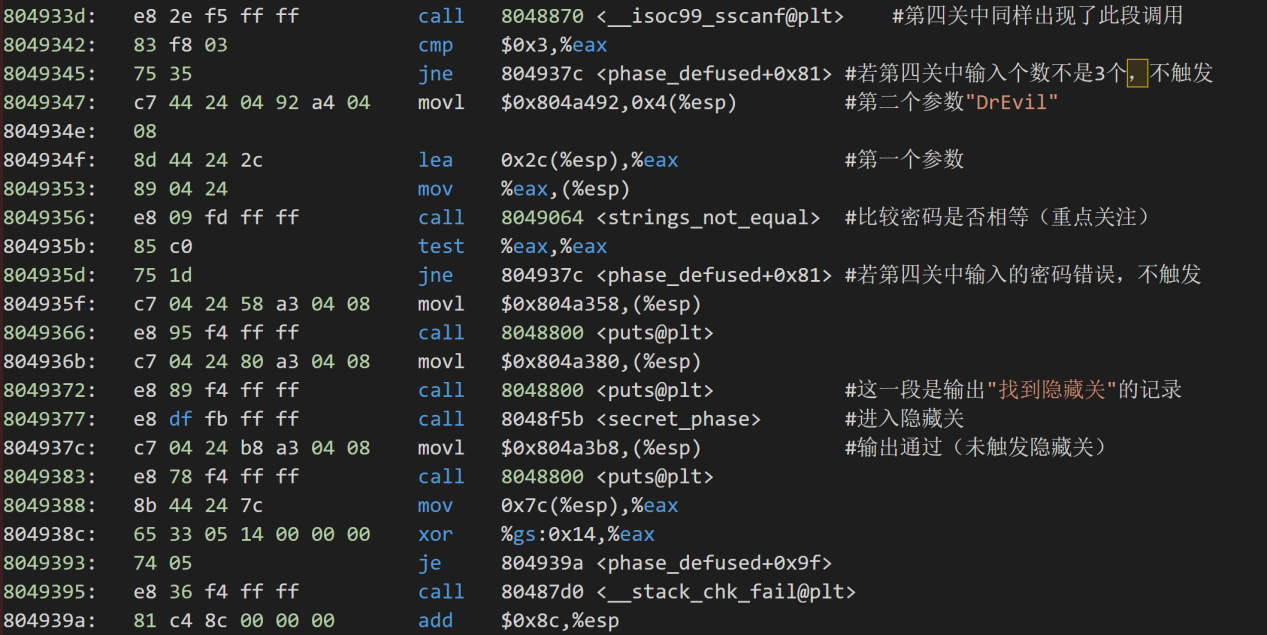


发现在<phase\_defused>内有一个判定，若前6关都通过才会触发隐藏关卡，否则跳过。触发隐藏关卡之后，首先判定第四关中输入的是不是两个整数与一个字符串。我们知道，第四关的答案是两个整数，但是如果要进入隐藏关卡，这里需要在两个整数之后再输入一个字符串。

（查看输入格式的截图）



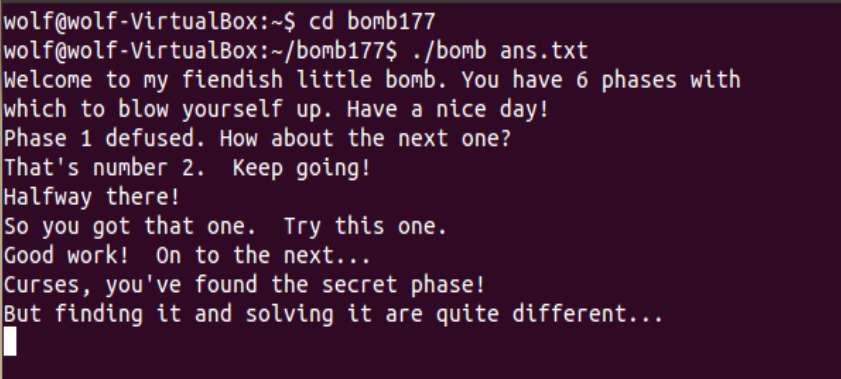
这里将我们在第四关中输入的字符串作为第一个参数，将给定字符串（也就是答案）作为第二个参数，传递给字符串相等性判定函数。若我们输入的与给定值一样，则会向终端输出信息，并进入隐藏关卡。进入隐藏关卡后，调用<secret\_phase>段。



（查看给定密码的截图）



（在第四关中输入正确的密码后，向终端提示已经成功进入隐藏关卡）

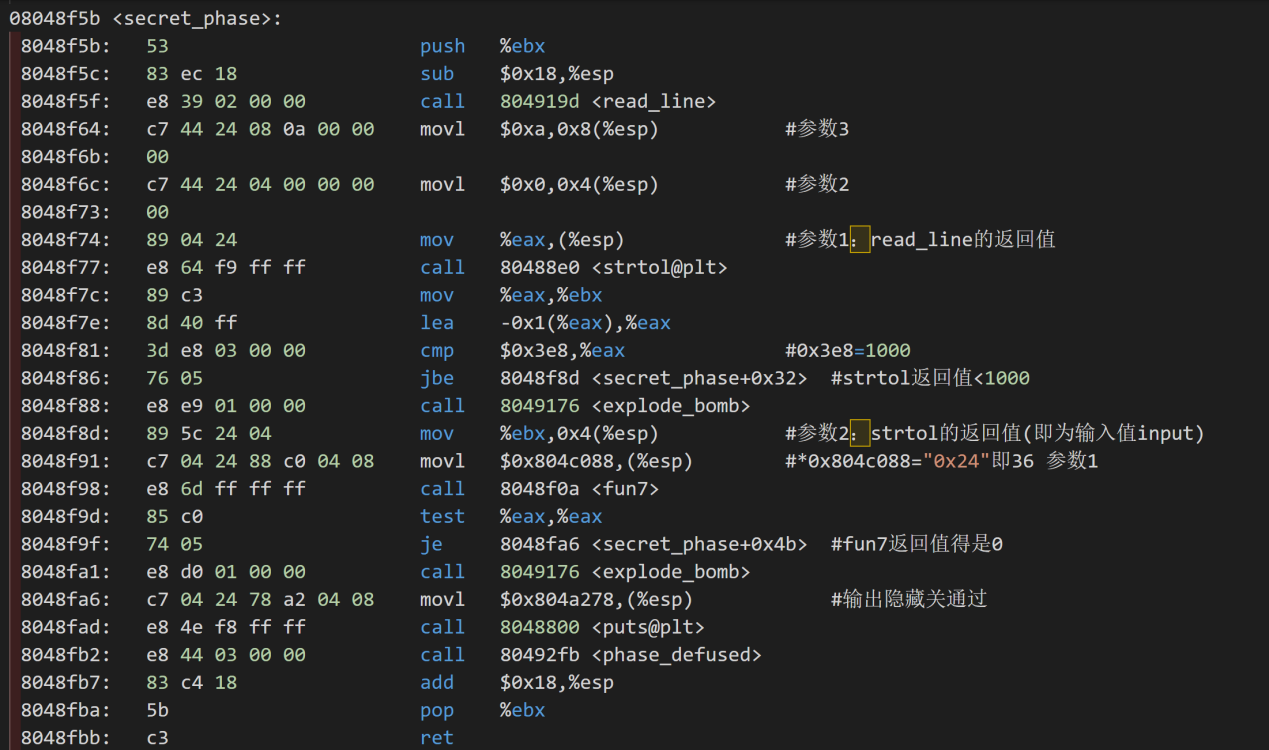


**<secret\_phase>隐藏关**

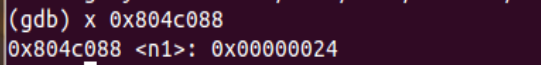
在这一段中，首先从终端读入字符串，通过strtol函数将字符串转换为长整型数。

Strol的原型函数为long int strtol(const char \*nptr,char \*\*endptr,int base);其中第一个参数为待转换的字符串首地址，第二个参数为转换的首地址，第三个参数为转换的进制（范围为2至36）这里是0xa，即转换为十进制。

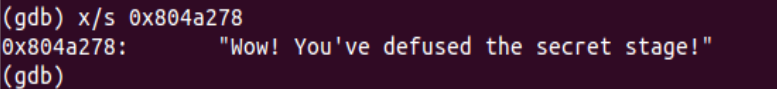
若我们输入的就是数值，通过strol函数返回的值即为我们输入的数值，不作改变。我们称它为input。而后，input作为第二个参数传递入fun7函数，给定地址作为第一个参数传递入fun7。



（查看给定地址值）



（查看最后会输出什么[不必要，纯粹为了好玩]）

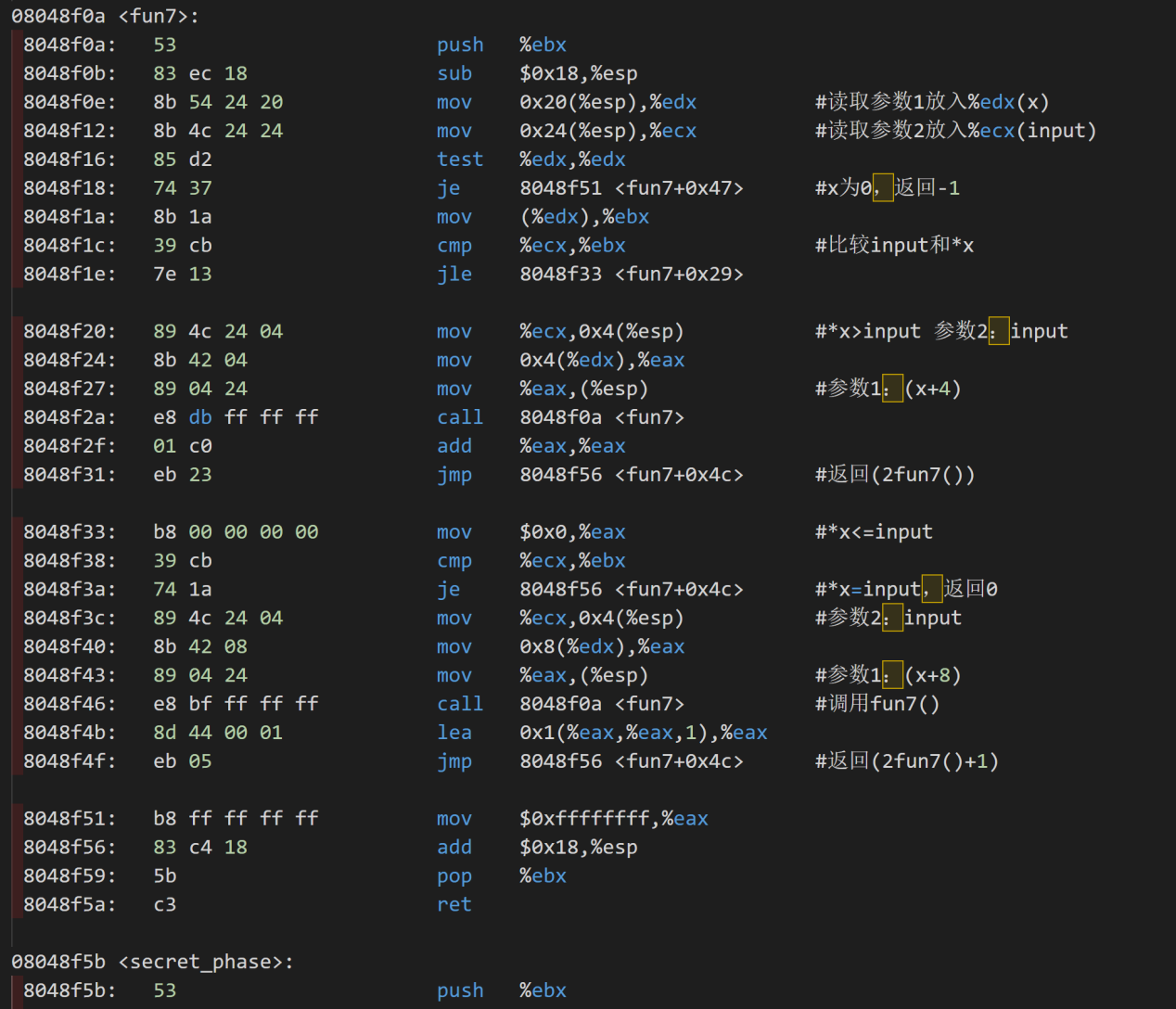


最后期待的是fun7返回的是0，这样就能够正确解出。

**<fun7>检索目标元素**

发现%ecx中一直存放我们之前输入的input值，从未被改变过。根据传入指针x所对应的值\*x的值与input的关系来确定下一步的走向。有如下递归关系：

递归边界为x=0，即访问到最低地址，就返回-1（错误），事实上永远不可能访问到这个地方，因为这一段代码不是程序能够访问到的。



就解这一道题而言，实际上十分轻松。因为要求返回0，我们只需要第一次进入这个递归结构的时候，就让input=x，它就退出并返回0了。在之前的<secret\_phase>中我们已经查看过\*0x804c088的值为0x24，即为36，所以这道题答案就是36，输入答案即可得到正确的结果。隐藏关卡就结束了。

**深度拓展与思考：**

事实上，之前的学长学姐和本级的同学有遇到要求返回值不是0的情况，这个时候就需要代入进去递归一下了。

此时有几个需要考虑的点：

1. 最深层的那一层递归返回的结果肯定是0；
2. 根据需要的结果来确定递归的层数以及所走路线。

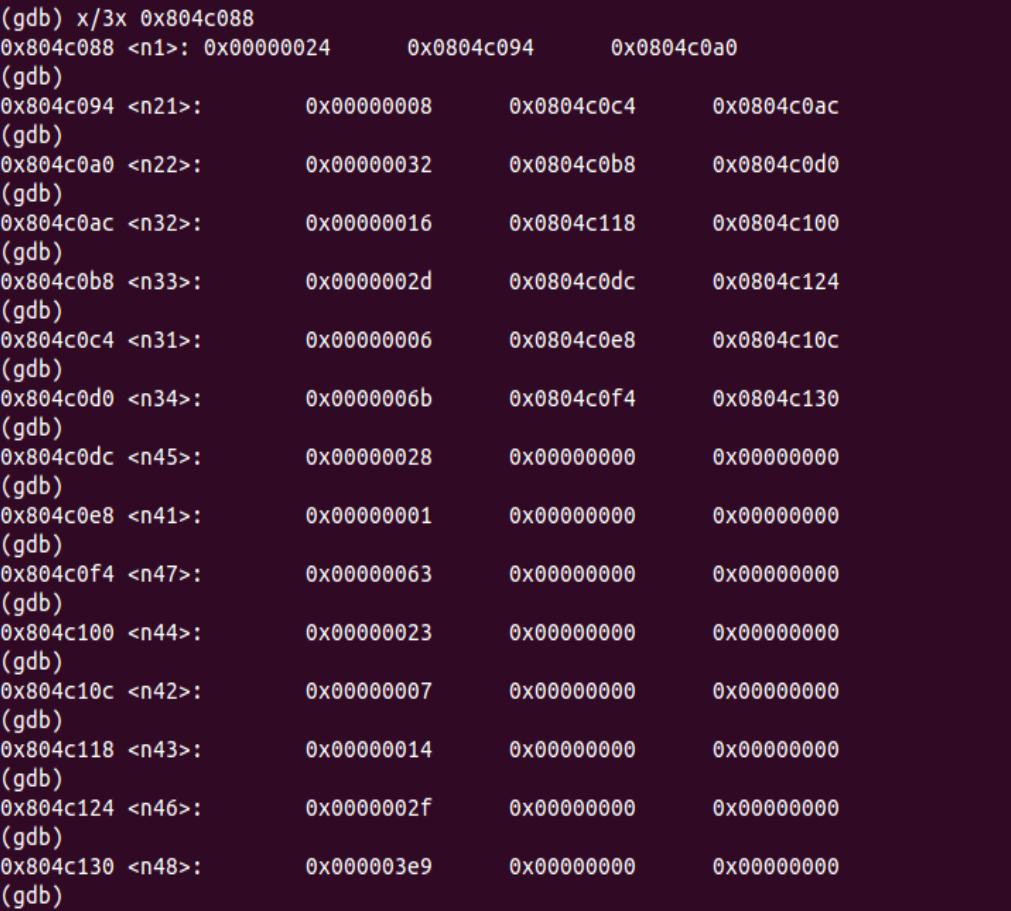
举例说明：

若要求返回2，则一定是fun7(x,input)=2fun7(x+4)=2fun7(x+4+8)+1，即第一层是\*x<input的，返回2fun7(x+4)；而第二层是\*x>input，返回2fun7(x+8)+1；第三层是\*x=input，返回0。在这种情况下只需要查看\*（\*（x+4）+8）即可找到正确的答案。

**进一步思考树结构的实现：**

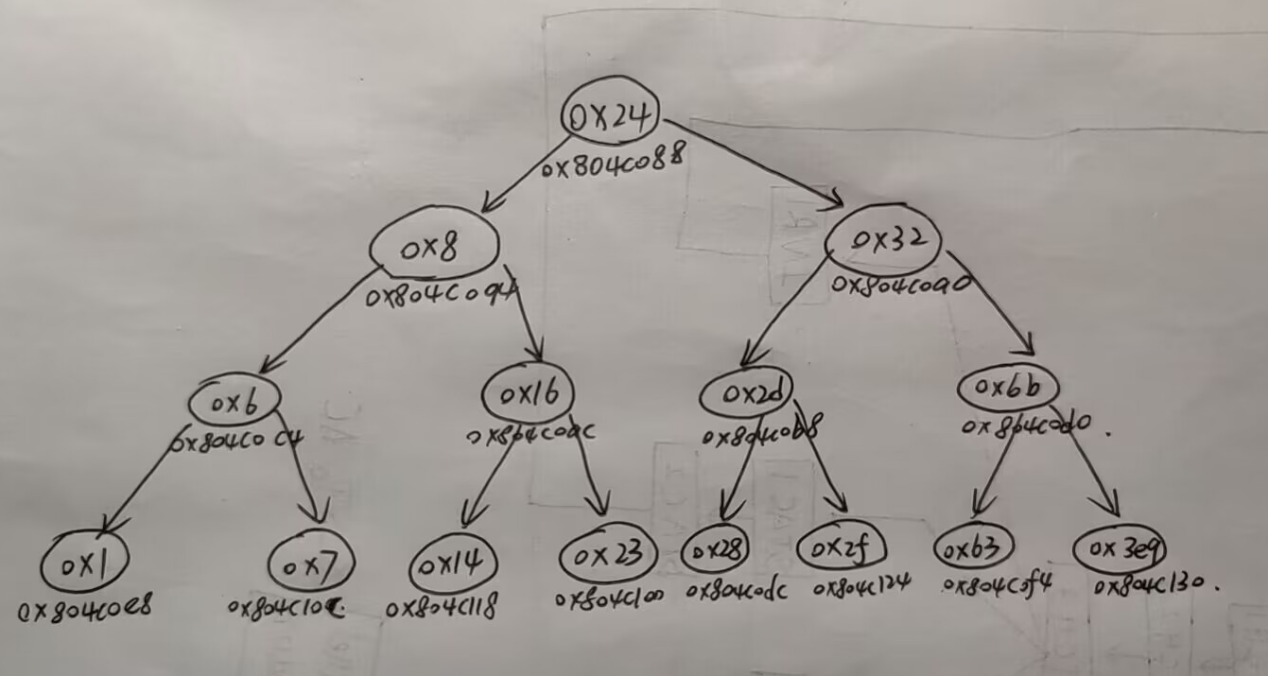
实际上，本题为二叉检索树，上面我是将其作为一个数组去计算的，虽然也能做出最终的结果，但是显然不容易理解，下面我将从二叉检索树的角度重新理解本题。

本质上，如果我们以12字节为步长查看从0x804c088开始的地址空间，我们会发现这实际上是一个链表结构，每个节点都有三个域。第一位的是数据域，第二、三位是指针域，分别指向左孩子和右孩子的地址。



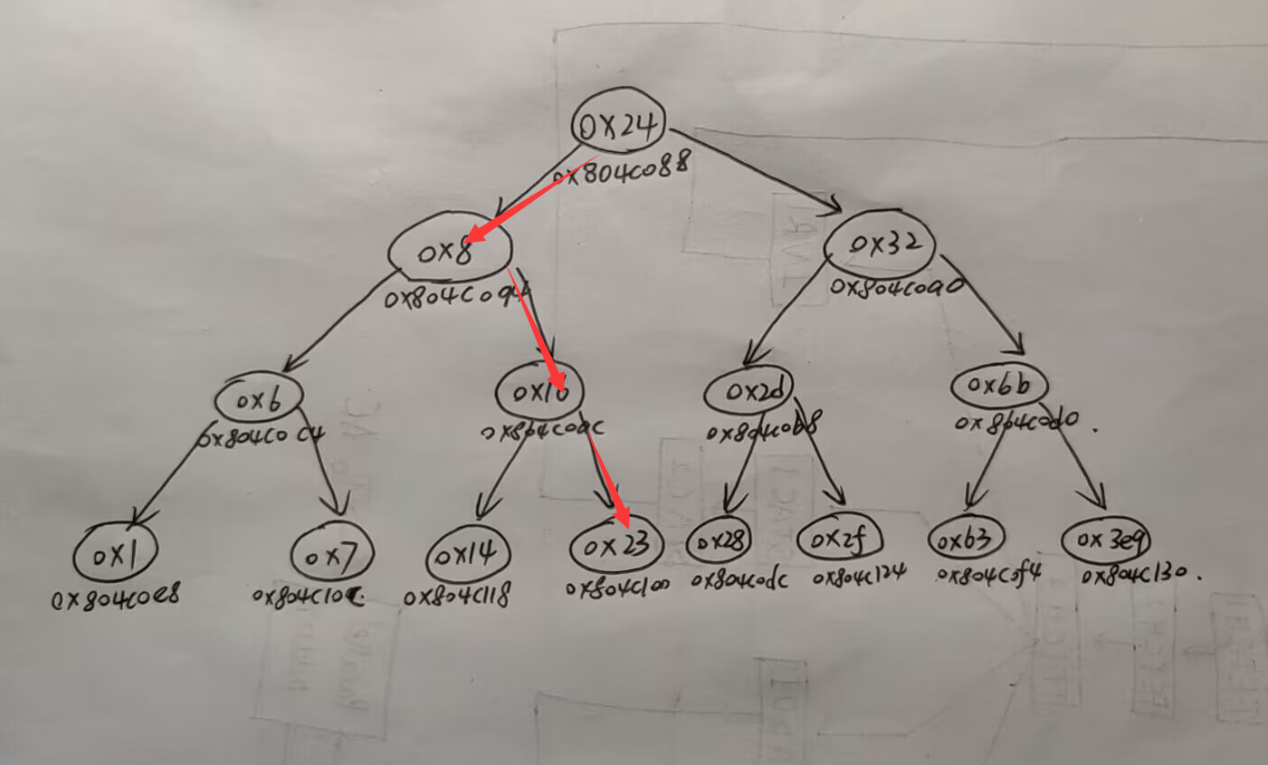
对于一个节点地址x来说，x为其数据域，（x+4）为其左孩子指针域，（x+8）为其右孩子指针域。所以上面我们在\*x<input时访问（x+8），即访问其右孩子，这是根据二叉检索树的性质，该节点右边的肯定比该节点的数据要大。若\*x>input也是同理的，本质上我们就是在模拟进行这个二叉检索树的检索过程。

如根据上面查看到的结果，我可以画出该二叉检索树的示意图如下。



研究到这个层面，就能够比较有效地解决问题较为复杂的问题了。

比如有同学遇到的要求fun7返回6，那么就可以看作是



即答案为0x23，十进制下为35。

解释原因也比较简单。

Fun7(x,input)=2\*fun7(x+4,input)=2\*[fun7(**\*(x+4)+8**,input)+1]

=2\*{2\*fun7(**\*(\*(x+4)+8)+8**,input)+1]+1}=2\*(2\*(0+1)+1)=6

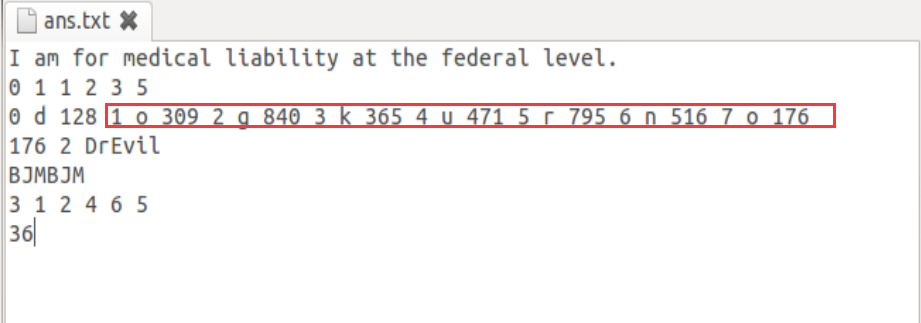
那么**\*(\*(x+4)+8)+8**就应该是最终的地址，也就是我们应该的输入值**input**，

因为**input=\*(\*(x+4)+8)+8**时才会返回0。

而根据上面这张图，就是先访问左孩子，再访问两次右孩子。

**<撒花完结>**

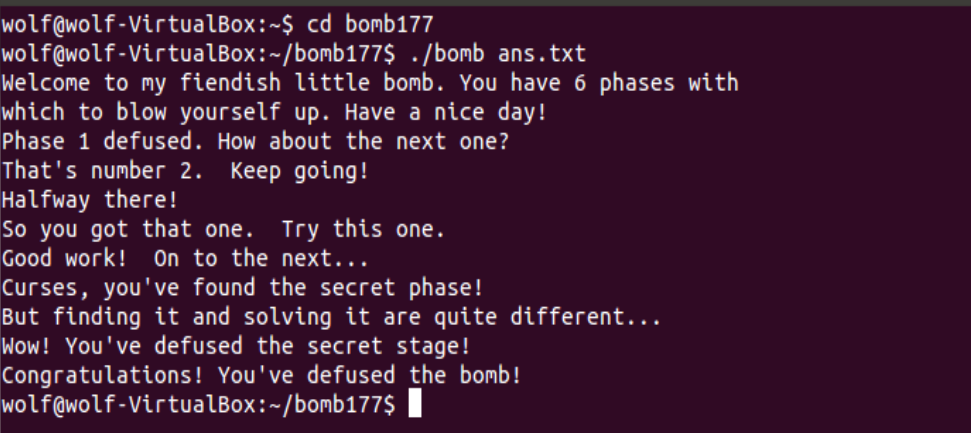
将所有答案保存进ans.txt内



（红框内可去掉，该题多解，只取一组即可）

（本来第四组也可以多解，但是要进入隐藏关，就只写了一组）

运行，可以得到所有炸弹都被成功拆除的提示。



至此，bomb-lab算是正式结束了。

**附录：自己写的各部分phase的c代码**

void bomb()

{

}

void phase\_1()

{

    string s\_ori = "I am for medical liability at the federal level.";

    string s\_input;

    scanf("%s", &s\_input);

    if (strings\_not\_equal(s\_input, s\_ori))

        bomb();

}

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)

        bomb();

    if (m[1] != 1)

        bomb();

    for (int i = 2; i <= 5; i++)

    {

        int temp = m[i - 1] + m[i - 2];

        if (temp != m[i])

            bomb();

    }

}

void phase\_3()

{

    int n;

    char c;

    int m;

    if (scanf("%d %c %d", n, c, m) <= 2)

        bomb();

    if (n > 7)

        bomb();

    char temp;

    switch (n)

    {

    case 0:

        if (m != 0x80)

            bomb();

        temp = 0x64;

        break;

    case 1:

        if (m != 0x135)

            bomb();

        temp = 0x6f;

        break;

    case 2:

        if (m != 0x348)

            bomb();

        temp = 0x67;

        break;

    case 3:

        if (m != 0x16d)

            bomb();

        temp = 0x6b;

        break;

    case 4:

        if (m != 0x1d7)

            bomb();

        temp = 0x75;

        break;

    case 5:

        if (m != 0x31b)

            bomb();

        temp = 0x72;

        break;

    case 6:

        if (m != 0x204)

            bomb();

        temp = 0x6e;

        break;

    case 7:

        if (m != 0xb0)

            bomb();

        temp = 0x6f;

        break;

    }

    if (temp != c)

        bomb();

}

int func4(int n, int x)

{

    if (n == 0)

        return 0;

    if (n == 1)

        return x;

    return (result(n - 1, x) + result(n - 2, x) + x);

}

void phase\_4()

{

    int check, x;

    if (scanf("%d %d", check, x) != 2)

        bomb();

    if (x <= 1)

        bomb();

    if (x > 4)

        bomb();

    if (func4(9, x) != check)

        bomb();

}

void phase\_5()

{

    string s;

    scanf("%s", s);

    if (string\_length(s) != 6)

        bomb();

    int total = 0;

    for (int i = 0; i < 6; i++)

    {

        total += m[s[i] & 0xf];

    }

    if (total != 0x38)

        bomb();

}

void phase\_6()

{

    int a[6];

    scanf("%d %d %d %d %d %d", &a[0], &a[1], &a[2], &a[3], &a[4], &a[5]);

    //  第一阶段：合规性检验

    if (a[0] > 6)

        bomb();

    for (int i = 1; i < 6; i++)

    {

        if (a[i] > 6)

            bomb();

        for (int j = i; j < 6; j++)

        {

            if (a[j] == a[i - 1])

                bomb();

        }

    }

    // 第二阶段：排序

    struct node

    {

        int data;

        int number;

        node \*next;

    } struct node head;

    for (int i = 0; i < 6; i++)

    {

        if (a[i] > 1) // 实际上这句可以不要，放这里可以看得更清晰一点

        {

            for (int j = 1; j < a[i]; j++)

            {

                head = head->next;

            }

            b[i] = &head;

        }

    }

    // 第三阶段：复写下一状态

    \*b[0]->next = \*b[1];

    \*b[1]->next = \*b[2];

    \*b[2]->next = \*b[3];

    \*b[3]->next = \*b[4];

    \*b[4]->next = \*b[5];

    // 第四阶段：检验数据域递增性

    struct node \*head;

    for (int i = 5; i > 0; i--)

    {

        if (head->data > head->next->data)

            bomb();

    }

}

int fun7(int \*x, int input)

{

    if (x == 0)

        return -1;

    if (\*x > input)

        return 2 \* fun7(x + 4, input); // 在左孩子中寻找

    else

    {

        if (\*x == input) // 寻找到了

            return 0;

        else                                   //\*x<input

            return 2 \* fun7(x + 8, input) + 1; // 在右孩子中寻找

    }

}

void secret\_phase()

{

    readline();

    long input = strtol(readline(), 0x0, 0xa);

    if (input > 1000)

        bomb();

    if (fun7(0x804c088, input))

        bomb();

}