**课 程 实 验 报 告**

**课程名称： 计算机组成原理**

**专业班级： 软工1809**

**学 号： 800211824**

**姓 名： 杨孟衡**

**指导教师： 胡勇**

**报告日期： 2019.11.23**

# 实验2： 拆弹实验

## 2.1 实验概述

实验目的：增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。

实验目标：需要拆除尽可能多的炸弹。

实验要求：使用gdb调试器和objdump来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。

实验语言：c。

实验环境：linux

## 2.2 实验内容

**2.2.1 阶段1 字符串比较**

1. 任务描述：通过phase\_1的反汇编代码找出要输入的字符串。
2. 实验设计：利用gdb结合断点来动态地分析。
3. 实验过程：

观察phase\_1的反汇编代码，如图2.1.1所示：

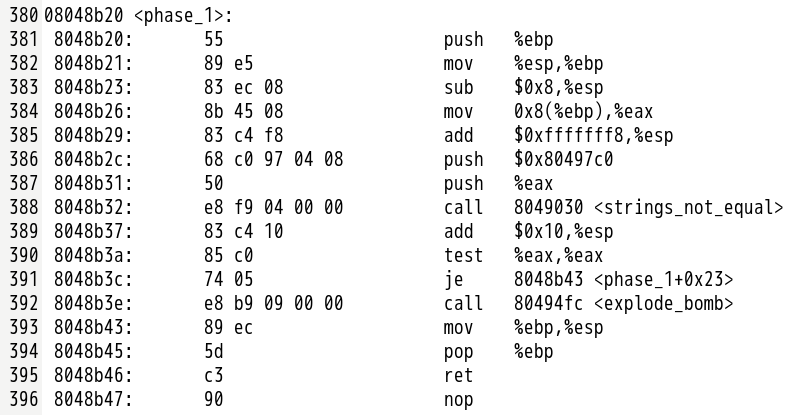


图2.1.1

通过分析，发现在调用strings\_not\_equal对比字符串之前，有两个压栈的指令，其中一个是将函数入参送入栈中，还有一个地址送入了栈，函数入参应该就是输入的字符串所在地址，猜测另一个地址就是正确字符串的首址，于是在phase\_1处下个断点，然后运行，随意输入一个字符，触发断点，再用x命令查看字符串。如图2.1.2所示：

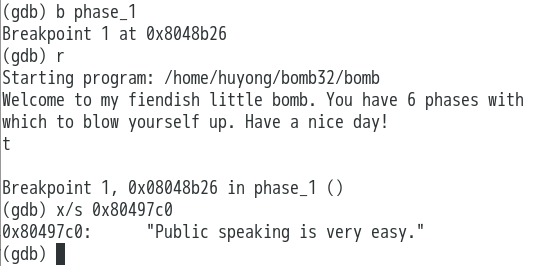


图2.1.2

故猜测“Public speaking is very easy.”就是所需字符串。重新执行该程序，直接输入该字符串，观察结果。如图2.1.3所示：

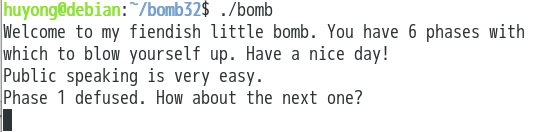


图2.1.3

1. 实验结果：如图2.1.3所示，阶段一拆弹成功！

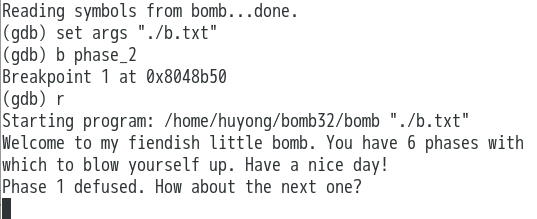
阶段一很简单，找出字符串首地址就可以了，没有什么大的难点。

为了提高调试效率，可以将解析的结果保存到文件中，在调试时以参数方式提供给程序运行，方法如下：

第一步：建立一个参数文件，比如“b.txt”文件，然后用vim或gedit编辑该文件，写入“Public speaking is very easy.”：



然后在gdb中调用该参数文件：



**2.2.2 阶段2 循环**

任务描述：通过phase\_2的反汇编代码推断第二阶段要输入的数据

**2.2.3 阶段3 条件/分支**

任务描述：通过phase\_3的反汇编代码推断第三阶段要输入的数据

**2.2.4 阶段4 递归调用和栈**

任务描述：通过phase\_4以及func4的反汇编代码推断第四阶段要输入的数据

**2.2.5 阶段5 指针**

任务描述：通过phase\_5的反汇编代码推断第五阶段要输入的数据

**2.2.6 阶段6 链表/指针/结构**

任务描述：通过phase\_6的反汇编代码推断第五阶段要输入的数据

**2.2.7 阶段7 隐藏阶段**

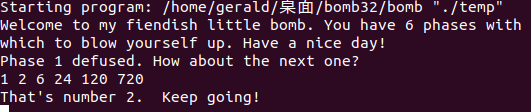
任务描述：找出隐藏阶段开启方式并且拆除隐藏阶段的炸弹。

## 2.3实验过程

阶段1：老师已经做好了示范，就不再赘述；

阶段2：首先反汇编bomb，得到对应汇编代码asm文件，然后仔细观察phase\_2任务函数，可知阶段2 是输入六个数字，若全对则拆除炸弹，分析汇编代码可知，存在一个整形数组a[]用于临时存储读入的六个数字，开始循环之前，先将a[0]置为1，然后除第一个元素之外，每一个元素都满足表达式：a[i] = a[i -1] \* (i + 1);执行5次，所以退出循环的条件为<=5;由表达式可知，当数组a[]读入的六个元素为1，2，6，24，120，720时符合拆弹要求，故拆弹成功！

运行结果：



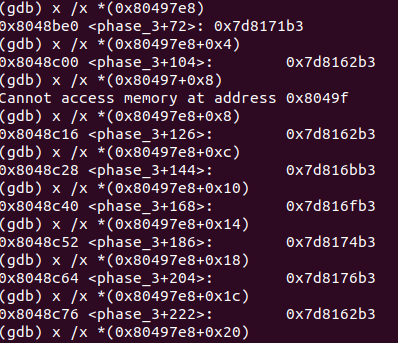
阶段3：

观察汇编代码，发现调用了0x8048860<sscanf@plt>函数，这是一个读入函数相当于scanf一样，而scanf函数是有一个返回值的，这个返回值等于读入的数据项数，比如读入了三个数据就返回3，所以分析可知phase\_3读入了3个数据项，然后再仔细一点点就可以发现（把栈帧画出来），这三个数据很像int,char,unsigned int，为了验证猜测，利用gdb调试器，观察字符串0x80497de的内容（如下第一张图所示），发现这是scanf函数的参数，也即读入的是一个整形，一个字符型，一个整形；

继续往下走，发现代码设置了一个爆炸条件，就是第一个输入的整形值被看作一个无符号整形数，且不能大于7，也即第一个整形数只能输入0-7中的任意一个数，推测本题目可能有多个答案！接着观察到之后的代码一大段都是cmp+jmp的条件跳转组合，推测可能是if-else多重或者switch语句，联想到第一个输入导致跳向不同的地址，于是利用gdb调试器查看对应存储单元的内容（存储单元内容第二张图），发现每4个字节对应一个地址，switch跳转表！于是到这里整个函数已经清晰了，就是一个switch跳转表的应用，根据第一个输入跳转到不同的分支执行，仔细观察每个地址的cmp语句即可得出最终答案，一共有七个，如下图所示。

存储单元内容：



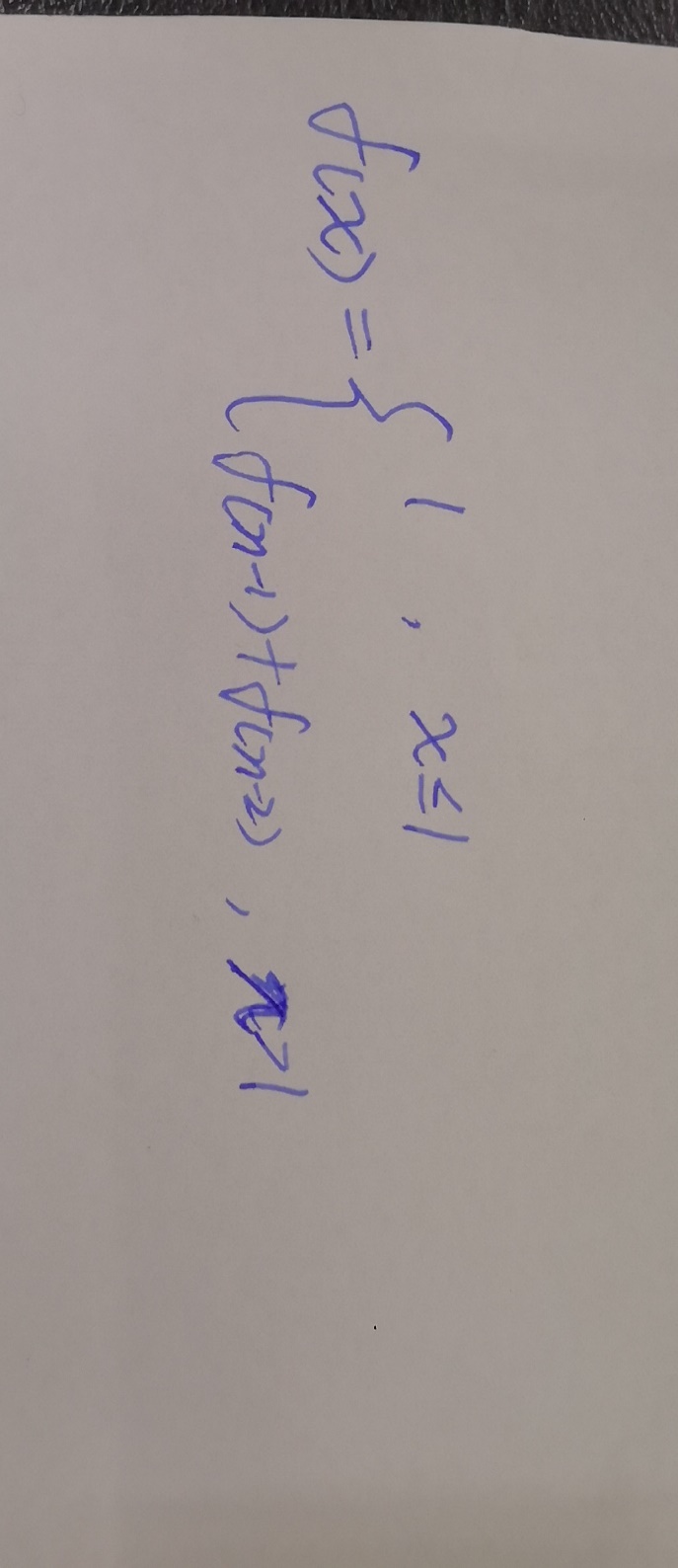


最终答案：

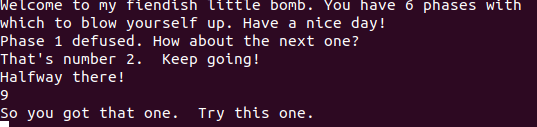
|  |  |
| --- | --- |
|  |  |
|  |  |

阶段4：  
本阶段是个人认为最简单的阶段，就是一个简单的递归函数，仔细观察汇编代码phase\_4和func4函数即可，没什么技术点。递归结束的条件为当入参x小于等于1时，返回值1退出递归，若不小于等于1则对x-1和x-2分别递归，再仔细观察代码引爆炸弹的条件可发现当输入的x的递归值不等于55时，炸弹爆炸，由此可知应输入9！

函数表达式如图：



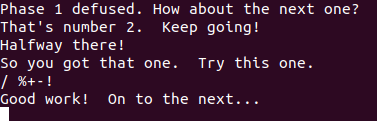
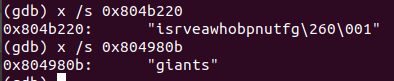
运行结果：



阶段5：

观察前面这部分的代码可以发现要求我们输入一个长度为6的字符串（字符数组），首地址在EBP+8处；观察代码，可以发现其中有两个特殊的地址0x804b220，0x804980b，利用gdb调试器观察可以发现是两个字符串，然后继续看代码，中段有一部分代码实现的是将我们输入的数组元素依次取出然后保留低4位，根据保留下来的低4位值到0x804b220地址对应的字符串中取对应当前edx偏移量的字符，然后把这个字符再送回我们输入的字符数组里对应的edx位置，再观察整个函数引爆炸弹的的条件，发现如果我们从0x804b220中取得的字符顺序是giants，那么通过函数<strings\_not\_equal>后，由于我们输入的字符数组处理后与giants字符串相同，所以返回值为1（送至eax里），然后对eax做test操作，由于返回值为1，test操作后置ZF位为0，所以满足0x8048d85地址对应的跳转指令，成功拆除炸弹！也即若经过处理后的字符数组不为giants，就会引爆炸弹，因而phase\_5拆除失败。

分析到此，答案呼之欲出，输入一个长度为6的字符串，同时每个字符的低4位取出来后应该是对应的0x804b220字符串中的我们需要的字符，第一个应取g，所以对应低4位为f，第二个取i，对应低4位为0，第三个取a，对应低4位为5，第四个取n，对应低4位为b，第五个取t，对应低4位为d，第六个取s，对应低4位为1，而高位不影响结果，所以我采用值为2f 20 25 2b 2d 21的字符串序列/ %+-!,中间有个空格（0x20）！



阶段6：（EIZ寄存器为伪寄存器，其值为0）

观察phase\_6代码段，可以发现一开始调用了函数<read\_six\_numbers>所以我们应该是输入六个数字，并存储为一个整形数组，首地址为ebp-18，继续观察代码，可以发现第一个循环是一个嵌套循环，该段指令地址为dc0-e00,这一段地址的指令为第一个嵌套循环，目的是判断我们输入的六个数字是否都小于等于6，同时每个数字不重复，也就是说我们需要输入1-6这6️个数字，并且不重复，详细说明内层循环为指令地址0x8048dd9-dfa，本内层循环是用于判断输入的六个数字中前五个数字不大于6，同时前五个数字没有重复；

第二段循环地址为e02-e42,这又是一个嵌套循环！一步一步观察代码，发现内层循环e26-e36是每一次循环取输入数据的数组元素，判断当前结点序号是否是与数据相等，直到找到对应序号的链表，然后将它的首地址取至esi寄存器保存结束后内层循环，外层循环确保每个数组元素都能遍历到并且将数组元素对应的结点数据域存储至地址为ebp-3c的存储单元中的内容，lea指令和mov指令最大的区别就是一个传地址，一个传值，所以看代码可以发现，ebp-3c的存储单元放的是ebp-30这个存储单元的地址，也就是说，其实我们是从ebp-30这个存储单元开始存放取出来的每个结点的数据值，也就是说这一个嵌套循环作用就是根据我们输入的数据，将每一个数组元素对应的结点的数据取到对应的存储单元地址，以便后续代码处理；

继续观察代码，第三段循环指令为e44-e5e，从中发现第二段循环存储的结点数据值数组（首地址为ebp-30）在地址为e44-e5e的指令摧残下，被依次取出至链表结点的数据域中，一个一个的取，一个一个结点的数据域中存，直到都存完，第三段循环的任务就完成了；

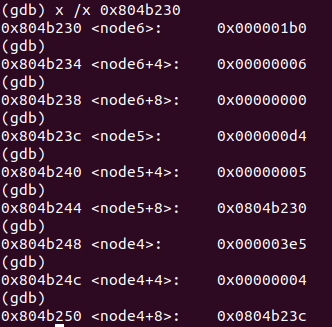
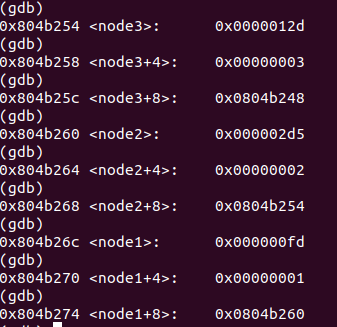
最后一段循环为e60-e77,仔细观察可以知道这一段代码是判断当前循环的结点数据值是否比下一个结点的的数据值大，若大那么进入下一次循环，否则BOOM!

到这里，phase\_6所有代码已经完全解析透彻，总结如下：

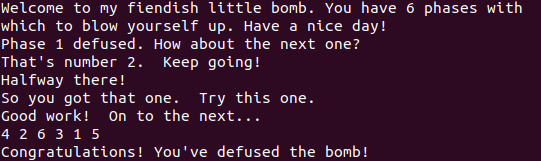
根据我们输入的6个数字存储为整形数组，且这六个数字一定要在1-6里，不可重复，根据我们输入的数字顺序不同，不一定为1-6的顺序，从源结点取出的数据值也会不同，因而每个结点的数据域作相应处理后也会不同，最后对每个结点遍历一次，判断当前结点的数据值是否大于下一个结点的数据值，若我们输入的数字顺序正好满足，那么炸弹成功拆除，否则BOOM！（答案为4 2 6 3 1 5）

gdb调试观察特殊内存单元内容后总结下表，数据来源如表格后图片所示；

|  |  |  |  |
| --- | --- | --- | --- |
| 链表（首地址） | 数据域(+0) | 序号域(+4) | next指针域(+8) |
| node1(0x0804 b26c) | 0x0000 00fd | 0x0000 0001 | 0x0804 b260 |
| node2(0x0804 b260) | 0x0000 02d5 | 0x0000 0002 | 0x0804 b254 |
| node3(0x0804 b254) | 0x0000 012d | 0x0000 0003 | 0x0804 b248 |
| node4(0x0804 b248) | 0x0000 03e5 | 0x0000 0004 | 0x0804 b23c |
| node5(0x0804 b23c) | 0x0000 00d4 | 0x0000 0005 | 0x0804 b230 |
| node6(0x0804 b230) | 0x0000 01b0 | 0x0000 0006 | 0x0000 0000(NULL) |

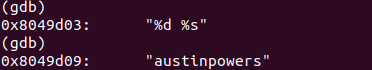


运行结果：



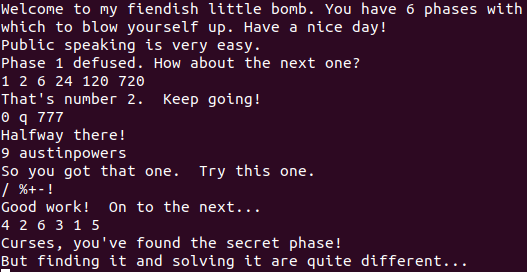
阶段7：

首先阐述一下如何找到secret\_phase的：老师上课的时候提到有这个隐藏阶段，实验报告里也写了，老师还说在phase\_4，后面做完了phase\_1-6都没发现，就很奇怪，然后就去看代码，发现多了一个fun7和secret\_phase，于是就去看main函数找入口，又没找到，然后接着看代码，直到看到了phase\_defused函数，发现这函数体内有很多push某个地址的操作，很奇怪，就用gdb调试器查看了这些地址对应的内容，结果就发现了隐藏阶段7，如图所示；



通过调试器观察以及代码检查，发现了在phase\_4阶段如果我们多输入一个字符串”austinpower”，那么就可以触发特殊条件，进入隐藏阶段。

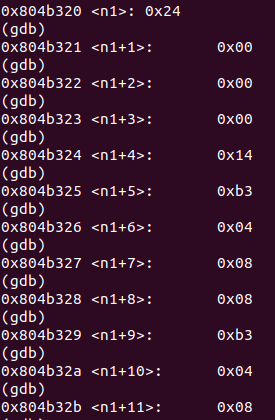
然后输入测试：



发现果然进入了隐藏阶段，正式开始解析隐藏阶段；

首先第一段代码（ee8-eef）是读入一行数据，也就是需要我们输入的数据，之后调用了C语言库函数strtol(eax,null,10),解读下来就是这个函数是用来在一个字符串中找数字，然后返回这个数字的数值，比如输入了“015ac”就返回015这个数值（按照第三个参数的值，10就代表10进制），所以也就是说在隐藏阶段需要我们输入一串字符串但是只需要整数部分，也就是说我们输入一个10进制整数就行，确定输入的数据是啥了，再观察下面的代码，可以发现如果我们输入的这个数大于等于1001那么就BOOM！所以我们输入的数还不能大于1001，最大为1001！

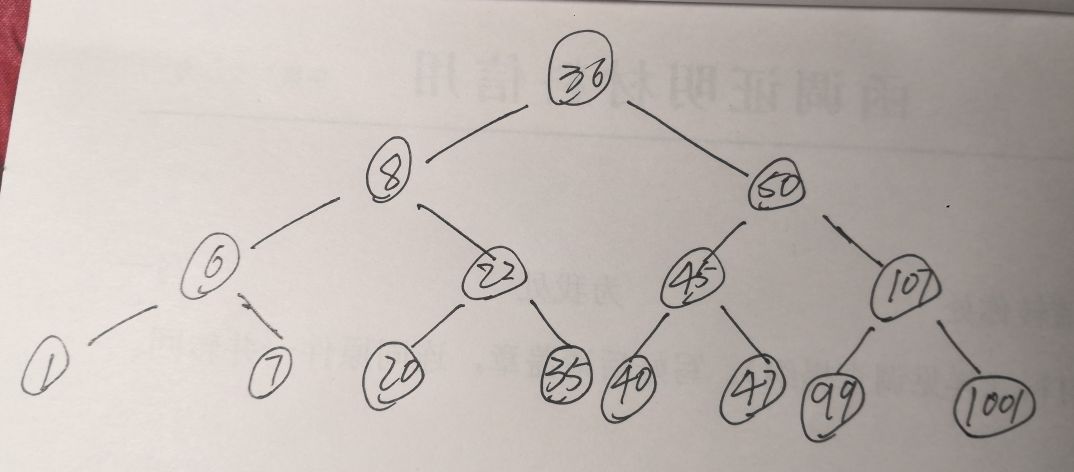
进入fun7函数前，push了两个参数，一个是0x804b320，一个是ebx（我们输入的数值），然后做了这么多的实验给我的经验告诉我，这个0x804b320一定不简单，所以我就仔细的查看了这个地址，得到了如图所示的很多很多很多数据…



发现这是一个容量为12的数组，再仔细一点点就能发现，每四个为一个类型，再仔细一丢丢，你就能发现，前四个字节是数据，中间四个字节是地址，后四个字节也是地址，也就是说，这是一个结构体，包含一个int型数据域，两个指针域，结合数据结构，这是二叉树…..真的没想到……….我菜还不行吗！

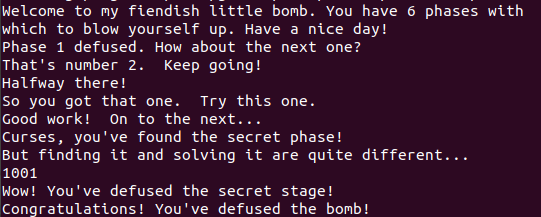
所以利用gdb调试器及二叉树的知识，就可以把下面这个表格列出来，得到所有的二叉树，然后画出如下下图所示的二叉树结构；

|  |  |  |  |
| --- | --- | --- | --- |
| 结点 | 数据域 | 左孩子 | 右孩子 |
| n1(0x804b320) | 36 | 0x804b314 | 0x804b308 |
| n21(0x804b314) | 8 | 0x804b2e4 | 0x804b2fc |
| n22(0x804b308) | 50 | 0x804b2f0 | 0x804b2d8 |
| n31(0x804b2e4) | 6 | 0x804b2c0 | 0x804b29c |
| n32(0x804b2fc) | 22 | 0x804b290 | 0x804b2a8 |
| n33(0x804b2f0) | 45 | 0x804b2cc | 0x804b284 |
| n34(0x804b2d8) | 107 | 0x804b2b4 | 0x804b278 |
| n41(0x804b2c0) | 1 | null | null |
| n42(0x804b29c) | 7 | null | null |
| n43(0x804b290) | 20 | null | null |
| n44(0x804b2a8) | 35 | null | null |
| n45(0x804b2cc) | 40 | null | null |
| n46(0x804b284) | 47 | null | null |
| n470x804b2b4) | 99 | null | null |
| n48(0x804b278) | 1001 | null | null |



到这里基本工作都完成了，接下来观察fun7函数，既然是个递归，这时我不免想到这是不是二叉树的查询……，带着这样的想法看fun7代码，发现还真的是…，两个参数，第一个是根结点，第二个是要查询的元素…..，分析fun7代码可知，初始化eax为0，若当前根结点为null，则返回-1，结束递归，另外当我们查询的元素值>=根结点值时，递归查询当前根结点的右子树，也就是将当前根结点首地址+8做入参，以及查询的元素值，并且返回2\*eax+1值，若查询的元素值<根结点值时递归查询当前根结点的左子树，将当前根结点首地址+4做入参，以及查询的元素值，并且返回2\*eax值，若相等则返回0，解析fun7代码完成，回到<secret\_phase>,观察最后的拆除炸弹条件，发现若执行了fun7函数后，返回值eax不等于7那么BOOM！！你想想，7=2\*3+1，这说明了啥，不断对右子树查找，并且查找的这个元素的度应该为4，怎么得来的？听我说，首先找到这个元素返回0吧（度+1=1），然后这个元素的根结点返回的是2\*0+1=1吧（度+1=2），这个根结点再返回一次那就是2\*1+1=3吧（度+1=3），再返回一次2\*3+1=7（度+1=4），由此由此得到结论，需要一个不断查询右子树，同时度为4的叶子结点，由上图可知，1001结点刚好符合，至此，隐藏阶段炸弹拆除！

运行结果：



## 2.4实验小结

本次实验非常有意义，首先第一点感受最深的就是机器级语言的不便性，汇编代码确实可读性不如高级程序语言，但同时效率更高，看汇编代码就像是自己的脑袋是计算机，一条一条的执行，如果跟不上就只有拿笔写了，不然真的是不知道哪个寄存器是啥，地址里是啥，高级程序设计语言里简易的循环语句，或者是条件语句，在汇编代码里会有很多篇幅去实现，但基本结构都差不多，通过这次实验，我还提高了自己的汇编语言水平，看了那么多代码，不会写真的是对不起自己了！

第二点感受就是这个实验的重要性和目的，计算机组成原理是一门较难的课程，很抽象，但是通过这些实验，就没那么抽象了，让我更深刻的认识到了计算机底部如何实现，理解了很多之前似懂非懂的问题。

第三点就是这个实验很有趣啊！很喜欢！计组的实验真的是上大学以来最有趣的，闯关很有成就感！谢谢老师！