

课程实验报告

课 程 名 称： 计算机系统试验

实验项目名称： BombLab

专 业 班 级： 计算机科学

姓 名： 周思宇

学 号： 201608030201

指 导 教 师：

完 成 时 间： 2018 年 4 月 29 日

信息科学与工程学院

|  |  |
| --- | --- |
| **实验题目：**bomblab | |
| **实验目的：**本次实验为熟悉汇编程序及其调试方法的实验。 | |
| **实验资源：Ubuntu环境，PC** | |
| **实验任务：**  实验内容包含2个文件bomb（可执行文件）和bomb.c（c源文件）。  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。  实验需要用到gdb工具，可到网上查找gdb使用方法和参数。   1. **第一关** 2. 原理设计：   需要了解栈的原理，读书P149-150，还需要会进入gdb调试状态，并能够查看某地址处的数据。   1. 实现与分析：   08048f61 <phase\_1>:  8048f61: 55 push %ebp  8048f62: 89 e5 mov %esp,%ebp  8048f64: 83 ec 18 sub $0x18,%esp  首先是为phase1这个函数开辟一个帧，申请24个内存空间  8048f67: c7 44 24 04 5c a1 04 movl $0x804a15c,0x4(%esp)  内存0x8048f67位置的movl指令将值0x804a15c放入内存的0x4+%esp位置  8048f6e: 08  8048f6f: 8b 45 08 mov 0x8(%ebp),%eax  接下来的mov指令从调用phase\_1函数的上级函数中获取参数，参数的位置是0x8+%ebp，并存入寄存器eax。  8048f72: 89 04 24 mov %eax,(%esp)  下一步将寄存器eax中的参数放入一个内存地址单元中，而这个单元的地址被存放在寄存器esp中，操作数的内存地址被存放在寄存器中，这样的行为称为寄存器间接寻址。其实也就是把0x8(%ebp)放到esp  8048f75: e8 31 00 00 00 call 8048fab <strings\_not\_equal>  8048f7a: 85 c0 test %eax,%eax  实现的功能是判断eax中的值是否为0，若为0，则跳转到0x8048f83函数结束，否则就会调用80490d1位置的<explode\_bomb>函数，炸弹爆炸。  而这个eax就是上面那个函数的返回值  8048f7c: 74 05 je 8048f83 <phase\_1+0x22>  8048f7e: e8 4e 01 00 00 call 80490d1 <explode\_bomb>  8048f83: c9 leave  8048f84: c3 ret  8048f85: 90 nop  8048f86: 90 nop  8048f87: 90 nop  8048f88: 90 nop  8048f89: 90 nop  8048f8a: 90 nop  8048f8b: 90 nop  8048f8c: 90 nop  8048f8d: 90 nop  8048f8e: 90 nop  8048f8f: 90 nop  可以推测出，进行比较的两个字符串分别是来自内存地址为0x804a15c位置的字符串和phase\_1上级函数的参数。进入gdb调试，对内存0x804a15c位置内容以字符串类型查看       1. 数据记录：完成帧堆栈数据记录表（1-3关）   IMG_256   1. 结论   对于反汇编代码中的立即数，一定要留意！！！   1. **第二关** 2. 原理设计：   能够分辨出循环，知道该怎么分析循环。这题没有用到设置断点什么的。   1. 实现与分析：   08048d6a <phase\_2>:  8048d6a: 55 push %ebp  8048d6b: 89 e5 mov %esp,%ebp  8048d6d: 56 push %esi  8048d6e: 53 push %ebx  8048d6f: 83 ec 30 sub $0x30,%esp  8048d72: 8d 45 e0 lea -0x20(%ebp),%eax  将%ebp-0x20位置的地址放入寄存器eax  8048d75: 89 44 24 04 mov %eax,0x4(%esp)  寄存器eax中的值（%ebp-0x20位置的地址）放入内存地址的0x4+%esp位置（通过寄存器作为中转在内存之间传递值）  8048d79: 8b 45 08 mov 0x8(%ebp),%eax  调用phase\_2的函数的返回地址传入phase\_2中，在内存的0x8+%ebp位置存放着上一级函数的返回地址，将返回地址存放在寄存器eax中  8048d7c: 89 04 24 mov %eax,(%esp)  8048d7f: e8 87 03 00 00 call 804910b <read\_six\_numbers>  推断出这一关的基本要求是输入6个数字  8048d84: 83 7d e0 00 cmpl $0x0,-0x20(%ebp)  将0和%ebp-0x20位置的值进行比较，若不相等则跳转至0x8048d90位置，往下看发现0x8048d90位置为<explode\_bomb>函数，即炸弹爆炸  8048d88: 75 06 jne 8048d90 <phase\_2+0x26>  8048d8a: 83 7d e4 01 cmpl $0x1,-0x1c(%ebp)  知道第二个参数放在内存的%ebp-0x1c位置且必须为1。到这里就已经推断出前两个参数的值为0和1  8048d8e: 74 05 je 8048d95 <phase\_2+0x2b>  8048d90: e8 3c 03 00 00 call 80490d1 <explode\_bomb>  8048d95: 8d 5d e8 lea -0x18(%ebp),%ebx  8048d98: 8d 75 f8 lea -0x8(%ebp),%esi  8048d9b: 8b 43 fc mov -0x4(%ebx),%eax  从这里开始，用模拟帧栈模拟图示意  /Users/zsy/Downloads/IMG_1913.JPG  8048d9e: 03 43 f8 add -0x8(%ebx),%eax  8048da1: 39 03 cmp %eax,(%ebx)  8048da3: 74 05 je 8048daa <phase\_2+0x40>  8048da5: e8 27 03 00 00 call 80490d1 <explode\_bomb>  8048daa: 83 c3 04 add $0x4,%ebx  8048dad: 39 f3 cmp %esi,%ebx  8048daf: 75 ea jne 8048d9b <phase\_2+0x31>  8048db1: 83 c4 30 add $0x30,%esp  8048db4: 5b pop %ebx  8048db5: 5e pop %esi  8048db6: 5d pop %ebp  8048db7: c3 ret  以下是我在草稿上做的标记    最后解决得出0 1 1 2 3 5     1. 数据记录：完成帧堆栈数据记录表（1-3关）   ../../../../Downloads/2236E4C16DDDE9F79EA30811F25420B0.pn   1. 结论   比较重要的是会分析循环条件。而且要弄清楚什么时候停止循环。   1. **第三关** 2. 原理设计：此题要用到gdb调试，将第一个参数设定为0，第二个参数设定为一个任意值（因为还不知道将会执行何种操作，并且在gdb调试的过程中，如果没有进行到0x8048f58位置则暂时不会引爆炸弹），这里选择10。打开调试功能，将断点设定在phase\_3函数位置，输入r开始调试，首先要将第1、2关的正确答案输入，否则将会引爆炸弹，接下来输入两个参数0和10（剩下具体调试内容在b中详细写） 3. 实现与分析：   8048ea1: 55 push %ebp  8048ea2: 89 e5 mov %esp,%ebp  8048ea4: 83 ec 28 sub $0x28,%esp  8048ea7: 8d 45 f0 lea -0x10(%ebp),%eax  8048eaa: 89 44 24 0c mov %eax,0xc(%esp)  8048eae: 8d 45 f4 lea -0xc(%ebp),%eax  8048eb1: 89 44 24 08 mov %eax,0x8(%esp)  8048eb5: c7 44 24 04 3e a2 04 movl $0x804a23e,0x4(%esp)  8048ebc: 08  8048ebd: 8b 45 08 mov 0x8(%ebp),%eax  8048ec0: 89 04 24 mov %eax,(%esp)  8048ec3: e8 78 f9 ff ff call 8048840 <\_\_isoc99\_sscanf@plt>  0x8048ec3开始读入数据，从0x8048ea7和0x8048eae可以看出，两个参数分别位于-0x10(%ebp)位置和-0xc(%ebp)位置，分别设为val2和val1（val1的输入顺序在val2之前）  8048ec8: 83 f8 01 cmp $0x1,%eax  8048ecb: 7f 05 jg 8048ed2 <phase\_3+0x31>  8048ecd: e8 ff 01 00 00 call 80490d1 <explode\_bomb>  8048ed2: 83 7d f4 07 cmpl $0x7,-0xc(%ebp)  从0x8048ed2位置的cmpl指令分析，-0xc(%ebp)位置的参数val1应该是一个不大于7的数字，否则，程序将跳转到0x8048f43位置的<explode\_bomb>，即引爆炸弹。  8048ed6: 77 6b ja 8048f43 <phase\_3+0xa2>  8048ed8: 8b 45 f4 mov -0xc(%ebp),%eax  8048edb: ff 24 85 a0 a1 04 08 jmp \*0x804a1a0(,%eax,4)  8048ee2: b8 00 00 00 00 mov $0x0,%eax  8048ee7: eb 53 jmp 8048f3c <phase\_3+0x9b>  8048ee9: b8 00 00 00 00 mov $0x0,%eax  8048eee: 66 90 xchg %ax,%ax  8048ef0: eb 45 jmp 8048f37 <phase\_3+0x96>  8048ef2: b8 00 00 00 00 mov $0x0,%eax  8048ef7: eb 39 jmp 8048f32 <phase\_3+0x91>  8048ef9: b8 00 00 00 00 mov $0x0,%eax  8048efe: 66 90 xchg %ax,%ax  8048f00: eb 2b jmp 8048f2d <phase\_3+0x8c>  8048f02: b8 00 00 00 00 mov $0x0,%eax  8048f07: eb 1f jmp 8048f28 <phase\_3+0x87>  8048f09: b8 00 00 00 00 mov $0x0,%eax  8048f0e: 66 90 xchg %ax,%ax  8048f10: eb 11 jmp 8048f23 <phase\_3+0x82>  8048f12: b8 14 03 00 00 mov $0x314,%eax  8048f17: eb 05 jmp 8048f1e <phase\_3+0x7d>  8048f19: b8 00 00 00 00 mov $0x0,%eax  8048f1e: 2d 5a 03 00 00 sub $0x35a,%eax  8048f23: 05 ef 02 00 00 add $0x2ef,%eax  8048f28: 2d 16 02 00 00 sub $0x216,%eax  8048f2d: 05 16 02 00 00 add $0x216,%eax  8048f32: 2d 16 02 00 00 sub $0x216,%eax  8048f37: 05 16 02 00 00 add $0x216,%eax  8048f3c: 2d 16 02 00 00 sub $0x216,%eax  8048f41: eb 0a jmp 8048f4d <phase\_3+0xac>  8048f43: e8 89 01 00 00 call 80490d1 <explode\_bomb>  8048f48: b8 00 00 00 00 mov $0x0,%eax  8048f4d: 83 7d f4 05 cmpl $0x5,-0xc(%ebp)  8048f51: 7f 05 jg 8048f58 <phase\_3+0xb7>  8048f53: 3b 45 f0 cmp -0x10(%ebp),%eax  所以输入的第一个参数val1是不大于7的一个整数，第二个参数val2在0x8048f53位置出现，与储存在eax中的计算结果进行比较，若不相等则跳转至0x8048f58位置，同样是引爆的结果，所以，参数1经过中间的一系列计算得到的结果必须和参数2相等，否则将引爆炸弹。  8048f56: 74 05 je 8048f5d <phase\_3+0xbc>  8048f58: e8 74 01 00 00 call 80490d1 <explode\_bomb>  8048f5d: c9 leave  8048f5e: 66 90 xchg %ax,%ax  8048f60: c3 ret    从地址08048eaa开始，两个数我就先设置为1，100，100是我瞎设置的，而1是我根据地址【8048ed2】的值第一个参数需要<7，于是我选择了1.  继续往下看    到edb的时候停止，因为有jmp了。看一下寄存器：    eax已经把我的第一个参数1存好啦  继续往下，到地址【8048f53】停下，查看寄存器  因为那个地址进行的是比较，ebp-10是不是等于eax里的值？？  Ebp-10是我第二个参数，我之前输入的是100，而看下图的eax：    是-641  所以下次我要是输入1 -641的话，就不会爆炸啦    第一个数可以取 0 1 2 3 4 5，分别有对应的第二个数   1. 数据记录：完成帧堆栈数据记录表（1-3关）   ../../../../Downloads/IMG_1914.JP  ../../../../Downloads/IMG_2140.PN   1. 结论   我认为这道题的关键是能分辨出有两个数字是我要输入的。  后面的一大串运算公式能把他们打包在一起，不能让它们扰乱思维，产生抵触情绪  设置断点单步运行，对照着代码来看地址运行到哪一步了  最最最重要的是要格外注意跳转！！！！！！   1. **第四关** 2. 原理设计：   第四题是最有意思的！！！！！！！！在破解的过程中发现这是一个递归函数，但是体现在汇编语言中就显得特征不是那么明显，而且在哪里循环，如何返回真的很难找！！！！！！所以采用了一种将汇编转化成与之等价的C语言语句，这样有助于更好的看清程序的运作过程。   1. 实现与分析：   08048e2e <phase\_4>:  8048e2e: 55 push %ebp  8048e2f: 89 e5 mov %esp,%ebp  8048e31: 83 ec 28 sub $0x28,%esp  8048e34: 8d 45 f0 lea -0x10(%ebp),%eax  8048e37: 89 44 24 0c mov %eax,0xc(%esp)  8048e3b: 8d 45 f4 lea -0xc(%ebp),%eax  8048e3e: 89 44 24 08 mov %eax,0x8(%esp)  8048e42: c7 44 24 04 3e a2 04 movl $0x804a23e,0x4(%esp)  8048e49: 08  8048e4a: 8b 45 08 mov 0x8(%ebp),%eax  8048e4d: 89 04 24 mov %eax,(%esp)  8048e50: e8 eb f9 ff ff call 8048840 <\_\_isoc99\_sscanf@plt>  8048e55: 83 f8 02 cmp $0x2,%eax  8048e58: 75 0c jne 8048e66 <phase\_4+0x38>  8048e5a: 8b 45 f4 mov -0xc(%ebp),%eax  8048e5d: 85 c0 test %eax,%eax  8048e5f: 78 05 js 8048e66 <phase\_4+0x38>  8048e61: 83 f8 0e cmp $0xe,%eax  8048e64: 7e 05 jle 8048e6b <phase\_4+0x3d>  8048e66: e8 66 02 00 00 call 80490d1 <explode\_bomb>  8048e6b: c7 44 24 08 0e 00 00 movl $0xe,0x8(%esp)  8048e72: 00  8048e73: c7 44 24 04 00 00 00 movl $0x0,0x4(%esp)  8048e7a: 00  8048e7b: 8b 45 f4 mov -0xc(%ebp),%eax  8048e7e: 89 04 24 mov %eax,(%esp)  8048e81: e8 da fc ff ff call 8048b60 <func4>  8048e86: 83 f8 01 cmp $0x1,%eax  8048e89: 75 06 jne 8048e91 <phase\_4+0x63>  8048e8b: 83 7d f0 01 cmpl $0x1,-0x10(%ebp)  8048e8f: 74 0c je 8048e9d <phase\_4+0x6f>  8048e91: 8d b4 26 00 00 00 00 lea 0x0(%esi,%eiz,1),%esi  8048e98: e8 34 02 00 00 call 80490d1 <explode\_bomb>  8048e9d: c9 leave  8048e9e: 66 90 xchg %ax,%ax  8048ea0: c3 ret  明显看到，phase4有调用一个函数，  把函数搜到，先研究这个函数  08048b60 <func4>:  8048b60: 55 push %ebp  8048b61: 89 e5 mov %esp,%ebp  8048b63: 83 ec 18 sub $0x18,%esp  8048b66: 89 5d f8 mov %ebx,-0x8(%ebp)  8048b69: 89 75 fc mov %esi,-0x4(%ebp)  先查看内存，我输入的数字11已经存在了内存里面。    8048b6c: 8b 55 08 mov 0x8(%ebp),%edx    8048b6f: 8b 45 0c mov 0xc(%ebp),%eax    8048b72: 8b 5d 10 mov 0x10(%ebp),%ebx  第二个输入的数据也存进去了。    8048b75: 89 d9 mov %ebx,%ecx  8048b77: 29 c1 sub %eax,%ecx  8048b79: 89 ce mov %ecx,%esi  8048b7b: c1 ee 1f shr $0x1f,%esi 就是esi  8048b7e: 8d 0c 0e lea (%esi,%ecx,1),%ecx  8048b81: d1 f9 sar %ecx  8048b83: 01 c1 add %eax,%ecx  8048b85: 39 d1 cmp %edx,%ecx  8048b87: 7e 17 jle 8048ba0 <func4+0x40>  调用自己  8048b89: 83 e9 01 sub $0x1,%ecx  减  8048b8c: 89 4c 24 08 mov %ecx,0x8(%esp)  8048b90: 89 44 24 04 mov %eax,0x4(%esp)  8048b94: 89 14 24 mov %edx,(%esp)  8048b97: e8 c4 ff ff ff call 8048b60 <func4>  8048b9c: 01 c0 add %eax,%eax  8048b9e: eb 20 jmp 8048bc0 <func4+0x60>  调用自己  8048ba0: b8 00 00 00 00 mov $0x0,%eax  8048ba5: 39 d1 cmp %edx,%ecx  8048ba7: 7d 17 jge 8048bc0 <func4+0x60>  调用自己  8048ba9: 89 5c 24 08 mov %ebx,0x8(%esp)  8048bad: 83 c1 01 add $0x1,%ecx  8048bb0: 89 4c 24 04 mov %ecx,0x4(%esp)  8048bb4: 89 14 24 mov %edx,(%esp)  8048bb7: e8 a4 ff ff ff call 8048b60 <func4>  调用自己  8048bbc: 8d 44 00 01 lea 0x1(%eax,%eax,1),%eax  8048bc0: 8b 5d f8 mov -0x8(%ebp),%ebx  8048bc3: 8b 75 fc mov -0x4(%ebp),%esi  8048bc6: 89 ec mov %ebp,%esp  8048bc8: 5d pop %ebp  8048bc9: c3 ret  明显是一个递归函数，按照这个函数的汇编代码，我写了一个c++程序，把每个寄存器都定义成变量：  Last login: Mon Apr 16 23:16:17 on console  zsydeMacBook-Pro:~ zsy$ vim 4,cpp  zsydeMacBook-Pro:~ zsy$ vim 4,cpp  #include<iostream>  using namespace std;  int func4(int edx,int eax,int ebx)  {  int ecx,esi;  ecx=ebx;  ecx=ecx-eax;  sei=ecx;  esi=esi>>31;  ecx=ecx+esi;  ecx=ecx/2;  ecx=ecx+eax;  if(edx>=ecx)  {  eax=0;  if(edx<=ecx)  {  return eax;  }  else  {  ecx=ecx+1;  eax=func4(edx,ecx,ebx);  eax=eax\*2+1;  return eax;  }  }  else  {  ecx=ecx-1;  eax=func4(edx,eax,ecx);  eax=eax\*2;  return eax;  }  }  int main()  {  for(int i=0;i<=14;i++)  {  if(func4(i,0,14)==1)  {  cout<<i<<endl;  }  }  }  "4,cpp" 46L, 445C  运行程序：      这个得出的结果就是，我第一个数据可以取得到的值：  那分别有8 9 11  于是，结果就是  8 1  9 1  11 1    随便找一个检查一下  成功～     1. 结论   我之前一直很疑惑如何能够得出函数的变量，怎么知道的那三个函数的变量就是我所写的这三个呢？后来问了老师，老师给出的一个特别好的方法就是不要老去纠结汇编语言，而是学着去理解汇编语言，如果不能知道，那就直接查看寄存器。   1. **第五关** 2. 原理设计：   这个的重点就是一定要看出来是数组操作（看到0x8048e0b位置对这个地址进行操作，以4单位为增长），否则会一头雾水。每次被加上的数组元素的下标是不连续的。循环进行了15次加法，且加上去的最后一个元素是15。所以可以通过第一个元素逆推出之前累加了哪些元素。   1. 实现与分析：   08048db8 <phase\_5>:  8048db8: 55 push %ebp  8048db9: 89 e5 mov %esp,%ebp  8048dbb: 56 push %esi  8048dbc: 53 push %ebx  8048dbd: 83 ec 20 sub $0x20,%esp  8048dc0: 8d 45 f0 lea -0x10(%ebp),%eax  8048dc3: 89 44 24 0c mov %eax,0xc(%esp)  8048dc7: 8d 45 f4 lea -0xc(%ebp),%eax  8048dca: 89 44 24 08 mov %eax,0x8(%esp)  8048dce: c7 44 24 04 3e a2 04 movl $0x804a23e,0x4(%esp)  8048dd5: 08  8048dd6: 8b 45 08 mov 0x8(%ebp),%eax  8048dd9: 89 04 24 mov %eax,(%esp)  8048ddc: e8 5f fa ff ff call 8048840 <\_\_isoc99\_sscanf@plt>  8048de1: 83 f8 01 cmp $0x1,%eax  8048de4: 7f 05 jg 8048deb <phase\_5+0x33>  8048de6: e8 e6 02 00 00 call 80490d1 <explode\_bomb>  8048deb: 8b 45 f4 mov -0xc(%ebp),%eax  8048dee: 83 e0 0f and $0xf,%eax  8048df1: 89 45 f4 mov %eax,-0xc(%ebp)  8048df4: 83 f8 0f cmp $0xf,%eax  8048df7: 74 29 je 8048e22 <phase\_5+0x6a>  8048df9: b9 00 00 00 00 mov $0x0,%ecx  8048dfe: ba 00 00 00 00 mov $0x0,%edx  8048e03: bb c0 a1 04 08 mov $0x804a1c0,%ebx  ebx在这个地址  8048e08: 83 c2 01 add $0x1,%edx  8048e0b: 8b 04 83 mov (%ebx,%eax,4),%eax  以ebx为基准，每次都是ebx+4\*eax  8048e0e: 01 c1 add %eax,%ecx  8048e10: 83 f8 0f cmp $0xf,%eax  这一部比较关键，edx从0到f，一共要循环16次  那么，每一次循环都要做什么呢？  通过上面的汇编，发现是在一段内存中，不断的获得某个特定地址里面存的数值  而这个地址是通过基地址加上一个偏移量得到的，基地址就是之前那个ebx存的地址，而偏移量就是当前的eax的值\*4，这个新的值又将被用到下一次偏移量的运算中去。  8048e13: 75 f3 jne 8048e08 <phase\_5+0x50>  8048e15: 89 45 f4 mov %eax,-0xc(%ebp)  8048e18: 83 fa 0f cmp $0xf,%edx  8048e1b: 75 05 jne 8048e22 <phase\_5+0x6a>  8048e1d: 39 4d f0 cmp %ecx,-0x10(%ebp)  ecx里的值和我输入的数据必须相等，而前面已经知道我这里存的是一个加和，是eax的加和  8048e20: 74 05 je 8048e27 <phase\_5+0x6f>  8048e22: e8 aa 02 00 00 call 80490d1 <explode\_bomb>  8048e27: 83 c4 20 add $0x20,%esp  8048e2a: 5b pop %ebx  8048e2b: 5e pop %esi  8048e2c: 5d pop %ebp  8048e2d: c3 ret    在上面那个数组中找我需要的特定数字是哪个。最后得出要的顺序为：  ../../../../Downloads/IMG_1954.JP  所以第一个数字应该是5！  第二个就是他们的总和，也就是115     1. 结论   总结来说，这是一个实现数组元素累加的程序。参数val1被用来计算每次的地址偏移量，初始的val1值由输入给出，参数val2被用来存放累加的结果。   1. **第六关** 2. 原理设计：   上一关是数组，这一关就是链表！太好玩了，当然，最后要把节点内的内容重新进行排序，排成递增序列。这个设置断点查看地址内容是最主要的操作，我会在b中详细说明。   1. 实现与分析：   08048c89 <phase\_6>:  8048c89: 55 push %ebp  8048c8a: 89 e5 mov %esp,%ebp  8048c8c: 57 push %edi  8048c8d: 56 push %esi  8048c8e: 53 push %ebx  8048c8f: 83 ec 5c sub $0x5c,%esp  8048c92: 8d 45 d0 lea -0x30(%ebp),%eax  8048c95: 89 44 24 04 mov %eax,0x4(%esp)  8048c99: 8b 45 08 mov 0x8(%ebp),%eax  8048c9c: 89 04 24 mov %eax,(%esp)  8048c9f: e8 67 04 00 00 call 804910b <read\_six\_numbers>  8048ca4: be 00 00 00 00 mov $0x0,%esi  8048ca9: 8d 7d d0 lea -0x30(%ebp),%edi  8048cac: 8b 04 b7 mov (%edi,%esi,4),%eax  8048caf: 83 e8 01 sub $0x1,%eax  8048cb2: 83 f8 05 cmp $0x5,%eax  8048cb5: 76 05 jbe 8048cbc <phase\_6+0x33>  8048cb7: e8 15 04 00 00 call 80490d1 <explode\_bomb>  eax<=6  8048cbc: 83 c6 01 add $0x1,%esi  8048cbf: 83 fe 06 cmp $0x6,%esi  8048cc2: 74 22 je 8048ce6 <phase\_6+0x5d>  **从0x8048cbc开始，%esi的值开始从1往6递增，直到等于6时才跳转到0x8048ce6，否则将执行8048cc4。这里应该是双层循环的外层。**  For(int i=0;i<6;i++)  {  8048cc4: 8d 1c b7 lea (%edi,%esi,4),%ebx  8048cc7: 89 75 b4 mov %esi,-0x4c(%ebp)  8048cca: 8b 44 b7 fc mov -0x4(%edi,%esi,4),%eax  8048cce: 3b 03 cmp (%ebx),%eax  8048cd0: 75 05 jne 8048cd7 <phase\_6+0x4e>  8048cd2: e8 fa 03 00 00 call 80490d1 <explode\_bomb>  前一个数不能和后一个数相等  8048cd7: 83 45 b4 01 addl $0x1,-0x4c(%ebp)  8048cdb: 83 c3 04 add $0x4,%ebx  8048cde: 83 7d b4 05 cmpl $0x5,-0x4c(%ebp)  8048ce2: 7e e6 jle 8048cca <phase\_6+0x41>  8048ce4: eb c6 jmp 8048cac <phase\_6+0x23>  内循环  前一个数值取出并加1，指针指向数组的下一个元素。  六个数不相等，  8048ce6: bb 00 00 00 00 mov $0x0,%ebx  0x8048ce6开始执行一个新的操作，将寄存器ebx的值清零，同样将地址-0x30(%ebp)传给%edi，然后直接跳转至0x8048d06处。  8048ceb: 8d 7d d0 lea -0x30(%ebp),%edi  8048cee: eb 16 jmp 8048d06 <phase\_6+0x7d>  8048cf0: 8b 52 08 mov 0x8(%edx),%edx  8048cf0到8048cf8是一个循环，每循环一次，就更新一次%edx中的值，更新为内存地址为0x8(%edx)中的值，  链表  直到%eax中的值和%ecx中的值相等为止。  8048cf3: 83 c0 01 add $0x1,%eax  8048cf6: 39 c8 cmp %ecx,%eax  8048cf8: 75 f6 jne 8048cf0 <phase\_6+0x67>  8048cfa: 89 54 b5 b8 mov %edx,-0x48(%ebp,%esi,4)  0x8048cfa处开始是一个mov指令，将前面那个循环产生的最终的%edx中的值存入-0x48(%ebp,%esi,4)位置，若为第一轮操作，就是-0x48(%ebp)，若为第二轮，就是-0x44(%ebp)，依此次序每次向上递增4个单元。  8048cfe: 83 c3 01 add $0x1,%ebx  8048d01: 83 fb 06 cmp $0x6,%ebx  8048d04: 74 16 je 8048d1c <phase\_6+0x93>  8048d06: 89 de mov %ebx,%esi  0x8048d06开始先是将%ebx（也就是0）赋给%esi，使%esi获得初始值0，再将(%edi,%ebx,4)位置的数取出，存入%ecx中  8048d08: 8b 0c 9f mov (%edi,%ebx,4),%ecx  8048d0b: ba c4 c0 04 08 mov $0x804c0c4,%edx 立即数  8048d10: b8 01 00 00 00 mov $0x1,%eax  8048d15: 83 f9 01 cmp $0x1,%ecx  %edi,%ebx,4和1比较  8048d18: 7f d6 jg 8048cf0 <phase\_6+0x67>  8048d1a: eb de jmp 8048cfa <phase\_6+0x71>  这里  8048d1c: 8b 5d b8 mov -0x48(%ebp),%ebx  8048d1f: 8b 45 bc mov -0x44(%ebp),%eax  8048d22: 89 43 08 mov %eax,0x8(%ebx)  8048d25: 8b 55 c0 mov -0x40(%ebp),%edx  8048d28: 89 50 08 mov %edx,0x8(%eax)  8048d2b: 8b 45 c4 mov -0x3c(%ebp),%eax  8048d2e: 89 42 08 mov %eax,0x8(%edx)  8048d31: 8b 55 c8 mov -0x38(%ebp),%edx  8048d34: 89 50 08 mov %edx,0x8(%eax)  8048d37: 8b 45 cc mov -0x34(%ebp),%eax  8048d3a: 89 42 08 mov %eax,0x8(%edx)  8048d3d: c7 40 08 00 00 00 00 movl $0x0,0x8(%eax)  8048d44: be 00 00 00 00 mov $0x0,%esi  8048d49: 8b 43 08 mov 0x8(%ebx),%eax  8048d4c: 8b 13 mov (%ebx),%edx  8048d4e: 3b 10 cmp (%eax),%edx  是否是递增序列  8048d50: 7d 05 jge 8048d57 <phase\_6+0xce>  8048d52: e8 7a 03 00 00 call 80490d1 <explode\_bomb>  8048d57: 8b 5b 08 mov 0x8(%ebx),%ebx  8048d5a: 83 c6 01 add $0x1,%esi  8048d5d: 83 fe 05 cmp $0x5,%esi  8048d60: 75 e7 jne 8048d49 <phase\_6+0xc0>  8048d62: 83 c4 5c add $0x5c,%esp  8048d65: 5b pop %ebx  8048d66: 5e pop %esi  8048d67: 5f pop %edi  8048d68: 5d pop %ebp  8048d69: c3 ret    561432    然后每次将%edx偏移0x8，查看后会发现，首先edx中是内容，然后edx+0x8内的内容是一个地址，然后再查看这个地址，地址内的内容是一个内容，然后在将这个地址偏移0x8得到的地址内的内容又是一个地址，可以看出来，这是一个典型的链表结构。链表就是不同的node节点，每个node节点在内存中存在不同的位置，不连续的位置中。每个node由内容和next指针构成，每个next指针内的值是一个指向下一个node节点的地址，最后一个node的next指向null。  这样结合前面的分析就可以知道，输入的6个数实际上就是不同的node节点数，根据我输入的节点号，把节点内的内容重新进行排序，排成递增序列。   1. 结论   这一关其实不只是一个简简单单的循环。比如后面的比较+循环结构实现的功能是判断链表元素是否是一个递增的序列……   1. **秘密关卡** 2. 原理设计：   这题关键是要把递归看清楚，可能不需要像第四关一样写出c语言代码，但是也要捋清跳转条件，如何跳转等等。  可以画图说明。  ../../../../Downloads/E05EA32DB747E69B965FD0B173C0C395.pn   1. 实现与分析：   首先，得知道怎么开启隐藏关卡！  找出defused的源码  08049014 <phase\_defused>:  8049014: 55 push %ebp  8049015: 89 e5 mov %esp,%ebp  8049017: 81 ec 88 00 00 00 sub $0x88,%esp  804901d: 65 a1 14 00 00 00 mov %gs:0x14,%eax  8049023: 89 45 f4 mov %eax,-0xc(%ebp)  8049026: 31 c0 xor %eax,%eax  8049028: 83 3d d0 c3 04 08 06 cmpl $0x6,0x804c3d0  这里有写！必须解出来六个答案之后才可以到隐藏关卡，  804902f: 0f 85 86 00 00 00 jne 80490bb <phase\_defused+0xa7>  8049035: 8d 45 a4 lea -0x5c(%ebp),%eax  8049038: 89 44 24 10 mov %eax,0x10(%esp)  804903c: 8d 45 9c lea -0x64(%ebp),%eax  804903f: 89 44 24 0c mov %eax,0xc(%esp)  8049043: 8d 45 a0 lea -0x60(%ebp),%eax  8049046: 89 44 24 08 mov %eax,0x8(%esp)  804904a: c7 44 24 04 00 a2 04 movl $0x804a200,0x4(%esp)  8049051: 08  8049052: c7 04 24 d0 c4 04 08 movl $0x804c4d0,(%esp)  8049059: e8 e2 f7 ff ff call 8048840 <\_\_isoc99\_sscanf@plt>  和这个函数有关？  <\_\_isoc99\_sscanf@plt>:  8048840: ff 25 3c c0 04 08 jmp \*0x804c03c  8048846: 68 78 00 00 00 push $0x78  804884b: e9 f0 fe ff ff jmp 8048740 <\_init+0x3c>  804905e: 83 f8 03 cmp $0x3,%eax  输入三个参数  8049061: 75 44 jne 80490a7 <phase\_defused+0x93>  8049063: c7 44 24 04 09 a2 04 movl $0x804a209,0x4(%esp)    所以，要加这个字符才可以！  在这里，就读取find secret phase  804906a: 08  804906b: 8d 45 a4 lea -0x5c(%ebp),%eax  804906e: 89 04 24 mov %eax,(%esp)  8049071: e8 35 ff ff ff call 8048fab <strings\_not\_equal>  8049076: 85 c0 test %eax,%eax  8049078: 75 2d jne 80490a7 <phase\_defused+0x93>  就是说，你的not equal那个要符合要求，才能开启    设置断点，查看这些地址里写的都是啥  804907a: c7 44 24 04 dc a2 04 movl $0x804a2dc,0x4(%esp)  8049081: 08  8049082: c7 04 24 01 00 00 00 movl $0x1,(%esp)  8049089: e8 e2 f7 ff ff call 8048870 <\_\_printf\_chk@plt>  804908e: c7 44 24 04 04 a3 04 movl $0x804a304,0x4(%esp)  8049095: 08  8049096: c7 04 24 01 00 00 00 movl $0x1,(%esp)  804909d: e8 ce f7 ff ff call 8048870 <\_\_printf\_chk@plt>  80490a2: e8 74 fb ff ff call 8048c1b <secret\_phase>  80490a7: c7 44 24 04 3c a3 04 movl $0x804a33c,0x4(%esp)  80490ae: 08  80490af: c7 04 24 01 00 00 00 movl $0x1,(%esp)  80490b6: e8 b5 f7 ff ff call 8048870 <\_\_printf\_chk@plt>  80490bb: 8b 45 f4 mov -0xc(%ebp),%eax  80490be: 65 33 05 14 00 00 00 xor %gs:0x14,%eax  80490c5: 74 05 je 80490cc <phase\_defused+0xb8>  80490c7: e8 e4 f6 ff ff call 80487b0 <\_\_stack\_chk\_fail@plt>  80490cc: c9 leave  80490cd: 8d 76 00 lea 0x0(%esi),%esi  80490d0: c3 ret    成功啦！！！  开始看秘密关卡  08048c1b <secret\_phase>:  8048c1b: 55 push %ebp  8048c1c: 89 e5 mov %esp,%ebp  8048c1e: 53 push %ebx  8048c1f: 83 ec 14 sub $0x14,%esp  8048c22: e8 df 05 00 00 call 8049206 <read\_line>  首先一句call  8049206 <read\_line>，表明程序先读入一行，随后返回值%eax作为函数<strtol@plt>的参数之一，  其中我和同学对于这个<strtol@plt>函数进行了讨论。我认为在看不懂代码时就应该直接看寄存器eax是不是在运行函数之前和运行函数之后是一样的。这个函数也可以根据名字来推断，是一个差不多算是转换字符的函数，把我输入的参数转化成一种极其能识别的语言，它返回的那个值将在下面有别的计算和约束条件。  8048c27: c7 44 24 08 0a 00 00 movl $0xa,0x8(%esp)  8048c2e: 00  8048c2f: c7 44 24 04 00 00 00 movl $0x0,0x4(%esp)  8048c36: 00  8048c37: 89 04 24 mov %eax,(%esp)  8048c3a: e8 71 fc ff ff call 80488b0 <strtol@plt>  8048c3f: 89 c3 mov %eax,%ebx  8048c41: 8d 40 ff lea -0x1(%eax),%eax  8048c44: 3d e8 03 00 00 cmp $0x3e8,%eax  的十进制数要小于等于1001  8048c49: 76 05 jbe 8048c50 <secret\_phase+0x35>  8048c4b: e8 81 04 00 00 call 80490d1 <explode\_bomb>  8048c50: 89 5c 24 04 mov %ebx,0x4(%esp)  8048c54: c7 04 24 78 c1 04 08 movl $0x804c178,(%esp)  8048c5b: e8 6a ff ff ff call 8048bca <fun7>  输入的数作为<fun7> 的参数之一  另外一个参数来自 0x804c178，查看为0x24。  8048c60: 83 f8 05 cmp $0x5,%eax  8048c63: 74 05 je 8048c6a <secret\_phase+0x4f>  8048c65: e8 67 04 00 00 call 80490d1 <explode\_bomb>  8048c6a: c7 44 24 04 34 a1 04 movl $0x804a134,0x4(%esp)  8048c71: 08  8048c72: c7 04 24 01 00 00 00 movl $0x1,(%esp)  8048c79: e8 f2 fb ff ff call 8048870 <\_\_printf\_chk@plt>  8048c7e: e8 91 03 00 00 call 8049014 <phase\_defused>  8048c83: 83 c4 14 add $0x14,%esp  8048c86: 5b pop %ebx  8048c87: 5d pop %ebp  8048c88: c3 ret  08048bca <fun7>:  8048bca: 55 push %ebp  8048bcb: 89 e5 mov %esp,%ebp  8048bcd: 53 push %ebx  8048bce: 83 ec 14 sub $0x14,%esp  8048bd1: 8b 55 08 mov 0x8(%ebp),%edx  8048bd4: 8b 4d 0c mov 0xc(%ebp),%ecx  8048bd7: b8 ff ff ff ff mov $0xffffffff,%eax  8048bdc: 85 d2 test %edx,%edx  8048bde: 74 35 je 8048c15 <fun7+0x4b> 结束  8048be0: 8b 1a mov (%edx),%ebx  8048be2: 39 cb cmp %ecx,%ebx  8048be4: 7e 13 jle 8048bf9 <fun7+0x2f>  8048be6: 89 4c 24 04 mov %ecx,0x4(%esp)  8048bea: 8b 42 04 mov 0x4(%edx),%eax  8048bed: 89 04 24 mov %eax,(%esp)  8048bf0: e8 d5 ff ff ff call 8048bca <fun7>  8048bf5: 01 c0 add %eax,%eax  8048bf7: eb 1c jmp 8048c15 <fun7+0x4b>  8048bf9: b8 00 00 00 00 mov $0x0,%eax  8048bfe: 39 cb cmp %ecx,%ebx  8048c00: 74 13 je 8048c15 <fun7+0x4b>  8048c02: 89 4c 24 04 mov %ecx,0x4(%esp)  8048c06: 8b 42 08 mov 0x8(%edx),%eax  8048c09: 89 04 24 mov %eax,(%esp)  8048c0c: e8 b9 ff ff ff call 8048bca <fun7>  递归返回值加倍，+1————下面  8048c11: 8d 44 00 01 lea 0x1(%eax,%eax,1),%eax  8048c15: 83 c4 14 add $0x14,%esp  8048c18: 5b pop %ebx  8048c19: 5d pop %ebp  8048c1a: c3 ret  在调用完<fun7>之后，紧跟着cmp   $0x5,%eax，即返回值必须为5。<fun7>分析如上，为递归函数，与第四题十分相似。递归最深处的返回值肯定为0，最外层返回值为5，可得出如下反递归过程：  A\*2+1=5 - ->A=2    即有\*A<B  A\*2=2  - ->A=1    有\*A>B  A\*2+1=1 - ->A=0    即有\*A<B  ../../../../Downloads/E05EA32DB747E69B965FD0B173C0C395.pn  也就是说在这三次递归中两次执行了“若\*A<B将(A+8)作为地址进入递归”系列代码，一次执行了“若\*A>b，将(A+4)作为地址进入递归”系列代码。使用gdb查询储存值：      为啥错了！  因为要输入十进制！    果然   1. 结论   通过对返回值的要求，来进行反递归，从不同条件来推断出这个结果是怎么得来的。比如5是奇数，那么肯定是2\*2+1，而2是偶数，那么就是1\*2……  收获与体会：  通过这个炸弹实验，因为是有目的性的去做实验，所以比较有趣味，完成实验也比较主动。这里我学到了很多知识，不仅仅是会看这些指令都是什么意思，更多的是形象的知道了地址里面的数据都是什么，链表会怎么存储，数组会怎么存储，如何通过跳转完成循环，知道寄存器外面那个数其实也是有意义的，一般是挨着存的，会发现两个int之间恰好是4个字节，而栈的空间的基数也恰好是4个字节。  这次试验会给我以后的学习带来很大的方便，不仅是为计算机系统这门课打下基础，而且以后再遇到程序的bug，我也至少有能力去看汇编代码来分析错误了。（比如有些错误没办法debug出来的话，就可以看汇编或者寄存器是不是溢出或者查看是什么导致了溢出，在想办法该如何对待溢出特别是浮点数的溢出……）  还有就是最后的秘密关卡，得要输入特定的字符才可以触发它，那我就得要找到defuse函数去解读，哇，一看defuse函数就发现真应该早点去看！之前疑惑的为什么输入三个数啊，为什么是某种顺序啊什么的突然说得通了。  然后秘密关卡为啥输入三个参数啊，啥时候读取Curses, you've found the secretphase!\n  什么的，都能通过查看地址知道了。  然后再做到后面实验的时候，确实有些函数看得云里雾里，push了一个0x60是什么意思？这个函数是干什么的？？？但是通过测试答案，大概就可以猜得到。比如那个秘密关卡，我之前就直接写0x2f就不对（明摆着的哈哈）所以就能推测到之前要输入的是十进制数，blabla  留有的疑惑：  8048e7b: 8b 45 f4 mov -0xc(%ebp),%eax  8048e7e: 89 04 24 mov %eax,(%esp)  8048e81: e8 da fc ff ff call 8048b60 <func4>  0x8048e7b开始再将eax中的值设为phase\_4的返回值。  我还是不太明白这是怎么做到的。我只能通过查看eax等寄存器来找到我的输入变量可能都存在哪里，并不能通过这几行代码了解清楚。  出了这个疑惑也没啥了，就是感觉题目数量不够，做着还是有点生疏。多练才能熟练吧。 | |
| 实  验成绩 |  |