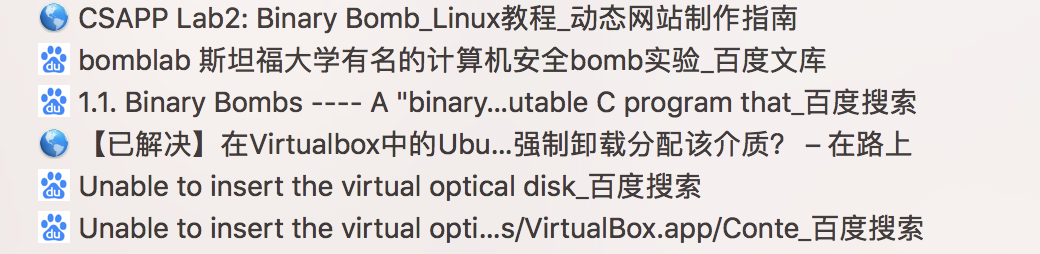
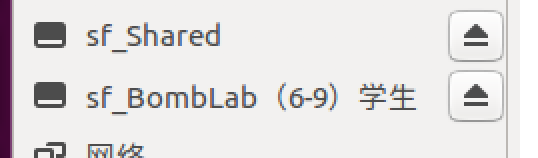
**Bomb实验日志集合**

2018年4 月 10日

第六周周二中午：把文件拖进ubuntu环境，由于不能直接拖进去，所以一节课都在查资料该怎么建立共享文件夹，设置挂载点等，最终成功。

以下是我的搜索记录和最终共享文件夹截图



2018年4 月 10日

第六周周二下午：

找到反汇编代码对应的phase1

08048f61 <phase\_1>:

8048f61: 55 push %ebp

8048f62: 89 e5 mov %esp,%ebp

8048f64: 83 ec 18 sub $0x18,%esp

8048f67: c7 44 24 04 5c a1 04 movl $0x804a15c,0x4(%esp)

8048f6e: 08

8048f6f: 8b 45 08 mov 0x8(%ebp),%eax

8048f72: 89 04 24 mov %eax,(%esp)

8048f75: e8 31 00 00 00 call 8048fab <strings\_not\_equal>

8048f7a: 85 c0 test %eax,%eax

8048f7c: 74 05 je 8048f83 <phase\_1+0x22>

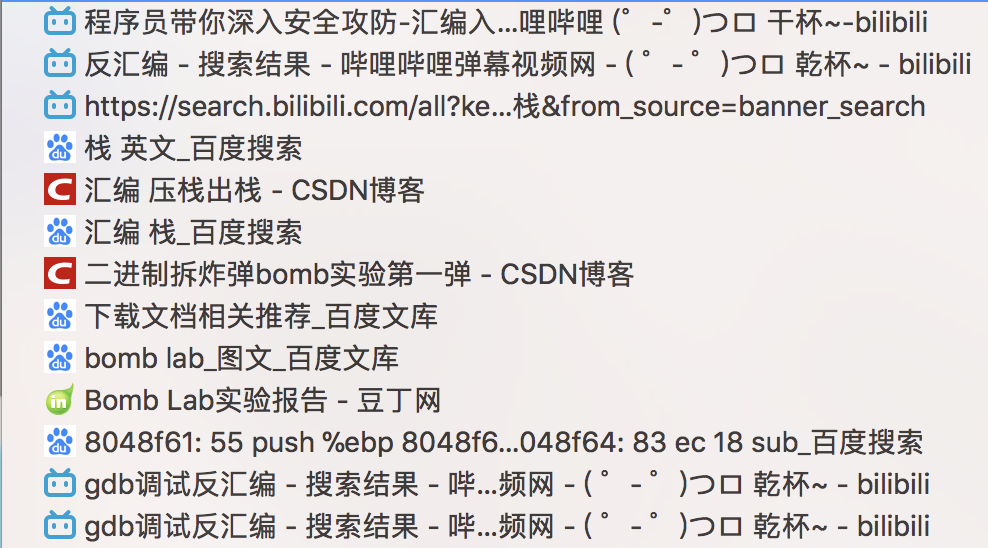
8048f7e: e8 4e 01 00 00 call 80490d1 <explode\_bomb>

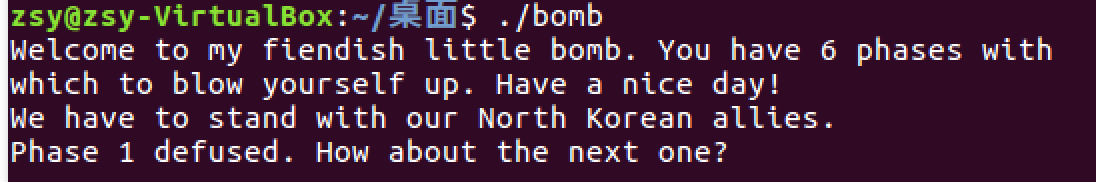
8048f83: c9 leave

8048f84: c3 ret

按照文件中对第一次汇编代码的分析来设置断点，查看密码。暂时不能自主解决，但是可以看懂。



在这之后我去网络上搜索了一些关于汇编的博客与慕课，在吃饭的时候看了一下



2018年4月 13日

第六周周五中午：开始研究第二题。

首先找到phase2代码：

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

8048d75: 89 44 24 04 mov %eax,0x4(%esp)

8048d79: 8b 45 08 mov 0x8(%ebp),%eax

8048d7c: 89 04 24 mov %eax,(%esp)

8048d7f: e8 87 03 00 00 call 804910b <read\_six\_numbers>

8048d84: 83 7d e0 00 cmpl $0x0,-0x20(%ebp) 0

8048d88: 75 06 jne 8048d90 <phase\_2+0x26>

8048d8a: 83 7d e4 01 cmpl $0x1,-0x1c(%ebp) 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 ebx在最后

8048d98: 8d 75 f8 lea -0x8(%ebp),%esi

8048d9b: 8b 43 fc mov -0x4(%ebx),%eax 1 ebx前一位

8048d9e: 03 43 f8 add -0x8(%ebx),%eax 0 ebx前两位

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 ebx往下移动一位

8048dad: 39 f3 cmp %esi,%ebx

8048daf: 75 ea jne 8048d9b <phase\_2+0x31>

8048db1: 83 c4 30 add $0x30,%esp

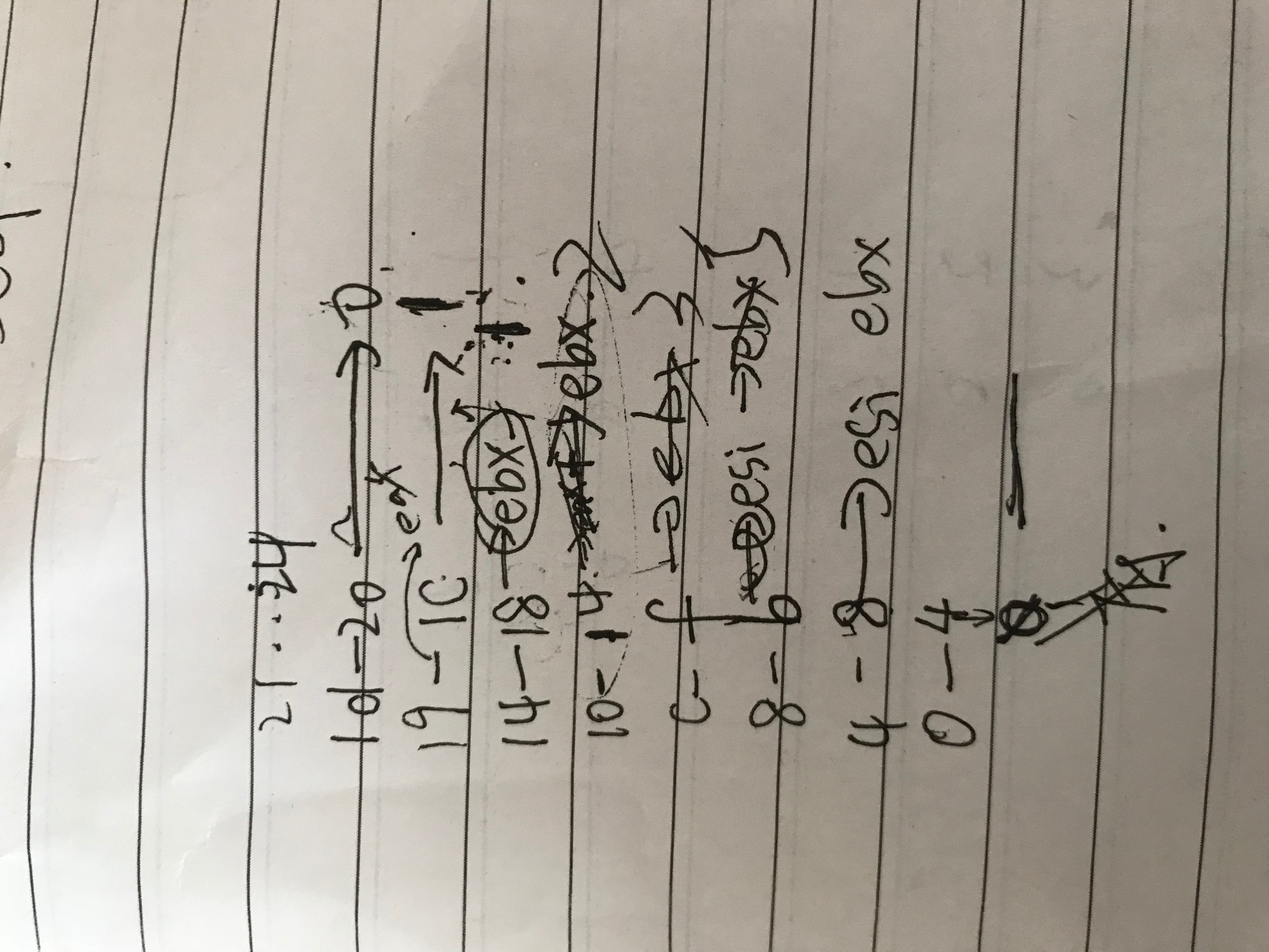
8048db4: 5b pop %ebx

8048db5: 5e pop %esi

8048db6: 5d pop %ebp

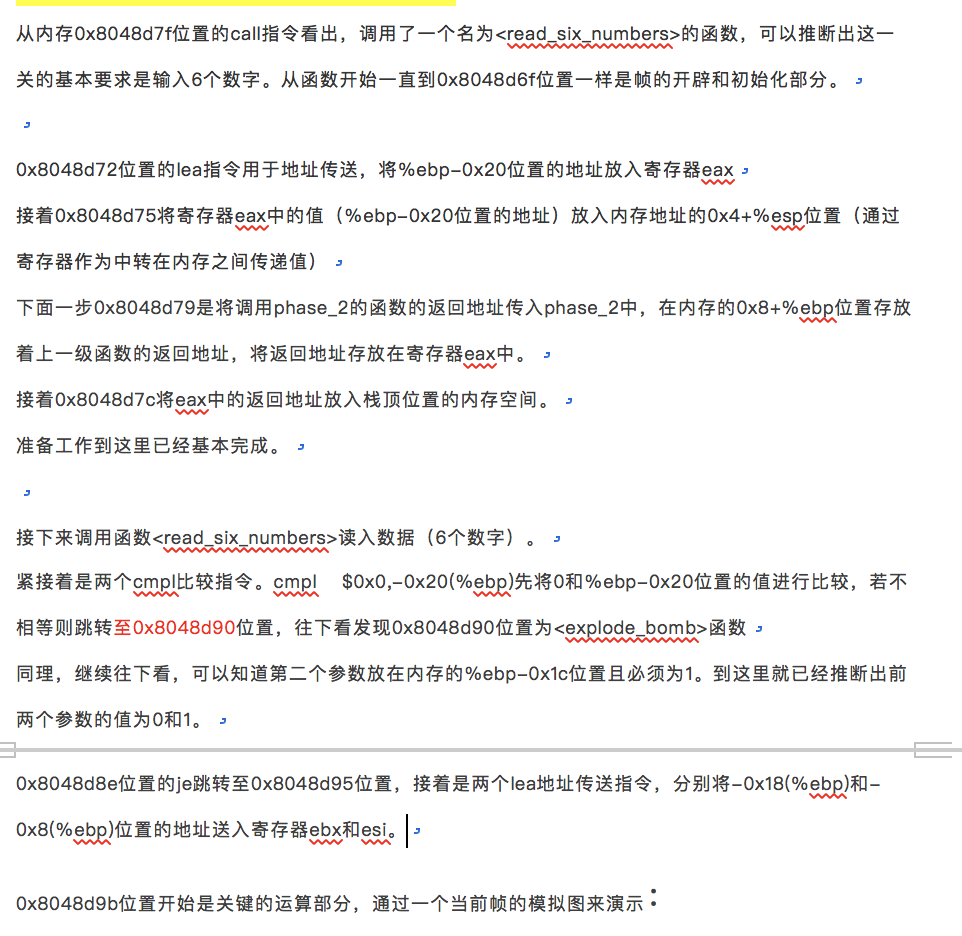
8048db7: c3 ret

为了看得明白，我画了一张图。



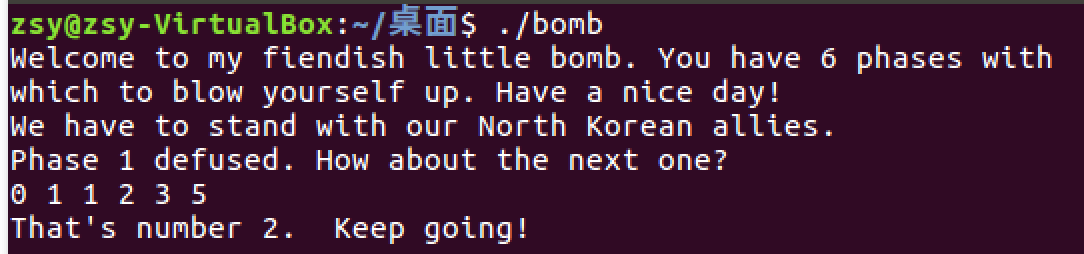


并且在代码中用不同颜色做标注。



在我的反汇编代码下面写下思路

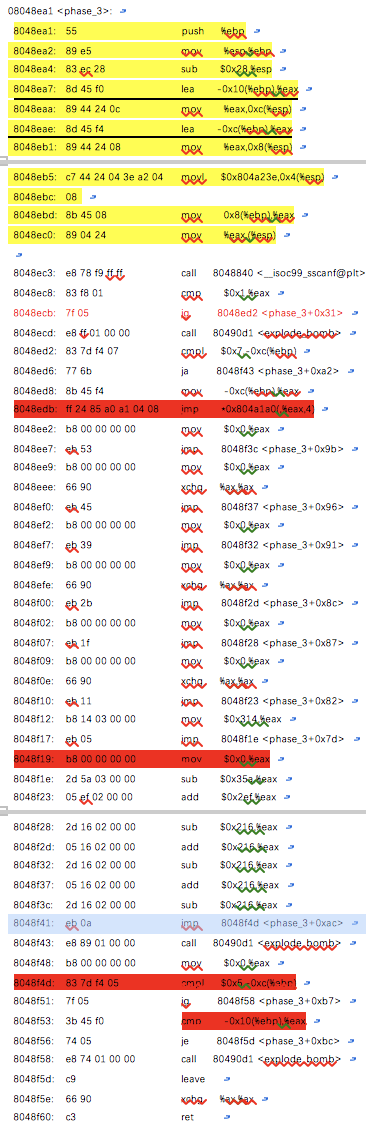
最后解决得出0 1 1 2 3 5



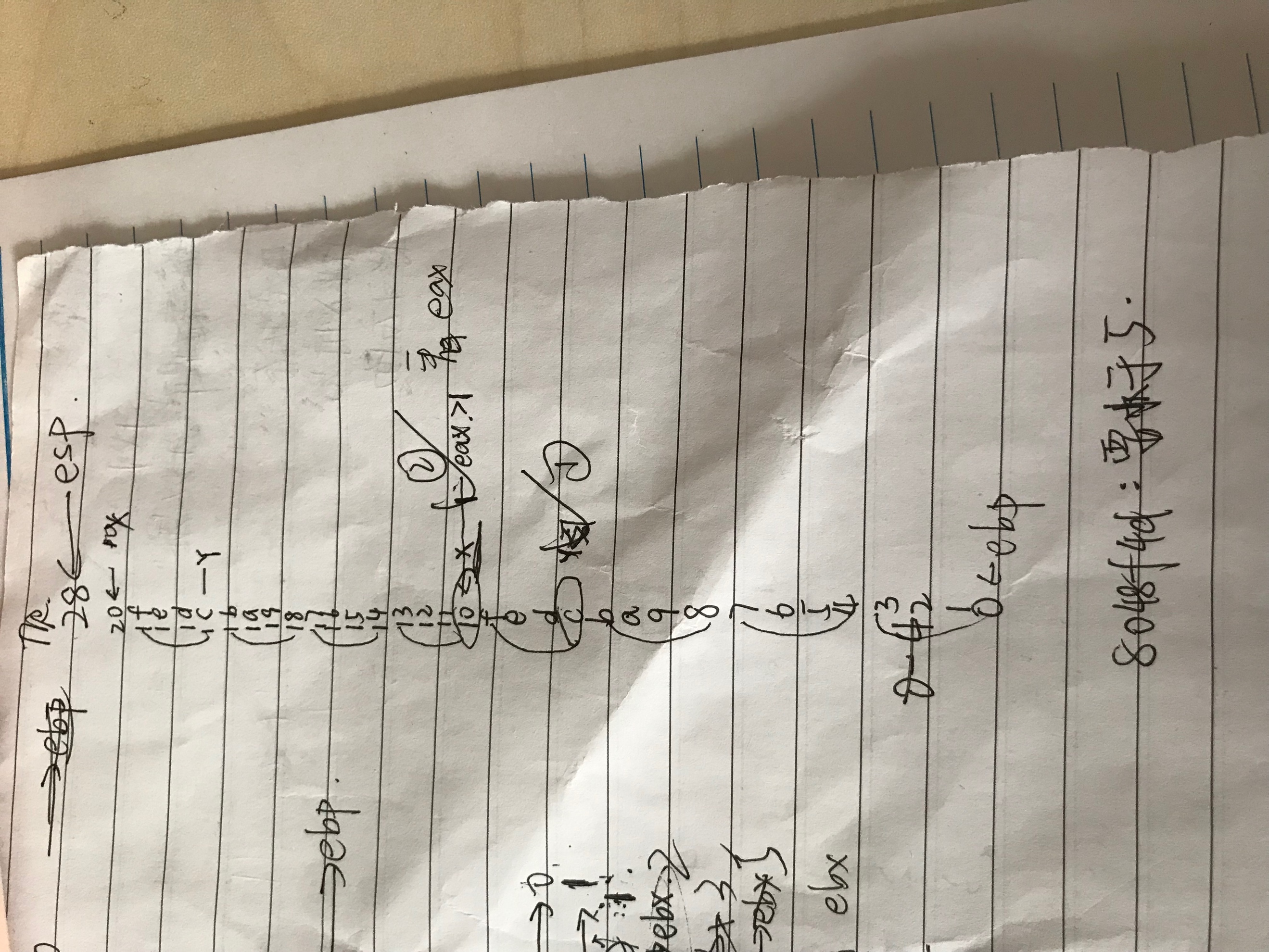
2018年4 月 15日

第六周周日下午： 开始解决第三题

找出对应的代码：



简易画个图图～

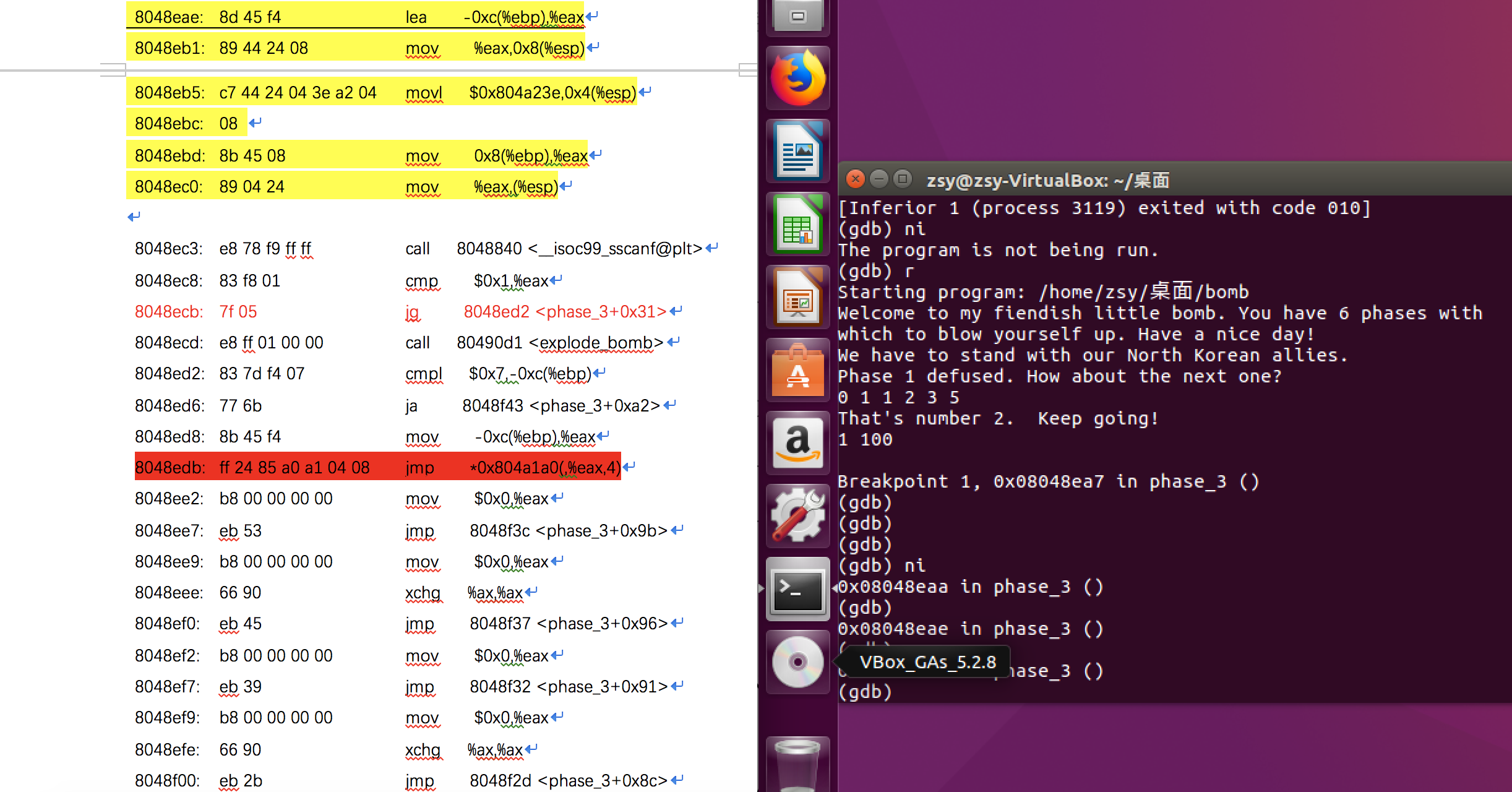


我认为这道题的关键是能分辨出有两个数字是我要输入的。

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（比如昨天的我）

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！！！！！！

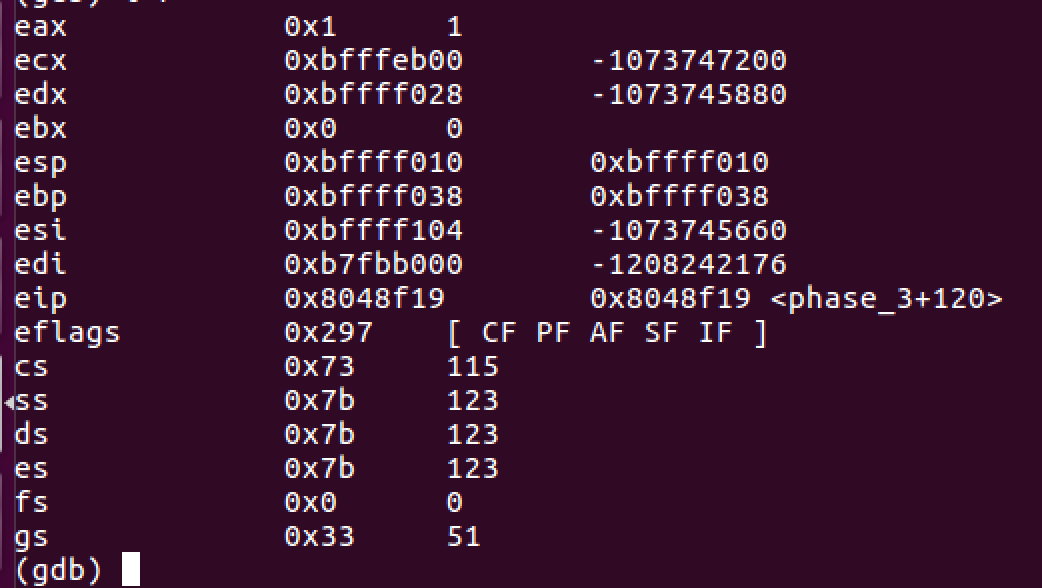


从地址08048eaa开始，两个数我就先设置为1，100，100是我瞎设置的，而1是我根据地址【8048ed2】的值第一个参数需要<7，于是我选择了1.

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

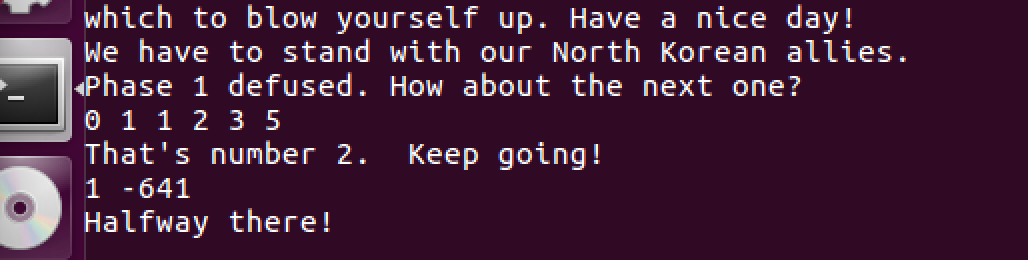
因为那个地址进行的是比较，ebp-10是不是等于eax里的值？？

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

2018年4 月 17日

第七周周二中午：开始处理phase4

首先把代码找出来

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

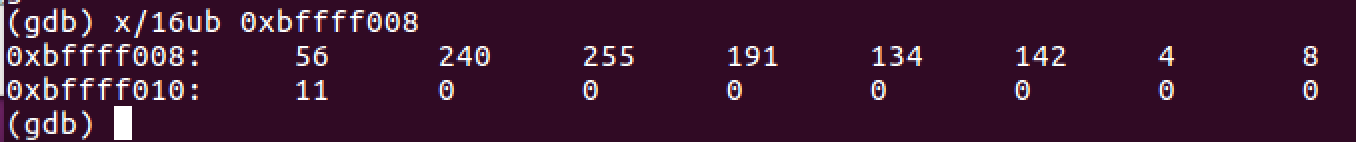
8048b61: 89 e5 mov %esp,%ebp

8048b63: 83 ec 18 sub $0x18,%esp

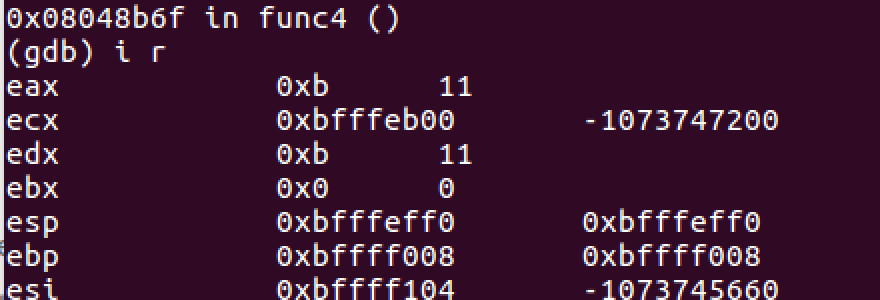
8048b66: 89 5d f8 mov %ebx,-0x8(%ebp)

8048b69: 89 75 fc mov %esi,-0x4(%ebp)

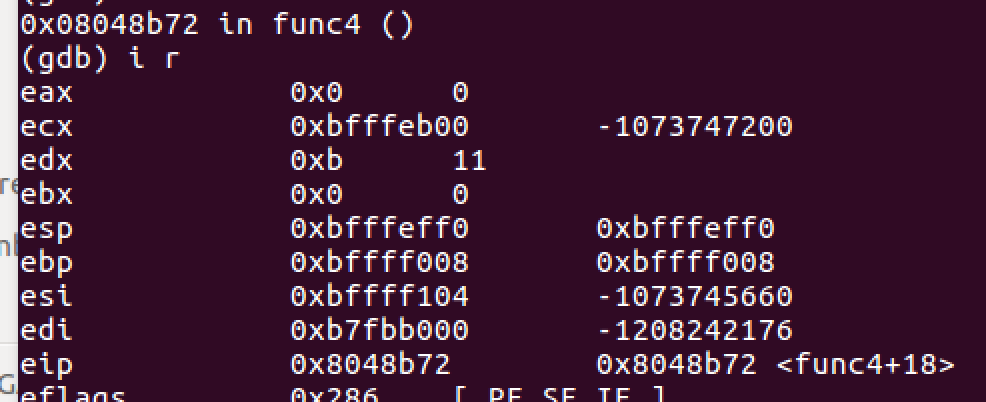
先查看内存，我输入的数字11已经存在了内存里面。



8048b6c: 8b 55 08 mov 0x8(%ebp),%edx

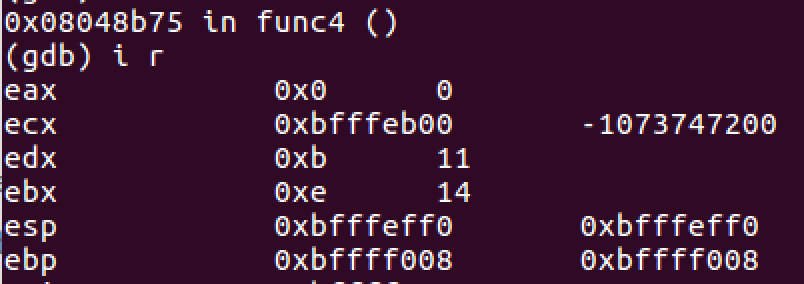


8048b6f: 8b 45 0c mov 0xc(%ebp),%eax



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;

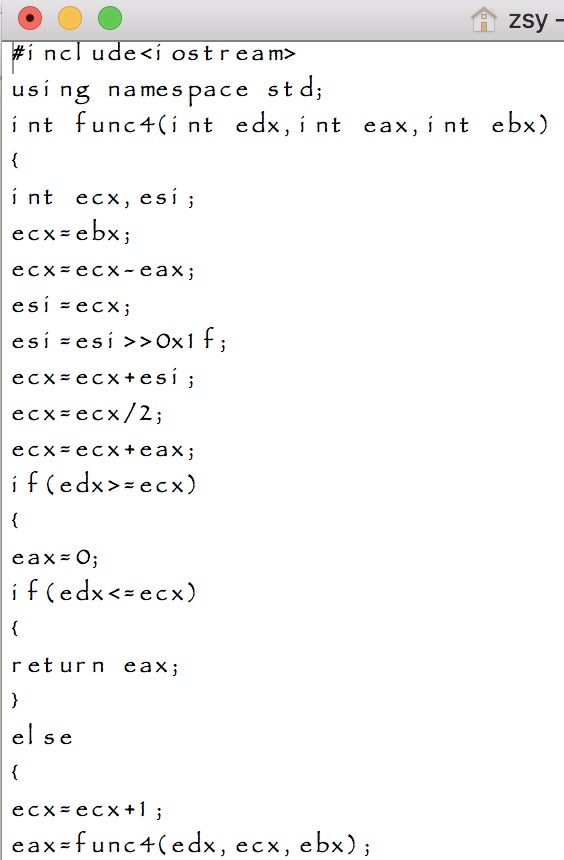
}

}

}

"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：

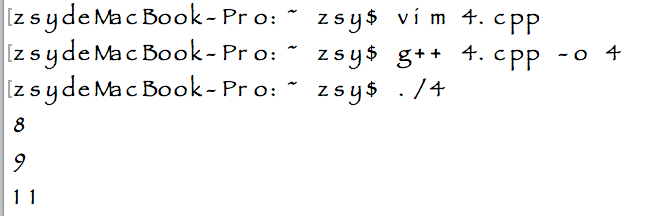
那分别有8 9 11

于是，结果就是

8 1

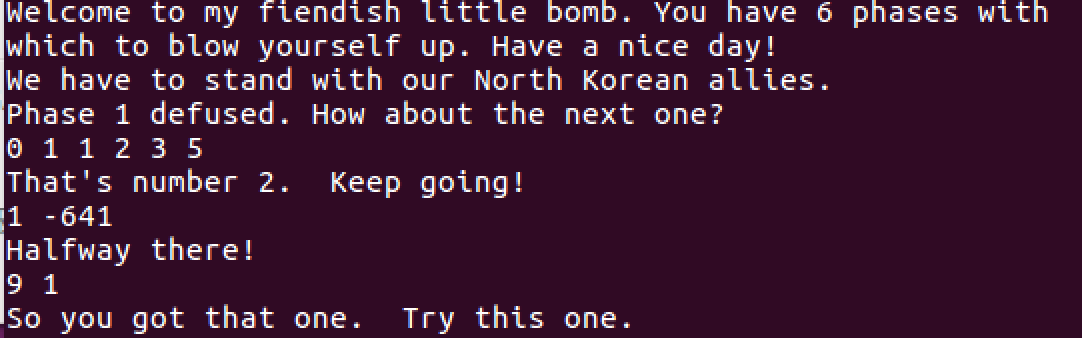
9 1

11 1



随便找一个检查一下

成功～



2018年4 月 20日

第七周周五下午：完成炸弹实验第五关：

经过了第四关，我的意志被磨练了，代码多烦我都不会害pia了（然而第六关轻易地打败了我）

代码找出来：

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>

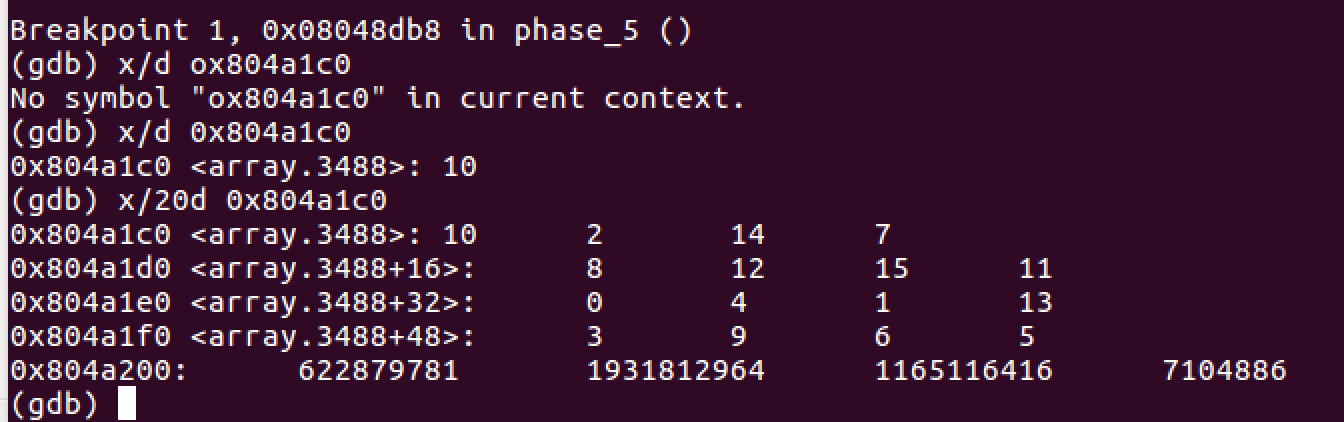
8048e27: 83 c4 20 add $0x20,%esp

8048e2a: 5b pop %ebx

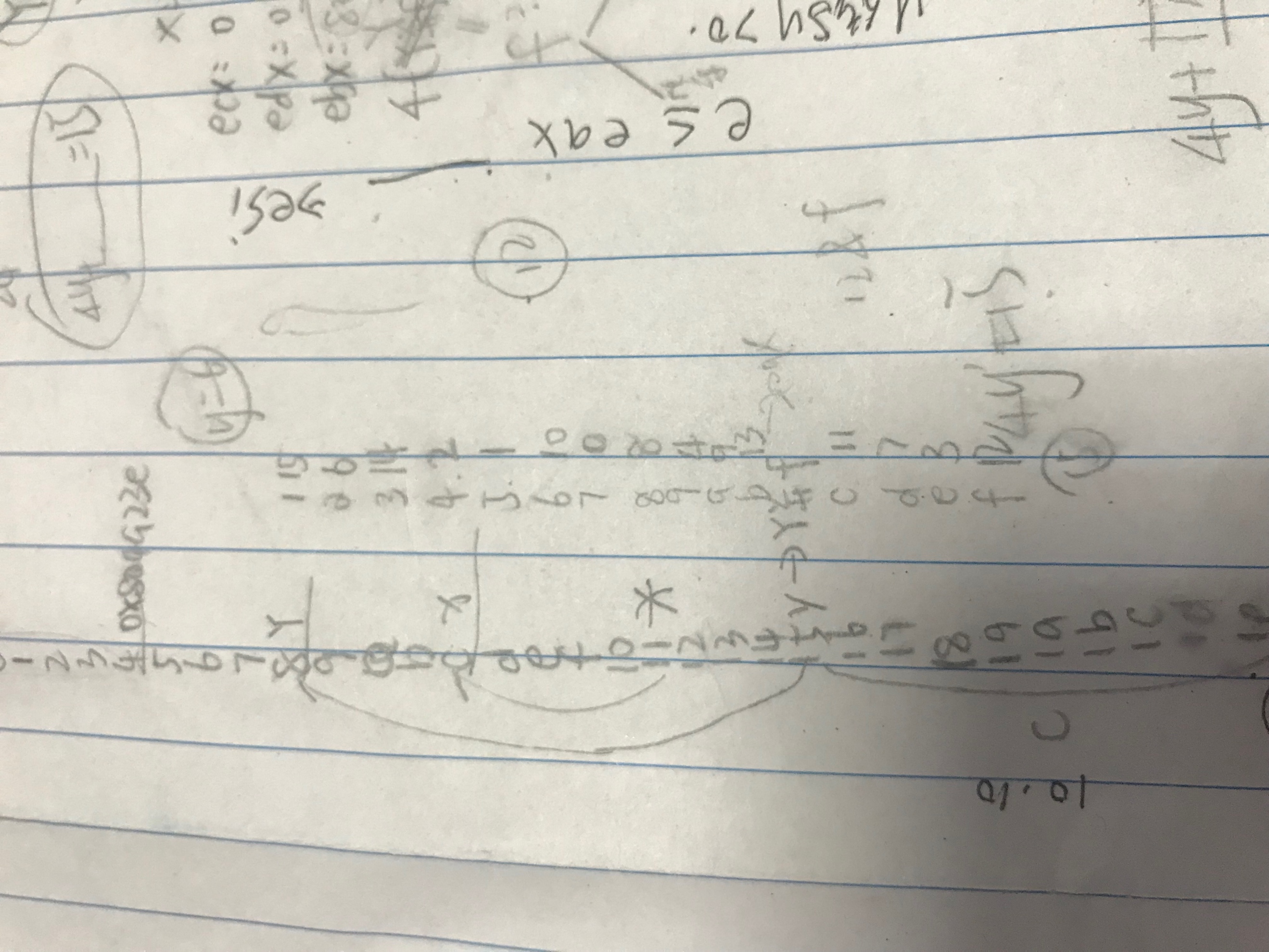
8048e2b: 5e pop %esi

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：



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

第八周周二中午：第六关

先把汇编代码找出来

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

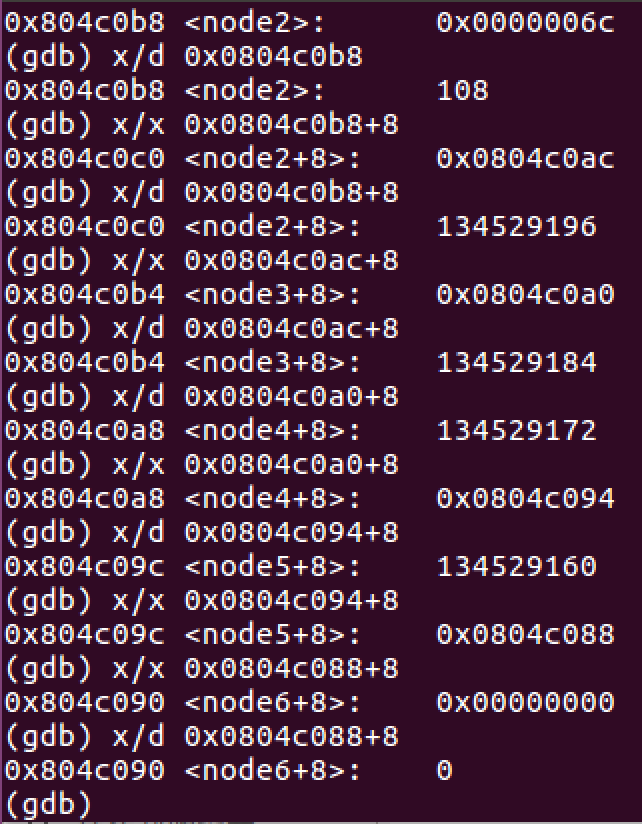
8048d65: 5b pop %ebx

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。

第八周周二下午：完成额外关卡

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，

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>

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

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：

