计算机系统原理实验之BombLab二进制炸弹5、6关

原创

Lily许 于 2018-05-23 15:02:08 发布 5966 收藏 24
文章标签： 计算机组成原理、炸弹实验、gdb、汇编语言

本文链接：https://blog.csdn.net/m0_37157965/article/details/80419909

实验目的：

通过二进制炸弹实验，熟悉汇编语言，反汇编工具objdump以及gdb调试工具。

实验内容：

1、炸弹实验第5关。

2、炸弹实验第6关。

实验过程：

第五关：

1、根据前几关的经验，进入bomb文件的gdb调试命令下，直接查看第五关的汇编代码。

```
xlc@xlc-VirtualBox:~$ gdb -q bomb
Reading symbols from bomb...done.
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x08048db8 <+0>:     push   %ebp
   0x08048db9 <+1>:     mov    %esp,%ebp
   0x08048dbb <+3>:     push   %esi
   0x08048dbc <+4>:     push   %ebx
   0x08048dbd <+5>:     sub    $0x20,%esp
   0x08048dc0 <+8>:     lea    -0x10(%ebp),%eax
   0x08048dc3 <+11>:    mov    %eax,0xc(%esp)
   0x08048dc7 <+15>:    lea    -0xc(%ebp),%eax
   0x08048dca <+18>:    mov    %eax,0x8(%esp)
   0x08048dce <+22>:    movl   $0x804a23e,0x4(%esp)
   0x08048dd6 <+30>:    mov    0x8(%ebp),%eax
   0x08048dd9 <+33>:    mov    %eax,(%esp)
   0x08048ddc <+36>:    call   0x8048840 <__isoc99_sscanf@plt>
   0x08048de1 <+41>:    cmp    $0x1,%eax
   0x08048de4 <+44>:    jg     0x8048deb <phase_5+51>
   0x08048de6 <+46>:    call   0x80490d1 <explode_bomb>
```


2、直接回车可以显示余下的phase_5函数的代码，浏览完一遍phase_5函数的汇编代码后，并不能直接发现此代码的精髓所在，因此，我开始逐条分析phase_5函数的代码。

首先是栈帧准备，开辟32字节的空间：

```
0x08048db8 <+0>:     push   %ebp
0x08048db9 <+1>:     mov    %esp,%ebp
0x08048dbb <+3>:     push   %esi
0x08048dbc <+4>:     push   %ebx
0x08048dbd <+5>:     sub    $0x20,%esp
```


接着加载%ebp-0x10处的有效地址间接存储到%esp+0xc处，也就是参数2存放的地方，同理，加载%ebp-0xc处的有效地址间接存储到%esp+0x8处，也就是参数1存放的地方。

```
0x08048dc0 <+8>:     lea    -0x10(%ebp),%eax
0x08048dc3 <+11>:    mov    %eax,0xc(%esp)
0x08048dc7 <+15>:    lea    -0xc(%ebp),%eax
0x08048dca <+18>:    mov    %eax,0x8(%esp)
```

接着看见了一串似曾相似的代码，老规矩，查看此地址处存放的数据，发现我们仍是输入两个整数。

```
(gdb) x/s 0x804a23e
0x804a23e:      "%d %d"
```

接着把%ebp+0x8处存储的值传送给%esp，然后调用__isoc99_sscanf@plt函数。

```
0x08048dd6 <+30>:    mov    0x8(%ebp),%eax
0x08048dd9 <+33>:    mov    %eax,(%esp)
0x08048ddc <+36>:    call   0x8048840 <__isoc99_sscanf@plt>
```

3、往下接着看，比较了%eax和1的大小，若%eax大于1，则跳转到<phase_5+51>处，否则，调用爆炸函数，引起爆炸，由此处可得，输入的参数个数必须大于1个。

```
0x08048de1 <+41>:    cmp    $0x1,%eax
0x08048de4 <+44>:    jg     0x8048deb <phase_5+51>
0x08048de6 <+46>:    call   0x80490d1 <explode_bomb>
0x08048deb <+51>:    mov    -0xc(%ebp),%eax
```

4、接着把参数1传给%eax，并对%eax与0xf作按位与运算，也就是取出参数1的低四位。

```
0x08048deb <+51>:    mov    -0xc(%ebp),%eax
0x08048dee <+54>:    and    $0xf,%eax
```

5、接着把取出来的参数1的低四位与0xf作比较，若相等，则跳转到爆炸函数，否则，执行下一步，由此可见，参数1的低四位不为1111。

```
0x08048df1 <+57>:    mov    %eax,-0xc(%ebp)
0x08048df4 <+60>:    cmp    $0xf,%eax
0x08048df7 <+63>:    je     0x8048e22 <phase_5+106>
```

6、接下来对%ecx，%edx赋值，初始化为0，%ebx则存储地址0x804a1c0。

```
0x08048df9 <+65>:    mov    $0x0,%ecx
Type <return> to continue, or q <return> to quit---
0x08048dfe <+70>:    mov    $0x0,%edx
0x08048e03 <+75>:    mov    $0x804a1c0,%ebx
```

7、接下来，执行%edx加1操作，而后执行操作得到eax=*(ebx+4eax)，由于之前得到%ebx存储地址0x804a1c0，并且此处以4*eax为步长，猜测0x804a1c0是一个数组的首地址，接着把存储在地址*(ebx+4eax)处的值与%ecx相加存储在%ecx中。

```
0x08048e08 <+80>:    add    $0x1,%edx
0x08048e0b <+83>:    mov    (%ebx,%eax,4),%eax
0x08048e0e <+86>:    add    %eax,%ecx
```

8、接着把%eax存储的值（应该是数组的序号）与0xf也就是15作比较，若不等于，则跳到<phase_5+80>处，否则，接着执行后续代码，由此可知，<phase_5+80>处开始应该是一个循环，每次循环取出数组中的一个值累加存储到%ecx中，最后一次取出的值应该是15，接着把%edx与15作比较，不相等则爆炸，反之，执行后续代码，由此可见循环需要进行15次，加上输入参数1（取值的时候只取了参数1二进制的后四位）的时候进行了一次循环，所以，总共取了16次数组的值。

```
0x08048e10 <+88>:    cmp    $0xf,%eax
0x08048e13 <+91>:    jne    0x8048e08 <phase_5+80>
0x08048e15 <+93>:    mov    %eax,-0xc(%ebp)
0x08048e18 <+96>:    cmp    $0xf,%edx
0x08048e1b <+99>:    jne    0x8048e22 <phase_5+106>
0x08048e1d <+101>:   cmp    %ecx,-0x10(%ebp)
0x08048e20 <+104>:   je     0x8048e27 <phase_5+111>
0x08048e22 <+106>:   call   0x80490d1 <explode_bomb>
0x08048e27 <+111>:   add    $0x20,%esp
0x08048e2a <+114>:   pop    %ebx
0x08048e2b <+115>:   pop    %esi
0x08048e2c <+116>:   pop    %ebp
0x08048e2d <+117>:   ret
```


9、分析到这里，大概可以理清了,输入参数1，调用循环，而后每次从数组中取出一个数累加到%ecx中，且16次循环的过程中直到最后一个才能取到15，由此，我们打印出数组连续的16位。

```
(gdb) p  *0x804a1c0@16
$1 = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}
```

另外我试着多打印几位看看数组里存储的到底是什么：

```
(gdb) p *0x804a1c0@20
$1 = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5, 622879781,
  1931812964, 1165116416, 7104886}
```

发现，其实这个数组里存储的刚好就只有16个有效元素，由此，推导出下面这个表。

数组序号和数组值相对应的表：

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	2	14	7	8	12	15	11	0	4	1	13	3	9	6	5

然后，根据上述推断则可以逆推出在数组里取到的值包括参数1在内的序列为：

5	12	3	7	11	13	9	4	8	0	10	1	2	14	6	15

又因为参数2为在数组里取到的值的累加值，所以参数2为：

12+3+7+11+13+9+4+8+0+10+1+2+14+6+15=115

也可以通过查看寄存器的值来得到参数2，通过以上分析，可以知道参数2被保存在%ecx中，在下图中可以看到%ecx中存储的值为115，和我求出来的一致，同时，也可以看到%eax和%edx存储的值都为15，与上述分析的也一致：

```
(gdb) b *0x08048e1d
Breakpoint 1 at 0x8048e1d
(gdb) r
Starting program: /home/xlc/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
We have to stand with our North Korean allies.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
0 147
Halfway there!
11 1
So you got that one.  Try this one.
5 5

Breakpoint 1, 0x08048e1d in phase_5 ()
(gdb) i r
eax            0xf      15
ecx            0x73     115
edx            0xf      15
ebx            0x804a1c0        134521280
esp            0xbffff120       0xbffff120
ebp            0xbffff148       0xbffff148
esi            0xbffff214       -1073745388
edi            0x0      0
eip            0x8048e1d        0x8048e1d <phase_5+101>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```


接下来测试一下5 、115是不是正确的通关密码：

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
We have to stand with our North Korean allies.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
0 147
Halfway there!
11 1
So you got that one.  Try this one.
5 115
Good work!  On to the next...
```

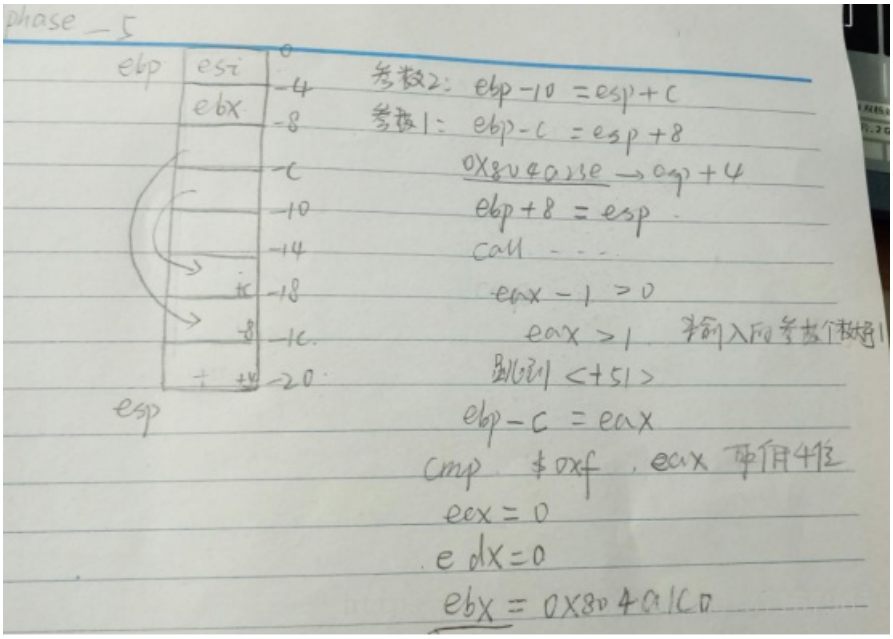

到这里，我不由得想起上次课程老师讲过的一点，在调用函数的之后，%eax和1作比较，得出的结论是，输入参数的个数必须大于1，那我是不是不用局限于两个，测试一下，随便输入第三个参数，看看会怎样，很惊讶，加了第三个参数，我随便输入了32，并没有影响通关密码的正确性，那么，就可以猜测，第五关的密码，只要前两个正确，后面应该随便组合都行吧，有无数种可能，这里，我就不一一去检验了：

```
0 1 1 2 3 5
That's number 2.  Keep going!
0 147
Halfway there!
11 1
So you got that one.  Try this one.
5 115 32
Good work!  On to the next...
```

紧接着，我想到了另外一点，前面有让%eax的低四位和1111作比较，得出的结论是参数1的二进制低四位不能为1111，而我求出来的答案是5,5的二进制表示为0101，于是，我有个大胆的猜想，是不是，只要输入的参数1的二进制形式的低四位不为1111，或者说刚好为0101的组合都符合通关密码的条件呢，我检验一下10101也就是21看看对不对，惊喜又出现了，真的可以，所以，答案应该是无数种，这里同样不一一检验了：

```
0 1 1 2 3 5
That's number 2.  Keep going!
0 147
Halfway there!
11 1
So you got that one.  Try this one.
21 115
Good work!  On to the next...
```


附上phase_5函数的栈帧图：



第五关成功通过。

第六关：

1、老规矩，查看phase_6的汇编代码。

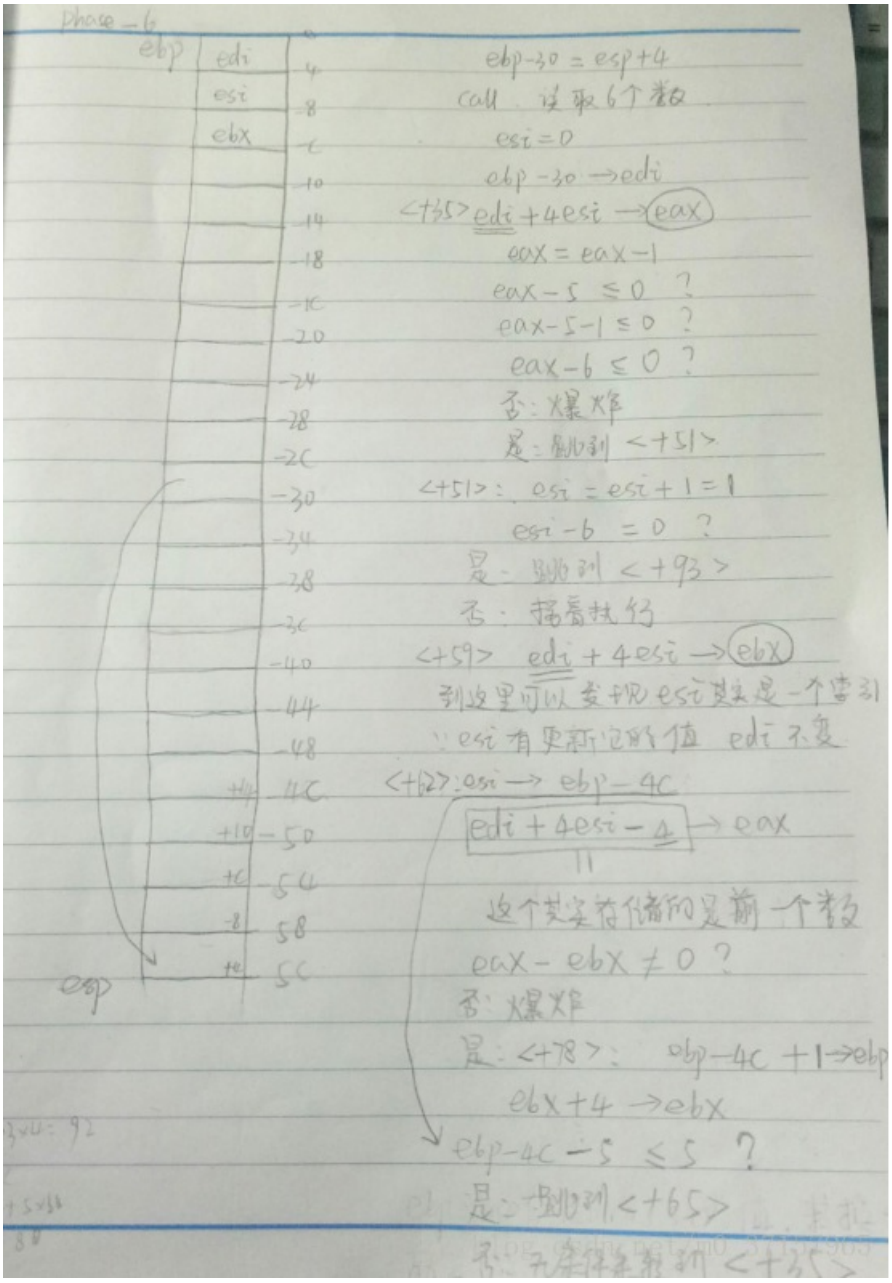
```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
   0x08048c89 <+0>:     push   %ebp
   0x08048c8a <+1>:     mov    %esp,%ebp
   0x08048c8c <+3>:     push   %edi
   0x08048c8d <+4>:     push   %esi
   0x08048c8e <+5>:     push   %ebx
   0x08048c8f <+6>:     sub    $0x5c,%esp
   0x08048c92 <+9>:     lea    -0x30(%ebp),%eax
   0x08048c95 <+12>:    mov    %eax,0x4(%esp)
   0x08048c99 <+16>:    mov    0x8(%ebp),%eax
   0x08048c9c <+19>:    mov    %eax,(%esp)
   0x08048c9f <+22>:    call   0x804910b <read_six_numbers>
   0x08048ca4 <+27>:    mov    $0x0,%esi
   0x08048ca9 <+32>:    lea    -0x30(%ebp),%edi
   0x08048cac <+35>:    mov    (%edi,%esi,4),%eax
   0x08048caf <+38>:    sub    $0x1,%eax
   0x08048cb2 <+41>:    cmp    $0x5,%eax
```

2、在粗略浏览完全部代码后，真的是不知所云，但还是看见了那么两个熟悉的函数名read_six_numbers和explode_bomb函数，由此，只能慢慢开始分析汇编代码，老实说，在画完栈帧图，一条一条过完全部指令后，我发现，单条指令是什么我知道，但综合所有的汇编代码，我真的是懵的，所以，我接着又重新再分析代码，仔细咀嚼。

首先是一些堆栈准备以及内存空间的开辟，这里申请了92字节的内存空间：

```
0x08048c89 <+0>:     push   %ebp
0x08048c8a <+1>:     mov    %esp,%ebp
0x08048c8c <+3>:     push   %edi
0x08048c8d <+4>:     push   %esi
0x08048c8e <+5>:     push   %ebx
0x08048c8f <+6>:     sub    $0x5c,%esp
```


下图是phase_6函数的栈帧图：

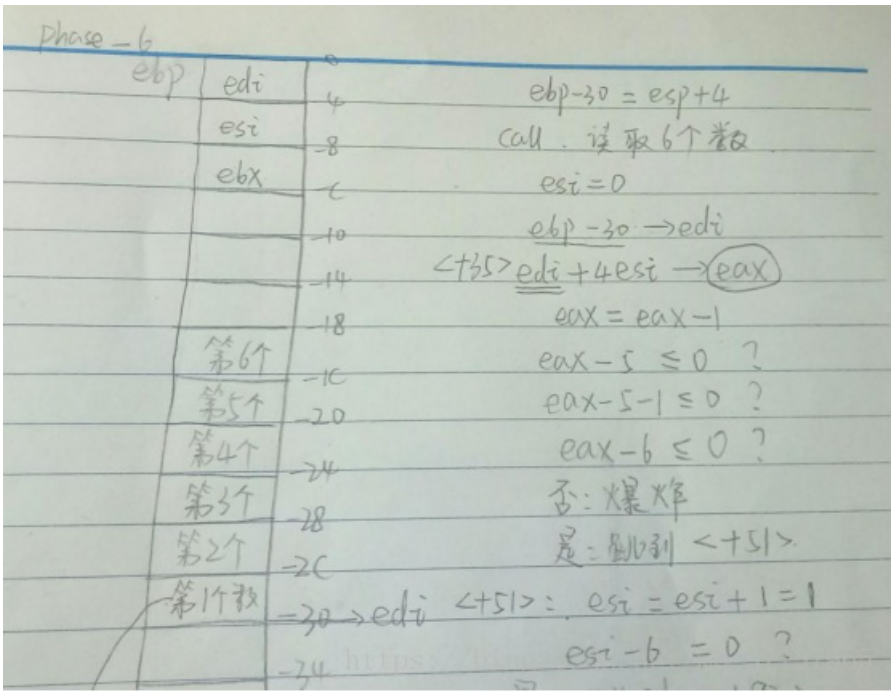


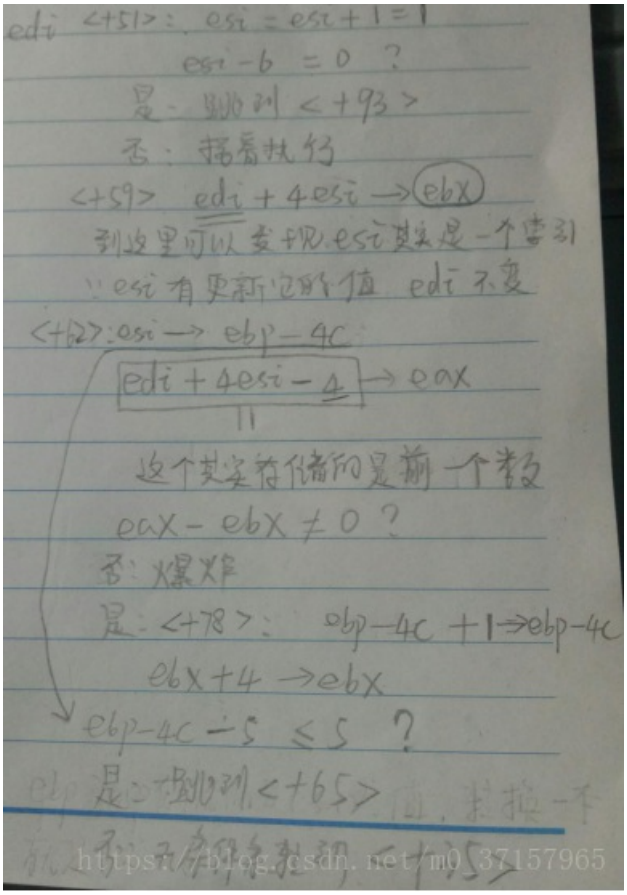
3、接着加载了%ebp-0x30存储的有效地址到%esp+0x4处，然后调用了read_six_numbers函数，查看此函数的汇编代码，结合前面第二关的经验，可以得到此函数是通过调用__isoc99_sscanf@plt函数来读取六个整数的：

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x0804910b <+0>:     push   %ebp
   0x0804910c <+1>:     mov    %esp,%ebp
   0x0804910e <+3>:     sub    $0x28,%esp
   0x08049111 <+6>:     mov    0xc(%ebp),%eax
   0x08049114 <+9>:     lea    0x14(%eax),%edx
   0x08049117 <+12>:    mov    %edx,0x1c(%esp)
   0x0804911b <+16>:    lea    0x10(%eax),%edx
   0x0804911e <+19>:    mov    %edx,0x18(%esp)
   0x08049122 <+23>:    lea    0xc(%eax),%edx
   0x08049125 <+26>:    mov    %edx,0x14(%esp)
   0x08049129 <+30>:    lea    0x8(%eax),%edx
   0x0804912c <+33>:    mov    %edx,0x10(%esp)
   0x08049130 <+37>:    lea    0x4(%eax),%edx
   0x08049133 <+40>:    mov    %edx,0xc(%esp)
   0x08049137 <+44>:    mov    %eax,0x8(%esp)
   0x0804913b <+48>:    movl   $0x804a232,0x4(%esp)
   0x08049143 <+56>:    mov    0x8(%ebp),%eax
   0x08049146 <+59>:    mov    %eax,(%esp)
   0x08049149 <+62>:    call   0x8048840 <__isoc99_sscanf@plt>
   0x0804914e <+67>:    cmp    $0x5,%eax
   0x08049151 <+70>:    jg     0x8049158 <read_six_numbers+77>
   0x08049153 <+72>:    call   0x80490d1 <explode_bomb>
```


4、在调用read_six_numbers函数读取完六个整数后，执行了如下操作，分析得到读取的六个整数存储在栈帧中的位置如下图所示，所以，当对%ebp-0x30处的数据进行操作时，即可知道是在对参数1进行操作，下图红色框内的语句可以得到参数1要小于等于6，通过蓝框和红框内的语句可以得到第二个数和第一个数不相同，结合这一部分的整体代码，可以推出读取到的六个数之间不相等且都小于等于6：

```
0x08048ca4 <+27>:    mov    $0x0,%esi
0x08048ca9 <+32>:    lea    -0x30(%ebp),%edi
0x08048cac <+35>:    mov    (%edi,%esi,4),%eax
0x08048caf <+38>:    sub    $0x1,%eax
0x08048cb2 <+41>:    cmp    $0x5,%eax
0x08048cb5 <+44>:    jbe    0x8048cbc <phase_6+51>
0x08048cb7 <+46>:    call   0x80490d1 <explode_bomb>
0x08048cbc <+51>:    add    $0x1,%esi
0x08048cbf <+54>:    cmp    $0x6,%esi
0x08048cc2 <+57>:    je     0x8048ce6 <phase_6+93>
0x08048cc4 <+59>:    lea    (%edi,%esi,4),%ebx
Type <return> to continue, or q <return> to quit---
0x08048cc7 <+62>:    mov    %esi,-0x4c(%ebp)
0x08048cca <+65>:    mov    -0x4(%edi,%esi,4),%eax
0x08048cce <+69>:    cmp    (%ebx),%eax
0x08048cd0 <+71>:    jne    0x8048cd7 <phase_6+78>
0x08048cd2 <+73>:    call   0x80490d1 <explode_bomb>
0x08048cd7 <+78>:    addl   $0x1,-0x4c(%ebp)
0x08048cdb <+82>:    add    $0x4,%ebx
0x08048cde <+85>:    cmpl   $0x5,-0x4c(%ebp)
0x08048ce2 <+89>:    jle    0x8048cca <phase_6+65>
```





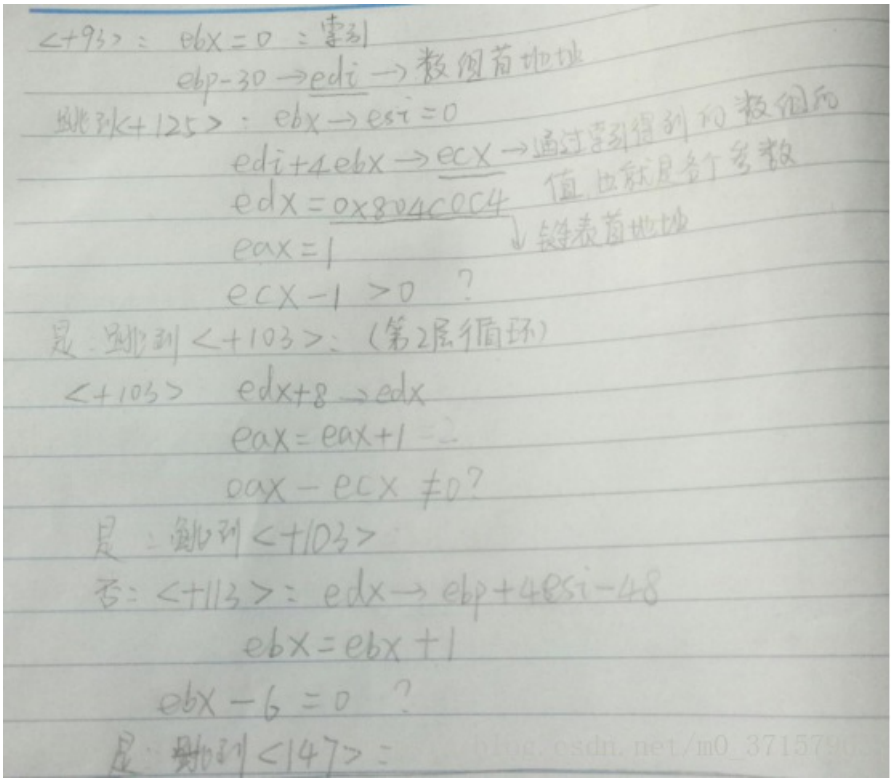
5、接着查看后续代码，分析以后可以得到，此段代码主要实现把一个单链表的节点信息根据输入参数的值提取出来存储在%ebp-0x48到%epb-0x34的地址处，也就是按照一定的顺序这里是降序（下图中蓝线部分，对新的链表的第一个节点的值和第二个结点的值作了比较，得出第一个节点的值大于等于第二个节点的值）把链表各个节点的地址存储在栈帧中：

得出降序排列的语句：

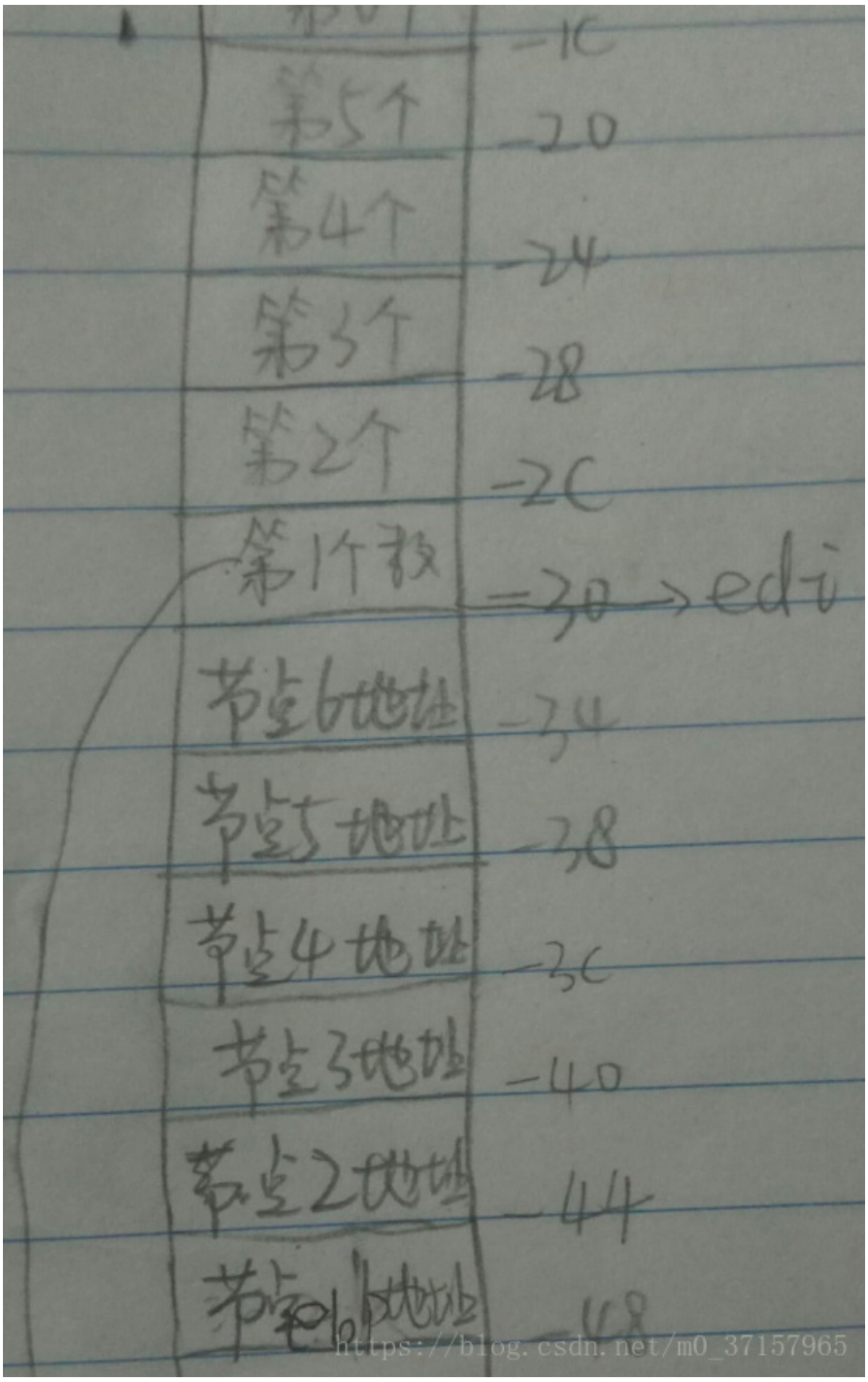
```
0x08048d49 <+192>:   mov    0x8(%ebx),%eax
0x08048d4c <+195>:   mov    (%ebx),%edx
0x08048d4e <+197>:   cmp    (%eax),%edx
0x08048d50 <+199>:   jge    0x8048d57 <phase_6+206>
```

```
0x08048ce6 <+93>:    mov    $0x0,%ebx
0x08048ceb <+98>:    lea    -0x30(%ebp),%edi
0x08048cee <+101>:   jmp    0x8048d06 <phase_6+125>
0x08048cf0 <+103>:   mov    0x8(%edx),%edx
0x08048cf3 <+106>:   add    $0x1,%eax
0x08048cf6 <+109>:   cmp    %ecx,%eax
0x08048cf8 <+111>:   jne    0x8048cf0 <phase_6+103>
0x08048cfa <+113>:   mov    %edx,-0x48(%ebp,%esi,4)
0x08048cfe <+117>:   add    $0x1,%ebx
0x08048d01 <+120>:   cmp    $0x6,%ebx
0x08048d04 <+123>:   je     0x8048d1c <phase_6+147>
0x08048d06 <+125>:   mov    %ebx,%esi
0x08048d08 <+127>:   mov    (%edi,%ebx,4),%ecx
Type <return> to continue, or q <return> to quit---
0x08048d0b <+130>:   mov    $0x804c0c4,%edx
0x08048d10 <+135>:   mov    $0x1,%eax
0x08048d15 <+140>:   cmp    $0x1,%ecx
0x08048d18 <+143>:   jg     0x8048cf0 <phase_6+103>
0x08048d1a <+145>:   jmp    0x8048cfa <phase_6+113>
```


推导过程图：



执行完这部分代码的部分栈帧图，当然，下图中的节点1或者节点6指的是根据输入参数选取出的节点，所以这里的节点1指的就是链表中存储最大值的那个节点，其它同理：



6、接下来的汇编代码则是根据上一步得到的降序排序得到的节点地址信息把链表恢复的这么一个过程：

```
0x08048d1c <+147>:   mov    -0x48(%ebp),%ebx
0x08048d1f <+150>:   mov    -0x44(%ebp),%eax
0x08048d22 <+153>:   mov    %eax,0x8(%ebx)
0x08048d25 <+156>:   mov    -0x40(%ebp),%edx
0x08048d28 <+159>:   mov    %edx,0x8(%eax)
0x08048d2b <+162>:   mov    -0x3c(%ebp),%eax
0x08048d2e <+165>:   mov    %eax,0x8(%edx)
0x08048d31 <+168>:   mov    -0x38(%ebp),%edx
0x08048d34 <+171>:   mov    %edx,0x8(%eax)
0x08048d37 <+174>:   mov    -0x34(%ebp),%eax
0x08048d3a <+177>:   mov    %eax,0x8(%edx)
0x08048d3d <+180>:   movl   $0x0,0x8(%eax)
0x08048d44 <+187>:   mov    $0x0,%esi
0x08048d49 <+192>:   mov    0x8(%ebx),%eax
0x08048d4c <+195>:   mov    (%ebx),%edx
0x08048d4e <+197>:   cmp    (%eax),%edx
0x08048d50 <+199>:   jge    0x8048d57 <phase_6+206>
```


7、所以，只要能得到以地址0x804c0c4为首地址存储的这个链表存储的数据得出即可以推出输入的六个参数：

```
(gdb) x &0x804c0c4
Attempt to take address of value not located in memory.
(gdb) x/4xw  0x804c0c4
0x804c0c4 <node1>:      0x000001a7      0x00000001      0x0804c0b8      0x000003
(gdb) x/4xw  0x804c0b8
0x804c0b8 <node2>:      0x0000006c      0x00000002      0x0804c0ac      0x000001
(gdb) x/4xw  0x804c0ac
0x804c0ac <node3>:      0x00000155      0x00000003      0x0804c0a0      0x000000
(gdb) x/4xw  0x804c0a0
0x804c0a0 <node4>:      0x00000187      0x00000004      0x0804c094      0x000001
(gdb) x/4xw  0x804c094
0x804c094 <node5>:      0x000003bd      0x00000005      0x0804c088      0x000001
(gdb) x/4xw  0x804c088
0x804c088 <node6>:      0x00000255      0x00000006      0x00000000      0x000003
```


8、得到六个结点分别存储的值为：

1	2	3	4	5	6
0x1a7	0x06c	0x155	0x187	0x3bd	0x255

降序排列后得：

5	6	1	4	3	2
0x3bd	0x255	0x1a7	0x187	0x155	0x06c

9、根据前面的推导过程可以得出输入的六个参数为5、6、1、4、3、2，检验正确性：

```
Halfway there!
11 1
So you got that one.  Try this one.
5 115
Good work!  On to the next...
5 6 1 4 3 2
Congratulations! You've defused the bomb!
[Inferior 1 (process 2957) exited normally]
```


第六关成功通过。

总结：

通过本次实验真的真的让我读汇编代码的能力提升了很多很多，虽然读汇编代码整体性的把控不是很好，但较以前真的大大提升了，也掌握了利用gdb调试的一些基本方法，颇有收获。