|  |
| --- |
| **实验报告** |

|  |  |  |  |
| --- | --- | --- | --- |
| **实验名称：** | **Lab2** | **日期：** | **2024.12.10** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **班级：** | **U10M21003.05** | **学号：** | **2024302589** | **姓名：** | **苗清皓** |

1. **实验概述：**

**DrEvil埋下了六颗炸弹！现在需要你来拆除炸弹，方法是输入六个对应的密码。**

**Lab2包括一个无法编译的C文件bomb.c，以及一个二进制可执行文件bomb，当运行bomb文件时，它会要求输入6个字符串，如果其中的任何一句是错的，炸弹就会爆炸。我们必须利用调试工具gdb和反汇编工具objdump，对bomb进行逆向分析，依次找到满足要求的6个字符串，从而拆除炸弹。**

1. **实验分析：**

**写在前面：**

**对bomb文件进行处理，打开终端，输入objdump -d bomb > asm.txt对bomb进行反汇编并将汇编代码输出到asm.txt文件（以AT&T风格）。**

**也可以直接输入gdb bomb进行调试，然后输入disas + 【要查看的函数名】，查看对应的汇编代码。**

1. **phase\_1**

Dump of assembler code for function phase\_1:

0x08048bc6 <+0>: push %ebp

0x08048bc7 <+1>: mov %esp,%ebp

0x08048bc9 <+3>: sub $0x10,%esp

0x08048bcc <+6>: push $0x804a2e8

0x08048bd1 <+11>: push 0x8(%ebp)

0x08048bd4 <+14>: call 0x804912d <strings\_not\_equal>

0x08048bd9 <+19>: add $0x10,%esp

0x08048bdc <+22>: test %eax,%eax

0x08048bde <+24>: je 0x8048be5 <phase\_1+31>

0x08048be0 <+26>: call 0x80493a1 <explode\_bomb>

0x08048be5 <+31>: leave

0x08048be6 <+32>: ret

End of assembler dump.

我们第一眼就看到了<strings\_not\_equal>，经过分析，这是将我们输入的密码与正确的密码比较看是否一致的函数，如果不一致，则引爆炸弹。那么在这个函数之前的，程序内置的值要引起我们的特别注意。

向上寻找，发现了两处push：

0x08048bcc <+6>: push $0x804a2e8

0x08048bd1 <+11>: push 0x8(%ebp)

一处可疑地址0x804a2e8，另一处是我们输入的字符串的起始地址，作为<strings\_not\_equal>的两个实参传入，然后比较这两个字符串是否一致。

我们立刻用x/s 0x804a2e8指令查看一下里面写了什么：

0x804a2e8: "I was trying to give Tina Fey more material."

我们现在就可以确定这就是第一个密码。

成功拆除第一个炸弹。

1. **phase\_2**

Dump of assembler code for function phase\_2:

0x08048be7 <+0>: push %ebp

0x08048be8 <+1>: mov %esp,%ebp

保护寄存器的作用，与实际题目无关，不用看这里了

0x08048bea <+3>: push %ebx

0x08048beb <+4>: sub $0x2c,%esp

0x08048bee <+7>: mov %gs:0x14,%eax

0x08048bf4 <+13>: mov %eax,-0xc(%ebp)

0x08048bf7 <+16>: xor %eax,%eax

0x08048bf9 <+18>: lea -0x24(%ebp),%eax //加载有效地址

0x08048bfc <+21>: push %eax //加载有效地址

0x08048bfd <+22>: push 0x8(%ebp) //加载实参

0x08048c00 <+25>: call 0x80493e1 <read\_six\_numbers> //读入6个整数

0x08048c05 <+30>: add $0x10,%esp //将栈顶向下移动4个单元，空出空间

0x08048c08 <+33>: cmpl $0x0,-0x24(%ebp)

0x08048c0c <+37>: jns 0x8048c13 <phase\_2+44>

//条件跳转指令。在 x86 汇编语言中，jns 指令代表 “Jump if Not Sign”，即如果符号标志（SF）为0（表示结果为非负数），则执行跳转，这里说明-0x24(%ebp)一定要大于等于0，才不会爆炸。

0x08048c0e <+39>: call 0x80493a1 <explode\_bomb>

0x08048c13 <+44>: mov $0x1,%ebx

0x08048c18 <+49>: mov %ebx,%eax

//eax和ebx为1

0x08048c1a <+51>: add -0x28(%ebp,%ebx,4),%eax

//eax=eax+R[ebp-0x28+4\*ebp]

0x08048c1e <+55>: cmp %eax,-0x24(%ebp,%ebx,4)

//这里把eax和后一位参数进行比较，而且二者一定相等才不会爆炸

0x08048c22 <+59>: je 0x8048c29 <phase\_2+66>

0x08048c24 <+61>: call 0x80493a1 <explode\_bomb>

0x08048c29 <+66>: add $0x1,%ebx

//ebx++

0x08048c2c <+69>: cmp $0x6,%ebx

//判断是否把六个数据都判断完了

0x08048c2f <+72>: jne 0x8048c18 <phase\_2+49>

//如果没完，开始循环

0x08048c31 <+74>: mov -0xc(%ebp),%eax

0x08048c34 <+77>: xor %gs:0x14,%eax

保护寄存器，与实际题目无关，不用看了。

0x08048c3b <+84>: je 0x8048c42 <phase\_2+91>

0x08048c3d <+86>: call 0x80487d0 <\_\_stack\_chk\_fail@plt>

0x08048c42 <+91>: mov -0x4(%ebp),%ebx

0x08048c45 <+94>: leave

0x08048c46 <+95>: ret

我们发现，每次循环，都把一个数加上当前循环的次数，而这个值与后一个数相等。

用C语言写出来：

int arr[6]; //这是输入的6个数字

for(int i=1;i<=6;i++)

{

If(arr[i-1]+i !=arr[i])

explode\_bomb;

}

于是我们令arr[0]等于1，得到1，2，4，7，11，16，但我们不知道输入的格式。于是我们查看一下<read\_six\_numbers>

以下是<read\_six\_numbers>的代码：

Dump of assembler code for function read\_six\_numbers:

0x080493e1 <+0>: push %ebp

0x080493e2 <+1>: mov %esp,%ebp

0x080493e4 <+3>: sub $0x8,%esp

0x080493e7 <+6>: mov 0xc(%ebp),%eax

0x080493ea <+9>: lea 0x14(%eax),%edx

0x080493ed <+12>: push %edx

0x080493ee <+13>: lea 0x10(%eax),%edx

0x080493f1 <+16>: push %edx

0x080493f2 <+17>: lea 0xc(%eax),%edx

0x080493f5 <+20>: push %edx

0x080493f6 <+21>: lea 0x8(%eax),%edx

0x080493f9 <+24>: push %edx

0x080493fa <+25>: lea 0x4(%eax),%edx

0x080493fd <+28>: push %edx

0x080493fe <+29>: push %eax

0x080493ff <+30>: push $0x804a589

0x08049404 <+35>: push 0x8(%ebp)

0x08049407 <+38>: call 0x8048860 <\_\_isoc99\_sscanf@plt>

//这里是sscanf函数调用

0x0804940c <+43>: add $0x20,%esp

0x0804940f <+46>: cmp $0x5,%eax

//意思是输入的数据必须比5大，要不然会爆炸

0x08049412 <+49>: jg 0x8049419 <read\_six\_numbers+56>

0x08049414 <+51>: call 0x80493a1 <explode\_bomb>

0x08049419 <+56>: leave

0x0804941a <+57>: ret

在调用函数前又出现了可疑的地址，再用同样的方法：

(gdb) x/s 0x804a589

0x804a589: "%d %d %d %d %d %d"

可以看出是读入6个int型数字，用空格间隔开，现在我们知道了输入的格式。

按照前面得出的规律，可知密码为：

1 2 4 7 11 16

第二个炸弹拆除成功！

1. **phase\_3**

Dump of assembler code for function phase\_3:

0x08048c47 <+0>: push %ebp

0x08048c48 <+1>: mov %esp,%ebp

0x08048c4a <+3>: sub $0x24,%esp

0x08048c4d <+6>: mov %gs:0x14,%eax

0x08048c53 <+12>: mov %eax,-0xc(%ebp)

0x08048c56 <+15>: xor %eax,%eax

//依然是从下面开始看

0x08048c58 <+17>: lea -0x10(%ebp),%eax

0x08048c5b <+20>: push %eax

0x08048c5c <+21>: lea -0x15(%ebp),%eax

0x08048c5f <+24>: push %eax

0x08048c60 <+25>: lea -0x14(%ebp),%eax

0x08048c63 <+28>: push %eax

0x08048c64 <+29>: push $0x804a33e

0x08048c69 <+34>: push 0x8(%ebp)

//这应该是我们输入的密码的起始地址

先查看输入什么样的密码：

(gdb) x/s 0x804a33e

0x804a33e: "%d %c %d"

0x08048c6c <+37>: call 0x8048860 <\_\_isoc99\_sscanf@plt>

0x08048c71 <+42>: add $0x20,%esp

0x08048c74 <+45>: cmp $0x2,%eax

//第7、8、9行，比较%eax与2，这是出于程序健壮性的考虑，%eax存放sscanf读到的字符个数，当读入的字小于2时，即引爆炸弹。否则跳转。

0x08048c77 <+48>: jg 0x8048c7e <phase\_3+55>

0x08048c79 <+50>: call 0x80493a1 <explode\_bomb>

0x08048c7e <+55>: cmpl $0x7,-0x14(%ebp)

0x08048c82 <+59>: ja 0x8048d74 <phase\_3+301>

//如果大于7，会爆炸，这个数字范围要小于等于7

0x08048c88 <+65>: mov -0x14(%ebp),%eax

//下面根据这个数进行跳转，phase\_3考察了分支选择结构。应该有不止一组密码，会根据第一个数字不同而改变，在这里我们假定这个数是3

0x08048c8b <+68>: jmp \*0x804a350(,%eax,4)

//跳转到0x804a350+12=0x804a35c的位置

查看一下：(gdb) x/a 0x804a35c 0x804a35c: 0x8048cf2 <phase\_3+171>，跳转！

0x08048c92 <+75>: mov $0x65,%eax

0x08048c97 <+80>: cmpl $0x48,-0x10(%ebp)

0x08048c9b <+84>: je 0x8048d7e <phase\_3+311>

0x08048ca1 <+90>: call 0x80493a1 <explode\_bomb>

0x08048ca6 <+95>: mov $0x65,%eax

0x08048cab <+100>: jmp 0x8048d7e <phase\_3+311>

0x08048cb0 <+105>: mov $0x64,%eax

0x08048cb5 <+110>: cmpl $0x389,-0x10(%ebp)

0x08048cbc <+117>: je 0x8048d7e <phase\_3+311>

0x08048cc2 <+123>: call 0x80493a1 <explode\_bomb>

0x08048cc7 <+128>: mov $0x64,%eax

0x08048ccc <+133>: jmp 0x8048d7e <phase\_3+311>

0x08048cd1 <+138>: mov $0x72,%eax

0x08048cd6 <+143>: cmpl $0xba,-0x10(%ebp)

0x08048cdd <+150>: je 0x8048d7e <phase\_3+311>

0x08048ce3 <+156>: call 0x80493a1 <explode\_bomb>

0x08048ce8 <+161>: mov $0x72,%eax

0x08048ced <+166>: jmp 0x8048d7e <phase\_3+311>

0x08048cf2 <+171>: mov $0x74,%eax

0x08048cf7 <+176>: cmpl $0x36c,-0x10(%ebp)

0x08048cfe <+183>: je 0x8048d7e <phase\_3+311>

//这里0x36c与-0x10(%ebp)一定要相等，转换成十进制就是876

0x08048d00 <+185>: call 0x80493a1 <explode\_bomb>

0x08048d05 <+190>: mov $0x74,%eax

0x08048d0a <+195>: jmp 0x8048d7e <phase\_3+311>

0x08048d0c <+197>: mov $0x73,%eax

0x08048d11 <+202>: cmpl $0x213,-0x10(%ebp)

0x08048d18 <+209>: je 0x8048d7e <phase\_3+311>

0x08048d1a <+211>: call 0x80493a1 <explode\_bomb>

0x08048d1f <+216>: mov $0x73,%eax

0x08048d24 <+221>: jmp 0x8048d7e <phase\_3+311>

0x08048d26 <+223>: mov $0x6c,%eax

0x08048d2b <+228>: cmpl $0x3bd,-0x10(%ebp)

0x08048d32 <+235>: je 0x8048d7e <phase\_3+311>

0x08048d34 <+237>: call 0x80493a1 <explode\_bomb>

0x08048d39 <+242>: mov $0x6c,%eax

0x08048d3e <+247>: jmp 0x8048d7e <phase\_3+311>

0x08048d40 <+249>: mov $0x63,%eax

0x08048d45 <+254>: cmpl $0x27d,-0x10(%ebp)

0x08048d4c <+261>: je 0x8048d7e <phase\_3+311>

0x08048d4e <+263>: call 0x80493a1 <explode\_bomb>

0x08048d53 <+268>: mov $0x63,%eax

0x08048d58 <+273>: jmp 0x8048d7e <phase\_3+311>

0x08048d5a <+275>: mov $0x73,%eax

0x08048d5f <+280>: cmpl $0x32e,-0x10(%ebp)

0x08048d66 <+287>: je 0x8048d7e <phase\_3+311>

0x08048d68 <+289>: call 0x80493a1 <explode\_bomb>

0x08048d6d <+294>: mov $0x73,%eax

0x08048d72 <+299>: jmp 0x8048d7e <phase\_3+311>

0x08048d74 <+301>: call 0x80493a1 <explode\_bomb>

0x08048d79 <+306>: mov $0x61,%eax

0x08048d7e <+311>: cmp -0x15(%ebp),%al

//跳转来到这里。%al是eax寄存器的低8位（eax在跳转前被赋值为0x74，不是0x61！一定要注意！）要与-0x15(%ebp)相等。这里只有8位，是ascii码的表示范围，而且地址偏移量不是4的倍数，我们大胆猜测这就是那个字符！

经计算，al等于0111 0100，是十进制116，对应字符 ” t ”。

至此，我们得到密码为：

3 t 876

输入后第三个炸弹拆除完毕！

0x08048d81 <+314>: je 0x8048d88 <phase\_3+321>

0x08048d83 <+316>: call 0x80493a1 <explode\_bomb>

0x08048d88 <+321>: mov -0xc(%ebp),%eax

0x08048d8b <+324>: xor %gs:0x14,%eax

0x08048d92 <+331>: je 0x8048d99 <phase\_3+338>

0x08048d94 <+333>: call 0x80487d0 <\_\_stack\_chk\_fail@plt>

0x08048d99 <+338>: leave

0x08048d9a <+339>: ret

1. **phase\_4**

Dump of assembler code for function phase\_4:

0x08048df2 <+0>: push %ebp

0x08048df3 <+1>: mov %esp,%ebp

0x08048df5 <+3>: sub $0x18,%esp

0x08048df8 <+6>: mov %gs:0x14,%eax

0x08048dfe <+12>: mov %eax,-0xc(%ebp)

0x08048e01 <+15>: xor %eax,%eax

//从下面开始

0x08048e03 <+17>: lea -0x10(%ebp),%eax

0x08048e06 <+20>: push %eax

0x08048e07 <+21>: lea -0x14(%ebp),%eax

0x08048e0a <+24>: push %eax

0x08048e0b <+25>: push $0x804a595

//同理

(gdb) x/s 0x804a595

0x804a595: "%d %d"

0x08048e10 <+30>: push 0x8(%ebp)

0x08048e13 <+33>: call 0x8048860 <\_\_isoc99\_sscanf@plt>

0x08048e18 <+38>: add $0x10,%esp

0x08048e1b <+41>: cmp $0x2,%eax

0x08048e1e <+44>: jne 0x8048e26 <phase\_4+52>

//同理

0x08048e20 <+46>: cmpl $0xe,-0x14(%ebp)

0x08048e24 <+50>: jbe 0x8048e2b <phase\_4+57>

//-0x14(%ebp)的值必须小于等于14

0x08048e26 <+52>: call 0x80493a1 <explode\_bomb>

0x08048e2b <+57>: sub $0x4,%esp

0x08048e2e <+60>: push $0xe

0x08048e30 <+62>: push $0x0

0x08048e32 <+64>: push -0x14(%ebp)

0x08048e35 <+67>: call 0x8048d9b <func4>

//这里调用了<func4>，使用的参数是-0x14(%ebp)，0x0和0xe

经查看，这是一个递归函数！先不急着看递归，先往下分析

0x08048e3a <+72>: add $0x10,%esp

0x08048e3d <+75>: cmp $0x15,%eax

//发现eax必须等于0x15，而我们知道，eax寄存器一般是放返回值的，那么，递归的结果肯定是21（十进制的0x15）！

0x08048e40 <+78>: jne 0x8048e48 <phase\_4+86>

0x08048e42 <+80>: cmpl $0x15,-0x10(%ebp)

//在这里我们发现，递归的返回值是与我们输入的第二个数相等的！那么我们要输入的第一个数字就是递归函数结果是21时对应的输入！

0x08048e46 <+84>: je 0x8048e4d <phase\_4+91>

0x08048e48 <+86>: call 0x80493a1 <explode\_bomb>

0x08048e4d <+91>: mov -0xc(%ebp),%eax

0x08048e50 <+94>: xor %gs:0x14,%eax

0x08048e57 <+101>: je 0x8048e5e <phase\_4+108>

0x08048e59 <+103>: call 0x80487d0 <\_\_stack\_chk\_fail@plt>

0x08048e5e <+108>: leave

--Type <RET> for more, q to quit, c to continue without paging--

0x08048e5f <+109>: ret

End of assembler dump.

我们现在看看<func4>：

Dump of assembler code for function func4:

0x08048d9b <+0>: push %ebp

0x08048d9c <+1>: mov %esp,%ebp

0x08048d9e <+3>: push %esi

0x08048d9f <+4>: push %ebx

0x08048da0 <+5>: mov 0x8(%ebp),%edx

0x08048da3 <+8>: mov 0xc(%ebp),%ecx

0x08048da6 <+11>: mov 0x10(%ebp),%esi

0x08048da9 <+14>: mov %esi,%eax

0x08048dab <+16>: sub %ecx,%eax

0x08048dad <+18>: mov %eax,%ebx

0x08048daf <+20>: shr $0x1f,%ebx

0x08048db2 <+23>: add %ebx,%eax

0x08048db4 <+25>: sar %eax

0x08048db6 <+27>: lea (%eax,%ecx,1),%ebx

0x08048db9 <+30>: cmp %edx,%ebx

0x08048dbb <+32>: jle 0x8048dd2 <func4+55>

0x08048dbd <+34>: sub $0x4,%esp

0x08048dc0 <+37>: lea -0x1(%ebx),%eax

0x08048dc3 <+40>: push %eax

0x08048dc4 <+41>: push %ecx

0x08048dc5 <+42>: push %edx

0x08048dc6 <+43>: call 0x8048d9b <func4>

0x08048dcb <+48>: add $0x10,%esp

0x08048dce <+51>: add %ebx,%eax

0x08048dd0 <+53>: jmp 0x8048deb <func4+80>

0x08048dd2 <+55>: mov %ebx,%eax

0x08048dd4 <+57>: cmp %edx,%ebx

0x08048dd6 <+59>: jge 0x8048deb <func4+80>

0x08048dd8 <+61>: sub $0x4,%esp

0x08048ddb <+64>: push %esi

0x08048ddc <+65>: lea 0x1(%ebx),%eax

0x08048ddf <+68>: push %eax

0x08048de0 <+69>: push %edx

0x08048de1 <+70>: call 0x8048d9b <func4>

0x08048de6 <+75>: add $0x10,%esp

0x08048de9 <+78>: add %ebx,%eax

0x08048deb <+80>: lea -0x8(%ebp),%esp

0x08048dee <+83>: pop %ebx

0x08048def <+84>: pop %esi

0x08048df0 <+85>: pop %ebp

0x08048df1 <+86>: ret

我们发现<func4>不仅是个递归，还是有三个函数的递归，而且是分情况的递归。

把<func4>转换成C语言得：

int func4(int a1, int a2, int a3)

{

int v3; // ebx

int result; // eax

v3 = (a3 - a2) / 2 + a2;

if ( v3 > a1 )

return v3 + func4(a1, a2, v3 - 1);

result = (a3 - a2) / 2 + a2;

if ( v3 < a1 )

return v3 + func4(a1, v3 + 1, a3);

return result;

}

单靠大脑分析有点困难，我把它扩展为一个C语言程序运行，在输入小于等于14的情况下穷举出第一个数字是6。

我们输入6 21，第四个炸弹拆除成功！

1. **phase\_5**

Dump of assembler code for function phase\_5:

0x08048e60 <+0>: push %ebp

0x08048e61 <+1>: mov %esp,%ebp

0x08048e63 <+3>: push %ebx

0x08048e64 <+4>: sub $0x20,%esp

0x08048e67 <+7>: mov 0x8(%ebp),%ebx

0x08048e6a <+10>: mov %gs:0x14,%eax

0x08048e70 <+16>: mov %eax,-0xc(%ebp)

0x08048e73 <+19>: xor %eax,%eax

0x08048e75 <+21>: push %ebx

0x08048e76 <+22>: call 0x804910b <string\_length>

0x08048e7b <+27>: add $0x10,%esp

0x08048e7e <+30>: cmp $0x6,%eax

0x08048e81 <+33>: je 0x8048e88 <phase\_5+40>

0x08048e83 <+35>: call 0x80493a1 <explode\_bomb>

0x08048e88 <+40>: mov $0x0,%eax

0x08048e8d <+45>: movzbl (%ebx,%eax,1),%edx

//把ebx+eax地址的值的低8位进行零扩展到32位，送edx，推测出放的应该是字符，通过循环读取

0x08048e91 <+49>: and $0xf,%edx

//edx的值经过运算，只保留低4位

0x08048e94 <+52>: movzbl 0x804a370(%edx),%edx

//从 %edx 寄存器中的地址加上 0x804a370 的偏移量处读取一个字节，这里的edx相当于一个下标的作用。四位二进制能表示的最大无符号整数为1111，即15，那么用edx作为下标，其范围是0~15，十六个数字。

查看0x804a370开始的字符串：

(gdb) x/s 0x804a370

0x804a370 <array.3252>: "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"

发现DrEvil的话前面还藏着一串字符，这应该是解决问题的关键。

我们数一数，情理之中地发现，这串字符正好16个。对应了edx能成为下标。

0x08048e9b <+59>: mov %dl,-0x13(%ebp,%eax,1)

//edx寄存器的低8位传送给-0x13(%ebp,%eax,1)

0x08048e9f <+63>: add $0x1,%eax

//这是个循环，eax递增，说明把6个字符的低8位放进以-0x13(%ebp)起始的地址里

0x08048ea2 <+66>: cmp $0x6,%eax

0x08048ea5 <+69>: jne 0x8048e8d <phase\_5+45>

0x08048ea7 <+71>: movb $0x0,-0xd(%ebp)

0x08048eab <+75>: sub $0x8,%esp

0x08048eae <+78>: push $0x804a347

//明显的提示，查看一下(gdb) x/s 0x804a347

0x804a347: "flyers"

0x08048eb3 <+83>: lea -0x13(%ebp),%eax

//把刚才字符串的起始地址加载到eax中

0x08048eb6 <+86>: push %eax

0x08048eb7 <+87>: call 0x804912d <strings\_not\_equal>

//又来<strings\_not\_equal>，说明刚才的字符串就是flyers

0x08048ebc <+92>: add $0x10,%esp

0x08048ebf <+95>: test %eax,%eax

0x08048ec1 <+97>: je 0x8048ec8 <phase\_5+104>

//eax要等于0

0x08048ec3 <+99>: call 0x80493a1 <explode\_bomb>

0x08048ec8 <+104>: mov -0xc(%ebp),%eax

0x08048ecb <+107>: xor %gs:0x14,%eax

0x08048ed2 <+114>: je 0x8048ed9 <phase\_5+121>

0x08048ed4 <+116>: call 0x80487d0 <\_\_stack\_chk\_fail@plt>

0x08048ed9 <+121>: mov -0x4(%ebp),%ebx

0x08048edc <+124>: leave

0x08048edd <+125>: ret

End of assembler dump.

经分析，我们发现我们输入的字符要与flyers和maduiersnfotvbyl之间有联系：

设输入的字符为str1，maduiersnfotvbyl为str2

我们输入的字符经过转换，只保留ascii码的低四位，赋给edx，然后edx作为str2的下标，对应找到各个字符，找到的字符会组合成flyers！

用逆向思维举例来说：

flyers第一个字符为f，即str2[9],9的二进制低四位为1001，那么str1[0]的第四位为1001，而前四位可变。我们设前四位为0110，得出第一个字符为i，以此类推，得到ionefg！

输入成功拆除第二个炸弹！

1. **phase\_6**

Dump of assembler code for function phase\_6:

0x08048ede <+0>: push %ebp

0x08048edf <+1>: mov %esp,%ebp

0x08048ee1 <+3>: push %esi

0x08048ee2 <+4>: push %ebx

0x08048ee3 <+5>: sub $0x48,%esp

0x08048ee6 <+8>: mov %gs:0x14,%eax

0x08048eec <+14>: mov %eax,-0xc(%ebp)

0x08048eef <+17>: xor %eax,%eax

0x08048ef1 <+19>: lea -0x3c(%ebp),%eax

0x08048ef4 <+22>: push %eax

0x08048ef5 <+23>: push 0x8(%ebp)

0x08048ef8 <+26>: call 0x80493e1 <read\_six\_numbers>

//看出还是输入6个int型数字

0x08048efd <+31>: add $0x10,%esp

0x08048f00 <+34>: mov $0x0,%esi

0x08048f05 <+39>: mov -0x3c(%ebp,%esi,4),%eax

0x08048f09 <+43>: sub $0x1,%eax

0x08048f0c <+46>: cmp $0x5,%eax

0x08048f0f <+49>: jbe 0x8048f16 <phase\_6+56>

//检验是否是6个数字

0x08048f11 <+51>: call 0x80493a1 <explode\_bomb>

0x08048f16 <+56>: add $0x1,%esi

0x08048f19 <+59>: cmp $0x6,%esi

0x08048f1c <+62>: je 0x8048f39 <phase\_6+91>

0x08048f1e <+64>: mov %esi,%ebx

0x08048f20 <+66>: mov -0x3c(%ebp,%ebx,4),%eax

0x08048f24 <+70>: cmp %eax,-0x40(%ebp,%esi,4)

0x08048f28 <+74>: jne 0x8048f2f <phase\_6+81>

0x08048f2a <+76>: call 0x80493a1 <explode\_bomb>

0x08048f2f <+81>: add $0x1,%ebx

0x08048f32 <+84>: cmp $0x5,%ebx

0x08048f35 <+87>: jle 0x8048f20 <phase\_6+66>

0x08048f37 <+89>: jmp 0x8048f05 <phase\_6+39>

//这是个双层循环，检验6个数字必须在[1,6]范围内，而且不能相等

0x08048f39 <+91>: lea -0x3c(%ebp),%eax

0x08048f3c <+94>: lea -0x24(%ebp),%ebx

0x08048f3f <+97>: mov $0x7,%ecx

0x08048f44 <+102>: mov %ecx,%edx

0x08048f46 <+104>: sub (%eax),%edx

0x08048f48 <+106>: mov %edx,(%eax)

0x08048f4a <+108>: add $0x4,%eax

0x08048f4d <+111>: cmp %eax,%ebx

0x08048f4f <+113>: jne 0x8048f44 <phase\_6+102>

//设输入的数字为x，则经过这部分变换，每个都变成了对应的7-x

0x08048f51 <+115>: mov $0x0,%ebx

0x08048f56 <+120>: jmp 0x8048f6e <phase\_6+144>

0x08048f58 <+122>: mov 0x8(%edx),%edx

0x08048f5b <+125>: add $0x1,%eax

0x08048f5e <+128>: cmp %ecx,%eax

0x08048f60 <+130>: jne 0x8048f58 <phase\_6+122>

0x08048f62 <+132>: mov %edx,-0x24(%ebp,%esi,4)

0x08048f66 <+136>: add $0x1,%ebx

0x08048f69 <+139>: cmp $0x6,%ebx

0x08048f6c <+142>: je 0x8048f85 <phase\_6+167>

0x08048f6e <+144>: mov %ebx,%esi

0x08048f70 <+146>: mov -0x3c(%ebp,%ebx,4),%ecx

0x08048f74 <+150>: mov $0x1,%eax

0x08048f79 <+155>: mov $0x804c154,%edx

//有地址就查看一下：

(gdb) x/d 0x804c154

0x804c154 <node1>: 261

发现有个node，是链表的英文！查看一下整个链表：

<node1>: 0x00000105 0x00000001 0x0804c160

<node2>: 0x000001d9 0x00000002 0x0804c16c

<node3>: 0x000002b8 0x00000003 0x0804c178

<node4>: 0x000003c4 0x00000004 0x0804c184

<node5>: 0x00000099 0x00000005 0x0804c190

<node6>: 0x00000128 0x00000006 0x00000000

0x08048f7e <+160>: cmp $0x1,%ecx

0x08048f81 <+163>: jg 0x8048f58 <phase\_6+122>

0x08048f83 <+165>: jmp 0x8048f62 <phase\_6+132>

0x08048f85 <+167>: mov -0x24(%ebp),%ebx

0x08048f88 <+170>: lea -0x24(%ebp),%eax

0x08048f8b <+173>: lea -0x10(%ebp),%esi

0x08048f8e <+176>: mov %ebx,%ecx

0x08048f90 <+178>: mov 0x4(%eax),%edx

0x08048f93 <+181>: mov %edx,0x8(%ecx)

0x08048f96 <+184>: add $0x4,%eax

0x08048f99 <+187>: mov %edx,%ecx

0x08048f9b <+189>: cmp %eax,%esi

0x08048f9d <+191>: jne 0x8048f90 <phase\_6+178>

0x08048f9f <+193>: movl $0x0,0x8(%edx)

0x08048fa6 <+200>: mov $0x5,%esi

0x08048fab <+205>: mov 0x8(%ebx),%eax

0x08048fae <+208>: mov (%eax),%eax

0x08048fb0 <+210>: cmp %eax,(%ebx)

0x08048fb2 <+212>: jge 0x8048fb9 <phase\_6+219>

0x08048fb4 <+214>: call 0x80493a1 <explode\_bomb>

0x08048fb9 <+219>: mov 0x8(%ebx),%ebx

0x08048fbc <+222>: sub $0x1,%esi

0x08048fbf <+225>: jne 0x8048fab <phase\_6+205>

//实现的效果是：将链表节点的顺序调换成数组中的顺序

0x08048fc1 <+227>: mov -0xc(%ebp),%eax

0x08048fc4 <+230>: xor %gs:0x14,%eax

0x08048fcb <+237>: je 0x8048fd2 <phase\_6+244>

0x08048fcd <+239>: call 0x80487d0 <\_\_stack\_chk\_fail@plt>

0x08048fd2 <+244>: lea -0x8(%ebp),%esp

0x08048fd5 <+247>: pop %ebx

0x08048fd6 <+248>: pop %esi

0x08048fd7 <+249>: pop %ebp

0x08048fd8 <+250>: ret

End of assembler dump.

现在要把链表第一个元素按降序排列，0x3c4>0x2b8>0x1d9>0x128>0x105>0x99,

对应就是4 3 2 6 1 5

但是我们输入数字x后还要进行7-x的操作。

综上，可以得出输入的密码是3 4 5 1 6 2

拆弹成功结束！

1. **实验总结：**

**Lab bomb的实验都十分精巧，尤其是phase\_5和phase**

**-6,都经过了多步变换，设计循环，分支选择，链表等结构。让我熟悉了多种结构的汇编语言的表示，由高级语言深入到机器级表示，也加深了对gdb调试和逆向工程的理解，对AT&T风格的**

**在分析汇编语言时，尤其要专注和仔细，把大脑当作一台CPU来进行计算，如果有一点不严谨或者动摇，就会失之毫厘，差之千里，使我理解到了掌握扎实的汇编代码和C语言基础对未来深入学习计算机科学与技术的重要性。**