Lab2 二进制炸弹

李仁杰 13307130279

一、实验目的

理解IA32汇编语言，学会使用Linux操作系统和gdb调试器。

二、实验介绍

邪恶博士(Dr. Evil)在城市的服务器上种植了一大批“二进制炸弹”。二进制炸弹是由一系列阶段(phase)组成的程序。在每个阶段，应当输入一个特定的字符串类型。如果输入正确的字符串，那么炸弹将被拆除，进入下一个阶段。否则，炸弹将爆炸(Boom!!!)然后终止。

为了阻止邪恶博士(Dr. Evil)的计划，拯救整个城市，勇敢的英雄需要通过反汇编和逆向工程来确定是哪六个字符串，从而解除他们各自炸弹的雷管。

三、实验过程

1.使用ssh登录、修改密码

(1)在Linux系统下，用Ctrl+Alt+T调出终端，使用命令ssh stu13307130279@10.92.13.8，输入密码(csapp)，使用命令passwd修改密码。

(2)使用命令ls后看到有一个文件bomb68.tar，输入 tar -xvf bomb68.tar 解压后，再用命令ls看到目录下新出现了bomb、bomb.c、ID和README四个文件。输入cat bomb.c后，把c代码看了一遍，发现原来这里只有主函数，没有每一关的具体代码。于是我们要从汇编代码入手。

(3)输入反汇编命令objdump -d bomb >> bomb.s，将汇编代码放入bomb.s文件中进行查看。

```
stu13307130279@ubuntu-catlab-8: ~
stu13307130279@ubuntu-catlab-8:~$ ls
bomb68.tar
stu13307130279@ubuntu-catlab-8:~$ tar -xvf bomb68.tar
bomb
bomb.c
ID
README
stu13307130279@ubuntu-catlab-8:~$ objdump -d bomb >> bomb.s
```

2.关卡1(phase_1)

首先我们找到main函数，发现它调用了从phase1到phase6这六个函数。这应该就是每一关需要看懂的函数。于是找到phase1,代码如下：

```
stu13307130279@ubuntu-catlab-8: ~
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x08048c10 <+0>:     sub    $0x1c,%esp
   0x08048c13 <+3>:     movl   $0x8049984,0x4(%esp)
   0x08048c1b <+11>:    mov    0x20(%esp),%eax
   0x08048c1f <+15>:    mov    %eax,(%esp)
   0x08048c22 <+18>:    call   0x8048fd4 <strings_not_equal>
   0x08048c27 <+23>:    test   %eax,%eax
   0x08048c29 <+25>:    je     0x8048c30 <phase_1+32>
   0x08048c2b <+27>:    call   0x8049563 <explode_bomb>
   0x08048c30 <+32>:    add    $0x1c,%esp
   0x08048c33 <+35>:    ret
```

可以看到在%eax!=0的时候就会调用<explode_bomb>，所以在调用<strings_not_equal> 函数之后的返回值%eax必须为0。继续往前，发现代码movl $0x8049984,0x4(%esp)有立即数，是将此处地址的值拿来用。于是我们在*0x8048c27处设置断点，紧接着我们随便输入一个字符串，用

print (char*)0x8049984查看内容，终端显示出字符串"For NASA, space is still a high priority."。

```
(gdb) break *0x8048c27
Breakpoint 1 at 0x8048c27
(gdb) run
Starting program: /home/13EE_CSAPP/student_users/stu13307130279/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Fuck you, leather man!

Breakpoint 1, 0x08048c27 in phase_1 ()
(gdb) print (char*)0x8049984
$1 = 0x8049984 "For NASA, space is still a high priority."
(gdb)
```

于是开始第一关的尝试。非常重要的一步是在<explode_bomb>之前设置断点，找到该函数的入口地址是0x8049563,即:break *0x8049563。然后输入命令run，进入程序，在输入提示的下一行输入"For NASA, space is still a high priority."，终端显示“Phase 1 defused. How about the next one?”也即第一关顺利通过。

3.关卡2(phase_2)

```
stu13307130279@ubuntu-catlab-8: ~
ump of assembler code for function phase_2:
  0x08048c34 <+0>:     push   %ebx
  0x08048c35 <+1>:     sub    $0x38,%esp
  0x08048c38 <+4>:     lea    0x18(%esp),%eax
  0x08048c3c <+8>:     mov    %eax,0x4(%esp)
  0x08048c40 <+12>:    mov    0x40(%esp),%eax
  0x08048c44 <+16>:    mov    %eax,(%esp)
  0x08048c47 <+19>:    call   0x8049668 <read_six_numbers>
  0x08048c4c <+24>:    cmpl   $0x1,0x18(%esp)
  0x08048c51 <+29>:    je     0x8048c58 <phase_2+36>
  0x08048c53 <+31>:    call   0x8049563 <explode_bomb>
  0x08048c58 <+36>:    mov    $0x1,%ebx
  0x08048c5d <+41>:    mov    %ebx,%eax
  0x08048c5f <+43>:    add    $0x1,%ebx
  0x08048c62 <+46>:    mov    %ebx,%edx
  0x08048c64 <+48>:    imul   0x14(%esp,%eax,4),%edx
  0x08048c69 <+53>:    cmp    %edx,0x18(%esp,%eax,4)
  0x08048c6d <+57>:    je     0x8048c74 <phase_2+64>
  0x08048c6f <+59>:    call   0x8049563 <explode_bomb>
  0x08048c74 <+64>:    cmp    $0x6,%ebx
  0x08048c77 <+67>:    jne    0x8048c5d <phase_2+41>
  0x08048c79 <+69>:    add    $0x38,%esp
  0x08048c7c <+72>:    pop    %ebx
```

首先要注意到的是<read_six_numbers>这个函数，根据名字的提示可以猜测这一关要我输入六个数字。紧接着这个函数三条指令的是cmpl $0x1,0x18(%esp)，je 0x8048c58 <phase_2+36>和call 0x8049563 <explode_bomb>，也就是如果输入的第一个数不等于1则炸弹爆炸。因此第一个数为1。接下来橙色方框中的内容是一个循环，两条重要指令是imul 0x14(%esp,%eax,4),%edx和cmp %edx,0x18(%esp,%eax,4)，其中%eax 和%edx在每次循环中加1。假设有数组A,由于数组地址是从小到大增长，所以地址数组索引b=a+1，根据指令可以得出A[b]=A[a]*%edx，其中%edx就

是%ebx，每次%edx就等于循环变量（其中变量=2，3，4，5，6）。初始时，A[1]=1。所以，A[2] = A[1] * 2 = 2，A[3] = A[2] * 3 = 6，A[4] = A[3] * 4 = 24，A[5] = A[4] * 5 = 120，A[6] = A[5] * 6 = 720。

4.关卡3(phase_3)

```
Dump of assembler code for function phase_3:
   0x08048c7e <+0>:     sub    $0x2c,%esp
   0x08048c81 <+3>:     lea    0x1c(%esp),%eax
   0x08048c85 <+7>:     mov    %eax,0xc(%esp)
   0x08048c89 <+11>:    lea    0x18(%esp),%eax
   0x08048c8d <+15>:    mov    %eax,0x8(%esp)
   0x08048c91 <+19>:    movl   $0x8049d7a,0x4(%esp)
   0x08048c99 <+27>:    mov    0x30(%esp),%eax
   0x08048c9d <+31>:    mov    %eax,(%esp)
   0x08048ca0 <+34>:    call   0x8048910 <__isoc99_sscanf@plt>
   0x08048ca5 <+39>:    cmp    $0x1,%eax
   0x08048ca8 <+42>:    jg     0x8048caf <phase_3+49>
   0x08048caa <+44>:    call   0x8049563 <explode_bomb>
   0x08048caf <+49>:    cmpl   $0x7,0x18(%esp)
   0x08048cb4 <+54>:    ja     0x8048cf2 <phase_3+116>
   0x08048cb6 <+56>:    mov    0x18(%esp),%eax
   0x08048cba <+60>:    jmp    *0x80499e0(,%eax,4)
   0x08048cc1 <+67>:    mov    $0x2ae,%eax
   0x08048cc6 <+72>:    jmp    0x8048d03 <phase_3+133>
   0x08048cc8 <+74>:    mov    $0x5e,%eax
   0x08048ccd <+79>:    jmp    0x8048d03 <phase_3+133>
   0x08048ccf <+81>:    mov    $0x1b4,%eax
   0x08048cd4 <+86>:    jmp    0x8048d03 <phase_3+133>
   0x08048cd6 <+88>:    mov    $0x115,%eax
   0x08048cdb <+93>:    jmp    0x8048d03 <phase_3+133>
   0x08048cdd <+95>:    mov    $0x60,%eax
   0x08048ce2 <+100>:   jmp    0x8048d03 <phase_3+133>
   0x08048ce4 <+102>:   mov    $0x294,%eax
   0x08048ce9 <+107>:   jmp    0x8048d03 <phase_3+133>
   0x08048ceb <+109>:   mov    $0x1fc,%eax
   0x08048cf0 <+114>:   jmp    0x8048d03 <phase_3+133>
   0x08048cf2 <+116>:   call   0x8049563 <explode_bomb>
   0x08048cf7 <+121>:   mov    $0x0,%eax
   0x08048cfc <+126>:   jmp    0x8048d03 <phase_3+133>
   0x08048cfe <+128>:   mov    $0x2b2,%eax
   0x08048d03 <+133>:   cmp    0x1c(%esp),%eax
   0x08048d07 <+137>:   je     0x8048d0e <phase_3+144>
   0x08048d09 <+139>:   call   0x8049563 <explode_bomb>
   0x08048d0e <+144>:   add    $0x2c,%esp
   0x08048d11 <+147>:   ret
```

注意到黄色下划线处movl $0x8049d7a,0x4(%esp)，输入指令print (char*)0x8049d7a，得到0x8049d7a: "%d %d"，显示出应该输入两个数字。而后面的绿色方框处cmp $0x1,%eax表明输入参数必须多于1个。再往下到达cmpl $0x7,0x18(%esp)，即输入的第一个参数值必须小于等于7。然后看到jmp *0x80499e0(,%eax,4)，这是典型的switch跳转语句，即跳转到以地址*0x80499e0

为基址的跳转表中。根据地址0x80496cc + 4%eax中值确定跳转地址，也就是说跳转表存在0x80496cc ~ 0x80496cc + 4 * 7 的地址段中。

```
(gdb) print /x *0x80499e0
$1 = 0x8048cc1
(gdb) print /x *(0x80499e0+4)
$2 = 0x8048cfe
(gdb) print /x *(0x80499e0+8)
$3 = 0x8048cc8
(gdb) print /x *(0x80499e0+12)
$4 = 0x8048ccf
(gdb) print /x *(0x80499e0+16)
$5 = 0x8048cd6
(gdb) print /x *(0x80499e0+20)
$6 = 0x8048cdd
(gdb) print /x *(0x80499e0+24)
$7 = 0x8048ce4
(gdb) print /x *(0x80499e0+28)
$8 = 0x8048ceb
```

输入print /x *0x80499e0，得到地址0x8048cc1，在代码中找到该处指令，得到第一个输入为0时对应的第二个输入为0x2ae,转换成十进制为686。这个关卡总共有七组答案如下：

0 686
1 94
2 436
3 277
4 96
5 660
6 508

5.关卡4(phase_4)

```
Dump of assembler code for function phase_4:
   0x08048d3c <+0>:     sub    $0x2c,%esp
   0x08048d3f <+3>:     lea    0x1c(%esp),%eax
   0x08048d43 <+7>:     mov    %eax,0x8(%esp)
   0x08048d47 <+11>:    movl   $0x8049d7d,0x4(%esp)
   0x08048d4f <+19>:    mov    0x30(%esp),%eax
   0x08048d53 <+23>:    mov    %eax,(%esp)
   0x08048d56 <+26>:    call   0x8048910 <__isoc99_sscanf@plt>
   0x08048d5b <+31>:    cmp    $0x1,%eax
   0x08048d5e <+34>:    jne    0x8048d67 <phase_4+43>
   0x08048d60 <+36>:    cmpl   $0x0,0x1c(%esp)    0x1c(%esp)>0
   0x08048d65 <+41>:    jg     0x8048d6c <phase_4+48>
   0x08048d67 <+43>:    call   0x8049563 <explode_bomb>
   0x08048d6c <+48>:    mov    0x1c(%esp),%eax
   0x08048d70 <+52>:    mov    %eax,(%esp)
   0x08048d73 <+55>:    call   0x8048d12 <func4>
   0x08048d78 <+60>:    cmp    $0x1cb91,%eax      0x1cb91=117649=7^6
   0x08048d7d <+65>:    je     0x8048d84 <phase_4+72>
   0x08048d7f <+67>:    call   0x8049563 <explode_bomb>
   0x08048d84 <+72>:    add    $0x2c,%esp
   0x08048d87 <+75>:    ret
```

同样的由movl $0x8049d7d,0x4(%esp)我们知道这一关是要输入一个数字。由cmpl $0x0,0x1c(%esp) 知道输入的参数必须大于0。注意到这里调用了函数<func4>，找到代码如下：

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x08048d12 <+0>:     sub    $0x1c,%esp
   0x08048d15 <+3>:     mov    0x20(%esp),%edx
   0x08048d19 <+7>:     mov    $0x1,%eax
   0x08048d1e <+12>:    test   %edx,%edx
   0x08048d20 <+14>:    jle    0x8048d38 <func4+38>
   0x08048d22 <+16>:    sub    $0x1,%edx
   0x08048d25 <+19>:    mov    %edx,(%esp)
   0x08048d28 <+22>:    call   0x8048d12 <func4>
   0x08048d2d <+27>:    lea    0x0(,%eax,8),%edx  8*eax -> edx
   0x08048d34 <+34>:    sub    %eax,%edx   edx= edx-eax
   0x08048d36 <+36>:    mov    %edx,%eax  eax <- edx
   0x08048d38 <+38>:    add    $0x1c,%esp
   0x08048d3b <+41>:    ret
End of assembler dump.
```

后面的意思是*7

由test %edx, %edx 及下面的跳转代码知如果所传递的参数等于0则结束<func4>，跳转到主结构，返回%eax=1，否则减1并继续调用<func4>并在调用的<func4>又一次调用其本身，也就是进入双层递归模式。在满足跳出条件后，将每一次的值乘7，把它写成C代码：

```c
int func4(int n)
{
  if(n == 0)
    return 1;
  else
    return 7*(func(n-1));
}
```

这其实就是7的整数次方。回到<phase_4>，紧接着的代码是cmp $0x1cb91,%eax，说明<func4>返回值应该是0x1cb91,转为十进制是117649,也就是7的6次方。所以这里输入的值应该为6。

6.关卡5(phase_5)

```
Dump of assembler code for function phase_5:
   0x08048d88 <+0>:     push   %ebx
   0x08048d89 <+1>:     sub    $0x18,%esp
   0x08048d8c <+4>:     mov    0x20(%esp),%ebx
   0x08048d90 <+8>:     mov    %ebx,(%esp)
   0x08048d93 <+11>:    call   0x8048fbb <string_length>
   0x08048d98 <+16>:    cmp    $0x6,%eax  长度为6
   0x08048d9b <+19>:    je     0x8048da2 <phase_5+26>
   0x08048d9d <+21>:    call   0x8049563 <explode_bomb>
   0x08048da2 <+26>:    mov    $0x0,%edx
   0x08048da7 <+31>:    mov    $0x0,%eax
   0x08048dac <+36>:    movsbl (%ebx,%eax,1),%ecx
   0x08048db0 <+40>:    and    $0xf,%ecx
   0x08048db3 <+43>:    add    0x8049a00(,%ecx,4),%edx
   0x08048dba <+50>:    add    $0x1,%eax
   0x08048dbd <+53>:    cmp    $0x6,%eax
   0x08048dc0 <+56>:    jne    0x8048dac <phase_5+36>
   0x08048dc2 <+58>:    cmp    $0x1e,%edx
   0x08048dc5 <+61>:    je     0x8048dcc <phase_5+68>
   0x08048dc7 <+63>:    call   0x8049563 <explode_bomb>
   0x08048dcc <+68>:    add    $0x18,%esp
   0x08048dcf <+71>:    pop    %ebx
   0x08048dd0 <+72>:    ret
```

符号扩展

ebx+eax -> ecx

取ecx后四位

由以下两句代码call 8048f60 <string_length>和cmp $0x6,%eax可知我们需要输入一个长度为6的字符串。之后我们用movsbl (%edx,%eax,1),%ecx把字符串中的字符的ASCII码值送入%ecx，紧接着我们调用and $0xf,%ecx取ecx的低四位。然后我们看到后面的指令：add 0x8049a00(,%ecx,4),%edx，在终端输入x/20 0x8049a00，得到如下图所示的存储器值。

```
(gdb) x/20 0x8049a00
0x8049a00 <array.2950>: 0x00000002      0x0000000a      0x00000006      0x00000001
0x8049a10 <array.2950+16>:      0x0000000c      0x00000010      0x00000009      0x00000003
0x8049a20 <array.2950+32>:      0x00000004      0x00000007      0x0000000e      0x00000005
0x8049a30 <array.2950+48>:      0x0000000b      0x00000008      0x0000000f      0x0000000d
0x8049a40:      0x79206f53      0x7420756f      0x6b6e6968      0x756f7920
(gdb)
```

在依次处理完六个字符后，我们调用cmp $0x1e,%edx，查看%edx的值是否为0x1e，也就是10进制的30。为了凑出来这个值，我们找到存储器值序列中的1和9，其中对应的位置分别为12和24，也就是%ecx*4的值为12和24。那么，我们只要保证有三个ASCII码低四位为3的字符以及三个ASCII码低四位为6的字符出现就好了。考虑到’0’的ASCII码是0x30，那么我们只要输入字符串“333666”即可通过这一关。

7.关卡6(phase_6)

这里我们先观察<explode_bomb>前面的跳转条件，这里是cmp %edx,(%eax)。在调用完 <fun6>之后是一系列链表操作，而mov 0x804c174,%edx直接将%edx值设置好了，也就是我们输入的参数。对于函数<strtol@plt>，作用是将我们输入的数字当作字符串处理，然后转为十进制。简单地说就是输入什么十进制数参数就是那个十进制数本身。于是我在此处输入500，进入调试，在0x8048e70的位置设置断点，可以得到如下图所示的值：

```
Breakpoint 9, 0x08048e70 in phase_6 ()
(gdb) print $eax
$8 = 134529480
(gdb) print *(int*)$eax
$9 = 575
(gdb) print $edx
$10 = 500
```

接着我用(gdb) kill指令结束程序的运行，并重新设置断点0x8048e61，在第六关输入575。之后通过四次查询内存，确定这里输入575是正确的值。这题的fun6函数的确有点长，但是理解fun6对解题的帮助并不大，所以我们选择直接看结果的方式。

```
Breakpoint 3, 0x08048e61 in phase_6 ()
(gdb) print /x *($eax+8)
$1 = 0x804c1d4
(gdb) print /x *(0x804c1d4)
$2 = 0x2d0
(gdb) print /x *(0x804c1d4+8)
$3 = 0x804c198
(gdb) print /x *(0x804c198+8)
$4 = 0x804c1c8
(gdb) print /x *(0x804c1c8+8)
$5 = 0x804c174
(gdb) break *0x8048e72
Breakpoint 4 at 0x8048e72
```

7.迷の关卡(secret_phase)

通过阅读bomb.c，发现邪恶博士(Dr. Evil)还做了一个隐藏关卡，等待我们来解除。仔细看了与唯一提到进入秘密关卡的<phase_defused>函数，首先发现cmpl $0x6,0x804c78c，是判断是否已经通过了前面6关，没有的话就直接跳到结束部分。若已经通过6关，应该会往下执行。又有一句出现了立即数的movl $0x8049d80,0x4(%esp)，用x/s 0x8049d80，得到"%d %s"。下面是movl

$0x804c890,(%esp)，同样地，得到该地址处为数字6。然后调用函数<__isoc99_sscanf@plt>并在这次要求返回值为2。因为接收了2个参数。因此这里要求0x804a970处存放的是一个数字和一个字符串。在cmp $0x2,%eax处设置断点，运行后发现%eax=1。而地址 0x804a970仍然为数字6,这个数字和第四关的输入相同，而且第四关也调用了<__isoc99_sscanf@plt>这个函数。

为了验证猜想，再次使用gdb调试，在输入第四关答案时故意在6后面输入了字符串“OK”，结果第四关仍能通过。于是继续调试，在输入第六关答案之后程序再次遇到断点，这时查看地址0x804a890，发现除了数字6,还有字符串“OK”。这说明秘密关卡和第四关是相联系的。这就解决了cmp $0x2,%eax 的问题。继续往下走是：movl $0x8049d86,0x4(%esp)，该地址处存储着字符串 "austinpowers"。接下来调用了函数<strings_not_equal>。函数的另一个参数正是在第四关的“额外输入”。也就是说我在第四关输入的字符串就应该是"austinpowers"。如此则可满足调用该函数之后返回值为0的问题。

于是把第四关的输入改为6 austinpowers,之后可以发现在解除炸弹之后出现字符串"Curses, you've found the secret phase!"也就是说此时已经进入了秘密关卡。

通过观察神秘关卡的汇编代码，发现在调用完<fun7>之后，紧跟着cmp $0x1,%eax，也就是返回值必须为1。分析函数<fun7>发现它也是一个递归函数。递归最深处的返回值肯定是0,即最外层返回值为1，尝试用C语言重现<fun7>：

```
int fun7(const int *a, int b)
{
    if (a == NULL)
        return -1;
    int ret = 0;
    if (*a - b > 0)
    {
        ret = fun7(*(a + 4), b);
        ret *= 2
    }
    else if (*a - b == 0)
        return 0;
    else
    {
        ret = fun7(*(a + 8), b);
        ret = ret * 2 + 1;
    }
    return ret;
}
```

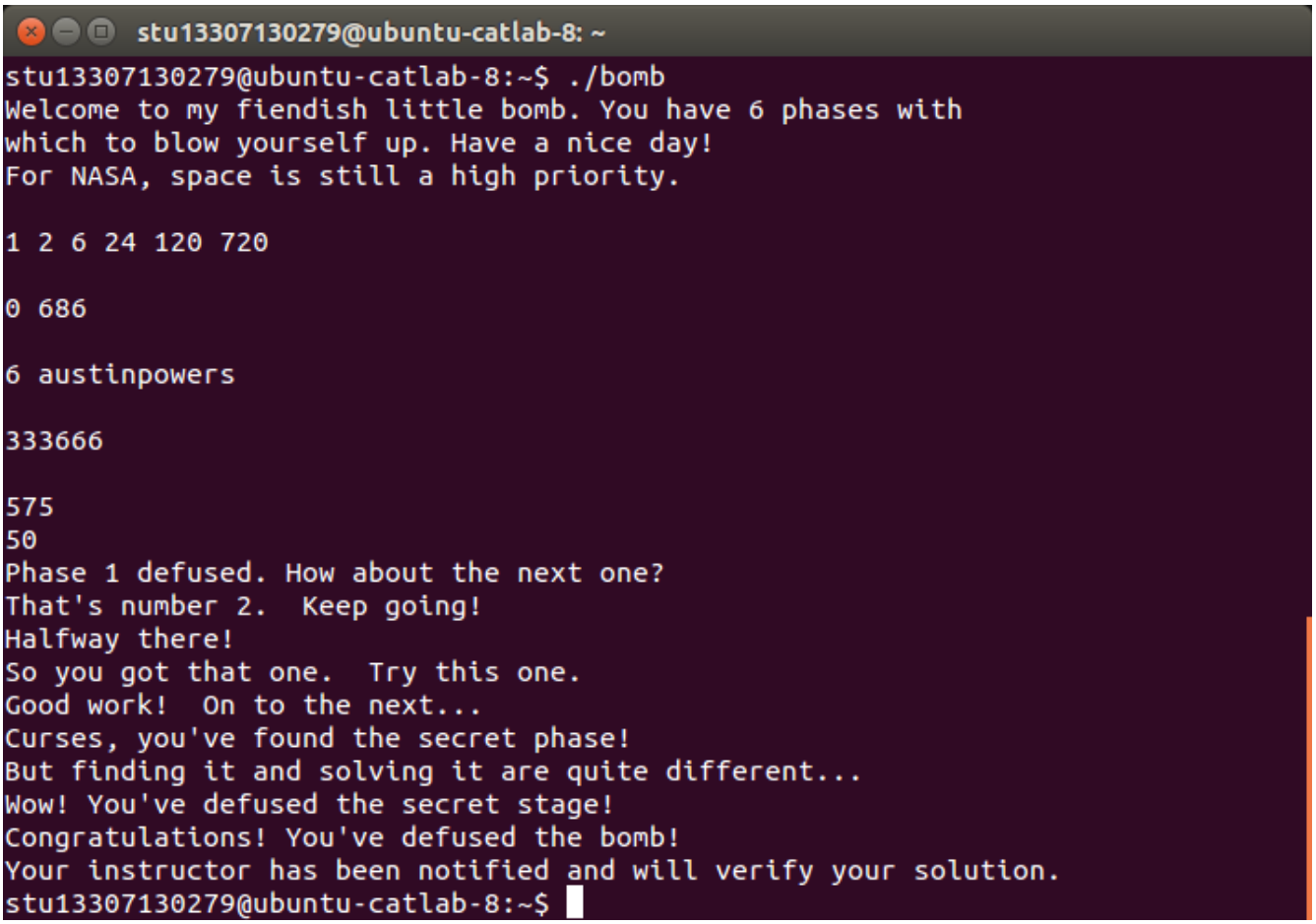
*a * 2 + 1 == 1 -> *a==0 得到*a < b

之后*a == 0，也就是*a == b。

```
(gdb) print *(int*)0x804c0c0
$1 = 36
(gdb) print *(int*)(0x804c0c0+8)
$2 = 134529240
(gdb) print *(int*)134529240
$3 = 50
(gdb) run
```

于是我们调用gdb输出其中的值，得到此时的输入应该为50。至此，七个关卡全部通过。

四、实验结论



直接用./bomb运行炸弹，把七个关卡得到的值输入，发现能够成功解除炸弹。最终英雄还是从邪恶博士(Dr. Evil)的手里拯救了整个城市。

五、实验体会

通过这次实验，我对于Linux系统的一些操作命令有了一些了解和掌握，学习了如何使用gdb这个强大的工具进行调试，以及加深了对于汇编语言的熟悉。在本次实验中，因为开始不知道可以在炸弹爆炸的函数之前设置断点，导致在第六关炸弹爆炸一次。我回头查看程序，发现此时的情况是设置的两个断点都因为跳转的原因没有运行到，才无法阻挡炸弹程序的运行。这一次的疏忽也让我明白了，不要存在侥幸心理，每一步都应该十分谨慎。

另外这次的报告也会在提交截止时间(4月19日中午12:00)之后在我的github上公开。地址是：https://github.com/lirenjie95/CSAPP