湖南大學
HUNAN UNIVERSITY

课程实验报告

课　程　名　称：　计算机组成与结构

实验项目名称：　bomb lab

专　业　班　级：　物联 1402

姓　　名：

学　　号：　2014080802--

指　导　教　师：　杨科华

完　成　时　间　：　2016　年　4　月　23　日

信息科学与工程学院

实验题目：bomb lab

实验目的：

程序运行在 linux 环境中。程序运行中有 6 个关卡（6 个 phase），每个 phase 需要用户在终端上输入特定的字符或者数字才能通关，否则会引爆炸弹！那么如何才能知道输入什么内容呢？这需要你使用 gdb 工具反汇编出汇编代码，结合 c 语言文件找到每个关卡的入口函数。然后分析汇编代码，找到在每个 phase 程序段中，引导程序跳转到“explode_bomb”程序段的地方，并分析其成功跳转的条件，以此为突破口寻找应该在命令行输入何种字符通关。

实验环境：联想 Y40，Ubuntu14.04 LTS 系统

实验内容及操作步骤：

一：用 linux 终端作准备

1、 Ubuntu 装好了之后，在终端输入 ssh username@10.92.13.8 连接到服务器，并开始新一轮的尝试。

2、输入 objdump -d bomb > 1.txt 将汇编代码输出到服务器上一个自动生成的叫 1.txt 的文件中。

3、中断连接，退回自己的系统桌面，使用命令 scp username@10.92.8:1.txt 1.txt 将在桌面复制生成一个也叫 1.txt 的文件。这时候就可以很方便的查看汇编代码了。

二： 开始拆炸弹

Phase_1：

```
08048f61 <phase_1>:
 8048f61:  55                       push   %ebp
 8048f62:  89 e5                    mov    %esp,%ebp
 8048f64:  83 ec 18                 sub    $0x18,%esp 栈的开辟
 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> 爆炸点
 8048f83:  c9                       leave
 8048f84:  c3                       ret
 8048f85:  90                       nop
```

分析：

首先找到炸弹爆炸点,可以看到是当调用 strings_not_equal 函数,当用户输入的字符 串

与程序的字符串不相等则就会爆炸，看到调用字符串比较函数之前传入的两个参数，第一个参数是放在 0x8（%ebp）处，可以推断出应该是我们输入的字符串，第二个参数是在 $0x804a15c 即本关的通关密码，因此我们只需要输入和 0x804a15c 地址处相同的字符串，通过匹配字符串，我们便可以把炸弹解开。
用命令 x/s 0x804a15c 查看其中的数据，得到密码：

```
(gdb) x/s 0x804a15c
0x804a15c:      "We have to stand with our North Korean allies."
```

将该串输入进行验证，由下图可知正确。

```
(gdb) r
Starting program: /home/dnoahsark/jizu3/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?
```

Phase_2：

```
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   栈的开辟
 8048d72:  8d 45 e0                lea    -0x20(%ebp),%eax 相当于 c 语言中的&
 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>
                                   //从函数名可猜测我们需要输入 6 个数
 8048d84:  83 7d e0 00             cmpl   $0x0,-0x20(%ebp)输入的第一个数
 8048d88:  75 06                   jne    8048d90 <phase_2+0x26>等于 0，不爆炸
 8048d8a:  83 7d e4 01             cmpl   $0x1,-0x1c(%ebp)输入的第二个数
 8048d8e:  74 05                   je     8048d95 <phase_2+0x2b>等于 1，不爆炸
 8048d90:  e8 3c 03 00 00          call   80490d1 <explode_bomb>爆炸点
 8048d95:  8d 5d e8                lea    -0x18(%ebp),%ebx 输入的第三个数
 8048d98:  8d 75 f8                lea    -0x8(%ebp),%esi
                                   //esi 指向第六个数字再向后移一位的地址。
 8048d9b:  8b 43 fc                mov    -0x4(%ebx),%eax
                                   //ebx 向前第一位的数字赋给 eax
 8048d9e:  03 43 f8                add    -0x8(%ebx),%eax
                                   //eax 加上 ebx 向前第二位的数字
 8048da1:  39 03                   cmp    %eax,(%ebx)
 8048da3:  74 05                   je     8048daa <phase_2+0x40>相等跳过爆炸
 8048da5:  e8 27 03 00 00          call   80490d1 <explode_bomb> 爆炸点
```

```
 8048daa:  83 c3 04                add    $0x4,%ebx  ebx 地址向后移动一位
 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
```

分析：

可以看出，本程序需要我们输入 6 个数，这 6 个数分别相对于 ebp 的偏移 16 进制为：-20，-1c，-18，-14，-10，-c。容易知道第一个数为 0，第二个数为 1，esi 出现一般会是循环或者数组中的变址，在此处考虑是循环的结束条件。若要不爆炸，第三个数等于第二个数+第一个数。因为有循环，可知 an=a_{n-1}+a_{n-2}。所以密码为：0 1 1 2 3 5

将该数据输入进行验证，由下图可知正确。

```
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2.  Keep going!
```

Phase_3：给出部分代码进行解释

```
08048ea1 <phase_3>:
 8048ea1:  55                      push   %ebp
 8048ea2:  89 e5                   mov    %esp,%ebp
 8048ea4:  83 ec 28                sub    $0x28,%esp  栈的开辟
 8048ea7:  8d 45 f0                lea    -0x10(%ebp),%eax  输入的第二个数
 8048eaa:  89 44 24 0c             mov    %eax,0xc(%esp)
 8048eae:  8d 45 f4                lea    -0xc(%ebp),%eax 输入的第一个数
 8048eb1:  89 44 24 08             mov    %eax,0x8(%esp)
 8048eb5:  c7 44 24 04 3e a2 04    movl   $0x804a23e,0x4(%esp)
                                   //用 gdb 查看一下$0x804a23e，可知里边儿存的是"%d %d"
 8048ebc:  08
 8048ebd:  8b 45 08                mov    0x8(%ebp),%eax
 8048ec0:  89 04 24                mov    %eax,(%esp)
 8048ec3:  e8 78 f9 ff ff          call   8048840 <__isoc99_sscanf@plt>
 8048ec8:  83 f8 01                cmp    $0x1,%eax 输入数据小于 1 个则爆炸
 8048ecb:  7f 05                   jg     8048ed2 <phase_3+0x31>
 8048ecd:  e8 ff 01 00 00          call   80490d1 <explode_bomb>爆炸点
```

分析：

这个爆炸点是说如果 eax 寄存器中的值小于 1 就会引爆，而 eax 寄存器中的值是调用 sscanf 函数的返回值，可以猜测该返回值可能是输入的数的个数。在看 lea 加载有效地

址，猜测这里应该是输入两个数，分别从-0x10（%ebp）（输入的第二个数）和-0xc（%ebp）（输入的第一个数）加载进来并进行保存。

```
8048ed2:  83 7d f4 07              cmpl   $0x7,-0xc(%ebp)第一个数大于 7 爆炸
8048ed6:  77 6b                    ja     8048f43 <phase_3+0xa2>
8048ed8:  8b 45 f4                 mov    -0xc(%ebp),%eax
8048edb:  ff 24 85 a0 a1 04 08     jmp    *0x804a1a0(,%eax,4)
                  //跳转至 0x804a1a0+eax*4(第一个数)内数据所指的地方   关键点
```

分析：

这里有一个非常关键的 jmp 语句，可以看出来这个是典型的 switch case 语句的汇编语句，对应的是一个跳转表，如果不熟悉也可以一步步的分析。现在假设你不知道。这里可以看出来跳转的地址跟你输入的第一个数有关，具体跳转到哪里，是要看 0x804a1a0 里面存的内容，用 gdb 调试一下，x/20xw 0x804a1a0 直接查看到每种 case 的跳转地址。

```
(gdb) x/20xw 0x804a1a0
0x804a1a0:      0x08048f12      0x08048f19      0x08048f09      0x08048f02
0x804a1b0:      0x08048ef9      0x08048ef2      0x08048ee9      0x08048ee2
```

```
8048f12:  b8 14 03 00 00           mov    $0x314,%eax
                                   //当第一个数为 0 时跳转到此处
8048f17:  eb 05                    jmp    8048f1e <phase_3+0x7d>
8048f19:  b8 00 00 00 00           mov    $0x0,%eax
8048f1e:  2d 5a 03 00 00           sub    $0x35a,%eax  x=x-858
8048f23:  05 ef 02 00 00           add    $0x2ef,%eax  x=x+751
8048f28:  2d 16 02 00 00           sub    $0x216,%eax  x=x-534
8048f2d:  05 16 02 00 00           add    $0x216,%eax  后边 4 行不用管
8048f32:  2d 16 02 00 00           sub    $0x216,%eax
8048f37:  05 16 02 00 00           add    $0x216,%eax
8048f3c:  2d 16 02 00 00           sub    $0x216,%eax
8048f41:  eb 0a                    jmp    8048f4d <phase_3+0xac>
8048f43:  e8 89 01 00 00           call   80490d1 <explode_bomb>爆炸点
8048f48:  b8 00 00 00 00           mov    $0x0,%eax
8048f4d:  83 7d f4 05              cmpl   $0x5,-0xc(%ebp)第一个数大于 5 爆炸
8048f51:  7f 05                    jg     8048f58 <phase_3+0xb7>
8048f53:  3b 45 f0                 cmp    -0x10(%ebp),%eax 第二个数
8048f56:  74 05                    je     8048f5d <phase_3+0xbc>
8048f58:  e8 74 01 00 00           call   80490d1 <explode_bomb>
8048f5d:  c9                       leave
8048f5e:  66 90                    xchg   %ax,%ax
8048f60:  c3                       ret
```

分析：

可以看出，先看出输入第一个数据小于等于 7.后边可知第一个数小于等于 5。然后 cmp 那句看出输入的第二个数要和每次根据第一个输入的 case 得到的 eax 的结果值相等。可

以输入 0-5 六个数，每次输入不同的数然后就可以直接查看它的 case 运算后的结果的 eax 寄存器的值得到第二个数。

最后答案：

0 147　　1 -641　　2 217　　3 -534　　4 0　　5 -534

以 0 147 为例，最后结果如图：

```
0 147
Halfway there!
```

Phase_4：给出部分代码进行解释

```
08048e2e <phase_4>:
 8048e2e:  55                       push   %ebp
 8048e2f:  89 e5                    mov    %esp,%ebp
 8048e31:  83 ec 28                 sub    $0x28,%esp 栈的开辟
 8048e34:  8d 45 f0                 lea    -0x10(%ebp),%eax
 8048e37:  89 44 24 0c              mov    %eax,0xc(%esp)第二个数字
 8048e3b:  8d 45 f4                 lea    -0xc(%ebp),%eax
 8048e3e:  89 44 24 08              mov    %eax,0x8(%esp)第一个数字
 8048e42:  c7 44 24 04 3e a2 04     movl   $0x804a23e,0x4(%esp)
                                    //$0x804a23e 存的是%d %d，即输入两个整数
 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
                                  //以上两行要求之前输入的时两个数据，否则爆炸
 8048e58:  75 0c                    jne    8048e66 <phase_4+0x38>
 8048e5a:  8b 45 f4                 mov    -0xc(%ebp),%eax
 8048e5d:  85 c0                    test   %eax,%eax
                                    //第一个数据大于等于 0，否则爆炸
 8048e5f:  78 05                    js     8048e66 <phase_4+0x38>
 8048e61:  83 f8 0e                 cmp    $0xe,%eax
 8048e64:  7e 05                    jle    8048e6b <phase_4+0x3d>
                                    //第一个数据小于等于 14，否则爆炸。
 8048e66:  e8 66 02 00 00           call   80490d1 <explode_bomb>爆炸点
 从以上代码可知第一个数为 0-14
 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>
                                   //调用递归函数 func4    func4（x，0,14）
 8048e86:  83 f8 01                cmp    $0x1,%eax
 8048e89:  75 06                   jne    8048e91 <phase_4+0x63>
 8048e8b:  83 7d f0 01             cmpl   $0x1,-0x10(%ebp)
                                   //第二个数据等于 1，否则爆炸
 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

08048b60 <func4>:
 对参数的存放：
 8048b6c:  8b 55 08                mov    0x8(%ebp),%edx  输入设为 x
 8048b6f:  8b 45 0c                mov    0xc(%ebp),%eax  0    设为 y
 8048b72:  8b 5d 10                mov    0x10(%ebp),%ebx 14   设为 z
 进行的一些处理：
 8048b75:  89 d9                   mov    %ebx,%ecx   a=z
 8048b77:  29 c1                   sub    %eax,%ecx   a=a-y
 8048b79:  89 ce                   mov    %ecx,%esi   b=a
 8048b7b:  c1 ee 1f                shr    $0x1f,%esi 逻辑右移 b=b>>31
 8048b7e:  8d 0c 0e                lea    (%esi,%ecx,1),%ecxa=b+a
 8048b81:  d1 f9                   sar    %ecx        算数右移 a=a/2
 8048b83:  01 c1                   add    %eax,%ecx   a=a+y

 8048b85:  39 d1                   cmp    %edx,%ecx
 8048b87:  7e 17                   jle    8048ba0 <func4+0x40>
```

分析：

设 ecx 寄存器中存储的是 a，esi 寄存器存储的是 b，按照汇编语句顺序对应操作 a=z，a=a-y，b=a，b=b>>31，a=b+a；a=a/2；a=a+y；即：a=（z-y）/2+y

将 a 与 x 相比较，有三个结果，将其代码转换成 c 语言

```
        int func4(int x,int y,int z)
        {
        int a;
        a=(z-y)/2+y;
        if(a<=x)
```

```
    {
        y=0;
        if(a>=x)
            return y; //a==x 时返回 0;
        else
        {
            a=a+1;
            y=func4(x,a,z);
            y=y*2;
            return y+1;
         }
        else
        {
            a=a-1;
            y=func4(x,y,z);
            y=y*2;
            return y;
         }
    }
```

分析：

第一次传参调用 func4（x，0,14），首先 a=（z-y）/2+y=7，因为 a<x,则 x 的取值范围要是 8-14.已知第二个数为 1，可以一个一个地尝试。
可以看出,第二个数据确定为 1。第一个数据范围为 0=<x=<14,并且由一个递归函数 func4 又一次缩小了范围。
我认为，因为函数太复杂，而且第一个数据范围又不大，可以直接将第一个数据从 0 尝试到 14，得到了三个数字 8、9、11 符合要求。故密码有三组：8 1；9 1；11 1。

Phase_5：给出部分代码进行解释

```
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    栈的开辟
 8048dc0:  8d 45 f0                    lea    -0x10(%ebp),%eax 第二个数字
 8048dc3:  89 44 24 0c                 mov    %eax,0xc(%esp)
 8048dc7:  8d 45 f4                    lea    -0xc(%ebp),%eax  第一个数字
 8048dca:  89 44 24 08                 mov    %eax,0x8(%esp)
 8048dce:  c7 44 24 04 3e a2 04        movl   $0x804a23e,0x4(%esp)
                                       //$0x804a23e 存的是%d %d，即输入两个整数
 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>
                                  //调用 sscanf 后返回值表示输入的数的个数要大于 1
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>爆炸点
// 看到有 ebx 和 esi 考虑下面的情况中有循环或者数组。

8048deb:  8b 45 f4                mov    -0xc(%ebp),%eax
8048dee:  83 e0 0f                and    $0xf,%eax 取第一个数的后四位
8048df1:  89 45 f4                mov    %eax,-0xc(%ebp)
8048df4:  83 f8 0f                cmp    $0xf,%eax 第一个数的后四位不为 1111
8048df7:  74 29                   je     8048e22 <phase_5+0x6a>跳到爆炸点
```

分析：

这段代码将第一个输入的参数与 0xf 进行位与运算保留输入参数的后四位并把结果放在 eax 中，若此时 eax 的值为 0xf 则会发生爆炸，于是可以知道第一个参数的后 4 位不能为 1111。

```
8048df9:  b9 00 00 00 00          mov    $0x0,%ecx 保存数组元素的和
8048dfe:  ba 00 00 00 00          mov    $0x0,%edx 用来循环计数
8048e03:  bb c0 a1 04 08          mov    $0x804a1c0,%ebx 数组的首地址
8048e08:  83 c2 01                add    $0x1,%edx
8048e0b:  8b 04 83                mov    (%ebx,%eax,4),%eax
8048e0e:  01 c1                   add    %eax,%ecx
8048e10:  83 f8 0f                cmp    $0xf,%eax
8048e13:  75 f3                   jne    8048e08 <phase_5+0x50>

8048e18:  83 fa 0f                cmp    $0xf,%edx 循环 15 次
8048e1b:  75 05                   jne    8048e22 <phase_5+0x6a>
8048e1d:  39 4d f0                cmp    %ecx,-0x10(%ebp)第二个参数值
8048e20:  74 05                   je     8048e27 <phase_5+0x6f>
8048e22:  e8 aa 02 00 00          call   80490d1 <explode_bomb>爆炸点
```

分析：

这里有一个循环，看到 8048e0b 中很熟悉的取数组元素的操作。但是取值的方式有点不同，我们输入的第一个数后四位 eax 用来被计算偏移下标了。也就是说下标不是连续的。每次取出来的元素的值用到下一次去数组元素时计算下标。可以看到 ecx 寄存器是一个累加器，先赋值为零，然后用来保存数组元素的和，然后 edx 用来循环计数，在这里，只要取到的数组中元素不为 15 就继续循环累加求和，可以看出数组中有一个元素是 15，然后 ebx 作为的是数组的首地址，所以可知已知数组存在这个地址中。用 gdb 调试查看一下 0x804a1c0 中的信息：

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

第二个数等于数组元素的累加和。

分析：

序号 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
数组 10 2 14 7 8 12 15 11 0 4 1 13 3 9 6 5
因为最后一个元素要为 15，倒推回去可知 n 依次为：5 12 3 7 11 13 9 4 8 0 10 1 2 14 6 15
S=115。所以第二个数为 115，然而第一个数最后四位为 0101 即可，所以可取很多值，这里就取 5 为例。则密码为：5 115。验证，由下图可知正确。

```
5 115
Good work!  On to the next...
```

Phase_6：给出部分代码进行解释

```
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  栈的开辟
 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>
```

分析：

看到有 edi，esi，ebx 就会联想到循环和数组，另外看到这里是调用了<read_six_numbers>，应该需要输入六个数字。

```
 8048cac:  8b 04 b7                 mov    (%edi,%esi,4),%eax
                                        //第 esi+1 个数给 eax。开始外层 for 循环
 8048caf:  83 e8 01                 sub    $0x1,%eax  eax--
 8048cb2:  83 f8 05                 cmp    $0x5,%eax     第 esi+1 个数小于等于 6
 8048cb5:  76 05                    jbe    8048cbc <phase_6+0x33>
 8048cb7:  e8 15 04 00 00           call   80490d1 <explode_bomb>爆炸点
```

分析：

edi 存的是数组的首地址，esi 作为数组坐标的偏移计算。首先数组的第一个元素减 1 要小于 5，也就是第一个数要小于 6.

```
 8048cbc:  83 c6 01                 add    $0x1,%esi   esi++
 8048cbf:  83 fe 06                 cmp    $0x6,%esi
 8048cc2:  74 22                    je     8048ce6 <phase_6+0x5d>
                                    //等于 6 时跳转至 0x8048ce6
 8048cc4:  8d 1c b7                 lea    (%edi,%esi,4),%ebx
                                    //第 esi+1 个数给 ebx
 8048cc7:  89 75 b4                 mov    %esi,-0x4c(%ebp)
```

```
8048cca:  8b 44 b7 fc          mov    -0x4(%edi,%esi,4),%eax
                               //第 esi 个数给 eax。开始内层 for 循环
8048cce:  3b 03                cmp    (%ebx),%eax
                               //若 eax 和 ebx 不相等则跳过炸弹
8048cd0:  75 05                jne    8048cd7 <phase_6+0x4e>
8048cd2:  e8 fa 03 00 00       call   80490d1 <explode_bomb>爆炸点
```

分析：

将 esi 的值加 1，我们令 esi 中的值为 i，那么 i<6,就在循环中，这个时候，把数组的第 esi+1 个数取出来放在 ebx 寄存器中，它将这个时候的 i 在栈中存储起来，相当于将 i 赋值给 k，然后又将第二个数的前一个数也就是第一个数放在 eax 中，将第二个数与第一个数比较，结果要不相等。

```
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>
```

分析：

将在栈中临时保存的 i 的值 k 加 1，然后 ebx 加 4 相当于取第三个数，只要 k 比 5 小，就跳进与 eax 寄存器的比较，也就是说通过 k 来实现第一个数与剩下五个数的比较，要第一个数跟其余五个数都不相等。当 k=5 时,即第一个数和其余 5 个数比较完后跳到 8048cac

```
8048ce6:  bb 00 00 00 00       mov    $0x0,%ebx
8048ceb:  8d 7d d0             lea    -0x30(%ebp),%edi
8048cee:  eb 16                jmp    8048d06 <phase_6+0x7d>

8048d06:  89 de                mov    %ebx,%esi
8048d08:  8b 0c 9f             mov    (%edi,%ebx,4),%ecx
8048d0b:  ba c4 c0 04 08       mov    $0x804c0c4,%edx
                               //0x804c0c4 查看可知是一个链表
8048d10:  b8 01 00 00 00       mov    $0x1,%eax
8048d15:  83 f9 01             cmp    $0x1,%ecx
8048d18:  7f d6                jg     8048cf0 <phase_6+0x67>
                               //数组元素大于 1
8048d1a:  eb de                jmp    8048cfa <phase_6+0x71>
```

分析：

//数组元素小于等于 1

将数组的首地址重新赋给 edi 寄存器。然后再次取数组中的元素，比较 1 的大小，然后进行相应的跳转。

```
8048cf0:  8b 52 08             mov    0x8(%edx),%edx
8048cf3:  83 c0 01             add    $0x1,%eax
8048cf6:  39 c8                cmp    %ecx,%eax
8048cf8:  75 f6                jne    8048cf0 <phase_6+0x67>
```

分析：

当数组元素比 1 大时，就将 edx 偏移 0x8，再次将 eax 加 1，将数组元素在跟 eax 相比较，只要不相等，就再次将 edx 偏移 0x8 直到数组元素和 eax 中的值相等。

```
8048cfa:  89 54 b5 b8             mov    %edx,-0x48(%ebp,%esi,4)
8048cfe:  83 c3 01                add    $0x1,%ebx
8048d01:  83 fb 06                cmp    $0x6,%ebx//当 6 个数执行完后
8048d04:  74 16                   je     8048d1c <phase_6+0x93>

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)对链表的值进行检测
8048d3d:  c7 40 08 00 00 00 00    movl   $0x0,0x8(%eax)
8048d44:  be 00 00 00 00          mov    $0x0,%esi
                                  //进行了一系列的操作，重新排列链表的值
8048d49:  8b 43 08                mov    0x8(%ebx),%eax
8048d4c:  8b 13                   mov    (%ebx),%edx
8048d4e:  3b 10                   cmp    (%eax),%edx
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
```

分析：

可以看到 esi 又是一个计数的，然后就是把新的映射中的内容做一个循环比较，通过比较的条件可以看出，新的序列中的内容要前面的大于等于后面的，也就是一个有序序列，也就是一个递增序列。而这个新序列经过我们分析可知是根据我们输入的六个数，按照我们输入的数的不同的顺序取出来的，最后取出来的顺序必须是增序。

分析：

综上可以看出，我们要输入 6 个不相同的数，而且都要小于等于 6，然后输入的顺序影响到后续得到的要是一个递增序列。

用 gdb 查看 0x804c04 得到一个<node1> 423。继续查看 node 可得到 6 个数据。

链表：

```
(gdb) x/d 0x804c0c4
0x804c0c4 <node1>:      423
(gdb) p node1
$1 = 423
(gdb) p node2
$2 = 108
(gdb) p node3
$3 = 341
(gdb) p node4
$4 = 391
(gdb) p node5
$5 = 957
(gdb) p node6
$6 = 597
(gdb) p node7
No symbol "node7" in current context.
```

收从上面可以看到内容按从大到小排列是：

957>597>423>391>341>108

也就是节点号按照这个顺序就是：

5 6 1 4 3 2

将密码输入验证，由下图可知正确：

```
5 6 1 4 3 2
Congratulations! You've defused the bomb!
```

至此，6 关已经全部通过！

三、验证

```
(gdb) r
Starting program: /home/dnoahsark/jizu3/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!
8 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 2695) exited normally]
(gdb) _
```

实验感想：

1.本次实验前 3 关比较容易，后 3 关就太烧脑了，看到那么多汇编语句真是头都要大了，其中我觉得第六关是最复杂的。通过这次实验，让我对汇编代码和堆栈的概念的理解更深了一步。

2.通过本实验，我熟悉了 gdb 的基本调试方法，提高了实践能力；掌握了一些汇编程序程序基本的书写格式并了解了某些寄存器的一般用途，对汇编程序有了更深入的了解。

实验成绩	