



原 深入理解计算机系统--bomblab

2018年05月03日 19:50:44 AC-NEWBIE 阅读数: 3733

版权声明：本文为博主原创文章，未经博主允许不得转载。 <https://blog.csdn.net/xbb224007/article/details/80155833>

前言：

这个是课程《深入理解计算机系统》中的一个实验，名称为bomblab，也就是炸弹实验。开始听这个名字就觉得挺有趣的有木有？那么这个实验要干嘛呢？老师稀里哗啦介绍了一番，也没有认真听。后来做了一下才了解了我们的任务。在此对这整个实验的操作过程进行回顾与记录。由于本文是我做完实验后写的，所以过程中可能有些许错误，如有发现，望各位朋友批评斧正。另外该实验过程中需要对函数的运行有一个比较直观的认识，需要的可以先看看[函数调用与控制流](#)。

实验任务：

首先，老师给了我们两个文件bomb.c和bomb.o，这个bomb.c文件中写了六个函数也就是对应着六个关卡，每个关卡都会提示你输入一些数据，当你输入符和函数的要求时，函数可以正常退出，如果不符不符合要求就bomb up!。那么任务很简单了，我们看代码还不容易吗？那我们就先来看看：

```
#include <stdio.h>
#include <stdlib.h>
#include "support.h"
#include "phases.h"

/*
 * Note to self: Remember to erase this file so my victims will have no
 * idea what is going on, and so they will all blow up in a
 * spectacularly fiendish explosion. -- Dr. Evil
 */

FILE *infile;

int main(int argc, char *argv[])
{
    char *input;

    /* Note to self: remember to port this bomb to Windows and put a
     * fantastic GUI on it. */

    /* When run with no arguments, the bomb reads its input lines
     * from standard input. */
    if (argc == 1) {
        infile = stdin;
    }

    /* When run with one argument <file>, the bomb reads from <file>
     * until EOF, and then switches to standard input. Thus, as you
     * defuse each phase, you can add its defusing string to <file> and
     * avoid having to retype it. */
    else if (argc == 2) {
        if (!(infile = fopen(argv[1], "r")))) {
            printf("%s: Error: Couldn't open %s\n", argv[0], argv[1]);
            exit(8);
        }
    }

    /* You can't call the bomb with more than 1 command line argument. */
    else {
        printf("Usage: %s [<input_file>]\n", argv[0]);
        exit(8);
    }

    /* Do all sorts of secret stuff that makes the bomb harder to defuse. */
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");
}
```



```

/* Hmm... Six phases must be more secure than the last one */
input = read_line(); /* Get input */
phase_1(input); /* Run the phase */
phase_defused(); /* Drat! They figured it out!
    * Let me know how they did it. */
printf("Phase 1 defused. How about the next one?\n");

/* The second phase is harder. No one will ever figure out
    * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2. Keep going!\n");

/* I guess this is too easy so far. Some more complex code will
    * confuse people. */
input = read_line();
phase_3(input);
phase_defused();
printf("Halfway there!\n");

/* Oh yeah? Well, how good is your math? Try on this saucy problem! */
input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one. Try this one.\n");

/* Round and 'round in memory we go, where we stop, the bomb blows! */
input = read_line();
phase_5(input);
phase_defused();
printf("Good work! On to the next...\n");

/* This phase will never be used, since no one will get past the
    * earlier ones. But just in case, make this one extra hard. */
input = read_line();
phase_6(input);
phase_defused();

/* Wow, they got it! But isn't something... missing? Perhaps
    * something they overlooked? Mua ha ha ha! */

return 0;
}

```

emmm, 貌似和我们想要的代码不一样, 这就只有函数名啊..., 不过想想也是, 直接给代码不就没啥意思了吗。不过我们还有一个可执行文件bomb.o, 一个可执行文件可以反汇编出这个程序的汇编代码(代码太长, 就不展示了), 然后通过看各个函数的反汇编代码来找到对应的提示、输入对应的数据, 然后过关, 我们这个实验需要完成的任务。

实验内容:

phase_1:

(一) 我们先得到反汇编代码:

```

Dump of assembler code for function phase_1:
0x08048f61 <+0>:    push   %ebp
0x08048f62 <+1>:    mov    %esp,%ebp
0x08048f64 <+3>:    sub    $0x18,%esp
0x08048f67 <+6>:    movl   $0x804a15c,0x4(%esp)
0x08048f6f <+14>:   mov    0x8(%ebp),%eax
0x08048f72 <+17>:   mov    %eax,(%esp)
0x08048f75 <+20>:   call   0x8048fab <strings_not_equal>
0x08048f7a <+25>:   test   %eax,%eax
0x08048f7c <+27>:   je    0x8048f83 <phase_1+34>
0x08048f7e <+29>:   call   0x80490d1 <explode_bomb>
0x08048f83 <+34>:   leave 
0x08048f84 <+35>:   ret    
End of assembler dump.

```

(二) 下面对其进行解释与说明:

1	0x08048f61 <+0>: push %ebp	// 前三句为栈帧的初始化
2	0x08048f62 <+1>: mov %esp,%ebp	
3	0x08048f64 <+3>: sub \$0x18,%esp	
4	0x08048f67 <+6>: movl \$0x804a15c,0x4(%esp)	// 将一个值放入0x4(%esp)
5	0x08048f6f <+14>: mov 0x8(%ebp),%eax	// 将0x8(%ebp)存的值存入%eax
6	0x08048f72 <+17>: mov %eax,(%esp)	// 将%eax的值存入(%esp)
7	0x08048f75 <+20>: call 0x8048fab <strings_not_equal>	// 调用函数, 由此可以知道前面两句赋值语句的作用为传参

```

8 | 0x08048f7a <+25>: test %eax,%eax           // 将函数的返回值做与运算
9 |
0x08048f7c <+27>: je    0x8048f83 <phase_1+34> // je判断相等或判断为0(相等跳转或结果为0跳转) 10 |
0x08048f7e <+29>: call   0x80490d1 <explode_bomb> // 若上述跳转语句没有执行则爆炸11 |
0x08048f83 <+34>: leave
0x08048f84 <+35>: ret

```

(三)再进行具体的分析:

首先我们看到调用的函数为<strings_not_equal>, 即比较两个字符串是否相等。可以想到该函数需要的两个参数为两个字符串的首地址。按照这个思路我们先查看一下地址0x804a15c:

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

发现的确是一个字符串的首地址。那么根据猜想, 另外一个参数自然是输入的字符串的首地址了, 我们同样地去验证一下。注意这里是把0x8(%ebp)存的值作为地址, 所以我们需要查看寄存器%ebp的值。这里设置一下断点然后查看即可, 如下:

```
(gdb) b phase_1
Breakpoint 1 at 0x8048f67
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Thank you!

Breakpoint 1, 0x08048f67 in phase_1 ()
```

我们设置断点以后运行, 然后随便输入一个字符串“Thank you!”。接下来我们查看寄存器%ebp的值:

```
(gdb) i r
eax            0x804c3e0      134530016
ecx            0x1          1
edx            0x0          0
ebx            0x0          0
esp            0xbffffef0    0xbffffef0
ebp            0xbffffef08    0xbffffef08
esi            0xbfffffd4    -1073745964
edi            0xb7fb8000    -1208254464
eip            0x8048f67     0x8048f67 <phase_1+6>
eflags          0x286      [ PF SF IF ]
cs             0x73        115
ss             0x7b        123
ds             0x7b        123
es             0x7b        123
fs             0x0          0
gs             0x33        51
```

寄存器%ebp的值为0xbffffef08, 那么0x8(%ebp)存的值(即地址单元0xbffffef10存的值)应该是我们输入字符串的首地址, 我们对对该地址的值addr进行查看, 并查看addr的内容如下:

```
(gdb) p/x *0xbffffef10
$1 = 0x804c3e0
(gdb) x/s 0x804c3e0
0x804c3e0 <input_strings>: "Thank you!"
```

我们发现addr=0x804c3e0, 并且该地址为我们输入字符串的首地址。所以我们上述的猜想正确。而函数<strings_not_equal>返回值为1时, 说明两个字符串不相等; 返回值为0时说明字符串相等。结合后面的跳转条件来看我们在这一关需要输入一个和给定字符串相等的字符串。所以phase_1的过关条件就是你输入的字符串和给定的字符串“We have to stand with our North Korean allies.”相等。上述我输入的“Thank you!”当然是无法过关的。如下图:

```
(gdb) c
Continuing.

BOOM!!!
The bomb has blown up.
```

我们再次运行, 这次输入给定的字符串:

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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.

Breakpoint 1, 0x08048f67 in phase_1 ()
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
```

可以看到phase_1过关成功，并给出了下一关的输入提示。

phase_2:

(一)同样地我们先得到反汇编代码：

```
(gdb) disass phase_2
Dump of assembler code for function phase_2:
0x08048d6a <+0>:    push   %ebp
0x08048d6b <+1>:    mov    %esp,%ebp
0x08048d6d <+3>:    push   %esi
0x08048d6e <+4>:    push   %ebx
0x08048d6f <+5>:    sub    $0x30,%esp
0x08048d72 <+8>:    lea    -0x20(%ebp),%eax
0x08048d75 <+11>:   mov    %eax,0x4(%esp)
0x08048d79 <+15>:   mov    0x8(%ebp),%eax
0x08048d7c <+18>:   mov    %eax,(%esp)
0x08048d7f <+21>:   call   0x804910b <read_six_numbers>
0x08048d84 <+26>:   cmpl   $0x0,-0x20(%ebp)
0x08048d88 <+30>:   jne    0x8048d90 <phase_2+38>
0x08048d8a <+32>:   cmpl   $0x1,-0x1c(%ebp)
0x08048d8e <+36>:   je     0x8048d95 <phase_2+43>
0x08048d90 <+38>:   call   0x80490d1 <explode_bomb>
0x08048d95 <+43>:   lea    -0x18(%ebp),%ebx
0x08048d98 <+46>:   lea    -0x8(%ebp),%esi
0x08048d9b <+49>:   mov    -0x4(%ebx),%eax
0x08048d9e <+52>:   add    -0x8(%ebx),%eax
0x08048da1 <+55>:   cmp    %eax,(%ebx)
0x08048da3 <+57>:   je     0x8048daa <phase_2+64>
0x08048da5 <+59>:   call   0x80490d1 <explode_bomb>
0x08048daa <+64>:   add    $0x4,%ebx
0x08048dad <+67>:   cmp    %esi,%ebx
0x08048daf <+69>:   jne    0x8048d9b <phase_2+49>
0x08048db1 <+71>:   add    $0x30,%esp
0x08048db4 <+74>:   pop    %ebx
0x08048db5 <+75>:   pop    %esi
0x08048db6 <+76>:   pop    %ebp
0x08048db7 <+77>:   ret

End of assembler dump.
```

(二)下面对其进行解释与说明：

1	0x08048d6a <+0>: push %ebp	// 保存旧的%ebp
2	0x08048d6b <+1>: mov %esp,%ebp	// 得到新栈帧的栈底
3	0x08048d6d <+3>: push %esi	// 保存旧的%esi与%ebx的值
4	0x08048d6e <+4>: push %ebx	
5	0x08048d6f <+5>: sub \$0x30,%esp	// 为新栈帧分配空间，这里分配了48个字节
6	0x08048d72 <+8>: lea -0x20(%ebp),%eax	// 将值-0x20(%ebp)(注意这里是lea指令，不是mov指令)存入寄存器%eax
7	0x08048d75 <+11>: mov %eax,0x4(%esp)	// 将%eax的值存入地址单元0x4(%esp)
8	0x08048d79 <+15>: mov 0x8(%ebp),%eax	// 将0x8(%ebp)存的值存入%eax
9	0x08048d7c <+18>: mov %eax,(%esp)	// 将%eax的值存入地址单元(%esp)
10	0x08048d7f <+21>: call 0x804910b <read_six_numbers>	// 调用函数<read_six_numbers>,所以上述的两个赋值为传参
11	0x08048d84 <+26>: cmpl \$0x0,-0x20(%ebp)	// 将地址-0x20(%ebp)存的值和0做比较
12	0x08048d88 <+30>: jne 0x8048d90 <phase_2+38>	// 不想等就跳转到+38的位置，即爆炸，故此时地址-0x20(%ebp)存的值应为0
13	0x08048d8a <+32>: cmpl \$0x1,-0x1c(%ebp)	// 将地址-0x1c(%ebp)存的值和1做比较
14	0x08048d8e <+36>: je 0x8048d95 <phase_2+43>	// 相等的话就跳过下一句(爆炸)到+43，故此时地址-0x1c(%ebp)存的值应为1
15	0x08048d90 <+38>: call 0x80490d1 <explode_bomb>	
16	0x08048d95 <+43>: lea -0x18(%ebp),%ebx	// 将值-0x18(%ebp)(lea指令)存入寄存器%ebx
17	0x08048d98 <+46>: lea -0x8(%ebp),%esi	// 将值-0x8(%ebp)(lea指令)存入寄存器%esi

```

18 | 0x08048d9b <+49>:    mov    -0x4(%ebx),%eax           // 将地址-0x4(%ebx)存的值存入寄存器%eax
19 |
| 0x08048d9e <+52>:    add    -0x8(%ebx),%eax           // 寄存器%eax的值加上地址-0x8(%ebx)存的值20
| 0x08048da1 <+55>:    cmp    %eax,(%ebx)               // 将%eax的值与地址(%ebx)存的值做比较, 不想等则爆炸21
| 0x08048da3 <+57>:    je     0x8048daa <phase_2+64>22 | 0x08048da5 <+59>:    call   0x80490d1 <explode_bomb>
23 | 0x08048da4 <+64>:    add    $0x4,%ebx             // 将寄存器%ebx的值+4
24 | 0x08048dad <+67>:    cmp    %esi,%ebx             // 将寄存器%ebx和寄存器%esi的值做比较, 不想等就跳转到+43的位置
25 | 0x08048daf <+69>:    jne    0x8048d9b <phase_2+49>
26 | 0x08048db1 <+71>:    add    $0x30,%esp            // phase_2结束, 接下来进行调用者栈帧的恢复操作
27 | 0x08048db4 <+74>:    pop    %ebx
28 | 0x08048db5 <+75>:    pop    %esi
29 | 0x08048db6 <+76>:    pop    %ebp
30 | 0x08048db7 <+77>:    ret

```

(三)接下来进行phase_2的具体分析:

我们先看函数<read_six_numbers>即读入6个数字。再看其上面的几个传参指令：一个是把地址-0x20(%ebp)传了进去；另一个是传入了地址0x8(%ebp)储存的值。按照第一关的思路，我们可以想到0x8(%ebp)储存的值为我们输入字符串的首地址。

这里对于输入做一个说明：我们通过看bomb.c文件可以知道在每一关执行之前，都要执行一个read_line()函数，也就是你的输入是在此时被读入的。而在具体的每一关里面的输入(比如这里的<read_six_numbers>)只是在你刚刚刚输入的一行数据里面进行操作，而你输入的数据(当作字符串处理)储存的首地址储存在地址0x8(%ebp)，后续几关的读入也是同样的原理。

既然0x8(%ebp)是用来进行<read_six_numbers>操作的，那么地址-0x20(%ebp)用来干嘛的呢？我们从输入的一行里面进行读入数据后，输入的数据需要储存起来的。更具后续对地址-0x20(%ebp)储存的值进行判断的操作，可以大概想到地址-0x20(%ebp)是用来保存读入的数据的。而我们需要保存的是六个数字，所以这个地址只是六个数据的第一个其中一个，其他的数据应该储存在该地址加上一个偏移量的地址中。当然这都是猜想，下面我们来进行验证：

同phase_1，我们先看0x8(%ebp)储存的是不是我们输入数据的首地址：

```

(gdb) b phase_2
Breakpoint 2 at 0x8048d6f
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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?
2 1 4 5 6 3

Breakpoint 2, 0x08048d6f in phase_2 ()
(gdb) i r
eax            0x804c430          134530096
ecx            0x2              2
edx            0x4              4
ebx            0x0              0
esp            0xbffffef00        0xbffffef00
ebp            0xbffffef08        0xbffffef08
esi            0xbffffefd4        -1073745964
edi            0xb7fb8000        -1208254464
eip            0x8048d6f         0x8048d6f <phase_2+5>
eflags          0x206          [ PF IF ]
cs              0x73            115
ss              0x7b            123
ds              0x7b            123
es              0x7b            123
fs              0x0              0
gs              0x33            51
(gdb) p/x *0xbffffef10
$2 = 0x804c430
(gdb) x/s 0x804c430
0x804c430 <input_strings+80>: "2 1 4 5 6 3"

```

至此说明读入一行数据以及储存的猜想没有错，由<read_six_numbers>读入的数据储存的位置是否也符合猜想呢，我们同样地设置断点、单步运行至函数<read_six_numbers>运行结束，然后去查看目标内存的值。如下：

```
Breakpoint 2, 0x08048d6f in phase_2 ()
(gdb) ni
0x08048d72 in phase_2 ()
(gdb)
0x08048d75 in phase_2 ()
(gdb)
0x08048d79 in phase_2 ()
(gdb)
0x08048d7c in phase_2 ()
(gdb)
0x08048d7f in phase_2 ()
(gdb)
0x08048d84 in phase_2 ()
(gdb)
0x08048d88 in phase_2 ()
(gdb) p/x *0xbffffee8@6
$11 = {0x2, 0x1, 0x4, 0x5, 0x6, 0x3}
(gdb) █
```

其中的 $0xbffffee8=0xbffffef08-0x20$, 即 $-0x20(%ebp)$ 的值, 我们可以发现从该地址开始, 更高地址连续几个地址的中值储存的值分别为2, 1, 4, 5, 6, 3。说明我们输入的数据的确被存到了这几个内存单元中, 并且第一关数据储存在地址 $-0x20(%ebp)$ 中, 第二个数据储存在 $-0x1c(%ebp)$ 中, 其余数据依次类推(每次+4是因为一个数据要四个字节储存)。

至此数据输入部分的说明结束。

接下来我们继续向下看:

首先是分别将地址 $-0x20(%ebp)$ 和地址 $-0x1c(%ebp)$ 的值分别和0与1比较, 不想等就爆炸。这里说明我们输入的前两个数应该为0和1。

接下来进入了一个循环, 我们先看循环的终止条件: `cmp %esi, %ebx`。当两个寄存器的值不相等时继续循环。再看他们的初值及变化: `%ebx`的初值为 $-0x18(%ebp)$, `%esi`的初值为 $-0x8(%ebp)$, 每次循环`%ebx`的值+4。这样一来的话该循环总共会进行4次。

对于每次循环的操作都是将当前 (`%ebx`) 的低两个地址储存的数据之和与地址 (`%ebx`) 储存的数据进行比较不相等则爆炸。

举个栗子:

对于第一次循环而言, `%ebx`储存的值为 $-0x18(%ebp)$, 低两个储存数据的地址分别为 $-0x1c(%ebp)$ 与 $-0x20(%ebp)$ 。

比较的时候将地址 $-0x1c(%ebp)$ 的值(你输入的第二个数1)与地址 $-0x20(%ebp)$ 的值(你输入的第一个数0)进行求和。

然后与地址 $-0x18(%ebx)$ 储存的值(你输入的第三个值)进行比较, 不相等则爆炸。相等话`ebx+4`(得到你输入的第四个数据储存的地址)。

那么下一次比较的时候, 进行就是你输入的第三个数与第二个数的和与你输入的第四个数的比较。

然后不断地循环, 直到`ebx==esi`结束循环, 实际上此时以恰好把第五个数与第四个数的和与第六个数比较完。

至此, 若以上比较均满足条件, 则phase_2过关成功。

总结一下通过密码就是: 你需要输入6个数, 第一个数为0, 第二个数为1, 其余的数就是其前面两个数之和。说白了就是要输入斐波拉契数列的前六项。如下:

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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

Breakpoint 2, 0x08048d6f in phase_2 ()
(gdb) c
Continuing.
That's number 2. Keep going!
```

至此第二关过关成功。

phase_3:

(一)首先查看其反汇编代码：

```
(gdb) disass phase_3
Dump of assembler code for function phase_3:
0x08048ea1 <+0>:    push   %ebp
0x08048ea2 <+1>:    mov    %esp,%ebp
0x08048ea4 <+3>:    sub    $0x28,%esp
0x08048ea7 <+6>:    lea    -0x10(%ebp),%eax
0x08048eaa <+9>:    mov    %eax,0xc(%esp)
0x08048eae <+13>:   lea    -0xc(%ebp),%eax
0x08048eb1 <+16>:   mov    %eax,0x8(%esp)
0x08048eb5 <+20>:   movl   $0x804a23e,0x4(%esp)
0x08048ebd <+28>:   mov    0x8(%ebp),%eax
0x08048ec0 <+31>:   mov    %eax,(%esp)
0x08048ec3 <+34>:   call   0x8048840 <__isoc99_sscanf@plt>
0x08048ec8 <+39>:   cmp    $0x1,%eax
0x08048ecb <+42>:   jg    0x8048ed2 <phase_3+49>
0x08048ecd <+44>:   call   0x80490d1 <explode_bomb>
0x08048ed2 <+49>:   cmpl   $0x7,-0xc(%ebp)
0x08048ed6 <+53>:   ja    0x8048f43 <phase_3+162>
0x08048ed8 <+55>:   mov    -0xc(%ebp),%eax
0x08048edb <+58>:   jmp    *0x804a1a0(,%eax,4)
0x08048ee2 <+65>:   mov    $0x0,%eax
0x08048ee7 <+70>:   jmp    0x8048f3c <phase_3+155>
0x08048ee9 <+72>:   mov    $0x0,%eax
0x08048eee <+77>:   xchg   %ax,%ax
0x08048ef0 <+79>:   jmp    0x8048f37 <phase_3+150>
0x08048ef2 <+81>:   mov    $0x0,%eax
0x08048ef7 <+86>:   jmp    0x8048f32 <phase_3+145>
0x08048ef9 <+88>:   mov    $0x0,%eax
0x08048efe <+93>:   xchg   %ax,%ax
0x08048f00 <+95>:   jmp    0x8048f2d <phase_3+140>
0x08048f02 <+97>:   mov    $0x0,%eax
0x08048f07 <+102>:  jmp    0x8048f28 <phase_3+135>
0x08048f09 <+104>:  mov    $0x0,%eax
0x08048f0e <+109>:  xchg   %ax,%ax
0x08048f10 <+111>:  jmp    0x8048f23 <phase_3+130>
0x08048f12 <+113>:  mov    $0x314,%eax
0x08048f17 <+118>:  jmp    0x8048f1e <phase_3+125>
0x08048f19 <+120>:  mov    $0x0,%eax
0x08048f1e <+125>:  sub    $0x35a,%eax
0x08048f23 <+130>:  add    $0x2ef,%eax
0x08048f28 <+135>:  sub    $0x216,%eax
0x08048f2d <+140>:  add    $0x216,%eax
0x08048f32 <+145>:  sub    $0x216,%eax
0x08048f37 <+150>:  add    $0x216,%eax
0x08048f3c <+155>:  sub    $0x216,%eax
0x08048f41 <+160>:  jmp    0x8048f4d <phase_3+172>
0x08048f43 <+162>:  call   0x80490d1 <explode_bomb>
0x08048f48 <+167>:  mov    $0x0,%eax
0x08048f4d <+172>:  cmpl   $0x5,-0xc(%ebp)
0x08048f51 <+176>:  jg    0x8048f58 <phase_3+183>
```

```

0x08048f53 <+178>:    cmp    -0x10(%ebp),%eax
0x08048f56 <+181>:    je     0x8048f5d <phase_3+188>
0x08048f58 <+183>:    call   0x80490d1 <explode_bomb>
0x08048f5d <+188>:    leave
0x08048f5e <+189>:    xchg   %ax,%ax
0x08048f60 <+191>:    ret
End of assembler dump.

```

(二)下面对其进行解释与说明:

```

1  0x08048ea1 <+0>:    push   %ebp
2  0x08048ea2 <+1>:    mov    %esp,%ebp
3  0x08048ea4 <+3>:    sub    $0x28,%esp           // 初始化新的栈帧
4  0x08048ea7 <+6>:    lea    -0x10(%ebp),%eax
5  0x08048eaa <+9>:    mov    %eax,0xc(%esp)      // 结合下面的函数调用的语句, 我们知道这里又是开始传参了
6  0x08048eae <+13>:   lea    -0xc(%ebp),%eax
7  0x08048eb1 <+16>:   mov    %eax,0x8(%esp)
8  0x08048eb5 <+20>:   movl   $0x804a23e,0x4(%esp) // 传入由四个参数, 其中的三个分别是地址: -0x10(%ebp), -0xc(%ebp)
9  0x08048ebd <+28>:   mov    0x8(%ebp),%eax
10 0x08048ec0 <+31>:   mov    %eax,(%esp)
11 0x08048ec3 <+34>:   call   0x8048840 <__isoc99_sscanf@plt> // 调用函数, 看sscanf可以大概猜出这里要进行读入操作
12 0x08048ec8 <+39>:   cmp    $0x1,%eax
13 0x08048ecb <+42>:   jg    0x8048ed2 <phase_3+49>
14 0x08048ecd <+44>:   call   0x80490d1 <explode_bomb>
15 0x08048ed2 <+49>:   cmpl   $0x7,-0xc(%ebp)      // 将地址-0xc(%ebp)储存的值与7做比较, 注意到-0xc(%ebp)是上述调用函数时
16                                         // 的一个参数, 类比phase_2, 我们可以知道-0xc(%ebp)储存的是我们输入的一个数
17 0x08048ed6 <+53>:   ja    0x8048f43 <phase_3+162> // 大于7则爆炸, 并且由于是无符号比较, 所以输入还要>=0
18 0x08048ed8 <+55>:   mov    -0xc(%ebp),%eax
19 0x08048edb <+58>:   jmp    *0x804a1a0(%eax,4) // 将-0xc(%ebp)的值存入%eax
20 0x08048ee2 <+65>:   mov    $0x0,%eax
21 0x08048ee7 <+70>:   jmp    0x8048f3c <phase_3+155> // 根据%eax的值以及0x804a1a0进行跳转, 具体的跳转过程后面解释
22 0x08048ee9 <+72>:   mov    $0x0,%eax
23 0x08048eee <+77>:   xchg   %ax,%ax
24 0x08048ef0 <+79>:   jmp    0x8048f37 <phase_3+150> // 接下来是一些的算数运算
25 0x08048ef2 <+81>:   mov    $0x0,%eax
26 0x08048ef7 <+86>:   jmp    0x8048f32 <phase_3+145> // 而上面的JMP跳转语句就是跳转到下面某一条语句, 然后开始执行
27 0x08048ef9 <+88>:   mov    $0x0,%eax
28 0x08048efe <+93>:   xchg   %ax,%ax
29 0x08048f00 <+95>:   jmp    0x8048f2d <phase_3+140>
30 0x08048f02 <+97>:   mov    $0x0,%eax
31 0x08048f07 <+102>:  jmp    0x8048f28 <phase_3+135>
32 0x08048f09 <+104>:  mov    $0x0,%eax
33 0x08048f0e <+109>:  xchg   %ax,%ax
34 0x08048f10 <+111>:  jmp    0x8048f23 <phase_3+130>
35 0x08048f12 <+113>:  mov    $0x314,%eax
36 0x08048f17 <+118>:  jmp    0x8048f1e <phase_3+125>
37 0x08048f19 <+120>:  mov    $0x0,%eax
38 0x08048f1e <+125>:  sub    $0x35a,%eax
39 0x08048f23 <+130>:  add    $0x2ef,%eax
40 0x08048f28 <+135>:  sub    $0x216,%eax
41 0x08048f2d <+140>:  add    $0x216,%eax
42 0x08048f32 <+145>:  sub    $0x216,%eax
43 0x08048f37 <+150>:  add    $0x216,%eax
44 0x08048f3c <+155>:  sub    $0x216,%eax
45 0x08048f41 <+160>:  jmp    0x8048f4d <phase_3+172> // 当运算结束以后再次进行跳转
46 0x08048f43 <+162>:  call   0x80490d1 <explode_bomb>
47 0x08048f48 <+167>:  mov    $0x0,%eax
48 0x08048f4d <+172>:  cmpl   $0x5,-0xc(%ebp)      // 将-0xc(%ebp)储存的值与5做比较
49 0x08048f51 <+176>:  jg    0x8048f58 <phase_3+183>
50 0x08048f53 <+178>:  cmp    -0x10(%ebp),%eax
51 0x08048f56 <+181>:  je     0x8048f5d <phase_3+188>
52 0x08048f58 <+183>:  call   0x80490d1 <explode_bomb>
53 0x08048f5d <+188>:  leave
54 0x08048f5e <+189>:  xchg   %ax,%ax
55 0x08048f60 <+191>:  ret

```

(三)接下来进行phase_3的具体分析:

首先是输入部分：我们先看那个不清楚意义的参数，查看其内容：

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

发现是“%d %d”，结合后面的输入函数以及其他参数，我们可以知道这一关需要我们输入两个整数，并且两个整数分别会被储存在地址-0x10(%ebp)和地址-0xc(%ebp)中。根据后面的一个比较我们可以知道输入的第一个数应该 ≤ 7 。这里怎么知道-0x10(%ebp)是第一个数？同前两关，设置断点，查看内存即可。

然后就是一个跟你输入的第一个数有关的跳转：jmp *0x804a1a0(%eax,4)。这里的jmp指令的格式说明进行的是间接跳转，这里分两步进行说明：

第一步：计算0x804a1a0+eax*4，这里的eax储存的是你输入的第一个数。设结果为ad1。

第二步：得到地址ad1储存的值ad2，然后跳转到地址ad2。

举个具体的栗子（假如我们输入的第一个数为0）：

第一步：得到ad1=0x804a1a0。

第二步：通过查看地址0x804a1a0储存的值，如下：

```
(gdb) p/x *0x804a1a0  
$12 = 0x8048f12
```

故跳转到地址0x8048f12，找到函数中对应的位置<phase_3+113>，接下来从这里开始执行指令。

再举个栗子（假如我们输入的第一数为2）：

第一步：得到ad1=0x804a1a8

第二步：通过查看地址0x804a1a0储存的值，如下：

```
(gdb) p/x *0x804a1a8  
$13 = 0x8048f09
```

故跳转到地址0x8048f09，对应函数中的位置<phase_3+104>，接下来从这里开始执行。

跳转完以后便是运算，这里没有太多的东西，就以上述的第一个栗子来说：

从<phase_3+113>开始：

1.%eax=0x314;

2.跳转到<phase_3+125>，%eax=%eax-0x35a

3.%eax=%eax+2ef;

4.%eax=%eax-216;

5.接下来的四次运算抵消。

6.最终的结果为：788-858+751-534=147

运算完以后是判断：

先是将你输入的第一个数（-0xc(%ebp)）与5做比较，大于5就爆炸，所以前面仅仅判断得出的第一个数 ≤ 7 还不够，第一个输入的数应该 ≤ 5 。

然后是将刚刚的运算结果与你输入的第二个数进行比较，不相等就爆炸。所以当你输入的第一个数是0时，你输入的第二个数就必须是147。

至此，梳理一下过第三关的思路：

1.输入两个数，第一个数必须大于等于0，且小于等于5。

2.根据第一个数的输入，计算出对应的运算后的结果，你输入的第二个数应该等于这个数，上述的(0,147)就是一个栗子。

最后面我们可以得到六组解，分别是：

(0,147) , (1,-641) , (2,217) , (3,-534) , (4,0) , (5,-534) 。

输入任意一组均可过关：

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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

Breakpoint 2, 0x08048d6f in phase_2 ()
(gdb) c
Continuing.
That's number 2. Keep going!
2 217
Halfway there!
```

至此，第三关已过，炸弹已拆除一半。

phase_4:

(一)首先查看其反汇编代码：

```
(gdb) disass phase_4
Dump of assembler code for function phase_4:
0x08048e2e <+0>:    push   %ebp
0x08048e2f <+1>:    mov    %esp,%ebp
0x08048e31 <+3>:    sub    $0x28,%esp
0x08048e34 <+6>:    lea    -0x10(%ebp),%eax
0x08048e37 <+9>:    mov    %eax,0xc(%esp)
0x08048e3b <+13>:   lea    -0xc(%ebp),%eax
0x08048e3e <+16>:   mov    %eax,0x8(%esp)
0x08048e42 <+20>:   movl   $0x804a23e,0x4(%esp)
0x08048e4a <+28>:   mov    0x8(%ebp),%eax
0x08048e4d <+31>:   mov    %eax,(%esp)
0x08048e50 <+34>:   call   0x8048840 <__isoc99_sscanf@plt>
0x08048e55 <+39>:   cmp    $0x2,%eax
0x08048e58 <+42>:   jne    0x8048e66 <phase_4+56>
0x08048e5a <+44>:   mov    -0xc(%ebp),%eax
0x08048e5d <+47>:   test   %eax,%eax
0x08048e5f <+49>:   js    0x8048e66 <phase_4+56>
0x08048e61 <+51>:   cmp    $0xe,%eax
0x08048e64 <+54>:   jle    0x8048e6b <phase_4+61>
0x08048e66 <+56>:   call   0x80490d1 <explode_bomb>
0x08048e6b <+61>:   movl   $0xe,0x8(%esp)
0x08048e73 <+69>:   movl   $0x0,0x4(%esp)
0x08048e7b <+77>:   mov    -0xc(%ebp),%eax
0x08048e7e <+80>:   mov    %eax,(%esp)
0x08048e81 <+83>:   call   0x8048b60 <func4>
0x08048e86 <+88>:   cmp    $0x1,%eax
0x08048e89 <+91>:   jne    0x8048e91 <phase_4+99>
0x08048e8b <+93>:   cmpl   $0x1,-0x10(%ebp)
0x08048e8f <+97>:   je    0x8048e9d <phase_4+111>
0x08048e91 <+99>:   lea    0x0(%esi,%eiz,1),%esi
0x08048e98 <+106>:  call   0x80490d1 <explode_bomb>
0x08048e9d <+111>:  leave 
0x08048e9e <+112>:  xchg   %ax,%ax
0x08048ea0 <+114>:  ret
```

End of assembler dump.

(二)下面对其进行解释与说明:

```
1 0x08048e2e <+0>: push %ebp //保存旧的%ebp
2 0x08048e2f <+1>: mov %esp,%ebp
3 0x08048e31 <+3>: sub $0x28,%esp //为新栈帧分配空间
4 0x08048e34 <+6>: lea -0x10(%ebp),%eax //下面几条语句又是在为函数传参
5 0x08048e37 <+9>: mov %eax,0xc(%esp) //首先是两个地址分别是-0x10(%ebp)、-0xc(%ebp)
6 0x08048e3b <+13>: lea -0xc(%ebp),%eax
7 0x08048e3e <+16>: mov %eax,0x8(%esp)
8 0x08048e42 <+20>: movl $0x804a23e,0x4(%esp) //这个我们在phase_3已经查看过了，说明这一关又要输入两个整数
9 0x08048e4a <+28>: mov 0x8(%ebp),%eax //这个就是我们输入字符串的首地址，下面的读入从这里面读数据
10 0x08048e4d <+31>: mov %eax,(%esp)
11 0x08048e50 <+34>: call 0x8048840 <__isoc99_sscanf@plt> //调用输入函数
12 0x08048e55 <+39>: cmp $0x2,%eax //读入的数不是两个话就爆炸
13 0x08048e58 <+42>: jne 0x8048e66 <phase_4+56>
14 0x08048e5a <+44>: mov -0xc(%ebp),%eax //将输入的第一个数存入寄存器%eax
15 0x08048e5d <+47>: test %eax,%eax
16 0x08048e5f <+49>: js 0x8048e66 <phase_4+56> //如果输入的第一个数是负数就爆炸
17 0x08048e61 <+51>: cmp $0xe,%eax //如果输入的第一数大于14的话就爆炸
18 0x08048e64 <+54>: jle 0x8048e6b <phase_4+61>
19 0x08048e66 <+56>: call 0x80490d1 <explode_bomb>
20 0x08048e6b <+61>: movl $0xe,0x8(%esp) //注意到下面又要调用函数<func4>，说明这里在传参
21 0x08048e73 <+69>: movl $0x0,0x4(%esp) //总该传了三个参数，一个是0，一个是14
22 0x08048e7b <+77>: mov -0xc(%ebp),%eax //另外一个就是你输入的那个数
23 0x08048e7e <+80>: mov %eax,(%esp)
24 0x08048e81 <+83>: call 0x8048b60 <func4> //调用函数<func4>
25 0x08048e86 <+88>: cmp $0x1,%eax //将<func4>的返回值与1做比较
26 0x08048e89 <+91>: jne 0x8048e91 <phase_4+99> //不相等就爆炸
27 0x08048e8b <+93>: cmpl $0x1,-0x10(%ebp) //将你输入的第二个数与1做比较
28 0x08048e8f <+97>: je 0x8048e9d <phase_4+111> //如果不相等则爆炸，说明你输入的第二个数必须为1
29 0x08048e91 <+99>: lea 0x0(%esi,%eiz,1),%esi
30 0x08048e98 <+106>: call 0x80490d1 <explode_bomb>
31 0x08048e9d <+111>: leave //phase_4运行结束，为返回调用者做准备
32 0x08048e9e <+112>: xchg %ax,%ax
33 0x08048ea0 <+114>: ret
```

(三)接下来进行phase_4的具体分析:

这次代码比较短，条理也比较清晰，还是比较容易读懂的。基本的运行流程如下：

首先是输入，输入和phase_3的输入一模一样，都是输入两个数，分别储存在地址-0x10(%ebp)和地址-0xc(%ebp)中。

然后根据两次判断我们知道第一个数的取值范围为[0, 14]。

接着将你输入的第一个数，以及0和14作为函数<func4>的参数，再调用函数<func4>。

根据函数调用以后的一次判断，我们知道我们输入的第一个数必须使<func4>的返回值为1，否则爆炸。

最后面就是告诉你，你输入的第二个数必须等于1。

主要的问题是输入的第一个数怎么确定，基本想法有两个：

第一个是我们已经知道了第一关数的取值范围是[0, 14]，那么我们最多尝试15次则一定会得到可行的结果。

第二个是查看函数<func4>，弄清他的运行过程，然后输入可行的解即可。

我这里尝试进行解读函数<func4>：

首先查看其反汇编代码：

```
(gdb) disass func4
Dump of assembler code for function func4:
0x08048b60 <+0>:    push   %ebp
0x08048b61 <+1>:    mov    %esp,%ebp
0x08048b63 <+3>:    sub    $0x18,%esp
0x08048b66 <+6>:    mov    %ebx,-0x8(%ebp)
0x08048b69 <+9>:    mov    %esi,-0x4(%ebp)
0x08048b6c <+12>:   mov    0x8(%ebp),%edx
0x08048b6f <+15>:   mov    0xc(%ebp),%eax
0x08048b72 <+18>:   mov    0x10(%ebp),%ebx
0x08048b75 <+21>:   mov    %ebx,%ecx
0x08048b77 <+23>:   sub    %eax,%ecx
0x08048b79 <+25>:   mov    %ecx,%esi
0x08048b7b <+27>:   shr    $0x1f,%esi
0x08048b7e <+30>:   lea    (%esi,%ecx,1),%ecx
0x08048b81 <+33>:   sar    %ecx
0x08048b83 <+35>:   add    %eax,%ecx
0x08048b85 <+37>:   cmp    %edx,%ecx
0x08048b87 <+39>:   jle    0x8048ba0 <func4+64>
0x08048b89 <+41>:   sub    $0x1,%ecx
0x08048b8c <+44>:   mov    %ecx,0x8(%esp)
0x08048b90 <+48>:   mov    %eax,0x4(%esp)
0x08048b94 <+52>:   mov    %edx,(%esp)
0x08048b97 <+55>:   call   0x8048b60 <func4>
0x08048b9c <+60>:   add    %eax,%eax
0x08048b9e <+62>:   jmp    0x8048bc0 <func4+96>
0x08048ba0 <+64>:   mov    $0x0,%eax
0x08048ba5 <+69>:   cmp    %edx,%ecx
0x08048ba7 <+71>:   jge    0x8048bc0 <func4+96>
0x08048ba9 <+73>:   mov    %ebx,0x8(%esp)
0x08048bad <+77>:   add    $0x1,%ecx
0x08048bb0 <+80>:   mov    %ecx,0x4(%esp)
0x08048bb4 <+84>:   mov    %edx,(%esp)
0x08048bb7 <+87>:   call   0x8048b60 <func4>
0x08048bbc <+92>:   lea    0x1(%eax,%eax,1),%eax
0x08048bc0 <+96>:   mov    -0x8(%ebp),%ebx
0x08048bc3 <+99>:   mov    -0x4(%ebp),%esi
0x08048bc6 <+102>:  mov    %ebp,%esp
0x08048bc8 <+104>:  pop    %ebp
0x08048bc9 <+105>:  ret
End of assembler dump.
```

没想到比phase_4还要长，并且可以看到它调用了自身，说明是一个递归函数，其具体的执行细节这里不予以说明，下面直接给出通过读汇编代码得到的C语言代码(主函数的五次输入是用来测试的)：

```

1 1 #include<stdio.h>
2 2 int func4(int x,int y,int z){
3 3     int mid1=x-y;
4 4     int mid2=(mid1>>31);
5 5 // printf("x=%d y=%d z=%d mid2=%d\n",x,y,z,mid2);
6 6     mid1+=mid2;
7 7     mid1/=2;
8 8     mid1+=y;
9 9     if(mid1<=z){
10 10         y=0;
11 11         if(mid1>=z){
12 12             return 0;
13 13         }
14 14         return 2*func4(x,mid1+1,z)+1;
15 15
16 16     }
17 17     return 2*func4(mid1-1,y,z);
18 18 }
19 19 int main(){
20 20     int in;
21 21     for(int i=0;i<5;i++){
22 22         scanf("%d",&in);
23 23         printf("%d\n",func4(14,0,in));
24 24     }
25 25     return 0;
26 26 }
```

暂时是还没有弄懂func4函数的具体作用，貌似就只是一系列操作而已。这份代码的正确性也不敢保证。但是根据测试结果而言，结论是没有问题的。测试的结果为输入为8,9或者11时，func4的返回值为1。其实这还是一个一个测试出来的。

得到这个结论以后，我们这一关就可以说已经过了，第一个数为8、9或者11，第二个数为1。

所以输入 (8,1) , (9,1) , (11,1) 三组中的任意一组即可过关：

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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!
2 217
Halfway there!
9 1
So you got that one. Try this one.
```

至此第四关已过，还剩两关。

phase_5:

(一)首先查看其反汇编代码：

```

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>
 0x08048deb <+51>:   mov    -0xc(%ebp),%eax
 0x08048dee <+54>:   and    $0xf,%eax
 0x08048df1 <+57>:   mov    %eax,-0xc(%ebp)
 0x08048df4 <+60>:   cmp    $0xf,%eax
 0x08048df7 <+63>:   je    0x8048e22 <phase_5+106>
 0x08048df9 <+65>:   mov    $0x0,%ecx
 0x08048dfe <+70>:   mov    $0x0,%edx
 0x08048e03 <+75>:   mov    $0x804a1c0,%ebx
 0x08048e08 <+80>:   add    $0x1,%edx
 0x08048e0b <+83>:   mov    (%ebx,%eax,4),%eax
 0x08048e0e <+86>:   add    %eax,%ecx
 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

End of assembler dump.

```

(二)下面对其进行解释与说明：

```

1  0x08048db8 <+0>:    push   %ebp           // 保存旧的%ebp
2  0x08048db9 <+1>:    mov    %esp,%ebp
3  0x08048dbb <+3>:    push   %esi           // 保存旧的%esi
4  0x08048dbc <+4>:    push   %ebx           // 保存旧的%ebx
5  0x08048dbd <+5>:    sub    $0x20,%esp
6  0x08048dc0 <+8>:    lea    -0x10(%ebp),%eax
7  0x08048dc3 <+11>:   mov    %eax,0xc(%esp)
8  0x08048dc7 <+15>:   lea    -0xc(%ebp),%eax
9  0x08048dca <+18>:   mov    %eax,0x8(%esp)
10 0x08048dce <+22>:   movl   $0x804a23e,0x4(%esp) // 输入两个整数
11 0x08048dd6 <+30>:   mov    0x8(%ebp),%eax
12 0x08048dd9 <+33>:   mov    %eax,(%esp)
13 0x08048ddc <+36>:   call   0x8048840 <__isoc99_sscanf@plt> // 读入数据，储存在地址-0xc(%ebp)和-0x10(%ebp)中
14 0x08048de1 <+41>:   cmp    $0x1,%eax
15 0x08048de4 <+44>:   jg    0x8048deb <phase_5+51> // 函数的返回值，即读入数据的个数要大于1
16 0x08048de6 <+46>:   call   0x80490d1 <explode_bomb>
17 0x08048deb <+51>:   mov    -0xc(%ebp),%eax // 将输入的第一个数存入%eax
18 0x08048dee <+54>:   and    $0xf,%eax // 将%eax的值和0xf做与运算，实际上就是取%eax的值的最低四位
19 0x08048df1 <+57>:   mov    %eax,-0xc(%ebp)
20 0x08048df4 <+60>:   cmp    $0xf,%eax // 将%eax的值和0xf比较
21 0x08048df7 <+63>:   je    0x8048e22 <phase_5+106> // 相等的话爆炸，说明输入的第一个数的最低四位的值不等于0xf
22 0x08048df9 <+65>:   mov    $0x0,%ecx // %ecx=0
23 0x08048dfe <+70>:   mov    $0x0,%edx // %edx=0
24 0x08048e03 <+75>:   mov    $0x804a1c0,%ebx // %ebx=0x804a1c0
25 0x08048e08 <+80>:   add    $0x1,%edx // %edx=%edx+1
26 0x08048e0b <+83>:   mov    (%ebx,%eax,4),%eax // 将地址(%ebx+%eax*4)的储存值赋值给%eax
27 0x08048e0e <+86>:   add    %eax,%ecx // %ecx=%ecx+%eax

```

```

28 | 0x08048e10 <+88>:    cmp    $0xf,%eax          // 将%eax的值与0xf做比较
29 |
| 0x08048e13 <+91>:    jne    0x8048e08 <phase_5+80> 30 |
| 0x08048e15 <+93>:    mov    %eax,-0xc(%ebp)
| 0x08048e18 <+96>:    cmp    $0xf,%edx          // 将寄存器%edx的值和0xf做比较32 |
| 0x08048e1b <+99>:    jne    0x8048e22 <phase_5+106> 33 |
| 0x08048e1d <+101>:   cmp    %ecx,-0x10(%ebp)    // 将寄存器%ecx的值和你输入的第二个数做比较34 |
| 0x08048e20 <+104>:   je     0x8048e27 <phase_5+111> 35 |
| 0x08048e22 <+106>:   call   0x80490d1 <explode_bomb>36 |
| 0x08048e27 <+111>:   add    $0x20,%esp          // phase_5运行结束37 |
| 0x08048e2a <+114>:   pop    %ebx             // 恢复调用者栈帧38 |
| 0x08048e2c <+116>:   pop    %ebp             0x08048e2b <+115>:   pop    %esi
39 | 0x08048e2d <+117>:   ret
40 |

```

(三)接下来进行phase_5的运行流程分析:

输入部分跟前两关一模一样。

接下来有一个对输入的一个数与0xf进行相与的操作，后面再由判断可以知道输入的数据的低四位的值不能是0xf。

根据后面<phase_5+91>处的跳转语句，我们可以知道，在<phase_5+80>~<phase_5+91>之间进行了循环。

<phase_5+65>~<phase_5+75>之间的语句为循环前的初始化操作，即%ecx=0,%edx=0,%ebx=0x804a1c0。

在循环里面进行的操作有：

1.%edx=%edx+1;

2.对寄存器%eac进行赋值；

3.寄存器%eac的值加上寄存器%eax的值；

4.如果寄存器%eax的值为0xf的话跳出循环；

循环结束后对%edx的进行一次判断，如果不等于0xf的话，就爆炸。说明循环结束后，%edx的值一定为0xf。

然后将你输入的第二个数与寄存器%ecx的值进行比较，如果不相等则爆炸。说明你输入的第二个数必须等于当前寄存器%ecx的值。

对运行过程中的循环进行具体分析：

循环前%edx的值为0，循环以后要求%edx=0xf，由于每次循环%edx的会+1，故要求循环进行15次。而循环退出的条件为%eax等于0xf，所以必须在循环恰好执行15次的时候，%eax的值为0xf。那么我们现在来看%eax的值的变化过程：

每次对%eax赋值的语句为：mov (%ebx,%eax,4), %eax。 %ebx=0x804a1c0。

也就是地址%ebx加上一个偏移量得到的地址里面储存的值赋值给%eax。我们先看一下%ebx后面多个内存单元的值：

```
(gdb) p *0x804a1c0@18
$3 = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5, 622879781, 1931812964}
```

那么当%eax的值为2时，执行mov (%ebx,%eax,4), %eax后，%eax的值就应该为14，再到下一次赋值结束以后%eax的值就应该为6，再下一次就应该为15。显然不符合要求，因为还没有循环15次。所以我们就是要确定一个%eax的初值，使循环在进行了15次以后，%eax的值恰好为15。

直接给定一个数，跟上述操作一样一步一步地去模拟显然不是一个好的选择。我们可以根据最终的结果来逆推%eax的初值。下面为推导过程：

最后一次%eax的值为15;

那么前一次%eax的值就应该为6

那么倒数第二次%eax的值为14;

那么倒数第三次%eax的值为2;

同理，这么推理下去。可以推理出一条数据链：

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

我们逆着推回去15步，得到%eax的值为5，那么从5开始，顺着循环15次得到值就是15，所以%eax的初值为15。即输入的第一个数的最低四位的值为5。

然后就是对%ecx的值进行分析，因为你输入的第二个数应该等于循环结束以后%ecx的值，对于%ecx来说，每次循环的过程中，%ecx的值会加上%eax的值，由于循环的时候是%eax的值先变化。所以%eax的初值5，并没有加到%ecx中，所以%ecx的值为%eax后面14次变化过程中所有数值的和，为115。所以我们输入的第二个数应该为115。

由此我们可以得到这一关的通关密码了，需要输入两个数，第一个数的最低四位的值为5，第二个数为115。所以满足 $(16X+5, 115)$ 形式的解($X \geq 0$)均可过关。如下：

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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!
2 217
Halfway there!
9 1
So you got that one. Try this one.
21 115
Good work! On to the next...
```

至此还剩下最后一关待解。

phase_6:

(一)首先查看其反汇编代码：

```
(gdb) disass 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
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
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>
0x08048ce4 <+91>:   jmp    0x8048cac <phase_6+35>
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
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>
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>
0x08048d52 <+201>:    call   0x80490d1 <explode_bomb>
0x08048d57 <+206>:    mov    0x8(%ebx),%ebx
0x08048d5a <+209>:    add    $0x1,%esi
0x08048d5d <+212>:    cmp    $0x5,%esi
0x08048d60 <+215>:    jne   0x8048d49 <phase_6+192>
0x08048d62 <+217>:    add    $0x5c,%esp
0x08048d65 <+220>:    pop    %ebx
0x08048d66 <+221>:    pop    %esi
0x08048d67 <+222>:    pop    %edi
0x08048d68 <+223>:    pop    %ebp
0x08048d69 <+224>:    ret

```

End of assembler dump.

(二)下面对其进行解释与说明:

```

1  0x08048c89 <+0>:    push   %ebp           //保存旧的%ebp
2  0x08048c8a <+1>:    mov    %esp,%ebp
3  0x08048c8c <+3>:    push   %edi           //保存旧的%edi,%esi,%ebx
4  0x08048c8d <+4>:    push   %esi
5  0x08048c8e <+5>:    push   %ebx
6  0x08048c8f <+6>:    sub    $0x5c,%esp      //为新的栈帧分配空间
7  0x08048c92 <+9>:    lea    -0x30(%ebp),%eax    //为函数<read_six_numbers>传参
8  0x08048c95 <+12>:   mov    %eax,0x4(%esp)    //输入的六个数的第一个数储存在地址-0x30(%ebp)中
9  0x08048c99 <+16>:   mov    0x8(%ebp),%eax    //其余的值顺序储存在从-0x30(%ebp)的递增的地址中
10 0x08048c9c <+19>:   mov    %eax,(%esp)
11 0x08048c9f <+22>:   call   0x804910b <read_six_numbers> //读入数据
12 <span style="color:#ff0000;"> 0x08048ca4 <+27>:   mov    $0x0,%esi          //%esi=0
13 0x08048ca9 <+32>:   lea    -0x30(%ebp),%edi    //%edi=-0x30(%ebp),lea指令, 存入的是地址, 不是值
14 0x08048cac <+35>:   mov    (%edi,%esi,4),%eax //%eax=(%edi+%esi*4),即把输入的第%esi+1个数赋值给%eax, 初始是第一个数
15 0x08048caf <+38>:   sub    $0x1,%eax          //%eax=%eax-1
16 0x08048cb2 <+41>:   cmp    $0x5,%eax          //如果%eax>5则爆炸, 说明%eax的应该<=5, 由于%eax等于输入的第%esi+1个数-1
17 0x08048cb5 <+44>:   jbe   0x8048cbc <phase_6+51> //所以输入的第%esi+1个数的值应该<=6
18 0x08048cb7 <+46>:   call   0x80490d1 <explode_bomb>
19 0x08048cbc <+51>:   add    $0x1,%esi          //%esi=%esi+1
20 0x08048cbf <+54>:   cmp    $0x6,%esi          //如果%esi==6, 则跳转到<phase_6+93>
21 0x08048cc2 <+57>:   je    0x8048ce6 <phase_6+93>
22 0x08048cc4 <+59>:   lea    (%edi,%esi,4),%ebx //把储存第%esi+1个数的地址赋值给%ebx
23 0x08048cc7 <+62>:   mov    %esi,-0x4c(%ebp) //把%esi的值存入地址-0x4c(%ebp)
24 0x08048cca <+65>:   mov    -0x4(%edi,%esi,4),%eax //把第%esi个数存入%eax, 第一次运行到这里的时候%esi=1
25 0x08048cce <+69>:   cmp    (%ebx),%eax    //将第%ebx地址储存的数与第%esi个进行比较, 第一次%ebx指向第%esi+1个数
26 0x08048cd0 <+71>:   jne   0x8048cd7 <phase_6+78> //相等则爆炸
27 0x08048cd2 <+73>:   call   0x80490d1 <explode_bomb>
28 0x08048cd7 <+78>:   addl   $0x1,-0x4c(%ebp) //地址-0x4c(%ebp)储存的值+1
29 0x08048cdb <+82>:   add    $0x4,%ebx          //%ebx=%ebx+4, 即%ebx指向下一个数
30 0x08048cde <+85>:   cmpl   $0x5,-0x4c(%ebp) //如果地址-0x4c(%ebp)储存的值<=5, 则跳转到<phase_6+65>
31 0x08048ce2 <+89>:   jle   0x8048cca <phase_6+65>
32 0x08048ce4 <+91>:   jmp   0x8048cac <phase_6+35> //跳转回到<phase_6+35></span>
33 <span style="color:#3333ff;"> 0x08048ce6 <+93>:   mov    $0x0,%ebx          //%ebx=0
34 0x08048ceb <+98>:   lea    -0x30(%ebp),%edi    //%edi=-0x30(%ebp)
35 0x08048cee <+101>:  jmp   0x8048d06 <phase_6+125> //跳转到<phase_6+125>
36 0x08048cf0 <+103>:  mov    0x8(%edx),%edx    //把地址0x8(%edx)的值赋值给%edx
37 0x08048cf3 <+106>:  add    $0x1,%eax          //%eax=%eax+1

```

```

38 0x08048cf6 <+109>: cmp %ecx,%eax //将%ecx的值与%eax的值进行比较39 |
 0x08048cf8 <+111>: jne 0x8048cf0 <phase_6+103> //如果不相等, 跳转到<phase_6+103>40 |
 0x08048cfa <+113>: mov %edx,-0x48(%ebp,%esi,4) //%edx的值储存到地址%ebp+%esi*4-0x4841 |
 0x08048cfe <+117>: add $0x1,%ebx //%ebx=%ebx+142 |
 0x08048d01 <+120>: cmp $0x6,%ebx //如果%ebx的值等于6, 跳转到<phase_6+147>43 |
 0x08048d04 <+123>: je 0x8048d1c <phase_6+147>44 | //将%ecx的值与%eax的值进行比较39 |
45 0x08048d08 <+127>: mov (%edi,%ebx,4),%ecx //将第%ebx+1个数赋值给%ecx, %ebx初值为0, 即把第一个输入的数赋值给%ecx
46 0x08048d0b <+130>: mov $0x804c0c4,%edx //%edx=0x804c0c4
47 0x08048d10 <+135>: mov $0x1,%eax //%eax=1
48 0x08048d15 <+140>: cmp $0x1,%ecx //比较%ecx与1的大小, 如果%ecx>1则跳转到<phase_6+103>
49 0x08048d18 <+143>: jg 0x8048cf0 <phase_6+103> //跳转到<phase_6+113></span>
50 0x08048d1a <+145>: jmp 0x8048cfa <phase_6+113> //地址-0x48(%ebp)储存的值赋值给%ebx
51 0x08048d1c <+147>: mov -0x48(%ebp),%ebx //地址-0x48(%ebp)储存的值赋值给%eax
52 0x08048d1f <+150>: mov -0x44(%ebp),%eax //地址-0x44(%ebp)储存的值赋值给%edx
53 0x08048d22 <+153>: mov %eax,0x8(%ebx) //将%eax的值存入地址0x8(%ebx), 注意%ebx的值是什么
54 0x08048d25 <+156>: mov -0x40(%ebp),%edx //地址-0x40(%ebp)储存的值赋值给%edx
55 0x08048d28 <+159>: mov %edx,0x8(%eax) //将%edx的值存入地址0x8(%eax), 注意%eax的值是什么
56 0x08048d2b <+162>: mov -0x3c(%ebp),%eax //地址-0x3c(%ebp)储存的值赋值给%eax
57 0x08048d2e <+165>: mov %eax,0x8(%edx) //将%eax的值存入地址0x8(%edx), 注意%edx的值是什么
58 0x08048d31 <+168>: mov -0x38(%ebp),%edx //地址-0x38(%ebp)储存的值赋值给%edx
59 0x08048d34 <+171>: mov %edx,0x8(%eax) //将%edx的值存入地址0x8(%eax), 注意%eax的值是什么
60 0x08048d37 <+174>: mov -0x34(%ebp),%eax //地址-0x34(%ebp)储存的值赋值给%eax
61 0x08048d3a <+177>: mov %eax,0x8(%edx) //将%eax的值存入地址0x8(%edx), 注意%edx的值是什么
62 0x08048d3d <+180>: movl $0x0,0x8(%eax) //将0存入地址0x8(%eax)
63 0x08048d44 <+187>: mov $0x0,%esi //%esi=0
64 0x08048d49 <+192>: mov 0x8(%ebx),%eax //将地址0x8(%ebx)的值存入%eax
65 0x08048d4c <+195>: mov (%ebx),%edx //将地址(%ebx)储存的值存入%edx
66 0x08048d4e <+197>: cmp (%eax),%edx //比较地址(%eax)储存的值与寄存器%edx的值的大小
67 0x08048d50 <+199>: jge 0x8048d57 <phase_6+206> //如果%edx的值小于地址(%eax)储存的值则爆炸
68 0x08048d52 <+201>: call 0x80490d1 <explode_bomb> //地址0x8(%ebx)储存的值存入%ebx
69 0x08048d57 <+206>: mov 0x8(%ebx),%ebx //%esi=%esi+1
70 0x08048d5a <+209>: add $0x1,%esi //如果%esi!=5, 则跳转到<phase_6+192>
71 0x08048d5d <+212>: cmp $0x5,%esi //恢复调用者栈帧
72 0x08048d60 <+215>: jne 0x8048d49 <phase_6+192> //<phase_6>运行结束, 释放内存
73 0x08048d62 <+217>: add $0x5c,%esp //恢复调用者栈帧
74 0x08048d65 <+220>: pop %ebx //恢复调用者栈帧
75 0x08048d66 <+221>: pop %esi //恢复调用者栈帧
76 0x08048d67 <+222>: pop %edi //恢复调用者栈帧
77 0x08048d68 <+223>: pop %ebp //恢复调用者栈帧
78 0x08048d69 <+224>: ret //返回调用者
79

0x08048ca4 <+27>: mov $0x0,%esi //%esi=0
0x08048ca9 <+32>: lea -0x30(%ebp),%edi //%edi=-0x30(%ebp), lea指令, 存入的是地址, 不是值
0x08048cac <+35>: mov (%edi,%esi,4),%eax //%eax=(%edi+%esi*4), 即把输入的第%esi+1个数赋值给%eax, 初始是第一个数
0x08048caf <+38>: sub $0x1,%eax //%eax=%eax-1
0x08048cb2 <+41>: cmp $0x5,%eax //如果%eax>5则爆炸, 说明%eax的应该<=5, 由于%eax等于输入的第%esi+1个数-1
0x08048cb5 <+44>: jbe 0x8048cbc <phase_6+51> //所以输入的第%esi+1个数的值应该<=6
0x08048cb7 <+46>: call 0x80490d1 <explode_bomb> //%esi=%esi+1
0x08048cbc <+51>: add $0x1,%esi //如果%esi==6, 则跳转到<phase_6+93>
0x08048cbf <+54>: cmp $0x6,%esi //把储存第%esi+1个数的地址赋值给%ebx
0x08048cc2 <+57>: je 0x8048ce6 <phase_6+93> //把%esi的值存入地址-0x4c(%ebp)
0x08048cc4 <+59>: lea (%edi,%esi,4),%ebx //把第%esi个数存入%eax, 第一次运行到这里的时候%esi=1
0x08048cc7 <+62>: mov %esi,-0x4c(%ebp) //将第%ebx地址储存的数与第%esi个进行比较, 第一次%ebx指向第%esi+1个数
0x08048cca <+65>: mov -0x4(%edi,%esi,4),%eax //相等则爆炸
0x08048cce <+69>: cmp (%ebx),%eax //地址-0x4c(%ebp)储存的值+1
0x08048cd0 <+71>: jne 0x8048cd7 <phase_6+78> //%ebx=%ebx+4, 即%ebx指向下一个数
0x08048cd2 <+73>: call 0x80490d1 <explode_bomb> //如果地址-0x4c(%ebp)储存的值<=5, 则跳转到<phase_6+65>
0x08048cd7 <+78>: addl $0x1,-0x4c(%ebp) //跳转回到<phase_6+35>
0x08048cdb <+82>: add $0x4,%ebx //%ebx=0
0x08048cde <+85>: cmpl $0x5,-0x4c(%ebp) //%edi=-0x30(%ebp)
0x08048ce2 <+89>: jle 0x8048cca <phase_6+65> //跳转到<phase_6+125>
0x08048ce4 <+91>: jmp 0x8048cac <phase_6+35> //把地址0x8(%edx)的值赋值给%edx
0x08048ce6 <+93>: mov $0x0,%ebx //%eax=%eax+1
0x08048ceb <+98>: lea -0x30(%ebp),%edi //将%ecx的值与%eax的值进行比较
0x08048cee <+101>: jmp 0x8048d06 <phase_6+125> //如果不相等, 跳转到<phase_6+103>
0x08048cf0 <+103>: mov 0x8(%edx),%edx //%edx的值储存到地址%ebp+%esi*4-0x48
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          // %ebx=%ebx+1
0x08048d01 <+120>: cmp    $0x6,%ebx          // 如果%ebx的值等于6, 跳转到<phase_6+147>
0x08048d04 <+123>: je     0x8048d1c <phase_6+147>
0x08048d06 <+125>: mov    %ebx,%esi          // %esi=%ebx
0x08048d08 <+127>: mov    (%edi,%ebx,4),%ecx // 将第%ebx+1个数赋值给%ecx,%ebx初值为0, 即把第一个输入的数赋值给%ecx
0x08048d0b <+130>: mov    $0x804c0c4,%edx // %edx=0x804c0c4
0x08048d10 <+135>: mov    $0x1,%eax          // %eax=1
0x08048d15 <+140>: cmp    $0x1,%ecx          // 比较%ecx与1的大小, 如果%ecx>1则跳转到<phase_6+103>
0x08048d18 <+143>: jg    0x8048cf0 <phase_6+103>
0x08048d1a <+145>: jmp    0x8048cfa <phase_6+113> // 跳转到<phase_6+113>
0x08048d1c <+147>: mov    -0x48(%ebp),%ebx // 地址-0x48(%ebp)储存的值赋值给%ebx
0x08048d1f <+150>: mov    -0x44(%ebp),%eax // 地址-0x44(%ebp)储存的值赋值给%eax
0x08048d22 <+153>: mov    %eax,0x8(%ebx) // 将%eax的值存入地址0x8(%ebx),注意%ebx的值是什么
0x08048d25 <+156>: mov    -0x40(%ebp),%edx // 地址-0x40(%ebp)储存的值赋值给%edx
0x08048d28 <+159>: mov    %edx,0x8(%eax) // 将%edx的值存入地址0x8(%eax),注意%eax的值是什么
0x08048d2b <+162>: mov    -0x3c(%ebp),%eax // 地址-0x3c(%ebp)储存的值赋值给%eax
0x08048d2e <+165>: mov    %eax,0x8(%edx) // 将%eax的值存入地址0x8(%edx),注意%edx的值是什么
0x08048d31 <+168>: mov    -0x38(%ebp),%edx // 地址-0x38(%ebp)储存的值赋值给%edx
0x08048d34 <+171>: mov    %edx,0x8(%eax) // 将%edx的值存入地址0x8(%eax),注意%eax的值是什么
0x08048d37 <+174>: mov    -0x34(%ebp),%eax // 地址-0x34(%ebp)储存的值赋值给%eax
0x08048d3a <+177>: mov    %eax,0x8(%edx) // 将%eax的值存入地址0x8(%edx),注意%edx的值是什么
0x08048d3d <+180>: movl   $0x0,0x8(%eax) // 将0存入地址0x8(%eax)
0x08048d44 <+187>: mov    $0x0,%esi          // %esi=0
0x08048d49 <+192>: mov    0x8(%ebx),%eax // 将地址0x8(%ebx)的值存入%eax
0x08048d4c <+195>: mov    (%ebx),%edx          // 将地址(%ebx)储存的值存入%edx
0x08048d4e <+197>: cmp    (%eax),%edx          // 比较地址(%eax)储存的值与寄存器%edx的值的大小
0x08048d50 <+199>: jge    0x8048d57 <phase_6+206> // 如果%edx的值小于地址(%eax)储存的值则爆炸
0x08048d52 <+201>: call   0x80490d1 <explode_bomb> // 地址0x8(%ebx)储存的值存入%ebx
0x08048d57 <+206>: mov    0x8(%ebx),%ebx // %esi=%esi+1
0x08048d5a <+209>: add    $0x1,%esi          // 如果%esi!=5, 则跳转到<phase_6+192>
0x08048d5d <+212>: cmp    $0x5,%esi          // <phase_6>运行结束, 释放内存
0x08048d60 <+215>: jne    0x8048d49 <phase_6+192> // 恢复调用者栈帧
0x08048d62 <+217>: add    $0x5c,%esp          // 返回调用者
0x08048d65 <+220>: pop    %ebx
0x08048d66 <+221>: pop    %esi
0x08048d67 <+222>: pop    %edi
0x08048d68 <+223>: pop    %ebp
0x08048d69 <+224>: ret

```

(三)对于上面的注释, 个人感觉如果不结合实际的例子来说, 是没啥实质上的用处的。所以下面将结合一个具体的栗子进行解释。并且整个关卡分三个主要讲: 红色部分、蓝色部分和最后的黑色部分。

首先是红色部(<+27~+91>之间的语句):

这里先给出phase_6的栈帧结构的示意图(A~F为输入的六个数):

stack "bottom"	
%ebp→	saved %ebp
-4	
-8	
-0C	
-10	
-14	
-18	
-1C	F
-20	E
-24	D
-28	C
-2C	B
%edi→	A
-30	
-34	ad6
-38	ad5
-3C	ad4
-40	ad3
-44	ad2
-48	ad1
-4C	X
-50	
-54	
%esp→	-58
stack "top"	

首先初始化%esi=0, %edi=-0x30(%ebp)。

执行<+35~+62>之间的语句：

先判断第一个数的数值(%esi=0, 即A)必须大于等于1, 小于等于6。

%esi+1, 把第二个数(B)的地址(%edi,%esi,4)存入%ebx。

把%esi的值(为1)存入地址-0x40(%edp), 即此时图中的X=1。

执行<+65~+89>之间的语句：

首先是第二个数B (由%ebx指向) 与第一个数A比较, 如果相等就爆炸。

然后将X+1, 即X=2。再%ebx+4, 即%ebx指向下一个数C。

判断：如果X<=5, 跳转回到<+65>, 下一次判断是进行C与A之间的判断。

然后将X+1, 即X=3。再%ebx+4, 即%ebx指向下一个数D。

判断：如果X<=5, 跳转回到<+65>, 下一次判断是进行D与A之间的判断。

如此循环下去, 当X>5时, 程序进行了完了F与A之间的判断, 即这个循环可以判断出A与B、C、D、E、F中的任意一个均不相等。

这个循环结束后, 程序跳转到<+35>处。

执行<+35~+62>之间的语句(此时%esi的值为1)：

先判断第二个数的数值(%esi=1, 即B)必须大于等于1, 小于等于6。

%esi+1, 把第三个数(C)的地址(%edi,%esi,4)存入%ebx。

把%esi的值(为2)存入地址-0x40(%edp), 即此时图中的X=2。

执行<+65~+89>之间的语句：

首先是第三个数C (由%ebx指向) 与第一个数B比较, 如果相等就爆炸。

然后将X+1, 即X=3。再%ebx+4, 即%ebx指向下一个数D。

判断：如果X<=5，跳转回到<+65>，下一次判断是进行D与B之间的判断。

然后将X+1，即X=3。再%ebx+4，即%ebx指向下一个数E。

判断：如果X<=5，跳转回到<+65>，下一次判断是进行E与B之间的判断。

如此循环下去，当X>5时，程序进行了完了F与B之间的判断，即这个这个循环可以判断出B与C、D、E、F中的任意一个均不相等。

这个循环结束后，程序跳转到<+35>处。

后面再执行<+35~+62>，<+65~+89>。这里不再给出具体的说明：直接给出得到相似的结论为：C与D、E、F均不相等。

同理当红色部分的循环结束时，得到的结论为：A、B、C、D、E、F的取值范围均为[1, 6]，且两两间互不相等。这就是红色部分程序对你的输入作出的限制条件。

然后是蓝色部分(<+93~+145>之间的语句)：

```
1 <span style="color:#3333ff"> 0x08048ce6 <+93>:    mov    $0x0,%ebx          //%ebx=0
2   0x08048ceb <+98>:    lea    -0x30(%ebp),%edi      //%edi=-0x30(%ebp)
3   0x08048cee <+101>:    jmp    0x8048d06 <phase_6+125> //跳转到<phase_6+125>
4   0x08048cf0 <+103>:    mov    0x8(%edx),%edx      //把地址0x8(%edx)的值赋值给%edx
5   0x08048cf3 <+106>:    add    $0x1,%eax        //%eax=%eax+1
6   0x08048cf6 <+109>:    cmp    %ecx,%eax        //将%ecx的值与%eax的值进行比较
7   0x08048cf8 <+111>:    jne    0x8048cf0 <phase_6+103> //如果不相等，跳转到<phase_6+103>
8   0x08048cfa <+113>:    mov    %edx,-0x48(%ebp,%esi,4) //%edx的值储存到地址%ebp+%esi*4-0x48
9   0x08048cfe <+117>:    add    $0x1,%ebx        //%ebx=%ebx+1
10  0x08048d01 <+120>:    cmp    $0x6,%ebx        //如果%ebx的值等于6，跳转到<phase_6+147>
11  0x08048d04 <+123>:    je     0x8048d1c <phase_6+147>
12  0x08048d06 <+125>:    mov    %ebx,%esi        //%esi=%ebx
13  0x08048d08 <+127>:    mov    (%edi,%ebx,4),%ecx //将第%ebx+1个数赋值给%ecx,%ebx初值为0，即把第一个输入的数赋值给%ecx
14  0x08048d0b <+130>:    mov    $0x804c0c4,%edx      //%edx=0x804c0c4
15  0x08048d10 <+135>:    mov    $0x1,%eax        //%eax=1
16  0x08048d15 <+140>:    cmp    $0x1,%ecx        //比较%ecx与1的大小，如果%ecx>1则跳转到<phase_6+103>
17  0x08048d18 <+143>:    jg    0x8048cf0 <phase_6+103>
18  0x08048d1a <+145>:    jmp    0x8048cfa <phase_6+113> //跳转到<phase_6+113></span>
```

首先是初始化：%ebx=0，%edi=-0x30(%ebp)。

为了便于后续分析，我们假设我们输入为4, 1, 3, 5, 6, 2。即A=4、B=1、C=3、D=5、E=6、F=2。

执行<+125~+145>之间的语句：

先把%ebx的值赋值给%esi。此时%esi=0。把第一个数(A=4)赋值给%ecx(指令mov(%edi,%ebx,4),%ecx)。

将值0x804c0c4存入寄存器%edx。将0x1存入寄存器%eax。

比较%ecx(此时%ecx=A=4)与1的大小，大于1跳转到<phase_6+103>

执行<+103~+123>之间的语句：

首先是mov 0x8(%edx), %edx。%edx之前被赋值为0x804c0c4。所以这里把地址0x804c0cc储存的值存入寄存器%edx。这里我们看一下地址0x804c0c4及其附近地址储存的内容，如下：

```
(gdb) p/x *0x804c0c4@3
$4 = {0x1a7, 0x1, 0x804c0b8}
```

所以这里把0x804c0b8存入寄存器%edx。

然后%eax的值+1，得到2，与%ecx的值(A=4)进行比较。不相等，跳转到<+103>。

执行mov 0x8(%edx), %edx。此时%ebx=0x804c0b8。所以这里把地址0x804c0c0储存的值存入寄存器%edx。

这里我们看一下地址0x804c0c0及其附近地址储存的内容，如下：

```
(gdb) p/x *0x804c0b8@3  
$11 = {0x6c, 0x2, 0x804c0ac}
```

所以这里把0x804c0ac存入寄存器%edx。

然后%eax的值+1，得到3，与%ecx的值(A=4)进行比较。不相等，跳转到<+103>。

然后再一次对%ebx进行迭代操作：

```
(gdb) p/x *0x804c0ac@3  
$12 = {0x155, 0x3, 0x804c0a0}
```

然后%eax的值+1，得到4，与%ecx的值(A=4)进行比较。相等，执行<+113>。

执行<+113~+123>之间的语句：

把%edx储存的值存入地址%ebp+%esi*4-0x48，这个时候%esi=%ebx=0，所以这个时候把%edx储存的值存入地址%ebp-0x48。即图中的ad1位置。

然后ebx的值+1，故%ebx=1。

(稍微回顾一下过程，这里由于%ecx=A=4，所以这里把%ebx迭代四次以后得到值存入ad1位置。)

执行<+125~+145>之间的语句：

现在%ebx的值为1了，故%esi的值也为1，并把第二个数(B=1)存入寄存器%ecx。

再把值0x804c0c4存入寄存器%edx。把0x1存入寄存器%eax。

判断，此时%ecx==1，故直接跳转到<+113>处。

执行<+113~+123>之间的语句：

此时%esi=1，%edx=0x804c0c4。

把%edx储存的值存入地址%ebp+%esi*4-0x48，这个时候%esi=%ebx=1，所以这个时候把%edx储存的值存入地址%ebp-0x44。即图中的ad2位置。

然后ebx的值+1，故%ebx=2。

执行<+125~+145>之间的语句、执行<+103~123>之间的语句、执行<+113~+123>之间的语句。同第一次的模拟，我们可以得出结论，将%edx迭代三次以后的结果存入ad3位置。

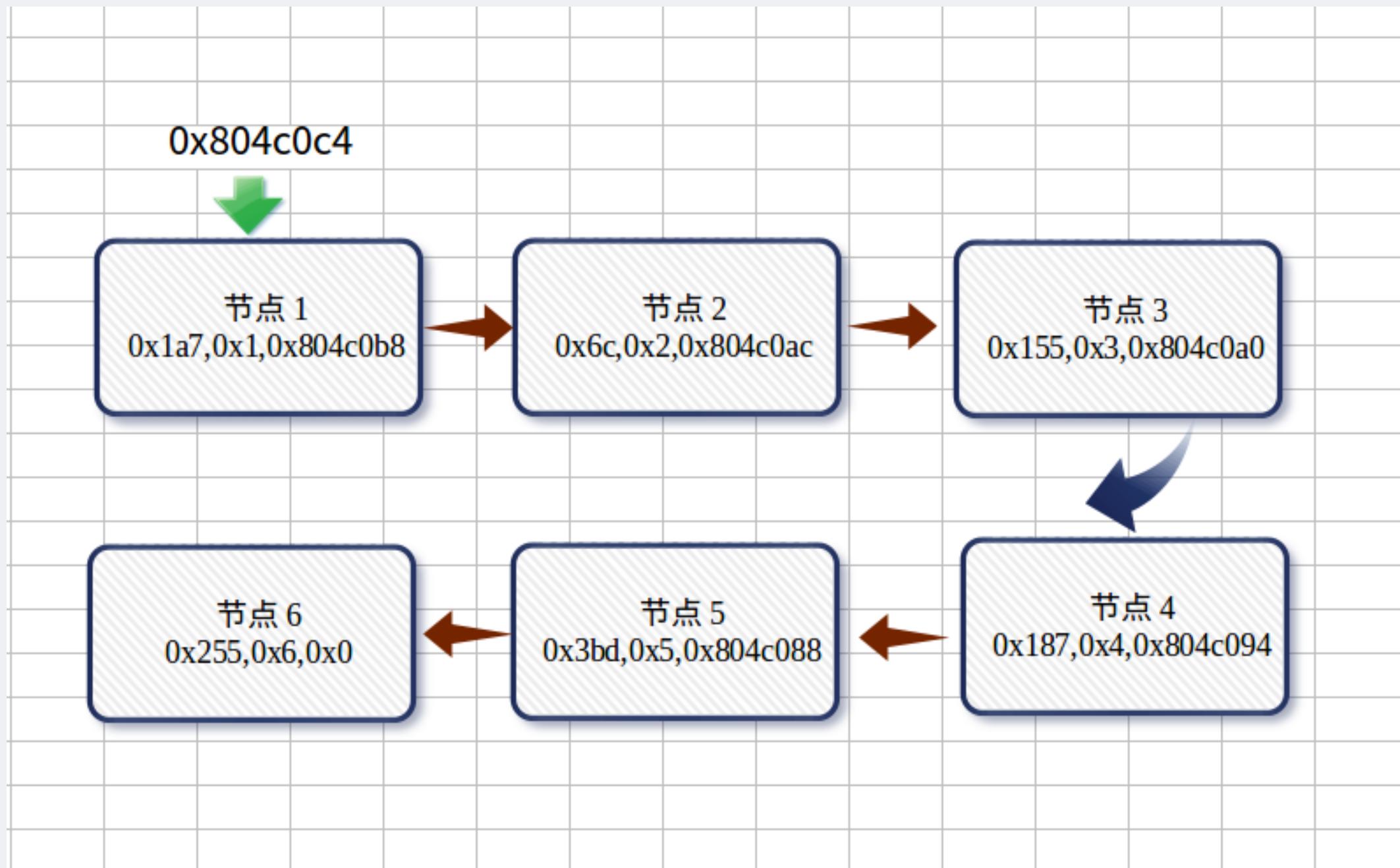
接着再循环一次就是把%edx迭代5次以后的结果存入ad4位置。再就是把%edx迭代6次以后的结果存入ad5位置，再把%edx迭代2次以后的结果存入ad6位置，此时%ebx在执行<+120>处的指令时，其值为6，跳出蓝色部分的循环。

回顾一下这个循环过程：

其实循环里面对于%ebx的迭代操作，其实像极了链表的操作，把最初的那个赋给%edx的值看作是链表的头节点的地址，然后每一个节点都有一个指针域指向下一个节点，那么这个迭代过程就是在节点之间移动。基于这个想法我们把各个节点的信息给查看一遍，如下：

```
(gdb) p/x *0x804c0c4@3
$13 = {0x1a7, 0x1, 0x804c0b8}
(gdb) p/x *0x804c0b8@3
$14 = {0x6c, 0x2, 0x804c0ac}
(gdb) p/x *0x804c0ac@3
$15 = {0x155, 0x3, 0x804c0a0}
(gdb) p/x *0x804c0a0@3
$16 = {0x187, 0x4, 0x804c094}
(gdb) p/x *0x804c094@3
$17 = {0x3bd, 0x5, 0x804c088}
(gdb) p/x *0x804c088@3
$18 = {0x255, 0x6, 0x0}
(gdb) p/x *0x804c0c0@3
```

更进一步地我们可以得到一个初始状态的各个节点之间的关系图如下：



而蓝色部分代码的作用呢，就是把这些节点进行排序，比如上述我输入的4、1、3、5、6、2。

操作以后的结果就是节点4放在了ad1位置、节点1放在了ad2位置、节点3放在了ad3位置、节点5放在了ad4位置、节点6放在了ad5位置、节点2放在了ad6位置。

最后是黑色部分代码的说明(<+147~+215>)：

1	0x08048d1c <+147>:	mov -0x48(%ebp),%ebx	//地址-0x48(%ebp)储存的值赋值给%ebx
2	0x08048d1f <+150>:	mov -0x44(%ebp),%eax	//地址-0x44(%ebp)储存的值赋值给%eax
3	0x08048d22 <+153>:	mov %eax,0x8(%ebx)	//将%eax的值存入地址0x8(%ebx)，注意%ebx的值是什么
4	0x08048d25 <+156>:	mov -0x40(%ebp),%edx	//地址-0x40(%ebp)储存的值赋值给%edx
5	0x08048d28 <+159>:	mov %edx,0x8(%eax)	//将%edx的值存入地址0x8(%eax)，注意%eax的值是什么
6	0x08048d2b <+162>:	mov -0x3c(%ebp),%eax	//地址-0x3c(%ebp)储存的值赋值给%eax
7	0x08048d2e <+165>:	mov %eax,0x8(%edx)	//将%eax的值存入地址0x8(%edx)，注意%edx的值是什么
8	0x08048d31 <+168>:	mov -0x38(%ebp),%edx	//地址-0x38(%ebp)储存的值赋值给%edx
9	0x08048d34 <+171>:	mov %edx,0x8(%eax)	//将%edx的值存入地址0x8(%eax)，注意%eax的值是什么
10	0x08048d37 <+174>:	mov -0x34(%ebp),%eax	//地址-0x34(%ebp)储存的值赋值给%eax
11	0x08048d3a <+177>:	mov %eax,0x8(%edx)	//将%eax的值存入地址0x8(%edx)，注意%edx的值是什么
12	0x08048d3d <+180>:	movl \$0x0,0x8(%eax)	//将0存入地址0x8(%eax)
13	0x08048d44 <+187>:	mov \$0x0,%esi	//%esi=0
14	0x08048d49 <+192>:	mov 0x8(%ebx),%eax	//将地址0x8(%ebx)的值存入%eax
15	0x08048d4c <+195>:	mov (%ebx),%edx	//将地址(%ebx)储存的值存入%edx
16	0x08048d4e <+197>:	cmp (%eax),%edx	//比较地址(%eax)储存的值与寄存器%edx的值的大小
17	0x08048d50 <+199>:	jge 0x8048d57 <phase_6+206>	//如果%edx的值小于地址(%eax)储存的值则爆炸
18	0x08048d52 <+201>:	call 0x80490d1 <explode_bomb>	
19	0x08048d57 <+206>:	mov 0x8(%ebx),%ebx	//地址0x8(%ebx)储存的值存入%ebx

```
20 | 0x08048d5a <+209>: add    $0x1,%esi          //%esi=%esi+1_21|
| 0x08048d5d <+212>: cmp    $0x5,%esi          //如果%esi!=5, 则跳转到<phase_6+192>22 |
| 0x08048d60 <+215>: jne    0x8048d49 <phase_6+192>
```

当蓝色部分代码结束后，我们可以理解为六个节点按照我们输入数字的顺序依次储存
在-0x48(%ebp)~-0x34(%ebp)中。在此基础上继续分析余下代码。

执行<+150~+180>之间的语句：

<+150~+153>: %ebx=ad1; %eax=ad2; *(ad1+8)=ad2。即把ad1处节点(节点4)的指
针域指向ad2处的节点(节点1);

<+156~+159>: %edx=ad3; *(ad2+8)=ad3。即把ad2处节点(节点1)的指针域指向ad3
处的节点(节点3)。

<+162~+165>: %eax=ad4; *(ad3+8)=ad4。即把ad3处节点(节点3)的指针域指向ad4
处的节点(节点5)。

<+168~+171>: %edx=ad5; *(ad4+8)=ad5。即把ad4处节点(节点5)的指针域指向ad5
处的节点(节点6)。

<+174~+177>: %eax=ad6; *(ad5+8)=ad6。即把ad5处节点(节点6)的指针域指向ad6
处的节点(节点1)。

<+180> : *(ad6+8)=NULL。即把ad6节点的指针域指向NULL。

执行完以后，相当于我们改变了各个节点的指针域，使各个节点按照你的输入顺序连接。就例子
而言，得到的结果为：

节点4-->节点1-->节点3-->节点5-->节点6-->节点2-->NULL。

接下来执行<+187~+215>之间的语句(开始时%ebx=ad1, %esi=0):

<+192>: %eax=*(ad1+8)。注意这里是把节点4指针域储存的值(节点1的地址)赋值
给%eax；

<+195>: %edx=*ad1。即把节点4储存的值赋值给%edx。

<+197>: 然后比较(%eax)与%edx的大小，(%eax)的值就是节点1储存的值。

<+199~+201>: 如果节点4的值小于节点1的值则爆炸。

<+206>: %ebx=*(%ebx+8); 即%ebx=ad2。

<+209~+215>: %esi的值+1。如果%esi的值!=5，跳转到<+192>处，继续循环。

下一次循环中进行的是节点1与节点3之间的比较。且节点3储存的值必须小于节点1储存的
值。然后%ebx继续指向下一个节点，%esi的值+1。

当%esi==5时，进行比较的是节点2储存的值与节点6之间的比较，且节点2储存的值必须小于节
点6储存的值。

综合<+187~+125>之间的语句来看，我们得到的结论为：ad1~ad6所对应节点储存的值应该
单调递减，否则爆炸。

到这里后我们就梳理一下phase_6所做的事情：

1、输入六个数字，这六个数字为1~6的一个排列。

2、程序里面有已经赋值好的六个节点，程序运行的时候，节点间的顺序会改变。具体来
说，如果你输入的是A[1~6]，那么第A[i]个节点会被置于第i个位置。比如输入4、1、3、5、6、
2，那么程序会把第四个节点换到第一位，第一个节点换到第二位，第三个节点换到第三位，依次
类推。

3、交换结束以后，程序会判断交换得到的节点1~节点6的值是否单调递减，单调递减则通
过，否则爆炸。

那么通关条件也就出来了：

我们输入1~6个一排列，使程序内部的节点按照这个顺序排序以后得到的各个节点值单调递减即可。那么我们使节点储存值大的节点排前面即可：其实上面我们已经查看了各个节点储存的值：

```
(gdb) p/x *0x804c0c4@3
$13 = {0x1a7, 0x1, 0x804c0b8}
(gdb) p/x *0x804c0b8@3
$14 = {0x6c, 0x2, 0x804c0ac}
(gdb) p/x *0x804c0ac@3
$15 = {0x155, 0x3, 0x804c0a0}
(gdb) p/x *0x804c0a0@3
$16 = {0x187, 0x4, 0x804c094}
(gdb) p/x *0x804c094@3
$17 = {0x3bd, 0x5, 0x804c088}
(gdb) p/x *0x804c088@3
$18 = {0x255, 0x6, 0x0}
(gdb) p/x *0x804c0c0@3
```

我们可以发现值最大的节点为节点5，值为0x3bd。

其次是节点6，值为0x255。再其次是节点1,值为0x1a7。

再其次是节点4，值为0x187。再就是节点3,值为0x155，最后是节点2，值为0x6c。

所以我们的输入应该为5、6、1、4、3、2。

来看一下输入以后的结果：

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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!
2 217
Halfway there!
9 1
So you got that one. Try this one.
21 115
Good work! On to the next...
5 6 1 4 3 2
Congratulations! You've defused the bomb!
[Inferior 1 (process 2858) exited normally]
(gdb) █
```

Congratulations! You've defused the bomb! 炸弹已拆除！至此六个关卡已经成功过关。

secret_phase:

做完了要求的六个关卡就很开心。但是身为社会主义的接班人怎么可以仅仅满足于此呢，不是还有一个传说中的“隐藏关卡”吗，这个自然也是想要完成的。多说，我们开始。这里将会分两个部分进行关卡的说明：

secret_phase的进入：

(一)但我们解完六个关卡后似乎程序已经运行完成了，说明隐藏关卡还需要一定的条件才能触发，那么我们首先就要先去找到触发隐藏关卡的条件。我们看bomb.c文件，每段phase函数运行完成以后又会运行一个phase_defused()函数，这个函数我们在上述拆炸弹过程中都没有用到，自然它的嫌疑就很大，故我们这个函数的具体内容：

```
(gdb) disass phase_defused
Dump of assembler code for function phase_defused:
0x08049014 <+0>:    push   %ebp
0x08049015 <+1>:    mov    %esp,%ebp
0x08049017 <+3>:    sub    $0x88,%esp
0x0804901d <+9>:    mov    %gs:0x14,%eax
0x08049023 <+15>:   mov    %eax,-0xc(%ebp)
0x08049026 <+18>:   xor    %eax,%eax
0x08049028 <+20>:   cmpl   $0x6,0x804c3d0
0x0804902f <+27>:   jne    0x80490bb <phase_defused+167>
0x08049035 <+33>:   lea    -0x5c(%ebp),%eax
0x08049038 <+36>:   mov    %eax,0x10(%esp)
0x0804903c <+40>:   lea    -0x64(%ebp),%eax
0x0804903f <+43>:   mov    %eax,0xc(%esp)
0x08049043 <+47>:   lea    -0x60(%ebp),%eax
0x08049046 <+50>:   mov    %eax,0x8(%esp)
0x0804904a <+54>:   movl   $0x804a200,0x4(%esp)
0x08049052 <+62>:   movl   $0x804c4d0,(%esp)
0x08049059 <+69>:   call   0x8048840 <__isoc99_sscanf@plt>
0x0804905e <+74>:   cmp    $0x3,%eax
0x08049061 <+77>:   jne    0x80490a7 <phase_defused+147>
0x08049063 <+79>:   movl   $0x804a209,0x4(%esp)
0x0804906b <+87>:   lea    -0x5c(%ebp),%eax
0x0804906e <+90>:   mov    %eax,(%esp)
0x08049071 <+93>:   call   0x8048fab <strings_not_equal>
0x08049076 <+98>:   test   %eax,%eax
0x08049078 <+100>:  jne    0x80490a7 <phase_defused+147>
0x0804907a <+102>:  movl   $0x804a2dc,0x4(%esp)
0x08049082 <+110>:  movl   $0x1,(%esp)
0x08049089 <+117>:  call   0x8048870 <__printf_chk@plt>
0x0804908e <+122>:  movl   $0x804a304,0x4(%esp)
0x08049096 <+130>:  movl   $0x1,(%esp)
0x0804909d <+137>:  call   0x8048870 <__printf_chk@plt>
0x080490a2 <+142>:  call   0x8048c1b <secret_phase>
0x080490a7 <+147>:  movl   $0x804a33c,0x4(%esp)
0x080490af <+155>:  movl   $0x1,(%esp)
0x080490b6 <+162>:  call   0x8048870 <__printf_chk@plt>
0x080490bb <+167>:  mov    -0xc(%ebp),%eax
0x080490be <+170>:  xor    %gs:0x14,%eax
0x080490c5 <+177>:  je    0x80490cc <phase_defused+184>
0x080490c7 <+179>:  call   0x80487b0 <__stack_chk_fail@plt>
0x080490cc <+184>:  leave 
0x080490cd <+185>:  lea    0x0(%esi),%esi
0x080490d0 <+188>:  ret    
End of assembler dump.
```

(二)我在<+142>的位置看到了调用函数<secret_phase>, 说明隐藏关卡的进入和这个函数有关无疑了。那么我们接下来具体分析其触发条件:

```

1  0x08049014 <+0>:    push   %ebp          //保存旧的%ebp
2  0x08049015 <+1>:    mov    %esp,%ebp
3  0x08049017 <+3>:    sub    $0x88,%esp      //为新的栈帧分配空间
4  0x0804901d <+9>:    mov    %gs:0x14,%eax    //!!! 待解决, 冒似这个以及后面和这条相似的语句是用来检测栈是否被破坏的
5  0x08049023 <+15>:   mov    %eax,-0xc(%ebp)
6  0x08049026 <+18>:   xor    %eax,%eax
7  0x08049028 <+20>:   cmpl   $0x6,0x804c3d0    //比较内存0x804c3d0储存的值与6的大小
8  0x0804902f <+27>:   jne    0x80490bb <phase_defused+167> //如果不等于6, 结束运行, 不会触发<secret_phase>
9  0x08049035 <+33>:   lea    -0x5c(%ebp),%eax    //看到后面的函数, 知道又开始传参了
10 0x08049038 <+36>:  mov    %eax,0x10(%esp)    //首先传入了三个地址-0x5c(%ebp)、-0x64(%ebp)和-0x60(%ebp)
11 0x0804903c <+40>:  lea    -0x64(%ebp),%eax    //应该有一种感觉, 这个是输入的数据存放的地址
12 0x0804903f <+43>:  mov    %eax,0xc(%esp)
13 0x08049043 <+47>:  lea    -0x60(%ebp),%eax
14 0x08049046 <+50>:  mov    %eax,0x8(%esp)
15 0x0804904a <+54>:  movl   $0x804a200,0x4(%esp) //然后又传入了两个未知的地址
16 0x08049052 <+62>:  movl   $0x804c4d0,(%esp)
17 0x08049059 <+69>:  call   0x8048840 <__isoc99_sscanf@plt> //调用输入函数
18 0x0804905e <+74>:  cmp    $0x3,%eax        //函数返回值必须是3, 说明这里要输入三个数据
19 0x08049061 <+77>:  jne    0x80490a7 <phase_defused+147>
20 0x08049063 <+79>:  movl   $0x804a209,0x4(%esp) //注意到下面的<string_not_equal>, 可知这里又是在传参
21 0x0804906b <+87>:  lea    -0x5c(%ebp),%eax    //传入了一个未知地址和一个保存我们数据输入的地址-0x5c(%ebp)
22 0x0804906e <+90>:  mov    %eax,(%esp)
```

```
23 0x08049071 <+93>: call 0x8048fab <strings_not_equal> //调用函数24 | 0x08049076 <+98>: test %eax,%eax
25 0x08049078 <+100>: jne 0x80490a7 <phase_defused+147> //如果返回值为1, 结束, 不会触发隐藏关, 说明进行比较的两个字符串要相等
26 0x0804907a <+102>: movl $0x804a2dc,0x4(%esp)           //传参, 为调用函数做准备
27 0x08049082 <+110>: movl $0x1,(%esp)
28 0x08049089 <+117>: call 0x8048870 <__printf_chk@plt> //调用函数, printf, 一猜就是要输出个什么东西
29 0x0804908e <+122>: movl $0x804a304,0x4(%esp)           //又是传参
30 0x08049096 <+130>: movl $0x1,(%esp)
31 0x0804909d <+137>: call 0x8048870 <__printf_chk@plt> //调用函数, 输出某些东西
32 0x080490a2 <+142>: call 0x8048c1b <secret_phase>      //终于可以调用<secret_phase>了
33 0x080490a7 <+147>: movl $0x804a33c,0x4(%esp)           //传参
34 0x080490af <+155>: movl $0x1,(%esp)
35 0x080490b6 <+162>: call 0x8048870 <__printf_chk@plt> //输出
36 0x080490bb <+167>: mov -0xc(%ebp),%eax                //下面两句话的作用, 按我现在的理解就是检测栈是否被破坏
37 0x080490be <+170>: xor %gs:0x14,%eax
38 0x080490c5 <+177>: je 0x80490cc <phase_defused+184> //相等就正常退出
39 0x080490c7 <+179>: call 0x80487b0 <__stack_chk_fail@plt> //否则检测错误原因
40 0x080490cc <+184>: leave                                //为返回调用者做准备
41 0x080490cd <+185>: lea 0x0(%esi),%esi
42 0x080490d0 <+188>: ret //返回
```

(三)下面我们从<+20>处开始讲:

<+20>: 先是比较内存0x804c3d0储存的值与6的大小, 为什么是6呢, 考虑到前面有六个关卡, 会不会是每解一关都会对该地址的值+1, 然后解完六关以后这个地址的值恰好为6, 满足条件, 带着这个疑问我们去验证一下。就是设置断点单步运行, 查看这个内存单元的值, 查看过程如下:

```
(gdb) b phase_1
Breakpoint 1 at 0x8048f67
(gdb) b phase_2
Breakpoint 2 at 0x8048d6f
(gdb) b phase_3
Breakpoint 3 at 0x8048ea7
(gdb) b phase_4
Breakpoint 4 at 0x8048e34
(gdb) b phase_5
Breakpoint 5 at 0x8048dbd
(gdb) b phase_6
Breakpoint 6 at 0x8048c8f
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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.

Breakpoint 1, 0x08048f67 in phase_1 ()
(gdb) p/x *0x804c3d0
$1 = 0x1
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
0 1 1 2 3 5

Breakpoint 2, 0x08048d6f in phase_2 ()
(gdb) p/x *0x804c3d0
$2 = 0x2
```

```
(gdb) c
Continuing.
That's number 2. Keep going!
2 217

Breakpoint 3, 0x08048ea7 in phase_3 ()
(gdb) p/x *0x804c3d0
$3 = 0x3
(gdb) c
Continuing.
Halfway there!
9 1

Breakpoint 4, 0x08048e34 in phase_4 ()
(gdb) p/x *0x804c3d0
$4 = 0x4
(gdb) c
Continuing.
So you got that one. Try this one.
21 115

Breakpoint 5, 0x08048dbd in phase_5 ()
(gdb) p/x *0x804c3d0
$5 = 0x5
(gdb) c
Continuing.
Good work! On to the next...
5 6 1 4 3 2

Breakpoint 6, 0x08048c8f in phase_6 ()
(gdb) p/x *0x804c3d0
$6 = 0x6
```

我们可以发现地址0x804c3d0储存的值的变化过程是符合我们的猜想的，那么它具体是在哪里变化的呢？显然我们在各个关卡的函数里并没有发现这一操作，那么改变这个值的操作就只会出现在<read_line>函数里，我们对其进行查看(这里只对这一操作段进行分析)：

0x08049311 <+267>:	mov	0x804c3d0,%eax
0x08049316 <+272>:	lea	0x0(%eax,4),%edx
0x0804931d <+279>:	lea	(%edx,%eax,1),%ecx
0x08049320 <+282>:	shl	\$0x4,%ecx
0x08049323 <+285>:	movb	\$0x0,0x804c3df(%ebx,%ecx,1)
0x0804932b <+293>:	lea	0x1(%eax),%ecx
0x0804932e <+296>:	mov	%ecx,0x804c3d0

主要看到<+267>和<+293>和<+296>着三个操作：先把地址0x804c3d0的值存入存入%eax，然后%ecx=%eax+1，然后再把%ecx的值存入地址0x804c3d0，整个过程就是实现地址0x804c3d0储存的值+1。也就是说我们读入六行数据以后地址0x804c3d0的值才为6，也说明了，最起码我们需要解决前面六个关卡才能开启隐藏关卡。

接着往下看：问题就到了那个未知的地址0x804a200，以及地址0x804a4d0。我们对这两个地址进行分析：

查看地址0x804a200的内容：

```
(gdb) x/s 0x804a200
0x804a200:      "%d %d %s"
```

说明下面的函数要输入三个数据，这也与后面对函数返回值的判断语句做了很好的说明。

然后是地址0x804a4d0，如果按照前面几关的思路的话，这个应该是我们输入字符串存放的首地址，但是我们之前这个参数是0x8(%ebp)，和这个不一样。再分析其原因：以前我们函数的输入是直接从当前输入的一行中读入数据，而我们输入的一行字符串的首地址就储存在0x8(%ebp)，所以以前是直接传的0x8(%ebp)，所以这里传的地址就极有可能是我们某次输入一行字符串的首地址。那么接下来我们去查看这个地址到底是哪次输入字符串的首地址，调试过程如下：

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/xbb0720/bomb/bomblab-20180409/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.

Breakpoint 1, 0x08048f67 in phase_1 ()
(gdb) i r
eax            0x804c3e0      134530016
ecx            0x1          1
edx            0x0          0
ebx            0x0          0
esp            0xbffffef0    0xbffffef0
ebp            0xbffffef08   0xbffffef08
esi            0xbffffefd4   -1073745964
edi            0xb7fb8000    -1208254464
eip            0x8048f67     0x8048f67 <phase_1+6>
eflags          0x286      [ PF SF IF ]
cs             0x73        115
ss             0x7b        123
ds             0x7b        123
es             0x7b        123
fs             0x0          0
gs             0x33        51
```

```
(gdb) p/x *0xbffffef10
$7 = 0x804c3e0
(gdb) c
Continuing.
Phase 1 defused. How about the next one?
0 1 1 2 3 5

Breakpoint 2, 0x08048d6f in phase_2 ()
(gdb) p/x *0xbffffef10
$8 = 0x804c430
(gdb) c
Continuing.
That's number 2. Keep going!
2 217

Breakpoint 3, 0x08048ea7 in phase_3 ()
(gdb) p/x *0xbffffef10
$9 = 0x804c480
(gdb) c
Continuing.
Halfway there!
9 1

Breakpoint 4, 0x08048e34 in phase_4 ()
(gdb) p/x *0xbffffef10
$10 = 0x804c4d0
(gdb) c
Continuing.
So you got that one. Try this one.
21 115

Breakpoint 5, 0x08048dbd in phase_5 ()
(gdb) p/x *0xbffffef10
$11 = 0x804c520
(gdb) c
Continuing.
Good work! On to the next...
5 6 1 4 3 2

Breakpoint 6, 0x08048c8f in phase_6 ()
(gdb) p/x *0xbffffef10
$12 = 0x804c570
```

分析：我们看到调用phase()函数时，%ebp的值为0xbffff08，所以我们每次在读入后查看地址0xbffffef10的内容。我们发现保存第一次至第六次输入的字符串的首地址分别为：

0x804c3e0、0x804c430、0x804c480、0x804c4d0、0x804c520、0x804c570

由此看来，这里数据的读入，是从第四次的输入中读入的。我们再回顾一下第四次的输入：当时是输入两个整数，而这里要是要输入两个整数以及一个字符串，说明我们需要在第四次输入时，输入完两个整数后还需要输入一个字符串。可能这里会有疑问，这里再输入一个字符串不会导致bomb up吗？其实是不会的，因为当时传的参数为“%d %d”，所以在读入两个整数以后便不会再读入。

phase_4还是可以正常运行。那么我们在后面需要就提输入什么内容呢？接着往下看：

下面是一个比较函数，其中一个参数是保存我们输入的字符串的首地址，另外一个想想就知道应该是进行比较的字符串的首地址了。我们对其内容进行查看：

```
(gdb) x/s 0x804a209
0x804a209:      "DrEvil"
```

这就说明了我们在进行phase_4的输入的还需要在两个数字后面输入一个字符串“DrEvil”。

再往下看可以看到两个输出函数，会对首地址分别为0x804a2dc和0x804a304的字符串进行输出：

```
0x0804907a <+102>:    movl    $0x804a2dc,0x4(%esp)
0x08049082 <+110>:    movl    $0x1,(%esp)
0x08049089 <+117>:    call    0x8048870 <__printf_chk@plt>
0x0804908e <+122>:    movl    $0x804a304,0x4(%esp)
0x08049096 <+130>:    movl    $0x1,(%esp)
0x0804909d <+137>:    call    0x8048870 <__printf_chk@plt>
```

我们对其进行查看：

```
(gdb) x/s 0x804a2dc
0x804a2dc:      "Curses, you've found the secret phase!\n"
(gdb) x/s 0x804a304
0x804a304:      "But finding it and solving it are quite different...\n"
```

发现程序就提示你找到隐藏关卡了。其实输出函数的下一句就是调用<secret_phase>

我们看一下实际操作：

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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!
2 217
Halfway there!
9 1 DrEvil
So you got that one. Try this one.
21 115
Good work! On to the next...
5 6 1 4 3 2
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

接下来就是输入<secret_phase>要求你输入的东西了。所以说...上述的分析还只是仅仅进入secret_phase，接下来就可以进入下一部分--解决<secret_phase>了。

secret_phase的解决：

(一)同以前的关卡一样，我们先查看其反汇编代码：

```

Dump of assembler code for function secret_phase:
0x08048c1b <+0>:    push    %ebp
0x08048c1c <+1>:    mov     %esp,%ebp
0x08048c1e <+3>:    push    %ebx
0x08048c1f <+4>:    sub     $0x14,%esp
0x08048c22 <+7>:    call    0x8049206 <read_line>
0x08048c27 <+12>:   movl    $0xa,0x8(%esp)
0x08048c2f <+20>:   movl    $0x0,0x4(%esp)
0x08048c37 <+28>:   mov     %eax,(%esp)
0x08048c3a <+31>:   call    0x80488b0 <strtol@plt>
0x08048c3f <+36>:   mov     %eax,%ebx
0x08048c41 <+38>:   lea    -0x1(%eax),%eax
0x08048c44 <+41>:   cmp    $0x3e8,%eax
0x08048c49 <+46>:   jbe    0x8048c50 <secret_phase+53>
0x08048c4b <+48>:   call    0x80490d1 <explode_bomb>
0x08048c50 <+53>:   mov     %ebx,0x4(%esp)
0x08048c54 <+57>:   movl    $0x804c178,(%esp)
0x08048c5b <+64>:   call    0x8048bca <fun7>
0x08048c60 <+69>:   cmp    $0x5,%eax
0x08048c63 <+72>:   je     0x8048c6a <secret_phase+79>
0x08048c65 <+74>:   call    0x80490d1 <explode_bomb>
0x08048c6a <+79>:   movl    $0x804a134,0x4(%esp)
0x08048c72 <+87>:   movl    $0x1,(%esp)
0x08048c79 <+94>:   call    0x8048870 <_printf_chk@plt>
0x08048c7e <+99>:   call    0x8049014 <phase_defused>
0x08048c83 <+104>:  add    $0x14,%esp
0x08048c86 <+107>:  pop    %ebx
0x08048c87 <+108>:  pop    %ebp
0x08048c88 <+109>:  ret
End of assembler dump.

```

(二)再对其进行解释说明:

```

1  0x08048c1b <+0>:    push    %ebp          //保存旧的%ebp
2  0x08048c1c <+1>:    mov     %esp,%ebp
3  0x08048c1e <+3>:    push    %ebx
4  0x08048c1f <+4>:    sub     $0x14,%esp      //为新栈帧分配空间
5  0x08048c22 <+7>:    call    0x8049206 <read_line> //读入一行数据
6  0x08048c27 <+12>:   movl    $0xa,0x8(%esp) //给后面的<strtol>函数传参
7  0x08048c2f <+20>:   movl    $0x0,0x4(%esp) //函数<strtol>调用时写为: strtol(char *ch1,char *ch2,int base){}, 作用
8  0x08048c37 <+28>:   mov     %eax,(%esp) //是把字符串*ch1当作一个base进制的整数并返回, 这里的base=0xa, 说明为10进制
9  0x08048c3a <+31>:   call    0x80488b0 <strtol@plt> //调用strtol, 说明把你输入的字符串转化为一个当作一个10进制的整数来使用
10 0x08048c3f <+36>:  mov     %eax,%ebx //把返回值存入寄存器%ebx
11 0x08048c41 <+38>:  lea    -0x1(%eax),%eax //%eax=%eax-1
12 0x08048c44 <+41>:  cmp    $0x3e8,%eax //如果%eax>0x3e8就爆炸, 说明你输入的数要小于等于0x3e9
13 0x08048c49 <+46>:  jbe    0x8048c50 <secret_phase+53> //并且这里是无符号比较, 所以你的输入还要大于等于1
14 0x08048c4b <+48>:  call    0x80490d1 <explode_bomb>
15 0x08048c50 <+53>:  mov     %ebx,0x4(%esp) //将你输入的那个数作为函数<fun7>的参数
16 0x08048c54 <+57>:  movl    $0x804c178,(%esp) //将一个地址作为函数<fun7>的参数
17 0x08048c5b <+64>:  call    0x8048bca <fun7>
18 0x08048c60 <+69>:  cmp    $0x5,%eax //将<fun7>的返回值和5做比较, 如果不等于5就爆炸
19 0x08048c63 <+72>:  je     0x8048c6a <secret_phase+79>
20 0x08048c65 <+74>:  call    0x80490d1 <explode_bomb>
21 0x08048c6a <+79>:  movl    $0x804a134,0x4(%esp) //看下面的函数又是要输出一个字符串, 所以这里要传参
22 0x08048c72 <+87>:  movl    $0x1,(%esp)
23 0x08048c79 <+94>:  call    0x8048870 <_printf_chk@plt> //输出一个字符串
24 0x08048c7e <+99>:  call    0x8049014 <phase_defused> //又递归调用<phase_defused>?, 不过好在地址0x804c3d0储存的值不会再等于6了
25 0x08048c83 <+104>: add    $0x14,%esp
26 0x08048c86 <+107>: pop    %ebx
27 0x08048c87 <+108>: pop    %ebp
28 0x08048c88 <+109>: ret

```

(三)这个函数的整体流程还是比较清晰的:

先输入一个字符串, 然后转化为数字(范围为[1~0x3e9]), 再把这个数作为函数<fun7>的参数。要求使<fun7>的返回值等于5。

那么要考虑的是<fun7>进行的是什么操作, 然后看要输入什么值, 才能使其返回值为5。所以我们先看到<fun7>:

```
(gdb) disass fun7
Dump of assembler code for function fun7:
0x08048bca <+0>:    push   %ebp
0x08048bcb <+1>:    mov    %esp,%ebp
0x08048bcd <+3>:    push   %ebx
0x08048bce <+4>:    sub    $0x14,%esp
0x08048bd1 <+7>:    mov    0x8(%ebp),%edx
0x08048bd4 <+10>:   mov    0xc(%ebp),%ecx
0x08048bd7 <+13>:   mov    $0xffffffff,%eax
0x08048bdc <+18>:   test   %edx,%edx
0x08048bde <+20>:   je     0x8048c15 <fun7+75>
0x08048be0 <+22>:   mov    (%edx),%ebx
0x08048be2 <+24>:   cmp    %ecx,%ebx
0x08048be4 <+26>:   jle    0x8048bf9 <fun7+47>
0x08048be6 <+28>:   mov    %ecx,0x4(%esp)
0x08048bea <+32>:   mov    0x4(%edx),%eax
0x08048bed <+35>:   mov    %eax,(%esp)
0x08048bf0 <+38>:   call   0x8048bca <fun7>
0x08048bf5 <+43>:   add    %eax,%eax
0x08048bf7 <+45>:   jmp    0x8048c15 <fun7+75>
0x08048bf9 <+47>:   mov    $0x0,%eax
0x08048bfe <+52>:   cmp    %ecx,%ebx
0x08048c00 <+54>:   je     0x8048c15 <fun7+75>
0x08048c02 <+56>:   mov    %ecx,0x4(%esp)
0x08048c06 <+60>:   mov    0x8(%edx),%eax
0x08048c09 <+63>:   mov    %eax,(%esp)
0x08048c0c <+66>:   call   0x8048bca <fun7>
0x08048c11 <+71>:   lea    0x1(%eax,%eax,1),%eax
0x08048c15 <+75>:   add    $0x14,%esp
0x08048c18 <+78>:   pop    %ebx
0x08048c19 <+79>:   pop    %ebp
0x08048c1a <+80>:   ret
End of assembler dump.
```

显然地，这又是一个递归函数。其具体内容不再展开细讲，下面直接给出该函数的伪代码：

```
1 int fun7(Node *root,int val){
2     if(root->value==val)
3         return 0;
4     else if(root->value>val)
5         return 2*fun7(root->left,val);
6     else
7         return 2*fun7(root->right,val)+1;
8 }
```

这个函数会用来对一颗二叉树进行查询操作，最初传入函数的那个地址就是根节点的地址，具体的返回值来说就像代码中写的那样：当当前节点的值等于你查询的值时候，返回0，否则根据值的大小的判断进行不同的递归查询。当查询值大于当前节点的值的时候，递归查询右子树，否则递归查询左子树。根据代码去读懂汇编代码应该就不是那么困难了。

那么我们知道了函数的功能及操作，那么怎么确定我们要输入什么数字呢？一个一个试显然是不可取的(取值范围有点大[1~0x3e9])。我们可以逆推，具体来说：

由于函数最终的返回值为5，那么我们可以推出上一次的返回值必定是2，并且是从右子树返回。

接着分析，返回值是2的话，那么再上一次的返回值必定是1，并且是从左子树返回。

返回值是1的话，那么再上一次的返回值必定是0，并且从右子树返回。

返回值是0的话，说明当前节点的值等于待查询的值，也就是你需要输入的值。

我们回顾一下函数查询过程，便可以得到这样的一条路径：root-->right-->left-->right。

我们需要输入的值就是最后一次查询到的结点储存的值，接下来，我们从给定的地址(根节点)出发查看这个值，如下：

```
(gdb) p/x *0x804c178@3
$3 = {0x24, 0x804c16c, 0x804c160}
(gdb) p/x *0x804c160@3
$4 = {0x32, 0x804c148, 0x804c130}
(gdb) p/x *0x804c148@3
$5 = {0x2d, 0x804c124, 0x804c0dc}
(gdb) p/x *0x804c0dc@3
$6 = {0x2f, 0x0, 0x0}
```

我们查询得到我们最终的目标节点储存的值为0x2f，即47。也就是说我们需要输入47。

那么我们来看一下组最终的运行结果：

```
(gdb) r
Starting program: /home/xbb0720/bomb/bomblab-20180409/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!
2 217
Halfway there!
9 1 DrEvil
So you got that one. Try this one.
21 115
Good work! On to the next...
5 6 1 4 3 2
Curses, you've found the secret phase!
But finding it and solving it are quite different...
47
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
[Inferior 1 (process 2385) exited normally]
```

Wow! You've defused the secret stage!至此炸弹已完美拆除。

CSAPP lab2 bomb (深入了解计算机系统 实验二) - Izjsqn的专栏

这个问题还得用GDB调试来做。截图做笔记吧，实在写不动了！1.执行反汇编 obj-dump -D bomb > mysrc.S 得到可执行文件的机器级程序（汇编文件）。2.搜索main（每一个应用...）

想对作者说点什么

 WinnocentLY: 感谢分享 secret_phase“当查询值大于当前节点的值的时候，递归查询左子树”这个描述反了吧？ (3周前 #1楼) [查看回复\(1\)](#)

计算机系统原理实验之BombLab二进制炸弹1、2关 - m0_37157965的博客

◎ 513

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... 来自：[m0_37157965...](#)

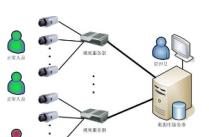
记一次bomblab实验 - 王者荣耀

◎ 420

首先反汇编代码，objdump -d bomb > asm.txt，对bomb进行反汇编并将汇...

来自：[王者荣耀](#)

开始之前 &#amp;&#amp;&#amp;&#amp;&#amp;&#amp;&#amp;&#amp;&#amp;&nbsp;&#amp;&#amp;...

来自: [lyh_lc的博客](#)

狄仁杰用一个方法找出蛇灵易容的人神探狄仁杰:寺里出现命案

百度广告

广告

ICS bomblab总结 - m0_57375647的博客

◎ 514

bomb ab 主要考察对汇编指令的阅读和理解. 由于不同人拿到的lab炸弹可能不一样, ... 来自: [m0_57375647...](#)

[ICS] Bomblab总结 - pku_15120的博客

◎ 760

bomblab总结

来自: [pku_15120的博客](#)

计算机系统原理实验之BombLab二进制炸弹3、4关 - m0_37157965的博客

◎ 459

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... 来自: [m0_37157965...](#)

Boom! ! ! 计算机系统, 从理解到爆炸, Bomblab - cww97的博客

◎ 7239

进入目录下 ./bomb 开始运行炸弹 对于炸弹command not found之类的鬼畜情况: ch... 来自: [cww97的博客](#)

CSAPP LAB——二进制炸弹 (bomblab) - The_V_的博客

◎ 1.8万

LAB3 预先准备 首先查看整个bomb.c的代码, 发现整个炸弹组是由6个小炸弹 (函数... 来自: [The_V_的博客](#)

重大通知! 11月起, 北京上班族可正式申读在名校职研究生

尚德机构 · 顶新



股市奇才17年不亏之谜, 方法令人意想不到

市沁 · 燐燚

下载 CSAPP bomb lab内容加解答

04-22

著名的bomb lab, CSAPP(深入理解计算机系统)一书中9个lab之一, 卡耐基 梅隆大学 Introduction to Computer 课程实验之一, 这里面包含实验内容及我的解答过程,

文章热词 机器学习 机器学习课程 机器学习教程 深度学习视频教程 深度学习学习**相关热词** zabbix深入 内存对齐的深入 storm 深入 tomcat深入 cocos 深入

CSAPP第二次实验 bomb二进制炸弹的破解 - ppy_的博客

◎ 2.1万

一个关于破解的初级实验。考的就是汇编代码的熟练程度和分析能力。不过有几个函... 来自: [ppy_的博客](#)

深入理解计算机系统(第二版) 家庭作业 第十一章 - zhanyu1990的专栏

◎ 2148

11.6 A. 因为read_requesthdrs中已经打印出了请求报头, 所以只要打印请求行即可... 来自: [zhanyu1990的...](#)

深入理解计算机系统attack lab - peanWang的博客

◎ 647

Attack lab 来自: [peanWang的博客](#)

<csapp> data lab (《深入理解计算机系统》lab1) - Stone

◎ 4649

po主是在读大学生一枚, 最近在 来自: [Stone](#)

恨不在北京, 11月起上班族可申读1年制成人本科学历, 毕业即大学

金领教育 · 顶新



呼和浩特2018赚钱新方法! 轻松月入高薪!

建木投资 · 燐燚

CSAPP Bomb Lab答案 Border relations with Canada have never been better. 1 2 4 ... 来自: [H-ZeX Coding Life](#)

CS:APP:BombLab 4 - bysoulwarden的博客

◎ 56

这篇文章意为记录下Bomb4中比较微妙的地方。之前经过分析, phase_4需要两个int... 来自: [bysoulwarden的...](#)

下载 深入理解计算机系统 lab - aaronguozheng

04-14

这是所有的lab, 可以去看一看, 应该对学习ICS很有帮助

深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phas...

◎ 1861

本文接 深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phase_2... 来自: [Void9711的博客](#)

深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phas...

◎ 2326

本文接 深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phase_3... 来自: [Void9711的博客](#)



恨不在北京, 11月起上班族可申读1年制成人本科学历, 毕业即大学

金领教育 · 顶新



男人房事时间短咋办? 教你一个技巧轻松达到30分钟!

珑悦 · 燐燚

下载 CSAPP Lab2 bomblab二进制炸弹 - 话歪之地

10-22

CSAPP Lab2 bomblab二进制炸弹 拆炸弹实验源代码 深入理解计算机系统课程实验二资料。程序设计与计算机系统课程。

深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phas...

◎ 1990

刚刚开始学习深入理解计算机系统 (CSAPP) (原书第二版), 初次接触到汇编语... 来自: [Void9711的博客](#)

CSAPP: bomb lab - xuzhezhaozhao的专栏

◎ 1.4万

有兴趣做这个lab的可以到这里下载, 里面包含原文档和bomb二进制文件和我的解答... 来自: [xuzhezhaozhao...](#)

The Report of the Bomb Lab - 毒液哥的专栏

◎ 36

The Report of the Bomb Lab The Lab2 in CS:APP 毒液哥 Fudan University The CS...

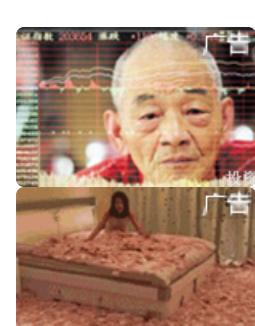
来自: [毒液哥的专栏](#)

CSAPP: Bomb Lab (3) - 未选之路

◎ 4458

Bomb 5 & Bomb 6

来自: [未选之路](#)



老股民酒后无意说漏: 20年炒股 坚持只看1指标

集升商贸 · 燐燚

偷偷告诉你一个呼和浩特不用上班就能轻松赚钱的办法

涌德投资 · 燐燚

CSAPP-BOMB-LAB - mbinary

◎ 736

这是2016版的bomb 下载得到bomb.tar文件,解压后只有bomb二进制文件,以及一个b...

来自: [mbinary](#)

CSAPP课程实验 bomb实验 拆炸弹实验 (1) - syq1207的博客

◎ 7617

由于内容较长, 所以打算分成几个部分来写。 实验准备知识: 实验三是CSAPP课程... 来自: [syq1207的博客](#)

计算机系统_炸弹 (boom) 实验/逆向工程实验 (phase_5) - Xindolia_Ri...

◎ 510

第五关 0000000000401002 <phase_5>: 401002: 48 83 ec 18 ... 来自: [Xindolia_Ring的...](#)

深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phas...

◎ 1753

本文接深入理解计算机系统 (CSAPP) 课程实验bomb程序炸弹实验日志 (phase_1... 来自: [Void9711的博客](#)

CSAPP LAB2 BombLab - P.H.S

◎ 573

CSAPP LAB2 BombLab给一个可执行文件, 需要按顺序输入6个字串, 每个字串是...

来自: [P.H.S](#)



一个长期喝土蜂蜜的人, 竟然变成了这样! 看到后立马转给家人!!

河北森和矿 · 燐燚



呼和浩特新出手机赚钱方式, 第一批90后已经赚翻了!

objdump命令的使用 - 北落师门'的专栏 ◎ 7.7万
objdump命令的使用 objdump命令是Linux下的反汇编目标文件或者可执行文件的命... 来自： 北落师门'的专栏

《深入理解计算机系统（原书第三版）》pdf - 程序员成长之路 ◎ 4.1万
下载 地址：网盘下载 内容简介 ····· 和第2版相比，本版内容上*大的变化是，从... 来自： 程序员成长之路

《深入理解计算机系统》学习总结一 - Katrina_ALi的博客 ◎ 1042
前几天开始看的《深入理解计算机系统》这本书，东西超级多，书老厚了。但是... 来自： Katrina_ALi的博客

下载 深入理解计算机系统上海交大课件 - qq_29719481 01-13
上海交通大学深入理解计算机系统的课件，物有所值。

《深入理解计算机系统》（原书第二版）粗读笔记 - wojiao555555的专栏 ◎ 2433
日记 2016年10月27日095534 2016年10月28日090501 2016年10月31日094406 20... 来自： wojiao555555的...

 一个长期喝土蜂蜜的人，竟然变成了这样！看到后立马转给家人！！
河北森和矿 · 爆款
 男人行房时间短咋办，老中医说：多吃它，让你延长40分钟。
北京悦鑫汇 · 爆款

下载 深入理解计算机系统（英文，完整，PDF修正版） 04-06
Prentice.Hall.Computer.Systems.A.Programmer's.Perspective 深入理解计算机系统 英文版，网上大部分都不是确第4章就是把第4章放在最后，影响阅读，我修正了下，把第4章插...

CSAPP: Bomb Lab (1) - 未选之路 ◎ 3442
CSAPP 实验：Bomb Lab (1) 相关知识准备以及bomb 1、bomb 2的解决过程 来自： 未选之路

BombLab Phase-3 & Phase-4 & Phase-5 - 启示录 ◎ 68
导航 BombLab Phase-1 & Phase-2 BombLab phase-6 & secr... 来自： 启示录

CSAPP深入理解计算机——bomblab (2018) - xiaolian_hust的博客 ◎ 157
准备工作1. 做该实验，务必已经看完了深入理解计算机系统的第三章节。了解常见c... 来自： xiaolian_hust的...

计算机系统基础——bomblab实验环境配置 - 扣扣biubiubiu~ ◎ 1075
一、实验目的 1) 理解arm汇编语言，学会使用调试器。2) 熟悉安卓开发板的使用和... 来自： 扣扣biubiubiu~

 老股民酒后无意说漏：20年炒股 坚持只看1指标
集升商贸 · 爆款
 揭露：呼和浩特宝妈在家用微信赚钱，一个月竟然给自己买了2克拉的钻戒！
涌德投资 · 爆款

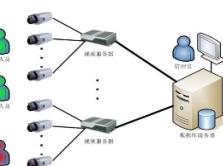
CMU bomb lab - BananaTree ◎ 1.2万
这个题的本质是让通过gdb和objdump的工具的使用，以及对汇编代码的理解找到程... 来自： BananaTree

objdump反汇编用法示例 - zoomdy's blog ◎ 2万
objdump反汇编，反汇编与源代码混合显示，C++符号逆向解析。 来自： zoomdy's blog

下载 深入理解计算机系统（第三版）英文 epub 03-17
深入理解计算机系统 英文版 epub格式的 Computer Systems Programmers Perspective 3rd

如何阅读深入理解计算机系统 - 路漫漫其修远 ◎ 1175
在找工作顺利结束之后，我又回顾了一下之前的标注，结合自己的笔试、面试经历，... 来自： 路漫漫其修远

下载 《深入理解计算机系统》课程主讲人本书作者pdf版本第10节 机器程序：程序 - php_ray 04-09
《深入理解计算机系统》作者本人的讲座pdf,视频及其他资料参考http://www.cs.cmu.edu/~213/schedule.html

 狄仁杰用一个方法找出蛇灵易容的人神探狄仁杰:寺里出现命案
百度广告 广告



下载 《深入理解计算机系统(原书第三版)》高清完整PDF下载 - I1505624

03-16

原书名: Computer Systems: A Programmer's Perspective, Third Edition 作者: (美) 兰德尔·E.布莱恩特 (Randal E.Bryant) 译者: 龚奕利 贺莲 丛书名: 计...

下载 深入理解计算机系统(中文PDF)(

06-29

·AMAZON五星图书，最伟大计算机科学教材之一；·卡耐基梅隆大学计算机学院院长，IEEE和ACM双院士倾力推出；·超过80所美国和世界一流大学计算机专业选用本书为教材。...

CSAPP Lab2: bomblab拆炸弹实验(汇编代码的理解) - 话歪之地的博客

◎ 1685

参考文章: 实验准备知识<http://blog.csdn.net/shiyuqing1207/article/details/4584941...> 来自: [话歪之地的博客](#)

计算机系统原理实验之BombLab二进制炸弹5、6关 - m0_37157965的博客

◎ 545

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... 来自: [m0_37157965...](#)

下载 ics lab2-bomblab - S-tone-R

05-01

《深入理解计算机系统》lab2-bomblab 利用assembly code破解bomb文件以拆除bomb文件中的炸弹, 使用的工具为gdb。

短信 接口

多用户商城系统

广告

在线客服系统

百度广告

csapp lab2 bomb 二进制炸弹 《深入理解计算机系统》 - fang92的专栏

◎ 4409

bomb炸弹实验 首先对bomb这个文件进行反汇编, 得到一个1000+的汇编程序, 看... 来自: [fang92的专栏](#)



AC-NEWBIE

关注

原创

78

粉丝

21

喜欢

评论

14

等级: 博客 3 访问: 1万+

积分: 963 排名: 6万+

勋章: 恒

intel OPTANE DC PERSISTENT MEMORY

内存革新
推动数据中心
价值腾飞

英特尔“傲腾”数据中心级持久性内存
大容量缔造大智慧
大幅拓展虚拟化服务规模
决胜数据未来

开启数据中心变革

广告

最新文章

CodeForces - 231E-Cactus (Tarjan缩点 +LCA)

CodeForces-238E-Meeting Her

Gym - 101158J -Cover the Polygon with Your Disk (模拟退火+多边形与圆面积的交)

P1337 [JSOI2004]平衡点 / 吊打XXX (模拟退火)

POJ-2420-A Star not a Tree?(模拟退火求

个人分类

CCF练习	6篇
搜索--BFS&&DFS	6篇
二分查找	3篇
思维	12篇
字符串	2篇

[展开](#)

归档

2018年11月	2篇
2018年10月	1篇
2018年9月	7篇
2018年8月	3篇
2018年7月	13篇

[展开](#)

热门文章

深入理解计算机系统--bomblab

阅读量: 3703

CCF201803-4-棋局评估

阅读量: 3594

CCF201803-3-URL映射

阅读量: 3555

计算机系统结构实验-模型机的组成

阅读量: 560

在Eclipse下新建工程及打开已有工程

阅读量: 390

最新评论

简易计算机系统综合设计--PC计数器

daima3: 好

深入理解计算机系统--bomblab

xbb224007: [reply]WinnocentLY[/reply] 的确反了，感谢指正，已修改O(∩_∩)O

深入理解计算机系统--bomblab

WinnocentLY: 感谢分享 secret_phase“当查询值大于当前节点的值的时候，递归查询左子树”这个描述反了吧？

第一场-E - Exponial

xbb224007: [reply]zhangzhenjunaixuxin[/reply] 本来是求 $n^m \% C$ ，而m特别...

第一场-E - Exponial

zhangzhenjunaixuxin: 大佬，为啥将模数由C降为 $\psi(C)$ ，为啥一直降下去



手绘网页



种头发危害



dji



手绘



联系我们



微信客服



QQ客服

QQ客服 kefu@csdn.net
客服论坛 400-660-0108
工作时间 8:00-22:00

[关于我们](#) | [招聘](#) | [广告服务](#) | [网站地图](#)

百度提供站内搜索 京ICP证09002463号
©1999-2018 江苏乐知网络技术有限公司
江苏知之为计算机有限公司 北京创新乐知
信息技术有限公司版权所有

[网络110报警服务](#) [经营性网站备案信息](#)
[北京互联网违法和不良信息举报中心](#)
[中国互联网举报中心](#)

联系我们

QQ客服 kefu@csdn.net
客服论坛 400-660-0108
工作时间 8:00-22:00

[关于我们](#) | [招聘](#) | [广告服务](#) | [网站地图](#)

百度提供站内搜索 京ICP证09002463号
©1999-2018 江苏乐知网络技术有限公司
江苏知之为计算机有限公司 北京创新乐知
信息技术有限公司版权所有

[网络110报警服务](#) [经营性网站备案信息](#)
[北京互联网违法和不良信息举报中心](#)
[中国互联网举报中心](#)