Lab Report of Lab2 Binary Bomb

一、 关卡密码

（一） Phase 1

```
Public speaking is very easy.
```

（二） Phase 2

```
1 2 6 24 120 720
```

（三） Phase 3

0 q 777	1 b 214	2 b 755	3 k 251	4 o 160	5 t 458	6 v 780	7 b 524

（四） Phase 4

```
9 austinpowers
```

（五） Phase 5

```
OPUKMA
```

（六） Phase 6

```
4 2 6 3 1 5
```

（七） Secret phase

```
1001
```

二、 推演过程

（零） 准备

形成 bomb.txt 文件，进入文件阅读代码：

```
objdump -d bomb > bomb.txt
```

阅读 main()，从：

```
8048a5b:    e8 c0 00 00 00        call   8048b20 <phase_1>
8048a60:    e8 c7 0a 00 00        call   804952c <phase_defused>
```

到：

```
8048b0a:    e8 89 02 00 00        call   8048d98 <phase_6>
8048b0f:    e8 18 0a 00 00        call   804952c <phase_defused>
```

不断调用<explode_bomb>函数，那么可以从 Phase 1 开始逐个处理。

（一） Phase 1

1.

```
8048b26:       8b 45 08           mov    0x8(%ebp),%eax
```

输入的数据储存在%ebp+0x8 中，并把值赋给了%eax

2.

```
8048b2c:    68 c0 97 04 08        push   $0x80497c0
8048b31:    50                    push   %eax
8048b32:    e8 f9 04 00 00        call   8049030 <strings_not_equal>
```

此处调用了<strings_not_equal>函数，易见此函数用于比较%eax, $0x80497c0 中的字符串是

否相等，并将返回值存到%eax 中

3.

```
8048b3a:   85 c0                test   %eax,%eax
8048b3c:   74 05                je     8048b43 <phase_1+0x23>
8048b3e:   e8 b9 09 00 00       call   80494fc <explode_bomb>
```

只有当%eax 中的值为 0，test 运算后%eax 值仍为 0，在下一步中跳转避开炸弹。因此调用字符比较函数后返回值必为 0，输入的字符串必须与$0x80497c0 中的相等。

4.

```
chenzhongyu@ubuntu:~/Desktop/lab2$ gdb bomb
```

开启调试

```
(gdb) b phase_1
Breakpoint 1 at 0x8048b26
```

设置断点

```
(gdb) print (char *) 0x80497c0
$1 = 0x80497c0 "Public speaking is very easy."
```

查找 0x80497c0 地址中的值即为密码。

5.

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
```

（二） Phase 2

1.

```
8048b5b:   e8 78 04 00 00       call   8048fd8 <read_six_numbers>
```

需要输入 6 个数字

2.

```
(gdb) b phase_2
Breakpoint 2 at 0x8048b50
(gdb) r
```

设置断点并运行

```
1 2 3 4 5 6
```

输入 6 个数字进行跟踪

3.

通过(gdb) nexti 向下运行若干步，以及(gdb) disas 查找当前运行到的位置，运行到：

```
=> 0x08048b63 <+27>:   cmpl   $0x1,-0x18(%ebp)
   0x08048b67 <+31>:   je     0x8048b6e <phase_2+38>
   0x08048b69 <+33>:   call   0x80494fc <explode_bomb>
```

查此地址所存的值，

```
(gdb) print *(int *) ($ebp-0x18)
$4 = 1
```

继续查找 4，8，12，16，20bit 后的地址的值，发现分别是 2，3，4，5，6，证明所输入的值储存在以($ebp-0x18)开头的位置。又当($ebp-0x18)中的值为 1 时，避开炸弹，那么第一个数为 1。

4.

```
8048b6e:    bb 01 00 00 00        mov    $0x1,%ebx
8048b73:    8d 75 e8              lea    -0x18(%ebp),%esi
8048b76:    8d 43 01              lea    0x1(%ebx),%eax
8048b79:    0f af 44 9e fc        imul   -0x4(%esi,%ebx,4),%eax
8048b7e:    39 04 9e              cmp    %eax,(%esi,%ebx,4)
8048b81:    74 05                 je     8048b88 <phase_2+0x40>
8048b83:    e8 74 09 00 00        call   80494fc <explode_bomb>
8048b88:    43                    inc    %ebx
8048b89:    83 fb 05              cmp    $0x5,%ebx
8048b8c:    7e e8                 jle    8048b76 <phase_2+0x2e>
```

翻译为

```
$ebx=1;
esi=ebp-0x18;//esi is the address of the first number
do
{
    $eax=$ebx+0x1;
    $eax*=$(esi+$ebx*4)//$eax*=a[$ebx]
    if($eax==$(esi+$ebx*4))
        do not explode;
    $ebx++;
}while($ebx<=0x5);
```

C 语言

```
i=1;
do
{
    temp=i+1;
    temp*=a[i-1];;
    if(temp==a[i])
        do not explode;
    i++;
}while(i<=5);
```

由于 a[0]=1，那么可以计算出来剩下 5 个数为 2，6，24，120，720。

5.

```
1 2 6 24 120 720
That's number 2.  Keep going!
```

（三） Phase 3

1.

```
8048bb0:    50                    push   %eax
8048bb1:    68 de 97 04 08        push   $0x80497de
8048bb6:    52                    push   %edx
8048bb7:    e8 a4 fc ff ff        call   8048860 <sscanf@plt>
8048bbc:    83 c4 20              add    $0x20,%esp
8048bbf:    83 f8 02              cmp    $0x2,%eax
8048bc2:    7f 05                 jg     8048bc9 <phase_3+0x31>
```

```
8048bc4:    e8 33 09 00 00          call   80494fc <explode_bomb>
```

在这里调用了 scanf()函数，返回输入参数的个数，并储存在%eax，而当$eax>2 时才避开炸弹，可知输入了 3 个参数，而且 scanf()需要一个匹配格式串，因此可以查看$0x80497de 中的值。

```
(gdb) print *(char *) 0x80497de
$16 = 37 '%'
(gdb) print *(char *) 0x80497df
$18 = 100 'd'
(gdb) print *(char *) 0x80497e0
$19 = 32 ' '
(gdb) print *(char *) 0x80497e1
$20 = 37 '%'
(gdb) print *(char *) 0x80497e2
$21 = 99 'c'
(gdb) print *(char *) 0x80497e3
$22 = 32 ' '
(gdb) print *(char *) 0x80497e4
$23 = 37 '%'
(gdb) print *(char *) 0x80497e5
$24 = 100 'd'
(gdb) print *(char *) 0x80497e6
$25 = 0 '\000'
```

那么格式串为“%d %c %d”，第一个和第三个是 int 型，第二个是 char 型。

2.

```
8048bc9:    83 7d f4 07             cmpl   $0x7,-0xc(%ebp)
8048bcd:    0f 87 b5 00 00 00       ja     8048c88 <phase_3+0xf0>
```

通过用和 Phase 2 中一样的手法，输入不同的参数，并查$(ebp-0xc)的值，发现其始终等于输入的第一个参数，而这里第一个参数 x 必须满足 0<=x<=7。

3.

```
8048bd3:    8b 45 f4                mov    -0xc(%ebp),%eax
8048bd6:    ff 24 85 e8 97 04 08    jmp    *0x80497e8(,%eax,4)
```

跳转到*0x80497e8(,%eax,4)这个地址，其中$eax∈[0, 7]，而每一个$eax（共 8 个）所对应的地址处，内容都相近（这里取第一个，当$eax==1 时）

```
8048be0:    b3 71                   mov    $0x71,%bl
8048be2:    81 7d fc 09 03 00 00    cmpl   $0x309,-0x4(%ebp)
8048be9:    0f 84 a0 00 00 00       je     8048c8f <phase_3+0xf7>
8048bef:    e8 08 09 00 00          call   80494fc <explode_bomb>
8048bf4:    e9 96 00 00 00          jmp    8048c8f <phase_3+0xf7>
```

通过同样的匹配方式，发现-0x4(%ebp)存储的是第三个参数，而第三个参数必须和 0x309 所对应的 777 相等，否则爆炸。而且发现，每一段相近内容最后都会跳转到 8048c8f 处，发现这是一个 switch 语句。跳转前，令$bl=0x71。

```
8048c8f:    3a 5d fb                cmp    -0x5(%ebp),%bl
8048c92:    74 05                   je     8048c99 <phase_3+0x101>
8048c94:    e8 63 08 00 00          call   80494fc <explode_bomb>
```

跳转后，当-0x5(%ebp)中的值与%bl 中的值相等时，避开炸弹。通过不断输入测试值、查-0x5(%ebp)的值，发现这是第二个参数，而且要与 0x71 对应，则为‘q’。另外还有 7 种组合是符合要求的。C 语言为：

```
int x1, x3;
char x2;
switc(x1)
{
case 0:if(x2==0x71&&x3==777) break;
case 1:if(x2==0x62&&x3==214) break;
case 2:if(x2==0x62&&x3==755) break;
case 3:if(x2==0x6b&&x3==251) break;
case 4:if(x2==0x6f&&x3==160) break;
case 5:if(x2==0x74&&x3==458) break;
case 6:if(x2==0x76&&x3==780) break;
case 7:if(x2==0x62&&x3==524) break;
default:explode;
}
```

4.

```
0 q 777
Halfway there!
```

（四） Phase 4

1.

```
8048cef:    50                  push   %eax
8048cf0:    68 08 98 04 08      push   $0x8049808
8048cf5:    52                  push   %edx
8048cf6:    e8 65 fb ff ff      call   8048860 <sscanf@plt>
8048cfb:    83 c4 10            add    $0x10,%esp
8048cfe:    83 f8 01            cmp    $0x1,%eax
8048d01:    75 06               jne    8048d09 <phase_4+0x29>
```

同样调用了 scanf()函数，先查看$0x8049808 中的格式串。

```
(gdb) print *(char *) (0x8049808)
$64 = 37 '%'
(gdb) print *(char *) (0x8049808+1)
$65 = 100 'd'
(gdb) print *(char *) (0x8049808+2)
$66 = 0 '\000'
```

格式串是”%d”，输入参数为 int 型。而且返回值必须为 1，否则跳转后爆炸，也证明了是一个参数。

2.

```
8048d03:    83 7d fc 00         cmpl   $0x0,-0x4(%ebp)
8048d07:    7f 05               jg     8048d0e <phase_4+0x2e>
8048d09:    e8 ee 07 00 00      call   80494fc <explode_bomb>
```

通过同样的查值，发现-0x4(%ebp)中的是所输入的参数，而且必须>0，否则爆炸。

3.

```
8048d11:   8b 45 fc             mov    -0x4(%ebp),%eax
8048d14:   50                   push   %eax
8048d15:   e8 86 ff ff ff       call    8048ca0 <func4>
8048d1a:   83 c4 10             add    $0x10,%esp
8048d1d:   83 f8 37             cmp    $0x37,%eax
8048d20:   74 05                je     8048d27 <phase_4+0x47>
8048d22:   e8 d5 07 00 00       call   80494fc <explode_bomb>
```

对输入的参数调用 fuc4，且返回值必须为 0x37。可知 func4 是一个运算的函数。

```
8048ca8:   8b 5d 08             mov    0x8(%ebp),%ebx
8048cab:   83 fb 01             cmp    $0x1,%ebx
8048cae:   7e 20                jle    8048cd0 <func4+0x30>
```

接下来跳转到：

```
8048cd0:   b8 01 00 00 00       mov    $0x1,%eax
```

发现 0x8(%ebp)中参数的值赋给%ebx，而当$ebx<=1 时，函数返回 1。这是函数的出口。

4.

```
8048cb3:   8d 43 ff             lea    -0x1(%ebx),%eax
8048cb6:   50                   push   %eax
8048cb7:   e8 e4 ff ff ff       call   8048ca0 <func4>
8048cbc:   89 c6                mov    %eax,%esi
```

对$ebx-1 调用 func4 函数，并把返回值储存在%esi 中。

```
8048cc1:   8d 43 fe             lea    -0x2(%ebx),%eax
8048cc4:   50                   push   %eax
8048cc5:   e8 d6 ff ff ff       call   8048ca0 <func4>
8048cca:   01 f0                add    %esi,%eax
```

对$ebx-2 调用 func4 函数，并把返回值返回到%eax 中，并且把%esi 中的值加到%eax 中。用 C 语言表达也就是

```
int func4(int x)
{
    if(x<=1)
        return 1;
    return func4(x-1)+func4(x-2);
}
```

这是一个以 f(0)=1, f(1)=1，开头的 Fibonacci 数列，解 fun4(x)=0x37 可以得 x=9

5.

```
9
So you got that one.  Try this one.
```

（五） Phase 5

1.

```
8048d34:   8b 5d 08             mov    0x8(%ebp),%ebx
8048d37:   83 c4 f4             add    $0xfffffff4,%esp
8048d3a:   53                   push   %ebx
8048d3b:   e8 d8 02 00 00       call   8049018 <string_length>
8048d40:   83 c4 10             add    $0x10,%esp
8048d43:   83 f8 06             cmp    $0x6,%eax
```

```
8048d46:   74 05                jе     8048d4d <phase_5+0x21>
8048d48:   e8 af 07 00 00       call   80494fc <explode_bomb>
```

通过不断输入值并对 0x8(%ebp)进行查值，发现输入的数据储存在这里。接下来对其中的值调用<string_length>函数，返回 string 的长度。当长度为 6 时，避开炸弹。那么密码是长度为 6 的字符串。

2.

```
8048d4d:   31 d2                xor    %edx,%edx
8048d4f:   8d 4d f8             lea    -0x8(%ebp),%ecx
8048d52:   be 20 b2 04 08       mov    $0x804b220,%esi
```

%edx 置 0，输入的数据赋值给%ecx，把$0x804b220 中的值赋给%esi，对此地址查值得

```
(gdb) print (char *) 0x804b220
$24 = 0x804b220 "isrveawhobpnutfg\260\001"
```

%esi 中存储这个字符串 s

3.

```
8048d57:   8a 04 1a             mov    (%edx,%ebx,1),%al
8048d5a:   24 0f                and    $0xf,%al
8048d5c:   0f be c0             movsbl %al,%eax
8048d5f:   8a 04 30             mov    (%eax,%esi,1),%al
8048d62:   88 04 0a             mov    %al,(%edx,%ecx,1)
8048d65:   42                   inc    %edx
8048d66:   83 fa 05             cmp    $0x5,%edx
8048d69:   7e ec                jle    8048d57 <phase_5+0x2b>
```

设数据存储在 c[6]中，C 语言表示为

```
i=0;
do
{
    int temp=c[i];
    temp&=0xf;
    c[i]=s[temp];
    i++;
}while(i<=5);
```

把 c[i]的低 4 位 ASCII 转换为 index，从模块串 s 中寻找对应的字符 s[index]，并对应赋值给 c[i]。如此对 6 个字符进行处理。

4.

```
8048d72:   68 0b 98 04 08       push   $0x804980b
8048d77:   8d 45 f8             lea    -0x8(%ebp),%eax
8048d7a:   50                   push   %eax
8048d7b:   e8 b0 02 00 00       call   8049030 <strings_not_equal>
8048d80:   83 c4 10             add    $0x10,%esp
8048d83:   85 c0                test   %eax,%eax
8048d85:   74 05                je     8048d8c <phase_5+0x60>
8048d87:   e8 70 07 00 00       call   80494fc <explode_bomb>
```

把处理好的字符串和$0x804980b 中的字符串，调用<string_not_equal>函数进行比对，只有当字符串相等，返回 0，使 test 结果仍为 0 时，才会跳转避开炸弹，因此两串必须相等。查

询$0x804980b 中的字符串。

```
(gdb) print (char *) 0x804980b
$26 = 0x804980b "giants"
```

5.

对于第一个字符 c[0]，要使 s[c[0]&0xf]==‘g’，那么查找模块串，可得 c[0]&0xf==15，那么只要 ASCII 码低 4 位为 1111（也就是 f）的字符均符合，而‘O’==0x4f，那么‘O’可以是第 1 个字符。同样的可以得到后 5 为字符。

6.

```
OPUKMA
Good work!  On to the next...
```

（六） Phase 6

1.

```
8048db3:    e8 20 02 00 00        call   8048fd8 <read_six_numbers>
```

调用读取 6 个数字的函数，可以知道读取的是 6 个数字。设置断点，输入 1 2 3 4 5 6，运行到此处，查看函数返回值%eax 中的值，

```
(gdb) x/6x $eax
0x804b810 <input_strings+400>:   0x20322031 0x20342033 0x00362035
0x00000000
0x804b820 <input_strings+416>:   0x00000000 0x00000000
(gdb) x/6x 0xbffff0c0
0xbffff0c0:    0xbffff1a4 0x00000000 0xbffff0e8 0x08049208
0xbffff0d0:    0x00000000 0x00007374
```

再运行到下一步，值从地址变为输入的数，

```
(gdb) x/6x 0xbffff0c0
0xbffff0c0:    0x00000001 0x00000002 0x00000003 0x00000004
0xbffff0d0:    0x00000005 0x00000006
```

确认了所输入的值为 6 个数字，并储存在 0xbffff0c0 开头的位置上。

2.

接下来阅读主体代码，发现有多次跳转，根据 jxx 指令的位置和跳转目的地，可以大致分为两个嵌套循环和两个单层循环，可以按顺序一个个处理。

3.

首先是一个嵌套的循环（为了节约空间，去掉了前面的地址和 ASCII 码）

xor %edi,%edi
add $0x10,%esp
lea 0x0(%esi),%esi
 	
lea -0x18(%ebp),%eax
mov (%eax,%edi,4),%eax
dec %eax
cmp $0x5,%eax
jbe 8048dd1 <phase_6+0x39>
//无符号比较大小，说明每个数都>0
call 80494fc <explode_bomb>
	edi=0;//index 初始设置
	
	do
	{
	$eax=ebp-0x18;
	$eax=$(eax+edi*4);
	$eax--;
	if($eax>5)
	explode;
	//6个数字都必须满足1<=x<=6

```
lea   0x1(%edi),%ebx
cmp   $0x5,%ebx
jg    8048dfc <phase_6+0x64>
lea   0x0(,%edi,4),%eax
mov   %eax,-0x38(%ebp)
lea   -0x18(%ebp),%esi
mov   -0x38(%ebp),%edx
mov   (%edx,%esi,1),%eax
cmp   (%esi,%ebx,4),%eax
jne   8048df6 <phase_6+0x5e>
call  80494fc <explode_bomb>
inc   %ebx
cmp   $0x5,%ebx
jle   8048de6 <phase_6+0x4e>
inc   %edi
cmp   $0x5,%edi
jle   8048dc0 <phase_6+0x28>
```

```
    $ebx=edi+1;
    if($ebx<=5)
    {
        $eax=edi*4+0;
        $(ebp-0x38)=$eax;
        $esi=ebp-0x18;
        do
        {
            $edx=$(ebp-0x38);
            $eax=$(edx+esi*1);
             if($eax==$(esi+ebx*4))
                explode;
            $ebx++;
        }while($ebx<=5);
        //每个数都互不相等
    }
    $edi++;
}while($edi<=5);
```

通过翻译成 C 语言，发现密码是 6 个[1, 6]的整数，且互不相等。设置断点，运行，输入测试数据 6 5 4 3 2 1 观察变量和内存的变化。

```
Breakpoint 2, 0x08048e02 in phase_6 ()
=> 0x08048e02 <phase_6+106>: 31 ff   xor    %edi,%edi
(gdb) set disassemble-next-line on
(gdb) ni
Breakpoint 3, 0x08048e04 in phase_6 ()
=> 0x08048e04 <phase_6+108>: 8d 4d e8   lea    -0x18(%ebp),%ecx
(gdb) info registers ebp
ebp            0xbffff0d8  0xbffff0d8
(gdb) ni
Breakpoint 5, 0x08048e07 in phase_6 ()
=> 0x08048e07 <phase_6+111>: 8d 45 d0   lea    -0x30(%ebp),%eax
(gdb)
Breakpoint 6, 0x08048e0a in phase_6 ()
=> 0x08048e0a <phase_6+114>: 89 45 c4   mov    %eax,-0x3c(%ebp)
(gdb) info registers eax ecx ebp
eax            0xbffff0a8  -1073745752
ecx            0xbffff0c0  -1073745728
ebp            0xbffff0d8  0xbffff0d8
(gdb) ni
Breakpoint 7, 0x08048e0d in phase_6 ()
=> 0x08048e0d <phase_6+117>: 8d 76 00   lea    0x0(%esi),%esi
(gdb)
Breakpoint 4, 0x08048e10 in phase_6 ()
=> 0x08048e10 <phase_6+120>: 8b 75 cc   mov    -0x34(%ebp),%esi
```

```
(gdb) info registers esi
esi            0xbffff0c0  -1073745728
(gdb) x/x $ebp-0x3c
0xbffff09c:    0xbffff0a8
```

发现$eax，$ecx 和 oxbffff09c 处的内容被更改了。接下来进入下一个循环。

4.

xor %edi,%edi	$edi=0;
lea -0x18(%ebp),%ecx	$ecx=ebp-0x18;
lea -0x30(%ebp),%eax	$eax=ebp-0x30;
mov %eax,-0x3c(%ebp)	$(ebp-0x3c)=$eax
lea 0x0(%esi),%esi	$esi=esi

以上是这个循环的准备环节。

```
8048e10:   8b 75 cc              mov    -0x34(%ebp),%esi
8048e13:   bb 01 00 00 00        mov    $0x1,%ebx
8048e18:   8d 04 bd 00 00 00 00  lea    0x0(,%edi,4),%eax
8048e1f:   89 c2                 mov    %eax,%edx
8048e21:   3b 1c 08              cmp    (%eax,%ecx,1),%ebx
8048e24:   7d 12                 jge    8048e38 <phase_6+0xa0>
8048e26:   8b 04 0a              mov    (%edx,%ecx,1),%eax
8048e29:   8d b4 26 00 00 00 00  lea    0x0(%esi,%eiz,1),%esi
8048e30:   8b 76 08              mov    0x8(%esi),%esi
8048e33:   43                    inc    %ebx
8048e34:   39 c3                 cmp    %eax,%ebx
8048e36:   7c f8                 jl     8048e30 <phase_6+0x98>
8048e38:   8b 55 c4              mov    -0x3c(%ebp),%edx
8048e3b:   89 34 ba              mov    %esi,(%edx,%edi,4)
8048e3e:   47                    inc    %edi
8048e3f:   83 ff 05              cmp    $0x5,%edi
8048e42:   7e cc                 jle    8048e10 <phase_6+0x78>
```

涉及到的最小内存是$ebp-0x3c，而大部分为内存变化，关系复杂，使用 gdb 查看循环前（0x8048e0d）和后（0x8048e44）的内存变化。

```
Breakpoint 9, 0x08048e0d in phase_6 ()
=> 0x08048e0d <phase_6+117>: 8d 76 00   lea    0x0(%esi),%esi
(gdb) x/3x $ebp-0x3c
0xbffff09c:    0xbffff0a8 0x00000010 0x0804b26c
(gdb) x/16x $ebp-0x3c
0xbffff09c:    0xbffff0a8 0x00000010 0x0804b26c 0xb7fbfc20
0xbffff0ac:    0xb7e6143f 0xbffff1a4 0x080497a0 0xbffff0d4
0xbffff0bc:    0x08048b37 0x00000006 0x00000005 0x00000004
0xbffff0cc:    0x00000003 0x00000002 0x00000001 0xbffff108
(gdb) c
Continuing.
Breakpoint 10, 0x08048e44 in phase_6 ()
=> 0x08048e44 <phase_6+172>: 8b 75 d0   mov    -0x30(%ebp),%esi
```

```
(gdb) x/16x $ebp-0x3c
0xbffff09c:    0xbffff0a8 0x00000010 0x0804b26c 0x0804b230
0xbffff0ac:    0x0804b23c 0x0804b248 0x0804b254 0x0804b260
0xbffff0bc:    0x0804b26c 0x00000006 0x00000005 0x00000004
0xbffff0cc:    0x00000003 0x00000002 0x00000001 0xbffff108
```

发现 0xbffff0a8 到 0xbffff0bc 的内存发生了变化。而 0xbffff0c0 到 0xbffff0d4 是输入的 6 个数字所在内存，接下来是可以把第二个循环变成 C 语言。设数字数组为*a，从 0xbffff0c0 到 0xbffff0d4，也就是*a = {6, 5, 4, 3, 2, 1}，而设会变化的内存为*p，从 0xbffff0a8 到 0xbffff0bc。

xor %edi,%edi
lea -0x18(%ebp),%ecx
lea -0x30(%ebp),%eax
mov %eax,-0x3c(%ebp)
lea 0x0(%esi),%esi
	$edi=0;
	$ecx=ebp-0x18;//$ecx=0xbffff0c0
	$eax=ebp-0x30;//$eax=0xbffff0a8
	$(ebp-
	0x3c)=$eax;//$eax->0xbffff0bc
	$esi=$esi+0;

mov -0x34(%ebp),%esi
mov $0x1,%ebx
lea 0x0(,%edi,4),%eax
mov %eax,%edx
 	
cmp (%eax,%ecx,1),%ebx
jge 8048e38 <phase_6+0xa0>
mov (%edx,%ecx,1),%eax
lea 0x0(%esi,%eiz,1),%esi
mov 0x8(%esi),%esi
inc %ebx
cmp %eax,%ebx
jl 8048e30 <phase_6+0x98>
 	
mov -0x3c(%ebp),%edx
mov %esi,(%edx,%edi,4)
inc %edi
cmp $0x5,%edi
jle 8048e10 <phase_6+0x78>
	do
	{
	$esi=M(0xbffff0c0)=0x0804b26c;
	$ebx=1;
	$eax=4*$edi;
	$eax=$edx=4*$edi;
	if($ebx<a[$eax/4])
	{
	do
	{
	$eax=a[$eax/4];
	$esi=$esi+0;
	$esi=M($esi+0x8);
	$ebx++;
	}while($ebx<$eax)
	}
	$edx=M(0xbffff0a0)= 0xbffff0a8
	f[$edi]=$esi;//f=0xbffff0a8
	$edi++;
	}while($edi<=5)

对整个单层循环单步调试，查看内存信息。

```
(gdb) x/x 0x804b26c+0x8
0x804b274 <node1+8>:  0x0804b260
(gdb) x/x 0x804b268
0x804b268 <node2+8>:  0x0804b254
(gdb) x/x 0x804b25c
0x804b25c <node3+8>:  0x0804b248
(gdb) x/x 0x804b250
0x804b250 <node4+8>:  0x0804b23c
(gdb) x/x 0x804b244
```

```
0x804b244 <node5+8>:  0x0804b230
(gdb) x/x 0x804b230
0x804b230 <node6>:0x000001b0
(gdb) x/x 0x804b23c
0x804b23c <node5>:0x000000d4
(gdb) x/x 0x804b248
0x804b248 <node4>:0x000003e5
(gdb) x/x 0x804b254
0x804b254 <node3>:0x0000012d
(gdb) x/x 0x804b260
0x804b260 <node2>:0x000002d5
(gdb) x/x 0x804b26c
0x804b26c <node1>:0x000000fd
(gdb) x/x 0x804b264
0x804b264 <node2+4>:  0x00000002
(gdb) x/x 0x804b26c
0x804b26c <node1>:0x000000fd
```

根据以上信息可有以下内存寻址模式：

地址	值	地址	值	地址	值
0xbffff0a4	0x0804b26c	0x804b230	0x1b0		
0xbffff0a8				0x804b25c	0x804b248
0xbffff0ac		0x804b23c	0xd4	0x804b260	0x2d5
0xbffff0b0					
0xbffff0b4		0x804b244	0x804b230	0x804b268	0x804b254
0xbffff0b8		0x804b248	0x3e5	0x804b26c	0xfd
0xbffff0bc					
		0x804b250	0x804b23c	0x804b274	0x804b260
		0x804b254	0x12d		

$edi=0	$eax=a[edi]=6, $esi=0x804b26c, $esi=M($esi+0x8)	
$edi=1	$esi=M(0x804b26c+0x8)=0x804b260	$eax=2
$edi=2	$esi=M(0x804b26c+0x8)=0x804b254	$eax=3
$edi=3	$esi=M(0x804b254+0x8)=0x804b248	$eax=4
$edi=4	$esi=M(0x804b248+0x8)=0x804b23c	$eax=5
$edi=5	$esi=M(0x804b23c+0x8)=0x804b230	$eax=6
$edi=6	$eax==6	end
M(0xbffff0a8+$edi*4)=$esi，在$eax==$ebx 时跳出循环，当为 1 时不进入内循环，直接存入 0x804b26c		

5.

接下来是下一个单层循环，同样像上一个循环那样，在循环前（0x08048e52）和循环后（0x08048e58）查看内存变化。

地址	值	地址	值	地址	值
0xbffff0a4	0x0804b230	0x804b230	0x1b0		

0xbffff0a8	0x0804b230	0x804b238	0x804b23c	0x804b25c	0x804b260
0xbffff0ac	0x0804b23c	0x804b23c	0xd4	0x804b260	0x2d5
0xbffff0b0	0x0804b248				
0xbffff0b4	0x0804b254	0x804b244	0x804b248	0x804b268	0x804b26c
0xbffff0b8	0x0804b260	0x804b248	0x3e5	0x804b26c	0xfd
0xbffff0bc	0x0804b26c				
		0x804b250	0x804b254	0x804b274	0x804b260
		0x804b254	0x12d		

mov %esi,-0x34(%ebp)
mov $0x1,%edi
lea -0x30(%ebp),%edx
mov (%edx,%edi,4),%eax
mov %eax,0x8(%esi)
mov %eax,%esi
inc %edi
cmp $0x5,%edi
jle 8048e52 <phase_6+0xba>
	$esi=0x804b230;//->0xbffff0a4
	$edi=1;
	$edx=0xbffff0a8
	do
	{
	$eax=p[$edi];
	$esi+0x8=$eax;
	$esi=$eax;
	$edi++;
	}while($edi<=5)

这次循环在上一次的基础上，继续更改内存。然后是下一个循环。

6.

movl $0x0,0x8(%esi)
mov -0x34(%ebp),%esi
xor %edi,%edi
lea 0x0(%esi,%eiz,1),%esi
mov 0x8(%esi),%edx
mov (%esi),%eax
cmp (%edx),%eax
jge 8048e7e <phase_6+0xe6>
call 80494fc <explode_bomb>
mov 0x8(%esi),%esi
inc %edi
cmp $0x4,%edi
jle 8048e70 <phase_6+0xd8>
	$esi+0x8=0;//0x804b26c=0
	$esi=M(0xbffff0a4)//=0x804b26c;
	$edi=0;
	$esi+=0;
	do
	{
	$edx=M($esi+0x8);
	$eax=M($esi);
	if($eax<M($edx))
	explode;
	$esi=M($esi+0x8);
	$edi++;
	}while($edi<=4)

这个循环，在测试根据输入的 6 个数字排好序的地址中存放的数字——输入的 6 个数字要使 0x804b230 到 0x804b26c 共 6 个地址中的数字按降序排列。而输入的 6 个数通过上述的寻址方式，去改变这 6 个地址在 0xbffff0a8 到 0xbffff0bc 中存储的顺序，再控制 6 个地址存放在 0x804b 开始的地址。

7.

可以推算，

0x804b248->0x3e5->4	0x804b260->0x2d5->2	0x804b230->0x1b0->6
0x804b254->0x12d->3	0x804b26c->0xfd->1	0x804b23c->0xd4->5

为 4 2 6 3 1 5 的顺序。

8.

```
4 2 6 3 1 5
Congratulations! You've defused the bomb!
[Inferior 1 (process 12224) exited normally]
```

（七） Secret phase

1.

在 main()函数里，每个关卡通过之后都会进入<phase_dufused>函数，而且函数中还会有<secret_phase>的入口，那么先分析<phase_dufused>函数。

```
push   %ebx
cmpl   $0x6,0x804b480
jne    804959f <phase_defused+0x73>
```

2.

在<phase_dufused>入口设置断点，在每一次调用<phase_dufused>函数后（每次通过后），查看 0x804b480 中的值。发现，其中储存的值从 1 开始，不断自增，到最后为 6，但没有进入 secret_phase。

```
push   %eax
push   $0x8049d03
push   $0x804b770
call   8048860 <sscanf@plt>
add    $0x10,%esp
cmp    $0x2,%eax
jne    8049592 <phase_defused+0x66>
```

发现由于$eax!=2，跳过了<secret_phase>的调用。而%eax 是 sccanf 函数的返回值，调用 sccanf 函数前还引用了$0x8049d03 和$0x804b770 这两个参数，可以猜测其中一个是自动输入的值，另外一个是格式串。那么查看其中的值，

```
(gdb) x/s 0x8049d03
0x8049d03: "%d %s"
(gdb) x/s 0x804b770
0x804b770 <input_strings+240>:   "9"
```

那么应该输入的是一个数字和一个字符串，才能触发 secret_phase，但是收到的是一个"9"（程序自带的），而 phase_4 的答案是 9，由于 phase_4 的格式串是%d，所以多余输入也不会影响的，那么应该在 phase_4 多输入一个字符串。

3.

```
push   $0x8049d09
push   %ebx
call   8049030 <strings_not_equal>
add    $0x10,%esp
test   %eax,%eax
jne    8049592 <phase_defused+0x66>
```

在这里，和之前调用<strings_not_equal>一样，必须引用的两个字符串参数相等，才不会跳转，否则就跳过了 secret_phase 的触发。而%ebx 中的是所多输入的字符串，那么$0x8049d09 中的是匹配串，查看它的值。

```
(gdb) x/s 0x8049d09
0x8049d09: "austinpowers"
```

那么 phase_4 应输入 9 austinpowers

```
9 austinpowers
So you got that one.  Try this one.
OPUKMA
Good work!  On to the next...
4 2 6 3 1 5
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

成功触发 secret_phase，那么接下来分析 secret_phase。

4.

```
push   %ebx
call   80491fc <read_line>
push   $0x0
push   $0xa
push   $0x0f
push   %eax
call   80487f0 <__strtol_internal@plt>
```

第一次调用函数，说明应该输入一个字符串。在第二个函数前设置断点，运行，随意输入"abc"，发现第二个函数引用的%eax 即为"abc"。Google 可得该函数可以将一个 string 转换成一个 long int，储存在%eax 中。

5.

```
mov    %eax,%ebx
lea    -0x1(%ebx),%eax
mp    $0x3e8,%eax
jbe    8048f14 <secret_phase+0x2c>
call   80494fc <explode_bomb>
```

那么函数返回值必须<=0x3e8+1（1001），而由于 jbe，那么返回值-1>=0，因此返回值为 1-1001 的一个整数。

6.

```
push   %ebx
push   $0x804b320
call   8048e94 <fun7>
add    $0x10,%esp
cmp    $0x7,%eax
je     8048f2f <secret_phase+0x47>
call   80494fc <explode_bomb>
```

可见对<__strtol_internal@plt>返回值调用<fun7>后，结果需等于 7，否则 explode，那么分析<fun7>。

7.

```
mov    0xc(%ebp),%eax
test   %edx,%edx
jne    8048eb0 <fun7+0x1c>
mov    $0xffffffff,%eax
jmp    8048ee2 <fun7+0x4e>
```

只有当传入的参数%edx 为 0 时，才会直接结束函数，返回$0xffffffff（-1）。用 gdb 来来查看

传入的值，发现是传入的地址，那么是不会出现$edx=0 的情况的，所以不影响，继续分析。

8.

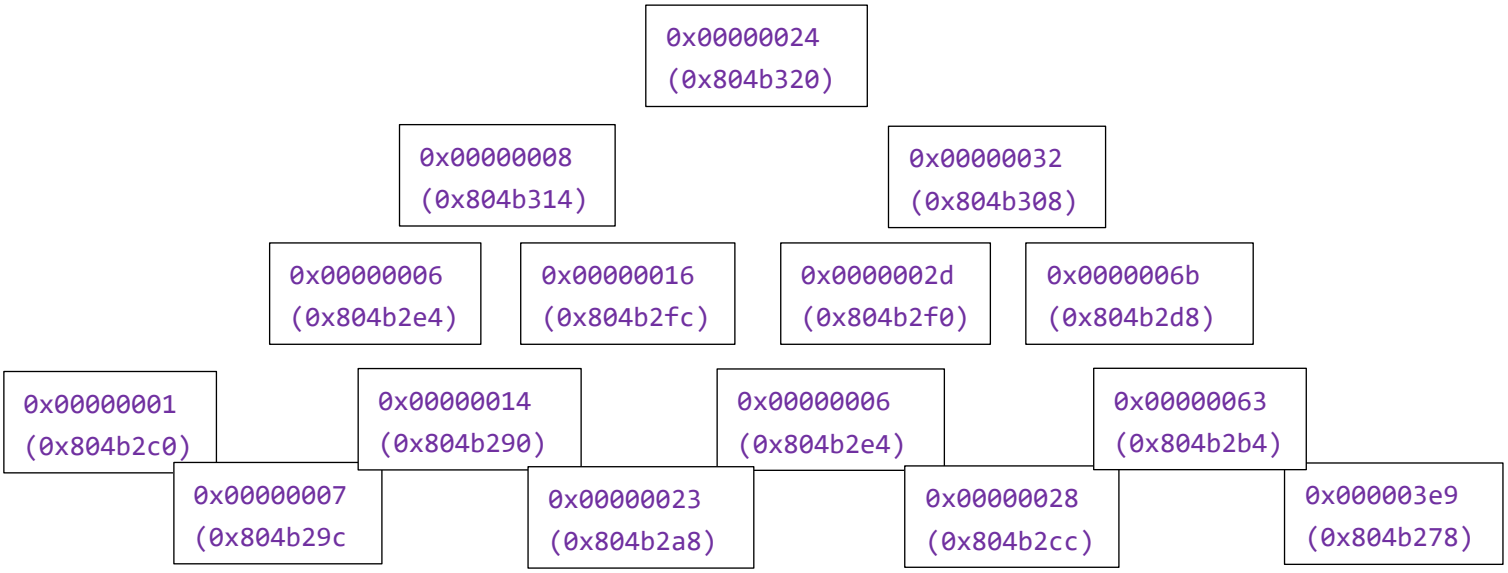
lea 0x0(%esi,%eiz,1),%esi
cmp (%edx),%eax
jge 8048ec5 <fun7+0x31>
add $0xfffffff8,%esp
push %eax
mov 0x4(%edx),%eax
push %eax
call 8048e94 <fun7>
add %eax,%eax
jmp 8048ee2 <fun7+0x4e>
cmp (%edx),%eax
je 8048ee0 <fun7+0x4c>
add $0xfffffff8,%esp
push %eax
mov 0x8(%edx),%eax
push %eax
call 8048e94 <fun7>
add %eax,%eax
inc %eax
jmp 8048ee2 <fun7+0x4e>
	if($eax<M($edx))
	{
	fun7($eax,M($edx+0x4));
	$eax*=2;
	}
	
	else
	if($eax==M($edx))
	$eax=0;
	
	else
	{
	fun7($eax,M($edx+0x8));
	$eax=$eax*2+1;
	}

这是有两个分支的递归，那么传入参数后，每次都会分开两条路，并不断分下去，直到$eax==M($edx)时，返回$eax=0，不再分下去，形成如同二叉树一般的结构。接下来在函数的分支处设置断点，并查看符合上述条件的可以递归的地址，以及其中的值，直到某个地址处的两个分支均为 0 时结束。

```
(gdb) x/3x $edx
0x804b320 <n1>:   0x00000024 0x0804b3140x0804b308
(gdb) x/3x $edx
0x804b320 <n1>:   0x00000024 0x0804b314 0x0804b308
(gdb) x/3x *($edx+0x4)
0x804b314 <n21>:  0x00000008 0x0804b2e4 0x0804b2fc
(gdb) x/3x *($edx+0x8)
0x804b308 <n22>:  0x00000032 0x0804b2f0 0x0804b2d8
(gdb) x/3x *(*($edx+0x4)+0x4)
0x804b2e4 <n31>:  0x00000006 0x0804b2c0 0x0804b29c
(gdb) x/3x *(*($edx+0x4)+0x8)
0x804b2fc <n32>:  0x00000016 0x0804b290 0x0804b2a8
(gdb) x/3x *(*($edx+0x8)+0x8)
0x804b2d8 <n34>:  0x0000006b 0x0804b2b4 0x0804b278
(gdb) x/3x *(*($edx+0x8)+0x4)
0x804b2f0 <n33>:  0x0000002d 0x0804b2cc 0x0804b284
(gdb) x/3x *(*(*($edx+0x4)+0x4)+0x4)
0x804b2c0 <n41>:  0x00000001 0x00000000 0x00000000
```

```
(gdb) x/3x *(*(*($edx+0x8)+0x4)+0x4)
0x804b2cc <n45>:  0x00000028 0x00000000 0x00000000
(gdb) x/3x *(*(*($edx+0x4)+0x8)+0x4)
0x804b290 <n43>:  0x00000014 0x00000000 0x00000000
(gdb) x/3x *(*(*($edx+0x4)+0x4)+0x8)
0x804b29c <n42>:  0x00000007 0x00000000 0x00000000
(gdb) x/3x *(*(*($edx+0x8)+0x4)+0x8)
0x804b284 <n46>:  0x0000002f 0x00000000 0x00000000
(gdb) x/3x *(*(*($edx+0x8)+0x8)+0x4)
0x804b2b4 <n47>:  0x00000063 0x00000000 0x00000000
(gdb) x/3x *(*(*($edx+0x4)+0x8)+0x8)
0x804b2a8 <n44>:  0x00000023 0x00000000 0x00000000
(gdb) x/3x *(*(*($edx+0x8)+0x8)+0x8)
0x804b278 <n48>:  0x000003e9 0x00000000 0x00000000
```

根据上述查询，可以总结出这样的二叉树结构，



而递归的结果必须为 7，那么可以推算 7=((0*2+1)*2+1)*2+1，而且只有这种分解方式了，所以应该有$eax>0x24，$eax>0x6b，$eax==0x3e9，即 1001，那么最终有$eax==0x3e9(1001)。

9.

```
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
[Inferior 1 (process 12337) exited normally]
```

三、 心得体会

（一） 需要熟练掌握 gdb 调试功能，包括设置断点、查内存的值、查寄存器的值等。

（二） 最好能有把汇编码翻译为高级语言的能力。分析时把汇编码转换为高级语言，有助于理解。

（三） 分析内存变化时，可以画出内存表格，一个地址对应一个值，并按顺序列出，实体化后有助于分析变化及取值。

（四） 分析汇编码要抓重点。分析时应该从 main()函数开始，并按照运行顺序去分析，不要盲目去随便阅读其他地方的代码，或者不按顺序阅读。

（五） 遇到函数时，可以观察其函数名，一般都会表示该函数功能（所以就知道了），那么只需要用 gdb 调试查值，获得引用的值、返回的值，就可以了，根本不需要去阅读该函数的汇编码。

（六） 知之为知之，不知 Google 知。