CSAPP BOMB writeup

可以使用gdb 的 peda 可以增加一点信息量，便于分析。

先来看 main函数

首先的补充一个知识是 x64 前六个参数依次保存在RDI, RSI, RDX, RCX, R8和 R9中，如果还有更多的参数的话才会保存在栈上。

所以主函数为 `main(argc,*argv[])` argc 存放在 rdi ，argv存放在 rsi

```
0x0000000000401076 <+0>:    push   rbx
0x0000000000401077 <+1>:    cmp    edi,0x1          #argc 与 1比较
0x000000000040107a <+4>:    jne    0x40108c <main+22> # 若不同则跳转 如果为1 就 fp = stdin （标准输入，从控制台输入）
0x000000000040107c <+6>:    mov    rax,QWORD PTR [rip+0x2042ad]        # 0x605330 <stdin@@GLIBC_2.2.5>
0x0000000000401083 <+13>:   mov    QWORD PTR [rip+0x2049a6],rax        # 0x605a30 <infile>
0x000000000040108a <+20>:   jmp    0x4010ef <main+121>
0x000000000040108c <+22>:   mov    rbx,rsi
0x000000000040108f <+25>:   cmp    edi,0x2      # 若argc =2 那么就打开 argv[1]这个文件 当作输入
0x0000000000401092 <+28>:   jne    0x4010ce <main+88>
0x0000000000401094 <+30>:   mov    rdi,QWORD PTR [rsi+0x8]
0x0000000000401098 <+34>:   mov    esi,0x403024
0x000000000040109d <+39>:   call   0x400e60 <fopen@plt>
0x00000000004010a2 <+44>:   mov    QWORD PTR [rip+0x204987],rax        # 0x605a30 <infile>
0x00000000004010a9 <+51>:   test   rax,rax
0x00000000004010ac <+54>:   jne    0x4010ef <main+121>      # 判断是否打开成功 成功就跳转到 0x4010ef 不成功就不跳转 会导致退出
0x00000000004010ae <+56>:   mov    rcx,QWORD PTR [rbx+0x8]
0x00000000004010b2 <+60>:   mov    rdx,QWORD PTR [rbx]
0x00000000004010b5 <+63>:   mov    esi,0x403026
0x00000000004010ba <+68>:   mov    edi,0x1
0x00000000004010bf <+73>:   call   0x400e50 <__printf_chk@plt>
0x00000000004010c4 <+78>:   mov    edi,0x8
0x00000000004010c9 <+83>:   call   0x400e80 <exit@plt>
0x00000000004010ce <+88>:   mov    rdx,QWORD PTR [rsi]
0x00000000004010d1 <+91>:   mov    esi,0x403043
0x00000000004010d6 <+96>:   mov    edi,0x1
0x00000000004010db <+101>:  mov    eax,0x0
0x00000000004010e0 <+106>:  call   0x400e50 <__printf_chk@plt>
0x00000000004010e5 <+111>:  mov    edi,0x8
0x00000000004010ea <+116>:  call   0x400e80 <exit@plt>
0x00000000004010ef <+121>:  call   0x401b10                 # 这个函数是用来初始化的，对后面 每一层的函数都会有影响
0x00000000004010f4 <+126>:  mov    edi,0x4030c8
0x00000000004010f9 <+131>:  call   0x400d10 <puts@plt>      # 输出0x4030c8 下面的puts都类似作用
0x00000000004010fe <+136>:  mov    edi,0x403100
0x0000000000401103 <+141>:  call   0x400d10 <puts@plt>
0x0000000000401108 <+146>:  call   0x401fd0 <read_line>
0x000000000040110d <+151>:  mov    rdi,rax
0x0000000000401110 <+154>:  call   0x401252 <phase1>        # 我们主要是分析phase1~6这几个函数
0x0000000000401115 <+159>:  call   0x402250 <phase_defused>
0x000000000040111a <+164>:  mov    edi,0x403130
0x000000000040111f <+169>:  call   0x400d10 <puts@plt>
0x0000000000401124 <+174>:  call   0x401fd0 <read_line>
0x0000000000401129 <+179>:  mov    rdi,rax
0x000000000040112c <+182>:  call   0x4012a9 <phase2>
0x0000000000401131 <+187>:  call   0x402250 <phase_defused>
0x0000000000401136 <+192>:  mov    edi,0x40305d
0x000000000040113b <+197>:  call   0x400d10 <puts@plt>
0x0000000000401140 <+202>:  call   0x401fd0 <read_line>
0x0000000000401145 <+207>:  mov    rdi,rax
0x0000000000401148 <+210>:  call   0x401378 <phase3>
0x000000000040114d <+215>:  call   0x402250 <phase_defused>
0x0000000000401152 <+220>:  mov    edi,0x40307b
0x0000000000401157 <+225>:  call   0x400d10 <puts@plt>
0x000000000040115c <+230>:  call   0x401fd0 <read_line>
0x0000000000401161 <+235>:  mov    rdi,rax
0x0000000000401164 <+238>:  call   0x401515 <phase4>
0x0000000000401169 <+243>:  call   0x402250 <phase_defused>
```

```
0x000000000040116e <+248>:  mov    edi,0x40308a
0x0000000000401173 <+253>:  call   0x400d10 <puts@plt>
0x0000000000401178 <+258>:  call   0x401fd0 <read_line>
0x000000000040117d <+263>:  mov    rdi,rax
0x0000000000401180 <+266>:  call   0x40168d <phase5>
0x0000000000401185 <+271>:  call   0x402250 <phase_defused>
0x000000000040118a <+276>:  mov    edi,0x4030a5
0x000000000040118f <+281>:  call   0x400d10 <puts@plt>
0x0000000000401194 <+286>:  call   0x401fd0 <read_line>
0x0000000000401199 <+291>:  mov    rdi,rax
0x000000000040119c <+294>:  call   0x401768 <phase6>
0x00000000004011a1 <+299>:  call   0x402250 <phase_defused>
0x00000000004011a6 <+304>:  mov    eax,0x0
0x00000000004011ab <+309>:  pop    rbx
0x00000000004011ac <+310>:  ret
```

------------------------------ phase1 ------------------------------

第一层很简单，直接来看主函数

```
gdb-peda$ disas phase1
Dump of assembler code for function phase1:
   0x0000000000401252 <+0>:     sub    rsp,0x8
   0x0000000000401256 <+4>:     mov    esi,0x6059a0
   0x000000000040125b <+9>:     call   0x4011c2 <strings_not_equal>
   0x0000000000401260 <+14>:    test   eax,eax
   0x0000000000401262 <+16>:    je     0x401269 <phase1+23>
   0x0000000000401264 <+18>:    call   0x401e40 <explode_bomb>
   0x0000000000401269 <+23>:    add    rsp,0x8
   0x000000000040126d <+27>:    ret
End of assembler dump.
```

前面提到过 64位系统 函数的前两个参数是 edi 和 esi `strings_not_equal` 就是比较edi和esi值 就是 输入的字符串（edi）和 0x6059a0（esi）的比较

相当于 `strings_not_equal(edi,esi)`

我们在0x40125B处 下断点

然后看一下 0x6059a0（esi） 内容,就为 key

```
gdb-peda$ b *0x40125B
Breakpoint 8 at 0x40125b
gdb-peda$ r
Welcome to my nasty little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
123456
gdb-peda$ x/s 0x6059a0
0x6059a0:   "Courage is fear holding on a minute longer."
```

这一题中的`strings_not_equal`函数就是比较两个字符串是否相同，相同就返回0就跳过bomb。

------------------------------ phase2 ------------------------------

```
Dump of assembler code for function phase2:
   0x00000000004012a9 <+0>: push   rbx
   0x00000000004012aa <+1>: sub    rsp,0x20
   0x00000000004012ae <+5>: mov    rax,QWORD PTR fs:0x28
   0x00000000004012b7 <+14>:    mov    QWORD PTR [rsp+0x18],rax
   0x00000000004012bc <+19>:    xor    eax,eax
   0x00000000004012be <+21>:    mov    rsi,rsp
   0x00000000004012c1 <+24>:    call   0x401e80 <read_six_numbers>
   0x00000000004012c6 <+29>:    mov    eax,DWORD PTR [rip+0x2046c0]  # 0x60598c  peda自动会帮你计算好 rip+0x2046c0 的值 就是 0x60598c
                                                                          这里就是 0x60598c 的值与 输入的第一个数字比较
   0x00000000004012cc <+35>:    cmp    DWORD PTR [rsp],eax      # rsp 位置保存着输入的 数字 当调试到这个位置的时候
                                                                  可以用命令 x/6wx $rsp 查看输入的参数
   0x00000000004012cf <+38>:    jne    0x4012dd <phase2+52>     # 第一个数字与0x60598c 比较 然后 0x60598c 的值为 2
                                                                  可以通过 x/wx 0x60598c 获得
   0x00000000004012d1 <+40>:    mov    eax,DWORD PTR [rip+0x2046b1]        # 0x605988
   0x00000000004012d7 <+46>:    cmp    DWORD PTR [rsp+0x4],eax
   0x00000000004012db <+50>:    je     0x4012e2 <phase2+57>     #与上面一块相同，第二个参数与0x605988的数字比较  这个数字为6
   0x00000000004012dd <+52>:    call   0x401e40 <explode_bomb>
   0x00000000004012e2 <+57>:    mov    ebx,0x2              #ebx当作一个计数器 这里的一个循环 相当于 for(ebx=2;ebx<=5;ebx++)
   0x00000000004012e7 <+62>:    jmp    0x40130a <phase2+97> #进入循环
   0x00000000004012e9 <+64>:    movsxd rdx,ebx
   0x00000000004012ec <+67>:    lea    ecx,[rbx-0x2]    #ecx = ebx -2
   0x00000000004012ef <+70>:    movsxd rcx,ecx
   0x00000000004012f2 <+73>:    lea    eax,[rbx-0x1]   #eax = ebx -1
   0x00000000004012f5 <+76>:    cdqe
   0x00000000004012f7 <+78>:    mov    eax,DWORD PTR [rsp+rax*4]  # 第rax个参数 这里*4是因为一个int型数字 占 4字节
   0x00000000004012fa <+81>:    add    eax,DWORD PTR [rsp+rcx*4]    #相当于 array[ebx-2]+array[ebx-1]
   0x00000000004012fd <+84>:    cmp    DWORD PTR [rsp+rdx*4],eax    # 相当于 array[ebx-2]+array[ebx-1] = array[ebx]
                                                                    所以这里的循环是第三位开始 每一位都为前两位的值相加
                                                                    结果为 2 6 8 14 22 36
   0x0000000000401300 <+87>:    je     0x401307 <phase2+94>
   0x0000000000401302 <+89>:    call   0x401e40 <explode_bomb>
   0x0000000000401307 <+94>:    add    ebx,0x1
   0x000000000040130a <+97>:    cmp    ebx,0x5             # ebx计数器 是否大于5
   0x000000000040130d <+100>:   jle    0x4012e9 <phase2+64> #若ebx大于5就进入下一步
   0x000000000040130f <+102>:   mov    rax,QWORD PTR [rsp+0x18]
   0x0000000000401314 <+107>:   xor    rax,QWORD PTR fs:0x28    #下面是判断溢出的，可以忽略
   0x000000000040131d <+116>:   je     0x401324 <phase2+123>
   0x000000000040131f <+118>:   call   0x400d40 <__stack_chk_fail@plt>
   0x0000000000401324 <+123>:   add    rsp,0x20
   0x0000000000401328 <+127>:   pop    rbx
   0x0000000000401329 <+128>:   ret
End of assembler dump.
```

根据上面的分析，可以得到

```
array[6]={2,6,?,?,?,?}

for(ebx =2;ebx<=5;ebx++){
    array[ebx]=array[ebx-1]+array[ebx-2];
}
```

下面是`read_six_numbers`的函数

```
Dump of assembler code for function read_six_numbers:
   0x0000000000401e80 <+0>: lea    rax,[rsi+0x14]
   0x0000000000401e84 <+4>: sub    rsp,0x8
   0x0000000000401e88 <+8>: lea    rcx,[rsi+0x4]
   0x0000000000401e8c <+12>:    lea    r9,[rsi+0xc]
   0x0000000000401e90 <+16>:    lea    r8,[rsi+0x8]
   0x0000000000401e94 <+20>:    mov    rdx,rsi
   0x0000000000401e97 <+23>:    push   rax
   0x0000000000401e98 <+24>:    lea    rax,[rsi+0x10]
   0x0000000000401e9c <+28>:    mov    esi,0x4039f0     # 0x4039f0 位置保存着 format 使用命令x/s 0x4039f0 读取 为 "%d %d %d %d %d %d"
   0x0000000000401ea1 <+33>:    push   rax
   0x0000000000401ea2 <+34>:    xor    eax,eax
   0x0000000000401ea4 <+36>:    call   0x400e30 <__isoc99_sscanf@plt> # int sscanf( const char *buffer, const char *format,
                                                                                [argument]...); 从buffer 里以format读取
   0x0000000000401ea9 <+41>:    cmp    eax,0x5      #sscanf 返回值为读取的参数个数 判断是否大于 5
   0x0000000000401eac <+44>:    pop    rdx
   0x0000000000401ead <+45>:    pop    rcx
   0x0000000000401eae <+46>:    jle    0x401eb5 <read_six_numbers+53>
   0x0000000000401eb0 <+48>:    add    rsp,0x8
   0x0000000000401eb4 <+52>:    ret
   0x0000000000401eb5 <+53>:    call   0x401e40 <explode_bomb>
End of assembler dump.
```

操作如下

```
gdb-peda$ x/s 0x4039f0
0x4039f0:   "%d %d %d %d %d %d"
gdb-peda$ x/wx 0x60598c
0x60598c:   0x00000002
gdb-peda$ x/wx 0x605988
0x605988:   0x00000006
gdb-peda$ x/wx 0x60598c
0x60598c:   0x00000002
gdb-peda$ x/wx 0x605988
0x605988:   0x00000006
```

------------------------------ phase3 ------------------------------

```
Dump of assembler code for function phase3:
   0x0000000000401378 <+0>: sub    rsp,0x18
   0x000000000040137c <+4>: mov    rax,QWORD PTR fs:0x28
   0x0000000000401385 <+13>:    mov    QWORD PTR [rsp+0x8],rax
   0x000000000040138a <+18>:    xor    eax,eax
   0x000000000040138c <+20>:    lea    rcx,[rsp+0x4]
   0x0000000000401391 <+25>:    mov    rdx,rsp
   0x0000000000401394 <+28>:    mov    esi,0x4039fc
   0x0000000000401399 <+33>:    call   0x400e30 <__isoc99_sscanf@plt>   # x/s 0x4039fc      0x4039fc:   "%d %d"
   0x000000000040139e <+38>:    cmp    eax,0x1                      #判断参数数量是否大于1
   0x00000000004013a1 <+41>:    jg     0x4013a8 <phase3+48>         #若参数数量只有1就bomb 否则进入 0x04013a8
   0x00000000004013a3 <+43>:    call   0x401e40 <explode_bomb>
   0x00000000004013a8 <+48>:    cmp    DWORD PTR [rsp],0x7          #前面提到过rsp存的就是输入的第一个值  [rsp+4]为第二个
   0x00000000004013ac <+52>:    ja     0x401418 <phase3+160>        #这里判断第一个参数是否大于7 大于7 就跳转到bomb
   0x00000000004013ae <+54>:    mov    eax,DWORD PTR [rsp]
   0x00000000004013b1 <+57>:    jmp    QWORD PTR [rax*8+0x403170]   #这里是一个switch 选择分值 分支表 保存在 0x403170 开始的区域,
                                                                     使用命令 x/16x 0x403170 来查看 这里的分支表 是依次跳到下面的
   0x00000000004013b8 <+64>:    mov    edi,0x3
   0x00000000004013bd <+69>:    call   0x40132a <getnumber>     #getnumber 操作 会传入一个参数 就是 edi
                                                                 执行的就是 getnumber(edi) 的操作 这里为 getnumber(3)
   0x00000000004013c2 <+74>:    jmp    0x401422 <phase3+170>
   0x00000000004013c4 <+76>:    mov    edi,0x2
   0x00000000004013c9 <+81>:    call   0x40132a <getnumber>     #getnumber(2)
   0x00000000004013ce <+86>:    jmp    0x401422 <phase3+170>
   0x00000000004013d0 <+88>:    mov    edi,0x1
   0x00000000004013d5 <+93>:    call   0x40132a <getnumber>     #getnumber(1)
   0x00000000004013da <+98>:    jmp    0x401422 <phase3+170>
   0x00000000004013dc <+100>:   mov    edi,0x6
   0x00000000004013e1 <+105>:   call   0x40132a <getnumber>     #getnumber(6)
   0x00000000004013e6 <+110>:   jmp    0x401422 <phase3+170>
   0x00000000004013e8 <+112>:   mov    edi,0x8
   0x00000000004013ed <+117>:   call   0x40132a <getnumber>     #getnumber(8)
   0x00000000004013f2 <+122>:   jmp    0x401422 <phase3+170>
   0x00000000004013f4 <+124>:   mov    edi,0x7
   0x00000000004013f9 <+129>:   call   0x40132a <getnumber>     #getnumber(7)
   0x00000000004013fe <+134>:   jmp    0x401422 <phase3+170>
   0x0000000000401400 <+136>:   mov    edi,0x3
   0x0000000000401405 <+141>:   call   0x40132a <getnumber>     #getnumber(3)
   0x000000000040140a <+146>:   jmp    0x401422 <phase3+170>
   0x000000000040140c <+148>:   mov    edi,0x9
   0x0000000000401411 <+153>:   call   0x40132a <getnumber>     #getnumber(9)
   0x0000000000401416 <+158>:   jmp    0x401422 <phase3+170>
   0x0000000000401418 <+160>:   call   0x401e40 <explode_bomb>
   0x000000000040141d <+165>:   mov    eax,0x0
   0x0000000000401422 <+170>:   cmp    eax,DWORD PTR [rsp+0x4]  #eax是getnumber() 的返回值 与 输入的第二个参数相比较
   0x0000000000401426 <+174>:   je     0x40142d <phase3+181>
   0x0000000000401428 <+176>:   call   0x401e40 <explode_bomb>
   0x000000000040142d <+181>:   mov    rax,QWORD PTR [rsp+0x8]
   0x0000000000401432 <+186>:   xor    rax,QWORD PTR fs:0x28    #下面是判断溢出的，可以忽略
   0x000000000040143b <+195>:   je     0x401442 <phase3+202>
   0x000000000040143d <+197>:   call   0x400d40 <__stack_chk_fail@plt>
   0x0000000000401442 <+202>:   add    rsp,0x18
   0x0000000000401446 <+206>:   ret
End of assembler dump.
```

通过上面的分析，phase3 要求输入两个参数，我记做 var1 和 var2

switch参数会根据 var1 的值 （当然 var1 要小于等与7） 来选择getnumber(edi)里的参数

```
+------+---------------+
| var1 | getnumber(edi)|
|------+---------------|
|  0   |  3            |
|  1   |  2            |
|  2   |  1            |
|  3   |  6            |
|  4   |  8            |
|  5   |  7            |
|  6   |  3            |
|  7   |  9            |
+------+---------------+
```

getnumber返回值与var2 比较 是否相同

所以 var1可以为 0 ~ 7的任意数字 然后 在0x401422 下断点比较var2的值就可以了

分析一下getnumber的流程

```
Dump of assembler code for function getnumber:
   0x000000000040132a <+0>:     push   r12
   0x000000000040132c <+2>:     push   rbp
   0x000000000040132d <+3>:     mov    r12d,edi
   0x0000000000401330 <+6>:     push   rbx
   0x0000000000401331 <+7>:     mov    ebp,0x20
   0x0000000000401336 <+12>:    call   0x400ec0 <rand@plt>
   0x000000000040133b <+17>:    mov    ecx,eax     #ecx=rand()
   0x000000000040133d <+19>:    mov    ebx,ecx      #ebx = ecx
   0x000000000040133f <+21>:    xor    ebx,r12d     #r12d=edi传入的参数 这里操作为 ebx^传入参数
   0x0000000000401342 <+24>:    call   0x400ec0 <rand@plt>
   0x0000000000401347 <+29>:    movzx  ecx,al       #注意这里取的是al 就 相当于 rand()&0xff
   0x000000000040134a <+32>:    imul   ecx,ebx      #ecx = ebx ^ al
   0x000000000040134d <+35>:    sub    ebp,0x1      # ebp = ebp -1
   0x0000000000401350 <+38>:    jne    0x40133d <getnumber+19> #若ebp不等于0就跳转到上面循环
                                                    # 好了 上面就是循环0x20次 ecx = (ecx ^ argv[0])*(rand()&0xff)
   0x0000000000401352 <+40>:    mov    eax,ecx
   0x0000000000401354 <+42>:    mov    edx,0x21195767
   0x0000000000401359 <+47>:    imul   edx
   0x000000000040135b <+49>:    pop    rbx
   0x000000000040135c <+50>:    pop    rbp
   0x000000000040135d <+51>:    mov    eax,edx
   0x000000000040135f <+53>:    mov    edx,ecx
   0x0000000000401361 <+55>:    sar    eax,0x8
   0x0000000000401364 <+58>:    sar    edx,0x1f
   0x0000000000401367 <+61>:    sub    eax,edx
   0x0000000000401369 <+63>:    imul   eax,eax,0x7bc    #这里一块的过程比较复杂，熟悉了就会知道这里是取mod的操作 为ecx % 0x7bc
   0x000000000040136f <+69>:    pop    r12
   0x0000000000401371 <+71>:    sub    ecx,eax
   0x0000000000401373 <+73>:    mov    eax,ecx
   0x0000000000401375 <+75>:    ret
End of assembler dump.
```

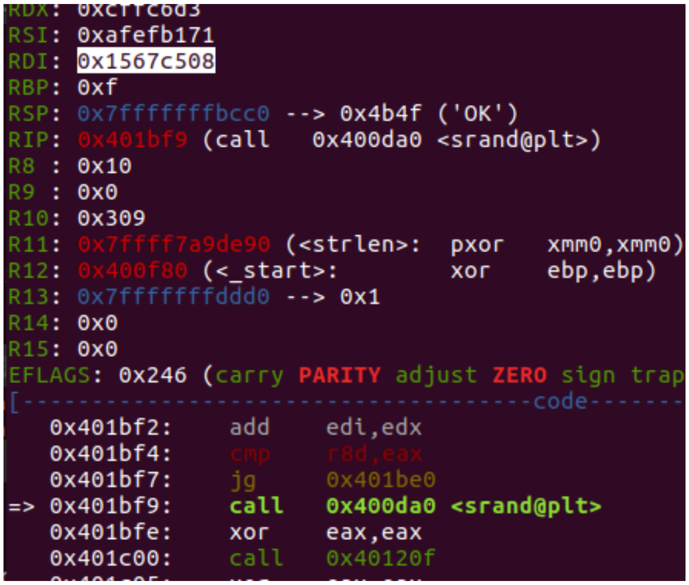
通过这样的分析，可以推断出 这里的操作其实为

```
ecx = rand()
while(循环0x20次)
    ecx = (ecx ^ argv[0])*(rand()&0xff)
```

```
out = ecx % 0x7bc
```

因为这里的rand()是伪随机

在初始化的那个函数里有用到srand 可以在 0x401BF9 断点 查看 edi的值 就是 seed的值



```
RDX: 0xcffc6d3
RSI: 0xafefb171
RDI: 0x1567c508
RBP: 0xf
RSP: 0x7fffffffbcc0 --> 0x4b4f ('OK')
RIP: 0x401bf9 (call   0x400da0 <srand@plt>)
R8 : 0x10
R9 : 0x0
R10: 0x309
R11: 0x7ffff7a9de90 (<strlen>:  pxor   xmm0,xmm0)
R12: 0x400f80 (<_start>:        xor    ebp,ebp)
R13: 0x7fffffffddd0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap
[-----------------------------------code-------
   0x401bf2:    add    edi,edx
   0x401bf4:    cmp    r8d,eax
   0x401bf7:    jg     0x401be0
=> 0x401bf9:    call   0x400da0 <srand@plt>
   0x401bfe:    xor    eax,eax
   0x401c00:    call   0x40120f
```

初始化的过程之中用到过很多次 rand() 所以这里很难推理,这里就不推理了。

因为rand()次数是相同的，所以这里的rand()值是可以固定下来的，所以最后的结果是不会变的

```
gdb-peda$ x/16x 0x403170
0x403170:   0x004013b8  0x00000000  0x004013c4  0x00000000
0x403180:   0x004013d0  0x00000000  0x004013dc  0x00000000
0x403190:   0x004013e8  0x00000000  0x004013f4  0x00000000
0x4031a0:   0x00401400  0x00000000  0x0040140c  0x00000000
```

------------------------------ phase4 ------------------------------

```
Dump of assembler code for function phase4:
   0x0000000000401515 <+0>: sub    rsp,0x18
   0x0000000000401519 <+4>: mov    rax,QWORD PTR fs:0x28
   0x0000000000401522 <+13>:    mov    QWORD PTR [rsp+0x8],rax
   0x0000000000401527 <+18>:    xor    eax,eax
   0x0000000000401529 <+20>:    lea    rcx,[rsp+0x4]
   0x000000000040152e <+25>:    mov    rdx,rsp
   0x0000000000401531 <+28>:    mov    esi,0x4039fc
   0x0000000000401536 <+33>:    call   0x400e30 <__isoc99_sscanf@plt>   #"%d %d" 同 phase3
   0x000000000040153b <+38>:    cmp    eax,0x1
   0x000000000040153e <+41>:    jg     0x401545 <phase4+48>     判断是否参数为2个（是否大于1）
   0x0000000000401540 <+43>:    call   0x401e40 <explode_bomb>
   0x0000000000401545 <+48>:    mov    esi,DWORD PTR [rsp+0x4]
   0x0000000000401549 <+52>:    mov    edi,0x6053a0
   0x000000000040154e <+57>:    call   0x4014db <fun4>      # fun4(0x6053a0, var2) 0x6053a0是一个地址,可以观察下面dump下来的地址
   0x0000000000401553 <+62>:    cmp    eax,DWORD PTR [rsp]  # fun4的结果和 var1来相比较 图方便可以直接在这里下端点取值
   0x0000000000401556 <+65>:    je     0x40155d <phase4+72> # 不同则bomb
   0x0000000000401558 <+67>:    call   0x401e40 <explode_bomb>
   0x000000000040155d <+72>:    mov    rax,QWORD PTR [rsp+0x8]
   0x0000000000401562 <+77>:    xor    rax,QWORD PTR fs:0x28    #下面是判断溢出的，可以忽略
   0x000000000040156b <+86>:    je     0x401572 <phase4+93>
   0x000000000040156d <+88>:    call   0x400d40 <__stack_chk_fail@plt>
   0x0000000000401572 <+93>:    add    rsp,0x18
   0x0000000000401576 <+97>:    ret
End of assembler dump.
```

操作

```
gdb-peda$ x/s 0x4039fc
0x4039fc:   "%d %d"
```

看了下这个函数也是比较简单的，就是 读取两个数 记作 var1，var2

执行判断 fun4(0x6053a0,var2)==var1

主要是要分析fun4这个函数的操作

```
Dump of assembler code for function fun4:
   0x00000000004014db <+0>: sub    rsp,0x8
   0x00000000004014df <+4>: mov    rdx,QWORD PTR [rdi+0x8]      # 注意是QWORD类型 8字节的 取
   0x00000000004014e3 <+8>: test   rdx,rdx                      # 若[rdi+0x8]非0则跳转
   0x00000000004014e6 <+11>:    jne    0x4014f3 <fun4+24>
   0x00000000004014e8 <+13>:    cmp    QWORD PTR [rdi+0x10],0x0 # 若[rdi+0x8]非0则跳转
   0x00000000004014ed <+18>:    jne    0x4014f3 <fun4+24>       #相当于([rdi+0x8]!=0||[rdi+0x10]!=0)
   0x00000000004014ef <+20>:    mov    eax,DWORD PTR [rdi]      #两者都不满足则直接 取这个值返回
   0x00000000004014f1 <+22>:    jmp    0x401510 <fun4+53>
   0x00000000004014f3 <+24>:    test   sil,0x1               #sil是rsi的低8位寄存器  来test 最后1bit是否为0
   0x00000000004014f7 <+28>:    je     0x401506 <fun4+43>       #相当于 if(sil & 1)
   0x00000000004014f9 <+30>:    mov    rdi,QWORD PTR [rdi+0x10]
   0x00000000004014fd <+34>:    sar    esi,1
   0x00000000004014ff <+36>:    call   0x4014db <fun4>          #fun4([rdi+0x10],esi>>1)  其实为 fun4(node->rightaddr,esi>>1)
   0x0000000000401504 <+41>:    jmp    0x401510 <fun4+53>
   0x0000000000401506 <+43>:    sar    esi,1
   0x0000000000401508 <+45>:    mov    rdi,rdx
   0x000000000040150b <+48>:    call   0x4014db <fun4>          #fun4([rdi+0x8], esi>>1) 其实为 fun4(node->leftaddr,esi>>1)
   0x0000000000401510 <+53>:    add    rsp,0x8
   0x0000000000401514 <+57>:    ret
End of assembler dump.
```

这里可以看作fun4的第一个参数为一个结构体

```
struct node{
    dword value;
    dword *leftaddr;
    dword *rightaddr;
}
```

整体上就是进行；这样的一个操作，就是

```
if ( node->leftaddr || node->rightaddr )
{
    if ( var & 1 )
        result = fun4(node->rightaddr, var >> 1);
    else
        result = fun4(node->leftaddr,  var >> 1);
}
else
{
    result = node->value;
}
return result;
```

是来根据 输入的 var2 来搜索一个数字的过程 我在这里输入var2 = 5 返回的fun4为 576

若有兴趣可以自己先定义结构体写出这样的一段寻址过程，然后写解决方案。

下面是 0x6053a0 的值

```
gdb-peda$ x/100x 0x6053a0
0x6053a0:   0x00000000000002ce  0x00000000006053b8
0x6053b0:   0x00000000006053d0  0x0000000000000257
```

```
0x6053c0:   0x00000000006053e8  0x0000000000605400
0x6053d0:   0x00000000000001c1  0x0000000000605418
0x6053e0:   0x0000000000605430  0x00000000000002cb
0x6053f0:   0x0000000000605448  0x0000000000605460
0x605400:   0x0000000000000105  0x0000000000605478
0x605410:   0x0000000000605490  0x00000000000000f4
0x605420:   0x00000000006054a8  0x00000000006054c0
0x605430:   0x00000000000001f3  0x00000000006054d8
0x605440:   0x00000000006054f0  0x00000000000003a2
0x605450:   0x0000000000605508  0x0000000000605520
0x605460:   0x0000000000000361  0x0000000000605538
0x605470:   0x0000000000605550  0x00000000000000e5
0x605480:   0x0000000000605568  0x0000000000605580
0x605490:   0x0000000000000088  0x0000000000605598
0x6054a0:   0x00000000006055b0  0x0000000000000033
0x6054b0:   0x00000000006055c8  0x00000000006055e0
0x6054c0:   0x000000000000033b  0x00000000006055f8
0x6054d0:   0x0000000000605610  0x0000000000000107
0x6054e0:   0x0000000000605628  0x0000000000605640
0x6054f0:   0x0000000000000221  0x0000000000605658
0x605500:   0x0000000000605670  0x0000000000000200
0x605510:   0x0000000000605688  0x00000000006056a0
0x605520:   0x0000000000000199  0x00000000006056b8
0x605530:   0x00000000006056d0  0x0000000000000199
0x605540:   0x00000000006056e8  0x0000000000605700
0x605550:   0x0000000000000327  0x0000000000605718
0x605560:   0x0000000000605730  0x000000000000023d
0x605570:   0x0000000000605748  0x0000000000605760
0x605580:   0x000000000000013a  0x0000000000605778
0x605590:   0x0000000000605790  0x0000000000000282
0x6055a0:   0x00000000006057a8  0x00000000006057c0
0x6055b0:   0x0000000000000142  0x00000000006057d8
0x6055c0:   0x00000000006057f0  0x0000000000000077
0x6055d0:   0x0000000000605808  0x0000000000605820
0x6055e0:   0x0000000000000114  0x0000000000605838
0x6055f0:   0x0000000000605850  0x0000000000000031
0x605600:   0x0000000000605868  0x0000000000605880
0x605610:   0x00000000000002c7  0x0000000000605898
0x605620:   0x00000000006058b0  0x00000000000002c9
0x605630:   0x00000000006058c8  0x00000000006058e0
0x605640:   0x0000000000000363  0x00000000006058f8
0x605650:   0x0000000000605910  0x00000000000002d6
0x605660:   0x0000000000605928  0x0000000000605940
0x605670:   0x0000000000000220  0x0000000000605958
0x605680:   0x0000000000605970  0x0000000000000249
0x605690:   0x0000000000000000  0x0000000000000000
0x6056a0:   0x000000000000002d  0x0000000000000000
0x6056b0:   0x0000000000000000  0x00000000000002c7
........后面还有一段忽略了
```

处理一下这些数字 可以看的清楚点，很明显就是一个结构体!

```
0x6053a0: 0x0002ce 0x6053b8 0x6053d0
0x6053b8: 0x000257 0x6053e8 0x605400
0x6053d0: 0x0001c1 0x605418 0x605430
0x6053e8: 0x0002cb 0x605448 0x605460
0x605400: 0x000105 0x605478 0x605490
0x605418: 0x0000f4 0x6054a8 0x6054c0
0x605430: 0x0001f3 0x6054d8 0x6054f0
0x605448: 0x0003a2 0x605508 0x605520
0x605460: 0x000361 0x605538 0x605550
0x605478: 0x0000e5 0x605568 0x605580
0x605490: 0x000088 0x605598 0x6055b0
0x6054a8: 0x000033 0x6055c8 0x6055e0
0x6054c0: 0x00033b 0x6055f8 0x605610
0x6054d8: 0x000107 0x605628 0x605640
```

```
0x6054f0: 0x000221 0x605658 0x605670
0x605508: 0x000200 0x605688 0x6056a0
0x605520: 0x000199 0x6056b8 0x6056d0
0x605538: 0x000199 0x6056e8 0x605700
0x605550: 0x000327 0x605718 0x605730
0x605568: 0x00023d 0x605748 0x605760
0x605580: 0x00013a 0x605778 0x605790
0x605598: 0x000282 0x6057a8 0x6057c0
0x6055b0: 0x000142 0x6057d8 0x6057f0
0x6055c8: 0x000077 0x605808 0x605820
0x6055e0: 0x000114 0x605838 0x605850
0x6055f8: 0x000031 0x605868 0x605880
0x605610: 0x0002c7 0x605898 0x6058b0
0x605628: 0x0002c9 0x6058c8 0x6058e0
0x605640: 0x000363 0x6058f8 0x605910
0x605658: 0x0002d6 0x605928 0x605940
0x605670: 0x000220 0x605958 0x605970
0x605688: 0x000249 0x000000 0x000000
.............
.............
```

------------------------------ phase5 ------------------------------

```
Dump of assembler code for function phase5:
   0x000000000040168d <+0>: push   r13
   0x000000000040168f <+2>: push   r12
   0x0000000000401691 <+4>: push   rbp
   0x0000000000401692 <+5>: push   rbx
   0x0000000000401693 <+6>: sub    rsp,0x8
   0x0000000000401697 <+10>:    mov    r12,rdi
   0x000000000040169a <+13>:    call   0x4011ad <string_length>     #string_length(rdi) 判断长度，这个函数在c里也包含了
   0x000000000040169f <+18>:    cmp    eax,0x6                      #判断输入字符串长度是否为6  我这里记作字符串为 char str[6]
   0x00000000004016a2 <+21>:    je     0x4016a9 <phase5+28>         #若不是则bomb
   0x00000000004016a4 <+23>:    call   0x401e40 <explode_bomb>
   0x00000000004016a9 <+28>:    mov    ebx,0x0                  #ebx = 0
   0x00000000004016ae <+33>:    mov    ebp,0x0                  #ebp = 0
   0x00000000004016b3 <+38>:    jmp    0x4016e2 <phase5+85>
   0x00000000004016b5 <+40>:    movsxd r13,ebx
   0x00000000004016b8 <+43>:    add    r13,r12              #r12为字符串的首地址 这两部为了获得 str[ebx] 的地址
   0x00000000004016bb <+46>:    movzx  eax,BYTE PTR [r13+0x0] #取str[ebx]
   0x00000000004016c0 <+51>:    sub    eax,0x61              # str[ebx]-0x61('a') < 0x19(26) 说明这里是在判断 str[] 要都是 a~z的字符
   0x00000000004016c3 <+54>:    cmp    al,0x19
   0x00000000004016c5 <+56>:    jbe    0x4016cc <phase5+63>
   0x00000000004016c7 <+58>:    call   0x401e40 <explode_bomb>
   0x00000000004016cc <+63>:    movsx  eax,BYTE PTR [r13+0x0]
   0x00000000004016d1 <+68>:    sub    eax,0x61             # 取str[ebx]-0x61('a') 的值 就是 'a'为0 'b'为1
   0x00000000004016d4 <+71>:    cdqe
   0x00000000004016d6 <+73>:    movsx  eax,BYTE PTR [rax+0x605370] #0x605370上有一个数组，取第str[ebx]-0x61('a')的值
   0x00000000004016dd <+80>:    add    ebp,eax                      #然后依次相加，加到寄存器ebp上
   0x00000000004016df <+82>:    add    ebx,0x1
   0x00000000004016e2 <+85>:    cmp    ebx,0x5                  # 这里是ebx做计数器功能
   0x00000000004016e5 <+88>:    jle    0x4016b5 <phase5+40>     # while(ebx <= 5)
   0x00000000004016e7 <+90>:    cmp    ebp,DWORD PTR [rip+0x203c77]     # 0x605364  经过6次循环之后然后和 0x605364位置上的数进行比较
   0x00000000004016ed <+96>:    je     0x4016f4 <phase5+103>
   0x00000000004016ef <+98>:    call   0x401e40 <explode_bomb>
   0x00000000004016f4 <+103>:   add    rsp,0x8
   0x00000000004016f8 <+107>:   pop    rbx
   0x00000000004016f9 <+108>:   pop    rbp
   0x00000000004016fa <+109>:   pop    r12
   0x00000000004016fc <+111>:   pop    r13
   0x00000000004016fe <+113>:   ret
End of assembler dump.
```

经过上面的分析应该大致可以清楚 是 输入一个 a~z的长度为 6 的字符串，然后转化为0~25

在基址为0x605370的数组上 寻找，6个值依次相加

最后与0x605364上的数组进行比较。

几个地址指向的信息如下

```
gdb-peda$ x/26bx 0x605370
0x605370:   0x0d    0x16    0x1a    0x09    0x0f    0x0c    0x02    0x05
0x605378:   0x0a    0x01    0x07    0x18    0x12    0x15    0x0b    0x13
0x605380:   0x11    0x0e    0x19    0x10    0x04    0x06    0x17    0x08
0x605388:   0x03    0x14
gdb-peda$ x/bx 0x605364
0x605364:   0x3f
```

伪代码如下

```
int num[26]={0x0d,0x16,0x1a,0x09,0x0f,0x0c,0x02,0x05,0x0a,0x01,0x07,0x18,0x12,0x15,
                0x0b,0x13,0x11,0x0e,0x19,0x10,0x04,0x06,0x17,0x08,0x03,0x14};
int result=0x3f;
char str[6]=?????;
int sum=0;
for(int i=0;i<6;i++){
    if(str[i]不为a~z){
        bomb();
    }
    sum+=num[str[i]-0x61];
}
if(sum!=result){
    bomb()
}
```

------------------------------ phase6 ------------------------------

国际惯例 先看phase6函数，这一层的循环比较多

```
Dump of assembler code for function phase6:
   0x0000000000401768 <+0>: push   r12
   0x000000000040176a <+2>: push   rbp
   0x000000000040176b <+3>: push   rbx
   0x000000000040176c <+4>: sub    rsp,0x60
   0x0000000000401770 <+8>: mov    rax,QWORD PTR fs:0x28        #检测溢出用
   0x0000000000401779 <+17>:    mov    QWORD PTR [rsp+0x58],rax
   0x000000000040177e <+22>:    xor    eax,eax
   0x0000000000401780 <+24>:    mov    rsi,rsp
   0x0000000000401783 <+27>:    call   0x401e80 <read_six_numbers>  #同第二层 读取6个数字
   #-------------一轮循环-----------
   0x0000000000401788 <+32>:    mov    ebp,0x0
   0x000000000040178d <+37>:    jmp    0x4017c9 <phase6+97>     #跳到+97 意思为 最外层循环为 for(ebp=0;ebp<=5;ebp++)
   0x000000000040178f <+39>:    movsxd rax,ebp
   0x0000000000401792 <+42>:    mov    eax,DWORD PTR [rsp+rax*4]    #rsp指向的是6个数字的首地址 我记作 char num[6],后面表现的清楚一点
                                                                    取值num[ebp]
   0x0000000000401795 <+45>:    sub    eax,0x1
   0x0000000000401798 <+48>:    cmp    eax,0x5
   0x000000000040179b <+51>:    jbe    0x4017a2 <phase6+58>         #num[ebp]-1>5 就bomb 代表 num[ebp] <= 6
   0x000000000040179d <+53>:    call   0x401e40 <explode_bomb>
   0x00000000004017a2 <+58>:    lea    r12d,[rbp+0x1]
   0x00000000004017a6 <+62>:    mov    ebx,r12d                 #ebx = ebp + 1 这里开始到 0x4017c4 范围是里面的一个小循环
                                                                    意思为 for(ebx=ebp+1;ebp<=5;ebp++)
   0x00000000004017a9 <+65>:    jmp    0x4017c1 <phase6+89>
   0x00000000004017ab <+67>:    movsxd rax,ebp
   0x00000000004017ae <+70>:    movsxd rdx,ebx
   0x00000000004017b1 <+73>:    mov    edi,DWORD PTR [rsp+rdx*4]    #取num[rdx]相当于 num[ebx]
   0x00000000004017b4 <+76>:    cmp    DWORD PTR [rsp+rax*4],edi    #比较num[ebx]与num[rax(ebp)]是否相同
   0x00000000004017b7 <+79>:    jne    0x4017be <phase6+86>
   0x00000000004017b9 <+81>:    call   0x401e40 <explode_bomb>
   0x00000000004017be <+86>:    add    ebx,0x1
   0x00000000004017c1 <+89>:    cmp    ebx,0x5                  #ebx<=5
   0x00000000004017c4 <+92>:    jle    0x4017ab <phase6+67>
   0x00000000004017c6 <+94>:    mov    ebp,r12d
   0x00000000004017c9 <+97>:    cmp    ebp,0x5                 #ebp<=5
   0x00000000004017cc <+100>:   jle    0x40178f <phase6+39>
   #-------------二轮循环-----------
   0x00000000004017ce <+102>:   mov    esi,0x0
   0x00000000004017d3 <+107>:   jmp    0x4017f8 <phase6+144>    #最外围也是一个 for(esi=0;esi<=5;esi++)的循环
   0x00000000004017d5 <+109>:   mov    rdx,QWORD PTR [rdx+0x8]  #这个就是0x605160的操作 可以看下下面dump下来的内存,
                                                                 会感觉到，就是依次往后移
   0x00000000004017d9 <+113>:   add    eax,0x1
   0x00000000004017dc <+116>:   jmp    0x4017e8 <phase6+128>
   0x00000000004017de <+118>:   mov    eax,0x1
   0x00000000004017e3 <+123>:   mov    edx,0x605160            #x/24wx 0x605160查看该地址内的 数值
   0x00000000004017e8 <+128>:   movsxd rcx,esi
   0x00000000004017eb <+131>:   cmp    eax,DWORD PTR [rsp+rcx*4]    for(eax=1;eax<num[rcx];eax++)
   0x00000000004017ee <+134>:   jl     0x4017d5 <phase6+109>        #若 eax< num[rcx]就走出循环
   0x00000000004017f0 <+136>:   mov    QWORD PTR [rsp+rcx*8+0x20],rdx   #上面循环结束之后 将rdx值储存到 rsp+0x20为基址的数组之中。
                                                                         我记作 new_num[]
   0x00000000004017f5 <+141>:   add    esi,0x1
   0x00000000004017f8 <+144>:   cmp    esi,0x5              #esi <= 5
   0x00000000004017fb <+147>:   jle    0x4017de <phase6+118>
   #-------------三轮循环-----------
   0x00000000004017fd <+149>:   mov    rbx,QWORD PTR [rsp+0x20]     #rbx = &new_num[0]
   0x0000000000401802 <+154>:   mov    rcx,rbx                      #rcx = &new_num[0]
   0x0000000000401805 <+157>:   mov    eax,0x1
   0x000000000040180a <+162>:   jmp    0x40181e <phase6+182>        #for(eax=1;eax<=5;eax++)
   0x000000000040180c <+164>:   movsxd rdx,eax
   0x000000000040180f <+167>:   mov    rdx,QWORD PTR [rsp+rdx*8+0x20]   #new_num[edx(eax)]
   0x0000000000401814 <+172>:   mov    QWORD PTR [rcx+0x8],rdx      #注意这个PTR 是取地址 *(new_num[rcx]+0x8)=new_num[edx]
                                                                        等同于 *(new_num[eax-1]+0x8)=new_num[eax]
   0x0000000000401818 <+176>:   add    eax,0x1
   0x000000000040181b <+179>:   mov    rcx,rdx                      # rcx=edx
   0x000000000040181e <+182>:   cmp    eax,0x5
   0x0000000000401821 <+185>:   jle    0x40180c <phase6+164>
   0x0000000000401823 <+187>:   mov    QWORD PTR [rcx+0x8],0x0      # &new_num[6]=0
   #-------------四轮循环-----------
   0x000000000040182b <+195>:   mov    ebp,0x0
   0x0000000000401830 <+200>:   jmp    0x401848 <phase6+224>        #for(ebp =0;ebp<=4;ebp++)
   0x0000000000401832 <+202>:   mov    rax,QWORD PTR [rbx+0x8]      #rax = new_num[rbx+1]
   0x0000000000401836 <+206>:   mov    eax,DWORD PTR [rax]          #eax = *new_num[rbx+1]
   0x0000000000401838 <+208>:   cmp    DWORD PTR [rbx],eax          #*new_num[rbx] 与 *new_num[rbx+1]做比较
   0x000000000040183a <+210>:   jge    0x401841 <phase6+217>        #如果*new_num[rbx+1]比较小那么 bomb
   0x000000000040183c <+212>:   call   0x401e40 <explode_bomb>
   0x0000000000401841 <+217>:   mov    rbx,QWORD PTR [rbx+0x8]      #rbx = *(rbx + 0x8)
   0x0000000000401845 <+221>:   add    ebp,0x1
   0x0000000000401848 <+224>:   cmp    ebp,0x4
   0x000000000040184b <+227>:   jle    0x401832 <phase6+202>
   #-------------溢出检测-----------
   0x000000000040184d <+229>:   mov    rax,QWORD PTR [rsp+0x58]
   0x0000000000401852 <+234>:   xor    rax,QWORD PTR fs:0x28
   0x000000000040185b <+243>:   je     0x401862 <phase6+250>
   0x000000000040185d <+245>:   call   0x400d40 <__stack_chk_fail@plt>
   0x0000000000401862 <+250>:   add    rsp,0x60
   0x0000000000401866 <+254>:   pop    rbx
   0x0000000000401867 <+255>:   pop    rbp
   0x0000000000401868 <+256>:   pop    r12
   0x000000000040186a <+258>:   ret
End of assembler dump.
```

根据第一轮的循环
可以写出下面的伪代码

```
for ( i = 0; i <= 5; ++i )
{
    if ( (num[i] - 1) > 5 )     // num[i] 为 1~6
```

```
        bomb();
    for ( j = i + 1; j <= 5; ++j )
    {
        if ( num[i] == num[j] )
            bomb();
    }
}
```

可以得出一个结论就是要输入一个1~6的数而且不重复

第二轮中

```
0x605160:   0x000003b8  0x00000001  0x00605170  0x00000000
0x605170:   0x0000a5db  0x00000002  0x00605180  0x00000000
0x605180:   0x0000c619  0x00000003  0x00605190  0x00000000
0x605190:   0x00008286  0x00000004  0x006051a0  0x00000000
0x6051a0:   0x0000293e  0x00000005  0x006051b0  0x00000000
0x6051b0:   0x00008ecb  0x00000006  0x00000000  0x00000000
```

看起来第二轮的操作比较混乱
这里我们可以总结一下，就是最外围一个for 使用 num[i]的值 来判断位置

里面的一个for 是为了选择第0x605160[num[i]]的地址

具体数字来看能够清楚一些。其实也就是一个排序的过程。

例如 num[6]={6,5,4,3,2,1};

new_num[6]={0x6051b0,0x6051a0,0x605190,0x605180,0x605170,0x605160};

第三轮操作 先看一下new_num[]的地址是否和上面所说的一样，结果是一样的。

```
gdb-peda$ x/10x $rsp+0x20
0x7fffffffdc90: 0x00000000006051b0  0x00000000006051a0
0x7fffffffdca0: 0x0000000000605190  0x0000000000605180
0x7fffffffdcb0: 0x0000000000605170  0x0000000000605160
0x7fffffffdcc0: 0x0000000000400f80  0xc0af1f658f851400
```

这里进行的操作是

```
for(int i=1;i<=5;i++){
     *(new_num[i-1]+0x8)=new_num[i];
}
```

用数字能清楚表示一点
0x6051b0+0x8的位置 写入 0x6051a0
0x6051a0+0x8的位置 写入 0x605190
0x605190+0x8的位置 写入 0x605180
....

在经过这一个循环之后再下一个断点，再来看这里的数值。

```
gdb-peda$ x/24wx 0x605160
0x605160:   0x000003b8  0x00000001  0x00000000  0x00000000
0x605170:   0x0000a5db  0x00000002  0x00605160  0x00000000
0x605180:   0x0000c619  0x00000003  0x00605170  0x00000000
0x605190:   0x00008286  0x00000004  0x00605180  0x00000000
0x6051a0:   0x0000293e  0x00000005  0x00605190  0x00000000
0x6051b0:   0x00008ecb  0x00000006  0x006051a0  0x00000000
```

第四个循环里就是比大小了

```
maxValueAddr = 0x6051b0
for(int i=0;i<=4;i++){
   if(*maxValueAddr < **(maxValueAddr+0x8)) {
       bomb();
   }
   maxValueAddr = *(maxValueAddr+0x8);
}
```

在这里就是 0x00008ecb 与 0x0000293e 比 大于没问题

0x0000293e 与 0x00008286 比。。就小于了然后就bomb了。

所以说最后phase6其实就是一个排序题目，根据0x605160这一个地址开始的 0x000003b8 0x0000a5db 0x0000c619 ... 来排序

最后可以推断出顺序为 3 2 6 4 5 1 希望可以自己按照这个顺序再调试看一遍。