利用 objdump > bomb_objdump.txt 将反编译的汇编代码输入到文本中，比较容易找到 main 函数和六个 phase 函数，看来这就是我们要破解的六个问题，以及引发爆炸的 explode_bomb 函数。接下来用 cgdb 进行调试，分别在六个 phase 函数前打上断点，然后在爆炸函数前打上断点，这样每次触发爆炸，到了爆炸函数入口处只需利用 kill 指令就可以退出本次调试但不退出 cgdb(这样可以保留断点)。下面来逐个击破。

Phase_1：

```
Dump of assembler code for function phase_1:
   0x0000000000400ee0 <+0>:  sub    $0x8,%rsp
   0x0000000000400ee4 <+4>:  mov    $0x402400,%esi
   0x0000000000400ee9 <+9>:  callq  0x401338 <strings_not_equal>
   0x0000000000400eee <+14>:    test   %eax,%eax
   0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
   0x0000000000400ef2 <+18>:    callq  0x40143a <explode_bomb>
   0x0000000000400ef7 <+23>:    add    $0x8,%rsp
   0x0000000000400efb <+27>:    retq
End of assembler dump.
```

上来先用 info register 查看寄存器，可以看到首个传入参数%rdi 存储的是我们输入的字符串地址，根据后续调用的函数<strings_not_equal>很容易想到传入%rsi 的地址 0x402400 即为要比较的字符串地址，利用 x/s 0x402400，即可得到答案，一阶段破之。

Phase_2:

```
Dump of assembler code for function phase_2:
   0x0000000000400efc <+0>: push   %rbp
   0x0000000000400efd <+1>: push   %rbx
   0x0000000000400efe <+2>: sub    $0x28,%rsp
   0x0000000000400f02 <+6>: mov    %rsp,%rsi
   0x0000000000400f05 <+9>: callq  0x40145c <read_six_numbers>
   0x0000000000400f0a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
   0x0000000000400f10 <+20>:    callq  0x40143a <explode_bomb>
   0x0000000000400f15 <+25>:    jmp    0x400f30 <phase_2+52>
   0x0000000000400f17 <+27>:    mov    -0x4(%rbx),%eax
   0x0000000000400f1a <+30>:    add    %eax,%eax
   0x0000000000400f1c <+32>:    cmp    %eax,(%rbx)
   0x0000000000400f1e <+34>:    je     0x400f25 <phase_2+41>
   0x0000000000400f20 <+36>:    callq  0x40143a <explode_bomb>
   0x0000000000400f25 <+41>:    add    $0x4,%rbx
   0x0000000000400f29 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f2c <+48>:    jne    0x400f17 <phase_2+27>
   0x0000000000400f2e <+50>:    jmp    0x400f3c <phase_2+64>
   0x0000000000400f30 <+52>:    lea    0x4(%rsp),%rbx
   0x0000000000400f35 <+57>:    lea    0x18(%rsp),%rbp
   0x0000000000400f3a <+62>:    jmp    0x400f17 <phase_2+27>
   0x0000000000400f3c <+64>:    add    $0x28,%rsp
   0x0000000000400f40 <+68>:    pop    %rbx
   0x0000000000400f41 <+69>:    pop    %rbp
   0x0000000000400f42 <+70>:    retq
End of assembler dump.
```

上来先开了一段栈的空间，然后将栈的地址作为传入参数，可以预见到后面调用函数会把数据保存在栈中返回给该函数。根据调用函数的名字<read_six_number>可以预见到需要输入六个数字，进入该函数以后也可以验证这一点。调用的函数比较简单，利用的是 sscanf 将字符串的字符解析为数字，返回值为成功输入的数字数量，如果不足六个就会爆炸，传入的参数给出了规定输入格式的字符串地址。回到 phase2 函数之后先检查字符串第一个数字是否为 1，不是则爆炸，之后进行五次循环，要求后一个数字依次为前面的两倍，因此得到答案：1 2 4 8 16 32，二阶段破之。

Phase_3:

```
Dump of assembler code for function phase_3:
=>   0x0000000000400f43 <+0>: sub    $0x18,%rsp
     0x0000000000400f47 <+4>: lea    0xc(%rsp),%rcx
     0x0000000000400f4c <+9>: lea    0x8(%rsp),%rdx
     0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
     0x0000000000400f56 <+19>:    mov    $0x0,%eax
     0x0000000000400f5b <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
     0x0000000000400f60 <+29>:    cmp    $0x1,%eax
     0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>
     0x0000000000400f65 <+34>:    callq  0x40143a <explode_bomb>
     0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
     0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
     0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
     0x0000000000400f75 <+50>:    jmpq   *0x402470(,%rax,8)
     0x0000000000400f7c <+57>:    mov    $0xcf,%eax
     0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
     0x0000000000400f83 <+64>:    mov    $0x2c3,%eax
     0x0000000000400f88 <+69>:    jmp    0x400fbe <phase_3+123>
     0x0000000000400f8a <+71>:    mov    $0x100,%eax
     0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
     0x0000000000400f91 <+78>:    mov    $0x185,%eax
     0x0000000000400f96 <+83>:    jmp    0x400fbe <phase_3+123>
     0x0000000000400f98 <+85>:    mov    $0xce,%eax
     0x0000000000400f9d <+90>:    jmp    0x400fbe <phase_3+123>
     0x0000000000400f9f <+92>:    mov    $0x2aa,%eax
     0x0000000000400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
     0x0000000000400fa6 <+99>:    mov    $0x147,%eax
     0x0000000000400fab <+104>:   jmp    0x400fbe <phase_3+123>
     0x0000000000400fad <+106>:   callq  0x40143a <explode_bomb>
     0x0000000000400fb2 <+111>:   mov    $0x0,%eax
     0x0000000000400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
     0x0000000000400fb9 <+118>:   mov    $0x137,%eax
     0x0000000000400fbe <+123>:   cmp    0xc(%rsp),%eax
     0x0000000000400fc2 <+127>:   je     0x400fc9 <phase_3+134>
     0x0000000000400fc4 <+129>:   callq  0x40143a <explode_bomb>
     0x0000000000400fc9 <+134>:   add    $0x18,%rsp
     0x0000000000400fcd <+138>:   retq
End of assembler dump.
```

阶段 3 同样是利用 sscanf 解析字符串，利用传入参数 0x4025cf 可以看到输入格式要求和上一阶段类似，不过这次只有 2 个数字，空格间隔。第一个数字保存在 0x8(%rsp)中，第二个数字保存在 0xc(%rsp)中，正确输入后先比较第一个数和 7，如果大于则爆炸，也就是限制第一个数字应在 0~7 之间。根据第一个数字和后续跳转指令可以得到第二个数字应该是多少，一共 8 种情况，任选一种匹配的情况即可通过，三阶段破之。

Phase_4:

```
Dump of assembler code for function phase_4:
   0x000000000040100c <+0>:  sub    $0x18,%rsp
   0x0000000000401010 <+4>:  lea    0xc(%rsp),%rcx
   0x0000000000401015 <+9>:  lea    0x8(%rsp),%rdx
   0x000000000040101a <+14>:     mov    $0x4025cf,%esi
   0x000000000040101f <+19>:     mov    $0x0,%eax
   0x0000000000401024 <+24>:     callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000401029 <+29>:     cmp    $0x2,%eax
   0x000000000040102c <+32>:     jne    0x401035 <phase_4+41>
   0x000000000040102e <+34>:     cmpl   $0xe,0x8(%rsp)
   0x0000000000401033 <+39>:     jbe    0x40103a <phase_4+46>
   0x0000000000401035 <+41>:     callq  0x40143a <explode_bomb>
   0x000000000040103a <+46>:     mov    $0xe,%edx
   0x000000000040103f <+51>:     mov    $0x0,%esi
   0x0000000000401044 <+56>:     mov    0x8(%rsp),%edi
   0x0000000000401048 <+60>:     callq  0x400fce <func4>
   0x000000000040104d <+65>:     test   %eax,%eax
   0x000000000040104f <+67>:     jne    0x401058 <phase_4+76>
   0x0000000000401051 <+69>:     cmpl   $0x0,0xc(%rsp)
   0x0000000000401056 <+74>:     je     0x40105d <phase_4+81>
   0x0000000000401058 <+76>:     callq  0x40143a <explode_bomb>
   0x000000000040105d <+81>:     add    $0x18,%rsp
   0x0000000000401061 <+85>:     retq
End of assembler dump.
```

解析字符串部分和上一阶段相同，一样是两个数字。之后调用<func4>函数，经过一顿复杂的操作以后你会发现这个函数要求的就是第一个数字等于 7，只有这样才能正常返回，之后检测第二个数字是否为 0，那么答案就显而易见了，四阶段破之。

Phase_5:

```
Dump of assembler code for function phase_5:
   0x0000000000401062 <+0>:   push   %rbx
   0x0000000000401063 <+1>:   sub    $0x20,%rsp
   0x0000000000401067 <+5>:   mov    %rdi,%rbx
   0x000000000040106a <+8>:   mov    %fs:0x28,%rax
   0x0000000000401073 <+17>:  mov    %rax,0x18(%rsp)
   0x0000000000401078 <+22>:  xor    %eax,%eax
   0x000000000040107a <+24>:  callq  0x40131b <string_length>
   0x000000000040107f <+29>:  cmp    $0x6,%eax
   0x0000000000401082 <+32>:  je     0x4010d2 <phase_5+112>
   0x0000000000401084 <+34>:  callq  0x40143a <explode_bomb>
   0x0000000000401089 <+39>:  jmp    0x4010d2 <phase_5+112>
   0x000000000040108b <+41>:  movzbl (%rbx,%rax,1),%ecx
   0x000000000040108f <+45>:  mov    %cl,(%rsp)
   0x0000000000401092 <+48>:  mov    (%rsp),%rdx
   0x0000000000401096 <+52>:  and    $0xf,%edx
   0x0000000000401099 <+55>:  movzbl 0x4024b0(%rdx),%edx
   0x00000000004010a0 <+62>:  mov    %dl,0x10(%rsp,%rax,1)
   0x00000000004010a4 <+66>:  add    $0x1,%rax
   0x00000000004010a8 <+70>:  cmp    $0x6,%rax
   0x00000000004010ac <+74>:  jne    0x40108b <phase_5+41>
   0x00000000004010ae <+76>:  movb   $0x0,0x16(%rsp)
   0x00000000004010b3 <+81>:  mov    $0x40245e,%esi
   0x00000000004010b8 <+86>:  lea    0x10(%rsp),%rdi
   0x00000000004010bd <+91>:  callq  0x401338 <strings_not_equal>
   0x00000000004010c2 <+96>:  test   %eax,%eax
   0x00000000004010c4 <+98>:  je     0x4010d9 <phase_5+119>
   0x00000000004010c6 <+100>: callq  0x40143a <explode_bomb>
   0x00000000004010cb <+105>: nopl   0x0(%rax,%rax,1)
   0x00000000004010d0 <+110>: jmp    0x4010d9 <phase_5+119>
   0x00000000004010d2 <+112>: mov    $0x0,%eax
   0x00000000004010d7 <+117>: jmp    0x40108b <phase_5+41>
   0x00000000004010d9 <+119>: mov    0x18(%rsp),%rax
   0x00000000004010de <+124>: xor    %fs:0x28,%rax
   0x00000000004010e7 <+133>: je     0x4010ee <phase_5+140>
   0x00000000004010e9 <+135>: callq  0x400b30 <__stack_chk_fail@plt>
   0x00000000004010ee <+140>: add    $0x20,%rsp
   0x00000000004010f2 <+144>: pop    %rbx
   0x00000000004010f3 <+145>: retq
End of assembler dump.
```

阶段五上来调用了一个<string_length>,根据检测的%rax 值可以判断需要我们输入一个长度为 6 的字符串。之后进一个循环，一通操作实际上是分别提取我们输入字符串 6 个字符的第四位作为偏移值，将 0x4024b0 位置(“maduiersnfotvbyl”)作为字符串首地址加上偏移值传送到栈中，根据后续调用<strings_not_equal>以及前面传入的参数 0x40245e 可确定需求字符串”flyers”从而确定偏移值，五阶段破之。

Phase_6：

代码很长，就不放图了，最后一阶段主要靠堆循环的数量来提高复杂度。将每个循环的伪代码写出来，可以看到第一个循环是要求输入的六个数字应该小于等于 6 且各不相同，因此这六个数字必然为集合{1,2,3,4,5,6}，剩下的就是确定排列方式。后面几个循环主要就是把地址来回捣腾，通过将地址列表记录的方式可以比较容易的确定每个循环是干什么的，最终每个地址还是链接了一个数字。进入最后一个循环发现需要比较最终一组地址处的数字大小，在需要比较的地址处可以找到六个数字，大小顺序为 432165，不难分析出最后一个循环的检测条件为从大到小排列，因此可以得到答案，六阶段破之。