基本思路：在各个 phase 函数打断点，进入函数后 stepi 单步调试，查看寄存器。

```
(gdb) break phase_1
Breakpoint 2 at 0x400e8d
(gdb) break phase_2
Breakpoint 3 at 0x400ea9
(gdb) break phase_3
Breakpoint 4 at 0x400f15
(gdb) break phase_4
Breakpoint 5 at 0x401020
(gdb) break phase_5
Breakpoint 6 at 0x40108d
(gdb) break phase_6
```

Phase 1：通过 disas phase_1 查看汇编代码：

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400e8d <+0>:     sub    $0x8,%rsp
   0x0000000000400e91 <+4>:     mov    $0x4023bc,%esi
   0x0000000000400e96 <+9>:     callq  0x40133c <strings_not_equal>
   0x0000000000400e9b <+14>:    test   %eax,%eax
   0x0000000000400e9d <+16>:    je     0x400ea4 <phase_1+23>
   0x0000000000400e9f <+18>:    callq  0x40143b <explode_bomb>
   0x0000000000400ea4 <+23>:    add    $0x8,%rsp
   0x0000000000400ea8 <+27>:    retq
```

注意到用于跳转的判断条件为调用 strings_not_equal 函数的返回值。

且函数的一个参数存储在%esi 寄存器中，则

```
(gdb) x /s $rsi
0x4023bc:       "Crikey! I have lost my mojo!"
```

可见 phase 1 的答案是 Crikey! I have lost my mojo! 经检验该输入可以通过。

Phase 2：通过 disas phase_2 查看汇编代码：

```
   0x0000000000400ec2 <+25>:    callq  0x40145d <read_six_numbers>
   0x0000000000400ec7 <+30>:    cmpl   $0x0,(%rsp)
   0x0000000000400ecb <+34>:    jne    0x400ed4 <phase_2+43>
   0x0000000000400ecd <+36>:    cmpl   $0x1,0x4(%rsp)
   0x0000000000400ed2 <+41>:    je     0x400ed9 <phase_2+48>
   0x0000000000400ed4 <+43>:    callq  0x40143b <explode_bomb>
   0x0000000000400ed9 <+48>:    mov    %rsp,%rbx
   0x0000000000400edc <+51>:    lea    0x10(%rsp),%rbp
   0x0000000000400ee1 <+56>:    mov    0x4(%rbx),%eax
   0x0000000000400ee4 <+59>:    add    (%rbx),%eax
   0x0000000000400ee6 <+61>:    cmp    %eax,0x8(%rbx)
   0x0000000000400ee9 <+64>:    je     0x400ef0 <phase_2+71>
   0x0000000000400eeb <+66>:    callq  0x40143b <explode_bomb>
   0x0000000000400ef0 <+71>:    add    $0x4,%rbx
   0x0000000000400ef4 <+75>:    cmp    %rbp,%rbx
   0x0000000000400ef7 <+78>:    jne    0x400ee1 <phase_2+56>
   0x0000000000400ef9 <+80>:    mov    0x18(%rsp),%rax
   0x0000000000400efe <+85>:    xor    %fs:0x28,%rax
   0x0000000000400f07 <+94>:    je     0x400f0e <phase_2+101>
   0x0000000000400f09 <+96>:    callq  0x400b00 <__stack_chk_fail@plt>
```

由函数名 read_six_numbers 及汇编代码，认为要输入 6 个整数。

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x000000000040145d <+0>:     sub    $0x8,%rsp
   0x0000000000401461 <+4>:     mov    %rsi,%rdx
   0x0000000000401464 <+7>:     lea    0x4(%rsi),%rcx
   0x0000000000401468 <+11>:    lea    0x14(%rsi),%rax
   0x000000000040146c <+15>:    push   %rax
   0x000000000040146d <+16>:    lea    0x10(%rsi),%rax
   0x0000000000401471 <+20>:    push   %rax
   0x0000000000401472 <+21>:    lea    0xc(%rsi),%r9
   0x0000000000401476 <+25>:    lea    0x8(%rsi),%r8
   0x000000000040147a <+29>:    mov    $0x40255b,%esi
   0x000000000040147f <+34>:    mov    $0x0,%eax
   0x0000000000401484 <+39>:    callq  0x400bb0 <__isoc99_sscanf@plt>
   0x0000000000401489 <+44>:    add    $0x10,%rsp
   0x000000000040148d <+48>:    cmp    $0x5,%eax
   0x0000000000401490 <+51>:    jg     0x401497 <read_six_numbers+58>
   0x0000000000401492 <+53>:    callq  0x40143b <explode_bomb>
   0x0000000000401497 <+58>:    add    $0x8,%rsp
   0x000000000040149b <+62>:    retq
End of assembler dump.
(gdb) x /s 0x40255b
0x40255b:       "%d %d %d %d %d %d"
```

由 cmpl $0x0,(%rsp)，cmpl $0x1,0x4(%rsp)及后续跳转判断第一个数字为 0，第二个为 1。单步运行 info registers 查看寄存器，%rsp 地址不断移动，每个地址存储的数据与前两个之和比较，不相等则跳转到 explode_bomb，判断斐波那契数列 0 1 1 2 3 5，经检验符合。

Phase 3：通过 disas phase_3 查看汇编代码：

```
   0x0000000000400f31 <+28>:    mov    $0x402567,%esi
   0x0000000000400f36 <+33>:    callq  0x400bb0 <__isoc99_sscanf@plt>
   0x0000000000400f3b <+38>:    cmp    $0x1,%eax
   0x0000000000400f3e <+41>:    jg     0x400f45 <phase_3+48>
   0x0000000000400f40 <+43>:    callq  0x40143b <explode_bomb>
   0x0000000000400f45 <+48>:    cmpl   $0x7,(%rsp)
   0x0000000000400f49 <+52>:    ja     0x400fb0 <phase_3+155>
   0x0000000000400f4b <+54>:    mov    (%rsp),%eax
```

可见输入的数量需大于 1，否则执行 explode_bomb。

```
(gdb) x /s 0x402567
0x402567:       "%d %d"
```

可见输入为两个整数。

```
   0x0000000000400f45 <+48>:    cmpl   $0x7,(%rsp)
   0x0000000000400f49 <+52>:    ja     0x400fb0 <phase_3+155>
   0x0000000000400f4b <+54>:    mov    (%rsp),%eax
```

```
   0x0000000000400fba <+165>:   cmpl   $0x5,(%rsp)
   0x0000000000400fbe <+169>:   jg     0x400fc6 <phase_3+177>
   0x0000000000400fc0 <+171>:   cmp    0x4(%rsp),%eax
   0x0000000000400fc4 <+175>:   je     0x400fcb <phase_3+182>
   0x0000000000400fc6 <+177>:   callq  0x40143b <explode_bomb>
```

可见第一个数字低于或等于 5（无符号），该数一系列操作后得到的值储存在%eax 中，该值应等于第二个数，否则将跳转执行 explode_bomb。
观察中间代码可见语句之间的跳转与第一个数字有关，难以直接推出算式。

```
   0x0000000000400fc0 <+171>:   cmp    0x4(%rsp),%eax
   0x0000000000400fc4 <+175>:   je     0x400fcb <phase_3+182>
   0x0000000000400fc6 <+177>:   callq  0x40143b <explode_bomb>
```

在*(0x400fc0)处断点，尝试输入 0 和一随机数字，直接跳转到该断点，查看%eax。

```
(gdb) print $eax
$4 = -540
```

则判断 0 -540 为一组符合要求的输入，经检验通过。
同理可得另外四组解，分别为 1 -591，2 -41，3 -914，4 0 和 5 -914。

Phase 4：通过 disas phase_4 查看汇编代码：

```
   0x000000000040103c <+28>:    mov    $0x402567,%esi
   0x0000000000401041 <+33>:    callq  0x400bb0 <__isoc99_sscanf@plt>
   0x0000000000401046 <+38>:    cmp    $0x2,%eax
   0x0000000000401049 <+41>:    jne    0x401056 <phase_4+54>
   0x000000000040104b <+43>:    mov    (%rsp),%eax
```

```
(gdb) x /s 0x402567
0x402567:       "%d %d"
```

可见需要输入两个整数。

随机输入两个数字，运行到 phase_4，分别查看(%rsp)和 0x4(%rsp)的值，发现第一个数字储存在 0x4(%rsp)中而第二个数字储存在(%rsp)中。

```
   0x000000000040104b <+43>:    mov    (%rsp),%eax
   0x000000000040104e <+46>:    sub    $0x2,%eax
   0x0000000000401051 <+49>:    cmp    $0x2,%eax
   0x0000000000401054 <+52>:    jbe    0x40105b <phase_4+59>
   0x0000000000401056 <+54>:    callq  0x40143b <explode_bomb>
   0x000000000040105b <+59>:    mov    (%rsp),%esi
```

通过上面这段代码可知第二个数字满足 0 <= x – 2 <= 2，即 2 <= x <= 4。

```
   0x000000000040105b <+59>:    mov    (%rsp),%esi
   0x000000000040105e <+62>:    mov    $0x8,%edi
   0x0000000000401063 <+67>:    callq  0x400fe5 <func4>
   0x0000000000401068 <+72>:    cmp    0x4(%rsp),%eax
   0x000000000040106c <+76>:    je     0x401073 <phase_4+83>
   0x000000000040106e <+78>:    callq  0x40143b <explode_bomb>
   0x0000000000401073 <+83>:    mov    0x8(%rsp),%rax
```

可以看到此处调用 func4，两个参数分别是第二个数和 8，返回值需等于第一个数。

通过 disas func4 查看汇编代码：

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x0000000000400fe5 <+0>:     test   %edi,%edi
   0x0000000000400fe7 <+2>:     jle    0x401014 <func4+47>
   0x0000000000400fe9 <+4>:     mov    %esi,%eax
   0x0000000000400feb <+6>:     cmp    $0x1,%edi
   0x0000000000400fee <+9>:     je     0x40101e <func4+57>
   0x0000000000400ff0 <+11>:    push   %r12
   0x0000000000400ff2 <+13>:    push   %rbp
   0x0000000000400ff3 <+14>:    push   %rbx
   0x0000000000400ff4 <+15>:    mov    %esi,%ebp
   0x0000000000400ff6 <+17>:    mov    %edi,%ebx
   0x0000000000400ff8 <+19>:    lea    -0x1(%rdi),%edi
   0x0000000000400ffb <+22>:    callq  0x400fe5 <func4>
   0x0000000000401000 <+27>:    lea    0x0(%rbp,%rax,1),%r12d
   0x0000000000401005 <+32>:    lea    -0x2(%rbx),%edi
   0x0000000000401008 <+35>:    mov    %ebp,%esi
   0x000000000040100a <+37>:    callq  0x400fe5 <func4>
   0x000000000040100f <+42>:    add    %r12d,%eax
   0x0000000000401012 <+45>:    jmp    0x40101a <func4+53>
   0x0000000000401014 <+47>:    mov    $0x0,%eax
   0x0000000000401019 <+52>:    retq
   0x000000000040101a <+53>:    pop    %rbx
   0x000000000040101b <+54>:    pop    %rbp
   0x000000000040101c <+55>:    pop    %r12
   0x000000000040101e <+57>:    repz retq
```

观察汇编代码可知该函数含有递归，其对应的 C 代码应为：

```
int func4(int x, int y) {
    if (y <= 0)
        return 0;
    else if (y == 1)
        return x;
    else
        return func4(x, y-1) + func4(x, y-2) + x;
}
```

则两个数满足关系 x = func4(y, 8) = 54 * y。

则由此可推出三组解，分别为 108 2，162 3 和 216 4，经检验均可通过。

Phase 5：通过 disas phase_5 查看汇编代码：

```
   0x00000000004010a5 <+24>:    callq  0x40131e <string_length>
   0x00000000004010aa <+29>:    cmp    $0x6,%eax
   0x00000000004010ad <+32>:    je     0x4010b4 <phase_5+39>
   0x00000000004010af <+34>:    callq  0x40143b <explode_bomb>
```

可见输入字符串的长度为 6，即共有 6 个字母。

```
   0x00000000004010b4 <+39>:    mov    $0x0,%eax
   0x00000000004010b9 <+44>:    movzbl (%rbx,%rax,1),%edx
   0x00000000004010bd <+48>:    and    $0xf,%edx
   0x00000000004010c0 <+51>:    movzbl 0x402420(%rdx),%edx
   0x00000000004010c7 <+58>:    mov    %dl,(%rsp,%rax,1)
   0x00000000004010ca <+61>:    add    $0x1,%rax
   0x00000000004010ce <+65>:    cmp    $0x6,%rax
   0x00000000004010d2 <+69>:    jne    0x4010b9 <phase_5+44>
   0x00000000004010d4 <+71>:    movb   $0x0,0x6(%rsp)
   0x00000000004010d9 <+76>:    mov    $0x4023d9,%esi
   0x00000000004010de <+81>:    mov    %rsp,%rdi
   0x00000000004010e1 <+84>:    callq  0x40133c <strings_not_equal>
```

可见字符串比较之前通过某些规则进行了加密。

既然总共也只有 26 个字母，那不妨 6 个一组试一试吧。

```
(gdb) x /s $esi
0x4023d9:       "oilers"
(gdb) x /s $rdi
0x7fffffffdd40: "aduier"
```

在*(0x40133c)处断点，查看寄存器%esi 和%rdi 的值，得知目标字符串为 oilers。

abcdef –> aduier，同理 ghijkl -> snfotv，mnopqr -> bylmad，stuvwx –> uiersn，yzabcd -> foadui，则根据这些转化关系可以列出 26 个字母的加密对照表：

a	b	c	d	e	f	g	h	i	j	k	l	m
a	d	u	i	e	r	s	n	f	o	t	v	b

n	o	p	q	r	s	t	u	V	w	x	y	z
y	l	m	a	d	u	i	e	R	s	n	f	o

由 oilers 逆推可得到其中一组符合条件的输入为 jdoefg，经检验通过。

Phase 6：通过 disas phase_6 查看汇编代码：

```
   0x0000000000401127 <+29>:    callq  0x40145d <read_six_numbers>
```

可见输入为 6 个整数，尝试 1 2 3 4 5 6。

```
   0x000000000040112c <+34>:    mov    %rsp,%r12
   0x000000000040112f <+37>:    mov    $0x0,%r13d
   0x0000000000401135 <+43>:    mov    %r12,%rbp
   0x0000000000401138 <+46>:    mov    (%r12),%eax
   0x000000000040113c <+50>:    sub    $0x1,%eax
   0x000000000040113f <+53>:    cmp    $0x5,%eax
   0x0000000000401142 <+56>:    jbe    0x401149 <phase_6+63>
   0x0000000000401144 <+58>:    callq  0x40143b <explode_bomb>
```

则 0 <= a[0] – 1 <= 5。

中间较长一段代码未见目标为 explode_bomb 的跳转，直到：

```
   0x00000000004011d7 <+205>:   mov    $0x5,%ebp
   0x00000000004011dc <+210>:   mov    0x8(%rbx),%rax
   0x00000000004011e0 <+214>:   mov    (%rax),%eax
   0x00000000004011e2 <+216>:   cmp    %eax,(%rbx)
   0x00000000004011e4 <+218>:   jle    0x4011eb <phase_6+225>
   0x00000000004011e6 <+220>:   callq  0x40143b <explode_bomb>
```

通过观察可知这是一个循环，链表中的每一个数字需比上一个大，否则爆炸。

```
   0x0000000000401190 <+134>:   mov    $0x0,%esi
   0x0000000000401195 <+139>:   mov    (%rsp,%rsi,1),%ecx
   0x0000000000401198 <+142>:   mov    $0x1,%eax
   0x000000000040119d <+147>:   mov    $0x6032f0,%edx
   0x00000000004011a2 <+152>:   cmp    $0x1,%ecx
   0x00000000004011a5 <+155>:   jg     0x401174 <phase_6+106>
```

链表储存在地址 0x6032f0 处，从该地址开始查看链表：

```
(gdb) x /24wd 0x6032f0
0x6032f0 <node1>:       334     1       6304512 0
0x603300 <node2>:       831     2       6304528 0
0x603310 <node3>:       299     3       6304544 0
0x603320 <node4>:       637     4       6304560 0
0x603330 <node5>:       852     5       6304576 0
0x603340 <node6>:       310     6       0       0
```

对该链表进行从小到大重新排序，应为 3 6 1 4 2 5，经检验通过。

Secret Phase：观察提供的 bomb.c 代码，注意到 phase_6(input)与 phase_defused 后没有其他操作了，进一步观察汇编代码可见 secret phase 隐藏在 phase_defused 中。

```
0x00000000004015d6 <+20>:    cmpl   $0x6,0x2021af(%rip)        # 0x60378c <num_input_strings>
0x00000000004015dd <+27>:    jne    0x40163d <phase_defused+123>
```

进一步确认 phase 6 的 defuse 过程与其他关卡不同，可能是开启隐藏关卡的入口。

```
0x00000000004015fd <+59>:    cmp    $0x3,%eax
0x0000000000401600 <+62>:    jne    0x401633 <phase_defused+113>
0x0000000000401602 <+64>:    mov    $0x4025ba,%esi
0x0000000000401607 <+69>:    lea    0x10(%rsp),%rdi
0x000000000040160c <+74>:    callq  0x40133c <strings_not_equal>
0x0000000000401611 <+79>:    test   %eax,%eax
0x0000000000401613 <+81>:    jne    0x401633 <phase_defused+113>
```

猜测此处的 strings_not_equal 用于测试 phase 4 的附加字符串是否正确。phase 4 附加随机字符串，在*(0x40160c)处断点，运行到该处查看寄存器%esi 和%rdi。

```
(gdb) x /s $esi
0x4025ba:       "DrEvil"
```

说明想要开启 secret phase，须在 phase 4 后附加的字符串是 DrEvil。

通过 disas secret_phase 查看汇编代码：

```
0x0000000000401253 <+1>:     callq  0x40149c <read_line>
0x0000000000401258 <+6>:     mov    $0xa,%edx
0x000000000040125d <+11>:    mov    $0x0,%esi
0x0000000000401262 <+16>:    mov    %rax,%rdi
0x0000000000401265 <+19>:    callq  0x400b90 <strtol@plt>
```

可见输入的字符串会被转化成十进制数。

```
0x000000000040126a <+24>:    mov    %rax,%rbx
0x000000000040126d <+27>:    lea    -0x1(%rax),%eax
0x0000000000401270 <+30>:    cmp    $0x3e8,%eax
0x0000000000401275 <+35>:    jbe    0x40127c <secret_phase+42>
0x0000000000401277 <+37>:    callq  0x40143b <explode_bomb>
```

```
0x000000000040127c <+42>:    mov    %ebx,%esi
0x000000000040127e <+44>:    mov    $0x603110,%edi
0x0000000000401283 <+49>:    callq  0x401214 <fun7>
0x0000000000401288 <+54>:    cmp    $0x1,%eax
0x000000000040128b <+57>:    je     0x401292 <secret_phase+64>
0x000000000040128d <+59>:    callq  0x40143b <explode_bomb>
```

可见这个数满足 0 <= x – 1 <= 0x3e8，即 1 <= x <= 1001，fun7 的返回值为 1。

通过 disas fun7 查看汇编代码，为一递归函数：

```
(gdb) disas fun7
Dump of assembler code for function fun7:
   0x0000000000401214 <+0>:     sub    $0x8,%rsp
   0x0000000000401218 <+4>:     test   %rdi,%rdi
   0x000000000040121b <+7>:     je     0x401248 <fun7+52>
   0x000000000040121d <+9>:     mov    (%rdi),%edx
   0x000000000040121f <+11>:    cmp    %esi,%edx
   0x0000000000401221 <+13>:    jle    0x401230 <fun7+28>
   0x0000000000401223 <+15>:    mov    0x8(%rdi),%rdi
   0x0000000000401227 <+19>:    callq  0x401214 <fun7>
   0x000000000040122c <+24>:    add    %eax,%eax
   0x000000000040122e <+26>:    jmp    0x40124d <fun7+57>
   0x0000000000401230 <+28>:    mov    $0x0,%eax
   0x0000000000401235 <+33>:    cmp    %esi,%edx
   0x0000000000401237 <+35>:    je     0x40124d <fun7+57>
   0x0000000000401239 <+37>:    mov    0x10(%rdi),%rdi
   0x000000000040123d <+41>:    callq  0x401214 <fun7>
   0x0000000000401242 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401246 <+50>:    jmp    0x40124d <fun7+57>
   0x0000000000401248 <+52>:    mov    $0xffffffff,%eax
   0x000000000040124d <+57>:    add    $0x8,%rsp
   0x0000000000401251 <+61>:    retq
```

如果是已知参数，求返回值，就不一定需要推出函数的具体内容，可以直接通过寄存器获得特定输入的返回值，已知返回值倒退参数，就必须知道函数的具体实现了。

```
int fun7(int val, Node *root) {
    if (val == 0)
        return -1;
    else if (root->val == val)
        return 0;
    else if (root->val < val)
        return 2 * fun7(val, root->right) + 1;
    else
        return 2 * fun7(val, root->left);
}
```

返回值为 1，推测上一次的返回值为 0，由右子树返回，其 val 的值为 0。

在*(0x401221)处断点，通过根节点的地址 0x603110，依次遍历该树。

```
(gdb) p /x *0x603110@5
$1 = {0x24, 0x0, 0x603130, 0x0, 0x603150}
(gdb) p /x *0x603150@5
$2 = {0x32, 0x0, 0x603190, 0x0, 0x6031d0}
(gdb)
```

由此可得 val 的值为 0x32 即 50，满足 val == root->right->val 且 val >root->val。

```
Crikey! I have lost my mojo!
0 1 1 2 3 5
0 -540
108 2 DrEvil
jdoefg
3 6 1 4 2 5
50
```

```
river@ubuntu:~/Desktop/workspace/lab05$ ./bomb solution
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```