题解（兼笔记）

gdb 的使用

先使用 make qemu-gdb，再另开一个 terminal 启动 make gdb，即可自动 remote 连接到我们的 bomb 程序。

基础命令：

- `b *0x00000a` 在 0x00000a 地址出设置断点
- `show breakpoints` 查看当前的所有断点
- `next` 运行下一条指令但不进入函数
- `step` 运行下一条指令且进入函数
- `x/wx 0x00000a` x 表示 examine，查看内存内容，/w 表示以 word 为单位显示（4 字节）（g 是 8 字节，b 是 2 字节，2b 表示读两个二字节），x 表示以十六进制显示，后面接内存地址就行
- `disassemble phase_2` 将 phase_2 内的汇编代码全部输出

phase0

```
0x400740 <phase_0+12>    adrp   x1, 0x4a0000        X1 => 0x4a0000 —  0x4236c0 (__memmove_
  0x400744 <phase_0+16>    ldr    w1, [x1, #0x54]     W1, [phase0_ans] => 0x7e6
  0x400748 <phase_0+20>    cmp    w1, w0              0x7e6 - 0x7b     CPSR => 0x20000000 
```

用 `x/wx 0x4a0000+0x54` 即可查看 w1 的值，即得到 key

phase1

```
0x400760 <phase_1>       stp    x29, x30, [sp, #-0x10]!
  0x400764 <phase_1+4>     mov    x29, sp                     X29 => 0x4000007ffaf0 —  0x4
  0x400768 <phase_1+8>     adrp   x1, 0x4a0000                X1 => 0x4a0000 —  0x4236c0 (
  0x40076c <phase_1+12>    ldr    x1, [x1, #0x58]             X1, [phase1_ans] => 0x464810
  0x400770 <phase_1+16>    bl     strcmp                      <strcmp>

  0x400774 <phase_1+20>    cbnz   w0, phase_1+32              <phase_1+32>

  0x400778 <phase_1+24>    ldp    x29, x30, [sp], #0x10
  0x40077c <phase_1+28>    ret
```

这里是把我们的输入和内存里的一个字符串作 strcmp，用 `x/s 0x464810` 查看传入的 x1 参数的值，即得到 key。

phase2

查看 phase_2 源代码，要输入八个整数，假设依次为 a1~a8 翻译一下代码要求的数字关系：

a1 = 1 a2 = 1 a1 + a2 + 3 = a3 a2 + a3 + 3 = a4 a3 + a4 + 3 = a5 a4 + a5 + 3 = a6 a5 + a6 + 3 = a7 a6 + a7 + 3 = a8

因此 a1~a8= 1 1 5 9 17 29 49 81

phase3

源代码：

```
0x0000000000400800 <+0>: stp x29, x30, [sp, #-32]!
0x0000000000400804 <+4>: mov x29, sp
0x0000000000400808 <+8>: add x3, sp, #0x18
0x000000000040080c <+12>:    add x2, sp, #0x1c
0x0000000000400810 <+16>:    adrp    x1, 0x464000 <free_mem+64>
0x0000000000400814 <+20>:    add x1, x1, #0x7d8
0x0000000000400818 <+24>:    bl  0x406d80 <__isoc99_sscanf>
0x000000000040081c <+28>:    cmp w0, #0x2
0x0000000000400820 <+32>:    b.ne    0x40084c <phase_3+76>  // b.any
0x0000000000400824 <+36>:    ldr w0, [sp, #28]
0x0000000000400828 <+40>:    cmp w0, #0x3
0x000000000040082c <+44>:    b.eq    0x400884 <phase_3+132>  // b.none
0x0000000000400830 <+48>:    cmp w0, #0x6
0x0000000000400834 <+52>:    b.eq    0x400854 <phase_3+84>  // b.none
0x0000000000400838 <+56>:    cmp w0, #0x2
0x000000000040083c <+60>:    b.eq    0x4008a0 <phase_3+160>  // b.none
0x0000000000400840 <+64>:    bl  0x400af4 <explode>
0x0000000000400844 <+68>:    ldp x29, x30, [sp], #32
0x0000000000400848 <+72>:    ret
0x000000000040084c <+76>:    bl  0x400af4 <explode>
0x0000000000400850 <+80>:    b   0x400824 <phase_3+36>
0x0000000000400854 <+84>:    ldr w2, [sp, #24]
0x0000000000400858 <+88>:    mov w0, #0x6667                 // #26215
0x000000000040085c <+92>:    movk    w0, #0x6666, lsl #16
0x0000000000400860 <+96>:    smull   x0, w2, w0
0x0000000000400864 <+100>:   asr x0, x0, #34
0x0000000000400868 <+104>:   sub w0, w0, w2, asr #31
0x000000000040086c <+108>:   add w1, w0, w0, lsl #2
0x0000000000400870 <+112>:   sub w1, w2, w1, lsl #1
0x0000000000400874 <+116>:   add w0, w1, w0
0x0000000000400878 <+120>:   cmp w0, #0x6
0x000000000040087c <+124>:   b.eq    0x400844 <phase_3+68>  // b.none
0x0000000000400880 <+128>:   bl  0x400af4 <explode>
0x0000000000400884 <+132>:   ldr w0, [sp, #24]
0x0000000000400888 <+136>:   eor w0, w0, w0, asr #3
0x000000000040088c <+140>:   and w0, w0, #0x7
0x0000000000400890 <+144>:   ldr w1, [sp, #28]
```

```
0x0000000000400894 <+148>:   cmp w0, w1
0x0000000000400898 <+152>:   b.eq    0x400844 <phase_3+68>  // b.none
0x000000000040089c <+156>:   bl  0x400af4 <explode>
0x00000000004008a0 <+160>:   ldr w0, [sp, #24]
0x00000000004008a4 <+164>:   ldr w1, [sp, #28]
0x00000000004008a8 <+168>:   and w2, w0, #0x7
0x00000000004008ac <+172>:   cmp w2, w1
0x00000000004008b0 <+176>:   b.eq    0x400844 <phase_3+68>  // b.none
0x00000000004008b4 <+180>:   ubfx    x0, x0, #3, #3
0x00000000004008b8 <+184>:   cmp w1, w0
0x00000000004008bc <+188>:   b.eq    0x400844 <phase_3+68>  // b.none
0x00000000004008c0 <+192>:   bl  0x400af4 <explode>
0x00000000004008c4 <+196>:   b   0x400840 <phase_3+64>
```

看看 sscanf 是怎么解析的：

```
0x400818 <phase_3+24>    bl     __isoc99_sscanf             <__isoc99_sscanf>
       s: 0x4a2138 (input_buf) — 0x2000006f6c6c6568 /* 'hello' */
       format: 0x4647d8 — 0x6425206425 /* '%d %d' */
       vararg: 0x4000007ffafc — 0x7ffb1000000000
```

说明应该三解析成两个整数，w0 存储的是解析出的数字的个数，不为 2 则爆炸。

随后就是走迷宫问题，假设我们输入的是 a 和 b，

先看 a=3，则跳到 132：

b=b xor (b » 3) b=b and bin(111) if (a=b) ret

反向解一下，b and bin(111)=bin(11)，b=10011 是可以的，然后继续往前推理
b=3

所以 a,b=3,3 即为 key

phase4

```
0x00000000004009e4 <+0>: stp x29, x30, [sp, #-32]!
0x00000000004009e8 <+4>: mov x29, sp
0x00000000004009ec <+8>: stp x19, x20, [sp, #16]
0x00000000004009f0 <+12>:    mov x19, x0
0x00000000004009f4 <+16>:    bl  0x400300 <strlen>
0x00000000004009f8 <+20>:    mov x20, x0
0x00000000004009fc <+24>:    cmp w0, #0xa
0x0000000000400a00 <+28>:    b.gt    0x400a3c <phase_4+88>
0x0000000000400a04 <+32>:    mov w1, w20
0x0000000000400a08 <+36>:    mov x0, x19
0x0000000000400a0c <+40>:    bl  0x4008c8 <encrypt_method1>
0x0000000000400a10 <+44>:    mov w1, w20
0x0000000000400a14 <+48>:    mov x0, x19
```

```
0x0000000000400a18 <+52>:    bl  0x400964 <encrypt_method2>
0x0000000000400a1c <+56>:    adrp    x0, 0x4a0000
0x0000000000400a20 <+60>:    ldr x1, [x0, #104]
0x0000000000400a24 <+64>:    mov x0, x19
0x0000000000400a28 <+68>:    bl  0x421b80 <strcmp>
0x0000000000400a2c <+72>:    cbnz    w0, 0x400a44 <phase_4+96>
0x0000000000400a30 <+76>:    ldp x19, x20, [sp, #16]
0x0000000000400a34 <+80>:    ldp x29, x30, [sp], #32
0x0000000000400a38 <+84>:    ret
0x0000000000400a3c <+88>:    bl  0x400af4 <explode>
0x0000000000400a40 <+92>:    b   0x400a04 <phase_4+32>
0x0000000000400a44 <+96>:    bl  0x400af4 <explode>
0x0000000000400a48 <+100>:   b   0x400a30 <phase_4+76>
```

试验破解一下两个 encrypt_method，qetwry encrypt_method1 是把字符串所有奇数位提到前面，偶数位放到后面：abcdef 变成 acebdf（不过字符串长度为奇数的时候好像会 bug

encrypt_method2 是一个替换加密，我们需要找到密钥表，

查看 0x4a0060 处的密钥表地址，是 0x4647f0，再查 0x4647f0 处开头的 26 个字节，得到密钥表：

原文—密文 a-q b-w c-e d-r e-t f-y g-u h-i i-o j-p k-a l-s m-d n-f o-g p-h q-j r-k s-l t-z u-x v-c w-v x-b y-n z-m

再看目标字符串是 isggstsvkt

按步骤还原出来 key 应该是：hloolelwre->helloworle

phase5

```
0x0000000000400ac0 <+0>:    stp x29, x30, [sp, #-16]!
0x0000000000400ac4 <+4>:    mov x29, sp
0x0000000000400ac8 <+8>:    bl  0x400bd4 <read_int>
0x0000000000400acc <+12>:   adrp    x1, 0x4a0000
0x0000000000400ad0 <+16>:   add x1, x1, #0x58
0x0000000000400ad4 <+20>:   add x1, x1, #0x18
0x0000000000400ad8 <+24>:   bl  0x400a4c <func_5>
0x0000000000400adc <+28>:   cmp w0, #0x3
0x0000000000400ae0 <+32>:   b.ne    0x400aec <phase_5+44>  // b.any
0x0000000000400ae4 <+36>:   ldp x29, x30, [sp], #16
0x0000000000400ae8 <+40>:   ret
0x0000000000400aec <+44>:   bl  0x400af4 <explode>
0x0000000000400af0 <+48>:   b   0x400ae4 <phase_5+36>
```

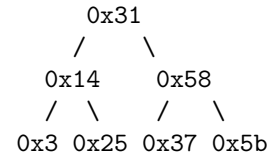
func5:

```
0x0000000000400a4c <+0>: cbz x1, 0x400ab8 <func_5+108>
```

```
0x0000000000400a50 <+4>: stp x29, x30, [sp, #-32]!
0x0000000000400a54 <+8>: mov x29, sp
0x0000000000400a58 <+12>:    stp x19, x20, [sp, #16]
0x0000000000400a5c <+16>:    mov w20, w0
0x0000000000400a60 <+20>:    mov x19, x1
0x0000000000400a64 <+24>:    ldr w0, [x1]
0x0000000000400a68 <+28>:    cmp w0, w20
0x0000000000400a6c <+32>:    b.eq    0x400a98 <func_5+76>  // b.none
0x0000000000400a70 <+36>:    ldr w0, [x19]
0x0000000000400a74 <+40>:    cmp w0, w20
0x0000000000400a78 <+44>:    b.le    0x400aa0 <func_5+84>
0x0000000000400a7c <+48>:    ldr x1, [x19, #8]
0x0000000000400a80 <+52>:    mov w0, w20
0x0000000000400a84 <+56>:    bl  0x400a4c <func_5>
0x0000000000400a88 <+60>:    lsl w0, w0, #1
0x0000000000400a8c <+64>:    ldp x19, x20, [sp, #16]
0x0000000000400a90 <+68>:    ldp x29, x30, [sp], #32
0x0000000000400a94 <+72>:    ret
0x0000000000400a98 <+76>:    bl  0x400af4 <explode>
0x0000000000400a9c <+80>:    b   0x400a70 <func_5+36>
0x0000000000400aa0 <+84>:    ldr x1, [x19, #16]
0x0000000000400aa4 <+88>:    mov w0, w20
0x0000000000400aa8 <+92>:    bl  0x400a4c <func_5>
0x0000000000400aac <+96>:    lsl w0, w0, #1
0x0000000000400ab0 <+100>:   add w0, w0, #0x1
0x0000000000400ab4 <+104>:   b   0x400a8c <func_5+64>
0x0000000000400ab8 <+108>:   mov w0, #0x0                    // #0
0x0000000000400abc <+112>:   ret
```

实际上 func5 实现的是一个二分查找的程序，同时用返回的 w0 来记录查找的路径（有点像 huffman-code），最后要求我们的 w0=3

通过读取内存可以知道树的结构大概长这样：



从右侧返回 w02+1，从左侧返回 w02，一共往上走三次，结合大小关系知道我们输入的 key 要在 0x58<w0<0x5b 之间。

比如输入 91, 即可。