范意阳 22307110117

bomb-lab 实验报告

第一题：

key：*Computer science is not a boring subject*

思路：跳转到 phase_1 后，总览一遍代码

首先循环调用 read_line 读取输入值，存储在 0x7fffffffe510

0x555555555558c <phase_1+52> call string_not_equal:

此指令调用 string_not_equal 函数，从函数名称推断它会比较 rdi 和 rsi 寄存器中的字符串（目标储存在 0x4061ae），查看它们是否不相等，这个函数的结果存储在 eax 寄存器中。只需要查看 0x4061ae 地址储存字符串即可。

功能：`phase_1` 函数将输入字符串与目标字符串进行比较

第二题：

key：1 -3 13 -51 205 -819（首项不为 0，后一个数为前一个数的负四倍加一）

思路：依然首先总览一遍代码

0x5555555555be <phase_2+30> call read_six_numbers <read_six_numbers>

看来第二题的 key 是六个整数，并存储在 rsi 寄存器中指向的地址中。

在<read_six_numbers> 中：

0x0000555555556225 <+109>: test eax,eax

0x0000555555556227 <+111>: jne 0x55555555622e <read_six_numbers+118>

发现没有对第一个数进行比较，说明第一个数没有限制，下面以输入 1 为例

0x5555555555f7 <phase_2+87> imul edx, eax

乘 4

0x5555555555e4 <phase_2+68> cdqe

将 EAX （储存输入值）的值符号扩展到 RAX。

0x555555555600 <phase_2+96> add eax, edx

info registers eax eax 0x1 1

加一

最后，每一位与上一位进行比较

功能：读取六个整数，检验第一个数是否为 0，然后检验后一个是否为前一个的-4 倍+1。

第三题：

key：610 987 w（之一）

思路：

lea rsi, [rip + 0x1bf0]：将 RIP 寄存器的当前值加上 0x1bf0，然后加载到寄存器 rsi 中。这可能是为了初始化 rsi 以指向某个数据或字符串。

通过在<__isoc99_vsscanf>打断点，读取寄存器 rsi 的值，得到输入为%d %d %c

0x555555555676 <phase_3+72> cmp eax, 0x262

0x55555555568c <phase_3+94> cmp eax, 0xe9

0x0000555555555676 <+72>: cmp eax,0x262

0x000055555555567b <+77>: je 0x55555555577c <phase_3+334>

eax 储存输入的第一个整数，如果为 610，则跳转到 0x55555555577c

0x000055555555577c <+334>: mov BYTE PTR [rbp-0x1],

0x77 0x0000555555555780 <+338>: mov eax,DWORD PTR [rbp-0x10]

0x0000555555555783 <+341>: cmp eax,0x3db

0x0000555555555788 <+346>: je 0x5555555557ad <phase_3+383>

0x000055555555578a <+348>: call 0x555555555605a <explode_bomb>

DWORD PTR [rbp-0x10] 储存的是第二个输入的整数值；如果为 0x3db 即 987，跳转到

0x00005555555557ad <+383>: nop

0x00005555555557ae <+384>: movzx eax,BYTE PTR [rbp-0x11]

0x00005555555557b2 <+388>: cmp BYTE PTR [rbp-0x1],al

0x00005555555557b5 <+391>: je 0x5555555557bc <phase_3+398>

判断第三个输入是否与寄存器中 al 相等，相等则判断为成功

其余还有 233 377 o

89 144 l

1 1 o

2 3 v

5 8 e

13 21 r

34 55 f

第四题：

Key：51539607553-51539607564 中一个数

范意阳 22307110117

思路：

读取了一个 64 位的数放在 0x7fffffffde70

```
0x40182f <phase_4+59>    mov    rax, qword ptr [rbp - 0x10]
► 0x401833 <phase_4+63>    sar    rax, 0x20
```

将读取的数据右移了 32 位，储存在 rbp - 4，

输入数据的全部，储存在 rbp - 8

```
0x40183e <phase_4+74>    mov    dword ptr [rbp - 8], eax
```

```
0x401841 <phase_4+77>    cmp    dword ptr [rbp - 4], 0
0x401848 <phase_4+84>    cmp    dword ptr [rbp - 4], 0xe
```

高位需要比 0 大比 15 小

```
0x401854 <phase_4+96>    cmp    dword ptr [rbp - 8], 0
0x401860 <phase_4+108>   cmp    dword ptr [rbp - 8], 0xe
```

低位需要比 0 大比 15 小

```
0x40187a <phase_4+134>   call   hope(int)                  <hope(int)>
```

Hope 函数递归把 rdi 存储的高位的数值传输了进去

```
0x4017b8 <hope(int)+12>    mov    dword ptr [rbp - 0x14], edi
0x4017bb <hope(int)+15>    cmp    dword ptr [rbp - 0x14], 0
0x4017bf <hope(int)+19>    jne    hope(int)+28                <hope(int)+28>
```

如果不是 0 大跳转到 28

```
0x4017c8 <hope(int)+28>    mov    eax, dword ptr [rbp - 0x14]
0x4017cb <hope(int)+31>    sar    eax, 1
0x4017cd <hope(int)+33>    mov    edi, eax
0x4017cf <hope(int)+35>    call   hope(int)                   <hope(int)>
```

右移一位然后重新进入循环，一直到变成 0，给 eax 赋值为 1

```
0x4017e1 <hope(int)+53>    mov    eax, dword ptr [rbp - 4]
0x4017e4 <hope(int)+56>    imul   eax, eax
0x4017e7 <hope(int)+59>    shl    eax, 2
0x4017ea <hope(int)+62>    jmp    hope(int)+70                <hope(int)+70>
```

开始把 eax 自己与自己相乘，不断左移两位，然后返回<hope(int)+35>的下一位

```
0x40187f <phase_4+139>   cmp    eax, 0x1000000
0x401884 <phase_4+144>   setne  al
```

如果结果为 0x1000000 即可通过

因此我需要将这个函数调用四次，即我的输入（高位）需要为二进制 1110 或者 1111，来保证调用了 hope 四次，因此高位只能为 0xc

第五题：

读取了一个字符串以及一个整数，存储到了 0x9de6006425207325

RSI 0x4032e5 ◂ — 0x9de6006425207325 n/* '%s %d' */

0x40193b <phase_5+168> lea rsi, [rip + 0x19c3]

0x401942 <phase_5+175> mov rdi, rax

0x401945 <phase_5+178> call strcmp@plt

调用 strcmp 函数。它用于比较两个字符串。

Strcmp 函数调用了两个地址

0x7fffffffe4d0 用于储存输入的字符串

0x4032eb ◂ — 0x9de6809de6809de6 储存用于比较的字符串

Strump 在函数中调用了三次

在 gdb 中查看这个地址，另外两次地址为

0x4032f8 ◂ — 0x80e98080e98080e9

0x403305 ◂ — 0x86e5b286e5b286e5

在 gdb 中调取他们的信息，得到三个字符串为

```
pwndbg> x/s0x4032eb
0x4032eb:       "杀杀杀！"
```

```
pwndbg> x/s 0x4032f8
0x4032f8:       "退退退。"
```

```
pwndbg> x/s 0x403305
0x403305:       "冲冲冲~"
```

0x4018fa <phase_5+103> call worldline1::worldline1() <worldline1::worldline1()>

当看到这个函数我脑子“噔噔蹬”了一下，有类存在，怕不是要找类里面的

冲冲冲~情况：

0x401f5f <worldline2::worldline2()+23> call worldline::worldline()

调用了 worldline::worldline() 函数，是初始化 worldline2 对象，将 worldline 对象与当前 worldline2 对象关联。

0x401e94 <worldline::worldline()+12> lea rdx, [rip + 0x3efd] <vtable for worldline+16>

提醒这里是一个存储类的虚函数

0x401963 <phase_5+208> mov qword ptr [rbp - 0x18], rbx

把这个函数的指针放进了 rbp - 0x18

0x40197c <phase_5+233> mov edx, dword ptr [rbp - 0x34]

把输入值存放到了 RDX

```
0x402039 <worldline3::dmail(int)+15>
cmp    dword ptr [rbp - 0xc], 0x7e7
```

我们处理后需要是 0x7e7 才能通过

```
0x401ff6 <worldline3::worldline3()+46>    mov    qword ptr [rax + 8], 0x1124fd
```

```
   0x401eb6 <worldline::is_phase5_passable()+16>    mov    rax, qword ptr [rax + 8]
   0x401eba <worldline::is_phase5_passable()+20>    cmp    rax, 0xf423f
```

这里需要 rax 值大于 0xf423f，而我们的值为 0x1124fd 成功

而在我尝试其他两种情况的时候，发现事情出现了不一样的情况

```
0x401ef6 <worldline1::worldline1()+46>    mov    qword ptr [rax + 8], 0x8b690
```

rax + 8 被赋值成了其他值，经过尝试只有冲冲冲~才能成功
但是我弄了好久也没有弄明白，都是`<worldline1::worldline1()+46>`，但是传输的数据不一样不太懂，但是还是过了。

第六题：
Key：6 1 6 1 6 1
思路：
读取了六个整数，存放在了 0x7fffffffe510

然后进入了一个循环，

```
0x401cb1 <phase_6+59>     mov    eax, dword ptr [rbp - 4]
 ► 0x401cb4 <phase_6+62>     cdqe
   0x401cb6 <phase_6+64>     lea    rdx, [rax*4]
   0x401cbe <phase_6+72>     mov    rax, qword ptr [rbp - 0x10]
   0x401cc2 <phase_6+76>     add    rax, rdx
   0x401cc5 <phase_6+79>     mov    eax, dword ptr [rax]
   0x401cc7 <phase_6+81>     cmp    eax, 6
```

从 rax = 1 一直到 rax = 5
把输入传递给 phase_6_nums 以及之后+16 +20 等等一系列地址 0x（输入一）0000000（输入二）以这种形式

之后进入 build_target

```
0x401aab <build_target+21>    mov    qword ptr [rbp - 0x98], rdi    <phase_6_nums>
```

把输入写入 rbp - 0x98

进入 build_queue

```
0x401a22 <build_queue+8>      lea    rax, [rip + 0x49d7]      <initialNodes+64>
```

initialNodes 里存储着六个数字，按顺序为 0x14 0x3c 0x28 0x32 0xa 0x1e

get_val 里

```
 0x401a03 <get_val+24>          mov    rax, qword ptr [rbp - 0x18]
   0x401a07 <get_val+28>          mov    rax, qword ptr [rax]
   0x401a0a <get_val+31>          mov    rdx, qword ptr [rax + 8]
   0x401a0e <get_val+35>          mov    rax, qword ptr [rbp - 0x18]
   0x401a12 <get_val+39>          mov    qword ptr [rax], rdx
```

把下一个数字读取进 0x7fffffffe450 —▸ 0x4063c0 (initialNodes) ◂ — 0x3c
上一个数字放入 0x7fffffffe400 ◂ — 0x14

第二次调用

```
 0x7fffffffe400 ◂ — 0x3c00000014
0x7fffffffe450 —▸ 0x406410 (initialNodes+80) ◂ — 0x28
```

第三次调用

```
0x7fffffffe400 ◂ — 0x3c00000014
0x7fffffffe450 —▸ 0x4063e0 (initialNodes+32) ◂ — 0x32
```

一直到五个数字全部读取

```
 0x7fffffffe400 ◂ — 0x3c00000014
 0x7fffffffe408 ◂ — 0x3200000028
0x7fffffffe410 ◂ — 0x7fff0000000a
```

put_val:

```
   0x4019f3 <get_val+8>     mov    qword ptr [rbp - 0x18], rdi
   0x4019f7 <get_val+12>    mov    rax, qword ptr [rbp - 0x18]
   0x4019fb <get_val+16>    mov    rax, qword ptr [rax]
   0x4019fe <get_val+19>    mov    eax, dword ptr [rax]
   0x401a00 <get_val+21>    mov    dword ptr [rbp - 4], eax
   0x401a03 <get_val+24>    mov    rax, qword ptr [rbp - 0x18]
   0x401a07 <get_val+28>    mov    rax, qword ptr [rax]
   0x401a0a <get_val+31>    mov    rdx, qword ptr [rax + 8]、
```

寄存器状态：

```
0x7fffffffe448 —▸ 0x406400 (initialNodes+64) ◂ — 0x14
0x406400 (initialNodes+64) ◂ — 0x14
0x406400 (initialNodes+64) ◂ — 0xa
0x7fffffffe448 —▸ 0x406400 (initialNodes+64) ◂ — 0xa
 RDX  0x4063c0 (initialNodes) ◂ — 0x3c

0x7fffffffe448 —▸ 0x4063c0 (initialNodes) ◂ — 0x3c
 0x4063c0 (initialNodes) ◂ — 0x3c
0x4063c0 (initialNodes) ◂ — 0x32
0x7fffffffe448 —▸ 0x4063c0 (initialNodes) ◂ — 0x32
```

```
0x406410 (initialNodes+80) ◂— 0x28
```

put_val 输入两个数据，第一个是一个指针指向现在这个数据的指针，第二个是一个数；把第二个数放进现在这个位置，然后原来的指针指向这个位置。

即把调用的两个值位置进行交换，接下来要找什么决定了我的 put_val 的输入

```
► 0x401c31 <build_target+411>    lea    rax, [rbp - 0x78]
  0x401c35 <build_target+415>    mov    esi, edx
  0x401c37 <build_target+417>    mov    rdi, rax
  0x401c3a <build_target+420>    call   put_val                    <put_val>
```

第一个输入存储在 rbp - 0x78

```
  0x401c24 <build_target+398>    mov    rax, qword ptr [rbp - 0x60]
  0x401c28 <build_target+402>    mov    edx, dword ptr [rbp - 0x34]
  0x401c2b <build_target+405>    movsxd rdx, edx
► 0x401c2e <build_target+408>    mov    edx, dword ptr [rax + rdx*4]
  0x401c31 <build_target+411>    lea    rax, [rbp - 0x78]
  0x401c35 <build_target+415>    mov    esi, edx
  0x401c37 <build_target+417>    mov    rdi, rax
  0x401c3a <build_target+420>    call   put_val
```

第二个存储在 rax + rdx*4，其中 rdx 为计数器

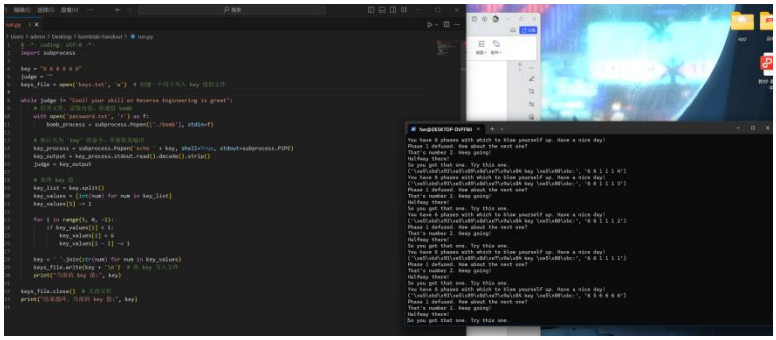
```
 RAX  0x7fffffffe400 ◂— 0x2800000032
```

我猜这是一个循环队列，get_val 和 put_val 前者把输入写进队列后者替换队列元素
0x7fffffffe400
0x7fffffffe408
0x7fffffffe410 储存这六个元素

```
  0x401a66 <check_answer+59>    shl    rax, 2
  0x401a6a <check_answer+63>    lea    rcx, [rax - 4]
► 0x401a6e <check_answer+67>    mov    rax, qword ptr [rbp - 0x18]
  0x401a72 <check_answer+71>    add    rax, rcx
  0x401a75 <check_answer+74>    mov    eax, dword ptr [rax]
  0x401a77 <check_answer+76>    cmp    edx, eax
```

通过查看寄存器的值很容易发现需要我们的队列是从小到大排列的才能通过

（看了将近十个小时 实在是找不到我的输入是怎么控制循环的，破防了；放弃正常写了）



运用 subprocess 库来控制终端的输入，在终端进入 wsl 后 python run.py 得到答案。

Secret_phase：

一开始助教提醒在第三题，我还以为是在第三题后面的某一行代码，或者里面的代码；一直 si 看了很多没用的外部函数，后来问了一下助教，助教让我注意第三题答案，结果拼出来了 overflow，“挠”的一下我就懂了。在输入后面加上 1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111over flow 后即可进入。

输入值存放在 0x7fffffffe510

存储到 rbp - 0x18

```
0x401dc1 <secret_phase+12>     mov     qword ptr [rbp - 0x18], rdi
```

把一个奇奇怪怪的值放进去了，感觉一定很有用

```
0x401dd4 <secret_phase+31>     mov     dword ptr [rbp - 4], 0xdeadc0de
0x401ddb <secret_phase+38>     lea     rdx, [rbp - 8]
*RDX  0x7fffffffe4f8 ◂— 0xdeadc0de0af5e100
```

```
   0x401e06 <secret_phase+81>     mov     eax, dword ptr [rbp - 8]（输入的十六进制）
 ► 0x401e09 <secret_phase+84>     xor     eax, dword ptr [rbp - 4]
   0x401e0c <secret_phase+87>     mov     dword ptr [rbp - 8], eax
   0x401e0f <secret_phase+90>     mov     eax, dword ptr [rbp - 8]
  0x401e12 <secret_phase+93>     cmp     eax, 0xbaadf00d
   0x401e17 <secret_phase+98>     je      secret_phase+105
```

重点就在这个 xor 上，这个数异或上 0xdeadc0de 等于 0xbaadf00d

```
#include <stdio.h>

int main()
{
    int result = 0xbaadf00d ^ 0xdeadc0de;
    printf("%d",result);
}
```

```
1677734099%
```

得到答案（隐藏意外的简单）

感受：

属于一个看山不是山-看山是山-看山又不是山了　这么一个过程，从最开始第一题，进度超级缓慢；然后二三四好像掌握了一些技巧，虽然过程曲折，但是结果是好的。到第五题，虽然有些地方不理解，但是整体是能懂在干什么；然后到了第六题，写的时间比我前四个题目加一起都多，到最后还是 get_val 和 put_val 能懂，但是就是不懂循环是怎么操纵的；做第六题的时候，晚上做梦都在地址倒来倒去。最后 secret_phase，在第三题看了好多好多遍，没想到在这么一个位置，有挺有意思。

总之感谢助教，学到了很多东西。