BombLab

分别有六个实验，给出了主函数，但是其他几个函数被定义在phases.h头文件中，没有调试信息。

```
#include <stdio.h>
#include <stdlib.h>
#include "support.h"
#include "phases.h"

/*
 * Note to self: Remember to erase this file so my victims will have no
 * idea what is going on, and so they will all blow up in a
 * spectaculary fiendish explosion. -- Dr. Evil
 */

FILE *infile;

int main(int argc, char *argv[])
{
    char *input;

    /* Note to self: remember to port this bomb to Windows and put a
     * fantastic GUI on it. */

    /* When run with no arguments, the bomb reads its input lines
     * from standard input. */
    if (argc == 1) {
    infile = stdin;
    }

    /* When run with one argument <file>, the bomb reads from <file>
     * until EOF, and then switches to standard input. Thus, as you
     * defuse each phase, you can add its defusing string to <file> and
     * avoid having to retype it. */
    else if (argc == 2) {
    if (!(infile = fopen(argv[1], "r"))) {
        printf("%s: Error: Couldn't open %s\n", argv[0], argv[1]);
        exit(8);
    }
    }

    /* You can't call the bomb with more than 1 command line argument. */
    else {
    printf("Usage: %s [<input_file>]\n", argv[0]);
    exit(8);
    }

    /* Do all sorts of secret stuff that makes the bomb harder to defuse. */
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");
```

```
    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();
    printf("That's number 2.  Keep going!\n");

    /* I guess this is too easy so far.  Some more complex code will
     * confuse people. */
    input = read_line();
    phase_3(input);
    phase_defused();
    printf("Halfway there!\n");

    /* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
    input = read_line();
    phase_4(input);
    phase_defused();
    printf("So you got that one.  Try this one.\n");

    /* Round and 'round in memory we go, where we stop, the bomb blows! */
    input = read_line();
    phase_5(input);
    phase_defused();
    printf("Good work!  On to the next...\n");

    /* This phase will never be used, since no one will get past the
     * earlier ones.  But just in case, make this one extra hard. */
    input = read_line();
    phase_6(input);
    phase_defused();

    /* Wow, they got it!  But isn't something... missing?  Perhaps
     * something they overlooked?  Mua ha ha ha ha! */

    return 0;
}
```

phase_1, phase_2, phase_3, phase_4, phase_5和phase_6对应着关键代码，从read_line输入的值应该会进入phase_n函数中进行校验。所以逆向的重点就是这六个函数。

第一关

```
0x400ee9 <phase_1+9>     call   strings_not_equal <strings_not_equal>
    rdi: 0x603780 (input_strings) ◂— 'aaaaaaaaaaaaaaaaaaaaaa'
    rsi: 0x402400 ◂— outsd  dx, dword ptr [rsi] /* 'Border relations with Canada have never been better.' */
    rdx: 0x1
    rcx: 0x17
```

第一关，跟进到phase_1中，步进到strings_note_equal函数，我猜测是要将输入值与后面的字符串做校验。

```
0x40134d <strings_not_equal+21>    call   string_length <string_length>
    rdi: 0x402400 ← outsd  dx, dword ptr [rsi] /* 'Border relations with Canada have never been better.' */
    rsi: 0x402400 ← outsd  dx, dword ptr [rsi] /* 'Border relations with Canada have never been better.' */
    rdx: 0x603797 (input_strings+23) ← 0x0
    rcx: 0x17
```

strings_note_equal函数首先校验了我们输入的字符串的长度，然后校验了0x402400处字符串的长度，将r12d寄存器的值（保留我们输入字符串长度）和rax寄存器的值（保留0x402400地址处字符串的长度）做了比较。

然后检查我们输入的字符串是否为空：

```
0x40135c <strings_not_equal+36>    movzx  eax, byte ptr [rbx]
0x40135f <strings_not_equal+39>    test   al, al
0x401361 <strings_not_equal+41>    je     strings_not_equal+80 <strings_not_equal+80>
```

接下来就校验字符串是否相等，rbx和rbp的值递增，逐位校验：

```
0x401363 <strings_not_equal+43>    cmp    al, byte ptr [rbp]
0x401366 <strings_not_equal+46>  ✔ je     strings_not_equal+58 <strings_not_equal+58>
 ↓
0x401372 <strings_not_equal+58>    add    rbx, 1
0x401376 <strings_not_equal+62>    add    rbp, 1
0x40137a <strings_not_equal+66>    movzx  eax, byte ptr [rbx]
0x40137d <strings_not_equal+69>    test   al, al
0x40137f <strings_not_equal+71>    jne    strings_not_equal+50 <strings_not_equal+50>
```

最终结果如下：

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

第二关

```
► 0x400f05 <phase_2+9>     call   read_six_numbers <read_six_numbers>
      rdi: 0x6037d0 (input_strings+80) ← 0x363636363636 /* '666666' */
      rsi: 0x7fffffffdf50 ← 0x0
      rdx: 0x2
      rcx: 0x6
```

read_six_number应该是要输入六个数字，先跟进看一下。

```
► 0x40148a <read_six_numbers+46>    call   __isoc99_sscanf@plt <__isoc99_sscanf@plt>
      s: 0x6037d0 (input_strings+80) ← 0x363636363636 /* '666666' */
      format: 0x4025c3 ← '%d %d %d %d %d %d'
      vararg: 0x7fffffffdf50 ← 0x0
```

```
0x40148a <read_six_numbers+46>    call   __isoc99_sscanf@plt <__isoc99_sscanf@plt>

0x40148f <read_six_numbers+51>    cmp    eax, 5
0x401492 <read_six_numbers+54>    jg     read_six_numbers+61 <read_six_numbers+61>
```

eax作为sscanf的返回值，做了一个校验，首先要大于5。可以看到，六个整型变量是被读入栈中的。

```
  0x401499 <read_six_numbers+61>    add    rsp, 0x18
  0x40149d <read_six_numbers+65>    ret
   ↓
► 0x400f0a <phase_2+14>             cmp    dword ptr [rsp], 1
  0x400f0e <phase_2+18>             je     phase_2+52 <phase_2+52>
   ↓
```

做完校验之后，返回到phase_2函数中，设置rsp指针指向地址处的值为1。

```
  0x400f17 <phase_2+27>     mov    eax, dword ptr [rbx - 4]
  0x400f1a <phase_2+30>     add    eax, eax
  0x400f1c <phase_2+32>     cmp    dword ptr [rbx], eax
► 0x400f1e <phase_2+34>   ✔ je     phase_2+41 <phase_2+41>
   ↓
  0x400f25 <phase_2+41>     add    rbx, 4
  0x400f29 <phase_2+45>     cmp    rbx, rbp
  0x400f2c <phase_2+48>     jne    phase_2+27 <phase_2+27>
```

```
  0x400f17 <phase_2+27>    mov    eax, dword ptr [rbx - 4]
  0x400f1a <phase_2+30>    add    eax, eax
  0x400f1c <phase_2+32>    cmp    dword ptr [rbx], eax
► 0x400f1e <phase_2+34>  ✓ je     phase_2+41 <phase_2+41>
    ↓
  0x400f25 <phase_2+41>    add    rbx, 4
  0x400f29 <phase_2+45>    cmp    rbx, rbp
  0x400f2c <phase_2+48>    jne    phase_2+27 <phase_2+27>
```

```
RBX  0x7fffffffdf54 ◂— 0x6037d000000002
```

rbx指向写入int类型变量的栈中的地址

```
0x400f17 <phase_2+27>    mov    eax, dword ptr [rbx - 4]
0x400f1a <phase_2+30>    add    eax, eax
0x400f1c <phase_2+32>    cmp    dword ptr [rbx], eax
0x400f1e <phase_2+34>    je     phase_2+41 <phase_2+41>
  ↓
0x400f25 <phase_2+41>    add    rbx, 4
0x400f29 <phase_2+45>    cmp    rbx, rbp
0x400f2c <phase_2+48>    jne    phase_2+27 <phase_2+27>
```

每次校验一个数，校验完之后，rbx的值+4，因为一个int类型变量大小就是四个字节，直到等于rbp的值的时候。

这个循环，其实就要求我们输入的数字是2的n-1次方。

```
Phase 1 defused. How about the next one?
1   2   4   8   16   32
That's number 2.  Keep going!
```

第三关

```
  0x400f6a <phase_3+39>     cmp    dword ptr [rsp + 8], 7
► 0x400f6f <phase_3+44>     ja     phase_3+106 <phase_3+106>
```

```
  0x400fa6 <phase_3+99>     mov    eax, 0x147
  0x400fab <phase_3+104>    jmp    phase_3+123 <phase_3+123>
```

其实比上一个好分析，步进之后，很容易就能找到关键代码。

```
7   327
Halfway there!
```

第三关通过。

第四关

```
0x40102e <phase_4+34>    cmp    dword ptr [rsp + 8], 0xe
0x401033 <phase_4+39>    jbe    phase_4+46 <phase_4+46>
```

需要输入两个数，第一个数由条件来看，需要小于0xe。第二个数是要和第一个数做一个运算，关键代码段在func4函数中，最后要实现的结果是：函数的返回值为0，函数的[rsp+0xc]地址处的值是否为0（要求我们输入的第二个数为0）。

```
► 0x401048 <phase_4+60>    call   func4 <func4>
      rdi: 0xe
      rsi: 0x0
      rdx: 0xe
      rcx: 0x0
```

func4函数我基本还原如下，比较丑，但是能看清楚。

```
int func4(int rdi,int rsi,int rdx,int rcx)
{
    int var1;
```

```
    var1=rdx-rsi;
    rcx=var1;
    rcx=rcx>>31;
    var1+=rcx;
    var1=var1/2;
    rcx=var1+rsi;   // rcx=(rdx-rsi)/2+rsi == rdi
    if(rcx <= rdi){
        var1=0;
        if(rcx >= rdi){
            return var1;
        }else{
            rsi=rcx+1;
            func4(rdi,rsi,rdx,rcx);
        }
    }else{
        rdx=rcx-1;
        func4(rdi,rsi,rdx,rcx);
    }
}
```

直接看汇编，关系比较凌乱，但是仔细梳理一下，就会发现rdi寄存器的值从来没有被改变，而要满足rcx<=rdi，rcx>=rdi，只能rcx==rdi。带入数学关系式，解得rdi==7。

所以要输入的两个数等于7和0。

```
7   0
So you got that one.  Try this one.
```

第五关

![image-20210207231006343](C:\Users\wolf\AppData\Roaming\Typora\typora-user-images\image-20210207231006343.png)

```
► 0x401073 <phase_5+17>    mov    qword ptr [rsp + 0x18], rax
  0x401078 <phase_5+22>    xor    eax, eax
  0x40107a <phase_5+24>    call   string_length <string_length>

  0x40107f <phase_5+29>    cmp    eax, 6
```

输入的字符串长度要为6，接下来，通过循环对字符串的值进行处理。

```
0x40108b <phase_5+41>     movzx  ecx, byte ptr [rbx + rax]
0x40108f <phase_5+45>     mov    byte ptr [rsp], cl
0x401092 <phase_5+48>     mov    rdx, qword ptr [rsp]
0x401096 <phase_5+52>     and    edx, 0xf <0x402231>
0x401099 <phase_5+55>     movzx  edx, byte ptr [rdx + 0x4024b0]
0x4010a0 <phase_5+62>     mov    byte ptr [rsp + rax + 0x10], dl
0x4010a4 <phase_5+66>     add    rax, 1
0x4010a8 <phase_5+70>     cmp    rax, 6
0x4010ac <phase_5+74>     jne    phase_5+41 <phase_5+41>
```

```
RSP  0x7fffffffdf60 —▸ 0x402231 (__libc_csu_init+33) ◂— 0x8948f8247c894cf0
```

rsp保留的值的最低位会被修改。处理过的字节序会被保留在rsp+0x10地址处。

```
► 0x4010ae <phase_5+76>    mov    byte ptr [rsp + 0x16], 0
  0x4010b3 <phase_5+81>    mov    esi, 0x40245e
  0x4010b8 <phase_5+86>    lea    rdi, [rsp + 0x10]
  0x4010bd <phase_5+91>    call   strings_not_equal <strings_not_equal>
```

最关键的校验肯定在string_not_equal函数中。

```
► 0x4010bd <phase_5+91>     call   strings_not_equal <strings_not_equal>
       rdi: 0x7fffffffdf70 ◂— 0x726569756461 /* 'aduier' */
       rsi: 0x40245e ◂— insb   byte ptr [rdi], dx /* 'flyers' */
       rdx: 0x72
       rcx: 0x36
```

flyers这个字符串，是目标字符串，是我们的输入值通过上面的算法处理，最终生成的值。上面的算法，是输入字符，最低字节与0xf异或，作为索引，确定0x4024b0这串字符串中的字符。

```
pwndbg> x/s 0x4024b0
0x4024b0 <array.3449>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

写个脚本逆向一下算法就可以了，脚本和最终结果如下。

```
s,new_s='flyers',''
target='maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?'
for i in range(len(s)):
    new_s+=chr((target.index(s[i]) & 0xf) + 0x60)

print(new_s)
```

```
ionefg
Good work!  On to the next...
```

第六关

按照要求是要读取6个数字。

```
  0x401117 <phase_6+35>     mov    eax, dword ptr [r13]
► 0x40111b <phase_6+39>     sub    eax, 1
  0x40111e <phase_6+42>     cmp    eax, 5
  0x401121 <phase_6+45>     jbe    phase_6+52 <phase_6+52>
```

所要写的数不能比6大，而且这些数要互不相等。

```
   0x401160 <phase_6+108>    mov    edx, ecx
   0x401162 <phase_6+110>    sub    edx, dword ptr [rax]
   0x401164 <phase_6+112>    mov    dword ptr [rax], edx
   0x401166 <phase_6+114>    add    rax, 4
   0x40116a <phase_6+118>    cmp    rax, rsi
   0x40116d <phase_6+121>    jne    phase_6+108 <phase_6+108>
```

以上代码转换为C代码

```
int nums[6]={...}
for(i=0;i<6;i++){
    nums[i]=7-nums[i];
}
```

继续步进，看到了一个定义在bss段的node1的变量。

```
0x40119f <phase_6+171>    mov    eax, 1
0x4011a4 <phase_6+176>    mov    edx, node1 <0x6032d0>
0x4011a9 <phase_6+181>    jmp    phase_6+130 <phase_6+130>
 ↓
0x401176 <phase_6+130>    mov    rdx, qword ptr [rdx + 8]
0x40117a <phase_6+134>    add    eax, 1
0x40117d <phase_6+137>    cmp    eax, ecx
0x40117f <phase_6+139>    jne    phase_6+130 <phase_6+130>
 ↓
0x401176 <phase_6+130>    mov    rdx, qword ptr [rdx + 8]
```

跟进观察一下0x6032d0这片内存地址。

```
pwndbg> x/12xg 0x6032d0
0x6032d0 <node1>:       0x000000010000014c      0x00000000006032e0
0x6032e0 <node2>:       0x00000002000000a8      0x00000000006032f0
0x6032f0 <node3>:       0x000000030000039c      0x0000000000603300
0x603300 <node4>:       0x00000004000002b3      0x0000000000603310
0x603310 <node5>:       0x00000005000001dd      0x0000000000603320
0x603320 <node6>:       0x00000006000001bb      0x0000000000000000
```

非常漂亮的一个单链表结构。

剪不断，理还乱，休息休息再看.....