bomblab 报告

总分	phase_1	phase_2	phase_3	phase_4	phase_5	phase_6	secret_phase
7	1	1	1	1	1	1	1

解题报告

phase_1

```
Just let it all out of you. Don't be swayed!
```

函数上来开栈帧空间，之后lea了一个非常奇怪的值作为strings_not_equal函数的参数，并且是作为第二个参数（在rsi寄存器）。

于是挨个看rsi和rdi里面指针指向的东西，发现能字符串打开，并且rdi存的是输入字符串，rsi是另一个字符串，经过验证发现这个字符串就是答案。

代码大致如下：

```
void phase_1(char *input) {
    const char *target_string = "预定义的目标字符串";、

    // 比较输入字符串和目标字符串
    if (strings_not_equal(input, target_string)) {
        explode_bomb();
    }
    return;
}
```

phase_2

```
1 3 1 2 3 5
```

这个程序先清空了栈帧中的6个地址，推测为读入地址，又因为函数read_six_numbers，猜测需要输入6个数字。

同时输入两串可疑的数字并且存到了0x48(%rsp)和0x40(%rsp)。之后把这两串数字存储的指针给到%rdi并且调用string length函数，结果相当于这两串数字二进制长度除8，猜测这两串数字是字符串。

之后函数循环遍历了这个字符串，将字符的ASCII码减去41之后加到了原本清空的6个地址中，猜测为字符统计。之后比较字符统计出的数和输入的数，只需要输入和字符统计数一一对应的数就可以拆除bomb。

代码大致如下：

```
void phase_2(void) {
    int numbers[6] = {0};
    int counts[6] = {0};
    const char *string = "BECDAEBEFFFDBFF";

    read_six_numbers(numbers);
```

```
    if (numbers[0] < 0) {
        explode_bomb();
    }

    for (int i = 0; i < string_length(string); i++) {
        char c = string[i];
        if (c >= 'A' && c <= 'F') {
            counts[c - 'A']++;
        }
    }

    for (int i = 0; i < 6; i++) {
        if (counts[i] != numbers[i]) {
            explode_bomb();
        }
    }
    return;
}
```

phase_3

```
4 11
```

在scanf的时候函数使用了模式字符串，查看字符串可以知道要输入两个数字。

查看之后的代码可以知道第一个数字不能输入大于7，之后函数会根据输入的第一个数字进行地址的跳转，且跳转地址相差4字节，猜测跳转根据是一个跳转表delta，原函数为switch函数。

只需要找到对应的跳转点，根据跳转代码计算出eax的值。函数最后拿第二个输入值和eax相比，则第二个值只需要和跳转后算出的eax相对应就可以了。

代码大致如下：

```
void phase_3(const char *input) {
    int x, y;
    int sscanf_result;
    int y_expected;
    sscanf_result = sscanf(input, "%d %d", &x, &y);

    if (sscanf_result < 1) {
        explode_bomb();
    }

    if (x > 7) {
        explode_bomb();
    }

    switch (x) {
        case 0:
            y_expected = 0x1F4 - delta_1; // 500 - delta_1
            break;
        case 1:
```

```
            y_expected = 0x1eb - delta_1; // 491 - delta_1
            break;
        case 2:
            y_expected = 0x1a1 - delta_1; // 417 - delta_1
            break;
        case 3:
            y_expected = 0x238 - delta_1; // 568 - delta_1
            break;
        case 4:
            y_expected = 0x116 - delta_1; // 278 - delta_1
            break;
        case 5:
            y_expected = 0x3d0 - delta_1; // 976 - delta_1
            break;
        case 6:
            y_expected = 0x264 - delta_1; // 612 - delta_1
            break;
        case 7:
            y_expected = 0;
            break;
        default:
            explode_bomb();
    }

    if (y < 0) {
        explode_bomb();
    }

    if (y != y_expected) {
        explode_bomb();
    }

    return;
}
```

phase_4

```
213 3
```

输入的时候scanf匹配了一个模式字符串，查看这个模式字符串发现是两个数字但是和要求的数据量不同，但是scanf之后又要求eax值是2，也就是说在phase_4的时候应该可以输入超过两个数字的数据，但是这个函数只需要前两个数据，这一点成为之后进入secret phase的关键。

之后将第二个参数减去2且要求减去后小于2，于是尝试输入3。之后发现这个数字和数字7作为两个参数传入了递归函数fun4。

fun4有两个递归出口：在7每次减一达到1（返回y），或者小于等于0（返回总加和的结果）。之后会同时调用两次递归，这个过程类似于计算斐波那契数列，但这个函数返回的是y参数，并且每次都多加上一次y值。func4函数代码大致如下：

```
int func4(int x, int y) {
    int result = 0;
```

```
    if (x <= 0) {
        return result;
    }

    if (x == 1) {
        return y;
    }

    int temp = y + 1;
    int sum1 = func4(x - 1, temp) + y;
    int sum2 = func4(x - 2, temp);

    result = sum1 + sum2;

    return result;
}
```

尝试调用这个func4函数，调用func4(7,3)就能得到结果213，为第一个输入的值。

phase_4的源代码的伪代码大致如下：

```
void phase_4(const char *input) {
    int var0 = 0, var1 = 0;
    int num_read = 0;

    num_read = sscanf(input, "%d %d", &var0, &var1);

    if (num_read != 2) {
        explode_bomb();
    }

    if ((var0 - 2) > 2) {
        explode_bomb();
    }

    int func4_result = func4(7, var0);

    if (func4_result != var1) {
        explode_bomb();
    }
    return;
}
```

phase_5

```
-6 38
```

输入的时候模式字符串也是“%d %d”则证明输入项为两个数字，之后将第一个参数和0相比，发现这个数据需要小于0。之后函数截取了这个负数的最低16进制位并且重新传入了(%rsp)，这个数字相当于原有负数加上16。之后验证了这个参数不能是15，也就是说第一个数不能是-1。

之后进入循环，这个循环大致的代码如下（其中edx和ecx初始值都是0）：

```
    while (1) {
        edx += 1;
        int value = array[var1];
        ecx += value;
        if (value == 0xF) {
            break;
        }
    }
```

之后会计算循环的次数是不是等于5，所以知道array的值非常重要。查看array的数据如下：

```
<array.0>:      0x0a    0x00    0x00    0x00    0x02    0x00    0x00    0x00
<array.0+8>:    0x0e    0x00    0x00    0x00    0x07    0x00    0x00    0x00
<array.0+16>:   0x08    0x00    0x00    0x00    0x0c    0x00    0x00    0x00
<array.0+24>:   0x0f    0x00    0x00    0x00    0x0b    0x00    0x00    0x00
<array.0+32>:   0x00    0x00    0x00    0x00    0x04    0x00    0x00    0x00
<array.0+40>:   0x01    0x00    0x00    0x00    0x0d    0x00    0x00    0x00
<array.0+48>:   0x03    0x00    0x00    0x00    0x09    0x00    0x00    0x00
<array.0+56>:   0x06    0x00    0x00    0x00    0x05    0x00    0x00    0x00
```

指向15的是array[6]，指向6的是array[14]，指向14的是array[2]，指向2的是array[1]，指向1的是array[10]，也就是说变量1加16是10，变量1是-6。之后计算1，2，14，6，15的和得出第二个变量是38。

整个函数代码大致如下：

```
void phase_5(const char *input) {
    int var1, var2;
    int num_read;

    num_read = sscanf(input, "%d %d", &var1, &var2);
    if (num_read <= 1) {
        explode_bomb();
    }
    if (var1 >= 0) {
        explode_bomb();
    }

    var1 &= 0xF;

    if (var1 == 0xF) {
        explode_bomb();
    }

    int ecx = 0;
    int edx = 0;
    while (1) {
        edx += 1;
        int value = array[var1];
        ecx += value;
        if (value == 0xF) {
            break;
        }
    }

    var1 = 0xF;
```

```
    if (edx != 5) {
        explode_bomb();
    }

    if (ecx != var2) {
        explode_bomb();
    }
}
```

phase_6

```
2 3 5 6 1 4
```

函数上来先是初始化（保存了一些callee保存寄存器，调用栈金丝雀之类的），之后把0x10(%rsp)的地址作为数组指针，调用read_six_numbers。

之后进入了一个二层的循环，这个循环内层是在17be—17e4行，外层是187c—18a3行，这个循环作用是比较这六个输入的数字两两是否相等。

之后函数跳到17e6—1805行，这个循环是将这六个输入的数字转换成7-x的形式。

之后函数跳到了1807—1839行，这些行对一个链表进行了一系列操作，找到了任意一个数字在链表中对应的节点，并且把这些节点的指针放到了一个顺序表中。

183b—187a行是把这些节点重新排列，按照顺序表的顺序重新连接链表的指针。

18a8—18c0行遍历了整个链表，检测重新排列后的链表是否按照降序排列。我们可以按照节点先降序排列，发现排列为5 4 2 1 6 3，之后将这些数字取1-x就可以知道答案。

节点的值查看如下：

```
(gdb) x/24x $rdx
0x5555555591e0 <node1>: 0x000002b8      0x00000001      0x555591f0      0x00005555
0x5555555591f0 <node2>: 0x00000308      0x00000002      0x55559200      0x00005555
0x555555559200 <node3>: 0x00000220      0x00000003      0x55559210      0x00005555
0x555555559210 <node4>: 0x0000037f      0x00000004      0x55559220      0x00005555
0x555555559220 <node5>: 0x0000039d      0x00000005      0x55559130      0x00005555
0x555555559230: 0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/4x $rdx
0x555555559130 <node6>: 0x00000297      0x00000006      0x00000000      0x00000000
```

原式的代码大致如下：

```
void phase_6() {
    int numbers[6];
    read_six_numbers(numbers);
    for (int i = 0; i < 6; i++) {
        if (numbers[i] < 1 || numbers[i] > 6) {
            explode_bomb();
        }
        for (int j = i + 1; j < 6; j++) {
            if (numbers[i] == numbers[j]) {
                explode_bomb();
            }
        }
```

```
    }

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

    struct node {
        int value;
        struct node* next;
    };

    struct node* node_pointers[6];

    for (int i = 0; i < 6; i++) {
        int index = numbers[i];
        struct node* p = &node1;
        for (int j = 1; j < index; j++) {
            p = p->next;
        }
        node_pointers[i] = p;
    }

    for (int i = 0; i < 5; i++) {
        node_pointers[i]->next = node_pointers[i + 1];
    }
    node_pointers[5]->next = NULL;

    struct node* current = node_pointers[0];
    while (current->next != NULL) {
        if (current->value > current->next->value) {
            explode_bomb();
        }
        current = current->next;
    }
}
```

secret_phase

```
phase4 : 213 3 Genshin Impact is an open-world action RPG developed by miHoYo
secret_phase : 00222220003310221113110033300002212211230002O2
```

首先发现代码里面能调用secret phase的只有phase defused函数，而这个函数0730d(%rip)是6的时候才跳转，猜测是需要解除第六个phase的时候才能跳转。

跳转后函数拿出一个字符串变量进行处理，访问这个字符串发现这个字符串就是phase4的字符串。之后进入一个循环，这个循环在遍历字符串的时候检查到两个空格的时候跳出，否则return。跳出后把这个字符串从第二个空格处截断，只保留后面的字符串，并且将这个字符串和另一个字符串"Genshin Impact is an open-world action RPG developed by miHoYo"一同调用strings_not_equal函数，因此只需要让phase4字符串之后加上这个字符串就可以进入secret_phase。

进入phase之后函数先是在rsp处存储了一个数组5 3 2 4，之后读入一行数据，并且要求这行数据长度小于70。之后进入fun7函数，要求这个函数的返回值是1，随后结束整个函数。secret_phase函数的代码大致如下：

```
void secret_phase() {
    char *input_line;
    int input_length;
    int result;
    int arr[5] = {5, 3, 2, 4};
    input_line = read_line();
    input_length = string_length(input_line);
    if (input_length > 0x46) {
        explode_bomb();
    }

    int *array = (int *)arr;
    int index = 0;
    int depth = 0;

    result = fun7(array, index, depth, input_line);

    if (result == 1) {
    } else {
        explode_bomb();
    }
}
```

进入fun7函数后，函数会声明两个数组，这两个数组大致是这样的：

```
arr1 = {0, 0, 1, -1};
arr2 = {1, -1, 0, 0};
```

这个数组非常像二维数组的跳转表。之后函数进行了一系列计算，这个计算大致上是把rsi这个寄存器之前存储的数字加7之后对7取余，随后将最低1-3位的数据存在rsi，4-6位存进rbx。随后规定了这个函数在rsi=rdx=7，r9=4的时候返回input_line是否读完的信息。从之后的代码可以知道仅有在这四个条件同时满足的时候代码才会返回1。

随后函数读入了一个line1的变量，经过查看可以知道这个变量的值是一个二维数组，如下（数据都是16进制）：

```
      0  1  2  3  4  5  6  7
line0 00 00 00 01 01 01 01 01
line1 01 01 00 00 00 02 00 01
line2 01 00 00 01 01 01 00 01
line3 01 01 00 01 05 00 00 01
line4 03 00 00 01 01 00 01 01
line5 01 01 00 00 00 00 00 01
line6 01 01 01 04 01 01 00 00
```

而函数则是找到了第rbx行，rsi列的数据，如果是1，就直接返回0，如果是0，就更新index的数据递归继续查找，如果这个数据和要查找的数据一样，就给r9加一。要查找的4个数据就是arr的5 3 2 4。至于查找的路径，正是根据input_line，按照跳转表：0：向右，1：向左，2：向下，3：向上。因此只需要找到从0，0不踩到1的情况下，依次找到四个数，并且最后到7，7的路径就行。

反馈/收获/感悟/总结

花的时间还挺长的，主要是因为没听完汇编的课程就急着做题导致的:(

觉得最有趣的一道题是secret。但是phase3和4，尤其是4，可以在完全看不懂代码的情况下做出来，我觉得可以改进下。

参考的重要资料