Bomblab 实验报告

一、实验目标

熟悉 gdb 调试、反汇编等等

二、实验方法

使用 gdb 调试的各种命令，尽量在不引爆炸弹的前提下发现正确答案

三、实验结果

1.拆弹程度：6 个 phase 全部解开
2.我的密码是：
Why make trillions when we could make... billions?
0 1 1 2 3 5
1 w 142
12 43
mfcdhg
4 6 5 1 2 3

四、详细过程

1.phase_1

- 观察这段代码，容易看出，主要是在调用 strings_not_equal 这个函数后，根据其返回结果判断是否触发爆炸，因此应该输入一个与地址 0x555555557150 处存放的字符串相同的字符串

```
B+> 0x555555555440 <phase_1>          sub    $0x8,%rsp
   0x555555555444 <phase_1+4>         lea    0x1cfc(%rip),%rsi        # 0x555555557150
   0x55555555544b <phase_1+11>        callq  0x555555555a1a <strings_not_equal>
   0x555555555450 <phase_1+16>        test   %eax,%eax
```

- ni 指令，移动到 call <strings_not_equal>函数处
- 运行指令 x/s 0x555555557150，得到答案

```
(gdb) x/s 0x555555557150
0x555555557150: "Why make trillions when we could make... billions?"
(gdb)
```

2.phase_2

- 观察代码，看到有一个函数 read_six_numbers，猜测本题要输入 6 个符合要求的数字

```
   0x555555555482 <phase_2+25>        callq  0x555555555cf0 <read_six_numbers>
   0x555555555487 <phase_2+30>        cmpl   $0x0,(%rsp)
   0x55555555548b <phase_2+34>        jne    0x555555555494 <phase_2+43>
```

- 查看 read_six_numbers 函数代码，发现在此函数内调用了 sscanf。
  输入 x/s 0x555555557409 查看其参数，根据结果可知 sscanf 从我的输入中读入 6 个整数。因此改变输入为 6 个空格分隔的整数（我在这里输入 012345）

```
0x555555555d09 <read_six_numbers+25>    lea    0x8(%rsi),%r8
0x555555555d0d <read_six_numbers+29>    lea    0x16f5(%rip),%rsi        # 0x555555557409
0x555555555d14 <read_six_numbers+36>    mov    $0x0,%eax
0x555555555d19 <read_six_numbers+41>    callq  0x555555555160 <__isoc99_sscanf@plt>
```

```
(gdb) x/s 0x555555557409
0x555555557409: "%d %d %d %d %d %d"
```

- 寄存器 rsp 中存放的是我输入的六个数字的首地址，根据标注的这一行，可以推出第一个数字是 0。继续下一行，0x4(%rsp)是我输入的第二个数字，应该等于 1。

```
0x555555555482 <phase_2+25>     callq  0x555555555cf0 <read_six_numbers>
>0x555555555487 <phase_2+30>     cmpl   $0x0,(%rsp)
0x55555555548b <phase_2+34>     jne    0x555555555494 <phase_2+43>
0x55555555548d <phase_2+36>     cmpl   $0x1,0x4(%rsp)
0x555555555492 <phase_2+41>     je     0x555555555499 <phase_2+48>
```

- 跳转到标注的这一行，分析这段代码，是一个循环。
- 分析循环内容，可以发现第三个数等于前两个数相加，且之后遵循相同的规律。因此推断出答案应该是 0 1 1 2 3 5

```
>0x555555555499 <phase_2+48>     mov    %rsp,%rbx
0x55555555549c <phase_2+51>     lea    0x10(%rsp),%rbp
0x5555555554a1 <phase_2+56>     jmp    0x5555555554ac <phase_2+67>
0x5555555554a3 <phase_2+58>     add    $0x4,%rbx
0x5555555554a7 <phase_2+62>     cmp    %rbp,%rbx
0x5555555554aa <phase_2+65>     je     0x5555555554bd <phase_2+84>
0x5555555554ac <phase_2+67>     mov    0x4(%rbx),%eax
0x5555555554af <phase_2+70>     add    (%rbx),%eax
0x5555555554b1 <phase_2+72>     cmp    %eax,0x8(%rbx)
0x5555555554b4 <phase_2+75>     je     0x5555555554a3 <phase_2+58>
0x5555555554b6 <phase_2+77>     callq  0x555555555cb4 <explode_bomb>
```

- 进行验证，确定答案就是 0 1 1 2 3 5

```
0 1 1 2 3 5
That's number 2.  Keep going!
```

3.phase_3

- 同上通过 sscanf 函数的参数，得到输入的字符串的形式应该是“%d %c %d”
  并且可以看出，%rsp+0xf 中存放输入的字符，%rsp+0x10 中存放第一个整数，%rsp+0x14 中存放第二个整数。尝试输入“1 a 2”，验证猜测正确。（代码截图不再重复贴出）
- 继续运行，看出第一个参数必须小于等于 7

```
>0x555555555504 <phase_3+52>     cmpl   $0x7,0x10(%rsp)
 0x555555555512 <phase_3+57>     ja     0x555555555624 <phase_3+331>
```

- 继续执行， 跳转到这里，可以看出第二个参数应该等于 0x8e（即 142）

```
 0x555555555555 <phase_3+124>    mov    $0x77,%eax
>0x55555555555a <phase_3+129>    cmpl   $0x8e,0x14(%rsp)
 0x555555555562 <phase_3+137>    je     0x55555555562e <phase_3+341>
 0x555555555568 <phase_3+143>    callq  0x555555555cb4 <explode_bomb>
```

- 同理可以看出，字符参数的值应该为寄存器 al 中存放的值：0x77(即 119)，去 ASCII 码表中寻找。对应的字符为 w，从而确定答案为 1 w 142

```
>0x55555555562e <phase_3+341>    cmp    %al,0xf(%rsp)
 0x555555555632 <phase_3+345>    jne    0x555555555649 <phase_3+368>
```

- 重新运行并输入 1 w 142，phase_3 成功通过

```
1 w 142
Halfway there!
```

4.phase_4

- 同样的，根据 sscanf 的参数，输入格式应为 “%d %d”，且两个参数分别存放在%rsp 和%rsp+4 的内存地址中。尝试输入“1 2”。（代码部分与之前重复，省略）
- 继续执行，可以看出第一个参数应该小于等于 14（0xe）

```
>0x5555555556b4 <phase_4+45>     cmpl   $0xe,(%rsp)
 0x5555555556b8 <phase_4+49>     jbe    0x5555555556bf <phase_4+56>
 0x5555555556ba <phase_4+51>     callq  0x555555555cb4 <explode_bomb>
 0x5555555556bf <phase_4+56>     mov    $0xe,%edx
```

- 注意到在 phase_4 中调用了函数 func4，且其三个参数分别存放在%edi、%esi、%edx

```
 0x5555555556bf <phase_4+56>     mov    $0xe,%edx
 0x5555555556c4 <phase_4+61>     mov    $0x0,%esi
 0x5555555556c9 <phase_4+66>     mov    (%rsp),%edi
>0x5555555556cc <phase_4+69>     callq  0x555555555555 <func4>
```

- 在 func4 处设置断点，根据汇编代码，发现这是一个递归函数，并据此写出 c 语言代码。

```
int func4(int x, int esi, int edx){
    int ebx = (((edx - esi) >> 31) + (edx - esi))/2 + esi;
    if(ebx > x){
        edx = ebx - 1;
        ebx += func4(x, esi, edx);
        return ebx;
    }else if(ebx < x){
        esi = ebx + 1;
        ebx += func4(x, esi, edx);
        return ebx;
    }
    return ebx;
}
```

- 回到 phase_4 函数查看剩余代码，容易看出 func4 函数的返回值应该等于 0x2b（43），且输入的第二个参数也应该等于 43。
- 还记得之前判断过第一个参数应该小于等于 14，所以在 c 代码中从 1 到 14 循环调用 func4 函数（初值为 x，0，14），寻找返回 43 的数

```
>0x5555555556d2 <phase_4+74>     cmp    $0x2b,%eax
 0x5555555556d4 <phase_4+77>     jne    0x5555555556dd <phase_4+86>
 0x5555555556d6 <phase_4+79>     cmpl   $0x2b,0x4(%rsp)
 0x5555555556db <phase_4+84>     je     0x5555555556e2 <phase_4+91>
```

```
int main(){
    for(int i = 0; i <= 14; i++){
        cout << i << "  " << func4(i, 0, 14) << endl;
    }
    return 0;
}
```

```
7  7
8  35
9  27
10  37
11  18
12  43
13  31
14  45
```

- 从输出结果中可以看出，12 应该就是我们寻找的第一个参数，故本题答案为“12 43”

```
12 43
So you got that one.  Try this one.
```

5.phase_5

- 该函数中调用了一个名叫 string_length 的函数，并且要求其返回值为 6。所以应该输入一个长度为 6 的字符串，尝试输入“abcdef”，且我的输入被放在寄存器 rbx 内

```
0x555555555714 <phase_5+24>     callq  0x5555555559fd <string_length>
0x555555555719 <phase_5+29>     cmp    $0x6,%eax
0x55555555571c <phase_5+32>     jne    0x555555555773 <phase_5+119>
0x55555555571e <phase_5+34>     mov    $0x0,%eax
```

- 继续向下看，是一段循环。分析其作用，可以看出这个循环是以 6 个输入字符的 ASCII 码的低 4 位作为下标，找到它在%rcx，（即 array.0）中对应的字符存到 %rsp+1，%rsp+2 一直到%rsp+6。

```
>0x555555555572a <phase_5+46>     movzbl (%rbx,%rax,1),%edx
 0x55555555572e <phase_5+50>     and    $0xf,%edx
 0x555555555731 <phase_5+53>     movzbl (%rcx,%rdx,1),%edx
 0x555555555735 <phase_5+57>     mov    %dl,0x1(%rsp,%rax,1)
 0x555555555739 <phase_5+61>     add    $0x1,%rax
 0x55555555573d <phase_5+65>     cmp    $0x6,%rax
 0x555555555741 <phase_5+69>     jne    0x55555555572a <phase_5+46>
```

```
(gdb) x/s $rcx
0x5555555571e0 <array.0>:       "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

- 继续执行，见到了熟悉的 strings_not_equal 函数，根据 phase_1 的分析，我们查看地址 0x5555555571b1 中的内容，并且应该让%rsp+1 处存放的字符串与之相等

```
0x55555555574d <phase_5+81>     lea    0x1a63(%rip),%rsi        # 0x5555555571b7
0x555555555754 <phase_5+88>     callq  0x555555555a1a <strings_not_equal>
```

```
(gdb) x/s 0x5555555571b7
0x5555555571b7: "bruins"
```

- 换言之，我们应该在 array.0，即 “maduiersnfotvbyl”（对应 0123456789ABCDEF） 中对应的找到 “bruins” 六个字母所对应的下标位置（从 0~15），即 D 6 3 4 8 7，然后去 ASCII 码表里找哪六个字符的 ASCII 码的低四位分别是 D 6 3 4 8 7 的二进制形式即可。
- 最终找到对应的字符为 mfcdhg

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

6.phase_6

- Phase_6 中，根据前面分析过的 read_six_numbers 函数，猜测答案为 6 个整数。尝试输入 1 2 3 4 5 6 并发现我输入的数字分别存放在%r14，%r14+0x4……%r14+0x14 内。

```
0x5555555557ae <phase_6+40>     mov    %r14,%rsi
0x5555555557b1 <phase_6+43>     callq  0x555555555cf0 <read_six_numbers>
```

- 继续执行这段代码，是一个大循环套小循环，最终实现的效果是：对每个参数都要求其小于等于 6，并调用各自的小循环，保证与后面的参数不相等，也就是说六个数互相不等，且都大于 0 小于等于 6。大循环结束后，跳转到 phase_6+105

```
<phase_6+255>     add    $0x1,%r15
<phase_6+259>     add    $0x4,%r14
<phase_6+263>     mov    %r14,%rbp
<phase_6+266>     mov    (%r14),%eax
<phase_6+269>     sub    $0x1,%eax
```

```
<phase_6+272>     cmp    $0x5,%eax
<phase_6+275>     ja     0x5555555557c7 <phase_6+65>
<phase_6+281>     cmp    $0x5,%r15d
<phase_6+285>     jg     0x5555555557ef <phase_6+105>
<phase_6+291>     mov    %r15,%rbx
<phase_6+294>     jmpq   0x5555555557de <phase_6+88>
```

```
<phase_6+75>     add    $0x1,%rbx
<phase_6+79>     cmp    $0x5,%ebx
<phase_6+82>     jg     0x555555555885 <phase_6+255>
<phase_6+88>     mov    0x0(%r13,%rbx,4),%eax
<phase_6+93>     cmp    %eax,0x0(%rbp)
<phase_6+96>     jne    0x5555555557d1 <phase_6+75>
<phase_6+98>     callq  0x555555555cb4 <explode_bomb>
<phase_6+103>    jmp    0x5555555557d1 <phase_6+75>
```

- 接下来的一段代码是用 7 依次减去原来的值并替换，我的输入 123456 变成 654321，不再展示
- 继续执行，在代码中发现一个特殊的地址<node1>，将其输出，发现这应该是一个依次连接的链表。但比较特殊的是 node6，它的地址在 0x555555559210 处。

```
(gdb) x/24xw 0x555555559330
0x555555559330 <node1>: 0x000002d0      0x00000001      0x55559340      0x00005555
0x555555559340 <node2>: 0x0000024a      0x00000002      0x55559350      0x00005555
0x555555559350 <node3>: 0x000002f6      0x00000003      0x55559360      0x00005555
0x555555559360 <node4>: 0x0000003c      0x00000004      0x55559370      0x00005555
0x555555559370 <node5>: 0x00000044      0x00000005      0x55559210      0x00005555
```

```
(gdb) x/4xw 0x555555559210
0x555555559210 <node6>: 0x000001ff      0x00000006      0x00000000      0x00000000
```

- 分析代码，发现它其实是按照我转变后的六个参数作为 node 的下标，按照其顺序依次链接 node(参数 1)->node(参数 2)……->node(参数 6)，即按照我目前的输入，它会链接出 node6->node5->node4->node3->node2->node1，并将它们分别存放在%rsp+0x30，%rsp+0x38……%rsp+0x58，然后建立链接

```
> 0x555555555810 <phase_6+138>    mov    $0x0,%esi
  0x555555555815 <phase_6+143>    mov    0x10(%rsp,%rsi,4),%ecx
  0x555555555819 <phase_6+147>    mov    $0x1,%eax
  0x55555555581e <phase_6+152>    lea    0x3b0b(%rip),%rdx        # 0x555555559330 <node1>
  0x555555555825 <phase_6+159>    cmp    $0x1,%ecx
  0x555555555828 <phase_6+162>    jle    0x555555555835 <phase_6+175>
  0x55555555582a <phase_6+164>    mov    0x8(%rdx),%rdx
  0x55555555582e <phase_6+168>    add    $0x1,%eax
  0x555555555831 <phase_6+171>    cmp    %ecx,%eax
  0x555555555833 <phase_6+173>    jne    0x55555555582a <phase_6+164>
  0x555555555835 <phase_6+175>    mov    %rdx,0x30(%rsp,%rsi,8)
  0x55555555583a <phase_6+180>    add    $0x1,%rsi
```

```
  0x55555555583e <phase_6+184>    cmp    $0x6,%rsi
  0x555555555842 <phase_6+188>    jne    0x555555555815 <phase_6+143>
```

- 继续执行，这段代码又是一个循环，它要求刚刚排好顺序的链表中结点的数值部分应该恰好也是降序排列，从大到小。否则就会爆炸。

```
  0x5555555558b1 <phase_6+299>    mov    0x8(%rbx),%rbx
  0x5555555558b5 <phase_6+303>    sub    $0x1,%ebp
  0x5555555558b8 <phase_6+306>    je     0x5555555558cb <phase_6+325>
> 0x5555555558ba <phase_6+308>    mov    0x8(%rbx),%rax
  0x5555555558be <phase_6+312>    mov    (%rax),%eax
  0x5555555558c0 <phase_6+314>    cmp    %eax,(%rbx)
  0x5555555558c2 <phase_6+316>    jge    0x5555555558b1 <phase_6+299>
  0x5555555558c4 <phase_6+318>    callq  0x555555555cb4 <explode_bomb>
  0x5555555558c9 <phase_6+323>    jmp    0x5555555558b1 <phase_6+299>
```

- 根据刚才列出的 6 个 node，可以降序排列出：
  node3(2f6)->node1(2d0)->node2(24a)->node6(1ff)->node5(44)->node4(3c)
- 所以对应的变换后的输入应该是：3 1 2 6 5 4
  所以我真正的输入，即答案应该用 7 减这个序列，得到 4 6 5 1 2 3
  验证答案正确，phase_6 解决。