实验报告

刘雅迪
计 26

Phase_1

考查的两字符串是否相等。

通过 disas 指令得到 phase_1 的汇编代码，从代码中可以得知在<+11>行中 call 了 strings_not_equal 函数，猜测作用是判断输入的字符串是否等于目标字符串，而在 call 函数之前将一个地址的值存到了%rsi 寄存器中，故使用 x/s 指令查看地址中的值，从而得到 phase_1 的输入。

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
=> 0x0000555555555419 <+0>:     sub    $0x8,%rsp
   0x000055555555541d <+4>:     lea    0x1d2c(%rip),%rsi        # 0x555555557150
   0x0000555555555424 <+11>:    call   0x555555555921 <strings_not_equal>
   0x0000555555555429 <+16>:    test   %eax,%eax
   0x000055555555542b <+18>:    jne    0x555555555432 <phase_1+25>
   0x000055555555542d <+20>:    add    $0x8,%rsp
   0x0000555555555431 <+24>:    ret
   0x0000555555555432 <+25>:    call   0x555555555a21 <explode_bomb>
   0x0000555555555437 <+30>:    jmp    0x55555555542d <phase_1+20>
End of assembler dump.
(gdb) x/s 0x555555557150
0x555555557150: "Why make trillions when we could make... billions?"
```

Phase_1 具体操作

Phase_2

考查循环。

由汇编代码中 call 了 read_six_numbers 函数可知，输入的格式是六个数字。

寄存器%ebx 中存了当前数与前一个数的差值，%ebx 初始值为 1，而在循环中%ebx 会每次加 1。从第一个数 0 开始，通过%rbp 指针的移动，来看输入的数是否满足程序要求的差值。当%ebx 等于 6 时，说明 6 个数全部判断完毕，%rsp 指针恢复，弹出%rbx 和%rbp，程序结束。

```
   0x000055555555545e <+37>:    add    $0x1,%ebx
   0x0000555555555461 <+40>:    add    $0x4,%rbp
   0x0000555555555465 <+44>:    cmp    $0x6,%ebx
   0x0000555555555468 <+47>:    je     0x55555555547b <phase_2+66>
   0x000055555555546a <+49>:    mov    %ebx,%eax
   0x000055555555546c <+51>:    add    0x0(%rbp),%eax
   0x000055555555546f <+54>:    cmp    %eax,0x4(%rbp)
   0x0000555555555472 <+57>:    je     0x55555555545e <phase_2+37>
```

Phase_2 中循环主体

Phase_3

考查 switch-case 语句。

由于 call 了 sscanf 函数，所以通过查%rsi 存的值可以得知 phase_3 的输入格式，如下图：

```
(gdb) x/sb 0x5555555571ae
0x5555555571ae: "%d %c %d"
```

查 phase_3 的输入格式

sscanf 会将参数 str 的字符串根据参数 format 字符串来转换格式并格式化数据，转换后的结果存于对应的参数内，成功则返回参数数目，失败则返回 0，返回值保存在%eax 中。

寄存器%rdi :存一开始的 input，%rsi 存输入格式“%d %c %d”，

所以输入的三个数分别依次由寄存器%rdx，%rcx，%r8 储存，它们的地址分别为 0xc(%rsp)、0x7(%rsp)、0x8(%rsp)，如下图：

```
0x0000000000001486 <+4>:     lea    0x7(%rsp),%rcx
0x000000000000148b <+9>:     lea    0xc(%rsp),%rdx
0x0000000000001490 <+14>:    lea    0x8(%rsp),%r8
```

寄存器的地址分配

由第<+41>和<+46>行可知第一个数%rdx 的值小于 7，如下图：

```
0x00000000000014ab <+41>:    cmpl   $0x7,0xc(%rsp)
0x00000000000014b0 <+46>:    ja     0x15bb <phase_3+313>
```

%rdx 的值范围

第<+52>~<+70>行是 switch-case 语句的条件判断部分，根据第一个数的不同跳转到不同的 case 语句，而跳转到的具体地址由%rax 的地址决定，如下图：

```
0x00000000000014b6 <+52>:    mov    0xc(%rsp),%eax
0x00000000000014ba <+56>:    lea    0x1cff(%rip),%rdx        # 0x31c0
0x00000000000014c1 <+63>:    movslq (%rdx,%rax,4),%rax
0x00000000000014c5 <+67>:    add    %rdx,%rax
0x00000000000014c8 <+70>:    jmp    *%rax
```

switch-case 条件判断部分

假设第一个数%rdx 为 1，由于我不知道程序运行中%rax 的具体地址，所以我通过 gdb 调试工具来辅助我判断程序跳转到哪个 case 语句。

给 phase_3 函数和 explode_bomb 函数设置断点，输入第一个数为 1 的三个数，在函数 explode_bomb 执行前通过 info reg 指令查看寄存器%rax 的地址，发现为 0x76，说明程序跳转到了第<+113>~<+126>行，如下图：

```
0x00005555555554f3 <+113>:   mov    $0x76,%eax
0x00005555555554f8 <+118>:   cmpl   $0xf1,0x8(%rsp)
0x0000555555555500 <+126>:   je     0x5555555555c5 <phase_3+323>
```

%rdx 为 1 时对应的 case 语句

此时第三个数%r8 = 0xf1 = 241

而由汇编代码第<+323>~<+329>行可知%rcx 的 ACSII 码为 0x76 即 118，对应的字符是 v，如下图：

```
0x00005555555555c5 <+323>:   cmp    %al,0x7(%rsp)
0x00005555555555c9 <+327>:   jne    0x5555555555d0 <phase_3+334>
0x00005555555555cb <+329>:   add    $0x18,%rsp
```

%rcx 的值

所以 phase_3 的输入为 1 v 241，这只是其中的一种情况，改变第一个数的值，后两个数的值也会跟着变化。

Phase_4

考查了递归函数。

输入格式的判断如 phase_3，如下图：

```
(gdb) x/sb 0x55555555732f
0x55555555732f: "%d %d"
```

查 phase_4 的输入格式

说明输入的是两个数字，分别储存在寄存器%rdx 和%rcx 中，其地址分别为 0x8(%rsp)和 0xc(%rsp)。

由第<+53>~<+62>行可知 phase_4 调用了函数 func4，传入的两个参数分别是 6 和%rcx，即调用函数 func4(6, %rcx)，如下图：

```
0x0000000000001643 <+53>:    mov    0xc(%rsp),%esi
0x0000000000001647 <+57>:    mov    $0x6,%edi
0x000000000000164c <+62>:    call   0x15d7 <func4>
```

func4 函数调用

查看 func4 函数的汇编代码，如下图，发现它在函数内部分不同情况调用了自身，说明这是个递归函数，%edi 和%esi 的初始值分别为 6 和%rcx。

```
Dump of assembler code for function func4:
   0x00000000000015d7 <+0>:     mov    $0x0,%eax
   0x00000000000015dc <+5>:     test   %edi,%edi
   0x00000000000015de <+7>:     jle    0x160d <func4+54>
   0x00000000000015e0 <+9>:     push   %r12
   0x00000000000015e2 <+11>:    push   %rbp
   0x00000000000015e3 <+12>:    push   %rbx
   0x00000000000015e4 <+13>:    mov    %edi,%ebx
   0x00000000000015e6 <+15>:    mov    %esi,%ebp
   0x00000000000015e8 <+17>:    mov    %esi,%eax
   0x00000000000015ea <+19>:    cmp    $0x1,%edi
   0x00000000000015ed <+22>:    je     0x1608 <func4+49>
   0x00000000000015ef <+24>:    lea    -0x1(%rdi),%edi
   0x00000000000015f2 <+27>:    call   0x15d7 <func4>
   0x00000000000015f7 <+32>:    lea    (%rax,%rbp,1),%r12d
   0x00000000000015fb <+36>:    lea    -0x2(%rbx),%edi
   0x00000000000015fe <+39>:    mov    %ebp,%esi
   0x0000000000001600 <+41>:    call   0x15d7 <func4>
   0x0000000000001605 <+46>:    add    %r12d,%eax
   0x0000000000001608 <+49>:    pop    %rbx
   0x0000000000001609 <+50>:    pop    %rbp
   0x000000000000160a <+51>:    pop    %r12
   0x000000000000160c <+53>:    ret
   0x000000000000160d <+54>:    ret
End of assembler dump.
```

递归函数 func4 的汇编代码

由于 phase_4 使用了递归函数，故我根据汇编代码写出了它的 C++函数，方便计算 func(6, %rcx)的值，如下图：

```cpp
#include <iostream>

int f(int x,int y){
    if (x == 0) return 0;
    if (x == 1) return y;
    return f(x-1,y) + f(x-2,y) + y;
}

int main() {
    // Example usage:
    int result = f(6, 3);
    std::cout << "Result: " << result << std::endl;

    return 0;
}

```

func4 函数的 C++代码

```
0x0000000000001632 <+36>:    mov    0xc(%rsp),%eax
0x0000000000001636 <+40>:    sub    $0x2,%eax
0x0000000000001639 <+43>:    cmp    $0x2,%eax
0x000000000000163c <+46>:    jbe    0x1643 <phase_4+53>
0x000000000000163e <+48>:    call   0x1a21 <explode_bomb>
```

%rcx 取值范围

从第<+36>~<+48>行可知第二个数%rcx 的取值应小于等于 4，不妨令其等于 3，算出第一个数%rdx 为 60。这只是其中的一种情况，改变第二个数的值，第一个数也会跟着发生变化。

Phase_5

考查数组。

通过 call 了 string_length 函数可知输入的是字符串且长度为 6，如下图：

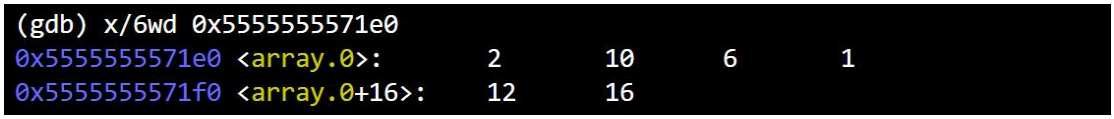
```
0x0000000000001667 <+4>:     call   0x1904 <string_length>
0x000000000000166c <+9>:     cmp    $0x6,%eax
```

phase_6 输入格式

由第<+51>行 cmp　　$0x32,%ecx 可知最后%ecx 要等于 0x32 即 50，分析汇编代码可知%ecx 初始值为 0，每次会加上数组 0~5 位的值。

Phase_5 汇编代码

而从<+26>行可知%rsi 储存的是数组的起始地址，通过 gdb 查看数组中的数，如下图：



数组中的数

可知数组 array[0]~array[5]的值分别为 2、10、6、1、12、16。

在这六个数里面凑六个相加等于 50 的数，如 50=16+16+12+2+2+2，对应的数组下标偏移量为 5 5 4 0 0 0。

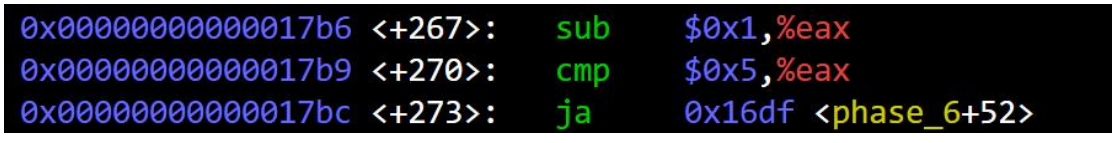
而由汇编代码第<+36>and 0xf %edx 可知，只取原字符的低四位，如字符'A'，ASCII 码为 0x41，取低四位后得到 0x01，所以查 ACSII 后可知，输入的字符串可以是 eedppp。

Phase_6

考查链表和结构体。

Phase_6 的汇编代码很长，我看了好久之后才大致看懂。可以分几个部分来分析：

首先由 call read_six_numbers 函数可知 phase_6 输入的也是六个数字。



输入数字的范围

从<+270>行得知%eax 减 1 后≤5，所以输入的数均≤6。

第一部分：

```
0x000055555555570f <+100>:   mov    $0x0,%esi
0x0000555555555714 <+105>:   mov    0x40(%rsp,%rsi,4),%ecx
0x0000555555555718 <+109>:   mov    $0x1,%eax
0x000055555555571d <+114>:   lea    0x3bcc(%rip),%rdx        # 0x5555555592f0 <node1>
0x0000555555555724 <+121>:   cmp    $0x1,%ecx
0x0000555555555727 <+124>:   jle    0x555555555734 <phase_6+137>
0x0000555555555729 <+126>:   mov    0x8(%rdx),%rdx
0x000055555555572d <+130>:   add    $0x1,%eax
0x0000555555555730 <+133>:   cmp    %ecx,%eax
0x0000555555555732 <+135>:   jne    0x555555555729 <phase_6+126>
0x0000555555555734 <+137>:   mov    %rdx,0x10(%rsp,%rsi,8)
0x0000555555555739 <+142>:   add    $0x1,%rsi
0x000055555555573d <+146>:   cmp    $0x6,%rsi
0x0000555555555741 <+150>:   jne    0x555555555714 <phase_6+105>
```

通过<+114>行我们可以得到 nodes 的值和地址，如下图：

```
(gdb) x/20 0x5555555592f0
0x5555555592f0 <node1>: 74      1       21248   0
0x555555559300 <node2>: 455     2       21264   0
0x555555559310 <node3>: 384     3       21280   0
0x555555559320 <node4>: 576     4       21296   0
0x555555559330 <node5>: 98      5       20976   0
```

nodes 的值和地址

这是个结构如下图的链表：

```
struct node{
    int val;
    int number;
    struct node* next;
}
// node[1]->next = node[2],...以此类推
```

链表结构

而上图这段汇编代码包含了内外两个循环，设输入的第 i 个数字为 n[i-1]，第 i 个结构体为 node[i]。第<+137>行可知 0x10(%rsp)是 node[1]的地址，而<+105>行可知%ecx = n[i]。

第<+109>~<+135>是内循环，可以写成如下 for 循环：

```
*rdx = node[1]
for(eax = 1, eax!=ecx, eax++)
    rdx = rdx->next
```

内部 for 循环

表明指针移动了 n[i] - 1 位，到了 node[n[i]]结点，将这个结点压入栈，外循环则是说明另 5 个 n[i]也这样做。所以从这段汇编代码我们可以知道程序想要将 node 结点们重新排序。第二部分：

```
0x00005555555557cd <+290>:   mov    0x8(%rbx),%rbx
0x00005555555557d1 <+294>:   sub    $0x1,%ebp
0x00005555555557d4 <+297>:   je     0x5555555557e7 <phase_6+316>
0x00005555555557d6 <+299>:   mov    0x8(%rbx),%rax
0x00005555555557da <+303>:   mov    (%rax),%eax
0x00005555555557dc <+305>:   cmp    %eax,(%rbx)
0x00005555555557de <+307>:   jge    0x5555555557cd <phase_6+290>
0x00005555555557e0 <+309>:   call   0x555555555a21 <explode_bomb>
0x00005555555557e5 <+314>:   jmp    0x5555555557cd <phase_6+290>
```

这段代码通过比较(%rbx)和 0x8(%rbx)的值，告诉我们新的 nodes 应该是个降序。而 node[4]>node[2]>node[3]>node[5]>node[1]，故顺序为 4 2 3 5 1。

第三部分：

```
0x00000000000016ce <+35>:    mov    %r15,%r12
0x00000000000016d1 <+38>:    mov    $0x1,%r14d
0x00000000000016d7 <+44>:    mov    %r15,%r13
0x00000000000016da <+47>:    jmp    0x17b0 <phase_6+261>
0x00000000000016df <+52>:    call   0x1a21 <explode_bomb>
0x00000000000016e4 <+57>:    cmp    $0x5,%r14d
0x00000000000016e8 <+61>:    jle    0x17c8 <phase_6+285>
0x00000000000016ee <+67>:    mov    0x8(%rsp),%rdx
0x00000000000016f3 <+72>:    add    $0x18,%rdx
0x00000000000016f7 <+76>:    mov    $0x7,%ecx
0x00000000000016fc <+81>:    mov    %ecx,%eax
0x00000000000016fe <+83>:    sub    (%r12),%eax
0x0000000000001702 <+87>:    mov    %eax,(%r12)
0x0000000000001706 <+91>:    add    $0x4,%r12
0x000000000000170a <+95>:    cmp    %rdx,%r12
0x000000000000170d <+98>:    jne    0x16fc <phase_6+81>
```

由<+76>~<+87>可知，程序将原始的数据进行了处理，%r12 = 7 - %r12，故最后还需用 7 减去得到的数字，才是我们输入的原始数据。

用 7 分别减去上述的 4 2 3 5 1，得到 3 5 4 2 6，而第一个数为 1，故输入的数据为 1 3 5 4 2 6。

实验感想

汇编的学习对我来说和之前学习微积分/线代等数学的感觉完全不一样，我经常在课堂上听的一头雾水，或者以为自己听懂的部分实际上没有完全听懂，导致作业下不了笔。写这次实验前我是十分恐惧的，因为觉得自己很多知识还没有搞懂，不知道能不能把实验写完，于是拖了很久才开始打开实验的文件。

在写 phase_2 时我就被难住了，我又回去把 C 语言-3 的课件以及回放看了一遍，才理解了一点栈指针%rsp 和帧指针%rbp 的使用，也读懂了 phase_2 其实考查的是一个循环，而在编输入数字的过程中我又因为搞错了地址的加和值的加而卡在了这一关，最后还是求助同学才知道我错在哪。

平心而论，这次实验对我来说还是很有难度的，由于每个人的代码都不一样，我也无法在网上找到参考资料，只能根据自己的理解编写输入的数据。又因为对汇编知识的不太熟悉，导致对别人来说可能一天就能写完的实验我写了好几天。但是在看到完成最后一个 phase 后终端出现的”congratulations”的时候，还是很高兴很有成就感的。我完成了一个对我来说有点难的任务，并且在这个过程中我又回顾理解了好几遍课件内容，还因为课件部分内容写的有点简略而又去阅读了 CSAPP 课本。随着对汇编知识点和实验的熟悉，明显感觉到解决后面的 phase 比解决前面的更得心应手，虽然后面的 phase 更难更复杂。

通过本次实验我学到了很多处理汇编语言的技巧，也巩固/重新学习了课堂上没有搞懂的知识，收获很大，也非常感谢助教和老师将本次实验的截止时间设置的那么晚，让我有机会完成本次实验。

就我来看，bomb 实验本身已经很精巧了，重要的知识点都考查到了，我暂时想不到什么创新点。但是我觉得可以给每个 phase 的输入排除一些“随便猜就可以猜中的答案”，要求同学们必须认真阅读理解汇编代码或者使用 gdb 调试工具后才能得到 phase 的答案。因为我有朋友在写 phase_6 时一开始就随便输入了 6 5 4 3 2 1，结果就过了，感觉对于 phase_6 这个比较复杂的 phase 来说有点荒谬与不公平。