CSAPP Lab2 Bomb实验记录

实验环境 : Ubuntu 22.04 X86_64架构
实验日期 : 2024.4.14 - 2024.4.21

预备知识

GDB有关指令

指令	简称	描述
r	run	开始执行程序，直到下一个断点或程序结束
q	quit	退出 GDB 调试器
ni	nexti	执行下一条指令，但不进入函数内部
si	stepi	执行当前指令，如果是函数调用则进入函数
b	break	在指定位置设置断点
c	cont	从当前位置继续执行程序，直到下一个断点或程序结束
p	print	打印变量的值
x	打印内存中的值	
disas	反汇编当前函数或指定的代码区域	
layout asm		显示汇编代码视图
layout regs		显示当前的寄存器状态和它们的值

实验任务开始前的准备

反编译汇编码

在做实验之前我们下载的`bomb`压缩文件只有`bomb`可执行程序,以及不完整的`bomb.cpp`文件,所以我们必须先反编译出`.asm`文件,才能窥探程序的内在逻辑。

```
objdump -d bomb > bomb.asm
```

利用上面的程序得到`bomb.asm`文件。

简单看一下编译后得出的main函数里面是个什么构造，帮助我们判断程序大体分几部分。

```
0000000000400da0 <main>:
  400e1e: bf 38 23 40 00        mov    $0x402338,%edi
  400e23: e8 e8 fc ff ff        call   400b10 <puts@plt>
```

```
400e28: bf 78 23 40 00          mov    $0x402378,%edi
400e2d: e8 de fc ff ff          call   400b10 <puts@plt>
400e32: e8 67 06 00 00          call   40149e <read_line>
400e37: 48 89 c7                mov    %rax,%rdi
400e3a: e8 a1 00 00 00          call   400ee0 <phase_1>
400e3f: e8 80 07 00 00          call   4015c4 <phase_defused>
400e44: bf a8 23 40 00          mov    $0x4023a8,%edi
400e49: e8 c2 fc ff ff          call   400b10 <puts@plt>
400e4e: e8 4b 06 00 00          call   40149e <read_line>
400e53: 48 89 c7                mov    %rax,%rdi
400e56: e8 a1 00 00 00          call   400efc <phase_2>
400e5b: e8 64 07 00 00          call   4015c4 <phase_defused>
400e60: bf ed 22 40 00          mov    $0x4022ed,%edi
400e65: e8 a6 fc ff ff          call   400b10 <puts@plt>
400e6a: e8 2f 06 00 00          call   40149e <read_line>
400e6f: 48 89 c7                mov    %rax,%rdi
400e72: e8 cc 00 00 00          call   400f43 <phase_3>
400e77: e8 48 07 00 00          call   4015c4 <phase_defused>
400e7c: bf 0b 23 40 00          mov    $0x40230b,%edi
400e81: e8 8a fc ff ff          call   400b10 <puts@plt>
400e86: e8 13 06 00 00          call   40149e <read_line>
400e8b: 48 89 c7                mov    %rax,%rdi
400e8e: e8 79 01 00 00          call   40100c <phase_4>
400e93: e8 2c 07 00 00          call   4015c4 <phase_defused>
400e98: bf d8 23 40 00          mov    $0x4023d8,%edi
400e9d: e8 6e fc ff ff          call   400b10 <puts@plt>
400ea2: e8 f7 05 00 00          call   40149e <read_line>
400ea7: 48 89 c7                mov    %rax,%rdi
400eaa: e8 b3 01 00 00          call   401062 <phase_5>
400eaf: e8 10 07 00 00          call   4015c4 <phase_defused>
400eb4: bf 1a 23 40 00          mov    $0x40231a,%edi
400eb9: e8 52 fc ff ff          call   400b10 <puts@plt>
400ebe: e8 db 05 00 00          call   40149e <read_line>
400ec3: 48 89 c7                mov    %rax,%rdi
400ec6: e8 29 02 00 00          call   4010f4 <phase_6>
400ecb: e8 f4 06 00 00          call   4015c4 <phase_defused>
400ed0: b8 00 00 00 00          mov    $0x0,%eax
400ed5: 5b                      pop    %rbx
400ed6: c3                      ret
```

可以看到`phase_1`——`phase_6`个函数执行完后，程序将进入`phase_defused`，拆弹就完成了。

配置.gdbinit脚本

由于使用gdb进入程序后，一不小心就会触发炸弹，导致我们看不到程序的执行过程，所以我们要提前打好断点，方便一步步拆弹。 创建.gdbinit文件，并写入下面的文本

```
# 设置初始读入文本
set args psol.txt

b phase_1
b phase_2
b phase_3
b phase_4
b phase_5
b phase_6

r
```

这里我们对phase_1——phase_6每个函数的起始命令地址打了断点，方便我们单步执行程序。

Phase_1

我们开始尝试完成第一个phase。

简单分析asm源码

```
0000000000400ee0 <phase_1>:
  400ee0: 48 83 ec 08          sub    $0x8,%rsp
  400ee4: be 00 24 40 00       mov    $0x402400,%esi
  400ee9: e8 4a 04 00 00       call   401338 <strings_not_equal>
  400eee: 85 c0                test   %eax,%eax
  400ef0: 74 05                je     400ef7 <phase_1+0x17>
  400ef2: e8 43 05 00 00       call   40143a <explode_bomb>
  400ef7: 48 83 c4 08          add    $0x8,%rsp
  400efb: c3                   ret
```

我们可以看到，在phase_1中，主要进行了以下几个操作：

- 1.压栈8个字节，创建局部变量

由下图可见这其实是为main函数中传入的寄存器%rdi中的数据创建临时变量（栈上）空间

```
  400e37: 48 89 c7             mov    %rax,%rdi
  400e3a: e8 a1 00 00 00       call   400ee0 <phase_1>
```

- 2.将一个地址为$0x402400的常量，放入寄存器%esi

- 3.将`%rsp`中存储的临时变量，和寄存器`%esi`存储的程序常量一起传给函数`<strings_not_equal>`~~（多么一目了然的函数名，由此见得有一个良好的程序命名风格是一件帮人帮己的事情...虽然我没有）~~
- 4.检测%eax的值是否为0,为零ZF置为1,否则置为0
- 5.如果ZF为1，则跳转到ret，不然炸弹bomb

这样我们就基本锁定了我们的拆弹目标——拿出来`$0x402400`里面的常量看一看。

gdb单步调试读取数据

通过命令行输入

```
gdb bomb
```

进入bomb程序，此时我们发现由于`ans.txt`文件中没有答案文本，所以需要我们手动输入一个字符串。

```
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
Breakpoint 1 at 0x400ee0
Breakpoint 2 at 0x400efc
Breakpoint 3 at 0x400f43
Breakpoint 4 at 0x40100c
Breakpoint 5 at 0x401062
Breakpoint 6 at 0x4010f4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

随便输入一个字符串，不妨输入`asc`，然后我们就能顺利的到达第一个断点处。

```
which to blow yourself up. Have a nice day!
123456

Breakpoint 1, 0x0000000000400ee0 in phase_1 ()
(gdb)
```

此时我们输入下面两条指定来观察程序运行到的指令地址，以及寄存器中的内容。

```
layout asm
layout regs
```

~~看上去还挺酷炫的~~

```
[ Register Values Unavailable ]

B+> 0x400ee0 <phase_1>       sub    $0x8,%rsp
    0x400ee4 <phase_1+4>     mov    $0x402400,%esi
    0x400ee9 <phase_1+9>     call   0x401338 <strings_not_equal>
    0x400eee <phase_1+14>    test   %eax,%eax
    0x400ef0 <phase_1+16>    je     0x400ef7 <phase_1+23>
    0x400ef2 <phase_1+18>    call   0x40143a <explode_bomb>
    0x400ef7 <phase_1+23>    add    $0x8,%rsp

multi-thre Thread 0x7ffff7f9d7 In: phase_1                L??   PC: 0x400ee0
(gdb) layout regs
(gdb)
```

然后我们再打印一下`$0x402400`地址里面的常量答案就不言而喻了。

```
x/s 0x402400
```

---

alt text

可以看到密码是`Border relations with Canada have never been better.`

~~这里Linux输入文本后要手动换行,不然io流会少读入一个.~~

输入这个密码到`ans.txt`文本，取消phase_1断点，再重新运行程序。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
```

显示phase_1已经被拆除，可以开始输入phase_2了。

phase_2

我们继续完成第二个phase

分析asm代码

```
400efc: 55                      push   %rbp
400efd: 53                      push   %rbx
400efe: 48 83 ec 28             sub    $0x28,%rsp
400f02: 48 89 e6                mov    %rsp,%rsi
400f05: e8 52 05 00 00          call   40145c <read_six_numbers>
400f0a: 83 3c 24 01             cmpl   $0x1,(%rsp)
400f0e: 74 20                   je     400f30 <phase_2+0x34>
400f10: e8 25 05 00 00          call   40143a <explode_bomb>
400f15: eb 19                   jmp    400f30 <phase_2+0x34>
400f17: 8b 43 fc                mov    -0x4(%rbx),%eax
400f1a: 01 c0                   add    %eax,%eax
400f1c: 39 03                   cmp    %eax,(%rbx)
400f1e: 74 05                   je     400f25 <phase_2+0x29>
400f20: e8 15 05 00 00          call   40143a <explode_bomb>
400f25: 48 83 c3 04             add    $0x4,%rbx
400f29: 48 39 eb                cmp    %rbp,%rbx
400f2c: 75 e9                   jne    400f17 <phase_2+0x1b>
400f2e: eb 0c                   jmp    400f3c <phase_2+0x40>
400f30: 48 8d 5c 24 04          lea    0x4(%rsp),%rbx
400f35: 48 8d 6c 24 18          lea    0x18(%rsp),%rbp
400f3a: eb db                   jmp    400f17 <phase_2+0x1b>
400f3c: 48 83 c4 28             add    $0x28,%rsp
400f40: 5b                      pop    %rbx
400f41: 5d                      pop    %rbp
400f42: c3                      ret
```

- 可以看到第五行的函数名`<read_six_numbers>`,也就是说这个部分程序要读入6个数字
- `cmpl $0x1,(%rsp)` 也就是栈顶的值与1比较,由于c系是从右往左入栈,也就是说我们第一个数字要为1

我们继续往下看,程序跳转到命令地址`400f30`处,然后又经过了一系列操作,再跳转回`400f17`
这个时候,我们就应该意识到,这很有可能是一个循环结构,所以我们要找到这个循环的终止条件

```
400f1c: 39 03                   cmp    %eax,(%rbx)
400f1e: 74 05                   je     400f25 <phase_2+0x29>
400f20: e8 15 05 00 00          call   40143a <explode_bomb>
```

程序要求循环执行过程中,`%eax`,`%rbx`的值必须相同

```
400f29: 48 39 eb                cmp    %rbp,%rbx
400f2c: 75 e9                   jne    400f17 <phase_2+0x1b>
400f2e: eb 0c                   jmp    400f3c <phase_2+0x40>
```

程序通过上面第三行跳出了循环,也就是说要求`%rbp` 和 `%rbx`内部值要相同
接下来我们就可以考虑通过gdb看一下`%rbx`内部值是如何变化的了

gdb调试

同理打开程序所在文件夹,用gdb运行bomb程序

我们先随便输入`1 2 3 4 5 6`

```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
Breakpoint 1 at 0x400efc
Breakpoint 2 at 0x400f43
Breakpoint 3 at 0x40100c
Breakpoint 4 at 0x401062
Breakpoint 5 at 0x4010f4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
1 2 3 4 5 6
```

通过`si`指令,一步步去观察寄存器内部值变化

我们直接设置断点到循环第一次开始的地方

```
b *(phase_2+18)
```

然后用`c`指令继续执行,可以看到我们成功进入了循环
alt text

继续执行,经过一系列操作,我们发现

```
400f17: 8b 43 fc             mov    -0x4(%rbx),%eax
400f1a: 01 c0                add    %eax,%eax
400f1c: 39 03                cmp    %eax,(%rbx)
```

这一部分事实上是把当前`%rbx`的存储地址前四个字节的内容存到`%eax`(也就是我们输入的这个数的前一个数)并将`%eax`的值翻倍,与`%rbx`内的值做比较
简单来说,就是要求我们`输入的数从左往右数的后一个数是前一个数的两倍`
由于我们已经知道,一个数必须是`1` 所以这个等比数列就非常明显了, 我们的答案就是`1 2 4 8 16 32`

输入`ans.txt`尝试一下能不能过

```
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
Breakpoint 1 at 0x400f43
Breakpoint 2 at 0x40100c
Breakpoint 3 at 0x401062
Breakpoint 4 at 0x4010f4
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
```

完美通过,继续继续

phase_3

分析并调试

```
00000400f43 <phase_3>:
400f43: 48 83 ec 18             sub    $0x18,%rsp
400f47: 48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
400f4c: 48 8d 54 24 08          lea    0x8(%rsp),%rdx
400f51: be cf 25 40 00          mov    $0x4025cf,%esi
400f56: b8 00 00 00 00          mov    $0x0,%eax
```

```
400f5b: e8 90 fc ff ff          call   400bf0 <__isoc99_sscanf@plt>
400f60: 83 f8 01                cmp    $0x1,%eax
400f63: 7f 05                   jg     400f6a <phase_3+0x27>
400f65: e8 d0 04 00 00          call   40143a <explode_bomb>
400f6a: 83 7c 24 08 07          cmpl   $0x7,0x8(%rsp)
400f6f: 77 3c                   ja     400fad <phase_3+0x6a>
400f71: 8b 44 24 08             mov    0x8(%rsp),%eax
400f75: ff 24 c5 70 24 40 00    jmp    *0x402470(,%rax,8)
400f7c: b8 cf 00 00 00          mov    $0xcf,%eax
400f81: eb 3b                   jmp    400fbe <phase_3+0x7b>
400f83: b8 c3 02 00 00          mov    $0x2c3,%eax
400f88: eb 34                   jmp    400fbe <phase_3+0x7b>
400f8a: b8 00 01 00 00          mov    $0x100,%eax
400f8f: eb 2d                   jmp    400fbe <phase_3+0x7b>
400f91: b8 85 01 00 00          mov    $0x185,%eax
400f96: eb 26                   jmp    400fbe <phase_3+0x7b>
400f98: b8 ce 00 00 00          mov    $0xce,%eax
400f9d: eb 1f                   jmp    400fbe <phase_3+0x7b>
400f9f: b8 aa 02 00 00          mov    $0x2aa,%eax
400fa4: eb 18                   jmp    400fbe <phase_3+0x7b>
400fa6: b8 47 01 00 00          mov    $0x147,%eax
400fab: eb 11                   jmp    400fbe <phase_3+0x7b>
400fad: e8 88 04 00 00          call   40143a <explode_bomb>
400fb2: b8 00 00 00 00          mov    $0x0,%eax
400fb7: eb 05                   jmp    400fbe <phase_3+0x7b>
400fb9: b8 37 01 00 00          mov    $0x137,%eax
400fbe: 3b 44 24 0c             cmp    0xc(%rsp),%eax
400fc2: 74 05                   je     400fc9 <phase_3+0x86>
400fc4: e8 71 04 00 00          call   40143a <explode_bomb>
400fc9: 48 83 c4 18             add    $0x18,%rsp
400fcd: c3                      ret
```

`400f51`: 查看`$0x4025cf` 发现要求输入两个数字

```
multi-thre Thread 0x7ffff7f9a7 In: phase_3                L??   PC: 0x400f43
(gdb) x/s 0x4025cf
0x4025cf:       "%d %d"
```

`400f6a`: 要求第一个数小于等于7大于等于0 第一个数等于1时 跳到`*0x402470(,%rax,8)`
查看`0x402470` 发现跳到`0xb9`

```
(gdb) x/x 0x402478
0x402478:       0xb9
```

`400fb9`:比较`$0x137 %eax` 所以一个答案为 `1 311`

phase_4

分析

```
40100c: 48 83 ec 18             sub    $0x18,%rsp
401010: 48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
401015: 48 8d 54 24 08          lea    0x8(%rsp),%rdx
40101a: be cf 25 40 00          mov    $0x4025cf,%esi
40101f: b8 00 00 00 00          mov    $0x0,%eax
401024: e8 c7 fb ff ff          call   400bf0 <__isoc99_sscanf@plt>
401029: 83 f8 02                cmp    $0x2,%eax
40102c: 75 07                   jne    401035 <phase_4+0x29>
40102e: 83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
401033: 76 05                   jbe    40103a <phase_4+0x2e>
401035: e8 00 04 00 00          call   40143a <explode_bomb>
40103a: ba 0e 00 00 00          mov    $0xe,%edx
40103f: be 00 00 00 00          mov    $0x0,%esi
401044: 8b 7c 24 08             mov    0x8(%rsp),%edi
401048: e8 81 ff ff ff          call   400fce <func4>
40104d: 85 c0                   test   %eax,%eax
40104f: 75 07                   jne    401058 <phase_4+0x4c>
401051: 83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
401056: 74 05                   je     40105d <phase_4+0x51>
401058: e8 dd 03 00 00          call   40143a <explode_bomb>
40105d: 48 83 c4 18             add    $0x18,%rsp
401061: c3                      ret
```

`401010` `401015`:存入两个数
`40102e`:第一个数要小于等于14 令`(%edx) = 0xe`,`(%esi) = 0x0`,`(%edi) = num1` 然后跳转到func4 且要求`func4返回0`

简单翻译一下func4

```
int func4 (int edi, int esi, int edx)
//初始值:edi = 第一个数, esi = 0, edx = 14
{
    // 返回值为eax
    eax = edx - esi;
    eax = (eax + (eax >> 31)) >> 1;
    ecx = eax + exi;
    if(edi < ecx)
        return  2 * func4(edi, esi, edx - 1);
    else if (edi > ecx)
        return  2 * func4(edi, esi + 1, edx) + 1;
    else
        return  0;
}
```

所以要返回0, edi的值由fun4反汇编写出函数前三行,可以算出等于7
所以答案是 `7 0`

phase_5

分析

```
401062: 53                      push   %rbx
401063: 48 83 ec 20             sub    $0x20,%rsp
401067: 48 89 fb                mov    %rdi,%rbx
40106a: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
401071: 00 00
401073: 48 89 44 24 18          mov    %rax,0x18(%rsp)
401078: 31 c0                   xor    %eax,%eax
40107a: e8 9c 02 00 00          call   40131b <string_length>
40107f: 83 f8 06                cmp    $0x6,%eax
401082: 74 4e                   je     4010d2 <phase_5+0x70>
401084: e8 b1 03 00 00          call   40143a <explode_bomb>
```

观察`40107a` `40107f` 知道这一部分要求输入一个长度为6的字符串

```
40108b: 0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
40108f: 88 0c 24                mov    %cl,(%rsp)
401092: 48 8b 14 24             mov    (%rsp),%rdx
401096: 83 e2 0f                and    $0xf,%edx
401099: 0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
4010a0: 88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
4010a4: 48 83 c0 01             add    $0x1,%rax
4010a8: 48 83 f8 06             cmp    $0x6,%rax
4010ac: 75 dd                   jne    40108b <phase_5+0x29>
```

这是一个以`(%rax)`为循环变量的循环,等于6跳出循环

`0x4024b0`存了一个字符串

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

这段代码大概是在从输入的六个字符中取他们的`低四位`, 然后从`0x4024b0`取`第低四位`个字符存入新字符串

`0x40245e`也存了一个字符串

```
(gdb) x/s 0x40245e
0x40245e:       "flyers"
```

让我们取低四位得到的新字符串,与`0x40245e`存的字符串相等即可

这里我选择的答案是`yonefg`

phase_6

分析

这段代码显然是在读入6个数

```
4010f4: 41 56                   push   %r14
4010f6: 41 55                   push   %r13
4010f8: 41 54                   push   %r12
4010fa: 55                      push   %rbp
4010fb: 53                      push   %rbx
4010fc: 48 83 ec 50             sub    $0x50,%rsp
401100: 49 89 e5                mov    %rsp,%r13
401103: 48 89 e6                mov    %rsp,%rsi
401106: e8 51 03 00 00          call   40145c <read_six_numbers>
```

这段代码在判断6个数是不是都在1-6之间

```
40110b: 49 89 e6                mov    %rsp,%r14
40110e: 41 bc 00 00 00 00       mov    $0x0,%r12d
401114: 4c 89 ed                mov    %r13,%rbp
401117: 41 8b 45 00             mov    0x0(%r13),%eax
40111b: 83 e8 01                sub    $0x1,%eax
40111e: 83 f8 05                cmp    $0x5,%eax
401121: 76 05                   jbe    401128 <phase_6+0x34>
401123: e8 12 03 00 00          call   40143a <explode_bomb>
```

这段代码在判断是不是有重复数字(要求不重复)

```
401128: 41 83 c4 01             add    $0x1,%r12d
40112c: 41 83 fc 06             cmp    $0x6,%r12d
401130: 74 21                   je     401153 <phase_6+0x5f>
401132: 44 89 e3                mov    %r12d,%ebx
401135: 48 63 c3                movslq %ebx,%rax
401138: 8b 04 84                mov    (%rsp,%rax,4),%eax
40113b: 39 45 00                cmp    %eax,0x0(%rbp)
40113e: 75 05                   jne    401145 <phase_6+0x51>
401140: e8 f5 02 00 00          call   40143a <explode_bomb>
401145: 83 c3 01                add    $0x1,%ebx
401148: 83 fb 05                cmp    $0x5,%ebx
40114b: 7e e8                   jle    401135 <phase_6+0x41>
40114d: 49 83 c5 04             add    $0x4,%r13
401151: eb c1                   jmp    401114 <phase_6+0x20>
```

这段代码在将原来的6个数数组,变成`arrs[i] = 7-arr[i]`的数组

```
401153: 48 8d 74 24 18          lea    0x18(%rsp),%rsi
401158: 4c 89 f0                mov    %r14,%rax
40115b: b9 07 00 00 00          mov    $0x7,%ecx
401160: 89 ca                   mov    %ecx,%edx
401162: 2b 10                   sub    (%rax),%edx
401164: 89 10                   mov    %edx,(%rax)
401166: 48 83 c0 04             add    $0x4,%rax
40116a: 48 39 f0                cmp    %rsi,%rax
40116d: 75 f1                   jne    401160 <phase_6+0x6c>
```

```
40116f: be 00 00 00 00          mov    $0x0,%esi
401174: eb 21                   jmp    401197 <phase_6+0xa3>
401176: 48 8b 52 08             mov    0x8(%rdx),%rdx
40117a: 83 c0 01                add    $0x1,%eax
40117d: 39 c8                   cmp    %ecx,%eax
40117f: 75 f5                   jne    401176 <phase_6+0x82>
401181: eb 05                   jmp    401188 <phase_6+0x94>
401183: ba d0 32 60 00          mov    $0x6032d0,%edx
401188: 48 89 54 74 20          mov    %rdx,0x20(%rsp,%rsi,2)
40118d: 48 83 c6 04             add    $0x4,%rsi
401191: 48 83 fe 18             cmp    $0x18,%rsi
401195: 74 14                   je     4011ab <phase_6+0xb7>
401197: 8b 0c 34                mov    (%rsp,%rsi,1),%ecx
40119a: 83 f9 01                cmp    $0x1,%ecx
40119d: 7e e4                   jle    401183 <phase_6+0x8f>
40119f: b8 01 00 00 00          mov    $0x1,%eax
4011a4: ba d0 32 60 00          mov    $0x6032d0,%edx
4011a9: eb cb                   jmp    401176 <phase_6+0x82>
```

当`(%ecx)=1`跳到`401183`

`401183`调了个地址存的值,查看`$0x6032d0`

```
(gdb) x/32x 0x6032d0
0x6032d0 <node1>:       0x4c    0x01    0x00    0x00    0x01    0x00    0x00    0x00
0x6032d8 <node1+8>:     0xe0    0x32    0x60    0x00    0x00    0x00    0x00    0x00
0x6032e0 <node2>:       0xa8    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x6032e8 <node2+8>:     0xf0    0x32    0x60    0x00    0x00    0x00    0x00    0x00
```

貌似是个链表

不等于1的时候,开始一个循环

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

接下来就是按我们输入的数字,程序重新排列原来的链表顺序

排序要求顺序递减

最后排序结果应该是`4 3 2 1 6 5`

附加题