北京郵電大學

实验报告

题目：拆解二进制炸弹

班　　级：2020211306

学　　号：2020211376

姓　　名：马天成

学　　院：计算机学院

2021 年 11 月 06 日

目录

一、实验目的

1.理解 C 语言程序的机器级表示。
2.初步掌握 GDB 调试器的用法。
3.阅读 C 编译器生成的 x86-64 机器代码，理解不同控制结构生成的基本指令模式，过程的实现。

二、实验环境

SecureCRT（10.120.11.12）
Linux
Objdump 命令反汇编
GDB 调试工具
积分榜（http://10.120.11.13:19220/scoreboard）
报告邮寄（X86 版本最迟时间：2021 年 11 月 17 日晚 23：59；Arm 版本最迟时间：2021 年 11 月 24 日晚 23：59）：大二班（5-8 班）：yangyyj98@bupt.edu.cn

三、实验内容

登录 bupt1 服务器，在 home 目录下可以找到 Evil 博士专门为你量身定制的一个 bomb，当运行时，它会要求你输入一个字符串，如果正确，则进入下一关，继续要求你输入下一个字符串；否则，炸弹就会爆炸，输出一行提示信息并向计分服务器提交扣分信息。因此，本实验要求你必须通过反汇编和逆向工程对 bomb 执行文件进行分析，找到正确的字符串来解除这个的炸弹。

本实验通过要求使用课程所学知识拆除一个“binary bombs”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。 “binary bombs”是一个 Linux 可执行程序，包含了 5 个阶段（或关卡）。炸弹运行的每个阶段要求你输入一个特定字符串，你的输入符合程序预期的输入，该阶段的炸弹就被拆除引信；否则炸弹“爆炸”，打印输出 “BOOM!!!”。炸弹的每个阶段考察了机器级程序语言的一个不同方面，难度逐级递增。

为完成二进制炸弹拆除任务，需要使用 gdb 调试器和 objdump 来反汇编 bomb 文件，可以单步跟踪调试每一阶段的机器代码，也可以阅读反汇编代码，从中理解每一汇编语言代码的行为或作用，进而设法推断拆除炸弹所需的目标字符串。实验 2 的具体内容见实验 2 说明。

四、实验步骤及实验分析

准备工作

先熟悉程序的机器级表示，如 move 指令集、跳转指令集、算术运算指令集、逻辑运算指令集；以及各种条件码和寄存器的固定使用方式；熟悉各类指令的基本组合，如 test %rax %rax + jne address。

然后熟悉 objdump 的基本操作（objdump -d bomb > bomb.o ……)。

熟悉 gdb 的基本操作（break 地址/函数名 ; disassemble (函数名) ; stepi ; print ; x……)。

阶段一：尝试调试

首先尝试上手 gdb。从开始调试，设置断点，以及最后的逐步调试。
第一个实验显然就是练手的。我简要概括说一下遇到的难点：

1. 看不太懂汇编代码。一开始看真的很难受，一堆指令根本不知道在干啥，不了解组合，以及一看到跳转就头疼。（当然现在也是）
2. 不会把握调试。一开始大刀阔斧让后边使劲 continue；炸了之后又一步步 stepi。
3. 不会看寄存器。一开始总是认为寄存器的值是胡乱的，是中间的计算量，无法跟踪。

当然，这些问题我在拆 7 个炸弹的时候就慢慢解决了所谓的心理障碍和能力障碍。

阶段二：拆 phase_1 -> phase_6

这就牵扯到详细的拆炸弹过程了。下面我详细介绍基础的 6 个炸弹是如何拆掉的。
难点主要从汇编代码反向理解源代码、调试并实时查看寄存器状态、理解寄存器和内部空间三个角度展开。

phase_1

```
(gdb) b strings_not_equal
Breakpoint 3 at 0x4013c2
(gdb) next

Breakpoint 2, 0x0000000000400f2d in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400f2d <+0>:     sub    $0x8,%rsp
   0x0000000000400f31 <+4>:     mov    $0x402670,%esi
   0x0000000000400f36 <+9>:     callq  0x4013c2 <strings_not_equal>
   0x0000000000400f3b <+14>:    test   %eax,%eax
   0x0000000000400f3d <+16>:    je     0x400f44 <phase_1+23>
   0x0000000000400f3f <+18>:    callq  0x401696 <explode_bomb>
   0x0000000000400f44 <+23>:    add    $0x8,%rsp
   0x0000000000400f48 <+27>:    retq
End of assembler dump.
(gdb) b *0x400f36
Breakpoint 4 at 0x400f36
(gdb) next
Single stepping until exit from function phase_1,
which has no line number information.

Breakpoint 4, 0x0000000000400f36 in phase_1 ()
(gdb) x /s $esi
0x402670:       "There are many handsome guys and beautiful girls on ShaHe campus."
(gdb)
```

phase_1

1. 初步理解程序

在 phase_1 处设了断点，并用 disassemble 指令观察了汇编代码。
可知大致意思为：调用 strings_not_equal 函数，判断返回值；如果返回值为 0，则爆炸。
基于此，可以判断：我们要**输入正确的字符串与答案匹配**才能使程序跳过爆炸阶段。

2. 寻找字符串

```
Breakpoint 4, 0x000000000400f3b in phase_1 ()
(gdb) x /s $esi
0x402670:       "There are many handsome guys and beautiful girls on ShaHe campus."
```

根据寄存器状态可知：进入 strings_not_equal 后，我们的寄存器参数 esi 打印字符串可得:
“There are many handsome guys and beautiful girls on ShaHe campus.”
由函数名易知，我们输入这个字符串，匹配后，就能使返回值为 1，跳过该次爆炸。

3. 增加的理解

- 用 x /s address 打印字符串
- rax (eax) 一般做函数返回值
- rdi rsi 一般做函数参数

phase_2

```
Dump of assembler code for function phase_2:
   0x0000000000400f49 <+0>:     push   %rbp
   0x0000000000400f4a <+1>:     push   %rbx
   0x0000000000400f4b <+2>:     sub    $0x28,%rsp
   0x0000000000400f4f <+6>:     mov    %fs:0x28,%rax
   0x0000000000400f58 <+15>:    mov    %rax,0x18(%rsp)
   0x0000000000400f5d <+20>:    xor    %eax,%eax
   0x0000000000400f5f <+22>:    mov    %rsp,%rsi
   0x0000000000400f62 <+25>:    callq  0x4016cc <read_six_numbers>
   0x0000000000400f67 <+30>:    cmpl   $0x0,(%rsp)
   0x0000000000400f6b <+34>:    jne    0x400f74 <phase_2+43>
   0x0000000000400f6d <+36>:    cmpl   $0x1,0x4(%rsp)
   0x0000000000400f72 <+41>:    je     0x400f79 <phase_2+48>
   0x0000000000400f74 <+43>:    callq  0x401696 <explode_bomb>
   0x0000000000400f79 <+48>:    mov    %rsp,%rbx
   0x0000000000400f7c <+51>:    lea    0x10(%rsp),%rbp
   0x0000000000400f81 <+56>:    mov    0x4(%rbx),%eax
   0x0000000000400f84 <+59>:    add    (%rbx),%eax
   0x0000000000400f86 <+61>:    cmp    %eax,0x8(%rbx)
   0x0000000000400f89 <+64>:    je     0x400f90 <phase_2+71>
   0x0000000000400f8b <+66>:    callq  0x401696 <explode_bomb>
   0x0000000000400f90 <+71>:    add    $0x4,%rbx
   0x0000000000400f94 <+75>:    cmp    %rbp,%rbx
   0x0000000000400f97 <+78>:    jne    0x400f81 <phase_2+56>
   0x0000000000400f99 <+80>:    mov    0x18(%rsp),%rax
   0x0000000000400f9e <+85>:    xor    %fs:0x28,%rax
   0x0000000000400fa7 <+94>:    je     0x400fae <phase_2+101>
   0x0000000000400fa9 <+96>:    callq  0x400b90 <__stack_chk_fail@plt>
   0x0000000000400fae <+101>:   add    $0x28,%rsp
   0x0000000000400fb2 <+105>:   pop    %rbx
   0x0000000000400fb3 <+106>:   pop    %rbp
   0x0000000000400fb4 <+107>:   retq
End of assembler dump.
```

phase_2

```
00000000004016cc <read_six_numbers>:
  4016cc:       48 83 ec 08             sub    $0x8,%rsp
  4016d0:       48 89 f2                mov    %rsi,%rdx
  4016d3:       48 8d 4e 04             lea    0x4(%rsi),%rcx
  4016d7:       48 8d 46 14             lea    0x14(%rsi),%rax
  4016db:       50                      push   %rax
  4016dc:       48 8d 46 10             lea    0x10(%rsi),%rax
  4016e0:       50                      push   %rax
  4016e1:       4c 8d 4e 0c             lea    0xc(%rsi),%r9
  4016e5:       4c 8d 46 08             lea    0x8(%rsi),%r8
  4016e9:       be a1 29 40 00          mov    $0x4029a1,%esi
  4016ee:       b8 00 00 00 00          mov    $0x0,%eax
  4016f3:       e8 48 f5 ff ff          callq  400c40 <__isoc99_sscanf@plt>
  4016f8:       48 83 c4 10             add    $0x10,%rsp
  4016fc:       83 f8 05                cmp    $0x5,%eax
  4016ff:       7f 05                   jg     401706 <read_six_numbers+0x3a>
  401701:       e8 90 ff ff ff          callq  401696 <explode_bomb>
  401706:       48 83 c4 08             add    $0x8,%rsp
  40170a:       c3                      retq
```

read_six_numbers

1. 初步理解程序

在汇编代码中，他告诉我 ”read_six_numbers”，所以我应该是要输留个数六个数进去。
但是这个 read_six_numbers 也是有炸弹的!

```
   0x00000000004016f3 <+39>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x00000000004016f8 <+44>:    add    $0x10,%rsp
   0x00000000004016fc <+48>:    cmp    $0x5,%eax
   0x00000000004016ff <+51>:    jg     0x401706 <read_six_numbers+58>
   0x0000000000401701 <+53>:    callq  0x401696 <explode_bomb>
```

第一遍我在输入时只输入了一个 0，导致在调用完这个函数后就炸了，Boom!

这个函数中 eax 是记录读入数字的个数。
按图中效果，如果输入数字个数小于等于 5，则函数会调用 explode_bomb 进行爆炸。

进这个函数后进了 scanf，并出现以下提示：

```
0x0000000000400c40 in __isoc99_sscanf@plt ()
(gdb) stepi
__GI___isoc99_sscanf (s=0x604810 <input_strings+80> "0 1 1 2 3 5", format=0x4029a1 "%d %d %d %d %d %d") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
(gdb)
```

scanf_2

输入格式是“%d %d %d %d %d %d”，所以我需要的输入是 6 个数。

2. 寻找六个数

```
(gdb) x /d $rsp
0x7fffffffea50: 0
(gdb) x /d $rsp+4
0x7fffffffea54: 1
(gdb) x /d $rsp+8
0x7fffffffea58: 1
(gdb) x /d $rsp+0xc
0x7fffffffea5c: 2
(gdb) x /d $rsp+0x10
0x7fffffffea60: 3
(gdb) x /d $rsp+0x14
0x7fffffffea64: 5
```

根据观测寄存器可知，我输入的数是被保存在了 rsp（开栈）的连续空间中。

查看源代码，它通过寄存器的来回赋值和逐步地址+0x4（数组是连续空间，地址+4 就是下标加一取值）移位操作，来达成逐步计算两项之和的功能，并且与第三项进行比较：

1. 匹配第一个输入是否为 $0x0；
2. 匹配第二个输入是否为 $0x1；
3. 循环 $0x6 次；
4. 循环算出前两个数之和并且与当前项比较，相等则继续循环；否则直接调用 explode_bomb；
5. 执行完循环则返回，并输出 phase_defused。

所以我只需要输入标准斐波那契数列前六项的数据，这个就能过了。
答案是（0，1，1，2，3，5）

3. 增加的理解

- 要理解机器代码实现循环：通过 cmp/test 配合 jump，相当于 do-while 或者 goto。
- 栈帧 rsp 很有用，是栈帧，保存当前函数地址，开栈存储输入的值。
- 理解了数组的机器级结构，主要是操作地址进行下标的变化和取值。

phase_3

```
0x0000000000400fb9 <+4>:     mov    %fs:0x28,%rax
0x0000000000400fc2 <+13>:    mov    %rax,0x8(%rsp)
0x0000000000400fc7 <+18>:    xor    %eax,%eax
0x0000000000400fc9 <+20>:    lea    0x4(%rsp),%rcx
0x0000000000400fce <+25>:    mov    %rsp,%rdx
0x0000000000400fd1 <+28>:    mov    $0x4029ad,%esi
0x0000000000400fd6 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
0x0000000000400fdb <+38>:    cmp    $0x1,%eax
0x0000000000400fde <+41>:    jg     0x400fe5 <phase_3+48>
0x0000000000400fe0 <+43>:    callq  0x401696 <explode_bomb>
0x0000000000400fe5 <+48>:    cmpl   $0x7,(%rsp)
0x0000000000400fe9 <+52>:    ja     0x401026 <phase_3+113>
0x0000000000400feb <+54>:    mov    (%rsp),%eax
0x0000000000400fee <+57>:    jmpq   *0x4026e0(,%rax,8)
0x0000000000400ff5 <+64>:    mov    $0x8e,%eax
0x0000000000400ffa <+69>:    jmp    0x401037 <phase_3+130>
0x0000000000400ffc <+71>:    mov    $0x37e,%eax
0x0000000000401001 <+76>:    jmp    0x401037 <phase_3+130>
0x0000000000401003 <+78>:    mov    $0xd0,%eax
0x0000000000401008 <+83>:    jmp    0x401037 <phase_3+130>
0x000000000040100a <+85>:    mov    $0x3d5,%eax
0x000000000040100f <+90>:    jmp    0x401037 <phase_3+130>
0x0000000000401011 <+92>:    mov    $0x3b2,%eax
0x0000000000401016 <+97>:    jmp    0x401037 <phase_3+130>
0x0000000000401018 <+99>:    mov    $0x122,%eax
0x000000000040101d <+104>:   jmp    0x401037 <phase_3+130>
0x000000000040101f <+106>:   mov    $0x42,%eax
0x0000000000401024 <+111>:   jmp    0x401037 <phase_3+130>
0x0000000000401026 <+113>:   callq  0x401696 <explode_bomb>
0x000000000040102b <+118>:   mov    $0x0,%eax
0x0000000000401030 <+123>:   jmp    0x401037 <phase_3+130>
0x0000000000401032 <+125>:   mov    $0x217,%eax
0x0000000000401037 <+130>:   cmp    0x4(%rsp),%eax
Type <RET> for more, q to quit, c to continue without paging--
0x000000000040103b <+134>:   je     0x401042 <phase_3+141>
0x000000000040103d <+136>:   callq  0x401696 <explode_bomb>
0x0000000000401042 <+141>:   mov    0x8(%rsp),%rax
0x0000000000401047 <+146>:   xor    %fs:0x28,%rax
0x0000000000401050 <+155>:   je     0x401057 <phase_3+162>
0x0000000000401052 <+157>:   callq  0x400b90 <__stack_chk_fail@plt>
0x0000000000401057 <+162>:   add    $0x18,%rsp
0x000000000040105b <+166>:   retq
```

phase_3

1. 初步理解程序

从这个程序开始，就没有明确的提示输入的信息了。需要我们自己进入 scanf 来看。

```
Dump of assembler code for function __isoc99_sscanf@plt:
=> 0x0000000000400c40 <+0>:     jmpq   *0x20345a(%rip)        # 0x6040a0 <__isoc99_sscanf@got.plt>
   0x0000000000400c46 <+6>:     pushq  $0x11
   0x0000000000400c4b <+11>:    jmpq   0x400b20
End of assembler dump.
(gdb) stepi
__GI___isoc99_sscanf (s=0x604860 <input_strings+160> "7 507", format=0x4029ad "%d %d") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
(gdb)
```

fomart=”%d %d”，表示此时我们要输入两个数字来读入程序。

2. 寻找二元组答案

info register 后，查看寄存器状态和地址取出来的值知道：

1. 我们存的第一个数是在 rsp 的地址中；
2. 第二个数就在它后面四个字节（数字占 4 个字节）。

所以我们要找出答案，用我们的输入来跳过相应的爆炸步骤。

第一个数：

```
0x0000000000400fd6 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
0x0000000000400fdb <+38>:    cmp    $0x1,%eax
0x0000000000400fde <+41>:    jg     0x400fe5 <phase_3+48>
0x0000000000400fe0 <+43>:    callq  0x401696 <explode_bomb>
0x0000000000400fe5 <+48>:    cmpl   $0x7,(%rsp)
0x0000000000400fe9 <+52>:    ja     0x401026 <phase_3+113>
```

根据 cmpl $0x7 , (%rsp) 知，第一个数要小于等于 7 才能跳过第一个炸弹。

第二个数：

根据 jmpq *0x4026e0(,%rax,8) 知：我们会根据第一次的输入跳转到相应行；而在执行完相应行之后都会跳出。显然，这是一个 switch 指令。

```
(gdb) print /x *(0x4026e0+0x8)
$9 = 0x400ff5
(gdb) print /x *(0x4026e0+0x10)
$10 = 0x400ffc
(gdb) print /x *(0x4026e0+0x18)
$11 = 0x401003
(gdb) print /x *(0x4026e0+0x20)
$12 = 0x40100a
(gdb) print /x *(0x4026e0+0x28)
$13 = 0x401011
(gdb) print /x *(0x4026e0+0x30)
$14 = 0x401018
(gdb) print /x *(0x4026e0+0x38)
$15 = 0x40101f
```

根据打印相应的值，1-7 会跳转到相应的行来比较我们输入的第二个数；而对于不同的行，要比较的数字也不一样，所以一共有 7 个答案。

(1 , 0x8e) (2 , 0x37e) (3 , oxd0) (4 , 0x3d5) (5 , 0x3b2) (6 , 0x122) (7 , 0x42)

我选择了一个比较小的值，就是 (7 , 66) -- 怕错 *(^ ~ ^)*

3. 增加的理解

- 了解了 switch 的执行方式：通过参数来选择要跳转的地址来执行相应的步骤。
- 懂得了拆这个炸弹要先学会尝试，现带入几个试试，然后逐个找到正确输入。
- 熟悉了寄存器取地址和取值的操作。

phase_4

```
Dump of assembler code for function phase_4:
   0x0000000000401097 <+0>:     sub    $0x18,%rsp
   0x000000000040109b <+4>:     mov    %fs:0x28,%rax
   0x00000000004010a4 <+13>:    mov    %rax,0x8(%rsp)
   0x00000000004010a9 <+18>:    xor    %eax,%eax
   0x00000000004010ab <+20>:    mov    %rsp,%rcx
   0x00000000004010ae <+23>:    lea    0x4(%rsp),%rdx
   0x00000000004010b3 <+28>:    mov    $0x4029ad,%esi
   0x00000000004010b8 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x00000000004010bd <+38>:    cmp    $0x2,%eax
   0x00000000004010c0 <+41>:    jne    0x4010cd <phase_4+54>
   0x00000000004010c2 <+43>:    mov    (%rsp),%eax
   0x00000000004010c5 <+46>:    sub    $0x2,%eax
   0x00000000004010c8 <+49>:    cmp    $0x2,%eax
   0x00000000004010cb <+52>:    jbe    0x4010d2 <phase_4+59>
   0x00000000004010cd <+54>:    callq  0x401696 <explode_bomb>
   0x00000000004010d2 <+59>:    mov    (%rsp),%esi
   0x00000000004010d5 <+62>:    mov    $0x8,%edi
   0x00000000004010da <+67>:    callq  0x40105c <func4>
   0x00000000004010df <+72>:    cmp    0x4(%rsp),%eax
   0x00000000004010e3 <+76>:    je     0x4010ea <phase_4+83>
   0x00000000004010e5 <+78>:    callq  0x401696 <explode_bomb>
   0x00000000004010ea <+83>:    mov    0x8(%rsp),%rax
   0x00000000004010ef <+88>:    xor    %fs:0x28,%rax
   0x00000000004010f8 <+97>:    je     0x4010ff <phase_4+104>
   0x00000000004010fa <+99>:    callq  0x400b90 <__stack_chk_fail@plt>
   0x00000000004010ff <+104>:   add    $0x18,%rsp
   0x0000000000401103 <+108>:   retq
End of assembler dump.
```

phase_4

```
000000000040105c <func4>:
  40105c:       85 ff                   test   %edi,%edi
  40105e:       7e 2b                   jle    40108b <func4+0x2f>
  401060:       89 f0                   mov    %esi,%eax
  401062:       83 ff 01                cmp    $0x1,%edi
  401065:       74 2e                   je     401095 <func4+0x39>
  401067:       41 54                   push   %r12
  401069:       55                      push   %rbp
  40106a:       53                      push   %rbx
  40106b:       89 f5                   mov    %esi,%ebp
  40106d:       89 fb                   mov    %edi,%ebx
  40106f:       8d 7f ff                lea    -0x1(%rdi),%edi
  401072:       e8 e5 ff ff ff          callq  40105c <func4>
  401077:       44 8d 64 05 00          lea    0x0(%rbp,%rax,1),%r12d
  40107c:       8d 7b fe                lea    -0x2(%rbx),%edi
  40107f:       89 ee                   mov    %ebp,%esi
  401081:       e8 d6 ff ff ff          callq  40105c <func4>
  401086:       44 01 e0                add    %r12d,%eax
  401089:       eb 06                   jmp    401091 <func4+0x35>
  40108b:       b8 00 00 00 00          mov    $0x0,%eax
  401090:       c3                      retq
  401091:       5b                      pop    %rbx
  401092:       5d                      pop    %rbp
  401093:       41 5c                   pop    %r12
  401095:       f3 c3                   repz retq

0000000000401097 <phase_4>:
  401097:       48 83 ec 18             sub    $0x18,%rsp
  40109b:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4010a2:       00 00
```

func4

从这一题开始，对于不完全熟悉汇编代码的我，题目变得抽象起来了。

1. 初步理解程序

本题是递归函数。首先，通读汇编代码，根据上一题的经验，rsp 是栈帧，保存我的输入地址。
通过 callq <func4> 可知，我们需要运行 func4，然后函数的返回值是 %eax。
通过 cmp 0x4(%rsp) %eax 可知，是我输入的第二个 int 与答案作比较。

初步解读：

1. 我的**第一个输入要合法**，跳过第一个炸弹步骤；
2. 将我的第一个输入传参进入 func4，然后将**跑出来的答案与我的第二个输入匹配**；
3. 如果第二次匹配正确，则输出 **phase_defused**；否调用 **explode_bomb**。

所以我完全可以先往里面跑一遍，然后获得 func4 的运行结果，就可以得到答案了！
当然这是拆炸弹的思路。这是练习，还是得看懂汇编代码。
通过两次比较，我大概猜测我的输入是两个 int；跑进 scanf 后也印证了我的猜测：

```
(gdb) disassemble
Dump of assembler code for function __isoc99_sscanf@plt:
=> 0x0000000000400c40 <+0>:     jmpq   *0x20345a(%rip)        # 0x6040a0 <__isoc99_sscanf@got.plt>
   0x0000000000400c46 <+6>:     pushq  $0x11
   0x0000000000400c4b <+11>:    jmpq   0x400b20
End of assembler dump.
(gdb) stepi
__GI___isoc99_sscanf (s=0x6048b0 <input_strings+240> "0 66", format=0x4029ad "%d %d") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
```

scanf_4

2. 寻找答案

第一个数：

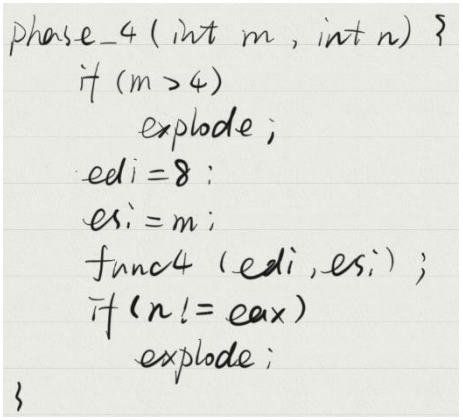
```
0x00000000004010b8 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
0x00000000004010bd <+38>:    cmp    $0x2,%eax
0x00000000004010c0 <+41>:    jne    0x4010cd <phase_4+54>
0x00000000004010c2 <+43>:    mov    (%rsp),%eax
0x00000000004010c5 <+46>:    sub    $0x2,%eax
0x00000000004010c8 <+49>:    cmp    $0x2,%eax
0x00000000004010cb <+52>:    jbe    0x4010d2 <phase_4+59>
0x00000000004010cd <+54>:    callq  0x401696 <explode_bomb>
```

1. **mov (%rsp) , %eax** ： 将我的第一个数赋给 **%eax** ；
2. **sub $0x2 , %eax** ： 将 **%eax** 减 2；
3. **cmp $0x2 , %eax** ： 将 **%eax** 与 2 比较；
4. 如果小于等于 4 则继续进行；否则调用 explode_bomb；

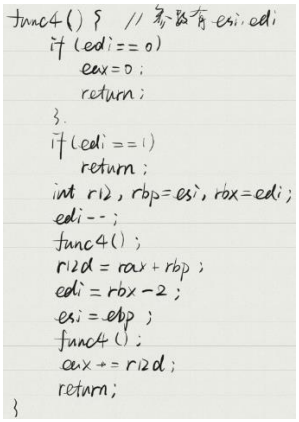
所以我们知道，输入必须小于等于 4；我这里选择 4。

第二个数：

首先找到传入的参数 **%edi = 8 , %esi = (%rsp)**；直观观察 func4 可知，我们的 edi 是控制深度的量。
函数手写代码如下：



phase_4 手稿



func4 手稿

根据深度控制条件反推，我这里代码的功能是：

Step 1. 一共最多开 8 层栈，从初到末记为 **1-8**；

Step 2. 对每一层，**eax**，**esi**，**edi** 都可以**视为全局变量**；

Step 3. 在每一层，**eax 加上计算前 eax+我输入的第一个值**；

Step 4. 每次阶段一结束，**edi-1**；每次阶段二结束，**edi-2**；

求出答案：

注意：我们的输入是反过来的：

***rsp 存的是第二个数，*(rsp+4)存的是第一个数**；

最终答案计算为（**214 , 4**）

3. 增加的理解

- 见识了**递归函数的恶心程度**，以后再也不写递归了！！！（快改了，但有时候转递推是真的有点难想）
- 同时也加深了**传入参数**和**返回参数**的理解，也重新认识了如何解决临时变量：

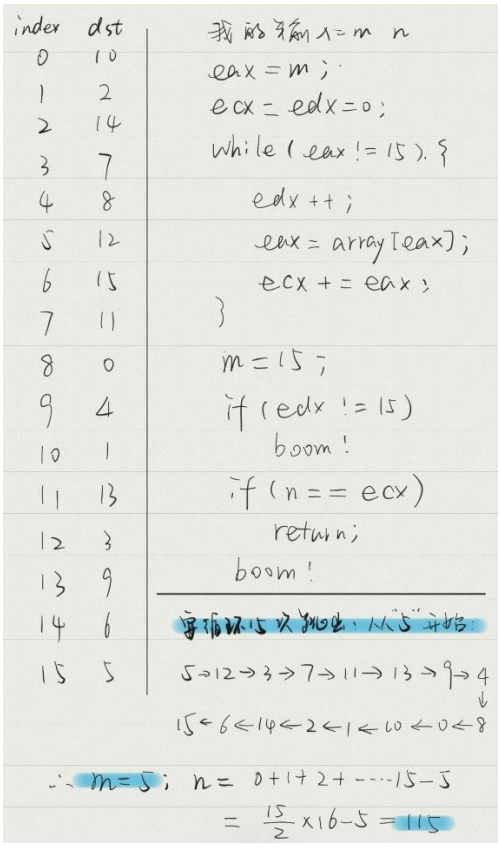
  push 参数寄存器（加入临时变量）

  pop 参数寄存器（释放临时变量）
- 认识到传入参数和返回值寄存器就是类似于全局变量的存在，而其他就是临时变量了。

phase_5

```
Dump of assembler code for function phase_5:
=> 0x0000000000401104 <+0>:     sub    $0x18,%rsp
   0x0000000000401108 <+4>:     mov    %fs:0x28,%rax
   0x0000000000401111 <+13>:    mov    %rax,0x8(%rsp)
   0x0000000000401116 <+18>:    xor    %eax,%eax
   0x0000000000401118 <+20>:    lea    0x4(%rsp),%rcx
   0x000000000040111d <+25>:    mov    %rsp,%rdx
   0x0000000000401120 <+28>:    mov    $0x4029ad,%esi
   0x0000000000401125 <+33>:    callq  0x400c40 <__isoc99_sscanf@plt>
   0x000000000040112a <+38>:    cmp    $0x1,%eax
   0x000000000040112d <+41>:    jg     0x401134 <phase_5+48>
   0x000000000040112f <+43>:    callq  0x401696 <explode_bomb>
   0x0000000000401134 <+48>:    mov    (%rsp),%eax
   0x0000000000401137 <+51>:    and    $0xf,%eax
   0x000000000040113a <+54>:    mov    %eax,(%rsp)
   0x000000000040113d <+57>:    cmp    $0xf,%eax
   0x0000000000401140 <+60>:    je     0x401171 <phase_5+109>
   0x0000000000401142 <+62>:    mov    $0x0,%ecx
   0x0000000000401147 <+67>:    mov    $0x0,%edx
   0x000000000040114c <+72>:    add    $0x1,%edx
   0x000000000040114f <+75>:    cltq
   0x0000000000401151 <+77>:    mov    0x402720(,%rax,4),%eax
   0x0000000000401158 <+84>:    add    %eax,%ecx
   0x000000000040115a <+86>:    cmp    $0xf,%eax
   0x000000000040115d <+89>:    jne    0x40114c <phase_5+72>
   0x000000000040115f <+91>:    movl   $0xf,(%rsp)
   0x0000000000401166 <+98>:    cmp    $0xf,%edx
   0x0000000000401169 <+101>:   jne    0x401171 <phase_5+109>
   0x000000000040116b <+103>:   cmp    0x4(%rsp),%ecx
   0x000000000040116f <+107>:   je     0x401176 <phase_5+114>
   0x0000000000401171 <+109>:   callq  0x401696 <explode_bomb>
   0x0000000000401176 <+114>:   mov    0x8(%rsp),%rax
   0x000000000040117b <+119>:   xor    %fs:0x28,%rax
   0x0000000000401184 <+128>:   je     0x40118b <phase_5+135>
   0x0000000000401186 <+130>:   callq  0x400b90 <__stack_chk_fail@plt>
   0x000000000040118b <+135>:   add    $0x18,%rsp
   0x000000000040118f <+139>:   retq
--Type <RET> for more, q to quit, c to continue without paging--
```

phase_5



phase_5 手稿

1. 初步理解程序

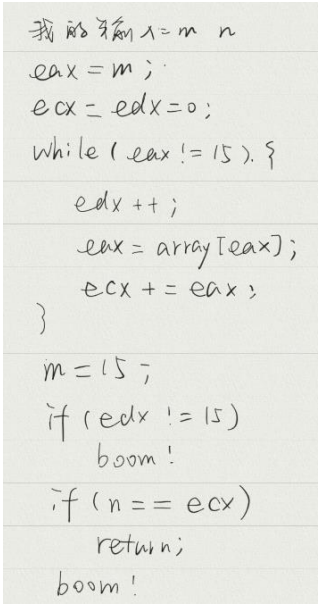
显然这次也是要输入几个数。不猜了，直接进去看：

```
(gdb)
__GI___isoc99_sscanf (s=0x604900 <input_strings+320> "5 115", format=0x4029ad "%d %d") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
(gdb)
```

“%d %d”，显然是要求输入两个数。
了解到这次是输入两个数，那么先来找找线索。
本次好像只需要比较我的第二个数？那么，找第一个数是干什么的就很重要了。

2. 寻找答案

根据我的反汇编至源程序的代码可知：



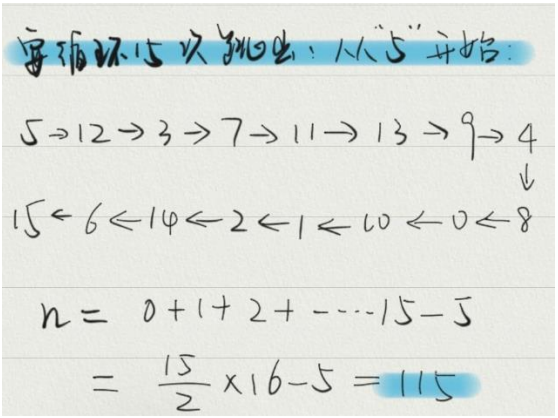
Step 1. 我的输入是 m，n；
Step 2. 有计数器 edx 初始值为 0；每次循环+1；
Step 3. 初始下标 eax 的值为我的输入 m。(eax = m)
Step 4. 循环中，在数组中根据 eax 作为下标找到对应内容，并将其给 eax；
Step 5. 一共要循环 15 次，要不然直接调用 explode_bomb。

图中的 **Array 数组**哪里来？观察到：mov 0x402720(,%rax,4) eax
这就是我的数组，且起始地址是 0x402720。
打印出来如下：

```
(gdb) x /16dw 0x402720
0x402720 <array.3599>:  10      2       14      7
0x402730 <array.3599+16>:       8       12      15      11
0x402740 <array.3599+32>:       0       4       1       13
0x402750 <array.3599+48>:       3       9       6       5
```

总共有 16 个，整理关系就是我在上面完整手稿写的那样。

根据**反推原则**，**最后一步计算是找到值为 15 的数组单元**



所以答案是：

第一个数：起始值 5，标志着要从 5 开始找；
第二个数：把路过的数组值相加之和，为 115。

3. 增加的理解

- 认识到数组在机器中的存储方式，以及 lea 指令对数组的重要性。
- 加强了对数组的理解。
- 加强了对循环的理解。

phase_6

```
(gdb) disassemble
Dump of assembler code for function phase_6:
=> 0x0000000000401190 <+0>:     push   %r13
   0x0000000000401192 <+2>:     push   %r12
   0x0000000000401194 <+4>:     push   %rbp
   0x0000000000401195 <+5>:     push   %rbx
   0x0000000000401196 <+6>:     sub    $0x68,%rsp
   0x000000000040119a <+10>:    mov    %fs:0x28,%rax
   0x00000000004011a3 <+19>:    mov    %rax,0x58(%rsp)
   0x00000000004011a8 <+24>:    xor    %eax,%eax
   0x00000000004011aa <+26>:    mov    %rsp,%rsi
   0x00000000004011ad <+29>:    callq  0x4016cc <read_six_numbers>
   0x00000000004011b2 <+34>:    mov    %rsp,%r12
   0x00000000004011b5 <+37>:    mov    $0x0,%r13d
   0x00000000004011bb <+43>:    mov    %r12,%rbp
   0x00000000004011be <+46>:    mov    (%r12),%eax
   0x00000000004011c2 <+50>:    sub    $0x1,%eax
   0x00000000004011c5 <+53>:    cmp    $0x5,%eax
   0x00000000004011c8 <+56>:    jbe    0x4011cf <phase_6+63>
   0x00000000004011ca <+58>:    callq  0x401696 <explode_bomb>
   0x00000000004011cf <+63>:    add    $0x1,%r13d
   0x00000000004011d3 <+67>:    cmp    $0x6,%r13d
   0x00000000004011d7 <+71>:    je     0x401216 <phase_6+134>
   0x00000000004011d9 <+73>:    mov    %r13d,%ebx
   0x00000000004011dc <+76>:    movslq %ebx,%rax
   0x00000000004011df <+79>:    mov    (%rsp,%rax,4),%eax
   0x00000000004011e2 <+82>:    cmp    %eax,0x0(%rbp)
   0x00000000004011e5 <+85>:    jne    0x4011ec <phase_6+92>
   0x00000000004011e7 <+87>:    callq  0x401696 <explode_bomb>
   0x00000000004011ec <+92>:    add    $0x1,%ebx
   0x00000000004011ef <+95>:    cmp    $0x5,%ebx
   0x00000000004011f2 <+98>:    jle    0x4011dc <phase_6+76>
   0x00000000004011f4 <+100>:   add    $0x4,%r12
   0x00000000004011f8 <+104>:   jmp    0x4011bb <phase_6+43>
   0x00000000004011fa <+106>:   mov    0x8(%rdx),%rdx
   0x00000000004011fe <+110>:   add    $0x1,%eax
   0x0000000000401201 <+113>:   cmp    %ecx,%eax
   0x0000000000401203 <+115>:   jne    0x4011fa <phase_6+106>
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401205 <+117>:   mov    %rdx,0x20(%rsp,%rsi,2)
   0x000000000040120a <+122>:   add    $0x4,%rsi
   0x000000000040120e <+126>:   cmp    $0x18,%rsi
```

phase_6_1

```
   0x000000000040120e <+126>:   cmp    $0x18,%rsi
   0x0000000000401212 <+130>:   jne    0x40121b <phase_6+139>
   0x0000000000401214 <+132>:   jmp    0x40122f <phase_6+159>
   0x0000000000401216 <+134>:   mov    $0x0,%esi
   0x000000000040121b <+139>:   mov    (%rsp,%rsi,1),%ecx
   0x000000000040121e <+142>:   mov    $0x1,%eax
   0x0000000000401223 <+147>:   mov    $0x6042f0,%edx
   0x0000000000401228 <+152>:   cmp    $0x1,%ecx
   0x000000000040122b <+155>:   jg     0x4011fa <phase_6+106>
   0x000000000040122d <+157>:   jmp    0x401205 <phase_6+117>
   0x000000000040122f <+159>:   mov    0x20(%rsp),%rbx
   0x0000000000401234 <+164>:   lea    0x20(%rsp),%rax
   0x0000000000401239 <+169>:   lea    0x48(%rsp),%rsi
   0x000000000040123e <+174>:   mov    %rbx,%rcx
   0x0000000000401241 <+177>:   mov    0x8(%rax),%rdx
   0x0000000000401245 <+181>:   mov    %rdx,0x8(%rcx)
   0x0000000000401249 <+185>:   add    $0x8,%rax
   0x000000000040124d <+189>:   mov    %rdx,%rcx
   0x0000000000401250 <+192>:   cmp    %rsi,%rax
   0x0000000000401253 <+195>:   jne    0x401241 <phase_6+177>
   0x0000000000401255 <+197>:   movq   $0x0,0x8(%rdx)
   0x000000000040125d <+205>:   mov    $0x5,%ebp
   0x0000000000401262 <+210>:   mov    0x8(%rbx),%rax
   0x0000000000401266 <+214>:   mov    (%rax),%eax
   0x0000000000401268 <+216>:   cmp    %eax,(%rbx)
   0x000000000040126a <+218>:   jle    0x401271 <phase_6+225>
   0x000000000040126c <+220>:   callq  0x401696 <explode_bomb>
   0x0000000000401271 <+225>:   mov    0x8(%rbx),%rbx
   0x0000000000401275 <+229>:   sub    $0x1,%ebp
   0x0000000000401278 <+232>:   jne    0x401262 <phase_6+210>
   0x000000000040127a <+234>:   mov    0x58(%rsp),%rax
   0x000000000040127f <+239>:   xor    %fs:0x28,%rax
   0x0000000000401288 <+248>:   je     0x40128f <phase_6+255>
   0x000000000040128a <+250>:   callq  0x400b90 <__stack_chk_fail@plt>
   0x000000000040128f <+255>:   add    $0x68,%rsp
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401293 <+259>:   pop    %rbx
   0x0000000000401294 <+260>:   pop    %rbp
   0x0000000000401295 <+261>:   pop    %r12
   0x0000000000401297 <+263>:   pop    %r13
   0x0000000000401299 <+265>:   retq
End of assembler dump.
```

phase_6_2

1. 初步理解

phase_6 显然是个难题。一看，这么长!

先看输入数据:

```
which has no line number information.
__GI___isoc99_sscanf (s=0x604950 <input_strings+400> "1 5 4 2 6 3 6", format=0x4029a1 "%d %d %d %d %d %d") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
(gdb)
```

“%d %d %d %d %d %d”，显然，他让我输入六个数字。

初步观察代码:

```
sub $0x1 %eax
cmp $0x5 %eax
```

意思显然是输入的第一个数不能大于 6。

不然而然想到我们的输入应该是小于等于 6 的整数。

2. 寻找答案

第六题分三个部分：

section_1：

检验所有数据 <6 且不相等；

section_2：

初始化地址和数组，用我输入的数据**将地址存到新的空间里**；

section_3：

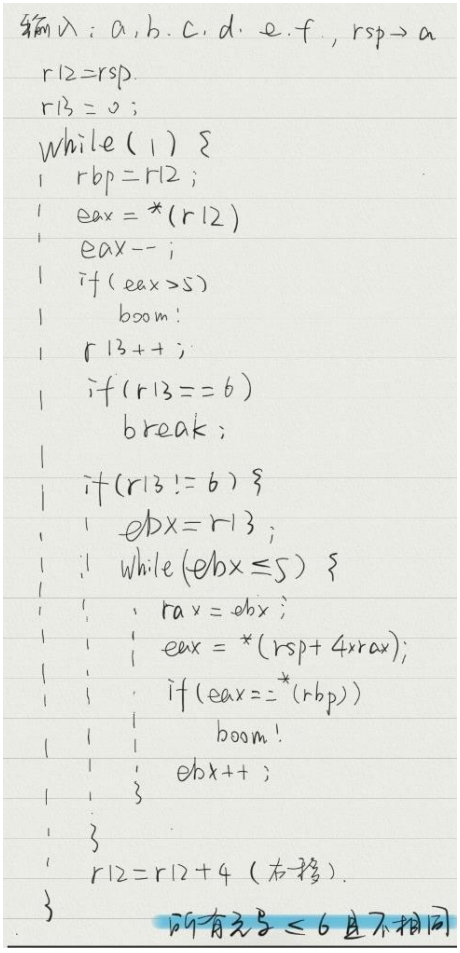
通过新数组比较大小，来判断新数组里**地址对应的值是否是从小到大排序**；即检验排序结果是否与输入匹配。

此外我将已经弄好的但没有排序的数组**数据**记录了下来，具体如下：

```
(gdb) x /24x $rdx
0x6042f0 <node1>:       0x000000fe      0x00000001      0x00604300      0x00000000
0x604300 <node2>:       0x000001dd      0x00000002      0x00604310      0x00000000
0x604310 <node3>:       0x0000032a      0x00000003      0x00604320      0x00000000
0x604320 <node4>:       0x000001a2      0x00000004      0x00604330      0x00000000
0x604330 <node5>:       0x00000162      0x00000005      0x00604340      0x00000000
0x604340 <node6>:       0x00000230      0x00000006      0x00000000      0x00000000
```

显然，虽然存储空间相邻，但这并不是一个严格的数组；因为每个对应的空间后面会跟着一个地址。这是一个**链表排序题**；不过他的方法比较特殊。下面详细分析上述三个过程。

section_1：检验输入数据



首先，输入格式如下：

```
format=0x4029a1 "%d %d %d %d %d %d"
```

所以我们的输入应该是 6 个数字。

第一个部分的主要功能有两个：

1. 第一个输入要小于等于 6；

   ```
   sub $0x1 %eax
   cmp $0x5 %eax
   ```

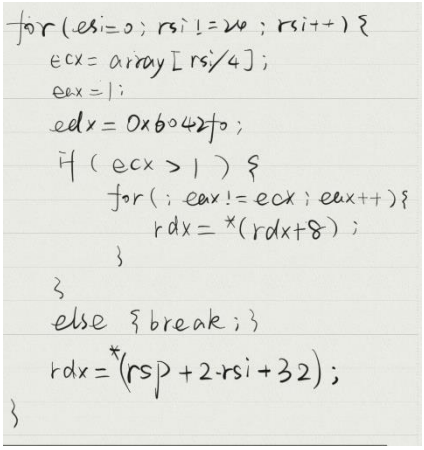
   所以第一个输入不可能大于 6，否则直接爆炸；
2. 所有输入不能相等。

   循环计数 6 次，每个循环内与后面几个数逐个匹配，若相等则爆炸。

根据这两个限制，我们很容易想到：
我们的输入是 123456 六个数的排列。

至于正确性，我们得知道其中真正的含义之后才知道。

section_2：创建新的地址数组



首先有个指令是将 0x6042f0 赋值给 edx 寄存器。
这是干什么呢？这是为排序开栈，0x6042f0 是排序首地址。

前提：

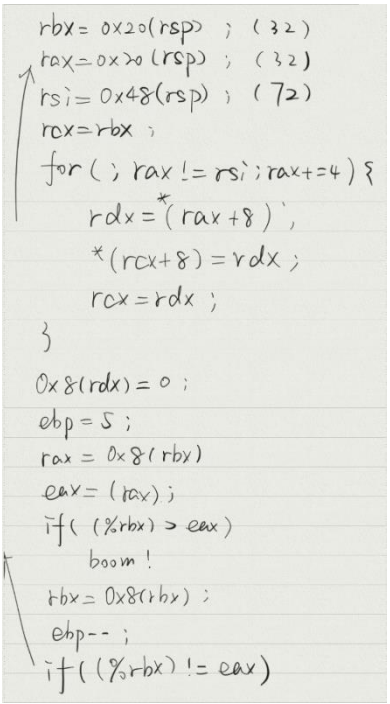
```
(gdb) x /6d $rsp
0x7fffffffea00: 1       5       4       2
0x7fffffffea10: 6       3
```

这是我的六个输入，被存在了 rsp 为首的连续空间中。
这就是我在这个循环中操作地址转移的关键。
（注，最后一行赋值代码写反了）

实现过程：

Step 1. 开辟新的临时变量以便于遍历我的输入数组；
Step 2. 每次循环找到我对应的输入数字，并将其赋值给 ecx；
Step 3. 通过 ecx 为位移量找到原来对应的链表的相应地址；
Step 4. 将得到的地址挨个存入到开辟空间中。

section_3：根据地址数组重新修改指针域



这段代码主要功能如下：

1. 通过双重循环：第一个循环找到对应要修改的指针域；
2. 将对应指针赋值回给原来的链表；
3. 判断当前元素是否小于指针域指向的元素（即下一个元素）；若大于，则直接爆炸，调用 explode_bomb 函数。

经过观察代码，知道是从小到大左右排序。

我们的输入是正确的元素顺序对应的原来链表中元素的位置

```
(gdb) x /24x $rdx
0x6042f0 <node1>:       0x000000fe      0x00000001      0x00604300      0x00000000
0x604300 <node2>:       0x000001dd      0x00000002      0x00604310      0x00000000
0x604310 <node3>:       0x0000032a      0x00000003      0x00604320      0x00000000
0x604320 <node4>:       0x000001a2      0x00000004      0x00604330      0x00000000
0x604330 <node5>:       0x00000162      0x00000005      0x00604340      0x00000000
0x604340 <node6>:       0x00000230      0x00000006      0x00000000      0x00000000
```

所以正确的答案是（1，5，4，2，6，3）。

运行结果：（显然已经排列好了，修改的是指针域，且进行的是类冒泡排序）

```
(gdb) x /24x 0x6042f0
0x6042f0 <node1>:       0x000000fe      0x00000001      0x00604330      0x00000000
0x604300 <node2>:       0x000001dd      0x00000002      0x00604340      0x00000000
0x604310 <node3>:       0x0000032a      0x00000003      0x00000000      0x00000000
0x604320 <node4>:       0x000001a2      0x00000004      0x00604300      0x00000000
0x604330 <node5>:       0x00000162      0x00000005      0x00604320      0x00000000
0x604340 <node6>:       0x00000230      0x00000006      0x00604310      0x00000000
```

3. 增加的理解

- 了解了链表的机器级存储结构，并且理解了链表的机器级相关操作；
- 熟悉了各类跳转操作，熟悉了循环操作。这题三个大循环，看循环的能力直接上了一档次；
- 理解了 rsp 的操作，即临时开栈，存储返回地址等。

阶段三：secret_phase

找到进入方式

首先，我知道有这个难题，但是找不到进入方法。它在 phase_defused 中可以被调用。

```
Dump of assembler code for function secret_phase:
=> 0x00000000004012d8 <+0>:     push   %rbx
   0x00000000004012d9 <+1>:     callq  0x40170b <read_line>
   0x00000000004012de <+6>:     mov    $0xa,%edx
   0x00000000004012e3 <+11>:    mov    $0x0,%esi
   0x00000000004012e8 <+16>:    mov    %rax,%rdi
   0x00000000004012eb <+19>:    callq  0x400c20 <strtol@plt>
   0x00000000004012f0 <+24>:    mov    %rax,%rbx
   0x00000000004012f3 <+27>:    lea    -0x1(%rax),%eax
   0x00000000004012f6 <+30>:    cmp    $0x3e8,%eax
   0x00000000004012fb <+35>:    jbe    0x401302 <secret_phase+42>
   0x00000000004012fd <+37>:    callq  0x401696 <explode_bomb>
   0x0000000000401302 <+42>:    mov    %ebx,%esi
   0x0000000000401304 <+44>:    mov    $0x604110,%edi
   0x0000000000401309 <+49>:    callq  0x40129a <fun7>
   0x000000000040130e <+54>:    cmp    $0x7,%eax
   0x0000000000401311 <+57>:    je     0x401318 <secret_phase+64>
   0x0000000000401313 <+59>:    callq  0x401696 <explode_bomb>
   0x0000000000401318 <+64>:    mov    $0x4026b8,%edi
   0x000000000040131d <+69>:    callq  0x400b70 <puts@plt>
   0x0000000000401322 <+74>:    callq  0x401831 <phase_defused>
   0x0000000000401327 <+79>:    pop    %rbx
   0x0000000000401328 <+80>:    retq
```

思考过程：

Step 1:

我想过在第三题换着七个答案输入，想着有一个能触发，但是都没有进入。

那我还有什么方法可以进入呢？

看见 secret_phase 中有 read_line。那我在第六题后面输入，会有什么发生吗？

我输入一个 6，就过了一个判断！

Step 2:

然后跟着找，在寄存器里找了个字符串，并且提醒我在第四题后边加上：

```
0x0000000000400c40 in __isoc99_sscanf@plt ()
(gdb) disas
Dump of assembler code for function __isoc99_sscanf@plt:
=> 0x0000000000400c40 <+0>:     jmpq   *0x20345a(%rip)        # 0x6040a0 <__isoc99_sscanf@got.plt>
   0x0000000000400c46 <+6>:     pushq  $0x11
   0x0000000000400c4b <+11>:    jmpq   0x400b20
End of assembler dump.
(gdb) stepi
__GI___isoc99_sscanf (s=0x6048b0 <input_strings+240> "216 4", format=0x4029f7 "%d %d %s") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
```

scanf_secret_phase

```
(gdb) x /s $esi
0x402a00:       "DrEvil"
```

“邪恶先生”

根据这个，再联系我们要求的输入”%d %d %s”，就可以得到答案了：

它**要求我们在第四题之后加上 DrEvil**。

于是，过了第二个关卡。

Step 3: 来到最后的输入 secret_phase 答案了。

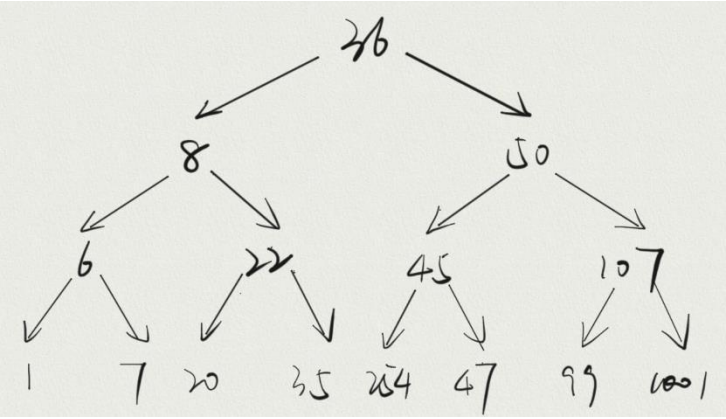
寻找 secret_phase 答案

直接放图，这题我做了蛮久的：

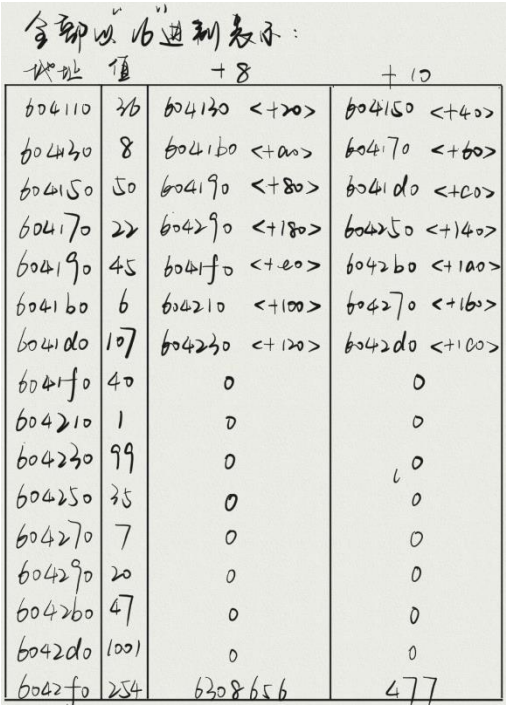
```
Continuing.
Curses, you've found the secret phase!
But finding it and solving it are quite different...

Breakpoint 3, 0x00000000004012d8 in secret_phase ()
(gdb) c
Continuing.
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```

这是我们正确输入后会出现的输出！

然后这是手写稿：



secret_phase_二叉树



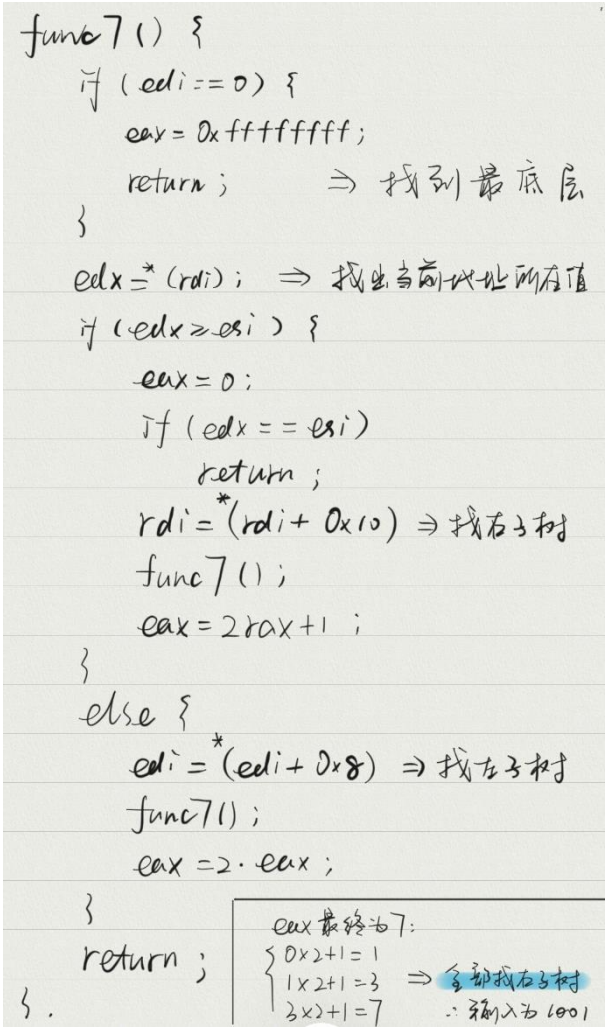
地址	值	+8	+10
6041l0	36	604130 <+20>	604150 <+40>
604130	8	6041b0 <+a0>	604170 <+60>
604150	50	604190 <+80>	6041d0 <+c0>
604170	22	6042l0 <+180>	6042l0 <+140>
604190	45	6041f0 <+e0>	604260 <+1a0>
6041b0	6	604210 <+100>	604270 <+160>
6041d0	107	604230 <+120>	6042d0 <+100>
6041f0	40	0	0
604210	1	0	0
604230	99	0	0
604250	35	0	0
604270	7	0	0
6042l0	20	0	0
6042b0	47	0	0
6042d0	1001	0	0
6042f0	254	630656	477

这是我在寄存器中发现的一个很奇怪的空间。

然后我把它抄下来后，经过链表的知识发现：

它是一个二叉树！

那他这是叫我干什么呢？接下来就是阅读代码了。



secret_phase

阅读代码，我马上接理解了代码的意思：
这个程序的意思就是，**要找我们输入的数**。

但会有一个初始值为 0 的变量：

1. 往**左子树**找一次：*2
2. 往**右子树**找一次：*2+1

而我要找的**立即数是** 7，但二叉树只有四层，能有三次计算的机会，我们需要把变量变成 7：

((2*0+1) *2 +1) *2 +1 = 7

所以我要一直找三次右子树，找到第四层。

找的步骤：

1. 根据我画的二叉树，我要找到 1001；
2. 如果数字匹配，则返回；
3. 数字比要找的数小，则找左子树；
4. 数字比要找的数字大，则找右子树。

所以我答案是 1001。

五、总结体会

遇到的问题&解决方法

Q1：一开始不会阅读汇编代码。
A1：经过 2*7 题的打磨，应该已经对机器及的指令表示比较熟悉。

Q2：不知道对应的寄存器的功能，以为只是和一般的临时变量一般。
A1：rax 为**返回值**；
rdi rsi r12 r13 r14 r15 为**被调用者寄存器**，可作为临时变量的作用参与子函数运行。
rsp 为**栈帧**，记录函数的返回地址，并且可以在上边开辟空间存储临时变量。

Q3：gdb 工具用不熟练，不知道该如何合理设置断点。
A3：失败是成功之母，BOOM!!! 一次之后就小心翼翼，慢慢就会了；
并且要学会将 break 和 continue 合起来用，这样能进行快速的调试。

Q4：不知道如何观测想要的值，打印想要的值。
A4：方法一：print value/*(address)；
方法二：x /d(/x/24d) address；
方法三：watch varible。

Q5：不清楚循环结构，导致一旦看到跳转指令就头疼。
A5：啃书，了解了循环体结构在机器级指令的基本实现形式。

Q6：不能很好的理解递归函数，尤其是关于传参和变量。
A6：做第四题：eax 是返回值；
pop/push 的寄存器在该次函数中可看作临时变量；
esi edi 等寄存器可看作默认传入参数。

Q7：不熟悉数组、链表、二叉树等数据结构的机器级表示；
A7：在做题中逐渐熟悉：
第五题：熟悉了数组的存储方式和使用方式
第六题：熟悉了链表的存储方式和使用方式，尤其是如何修改指针域；
第七题：熟悉了二叉树的存储方式和遍历方式（与链表如出一辙）。

Q8：不了解地址的妙用，尤其是在遇到 lea 指令的时候；
A8：一般对数据的间接操作都是用地址实现，除非必须用 mov 修改值或者做下标参数，否则不会轻易修改地址里面的值（在机器级可以把取值看作间接操作，操作地址才是直接操作）。

Q9：不清楚如何找到各个题目的正确输入形式；
A9：方法一：进入读取函数看 format 是什么；
方法二：打印相应寄存器（如 esi/edi）存的字符串，一般 scanf 会根据相应字符串形式读取数据。
补充：我们的输入都会以字符串形式存在相应位置，所谓的读取都是操作字符串。

挫败的感受：
我连着三四天拆炸弹拆到 1、2 点；面对不熟悉的指令和操作组合会感到很烦躁，但又很无力。

过关的感受：
感觉自己又行了，直到翻开下一个炸弹。

实验投入的精力：
64h+；总共 2*7 题加两个实验报告。
以 x86 为例：
1. 前三题平均 1h；
2. 后三题带完全理解所有指令 6h 一题；
3. 进入 secret_phase 1.5h；
4. 解决 secret_phase 4h；
5. 实验报告 12h。

建议：最好把时间延长一点，因为这题需要消化的东西还挺多的。

六、诚信声明（不签扣 10 分）

需要填写如下声明，并在底部给出手写签名的电子版。

我参考了以下资料：

1. 搜索如何打印地址里的字符串；
2. 搜索如何打印链表；
3. 《深入理解计算机系统》书籍中关于机器指令功能的表格。

在我提交的程序中，还在对应的位置以注释形式记录了具体的参考内容。

我独立完成了本次实验除以上方面之外的所有工作，包括分析、设计、编码、调试与测试。

我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

我从未使用他人代码，不管是原封不动地复制，还是经过某些等价转换。

我未曾也不会向同一课程（包括此后各届）的同学复制或公开我这份程序的代码，我有义务妥善保管好它们。

我编写这个程序无意于破坏或妨碍任何计算机系统的正常运行。

我清楚地知道，以上情况均为本课程纪律所禁止，若违反，对应的实验成绩将按照 0 分计。

（签名）