

# 北京邮电大学

## 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 ..... 3

二、实验环境 ..... 3

三、实验内容 ..... 3

四、实验步骤及实验分析 ..... 3

    准备工作 ..... 3

    阶段一：尝试调试 ..... 4

    阶段二：拆 phase\_1 -> phase\_6 ..... 4

        phase\_1 ..... 4

        phase\_2 ..... 5

        phase\_3 ..... 7

        phase\_4 ..... 9

        phase\_5 ..... 11

        phase\_6 ..... 13

    阶段三：secret\_phase ..... 18

        找到进入方式 ..... 18

        寻找 secret\_phase 答案 ..... 19

五、总结体会 ..... 21

六：彩蛋 ..... 23

七、诚信声明（不签扣 10 分） ..... 23

# 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 ARM 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): [yangyj98@bupt.edu.cn](mailto:yangyj98@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

本实验是建立在 x86 六个炸弹拆完的情况下进行的。所以在解题上会有很强的能力，但看懂 ARM 汇编代码的能力基本为 0。

所以我的准备工作是：

1. 熟悉 ARM 的各个操作指令；

2. 熟悉 ARM 寄存器的各种功能;
3. 熟悉 ARM 的指令特点和组合方法。

其实，在看完 ARM 后，我发现他是极为强大的。它相比汇编显得不那么汇编而更像汇编中的高级语言。因为它有太多组合起来的指令去适应常见数据结构的计算。

## 阶段一：尝试调试

首先尝试上手 ARM。

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. 不会看寄存器。一开始总是认为不同寄存器的值是胡乱的，是中间的计算量，无法跟踪；
3. 不能理解寄存器为什么会是数字命名法！

## 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

```
0000000000401028 <phase_1>:
401028: a9bf7bfd stp x29, x30, [sp, #-16]!
40102c: 910003fd mov x29, sp
401030: b0000001 adrp x1, 402000 <submitr+0x3a4>
401034: 911b2021 add x1, x1, #0x6c8
401038: 9400016f bl 4015f4 <strings_not_equal>
40103c: 35000060 cbnz w0, 401048 <phase_1+0x20>
401040: a8c17bfd ldp x29, x30, [sp], #16
401044: d65f03c0 ret
401048: 9400022c bl 4018f8 <explode_bomb>
40104c: 17fffffd b 401040 <phase_1+0x18>
```

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

```
(gdb) x /s $x0
0x420728 <input_strings>:      "Only you can give me that feeling."
```

根据寄存器状态可知：进入 strings\_not\_equal 后，我们的寄存器参数 x0 打印字符串可得：“Only you can give me that feeling.”

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
- w0 一般做函数返回值
- w29 是重要函数参数
- sp 是栈帧

## phase\_2

```
000000000401050 <phase_2>:
401050: a9bb7bfd      stp     x29, x30, [sp, #-80]!
401054: 910003fd      mov     x29, sp
401058: a90153f3      stp     x19, x20, [sp, #16]
40105c: f90013f5      str     x21, [sp, #32]
401060: 9100e3a1      add     x1, x29, #0x38
401064: 94000234      bl     401934 <read_six_numbers>
401068: b9403ba0      ldr     w0, [x29, #56]
40106c: 37f800a0      tbnz    w0, #31, 401080 <phase_2+0x38>
401070: d2800014      mov     x20, #0x0
401074: d2800033      mov     x19, #0x1
401078: 9100e3b5      add     x21, x29, #0x38
40107c: 14000007      b       401098 <phase_2+0x48>
401080: 9400021e      bl     4018f8 <explode_bomb>
401084: 17fffffb      b       401070 <phase_2+0x20>
401088: 91000673      add     x19, x19, #0x1
40108c: 91001294      add     x20, x20, #0x4
401090: f1001a7f      cmp     x19, #0x6
401094: 54000100      b.eq    4010b4 <phase_2+0x64> // b.none
401098: b8756a80      ldr     w0, [x20, x21]
40109c: 0b130000      add     w0, w0, w19
4010a0: b8737aa1      ldr     w1, [x21, x19, lsl #2]
4010a4: 6b00003f      cmp     w1, w0
4010a8: 54fffff0      b.eq    401088 <phase_2+0x38> // b.none
4010ac: 94000213      bl     4018f8 <explode_bomb>
4010b0: 17fffffb      b       401080 <phase_2+0x38>
4010b4: a94153f3      ldp     x19, x20, [sp, #16]
4010b8: f94013f5      ldr     x21, [sp, #32]
4010bc: a8c57bfd      ldp     x29, x30, [sp], #80
4010c0: d65f03c0      ret

000000000401934 <read_six_numbers>:
401934: a9bf7bfd      stp     x29, x30, [sp, #-16]!
401938: 910003fd      mov     x29, sp
40193c: 91005027      add     x7, x1, #0x14
401940: 91004026      add     x6, x1, #0x10
401944: 91003025      add     x5, x1, #0xc
401948: 91002024      add     x4, x1, #0x8
40194c: 91001023      add     x3, x1, #0x4
401950: aa0103e2      mov     x2, x1
401954: b0000001      adrp    x1, 402000 <submitr+0x3a4>
401958: 91232021      add     x1, x1, #0x8c8
40195c: 97ffffd05     bl     400d70 <__isoc99_sscanf@plt>
401960: 7100141f      cmp     w0, #0x5
401964: 5400006d      b.le    401970 <read_six_numbers+0x3c>
401968: a8c17bfd      ldp     x29, x30, [sp], #16
40196c: d65f03c0      ret
401970: 97ffffe2      bl     4018f8 <explode_bomb>
```

phase\_2

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！

```
40195c: 97ffffd05     bl     400d70 <__isoc99_sscanf@plt>
401960: 7100141f      cmp     w0, #0x5
401964: 5400006d      b.le    401970 <read_six_numbers+0x3c>
401968: a8c17bfd      ldp     x29, x30, [sp], #16
40196c: d65f03c0      ret
401970: 97ffffe2      bl     4018f8 <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：

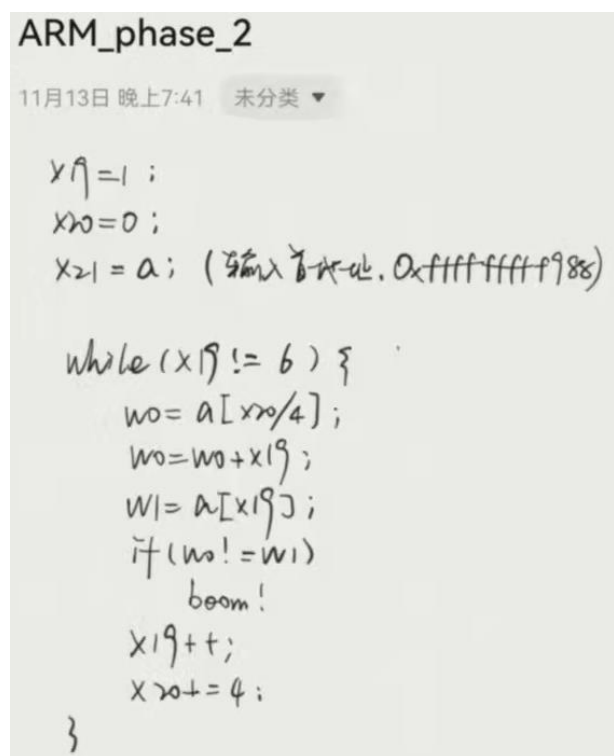
```
0x0000000000400c40 in __isoc99_sscanf@plt ()
(gdb) stepi
GI __isoc99_sscanf (s=0x604810 <input_strings+80> "0 1 1 2 3 5", format=0x4029a1 "%d %d %d %d %d %d") at isoc99_sscanf.c:24
24      isoc99_sscanf.c: No such file or directory.
(gdb)
```

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

根据观测寄存器可知，我输入的数是被保存在了 **sp**（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. 有一个临时变量 **x19** 初始值为 **#1**
2. 匹配第一个输入是否为 **\$0x0**;
3. 匹配第二个输入是否为 **x19+前一个数字**;
4. 循环 **\$0x6** 次;
5. 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**;
6. 执行完循环则返回，并输出 **phase\_defused**。

所以我只需要输入数列前六项的数据，这个就能过了。

答案是 **(0, 1, 3, 6, 10, 15)**

## 3. 增加的理解

- 要理解机器代码实现**循环**：通过 **cmp/test** 配合 **b**，相当于 **do-while** 或者 **goto**。
- **栈帧 sp** 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

```
0000000004010c4: <phase_3>;
4010c4: a9be7b7d    stp     x29, x30, [sp, #-32]!
4010c8: 910003fd    mov     x29, sp
4010cc: 910063a4    add     x4, x29, #0x18
4010d0: 91005fa3    add     x3, x29, #0x17
4010d4: 910073a2    add     x2, x29, #0x1c
4010d8: b0000001    adrp    x1, 402000 <submitr+0x3a4>
4010dc: 911bc021    add     x1, x1, #0x6f0
4010e0: 97ffff74    bl      400d70 <_isoc99_sscanf@plt>
4010e4: 7100081f    cmp     w0, #0x2
4010e8: 5400026d    b.le    401134 <phase_3+0x70>
4010ec: b9401fa9    ldr     w0, [x29, #28]
4010f0: 71000c1f    cmp     w0, #0x3
4010f4: 54000560    b.eq    4011a0 <phase_3+0xdc> // b.none
4010f8: 5400022d    b.le    40113c <phase_3+0x78>
4010fc: 7100141f    cmp     w0, #0x5
401100: 540006c0    b.eq    4011d0 <phase_3+0x114> // b.none
401104: 540005cb    b.lt    4011bc <phase_3+0xf8> // b.tstop
401108: 7100181f    cmp     w0, #0x6
40110c: 54000740    b.eq    4011f4 <phase_3+0x130> // b.none
401110: 71001c1f    cmp     w0, #0x7
401114: 540007e1    b.ne    401210 <phase_3+0x14c> // b.any // #122
401118: 52800f40    mov     w0, #0x7a
40111c: b9401ba1    ldr     w1, [x29, #24]
401120: 7105243f    cmp     w1, #0x149
401124: 540007a0    b.eq    401218 <phase_3+0x154> // b.none
401128: 940001f4    bl      4018f8 <explode_bomb>
40112c: 52800f40    mov     w0, #0x7a // #122
401130: 1400003a    b       401218 <phase_3+0x154>
401134: 940001f1    bl      4018f8 <explode_bomb>
401138: 17fffffd    b       4010ec <phase_3+0x28>
40113c: 7100041f    cmp     w0, #0x1
401140: 54000140    b.eq    401168 <phase_3+0xa4> // b.none
401144: 5400020c    b.gt    401184 <phase_3+0xc0>
401148: 35000640    cbnz    w0, 401210 <phase_3+0x14c>
40114c: 52800da0    mov     w0, #0xd // #109
401150: b9401ba1    ldr     w1, [x29, #24]
401154: 7106083f    cmp     w1, #0x182
401158: 54000600    b.eq    401218 <phase_3+0x154> // b.none
40115c: 940001e7    bl      4018f8 <explode_bomb>
401160: 52800da0    mov     w0, #0xd // #109
```

phase\_3\_1

```
401200: 54000c0     b.eq     401218 <phase_3+0x154> // b.none
401204: 940001bd    bl      4018f8 <explode_bomb>
401208: 52800d00    mov     w0, #0x68 // #104
40120c: 14000003    b       401218 <phase_3+0x154>
401210: 940001ba    bl      4018f8 <explode_bomb>
401214: 52800e40    mov     w0, #0x72 // #114
401218: 39405fa1    ldrb    w1, [x29, #23]
40121c: 6b00003f    cmp     w1, w0
401220: 54000040    b.eq    401228 <phase_3+0x164> // b.none
401224: 940001b5    bl      4018f8 <explode_bomb>
401228: a8c27bfd    ldp     x29, x30, [sp], #32
40122c: d65f03c0    ret
```

phase\_3\_3

```
401160: 52800da0    mov     w0, #0xd // #109
401164: 1400002d    b       401218 <phase_3+0x154>
401168: 52800d20    mov     w0, #0x69 // #105
40116c: b9401ba1    ldr     w1, [x29, #24]
401170: 7109c03f    cmp     w1, #0x270
401174: 54000520    b.eq    401218 <phase_3+0x154> // b.none
401178: 940001e0    bl      4018f8 <explode_bomb>
40117c: 52800d20    mov     w0, #0x69 // #105
401180: 14000026    b       401218 <phase_3+0x154>
401184: 52800ec0    mov     w0, #0x76 // #118
401188: b9401ba1    ldr     w1, [x29, #24]
40118c: 7101783f    cmp     w1, #0x4a
401190: 54000440    b.eq    401218 <phase_3+0x154> // b.none
401194: 940001d9    bl      4018f8 <explode_bomb>
401198: 52800ec0    mov     w0, #0x76 // #118
40119c: 1400001f    b       401218 <phase_3+0x154>
4011a0: 52800e20    mov     w0, #0x71 // #113
4011a4: b9401ba1    ldr     w1, [x29, #24]
4011a8: 710b4c3f    cmp     w1, #0x2d3
4011ac: 54000360    b.eq    401218 <phase_3+0x154> // b.none
4011b0: 940001d2    bl      4018f8 <explode_bomb>
4011b4: 52800e20    mov     w0, #0x71 // #113
4011b8: 14000018    b       401218 <phase_3+0x154>
4011bc: 52800f00    mov     w0, #0x78 // #120
4011c0: b9401ba1    ldr     w1, [x29, #24]
4011c4: 710df43f    cmp     w1, #0x37d
4011c8: 54000280    b.eq    401218 <phase_3+0x154> // b.none
4011cc: 940001cb    bl      4018f8 <explode_bomb>
4011d0: 52800f00    mov     w0, #0x78 // #120
4011d4: 14000011    b       401218 <phase_3+0x154>
4011d8: 52800d40    mov     w0, #0x6a // #106
4011dc: b9401ba1    ldr     w1, [x29, #24]
4011e0: 7103e03f    cmp     w1, #0xf8
4011e4: 540001a0    b.eq    401218 <phase_3+0x154> // b.none
4011e8: 940001c4    bl      4018f8 <explode_bomb>
4011ec: 52800d40    mov     w0, #0x6a // #106
4011f0: 1400000a    b       401218 <phase_3+0x154>
4011f4: 52800d00    mov     w0, #0x68 // #104
4011f8: b9401ba1    ldr     w1, [x29, #24]
4011fc: 710be03f    cmp     w1, #0x2f8
401200: 540000c0    b.eq    401218 <phase_3+0x154> // b.none
```

phase\_3\_2

## 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。

```
(gdb) x /s $x1
0x4026f0: "%d %c %d"
```

format=" %d %c %d"，我们要输入两个数字中间加一个字符来读入程序。

## 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：

```
(gdb) x /d $x29+28
0xffffffff99c: 3
(gdb) x /c $x29+23
0xffffffff997: 113 'q'
```

1. 我们存的第一个数是在 sp 的地址中；
2. 字符就在它后面四个字节（数字占 4 个字节）。
3. 第二个数字在字符后边一个字节



第一个数:

```
0x00000000004010e0 <+28>: bl 0x400d70 <_isoc99_sscanf@plt>
0x00000000004010e4 <+32>: cmp w0, #0x2
0x00000000004010e8 <+36>: b.le 0x401134 <phase_3+112>
0x00000000004010ec <+40>: ldr w0, [x29, #28]
0x00000000004010f0 <+44>: cmp w0, #0x3
0x00000000004010f4 <+48>: b.eq 0x4011a0 <phase_3+220> // b.none
0x00000000004010f8 <+52>: b.le 0x40113c <phase_3+120>
0x00000000004010fc <+56>: cmp w0, #0x5
0x0000000000401100 <+60>: b.eq 0x4011d8 <phase_3+276> // b.none
0x0000000000401104 <+64>: b.lt 0x4011bc <phase_3+248> // b.tstop
0x0000000000401108 <+68>: cmp w0, #0x6
0x000000000040110c <+72>: b.eq 0x4011f4 <phase_3+304> // b.none
0x0000000000401110 <+76>: cmp w0, #0x7
0x0000000000401114 <+80>: b.ne 0x401210 <phase_3+332> // b.any
0x0000000000401118 <+84>: mov w0, #0x7a // #122
0x000000000040111c <+88>: ldr w1, [x29, #24]
0x0000000000401120 <+92>: cmp w1, #0x149
0x0000000000401124 <+96>: b.eq 0x401218 <phase_3+340> // b.none
0x0000000000401128 <+100>: bl 0x4018f8 <explode_bomb>
0x000000000040112c <+104>: mov w0, #0x7a // #122
0x0000000000401130 <+108>: b 0x401218 <phase_3+340>
0x0000000000401134 <+112>: bl 0x4018f8 <explode_bomb>
```

根据 `cmpl $0x7, (%rsp)` 知, 第一个数要等于 3, 5, 6 或者 7 等才能才能跳过第一个炸弹。

这和 x86 的基本一样, 是一个 `switch` 类型题目。

那么还是一样, 我选择 **3** 作为我的第一个输入, 简单。

第二个数:

```
0x00000000004011a0 <+220>: mov w0, #0x71 // #113
0x00000000004011a4 <+224>: ldr w1, [x29, #24]
0x00000000004011a8 <+228>: cmp w1, #0x2d3
0x00000000004011ac <+232>: b.eq 0x401218 <phase_3+340> // b.none
0x00000000004011b0 <+236>: bl 0x4018f8 <explode_bomb>
```

这是我在输入 3 后回跳转到的地方。这里的功能是把我的第二个数与 723 做比较, 不相等则爆炸。此外, 这里还把 113 赋值给了 w0, 后面有用。

字符:

这是上一个数字相等后跳转到的地方。

```
0x0000000000401218 <+340>: ldrrb w1, [x29, #23]
0x000000000040121c <+344>: cmp w1, w0
0x0000000000401220 <+348>: b.eq 0x401228 <phase_3+356> // b.none
0x0000000000401224 <+352>: bl 0x4018f8 <explode_bomb>
0x0000000000401228 <+356>: ldp x29, x30, [sp], #32
0x000000000040122c <+360>: ret
```

这里是将刚刚被赋值成 **113** 的 w0 与我的字符作比较。显然, 我的字符 ascii 码应该是 113 所以字符是 'q'。

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。同时, 比较的字符也不一样。

我选择了一个比较小的值作为开始, 就是 **(3, q, 723)** -- 怕错  $(\wedge \sim \wedge)^*$

### 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

```
0000000000401284 <phase_4>:
401284: a9be7bfd      stp     x29, x30, [sp, #-32]!
401288: 910003fd      mov     x29, sp
40128c: 910063a3      add     x3, x29, #0x18
401290: 910073a2      add     x2, x29, #0x1c
401294: b0000001      adrp    x1, 402000 <subm1tr+0x3a4>
401298: 911c0021      add     x1, x1, #0x700
40129c: 97fff0b5      bl      400d70 <__isoc99_sscanf@plt>
4012a0: 7100001f      cmp     w0, #0x2
4012a4: 54000001      b.ne    4012b4 <phase_4+0x30> // b.any
4012a8: b9401fa0      ldr     w0, [x29, #28]
4012ac: 7100381f      cmp     w0, #0xe
4012b0: 54000049      b.ls    4012b8 <phase_4+0x34> // b.plast
4012b4: 94000191      bl      4018f8 <explode_bomb>
4012b8: 520001c2      mov     w2, #0xe // #14
4012bc: 52000001      mov     w1, #0x0 // #0
4012c0: b9401fa0      ldr     w0, [x29, #28]
4012c4: 97ffffdb      bl      401230 <func4>
4012c8: 71003c1f      cmp     w0, #0xf
4012cc: 54000001      b.ne    4012dc <phase_4+0x58> // b.any
4012d0: b9401ba0      ldr     w0, [x29, #24]
4012d4: 71003c1f      cmp     w0, #0xf
4012d8: 54000040      b.eq    4012e0 <phase_4+0x5c> // b.none
4012dc: 94000187      bl      4018f8 <explode_bomb>
4012e0: a8c27bfd      ldp     x29, x30, [sp], #32
4012e4: d65f03c0      ret

0000000000401230 <func4>:
401230: a9be7bfd      stp     x29, x30, [sp, #-32]!
401234: 910003fd      mov     x29, sp
401238: f9000bf3      str     x19, [sp, #16]
40123c: 4b010053      sub     w19, w2, w1
401240: 0b537e73      add     w19, w19, w19, lsr #31
401244: 0b930433      add     w19, w1, w19, asr #1
401248: 6b00027f      cmp     w19, w0
40124c: 540000cc      b.gt    401264 <func4+0x34>
401250: 5400012b      b.lt    401274 <func4+0x44> // b.tstop
401254: 2a1303e0      mov     w0, w19
401258: f9400bf3      ldr     x19, [sp, #16]
40125c: a8c27bfd      ldp     x29, x30, [sp], #32
401260: d65f03c0      ret
401264: 51000662      sub     w2, w19, #0x1
401268: 97fffff2      bl      401230 <func4>
40126c: 0b000273      add     w19, w19, w0
401270: 17fffff9      b       401254 <func4+0x24>
401274: 11000661      add     w1, w19, #0x1
401278: 97ffffee      bl      401230 <func4>
40127c: 0b000273      add     w19, w19, w0
401280: 17fffff5      b       401254 <func4+0x24>
```

phase\_4

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，**sp 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 **x0**。

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**；否则调用 **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。

跑进 scanf 后：

```
(gdb) x /s $x1
0x402700:      "%d %d"
```

scanf\_4

所以我要输入两个数字。

### 2. 寻找答案

第一个数：

```
ldr     w0, [x29, #28]
cmp     w0, #0xe
b.ls    4012b8 <phase_4+0x34> // b.plast
bl      4018f8 <explode_bomb>
```

首先知道，w0 就是我的第一个数字。

然后，我必须让他小于等于 14，否则，直接爆炸。

但我不知道其他任何信息。得看看 func4 里面有什么。

### 第二个数：

当第一个数通过检测后，就会将 w0 传参进入 func4，然后跑出来一个答案。

显然，这里的功能是二分查找：

首先找到传入的参数直观观察 func4 可知，我们的 edi 是控制深度的量。

函数如下：

根据深度控制条件反推，我这里代码的功能是：

Step 1. sp 开栈，从初到末记数；

Step 2. 对每一层，sp, x29, x19 都可以视为全局变量；

Step 3. 在每一层，找数，大了往左，小了往右

### 求出答案：

最终答案计算为 ( 5, 15 )

## 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)
- 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

```
00000000004012e8 <phase_5>:
4012e8: a9bd7bfd      stp     x29, x30, [sp, #-48]!
4012ec: 910003fd      mov     x29, sp
4012f0: f9000bf3      str     x19, [sp, #16]
4012f4: aa0003f3      mov     x19, x0
4012f8: 940000b4      bl      4015c8 <string_length>
4012fc: 7100181f      cmp     w0, #0x6
401300: 540002a1      b.ne    401354 <phase_5+0x6c> // b.any
401304: d2800000      mov     x0, #0x0 // #0
401308: 9100a3a3      add     x3, x29, #0x28
40130c: b0000002      adrp    x2, 402000 <submitr+0x3a4>
401310: 911ae042      add     x2, x2, #0x6b8
401314: 38606a61      ldrb    w1, [x19, x0]
401318: 12000c21      and     w1, w1, #0xf
40131c: 3861c841      ldrb    w1, [x2, w1, sxtw]
401320: 38236801      strb    w1, [x0, x3]
401324: 91000400      add     x0, x0, #0x1
401328: f100181f      cmp     x0, #0x6
40132c: 54ffff41      b.ne    401314 <phase_5+0x2c> // b.any
401330: 3900bbbf      strb    wzr, [x29, #46]
401334: b0000001      adrp    x1, 402000 <submitr+0x3a4>
401338: 911c2021      add     x1, x1, #0x708
40133c: 9100a3a0      add     x0, x29, #0x28
401340: 940000ad      bl      4015f4 <strings_not_equal>
401344: 350000c0      cbnz    w0, 40135c <phase_5+0x74>
401348: f9400bf3      ldr     x19, [sp, #16]
40134c: a8c37bfd      ldp     x29, x30, [sp], #48
401350: d65f03c0      ret
401354: 94000169      bl      4018f8 <explode_bomb>
401358: 17ffffeb      b       401304 <phase_5+0x1c>
40135c: 94000167      bl      4018f8 <explode_bomb>
401360: 17fffffa      b       401348 <phase_5+0x60>
```

phase\_5

### 1. 初步理解程序

```
0x00000000004012f8 <+16>:    bl      0x4015c8 <string_length>
0x00000000004012fc <+20>:    cmp     w0, #0x6
0x0000000000401300 <+24>:    b.ne    0x401354 <phase_5+108> // b.any
```

<string\_length>显然就是让我们输入字符串了。

然后判断输入个数是否为 6，不为 6 则直接爆炸。所以我的字符串是六个字符。

### 2. 寻找答案

根据我的反汇编至源程序的代码可知：

```
while (x0++ != 6)
    [x3, x0] = [x2, a[x0] mod 16];
callq <string_not_equal>
```

这里有点问题，应该是++x0

它通过我的输入字符的 Ascii 码 mod16 去找对应下标的字符来凑出 **flyers**

```
(gdb) x /s $x2
0x4026b8 <array.4328>: "maduiersnfotvbylonly you can give me that feeling."
(gdb) x /s $x1
0x402708: "flyers"
```

- Step 1. 用下标 x0 循环六次;
- Step 2. 通过我的输入字符的 Ascii 码 mod16 算出对应字符串的下标;
- Step 3. 将对应字符串下标的字符传送到新开辟的一片地址中;
- Step 4. 将新生成的六个字符形成的字符串与“flyers”比较; 正确则跳出, 否则爆炸。

图中的 Array 数组哪里来?

```
(gdb) x /s $x2
0x4026b8 <array.4328>: "maduiersnfotvbylonly you can give me that feeling."
```

就在 x2 所指向的地址里。

数据关系整理如下:

maduiersnfotvbyl  
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  
109 97 100 117 105 101 114 115 110 102 111 116 118 93 121 108  
13 1 4 5 9 5 2 3 14 6 15 4 6 2 9 12

flyers: 9 15 14 5 6 7  
从a开始找: i o n e f g

功能: 找  $\text{Ascii} \bmod 16$  的下标  
字母来凑出 "flyers".

如左图所示, 我要找出“flyers”, 只需要输入 Ascii 码 mod16 分别为:  
“9 15 14 5 6 7”的一个字符串。

为了方便找, 我就用了 26 个英文字母了里面找了。

所以答案是:  
“ione fg”

### 3. 增加的理解

- 认识到数组在机器中的存储方式, 以及 **str** 和 **ldr** 指令对地址空间的重要性。
- 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:
=> 0x00000000401190 <+0>: push %r13
0x00000000401192 <+2>: push %r12
0x00000000401194 <+4>: push %rbp
0x00000000401195 <+6>: push %rbx
0x00000000401196 <+8>: sub $0x68,%rsp
0x0000000040119a <+10>: mov %fs:0x28,%rax
0x000000004011a3 <+19>: mov %rax,0x58(%rsp)
0x000000004011a8 <+24>: xor %eax,%eax
0x000000004011aa <+26>: mov %rsp,%rsi
0x000000004011ad <+29>: callq 0x4016cc <read_six_numbers>
0x000000004011b2 <+34>: mov %rsp,%r12
0x000000004011b5 <+37>: mov $0x0,%r13d
0x000000004011b8 <+40>: mov %r12,%rbp
0x000000004011be <+46>: mov (%r12),%eax
0x000000004011c2 <+50>: sub $0x1,%eax
0x000000004011c5 <+53>: cmp $0x5,%eax
0x000000004011c8 <+56>: jbe 0x4011cf <phase_6+63>
0x000000004011ca <+58>: callq 0x401696 <explode_bomb>
0x000000004011cf <+63>: add $0x1,%r13d
0x000000004011d3 <+67>: cmp $0x6,%r13d
0x000000004011d7 <+71>: je 0x401216 <phase_6+134>
0x000000004011d9 <+73>: mov %r13d,%ebx
0x000000004011dc <+76>: movslq %ebx,%rax
0x000000004011df <+79>: mov (%rsp,%rax,4),%eax
0x000000004011e2 <+82>: cmp %eax,0x0(%rbp)
0x000000004011e5 <+85>: jne 0x4011ec <phase_6+92>
0x000000004011e7 <+87>: callq 0x401696 <explode_bomb>
0x000000004011ec <+92>: add $0x1,%ebx
0x000000004011ef <+95>: cmp $0x5,%ebx
0x000000004011f2 <+98>: jle 0x4011dc <phase_6+76>
0x000000004011f4 <+100>: add $0x4,%r12
0x000000004011f8 <+104>: jmp 0x4011bb <phase_6+43>
0x000000004011fa <+106>: mov 0x8(%rdx),%rdx
0x000000004011fe <+110>: add $0x1,%eax
0x00000000401201 <+113>: cmp %ecx,%eax
0x00000000401203 <+115>: jne 0x4011fa <phase_6+106>
--Type <RET> for more, q to quit, c to continue without paging--
0x00000000401205 <+117>: mov %rdx,0x20(%rsp,%rsi,2)
0x0000000040120a <+122>: add $0x4,%rsi
0x0000000040120e <+126>: cmp $0x18,%rsi
```

phase\_6\_1

```
0x0000000040120e <+126>: cmp $0x18,%rsi
0x00000000401212 <+130>: jne 0x40121b <phase_6+139>
0x00000000401214 <+132>: jmp 0x40122f <phase_6+159>
0x00000000401216 <+134>: mov $0x0,%esi
0x0000000040121b <+139>: mov (%rsp,%rsi,1),%ecx
0x0000000040121e <+142>: mov $0x1,%eax
0x00000000401223 <+147>: mov $0x6042f0,%edx
0x00000000401228 <+152>: cmp $0x1,%ecx
0x0000000040122b <+155>: jg 0x4011fa <phase_6+106>
0x0000000040122d <+157>: jmp 0x401205 <phase_6+117>
0x0000000040122f <+159>: mov 0x20(%rsp),%rbx
0x00000000401234 <+164>: lea 0x20(%rsp),%rax
0x00000000401239 <+169>: lea 0x48(%rsp),%rsi
0x0000000040123e <+174>: mov %rbx,%rcx
0x00000000401241 <+177>: mov 0x8(%rax),%rdx
0x00000000401245 <+181>: mov %rdx,0x8(%rcx)
0x00000000401249 <+185>: add $0x8,%rax
0x0000000040124d <+189>: mov %rdx,%rcx
0x00000000401250 <+192>: cmp %rsi,%rax
0x00000000401253 <+195>: jne 0x401241 <phase_6+177>
0x00000000401255 <+197>: movq $0x0,0x0(%rdx)
0x0000000040125d <+205>: mov $0x5,%ebp
0x00000000401262 <+210>: mov 0x8(%rbx),%rax
0x00000000401266 <+214>: mov (%rax),%eax
0x00000000401268 <+216>: cmp %eax,0(%rbx)
0x0000000040126a <+218>: jle 0x401271 <phase_6+225>
0x0000000040126c <+220>: callq 0x401696 <explode_bomb>
0x00000000401271 <+225>: mov 0x8(%rbx),%rbx
0x00000000401275 <+229>: sub $0x1,%ebp
0x00000000401278 <+232>: jne 0x401262 <phase_6+210>
0x0000000040127a <+234>: mov 0x58(%rsp),%rax
0x0000000040127f <+239>: xor %fs:0x28,%rax
0x00000000401288 <+248>: je 0x40128f <phase_6+255>
0x0000000040128a <+250>: callq 0x400b90 <__stack_chk_fail@plt>
0x0000000040128f <+255>: add $0x68,%rsp
--Type <RET> for more, q to quit, c to continue without paging--
0x00000000401293 <+259>: pop %rbx
0x00000000401294 <+260>: pop %rbp
0x00000000401295 <+261>: pop %r12
0x00000000401297 <+263>: pop %r13
0x00000000401299 <+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") 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。

这个题目和 x86 的如出一辙，这里我就不太过赘述。

## 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链表数据记录了下来，具体如下：

◦ node1	0x420110	0x291
+8		0x420288
◦ n1	0x420120	0x24
+8		0x420138
+16		0x420150
◦ n21	0x420138	0x8
+8		0x420198
+16		0x420168
◦ n22	0x420150	0x420180
+8		0x420180
+16		0x420160
◦ n32	0x420168	0x16
+8		0x420240
+16		0x420210
◦ n33	0x420180	0x2d
+8		0x4201c8
+16		0x420258
◦ n31	0x420198	0x6
+8		0x4201e0
+16		0x420228
◦ n34	0x420160	0x6b
+8		0x4201f8
+16		0x420270
◦ n45	0x4201c8	0x28
◦ n41	0x4201e0	0x1
◦ n47	0x4201f8	0xb3
◦ n44	0x420210	0x23
◦ n42	0x420228	0x7
◦ n43	0x420240	0x14
◦ n46	0x420258	0x2f
◦ n48	0x420270	0x3e9
◦ node2	0x420288	0x3a6
+8		0x98
◦ node3	0x420298	0xc4

可见，它在链表头和后面之间直接塞了个第七题的二叉树！



所以我直接打印出了比较好看一点的格式：

```
(gdb) x /4xw 0x420110
0x420110 <node1>:      0x00000291      0x00000001      0x00420288      0x00000000
(gdb) x /20xw 0x420288
0x420288 <node2>:      0x000003a6      0x00000002      0x00420298      0x00000000
0x420298 <node3>:      0x000000c4      0x00000003      0x004202a8      0x00000000
0x4202a8 <node4>:      0x00000060      0x00000004      0x004202b8      0x00000000
0x4202b8 <node5>:      0x000001ca      0x00000005      0x004202c8      0x00000000
0x4202c8 <node6>:      0x00000398      0x00000006      0x00000000      0x00000000
```

这个是没有排好序的状态。

## 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 (x86)

```
for (esi=0; rsi!=20; rsi++){
    ecx=array[rsi/4];
    eax=1;
    edx=0x6042f0;
    if (ecx>1){
        for (; eax!=ecx; eax++){
            rdx=*(rdx+8);
        }
    }
    else {break;}
    rdx=*(rsp+2*rsi+32);
}
```

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
0x7fffffffefa0: 1      5      4      2
0x7fffffffefa10: 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。

ARM 的如出一辙:

```

W3=0.
while (X3!=6)
    W2=a[W3]; //输入
    W0=1;
    W1=X4+0x110; (0x420110)
    while (W2>W0) {
        X1=[X1+8];
        W0++;
    }
    [X5,X3]=X1;
    X3++;
}
X19=[X29,7x8]
X0=[X29,8x8].
[X19,9x8]=X0;
X1=[X29,9x8].
[X0,8]=X1;
X0=[X29,10x8].
[X1,8]=X0.
X1=[X29,11x8].
[X0,8]=X1.
X0=[X29,12x8].
[X1,8]=X0;
[X5,8]=X29;
W0=5.
X19=[X19,8];
W0--;
do {
    X0=[X19,8]
    W1=[X19]
    W0=[X0].
} while (W1≤W0)

```

都是修改尾链表, 来达到排序的效果。

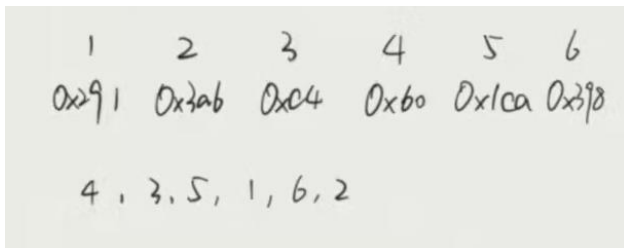
### section\_3: 根据地址数组重新修改指针域

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



1 2 3 4 5 6  
0x291 0x3ab 0x04 0xb0 0x1ca 0x398  
4, 3, 5, 1, 6, 2

所以正确的答案是 (4, 3, 5, 1, 6, 2)。

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 /4xw 0x420110
0x420110 <node1>:      0x00000291      0x00000001      0x004202c8      0x00000000
(gdb) x /20xw 0x420288
0x420288 <node2>:      0x000003a6      0x00000002      0x00000000      0x00000000
0x420298 <node3>:      0x000000c4      0x00000003      0x004202b8      0x00000000
0x4202a8 <node4>:      0x00000060      0x00000004      0x00420298      0x00000000
0x4202b8 <node5>:      0x000001ca      0x00000005      0x00420110      0x00000000
0x4202c8 <node6>:      0x00000398      0x00000006      0x00420288      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

前景提要：我发现，[这题和 x86 一模一样](#)，所以照搬一些 x86 的内容：

## 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:
0x0000000004014e8 <+0>:    stp     x29, x30, [sp, #-32]!
0x0000000004014ec <+4>:    mov     x29, sp
0x0000000004014f0 <+8>:    str     x19, [sp, #16]
=> 0x0000000004014f4 <+12>:   bl      0x401974 <read_line>
0x0000000004014f8 <+16>:   mov     w2, #0xa                                // #10
0x0000000004014fc <+20>:   mov     x1, #0x0                                // #0
0x000000000401500 <+24>:   bl      0x400d00 <strtol@plt>
0x000000000401504 <+28>:   mov     x19, x0
0x000000000401508 <+32>:   sub     w0, w0, #0x1
0x00000000040150c <+36>:   cmp     w0, #0x3e8
0x000000000401510 <+40>:   b.hi    0x401548 <secret_phase+96> // b.pmore
0x000000000401514 <+44>:   mov     w1, w19
0x000000000401518 <+48>:   adrp    x0, 0x420000 <strlen@got.plt>
0x00000000040151c <+52>:   add     x0, x0, #0x110
0x000000000401520 <+56>:   add     x0, x0, #0x10
0x000000000401524 <+60>:   bl      0x401490 <fun7>
0x000000000401528 <+64>:   cbnz    w0, 0x401550 <secret_phase+104>
0x00000000040152c <+68>:   adrp    x0, 0x402000 <submitr+932>
0x000000000401530 <+72>:   add     x0, x0, #0x710
0x000000000401534 <+76>:   bl      0x400cd0 <puts@plt>
0x000000000401538 <+80>:   bl      0x401aa0 <phase_defused>
0x00000000040153c <+84>:   ldr     x19, [sp, #16]
0x000000000401540 <+88>:   ldp     x29, x30, [sp], #32
0x000000000401544 <+92>:   ret
0x000000000401548 <+96>:   bl      0x4018f8 <explode_bomb>
0x00000000040154c <+100>:  b       0x401514 <secret_phase+44>
0x000000000401550 <+104>:  bl      0x4018f8 <explode_bomb>
0x000000000401554 <+108>:  b       0x40152c <secret_phase+68>
```

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：

```
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

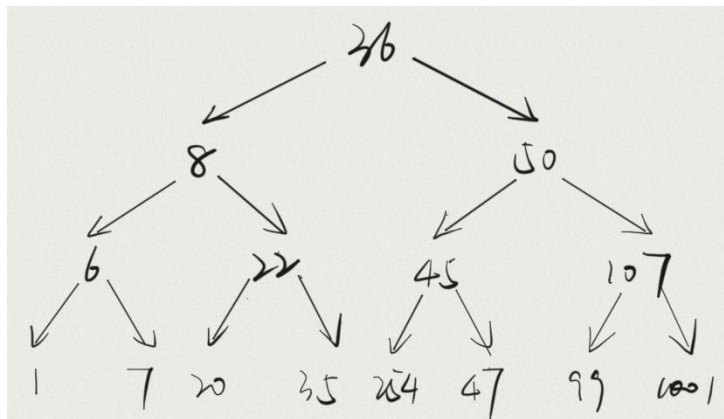
直接放图，这题我做了蛮久的：

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

Breakpoint 3, 0x0000000004012d8 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

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！(x86)

那他这是叫我干什么呢？接下来就是阅读代码了。

全部以16进制表示：

地址	值	+8	+10
604110	36	604130 <+20>	604150 <+40>
604130	8	604150 <+20>	604170 <+60>
604150	50	604190 <+40>	6041d0 <+80>
604170	22	604190 <+20>	6041d0 <+80>
604190	45	6041f0 <+20>	6042b0 <+120>
6041b0	6	6041d0 <+20>	604270 <+60>
6041d0	107	604230 <+80>	6042d0 <+100>
6041f0	40	0	0
604210	1	0	0
604230	99	0	0
604250	35	0	0
604270	7	0	0
604290	20	0	0
6042b0	47	0	0
6042d0	1001	0	0
6042f0	254	6308656	477

本题二叉树如下：

◦ n1	0x420120	0x24
+8		0x420138
+16		0x420150
◦ n21	0x420138	0x8
+8		0x420198
+16		0x420168
◦ n22	0x420150	0x420180
+8		0x420180
+16		0x420160
◦ n32	0x420168	0x16
+8		0x420240
+16		0x420210
◦ n33	0x420180	0x2d
+8		0x4201c8
+16		0x420258
◦ n31	0x420198	0x6
+8		0x4201e0
+16		0x420228
◦ n34	0x420160	0x6b
+8		0x4201f8
+16		0x420270
◦ n45	0x420108	0x28
◦ n41	0x4201e0	0x1
◦ n47	0x4201f8	0xb3
◦ n44	0x420210	0x23
◦ n42	0x420228	0x7
◦ n43	0x420240	0x14
◦ n46	0x420258	0x2f
◦ n48	0x420270	0x3e9

0x420120	<n1>:	0x00000024	0x00000000	0x00420138	0x00000000
0x420130	<n1+16>:	0x00420150	0x00000000	0x00000008	0x00000000
0x420140	<n21+8>:	0x00420198	0x00000000	0x00420168	0x00000000
0x420150	<n22>:	0x00000032	0x00000000	0x00420180	0x00000000
0x420160	<n22+16>:	0x004201b0	0x00000000	0x00000016	0x00000000
0x420170	<n32+8>:	0x00420240	0x00000000	0x00420210	0x00000000
0x420180	<n33>:	0x0000002d	0x00000000	0x004201c8	0x00000000
0x420190	<n33+16>:	0x00420258	0x00000000	0x00000006	0x00000000
0x4201a0	<n31+8>:	0x004201e0	0x00000000	0x00420228	0x00000000
0x4201b0	<n34>:	0x0000006b	0x00000000	0x004201f8	0x00000000
0x4201c0	<n34+16>:	0x00420270	0x00000000	0x00000028	0x00000000
0x4201d0	<n45+8>:	0x00000000	0x00000000	0x00000000	0x00000000
0x4201e0	<n41>:	0x00000001	0x00000000	0x00000000	0x00000000
0x4201f0	<n41+16>:	0x00000000	0x00000000	0x00000063	0x00000000
0x420200	<n47+8>:	0x00000000	0x00000000	0x00000000	0x00000000
0x420210	<n44>:	0x00000023	0x00000000	0x00000000	0x00000000
0x420220	<n44+16>:	0x00000000	0x00000000	0x00000007	0x00000000
0x420230	<n42+8>:	0x00000000	0x00000000	0x00000000	0x00000000
0x420240	<n43>:	0x00000014	0x00000000	0x00000000	0x00000000
0x420250	<n43+16>:	0x00000000	0x00000000	0x0000002f	0x00000000
0x420260	<n46+8>:	0x00000000	0x00000000	0x00000000	0x00000000
0x420270	<n48>:	0x000003e9	0x00000000	0x00000000	0x00000000
0x420280	<n48+16>:	0x00000000	0x00000000	0x000003a6	0x00000002



```

func7() {
    if (edi == 0) {
        eax = 0xffffffff;
        return;    ⇒ 找到最底层
    }
    edx = *(rdi);    ⇒ 找出当前地址所存值
    if (edx >= esi) {
        eax = 0;
        if (edx == esi)
            return;
        rdi = *(rdi + 0x10) ⇒ 找右子树
        func7();
        eax = 2 * eax + 1;
    }
    else {
        edi = *(edi + 0x8) ⇒ 找左子树
        func7();
        eax = 2 * eax;
    }
    return;
}

```

eax 最终为 7:

0 × 2 + 1 = 1	⇒ 全部找右子树
1 × 2 + 1 = 3	
3 × 2 + 1 = 7	

∴ 输入为 1001

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 0**，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 0：  
所以我要往左找左子树。

找的步骤：

1. 根据我画的二叉树，我要找到？；
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 **36861**。  
因为最左边一列答案全是 0！

## 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：**x0** 为**返回值**；

**x19 x21 x29** 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 variable`。

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: `x0` 是返回值;

由 `x` 变成 `w` 的寄存器在该次函数中可看作临时变量!

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 `x0/x1/x2`) 存的字符串, `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：彩蛋

```
(gdb) x /s $x0+24
0x4202e0 <user_password+8>: "p7KLE1NQd18L"
(gdb) x /s $x0+40
0x4202f0 <userid>: "2020211376"
```

我可能找到了我的真正密码，还找到了我的学号。

不仅如此，我还能看到一些"bupt1""bupt3"等的字符串。

而且一片区域里面存了所有的信息，有些很好玩的词汇……

## 七、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) 