哈爾濱工業大學

实验报告

实 验（三）

题　　目 Binary Bomb

二进制炸弹

专　　业 计算机类

学　　号 1190200122

班　　级 1903001

学　　生 袁野

指 导 教 师 郑贵滨

实 验 地 点 G709

实 验 日 期 2020-4-16

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

熟练掌握计算机系统的 ISA 指令系统与寻址方式
熟练掌握 Linux 下调试器的反汇编调试跟踪分析机器语言的方法
增强对程序机器级表示、汇编语言、调试器和逆向工程等的理解

1.2 实验环境与工具

1.2.1 硬件环境

Legion Y7000P 2019 PG0
CPU:Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (12 CPUs), ~2.6GHz
RAM: 16384MB

1.2.2 软件环境

Windows 10 家庭中文版 64-bit
Ubuntu 20.04.2 LTS
VMware® Workstation 16 Player 16.1.0 build-17198959

1.2.3 开发工具

Microsoft Visual Studio Community 2019 版本 16.9.2
Microsoft Visual 1.54.3
GCC 9.3.0

1.3 实验预习

■ **上实验课前，必须认真预习实验指导书（PPT 或 PDF）**

■ **了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。**

■ **请写出 C 语言下包含字符串比较、循环、分支（含 switch）、函数**

调用、递归、指针、结构、链表等的例子程序 **sample.c**。

- 生成执行程序 **sample.out**。
- 用 **gcc –S** 或 **CodeBlocks** 或 **GDB** 或 **OBJDUMP** 等，反汇编，比较。
- 列出每一部分的 **C** 语言对应的汇编语言。
- 修改编译选项**-O (**缺省 **2)**、**O0**、**O1**、**O2**、**O3**，**-m32/m64**。再次查看生成的汇编语言与原来的区别。
- 注意 **O1** 之后无栈帧，**EBP** 做别的用途。**-fno-omit-frame-pointer** 加上栈指针。
- **GDB** 命令详解 **–tui** 模式 **^XA** 切换 **layout** 改变等等
- 有目的地学习：看 **VS** 的功能 **GDB** 命令用什么？

第 2 章 实验环境建立

2.1 Ubuntu 下 CodeBlocks 反汇编（10 分）

CodeBlocks 运行 hellolinux.c。反汇编查看 printf 函数的实现。

要求：C、ASM、内存(显示 hello 等内容)、堆栈（call printf 前）、寄存器同时在一个窗口。

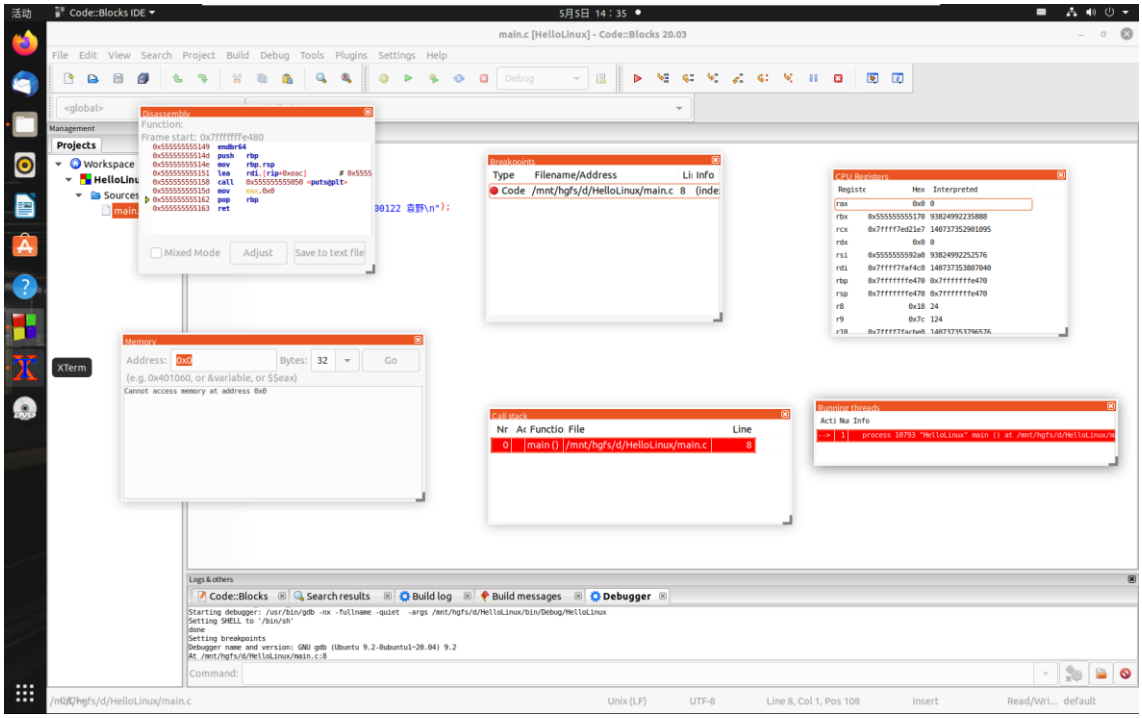

图 2-1 Ubuntu 下 CodeBlocks 反汇编截图

2.2 Ubuntu 下 EDB 运行环境建立（10 分）

用 EDB 调试 hellolinux.c 的执行文件，截图，要求同 2.1

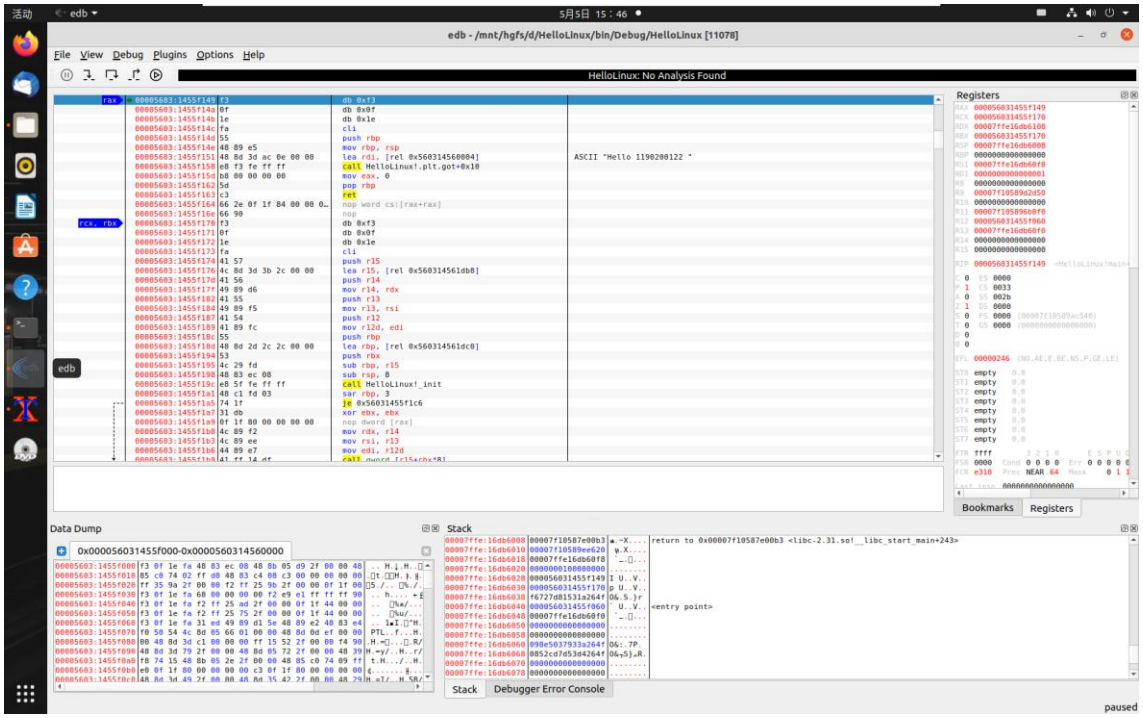

图 2-2　Ubuntu 下 EDB 截图

第 3 章 各阶段炸弹破解与分析

每阶段 15 分（密码 10 分，分析 5 分），总分不超过 80 分

3.1 阶段 1 的破解与分析

密码如下：I turned the moon into something I call a Death Star.

破解过程：

我们观察 phase_1 的反汇编代码

```
=> 0x00000000004013ec <+0>:     sub    $0x8,%rsp
   0x00000000004013f0 <+4>:     mov    $0x403150,%esi
   0x00000000004013f5 <+9>:     callq  0x40188f <strings_not_equal>
   0x00000000004013fa <+14>:    test   %eax,%eax
   0x00000000004013fc <+16>:    jne    0x401403 <phase_1+23>
   0x00000000004013fe <+18>:    add    $0x8,%rsp
   0x0000000000401402 <+22>:    retq
   0x0000000000401403 <+23>:    callq  0x401971 <explode_bomb>
   0x0000000000401408 <+28>:    jmp    0x4013fe <phase_1+18>
```

观察代码可知，在<phase_1+14>处%eax 为 1 时，程序会运行<explode_bomb>函数二引爆炸弹，因此我们需要使%eax 为 0,因此我们需要接着观察<string_not_equal>函数。

```
Dump of assembler code for function strings_not_equal:
   0x000000000040188f <+0>:     push   %r12
   0x0000000000401891 <+2>:     push   %rbp
   0x0000000000401892 <+3>:     push   %rbx
   0x0000000000401893 <+4>:     mov    %rdi,%rbx
   0x0000000000401896 <+7>:     mov    %rsi,%rbp
   0x0000000000401899 <+10>:    callq  0x40187b <string_length>
   0x000000000040189e <+15>:    mov    %eax,%r12d
   0x00000000004018a1 <+18>:    mov    %rbp,%rdi
   0x00000000004018a4 <+21>:    callq  0x40187b <string_length>
   0x00000000004018a9 <+26>:    cmp    %eax,%r12d
   0x00000000004018ac <+29>:    jne    0x4018cb <strings_not_equal+60>
   0x00000000004018ae <+31>:    movzbl (%rbx),%eax
   0x00000000004018b1 <+34>:    test   %al,%al
   0x00000000004018b3 <+36>:    je     0x4018c4 <strings_not_equal+53>
   0x00000000004018b5 <+38>:    cmp    %al,0x0(%rbp)
   0x00000000004018b8 <+41>:    jne    0x4018d5 <strings_not_equal+70>
   0x00000000004018ba <+43>:    add    $0x1,%rbx
   0x00000000004018be <+47>:    add    $0x1,%rbp
   0x00000000004018c2 <+51>:    jmp    0x4018ae <strings_not_equal+31>
   0x00000000004018c4 <+53>:    mov    $0x0,%eax
   0x00000000004018c9 <+58>:    jmp    0x4018d0 <strings_not_equal+65>
   0x00000000004018cb <+60>:    mov    $0x1,%eax
```

通过观察发现，该函数是比较%rdi 与%rsi 中的字符串是否相等，若相等返回 0，不等返回 1，因此我们需要知道%rdi 和%rai 中的字符串是什么。

<phase_1+4>处将地址 0x403150 赋给%rsi。

```
mp of assembler code for function main:
 0x00000000004012a6 <+0>:     push   %rbx
 0x00000000004012a7 <+1>:     cmp    $0x1,%edi
 0x00000000004012aa <+4>:     je     0x401398 <main+242>
 0x00000000004012b0 <+10>:    mov    %rsi,%rbx
 0x00000000004012b3 <+13>:    cmp    $0x2,%edi
 0x00000000004012b6 <+16>:    jne    0x4013cb <main+293>
 0x00000000004012bc <+22>:    mov    0x8(%rsi),%rdi
 0x00000000004012c0 <+26>:    mov    $0x403004,%esi
 0x00000000004012c5 <+31>:    callq  0x401130 <fopen@plt>
 0x00000000004012ca <+36>:    mov    %rax,0x449f(%rip)        # 0
>
 0x00000000004012d1 <+43>:    test   %rax,%rax
 0x00000000004012d4 <+46>:    je     0x4013ab <main+261>
 0x00000000004012da <+52>:    callq  0x4018dc <initialize_bomb>
 0x00000000004012df <+57>:    mov    $0x403088,%edi
 0x00000000004012e4 <+62>:    callq  0x401060 <puts@plt>
 0x00000000004012e9 <+67>:    mov    $0x4030c8,%edi
 0x00000000004012ee <+72>:    callq  0x401060 <puts@plt>
 0x00000000004012f3 <+77>:    callq  0x4019d4 <read_line>
 0x00000000004012f8 <+82>:    mov    %rax,%rdi
 0x00000000004012fb <+85>:    callq  0x4013ec <phase_1>
 0x0000000000401300 <+90>:    callq  0x401b02 <phase_defused>
```

main 函数的反汇编代码中<read_line>将用户输入的字符串的返回值转移给%rdi，因此我们只需要知道 0x403150 储存的字符串即可。

```
(gdb) x/s 0x403150
0x403150:       "I turned the moon into something I call a Death Star."
```

3.2 阶段 2 的破解与分析

密码如下：1 2 4 7 11 16

破解过程：

```
Dump of assembler code for function phase_2:
=> 0x000000000040140a <+0>:     push   %rbx
   0x000000000040140b <+1>:     sub    $0x20,%rsp
   0x000000000040140f <+5>:     mov    %rsp,%rsi
   0x0000000000401412 <+8>:     callq  0x401995 <read_six_numbers>
   0x0000000000401417 <+13>:    cmpl   $0x0,(%rsp)
   0x000000000040141b <+17>:    js     0x401424 <phase_2+26>
   0x000000000040141d <+19>:    mov    $0x1,%ebx
   0x0000000000401422 <+24>:    jmp    0x401433 <phase_2+41>
   0x0000000000401424 <+26>:    callq  0x401971 <explode_bomb>
   0x0000000000401429 <+31>:    jmp    0x40141d <phase_2+19>
   0x000000000040142b <+33>:    callq  0x401971 <explode_bomb>
   0x0000000000401430 <+38>:    add    $0x1,%ebx
   0x0000000000401433 <+41>:    cmp    $0x5,%ebx
   0x0000000000401436 <+44>:    jg     0x40144d <phase_2+67>
   0x0000000000401438 <+46>:    movslq %ebx,%rax
   0x000000000040143b <+49>:    lea    -0x1(%rbx),%edx
   0x000000000040143e <+52>:    movslq %edx,%rdx
   0x0000000000401441 <+55>:    mov    %ebx,%ecx
   0x0000000000401443 <+57>:    add    (%rsp,%rdx,4),%ecx
   0x0000000000401446 <+60>:    cmp    %ecx,(%rsp,%rax,4)
   0x0000000000401449 <+63>:    je     0x401430 <phase_2+38>
   0x000000000040144b <+65>:    jmp    0x40142b <phase_2+33>
   0x000000000040144d <+67>:    add    $0x20,%rsp
   0x0000000000401451 <+71>:    pop    %rbx
   0x0000000000401452 <+72>:    retq
```

观察 phase_2 的反汇编代码可知，该函数首先进入了<read_six_numbers>函数，观察子函数内容可知该函数读入六个整型变量，当读入数量不够时就会爆炸。此时我们回到 phase_2。

```
   0x0000000000401417 <+13>:    cmpl   $0x0,(%rsp)
   0x000000000040141b <+17>:    js     0x401424 <phase_2+26>
```

由这两句话得知，我们需要让栈指针指向的元素，也就是第一个输入的数字大于等于 0，否则爆炸。

```
0x000000000040141d <+19>:    mov    $0x1,%ebx
0x0000000000401422 <+24>:    jmp    0x401433 <phase_2+41>
0x0000000000401424 <+26>:    callq  0x401971 <explode_bomb>
0x0000000000401429 <+31>:    jmp    0x40141d <phase_2+19>
0x000000000040142b <+33>:    callq  0x401971 <explode_bomb>
0x0000000000401430 <+38>:    add    $0x1,%ebx
0x0000000000401433 <+41>:    cmp    $0x5,%ebx
0x0000000000401436 <+44>:    jg     0x40144d <phase_2+67>
0x0000000000401438 <+46>:    movslq %ebx,%rax
0x000000000040143b <+49>:    lea    -0x1(%rbx),%edx
0x000000000040143e <+52>:    movslq %edx,%rdx
0x0000000000401441 <+55>:    mov    %ebx,%ecx
0x0000000000401443 <+57>:    add    (%rsp,%rdx,4),%ecx
0x0000000000401446 <+60>:    cmp    %ecx,(%rsp,%rax,4)
0x0000000000401449 <+63>:    je     0x401430 <phase_2+38>
0x000000000040144b <+65>:    jmp    0x40142b <phase_2+33>
```

紧接着进入一个循环，<phase_2+19>为循环的初始化，<phase_2+41>和<phase_2+44>为跳出循环的判断。

```
0x0000000000401438 <+46>:    movslq %ebx,%rax
0x000000000040143b <+49>:    lea    -0x1(%rbx),%edx
0x000000000040143e <+52>:    movslq %edx,%rdx
0x0000000000401441 <+55>:    mov    %ebx,%ecx
0x0000000000401443 <+57>:    add    (%rsp,%rdx,4),%ecx
0x0000000000401446 <+60>:    cmp    %ecx,(%rsp,%rax,4)
0x0000000000401449 <+63>:    je     0x401430 <phase_2+38>
0x000000000040144b <+65>:    jmp    0x40142b <phase_2+33>
```

观察代码我们发现，%rdx 赋值为上一个数字的下标，%ecx 先赋值为当前数字的下标，然后加上通过%rdx 而寻得的上一个数字的值，而在<phase_2+60>处与%ecx 比较的正是当前数字，如果不相等则爆炸，因此正确的密码只需要保证第一个数字为非负数，剩下的数字与前一个数字的差值依次为 1，2，3，4，5 即可。

3.3 阶段 3 的破解与分析

密码如下：

0 n 279　或

1 l 893　或

2 t 813　或

3 z 745　或

4 g 483　或

5 l 119　或

6 d 510　或

7 c 154

破解过程：

```
Dump of assembler code for function phase_3:
   0x0000000000401453 <+0>:     sub    $0x18,%rsp
   0x0000000000401457 <+4>:     lea    0x8(%rsp),%r8
   0x000000000040145c <+9>:     lea    0x7(%rsp),%rcx
   0x0000000000401461 <+14>:    lea    0xc(%rsp),%rdx
   0x0000000000401466 <+19>:    mov    $0x4031ae,%esi
   0x000000000040146b <+24>:    mov    $0x0,%eax
   0x0000000000401470 <+29>:    callq  0x401110 <__isoc99_sscanf@plt>
   0x0000000000401475 <+34>:    cmp    $0x2,%eax
   0x0000000000401478 <+37>:    jle    0x401490 <phase_3+61>
   0x000000000040147a <+39>:    mov    0xc(%rsp),%eax
   0x000000000040147e <+43>:    cmp    $0x7,%eax
   0x0000000000401481 <+46>:    ja     0x401591 <phase_3+318>
```

在开始阶段是数据的输入，我们观察汇编代码发现栈中留出了两个四字节的空间和一个一字节的空间，紧接着我们可以输出 0x4031ae 地址的内容发现该字符串为“%d %c %d”，因此我们可以得知输入的为两个整数和一个字符。

```
End of assembler dump.
(gdb) x/s 0x4031ae
0x4031ae:       "%d %c %d"
```

而我们经过测试得知 0xc(%rsp)为第一个整数，0x8(%rsp)为第二个整数，0x7(%rsp)为第二个字符。

```
(gdb) x/x $rsp+0x8
0x7fffffffdde8: 0x02
(gdb) x/x $rsp+0xc
0x7fffffffddec: 0x01
(gdb) x/x $rsp+0x7
0x7fffffffdde7: 0x2d
```

、

若输入参数不够 3 则会爆炸。

紧接着将第一个整数与 7 进行比较，如果大于 7 也会爆炸，而且由指令 ja 可知该数字为无符号整数，因此第一个参数的范围是 0~7。

```
0x0000000000401489 <+54>:    jmpq   *0x4031c0(,%rax,8)
```

而这句话是一个由第一个参数的数值决定的跳转表，我们将其输出。

```
(gdb) x/32gx 0x4031c0
0x4031c0:       0x0000000000401497      0x00000000004014ba
0x4031d0:       0x00000000004014dd      0x0000000000401500
0x4031e0:       0x0000000000401520      0x000000000040153d
0x4031f0:       0x0000000000401557      0x0000000000401574
```

我们会发现内容为八个代码段的地址。正是如下地址

这些地址根据输入的第一个整数的不同来决定跳转位置的不同，这些部分的结构基本一致，我们观察其中的一个部分即可。

```
0x0000000000401497 <+68>:    cmpl   $0x117,0x8(%rsp)
0x000000000040149f <+76>:    jne    0x4014ab <phase_3+88>
0x00000000004014a1 <+78>:    mov    $0x6e,%eax
0x00000000004014a6 <+83>:    jmpq   0x40159b <phase_3+328>
0x00000000004014ab <+88>:    callq  0x401971 <explode_bomb>
0x00000000004014b0 <+93>:    mov    $0x6e,%eax
0x00000000004014b5 <+98>:    jmpq   0x40159b <phase_3+328>
```

上述代码为当第一个整数为 0 时的部分代码，观察代码会发现，首先将第三个输入的整数与给定的数字 279 进行比较，如果不相等爆炸，因此此时输入的第三个整数应为 279。

随后将地址 0x6a 赋值给%eax 并跳转到如下部分

```
0x000000000040159b <+328>:   cmp    %al,0x7(%rsp)
0x000000000040159f <+332>:   jne    0x4015a6 <phase_3+339>
```

将%al 指向的字符输出

```
(gdb) p $al
$2 = 110
```

可知只有当我们输入的第二个字符的 ASCII 为 110，即字符‘n’时程序才能顺利结束。

因此我们可以得到一组密码为“0 n 279”。同理我们可以根据第一个整数的不同情况推得其他七组密码。

3.4 阶段 4 的破解与分析

密码如下：

8 1 或

9 1 或

11 1

破解过程：

```
Dump of assembler code for function phase_4:
=> 0x00000000004015ea <+0>:     sub    $0x18,%rsp
   0x00000000004015ee <+4>:     lea    0x8(%rsp),%rcx
   0x00000000004015f3 <+9>:     lea    0xc(%rsp),%rdx
   0x00000000004015f8 <+14>:    mov    $0x40334f,%esi
   0x00000000004015fd <+19>:    mov    $0x0,%eax
   0x0000000000401602 <+24>:    callq  0x401110 <__isoc99_sscanf@p
   0x0000000000401607 <+29>:    cmp    $0x2,%eax
   0x000000000040160a <+32>:    jne    0x401619 <phase_4+47>
   0x000000000040160c <+34>:    mov    0xc(%rsp),%eax
   0x0000000000401610 <+38>:    test   %eax,%eax
   0x0000000000401612 <+40>:    js     0x401619 <phase_4+47>
   0x0000000000401614 <+42>:    cmp    $0xe,%eax
   0x0000000000401617 <+45>:    jle    0x40161e <phase_4+52>
   0x0000000000401619 <+47>:    callq  0x401971 <explode_bomb>
   0x000000000040161e <+52>:    mov    $0xe,%edx
   0x0000000000401623 <+57>:    mov    $0x0,%esi
   0x0000000000401628 <+62>:    mov    0xc(%rsp),%edi
   0x000000000040162c <+66>:    callq  0x4015ad <func4>
   0x0000000000401631 <+71>:    cmp    $0x1,%eax
   0x0000000000401634 <+74>:    jne    0x40163d <phase_4+83>
   0x0000000000401636 <+76>:    cmpl   $0x1,0x8(%rsp)
   0x000000000040163b <+81>:    je     0x401642 <phase_4+88>
   0x000000000040163d <+83>:    callq  0x401971 <explode_bomb>
   0x0000000000401642 <+88>:    add    $0x18,%rsp
   0x0000000000401646 <+92>:    retq
```

我们观察 phase_4 的反汇编代码，首先将%esi 的字符串输出

```
(gdb) x/s 0x40334f
0x40334f:       "%d %d"
```

由此我们得知，我们需要输入两个整数

```
0x000000000040160c <+34>:    mov    0xc(%rsp),%eax
0x0000000000401610 <+38>:    test   %eax,%eax
0x0000000000401612 <+40>:    js     0x401619 <phase_4+47>
0x0000000000401614 <+42>:    cmp    $0xe,%eax
0x0000000000401617 <+45>:    jle    0x40161e <phase_4+52>
```

而由这句话我们可知，输入的第一个参数的范围为 0~14，此后就是 func4 函数，我们将其反编译

```
Dump of assembler code for function func4:
   0x00000000004015ad <+0>:     sub    $0x8,%rsp
   0x00000000004015b1 <+4>:     mov    %edx,%ecx
   0x00000000004015b3 <+6>:     sub    %esi,%ecx
   0x00000000004015b5 <+8>:     mov    %ecx,%eax
   0x00000000004015b7 <+10>:    shr    $0x1f,%eax
   0x00000000004015ba <+13>:    add    %ecx,%eax
   0x00000000004015bc <+15>:    sar    %eax
   0x00000000004015be <+17>:    add    %esi,%eax
   0x00000000004015c0 <+19>:    cmp    %edi,%eax
   0x00000000004015c2 <+21>:    jg     0x4015d0 <func4+35>
   0x00000000004015c4 <+23>:    jl     0x4015dc <func4+47>
   0x00000000004015c6 <+25>:    mov    $0x0,%eax
   0x00000000004015cb <+30>:    add    $0x8,%rsp
   0x00000000004015cf <+34>:    retq
   0x00000000004015d0 <+35>:    lea    -0x1(%rax),%edx
   0x00000000004015d3 <+38>:    callq  0x4015ad <func4>
   0x00000000004015d8 <+43>:    add    %eax,%eax
   0x00000000004015da <+45>:    jmp    0x4015cb <func4+30>
   0x00000000004015dc <+47>:    lea    0x1(%rax),%esi
   0x00000000004015df <+50>:    callq  0x4015ad <func4>
   0x00000000004015e4 <+55>:    lea    0x1(%rax,%rax,1),%eax
   0x00000000004015e8 <+59>:    jmp    0x4015cb <func4+30>
```

通过观察我们发现，该 func4 函数是一个递归函数，翻译为 C 代码后如下

```c
// a -> %esi
// b -> %edx
// c -> %edi

int fun4(int a,int b,int c){
    int x = (b-a)/2+a;
    if (x > c) return fun4(a,x-1,c)*2;
    else if (x < c) return fun4(x+1,b,c)*2+1;
    else return 0;
}
```

```
0x000000000040162c <+66>:    callq  0x4015ad <func4>
0x0000000000401631 <+71>:    cmp    $0x1,%eax
```

而由上述代码可知我们需要将函数的返回值为 1，我们通过代码

```c
int main(){
    for (int i=0; i<=14; ++i)
        if (fun4(0,14,i) == 1) printf("%d ",i);
    printf("\n");
}
```

运行结果为

```
yuanye@1190200122-yuanye:/mnt/hgfs/d$ ./a.out
8 9 11
```

因此第一个数字为他们三个之一。

```
0x0000000000401636 <+76>:    cmpl   $0x1,0x8(%rsp)
```

有上述代码可知，第二个整数为 1。

因此密码得出。

3.5 阶段 5 的破解与分析

密码如下：

破解过程：

3.6 阶段 6 的破解与分析

密码如下：

破解过程：

3.7 阶段 7 的破解与分析(隐藏阶段)

密码如下：

破解过程：

第 4 章 总结

4.1 请总结本次实验的收获

本次实验学会了 codeblocks 的反汇编调试方法，学会了 edb 的调试方法，更加熟练运用了 gdb 调试，学会了分析反汇编代码来寻找其中储存的某些关键信息进而破解密码，对反汇编语句运用更加熟练，明白了汇编语言中各个函数之间的调用关系，对堆栈有了更深的理解，学会了对教材知识熟练运用

4.2 请给出对本次实验内容的建议

不同环境下的汇编代码会有所差异，希望可以比较一下。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.