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 计算机科学与技术

学　　号 1171000410

班　　级 1703005

学　　生 强文杰

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2018.10.21

计算机科学与技术学院

目 录

第 1 章 实验基本信息 - 3 -

1.1 实验目的 - 3 -
1.2 实验环境与工具 - 3 -
1.2.1 硬件环境 - 3 -
1.2.2 软件环境 - 3 -
1.2.3 开发工具 - 3 -
1.3 实验预习 - 3 -

第 2 章 实验环境建立 - 5 -

2.1 UBUNTU 下 CODEBLOCKS 反汇编（10 分） - 5 -
2.2 UBUNTU 下 EDB 运行环境建立（10 分） - 5 -

第 3 章 各阶段炸弹破解与分析 - 7 -

3.1 阶段 1 的破解与分析 - 7 -
3.2 阶段 2 的破解与分析 - 7 -
3.3 阶段 3 的破解与分析 - 9 -
3.4 阶段 4 的破解与分析 - 11 -
3.5 阶段 5 的破解与分析 - 14 -
3.6 阶段 6 的破解与分析 - 15 -
3.7 阶段 7 的破解与分析(隐藏阶段) - 16 -

第 4 章 总结 - 20 -

4.1 请总结本次实验的收获 - 20 -
4.2 请给出对本次实验内容的建议 - 20 -

参考文献 - 21 -

第 1 章 实验基本信息

1.1 实验目的

1．熟练掌握计算机系统的 ISA 指令系统与寻址方式
2．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
3．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

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

GDB/OBJDUMP；EDB；KDD 等

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。

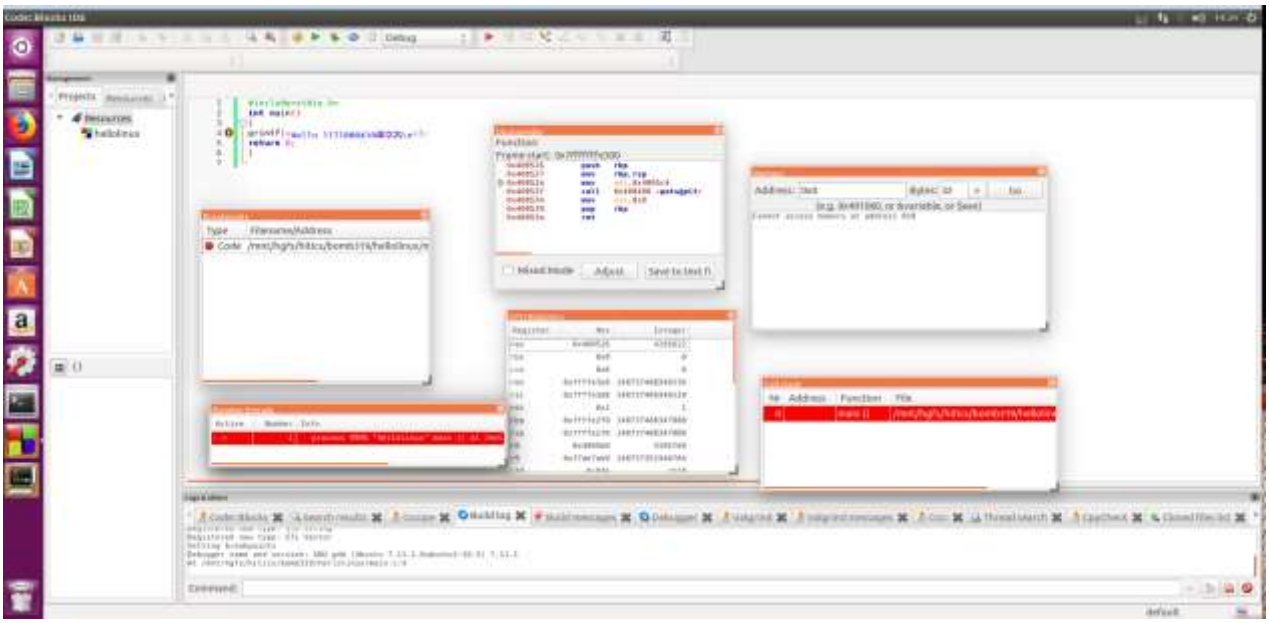


图 2-1　Ubuntu 下 CodeBlocks 反汇编截图

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

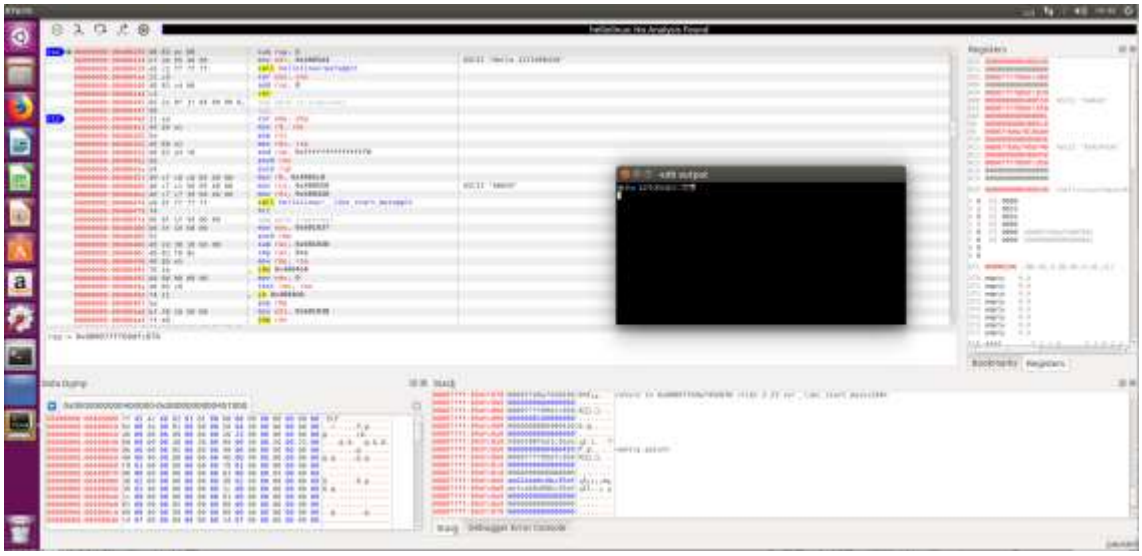


图 2-2　Ubuntu 下 EDB 截图

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：Why make trillions when we could make... billions?

```
qwj@qwj-virtual-machine:~/hitics/bomb319$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Why make trillions when we could make... billions?
Phase 1 defused. How about the next one?
```

破解过程：strings_not_equal 函数是判断输入的字符串和%rsi 中的字符串是否相等，如果相等则令%eax 为 0，否则为 1。

并且后面的 test %eax,%eax 和 jne 0x55555555518d 则是判断若%eax 不为 0，就执行 explode_bomb。因此需要输入的密码必须与%rsi 中的字符串相等。

```
(gdb) disassemble phase_1
Dump of assembler code for function phase_1:
   0x0000555555555174 <+0>:     sub    $0x8,%rsp
   0x0000555555555178 <+4>:     lea    0x14f1(%rip),%rsi        # 0x555555556670
   0x000055555555517f <+11>:    callq  0x5555555555cf <strings_not_equal>
   0x0000555555555184 <+16>:    test   %eax,%eax
   0x0000555555555186 <+18>:    jne    0x55555555518d <phase_1+25>
   0x0000555555555188 <+20>:    add    $0x8,%rsp
   0x000055555555518c <+24>:    retq
   0x000055555555518d <+25>:    callq  0x5555555556db <explode_bomb>
   0x0000555555555192 <+30>:    jmp    0x555555555188 <phase_1+20>
End of assembler dump.
(gdb) x/s 0x555555556670
0x555555556670: "Why make trillions when we could make... billions?"
```

gdb 通过 x/s 0x555555556670 查看其中的数据，即得密码。

3.2 阶段 2 的破解与分析

密码如下：其中一组密码为：0 1 3 6 10 15

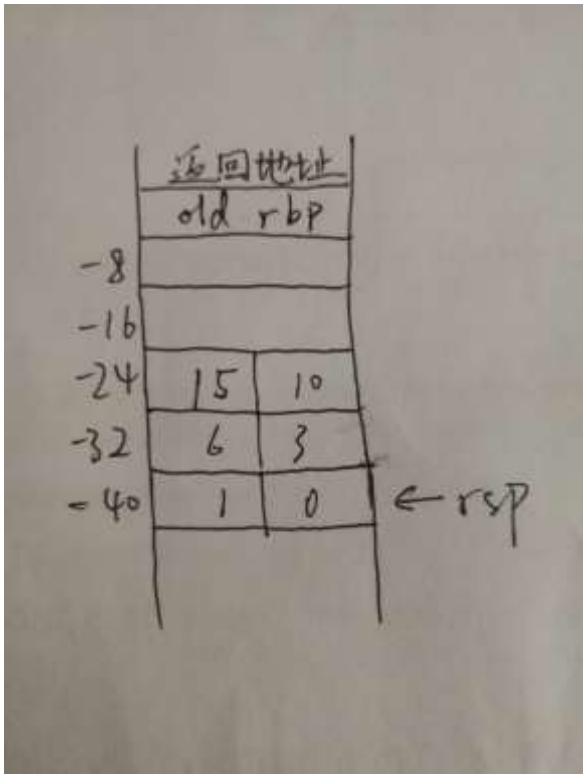
```
qwj@qwj-virtual-machine:~/hitics/bomb319$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Why make trillions when we could make... billions?
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2.  Keep going!
```

破解过程：第一步：phase_2 的代码需要输入 6 个数字，并且当我进入 read_six_numbers 函数，通过查看%rsi 时，发现其为%d %d %d %d %d %d，由此可知，函数读入的数据是 6 个 int 类型，这对栈的分析至关重要。

第二步：查看函数 phase_2。通过 cmpl $0x0,(%rsp)和后一句 js 条件跳转语句可以分析第一个数大于等于 0。暂且输入第一个数为 0。%ebx 的值由 0x1 执行循环，每次加 0x1，当%ebx 为 0x6 时退出循环。每次循环，将%ebx 赋值给%eax。分析主要的语句可以得出每次循环内，%eax+（4%rbx-4+%rbp）与（4%rbx+%rbp）的值相等，而（4%rbx-4+%rbp）取址后为上一个输入的值，（4%rbx+%rbp）为本次输入的值。由此不难得出函数的循环大致为：

```
int a[6];
a[0]=0;
for (i=1;i<6;i++)
{   a[i]=a[i-1]+I ;   }
```

附上此时栈的示意图：



3.3 阶段 3 的破解与分析

密码如下：其中一组密码为：2 44

```
2 44
Halfway there!
```

破解过程：

```
(gdb) x/s 0x55555555684f
0x55555555684f: "%d %d"
```

通过查看%rsi 中的字符，可知输入的是两个 int 型数据。sscanf 返回值为读入有效数据的个数，并且由$0x7,0xc(%rsp)可知，第一个数小于 7。

于是先输入 2 2，使用 gdb 观察下面的执行。

观察到 jmpq　*%rax 语句，表示以%rax 中的值作为跳转目标，此时查看%rax 中十六进制的值如下：

```
(gdb) x/x $rax
0x55555555526d <phase_3+144>:    0x000000b8
```

因此 jmp　*rax 即表示跳转到 0x55555555526d，再单步执行程序。

```
e0
   0x0000555555555217 <+58>:    movslq (%rdx,%rax,4),%rax
   0x000055555555521b <+62>:    add    %rdx,%rax
=> 0x000055555555521e <+65>:    jmpq   *%rax
   0x0000555555555220 <+67>:    callq  0x5555555556db <explode_bomb>
   0x0000555555555225 <+72>:    jmp    0x555555555201 <phase_3+36>
   0x0000555555555227 <+74>:    mov    $0x21a,%eax
   0x000055555555522c <+79>:    jmp    0x555555555233 <phase_3+86>
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) ni
```

```
   0x0000555555555238 <+91>:    add    $0x246,%eax
   0x000055555555523d <+96>:    sub    $0x21a,%eax
   0x0000555555555242 <+101>:   add    $0x21a,%eax
   0x0000555555555247 <+106>:   sub    $0x21a,%eax
   0x000055555555524c <+111>:   add    $0x21a,%eax
   0x0000555555555251 <+116>:   sub    $0x21a,%eax
   0x0000555555555256 <+121>:   cmpl   $0x5,0xc(%rsp)
   0x000055555555525b <+126>:   jg     0x555555555263 <phase_3+134>
   0x000055555555525d <+128>:   cmp    %eax,0x8(%rsp)
   0x0000555555555261 <+132>:   je     0x555555555268 <phase_3+139>
   0x0000555555555263 <+134>:   callq  0x5555555556db <explode_bomb>
   0x0000555555555268 <+139>:   add    $0x18,%rsp
   0x000055555555526c <+143>:   retq
=> 0x000055555555526d <+144>:   mov    $0x0,%eax
   0x0000555555555272 <+149>:   jmp    0x555555555238 <phase_3+91>
   0x0000555555555274 <+151>:   mov    $0x0,%eax
   0x0000555555555279 <+156>:   jmp    0x55555555523d <phase_3+96>
   0x000055555555527b <+158>:   mov    $0x0,%eax
```

跳转到 0x55555555526d 后，mov　$0x0,%eax　表示将%rax 赋值为 0x0 后再跳转到 0x555555555238 进行计算。

```
0x0000555555555238 <+91>:    add    $0x246,%eax
0x000055555555523d <+96>:    sub    $0x21a,%eax
0x0000555555555242 <+101>:   add    $0x21a,%eax
0x0000555555555247 <+106>:   sub    $0x21a,%eax
0x000055555555524c <+111>:   add    $0x21a,%eax
0x0000555555555251 <+116>:   sub    $0x21a,%eax
0x0000555555555256 <+121>:   cmpl   $0x5,0xc(%rsp)
0x000055555555525b <+126>:   jg     0x555555555263 <phase_3+134>
0x000055555555525d <+128>:   cmp    %eax,0x8(%rsp)
0x0000555555555261 <+132>:   je     0x555555555268 <phase_3+139>
0x0000555555555263 <+134>:   callq  0x5555555556db <explode_bomb>
```

由 cmpl　$0x5 , 0xc(%rsp)和 jg　0x555555555263 可知第一个值为小于 5 的无符号数，而第二个数值等于计算后%eax 中的值。

计算过程是从地址 0x555555555238 开始：

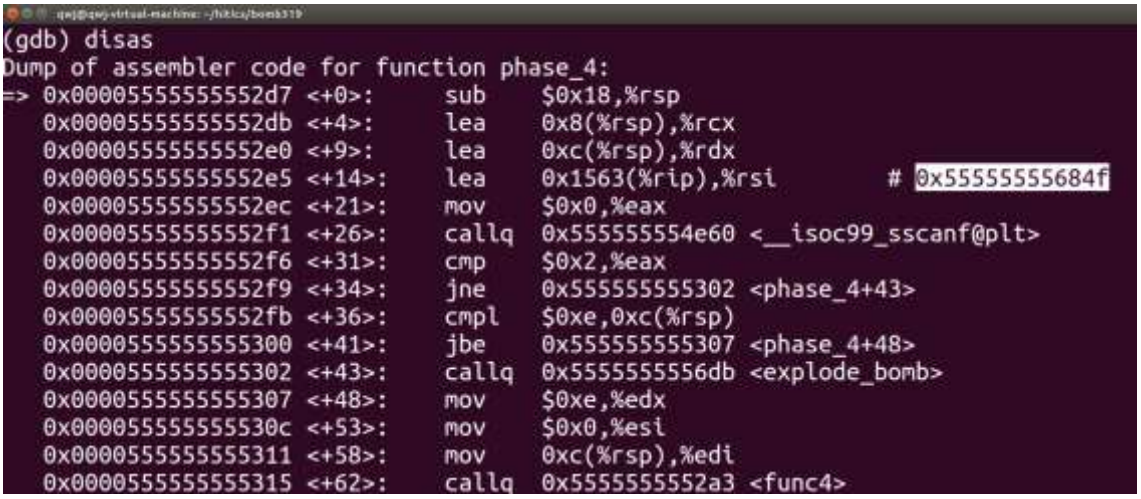
0+0x246-0x21a+0x21a-0x21a+0x21a -0x21a =44，因此第一个数输入 2 时，第二个数为 44.

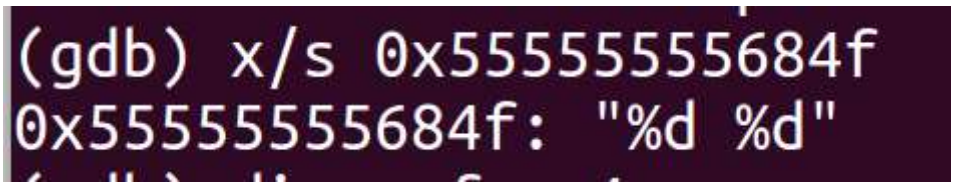
综上，此题其实是一条 switch 语句，根据输入小于 5 的第一个数值，可计算出 *%rax，即跳转的地址，然后计算求出第二个数值，即得通关密码。

3.4 阶段 4 的破解与分析

密码如下：8 35

破解过程：





第一步，先观察调用 func4 之前的 phase_4，通过查看%rsi，和调用 __isoc99_sscanf 时返回值为 2 可以知道，我们输入的是两个 int 型数据，并且由 cmpl　$0xe,0xc(%rsp) 和 jbe　0x555555555307 可以知道输入的第一个数小于 14，进入 func4 之前%edx 被赋值为 14，%esi 被赋值为 0，%edi 被赋值为我们输入的第一个数。

```
   0x0000555555555315 <+62>:    callq  0x5555555552a3 <func4>
   0x000055555555531a <+67>:    cmp    $0x23,%eax
   0x000055555555531d <+70>:    jne    0x555555555326 <phase_4+79>
   0x000055555555531f <+72>:    cmpl   $0x23,0x8(%rsp)
   0x0000555555555324 <+77>:    je     0x55555555532b <phase_4+84>
   0x0000555555555326 <+79>:    callq  0x5555555556db <explode_bomb>
   0x000055555555532b <+84>:    add    $0x18,%rsp
```

第二步，观察 func4 调用之后的 phase_4，可以知道输入的第二个数是 35，并且 fun4 的返回值%eax 也是 35。

```
   0x00000000000012a3 <+0>:     push   %rbx
   0x00000000000012a4 <+1>:     mov    %edx,%eax
   0x00000000000012a6 <+3>:     sub    %esi,%eax
   0x00000000000012a8 <+5>:     mov    %eax,%ebx
   0x00000000000012aa <+7>:     shr    $0x1f,%ebx
   0x00000000000012ad <+10>:    add    %eax,%ebx
   0x00000000000012af <+12>:    sar    %ebx
   0x00000000000012b1 <+14>:    add    %esi,%ebx
   0x00000000000012b3 <+16>:    cmp    %edi,%ebx
   0x00000000000012b5 <+18>:    jg     0x12bf <func4+28>
   0x00000000000012b7 <+20>:    cmp    %edi,%ebx
   0x00000000000012b9 <+22>:    jl     0x12cb <func4+40>
   0x00000000000012bb <+24>:    mov    %ebx,%eax
   0x00000000000012bd <+26>:    pop    %rbx
   0x00000000000012be <+27>:    retq
   0x00000000000012bf <+28>:    lea    -0x1(%rbx),%edx
   0x00000000000012c2 <+31>:    callq  0x12a3 <func4>
   0x00000000000012c7 <+36>:    add    %eax,%ebx
   0x00000000000012c9 <+38>:    jmp    0x12bb <func4+24>
   0x00000000000012cb <+40>:    lea    0x1(%rbx),%esi
   0x00000000000012ce <+43>:    callq  0x12a3 <func4>
   0x00000000000012d3 <+48>:    add    %eax,%ebx
---Type <return> to continue, or q <return> to quit---r
   0x00000000000012d5 <+50>:    jmp    0x12bb <func4+24>
```

第三步，进入 func4 查看。

通过对上语句的分析，我将 func 简化为如下形式：

%edx 初值为 14，%esi 初值为 0,

x=%rbx, y=%eax, z=%edx, a=%esi, a1=%edi

<fun4>

y=z ;

x=(y+a)/2;

1 .if (x>a1)

z=x-1

call <func4>

y+=x;

2 .if (x<a1)

a =x+1;

call<fun4>

y+=x;

3 .if (x=a1)

y=x;

由以上简化的调用形式可知，递归的最里层是 x=a1，由于每次调用<fun4>之前都要 pop %rbx，因此递归返回计算时 x 的值是不断变化的。

返回值%rax，即设置的 y，有 y= a1 +x1 +x2 +x3…

进行多次计算如下：

%edi = 0 , %rax = 11

%edi = 1 , %rax = 11

%edi = 2, %rax = 13

%edi = 3 , %rax = 10

%edi = 4 , %rax = 19

%edi = 5 , %rax = 15

%edi = 6 , %rax = 21

%edi = 7 , %rax = 7

%edi = 8 , %rax = 35

%edi = 9 , %rax = 27

%edi = 10 , %rax = 37

%edi = 11 , %rax = 18

%edi = 12 , %rax = 43

%edi = 13 , %rax = 31

%edi = 14 , %rax = 45

由返回值%eax 为 35 可以知道输入第一个数为 8。综上，输入 8　35.

3.5 阶段 5 的破解与分析

密码如下：其中一组为 444440

```
qwj@qwj-virtual-machine:~/hitics/bomb319$ ./bomb ans.txt
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!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
```

破解过程：

```
0x0000000000001330 <+0>:     push   %rbx
0x0000000000001331 <+1>:     mov    %rdi,%rbx
0x0000000000001334 <+4>:     callq  0x15b2 <string_length>
0x0000000000001339 <+9>:     cmp    $0x6,%eax
0x000000000000133c <+12>:    jne    0x136f <phase_5+63>
```

首先通过 string_length 函数，及 cmp　　$0x6,%eax 语句，可知密码为长度为 6 的字符串。

再由 movzbl (%rax),%edx 及 and　　$0xf,%edx 可知，将输入字符 0 拓展为 32 为赋值给%edx 再取%edx 的后四位。

另外，%rax 每次加一，cmp　　%rdi,%rax 条件判断语句控制循环。

其中最重要的是 add　　(%rsi,%rdx,4), %ecx,这条语句是指将（%rsi+4%rdx）该地址对应的值赋给%ecx，于是 gdb 调试出当%rdx 分别为 0，1，2，3，4，时，（%rsi+4%rdx）地址对应的值：

```
(gdb) x/d Quit
(gdb) x/d 0x555555556700
0x555555556700 <array.3415>:    2
(gdb) x/d 0x555555556704
0x555555556704 <array.3415+4>:  10
(gdb) x/d 0x555555556708
0x555555556708 <array.3415+8>:  6
(gdb) x/d 0x55555555670c
0x55555555670c <array.3415+12>: 1
(gdb) x/d 0x555555556710
0x555555556710 <array.3415+16>: 12
(gdb) q
A debugging session is active.
```

由于输出的%ecx 的值为 62，又根据累加出%ecx 的一种计算可以为 62=12+12+12+12+12+2，故逆推输入的字符串相对应为 444440 。

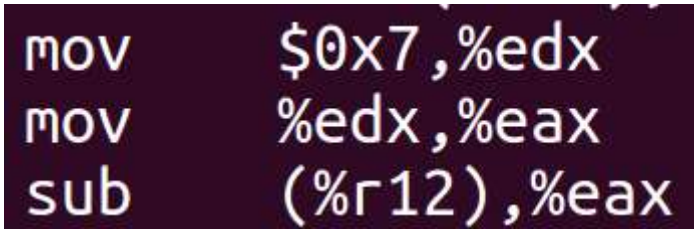
3.6 阶段 6 的破解与分析

密码如下：1 6 4 5 2 3

```
qwj@qwj-virtual-machine:~/hitics/bomb319$ ./bomb ans.txt
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!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Congratulations! You've defused the bomb!
```

破解过程：第一步：分析前面的的循环可知，read_six_numbers 的六个参数，全部不大于 6，并且互不相等。

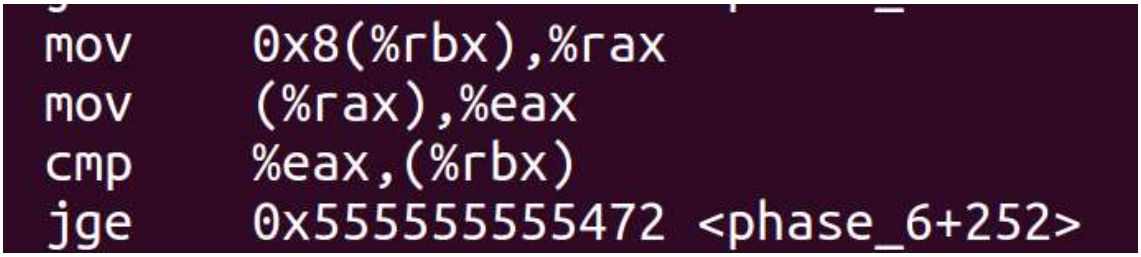
第二步：



分析后一个循环可知，输入的 6 个数，分别被 7 减，并且保存在原处。

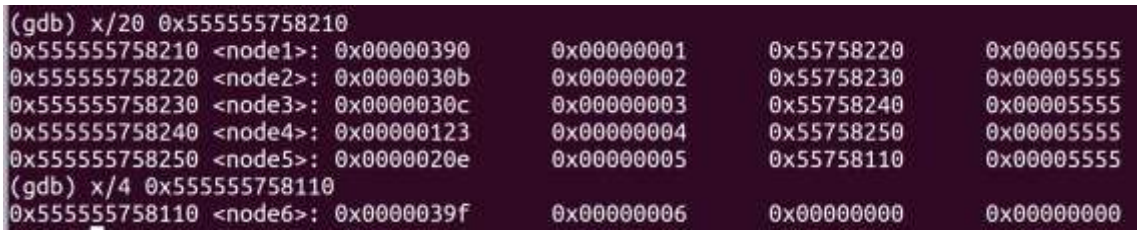
第三步：根据 7 减去输入的六个数字的值，通过循环，%rsp+8n 中分别存入不同的地址值。存值的方式是循环时不断对新的%rdx 加 0x8，其中%rdx 初始值为 0x555555758210

第四步：



由此可知，链表值由大到小排列。

第五步：



链表值由大到小排列为：0x39f 0x390 0x30c 0x30b 0x20e 0x123

对应的值为：6 1 3 2 5 4

又因为对应的值是 7 减去输入数后的值。

故输入的数对应为：1 6 4 5 2 3

3.7 阶段 7 的破解与分析(隐藏阶段)

密码如下：1001

```
qwj@qwj-virtual-machine:~/hitics/bomb319$ ./bomb ans.txt
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!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

破解过程：根据 phase_defused 的分析，可以知道必须解决完前六关才能进入隐藏阶段。

```
(gdb) x/s 0x2899
0x2899: "%d %d %s"
```

```
(gdb) x/s 0x28a2
0x28a2: "DrEvil"
```

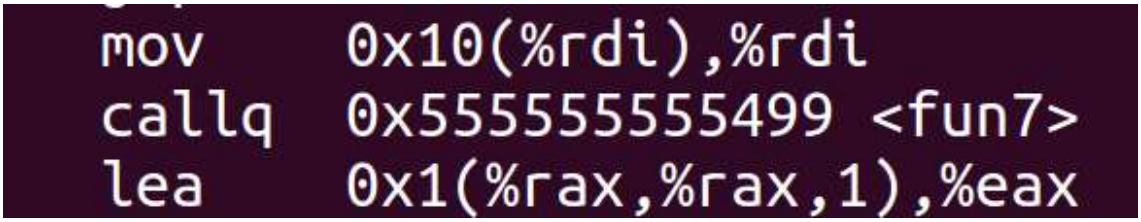
由以上截图可知，隐藏阶段是在输入两个 int 型数据后，再输入 DrEvil，才可进入。

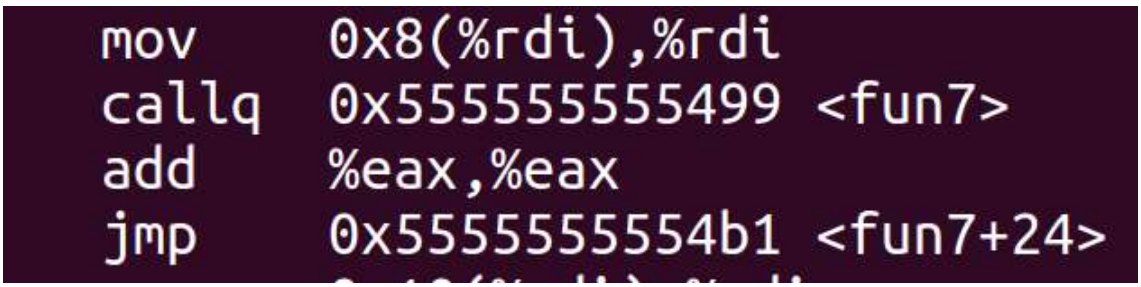
首先分析 secret_phase,由 read_line 可知，输入的是一个字符串，并且 strtol 函数是将一个字符串转化为十进制长整数赋给%rax 作为返回值，调用 func7 之前，%rdi 被赋值为 36，即第一个参数 a1=0x24，a2 为要输入的数。再由 func7 之后，可以知道返回值是 0x7.

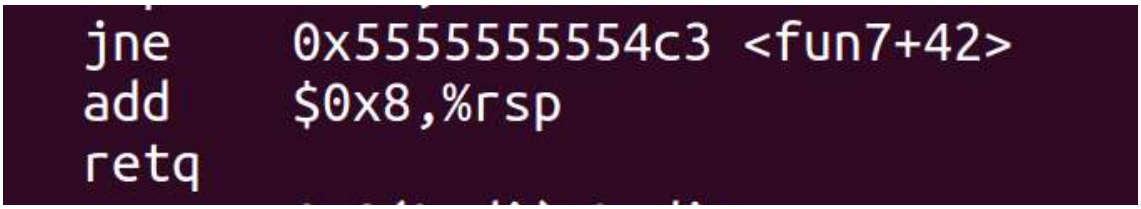
```
(gdb) x/x 0x555555758130
0x555555758130 <n1>:    0x00000024
```

```
  cmp    $0x7,%eax
  je     0x555555555515 <secret_phase+61>
```

观察 func7 内的语句，核心部分是递归。







由以上三张截图可知，

若*a1>a2 , a1=*(a1+8) , call func7 , %eax=%eax*2;

若*a1<a2 , a1=*(a1+16) , call fun7 , %eax=%eax*2+1;

若*a1=a2 , 跳出。

由此可知，最深层的%eax=0，并且如果*a1=a2，则推出最里面的递归条件。

因为 func7 执行完后返回值是 7，而逆推出 7 的产生过程为：

7=（（0*2+1）*2+1）*2+1

则递归时 a2=*（*（*（a1+0x10）+0x10）+0x10）

```
(gdb) p/x *(0x555555758130+0x10)
$2 = 0x55758170
(gdb) p/x *(0x555555758170+0x10)
$3 = 0x557581f0
(gdb) p/x *(0x5555557581f0+0x10)
$4 = 0x557580f0
(gdb) x/x 0x5555557580f0
0x5555557580f0 <n48>:   0x000003e9
(gdb) x/x 0x5555557581f0
0x5555557581f0 <n34>:   0x0000006b
(gdb) x/x 0x555555758170
0x555555758170 <n22>:   0x00000032
```

再用 gdb 调试，可知 a2=0x3e9，即 1001。

综上可知，输入的密码是 1001.

第 4 章 总结

4.1 请总结本次实验的收获

1．学会了 gdb 的调节和各种命令。
2．对 C 语言下字符串比较、循环、分支（含 switch）、函数调用、递归、指针、结构、链表等有了更深刻的理解。
3.更加深刻地理解了汇编语言，程序机器级表示以及逆向工程。

4.2 请给出对本次实验内容的建议

希望对 gdb 调试的讲解更加细致一些。

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.