CSAPP Bomblab 详解

该实验含有六个phase，通过gdb调试分析程序获得六个phase的答案即可完成

x86汇编指令

Register

保存返回值的寄存器：%rax=%rax[63:0],%eax=%rax[31:0],%ax=rax[15:0],%al=%rax[7:0]

栈指针寄存器：%rsp=%rsp[63:0],%esp=%rsp[31:0],%sp=%rsp[15:0],%spl=%rsp[7:0]

被调用者保存的寄存器：

- %rbx=%rbx[63:0],%ebx=%rbx[31:0],%bx=%rbx[15:0],%bl=%rbx[7:0]
- %rbp=%rbp[63:0],%ebp=%rbp[31:0],%bp=%rbx[15:0],%bpl=%rbp[7:0]
- %r12=%r12[63:0],%r12d=%r12[31:0],%r12w=%r12[15:0],%r12b=%r12[7:0]
- %r13=%r13[63:0],%r13d=%r13[31:0],%r13w=%r13[15:0],%r13b=%r13[7:0]
- %r14=%r14[63:0],%r14d=%r14[31:0],%r14w=%r14[15:0],%r14b=%r14[7:0]
- %r15=%r15[63:0],%r15d=%r15[31:0],%r15w=%r15[15:0],%r15b=%r15[7:0]

函数参数寄存器：(按参数顺序列举)

- %rdi=%rdi[63:0],%edi=%rdi[31:0],%di=%rdi[15:0],%dil=%rdi[7:0]
- %rsi=%rsi[63:0],%rsi=%rsi[31:0],%si=%rsi[15:0],%sil=%rsi[7:0]
- %rdx=%rdx[63:0],%edx=%rdx[31:0],%dx=%rdx[15:0],%dl=%rdx[7:0]
- %rcx=%rcx[63:0],%ecx=%rcx[31:0],%cx=%rcx[15:0],%cl=%rcx[7:0]
- %r8=%r8[63:0],%r8d=%r8[31:0],%r8w=%r8[15:0],%r8b=%r8[7:0]
- %r9=%r9[63:0],%r9d=%r9[31:0],%r9w=%r9[15:0],%r9b=%r9[7:0]

调用者保存：

- %r10=%r10[63:0],%r10d=%r10[31:0],%r10w=%r10[15:0],%r10b=%r10[7:0]
- %r11=%r11[63:0],%r11d=%r11[31:0],%r11w=%r11[15:0],%r11b=%r11[7:0]

寻址方式

立即数寻址

- 格式：$$Imm$ 操作数值：Imm

寄存器寻址

- 格式：r_a 操作数值：$R[r_a]$

绝对寻址

- 格式：Imm 操作数值：$M[Imm]$

间接寻址

- 格式：(r_a) 操作数值：$M[R[r_a]]$

(基址+偏移量)寻址

- 格式：$Imm(r_b)$ 操作数值：$M[Imm+R[r_b]]$

变址寻址

- 格式：(r_b,r_i) 操作数值：$M[R[r_b]+R[r_i]]$
- 格式：$Imm(r_b,r_i)$ 操作数值：$M[Imm+R[r_b]+R[r_i]]$

比例变址寻址

- 格式：$(,r_i,s)$ 操作数值：$M[R[r_i] \cdot s]$
- 格式：$Imm(,r_i,s)$ 操作数值：$M[Imm + R[r_i] \cdot s]$
- 格式：(r_b,r_i,s) 操作数值：$M[R[r_b] + R[r_i] \cdot s]$
- 格式：$Imm(r_b,r_i,s)$ 操作数值：$M[Imm + R[r_b] + R[r_i] \cdot s]$

指令

数据传送指令

```
;普通传送指令
mov a,b          ;将a放入b
movb a,b         ;传送字节
movw a,b         ;传送字
movl a,b         ;传送双字
movq a,b         ;传送四字
movabq a,b       ;传送绝对的四字

;零扩展传送
movz a,b         ;把零扩展a传送到b
movzbw a,b       ;零扩展字节传送到字
movzbl a,b       ;零扩展字节传送到双字
movzwl a,b       ;零扩展字传送到双字
movzbq a,b       ;零扩展字节传送到四字
movzwq a,b       ;零扩展字传送到四字

;符号扩展传送
movs a,b         ;把符号扩展a传送到b
movsbw a,b       ;符号扩展字节传送到字
movsbl a,b       ;符号扩展字节传送到双字
movswl a,b       ;符号扩展字传送到双字
movsbq a,b       ;符号扩展字节传送到四字
movswq a,b       ;符号扩展字传送到四字
movslq a,b       ;符号扩展双字传送到四字
cltq             ;把%eax符号扩展到%rax

;栈操作指令
```

```
pushq a          ;将四字压入栈: R[%rsp] = R[%rsp] - 8 && M[R[%rsp]] = a
popq a           ;将四字弹出栈: a = M[R[%rsp]] && R[%rsp] = R[%rsp] + 8
```

算术逻辑运算

```
;除leaq指令均会改变条件码！！！
leaq a,b         ;b = M[a]
inc a            ;a++
dec a            ;a--
neg a            ;a = -a
not a            ;a = ~a
add a,b          ;b = a + b
sub a,b          ;b = b - a
imul a,b         ;b = a * b
xor a,b          ;b = a ^ b
or a,b           ;b = a | b
and a,b          ;b = a & b
sal k,d          ;d = d << k
shl k,d          ;d = d << k
sar k,d          ;算术右移
shr k,d          ;逻辑右移

;特殊算术操作???
imulq a          ;有符号全乘法
mulq a           ;无符号全乘法
cqto             ;转换为八字
idivq a          ;有符号除法
divq a           ;无符号除法
```

注：ATT汇编格式中操作数的顺序与一般直觉相反

控制指令

条件码

- CF：最高位进位标志，检查上一个操作是否发生溢出
- ZF：零标志，上一个操作是否结果为0
- SF：符号标志，上一个操作是否结果为负数
- OF：溢出标志，上一个操作导致一个补码溢出——正溢出或负溢出

指令

```
cmp a,b          ;比较a,b，根据b-a设置条件码
cmpb a,b         ;比较字节
cmpw a,b         ;比较字
cmpl a,b         ;比较双字
cmpq a,b         ;比较四字
```

```
test a,b        ;测试，根据a&b设置条件码
testb a,b       ;测试字节
testw a,b       ;测试字
testl a,b       ;测试双字
testq a,b       ;测试四字
```

跳转指令

```
jmp Label       ;直接跳转
je Label        ;等于0即跳转(别名jz)
jne Label       ;不为0即跳转(jnz)、
js Label        ;为负即跳转
jns Label       ;不为负即跳转
jg Label        ;有符号大于即跳转(jnle)
jge Label       ;有符号大于等于即跳转(jnl)
jl Label        ;有符号小于即跳转(jnge)
jle Label       ;有符号小于等于即跳转(jng)
ja Label        ;无符号大于即跳转(jnbe)
jae Label       ;无符号大于等于即跳转(jnb)
jb Label        ;无符号小于即跳转(jnae)
jbe Label       ;无符号小于等于即跳转(jna)
```

转移控制指令

```
call Label      ;过程调用
ret             ;从过程调用返回
```

源代码

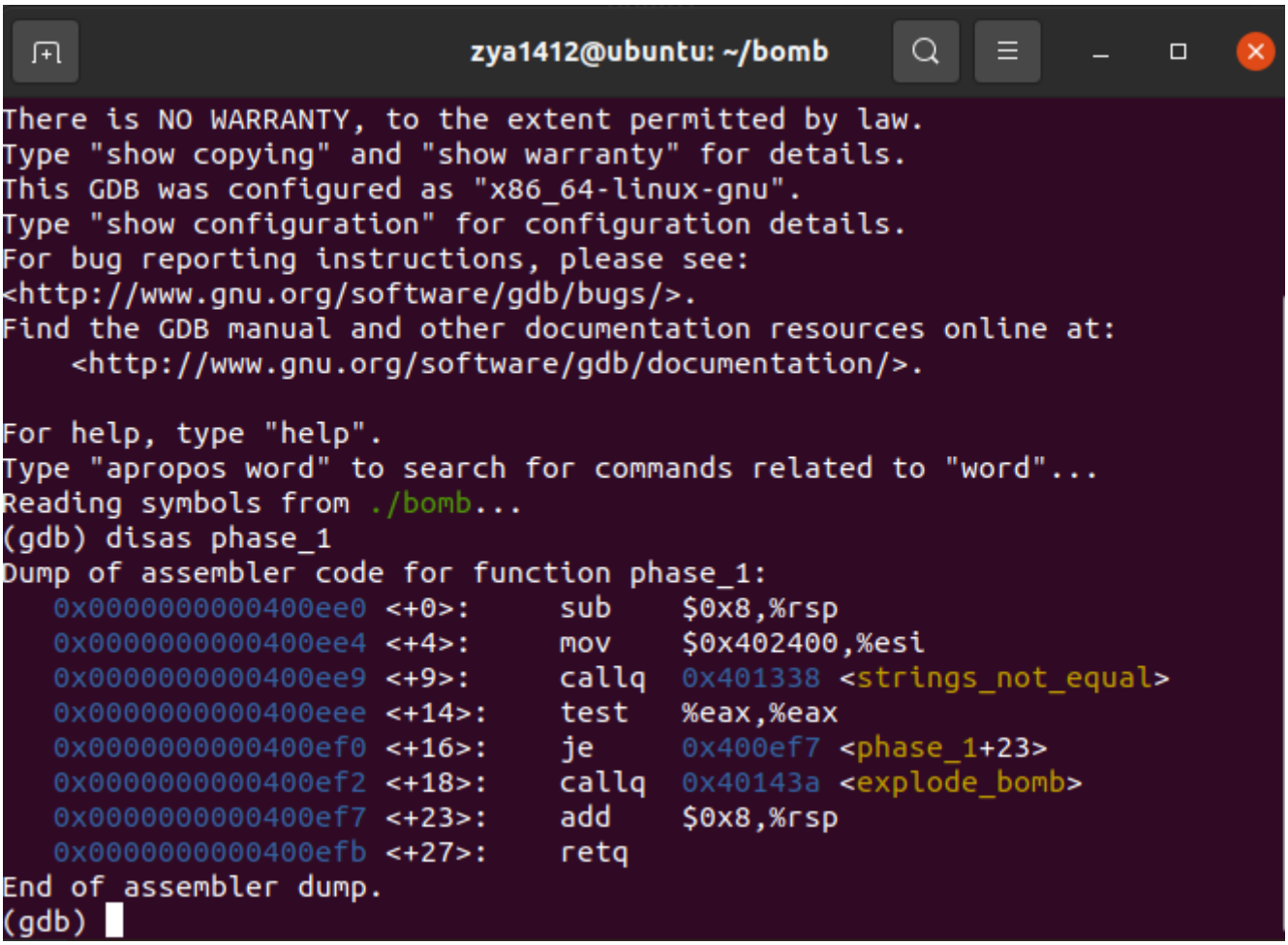
```
/***************************************************************************
 * Dr. Evil's Insidious Bomb, Version 1.1
 * Copyright 2011, Dr. Evil Incorporated. All rights reserved.
 *
 * LICENSE:
 *
 * Dr. Evil Incorporated (the PERPETRATOR) hereby grants you (the
 * VICTIM) explicit permission to use this bomb (the BOMB).  This is a
 * time limited license, which expires on the death of the VICTIM.
 * The PERPETRATOR takes no responsibility for damage, frustration,
 * insanity, bug-eyes, carpal-tunnel syndrome, loss of sleep, or other
 * harm to the VICTIM.  Unless the PERPETRATOR wants to take credit,
 * that is.  The VICTIM may not distribute this bomb source code to
 * any enemies of the PERPETRATOR.  No VICTIM may debug,
 * reverse-engineer, run "strings" on, decompile, decrypt, or use any
```

```
 * other technique to gain knowledge of and defuse the BOMB.  BOMB
 * proof clothing may not be worn when handling this program.  The
 * PERPETRATOR will not apologize for the PERPETRATOR's poor sense of
 * humor.  This license is null and void where the BOMB is prohibited
 * by law.
 ********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "support.h"
#include "phases.h"

/*
 * Note to self: Remember to erase this file so my victims will have no
 * idea what is going on, and so they will all blow up in a
 * spectaculary fiendish explosion. -- Dr. Evil
 */

FILE *infile;

int main(int argc, char *argv[])
{
    char *input;

    /* Note to self: remember to port this bomb to Windows and put a
     * fantastic GUI on it. */

    /* When run with no arguments, the bomb reads its input lines
     * from standard input. */
    if (argc == 1) {
    infile = stdin;
    }

    /* When run with one argument <file>, the bomb reads from <file>
     * until EOF, and then switches to standard input. Thus, as you
     * defuse each phase, you can add its defusing string to <file> and
     * avoid having to retype it. */
    else if (argc == 2) {
    if (!(infile = fopen(argv[1], "r"))) {
        printf("%s: Error: Couldn't open %s\n", argv[0], argv[1]);
        exit(8);
    }
    }

    /* You can't call the bomb with more than 1 command line argument. */
    else {
    printf("Usage: %s [<input_file>]\n", argv[0]);
    exit(8);
    }

    /* Do all sorts of secret stuff that makes the bomb harder to defuse. */
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
```

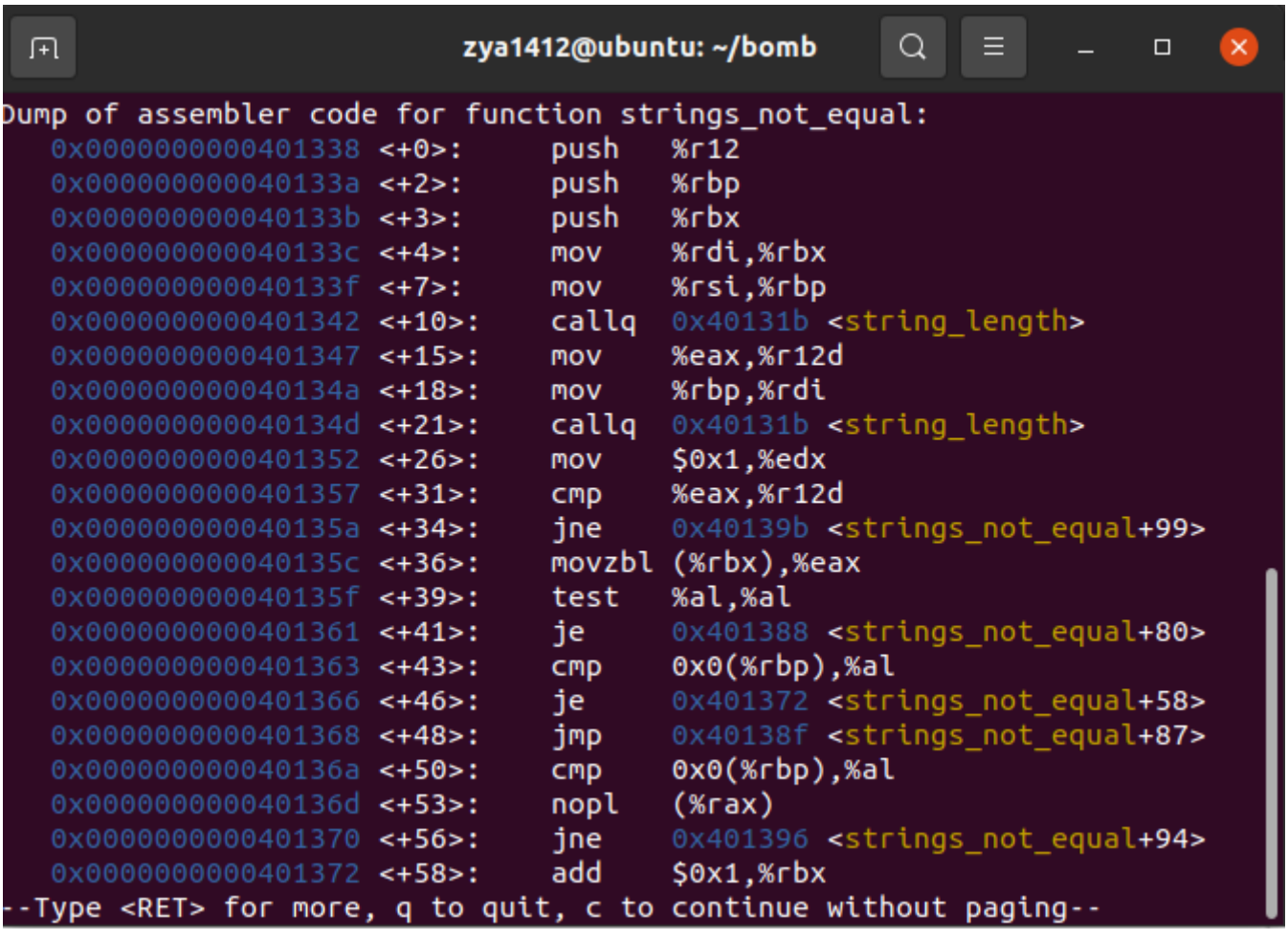
```
    printf("which to blow yourself up. Have a nice day!\n");

    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();
    printf("That's number 2.  Keep going!\n");

    /* I guess this is too easy so far.  Some more complex code will
     * confuse people. */
    input = read_line();
    phase_3(input);
    phase_defused();
    printf("Halfway there!\n");

    /* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
    input = read_line();
    phase_4(input);
    phase_defused();
    printf("So you got that one.  Try this one.\n");

    /* Round and 'round in memory we go, where we stop, the bomb blows! */
    input = read_line();
    phase_5(input);
    phase_defused();
    printf("Good work!  On to the next...\n");

    /* This phase will never be used, since no one will get past the
     * earlier ones.  But just in case, make this one extra hard. */
    input = read_line();
    phase_6(input);
    phase_defused();

    /* Wow, they got it!  But isn't something... missing?  Perhaps
     * something they overlooked?  Mua ha ha ha ha! */

    return 0;
}
```

Phase 1

从源代码中可以看出，phase1先读取一个字符串，然后进入phase_1：

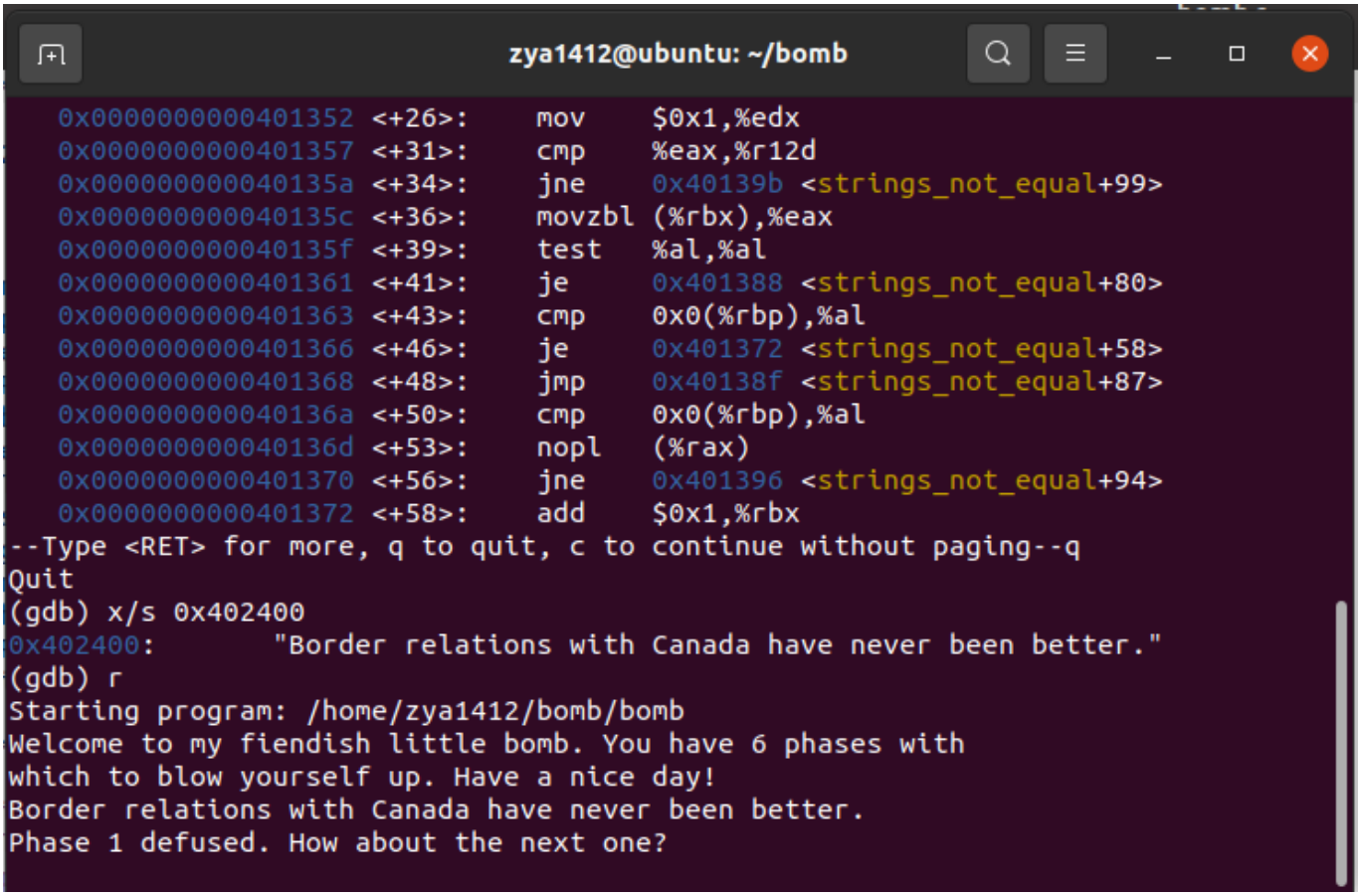


进一步反汇编其中的strings_not_equal函数：



我们可以看到strings_not_equal比较输入字符串和函数中的字符串，若不一样则引爆炸弹

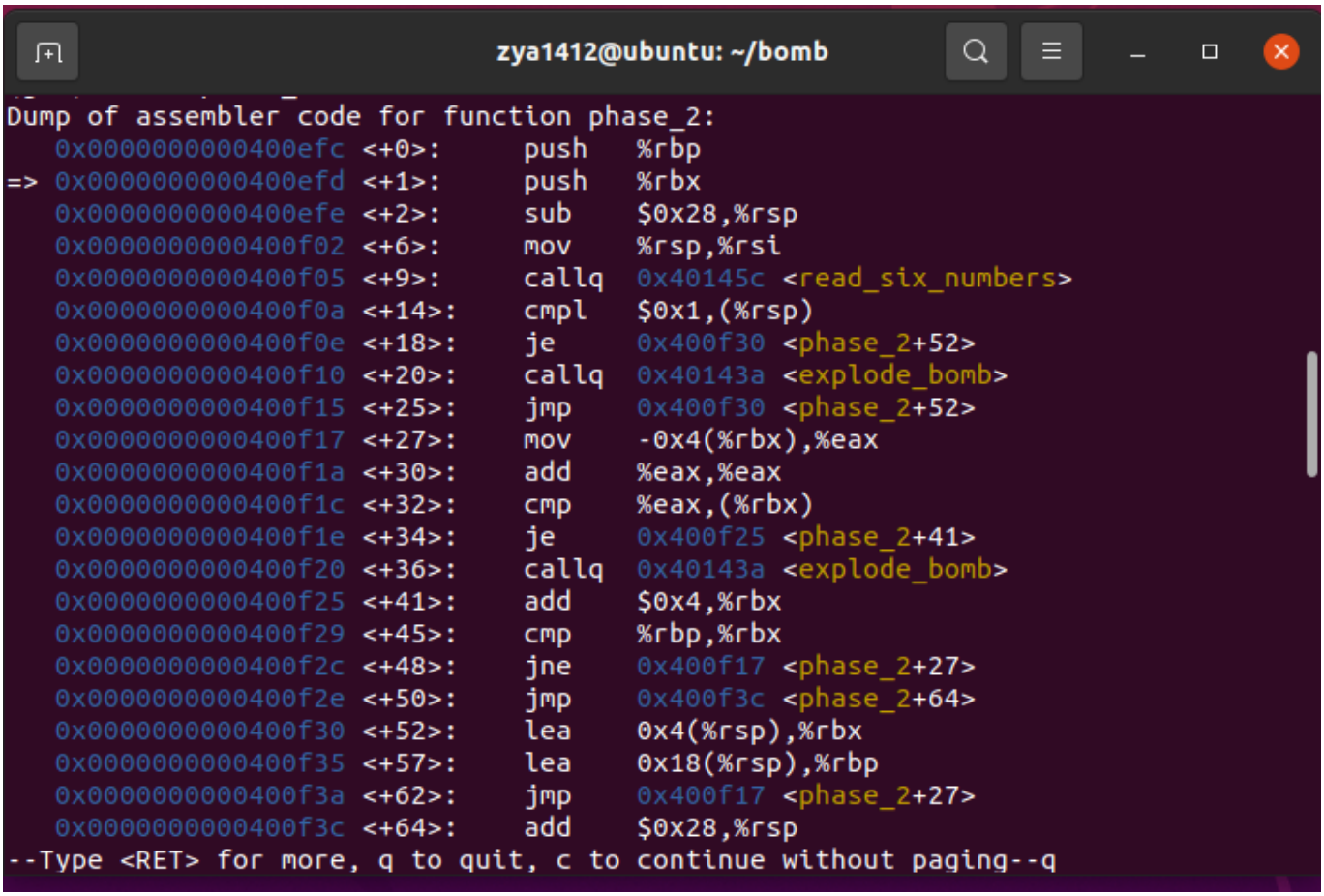
于是观察到phase_1中进入string_not_equal前只有两个初始化操作，那么应该一个为保存在栈中的参数即输入字符串；另一个为函数中的默认值，所以查看该地址0x402400中的值就可以得到我们想要的正确结果：



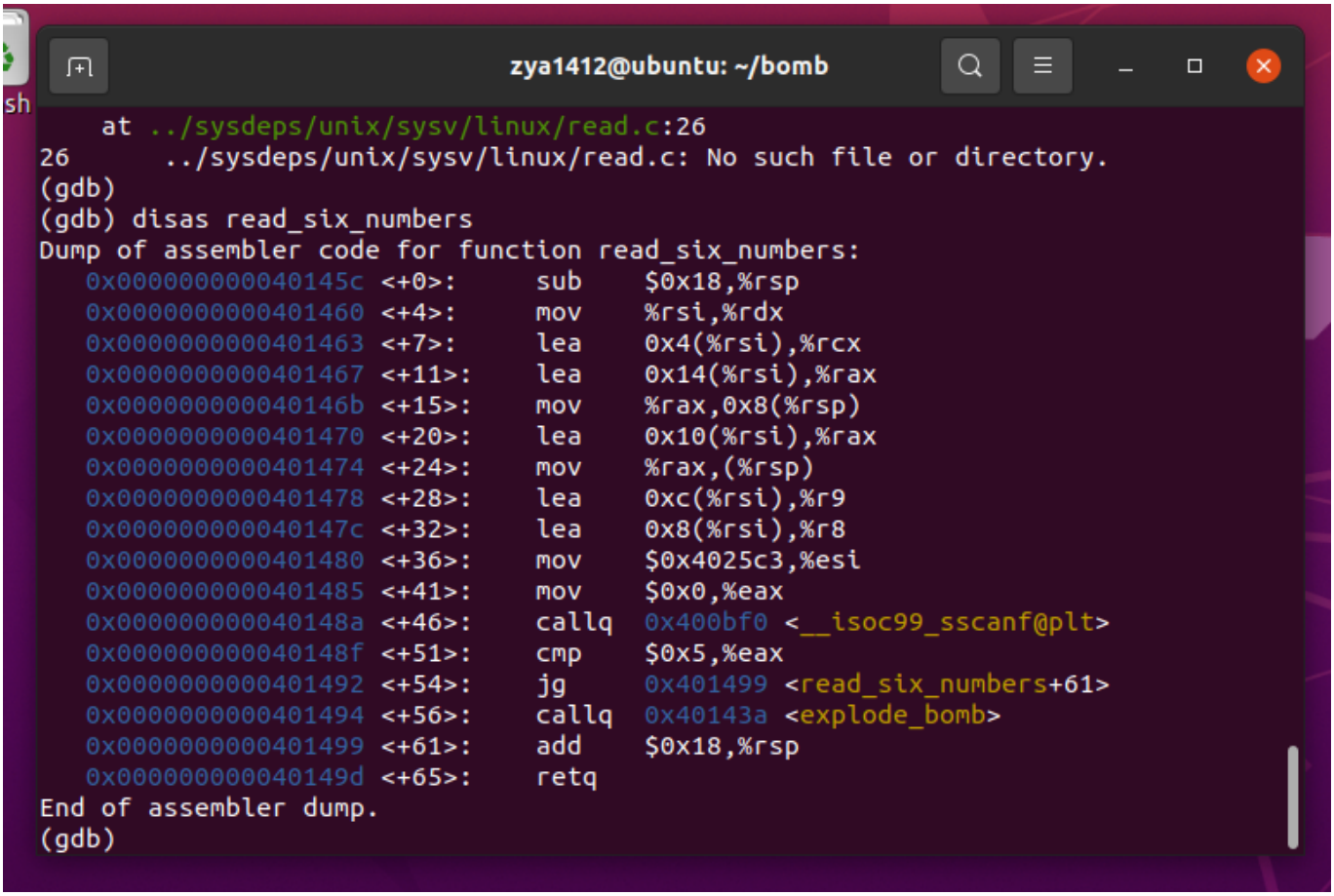
phase 1完成!

Phase 2

phase_2函数的反汇编如下：



前四条指令做了初始化，然后进入了read_six_numbers函数，从名字上来看是读取了六个数字，我们反汇编该函数：

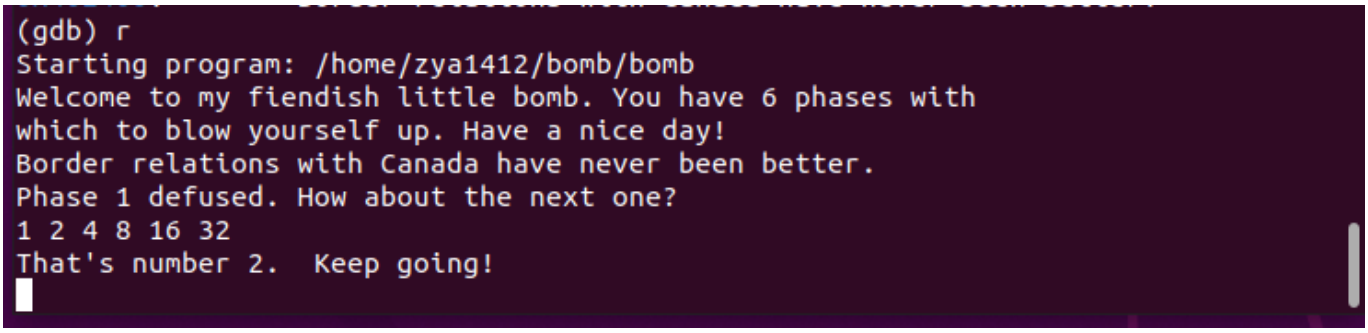


我们可以看到，同phase1，在sscanf函数之前读入的常值字符串储存在0x4025c3，我们读取该地址中的内容可得：

```
(gdb) x/s 0x4025c3
0x4025c3:    "%d %d %d %d %d %d"
```

所以可以看到是空格相隔的六个整数

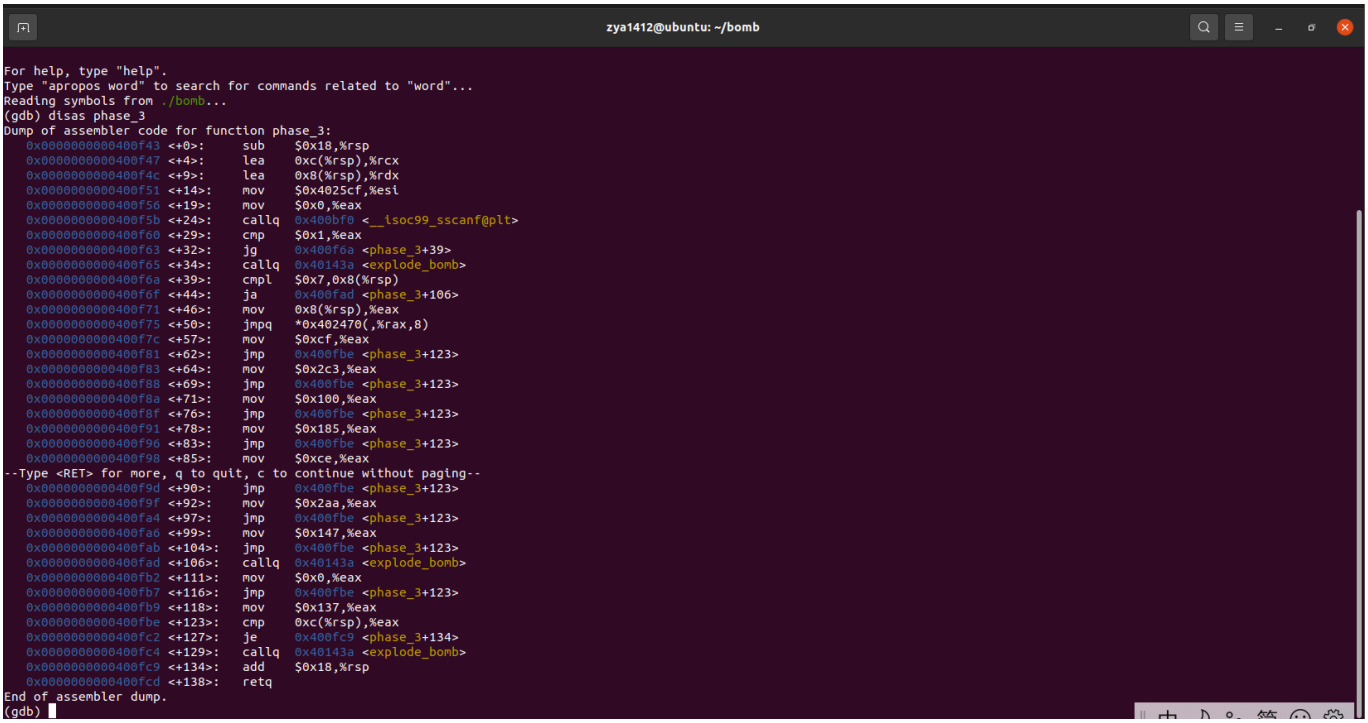
然后回来分析phase_2中的函数逻辑，先比较第一个栈中第一个数据，判断是否为1(0x400f0a)，若是则转到0x400f30：将栈指针的下一个存到rbx寄存器，将第六个参数的栈地址存到rbp寄存器，然后跳转到0x400f17：将栈中之前一个（rbp-4中）的参数，在首次操作中也就是1，放入eax，然后将eax乘2与下一个参数作比较，重复这样的操作六次，这也就是说，我们要输入的整数序列为一个首项为1公比为2的等比数列，这样我们得到了phase2的答案字符串：1 2 4 8 16 32



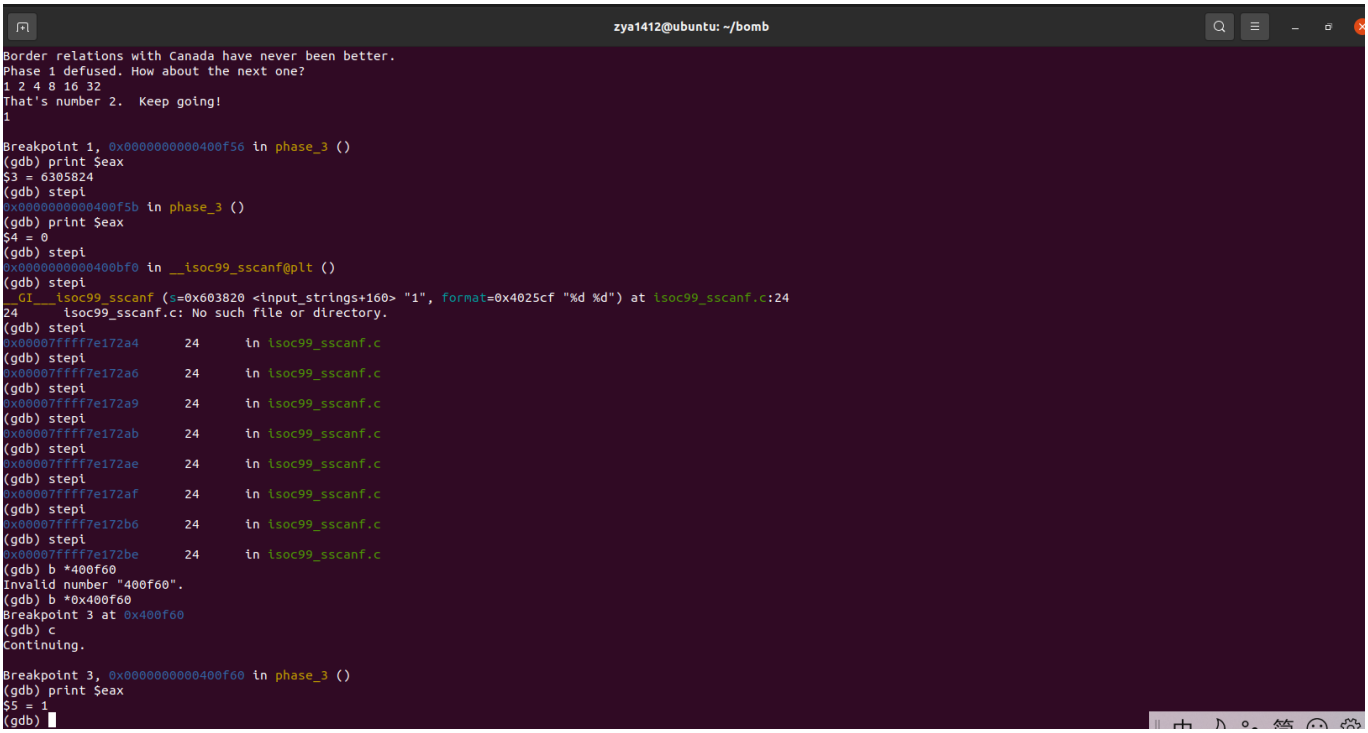
phase2完成!

Phase 3

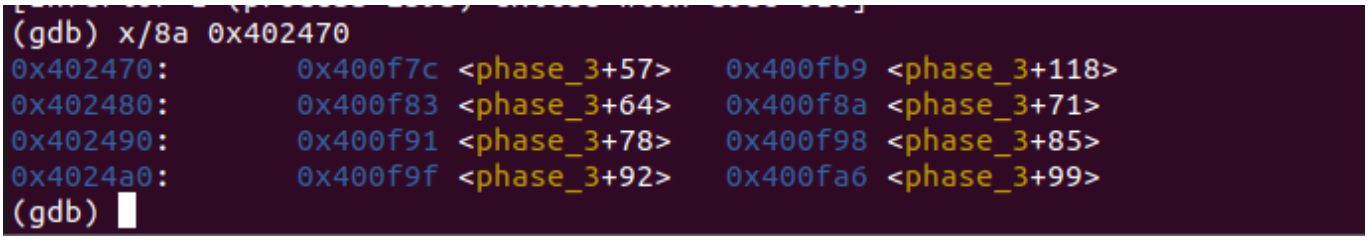
phase_3函数的反汇编如下：



对0x400f56和0x400f60两处的指令下断点进行调试，我们可以得到sscanf的常值字符串参数为%d %d，且eax中储存的是sscanf读入的数字个数：



接着我们重新观察phase_3的函数逻辑：在读入数字后，比较读入数字是否大于1，若非大于1，则跳转到0x400f65引爆炸弹。故我们需要输入大于1个数字(从调试得到的信息可知我们需要输入两个数字)，然后将第一个数字放入eax中，比较第一个数字是否大于等于7，若是则引爆炸弹，故我们应输入一个小于7的参数，继续向下看，将第二个参数放入eax寄存器，然后跳转到*(0x402470)+($%rax \times 8$)，这里我们需要查看0x402470及其后7个内存中保存的内容：



也就是0x400f7c，所以我们可以看到，根据输入不同参数值跳转到下面不同的位置，在将一个值放入eax后，跳转到0x400fbe，比较eax和第二个参数的值，若相等则完成phase_3，反之炸弹爆炸。

由上述程序逻辑，我们可以得到7组对应的参数数值：

- 0x0 0xcf
- 0x1 0x137
- 0x2 0x2c3
- 0x3 0x100
- 0x4 0x185
- 0x5 0xce
- 0x6 0x2aa
- 0x7 0x147

```
(gdb) r
Starting program: /home/zya1412/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
1 311
Halfway there!
^C
```

Phase3完成!

Phase 4

先来观察phase_4的汇编代码:

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x000000000040100c <+0>:     sub    $0x18,%rsp
   0x0000000000401010 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000401015 <+9>:     lea    0x8(%rsp),%rdx
   0x000000000040101a <+14>:    mov    $0x4025cf,%esi
   0x000000000040101f <+19>:    mov    $0x0,%eax
   0x0000000000401024 <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000401029 <+29>:    cmp    $0x2,%eax
   0x000000000040102c <+32>:    jne    0x401035 <phase_4+41>
   0x000000000040102e <+34>:    cmpl   $0xe,0x8(%rsp)
   0x0000000000401033 <+39>:    jbe    0x40103a <phase_4+46>
   0x0000000000401035 <+41>:    callq  0x40143a <explode_bomb>
   0x000000000040103a <+46>:    mov    $0xe,%edx
   0x000000000040103f <+51>:    mov    $0x0,%esi
   0x0000000000401044 <+56>:    mov    0x8(%rsp),%edi
   0x0000000000401048 <+60>:    callq  0x400fce <func4>
   0x000000000040104d <+65>:    test   %eax,%eax
   0x000000000040104f <+67>:    jne    0x401058 <phase_4+76>
   0x0000000000401051 <+69>:    cmpl   $0x0,0xc(%rsp)
   0x0000000000401056 <+74>:    je     0x40105d <phase_4+81>
   0x0000000000401058 <+76>:    callq  0x40143a <explode_bomb>
   0x000000000040105d <+81>:    add    $0x18,%rsp
   0x0000000000401061 <+85>:    retq
```

同上题读取0x4025cf可看出要输入两个整数，必须满足第一个参数小于14(0x40102e处指令)，然后进入func4，先抛开func4不谈看func4后面的指令，判断第二个参数是否为0，只有为0时才可避免explode，第二个参数显然可得，接下来看func4来寻找第一个参数的有关信息:

```
Dump of assembler code for function func4:
   0x0000000000400fce <+0>:     sub    $0x8,%rsp
   0x0000000000400fd2 <+4>:     mov    %edx,%eax
   0x0000000000400fd4 <+6>:     sub    %esi,%eax
   0x0000000000400fd6 <+8>:     mov    %eax,%ecx
   0x0000000000400fd8 <+10>:    shr    $0x1f,%ecx
   0x0000000000400fdb <+13>:    add    %ecx,%eax
   0x0000000000400fdd <+15>:    sar    %eax
   0x0000000000400fdf <+17>:    lea    (%rax,%rsi,1),%ecx
   0x0000000000400fe2 <+20>:    cmp    %edi,%ecx
   0x0000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
   0x0000000000400fe6 <+24>:    lea    -0x1(%rcx),%edx
   0x0000000000400fe9 <+27>:    callq  0x400fce <func4>
   0x0000000000400fee <+32>:    add    %eax,%eax
   0x0000000000400ff0 <+34>:    jmp    0x401007 <func4+57>
   0x0000000000400ff2 <+36>:    mov    $0x0,%eax
   0x0000000000400ff7 <+41>:    cmp    %edi,%ecx
   0x0000000000400ff9 <+43>:    jge    0x401007 <func4+57>
   0x0000000000400ffb <+45>:    lea    0x1(%rcx),%esi
   0x0000000000400ffe <+48>:    callq  0x400fce <func4>
   0x0000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401007 <+57>:    add    $0x8,%rsp
   0x000000000040100b <+61>:    retq
```

func4手动逆向为c语言代码如下：

```c
//直接逆向版
int func4(int arg1,int esi,int edx)
{
    edi = arg1;
    //edx = 14;
    //esi = 0;
    eax = edx;
    ecx = edx;
    ecx = (unsigned)ecx >> 31;
    eax = eax + (int)ecx;
    eax = eax >> 1;
    ecx = eax;
    if(edi <= ecx)
    {
        eax = 0;
        if(edi >= ecx) return 0;
        else return 2 * func4(arg1,ecx + 1,edx);
    }
    else
    {
        edx = 6;
        func4(arg1,esi,ecx - 1);
    }
    eax <<= 1;
    retern eax;
}
```

```
//优化版
int func4(int arg1,int esi,int edx)
{
    int flag = edx - esi;
    flag = (int)(((unsigned)flag >> 31) + flag);
    flag >>= 1;
    flag += esi;
    if(flag <= arg1)
    {
        if(arg1 == flag) return 0;
        else return 2 * func4(arg1,ecx + 1,edx);
    }
    else
    {
        return 2*func4(arg1,esi,eci - 1);
    }
}
```

可以看到，不去算复杂递归函数结果的话，7是一个满足条件的解，输入7 0即可

```
(gdb) r ans.txt
Starting program: /home/zya1412/bomb/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!
7 0
So you got that one.  Try this one.
```

phase_4完成!

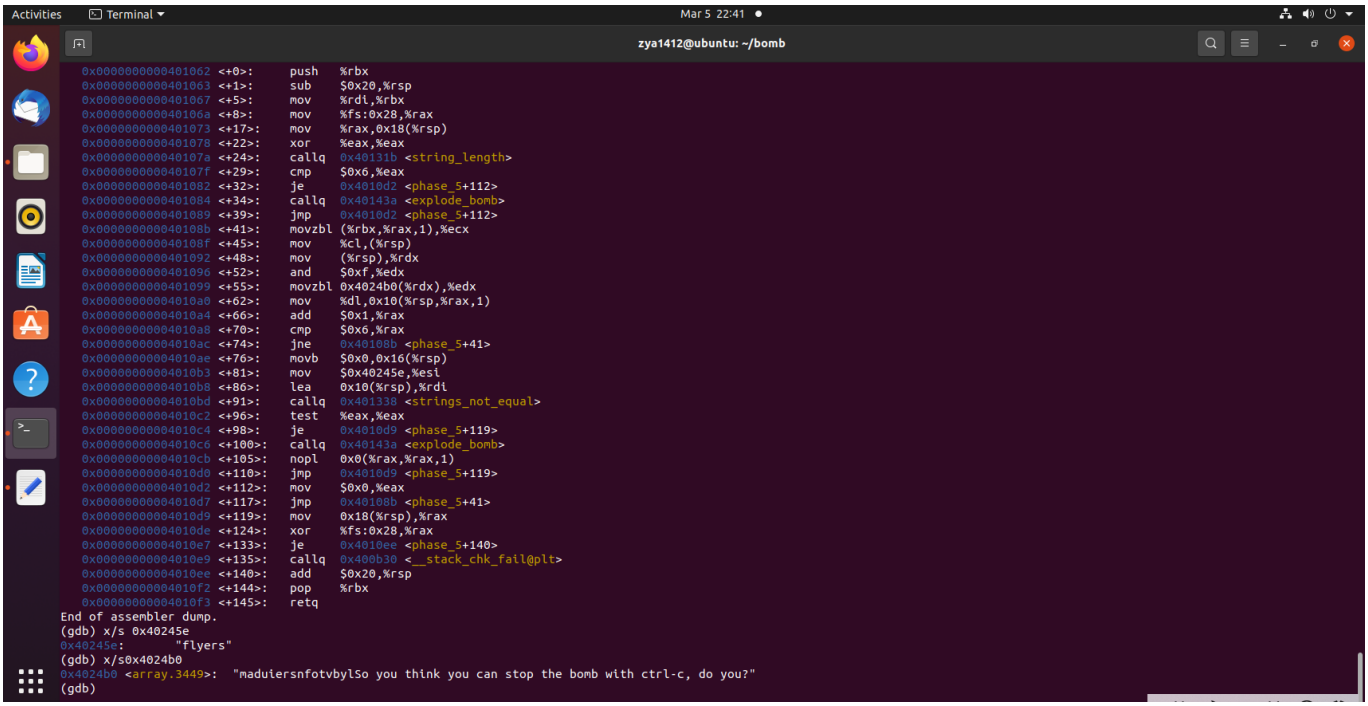
Phase 5

phase_5的汇编代码如下:

```
Dump of assembler code for function phase_5:
   0x0000000000401062 <+0>:     push   %rbx
   0x0000000000401063 <+1>:     sub    $0x20,%rsp
   0x0000000000401067 <+5>:     mov    %rdi,%rbx
   0x000000000040106a <+8>:     mov    %fs:0x28,%rax
   0x0000000000401073 <+17>:    mov    %rax,0x18(%rsp)
   0x0000000000401078 <+22>:    xor    %eax,%eax
   0x000000000040107a <+24>:    callq  0x40131b <string_length>
   0x000000000040107f <+29>:    cmp    $0x6,%eax
   0x0000000000401082 <+32>:    je     0x4010d2 <phase_5+112>
   0x0000000000401084 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000401089 <+39>:    jmp    0x4010d2 <phase_5+112>
   0x000000000040108b <+41>:    movzbl (%rbx,%rax,1),%ecx
   0x000000000040108f <+45>:    mov    %cl,(%rsp)
   0x0000000000401092 <+48>:    mov    (%rsp),%rdx
   0x0000000000401096 <+52>:    and    $0xf,%edx
   0x0000000000401099 <+55>:    movzbl 0x4024b0(%rdx),%edx
   0x00000000004010a0 <+62>:    mov    %dl,0x10(%rsp,%rax,1)
   0x00000000004010a4 <+66>:    add    $0x1,%rax
   0x00000000004010a8 <+70>:    cmp    $0x6,%rax
   0x00000000004010ac <+74>:    jne    0x40108b <phase_5+41>
   0x00000000004010ae <+76>:    movb   $0x0,0x16(%rsp)
   0x00000000004010b3 <+81>:    mov    $0x40245e,%esi
   0x00000000004010b8 <+86>:    lea    0x10(%rsp),%rdi
   0x00000000004010bd <+91>:    callq  0x401338 <strings_not_equal>
   0x00000000004010c2 <+96>:    test   %eax,%eax
   0x00000000004010c4 <+98>:    je     0x4010d9 <phase_5+119>
   0x00000000004010c6 <+100>:   callq  0x40143a <explode_bomb>
   0x00000000004010cb <+105>:   nopl   0x0(%rax,%rax,1)
   0x00000000004010d0 <+110>:   jmp    0x4010d9 <phase_5+119>
   0x00000000004010d2 <+112>:   mov    $0x0,%eax
   0x00000000004010d7 <+117>:   jmp    0x40108b <phase_5+41>
   0x00000000004010d9 <+119>:   mov    0x18(%rsp),%rax
   0x00000000004010de <+124>:   xor    %fs:0x28,%rax
   0x00000000004010e7 <+133>:   je     0x4010ee <phase_5+140>
   0x00000000004010e9 <+135>:   callq  0x400b30 <__stack_chk_fail@plt>
   0x00000000004010ee <+140>:   add    $0x20,%rsp
   0x00000000004010f2 <+144>:   pop    %rbx
   0x00000000004010f3 <+145>:   retq
```

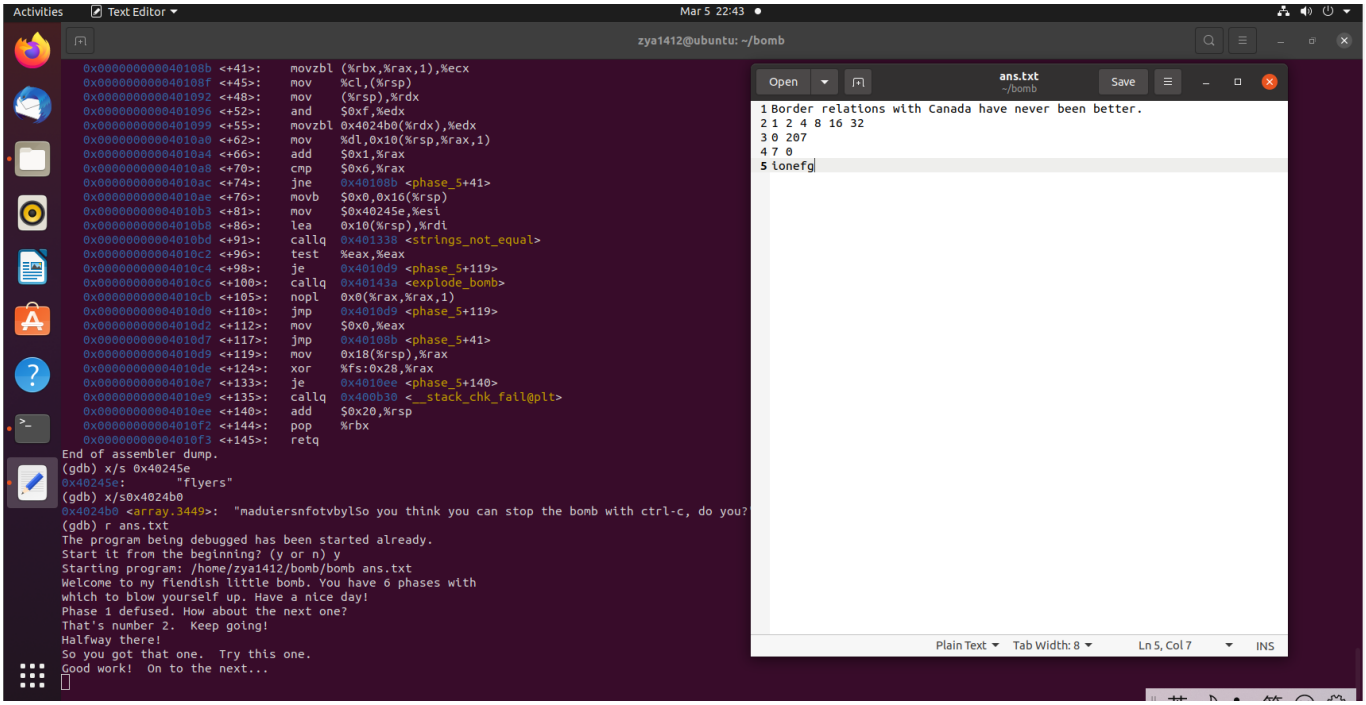
分析函数逻辑：先对局部变量进行初始化，xor给eax清0，检查字符串的长度，若长度不为6的爆炸，反之则跳转到0x4010d2，清0eax后进入循环：从0x40108b到0x4010ac是一个循环：先将输入的值与0xf做与运算，也就是取这个字符的最后一位(16进制)，然后以该位为偏移量shift访问0x4024b0+shift放入栈中，循环六次后跳出，然后判断这样取出的六个字符是否与0x40245e中储存的字符串相同，若不同炸弹爆炸；相同则完成此关。

故问题的关键是0x4024b0和0x40245e中储存的字符串：



所以要取的数据下标分别为：9 f e 5 6 7

对照ascii码我们可以取i o n e f g



phase_5完成!

Phase 6

查看phase_6的反汇编代码：

```
Dump of assembler code for function phase_6:
   0x00000000004010f4 <+0>: push   %r14
   0x00000000004010f6 <+2>: push   %r13
   0x00000000004010f8 <+4>: push   %r12
```

```
0x00000000004010fa <+6>: push   %rbp
0x00000000004010fb <+7>: push   %rbx
0x00000000004010fc <+8>: sub    $0x50,%rsp
0x0000000000401100 <+12>:    mov    %rsp,%r13
0x0000000000401103 <+15>:    mov    %rsp,%rsi
0x0000000000401106 <+18>:    callq  0x40145c <read_six_numbers>
0x000000000040110b <+23>:    mov    %rsp,%r14
0x000000000040110e <+26>:    mov    $0x0,%r12d
0x0000000000401114 <+32>:    mov    %r13,%rbp
0x0000000000401117 <+35>:    mov    0x0(%r13),%eax
0x000000000040111b <+39>:    sub    $0x1,%eax
0x000000000040111e <+42>:    cmp    $0x5,%eax
0x0000000000401121 <+45>:    jbe    0x401128 <phase_6+52>
0x0000000000401123 <+47>:    callq  0x40143a <explode_bomb>
0x0000000000401128 <+52>:    add    $0x1,%r12d
0x000000000040112c <+56>:    cmp    $0x6,%r12d
0x0000000000401130 <+60>:    je     0x401153 <phase_6+95>
0x0000000000401132 <+62>:    mov    %r12d,%ebx
0x0000000000401135 <+65>:    movslq %ebx,%rax
0x0000000000401138 <+68>:    mov    (%rsp,%rax,4),%eax
0x000000000040113b <+71>:    cmp    %eax,0x0(%rbp)
0x000000000040113e <+74>:    jne    0x401145 <phase_6+81>
0x0000000000401140 <+76>:    callq  0x40143a <explode_bomb>
0x0000000000401145 <+81>:    add    $0x1,%ebx
0x0000000000401148 <+84>:    cmp    $0x5,%ebx
0x000000000040114b <+87>:    jle    0x401135 <phase_6+65>
0x000000000040114d <+89>:    add    $0x4,%r13
0x0000000000401151 <+93>:    jmp    0x401114 <phase_6+32>
0x0000000000401153 <+95>:    lea    0x18(%rsp),%rsi
0x0000000000401158 <+100>:   mov    %r14,%rax
0x000000000040115b <+103>:   mov    $0x7,%ecx
0x0000000000401160 <+108>:   mov    %ecx,%edx
0x0000000000401162 <+110>:   sub    (%rax),%edx
0x0000000000401164 <+112>:   mov    %edx,(%rax)
0x0000000000401166 <+114>:   add    $0x4,%rax
0x000000000040116a <+118>:   cmp    %rsi,%rax
0x000000000040116d <+121>:   jne    0x401160 <phase_6+108>
0x000000000040116f <+123>:   mov    $0x0,%esi
0x0000000000401174 <+128>:   jmp    0x401197 <phase_6+163>
0x0000000000401176 <+130>:   mov    0x8(%rdx),%rdx
0x000000000040117a <+134>:   add    $0x1,%eax
0x000000000040117d <+137>:   cmp    %ecx,%eax
0x000000000040117f <+139>:   jne    0x401176 <phase_6+130>
0x0000000000401181 <+141>:   jmp    0x401188 <phase_6+148>
0x0000000000401183 <+143>:   mov    $0x6032d0,%edx
0x0000000000401188 <+148>:   mov    %rdx,0x20(%rsp,%rsi,2)
0x000000000040118d <+153>:   add    $0x4,%rsi
0x0000000000401191 <+157>:   cmp    $0x18,%rsi
0x0000000000401195 <+161>:   je     0x4011ab <phase_6+183>
0x0000000000401197 <+163>:   mov    (%rsp,%rsi,1),%ecx
0x000000000040119a <+166>:   cmp    $0x1,%ecx
0x000000000040119d <+169>:   jle    0x401183 <phase_6+143>
0x000000000040119f <+171>:   mov    $0x1,%eax
0x00000000004011a4 <+176>:   mov    $0x6032d0,%edx
```

```
   0x00000000004011a9 <+181>:   jmp    0x401176 <phase_6+130>
   0x00000000004011ab <+183>:   mov    0x20(%rsp),%rbx
   0x00000000004011b0 <+188>:   lea    0x28(%rsp),%rax
   0x00000000004011b5 <+193>:   lea    0x50(%rsp),%rsi
   0x00000000004011ba <+198>:   mov    %rbx,%rcx
   0x00000000004011bd <+201>:   mov    (%rax),%rdx
   0x00000000004011c0 <+204>:   mov    %rdx,0x8(%rcx)
   0x00000000004011c4 <+208>:   add    $0x8,%rax
   0x00000000004011c8 <+212>:   cmp    %rsi,%rax
   0x00000000004011cb <+215>:   je     0x4011d2 <phase_6+222>
   0x00000000004011cd <+217>:   mov    %rdx,%rcx
   0x00000000004011d0 <+220>:   jmp    0x4011bd <phase_6+201>
   0x00000000004011d2 <+222>:   movq   $0x0,0x8(%rdx)
   0x00000000004011da <+230>:   mov    $0x5,%ebp
   0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
   0x00000000004011e3 <+239>:   mov    (%rax),%eax
   0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
   0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250>
   0x00000000004011e9 <+245>:   callq  0x40143a <explode_bomb>
   0x00000000004011ee <+250>:   mov    0x8(%rbx),%rbx
   0x00000000004011f2 <+254>:   sub    $0x1,%ebp
   0x00000000004011f5 <+257>:   jne    0x4011df <phase_6+235>
   0x00000000004011f7 <+259>:   add    $0x50,%rsp
   0x00000000004011fb <+263>:   pop    %rbx
   0x00000000004011fc <+264>:   pop    %rbp
   0x00000000004011fd <+265>:   pop    %r12
   0x00000000004011ff <+267>:   pop    %r13
   0x0000000000401201 <+269>:   pop    %r14
   0x0000000000401203 <+271>:   retq
```

可以看到该程序先读入六个数字，给第一个参数减一后与5比较，第一个数小于等于5时继续，否则炸弹引爆；然后给一个初始化为0的寄存器r12加1(r12是一个判断是否跳转的计数器，当r12到6时跳转)，将下一个数放入eax，若下一个数与上一个相等则炸弹引爆；然后给ebx加1(ebx同r12，不过边界为5)，取下一个参数，跳回0x401114，即循环入口，该循环的c语言表示如下：

```c
for(inr i = 0;i < 6;i++)
{
    if(num[i] > 6)
    {
        explode_bomb();
    }
    for(int j = i + 1;j <= 5;j++)
    {
        if(num[j] == num[i])
        {
            explode_bomb();
        }
    }
}
```

将第六个参数放入rsi，将第一个数的地址放入rax，令ecx=edx=7，令edx=7 - edx，使第一个参数arg1变为7 - arg1，再给此时的arg1加4，取下一个数，判断是否变为最后一个参数，若否则对每个参数都进行上述操作，结束循环后跳转到0x4001160，然后依次取栈中的参数，若数字小于等于1则跳转到0x4001183，然后将0x6032d0放入edx，每次取rdx放入$M[%rsp+%rsi*2]$ *中(作为下一个地址，像链表的next)，然后给rsi+4(进行偏移)，直到循环至最后一个参数为止，循环结束后跳转到0x40011ab，将第一个参数放入rbx，rax指向第二个参数的地址，rsi指向最后一个参数的next地址(当作结束地址)，令rcx=rbx(第一个参数)，rdx*=(rax)(下一个地址的值，即下一个参数)，然后指向第一个参数->next->next，判断是否到最后一个节点，若否则将第一个第一个值改为下一个的值，然后根据不同情况跳转。

此处我们输入一组数据发现栈中以三个地址为一个结构体，第一个数字储存的是一个特殊值，第二个是我们的参数，第三个是指向下一个结构体的地址的指针；而不是刚才以为的第一个即是参数。而重新分析上述代码可知，我们输入的序列就是每个节点的连接顺序(0x4011ab-0x4011d2处的循环)

```
Breakpoint 2, 0x00000000004011da in phase_6 ()
(gdb) x/24w 0x6032d0
0x6032d0 <node1>:       332     1       6304480 0
0x6032e0 <node2>:       168     2       0       0
0x6032f0 <node3>:       924     3       6304512 0
0x603300 <node4>:       691     4       6304528 0
0x603310 <node5>:       477     5       6304544 0
0x603320 <node6>:       443     6       6304464 0
(gdb)
```

而最后的程序逻辑如下：

先令ebp=5，然后令eax指向ebx的下一个结构体，然后令eax保存它内部的值，比较eax和ebx的值，如果eax小于ebx则爆炸，反之则跳转到0x4011ee，即跳过爆炸，然后循环。也就是说我们要做的就是保证到这里每个节点的第一个值是按倒序排列的。而从上图我们可以看出序列应该为3 4 5 6 1 2，而我们这时得到的序列是由7 - 原数得到的，故原序列应为4 3 2 1 6 5

```
zya1412@ubuntu:~/bomb$ ./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!
zya1412@ubuntu:~/bomb$
```

phase_6完成!

最后答案的文本文件内容为：

```
Border relations with Canada have never been better.
1 2 4 8 16 32
0 207
7 0
ionefg
4 3 2 1 6 5
```

总结

本次实验考察了对代码逆向工程的能力(手撕汇编干一次就再也不想干了...)，感觉比datalab难了很多orz，耗时一周，收获了很多，例如对程序逻辑的分析：数据在函数中是如何传递，运算；各种跳转以及递归如何进行。对gdb的使用：下断点进行调试，用命令查看地址各种形式的值，后面基本可以脱离手册来操作gdb等。最后phase_6分析链表尤其让我印象深刻...本来以为链表是一种十分简单浅显的数据结构，但是经过phase_6分析之后才知道在汇编下链表的操作也显得十分复杂。

做完本次实验后其实还有一个预料之外的收获：对编译过程的兴趣upup...分析这六个phase后感觉这六个phase的题在高级语言中都是很简单的逻辑，但是如果从汇编层面的角度来看都显得复杂而繁琐，那么汇编是如何处理那些高级数据结构或者是复杂逻辑的程序呢？希望以后能在这方面多了解一下吧大概...