Border relations with Canada have never been better.

Data Lab & Bomb Lab讲解

Copyright © 2021 ECNU Corporation. All rights reserved.
Tel:+86-021-62233586 Fax:+86-021-62606775
E-mail: ecnu@ecnu.com.cn Http: //www.ecnu.edu.cn

BEFORE WE GO

我们先来熟悉一些常用的位运算技巧：

- 对第K位置位
- 清除第K位
- 对第K位取反
- 提取第K位内容
- 替换一段二进制内容

C Bitwise Operators

Operator	Description
&	AND
|	OR
^	XOR (exclusive OR)
~	NOT (one's complement)
<<	shift left
>>	shift right

Examples (8-bit word)

```
A = 0b10110011
B = 0b01101001

A&B = 0b00100001        ~A = 0b01001100
A|B = 0b11111011    A >> 3 = 0b00010110
A^B = 0b11011010    A << 2 = 0b11001100
```

© 2008–2018 by the MIT 6.172 Lecturers

BEFORE WE GO

我们先来熟悉一些常用的位运算技巧：

- 对第K位置位
- 清除第K位
- 对第K位取反
- 提取第K位内容
- 替换一段二进制内容

Set the kth Bit

Problem

Set kth bit in a word x to 1.

Idea

Shift and OR.

```
y = x | (1 << k);
```

Example

k = 7

x	1011110101101101
1 << k	0000000010000000
x | (1 << k)	1011110111101101

© 2008–2018 by the MIT 6.172 Lecturers

BEFORE WE GO

我们先来熟悉一些常用的位运算技巧：

- 对第K位置位
- 清除第K位
- 对第K位取反
- 提取第K位内容
- 替换一段二进制内容

Clear the kth Bit

Problem

Clear the kth bit in a word x.

Idea

Shift, complement, and AND.

```
y = x & ~(1 << k);
```

Example

k = 7

x	1011110111101101
1 << k	0000000010000000
~(1 << k)	1111111101111111
x & ~(1 << k)	1011110101101101

© 2008–2018 by the MIT 6.172 Lecturers

BEFORE WE GO

我们先来熟悉一些常用的位运算技巧：

- 对第K位置位
- 清除第K位
- 对第K位取反
- 提取第K位内容
- 替换一段二进制内容

Toggle the kth Bit

Problem

Flip the k th bit in a word x .

Idea

Shift and XOR.

```
y = x ^ (1 << k);
```

Example (1 → 0)

k = 7

x	1011110111101101
1 << k	0000000010000000
x ^ (1 << k)	1011110101101101

© 2008–2018 by the MIT 6.172 Lecturers

BEFORE WE GO

我们先来熟悉一些常用的位运算技巧：

- 对第K位置位
- 清除第K位
- 对第K位取反
- 提取第K位内容
- 替换一段二进制内容

Extract a Bit Field

Problem

Extract a bit field from a word `x`.

Idea

Mask and shift.

```
(x & mask) >> shift;
```

Example

`shift = 7`

x	1011110101101101
mask	0000011110000000
x & mask	0000010100000000
x & mask >> shift	0000000000001010

© 2008–2018 by the MIT 6.172 Lecturers

BEFORE WE GO

我们先来熟悉一些常用的位运算技巧：

- 对第K位置位
- 清除第K位
- 对第K位取反
- 提取第K位内容
- 替换一段二进制内容

Set a Bit Field

Problem

Set a bit field in a word x to a val

For safety's sake:
((y << shift) & mask)

Idea

Invert mask to clear, and OR the sh value.

```
x = (x & ~mask) | (y << shift);
```

Example

shift = 7

x	1011110101101101
y	0000000000000011
mask	0000011110000000
x & ~mask	1011100001101101
x = (x & ~mask) | (y << shift);	1011100111101101

© 2008–2018 by the MIT 6.172 Lecturers

Data Lab关键点

1. 位运算技巧
2. 补码的特点
3. 浮点数的表示方法

符号									
0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	-1
1	1	1	1	1	1	1	0	=	-2
1	0	0	0	0	0	0	1	=	-127
1	0	0	0	0	0	0	0	=	-128

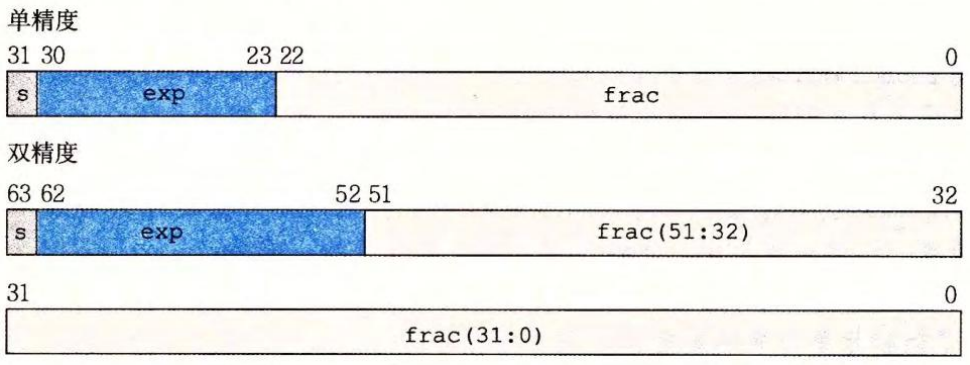


图 2-32 标准浮点格式(浮点数由 3 个字段表示。两种最常见的格式是它们被封装到 32 位(单精度)和 64 位(双精度)的字中)

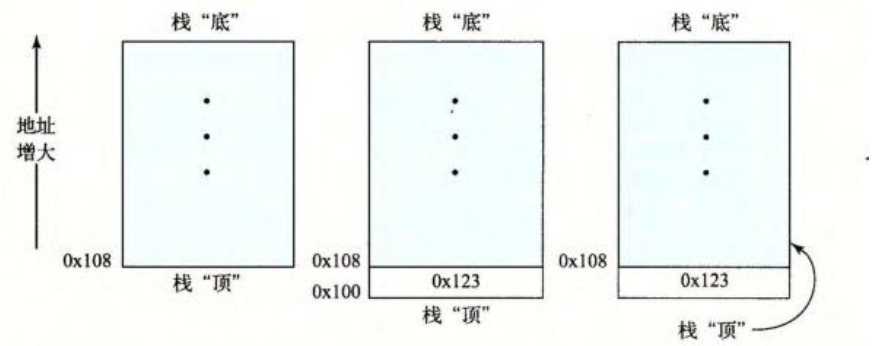
比较难的一个函数：二分法

```
int howManyBits(int x)
{
    int signMask = x >> 31;
    int y = (signMask ^ x); // bit nor only for negs
    int sum = 0;
    sum += !!(y & ((0xFF << 24) + (0xFF << 16))) << 4;
    sum += !!(y & ((0xFF << 8) << sum)) << 3;
    sum += !!(y & (0xF0 << sum)) << 2;
    sum += !!(y & (0xC << sum)) << 1; // C:1100
    sum += !!(y & (0x2 << sum));
    sum += !!(y & (0x1 << sum));
    return sum + 1;
}
```

Bomb Lab关键点

1. 参数是如何传递的
2. 变量在内存中如何组织
3. 汇编的基础知识

- test %reg,%reg是在干什么?
- 使用lea进行加法和乘法的组合运算



-9 栈操作说明。根据惯例，我们的栈是倒过来画的，因而栈“顶”在底部。x86-64 中，栈向低地址方向增长，所以压栈是减小栈指针(寄存器%rsp)的值，并将数据存放到内存中，而出栈是从内存中读数据，并增加栈指针的值

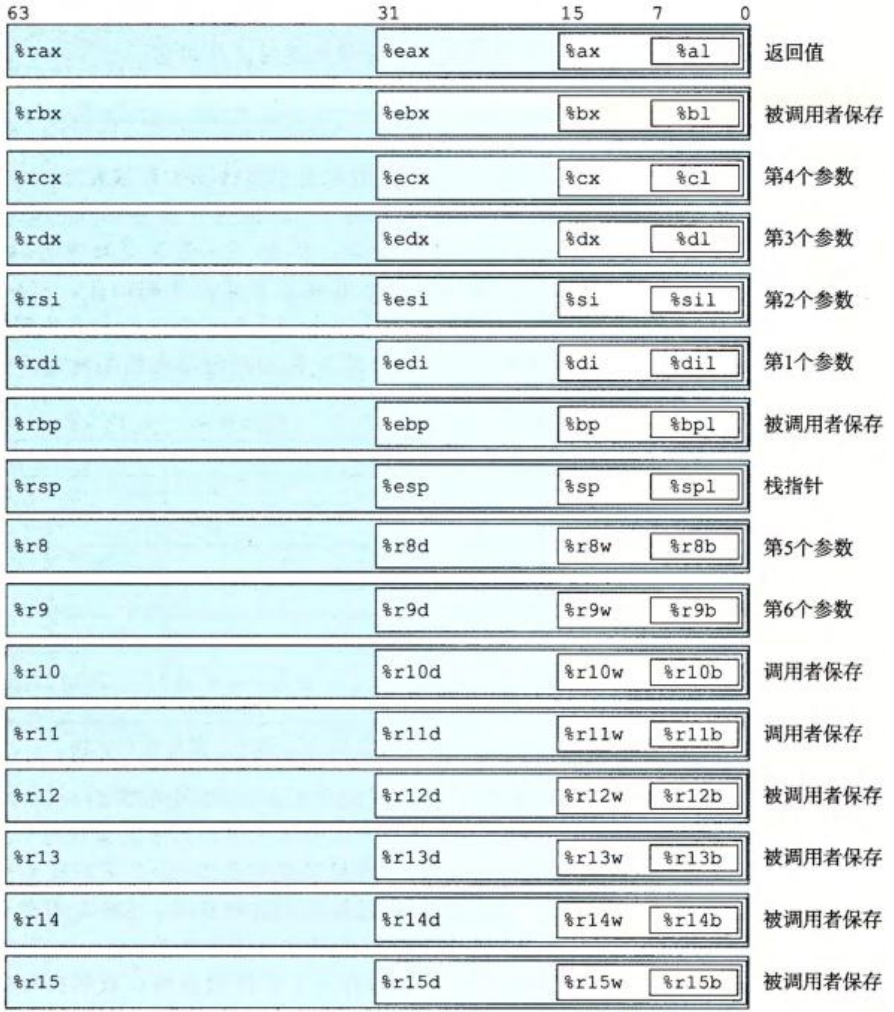
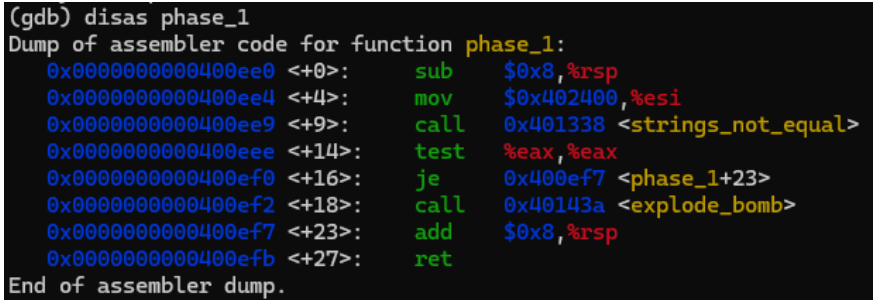


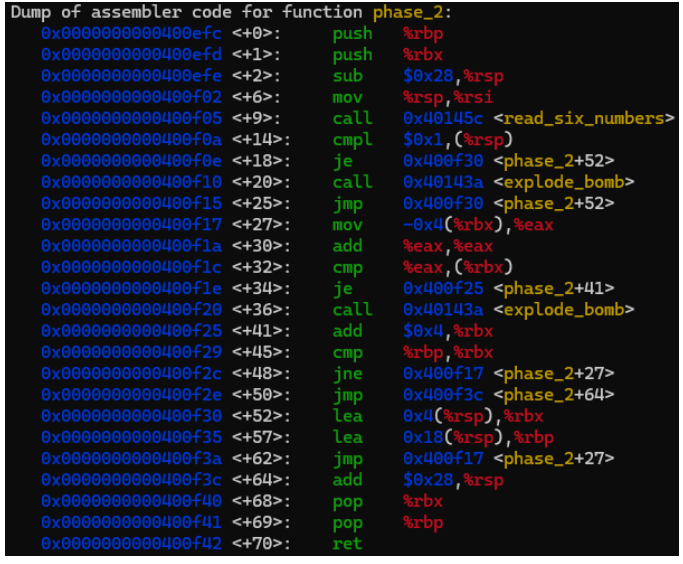
图 3-2 整数寄存器。所有 16 个寄存器的低位部分都可以作为字节、字(16 位)、双字(32 位)和四字(64 位)数字来访问

Phase 1

- strings_not_equal说明了一切。解法十分直接。
- Border relations with Canada have never been better.
- 输入了错误的Phase会发生什么?

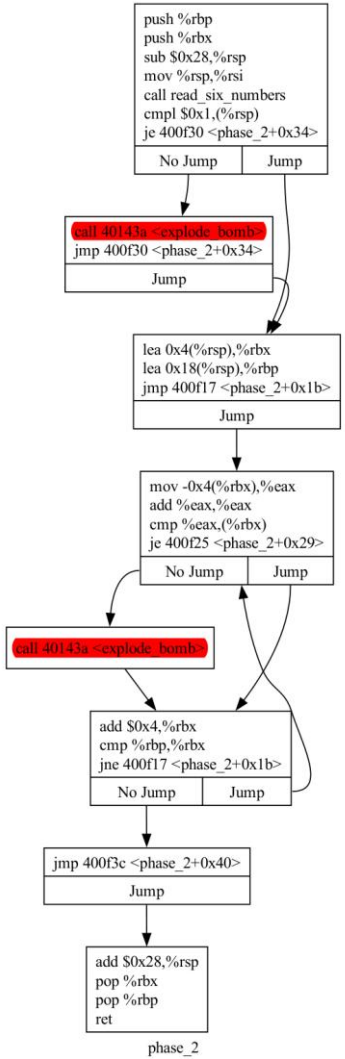


Phase 2-循环



```
Dump of assembler code for function phase_2:
   0x0000000000400efc <+0>:     push   %rbp
   0x0000000000400efd <+1>:     push   %rbx
   0x0000000000400efe <+2>:     sub    $0x28,%rsp
   0x0000000000400f02 <+6>:     mov    %rsp,%rsi
   0x0000000000400f05 <+9>:     call   0x40145c <read_six_numbers>
   0x0000000000400f0a <+14>:    cmpl   $0x1,(%rsp)
   0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
   0x0000000000400f10 <+20>:    call   0x40143a <explode_bomb>
   0x0000000000400f15 <+25>:    jmp    0x400f30 <phase_2+52>
   0x0000000000400f17 <+27>:    mov    -0x4(%rbx),%eax
   0x0000000000400f1a <+30>:    add    %eax,%eax
   0x0000000000400f1c <+32>:    cmp    %eax,(%rbx)
   0x0000000000400f1e <+34>:    je     0x400f25 <phase_2+41>
   0x0000000000400f20 <+36>:    call   0x40143a <explode_bomb>
   0x0000000000400f25 <+41>:    add    $0x4,%rbx
   0x0000000000400f29 <+45>:    cmp    %rbp,%rbx
   0x0000000000400f2c <+48>:    jne    0x400f17 <phase_2+27>
   0x0000000000400f2e <+50>:    jmp    0x400f3c <phase_2+64>
   0x0000000000400f30 <+52>:    lea    0x4(%rsp),%rbx
   0x0000000000400f35 <+57>:    lea    0x18(%rsp),%rbp
   0x0000000000400f3a <+62>:    jmp    0x400f17 <phase_2+27>
   0x0000000000400f3c <+64>:    add    $0x28,%rsp
   0x0000000000400f40 <+68>:    pop    %rbx
   0x0000000000400f41 <+69>:    pop    %rbp
   0x0000000000400f42 <+70>:    ret
```

- read_six_numbers 说 明 这 一 个 phase答案如何组成
- 需要略微了解程序运行时内存结构
- 注意汇编代码特征



梳理一下程序结构

注意图中的反向边，这说明了phase_2中存在循环结构

- 循环条件是什么?
- 循环体是什么样的?

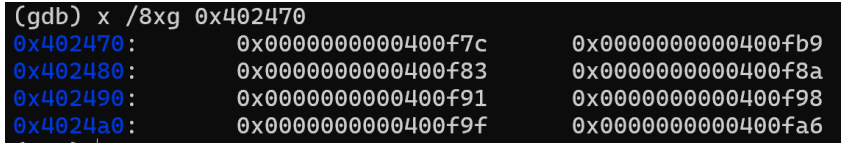
```
Dump of assembler code for function phase_3:
   0x0000000000400f43 <+0>:     sub    $0x18,%rsp
   0x0000000000400f47 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000400f4c <+9>:     lea    0x8(%rsp),%rdx
   0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
   0x0000000000400f56 <+19>:    mov    $0x0,%eax
   0x0000000000400f5b <+24>:    call   0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000400f60 <+29>:    cmp    $0x1,%eax
   0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>
   0x0000000000400f65 <+34>:    call   0x40143a <explode_bomb>
   0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
   0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
   0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
   0x0000000000400f75 <+50>:    jmp    *0x402470(,%rax,8)
   0x0000000000400f7c <+57>:    mov    $0xcf,%eax
   0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f83 <+64>:    mov    $0x2c3,%eax
   0x0000000000400f88 <+69>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f8a <+71>:    mov    $0x100,%eax
   0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f91 <+78>:    mov    $0x185,%eax
   0x0000000000400f96 <+83>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f98 <+85>:    mov    $0xce,%eax
   0x0000000000400f9d <+90>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f9f <+92>:    mov    $0x2aa,%eax
   0x0000000000400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400fa6 <+99>:    mov    $0x147,%eax
   0x0000000000400fab <+104>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fad <+106>:   call   0x40143a <explode_bomb>
   0x0000000000400fb2 <+111>:   mov    $0x0,%eax
   0x0000000000400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fb9 <+118>:   mov    $0x137,%eax
   0x0000000000400fbe <+123>:   cmp    0xc(%rsp),%eax
   0x0000000000400fc2 <+127>:   je     0x400fc9 <phase_3+134>
   0x0000000000400fc4 <+129>:   call   0x40143a <explode_bomb>
   0x0000000000400fc9 <+134>:   add    $0x18,%rsp
   0x0000000000400fcd <+138>:   ret
```

Phase 3-跳转表

- 注意jmp *，它似乎与我们之前见到的跳转指令不太一样
- 后面的mov...jmp..序列在干什么?

0x402470处是什么?

jmp *语句似乎没有指明需要跳转到哪里，0x402470(,%rax,8)=0x402470+%rax*8,具体的跳转位置似乎要根据rax寄存器的值得出，此外，0x402470处的内容似乎也非常重要。



0x402470处是什么?

联系到先前phase_3的反汇编代码，我们应该能从中看出一些对应关系：

- 0x400f7c=<phase_3+57>
- 0x400fb9=<phase_3+118>
- ……

```
(gdb) x /8xg 0x402470
0x402470:       0x0000000000400f7c      0x0000000000400fb9
0x402480:       0x0000000000400f83      0x0000000000400f8a
0x402490:       0x0000000000400f91      0x0000000000400f98
0x4024a0:       0x0000000000400f9f      0x0000000000400fa6
```

```
Dump of assembler code for function phase_3:
   0x0000000000400f43 <+0>:     sub    $0x18,%rsp
   0x0000000000400f47 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000400f4c <+9>:     lea    0x8(%rsp),%rdx
   0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
   0x0000000000400f56 <+19>:    mov    $0x0,%eax
   0x0000000000400f5b <+24>:    call   0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000400f60 <+29>:    cmp    $0x1,%eax
   0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>
   0x0000000000400f65 <+34>:    call   0x40143a <explode_bomb>
   0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
   0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
   0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
   0x0000000000400f75 <+50>:    jmp    *0x402470(,%rax,8)
   0x0000000000400f7c <+57>:    mov    $0xcf,%eax
   0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f83 <+64>:    mov    $0x2c3,%eax
   0x0000000000400f88 <+69>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f8a <+71>:    mov    $0x100,%eax
   0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f91 <+78>:    mov    $0x185,%eax
   0x0000000000400f96 <+83>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f98 <+85>:    mov    $0xce,%eax
   0x0000000000400f9d <+90>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f9f <+92>:    mov    $0x2aa,%eax
   0x0000000000400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400fa6 <+99>:    mov    $0x147,%eax
   0x0000000000400fab <+104>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fad <+106>:   call   0x40143a <explode_bomb>
   0x0000000000400fb2 <+111>:   mov    $0x0,%eax
   0x0000000000400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fb9 <+118>:   mov    $0x137,%eax
   0x0000000000400fbe <+123>:   cmp    0xc(%rsp),%eax
   0x0000000000400fc2 <+127>:   je     0x400fc9 <phase_3+134>
   0x0000000000400fc4 <+129>:   call   0x40143a <explode_bomb>
   0x0000000000400fc9 <+134>:   add    $0x18,%rsp
   0x0000000000400fcd <+138>:   ret
```

Phase 4-递归

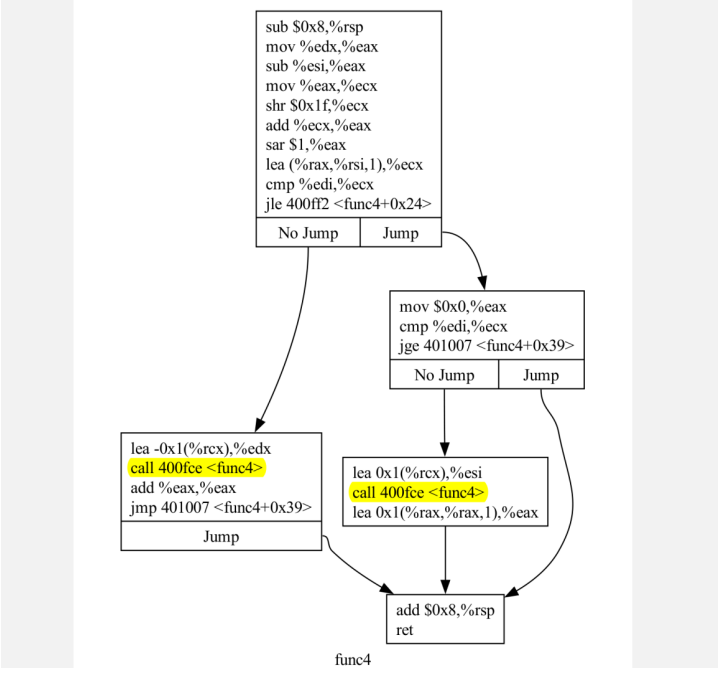
- 函数结构看起来似乎很简单……吗?

```
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>:    call   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>:    call   0x40143a <explode_bomb>
   0x000000000040103a <+46>:    mov    $0xe,%edx
   0x000000000040103f <+51>:    mov    $0x0,%esi
   0x0000000000401044 <+56>:    mov    0x8(%rsp),%edi
   0x0000000000401048 <+60>:    call   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>:    call   0x40143a <explode_bomb>
   0x000000000040105d <+81>:    add    $0x18,%rsp
   0x0000000000401061 <+85>:    ret
End of assembler dump.
```

```
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    $1,%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>:    call   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>:    call   0x400fce <func4>
   0x0000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401007 <+57>:    add    $0x8,%rsp
   0x000000000040100b <+61>:    ret
End of assembler dump.
```

Phase 4-递归

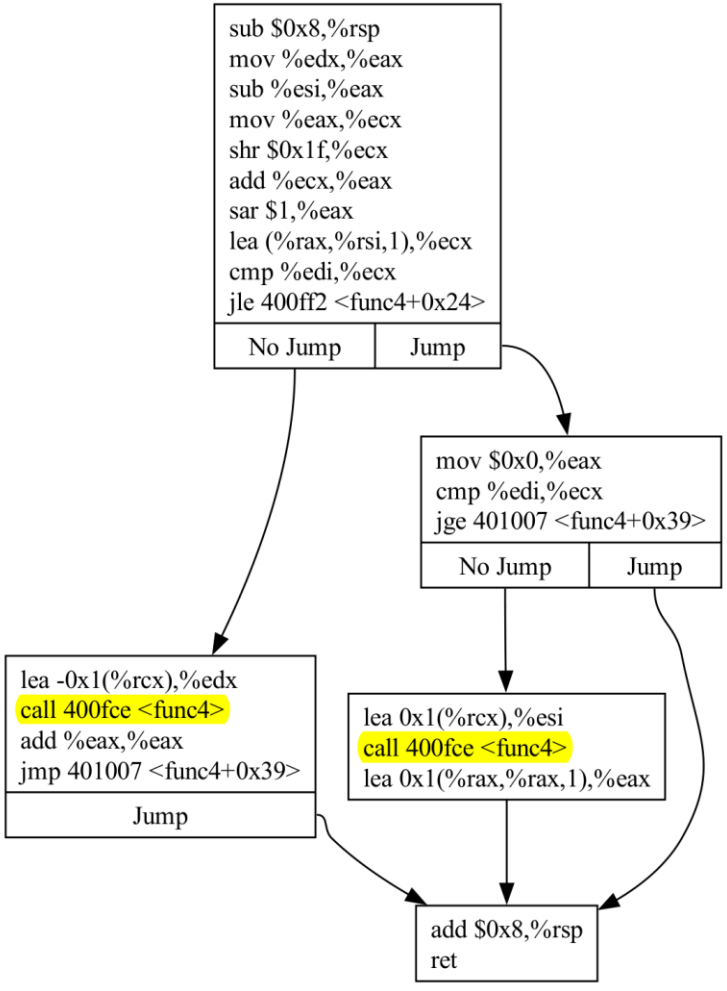
- 函数结构看起来似乎很简单……吗?
- 递归的出现让函数的分析变得较为棘手




```
Dump of assembler code for function func4:
   0x00000000000400fce <+0>:     sub    $0x8,%rsp
   0x00000000000400fd2 <+4>:     mov    %edx,%eax
   0x00000000000400fd4 <+6>:     sub    %esi,%eax
   0x00000000000400fd6 <+8>:     mov    %eax,%ecx
   0x00000000000400fd8 <+10>:    shr    $0x1f,%ecx
   0x00000000000400fdb <+13>:    add    %ecx,%eax
   0x00000000000400fdd <+15>:    sar    $1,%eax
   0x00000000000400fdf <+17>:    lea    (%rax,%rsi,1),%ecx
   0x00000000000400fe2 <+20>:    cmp    %edi,%ecx
   0x00000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
   0x00000000000400fe6 <+24>:    lea    -0x1(%rcx),%edx
   0x00000000000400fe9 <+27>:    call   0x400fce <func4>
   0x00000000000400fee <+32>:    add    %eax,%eax
   0x00000000000400ff0 <+34>:    jmp    0x401007 <func4+57>
   0x00000000000400ff2 <+36>:    mov    $0x0,%eax
   0x00000000000400ff7 <+41>:    cmp    %edi,%ecx
   0x00000000000400ff9 <+43>:    jge    0x401007 <func4+57>
   0x00000000000400ffb <+45>:    lea    0x1(%rcx),%esi
   0x00000000000400ffe <+48>:    call   0x400fce <func4>
   0x00000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x00000000000401007 <+57>:    add    $0x8,%rsp
   0x0000000000040100b <+61>:    ret
End of assembler dump.
```

Phase 4-递归

- 函数结构看起来似乎很简单……吗?
- 递归的出现让函数的分析变得较为棘手
- 我们用程序控制流图分析一下func4在做什么



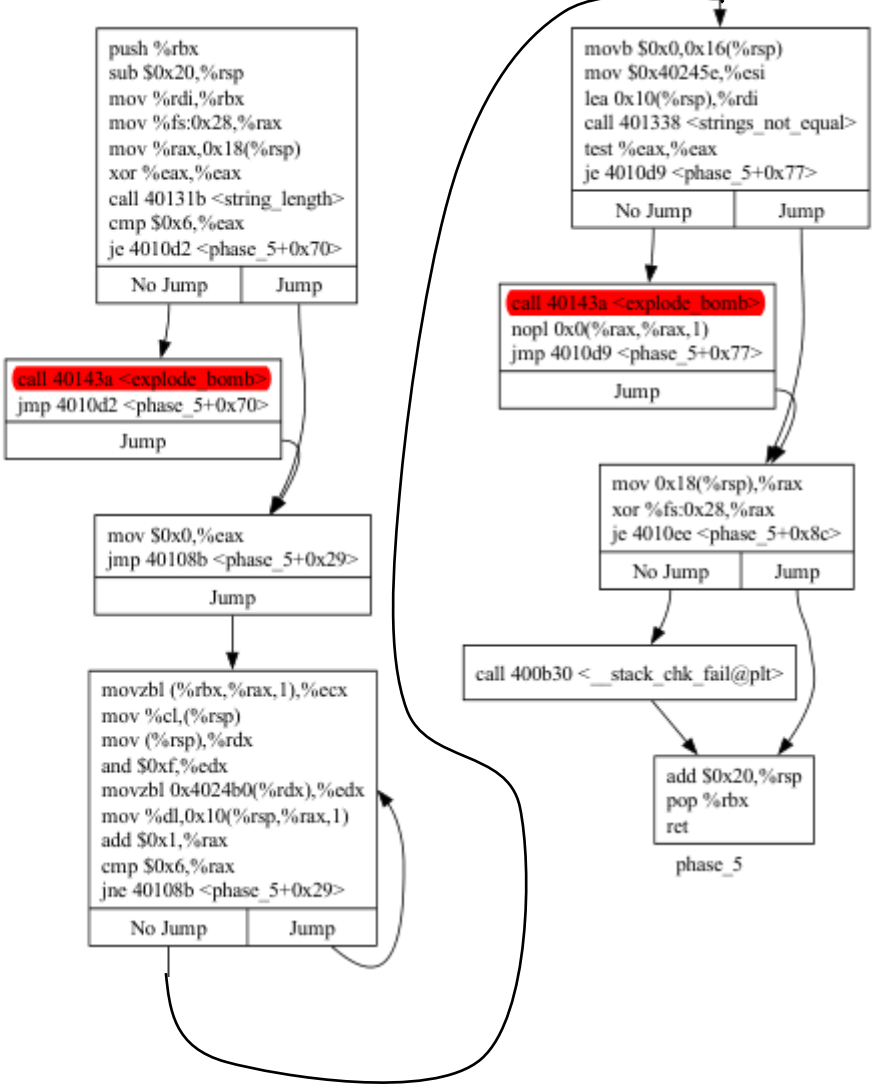
Phase 5-索引

- string_length提示输入为字符串，且对这个字符串长度有一定要求

```
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>:    call   0x40131b <string_length>
   0x000000000040107f <+29>:    cmp    $0x6,%eax
   0x0000000000401082 <+32>:    je     0x4010d2 <phase_5+112>
   0x0000000000401084 <+34>:    call   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>:    call   0x401338 <strings_not_equal>
   0x00000000004010c2 <+96>:    test   %eax,%eax
   0x00000000004010c4 <+98>:    je     0x4010d9 <phase_5+119>
   0x00000000004010c6 <+100>:   call   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>:   call   0x400b30 <__stack_chk_fail@plt>
   0x00000000004010ee <+140>:   add    $0x20,%rsp
   0x00000000004010f2 <+144>:   pop    %rbx
   0x00000000004010f3 <+145>:   ret
End of assembler dump.
```

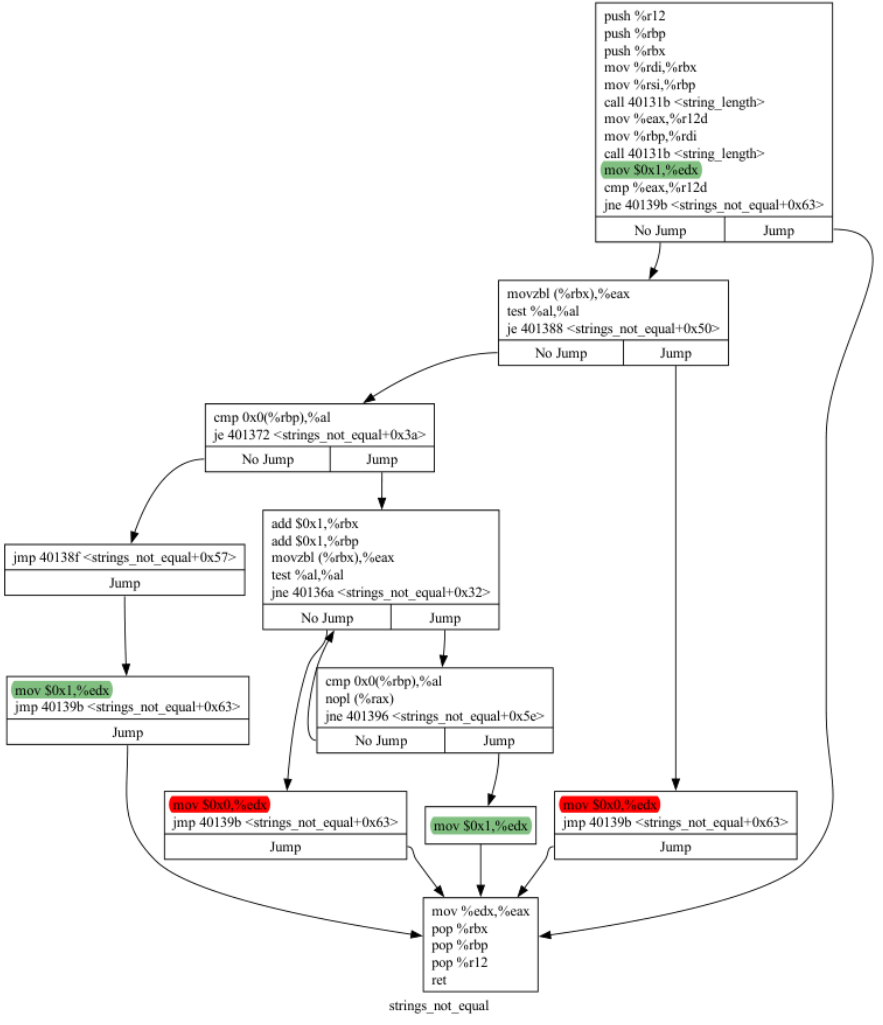
Phase 5-索引

- 字符串长度必须为6
- 需要让strings_not_equal返回1
- 把某些东西放到0x10(%rsp)中并将其传递给了strings_not_equal



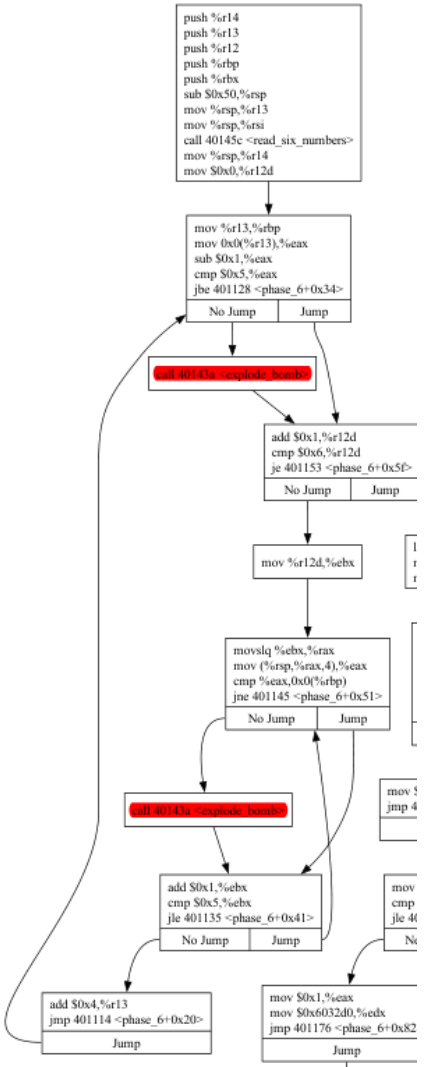
Phase 5-strings_not_equal

- 哪些情况使得其返回0?
- 哪些情况返回1？（这是我们想要的）



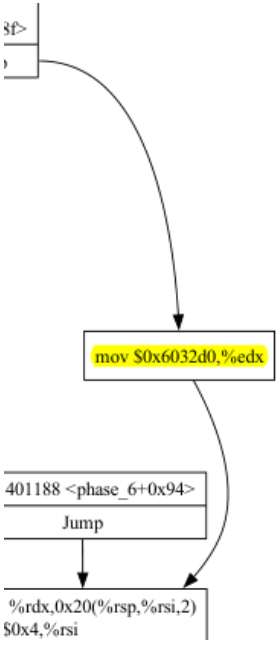
Phase 6-链表

- 很长
- 拆分成几部分来分析
- 有没有什么地方存储了一些值为我们提供线索?
- 右图为输入字串要求



Phase 6-链表

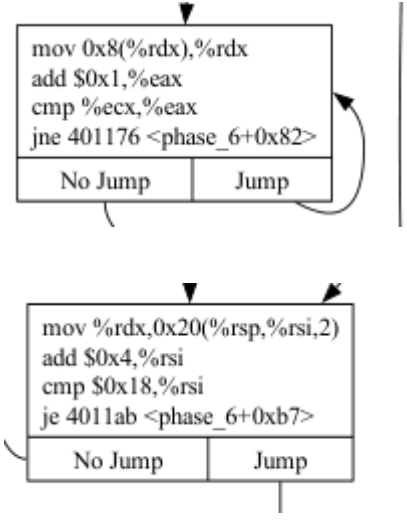
- 很长
- 拆分成几部分来分析
- 有没有什么地方存储了一些值为我们提供线索?
- 右图为输入字串要求
- 我们怎么查看这个内存地址位置的内容呢?




```
(gdb) x /??? 0x6032d0
```

Phase 6-链表

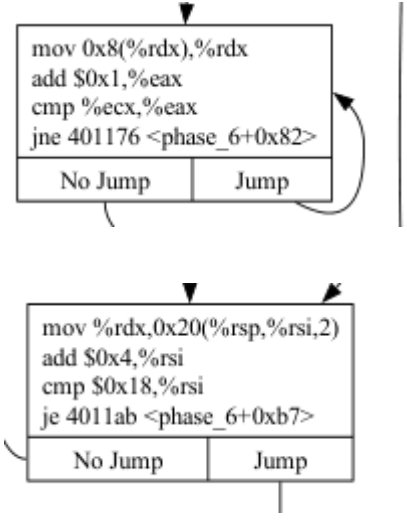
- 很长
- 拆分成几部分来分析
- 有没有什么地方存储了一些值为我们提供线索?
- 右图为输入字串要求
- 我们怎么查看这个内存地址位置的内容呢?
- 两个地方似乎都是以8字节为单位传递的，所以/?xg




```
(gdb) x /8xg 0x6032d0
0x6032d0 <node1>:       0x000000010000014c      0x00000000006032e0
0x6032e0 <node2>:       0x00000002000000a8      0x00000000006032f0
0x6032f0 <node3>:       0x000000030000039c      0x0000000000603300
0x603300 <node4>:       0x00000004000002b3      0x0000000000603310
```

Phase 6-链表

- 很长
- 拆分成几部分来分析
- 有没有什么地方存储了一些值为我们提供线索?
- 右图为输入字串要求
- 我们怎么查看这个内存地址位置的内容呢?
- 两个地方似乎都是以8字节为单位传递的，所以/?xg
- 后面的字段似乎是地址?




```
(gdb) x /8xg 0x6032d0
0x6032d0 <node1>:       0x000000010000014c      0x00000000006032e0
0x6032e0 <node2>:       0x00000002000000a8      0x00000000006032f0
0x6032f0 <node3>:       0x000000030000039c      0x0000000000603300
0x603300 <node4>:       0x00000004000002b3      0x0000000000603310
```

Phase 6-链表

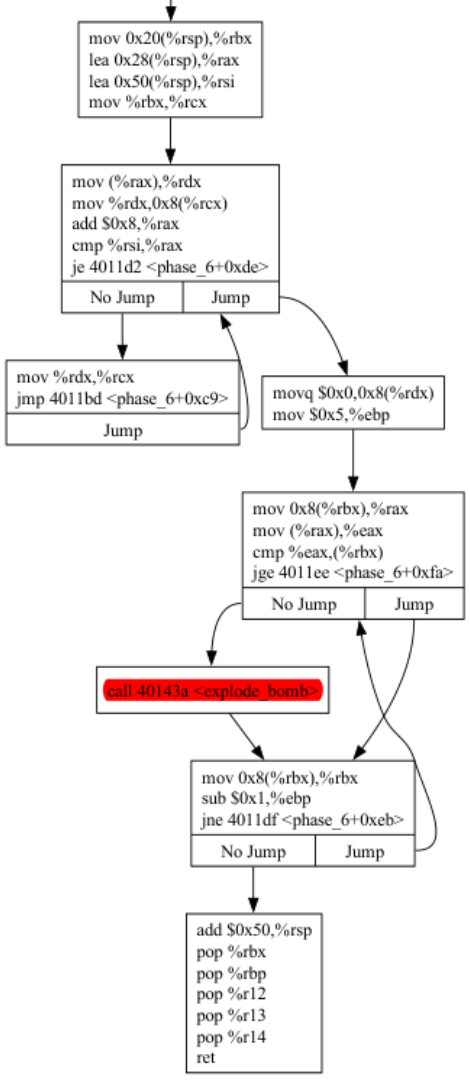
```
(gdb) x /13xg 0x6032d0
0x6032d0 <node1>:       0x000000010000014c      0x00000000006032e0
0x6032e0 <node2>:       0x00000002000000a8      0x00000000006032f0
0x6032f0 <node3>:       0x000000030000039c      0x0000000000603300
0x603300 <node4>:       0x00000004000002b3      0x0000000000603310
0x603310 <node5>:       0x00000005000001dd      0x0000000000603320
0x603320 <node6>:       0x00000006000001bb      0x0000000000000000
0x603330:       0x0000000000000000
```

- 很长
- 拆分成几部分来分析
- 有没有什么地方存储了一些值为我们提供线索?
- 右图为输入字串要求
- 我们怎么查看这个内存地址位置的内容呢?
- 两个地方似乎都是以8字节为单位传递的，所以/?xg
- 后面的字段似乎是地址?
- 根据链表的特点重新分析

```
struct node{
        int num;
        int ???;
        struct node *next;
};
```

Phase 6-链表

- 很长
- 拆分成几部分来分析
- 有没有什么地方存储了一些值为我们提供线索?
- 右图为输入字串要求
- 我们怎么查看这个内存地址位置的内容呢?
- 两个地方似乎都是以8字节为单位传递的，所以/?xg
- 后面的字段似乎是地址?
- 根据链表的特点重新分析
- 这里的循环告诉我们最终的输入顺序



结束了……吗?

让我们仔细看看objdump输出的反汇编代码，有一个我们似乎没有注意到的函数：secret_phase

```
0000000000401242 <secret_phase>:
  401242: 53                    push   %rbx
  401243: e8 56 02 00 00        call   40149e <read_line>
  401248: ba 0a 00 00 00        mov    $0xa,%edx
  40124d: be 00 00 00 00        mov    $0x0,%esi
  401252: 48 89 c7              mov    %rax,%rdi
  401255: e8 76 f9 ff ff        call   400bd0 <strtol@plt>
  40125a: 48 89 c3              mov    %rax,%rbx
  40125d: 8d 40 ff              lea    -0x1(%rax),%eax
  401260: 3d e8 03 00 00        cmp    $0x3e8,%eax
  401265: 76 05                 jbe    40126c <secret_phase+0x2a>
  401267: e8 ce 01 00 00        call   40143a <explode_bomb>
  40126c: 89 de                 mov    %ebx,%esi
  40126e: bf f0 30 60 00        mov    $0x6030f0,%edi
  401273: e8 8c ff ff ff        call   401204 <fun7>
  401278: 83 f8 02              cmp    $0x2,%eax
  40127b: 74 05                 je     401282 <secret_phase+0x40>
  40127d: e8 b8 01 00 00        call   40143a <explode_bomb>
  401282: bf 38 24 40 00        mov    $0x402438,%edi
  401287: e8 84 f8 ff ff        call   400b10 <puts@plt>
  40128c: e8 33 03 00 00        call   4015c4 <phase_defused>
  401291: 5b                    pop    %rbx
  401292: c3                    ret
  401293: 90                    nop
  401294: 90                    nop
  401295: 90                    nop
  401296: 90                    nop
  401297: 90                    nop
  401298: 90                    nop
  401299: 90                    nop
  40129a: 90                    nop
  40129b: 90                    nop
  40129c: 90                    nop
  40129d: 90                    nop
  40129e: 90                    nop
  40129f: 90                    nop
```

参考资料

本PPT参考了如下资料：

上海交通大学并行与分布式系统研究所：ICS-tutorial-5-asm

CMU15213 Introduction to Computer Systems

MIT6.172 Performance Engineering of Software Systems

TO BE CONTINUED

Copyright © 2021 ECNU Corporation. All rights reserved.
Tel:+86-021-62233586 Fax:+86-021-62606775
E-mail: ecnu@ecnu.com.cn Http: //www.ecnu.edu.cn