概览

首先我们从cmu15-213的官网 下载实验文件压缩包, 打开后我们看到有3个文件：

```
root@LAPTOP-RI965SV0:/mnt/d/bomb# ls
README  bomb  bomb.c
```

其中`bomb.c`为c源文件，`bomb`为其的object file。

打开`bomb.c`我们可以看到：

```
/* 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");
```

我们需要输入一个`input`，然后函数`phase_1`会将`input`作为参数并执行， 如果我们的`input`是`phase_1`所想要的输入，那么该炸弹即可被拆除。那么我们的任务便是找出`phase_1`的正确输入是什么。

实验开始

拆除第一个炸弹：

我们首先进入`gdb`：

```
root@LAPTOP-RI965SV0:/mnt/d/bomb# gdb bomb
```

然后反汇编`phase_1`：

```
(gdb) disassemble phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ee0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ee4 <+4>:     mov    $0x402400,%esi
   0x0000000000400ee9 <+9>:     call   0x401338 <strings_not_equal>
   0x0000000000400eee <+14>:    test   %eax,%eax
   0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
   0x0000000000400ef2 <+18>:    call   0x40143a <explode_bomb>
   0x0000000000400ef7 <+23>:    add    $0x8,%rsp
   0x0000000000400efb <+27>:    ret
End of assembler dump.
```

我们逐句解析这段汇编代码：

- `sub $0x8,%rsp`：栈指针减8，分配栈上的存储空间
- `mov $0x402400, %esi`：把0x402400 放到`%esi`，参考教材P245， `%esi`是放函数参数的。
- `call 0x401338<strings_not_equal>` ：调用函数`strings_not_equal`，显然这是一个判断字符串是否相同的函数， 函数返回值放到了`%eax`.

- `test %eax,%eax`
- `je <phase_1+23>`：如果`%eax`为0， 跳转到目标地址
- `call <explode_bomb>` ： 如果上一步不跳转，引爆炸断
- `add $0x8,%rsp` 恢复栈指针

经过上述分析，结果已经很明显了，我们只需去看`0x402400`处存的是什么即可。

使用命令`x/s`：

```
(gdb) x/s 0x402400
0x402400:       "Border relations with Canada have never been better."
```

这便是我们需要的答案。

拆除第二个炸弹

同样先反汇编`phase_2`：

```
(gdb) disassemble phase_2
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
End of assembler dump.
```

- 想看前三行：`push %rbq`，`push %rbx`，`sub $0x28,%rsp`是对栈的操作。
  - `mov %rsp,%rsi`把栈指针作为函数参数，随后调用`read_six_numbers`， 看名字应该是读入六个数。
- `cmpl $0x1, (%rsp)` ：看到这里，我们发现如果栈指针不为1，那么炸弹爆炸
- 我们现在从跳转到`0x400f30`开始看：

  首先把`0x4(%rsp)`拿给`%rbx`，把`0x18(%rsp)`拿给`%rbp`。

然后直接跳转到 `0x400f17`：

把 `-0x4(%rbx)` 拿给 `%eax`，随后 `%eax` 乘2， 下面是一个是否相等的判断，如果不等，炸弹爆炸，如果相等，又跳到 `0x400f25`：

更新 `%rbx` 为 `4+%rbx`，然后判断 `%rbx` 和 `%rbp` 是否相等，不相等的话就跳到 `0x400f17`，又开启一次新的循环。

我们来梳理一下上面这个流程：

初始时：`%rps=1 ， %rbx=0x4+%rsp`，`%rbp = 0x18+%rsp`，注意 `0x18 = 24`(妈的，差点搞错了)

然后对于每次循环，`%rbx+=0x4`，继续条件：`%rbx!=rsp`

循环内部： 比较两个数： `(%rbx)`，`0x2 * (%rbx-0x4)`，如果这两个数不相等那么炸弹爆炸， 也就是说，我们输入的数（存在栈上面），满足后一个是前一个的两倍。 然后初始值又为1： 这六个数为1,2,4,8,16,32

- 现在我们可以判断，函数 `read_six_numbers` 就是给我们所需要的栈上的那几个位置赋值的。也就是说，我们输入的6个数字会对应地赋值到栈上去。
- 下面我们看看 `read_six_numbers` 干了什么。

```
(gdb) disassemble read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x000000000040145c <+0>:     sub    $0x18,%rsp
   0x0000000000401460 <+4>:     mov    %rsi,%rdx
   0x0000000000401463 <+7>:     lea    0x4(%rsi),%rcx
   0x0000000000401467 <+11>:    lea    0x14(%rsi),%rax
   0x000000000040146b <+15>:    mov    %rax,0x8(%rsp)
   0x0000000000401470 <+20>:    lea    0x10(%rsi),%rax
   0x0000000000401474 <+24>:    mov    %rax,(%rsp)
   0x0000000000401478 <+28>:    lea    0xc(%rsi),%r9
   0x000000000040147c <+32>:    lea    0x8(%rsi),%r8
   0x0000000000401480 <+36>:    mov    $0x4025c3,%esi
   0x0000000000401485 <+41>:    mov    $0x0,%eax
   0x000000000040148a <+46>:    call   0x400bf0 <__isoc99_sscanf@plt>
   0x000000000040148f <+51>:    cmp    $0x5,%eax
   0x0000000000401492 <+54>:    jg     0x401499 <read_six_numbers+61>
   0x0000000000401494 <+56>:    call   0x40143a <explode_bomb>
   0x0000000000401499 <+61>:    add    $0x18,%rsp
   0x000000000040149d <+65>:    ret
End of assembler dump.
```

我们先关注前面几行， 出现了这么几个元素：

`%rdx`，`%rcx`，`%rax`，`0x8(%rsp)`，`(%rsp)` `%r9`，`%r8`， 我们可以判断这是在分配参数的存储空间。

拆除第三个炸弹

```
(gdb) disassemble phase_3
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
End of assembler dump.
```

- `x/s $4025cf` 查看 `scanf` 的参数

```
(gdb) x/s 0x4025cf
0x4025cf:       "%d %d"
```

也就是说我们需要输入两个数字。

- 从 `cmpl $0x7, 0x8(%rsp), ja` 知： 第一个数必须小于等于7.
-

```
mov     0x8(%rsp),%eax
jmp     *0x402470(,%rax,8)
mov     $0xcf,%eax
```

把第一个数拿给 `%eax`， `jmp*` 的跳转地址为 `(%rax*8 + 0x402470)`，就是第一个数乘8 + 0x402470 所在位置存储的数字。

- 如果`0x8(%rsp) = 1`，那么 跳转地址为`(0x402478)`，我们查看一下这个位置：

```
(gdb) x/x 0x402478
0x402478:       0xb9
```

- 我们去看一下`0x400fb9`这个位置：

  得到`0xc(%rsp) = 0x137 = 311`，就是我们第二个需要输入的数。

拆除第四个炸弹

```
(gdb) disassemble 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>:    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.
```

与上面两题类似，我们通过`scanf`输入两个数， 其分别存放在`0x8(%rsp), 0xc(%rsp)`上。

- ```
  0x0000000000401051 <+69>:    cmpl   $0x0,0xc(%rsp)
  0x0000000000401056 <+74>:    je     0x40105d <phase_4+81>
  0x0000000000401058 <+76>:    call   0x40143a <explode_bomb>
  0x000000000040105d <+81>:    add    $0x18,%rsp
  ```

  从这里可以看出第二个数必须为0.

- ```
  cmp    $0x2,%eax
  jne    0x401035 <phase_4+41>
  ```

  此处表明第一个数必须小于等于`0xe`，随后跳转到`0x4013a`

-

```
mov    $0xe,%edx
mov    $0x0,%esi
mov    0x8(%rsp),%edi
call   0x400fce <func4>
test   %eax,%eax
jne    0x401058 <phase_4+76>
```

此处是调用函数`func4`的过程，其形式如下`func4(num,0,0xe)`，其中`num`是我们需要输入的第一个数字。

由`test`可知：`func4`的输出必须为0.

- 下面我们研究`func4`：

```
(gdb) disassemble func4
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>:    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.
```

我们看到在`func4`内部还调用了自己，判断这是一个递归函数。下面尝试逐句翻译成c语言。

```c
int func4(int edi,int esi, int edx){
    int eax =  edx - esi  ;
    int ecx = eax >> (0x1f);
    eax += ecx;
    eax =eax>>1;
    ecx = eax + esi;

    if (ecx>edi){
```

```
        return 2* func4(edi,esi,ecx-1);
    }
    if (ecx<edi){
        return 2*func4(edi,ecx+1,edx)+1;
    }
    return 0;

}
```

- 第一个参数为多少时，`func4` 的输出为0?

```
#include <iostream>
using namespace std;
int main(){
    for(int i=0;i<=14;i++){
        if (func4(i,0,14) ==0) cout<<i<<endl;
    }
    return 0;
}
```

- 最终结果为`0,1,3,7`，所以我们输入`0 0; 1 0; 3 0; 7 0`均可以。

拆除第五个炸弹

```
(gdb) disassemble phase_5
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.
```

我们一段一段地分析：

- ```
  call   0x40131b <string_length>
  cmp    $0x6,%eax
  ```

表明我们输入的字符串长度必须是6

-
  ```

```
<+41>:    movzbl (%rbx,%rax,1),%ecx
<+45>:    mov    %cl,(%rsp)
<+48>:    mov    (%rsp),%rdx
<+52>:    and    $0xf,%edx
<+55>:    movzbl 0x4024b0(%rdx),%edx
<+62>:    mov    %dl,0x10(%rsp,%rax,1)
<+66>:    add    $0x1,%rax
<+70>:    cmp    $0x6,%rax
<+74>:    jne    0x40108b <phase_5+41>
```

这段代码明显是一个循环， 推出循环的条件是`%rax == 6`

由最开头我们知道`%rbx=%rdi`, 因此`%rbx`应该为一个指向字符串的指针。

那么`%ecx = (%rbx+%rax)`, 并且`%rax`初始值为0， 因此不难判断`%ecx`存的是字符串中的单个字符。

`%cl`为`%ecx`的低八位，然后把`%cl`拿给`%edx`.

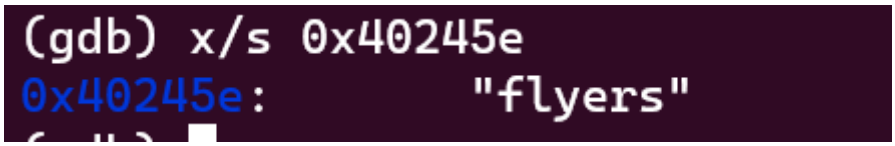
`movzbl 0x4024b0(%rdx), %edx` ： 把地址`0x4024b0(%rdx)`存的东西拿给`%edx`, 再把其第四位拿到栈上`0x10(%rsp,%rax,1)` (什么玩意儿)， 随后`%rax`增加1，开启下一次循环。

我们先不管这段代码干了什么，反正我们清楚现在栈上有几个新添的东西 （`0x4024b0(%rdx)`处存的东西），它们原来的地址和我们输入的字符串中的每个字符的ascii码有关系。

- 下面我们研究退出循环后的行为：

```
movb   $0x0,0x16(%rsp)
mov    $0x40245e,%esi
lea    0x10(%rsp),%rdi
call   0x401338 <strings_not_equal>
test   %eax,%eax
je     0x4010d9 <phase_5+119>
call   0x40143a <explode_bomb>
nopl   0x0(%rax,%rax,1)
jmp    0x4010d9 <phase_5+119>
mov    $0x0,%eax
jmp    0x40108b <phase_5+41>
mov    0x18(%rsp),%rax
xor    %fs:0x28,%rax
je     0x4010ee <phase_5+140>
call   0x400b30 <__stack_chk_fail@plt>
add    $0x20,%rsp
pop    %rbx
ret
```

我们又发现了这个`strings_not_equal`函数， 它的参数之一为`(0x40245e)`，我们去看一下这是个什么东西：



也就是说，我们放到栈上的东西就应该是这么一串字符。

我们先看看`0x4024b0`是什么：

```
(gdb) x/s 0x4024b0
0x4024b0 <array.3449>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

因此`%rdx`相当于这个字符串的索引。我们找到`f,l,y,e,r,s`对应的索引应该是什么。(从0开始)

`f:9，l:15，y:14，e:5，r:6，s: 7`，注意我们只有4个bits，其能表示的最大值为15

那么也就是说，我们输入的6个字符，其ASCII码的低四位所代表的数字为`1001，1111,1110，0101，0110，0111`

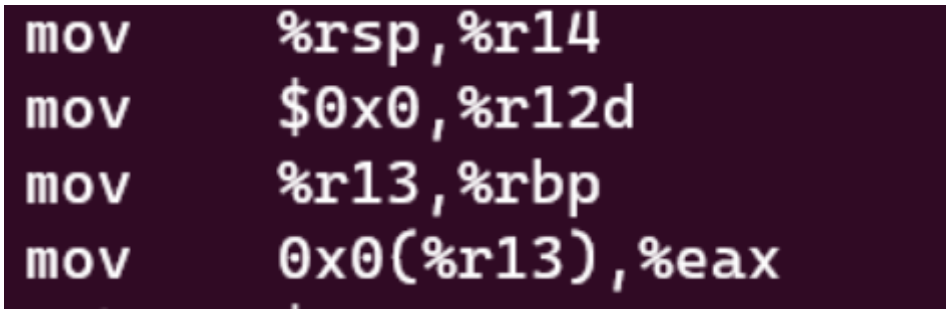
这样的答案不唯一：`ionuvw`，`)/>5&7`等均为正确答案。

拆除第六个炸弹

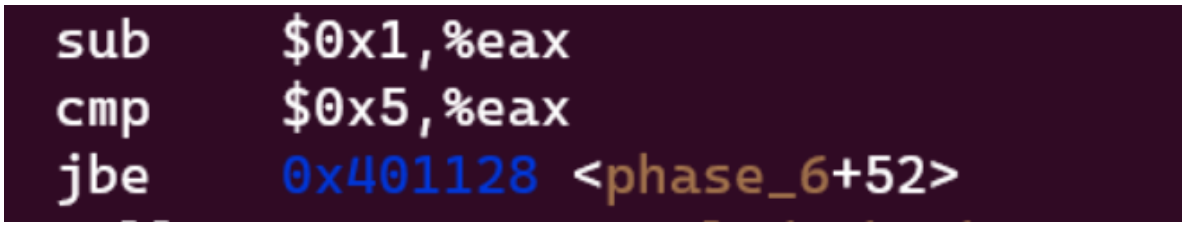
```
(gdb) disassemble phase_6
Dump of assembler code for function phase_6:
   0x00000000004010f4 <+0>:     push   %r14
   0x00000000004010f6 <+2>:     push   %r13
   0x00000000004010f8 <+4>:     push   %r12
   0x00000000004010fa <+6>:     push   %rbp
   0x00000000004010fb <+7>:     push   %rbx
   0x00000000004010fc <+8>:     sub    $0x50,%rsp
   0x0000000000401100 <+12>:    mov    %rsp,%r13
   0x0000000000401103 <+15>:    mov    %rsp,%rsi
   0x0000000000401106 <+18>:    call   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>:    call   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>:    call   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>:   call   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>:   ret
End of assembler dump.
```

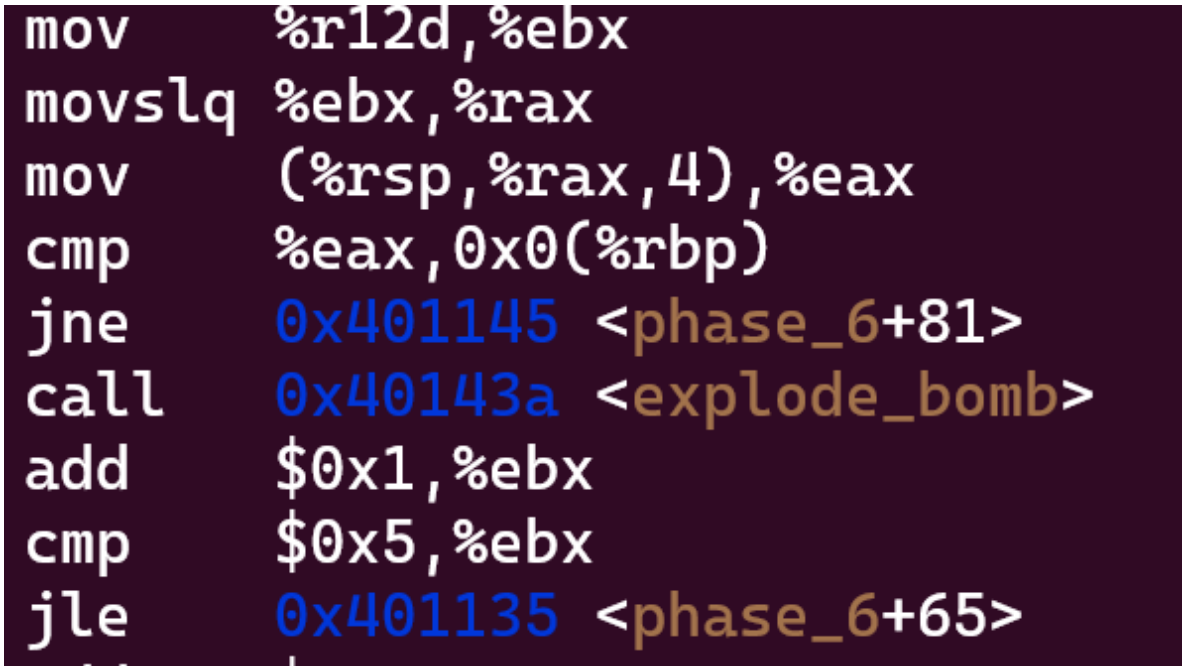
我们一点一点从上往下看吧：

- 首先我们看到`read_six_numbers`，表明这次依然读入6个数，然后把它们按照输入的顺序放到栈上去。记为`nums[i], (i=0,1,2,3,4,5)`
- 

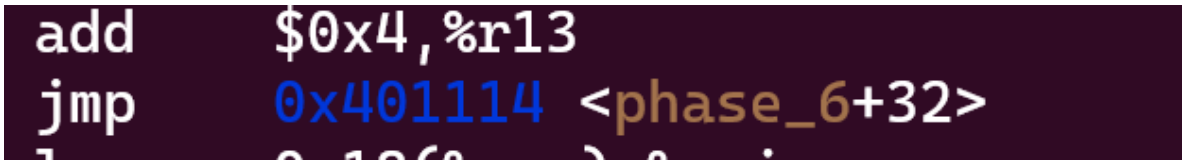
  取出栈上的第一个数`nums[0]`，放到`%eax`.
-



如果`nums[0]-1<=5`，跳到`+52`，否则爆炸。

- `%r12d`之前为0，现在是加1，随后如果`%r12d==6`，跳到`+95`，这应该是循环的判断条件，`%r12d`就是循环量`i`，下面我们看循环内部逻辑。
- 

`mov (%rsp,%rax,4)`把`nums[i]`取出来，拿给`%eax`. 然后由最开始知道，`%rbp`是指针的初始位置，那么就是在判断`nums[i]!=nums[0]`，若不成立便爆炸，随后再次进入循环。

- 

把`%r13`加4后再次进入外层循环，到此为止，这段代码我们已经清楚了。简单来说就是两层循环，外面那层判断每个`nums[i]`不大于`6`，里面那层判断两两`nums[i],nums[j]`不能相等。

- 接着往下看：

```
lea     0x18(%rsp),%rsi
mov     %r14,%rax
mov     $0x7,%ecx
mov     %ecx,%edx
sub     (%rax),%edx
mov     %edx,(%rax)
add     $0x4,%rax
cmp     %rsi,%rax
jne     0x401160 <phase_6+108>
```

`0x18(%rsp)` 是 `nums[5]`，刚开始`%r14` 为 `%rsp` 的初始值， 比较容易看出，这段代码就是把每个 `nums[i]` 换成`7-nums[i]`

- ```
  <+123>:    mov     $0x0,%esi
   q to quit, c to continue without paging--c
  <+128>:    jmp     0x401197 <phase_6+163>
  <+130>:    mov     0x8(%rdx),%rdx
  <+134>:    add     $0x1,%eax
  <+137>:    cmp     %ecx,%eax
  <+139>:    jne     0x401176 <phase_6+130>
  <+141>:    jmp     0x401188 <phase_6+148>
  <+143>:    mov     $0x6032d0,%edx
  <+148>:    mov     %rdx,0x20(%rsp,%rsi,2)
  <+153>:    add     $0x4,%rsi
  <+157>:    cmp     $0x18,%rsi
  <+161>:    je      0x4011ab <phase_6+183>
  <+163>:    mov     (%rsp,%rsi,1),%ecx
  <+166>:    cmp     $0x1,%ecx
  <+169>:    jle     0x401183 <phase_6+143>
  <+171>:    mov     $0x1,%eax
  <+176>:    mov     $0x6032d0,%edx
  <+181>:    jmp     0x401176 <phase_6+130>
  ```

我们尝试解析这段代码:

- 现在我们首先去看在`0x6032d0` 处存的是什么?
  ```

```
(gdb) x/x 0x6032d0
0x6032d0 <node1>:       0x0000014c
(gdb) x/18x 0x6032d0
0x6032d0 <node1>:       0x0000014c      0x00000001      0x006032e0      0x00000000
0x6032e0 <node2>:       0x000000a8      0x00000002      0x006032f0      0x00000000
0x6032f0 <node3>:       0x0000039c      0x00000003      0x00603300      0x00000000
0x603300 <node4>:       0x000002b3      0x00000004      0x00603310      0x00000000
0x603310 <node5>:       0x000001dd      0x00000005
(gdb) x/50 0x6032d0
0x6032d0 <node1>:       0x0000014c      0x00000001      0x006032e0      0x00000000
0x6032e0 <node2>:       0x000000a8      0x00000002      0x006032f0      0x00000000
0x6032f0 <node3>:       0x0000039c      0x00000003      0x00603300      0x00000000
0x603300 <node4>:       0x000002b3      0x00000004      0x00603310      0x00000000
0x603310 <node5>:       0x000001dd      0x00000005      0x00603320      0x00000000
0x603320 <node6>:       0x000001bb      0x00000006      0x00000000      0x00000000
0x603330:       0x00000000      0x00000000      0x00000000      0x00000000
0x603340 <host_table>:  0x00402629      0x00000000      0x00402643      0x00000000
0x603350 <host_table+16>:       0x0040265d      0x00000000      0x00000000      0x00000000
0x603360 <host_table+32>:       0x00000000      0x00000000      0x00000000      0x00000000
0x603370 <host_table+48>:       0x00000000      0x00000000      0x00000000      0x00000000
0x603380 <host_table+64>:       0x00000000      0x00000000      0x00000000      0x00000000
0x603390 <host_table+80>:       0x00000000      0x00000000
(gdb)
```

我们发现这里存了一些 `node`, 应该是一个结构体。 每个 `node` 存了这么些东西（以node1为例）：

- `0x0000014c`
- `0x00000001`
- `0x006032e0 00000000` : 明显为下一个node的地址

所以这个结构体大概长这个样子：

```
struct node{
    int val;
    int idx;
    struct node* next;
}
```

所以初始时：`%edx` 是第一个 `node` 的首地址， `(%rdx+8)` 正好是 `next` 指针。

- 现在把 `%esi` 置为0，并跳到 `+163`

`+163`： `(%rsp,%rsi,1), %ecx` ： 取出 `nums[0]`, 放到 `%ecx`，随后将 `%ecx` 与 1 比较

如果 `%ecx<=1`: 跳到 `+143` ，把 `%edx` 置为 `0x6032d0`, 把 `%rdx 即 0x6032d0` 拿给 `0x20+%rsp+%rsi`, `%rsi+4`, 比较 `%rsi` 与 `%0x18`

的关系，如果不等于，取出 `nums[1] 到 %ecx`， 随后和 `1` 比较

如果 `%ecx>1`， `%eax=1, %edx = 0x6032d0`, 随后跳到 `+130`, `%rdx = (%rdx+8) = (0x6032d0+8)` 及 `next` 指针 及 下一节点的首地址。然后不断给 `%eax` 加1， 知道和 `%ecx = nums[1]` 相同， 然后跳到 `+148`, 把 `%rdx` 存到栈上去。

综上所述，这一大段代码似乎是在把 `node[nums[i]]` 的首地址 按照 `i` 的顺序依次存到栈上去， 关系大概是这样的：

`node[nums[0]]: %rsp+0x20;  node[nums[1]]: %rsp+0x28 .....`， 这一段到此结束

- ```
  <+183>:   mov    0x20(%rsp),%rbx
  <+188>:   lea    0x28(%rsp),%rax
  <+193>:   lea    0x50(%rsp),%rsi
  <+198>:   mov    %rbx,%rcx
  <+201>:   mov    (%rax),%rdx
  <+204>:   mov    %rdx,0x8(%rcx)
  <+208>:   add    $0x8,%rax
  <+212>:   cmp    %rsi,%rax
  <+215>:   je     0x4011d2 <phase_6+222>
  <+217>:   mov    %rdx,%rcx
  <+220>:   jmp    0x4011bd <phase_6+201>
  ```

`mov 0x20(%rsp),%rbx` 取出 `node[nums[0]]` 的首地址， `mov 0x28(%rsp),%rax` 取出 `node[nums[1]]` 的首地址，

`0x8(%rcx)` 是 `node[nums[0]].next`，而`%rdx` 是 `node[nums[1]]` 即`node[nums[0]].next = node[nums[1]]`

随后是一个滚动循环。写成c语言大概是：

```c
for(int i = 0, j = 1; j <= 5; i++, j++){
    node[n[i]].next = node[n[j]];
}
```

到目前为止，这个链表结构为:

`node[nums[0]].next  = node[nums[1]]; node[nums[1]].next = node[nums[2]]; node[nums[2]].next = node[nums3]; node[nums[3]].next = node[nums[4]]; node[nums[4]].next = node[nums[5]], node[nums[5]].next = 0`

-
```

```
<+222>:    movq   $0x0,0x8(%rdx)
<+230>:    mov    $0x5,%ebp
<+235>:    mov    0x8(%rbx),%rax
<+239>:    mov    (%rax),%eax
<+241>:    cmp    %eax,(%rbx)
<+243>:    jge    0x4011ee <phase_6+250>
<+245>:    call   0x40143a <explode_bomb>
<+250>:    mov    0x8(%rbx),%rbx
<+254>:    sub    $0x1,%ebp
<+257>:    jne    0x4011df <phase_6+235>
<+259>:    add    $0x50,%rsp
<+263>:    pop    %rbx
<+264>:    pop    %rbp
<+265>:    pop    %r12
<+267>:    pop    %r13
<+269>:    pop    %r14
<+271>:    ret
```

`%eax` 为 `node[nums[1]].val`，`(%rbx) = node[nums[0]].val`，为了不让炸弹爆炸，必须使得 `node[nums[0]].val >= node[nums[1]].val`，随后 向前滚动一下，循环变量`%ebx-=1`. 也就是说，我们的节点序列的`val`必须是单调递减的. 我们打印出所有的`val`：

```
(gdb) x/24d 0x6032d0
0x6032d0 <node1>:       332     1       6304480 0
0x6032e0 <node2>:       168     2       6304496 0
0x6032f0 <node3>:       924     3       6304512 0
0x603300 <node4>:       691     4       6304528 0
0x603310 <node5>:       477     5       6304544 0
0x603320 <node6>:       443     6       0       0
```

发现：`node[3]>node[4]>node[5]>node[6]>node[1]>node[2]`

因此`nums = [3,4,5,6,1,2]`，但是在前面我们用7去减了每个数，那么最终答案为: `nums = [4,3,2,1,6,5]`

总结

```
root@LAPTOP-RI965SV0:/mnt/d/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!
3 0
So you got that one.  Try this one.
ionuvw
Good work!  On to the next...
4 3 2 1 6 5
Congratulations! You've defused the bomb!
```

由于对GDB不太熟练，导致需要人为的去记录每一时刻寄存器的值，这导致分析起来及其麻烦，尤其是遇到phase_6这样困难的问题时。