boomlab 实验 炸弹实验 系统级程序设计 CMU

原创

想想虔诚怎么做 于 2019-12-01 12:45:25 发布 13269 收藏 30

文章标签： boomlab 炸弹实验 汇编语言 系统级程序设计

本文链接：https://blog.csdn.net/qq_41409438/article/details/102876005

系统级程序设计 专栏收录该内容

7 篇文章 1 订阅

订阅专栏

MENU

大家如果做boomlab还碰到了什么有意思的错误可以留言或者发我邮箱(hwk2019ucb@163.com)哈

boomlab还有30s到达

嘎嘎，今天有机会做了系统级程序设计课程布置的作业！
韩波老师真的非常nice！！！助教也很nice！！！
下面开始本次实验的介绍：
首先我是在虚拟机中使用！
配置的是VMware 15和Ubuntu18（好像是的，不确定,不重要）

实验1

首先我们会获得一个可执行文件和一个.c的文件，readme被我删掉了
肯定有人想和我一样想直接跑一跑可执行文件！然后你可能就会碰到问题！

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ ./bomb
bash: ./bomb: Permission denied
```

那么有事问百度，我找到了解决办法
果不其然我找到了解决办法，好像是如此就可以给予权限还是啥的，但不管了，可以执行！
然后我随便输入了我的名字缩写！
果然炸了！
那么开始我们的下一步正式研究了！

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ sudo chmod -R 777 ~/Desktop/bomblab/bomblab/bomb
[sudo] password for hwk0901:
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
hwk

BOOM!!!
The bomb has blown up.
```

Step1–反汇编

我们先把可执行文件反汇编成一个.s文件

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ objdump -d bomb >bomb.s
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ ls
bomb  bomb.c  bomb.s
```

vim大法

然后我们可以装模作用的来看看这里都是些啥
但肯定像我这种性格，没有耐心看完
想看的可以

```
vim bomb.s
```

问题来了，记得安装vim...

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ vim bomb.s

Command 'vim' not found, but can be installed with:

sudo apt install vim
sudo apt install vim-gtk3
sudo apt install vim-tiny
sudo apt install neovim
sudo apt install vim-athena
sudo apt install vim-gtk
sudo apt install vim-nox

hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ sudo apt install vim
Reading package lists... Done
```


这里有两种很酷炫的操作，其实也很normal haha

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ vim -on bomb.s bomb.c
2 files to edit
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ vim -On bomb.s bomb.c
2 files to edit
```

第一种是垂直显示两个，第二个是竖直的并列两个！
我个人喜欢第二个

检查boom原因

```
0000000000400ee0 <phase_1>:
  400ee0:       48 83 ec 08             sub    $0x8,%rsp
  400ee4:       be 00 24 40 00          mov    $0x402400,%esi
  400ee9:       e8 4a 04 00 00          callq  401338 <strings_not_equal>
  400eee:       85 c0                   test   %eax,%eax
  400ef0:       74 05                   je     400ef7 <phase_1+0x17>
  400ef2:       e8 43 05 00 00          callq  40143a <explode_bomb>
  400ef7:       48 83 c4 08             add    $0x8,%rsp
  400efb:       c3                      retq
```

我们可以发现mov了一个值进$esi然后比较了**string not equal**
这肯定是string不一致就boomb，然后我们可以用gdb调试

gdb调试出结果

先安装gdb！

```
2 files to edit
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ sudo apt-get install gdb
[sudo] password for hwk0901:
Reading package lists... Done
Building dependency tree
```

然后输入指令

gdb bomb

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/bomb$ gdb bomb
GNU gdb (Ubuntu 8.1-0ubuntu3.1) 8.1.0.20180409-git
```

就可以快乐的调试辣

我们在phase_1处加断点！

```
Reading symbols from bomb...done.
(gdb) b phase_1
Breakpoint 1 at 0x400ee0
(gdb) r
```

(gdb) r 表示运行代码哦

随便输入一个字符串

disass 用于反汇编

```
(gdb) r
Starting program: /home/hwk0901/Desktop/bomblab/bomblab/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
da

Breakpoint 1, 0x0000000000400ee0 in phase_1 ()
(gdb) disass
Dump of assembler code for function phase_1:
=> 0x0000000000400ee0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ee4 <+4>:     mov    $0x402400,%esi
   0x0000000000400ee9 <+9>:     callq  0x401338 <strings_not_equal>
   0x0000000000400eee <+14>:    test   %eax,%eax
   0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
   0x0000000000400ef2 <+18>:    callq  0x40143a <explode_bomb>
   0x0000000000400ef7 <+23>:    add    $0x8,%rsp
   0x0000000000400efb <+27>:    retq
End of assembler dump.
```


examine

可以使用examine命令(简写是x)来查看内存地址中的值

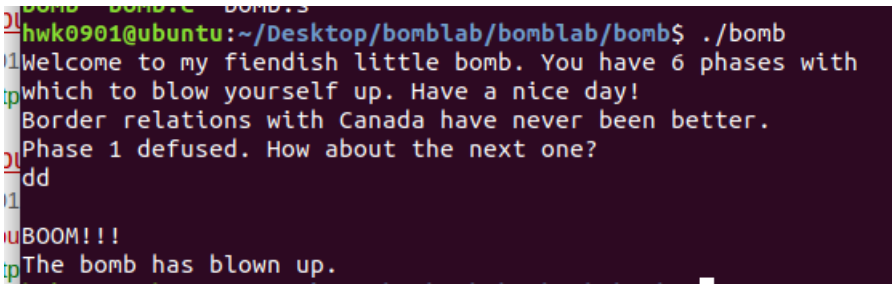
那么我们查看0x402400

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

答案出来啦！！！！

quit

输入quit退出gdb调试哦



实验1至此结束

撒花撒花

Border relations with Canada have never been better.

实验二

分析汇编语言

```
0000000000400efc <phase_2>:
  400efc:       55                      push   %rbp
  400efd:       53                      push   %rbx
  400efe:       48 83 ec 28             sub    $0x28,%rsp
  400f02:       48 89 e6                mov    %rsp,%rsi
  400f05:       e8 52 05 00 00          callq  40145c <read_six_numbers>
  400f0a:       83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:       74 20                   je     400f30 <phase_2+0x34>
  400f10:       e8 25 05 00 00          callq  40143a <explode_bomb>
  400f15:       eb 19                   jmp    400f30 <phase_2+0x34>
  400f17:       8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:       01 c0                   add    %eax,%eax
  400f1c:       39 03                   cmp    %eax,(%rbx)
  400f1e:       74 05                   je     400f25 <phase_2+0x29>
  400f20:       e8 15 05 00 00          callq  40143a <explode_bomb>
  400f25:       48 83 c3 04             add    $0x4,%rbx
  400f29:       48 39 eb                cmp    %rbp,%rbx
  400f2c:       75 e9                   jne    400f17 <phase_2+0x1b>
  400f2e:       eb 0c                   jmp    400f3c <phase_2+0x40>
  400f30:       48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35:       48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a:       eb db                   jmp    400f17 <phase_2+0x1b>
  400f3c:       48 83 c4 28             add    $0x28,%rsp
  400f40:       5b                      pop    %rbx
  400f41:       5d                      pop    %rbp
  400f42:       c3                      retq
```


push %rbx 保存rbx的值

sub $0x28,%rsp 将栈的长度增加0x28以便字符串存储

callq <read_six_numbers> 调用函数读取六个字符（应该会把字符串首地址给rsi 也就是放在上一帧栈顶rsp）

通过callq调用函数read_six_numbers，从函数名就可以知道是从我们的输入中获取6个数字。那么就可以大胆的猜测，刚才开辟栈空间的操作是在栈上分配了一个具有**6个元素的数组**，至于数组元素是什么类型，目前不得而知。并且可以判断函数read_six_numbers具有两个参数，第一个参数使我们的输入input，第二个参数就是数组第一个元素的首地址，即函数read_six_numbers的原型为**void read_six_numbers(const char *input*， *TYPE p)*

这里我研究了很久最后看了半天没看出个所以然，我就用gdb调试了下

```
(gdb) disass /m 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>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x000000000040148f <+51>:    cmp    $0x5,%eax
   0x0000000000401492 <+54>:    jg     0x401499 <read_six_numbers+61>
   0x0000000000401494 <+56>:    callq  0x40143a <explode_bomb>
   0x0000000000401499 <+61>:    add    $0x18,%rsp
   0x000000000040149d <+65>:    retq
End of assembler dump.
```


当当当！！！我发现了！！！！

PS:

- MOV指令的功能是传送数据，例如MOV AX,[1000H]，作用是将1000H作为偏移地址，寻址找到内存单元，将该内存单元中的数据送至AX；
- LEA指令的功能是取偏移地址，例如LEA AX,[1000H]，作用是将源操作数[1000H]的偏移地址1000H送至AX。理解时，可直接将[ ]去掉，等同于MOV AX,1000H。

原来函数read_six_numbers是通过函数sscanf从我们的输入中获得6个数字的。这就好办了，根据函数sscanf的原型，我们来仔细看看调用函数sscanf之前，它需要的参数是如何传递进去的。
第一个参数肯定是我们的输入input的首地址，它目前储存在寄存器%rdi中，那么第二个参数应该是sscanf需要的格式化字符串，从mov $0x4025c3,%esi中知道这个格式化字符串存储在地址0x4025c3处，用x/s命令查看，原来是"%d %d %d %d %d %d"，这下我们可以断定，在函数phase_2中的数组元素类型是int型，即int a[6]。

Register	Usage	function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12-r14	callee-saved registers	Yes
%r15	callee-saved register; optionally used as GOT base pointer	Yes

既然格式化字符串中有个6个%d，自然的sscanf函数还需要6个参数，应该分别是数组a每个元素的地址，即分别是**&a[0]，&a[1]，&a[2]，&a[3]，&a[4]，&a[5]。
根据调用约定，前4个元素的地址应该用寄存器传递，分别是%rdx、%rcx、%r8、%r9**
最后两个通过栈来传递，通过**lea 0x10(%rsi),%rax，mov %rax,(%rsp)和lea 0x14(%rsi),%rax**，**mov %rax,0x8(%rsp)**分别将&a[4]和&a[5]压栈，这就完成了sscanf所有参数的传递任务。
如有不懂请反复看上面那张图
&a[0]

```
0x0000000000401460 <+4>:     mov    %rsi,%rdx
```

&a[1]

```
0x0000000000401463 <+7>:     lea    0x4(%rsi),%rcx
```

&a[2]

```
0x000000000040147c <+32>:    lea    0x8(%rsi),%r8
```

&a[3]

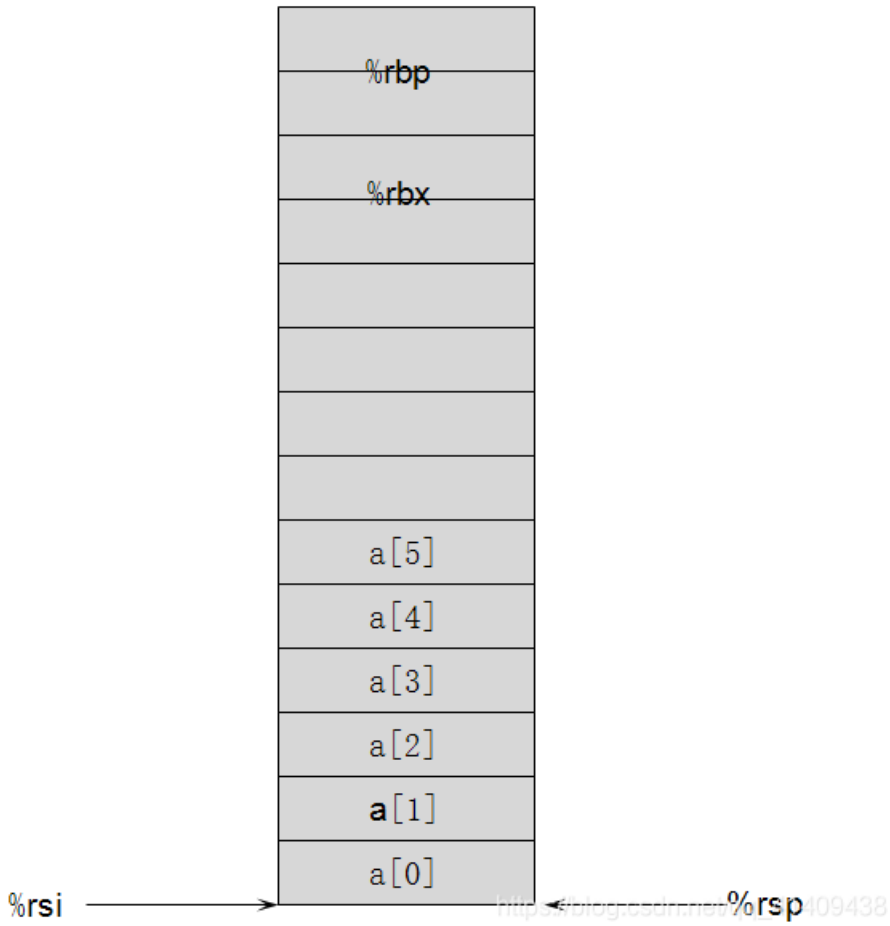
```
0x0000000000401478 <+28>:    lea    0xc(%rsi),%r9
```

&a[4]

```
0x0000000000401470 <+20>:    lea    0x10(%rsi),%rax
0x0000000000401474 <+24>:    mov    %rax,(%rsp)
```

一定要注意！！！这里tmd是16进制！！！！！！我看了好久为什么是10！！！！！！！！
&a[5]

```
0x0000000000401467 <+11>:    lea    0x14(%rsi),%rax
0x000000000040146b <+15>:    mov    %rax,0x8(%rsp)
```



写成这样了如果还看不懂的话，建议原地爆炸
如果没仔细看的话，你下棋必被祖安人民指指点点

%rsp %rsp+0x4 %rsp+0x8 %rsp+0xC %rsp+0x10 %rsp+0x14 %rbp=%rsp+0x18

Number[0]	Number[1]	Number[2]	Number[3]	Number[4]	Number[5]	Edge

让我们回到phase_2函数

mov %rsp,%rsi 将栈顶的的地址送给指针寄存器,rsp为字符串首地址

cmpl $0x1,(%rsp) 把第一个数字和1比较！

je 400f30 <phase_2+0x34> 如果相等跳到400f30，让我们思考下这里写了什么东西

```
0x0000000000400f30 <+52>:    lea    0x4(%rsp),%rbx
```

```
0x0000000000400f35 <+57>:    lea    0x18(%rsp),%rbp
```

艹，这里写到了函数一开始就压入栈的两个寄存器，使之分别指向&a[1]和&a[6]。读者可能会问，数组a只有6个元素啊，那么最后一个元素不是应该是a[5]吗？ 注意，这里取的是a[6]的地址，这在标准C中是允许的，只要不取这个地址处的值都是OK的，而标准允许这样做是有原因的，这样就可以很方便在循环中来遍历数组了。

jmp 0x400f17 <phase_2+27>，jmp到地址0x0000000000400f17处继续执行。

如果仔细观察地址0x0000000000400f2c处也有个jmp到地址0x0000000000400f17处的指令，我们可以判定这里应该是个循环语句，那么循环体的内容可以根据地址0x0000000000400f17 ~ 地址0x0000000000400f29处的指令来获知。

假设第一次进入循环体，现在寄存器**%rbx**指向**&a[1]，那么mov -0x4(%rbx),%eax**就是把a[0]的值存储寄存器%eax，add %eax,%eax将寄存器%eax的值乘以2，**cmp %eax,(%rbx)就是将现在%eax的值与a[1]比较**，也就是a[0]乘以2后的值与a[1]比较。如果不相等， 则触发炸弹，相等则%rbx指向下一个元素的地址，在这里就是&a[3]。好了我懂了就是乘以2从1开始

1 2 4 8 16 32

ENDING

```
(gdb) quit
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/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!
```

tql，柯少是真的智慧

实验三

个人感觉这个实验有点脑瘫，其实韩老师上课基本都说了，还蛮简单的...、
依旧断点找到phase_3的函数来看干了些啥

```
1 2 4 8 16 32
That's number 2.  Keep going!
ss

Breakpoint 1, 0x0000000000400f43 in phase_3 ()
(gdb) disass
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>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000400f60 <+29>:    cmp    $0x1,%eax
   0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>
   0x0000000000400f65 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
   0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
   0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
   0x0000000000400f75 <+50>:    jmpq   *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>:   callq  0x40143a <explode_bomb>
```


有了分析phase_2的经验，很容易从寄存器%esi中获得格式化字符串，x/s显示的结果是"%d %d"，即从我们的输入中获得两个int型整数。我们不妨将第一个整数命令为num1，第二个为num2。
紧接着判断sscanf的返回值，如果小于等于1，则触发炸弹。否则继续，这里都还蛮简单，后面有意思了！

cmpl $0x7,0x8(%rsp)，判断num1是否大于7，如果大于，则调整到地址0x0000000000400fad处，触发炸弹。否则继续。

```
0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
```

下一步就是核心了

```
0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
0x0000000000400f75 <+50>:    jmpq   *0x402470(,%rax,8)
```

熟悉不熟悉！！！！！韩老师牛逼！！！

第一条指令，**mov 0x8(%rsp),%eax**，将num1的值存储寄存器%eax中，第二条指令**jmpq *0x402470(,%rax,8)**，这条指令什么意思呢？

可能AT&T的汇编指令不太容易看懂，那我们通过set disassembly-flavor intel来查看intel形式的这条指令为**jmp QWORD PTR [rax*8+0x402470]**，这下就容易多了 – 取出地址rax*8+0x402470处的值，并调转到这个值指示的内存地址处继续执行。

起飞，原地起飞，快来看看这个地址放了些啥

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

因为上面判断num1小于8，因此可知跳转表中应该存储有8个地址。x表明以十六进制的形式显示地址，g表示每8个字节的内存，因为这是x64平台，所以地址占8个字节

这些地址都是在phase_3中存在的！！！！！

答案

当num1等于0时，跳转到0x0000000000400f7c处执行。如果num2不等于0xcf，则触发炸弹。

当num1等于1时，跳转到0x0000000000400fb9处执行。如果num2不等于0x137，则触发炸弹。

当num1等于2时，跳转到0x0000000000400f83处执行。如果num2不等于0x2c3，则触发炸弹。

当num1等于3时，跳转到0x0000000000400f8a处执行。如果num2不等于0x100，则触发炸弹。

当num1等于4时，跳转到0x0000000000400f91处执行。如果num2不等于0x185，则触发炸弹。

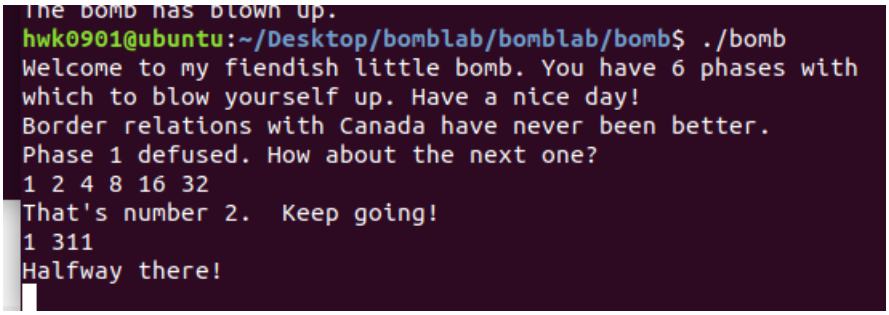
当num1等于5时，跳转到0x0000000000400f98处执行。如果num2不等于0xce，则触发炸弹。

当num1等于6时，跳转到0x0000000000400f9f处执行。如果num2不等于0x2aa，则触发炸弹。

当num1等于7时，跳转到0x0000000000400fa6处执行。如果num2不等于0x147，则触发炸弹。

所以答案这不就好起来了？？？？？

随便拿一个去测试就完事了，快快乐乐



答案如下

（0，207）、（1，311）、（2，707）、（3，256）、（4，389）、（5，206）、（6，682）、（7，327)

实验四

这个实验还是老办法gdb调试调试

```
000000000040100c <phase_4>:
  40100c:       48 83 ec 18             sub    $0x18,%rsp
  401010:       48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  401015:       48 8d 54 24 08          lea    0x8(%rsp),%rdx
  40101a:       be cf 25 40 00          mov    $0x4025cf,%esi
  40101f:       b8 00 00 00 00          mov    $0x0,%eax
  401024:       e8 c7 fb ff ff          callq  400bf0 <__isoc99_sscanf@plt>
  401029:       83 f8 02                cmp    $0x2,%eax
  40102c:       75 07                   jne    401035 <phase_4+0x29>
  40102e:       83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:       76 05                   jbe    40103a <phase_4+0x2e>
  401035:       e8 00 04 00 00          callq  40143a <explode_bomb>
  40103a:       ba 0e 00 00 00          mov    $0xe,%edx
  40103f:       be 00 00 00 00          mov    $0x0,%esi
  401044:       8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:       e8 81 ff ff ff          callq  400fce <func4>
  40104d:       85 c0                   test   %eax,%eax
  40104f:       75 07                   jne    401058 <phase_4+0x4c>
  401051:       83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  401056:       74 05                   je     40105d <phase_4+0x51>
  401058:       e8 dd 03 00 00          callq  401438 <explode_bomb>
```


我们首先分析是输入的是什么：

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

又是两个数字，那么我们可以暂时把他们假设成num1，num2
在代码中有一句话我们可以看到：

```
  40102e:       83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:       76 05                   jbe    40103a <phase_4+0x2e>
  401035:       e8 00 04 00 00          callq  40143a <explode_bomb>
```

这里比较的是num1和14的大小关系，当低于或等于时候就跳转，否则就炸了
所以我们可以知道num是要小于等于14的

```
401051:       83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
401056:       74 05                   je     40105d <phase_4+0x51>
```

这里我们可以知道num2的值不为0也会炸，所以两个参数，第二个参数一定是0
之后我们需要分析func4

func4

```
  40103a:       ba 0e 00 00 00          mov    $0xe,%edx
  40103f:       be 00 00 00 00          mov    $0x0,%esi
  401044:       8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:       e8 81 ff ff ff          callq  400fce <func4>
```

大致意思就是传了三个参数进去
% edx 是
% esi 是
% edi 中是第一个参数
mov %edx,%eax，将参数c存储在寄存器%eax中。
sub %esi,%eax，用%eax中的值减去%esi，结果存在寄存器%eax中，即c - b的值存储在寄存器%eax中。（14）
为了直观，我们将%eax定义为一个局部变量int x，即int x = c - b;

```
0x0000000000400fd6 <+8>:     mov    %eax,%ecx
0x0000000000400fd8 <+10>:    shr    $0x1f,%ecx
0x0000000000400fdb <+13>:    add    %ecx,%eax
0x0000000000400fdd <+15>:    sar    %eax
```

这几句整合起来就是x = (x>>31 + x) >> 1，大家可以自己推下，蛮简单的

lea (%rax,%rsi,1),%ecx，这句意思是%ecx = %rax + %rsi * 1，其中%rax就是我们刚刚求得的x，%rsi是参数b。因此%ecx = (x>>31 + x) >> 1 + b;
我们将寄存器%ecx定义一个局部变量，名为tmp，即int tmp = (x>>31 + x) >> 1 + b；将x带进去就是int tmp = ((c - b) >> 31 + (c - b)) >> 1 + b;

```
0x0000000000400fe2 <+20>:    cmp    %edi,%ecx
0x0000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
```

这里就是比较这个tmp和num1的关系，如果小于等于就跳到0x400ff2
处

```
  0x0000000000400ff2 <+36>:    mov    $0x0,%eax
  0x0000000000400ff7 <+41>:    cmp    %edi,%ecx
  0x0000000000400ff9 <+43>:    jge    0x401007 <func4+57>
```

这里的意思是比较tmp和num1如果大于等于就去0x401007，原来如此通过小于等于和大于等于，即如果等于就ok！那么最基本的情况就结束了
我们可以继续分析代码得到如下C语言代码

```c
//               %edi    %esi   %edx
static int func4(int a, int b, int c)
{
    int tmp = (((c - b) + ((c - b) >> 31)) >> 1) + b;

    if (tmp <= a) {
        if (tmp == a) {
            return (0);
        } else {
            return func4(a, tmp + 1, c) * 2 + 1;
        }
    } else {
        return func4(a, b, tmp - 1) * 2;
    }
}
```

然后我们写一个测试程序就知道答案有哪些了

```
#include <stdio.h>
#include <stdlib.h>
//                %edi    %esi   %edx
static int func4(int a, int b, int c)
{
    int tmp = (((c - b) + ((c - b) >> 31)) >> 1) + b;

    if (tmp <= a) {
        if (tmp == a) {
            return (0);
        } else {
            return func4(a, tmp + 1, c) * 2 + 1;
        }
    } else {
        return func4(a, b, tmp - 1) * 2;
    }
}

int main(int argc, const char *argv[])
{
    int i, result;

    for (i = 0; i < 14; ++i) {
        result = func4(i, 0, 14);
        if (result == 0) {
            printf("%d\n", i);
        }
    }
    return 0;
}
```

程序输出0 1 3 7，因此本阶段的答案有4组，分别为（0,0）、（1,0）、（3,0）、（7,0）。

```
hwk0901@ubuntu:~/Desktop/bomblab/bomblab/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!
0 0
So you got that one.  Try this one.
```


实验五

先看看phase_5的汇编语言

```
hwk0901@ubuntu: ~/Desktop/bomblab/bomblab/bomb
File  Edit  View  Search  Terminal  Help
  40105d:       48 83 c4 18             add    $0x18,%rsp
  401061:       c3                      retq

0000000000401062 <phase_5>:
  401062:       53                      push   %rbx
  401063:       48 83 ec 20             sub    $0x20,%rsp
  401067:       48 89 fb                mov    %rdi,%rbx
```

```
  40106a:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  401071:       00 00
  401073:       48 89 44 24 18          mov    %rax,0x18(%rsp)
  401078:       31 c0                   xor    %eax,%eax
  40107a:       e8 9c 02 00 00          callq  40131b <string_length>
  40107f:       83 f8 06                cmp    $0x6,%eax
  401082:       74 4e                   je     4010d2 <phase_5+0x70>
  401084:       e8 b1 03 00 00          callq  40143a <explode_bomb>
  401089:       eb 47                   jmp    4010d2 <phase_5+0x70>
  40108b:       0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
  40108f:       88 0c 24                mov    %cl,(%rsp)
  401092:       48 8b 14 24             mov    (%rsp),%rdx
  401096:       83 e2 0f                and    $0xf,%edx
  401099:       0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
  4010a0:       88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
  4010a4:       48 83 c0 01             add    $0x1,%rax
  4010a8:       48 83 f8 06             cmp    $0x6,%rax
  4010ac:       75 dd                   jne    40108b <phase_5+0x29>
  4010ae:       c6 44 24 16 00          movb   $0x0,0x16(%rsp)
  4010b3:       be 5e 24 40 00          mov    $0x40245e,%esi
  4010b8:       48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  4010bd:       e8 76 02 00 00          callq  401338 <strings_not_equa
l>
  4010c2:       85 c0                   test   %eax,%eax
  4010c4:       74 13                   je     4010d9 <phase_5+0x77>
  4010c6:       e8 6f 03 00 00          callq  40143a <explode_bomb>
  4010cb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  4010d0:       eb 07                   jmp    4010d9 <phase_5+0x77>
  4010d2:       b8 00 00 00 00          mov    $0x0,%eax
  4010d7:       eb b2                   jmp    40108b <phase_5+0x29>
  4010d9:       48 8b 44 24 18          mov    0x18(%rsp),%rax
  4010de:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
                                                      https://blog.csdn.net/qq_41409438
                                                      471,22-28     27%
```

看着这么多有点压迫感，但是应该不难，一开始的push，submov其实都还好我感觉然后一直到callq语句核心来了
他比较了返回值%eax和6那么这里返回的是字符串的长度，如果不是6就boom，好了这里解决了，不是很难，那么我们看下je语句跳到了什么地方，**4010d2**
在这里设置寄存器%eax值为0，再跳转到0x000000000040108b指定位置。

```
  40108b:       0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
  40108f:       88 0c 24                mov    %cl,(%rsp)
  401092:       48 8b 14 24             mov    (%rsp),%rdx
  401096:       83 e2 0f                and    $0xf,%edx
  401099:       0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
  4010a0:       88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
  4010a4:       48 83 c0 01             add    $0x1,%rax
  4010a8:       48 83 f8 06             cmp    $0x6,%rax
  4010ac:       75 dd                   jne    40108b <phase_5+0x29>
```

然后我们发现这里有个循环体，那么这个是干什么的呢？
mov %cl,(%rsp)
mov (%rsp),%rdx
and $0xf,%edx
这三行只需关注最后一行，即只取字符数值的低4位。例如字符'A'，ascii码值为0x41，取低4位后得到0x01。
movzbl 0x4024b0(%rdx),%edx，又是访问数组，并且是用刚刚才算得的**字符低4位数值**作为索引来访问这个值。从movzbl得到这是个单字节数组，数组首地址是0x4024b0。我们使用x命令查看下这个数组的内容是什么。
mov %dl,0x10(%rsp,%rax,1)，这句又是在访问数组，而且是个局部单字节数组，数组首地址在%rsp + 0x10处，目前猜测这个数组至少6个字节大小。
循环结束后，movb $0x0,0x16(%rsp)，将局部单字节数组第7个元素复制为0，所以我们可以得出这个局部数组至少7个字节大小。

最后又调用函数strings_not_equal来比较字符串，这个函数在phase_1中就已经分析过了。

待比较的两个字符串一个是刚刚的局部数组，一个是存储在地址0x40245e处的字符串，我们使用x/s查看是"flyers"。

```
Quit
(gdb) x/s 0x40245e
0x40245e:       "flyers"
(gdb)
```

…然后事情就变得简单了。记得之前那个莫名奇妙的数组么

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

看来我们输入的6个字符是array数组的下标值。得到的6个字符必须是"flyers"，得出这个结论就好办了，接下来就是数数游戏了。要得到字符"flyers"，下标值必须依次为9、15、14、5、6和7。用16进制来表示的话，则为0x09、0x0f，0x0e，0x05，0x06和0x07。

也就是说我们输入的

- 第一个字符的低4位的值必须是0x09，查表得出字符')'、'9'、'I'、'Y'、'i'、'y'符合条件。
- 第二字字符的低4位的值必须是0x0f，查表得出字符'/'、'? '、'O'、'_'、'o'符合条件。
- 第三个字符的低4位的值必须是0x0e，查表得出字符'.'、'>'、'N'、'^'、n'、'~'符合条件。
- 第四个字符的低4位的值必须是0x05，查表得出字符'%'、'5'、'E'、'U'、'e'、'u'符合条件。
- 第五个字符的低4位的值必须是0x06，查表得出字符'&'、'6'、'F'、'V'、'f'、'v'符合条件。
- 第六个字符的低4位的值必须是0x07，查表得出字符'''、'7'、'G'、'W'、'g'、'w'符合条件。

因此phase_5的答案就多了，这6组字符随便按顺序组合即可。

```
mblab/bomb/bomb
Welcome to my fiendish little bomb. You have 6 pha
ses with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been bette
r.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
1 311
Halfway there!
0 0
So you got that one.  Try this one.
9?>567
Good work!  On to the next...
```


然后是我对于实验五的一些更细的思考，与答案无关，答案呢很好做，但是我想弄明白经历每一步：
先列个总结怕自己忘记

- %rax 作为函数返回值使用。
- %rsp 栈指针寄存器，指向栈顶
- %rdi，%rsi，%rdx，%rcx，%r8，%r9 用作函数参数，依次对应第1参数，第2参数。。。
- %rbx，%rbp，%r12，%r13，%14，%15 用作数据存储，遵循被调用者使用规则，简单说就是随便用，调用子函数之前要备份它，以防他被修改
- %r10，%r11 用作数据存储，遵循调用者使用规则，简单说就是使用之前要先保存原值

push %rbx 把rbx寄存器压栈
sub %0x20 %rsp 开辟一个长度是20的空间出来，这些之后应该用得到
mov %rdi % rax 把接收到的第一个参数传递给%rax
mov %fs:0x28,%rax 不知道在干啥，有知道的可以告诉我下，据说是gcc做的一个栈保护检测机制
mov %rax,0x18(rsp)这里我怀疑是把%rax的值给到了rsp的24位置偏移的地方，但是这个是在干啥我没想明白
xor %eax %eax 这里在干啥我也没搞清楚，但不妨碍我做题…
callq 4014b <string_length> 这里调用了字符串长度的函数那么这里会有一个返回值给到了eax
cmp %0x6 %eax 这里应该就是看是否是六个字符吧，如果不是就直接炸了
然后就会看到这个函数函数跳走了
mov %0x0 %eax 设置寄存器%eax值为0，再跳转到0x000000000040108b指定…有一说一不干扰我做题，但是这是在干什么我还是不知道，暂且当作置零
之后跳走了
之后的分析就是上面对循环分析了，easy了，做题可以做，细究我炸了
开始最后一个实验！！！！！！

实验六

这个phase_6也太长了…我裂开…甚至有点想打人…如果暴力求解的话会不会有答案…

gdb调试

```
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
```

前6行很老套，分别将寄存器%r14、%r13、%r12、%rbp和%rbx入栈，然后开辟栈空间，这次开辟的栈空间可不小哦，足足0x50个字节

接下来调用函数read_six_numbers，这个函数我们可不是第一次遇到了哦！在phase_2中详细分析过，这个函数需要两个参数，**第一个是我们输入的字符串，目前存储在寄存器%rdi中；第二个参数是一个6个int型元素数组的首地址**，这里通过寄存器**%rsi**传递。我们断定phase_6内定义了int a[6]；它的首地址就是当前栈顶，这点要牢记，因为后续有很多以%rsp为基址的内存访问，看到就会意识到是在访问数组a。

```
0x000000000040110b <+23>:    mov    %rsp,%r14
0x000000000040110e <+26>:    mov    $0x0,%r12d
0x0000000000401114 <+32>:    mov    %r13,%rbp
```

- 使寄存器指向栈顶，也就是数组a的首地址。
- 寄存器%r12d初始化为0，从后续的add r12d, 1来看，它应该是充当一个计数器的作用
- **%r13是指向数组a首地址的**，因此这里是使寄存器%rbp指向数组a首地址。到目前为止，指向数组a首地址的寄存器有不少，分别为%r13、%r14、%rsi、%rbp。
-

```
   0x0000000000401117 <+35>:    mov    0x0(%r13),%
eax
   0x000000000040111b <+39>:    sub    $0x1,%eax
   0x000000000040111e <+42>:    cmp    $0x5,%eax
   0x0000000000401121 <+45>:    jbe    0x401128 <p
hase_6+52>
   0x0000000000401123 <+47>:    callq  0x40143a <e
xplode_bomb>
```

这5条汇编指令很清晰，取出当前**%rbp指向的数组a元素值与6比较**，如果大于6则触发炸弹，否则跳转到0x401128继续执行。

```
   0x0000000000401128 <+52>:    add    $0x1,%r12d
   0x000000000040112c <+56>:    cmp    $0x6,%r12d
   0x0000000000401130 <+60>:    je     0x401153 <p
hase_6+95>
```

跳转到地址0x0000000000401128处，寄存器%r12d这个计数器增加1，接着判断是否等于6了，如果等于6了，则跳转到地址0x401153处，观察到这里已经跳出了最外层循环，结束了。所以可以判断%12d是否等于6是结束循环的条件。
我的脑袋要炸了…

```
   0x0000000000401132 <+62>:    mov    %r12d,%ebx
   0x0000000000401135 <+65>:    movslq %ebx,%rax
   0x0000000000401138 <+68>:    mov    (%rsp,%rax,
4),%eax
   0x000000000040113b <+71>:    cmp    %eax,0x0(%r
bp)
   0x000000000040113e <+74>:    jne    0x401145 <p
hase_6+81>
   0x0000000000401140 <+76>:    callq  0x40143a <e
xplode_bomb>
   0x0000000000401145 <+81>:    add    $0x1,%ebx
   0x0000000000401148 <+84>:    cmp    $0x5,%ebx
   0x000000000040114b <+87>:    jle    0x401135 <p
hase_6+65>
```


如果%12d小于6，继续执行。将%12d的值赋给寄存器%ebx后，后面的一些列操作又是一个循环，从地址0x0040114b处的jmp指令可以判断出来。这个循环做了什么呢？很简单，就是将后续的数组a元素值与a[%12d]比较，如果相等则触发炸弹。这意味着什么？ 意思是说a[0]和a[%12d]不能相等。

然后他把ebx加一，之后把ebx和5比较大小，如果小于或等于则跳转了，这里其实是个越界符号判断？然后他又跳回去了，那么这里循环了六个元素并且他们都和第一个不一样，这就是核心了，因为ebx某种程度上是数组的index，六个元素 0 1 2 3 4 5 大循环结束，到了这里，我们得出结论：数组a的6个元素值不能大于，并且它们彼此不相等。

```
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
-Type <return> to continue, or q <return> to quit---
0x000000000040116a <+118>:   cmp    %rsi,%rax
0x000000000040116d <+121>:   jne    0x401160 <phase_6+108>
```

这段比较简单，寄存器%rax指向数值a首地址，而寄存器%rsi指向a[6]，即数组a的最后一个元素的末端，以此为循环条件，分别用7减去数组a的元素，结果再存回数组a中。

```
   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
```


首先这里有两层循环，外层循环以寄存器%esi为计数器，初始值为0； 里层循环以寄存器%eax为计数器，并且初值为1。

最为关键的是这里牵扯到两个内存地址，一个是以%rsp + 0x20为基址的内存区域，步长为8，这个肯定是函数phase_6内的局部变量，就跟数组a一样。

另一个地址为0x6032d0，通过调试，发现这个地址是位于数据段.data内的地址，用x命令观察如下：

```
End of assembler dump.
(gdb) x/12xg 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
(gdb)
```

卧槽，这tm竟然是个链表，我裂开
目前可以猜测nodeX的类型是个结构体，最后一个成员是指向同类型的指针，其他成员类型不得而知

```
0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
0x00000000004011e3 <+239>:   mov    (%rax),%eax
0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250>
```

可以看到这里在比较结构体struct node的前4个字节，因此可以初步判断出struct node的第一个成员是int型的整数。

struct node {

int value;

struct node *next;

};

继续观察， 后续代码中访问next时，为什么都是结构体首地址加上偏移值8呢？
原因在于这是x64平台，指针大小占8个字节，而int型占4个字节，因此struct node中的next成员为了内存对齐的需要，会在value和next之间填充4个字节的padding，不过仔细观察上面的截图，这里所谓的padding分别是1、2、3、4、5和6，这些数值好像不是内存里的随意值，更像是特意填充的，因此我们可以假设struct node的完整类型如下：

struct node {

int value;

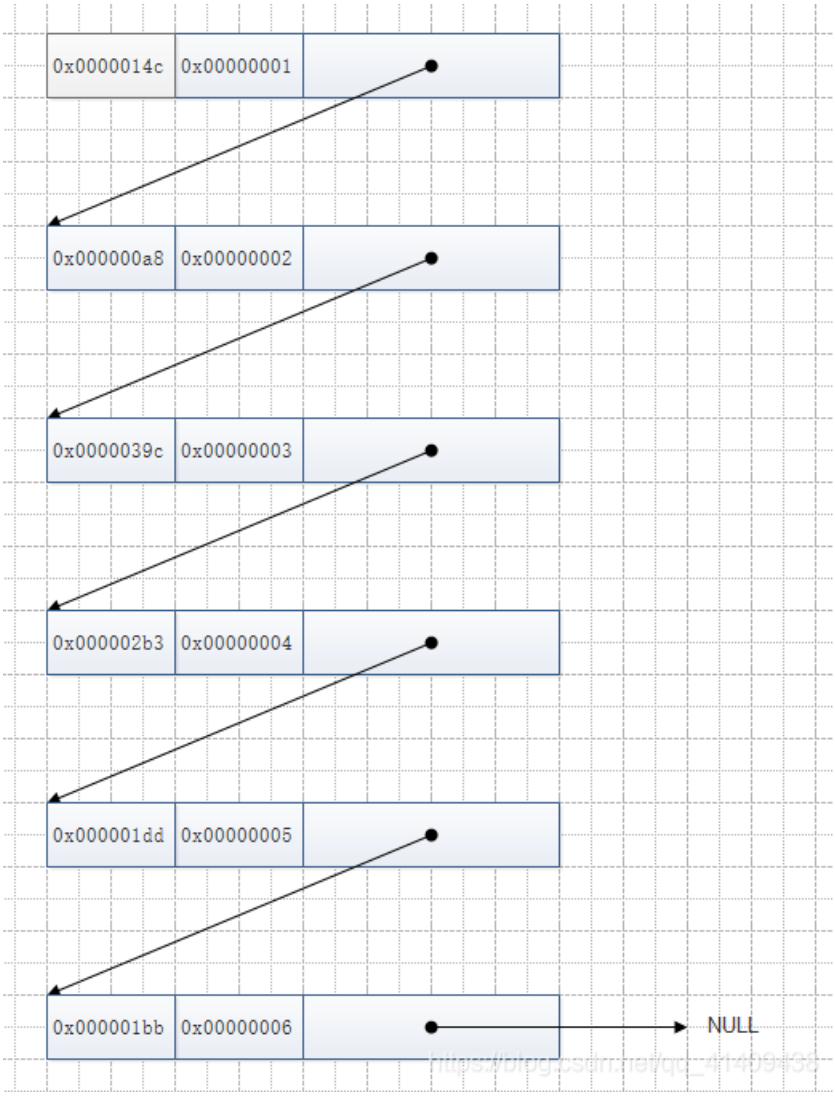
int seq;

struct node *next;

};

当然了，成员seq在后续代码中并没有什么作用，最后关键部分用到的是成员value；

好了，因此我们可以得出链表在内存中的初始状态如下：



回到代码，从—> **0x00401188 *，*可以猜测以%rsp + 0x20为基址的局部变量是个数组，并且数组元素类型为struct node*，即指向struct node的指针。因此可以判定函数内定义了一个struct node *nodes[6];数组。

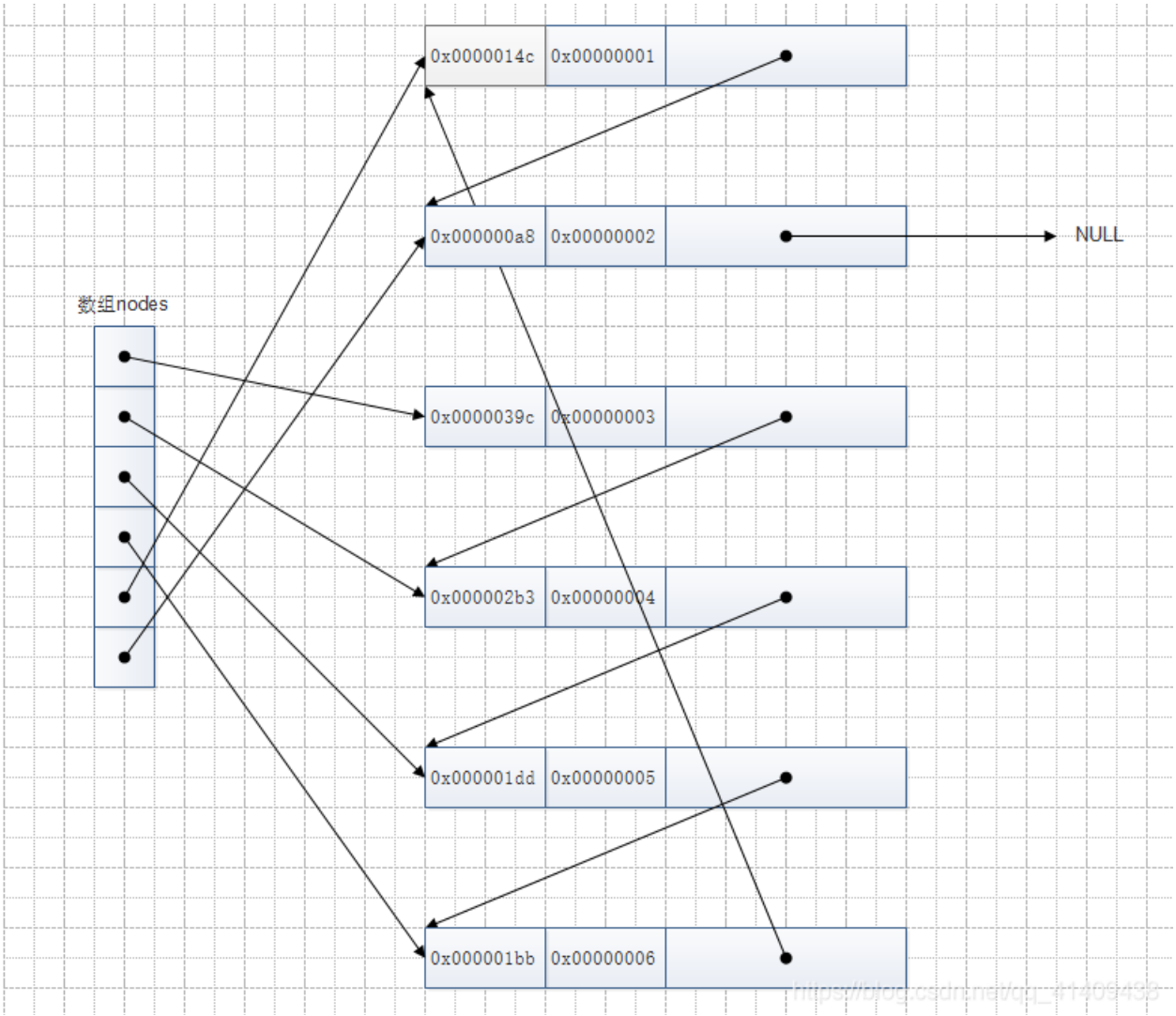
这段代码是在给数组nodes的每个元素赋值，根据对应数组a的元素分别指向不同的nodeX。如果a[%esi]的值小于等于1则nodes[%esi]直接指向node1。否则指向对应的nodeX，其中X的值与a[%esi]的值相等。

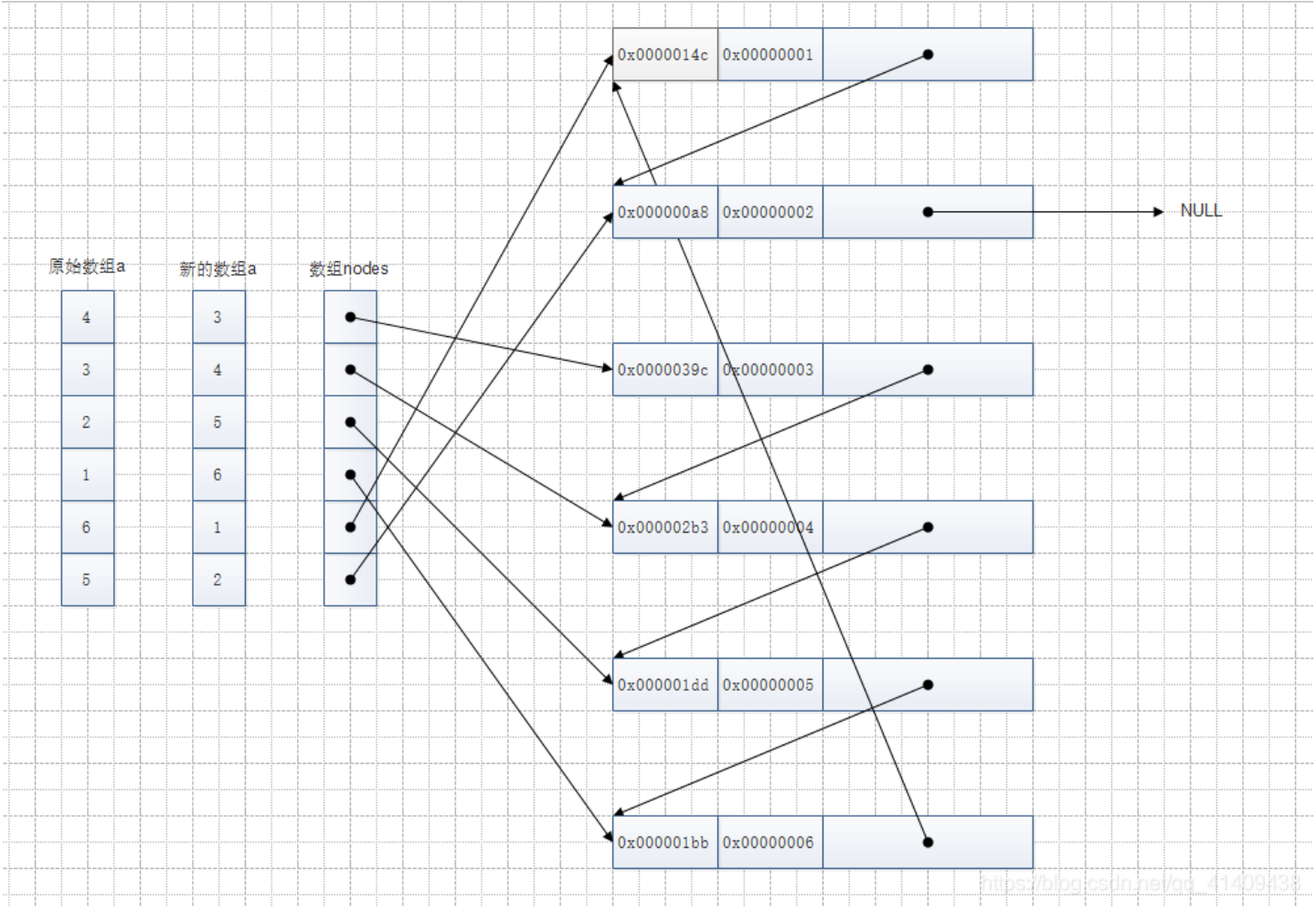
```
0x00000000004011ab <+183>:    mov    0x20(%rsp),%rbx
0x00000000004011b0 <+188>:    lea    0x28(%rsp),%rax
0x00000000004011b5 <+193>:    lea    0x50(%rsp),%rsi
0x00000000004011ba <+198>:    mov    %rbx,%rcx
0x00000000004011bd <+201>:    mov    (%rax),%rdx
0x00000000004011c0 <+204>:    mov    %rdx,0x8(%rcx)
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)
```


循环遍历数组nodes，调整各自的next值。

```
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
Type <return> to continue, or q <return> to quit---
0x00000000004011f2 <+254>:    sub    $0x1,%ebp
0x00000000004011f5 <+257>:    jne    0x4011df <phase_6+235>
```

最后的这段代码最为关键，它告诉我们链表最终的内存状态，得出的结论是：从nodes[0]开始遍历链表，nodeX.i的值是从大到小的顺序排列的，即如下这样：





CNM 我做出来了！！！！！！！！！

因此得到最终的输入应该是:4 3 2 1 6 5。 Bingo！！！

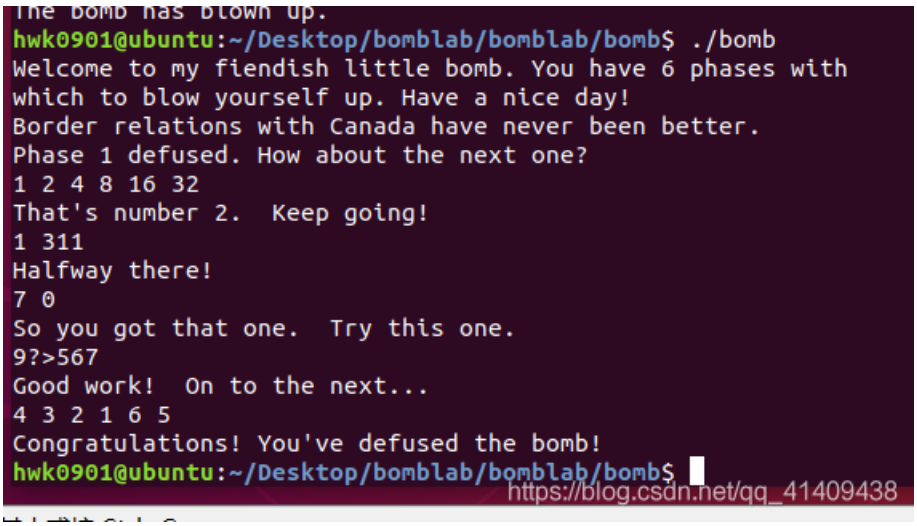
我从网上找到了牛逼的程序员复原的程序：

```
#include <stdio.h>
#include <stdlib.h>

static void read_six_numbers(const char *input, int *a)
{
 //     %rdi      %rsi   %rdx    %rcx   %r8   %r9    (%rsp)  *(%rsp)
 int result = sscanf(input, "%d %d %d %d %d %d", &a[0], &a[1], &a[2], &a[3], &a[4], &a[5]);
 if (result <= 5) {
  explode_bomb();
 }
}

struct node {
 int value;
 int seq;
 struct node *next;
};

struct node node6 = {
 0x000001bb, 6, NULL
};

struct node node5 = {
```

```
 0x000001dd, 5, &node6
};

struct node node4 = {
 0x000002b3, 4, &node5
};

struct node node3 = {
 0x0000039c, 3, &node4
};

struct node node2 = {
 0x000000a8, 2, &node3
};

struct node node1 = {
 0x0000014c, 1, &node2
};

void phase_6(const char *input)
{
 int a[6];
 struct node *nodes[6];

 read_six_numbers(input, a);

 // %r13
 int *pa = a;
 // %r12d
 int i = 0;

 for ( ;; ) {
  if (*pa > 6) {
   explode_bomb();
  }
  ++i;

  if (i == 6) break;

  int j = i;
  do {
   if (*pa == a[j]) {
    explode_bomb();
   }
   j++;
  } while (j <= 5);

  ++pa;
 }

 int *begin = &a[0];
 int *end = &a[6];
 do {
  *begin = 7 - *begin;
  ++begin;
 } while (begin != end);

 i = 0;
 do {
```

```
  if (a[i] <= 1) {
   nodes[i] = &node1;
  } else {
   int j = 1;
   struct node *pnode = &node1;
   do {
    pnode = pnode->next;
    ++j;
   } while (j != a[i]);
   nodes[i] = pnode;
  }
  ++i;
 } while (i != 6);

 struct node *head = nodes[0];  // %rbx
 struct node **begin_node = &nodes[1]; // %rax
 struct node **end_node = &nodes[6];  // %rsi
 struct node *tmp;    // %rdx

 for ( ;; ) {
  // %rdx
  tmp = *begin_node;
  head->next = tmp;
  ++begin_node;
  if (begin_node == end_node) break;
  head = tmp;
 }
 tmp->next = NULL;

 i = 5;
 head = nodes[0];
 do {
  tmp = head->next;
  if (head->value < tmp->value) {
   explode_bomb();
  }
  head = tmp;
  --i;
 } while (i != 0);

 printf("sucess\n");
}

#if 0
int main(int argc, const char *argv[])
{
 phase_6(argv[1]);
 return 0;
}
#endif
```

这个代码都把我看晕了，让我们快去试试对不对！



答案汇总

- Border relations with Canada have never been better.
- 1 2 4 8 16 32
- （0，207）、（1，311）、（2，707）、（3，256）、（4，389）、（5，206）、（6，682）、（7，327)
- （0,0）、（1,0）、（3,0）、（7,0）
- 第一个字符的低4位的值必须是0x09，查表得出字符')'、'9'、'I'、'Y'、'i'、'y'符合条件。
- 第二字字符的低4位的值必须是0x0f，查表得出字符'/'、'？'、'O'、'_'、'o'符合条件。
- 第三个字符的低4位的值必须是0x0e，查表得出字符'.'、'>'、'N'、'^'、n'、'~'符合条件。
- 第四个字符的低4位的值必须是0x05，查表得出字符'%'、'5'、'E'、'U'、'e'、'u'符合条件。
- 第五个字符的低4位的值必须是0x06，查表得出字符'&'、'6'、'F'、'V'、'f'、'v'符合条件。
- 第六个字符的低4位的值必须是0x07，查表得出字符'''、'7'、'G'、'W'、'g'、'w'符合条件。
- 4 3 2 1 6 5

ENDING

问题解决（By hjq）

Question：

mov %fs:0x28,%rax 不知道在干啥，有知道的可以告诉我下，据说是gcc做的一个栈保护检测机制

mov %rax,0x18(rsp)这里我怀疑是把%rax的值给到了rsp的24位置偏移的地方，但是这个是在干啥我没想明白

xor %eax %eax 这里在干啥我也没搞清楚，但不妨碍我做题…

Answer：

这段话的意思大概是说 FS: %0x28 存储了一个特殊的标记堆栈的保护值，执行堆栈的保护检查，后面常常有 FS: %0x28 处原始值的XOR操作。若两值相等，则返回零，因为异或操作在两相同值之间的操作结果为0；相反，若是XOR操作不为0，则跳到一个特殊的处理堆栈错误的函数，因为两值不相同表明寄存器损坏了，导致存储在堆栈上的标记值发生了变化。

彩蛋

听闻这个Boomlab实验还有隐藏实验…大家可以试试…