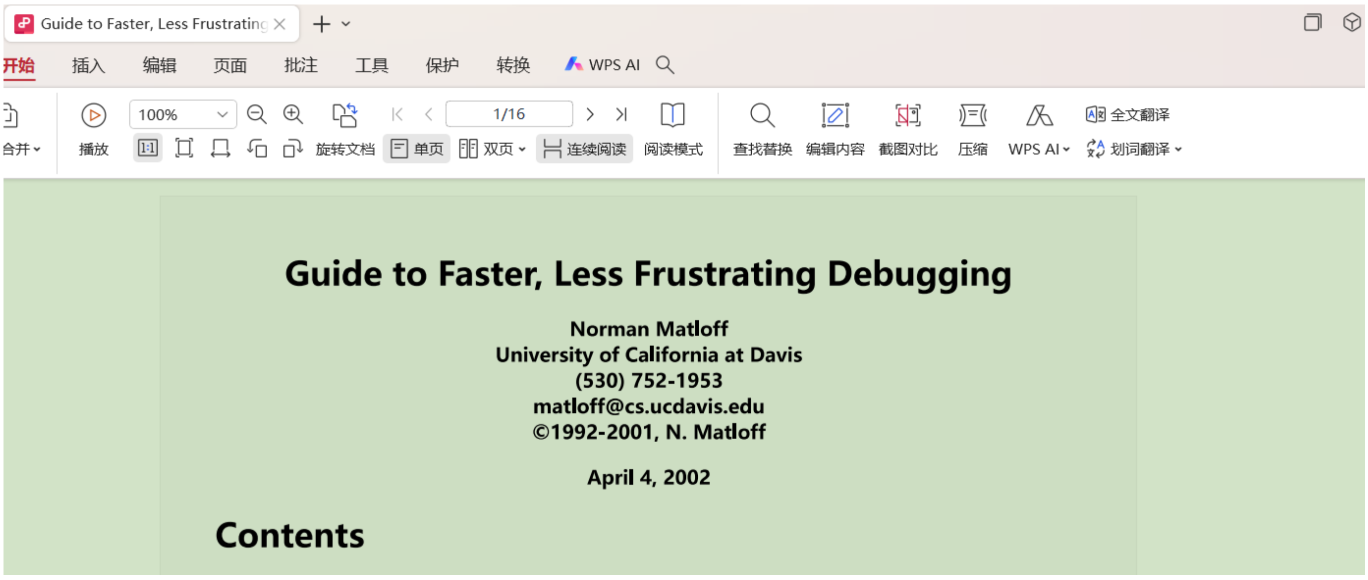
lab2

2024/3/24

PB22111702 李岱峰

一.实验准备

阅读说明文档，发现需要使用gdb调试，故找到校方的gdb调试summary，共17页打印阅读。



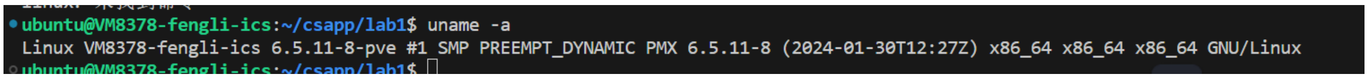
Guide to Faster, Less Frustrating Debugging

Norman Matloff
University of California at Davis
(530) 752-1953
matloff@cs.ucdavis.edu
©1992-2001, N. Matloff

April 4, 2002

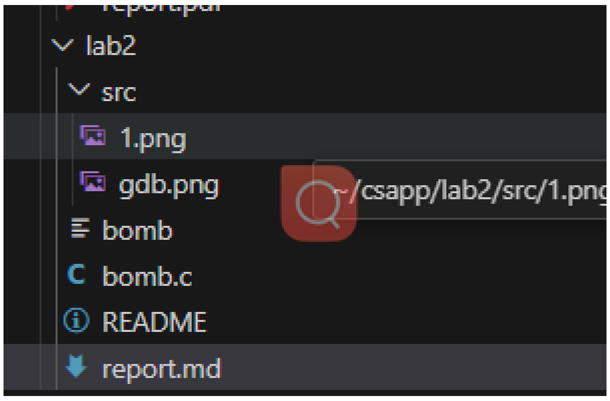
Contents

linux环境准备就绪，配置如下：



```
ubuntu@VM8378-fengli-ics:~/csapp/lab1$ uname -a
Linux VM8378-fengli-ics 6.5.11-8-pve #1 SMP PREEMPT_DYNAMIC PMX 6.5.11-8 (2024-01-30T12:27Z) x86_64 x86_64 x86_64 GNU/Linux
ubuntu@VM8378-fengli-ics:~/csapp/lab1$
```

文件已经就绪，文件树如下：

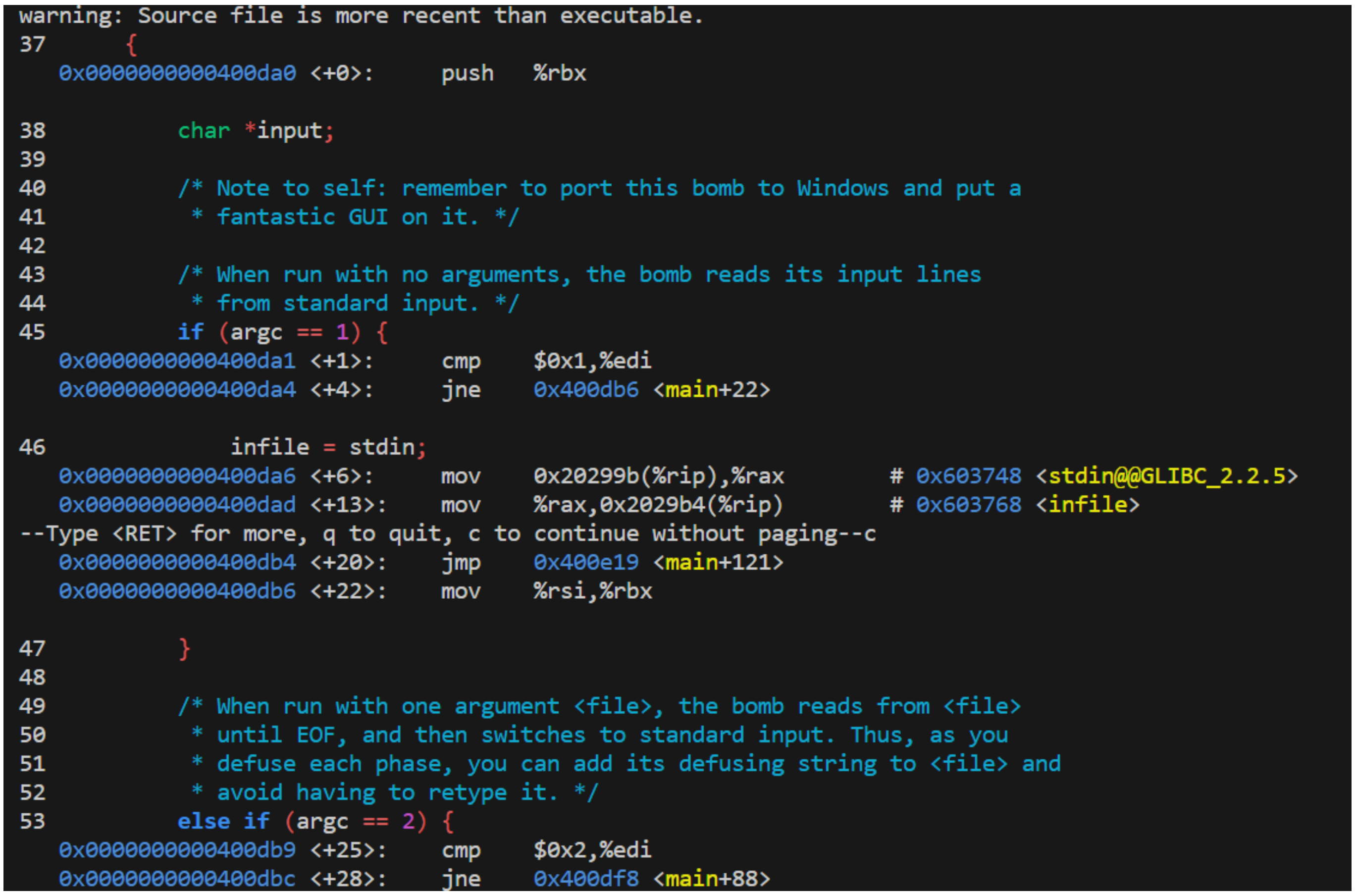


```
lab2
  src
    1.png
    gdb.png
  bomb
  bomb.c
  README
  report.md
```

二.实验过程

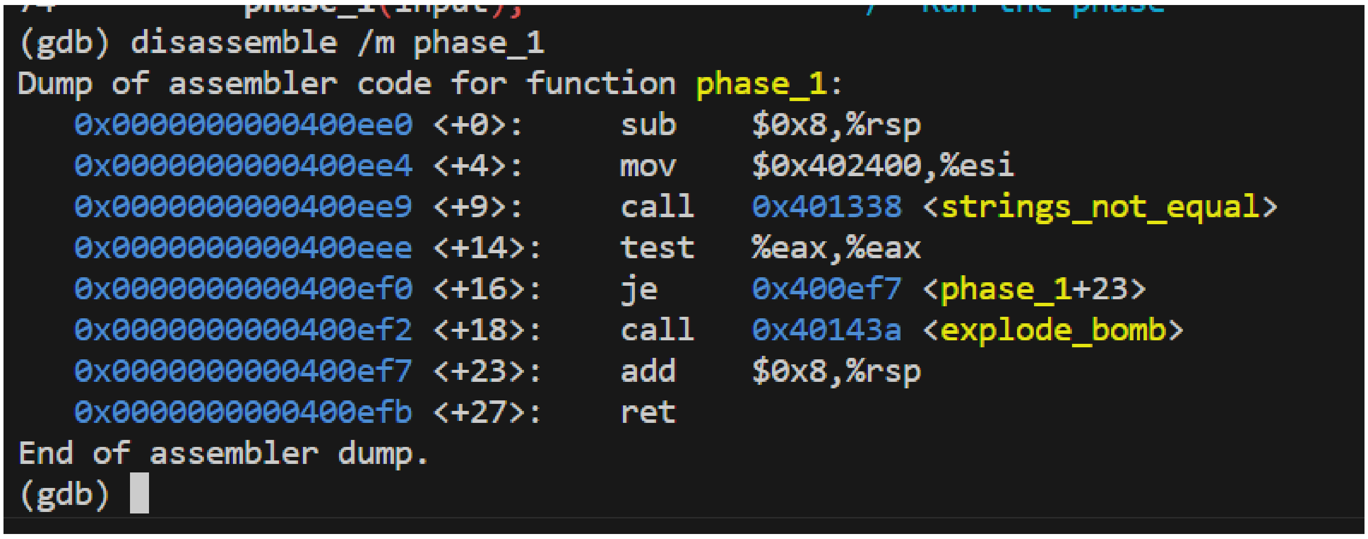
1. 尝试

首先我尝试使用gdb disassemble /m main ，看到了main函数的汇编代码，然而这没有什么用



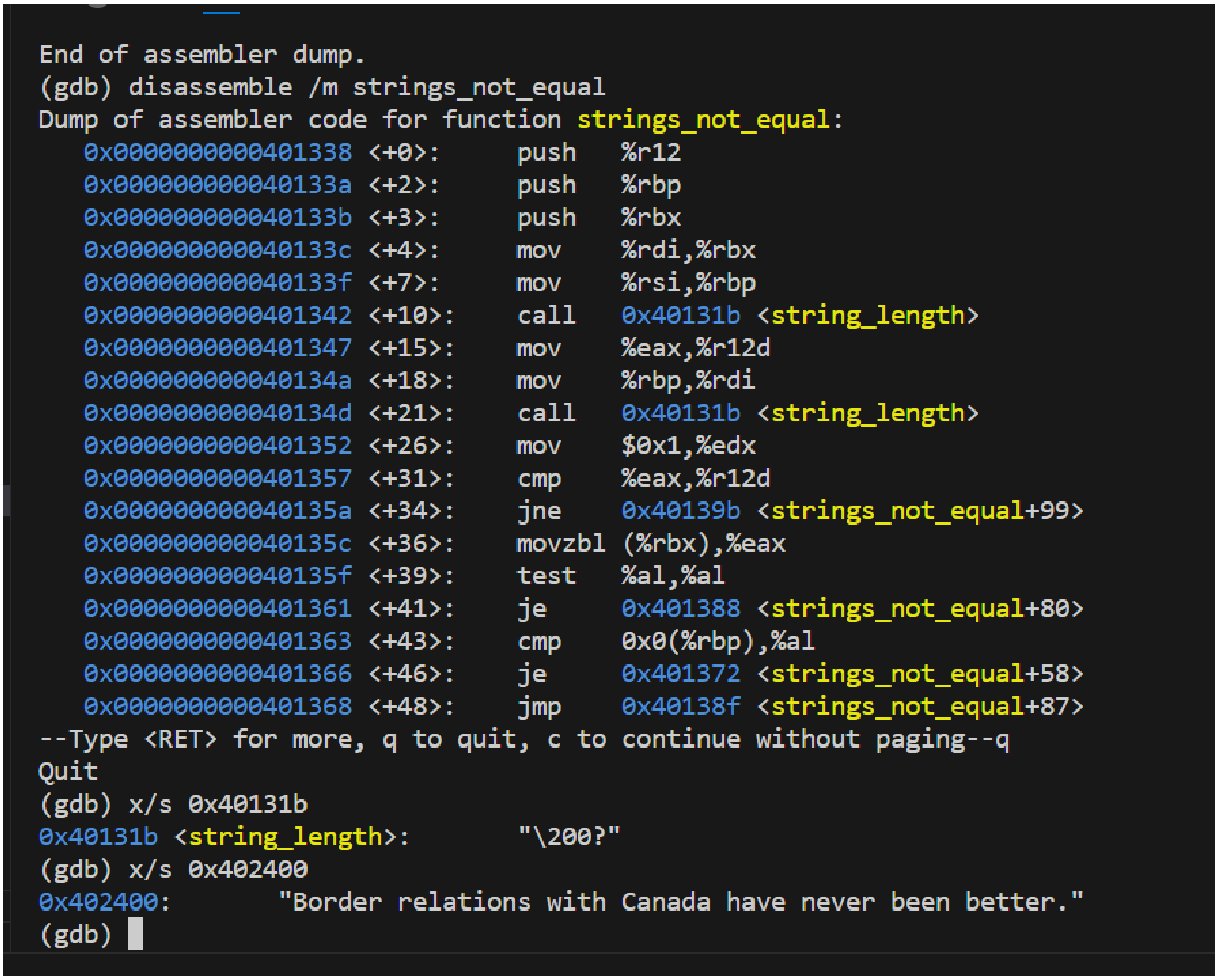
```
warning: Source file is more recent than executable.
37      {
   0x0000000000400da0 <+0>:     push   %rbx

38          char *input;
39
40          /* Note to self: remember to port this bomb to Windows and put a
41           * fantastic GUI on it. */
42
43          /* When run with no arguments, the bomb reads its input lines
44           * from standard input. */
45          if (argc == 1) {
   0x0000000000400da1 <+1>:     cmp    $0x1,%edi
   0x0000000000400da4 <+4>:     jne    0x400db6 <main+22>

46              infile = stdin;
   0x0000000000400da6 <+6>:     mov    0x20299b(%rip),%rax        # 0x603748 <stdin@@GLIBC_2.2.5>
   0x0000000000400dad <+13>:    mov    %rax,0x2029b4(%rip)        # 0x603768 <infile>
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000400db4 <+20>:    jmp    0x400e19 <main+121>
   0x0000000000400db6 <+22>:    mov    %rsi,%rbx

47          }
48
49          /* When run with one argument <file>, the bomb reads from <file>
50           * until EOF, and then switches to standard input. Thus, as you
51           * defuse each phase, you can add its defusing string to <file> and
52           * avoid having to retype it. */
53          else if (argc == 2) {
   0x0000000000400db9 <+25>:    cmp    $0x2,%edi
   0x0000000000400dbc <+28>:    jne    0x400df8 <main+88>
```

2. 解析字符串1



```
(gdb) disassemble /m 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.
(gdb)
```

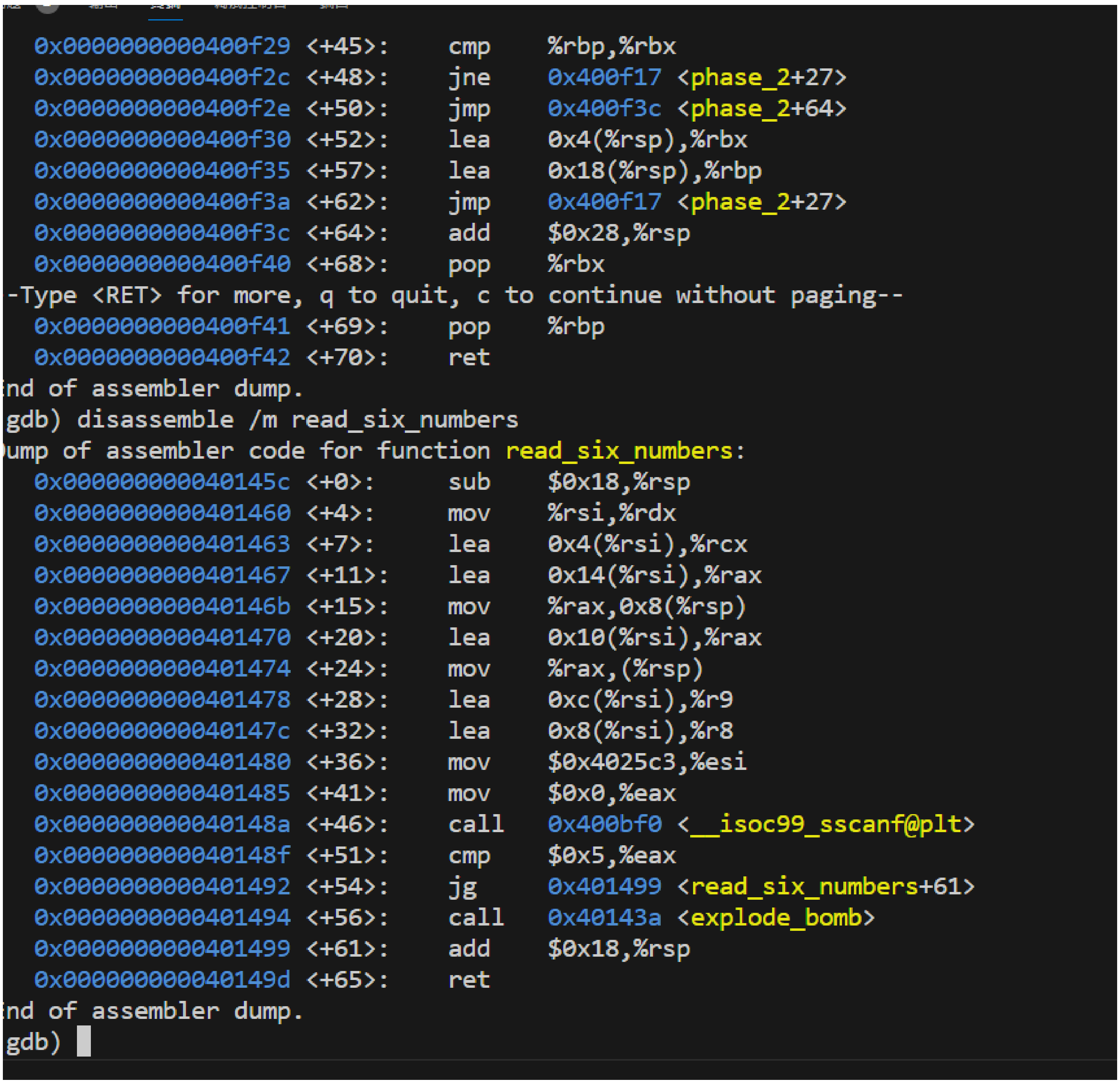
尝试解析一个未知的函数，得到如上图结果，可以得知第一个炸弹的输入函数是将一个字符串传入，然后call了一个比较字符串的函数，test检验函数然后得到结果。



```
End of assembler dump.
(gdb) disassemble /m strings_not_equal
Dump of assembler code for function strings_not_equal:
   0x0000000000401338 <+0>:     push   %r12
   0x000000000040133a <+2>:     push   %rbp
   0x000000000040133b <+3>:     push   %rbx
   0x000000000040133c <+4>:     mov    %rdi,%rbx
   0x000000000040133f <+7>:     mov    %rsi,%rbp
   0x0000000000401342 <+10>:    call   0x40131b <string_length>
   0x0000000000401347 <+15>:    mov    %eax,%r12d
   0x000000000040134a <+18>:    mov    %rbp,%rdi
   0x000000000040134d <+21>:    call   0x40131b <string_length>
   0x0000000000401352 <+26>:    mov    $0x1,%edx
   0x0000000000401357 <+31>:    cmp    %eax,%r12d
   0x000000000040135a <+34>:    jne    0x40139b <strings_not_equal+99>
   0x000000000040135c <+36>:    movzbl (%rbx),%eax
   0x000000000040135f <+39>:    test   %al,%al
   0x0000000000401361 <+41>:    je     0x401388 <strings_not_equal+80>
   0x0000000000401363 <+43>:    cmp    0x0(%rbp),%al
   0x0000000000401366 <+46>:    je     0x401372 <strings_not_equal+58>
   0x0000000000401368 <+48>:    jmp    0x40138f <strings_not_equal+87>
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) x/s 0x40131b
0x40131b <string_length>:       "\200?"
(gdb) x/s 0x402400
0x402400:       "Border relations with Canada have never been better."
(gdb) 
```

查看第一个密码。阅读该函数我们得知，phase_1从0x402400处mov了一个字符串上来，调用了字符串比较程序(call 0x401338) esi寄存器中就是等待被比较的字符串，

3. 解析字符串2



```
   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
-Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000400f41 <+69>:    pop    %rbp
   0x0000000000400f42 <+70>:    ret
ind of assembler dump.
gdb) disassemble /m read_six_numbers
ump 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
ind of assembler dump.
gdb) 
```

仔细阅读代码。

阅读教材，发现该函数逻辑为：rsp减0x28(分配栈)，将栈帧给rsi，然后读6个数字，最后比较rsp栈帧指向的数据是否为1，经过某些比较后，程序有跳转到phase_2+27和phase_2+52、phase_2+41、phase_2+64。其中+41是add指令，这里会进行栈帧的移动，移动到下一个内存空间，然后进行+45行的某种比较判断。整体是一个循环过程，猜测为比较6个数字，一个一个比较。

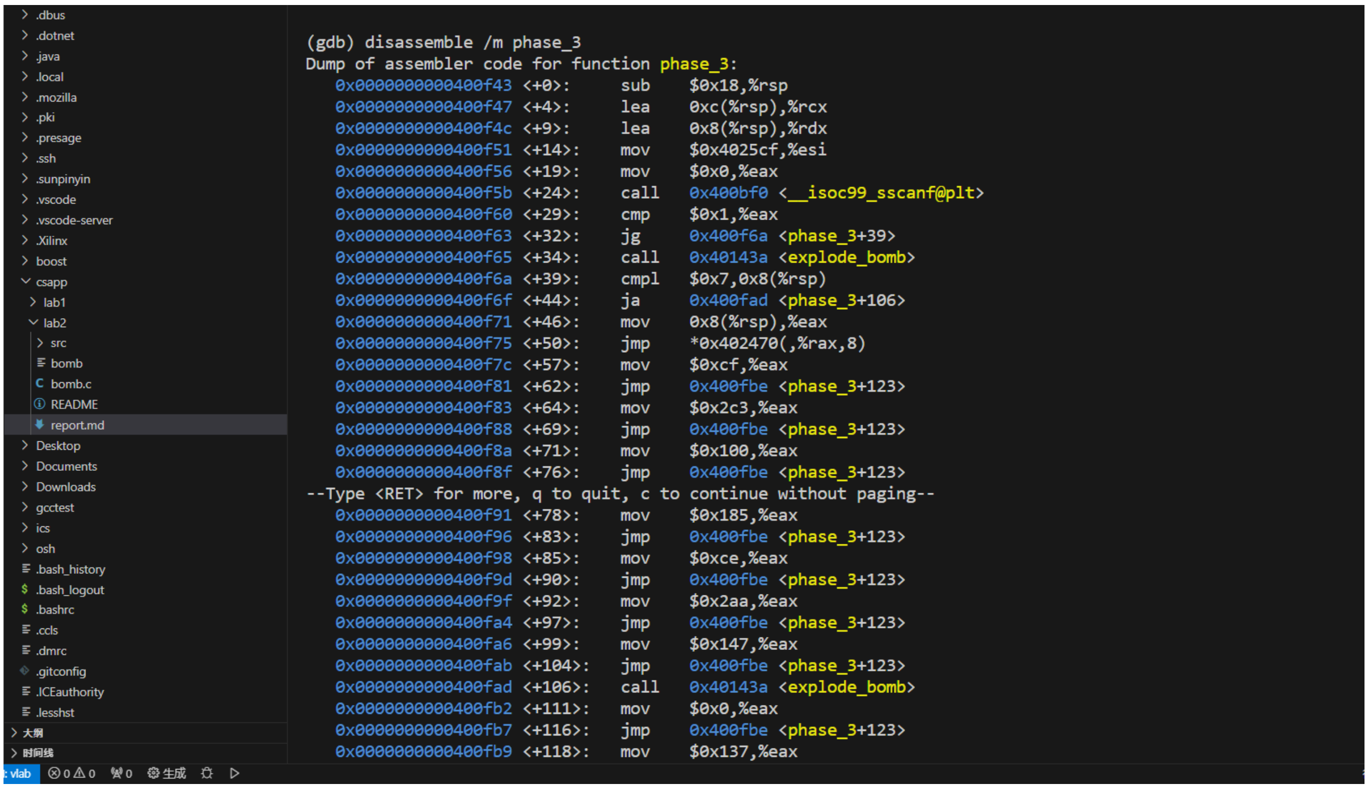
观察到循环过程为在读完数字后:+18 跳转 +52 顺序 +62 跳转 +27 顺序 +34 跳转 +41 顺序 +48 跳转回+27. 当6个数字读完后，+48后不再跳转+27 而是+50跳转+64，收回分配的栈空间。

根据循环，发现其中是在对%eax,(%rbx)在进行数值比较，是对%rbp和%rbx进行判段栈是否为空。

每次循环过程中，都有add eax+eax，说明密码中相邻数字扩大一倍，而从数字1开始(+14行说明)

得到密码：1 2 4 8 16 32

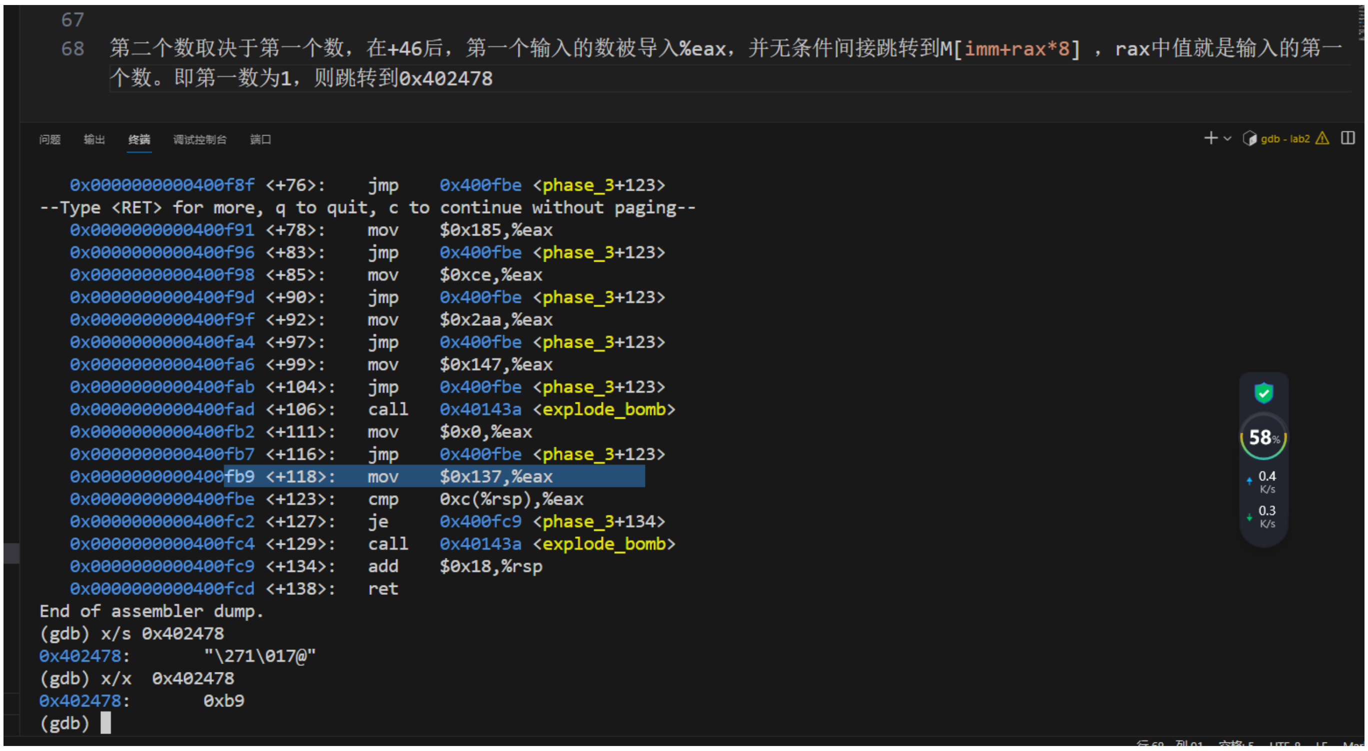
4. 解析字符串3



```
(gdb) disassemble /m 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>
--Type <RET> for more, q to quit, c to continue without paging--
   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
```

该程序中发现了大量的跳转，但每一个跳转都向下运行，指向phase_3+123,到达+123后是一个cmp比较，然后结束。说明输入应该是两个数，第一个数在+39中，比较0x7与0x8(%rsp)，如果输入的第一个数大于7，就引爆炸弹+106，所以第一个数是一个小于7的数

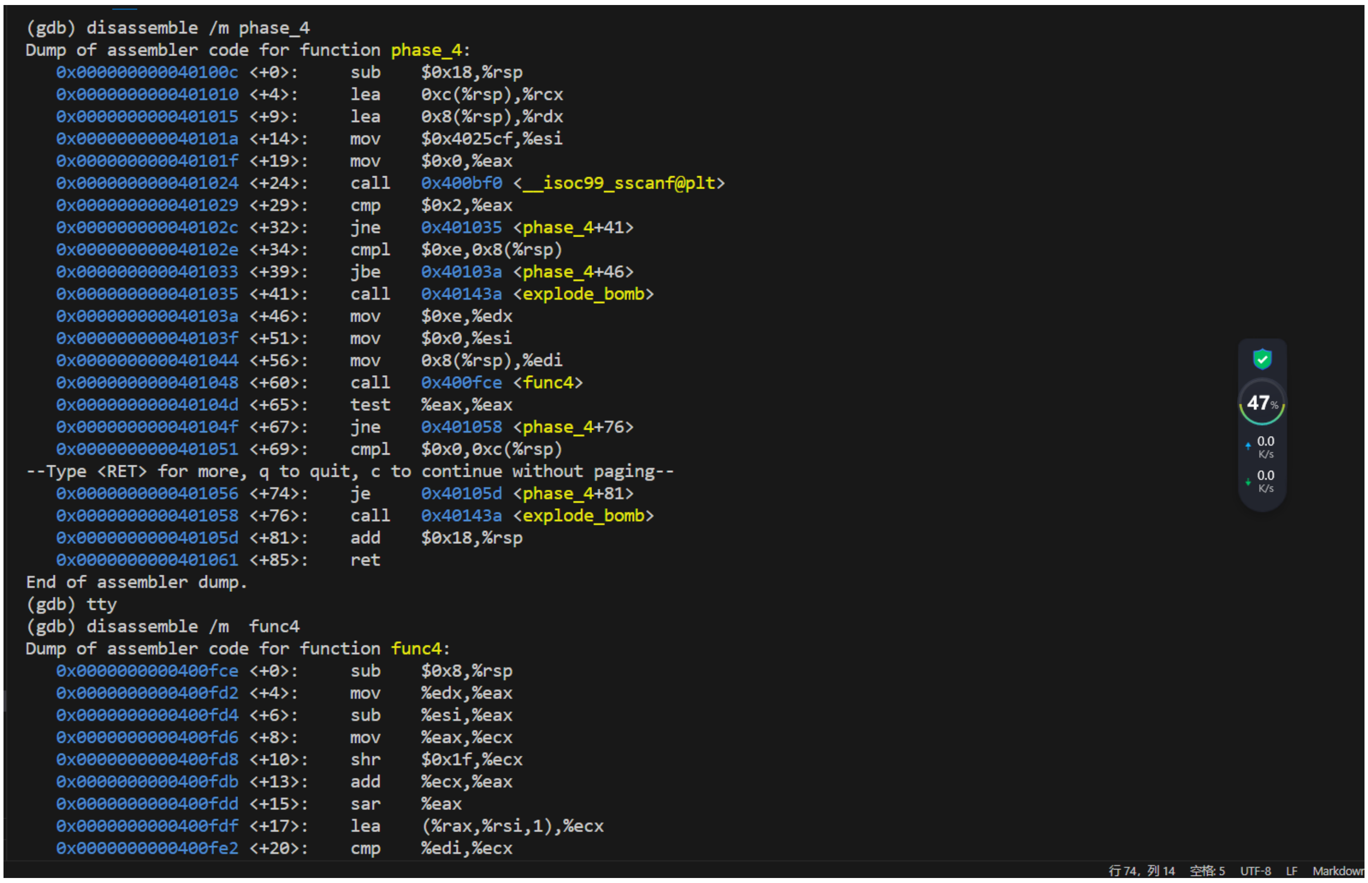
第二个数取决于第一个数，在+46后，第一个输入的数被导入%eax，并无条件间接跳转到M[imm+rax*8] 以这里面的值作为跳转目标，rax中值就是输入的第一个数。即第一数为1，则跳转到0x402478，观察到如下图，0x402478中存储的值是0x4024b9，跳转到+118，发现是move指令，给eax寄存器写入了0x137.如此第二个数就是0x137


第二个数取决于第一个数，在+46后，第一个输入的数被导入%eax，并无条件间接跳转到M[imm+rax*8] ，rax中值就是输入的第一个数。即第一数为1，则跳转到0x402478

```
   0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
--Type <RET> for more, q to quit, c to continue without paging--
   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.
(gdb) x/s 0x402478
0x402478:       "\271\017@"
(gdb) x/x  0x402478
0x402478:       0xb9
(gdb)
```

故密码为:1 311 (不唯一)

5. 解析字符串4

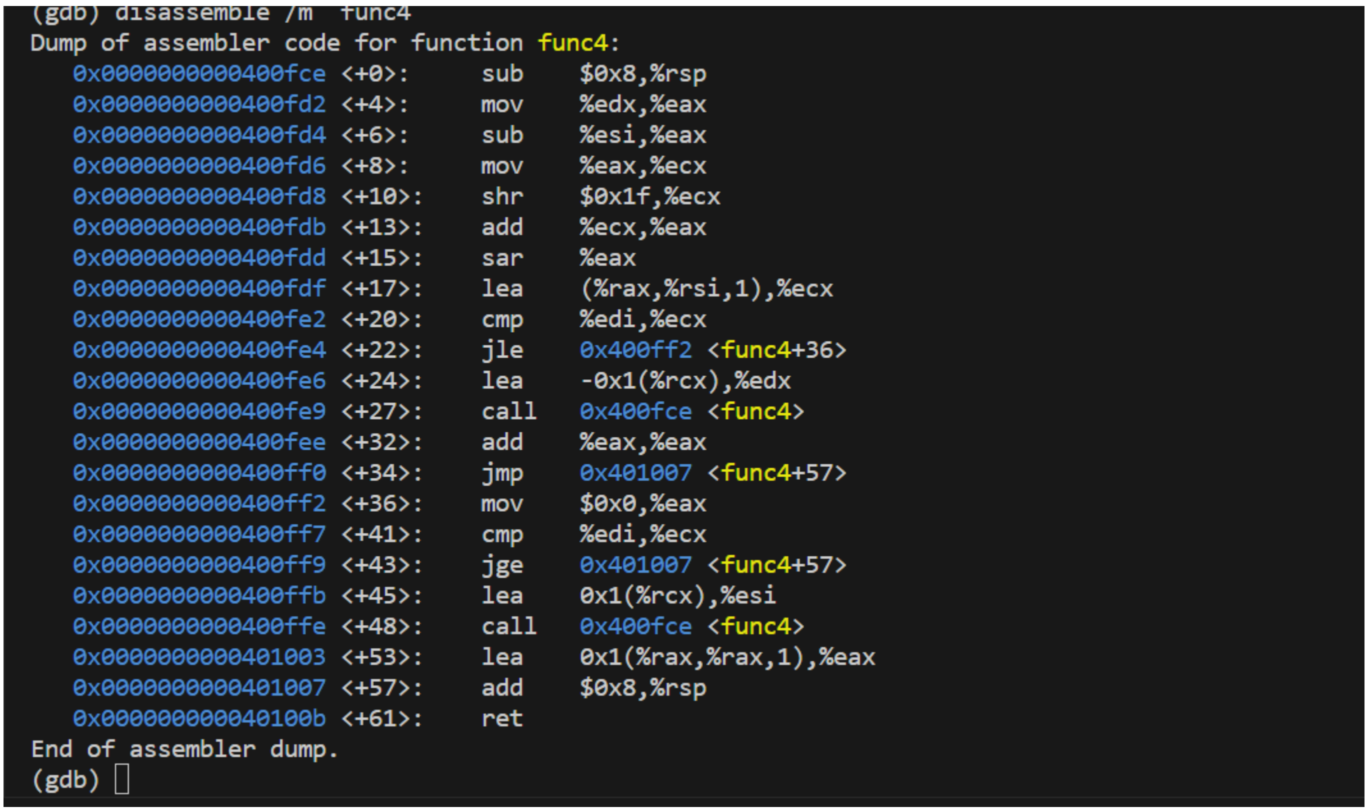


```
(gdb) disassemble /m phase_4
Dump of assembler code for function phase_4:
   0x000000000040100c <+0>:     sub    $0x18,%rsp
   0x0000000000401010 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000401015 <+9>:     lea    0x8(%rsp),%rdx
   0x000000000040101a <+14>:    mov    $0x4025cf,%esi
   0x000000000040101f <+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)
--Type <RET> for more, q to quit, c to continue without paging--
   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.
(gdb) tty
(gdb) disassemble /m  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
```

如图为bomb4的代码，同样的分配栈空间，从内存上提取数据，寄存器初始化后接收字符串，可以看到要求第一个数字小于等于0xe，并且第一个数字为%rdx ,第二个数字为%rcx,+29行标志一共有两个参数。

在+46行开始进行了三个mov，对edx，esi，edi进行了更改，此时在+60，%edx中是0xe，%esi是0x0,%edi中为%rdx即第一个数字。这三个东西作为func的参数被传入。在经过func函数后，判断eax中值是否为0，若不为0则爆炸，否则比较第二个数。若第二个数等于0x0则通过。

下面看func中的操作：



```
(gdb) disassemble /m  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.
(gdb) 
```

在func函数中，发现+27call了func，说明是一个递归函数。首先进入函数，分配空间，%eax=%edx=0xe,%eax=%eax-%esi=0xe,%ecx=%eax=0xe;

+10:%ecx>>>0x1f=>%ecx=0;(取符号)

%eax=%ecx+%eax=0xe；

%eax>>1=>0111(b)；

%ecx=(%rax+%rsi*1)=0x7；

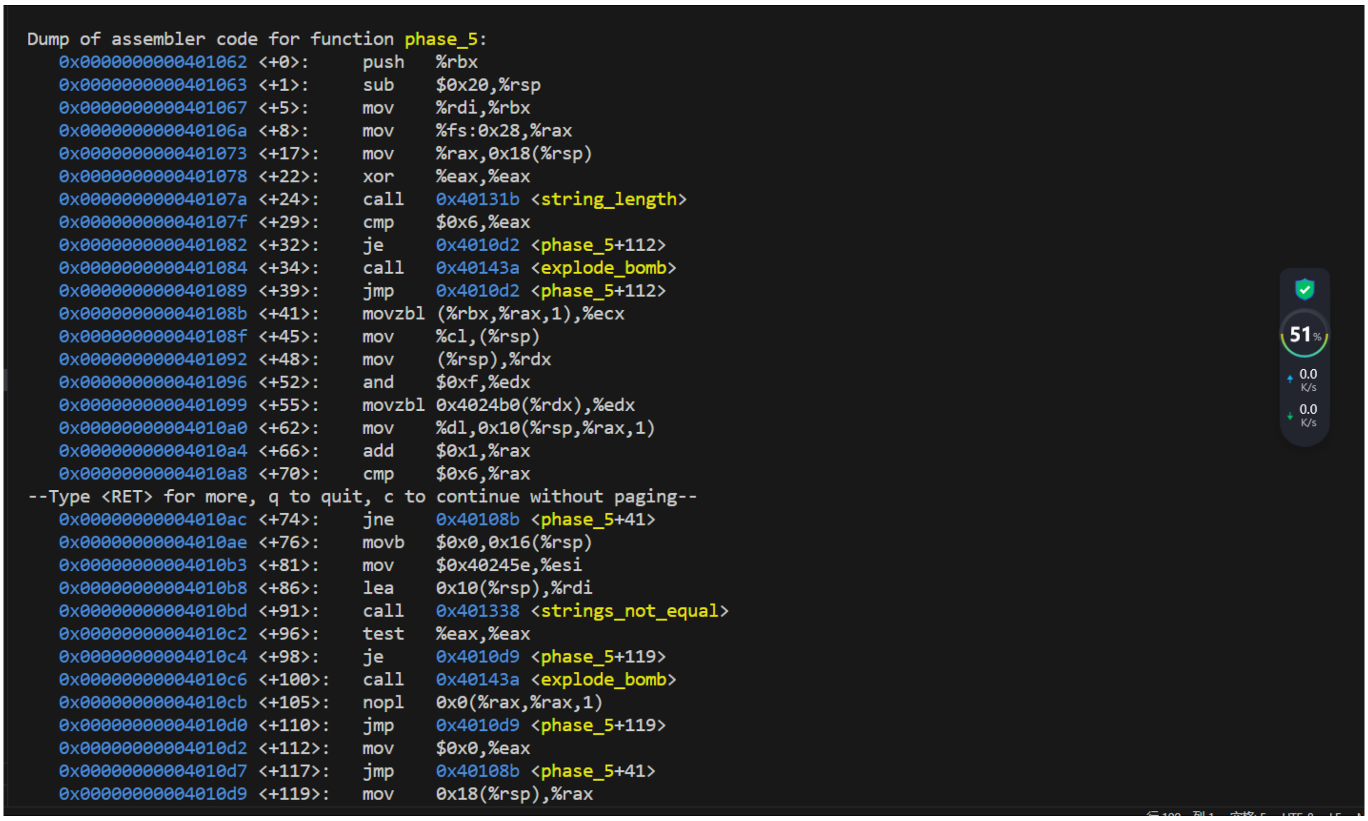
以上计算后，程序到达+20,是一个比较判断%edi,%ecx。根据上述，%edi是第一个数字，%ecx是0x7，如果两个值小于或相等，跳转+36。我们假设跳转成功，则%eax=0,比较%edi,%ecx，若大于或等于，继续跳转+57，函数结束ret。

根据上述发现0x7为递归终止条件(两个判断一个大于等于0x7，一个小于等于0x7)，则该函数入口参数，即第一个数字为0x7，返回phase_4.

如果递归没有进入，此时%eax为0x0，test结果为0，jne跳转不成立，进入cmpl比较，发现是0和第二个数进行比较，如果相等则跳转+81，成功。所以第二个操作数一定是0。

综上得到两个操作数：7 0 。

6. 解析字符串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
--Type <RET> for more, q to quit, c to continue without paging--
   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
```

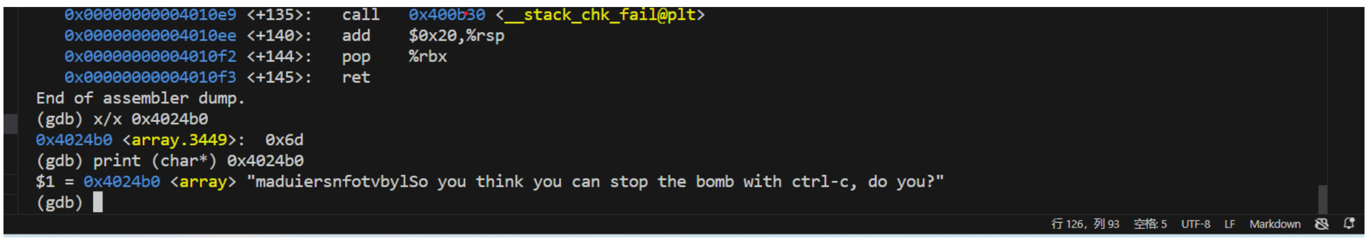
代码过长，不全文展示

首先观察到+29行，说明string_length应该为6。然后注意到+32行跳转+112行后再跳回+41行，是一个循环过程。循环的跳出条件在+98行，会将程序跳出至+119行，条件为%eax为0。综上，程序在+41行和+74行间循环，读入6个字符串后由+76行开始进行，调用字符串比较函数，如果%eax为0即字符串相等，到+119行进行栈越界检查，然后程序结束。

分析+41到+74的代码，总结如下：(在+41到+74内，%rax每次递增1，是计数器)

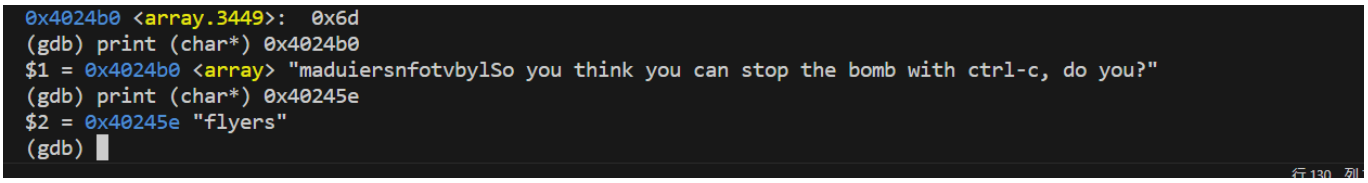
```
for(int rax=0;rax<=6;rax++){
     long a=c[rbx+rax*1]                      //这里会把a的高32位置0    movzbl (%rbx,%rax,1),%ecx
    char tmp=a[7:0]                          //
    //(rsp)=tmp tmp输入的第一个字符
    long rdx=tmp;                      //
    edx=edx&0xf //也就是只保存后4位          mov     (%rsp),%rdx   ; and $0xf,%edx
    edx=m[0x4024b0+rdx] //这里的rdx里保存的就是我们输入的第一个字符    movzbl 0x4024b0(%rdx),%edx
    m(rsp+10+rax)=edx    //低8位                 mov     %dl,0x10(%rsp,%rax,1)
}
```

这段代码的工作就是将输入的字符串一个一个取字符，然后将低八位存入rdx，再取后四位，加偏移量，在0x4024b0处找到相应位置，放到edx中，存入栈中。那么如下图，发现0x4024b0开始是一个字符串，显示"maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"。那么我们输入的字符的作用就是用这些字符的ascll码后4位的数值作为偏移量，找到相应位置的字符，来形成一个新字符串，存在栈rsp中。



```
    0x00000000004010e9 <+135>:   call   0x400b30 <__stack_chk_fail@plt>
    0x00000000004010ee <+140>:   add    $0x20,%rsp
    0x00000000004010f2 <+144>:   pop    %rbx
    0x00000000004010f3 <+145>:   ret
 End of assembler dump.
 (gdb) x/x 0x4024b0
 0x4024b0 <array.3449>:  0x6d
 (gdb) print (char*) 0x4024b0
 $1 = 0x4024b0 <array> "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
 (gdb) 
```

从上向下找密文所在位置，即提取出输入字符后，跟谁比较。从第+91行发现，在此调用比较函数，那在上面一定mov了一个字符串上来，调查0x40245e，



```
0x4024b0 <array.3449>:  0x6d
(gdb) print (char*) 0x4024b0
$1 = 0x4024b0 <array> "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
(gdb) print (char*) 0x40245e
$2 = 0x40245e "flyers"
(gdb) 
```

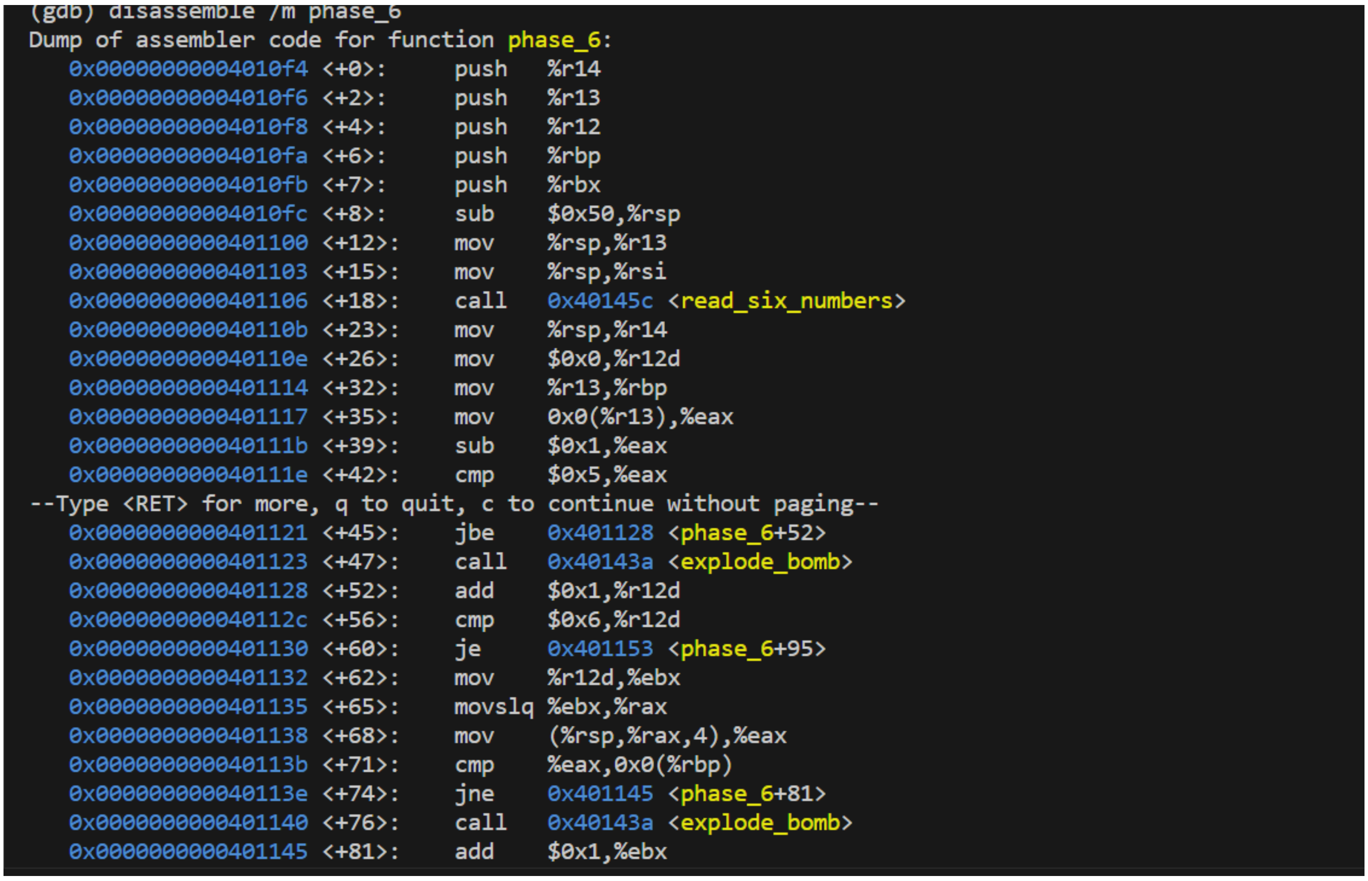
发现密文“flyers”。将每个字符在0x4024b0字符串出现位置(偏移量)找出，分别为：f=9,l=15,y=14,e=5,r=6,s=7. 则ascll码中后4位bit位这6个数字的字符就是密码。

密码：0100_1001:I;0100_1111:O;0100_1110:N;0100_0101:E;0100_0110:F;0100_0111:G

IONEFG

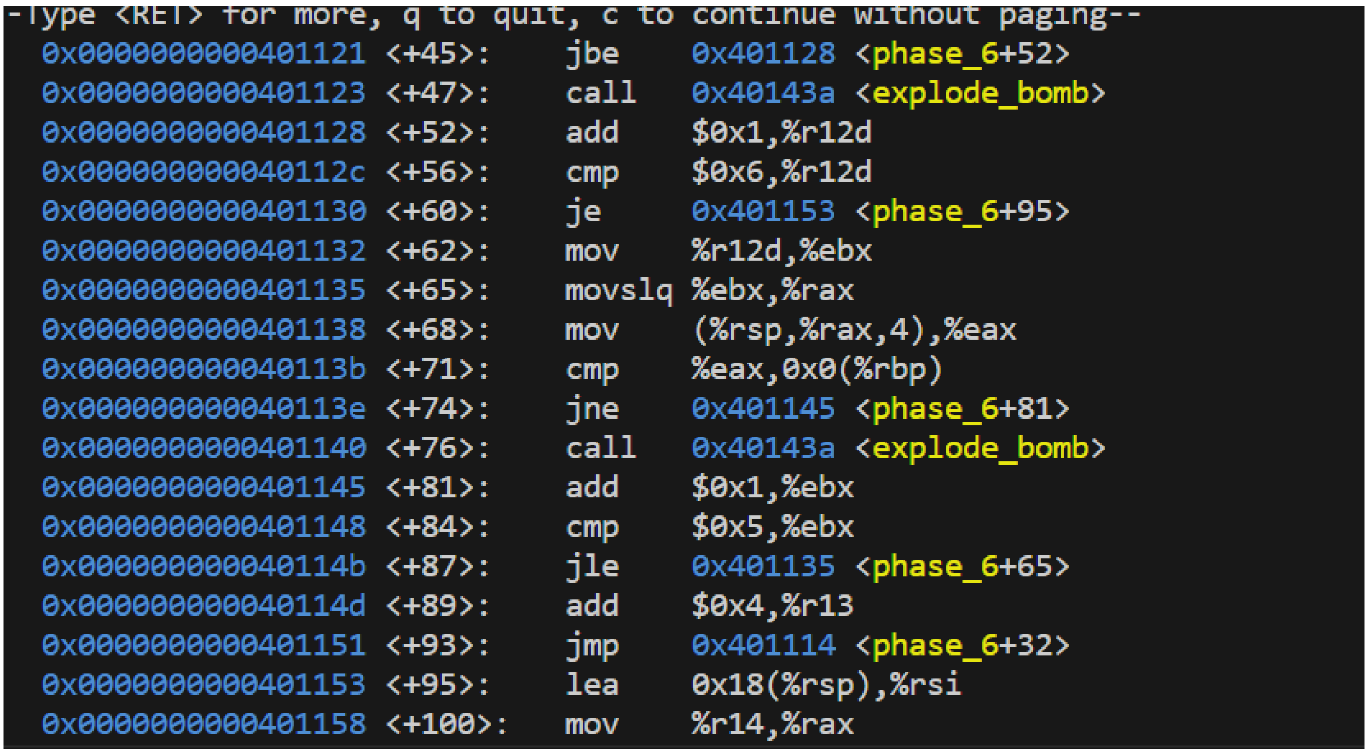
7. 解析字符串6

太长了。



```
 (gdb) disassemble /m 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
 --Type <RET> for more, q to quit, c to continue without paging--
    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
```

+18行揭示读入的是6个数字。这六个数字均不能大于6：(+39行)。+52、+56、+60是计数器，揭示着比较的出口。



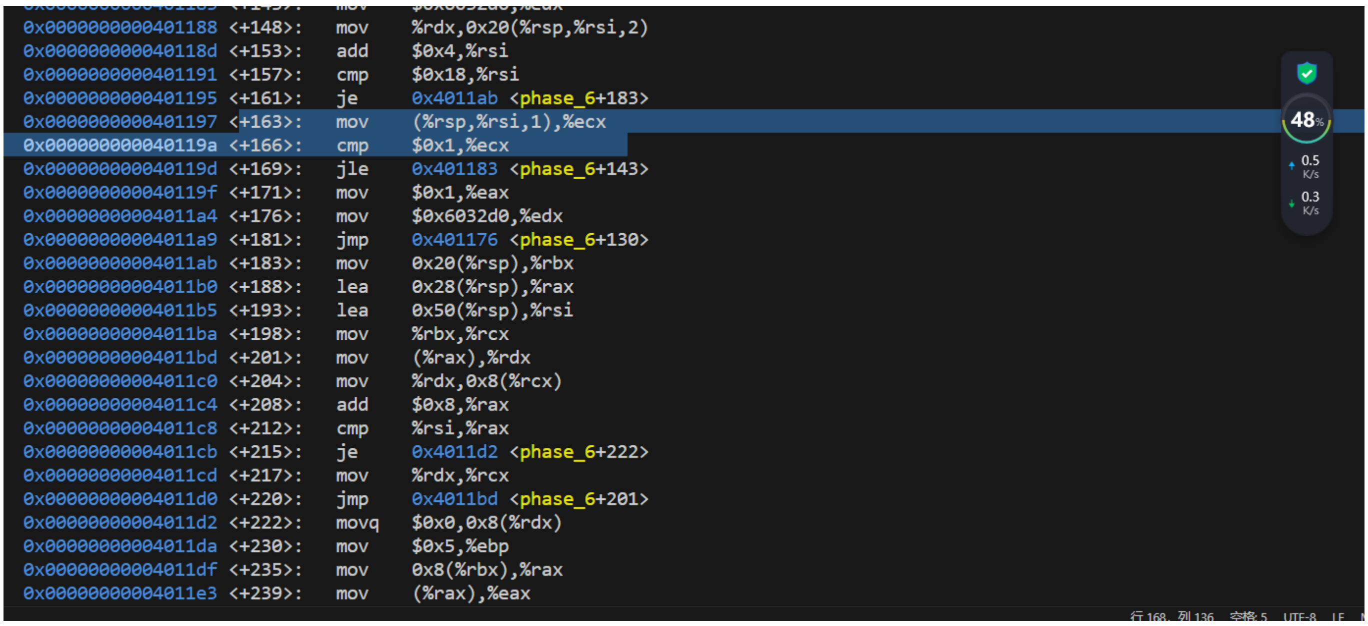
```
-Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000401121 <+45>:    jbe    0x401128 <phase_6+52>
   0x0000000000401123 <+47>:    call   0x40143a <explode_bomb>
   0x0000000000401128 <+52>:    add    $0x1,%r12d
   0x000000000040112c <+56>:    cmp    $0x6,%r12d
   0x0000000000401130 <+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
```

+74行揭示了两个数组空间内的值不能相同，而且这里是一个双重循环：

```
if(a[i]>6) bomb!
r12d+=1;                        //<+52>:    add    $0x1,%r12d
if (r12d=6){call <phase_6+95>}    //<+56\+60>
int ebx=r12d;                   //<+62>
while(ebx<=5) {                 //
  int rax=ebx;                    //<+65>    movslq %ebx,%rax
  int eax=a[rax*4+rsp];           //<+68>     mov    (%rsp,%rax,4),%eax
  if(eax==a[i]) bomb!;            //<+71><+74>
  else{
    ebx+=1;                         //<+81>:    add    $0x1,%ebx
  }
}
```

如此，这6个数均不能相同，且都小于等于6.然后跳转<+95>

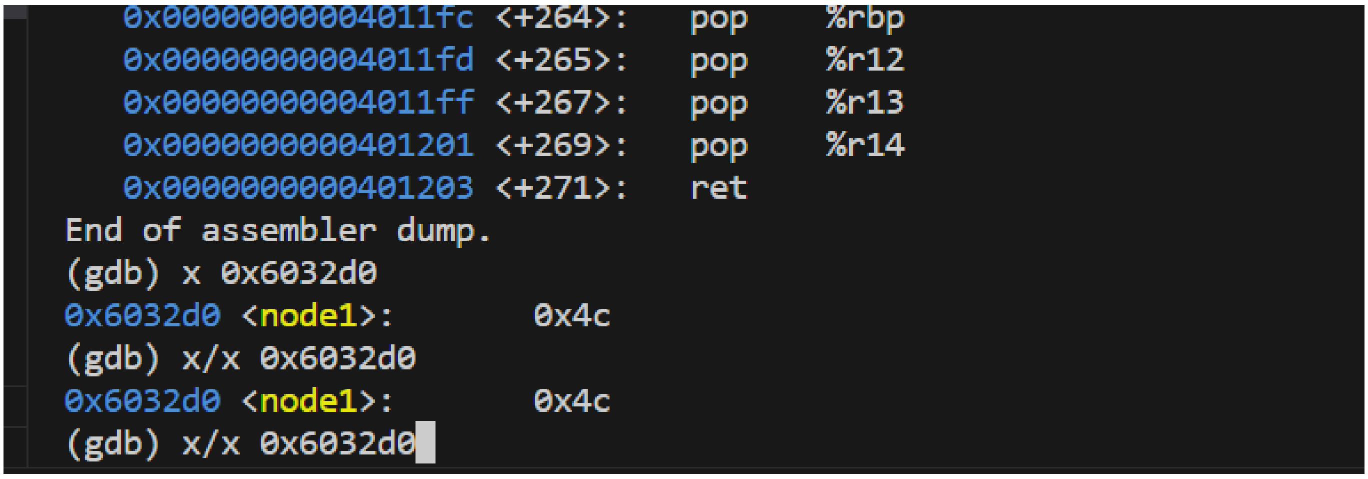
<+95>开始，又是一个循环，这个循环出口是rsi==rax,跳转<+163>.循环内进行了%edx的操作，%edx=7-%rax,又将edx值写入rax，就是说当rsi!=rax时(rsi是非法栈帧，就是说数组没越界时)，将r14=7-r14。进行完毕后进入<+163>



```
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
```

来之前，esi清零。mov将数组第一个元素写入ecx，如果是1<=该元素，跳回<+143>.<143>给edx一个神奇的值0x6032d0，将rdx写给一个神奇的地址，然后切换到数组下一个元素，比较是否越界后进入<+163>.可见这里有一个神奇的循环。

所以开始查看0x6032d0这个神奇的值的意思。如下图



```
   0x00000000004011fc <+264>:   pop    %rbp
   0x00000000004011fd <+265>:   pop    %r12
   0x00000000004011ff <+267>:   pop    %r13
   0x0000000000401201 <+269>:   pop    %r14
   0x0000000000401203 <+271>:   ret
End of assembler dump.
(gdb) x 0x6032d0
0x6032d0 <node1>:       0x4c
(gdb) x/x 0x6032d0
0x6032d0 <node1>:       0x4c
(gdb) x/x 0x6032d0
```

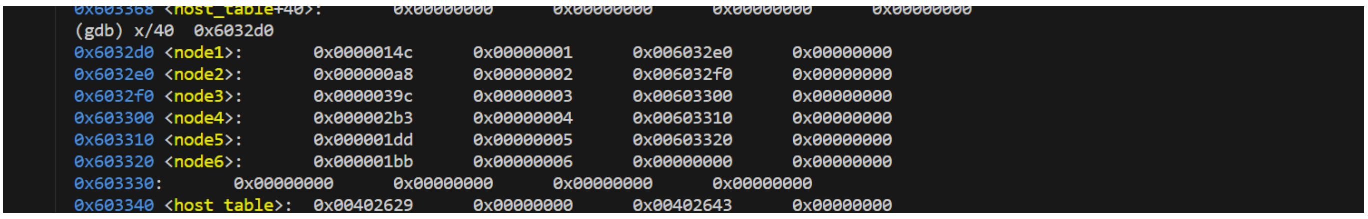
发现了node1，大概率是一个链表。而这个循环里给node附了值。从<+148>到<+169>讲述了给数组重排的规则.而<+181>到<+257>给出了链表连接的方式，具体逻辑我已伪代码形式给出：

```
for(int i=1;i<=6;i++){
  if(a[i]==6)
      L[i]=node1; //L[i]表示我们新链表的第i个结点。
  else{
      int b=7-a[i],p=node1;
    while(b--){
      p=p->next;
    }
    L[i]=p;
  }
}
```

故而，重拍揭示了规则，经过上述逻辑重拍后，会得到全新的链表，并且由<+243>行注意，重拍后的链表一定是单调递减的。

总逻辑：我输入6个数，这六个数满足不大于6且互相不同，6个数会存放在一个给定的链表中，按照L[i]=node[7-a[i]]的方式，将旧链表中的数赋给新链表，并使新链表的结构满足数值单调递减。

6个节点存储的初始值情况如下：



```
0x603368 <host_table+40>:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) x/40  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
```

观察到了6个节点的链表，发现排完序的链表应是node3>node4>node5>node6>node1>node2（比的是0x6032d0+n*0x10）

可以得到输入的序列应是

L[1]=node3=node[7-a[1]] a[1]=4;

L[2]=node4=node[7-a[2]] a[2]=3;

L[3]=node5=node[7-a[3]] a[3]=2;

L[4]=node6=node[7-a[4]] a[4]=1;

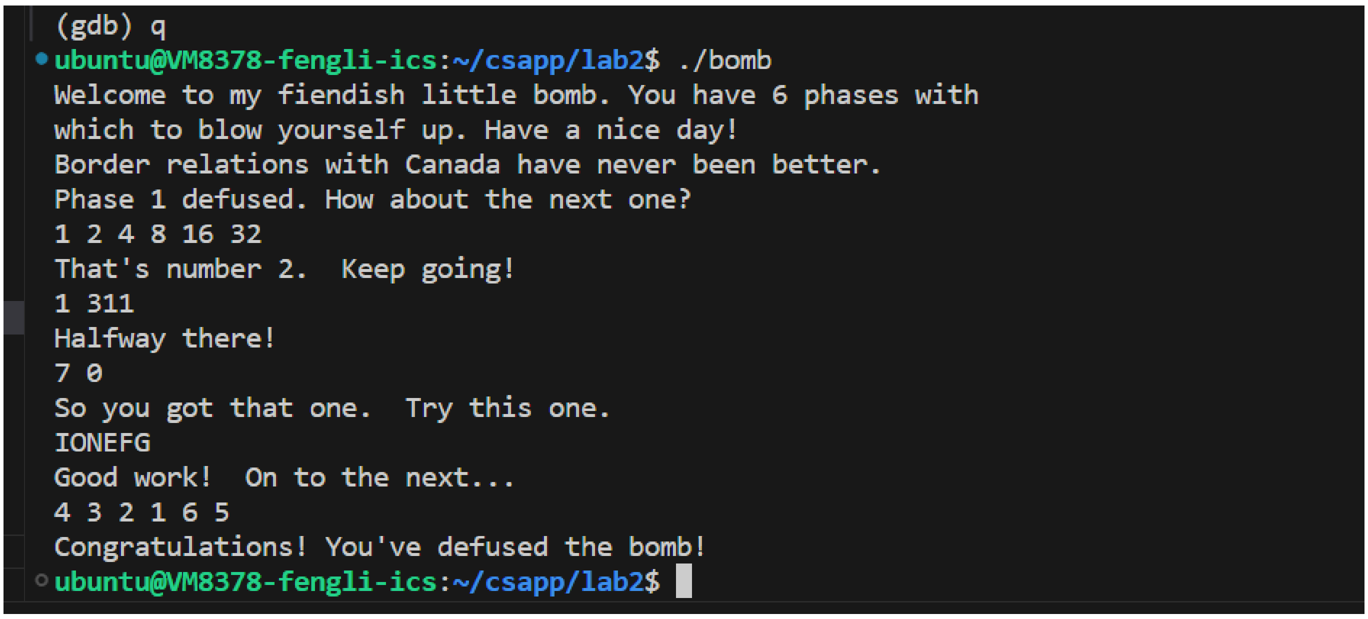
L[5]=node1=6;

L[6]=node2=node[7-a[6]] a[1]=5;

密码为4 3 2 1 6 5

实验结果和总结

1. 实验结果



```
(gdb) q
ubuntu@VM8378-fengli-ics:~/csapp/lab2$ ./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!
7 0
So you got that one.  Try this one.
IONEFG
Good work!  On to the next...
4 3 2 1 6 5
Congratulations! You've defused the bomb!
ubuntu@VM8378-fengli-ics:~/csapp/lab2$
```

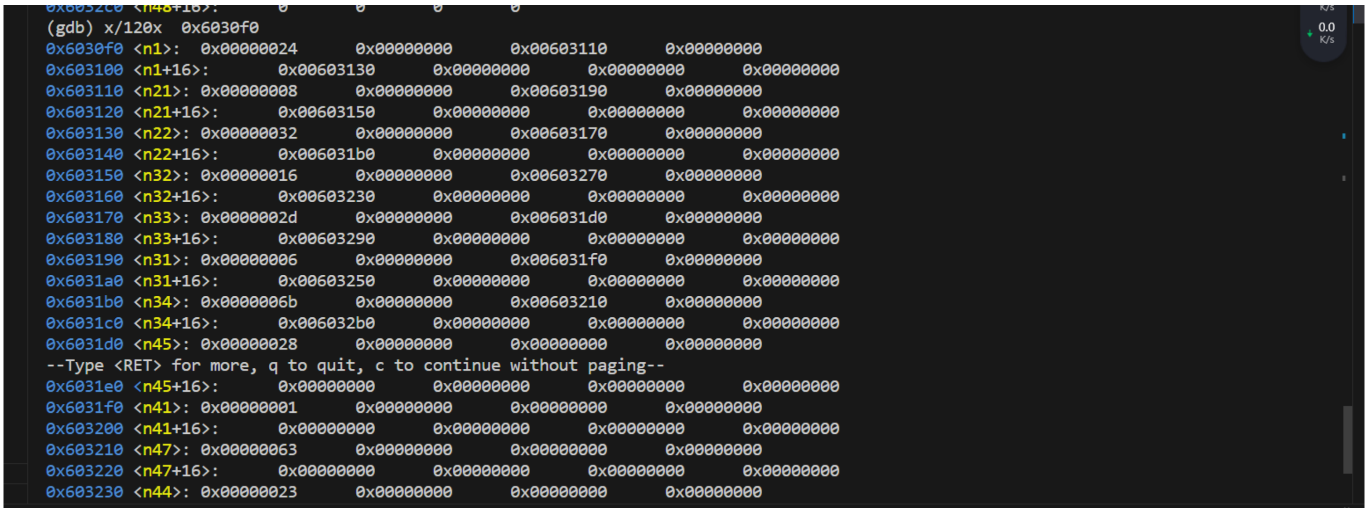
如上图，炸弹解除。

2. 总结

因为没有接触过x86这样的复杂指令集，从实验开始到结束用了整整两天。客观来讲，收获是极大的，x86堪称2天速成，各种取址方式都用到了，各种跳转，空间分配都见过了，甚至还有链表的存在，让我对链表的结构有了更深层次的认识。

该实验任务量大，难度高，出题诡异，但很有趣，收获多。

甚至在阅读解析时还发现有隐藏关卡，树的存在......



```
0x6032c0 <n48+16>:      0       0       0       0
(gdb) x/120x  0x6030f0
0x6030f0 <n1>:  0x00000024      0x00000000      0x00603110      0x00000000
0x603100 <n1+16>:       0x00603130      0x00000000      0x00000000      0x00000000
0x603110 <n21>: 0x00000008      0x00000000      0x00603190      0x00000000
0x603120 <n21+16>:      0x00603150      0x00000000      0x00000000      0x00000000
0x603130 <n22>: 0x00000032      0x00000000      0x00603170      0x00000000
0x603140 <n22+16>:      0x006031b0      0x00000000      0x00000000      0x00000000
0x603150 <n32>: 0x00000016      0x00000000      0x00603270      0x00000000
0x603160 <n32+16>:      0x00603230      0x00000000      0x00000000      0x00000000
0x603170 <n33>: 0x0000002d      0x00000000      0x006031d0      0x00000000
0x603180 <n33+16>:      0x00603290      0x00000000      0x00000000      0x00000000
0x603190 <n31>: 0x00000006      0x00000000      0x006031f0      0x00000000
0x6031a0 <n31+16>:      0x00603250      0x00000000      0x00000000      0x00000000
0x6031b0 <n34>: 0x0000006b      0x00000000      0x00603210      0x00000000
0x6031c0 <n34+16>:      0x006032b0      0x00000000      0x00000000      0x00000000
0x6031d0 <n45>: 0x00000028      0x00000000      0x00000000      0x00000000
--Type <RET> for more, q to quit, c to continue without paging--
0x6031e0 <n45+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x6031f0 <n41>: 0x00000001      0x00000000      0x00000000      0x00000000
0x603200 <n41+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x603210 <n47>: 0x00000063      0x00000000      0x00000000      0x00000000
0x603220 <n47+16>:      0x00000000      0x00000000      0x00000000      0x00000000
0x603230 <n44>: 0x00000023      0x00000000      0x00000000      0x00000000
```

这未免太极端了。但原来树是这样存的。