

# 计算机科学与技术学院计算机系统基础课程实验报告

实验题目：拆除二进制炸弹		学号：202000130198
班级：20.4	姓名：隋春雨	
Email：2018640800@qq.com		
<b>实验目的：</b> <ol style="list-style-type: none"><li>1. 熟悉汇编语言命令与 MIPS 指令集</li><li>2. 熟悉 LINUX 环境</li><li>3. 根据反汇编程序分析功能</li><li>4. 熟悉 GDB 调试工具</li></ol>		
<b>实验软件和硬件环境：</b> vmware ubuntu		
<b>实验原理和方法：</b> 结合指令集对汇编语言进行理解 使用 GDB 调试工具查看寄存器和内存的值 结合反汇编程序分析功能		
<b>实验步骤：</b> <ol style="list-style-type: none"><li>1. 本篇实验报告相关的知识前备和一些疑难问题已经上传到我的博客上，博客链接为 <a href="https://blog.csdn.net/capsnever/article/details/121524876">https://blog.csdn.net/capsnever/article/details/121524876</a>  欢迎关注 我的博客部分图片如下：</li></ol>		
		

stack如何保存现场？

### 1. 保护现场

现场/上下文相当于案发现场，总有一些案发现场，要记录下来，否则被别人破坏，便无法恢复。而此处说的现场，是指CPU运行时，用到的一些寄存器，比如r0,r1等，对于这些寄存器的值，如果不保存而直接跳转到子函数中执行，其很可能被破坏，因为其函数执行也要用到这些寄存器。因此，在函数调用之前，应该将这些寄存器等现场暂时保存(入栈push)，等调用函数执行完毕后出栈(pop)再恢复现场。这样CPU就可以正确的继续执行了。

保存寄存器的值，一般用push指令，将对应的某些寄存器的值，一个个放到栈中，即所谓的压栈。然后待被调用的子函数执行完毕后再调用pop，把栈中的一个一个的值，赋值给对应的那些你刚开始压栈时用到的寄存器，把对应的值从栈中弹出去，即所谓的出栈。

其中保存的寄存器中，也包括lr的值（因为用bl指令进行跳转的话，之前的pc值存在lr中），在子程序执行完毕后，再把栈中的lr值pop出来，赋值给pc，这样就实现了子函数的正确的返回。

CSDN @Capsfly

常见寄存器：

寄存器特点

编号	助记符	用法
0	zero	永远为0

## 2. 知识预备

Wrote by CS 20.4 隋春雨

参考文章：[常见指令](#)

[MIPS 指令官方文档](#)

[gdb 调试](#)

stack 如何保存现场？

### 1. 保护现场

现场/上下文相当于案发现场，总有一些案发现场，要记录下来，否则被别人破坏，便无法恢复。而此处说的现场，是指CPU运行时，用到的一些寄存器，比如r0,r1等，对于这些寄存器的值，如果不保存而直接跳转到子函数中执行，其很可能被破坏，因为其函数执行也要用到这些寄存器。因此，在函数调用之前，应该将这些寄存器等现场暂时保存(入栈push)，等调用函数执行完毕后出栈(pop)再恢复现场。这样CPU就可以正确的继续执行了。

保存寄存器的值，一般用push指令，将对应的某些寄存器的值，一个个放到栈中，即所谓的压栈。然后待被调用的子函数执行完毕后再调用pop，把栈中的一个一个的值，赋值给对应的那些你刚开始压栈时用到的寄存器，把对应的值从栈中弹出去，即所谓的出栈。

其中保存的寄存器中，也包括lr的值（因为用bl指令进行跳转的话，之前的pc值存在lr中），在子程序执行完毕后，再把栈中的lr值pop出来，赋值给pc，这样就实现了子函数的正确的返回。

CSDN @Capsfly

常见寄存器：

寄存器特点		
编号	助记符	用法
0	zero	永远为0
1	at	用做汇编器的暂时变量
2-3	v0,v1	子函数调用返回结果
4-7	a0-a3	子函数调用的参数
8-15	t0-t7	暂时变量，子函数使用时不需要保存与恢复
24-25	t8-t9	暂时变量，子函数使用时不需要保存与恢复
16-23	s0-s7	子函数寄存器变量。在返回之前子函数必须保存和恢复使用过的变量，从而调用函数知道这些寄存器的值没有变化
26-27	k0,k1	通常被中断或异常处理程序使用作为保存一些系统参数
28	gp	全局指针。一些运行系统维护这个指针来更方便的存取static和extern变量
29	sp	堆栈指针
30	s8/fp	第9个寄存器变量。/ 框架指针
31	ra	子函数的返回地址

[https://blog.csdn.net/qq\\_41191281](https://blog.csdn.net/qq_41191281)  
CSDN @Capsfly

# 1. 第一关

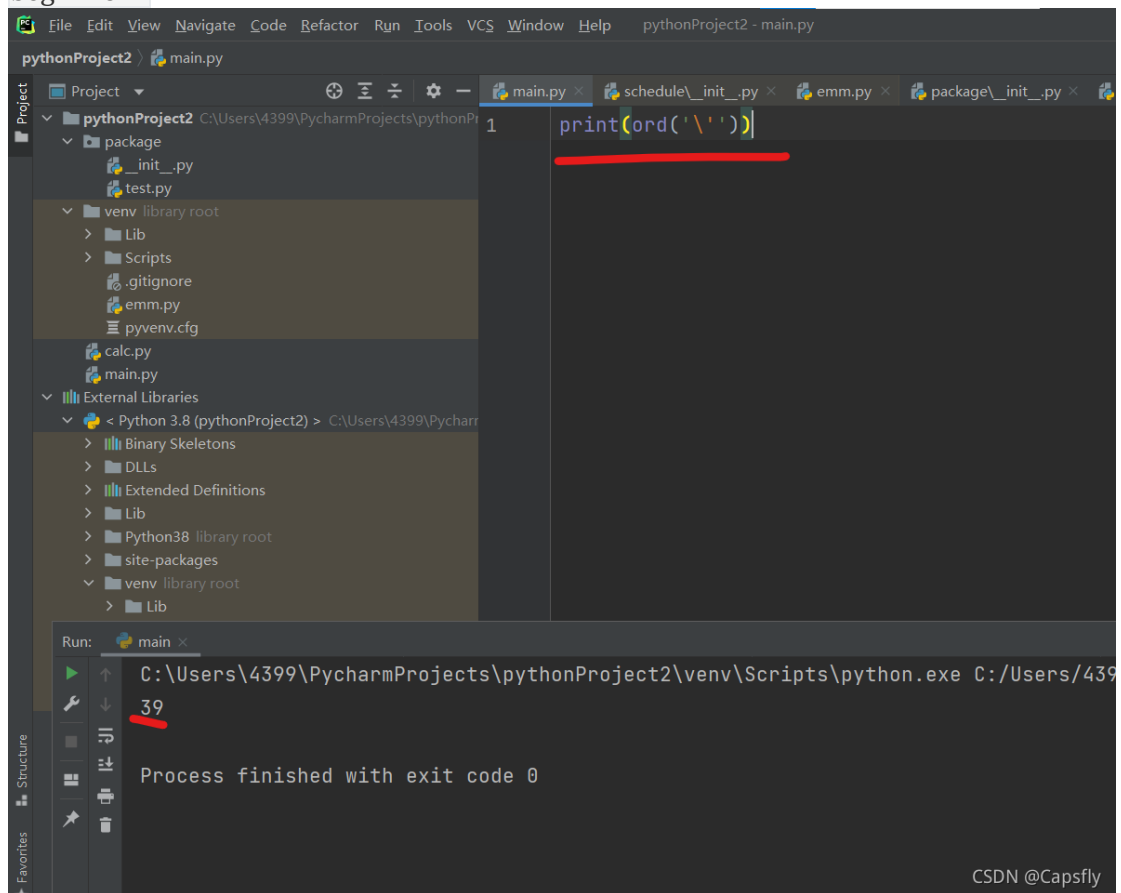
```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0x7f7c9cd0 in ?? ()
(gdb) disas phase_1
Dump of assembler code for function phase_1:
    0x00400d6c <+0>:      addiu    sp,sp,-32
    0x00400d70 <+4>:      sw        ra,28(sp)
    0x00400d74 <+8>:      sw        s8,24(sp)
    0x00400d78 <+12>:     move     s8,sp
    0x00400d7c <+16>:     sw        a0,32(s8)
    0x00400d80 <+20>:     lw        a0,32(s8)
    0x00400d84 <+24>:     lui       v0,0x40
    0x00400d88 <+28>:     addiu    a1,v0,10092
    0x00400d8c <+32>:     jal       0x401cf8 <strings_not_equal>
    0x00400d90 <+36>:     nop
    0x00400d94 <+40>:     beqz     v0,0x400da4 <phase_1+56>
    0x00400d98 <+44>:     nop
    0x00400d9c <+48>:     jal       0x4021f0 <explode_bomb>
    0x00400da0 <+52>:     nop
    0x00400da4 <+56>:     move     sp,s8
    0x00400da8 <+60>:     lw        ra,28(sp)
    0x00400dac <+64>:     lw        s8,24(sp)
    0x00400db0 <+68>:     addiu    sp,sp,32
    0x00400db4 <+72>:     jr        ra
    0x00400db8 <+76>:     nop
End of assembler dump.
(gdb) s
```

CSDN @Capsfly

从 beqz 指令那里可以看出，如果是字符串相等的话，那么我们就直接跳到 0x400da4，可以看到 0x00400d9c 是炸弹爆炸的地方，所以这一关的要点就变成了，查出原来的函数中保存的 string 的内容，然后输入即可

```
0x40276c:      76 'L' 101 'e' 116 't' 39 '\\' 115 's' 32 ' ' 98 'b' 101 'e'
0x402774:      103 'g' 105 'i' 110 'n' 32 ' ' 110 'n' 111 'o' 119 'w' 33 '!'
(gdb) █
```

本来我最开始看到这个\就不太理解,后来用 pycharm 输出了一下'的 asc 码发现就是 39,才发现是哪个'太小了没看到,所以最终拆除炸弹的结果就是 Let's begin now!



PS: [GDB 中 X 的用法](#)

# GDB中x的使用语法

原创

SkYe231\_

2020-01-17 01:00:30

646

★ 收藏 3

分类专栏: C语言

文章标签: gdb

内存

查看内存



C语言 专栏收录该内容

## 简介

x 用于在 gdb 中查看内存的内容

格式: x /nuf <addr>

## 说明

- x 是 examine 的缩写
- n 表示要显示的内存单元的个数
- u 表示一个地址单元的长度:
  - b 表示单字节
  - h 表示双字节
  - w 表示四字节

CSDN @Capsfly

x/16c \$a0 类似于事后诸葛亮了，因为这是我们在知道这个结果是 16 位的情况下，才能 x/16c \$a0,才平时做实验的时候，可以多扩展几位看一看结果

## 2. 第二关

```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
Do you need "set solib-search-path" or "set sysroot"?
^C^CThe target is not responding to interrupt requests.
Stop debugging it? (y or n) y
Disconnected from target.
(gdb) disas phase_2
Dump of assembler code for function phase_2:
0x00400dbc <+0>:      addiu    sp,sp,-64
0x00400dc0 <+4>:      sw       ra,60(sp)
0x00400dc4 <+8>:      sw       s8,56(sp)
0x00400dc8 <+12>:     move     s8,sp
0x00400dcc <+16>:     lui      gp,0x42
0x00400dd0 <+20>:     addiu    gp,gp,-20080
0x00400dd4 <+24>:     sw       gp,16(sp)
0x00400dd8 <+28>:     sw       a0,64(s8)
0x00400ddc <+32>:     addiu    v0,s8,28
0x00400de0 <+36>:     lw       a0,64(s8)
0x00400de4 <+40>:     move     a1,v0
0x00400de8 <+44>:     jal      0x401ba8 <read_six_numbers>
0x00400dec <+48>:     nop
0x00400df0 <+52>:     lw       gp,16(s8)
0x00400df4 <+56>:     lw       v1,28(s8)
0x00400df8 <+60>:     li       v0,1
0x00400dfc <+64>:     beq     v1,v0,0x400e10 <phase_2+84>
0x00400e00 <+68>:     nop
0x00400e04 <+72>:     jal      0x4021f0 <explode_bomb>
0x00400e08 <+76>:     nop
0x00400e0c <+80>:     lw       gp,16(s8)
0x00400e10 <+84>:     li       v0,1
---Type <return> to continue, or q <return> to quit---
```

CSDN @Capsfly

li

### load immediate:

```
li      register_destination, value
```

#load immediate value into destination register

顾名思义，这里的 li 意为 load immediate

CSDN @Capsfly

\$v0:子函数调用返回的结果 参考了一下 [关于 mips-fp（帧指针）寄存器的理解](#) 和 [MIPS 汇编角度看 C 语言的指针](#) 和 [mips 中 gp 寄存器的用法](#) 简单来说，用 fp 保存栈底的位置 fp:栈基址寄存器

值得注意的是：0x00400dc0 <+4>: sw ra,60(sp) 这条指令，按照 mips 官方文档的解释，是这样的

**Format:** SW rt, offset(base)

**Purpose:** Store Word

To store a word to memory.

**Description:** memory[GPR[base] + offset] ← GPR[rt]

个人感觉可能用 `lw` 更合适一点，为了验证一下是否保存的是返回地址，找了一下资料

```
34  ## Function int factorial(int n)
35  factorial:
36      ## YOUR CODE HERE
37      addi $sp,$sp,-8          #adjust stack for 2 items
38      sw   $ra,4($sp)         #save return address
39      sw   $a0,0($sp)         #save the argument n
40
41      slti $t0,$a0,1          #if n < 1, then set $t0 as 1
42      beq  $t0,$zero,L1       #if equal, then jump L1
43                                     #above all, if n >= 1, then jump L1
44      #if(n < 1)
45      addi $v0,$zero,1        #return 1
46      addi $sp,$sp,8          #pop 2 items off stack
47      jr   $ra                #return to caller
48      #else
49      L1:
50          add $a0,$a0,-1       #argument :n - 1
51          jal factorial        #call factorial with (n-1)
52
53          lw   $a0,0($sp)      #restore argument n
54          lw   $ra,4($sp)      #restore address
55          addi $sp,$sp,8       #adjust stack pionter
56          mul  $v0,$a0,$v0     #return n * factorial(n-1)
57          jr   $ra            #return to caller
58      ## END OF YOUR CODE
59      #jr $ra
```

CSDN @Capsfly

可以看出作者认为，它保存的确实是返回地址。。。。Ok。。。。那就先这样

- 对于 `move` 指令，查找了一下资料，可以发现 `move $a0 $a1` 的含义就是把 `$a1` 的值送到 `$a0` 寄存器

# Difference between "move" and "li" in MIPS assembly language

Asked 8 years ago   Active 6 years, 11 months ago   Viewed 134k times



35



I was practicing converting C code into MIPS assembly language, and am having trouble understanding the usage of `move` and `li` in variable assignment.

For example, to implement the following C line in MIPS:



```
int x = 0;
```

10



If I understand it correctly (I highly doubt this, though), it looks like both of these work in MIPS assembler:

```
move $s0, $zero
li $s0, $zero
```

Am I wrong? What is the difference between these two lines?

`c` `assembly` `mips`

CSDN @Capsf

## 1 Answer

Active

Oldest

Votes



58



The `move` instruction copies a value from one register to another. The `li` instruction loads a specific numeric value into that register.

For the *specific* case of zero, you can use either the constant zero or the zero register to get that:

```
move $s0, $zero
li $s0, 0
```



There's no register that generates a value other than zero, though, so you'd have to use `li` if you wanted some other number, like:

```
li $s0, 12345678
```

CSDN @Capsfly

## 4. 思考：为什么

```
0x00400dbc <+0>: addiu sp,sp,-64
0x00400dc0 <+4>: sw ra,60(sp)
0x00400dc4 <+8>: sw s8,56(sp)
0x00400dc8 <+12>: move s8,sp
0x00400dcc <+16>: lui gp,0x42
0x00400dd0 <+20>: addiu gp,gp,-20080
0x00400dd4 <+24>: sw gp,16(sp)
0x00400dd8 <+28>: sw a0,64(s8)
0x00400ddc <+32>: addiu v0,s8,28
0x00400de0 <+36>: lw a0,64(s8)
0x00400de4 <+40>: move a1,v0
0x00400de8 <+44>: jal 0x401ba8 <read_six_numbers>
0x00400dec <+48>: nop
```



```

0x00400df0 <+52>: lw gp,16(s8)
0x00400df4 <+56>: lw v1,28(s8)
0x00400df8 <+60>: li v0,1
0x00400dfc <+64>: beq v1,v0,0x400e10 <phase_2+84>
0x00400e00 <+68>: nop
0x00400e04 <+72>: jal 0x4021f0 <explode_bomb>
0x00400e08 <+76>: nop
0x00400e0c <+80>: lw gp,16(s8)
0x00400e10 <+84>: li v0,1
0x00400e14 <+88>: sw v0,24(s8)
0x00400e18 <+92>: b 0x400ea8 <phase_2+236>
0x00400e1c <+96>: nop
0x00400e20 <+100>: lw v0,24(s8)
0x00400e24 <+104>: nop

```

每条指令之间的间隔都是 4 呢？

解决：查了一下 MIPS 指令的构成发现，无论是 R、I、J 型指令，都是 32 位的，4 个字节，所以 MIPS 指令的间隔都是 4。同时查了一下 beqz 指令的用法 beqz \$v location:若 \$v 寄存器的值为 0，那么程序跳转到 location 在实验二中的 0x00400dfc <+64>: beq v1,v0,0x400e10 <phase\_2+84> 指令中

如果 \$v1==\$v0,那么跳转到 0x400e10,不会爆炸。

如果没有,那么一定会因为 0x00400e04 <+72>: jal 0x4021f0 <explode\_bomb> 而爆炸

CODE: SELECT ALL

```

location_0001:
....
beqz $t0, location_0002
lw $a0, 0($s1)
....

location_0002:
beqzl $t0, location_0001
lw $s0, 0($s1)
....

```

CSDN @Capsfly

到这里的时候，总是遇到 ---Type <return> to continue, or q <return> to quit---q 的问题，查了一下，发现设置一下不分页显示就好了 set pagination off

Dump of assembler code for function phase\_2:

```

0x00400dbc <+0>: addiu sp,sp,-64
0x00400dc0 <+4>: sw ra,60(sp)
0x00400dc4 <+8>: sw s8,56(sp)
0x00400dc8 <+12>: move s8,sp
0x00400dcc <+16>: lui gp,0x42
0x00400dd0 <+20>: addiu gp,gp,-20080
0x00400dd4 <+24>: sw gp,16(sp)
0x00400dd8 <+28>: sw a0,64(s8)
0x00400ddc <+32>: addiu v0,s8,28
0x00400de0 <+36>: lw a0,64(s8)
0x00400de4 <+40>: move a1,v0
0x00400de8 <+44>: jal 0x401ba8 <read_six_numbers>
0x00400dec <+48>: nop
0x00400df0 <+52>: lw gp,16(s8)
0x00400df4 <+56>: lw v1,28(s8)
0x00400df8 <+60>: li v0,1

```

```

0x00400dfc <+64>: beq v1,v0,0x400e10 <phase_2+84>
0x00400e00 <+68>: nop
0x00400e04 <+72>: jal 0x4021f0 <explode_bomb>
0x00400e08 <+76>: nop
0x00400e0c <+80>: lw gp,16(s8)
0x00400e10 <+84>: li v0,1
0x00400e14 <+88>: sw v0,24(s8)
0x00400e18 <+92>: b 0x400ea8 <phase_2+236>
0x00400e1c <+96>: nop
0x00400e20 <+100>: lw v0,24(s8)// 当前循环次数
0x00400e24 <+104>: nop
0x00400e28 <+108>: addiu v0,v0,-1//上一次的循环次数
0x00400e2c <+112>: sll v0,v0,0x2
0x00400e30 <+116>: addiu v1,s8,24//v1 保存的起始位置
0x00400e34 <+120>: addu v0,v1,v0//找到上一个元素的位置
0x00400e38 <+124>: lw a0,4(v0)//当前的数组中的元素放到了$a0
0x00400e3c <+128>: li v1,12
0x00400e40 <+132>: lw v0,24(s8)//循环次数
0x00400e44 <+136>: nop
0x00400e48 <+140>: subu v0,v1,v0//v0=12-循环次数
0x00400e4c <+144>: lw v1,-32660(gp)//v1=$gp-32660
0x00400e50 <+148>: sll v0,v0,0x2//v0=(12-循环次数)*4
0x00400e54 <+152>: addu v0,v1,v0//v0=$gp-32660+(12-循环次数)*4
0x00400e58 <+156>: lw v0,0(v0)//v0=Memory[v0]
0x00400e5c <+160>: nop
0x00400e60 <+164>: mult a0,v0
0x00400e64 <+168>: mflo a0 //这里是 a0 最后一次被改变的位置
0x00400e68 <+172>: lw v0,24(s8)//这里是 v0 被改变的有效位置, $v0=循环次数
0x00400e6c <+176>: nop
0x00400e70 <+180>: sll v0,v0,0x2//v0=循环次数*4
0x00400e74 <+184>: addiu v1,s8,24//v1=2147479848
0x00400e78 <+188>: addu v0,v1,v0
0x00400e7c <+192>: lw v0,4(v0)
0x00400e80 <+196>: nop
0x00400e84 <+200>: beq a0,v0,0x400e98 <phase_2+220>
0x00400e88 <+204>: nop
0x00400e8c <+208>: jal 0x4021f0 <explode_bomb>
0x00400e90 <+212>: nop
0x00400e94 <+216>: lw gp,16(s8)
0x00400e98 <+220>: lw v0,24(s8)
0x00400e9c <+224>: nop
0x00400ea0 <+228>: addiu v0,v0,1
0x00400ea4 <+232>: sw v0,24(s8)
0x00400ea8 <+236>: lw v0,24(s8)
0x00400eac <+240>: nop
0x00400eb0 <+244>: slti v0,v0,6
0x00400eb4 <+248>: bnez v0,0x400e20 <phase_2+100>
0x00400eb8 <+252>: nop
0x00400ebc <+256>: move sp,s8
0x00400ec0 <+260>: lw ra,60(sp)
0x00400ec4 <+264>: lw s8,56(sp)
0x00400ec8 <+268>: addiu sp,sp,64

```

```
0x00400ecc <+272>: jr ra
0x00400ed0 <+276>: nop
End of assembler dump.
```

其中对于 `0x00400e18 <+92>: b 0x400ea8 <phase_2+236>`：猜也能猜到跳转到 `0x400ea8 <phase_2+236>`，查了一下官方文档

B offset is the assembly idiom used to denote an unconditional branch. The actual instruction is interpreted by the hardware as BEQ r0, r0, offset.

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself), in the branch delay slot, to form a PC-relative effective target address.

发现确实是这样

然后我想像 vs 那样单步查看变量的值，于是查询了一下单步执行的命令 `ni` 例子：

```
(gdb) b* 0x00400dd4
Breakpoint 1 at 0x400dd4
(gdb) continue
Continuing.
warning: Could not load shared library symbols for 2 libraries, e.g. /lib/libc.so.6.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?

Breakpoint 1, 0x00400dd4 in phase_2 ()
(gdb) ni
0x00400dd8 in phase_2 ()
(gdb) ni
0x00400ddc in phase_2 ()
(gdb) ni
0x00400de0 in phase_2 ()
(gdb) █
```

CSDN @Capsfly

在第一个值得怀疑的地方设置一个断点

```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
(gdb) disas phase_2
Dump of assembler code for function phase_2:
0x00400dbc <+0>:      addiu    sp,sp,-64
0x00400dc0 <+4>:      sw        ra,60(sp)
0x00400dc4 <+8>:      sw        s8,56(sp)
0x00400dc8 <+12>:     move     s8,sp
0x00400dcc <+16>:     lui       gp,0x42
0x00400dd0 <+20>:     addiu    gp,gp,-20080
0x00400dd4 <+24>:     sw        gp,16(sp)
0x00400dd8 <+28>:     sw        a0,64(s8)
0x00400ddc <+32>:     addiu    v0,s8,28
0x00400de0 <+36>:     lw        a0,64(s8)
0x00400de4 <+40>:     move     a1,v0
0x00400de8 <+44>:     jal       0x401ba8 <read_six_numbers>
0x00400dec <+48>:     nop
0x00400df0 <+52>:     lw        gp,16(s8)
0x00400df4 <+56>:     lw        v1,28(s8)
0x00400df8 <+60>:     li        v0,1
=> 0x00400dfc <+64>:     beq       v1,v0,0x400e10 <phase_2+84>
0x00400e00 <+68>:     nop
0x00400e04 <+72>:     jal       0x4021f0 <expcode_bomb>
0x00400e08 <+76>:     nop
0x00400e0c <+80>:     lw        gp,16(s8)
0x00400e10 <+84>:     li        v0,1
0x00400e14 <+88>:     sw        v0,24(s8)
0x00400e18 <+92>:     b         0x400ea8 <phase_2+236>
0x00400e1c <+96>:     nop
0x00400e20 <+100>:    lw        v0,24(s8)
```

p \$v0 发现是 1，当然从上一行的指令中我们也能看到。所以第一个数字一定是 1 然后一步一步使用 ni 指令一步一步走，如下图

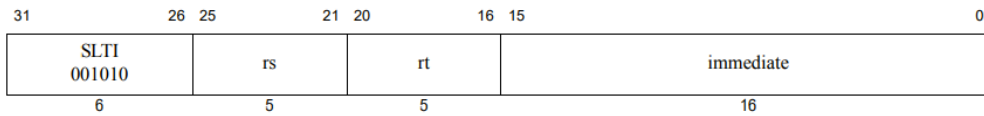
```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
0x00400ea4 <+232>: sw      v0,24(s8)
0x00400ea8 <+236>: lw      v0,24(s8)
0x00400eac <+240>: nop
0x00400eb0 <+244>: slti    v0,v0,6
0x00400eb4 <+248>: bnez    v0,0x400e20 <phase_2+100>
0x00400eb8 <+252>: nop
0x00400ebc <+256>: move    sp,s8
0x00400ec0 <+260>: lw      ra,60(sp)
0x00400ec4 <+264>: lw      s8,56(sp)
0x00400ec8 <+268>: addiu   sp,sp,64
0x00400ecc <+272>: jr      ra
0x00400ed0 <+276>: nop
End of assembler dump.
(gdb) p $v0
$1 = 1
(gdb) p $v1
$2 = 1
(gdb) ni
0x00400e10 in phase_2 ()
(gdb) ni
0x00400e14 in phase_2 ()
(gdb) ni
0x00400e18 in phase_2 ()
(gdb) ni
0x00400ea8 in phase_2 ()
(gdb) bt
#0  0x00400ea8 in phase_2 ()
#1  0x00400c10 in main ()
warning: GDB can't find the start of the function at 0x7f63f03b.
(gdb) ni
0x00400eac in phase_2 ()
(gdb) ni
0x00400eb0 in phase_2 ()
(gdb) █
```

CSDN @Capsfly

然后碰到了 `0x00400eb0 <+244>: slti v0,v0,6` 查了一下 这条指令的含义是 `if $v0<6,then $v0=1,else $v0=0`

## SLTI

## Set on Less Than Immediate



**Format:** SLTI *rt*, *rs*, *immediate*

MIPS32

**Purpose:** Set on Less Than Immediate

To record the result of a less-than comparison with a constant.

**Description:**  $GPR[rt] \leftarrow (GPR[rs] < \text{sign\_extend}(\text{immediate}))$

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

```
if GPR[rs] < sign_extend(immediate) then
    GPR[rt] ← 0GPRLEN-1 || 1
else
    GPR[rt] ← 0GPRLEN
endif
```

**Exceptions:**

None

CSDN @Capsfly

5. 继续就这么执行，直到跳转到了 `0x00400e20 <+100>: lw v0,24(s8)` 我们之前手动分析的时候，这个地方保存的值就是 `1`，如果不放心的话可以打印看一看 `(gdb) x/1ub $s8+24` 结果

```
(gdb) p $v0
$6 = 1
(gdb) p $s8
$7 = 2147479824
(gdb) p $s8+16
$8 = 2147479840
(gdb) p s8
No symbol table is loaded. Use the "file" command.
(gdb) x/1ub $s8+16
0x7ffff120: 144
(gdb) x/1ub $s8+24
0x7ffff128: 1
(gdb)
```

CSDN @Capsfly

继续执行，发现程序总是在调用

```
34 0x00400e3c <+128>: li v1,12
35 0x00400e40 <+132>: lw v0,24(s8)//循环次数
36 0x00400e44 <+136>: nop
37 0x00400e48 <+140>: subu v0,v1,v0// $v0=12-循环次数
38 0x00400e4c <+144>: lw v1,-32660(gp)// $v1=$gp-32660
39 0x00400e50 <+148>: sll v0,v0,0x2// $v0=(12-循环次数)*4
40 0x00400e54 <+152>: addu v0,v1,v0// $v0=$gp-32660+(12-循环次数)*4
41 0x00400e58 <+156>: lw v0,0(v0)
```

CSDN @Capsfly

`$gp-32660` 相当于基址

打印一下发现

```

1 0x413284 <ID_num+32>: 6
1 (gdb) x/1un 4272768
0x413280 <ID_num+28>: Undefined output format "n".
(gdb) x/1ub 4272768
0x413280 <ID_num+28>: 9
(gdb) x/1ub 4272764
0x41327c <ID_num+24>: 9
(gdb) x/1ub 4272776
0x413288 <ID_num+36>: 0
(gdb) x/1ub 4272780
0x41328c <ID_num+40>: 0
(gdb) x/1ub 4272784
0x413290 <ID_num+44>: 7
(gdb)

```

CSDN @Capsfly

这正好是我输入的 ID-number

就 这么一步一步拆，最后发现结果是

170000，当然这组数据可能跟我取的值有关系，我输入的 ID-number=996007，

### 3. 第三关

Dump of assembler code for function phase\_3:

```

0x00400ed4 <+0>: addiu sp,sp,-56
0x00400ed8 <+4>: sw ra,52(sp)
0x00400edc <+8>: sw s8,48(sp)
0x00400ee0 <+12>: move s8,sp
0x00400ee4 <+16>: lui gp,0x42
0x00400ee8 <+20>: addiu gp,gp,-20080
0x00400eec <+24>: sw gp,24(sp)
0x00400ef0 <+28>: sw a0,56(s8)
0x00400ef4 <+32>: lw a0,56(s8)
0x00400ef8 <+36>: lui v0,0x40
0x00400efc <+40>: addiu a1,v0,10112
0x00400f00 <+44>: addiu v1,s8,44
0x00400f04 <+48>: addiu v0,s8,40
0x00400f08 <+52>: addiu a2,s8,36
0x00400f0c <+56>: sw a2,16(sp)
0x00400f10 <+60>: move a2,v1
0x00400f14 <+64>: move a3,v0
0x00400f18 <+68>: lw v0,-32636(gp)
0x00400f1c <+72>: nop
0x00400f20 <+76>: move t9,v0
0x00400f24 <+80>: jalr t9
0x00400f28 <+84>: nop
0x00400f2c <+88>: lw gp,24(s8)
0x00400f30 <+92>: slti v0,v0,3
0x00400f34 <+96>: beqz v0,0x400f48 <phase_3+116>
0x00400f38 <+100>: nop
0x00400f3c <+104>: jal 0x4021f0 <explode_bomb>
0x00400f40 <+108>: nop
0x00400f44 <+112>: lw gp,24(s8)

```



```

0x00400f48 <+116>: lw v0,44(s8)
0x00400f4c <+120>: nop
0x00400f50 <+124>: sltiu v1,v0,8
0x00400f54 <+128>: beqz v1,0x401190 <phase_3+700>
0x00400f58 <+132>: nop
0x00400f5c <+136>: sll v1,v0,0x2
0x00400f60 <+140>: lui v0,0x40
0x00400f64 <+144>: addiu v0,v0,10124
0x00400f68 <+148>: addu v0,v1,v0
0x00400f6c <+152>: lw v0,0(v0)
0x00400f70 <+156>: nop
0x00400f74 <+160>: jr v0
0x00400f78 <+164>: nop
0x00400f7c <+168>: li v0,113
0x00400f80 <+172>: sb v0,32(s8)
0x00400f84 <+176>: lw v0,-32660(gp)
0x00400f88 <+180>: nop
0x00400f8c <+184>: lw v1,44(v0)
0x00400f90 <+188>: lw v0,36(s8)
0x00400f94 <+192>: nop
0x00400f98 <+196>: mult v1,v0
0x00400f9c <+200>: mflo v1
0x00400fa0 <+204>: li v0,777
0x00400fa4 <+208>: beq v1,v0,0x4011ac <phase_3+728>
0x00400fa8 <+212>: nop
0x00400fac <+216>: jal 0x4021f0 <explode_bomb>
0x00400fb0 <+220>: nop
0x00400fb4 <+224>: lw gp,24(s8)
0x00400fb8 <+228>: b 0x4011f8 <phase_3+804>
0x00400fbc <+232>: nop
0x00400fc0 <+236>: li v0,98
0x00400fc4 <+240>: sb v0,32(s8)
0x00400fc8 <+244>: lw v0,-32660(gp)
0x00400fcc <+248>: nop
0x00400fd0 <+252>: lw v1,44(v0)
0x00400fd4 <+256>: lw v0,36(s8)
0x00400fd8 <+260>: nop
0x00400fdc <+264>: mult v1,v0
0x00400fe0 <+268>: mflo v1
0x00400fe4 <+272>: li v0,214
0x00400fe8 <+276>: beq v1,v0,0x4011b8 <phase_3+740>
0x00400fec <+280>: nop
0x00400ff0 <+284>: jal 0x4021f0 <explode_bomb>
0x00400ff4 <+288>: nop
0x00400ff8 <+292>: lw gp,24(s8)
0x00400ffc <+296>: b 0x4011f8 <phase_3+804>
0x00401000 <+300>: nop
0x00401004 <+304>: li v0,98
0x00401008 <+308>: sb v0,32(s8)
0x0040100c <+312>: lw v0,-32660(gp)
0x00401010 <+316>: nop
0x00401014 <+320>: lw v1,44(v0)
0x00401018 <+324>: lw v0,36(s8)
0x0040101c <+328>: nop

```



```
0x00401020 <+332>: mult v1,v0
0x00401024 <+336>: mflo v1
0x00401028 <+340>: li v0,755
0x0040102c <+344>: beq v1,v0,0x4011c4 <phase_3+752>
0x00401030 <+348>: nop
0x00401034 <+352>: jal 0x4021f0 <explode_bomb>
0x00401038 <+356>: nop
0x0040103c <+360>: lw gp,24(s8)
0x00401040 <+364>: b 0x4011f8 <phase_3+804>
0x00401044 <+368>: nop
0x00401048 <+372>: li v0,107
0x0040104c <+376>: sb v0,32(s8)
0x00401050 <+380>: lw v0,-32660(gp)
0x00401054 <+384>: nop
0x00401058 <+388>: lw v1,44(v0)
0x0040105c <+392>: lw v0,36(s8)
0x00401060 <+396>: nop
0x00401064 <+400>: mult v1,v0
0x00401068 <+404>: mflo v0
0x0040106c <+408>: beqz v0,0x4011d0 <phase_3+764>
0x00401070 <+412>: nop
0x00401074 <+416>: jal 0x4021f0 <explode_bomb>
0x00401078 <+420>: nop
0x0040107c <+424>: lw gp,24(s8)
0x00401080 <+428>: b 0x4011f8 <phase_3+804>
0x00401084 <+432>: nop
0x00401088 <+436>: li v0,111
0x0040108c <+440>: sb v0,32(s8)
0x00401090 <+444>: lw v0,-32660(gp)
0x00401094 <+448>: nop
0x00401098 <+452>: lw v1,44(v0)
0x0040109c <+456>: lw v0,36(s8)
0x004010a0 <+460>: nop
0x004010a4 <+464>: mult v1,v0
0x004010a8 <+468>: mflo v1
0x004010ac <+472>: li v0,228
0x004010b0 <+476>: beq v1,v0,0x4011dc <phase_3+776>
0x004010b4 <+480>: nop
0x004010b8 <+484>: jal 0x4021f0 <explode_bomb>
0x004010bc <+488>: nop
0x004010c0 <+492>: lw gp,24(s8)
0x004010c4 <+496>: b 0x4011f8 <phase_3+804>
0x004010c8 <+500>: nop
0x004010cc <+504>: li v0,116
0x004010d0 <+508>: sb v0,32(s8)
0x004010d4 <+512>: lw v0,-32660(gp)
0x004010d8 <+516>: nop
0x004010dc <+520>: lw v1,44(v0)
0x004010e0 <+524>: lw v0,36(s8)
0x004010e4 <+528>: nop
0x004010e8 <+532>: mult v1,v0
0x004010ec <+536>: mflo v1
0x004010f0 <+540>: li v0,513
0x004010f4 <+544>: beq v1,v0,0x4011e8 <phase_3+788>
```

```
0x004010f8 <+548>: nop
0x004010fc <+552>: jal 0x4021f0 <explode_bomb>
0x00401100 <+556>: nop
0x00401104 <+560>: lw gp,24(s8)
0x00401108 <+564>: b 0x4011f8 <phase_3+804>
0x0040110c <+568>: nop
0x00401110 <+572>: li v0,118
0x00401114 <+576>: sb v0,32(s8)
0x00401118 <+580>: lw v0,-32660(gp)
0x0040111c <+584>: nop
0x00401120 <+588>: lw v1,44(v0)
0x00401124 <+592>: lw v0,36(s8)
0x00401128 <+596>: nop
0x0040112c <+600>: mult v1,v0
0x00401130 <+604>: mflo v1
0x00401134 <+608>: li v0,780
0x00401138 <+612>: beq v1,v0,0x40114c <phase_3+632>
0x0040113c <+616>: nop
0x00401140 <+620>: jal 0x4021f0 <explode_bomb>
0x00401144 <+624>: nop
0x00401148 <+628>: lw gp,24(s8)
0x0040114c <+632>: li v0,98
0x00401150 <+636>: sb v0,32(s8)
0x00401154 <+640>: lw v0,-32660(gp)
0x00401158 <+644>: nop
0x0040115c <+648>: lw v1,44(v0)
0x00401160 <+652>: lw v0,36(s8)
0x00401164 <+656>: nop
0x00401168 <+660>: mult v1,v0
0x0040116c <+664>: mflo v1
0x00401170 <+668>: li v0,824
0x00401174 <+672>: beq v1,v0,0x4011f4 <phase_3+800>
0x00401178 <+676>: nop
0x0040117c <+680>: jal 0x4021f0 <explode_bomb>
0x00401180 <+684>: nop
0x00401184 <+688>: lw gp,24(s8)
0x00401188 <+692>: b 0x4011f8 <phase_3+804>
0x0040118c <+696>: nop
0x00401190 <+700>: li v0,120
0x00401194 <+704>: sb v0,32(s8)
0x00401198 <+708>: jal 0x4021f0 <explode_bomb>
0x0040119c <+712>: nop
0x004011a0 <+716>: lw gp,24(s8)
0x004011a4 <+720>: b 0x4011f8 <phase_3+804>
0x004011a8 <+724>: nop
0x004011ac <+728>: nop
0x004011b0 <+732>: b 0x4011f8 <phase_3+804>
0x004011b4 <+736>: nop
0x004011b8 <+740>: nop
0x004011bc <+744>: b 0x4011f8 <phase_3+804>
0x004011c0 <+748>: nop
0x004011c4 <+752>: nop
0x004011c8 <+756>: b 0x4011f8 <phase_3+804>
0x004011cc <+760>: nop
```



的 debug 发现是输出格式的错误，应该是 `x/1uw %gp-32636`，这样就是 2137465136 了，确认过眼神（下次一定好好看输出格式）

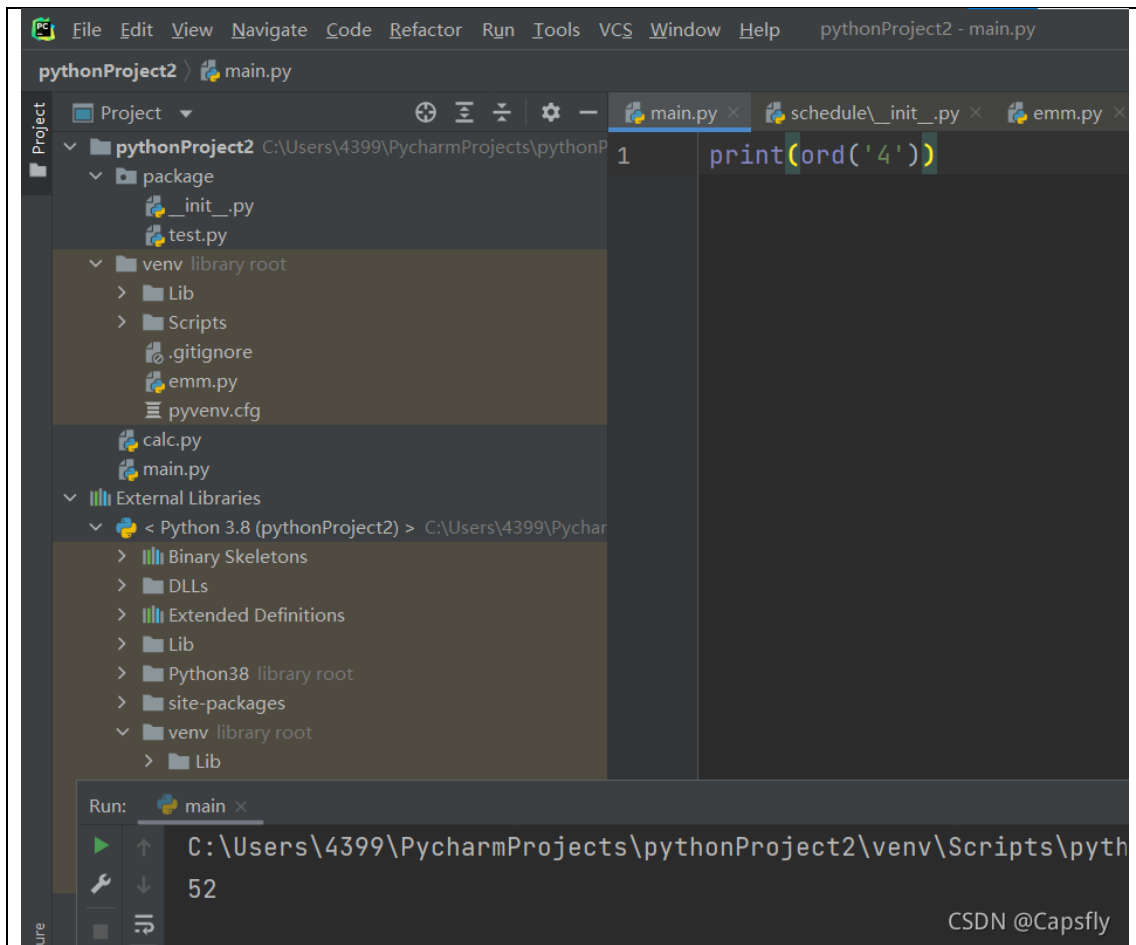
```
(gdb) p $v0
$2 = 2137465136
(gdb) p $gp-32636
$3 = 4272660
(gdb) x/1ub $gp-32636
0x413214: 48
(gdb) x/1uw $gp-32636
0x413214: 2137465136
```

Ubuntu 64 位- VMware Workstation 16 Player (仅用于非商业用途)

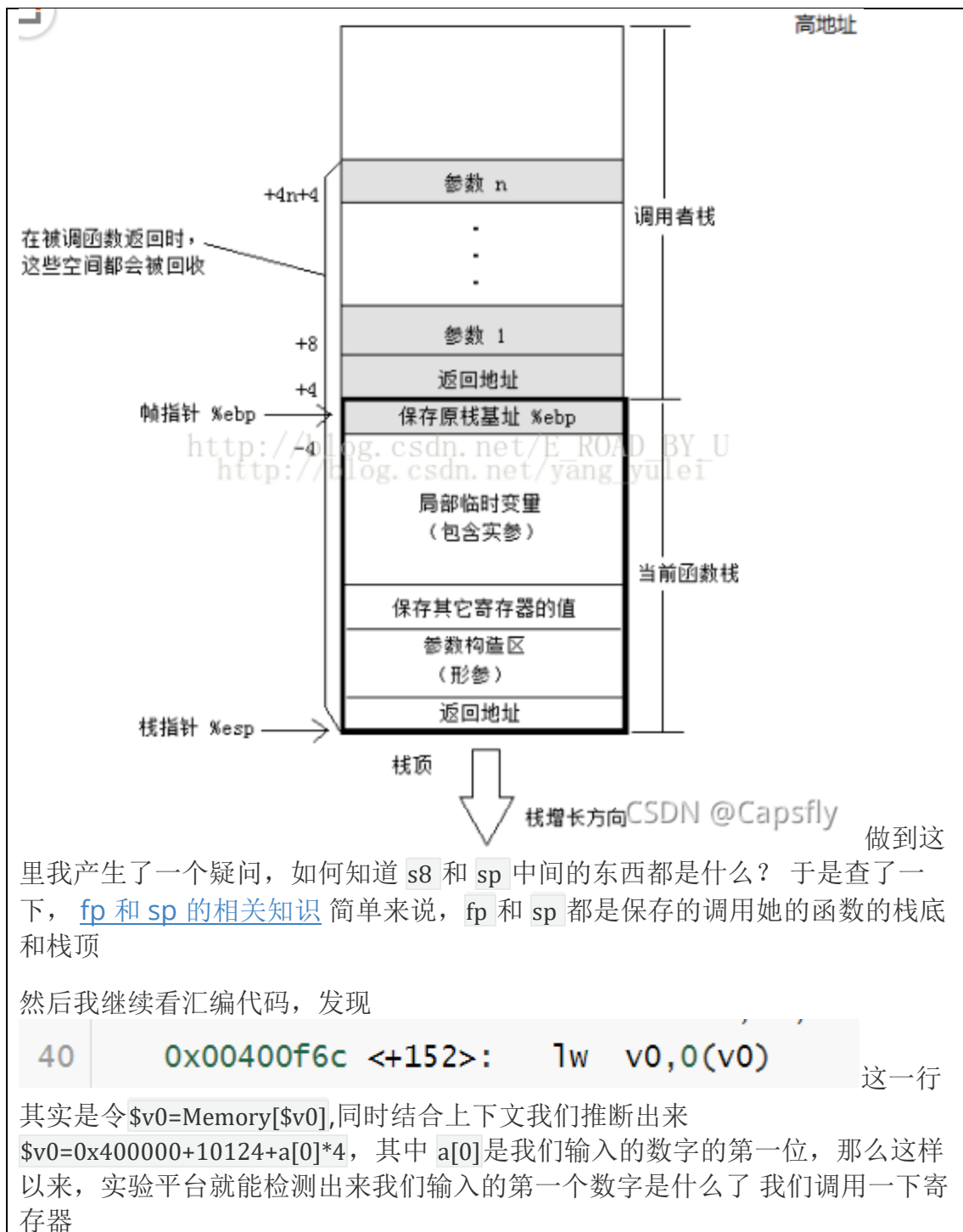
Player(P) | [Icons] | [Progress Bar]

```
Activities | Terminal | Nov 18 04:50
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
(gdb) p $v0
$5 = 4198336
(gdb) i r
zero at v0 v1 a0 a1
R0 00000000 00000001 00400fc0 00000004 7a617400 00000000
t0 t1 t2 t3 t4 t5
R8 00000000 19999999 7f774378 7f774c78 00000005 ffffffff
s0 s1 s2 s3 s4 s5
R16 00000000 00000000 00000000 00000000 00000000 00000000
t8 t9 k0 k1 gp sp
R24 00000001 7f6a779c 00000000 00000000 0041b190 7ffff118
sr lo hi bad cause pc
24000010 00000000 00000000 00000000 00000000 00400f70
fsr fir
00000000 00739300
capsfly@ubuntu:~/Desktop/mipsel-gdb/bin (gdb) b* 0x00400fd8
Breakpoint 4 at 0x400fd8
(gdb) continue
Please input: Continuing.
996007
Welcome to Breakpoint 4, 0x00400fd8 in phase_3 ()
which to b' (gdb) x/3uw $s8+36
Let's begin! 0x7ffff13c: 0 52 1
Phase 1 de: (gdb)
1 7 0 0 0 0
That's number 2. Keep going!
```

我输入的是 1 4 6，但是因为它是按照字符读取的，所以我们输出它的 4 的 asc



可以发现，是一样的，同时我们观察 `stack` 中保存变量的顺序，正好从低到高



```

(gdb) p $v0
$9 = 4204432
(gdb) bt
#0  0x00400f6c in phase_3 ()
#1  0x00400c5c in main ()
warning: GDB can't find the start of the function at 0x7f63e03b.
(gdb) ni
0x00400f70 in phase_3 ()
(gdb) p $v0
$10 = 4198336
(gdb) i r

      zero      at      v0      v1      a0      a1      a2      a3
R0     00000000 00000001 00400fc0 00000004 43d1e700 00000000 0000000a 7fffebe9
      t0       t1       t2       t3       t4       t5       t6       t7
R8     00000000 19999999 7f774378 7f774c78 00000005 ffffffff 7f627750 000007fc
      s0       s1       s2       s3       s4       s5       s6       s7
R16    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
      t8       t9       k0       k1       gp       sp       s8       ra
R24    00000001 7f6a779c 00000000 00000000 0041b190 7ffff118 7ffff118 00400f2c
      sr       lo       hi       bad      cause   pc
      24000010 00000000 00000000 00000000 00000000 00400f70
      fsr      fir
      00000000 00739300
(gdb)

```

CSDN @Capsfly

发现\$*v0* 保存的是一个地址

```
61      0x00400fc0 <+236>:  li  v0,98
```

那么问题来

了, 如果我们输入的第一位数字不是 1 而是别的数字会怎么办?

```

1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int i=0;i<10;i++)
7      {
8          cout<<(0x400000)+10124+i*4<<endl;
9      }
10     return 0;
11 }

```

```

运行: untitled102 x
C:\Users\4399\untitled102\cmake-build-debug\untitled102.exe
4204428
4204432
4204436
4204440
4204444
4204448
4204452
4204456
4204460

```

CSDN @Capsfly

打印输出了 10 个值, 结果发现正符合预期, 最后 2 个乱码是因为 8 9 是 UI 直接爆炸, 根本不会执行到这里

```

00000000 00739300
(gdb) x/10uw 4204428
0x40278c:  4198268 4198336 4198404 4198472
0x40279c:  4198536 4198604 4198672 4198732
0x4027ac:  25637  1851877735
(gdb)

```

## 我们检测一下， 在线进制转换

支持在2~36进制之间进行任意转换，支持浮点型

☐ 2进制 ☐ 4进制 ☐ 8进制 ☒ 10进制 ☐ 16进制 ☐ 32进制 10进制 ▼

转换数字 4198732

☐ 2进制 ☐ 4进制 ☐ 8进制 ☐ 10进制 ☒ 16进制 ☐ 32进制 16进制 ▼

转换结果 40114c

CSDN @Capsfly

160 0x0040114c <+632>: li v0,98

发现正好有（当然也必须有），同时看上下文，这个地址后面也只有一个 `li` 指令，符合我们的预期

```
46 0x00400f84 <+176>: lw v0,-32660(gp)
47 0x00400f88 <+180>: nop
48 0x00400f8c <+184>: lw v1,44(v0)
```

看到这几行指令，我认为那里保存的是我输入的数据，输出一看，确实是 `ID-num`，同时温馨提示，尽量不要选择特别奇怪的数字，比如连着 2 个 0，你很有可能理解为那个是未初始化的内存单元的值

```
(gdb) X3uw +$v0+44
Undefined command: "X3uw". Try "help".
(gdb) x/3uw $v0+44
0x413290 <ID_num+44>: 7      661939532      1700929651
(gdb) x/1uw $v0+48
0x413294 <input_strings>: 661939532
(gdb) x/1uw $v0+40
0x41328c <ID_num+40>: 0
(gdb) x/6uw $v0+24
0x41327c <ID_num+24>: 9      9      6      0
0x41328c <ID_num+40>: 0      7
(gdb)
```

CSDN @Capsfly

在分析的过程中，观察这段代码，最终发现这段其实就做了两件事情 1.将 `v0` 寄存器的值放到了 `Memory[$s8+32]`

`ID-num` 的最后一位和输入的最后一个数字相乘，最终和 `$v0` 比较是否相等，若相等，则跳转，否则爆炸。因为我输入的学号的最后一位是 7，所以需要找到一个 7 的倍数的 `$v0`，最终发现是第一个函数段，所以我们输入的第一个数字应该是 0



```

44 0x00400f7c <+168>: li v0,113
45 0x00400f80 <+172>: sb v0,32(s8)
46 0x00400f84 <+176>: lw v0,-32660(gp)
47 0x00400f88 <+180>: nop
48 0x00400f8c <+184>: lw v1,44(v0)
49 0x00400f90 <+188>: lw v0,36(s8)
50 0x00400f94 <+192>: nop
51 0x00400f98 <+196>: mult v1,v0
52 0x00400f9c <+200>: mflo v1
53 0x00400fa0 <+204>: li v0,777
54 0x00400fa4 <+208>: beq v1,v0,0x4011ac <phase_3+728>
55 0x00400fa8 <+212>: nop
56 0x00400fac <+216>: jal 0x4021f0 <explode_bomb>

```

CSDN @Capsfly

最终我们成功跳转到了

```

202 0x004011f8 <+804>: lb v0,40(s8)
203 0x004011fc <+808>: lb v1,32(s8)
204 0x00401200 <+812>: nop
205 0x00401204 <+816>: beq v1,v0,0x401218 <phase_3+836>
206 0x00401208 <+820>: nop
207 0x0040120c <+824>: jal 0x4021f0 <explode_bomb>
208 0x00401210 <+828>: nop
209 0x00401214 <+832>: lw gp,24(s8)
210 0x00401218 <+836>: move sp,s8
211 0x0040121c <+840>: lw ra,52(sp)
212 0x00401220 <+844>: lw s8,48(sp)
213 0x00401224 <+848>: addiu sp,sp,56
214 0x00401228 <+852>: jr ra
215 0x0040122c <+856>: nop

```

CSDN @Capsfly

从上文的分析中，我们知道 `Memory[$s8+40]` 保存的是输入的第 2 个参数，而 `Memory[$s8+32]` 保存的是 113

```

44 0x00400f7c <+168>: li v0,113
45 0x00400f80 <+172>: sb v0,32(s8)

```

经过分析得出，这个 113 其实表示的是一个字符，所以我们第 2 个输入 `q` 即可

```

capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
capsfly@ubuntu:~/Desktop/mipsel-gdb/bin$ qemu-mipsel -g 2333 /home/capsfly/Desktop/mipsel-gdb/bin/bomb
Please input your ID_number:
9966007
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Let's begin now!
Phase 1 defused. How about the next one?
1 7 0 0 0 0
That's number 2. Keep going!
0 q 113
Halfway there!

```

CSDN @Capsfly

## 4. 第四关

碰到的难点如下

```
46      0x0040136c <+176>:    jal 0x401230 <func4>
```

func4 做了什么？我们看一下 func4 的代码

Dump of assembler code for function func4:

```
0x00401230 <+0>: addiu sp,sp,-40
0x00401234 <+4>: sw ra,36(sp)
0x00401238 <+8>: sw s8,32(sp)
0x0040123c <+12>: sw s0,28(sp)
0x00401240 <+16>: move s8,sp
0x00401244 <+20>: sw a0,40(s8)
0x00401248 <+24>: lw v0,40(s8)
0x0040124c <+28>: nop
0x00401250 <+32>: slti v0,v0,2
0x00401254 <+36>: bnez v0,0x40129c <func4+108>
0x00401258 <+40>: nop
0x0040125c <+44>: lw v0,40(s8)
0x00401260 <+48>: nop
0x00401264 <+52>: addiu v0,v0,-1
0x00401268 <+56>: move a0,v0
0x0040126c <+60>: jal 0x401230 <func4>
0x00401270 <+64>: nop
0x00401274 <+68>: move s0,v0
0x00401278 <+72>: lw v0,40(s8)
0x0040127c <+76>: nop
0x00401280 <+80>: addiu v0,v0,-2
0x00401284 <+84>: move a0,v0
0x00401288 <+88>: jal 0x401230 <func4>
0x0040128c <+92>: nop
0x00401290 <+96>: addu v0,s0,v0
0x00401294 <+100>: b 0x4012a0 <func4+112>
0x00401298 <+104>: nop
0x0040129c <+108>: li v0,1
0x004012a0 <+112>: move sp,s8
0x004012a4 <+116>: lw ra,36(sp)
0x004012a8 <+120>: lw s8,32(sp)
0x004012ac <+124>: lw s0,28(sp)
0x004012b0 <+128>: addiu sp,sp,40
0x004012b4 <+132>: jr ra
0x004012b8 <+136>: nop
```

End of assembler dump.

7. 我们先看这么几个关键点

```
11      0x00401254 <+36>:    bnez    v0,0x40129c <func4+108> // $v0<2 就跳转到 <func4+108>
```

```

29 0x0040129c <+108>: li    v0,1
30 0x004012a0 <+112>: move   sp,s8
31 0x004012a4 <+116>: lw     ra,36(sp)
32 0x004012a8 <+120>: lw     s8,32(sp)
33 0x004012ac <+124>: lw     s0,28(sp)
34 0x004012b0 <+128>: addiu  sp,sp,40
35 0x004012b4 <+132>: jr     ra
36 0x004012b8 <+136>: nop
12 0x00401258 <+40>: nop
13 0x0040125c <+44>: lw     v0,40(s8)
14 0x00401260 <+48>: nop
15 0x00401264 <+52>: addiu  v0,v0,-1
16 0x00401268 <+56>: move   a0,v0
17 0x0040126c <+60>: jal    0x401230 <func4>

```

CSDN @Capsf

看这一段汇编代码，我们可以很明显的看出，这一段调用了 `func4($v0-1)` 再看

```

18 0x00401270 <+64>: nop
19 0x00401274 <+68>: move   s0,v0
20 0x00401278 <+72>: lw     v0,40(s8)
21 0x0040127c <+76>: nop
22 0x00401280 <+80>: addiu  v0,v0,-2
23 0x00401284 <+84>: move   a0,v0
24 0x00401288 <+88>: jal    0x401230 <func4>

```

它调用了 `func($v0-2)` 这是什么？这不是斐波那契数列吗？！ 再看这一段

```

26 0x00401290 <+96>: addu    v0,s0,v0

```

果然 所以接下来的问题就变成了，斐波那契的第几项是 8 我们打表看一下

运行: untitled102 x		
C:\Users\4399\untitled102		
2	2	
3	3	
4	5	
5	8	
6	13	
7	21	
8	34	
9	55	
10	89	

CSDN @Capsfly

所以答案很明

显，是 5

```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
capsfly@ubuntu:~/Desktop/mipsel-gdb/bin$ qemu-mipsel -g 2333 /home/capsfly/Desktop/mipsel-gdb/bin/bomb
Please input your ID_number:
996007
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Let's begin now!
Phase 1 defused. How about the next one?
1 7 0 0 0 0
That's number 2. Keep going!
0 q 111
Halfway there!
5
So you got that one. Try this one.
```

CSDN @Capsfly

## 5. 第五关

```
0x004013e8 <+0>: addiu sp,sp,-72
0x004013ec <+4>: sw ra,68(sp)
0x004013f0 <+8>: sw s8,64(sp)
0x004013f4 <+12>: move s8,sp
0x004013f8 <+16>: sw a0,72(s8)
0x004013fc <+20>: lw a0,72(s8)
0x00401400 <+24>: jal 0x401c78 <string_length>
0x00401404 <+28>: nop
0x00401408 <+32>: move v1,v0
0x0040140c <+36>: li v0,6
0x00401410 <+40>: beq v1,v0,0x401420 <phase_5+56>
0x00401414 <+44>: nop
0x00401418 <+48>: jal 0x4021f0 <explode_bomb>
0x0040141c <+52>: nop
0x00401420 <+56>: sw zero,24(s8)
0x00401424 <+60>: b 0x4014a8 <phase_5+192>
0x00401428 <+64>: nop
0x0040142c <+68>: lw v0,24(s8)
0x00401430 <+72>: lw v1,24(s8)
0x00401434 <+76>: lw a0,72(s8)
0x00401438 <+80>: nop
0x0040143c <+84>: addu v1,a0,v1
0x00401440 <+88>: lb v1,0(v1)
0x00401444 <+92>: nop
0x00401448 <+96>: andi v1,v1,0xff
0x0040144c <+100>: andi v1,v1,0xf
0x00401450 <+104>: sll v0,v0,0x2
0x00401454 <+108>: addiu a0,s8,24
0x00401458 <+112>: addu v0,a0,v0
0x0040145c <+116>: sw v1,12(v0)
0x00401460 <+120>: lw a0,24(s8)
0x00401464 <+124>: lw v0,24(s8)
0x00401468 <+128>: nop
```

```

0x0040146c <+132>: sll v0,v0,0x2
0x00401470 <+136>: addiu v1,s8,24
0x00401474 <+140>: addu v0,v1,v0
0x00401478 <+144>: lw v1,12(v0)
0x0040147c <+148>: lui v0,0x41
0x00401480 <+152>: addiu v0,v0,12524
0x00401484 <+156>: addu v0,v1,v0
0x00401488 <+160>: lb v1,0(v0)
0x0040148c <+164>: addiu v0,s8,24
0x00401490 <+168>: addu v0,v0,a0
0x00401494 <+172>: sb v1,4(v0)
0x00401498 <+176>: lw v0,24(s8)
0x0040149c <+180>: nop
0x004014a0 <+184>: addiu v0,v0,1
0x004014a4 <+188>: sw v0,24(s8)
0x004014a8 <+192>: lw v0,24(s8)
0x004014ac <+196>: nop
0x004014b0 <+200>: slti v0,v0,6
0x004014b4 <+204>: bnez v0,0x40142c <phase_5+68>
0x004014b8 <+208>: nop
0x004014bc <+212>: sb zero,34(s8)
0x004014c0 <+216>: addiu v0,s8,28
0x004014c4 <+220>: move a0,v0
0x004014c8 <+224>: lui v0,0x40
0x004014cc <+228>: addiu a1,v0,10160
0x004014d0 <+232>: jal 0x401cf8 <strings_not_equal>
0x004014d4 <+236>: nop
0x004014d8 <+240>: beqz v0,0x4014e8 <phase_5+256>
0x004014dc <+244>: nop
0x004014e0 <+248>: jal 0x4021f0 <explode_bomb>
0x004014e4 <+252>: nop
0x004014e8 <+256>: move sp,s8
0x004014ec <+260>: lw ra,68(sp)
0x004014f0 <+264>: lw s8,64(sp)
0x004014f4 <+268>: addiu sp,sp,72
0x004014f8 <+272>: jr ra
0x004014fc <+276>: nop

```

8. 首先我们要看一下 `string_length` 的代码，我们大体猜测，这个函数应该是计算我们输入的字符串的长度，不过还需要看看代码验证一下

```

0x00401c78 <+0>: addiu sp,sp,-24
0x00401c7c <+4>: sw s8,20(sp)
0x00401c80 <+8>: move s8,sp
0x00401c84 <+12>: sw a0,24(s8)
0x00401c88 <+16>: lw v0,24(s8)
0x00401c8c <+20>: nop
0x00401c90 <+24>: sw v0,12(s8)
0x00401c94 <+28>: sw zero,8(s8)
0x00401c98 <+32>: b 0x401cb0 <string_length+56>
0x00401c9c <+36>: nop

```

```

0x00401ca0 <+40>: lw v0,8(s8)
0x00401ca4 <+44>: nop
0x00401ca8 <+48>: addiu v0,v0,1
0x00401cac <+52>: sw v0,8(s8)
0x00401cb0 <+56>: lw v0,12(s8)
0x00401cb4 <+60>: nop
0x00401cb8 <+64>: lb v0,0(v0)
0x00401cbc <+68>: nop
0x00401cc0 <+72>: sltu v0,zero,v0
0x00401cc4 <+76>: andi v0,v0,0xff
0x00401cc8 <+80>: lw v1,12(s8)
0x00401ccc <+84>: nop
0x00401cd0 <+88>: addiu v1,v1,1
0x00401cd4 <+92>: sw v1,12(s8)
0x00401cd8 <+96>: bnez v0,0x401ca0 <string_length+40>
0x00401cdc <+100>: nop
0x00401ce0 <+104>: lw v0,8(s8)
0x00401ce4 <+108>: move sp,s8
0x00401ce8 <+112>: lw s8,20(sp)
0x00401cec <+116>: addiu sp,sp,24
0x00401cf0 <+120>: jr ra
0x00401cf4 <+124>: nop

```

End of assembler dump.

19      0x00401cc0 <+72>:      sltu      v0,zero,v0

做实验的时候，遇到了这行代码，不太理解，翻了一下指令集手册，发现这样

**SLTU** Set on Less Than Unsigned

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000			rs		rt		rd		0 00000		SLTU 101011
6			5		5		5		5		6

**Format:** SLTU rd, rs, rt MIPS32

**Purpose:** Set on Less Than Unsigned  
To record the result of an unsigned less-than comparison.

**Description:**  $GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$   
Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).  
The arithmetic comparison does not cause an Integer Overflow exception.

CSDN @Capsfly

本来以为，这是个寄存器值和立即数的比较，但是看了一下，发现是两个寄存器，我就不太理解，后来恍然大悟，这个 **zero** 有没有可能是 **0** 号寄存器？！于是输出了一下，果然是

```

End of assembler dump.
(gdb) i r
      zero      at      v0      v1      a0      a1      a2      a3
R0    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
      t0      t1      t2      t3      t4      t5      t6      t7
R8    00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
      s0      s1      s2      s3      s4      s5      s6      s7
R16   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
      t8      t9      k0      k1      gp      sp      s8      ra
R24   00000000 00000000 00000000 00000000 00000000 7ffff250 00000000 00000000
      sr      lo      hi      bad      cause      pc
      24000010 00000000 00000000 00000000 00000000 7f7c9cd0
      fsr      fir
      00000000 00739300
(gdb)

```

当然，这个\$zero 保存的一直都是 0 继续分析 string\_length 的代码

```

17 0x00401cb8 <+64>: 1b v0,0(v0)//可能存放的是我们输入的字符

```

因为 1b 中 0 的参数一定是一个地址，所以我们往前找，

```

15 0x00401cb0 <+56>: 1w v0,12(s8)//可能存放了一个地址

```

我们推测 \$s8+12 的值也一定是一个地址 继续往下看

```

19 0x00401cc0 <+72>: sltu v0,zero,v0//if $v0>0 then $v0=1 else 0

```

9. 这行代码其实是最令我无法理解的，我最开始以为，难道它的循环次数是从负数开始？其次，那他怎么判断循环终结？（当然，最后我都搞懂了这些东西）不过分析还是要一步一步来

```

21 0x00401cc8 <+80>: 1w v1,12(s8)//可能保存的是读取到了哪里，因为每一个字符都是1byte
22 0x00401ccc <+84>: nop
23 0x00401cd0 <+88>: addiu v1,v1,1
24 0x00401cd4 <+92>: sw v1,12(s8)//保存循环次数?

```

我们推测，\$v1 才是保存的循环次数，因为他有很明显的 +1 并且保存到了 \$s8+12,那么原来的 \$v0 就肯定不是了,并且我们结合上下文也能发现,\$v0 中的值并没有保存，所以我们推测，v0 中的保存的值有别的含义 我们想，如果它读取结束了，那么它一定会退出，也就是

```

25 0x00401cd8 <+96>: bnez v0,0x401ca0 <string_length+40>

```

那么我们往前看 什么时候 \$v0 会等于 0 呢？

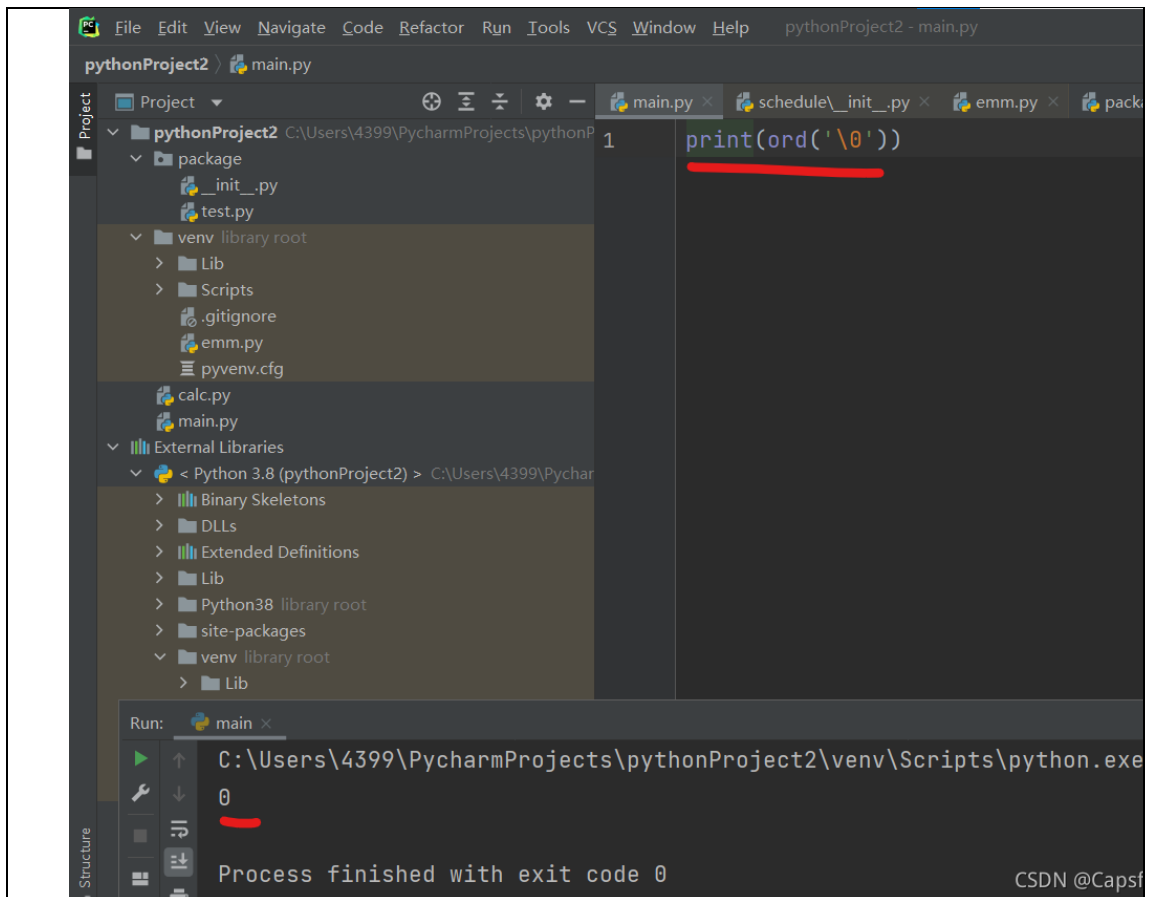
```

19 0x00401cc0 <+72>: sltu v0,zero,v0//if $v0>0 then $v0=1 else 0

```

我们突然想起，上一个实验的 113 其实是 q 的 asc 码 那么这个 0 是什么？这个不是 '\0' 的 asc 码吗？！ 我们输出一下看看





10. 所以这样，这个函数就完全搞清除了 对于第 5 关，我的输入是 `half`

`So you got that one. Try this one.`  
`half`

```

1 0x004013e8 <+0>: addiu sp,sp,-72
2 0x004013ec <+4>: sw ra,68(sp)
3 0x004013f0 <+8>: sw s8,64(sp)
4 0x004013f4 <+12>: move s8,sp
5 0x004013f8 <+16>: sw a0,72(s8)
6 0x004013fc <+20>: lw a0,72(s8)

```

对于这几行代码，我的猜测就是 `$a0` 中保存的地址存放着输入的字符串，输出

```

(gdb) p $a0
$1 = 4273108
(gdb) x/10uw $a0
0x4133d4 <input_strings+320>: 1718378856 0 0 0
0x4133e4 <input_strings+336>: 0 0 0 0
0x4133f4 <input_strings+352>: 0 0
(gdb) x/16c $a0
0x4133d4 <input_strings+320>: 104 'h' 97 'a' 108 'l' 102 'f' 0 '\000'
0 '\000' 0 '\000' 0 '\000'
0x4133dc <input_strings+328>: 0 '\000' 0 '\000' 0 '\000'
0 '\000' 0 '\000' 0 '\000'
(gdb)

```



确实是

11. 在做实验的时候，也遇到这样的错误

```
7      0x00401400 <+24>:      jal 0x401c78 <string_length>
```

我想进入 `string_length` 函数单步调试，但是使用 `ni` 直接跳过，使用 `step` 直接爆炸，后来发现，应该使用 `si` 命令（或者 `stepi`）同时，对于一个函数，如果被调用了很多次，我们不应该直接在那个位置上设置断点，在它前面的某个位置设置一个断点，然后 `si` 即可 否则一直 `c` 会很麻烦 同时

```
11     0x00401ca0 <+40>:      lw  v0,8(s8) //从<+96>跳转过来的
12     0x00401ca4 <+44>:      nop
13     0x00401ca8 <+48>:      addiu v0,v0,1
14     0x00401cac <+52>:      sw  v0,8(s8)
15     0x00401cb0 <+56>:      lw  v0,12(s8) // $v0保存我们输入的字符串在内存中的地址
16     0x00401cb4 <+60>:      nop
17     0x00401cb8 <+64>:      lb  v0,0(v0) //存放的是我们读取到的字符
18     0x00401cbc <+68>:      nop
19     0x00401cc0 <+72>:      sltu v0,zero,v0 //判断是否读取到了'\0'，如果读取到了，直接退出
20     0x00401cc4 <+76>:      andi v0,v0,0xff //其实没什么用，这一行
21     0x00401cc8 <+80>:      lw  v1,12(s8) //加载地址
22     0x00401ccc <+84>:      nop
23     0x00401cd0 <+88>:      addiu v1,v1,1 //没有读取到'\0'，地址的位置+1
24     0x00401cd4 <+92>:      sw  v1,12(s8) //保存读取到了哪里
25     0x00401cd8 <+96>:      bnez v0,0x401ca0 <string_length+40>
```

CSDN @Capsf

我最开始读这段代码的时候没有搞懂，为什么 `v0` 中存放的是保存的循环次数，你刚开始就+1 然后保存，不怕读取到了 `'\0'` 吗？后来仔细阅读发现，这个+1 加的其实是上一次读取到的，如果上一次读取到了 `'\0'` 那么它根本不会跳转到这里！ 它就直接退出了！ 所以它保存的一定是循环次数！ 也就是，字符串的长度！

然后继续往下看

```
15     0x00401420 <+56>:      sw  zero,24(s8)
```

不难猜出来，`Memory[$s8+24]` 保存的是 `for` 循环的执行次数

```
22     0x0040143c <+84>:      addu  v1,a0,v1
23     0x00401440 <+88>:      lb  v1,0(v1)
```

从这两行代码中，我们可以才出来，`a0` 保存的是一个内存地址 那么我们输出一下

```
(gdb) x/6c $a0
0x4133d4 <input_strings+320>:  115 's' 99 'c' 121 'y' 116 't' 99 'c' 108 'l'
(gdb)
```

是我们输入的字符串 所以以下这两行的代码的作用就明晰了

```
22     0x0040143c <+84>:      addu  v1,a0,v1/
23     0x00401440 <+88>:      lb  v1,0(v1) //取出 22
```

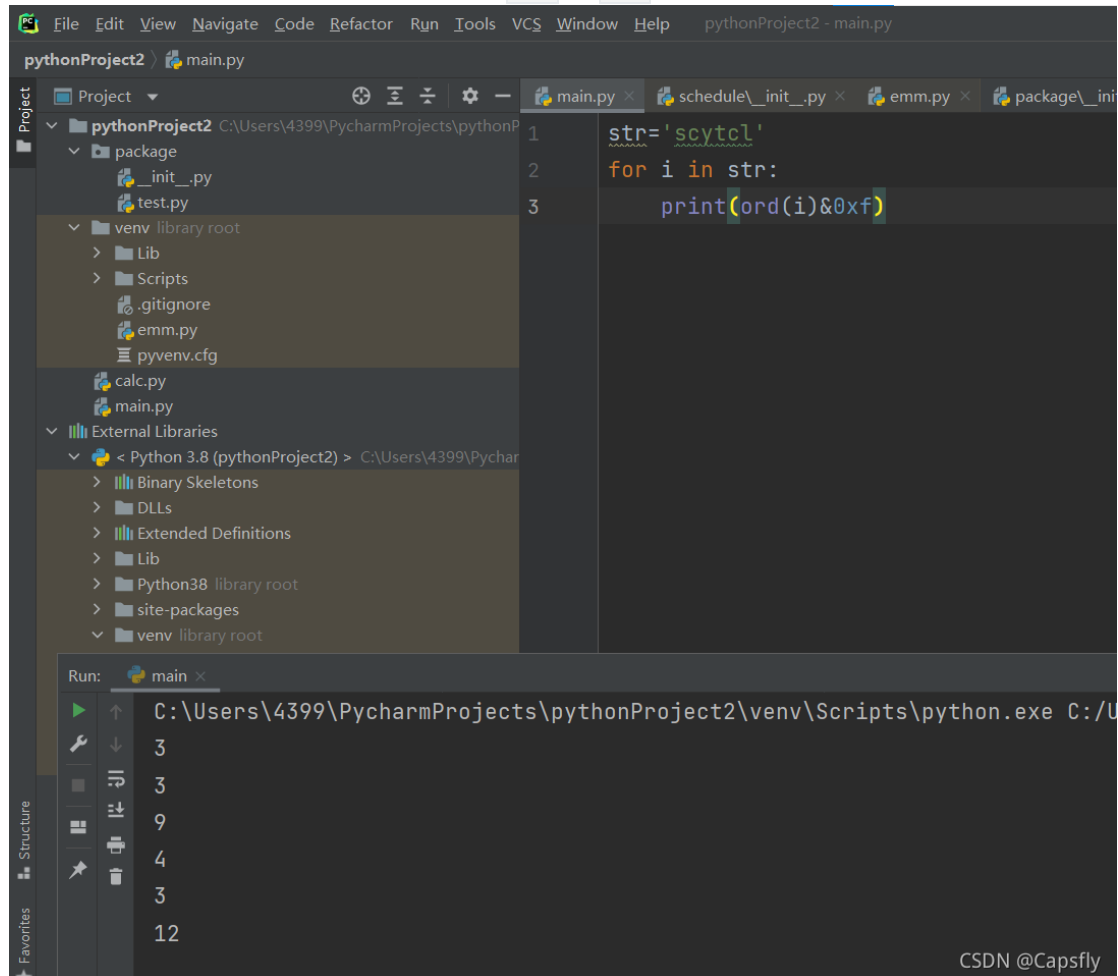
行是用来保存当前读取到的位置 `v1` 用来保存读取出来的字符

所以

```
0x00401448 <+96>: andi v1,v1,0xff
```

0x0040144c <+100>: andi v1,v1,0xf//字符的 asc 和 0xf 相与

那我们输出一下我们输入的字符的 asc 和 0xf 相与之后的结果



The screenshot shows the PyCharm IDE interface. The main editor displays a Python script in `main.py` with the following code:

```
1 str='scytcl'
2 for i in str:
3     print(ord(i)&0xf)
```

The left sidebar shows the project structure for `pythonProject2`, including `package`, `test.py`, and `venv` directory. The bottom panel shows the output of the script, displaying the results of the bitwise AND operation for each character in the string 'scytcl':

```
Run: main x
C:\Users\4399\PycharmProjects\pythonProject2\venv\Scripts\python.exe C:/U
3
3
9
4
3
12
```

The output shows the decimal values of the characters 's', 'c', 'y', 't', 'c', 'l' after being ANDed with 0xf, which are 3, 3, 9, 4, 3, and 12 respectively.

那么我们在虚拟机上看一看，第一位字符处理之后的结果是不是 3

```
(gdb) p $v1
$2 = 3
(gdb)
```

果然 并且我们知道

```
27 0x00401450 <+104>: sll v0,v0,0x2//v0=4*循环次数
28 0x00401454 <+108>: addiu a0,s8,24
29 0x00401458 <+112>: addu v0,a0,v0
30 0x0040145c <+116>: sw v1,12(v0)//Memory[$s8+36+4*循环次数]=当前读取到的字符asc和0xf
相与
```

```
(gdb) i r
zero at v0
a3
R0 00000000 00000001 00000000
133db
```

那么我们来调用一下看看

```
0x00401464 in phase_5 ()
(gdb) x/1uw $s8+36
0x7ffff12c: 3
```

对于 37 0x00401478 <+144>: lw v1,12(v0)

`$v1=Memory[$s8+36+4*循环次数]`,读取到的字符 `asc` 和 `0xf` 相与的结果 查看寄存

器

```
i r
zero at v0 v1
00000000 00000001 00410000 00000003
```

果然

```
38 0x0040147c <+148>: lui v0,0x41
39 0x00401480 <+152>: addiu v0,v0,12524
40 0x00401484 <+156>: addu v0,v1,v0//对于
```

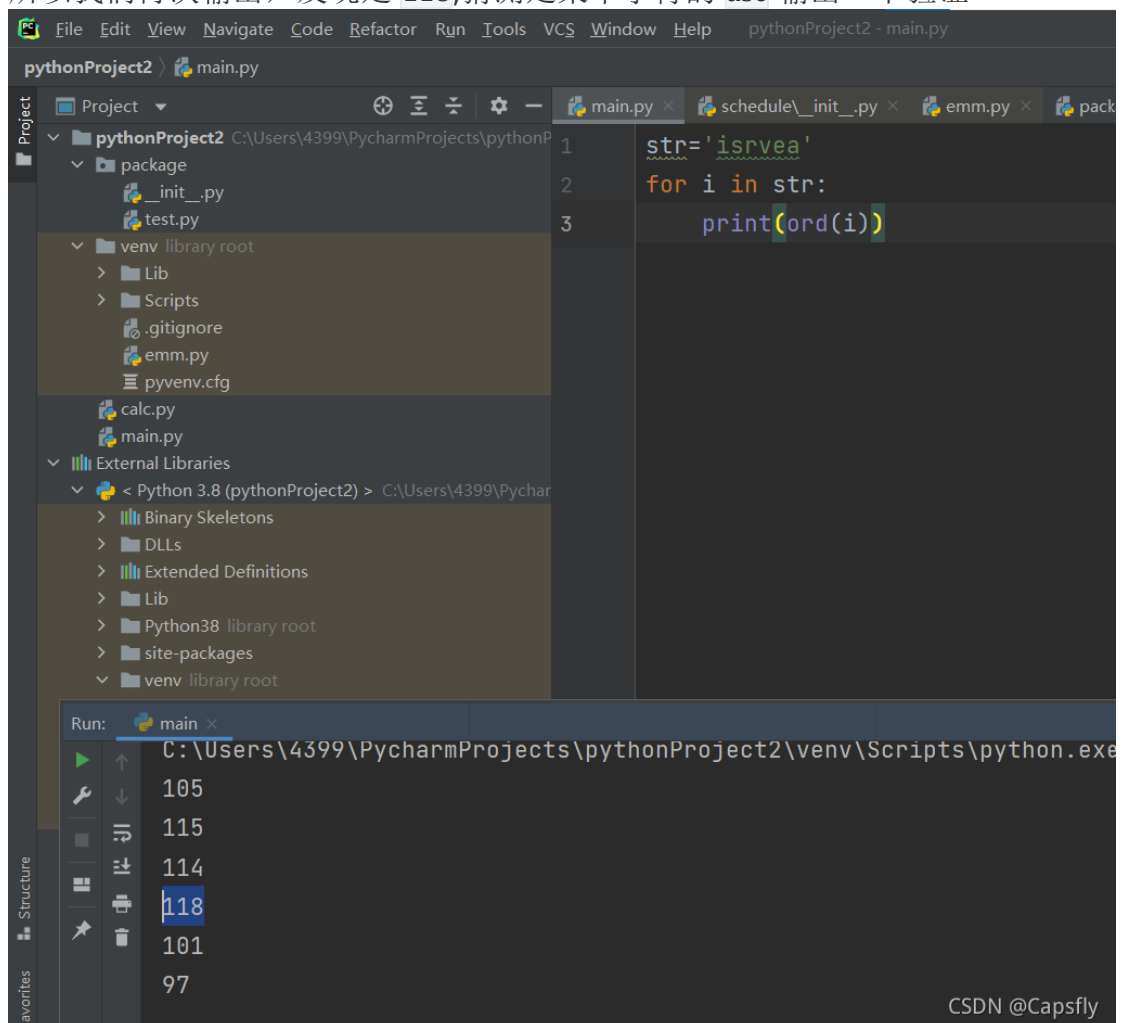
9. 这一段,基本上看到就能猜到,这肯定是一个地址了,那么我们输出一下,最开始我的猜测是,这个可能是我们输入的信息之类的,比如学号或者你输入的字符串 but....

```
(gdb) x/6c 4272364
0x4130ec <array.3607>: 105 'i' 115 's' 114 'r' 118 'v' 101 'e' 97 'a'
```

又因为

```
41 0x00401488 <+160>: lb v1,0(v0)//一个固定的地址+读取到的字符asc和0xf相与的结果
```

所以我们再次输出,发现是 118,猜测是某个字符的 `asc` 输出一下验证



的确!也正好是我们推测的结果!对于

```
0x00401494 <+172>: sb v1,4(v0)
```

我们知道，把\$**v1** 储存到了循环次数在内存中的地址+循环次数

12.再检验一次 设置断点在 `0x0040144c <+100>: andi v1,v1,0xf`//字符的 asc 和 0xf 相与 这里，输出一下 register

```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
capsfly@ubuntu: ~/Desktop/mip... x capsfly@ubuntu: ~/Desktop/mip... x
gdb) ni
x00401450 in phase_5 ()
gdb) i r
      zero      at      v0      v1      a0      a1      a2
a3
R0      00000000 00000001 00000001 00000003 004133d4 00000100 00000006
33db
      t0      t1      t2      t3      t4      t5      t6
t7
R8      0041459e 0a0a0a0a 00000001 20746168 2e656e6f 72542020 68742079
7369
      s0      s1      s2      s3      s4      s5      s6
s7
R16     00000000 00000000 00000000 00000000 00000000 00000000 00000000
0000
      t8      t9      k0      k1      gp      sp      s8
ra
R24     00000002 7f64e7d0 00000000 00000000 0041b190 7fffffff 7fffffff
CSDN @Capsfly
```

12. 我们之前用 `pycharm` 输出了 `scytcl` 的 asc 如下 发现正确 事实上，形如这种的指令（虽然对解题无帮助，但是个人觉得还是很有必要理解的）`0x0040145c <+116>: sw v1,12(v0)` 这种，修改了系统内存的东西（类似于 `c++` 中修改类内私有变量的成员的值，曾经在这个点上 `WA` 了无数次。。。。）对于 `0x0040145c <+116>` 的指令，验证第 1 次循环如下（循环从 0 计数）

```
(gdb) x/1uw $s8+36+4
0x7ffff130:      3
(gdb)
```

对于 `0x00401478 <+144>: lw v1,12(v0)`//`$v1=Memory[$s8+36+4*循环次数]` 指令，

```
(gdb) x/1uw $s8+36+4
0x7ffff130:      3
(gdb)
```

验证如下 对于 `0x00401488 <+160>: lb v1,0(v0)`//一个固定的地址+读取到的字符 asc 和 0xf 相与的结果 验证如下

```
breakpoint 4, 0x
(gdb) p $v1
$6 = 118
(gdb) █
```

对于 0x00401494 <+172>: sb v1,4(v0)//把\$v1 储存到了循环次数在内存中的地址+循环次数+4 验证如下

```
x/1uw $s8+24+1+4
ff125:      83902582
x/1ub $s8+24+1+4
ff125:      118
```

哦对了，验证了时候发生了一点小的事故，其实从我的输入中也能看出来错在了哪里，输出格式不正确(这个也是我数学建模的队友 lxr 巨巨跟我讨论了一节课的问题)，观察一下汇编的输入

```
0x00401494 <+172>:  sb  v1,4(v0)// 这个是 store byte 不是 store word!!!!
```

再然后就碰到了这么一个函数

```
59  0x004014d0 <+232>:  jal 0x401cf8 <strings_not_equal>
```

Dump of assembler code for function strings\_not\_equal:

```
0x00401cf8 <+0>:      addiu    sp,sp,-48
0x00401cfc <+4>:      sw        ra,44(sp)
0x00401d00 <+8>:      sw        s8,40(sp)
0x00401d04 <+12>:     move     s8,sp
0x00401d08 <+16>:     sw        a0,48(s8)
0x00401d0c <+20>:     sw        a1,52(s8)
0x00401d10 <+24>:     lw        a0,48(s8)
0x00401d14 <+28>:     jal        0x401c78 <string_length>
0x00401d18 <+32>:     nop
0x00401d1c <+36>:     sw        v0,36(s8)
0x00401d20 <+40>:     lw        a0,52(s8)
0x00401d24 <+44>:     jal        0x401c78 <string_length>
0x00401d28 <+48>:     nop
0x00401d2c <+52>:     sw        v0,32(s8)
0x00401d30 <+56>:     lw        v0,48(s8)
0x00401d34 <+60>:     nop
0x00401d38 <+64>:     sw        v0,28(s8)
0x00401d3c <+68>:     lw        v0,52(s8)
0x00401d40 <+72>:     nop
0x00401d44 <+76>:     sw        v0,24(s8)
0x00401d48 <+80>:     lw        v1,36(s8)
0x00401d4c <+84>:     lw        v0,32(s8)
0x00401d50 <+88>:     nop
0x00401d54 <+92>:     beq      v1,v0,0x401dc4 <strings_not_equal+204>
0x00401d58 <+96>:     nop
0x00401d5c <+100>:    li        v0,1
0x00401d60 <+104>:    b        0x401de4 <strings_not_equal+236>
0x00401d64 <+108>:    nop
0x00401d68 <+112>:    lw        v0,28(s8)
0x00401d6c <+116>:    nop
0x00401d70 <+120>:    lb        v1,0(v0)
0x00401d74 <+124>:    lw        v0,24(s8)
```

0x00401d78 <+128>:	nop	
0x00401d7c <+132>:	lb	v0,0(v0)
0x00401d80 <+136>:	nop	
0x00401d84 <+140>:	xor	v0,v1,v0
0x00401d88 <+144>:	sltu	v0,zero,v0
0x00401d8c <+148>:	andi	v0,v0,0xff
0x00401d90 <+152>:	lw	v1,28(s8)
0x00401d94 <+156>:	nop	
0x00401d98 <+160>:	addiu	v1,v1,1
0x00401d9c <+164>:	sw	v1,28(s8)
0x00401da0 <+168>:	lw	v1,24(s8)
0x00401da4 <+172>:	nop	
0x00401da8 <+176>:	addiu	v1,v1,1
0x00401dac <+180>:	sw	v1,24(s8)
0x00401db0 <+184>:	beqz	v0,0x401dc8 <strings_not_equal+208>
0x00401db4 <+188>:	nop	
0x00401db8 <+192>:	li	v0,1
0x00401dbc <+196>:	b	0x401de4 <strings_not_equal+236>
0x00401dc0 <+200>:	nop	
0x00401dc4 <+204>:	nop	
0x00401dc8 <+208>:	lw	v0,28(s8)
0x00401dcc <+212>:	nop	
0x00401dd0 <+216>:	lb	v0,0(v0)
0x00401dd4 <+220>:	nop	
0x00401dd8 <+224>:	bnez	v0,0x401d68 <strings_not_equal+112>
0x00401ddc <+228>:	nop	
0x00401de0 <+232>:	move	v0,zero
0x00401de4 <+236>:	move	sp,s8
0x00401de8 <+240>:	lw	ra,44(sp)
0x00401dec <+244>:	lw	s8,40(sp)
0x00401df0 <+248>:	addiu	sp,sp,48
0x00401df4 <+252>:	jr	ra
0x00401df8 <+256>:	nop	
End of assembler dump.		

13. 我们看这一行		
0x00401d5c <+100>:	li	v0,1
0x00401d60 <+104>:	b	0x401de4 <strings_not_equal+236>

我们结合函数调用的上下文可以得出，如果这一行被执行了，那么一定会爆炸 如果很幸运，我们没有爆炸 那么我们继续往下看		
54	0x00401dc8 <+208>:	lw v0,28(s8) // ?
55	0x00401dcc <+212>:	nop
56	0x00401dd0 <+216>:	lb v0,0(v0) // 猜测这里存在经过偏移处理后的结果
毫无疑问，从内存中读取出来了一组数据，我们非常有理由怀疑 读取出来的是我们在 phase_5 中经过处理后的数据或者是本来我们输入的数据		
我们往前找一下，因为 s8 很少改变 所以我们在 phase_5 中找一找 我们发现		
44	0x00401494 <+172>:	sb v1,4(v0) // 把 \$v1 储存到了循环次数在内存中的地址+循环次数+4，储存的是修改之后的结果，需要注意

所以更坚定了我们认为这里读取到的是经过处理后的数据

我们再往下看

```
0x00401d84 <+140>: xor    v0,v1,v0//按位或
0x00401db0 <+184>: beqz   v0,0x401dc8 <strings_not_equal+208>
```

这两行其实是这个函数的精髓所在（虽然其实也没有多难）如果 `$v0 xor $v1==0`，那么 `$v0==$v1`

然后

```
38 0x00401d88 <+144>: sltu    v0,zero,v0//if $v0==0 $v0=0 else 1
```

最后

```
48 0x00401db0 <+184>: beqz   v0,0x401dc8 <strings_not_equal+208>//
```

判断是否为 0，如果是，则跳转，否则 `$v0=1`，爆炸 所以到这里，我们猜测，是要进行处理后的字符串和给定的字符串进行比较，如果处理后的字符串和给定的字符串相同，那么不爆炸，否则爆炸

那么接下来的问题就变成了

1. 它给定的字符串是什么？
2. 找到那个转换函数，使得经过映射后的字符串等于给定的字符串

好，知道这些后，我们先来看一下上文中提到的，那个位置存储的是不是我们经过处理后的字符串 我们重新开

```
Breakpoint 1, 0x004014bc in phase_5 ()
(gdb) x/6ub $s8+28
0x7ffff124: 118 118 98 101 118 117
(gdb)
```

果然

我们再从

```
0x00401d74 <+124>: lw     v0,24(s8)//推测是个地址
0x00401d78 <+128>: nop
0x00401d7c <+132>: lb     v0,0(v0)
0x00401d80 <+136>: nop
0x00401d84 <+140>: xor    v0,v1,v0//按位或
```

写到这里我突然产生了疑问，如果按位或的话，那岂不是最小值取决于给定的字符串？于是百度了一下 `xor` 发现是异或

但是我们的思路方向还是正确的，必须相等，否则爆炸

这里，查看一下给定的字符串

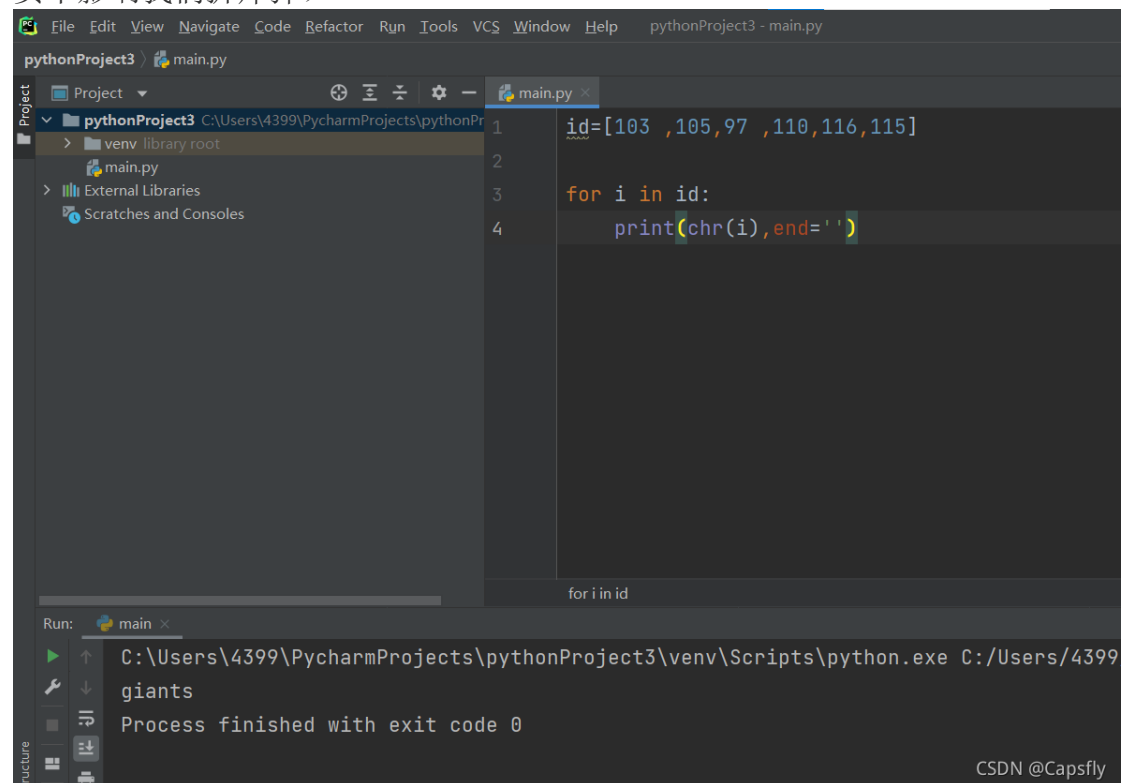


```

0x7ffff0f0:    176    39    64    0    36    241
(gdb) bt
#0  0x00401d88 in strings_not_equal ()
#1  0x004014d8 in phase_5 ()
#2  0x00400cf4 in main ()
warning: GDB can't find the start of the function at 0x7f63e03b.
(gdb) x/6uw $s8+24
0x7ffff0f0:    4204464 2147479844    6    6
0x7ffff100:    2147479816    4199640
(gdb) x/1uw $s8+24
0x7ffff0f0:    4204464
(gdb) x/6uw 4204464
0x4027b0:    1851877735    29556    561475415    1970231584
0x4027c0:    543520295    1969644900
(gdb) x/6ub 4204464
0x4027b0:    103    105    97    110    116
(gdb)

```

当然，我相信你从我的繁杂的输入中也能看到我错在了哪里 如果没有，请仔细阅读前文 输出这个字符串 可以看出 字符串为 `giants`（当然，如果只知道 `asc` 其实不影响我们拆炸弹）



The screenshot shows the PyCharm IDE interface. The main editor displays a Python script named `main.py` with the following code:

```

1 id=[103 ,105,97 ,110,116,115]
2
3 for i in id:
4     print(chr(i),end=' ')

```

The Run window at the bottom shows the execution output:

```

Run: main x
C:\Users\4399\PycharmProjects\pythonProject3\venv\Scripts\python.exe C:/Users/4399
giants
Process finished with exit code 0

```

The output clearly shows the string `giants` printed on the console.

那么接下来我们就需要找到那个函数

经过分析，函数主要就是这一段



```

30 0x0040145c <+116>: sw v1,12(v0)//Memory[$s8+36+4*循环次数]=当前读取到的字符asc和0xf相与，修改了内存
31 0x00401460 <+120>: lw a0,24(s8)//循环次数
32 0x00401464 <+124>: lw v0,24(s8)//循环次数
33 0x00401468 <+128>: nop
34 0x0040146c <+132>: sll v0,v0,0x2//$v0=4*循环次数
35 0x00401470 <+136>: addiu v1,s8,24//v1保存的是循环次数在内存中的地址
36 0x00401474 <+140>: addu v0,v1,v0//v0保存的是循环次数在内存中的地址+4*循环次数
37 0x00401478 <+144>: lw v1,12(v0)//$v1=Memory[$s8+36+4*循环次数]，读取到的字符asc和0xf相与的结果
38 0x0040147c <+148>: lui v0,0x41
39 0x00401480 <+152>: addiu v0,v0,12524//0x410000+12524
40 0x00401484 <+156>: addu v0,v1,v0//0x410000+12524+偏移（读取到的字符asc和0xf相与的结果）
41 0x00401488 <+160>: lb v1,0(v0)//一个固定的地址+读取到的字符asc和0xf相与的结果
42 0x0040148c <+164>: addiu v0,s8,24//循环次数的地址
43 0x00401490 <+168>: addu v0,v0,a0//$v0=循环次数在内存中的地址+循环次数
44 0x00401494 <+172>: sb v1,4(v0)//把$v1储存到了循环次数在内存中的地址+循环次数+4，储存的是修改之后的结果，需要注
意 CSDN @Capsfly

```

注释写的比较详细 分析可得，我们只需要找到是 giants 在内存中的地址即可

好的，问题来了 我们需要查看多少内存空间？ ans:16 因为是输入的字符和 0xf 相与，所以结果一定是 0-e，共 16 个

```

(gdb) x/16ub
0x4027b6:  0      0      87     111     119     33     32     89
0x4027be:  111     117     39     118     101     32     100    101
(gdb)
0x7ffff0f0:  176     39     64      0     36     241
(gdb) bt
#0  0x00401d88 in strings_not_equal ()
#1  0x004014d8 in phase_5 ()
#2  0x00400cf4 in main ()
warning: GDB can't find the start of the function at 0x7f63e03b.
(gdb) x/6uw $s8+24
0x7ffff0f0:  4204464 2147479844      6      6
0x7ffff100:  2147479816 4199640
(gdb) x/1uw $s8+24
0x7ffff0f0:  4204464
(gdb) x/6uw 4204464
0x4027b0:  1851877735 29556 561475415 1970231584
0x4027c0:  543520295 1969644900
(gdb) x/6ub 4204464
0x4027b0:  103     105     97     110     116 CSDN @Capsfly
(gdb)

```

所以我们接下来只需要寻找 giants 在内存中的地址即可

```

26 0x0040144c <+100>: andi v1,v1,0xf//字符的asc和0xf相与
27 0x00401450 <+104>: sll v0,v0,0x2//$v0=4*循环次数
28 0x00401454 <+108>: addiu a0,s8,24
29 0x00401458 <+112>: addu v0,a0,v0
30 0x0040145c <+116>: sw v1,12(v0)//Memory[$s8+36+4*循环次数]=当前读取到的字符asc和0xf相与，修改了内存
31 0x00401460 <+120>: lw a0,24(s8)//循环次数
32 0x00401464 <+124>: lw v0,24(s8)//循环次数
33 0x00401468 <+128>: nop
34 0x0040146c <+132>: sll v0,v0,0x2//$v0=4*循环次数
35 0x00401470 <+136>: addiu v1,s8,24//v1保存的是循环次数在内存中的地址
36 0x00401474 <+140>: addu v0,v1,v0//v0保存的是循环次数在内存中的地址+4*循环次数
37 0x00401478 <+144>: lw v1,12(v0)//$v1=Memory[$s8+36+4*循环次数]，读取到的字符asc和0xf相与的结果
38 0x0040147c <+148>: lui v0,0x41
39 0x00401480 <+152>: addiu v0,v0,12524//0x410000+12524
40 0x00401484 <+156>: addu v0,v1,v0//0x410000+12524+偏移（读取到的字符asc和0xf相与的结果）
41 0x00401488 <+160>: lb v1,0(v0)//一个固定的地址+读取到的字符asc和0xf相与的结果
42 0x0040148c <+164>: addiu v0,s8,24//循环次数的地址
43 0x00401490 <+168>: addu v0,v0,a0//$v0=循环次数在内存中的地址+循环次数
44 0x00401494 <+172>: sb v1,4(v0)//把$v1储存到了循环次数在内存中的地址+循环次数+4，储存的是修改之后的结果，需要注
意 CSDN @Capsfly

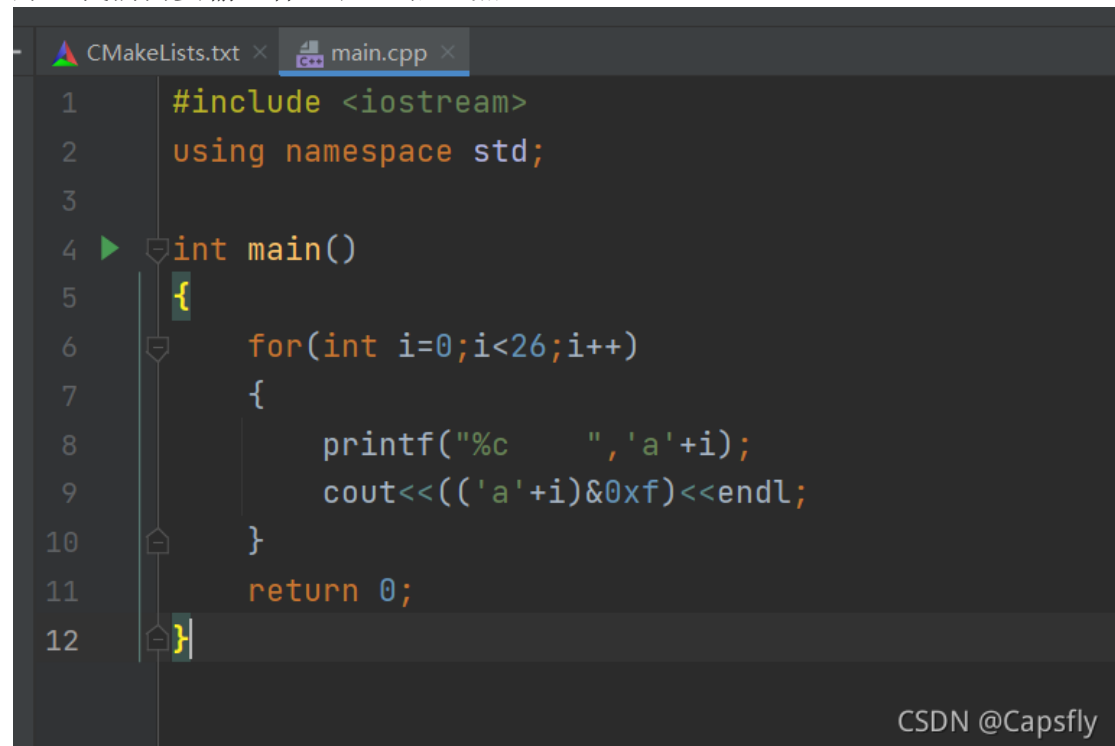
```

我们从这几段代码中可以发现，我们先从一个偏移的位置找到了函数映射后的结果，然后将这个映射后的结果保存到了一个位置，也正是 `strings_not_equal` 调用出来的位置

所以这样问题就化简成了，在那段地址中，寻找 `giants`

```
(gdb) x/16ub 0x410000+12524
0x4130ec <array.3607>: 105      115      114      118      101      97       119      104
0x4130f4 <array.3607+8>: 111      98       112      110      117      116      102      103
(gdb)
```

可以发现，这段内存中的值包含 `giants` 最终查找发现，偏移量是 `15 0 5 12 14 1` 那么我们需要输入什么呢？很显然



```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int i=0;i<26;i++)
7      {
8          printf("%c    ", 'a'+i);
9          cout<<(('a'+i)&0xf)<<endl;
10     }
11     return 0;
12 }
```

CSDN @Capsfly

```
a 1
b 2
c 3
d 4
e 5
f 6
g 7
h 8
i 9
j 10
k 11
l 12
m 13
n 14
o 15
p 0
q 1
r 2
s 3
t 4
```

u 5  
v 6  
w 7  
x 8  
y 9  
z 10

最终结果 opekma

```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin$ qemu-mipsel -g 2333 /home/capsfly/Desktop/mipsel-gdb/bin/bomb
Please input your ID_number:
996007
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Let's begin now!
Phase 1 defused. How about the next one?
1 7 0 0 0 0
That's number 2. Keep going!
0 q 111
Halfway there!
5
So you got that one. Try this one.
opekma
Good work! On to the next...
```

CSDN @Capsfly

## 6. 第六关

```
23 0x00401558 <+88>: sll v0,v0,0x2// $v0=4*$v0
24 0x0040155c <+92>: addiu v1,s8,24// $v1=$s8+24
25 0x00401560 <+96>: addu v0,v1,v0// $v0=$s8+24+4*$v0, 猜测v0是循环次数
26 0x00401564 <+100>: lw v0,12(v0)
```

我们猜测  $\$s8+36+4*\$v0$  保存的是我们输入的 6 个数字，输出一看，果然是

```
0x7ffff118: 7
(gdb) x/6uw $s8+36
0x7ffff114: 6 7 0 4
0x7ffff124: 2 3
(gdb)
```

写到这里编辑器已经无比的卡，简单说一下思路吧

首先它判定了一下你输入的 6 个数字是否处于(0,7)之间, 并且必须是全排列  
然后它看了一下你的学号的最后一位, 是奇数还是偶数,

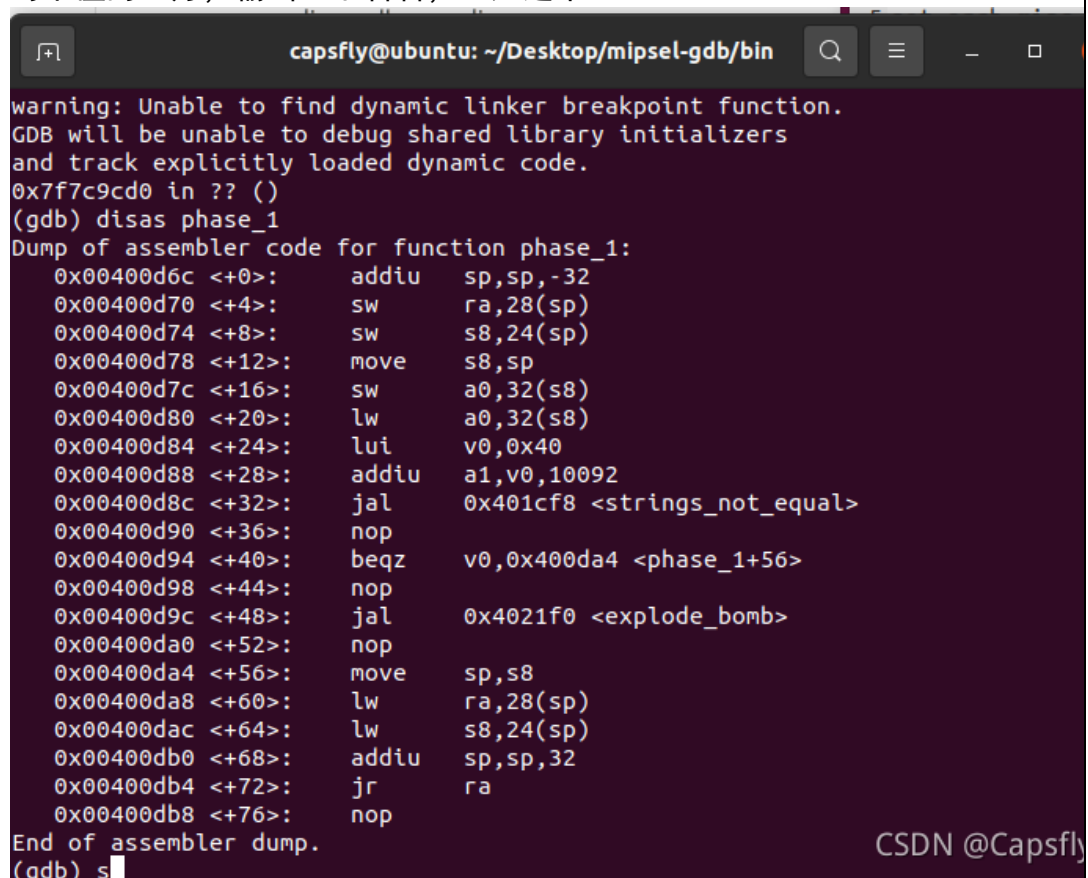
最后你发现它用了一个链表存储, 需要在循环中找到单调不减或者单调不增加的一个序列

我的学号最后一位为奇数, 所以是单调不减少的

最后结果就是 426315

### 结论分析与体会:

1. 学会看官方指令集, 有很多个人博客上的指令都是不全的。看官方指令集能最方便的找到我们想要查看的指令, 这也告诉我们要学好英语
2. 对不懂的地方, 输出一下看看, 比如这个



```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
0x7f7c9cd0 in ?? ()
(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x00400d6c <+0>:      addiu    sp,sp,-32
0x00400d70 <+4>:      sw       ra,28(sp)
0x00400d74 <+8>:      sw       s8,24(sp)
0x00400d78 <+12>:     move     s8,sp
0x00400d7c <+16>:     sw       a0,32(s8)
0x00400d80 <+20>:     lw       a0,32(s8)
0x00400d84 <+24>:     lui      v0,0x40
0x00400d88 <+28>:     addiu   a1,v0,10092
0x00400d8c <+32>:     jal      0x401cf8 <strings_not_equal>
0x00400d90 <+36>:     nop
0x00400d94 <+40>:     beqz    v0,0x400da4 <phase_1+56>
0x00400d98 <+44>:     nop
0x00400d9c <+48>:     jal      0x4021f0 <explode_bomb>
0x00400da0 <+52>:     nop
0x00400da4 <+56>:     move     sp,s8
0x00400da8 <+60>:     lw       ra,28(sp)
0x00400dac <+64>:     lw       s8,24(sp)
0x00400db0 <+68>:     addiu   sp,sp,32
0x00400db4 <+72>:     jr      ra
0x00400db8 <+76>:     nop
End of assembler dump.
(gdb) s
```

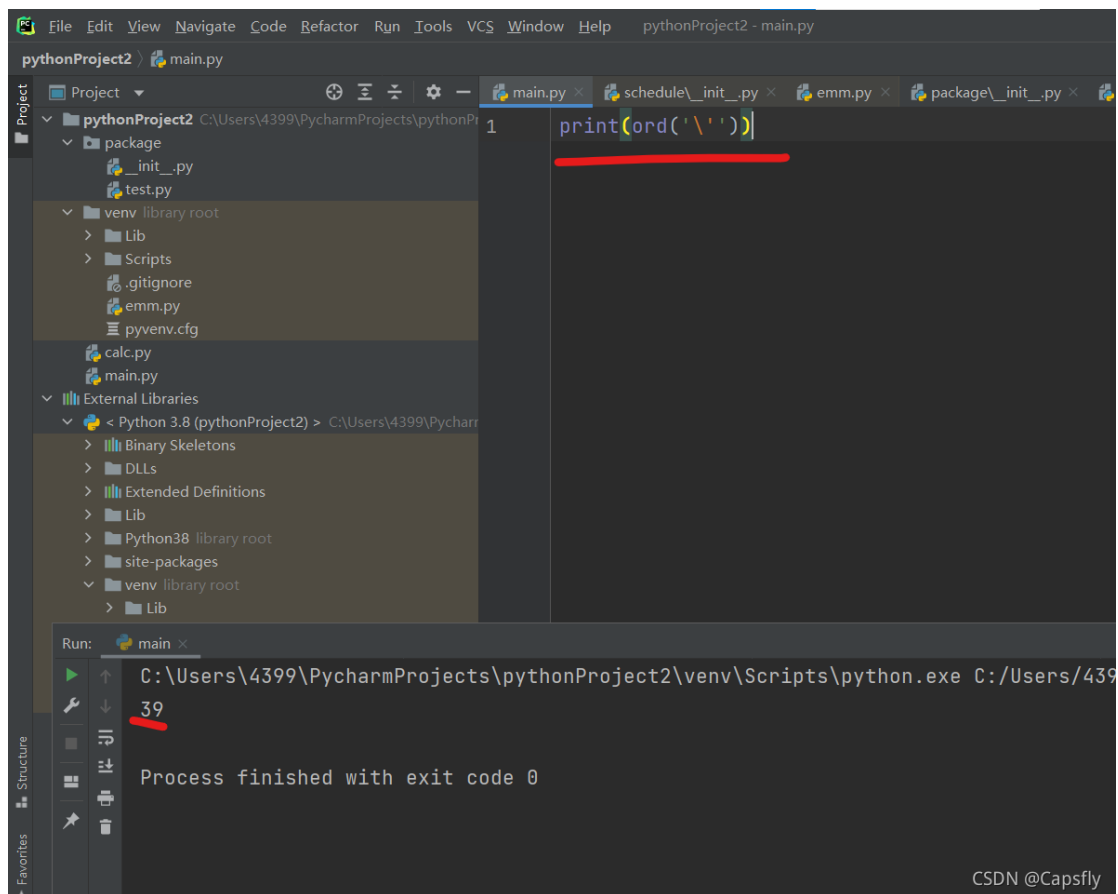
从 beqz 指令那里可以看出, 如果是字符串相等的话, 那么我们就直接跳到 0x400da4, 可以看到 0x00400d9c 是炸弹爆炸的地方, 所以这一关的要点就

变成了，查出原来的函数中保存的 `string` 的内容，然后输入即可

```
0x40276c: 76 'L' 101 'e' 116 't' 39 '\\' 115 's' 32 ' ' 98 'b' 101 'e'
0x402774: 103 'g' 105 'i' 110 'n' 32 ' ' 110 'n' 111 'o' 119 'w' 33 '!'
(gdb) █
```

本来我最开始看到这个 `\` 就不太理解，后来用 `pycharm` 输出了一下 `'\'` 的 `asc` 码发现就是 39, 才发现是哪个 `'\'` 太小了没看到，所以最终拆除炸弹的结果就是 `Let's`

`begin` `now!`



3. 每条指令之间的间隔都是 4 呢？

解决：查了一下 `MIPS` 指令的构成发现，无论是 `R`、`I`、`J` 型指令，都是 32 位的，4 个字节，所以 `MIPS` 指令的间隔都是 4

4. 然后碰到了 `0x00400eb0 <+244>: slti v0,v0,6` 查了一下 这条指令的含义是 `if $v0<6,then $v0=1,else $v0=0`

### Set on Less Than Immediate

31	26	25	21	20	16	15	0
SLTI 001010		rs	rt		immediate		
6		5	5		16		

**Format:** SLTI rt, rs, immediate

## MIPS32

**Purpose:** Set on Less Than Immediate

To record the result of a less-than comparison with a constant.

**Description:**  $\text{GPR}[\text{rt}] \leftarrow (\text{GPR}[\text{rs}] < \text{sign\_extend}(\text{immediate}) )$

Compare the contents of GPR *rs* and the 16-bit signed *immediate* as signed integers; record the Boolean result of the comparison in GPR *rt*. If GPR *rs* is less than *immediate*, the result is 1 (true); otherwise, it is 0 (false).

The arithmetic comparison does not cause an Integer Overflow exception.

**Restrictions:**

None

**Operation:**

```

if GPR[rs] < sign_extend(immediate) then
    GPR[rt] ← 0GPRLEN-1 || 1
else
    GPR[rt] ← 0GPRLEN
endif

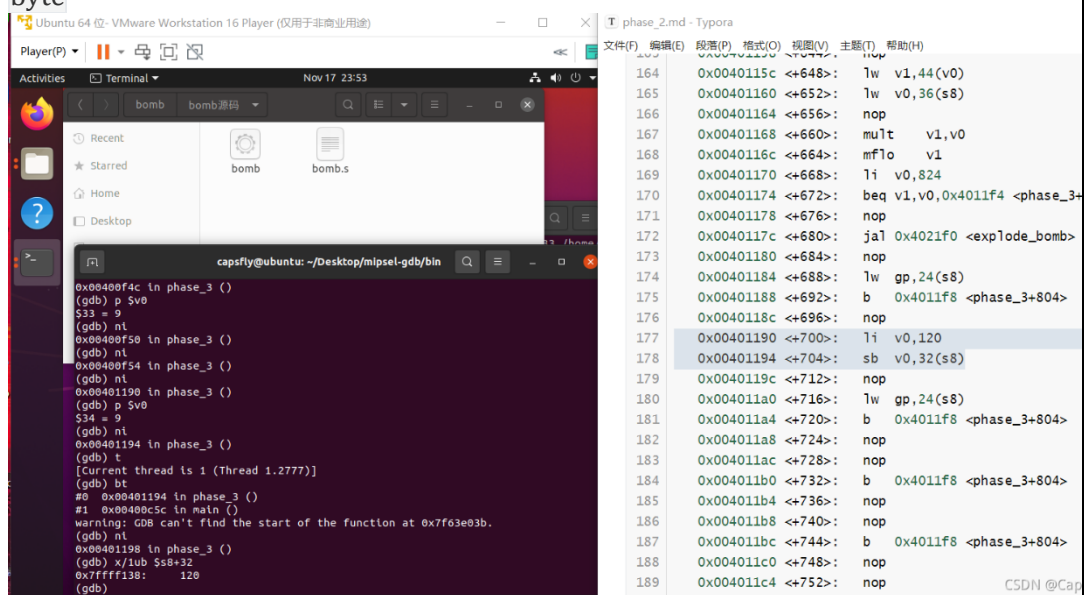
```

**Exceptions:**

None

CSDN @Capsfly

5. 做实验的时候发生了很多问题, 比如说, 输出寄存器的值没有得到预期的结果, 后来发现, 是 `x` 的用法有错误, `sw` 指令, 读取的时候应该读出来的是一个 `word` 而不是 `byte`, 像以下的结果就是正确的, 输入一个 `byte`, 读取一个 `byte`



6. 在拆除实验三的时候，我跟我数学建模的队友发现了一个问题，

```
19 0x00400f18 <+68>: 1w v0,-32636(gp)//$v0=2137465136????
```



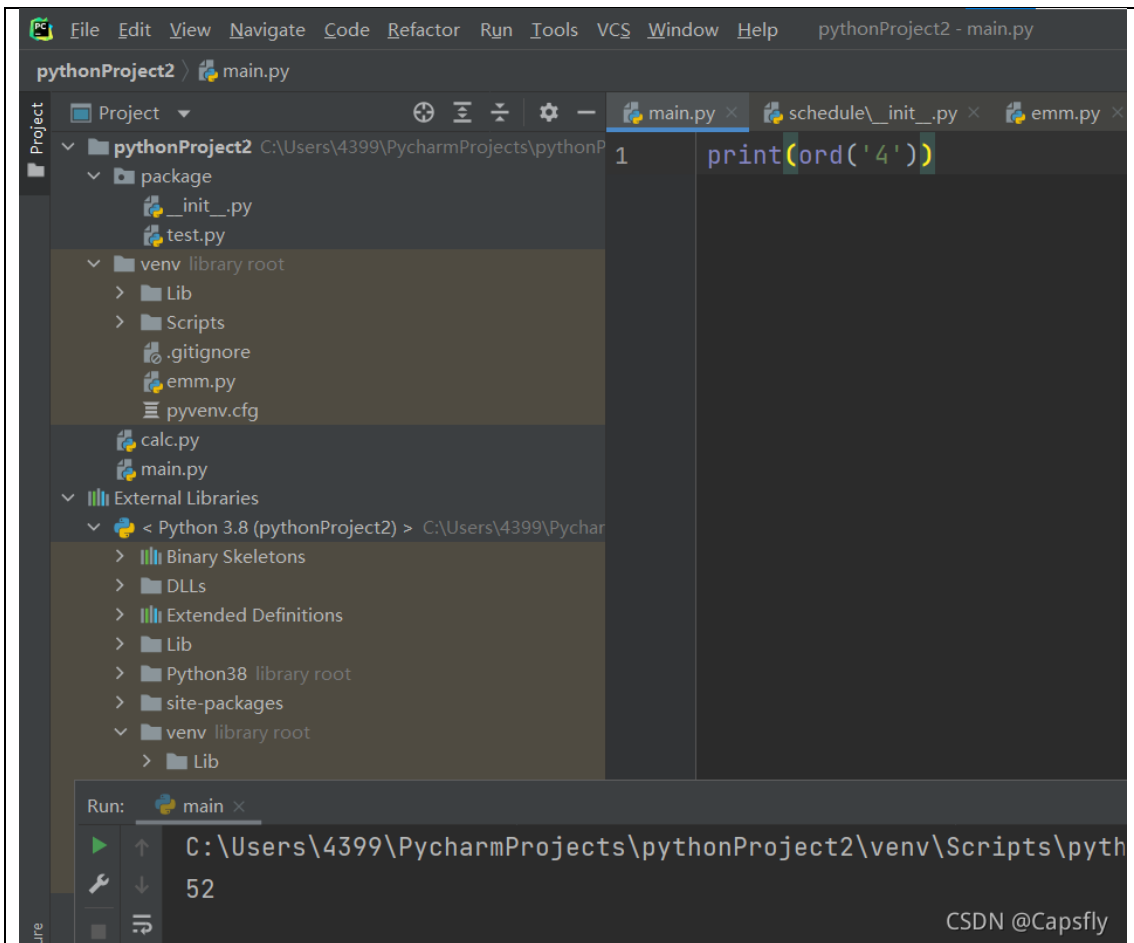
在这条指令中，我们执行后输出\$**v0** 发现是 2137465136，但是 **x/1ub \$gp-32636** 发现却是 48，最后经过一个形势政策的讨论，也没有结果 经过顽强的 debug 发现是输出格式的错误，应该是 **x/1uw %gp-32636**，这样就是 2137465136 了

```
(gdb) p $v0
$2 = 2137465136
(gdb) p $gp-32636
$3 = 4272660
(gdb) x/1ub $gp-32636
0x413214:      48
(gdb) x/1uw $gp-32636
0x413214:      2137465136
```

Ubuntu 64 位- VMware Workstation 16 Player (仅用于非商业用途)

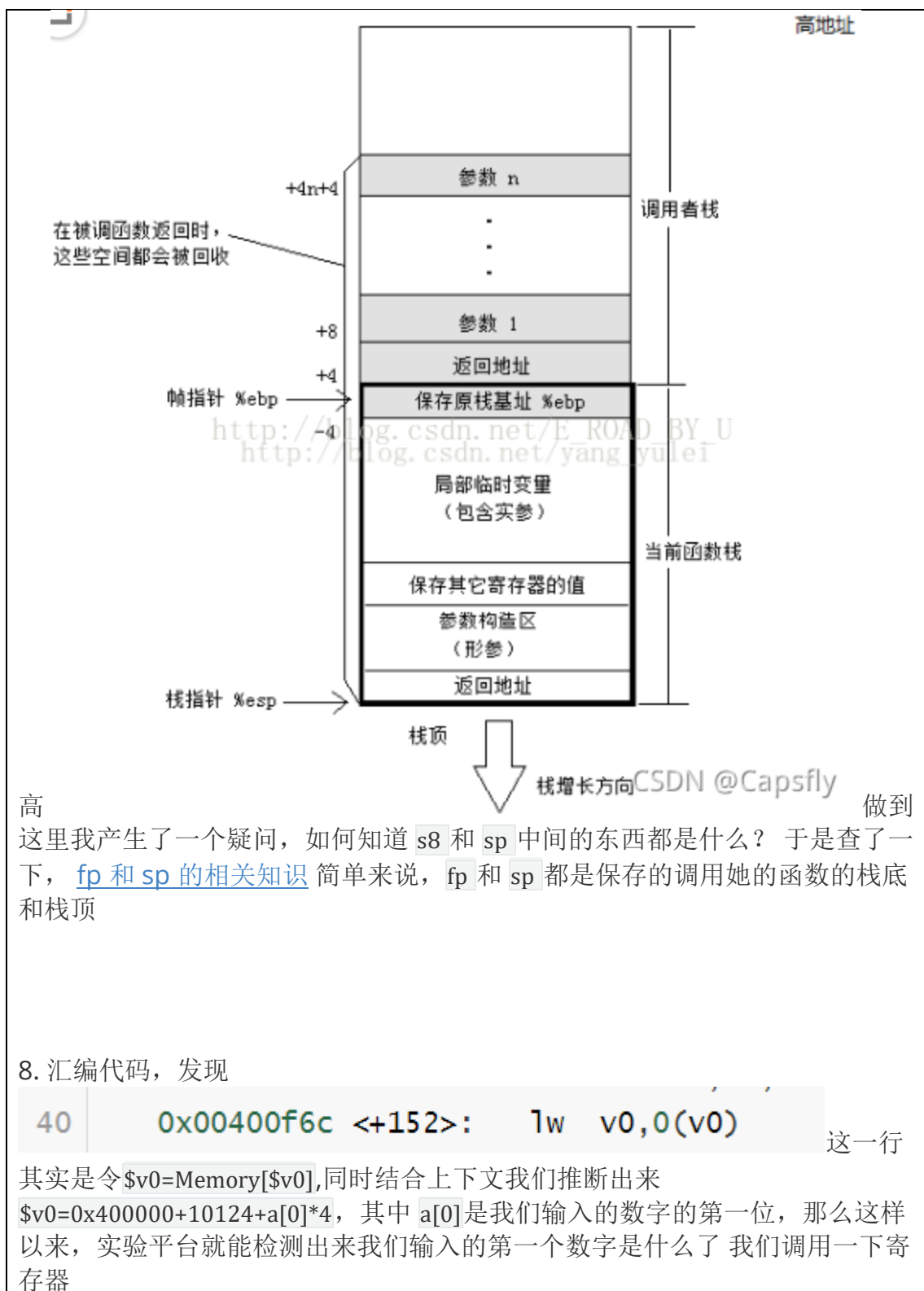
```
Player(P) | [Icons] | [Maximize] [Close]
Activities | Terminal | Nov 18 04:50
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin
(gdb) p $v0
$5 = 4198336
(gdb) i r
zero      at      v0      v1      a0      a1
R0 00000000 00000001 00400fc0 00000004 7a617400 00000000
      t0      t1      t2      t3      t4      t5
R8 00000000 19999999 7f774378 7f774c78 00000005 ffffffff
      s0      s1      s2      s3      s4      s5
R16 00000000 00000000 00000000 00000000 00000000 00000000
      t8      t9      k0      k1      gp      sp
R24 00000001 7f6a779c 00000000 00000000 0041b190 7fffffff
      sr      lo      hi      bad      cause  pc
24000010 00000000 00000000 00000000 00000000 00400f70
      fsr      fir
00000000 00739300
capsfly@ubuntu:~/Desktop/mipsel-gdb/bin(gdb) b* 0x00400fd8
op/mipsel-gdb: Breakpoint 4 at 0x400fd8
(gdb) continue
Please input: Continuing.
996007
Welcome to Breakpoint 4, 0x00400fd8 in phase_3 ()
which to b'(gdb) x/3uw $s8+36
Let's begin0x7ffff13c. 0 52 1
Phase 1 de'(gdb)
1 7 0 0 0 0
That's number 2. Keep going!
```

我输入的是 1 4 6，但是因为它是按照字符读取的，所以我们输出它的 4 的 asc



7. 可以发现，是一样的，同时我们观察 stack 中保存变量的顺序，正好从低到





```

(gdb) p $v0
$9 = 4204432
(gdb) bt
#0  0x00400f6c in phase_3 ()
#1  0x00400c5c in main ()
warning: GDB can't find the start of the function at 0x7f63e03b.
(gdb) ni
0x00400f70 in phase_3 ()
(gdb) p $v0
$10 = 4198336
(gdb) i r

      zero      at      v0      v1      a0      a1      a2      a3
R0     00000000  00000001  00400fc0  00000004  43d1e700  00000000  0000000a  7fffebe9
      t0        t1        t2        t3        t4        t5        t6        t7
R8     00000000  19999999  7f774378  7f774c78  00000005  ffffffff  7f627750  000007fc
      s0        s1        s2        s3        s4        s5        s6        s7
R16    00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
      t8        t9        k0        k1        gp        sp        s8        ra
R24    00000001  7f6a779c  00000000  00000000  0041b190  7ffff118  7ffff118  00400f2c
      sr        lo        hi        bad      cause    pc
      24000010  00000000  00000000  00000000  00000000  00400f70
      fsr       fir
      00000000  00739300
(gdb)

```

CSDN @Capsfly

发现\$*v0* 保存的是一个地址

```
61      0x00400fc0 <+236>:  1i  v0,98
```

那么问题来

了, 如果我们输入的第一位数字不是 1 而是别的数字会怎么办?

```

untitled102 - main.cpp
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int i=0;i<10;i++)
7      {
8          cout<<(0x400000)+10124+i*4<<endl;
9      }
10     return 0;
11 }

```

```

运行:  untitled102 x
C:\Users\4399\untitled102\cmake-build-debug\untitled102.exe
4204428
4204432
4204436
4204440
4204444
4204448
4204452
4204456
4204460

```

CSDN @Capsfly

打印输出了 10 个值, 结果发现正符合预期, 最后 2 个乱码是因为 8 9 是 UI 直接爆炸, 根本不会执行到这里

```

00000000 00739300
(gdb) x/10uw 4204428
0x40278c:  4198268 4198336 4198404 4198472
0x40279c:  4198536 4198604 4198672 4198732
0x4027ac:  25637  1851877735
(gdb)

```

## 我们检测一下， 在线进制转换

支持在2~36进制之间进行任意转换，支持浮点型

☐ 2进制 ☐ 4进制 ☐ 8进制 ☒ 10进制 ☐ 16进制 ☐ 32进制 10进制 ▼

转换数字 4198732

☐ 2进制 ☐ 4进制 ☐ 8进制 ☐ 10进制 ☒ 16进制 ☐ 32进制 16进制 ▼

转换结果 40114c

CSDN @Capsfly

160 0x0040114c <+632>: li v0,98

发现正好有（当然也必须有），同时看上下文，这个地址后面也只有一个 `li` 指令，符合我们的预期

```
46 0x00400f84 <+176>: lw v0,-32660(gp)
47 0x00400f88 <+180>: nop
48 0x00400f8c <+184>: lw v1,44(v0)
```

看到这几行指令，我认为那里保存的是我输入的数据，输出一看，确实是 `ID-num`，同时温馨提示，尽量不要选择特别奇怪的数字，比如连着 2 个 0，你很有可能理解为那个是未初始化的内存单元的值

```
(gdb) X3uw $v0+44
Undefined command: "X3uw". Try "help".
(gdb) x/3uw $v0+44
0x413290 <ID_num+44>: 7 661939532 1700929651
(gdb) x/1uw $v0+48
0x413294 <input_strings>: 661939532
(gdb) x/1uw $v0+40
0x41328c <ID_num+40>: 0
(gdb) x/6uw $v0+24
0x41327c <ID_num+24>: 9 9 6 0
0x41328c <ID_num+40>: 0 7
(gdb)
```

CSDN @Capsfly

在分析的过程中，观察这段代码，最终发现这段其实就做了两件事情 1.将 `v0` 寄存器的值放到了 `Memory[$s8+32]`

`ID-num` 的最后一位和输入的最后数字相乘，最终和 `$v0` 比较是否相等，若相等，则跳转，否则爆炸。因为我输入的学号的最后一位是 7，所以需要找到一个 7 的倍数的 `$v0`，最终发现是第一个函数段，所以我们输入的第一个数字应该是 0

```
44 0x00400f7c <+168>: li v0,113
45 0x00400f80 <+172>: sb v0,32(s8)
46 0x00400f84 <+176>: lw v0,-32660(gp)
47 0x00400f88 <+180>: nop
48 0x00400f8c <+184>: lw v1,44(v0)
49 0x00400f90 <+188>: lw v0,36(s8)
50 0x00400f94 <+192>: nop
51 0x00400f98 <+196>: mult v1,v0
52 0x00400f9c <+200>: mflo v1
53 0x00400fa0 <+204>: li v0,777
54 0x00400fa4 <+208>: beq v1,v0,0x4011ac <phase_3+728>
55 0x00400fa8 <+212>: nop
56 0x00400fac <+216>: jal 0x4021f0 <explode_bomb>
```

CSDN @Capsfly

最终我们成功跳转到了

```
202 0x004011f8 <+804>: lb v0,40(s8)
203 0x004011fc <+808>: lb v1,32(s8)
204 0x00401200 <+812>: nop
205 0x00401204 <+816>: beq v1,v0,0x401218 <phase_3+836>
206 0x00401208 <+820>: nop
207 0x0040120c <+824>: jal 0x4021f0 <explode_bomb>
208 0x00401210 <+828>: nop
209 0x00401214 <+832>: lw gp,24(s8)
210 0x00401218 <+836>: move sp,s8
211 0x0040121c <+840>: lw ra,52(sp)
212 0x00401220 <+844>: lw s8,48(sp)
213 0x00401224 <+848>: addiu sp,sp,56
214 0x00401228 <+852>: jr ra
215 0x0040122c <+856>: nop
```

CSDN @Capsfly

从上文的分析中，我们知道 `Memory[$s8+40]` 保存的是输入的第 2 个参数，而 `Memory[$s8+32]` 保存的是 113

```
44 0x00400f7c <+168>: li v0,113
45 0x00400f80 <+172>: sb v0,32(s8)
```

经过分析得出，这个 113 其实表示的是一个字符，所以我们第 2 个输入 `q` 即可

9. 第 4 关的理解：其实第 4 关的斐波那契数列是整个炸弹中最简单的一关

碰到如下

```
46 0x0040136c <+176>: jal 0x401230 <func4>
```

`func4` 做了什么？我们看一下 `func4` 的代码

Dump of assembler code for function `func4`:

```
0x00401230 <+0>: addiu sp,sp,-40
0x00401234 <+4>: sw ra,36(sp)
0x00401238 <+8>: sw s8,32(sp)
0x0040123c <+12>: sw s0,28(sp)
0x00401240 <+16>: move s8,sp
0x00401244 <+20>: sw a0,40(s8)
0x00401248 <+24>: lw v0,40(s8)
0x0040124c <+28>: nop
0x00401250 <+32>: slti v0,v0,2
0x00401254 <+36>: bnez v0,0x40129c <func4+108>
0x00401258 <+40>: nop
0x0040125c <+44>: lw v0,40(s8)
0x00401260 <+48>: nop
0x00401264 <+52>: addiu v0,v0,-1
0x00401268 <+56>: move a0,v0
```

```

0x0040126c <+60>: jal 0x401230 <func4>
0x00401270 <+64>: nop
0x00401274 <+68>: move s0,v0
0x00401278 <+72>: lw v0,40(s8)
0x0040127c <+76>: nop
0x00401280 <+80>: addiu v0,v0,-2
0x00401284 <+84>: move a0,v0
0x00401288 <+88>: jal 0x401230 <func4>
0x0040128c <+92>: nop
0x00401290 <+96>: addu v0,s0,v0
0x00401294 <+100>: b 0x4012a0 <func4+112>
0x00401298 <+104>: nop
0x0040129c <+108>: li v0,1
0x004012a0 <+112>: move sp,s8
0x004012a4 <+116>: lw ra,36(sp)
0x004012a8 <+120>: lw s8,32(sp)
0x004012ac <+124>: lw s0,28(sp)
0x004012b0 <+128>: addiu sp,sp,40
0x004012b4 <+132>: jr ra
0x004012b8 <+136>: nop

```

End of assembler dump.

A. 我们先看这么几个关键点

B.

11	0x00401254 <+36>:	bnez	v0,0x40129c <func4+108> // \$v0<2就跳转到<func4+108>
29	0x0040129c <+108>:	li	v0,1
30	0x004012a0 <+112>:	move	sp,s8
31	0x004012a4 <+116>:	lw	ra,36(sp)
32	0x004012a8 <+120>:	lw	s8,32(sp)
33	0x004012ac <+124>:	lw	s0,28(sp)
34	0x004012b0 <+128>:	addiu	sp,sp,40
35	0x004012b4 <+132>:	jr	ra
36	0x004012b8 <+136>:	nop	
12	0x00401258 <+40>:	nop	
13	0x0040125c <+44>:	lw	v0,40(s8)
14	0x00401260 <+48>:	nop	
15	0x00401264 <+52>:	addiu	v0,v0,-1
16	0x00401268 <+56>:	move	a0,v0
17	0x0040126c <+60>:	jal	0x401230 <func4>

CSDN @Caps

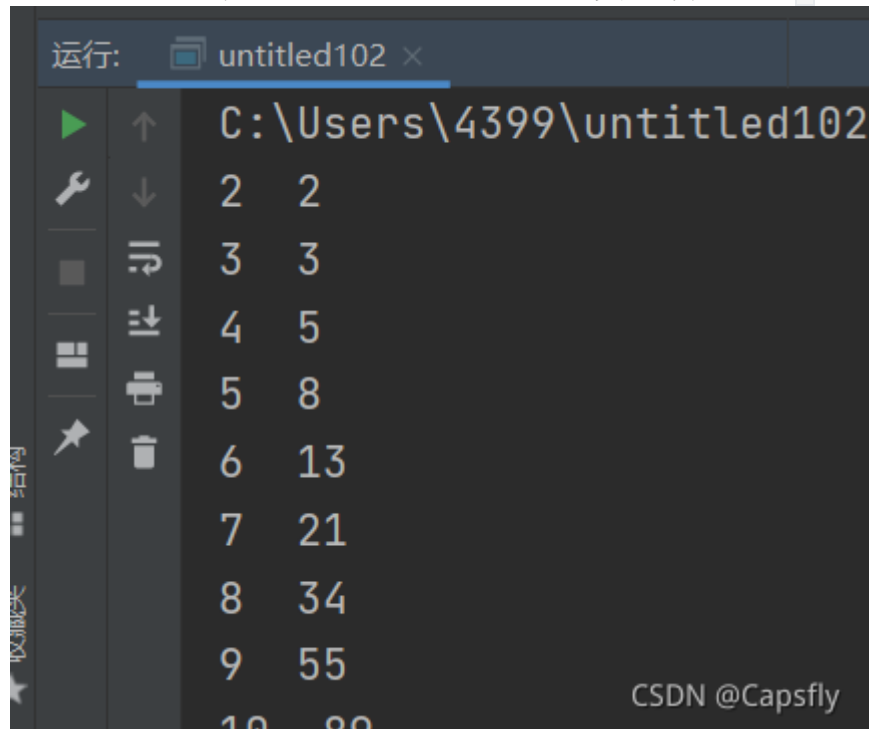
看这一段汇编代码，我们可以很明显的看出，这一段调用了 `func4($v0-1)` 再看

18	0x00401270 <+64>:	nop
19	0x00401274 <+68>:	move s0,v0
20	0x00401278 <+72>:	lw v0,40(s8)
21	0x0040127c <+76>:	nop
22	0x00401280 <+80>:	addiu v0,v0,-2
23	0x00401284 <+84>:	move a0,v0
24	0x00401288 <+88>:	jal 0x401230 <func4>

C. 它调用了 `func($v0-2)` 这是什么？这不是斐波那契数列吗？！ 再看这一段

```
26      0x00401290 <+96>:      addu      v0,s0,v0
```

果然 所以接下来的问题就变成了，斐波那契的第几项是 8 我们打表看一下

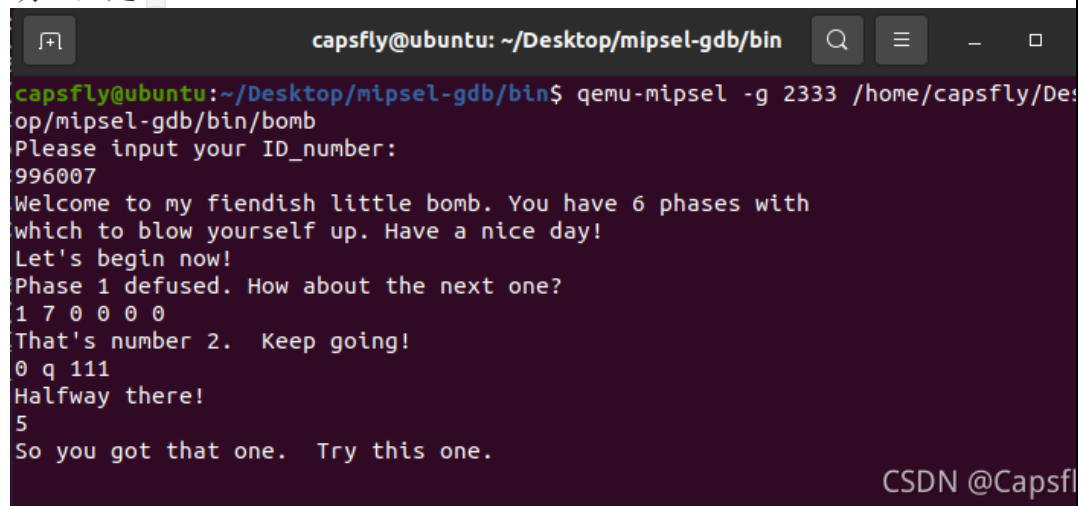


The screenshot shows a debugger window titled '运行: untitled102'. The main pane displays a table of Fibonacci numbers. The first column represents the index (from 2 to 10), and the second column represents the value (from 2 to 89). The text 'CSDN @Capsfly' is visible in the bottom right corner of the window.

Index	Value
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55
10	89

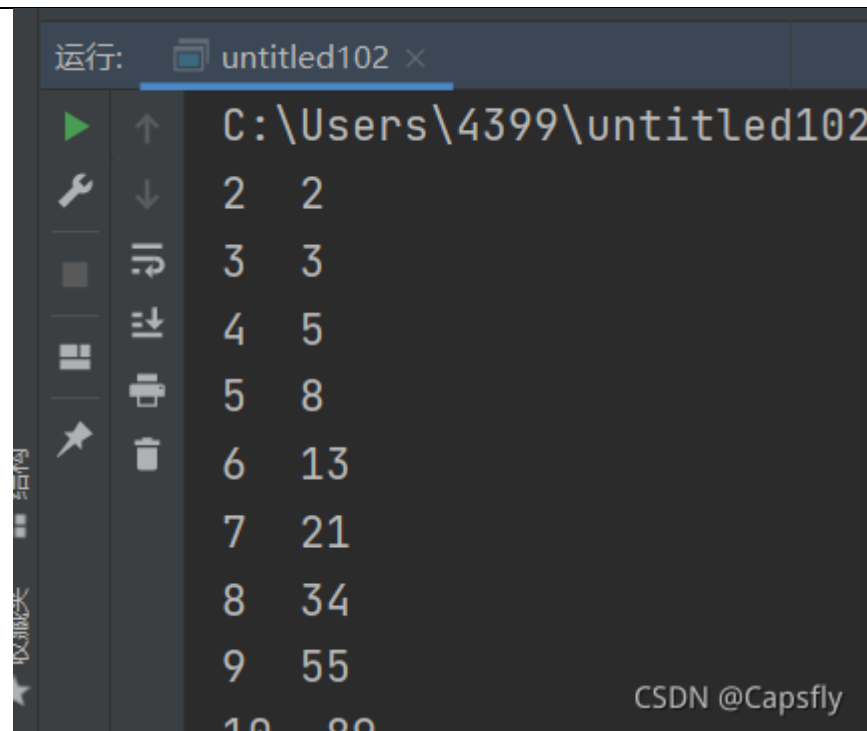
所以答案很

明显，是 5



The screenshot shows a terminal window with the prompt 'capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin'. The user has run the command 'qemu-mipsel -g 2333 /home/capsfly/Desktop/mipsel-gdb/bin/bomb'. The program prompts for an ID number, and the user enters '996007'. The program then displays a series of messages: 'Welcome to my fiendish little bomb. You have 6 phases with which to blow yourself up. Have a nice day!', 'Let's begin now!', 'Phase 1 defused. How about the next one?', '1 7 0 0 0 0', 'That's number 2. Keep going!', '0 q 111', 'Halfway there!', '5', and 'So you got that one. Try this one.' The text 'CSDN @Capsfly' is visible in the bottom right corner.

```
capsfly@ubuntu: ~/Desktop/mipsel-gdb/bin$ qemu-mipsel -g 2333 /home/capsfly/Desktop/mipsel-gdb/bin/bomb
Please input your ID_number:
996007
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Let's begin now!
Phase 1 defused. How about the next one?
1 7 0 0 0 0
That's number 2. Keep going!
0 q 111
Halfway there!
5
So you got that one. Try this one.
```



10.

19 0x00401cc0 <+72>: sltu v0,zero,v0

做实验的时候，遇到了这行代码，不太理解，翻了一下指令集手册，发现这样

**SLTU** Set on Less Than Unsigned

31	26	25	21	20	16	15	11	10	6	5	0
SPECIAL 000000						rs		rt		rd	
6						5		5		5	
								0 00000		SLTU 101011	
								5		6	

**Format:** SLTU rd, rs, rt MIPS32

**Purpose:** Set on Less Than Unsigned  
To record the result of an unsigned less-than comparison.

**Description:**  $GPR[rd] \leftarrow (GPR[rs] < GPR[rt])$   
Compare the contents of GPR *rs* and GPR *rt* as unsigned integers; record the Boolean result of the comparison in GPR *rd*. If GPR *rs* is less than GPR *rt*, the result is 1 (true); otherwise, it is 0 (false).  
The arithmetic comparison does not cause an Integer Overflow exception.

CSDN @Capsfly

本来以为，这是个寄存器值和立即数的比较，但是看了一下，发现是两个寄存器，我就不太理解，后来恍然大悟，这个 `zero` 有没有可能是 `0` 号寄存器？！于是输出了一下，果然是

11. 这行代码其实是最令我无法理解的，我最开始以为，难道它的循环次数是从负数开始？其次，那他怎么判断循环终结？（当然，最后我都搞懂了这些东西）不过分析还是要一步一步来

```

21 0x00401cc8 <+80>: lw v1,12(s8)//可能保存的是读取到了哪里,因为每一个字符都是1byte
22 0x00401ccc <+84>: nop
23 0x00401cd0 <+88>: addiu v1,v1,1
24 0x00401cd4 <+92>: sw v1,12(s8)//保存循环次数?

```

我们推测, `$v1` 才是保存的循环次数, 因为他有很明显的 `+1` 并且保存到了 `$s8+12`, 那么原来的 `$v0` 就肯定不是了, 并且我们结合上下文也能发现, `$v0` 中的值并没有保存, 所以我们推测, `v0` 中的保存的值有别的含义 我们想, 如果它读取结束了, 那么它一定会退出, 也就是

```

25 0x00401cd8 <+96>: bnez v0,0x401ca0 <string_length+40>

```

那么我们往前看 什么时候 `$v0` 会等于 0 呢?

```

19 0x00401cc0 <+72>: sltu v0,zero,v0//if $v0>0 then $v0=1 else 0

```

我们突然想起, 上一个实验的 113 其实是 `q` 的 `asc` 码 那么这个 0 是什么? 这个不是 `'\0'` 的 `asc` 码吗?! 我们输出一下看看

The screenshot shows the PyCharm IDE interface. The main editor window displays a Python script with the following code:

```

1 print(ord('\0'))

```

The left sidebar shows the project structure for 'pythonProject2'. The bottom panel shows the 'Run' output, which displays the value '0' and the message 'Process finished with exit code 0'.

12. 在做实验的时候, 也遇到这样的错误

```

7 0x00401400 <+24>: jal 0x401c78 <string_length>

```

我想进入 `string_length` 函数单步调试, 但是使用 `ni` 直接跳过, 使用 `step` 直接爆炸, 后来发现, 应该使用 `si` 命令 (或者 `stepi`) 同时, 对于一个函数, 如果被调用了很多次, 我们不应该直接在那个位置上设置断点, 在它前面的某个位置设置一个断点, 然后 `si` 即可 否则一直 `c` 会很麻烦 同时



```

11 0x00401ca0 <+40>: lw v0,8(s8)//从<+96>跳转过来的
12 0x00401ca4 <+44>: nop
13 0x00401ca8 <+48>: addiu v0,v0,1
14 0x00401cac <+52>: sw v0,8(s8)
15 0x00401cb0 <+56>: lw v0,12(s8)//$v0保存我们输入的字符串在内存中的地址
16 0x00401cb4 <+60>: nop
17 0x00401cb8 <+64>: lb v0,0(v0)//存放的是我们读取到的字符
18 0x00401cbc <+68>: nop
19 0x00401cc0 <+72>: sltu v0,zero,v0//判断是否读取到了'\0', 如果读取到了, 直接退出
20 0x00401cc4 <+76>: andi v0,v0,0xff//其实没什么用, 这一行
21 0x00401cc8 <+80>: lw v1,12(s8)//加载地址
22 0x00401ccc <+84>: nop
23 0x00401cd0 <+88>: addiu v1,v1,1//没有读取到'\0', 地址的位置+1
24 0x00401cd4 <+92>: sw v1,12(s8)//保存读取到了哪里
25 0x00401cd8 <+96>: bnez v0,0x401ca0 <string_length+40>

```

CSDN @Capsf

我最开始读这段代码的时候没有搞懂, 为什么 `v0` 中存放的是保存的循环次数, 你刚开始就+1 然后保存, 不怕读取到了 `'\0'` 吗? 后来仔细阅读发现, 这个+1 加的其实是上一次读取到的, 如果上一次读取到了 `'\0'` 那么它根本不会跳转到这里! 它就直接退出了! 所以它保存的一定是循环次数! 也就是, 字符串的长度!

然后继续往下看

```

15 0x00401420 <+56>: sw zero,24(s8)

```

不难猜出来, `Memory[$s8+24]` 保存的是 `for` 循环的执行次数

```

22 0x0040143c <+84>: addu v1,a0,v1
23 0x00401440 <+88>: lb v1,0(v1)

```

从这两行代码中, 我们可以才出来, `a0` 保存的是一个内存地址 那么我们输出一下

```

(gdb) x/6c $a0
0x4133d4 <input_strings+320>: 115 's' 99 'c' 121 'y' 116 't' 99 'c' 108 'l'
(gdb)

```

是我们输入的字符串 所以以下这两行的代码的作用就明晰了

```

22 0x0040143c <+84>: addu v1,a0,v1/
23 0x00401440 <+88>: lb v1,0(v1)//取出

```

22

行是用来保存当前读取到的位置 `v1` 用来保存读取出来的字符

13.

```

38 0x0040147c <+148>: lui v0,0x41
39 0x00401480 <+152>: addiu v0,v0,12524
40 0x00401484 <+156>: addu v0,v1,v0//—

```

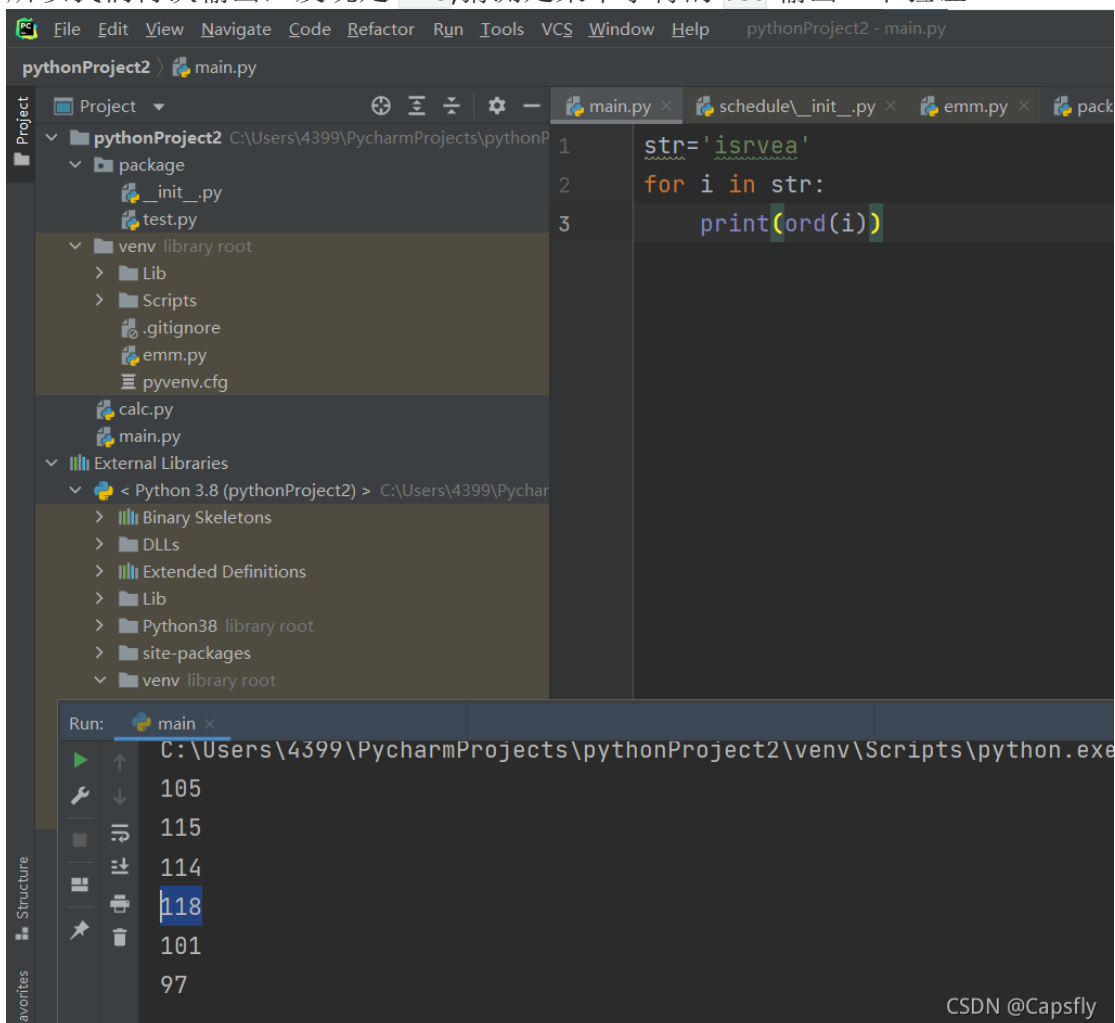
这一段, 基本上看到就能猜到, 这肯定是一个地址了, 那么我们输出一下, 最开始我的猜测是, 这个可能是我们输入的信息之类的, 比如学号或者你输入的字符串 but....

```
(gdb) x/6c 4272364
0x4130ec <array.3607>: 105 'i' 115 's' 114 'r' 118 'v' 101 'e' 97 'a'
```

又因为

```
41 0x00401488 <+160>: 1b v1,0(v0)//一个固定的地址+读取到的字符asc和0xf相与的结果
```

所以我们再次输出，发现是 118,猜测是某个字符的 asc 输出一下验证



的确！也正好是我们推测的结果！对于

```
0x00401494 <+172>: sb v1,4(v0)
```

我们知道，把 \$v1 储存到了循环次数在内存中的地址+循环次数



这种的指令（虽然对解题无帮助，但是个人觉得还是很有必要理解的）  
0x0040145c <+116>: sw v1,12(v0) 这种，修改了系统内存的东西（类似于 C++ 中修改类内私有变量的成员的值，曾经在这个点上 WA 了无数次。。。。）  
对于 0x0040145c <+116> 的指令，验证第 1 次循环如下（循环从 0 计数）

```
(gdb) x/1uw $s8+36+4
0x7ffff130:      3
(gdb)
```

对于 0x00401478 <+144>: lw v1,12(v0)//\$v1=Memory[\$s8+36+4\*循环次数] 指令，

```
(gdb) x/1uw $s8+36+4
0x7ffff130:      3
```

验证如下

```
(gdb)
```

对于 0x00401488 <+160>: lb

v1,0(v0)//\$v1=Memory[\$s8+36+4\*循环次数] 验证如下

```
breakpoint 4, 0x
(gdb) p $v1
$6 = 118
(gdb)
```

对于 0x00401494 <+172>: sb v1,4(v0)//\$v1 储存到了循环次数在内存中的地址+循环次数+4 验证如下

```
x/1uw $s8+24+1+4
ff125:      83902582
x/1ub $s8+24+1+4
ff125:      118
```

哦对了，验证了时候发生了一点小的事故，其实从我的输入中也能看出来错在了哪里，输出格式不正确(这个也是我数学建模的队友 lxr 巨巨跟我讨论了一节课的问题)，观察一下汇编的输入

```
0x00401494 <+172>: sb v1,4(v0) 这个是 store byte 不是 store word!!!!
```

15.看这一行

```
0x00401d5c <+100>: li v0,1
0x00401d60 <+104>: b 0x401de4 <strings_not_equal+236>
```

我们结合函数调用的上下文可以得出，如果这一行被执行了，那么一定会爆炸 如果很幸运，我们没有爆炸 那么我们继续往下看

```
54 0x00401dc8 <+208>: lw v0,28(s8)  //?
55 0x00401dcc <+212>: nop
56 0x00401dd0 <+216>: lb v0,0(v0)  //猜测这里存在经过偏移处理后的结果
```

毫无疑问，从内存中读取出来了一组数据，我们非常有理由怀疑 读取出来的是我们在 phase\_5 中经过处理后的数据或者是本来我们输入的数据

我们往前找一下，因为 s8 很少改变 所以我们在 phase\_5 中找一找 我们发现

```
44      0x00401494 <+172>:  sb  v1,4(v0)//把$v1储存到了循环次数在内存中的地址+循环次数+4，储存的是修改之后的结果，需要注意
```

所以更坚定了我们认为这里读取到的是经过处理后的数据

我们再往下看

```
0x00401d84 <+140>:  xor    v0,v1,v0//按位或
0x00401db0 <+184>:  beqz   v0,0x401dc8 <strings_not_equal+208>
```

这两行其实是这个函数的精髓所在（虽然其实也没有多难）如果 `$v0 xor $v1==0`，那么 `$v0==$v1`

然后

```
38      0x00401d88 <+144>:  sltu   v0,zero,v0//if $v0==0 $v0=0 else 1
```

最后

```
48      0x00401db0 <+184>:  beqz   v0,0x401dc8 <strings_not_equal+208>//
```

判断是否为 0，如果是，则跳转，否则 `$v0=1`，爆炸 所以到这里，我们猜测，是要进行处理后的字符串和给定的字符串进行比较，如果处理后的字符串和给定的字符串相同，那么不爆炸，否则爆炸

16.那么接下来的问题就变成了

3. 它给定的字符串是什么？
4. 找到那个转换函数，使得经过映射后的字符串等于给定的字符串

好，知道这些后，我们先来看一下上文中提到的，那个位置存储的是不是我们经过处理后的字符串 我们重新开

```
Breakpoint 1, 0x004014bc in phase_5 ()
(gdb) x/6ub $s8+28
0x7ffff124:  118  118  98  101  118  117
(gdb) █
```

果然

我们再从

```
0x00401d74 <+124>:  lw     v0,24(s8)//推测是个地址
0x00401d78 <+128>:  nop
0x00401d7c <+132>:  lb     v0,0(v0)
0x00401d80 <+136>:  nop
0x00401d84 <+140>:  xor    v0,v1,v0//按位或
```

写到这里我突然产生了疑问，如果按位或的话，那岂不是最小值取决于给定的字符串？于是百度了一下 `xor` 发现是异或

但是我们的思路方向还是正确的，必须相等，否则爆炸

这里，查看一下给定的字符串

```

0x7ffff0f0:    176    39    64    0    36    241
(gdb) bt
#0  0x00401d88 in strings_not_equal ()
#1  0x004014d8 in phase_5 ()
#2  0x00400cf4 in main ()
warning: GDB can't find the start of the function at 0x7f63e03b.
(gdb) x/6uw $s8+24
0x7ffff0f0:    4204464 2147479844    6    6
0x7ffff100:    2147479816    4199640
(gdb) x/1uw $s8+24
0x7ffff0f0:    4204464
(gdb) x/6uw 4204464
0x4027b0:    1851877735    29556    561475415    1970231584
0x4027c0:    543520295    1969644900
(gdb) x/6ub 4204464
0x4027b0:    103    105    97    110    116 CSDN @Capsfly
(gdb)

```

当然，我相信你从我的繁杂的输入中也能看到我错在了哪里 可以看出 字符串为 giants

17.我们只需要找到是 giants 在内存中的地址即可

好的，问题来了 我们需要查看多少内存空间？ ans:16 因为是输入的字符和 0xf 相与，所以结果一定是 0-e，共 16 个

```

(gdb) x/16ub
0x4027b6:    0    0    87    111    119    33    32    89
0x4027be:    111    117    39    118    101    32    100    101
(gdb)

```

```

0x7ffff0f0:    176    39    64    0    36    241
(gdb) bt
#0  0x00401d88 in strings_not_equal ()
#1  0x004014d8 in phase_5 ()
#2  0x00400cf4 in main ()
warning: GDB can't find the start of the function at 0x7f63e03b.
(gdb) x/6uw $s8+24
0x7ffff0f0:    4204464 2147479844    6    6
0x7ffff100:    2147479816    4199640
(gdb) x/1uw $s8+24
0x7ffff0f0:    4204464
(gdb) x/6uw 4204464
0x4027b0:    1851877735    29556    561475415    1970231584
0x4027c0:    543520295    1969644900
(gdb) x/6ub 4204464
0x4027b0:    103    105    97    110    116 CSDN @Capsfly
(gdb)

```

所以我们接下来只需要寻找 giants 在内存中的地址即可

```

26 0x0040144c <+100>: andi    v1,v1,0xf//字符的asc和0xf相与
27 0x00401450 <+104>: sll    v0,v0,0x2//v0=4*循环次数
28 0x00401454 <+108>: addiu   a0,s8,24
29 0x00401458 <+112>: addu    v0,a0,v0
30 0x0040145c <+116>: sw      v1,12(v0)//Memory[$s8+36+4*循环次数]=当前读取到的字符asc和0xf相与,修改了内存
31 0x00401460 <+120>: lw      a0,24(s8)//循环次数
32 0x00401464 <+124>: lw      v0,24(s8)//循环次数
33 0x00401468 <+128>: nop
34 0x0040146c <+132>: sll     v0,v0,0x2//v0=4*循环次数
35 0x00401470 <+136>: addiu   v1,s8,24//v1保存的是循环次数在内存中的地址
36 0x00401474 <+140>: addu    v0,v1,v0//v0保存的是循环次数在内存中的地址+4*循环次数
37 0x00401478 <+144>: lw      v1,12(v0)//v1=Memory[$s8+36+4*循环次数],读取到的字符asc和0xf相与的结果
38 0x0040147c <+148>: lui     v0,0x41
39 0x00401480 <+152>: addiu   v0,v0,12524//0x410000+12524
40 0x00401484 <+156>: addu    v0,v1,v0//0x410000+12524+偏移(读取到的字符asc和0xf相与的结果)
41 0x00401488 <+160>: lb      v1,0(v0)//一个固定的地址+读取到的字符asc和0xf相与的结果
42 0x0040148c <+164>: addiu   v0,s8,24//循环次数的地址
43 0x00401490 <+168>: addu    v0,v0,a0//v0=循环次数在内存中的地址+循环次数
44 0x00401494 <+172>: sb      v1,4(v0)//把v1储存到了循环次数在内存中的地址+循环次数+4,储存的是修改之后的结果,需要注
意

```

我们从这几段代码中可以发现,我们先从一个偏移的位置找到了函数映射后的结果,然后将这个映射后的结果保存到了一个位置,也正是 `strings_not_equal` 调用出来的位置


所以这样问题就化简成了,在那段地址中,寻找 `giants`

```

(gdb) x/16ub 0x410000+12524
0x4130ec <array.3607>: 105    115    114    118    101    97    119    104
0x4130f4 <array.3607+8>: 111    98    112    110    117    116    102    103
(gdb)

```

可以发现,这段内存中的值包含 `giants` 最终查找发现,偏移量是 `15 0 5 12 14 1` 那么我们需要输入什么呢? 很显然



```

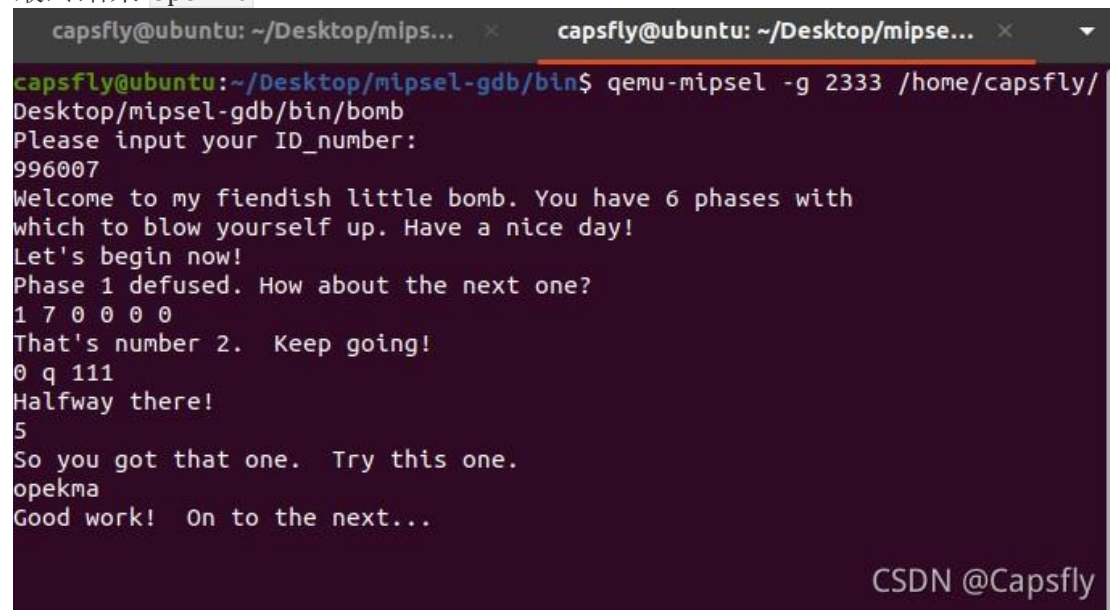
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      for(int i=0;i<26;i++)
7      {
8          printf("%c    ", 'a'+i);
9          cout<<(('a'+i)&0xf)<<endl;
10     }
11     return 0;
12 }

```

- a 1
- b 2
- c 3
- d 4

```
e 5
f 6
g 7
h 8
i 9
j 10
k 11
l 12
m 13
n 14
o 15
p 0
q 1
r 2
s 3
t 4
u 5
v 6
w 7
x 8
y 9
z 10
```

最终结果 opekma



```
capsfly@ubuntu: ~/Desktop/mipsel... x capsfly@ubuntu: ~/Desktop/mipse... x
capsfly@ubuntu:~/Desktop/mipsel-gdb/bin$ qemu-mipsel -g 2333 /home/capsfly/
Desktop/mipsel-gdb/bin/bomb
Please input your ID_number:
996007
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Let's begin now!
Phase 1 defused. How about the next one?
1 7 0 0 0 0
That's number 2. Keep going!
0 q 111
Halfway there!
5
So you got that one. Try this one.
opekma
Good work! On to the next...
```

CSDN @Capsfly

