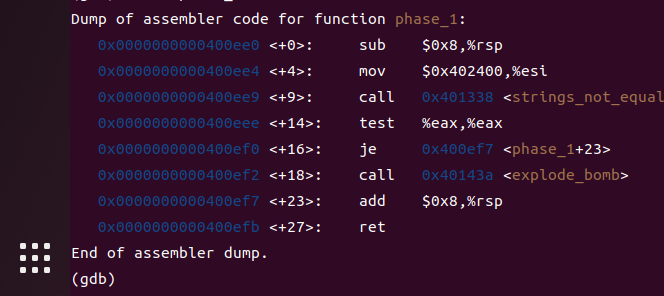
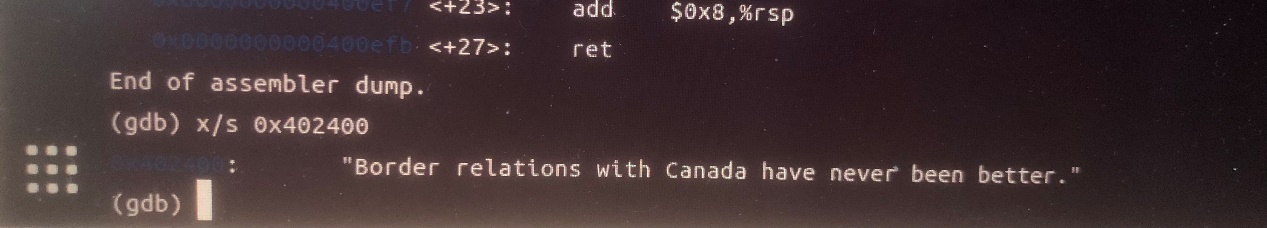
(注意：本报告中所说的行数并不是指汇编代码的真实行数，而是汇编中对应函数首地址的偏移量对应的行)

运行结果的截图位于报告的最底部

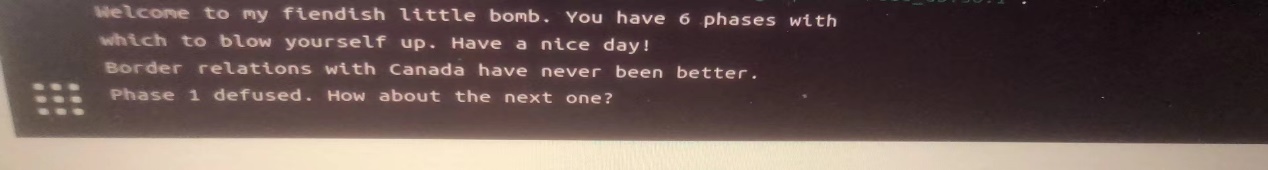
phase\_1:



Phase\_1中，读入我们的一个字符串，同时将位于0x402400的数据存放在%esi中，接着调用strings\_not\_equal函数，可以猜测该函数将我们的字符串与一个字符串作比较，返回一个bool值，再判断该bool值是否为零(test %eax %eax),为零则出栈，结束函数，而位于0x402400的数据是：



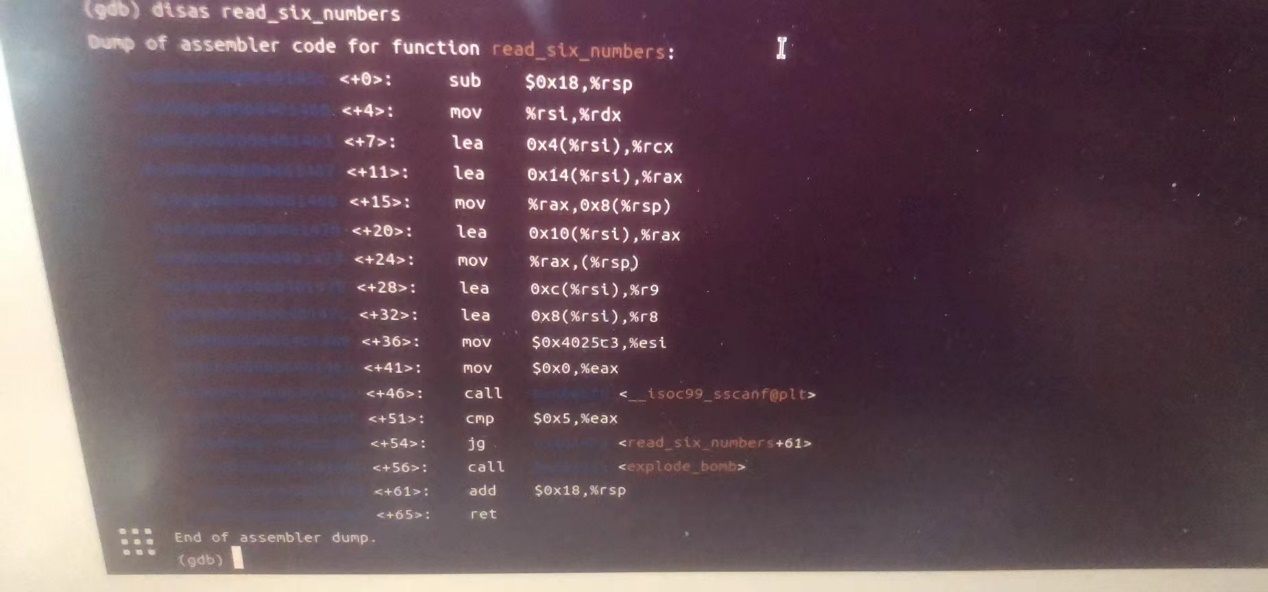
我们输入这一段话：



通过。

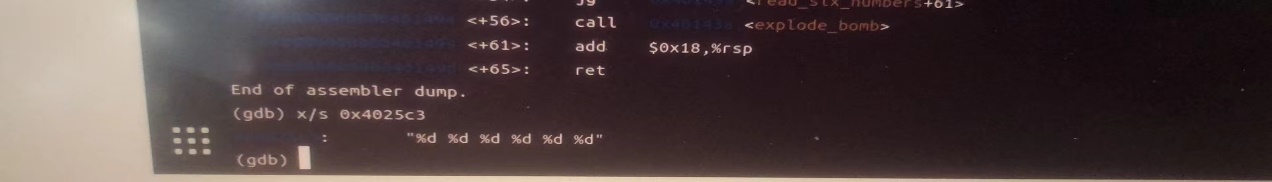
Phase\_2:



在压栈之后，函数将栈顶地址传给%rsi, 并调用函数read\_six\_numbers, 该函数汇编如下：

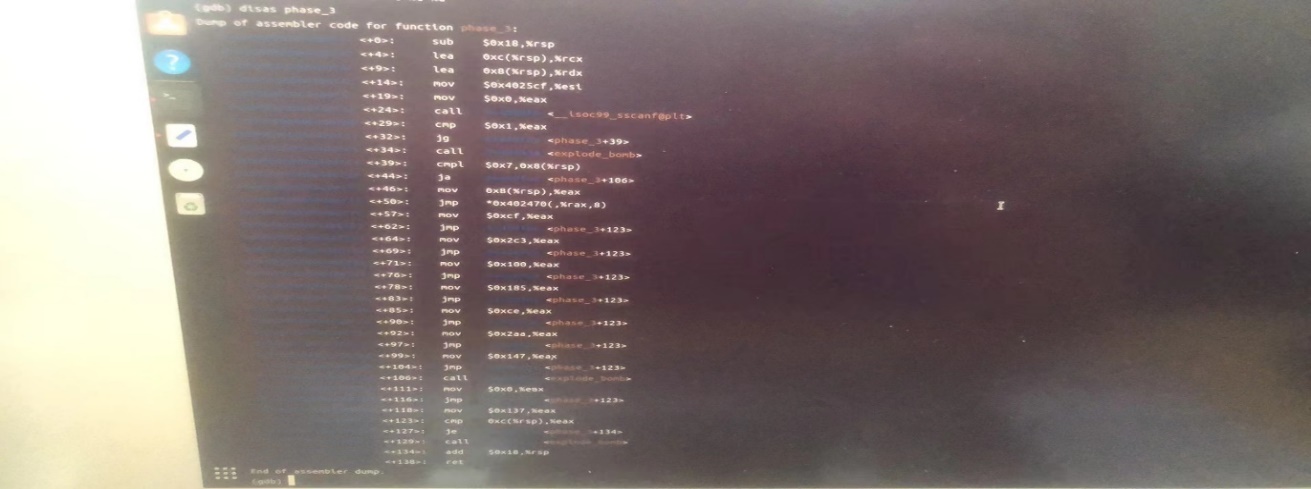
可以看出，除去所有的加载有效地址和mov及压栈操作，该函数调用了\_\_isoc99\_sscanf@plt,

Sscanf在C语言中可将字符串转化为数字存入数组中(栈顶为起点，这也是为什么将栈顶地址作为参数传进去的原因)，并返回成功转化的个数，而指令(mov 0x4025c3, %esi), 查看0x4025c3:

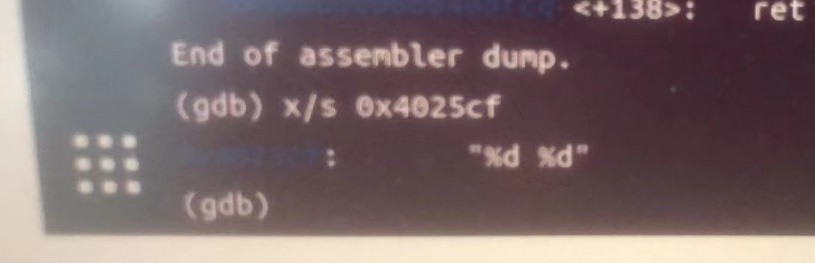


该格式为sscanf进行读入，即，将6个字符转化为对应的数字，此时可以知道read\_six\_numbers的作用为：读我们输入的六个字符，并转化为数字存在栈上(开头的压栈操作)。此时，我们跳回原函数，原函数接着将1和栈顶的第一个元素做对比，即，将1和我们输入的第一个数做对比，因此，我们确定第一个数是1；之后，函数跳到52，将两个地址给到%rbx和%rbp,并跳到27，由于下面有条件判断，且有可能再次跳回27，因此这可能是一个循环，在第一次循环中，%eax保存了栈顶值，并自加(乘2)，将自加的结果与下一个输入的数作比较(%rbx比%eax存值的地址大4，可能是下一个我们的数)，再往下看，%rbx每次加4，而%rbp存的地址正好是%rbx+6\*4次的地址，此时知道，那六个数字的具体位置了，也证实了“将自加的结果与下一个输入的数作比较”，比较结果不一样的话引爆炸弹，因而序列为：1 2 4 8 16 32；

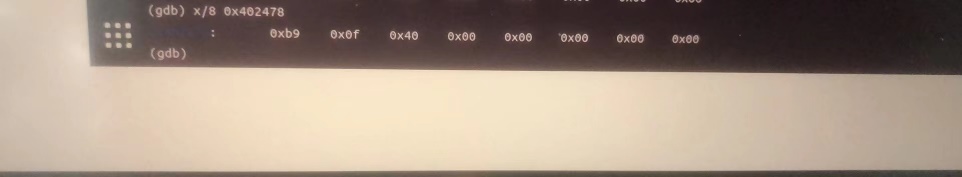
Phase\_3:



开头调用函数sscanf，与phase\_2类似，直接查看0x4025cf的内容：



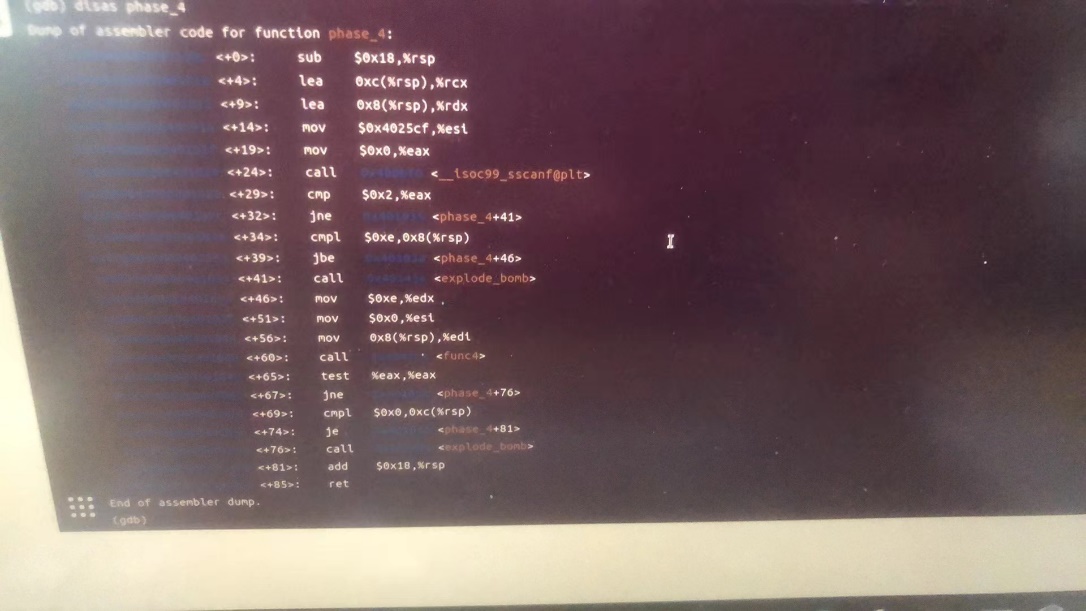
此时可以猜测要输入两个数字，继续读主函数，sscanf返回转化的个数，存到%eax中，接着与1对比，大于1才不会炸，说明我们之前的猜测是有道理的，可能要输入两个数据，继续读，将7和栈上加8的位置的数据做对比，大了的话就引爆炸弹，此时，我们知道开头有两个加载有效地址的操作，一个是%rsp+c,另一个是%rsp+8, c > 8, 因此栈上加8位置的数应该是第一个，即第一个数要小于等于7，而向下读，%eax保存第一个值，接着出现这个指令：jmp \*0x402470(,%rax,8) 即switch语句，我们去看下面每一个分支，都不会直接引爆炸弹，除最后几行外，都是跳转123行，因此，可能有多个答案，我们假定第一个数为1，计算跳转的位置为：



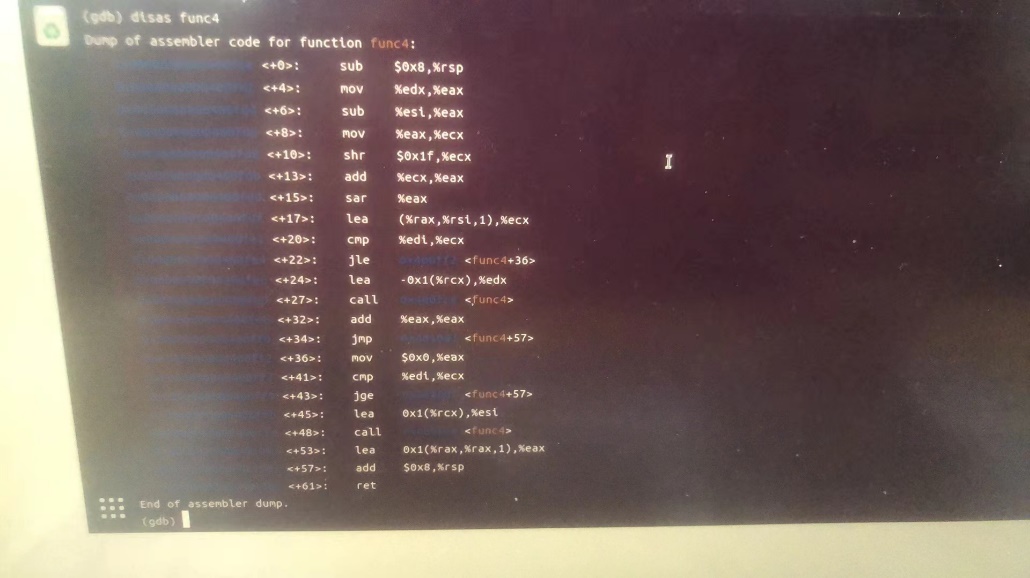
由于x86-64为小端法机器，因此跳转地址为0x400fb9，继续读，此时跳到118行，将0x137(311)放到%eax中，与第二个数比较，相等的话出栈，函数结束，因此，第二个数就是0x137(311),

得到答案：1，311；(也有其他答案，因为123行是比较行，而每个分支都有具体值)

Phase\_4:



前面内容与phase\_3高度雷同，我们可以知道，应该也要输入两个数字，29行中sscanf与2对比的式子证明了我们的猜测，继续读，他将第一个数与14对比，小于等于就不会炸，此时我们确定第一个数小于等于14，接着是三个传参操作，%edx存下14，%esi存下0，%edi存下第一个数，调用func4: (下一页)

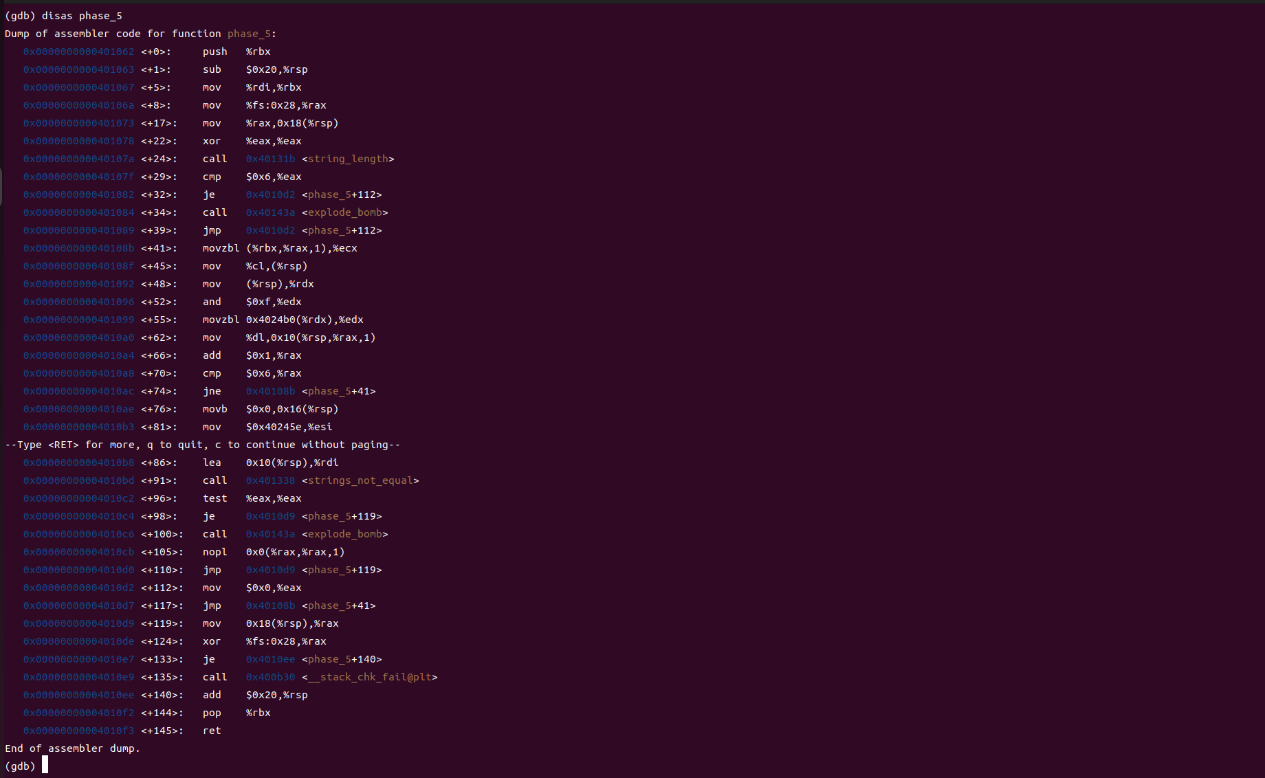


从第0行到第22行执行了如下操作：将%edx(14)移给%eax，%eax自减0，将结果移给%ecx之后逻辑右移31位(取符号位)，加给%eax，%eax为正数，则eax不变，接着算术右移一位，eax此时为7，下一个操作，计算7+0\*1，赋给%ecx，接着比较%edi和%ecx(7),若小于等于7,则跳到36行。在36行%eax赋0操作之后，再一次将%edi和%ecx(7)比较，若大于等于则跳到57行，出栈，结束函数，此时我们可以假定7是第一个数(一路通畅走完func4),该函数有别的分支，取其他数可能会进行递归调用，或引爆炸弹。

第65行是一个test语句，执行与操作，由于%eax为0，接着读69行，69行将0和第二个数对比，不相等则引爆炸弹，因此第二个数为0；

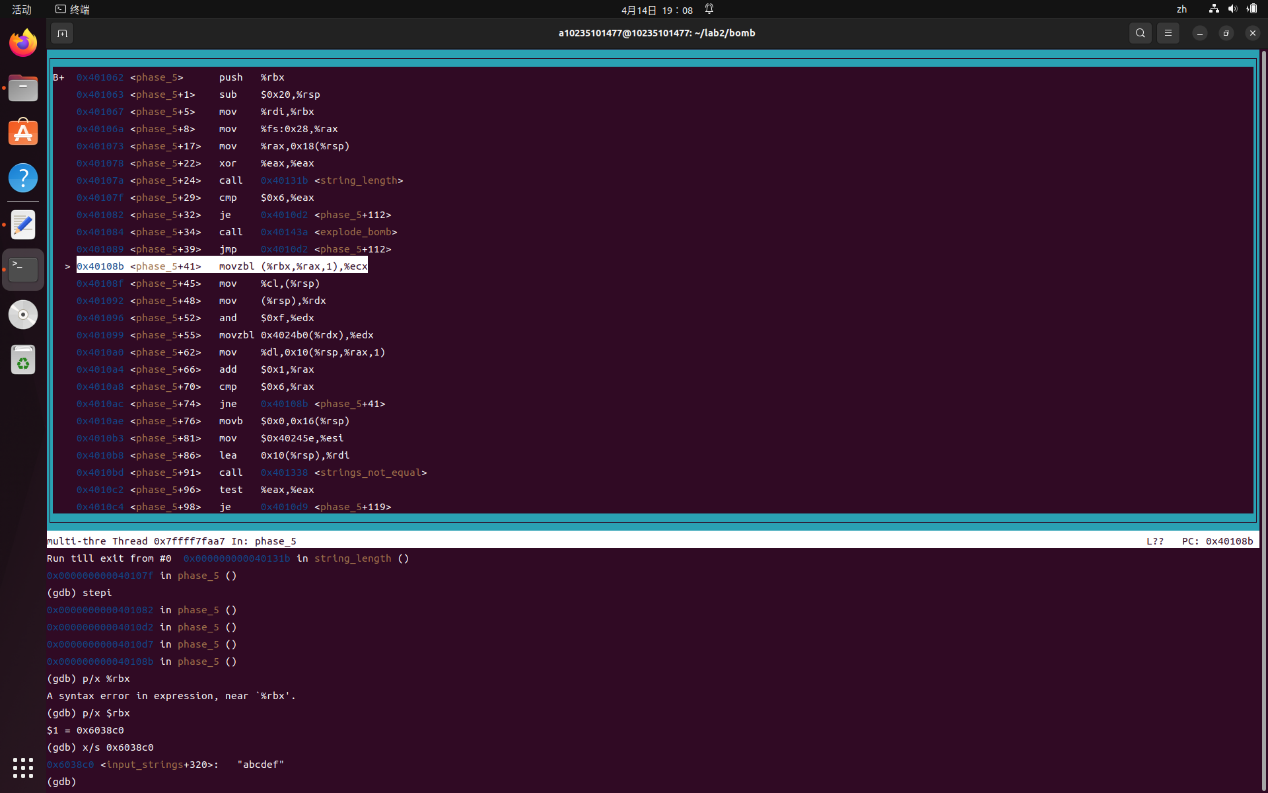
故答案为：7，0；

Phase\_5:

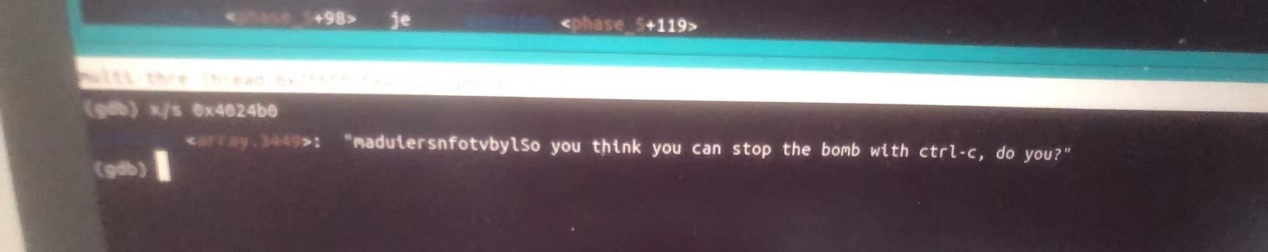


前面几行都是常规的压栈操作，以及一个设立金丝雀值的操作(并将该值放到%rsp+24的位置),接着%eax置零，调用string\_length，由名字可知，该函数计算我们输入字符的长度，并返回该长度，接着与6比较，不相等就引爆炸弹，此时我们知道要输入一个长度为六的字符串，继续读，跳转到112行，将%eax置零后跳回41行，41行到55行的操作：将%rbx+%rax\*1算出来的内存地址中的值进行零扩展后赋给%ecx，接着，将%ecx的低八位(%cl,一个字节)放到栈顶和%rdx中，接着，将该值的低四位(and语句)取出。此时%edx是一个0到15的数值。继续读，接下来两句话是将内存0x4024b0+(0~15)中的内存值放到%rsp+0x10处，然后%rax++，下面是一个跳转比较，可以知道是跳回41行，故41行到74行形成了一个循环，循环六次，

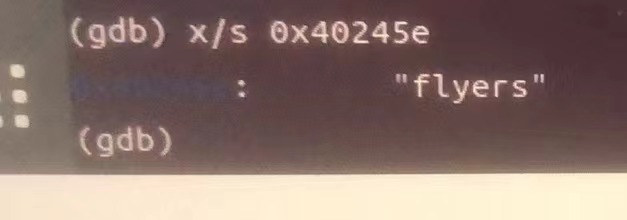
此时与我们的字符串长度相等，有可能这个循环是对我们的字符串做操作，而由于每次%rax+1，%rbx+%rax\*1算出来的值也不一样，因此我们在phase\_5处设一个断点，向下执行到41行，查看一下%rbx的值，并写入字符串“abcdef”，看这个%rbx指向的值是不是我们的字符串：



由图可知，%rbx指向的值正是我们的字符串，此时我们可以确定，该循环对我们的字符串做了操作，通过该操作将内存0x4024b0的某个东西取出，并保存到%rsp的位置，此时我们查看0x4024b0：

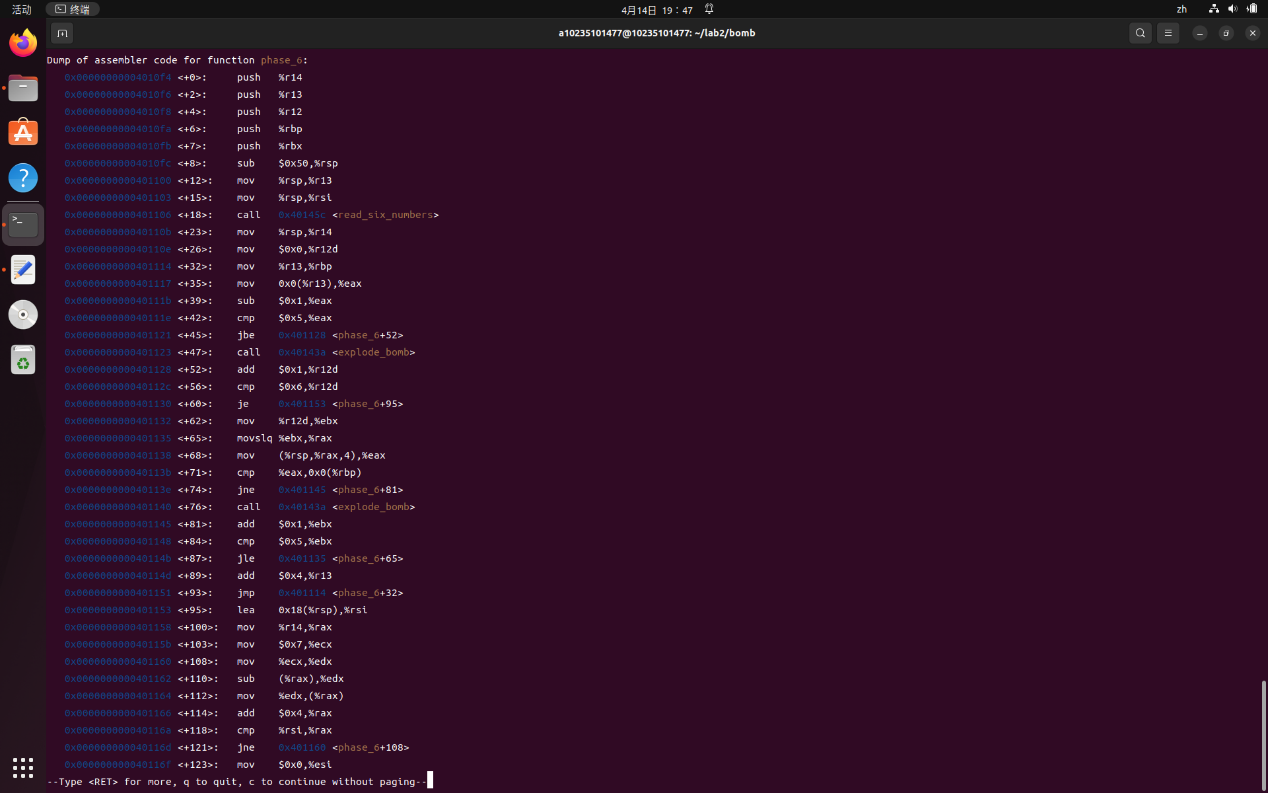


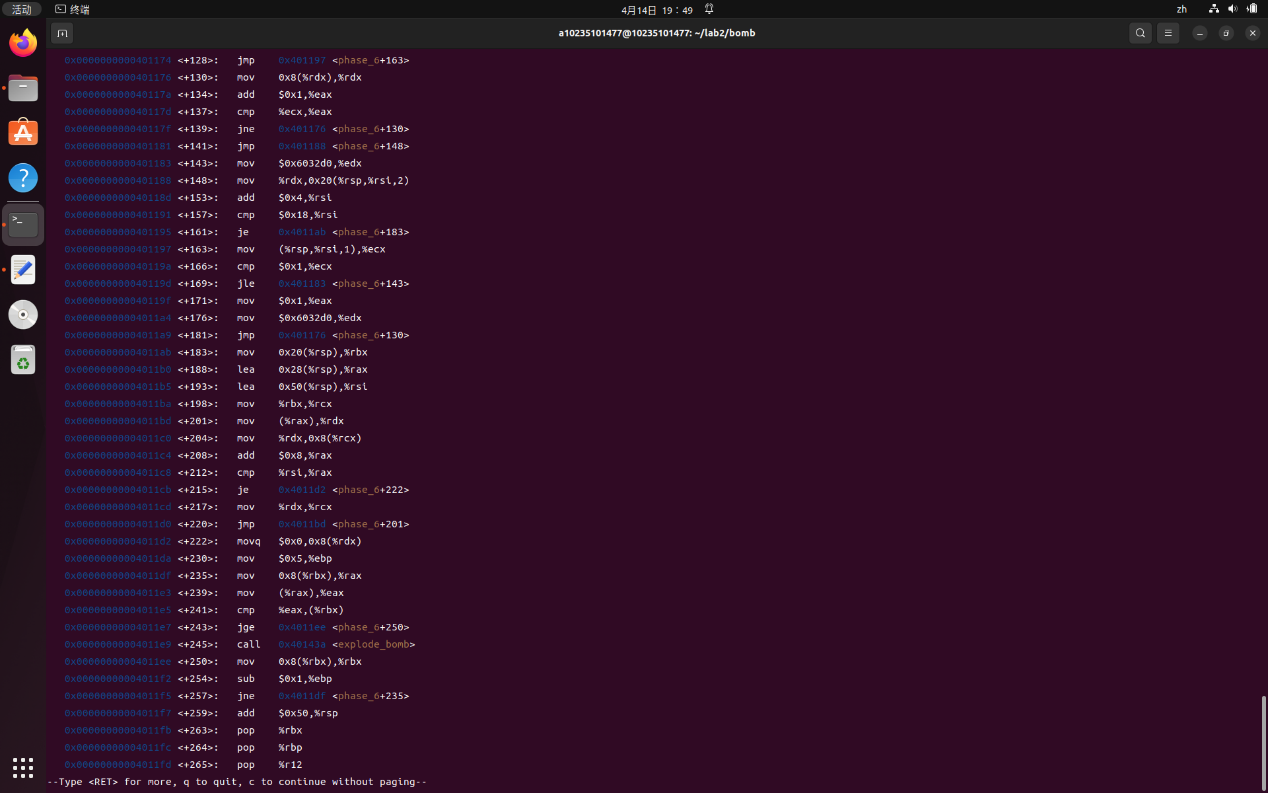
由于%rdx只能是0~15，故只看前16个，此时并无头绪，只能继续读，接下来是两个传参的操作，调用strings\_not\_equal函数，此时观察到%rdi指向的值就是我们的字符串，而%rsi是一个内存保存的值，查看该值：



我们此时可以知道，转换后的字符串应该与“flyers”一样，而转换后的字符串是从“maduiersnfotvbyl”中提取，找到对应位置，即为%rdx在六次循环中分别的值(二进制)：1001，1111，1110，0101，0110，0111.而%rdx的低四位就是%rbx指向的值对应的低四位(继续读，下面查看返回值是否为零，即不相等就爆炸，接着就是查看金丝雀值是否修改，此时已与拆弹无关,故此处加括号)故拆弹的密码需满足：按顺序各字符ASCLL码的低四位必须分别是1001，1111，1110，0101，0110，0111. 找到这样一组字符串：ionefg。即为答案。

Phase\_6:





剩下一小段为弹栈操作，不再贴出。

该题特别困难，先分段处理，第一段：0~26行；第二段：32~93行；第三段：95~121行；

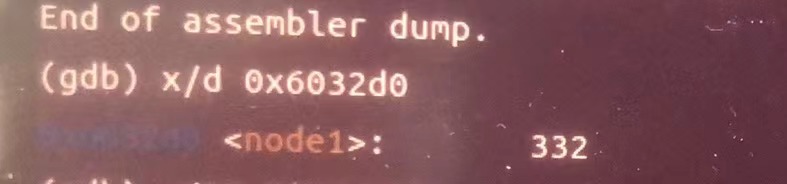
第四段：123~181行；第五段：剩下所有。

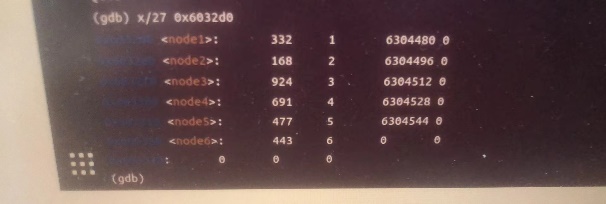
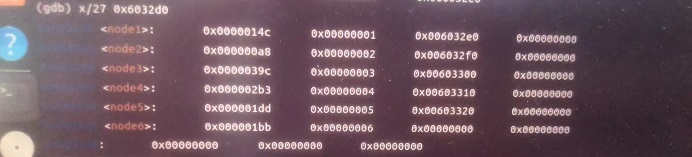
第一段：被调用者保存五个值，调用函数read\_six\_numbers，和phase\_2一样，将我们读入的6个数转化为数组存到栈顶；

第二段：%r14指向数组的第一个值，接着将0赋给%r12，再将%r13的值赋给%rbp，接着第35行，%r13指向栈顶(第12行)，将%r13指向的值赋给eax，若%eax-1小于等于5，则，%r12++，此时%r12为1，小于6，继续向下读，接下来三句话将数组的第二个元素赋给%eax，将该元素与0x0(%rbp)，第一个元素作比较，一样的话就引爆炸弹，接着%ebx++，重复65~87行5次，此时可以推出，第一个数与后面五个数都不一样。走出这个循环，%r13+4，回到32行，可以看出这是一个嵌套循环，%r12存储了循环的次数。%rbp此时指向%r13+4指向的内容，而%r13自身也更新为%r13+4，即这个大的循环，是在遍历读入的数组，将数组的每一个元素与其他元素做对比，相等就炸，故第二段说明，六个数都不一样。

第三段：这一段对数组中的每个数都做了处理，将(7-自身)存到自身的位置。由68行的乘数因子4可以知道这是一个int类型的数组，由于有6个数，栈顶的数组应存到0x18(%rsp)为止，而lea指令将这个截止地址赋给%rsi，下面的cmp指令将%rsi与%eax做对比(%eax的值来自于%r14，而%r14指向栈顶，而%eax又有自身+4的操作，即%eax在这个循环里是一个数组下标)，不相等就进入循环，所以循环进行了6次，每次按顺序处理数组的中的一个数，第103行到112行，先将7赋给%edx(%ecx在这个循环中一直是7)，然后edx减去%rax指向的值，也就是对应数组的值，之后再赋给数组对应位置的值，即所有数变成了7-自身。

第四段：这一段也是对数组进行处理，只不过处理是有条件的，对于此时数组中的每个数，若该数小于等于一，则将位于内存0x6032d0的数据传给一个新数组中的对应位置(0x20(%rsp)作为新数组的开头，每个元素为八个字节)，如果不是，更新rdx，且更新的rdx必须满足这个条件：更新次数为这个数组对应数的值。接着让rdx更新后指向的值存入数组中，接下来，我们查看0x6032d0的值：

  
此时出现了一个node1，即一号节点，而在130行，%rdx=0x8(%rdx)，又因为后面有将%rdx指向的值赋值给新数组的操作，考虑这里有一个链表，则需要知道该链表的存储值，因此我们让链表尽可能多的显示：

(十进制形式)  
(16进制)

故该链表有6个节点，注意：%rdx=0x8(%rdx),这条语句跳了八个字节，也就是说，rdx每次更新的值是下一个结点的地址(每个结点的最后一个参数)，并将这个地址的值存到新数组中(由于相应地址的值根据原数组的相应的值变化而变化，这里并不能告诉我们拆弹的密码)，以上就是这段的功能，该段功能显得怪异，需与下一段结合来看。

第五段：%rbx存储了新数组的第一个元素(链表的某个节点)，%rax是下一个元素，而%rsi就是尾部了(一个指针8字节，0x20到0x50正好6个指针，对上6个节点)，接下来198行到220行，是将链表按照新数组的顺序进行重新组合(%rcx是%rbx，因此0x8(%rcx)是该节点的指针域，而指针域被替换为%rdx，即新数组的下一个元素(%rdx是%rax指向的节点，也就是下一个节点))，而比较%rax和%rsi的值是为了保证所有的节点被重排，此时链表已经根据新数组被重排了，继续读，222行到257行是为了判断重排后的链表是否按照递减的顺序排列，%rbx，由上一个循环可知，仍指向第一个节点，%rax=0x8(%rbx)，即%rbx的指针域，也就是下一个节点，而接下来%eax又保存了其节点的值，并与%rbx结点的值做比较，较大才不引爆炸弹，又因为%rbx和%rax会不断更新(%rbx指向自己的指针域，即下一个节点)可以知道，前一个结点的值要大于后一个结点的值，我们根据节点的值对节点序号进行如下排列：node3 > node4 > node5 > node6 > node1 > node2 ，也就是说，我们输入的数(设为Ai，0 < i < 7)，node[7-Ai(第三段的结论)]=node[j]，j表示重排后的节点序号，由此我们计算出6个数：7-A1=3，7-A2=4，7-A3=5，7-A4=6，7-A5=1，7-A6=2，故密码为：4，3，2，1，6，5。

完整答题截图：

