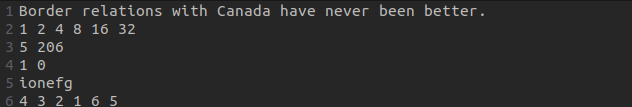
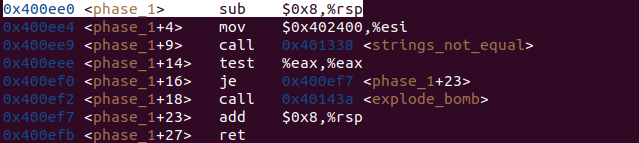
**Bomb Lab**

1. **Answer:**



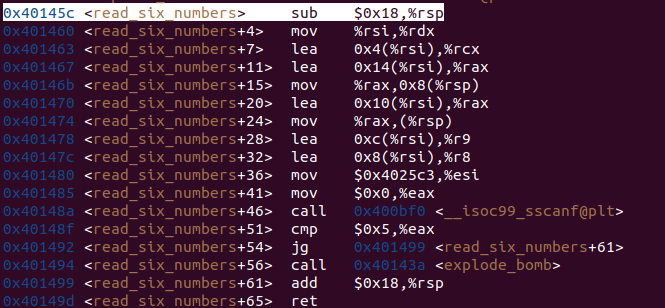
1. **Analyses:**
2. **Phase 1**

通过检验储存在0x402400的字符串(gdb: x/s 0x402400)可得到一个字符串“Border relations with Canada have never been better.”，再通过查看寄存器中值的情况(gdb: info registers)，可以观察到寄存器%rdi（作为传入string\_not\_equal()函数的第一个参数（%rsi为第二个传入的参数，储存的是0x402400，即目标字符串的地址））的值为0x603780，通过检验储存在其中的字符串(gdb: x/s $rdi)可知，%rdi寄存器存储的是输入的字符串。

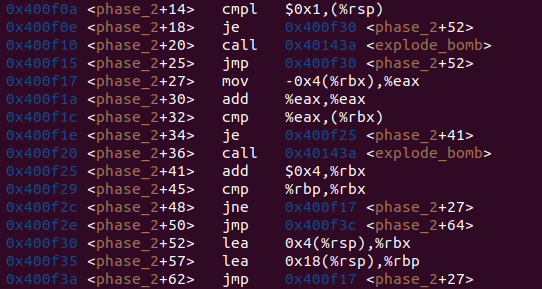
通过测试可以知道string\_not\_equal()函数是用来检验传入的两个字符串是否不相等，相等则返回0，不相等则返回1。通过后续的指令知道，如果返回值是1，就会调用explode\_bomb()函数导致失败，而如果返回值是0，就跳过explode\_bomb()函数。因此输入的字符串必须与存储在0x402400的字符串相等。

通过以上分析可知Phase 1的答案为：Border relations with Canada have never been better.。

1. **Phase 2**

****

通过<read\_six\_numbers+51> ~ <read\_six\_numbers+56>的指令可以知道，scanf函数的返回值必须大于5，即必须读入至少6个数，又通过检验储存在0x4025c3的字符串(gdb: x/s 0x4025c3)可得到 “%d %d %d %d %d %d”，可知系统需要读入6个整数。



通过观察以上汇编指令，可以推断出上面的汇编指令可以大概等价为以下的C语言代码：

for (int i = 1; i < 6; i++) {

  if ((arr[i - 1] + arr[i - 1]) != arr[i]) {

    explode\_bomb();

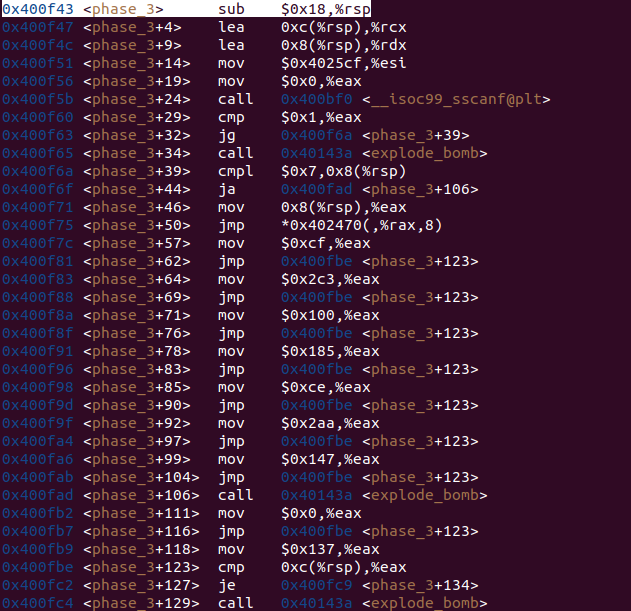
  }

}

通过这段代码可知，读入的这6个数应该组成一个公比为2的等比数列。又通过<phase\_2+14> ~ <phase\_2+20>之间的指令可知，输入的第一个数必须要为1。

综上，答案Phase 2的答案为：1 2 4 8 16 32。

1. **Phase 3**

****

通过<phase\_3> ~ <phase\_3+34>的指令并且通过检验储存在0x4025cf的字符串（得到“%d %d”）可知，系统首先需要读入2个整数。同时也可以知道输入的两个整数分别存储在%rsp + 0x8和%rsp + 0xc中。

通过<phase\_3+39> ~ <phase\_3+46>的指令可知输入的第一个整数不能大于0x7。以此我们按照要求输入一个不大于7的数。通过<phase\_3+46> ~ <phase\_3+104>的指令（加上手动逐步的测试）可以知道，这其间的指令会通过我们输入的第一个数来跳转到特定的地方（通过<phase\_3+50>的指令知道这个跳转地址与输入的第一个数的关系是：%rax + 0x8 + 0x402470），并且跳转过后会执行指令将特定的数字赋值到%eax中。（比如输入第一个数是5，在这里会跳转到0x400f98，然后执行mov $0xce,%eax，即此时%eax赋值为0xce）。

通过<phase\_3+123> ~ <phase\_3+129>之间的指令可以知道，%eax中数字的值要与我们输入的第二个数相等。

综上可以知道，Phase 3所做的可以总结为：通过输入的第一个数的多少跳转到指定的地方然后赋特定值到一个变量中，将这个变量与输入的第二个数进行比较，相等则成功，不相等则失败。

上述过程可以等价成下面的C语言代码：

int val;

switch (number1) {

case x: val = xxx; break;

case x: val = xxx; break;

……

}

if (number2 != val) {

explode\_bomb();

}

return;

综上，Phase 3的答案可以是：5 206（不唯一）。

1. **Phase4**

文本

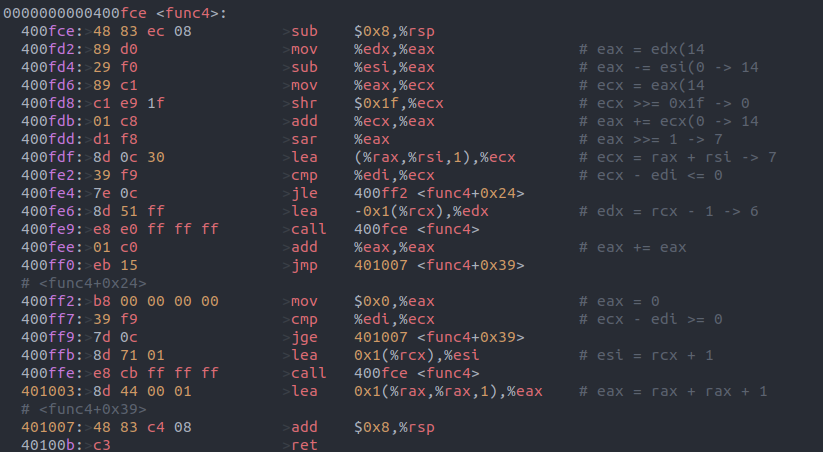
描述已自动生成

通过<phase\_4>~<phase\_4+32>的指令可知需要读入两个整数（同时也可以知道输入的两个整数分别存储在%rsp + 0x8和%rsp + 0xc中），并且由<phase\_34>~<phase\_4+41>的指令可知读入的第一个整数不能大于0xe。

通过<phase\_4+46>~<phase\_4+60>的指令可知调用func4()函数时，分别传入了输入的第一个整数、0和0xe。

通过<phase\_4+65>~<phase\_4+67>的指令可知函数func4()必须返回0才能使得不调用explode\_bomb()函数，并且通过<phase\_4+69>~<phase\_4+76>的指令可知输入的第二个整数必须是0。

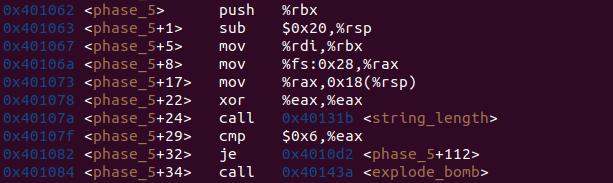
通过以上分析目前知道：输入第一个整数不能大于0xe且第二个整数值为0和func4()的参数列表及返回值，要具体确定第一个整数的值还需要分析func4()函数。



通过观察func4的代码发现很难推断出其具体的意图，但是通过我们之前的分析知道func4的返回值必须是0。那么在最简单的情况下（不去考虑函数的实际逻辑）可以让寄存器%edi中的值等于7，此时可以让函数的返回值为0。

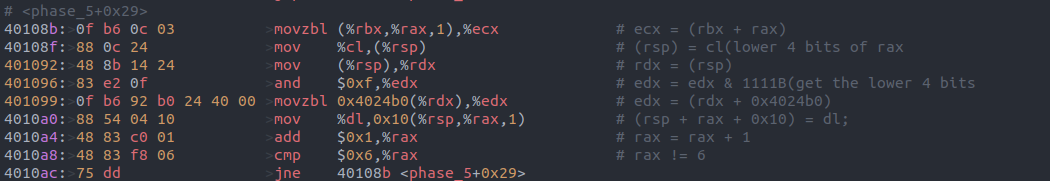
综上，Phase 4的答案为：7 0（在测试过程中发现，第一个整数的值实际上不唯一，实际的规律可以通过分析func4来进行总结，在此先不做探讨）。

1. **Phase 5**



通过测试可以发现string\_length函数的返回值是传入的字符串的长度，以此通过<phase\_5+29>到<phase\_5+34>指令可以知道这个阶段的输入是一个长度为6的字符串（同时也可以通过检查寄存器的值，发现输入的字符串储存在了%rbx寄存器中）。

接着分析后面的指令：



对存储在0x4024b0的字符串进行检验可以得到：



此时知道0x4024b0存有字符串 “maduiersnfotvbyl”。

观察上面指令可以推断出其进行的操作是：每次从输入的字符串中依次取一个字符，然后取得这个字符对应ASCII码的二进制表示的后四位，然后通过这后四位的数的大小在字符串 “maduiersnfotvbyl”中获取一个字符后，将其依次保存到%rsp+0x10到%rsp+0x16处，并且在字符串后面补充‘\0’作为字符串的结尾（movb $0x0, 0x16(%rsp)）

以上的过程可以等价为以下的C语言代码：

for (int i = 0; i < 6; i++) {

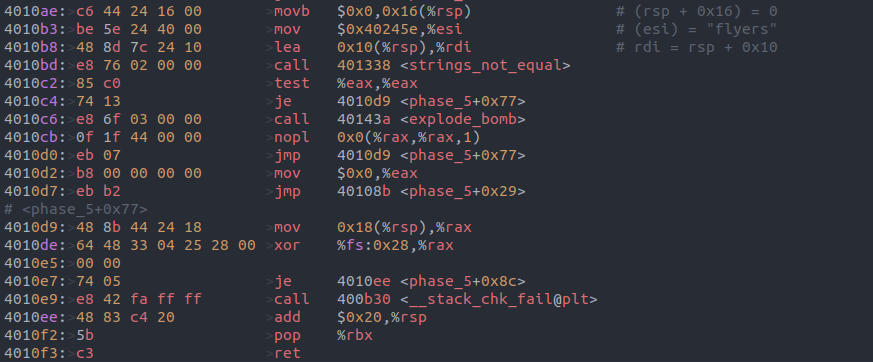
char ch = input\_string[i];

ch = ch & 0xf;

char ch1 = stored\_string[ch];

new\_string[i] = ch1;

}



（检验存储在0x40245e的字符串得到“flyer”）

通过观察上面的指令可以推断出我们在上一步得到的字符串必须要与 “flyer”相等才能成功。

综上，我们需要通过输入一个特定的字符串，这个字符串的每个字符对应的ASCII码的二进制表示形式的后四位的值作为索引，来获取字符串 “maduiersnfotvbyl”中特定的字符，这些字符依次构成一个“flyer”字符串。

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

m a d u i e r s n f  o  t  v  b  y  l

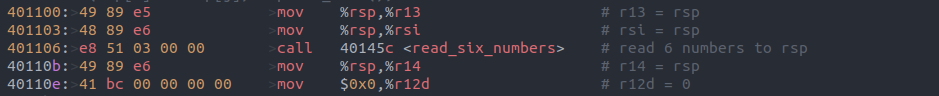
通过观察，我们需要依次获得索引值9 15 14 5 6 7（对应的二进制表示分别为：1001 1111 1110 0101 0110 0111）。

ASCII码的二进制表示的后四位为这些数字的字符依次是：

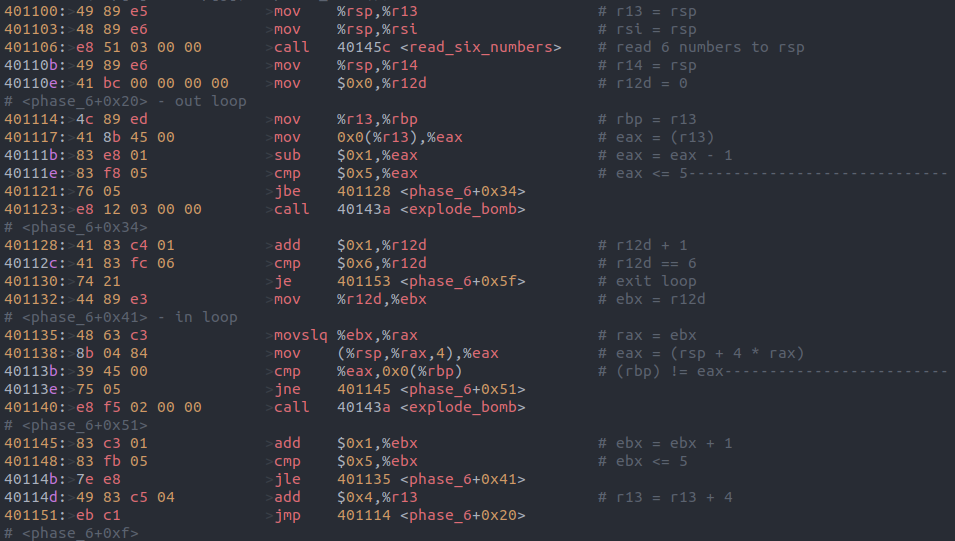
‘I’（或者是‘i’, ‘Y’, ‘y’, ‘9’, ‘)’）、‘O’（或者是‘o’, ‘/’, ‘?’, ‘\_’）、‘N’（或者是‘n’, ‘.’, ‘>’, ‘^’, ‘~’）、‘E’（或者是‘e’, ‘U’, ‘u’, ‘5’, ‘%’）、‘F’（或者是‘f’, ‘V’, ‘v’, ‘6’, ‘&’）、‘G’（或者是‘g’, ‘W’, ‘w’, ‘7’, ‘’’）。

综上，Phase 5的答案可以是IONEFG（不唯一）

1. **Phase 6**



与其他的Phase同理，此时读入6个整数，并且存放在内存中。



通过对以上指令的观察，可以将其等价为以下的C语言代码：

for (int i = 0; i < 6; i++) {

    if (six\_numbers[i] > 6) {

        explode\_bomb();

    }

    for (int j = i + 1; j < 6; j++) {

        if (six\_numbers[i] == six\_numbers[j]) {

            explode\_bomb();

        }

    }

}

也就是说，读入的这6个数每一个都不能大于6，且没有出现重复的数。

文本

描述已自动生成

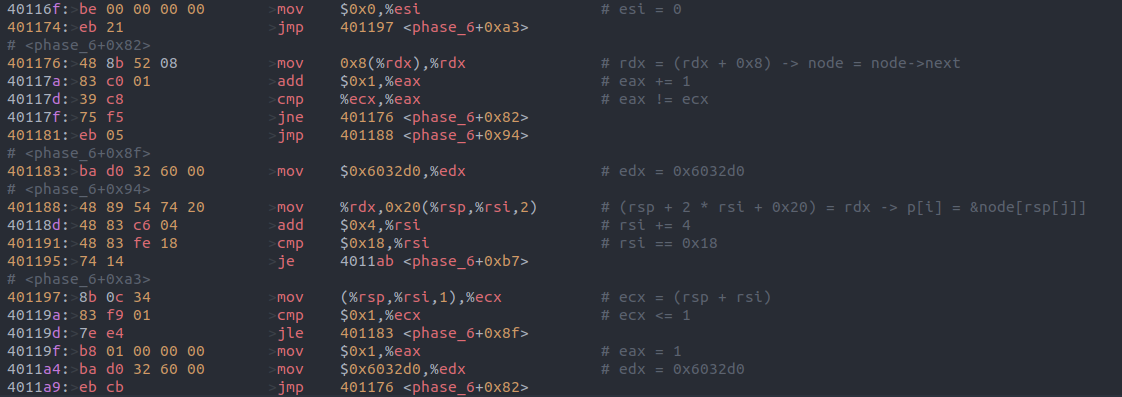
通过对以上指令的观察，可以将其等价为以下的C语言代码：

for (int i = 0; i < 6; i++) {

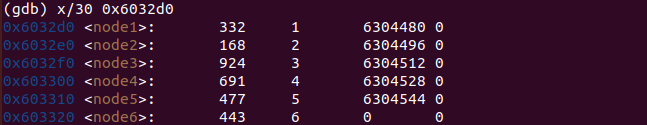
    six\_numbers[i] = 7 - six\_numbers[i];

}

此时是将读入的6个数字让7与其作差。



在分析上面的指令时出现了一个没有根据的立即数0x6032d0，通过打印地址是0x6032d0的内存周围中一些值时并没有什么特别的发现，而是有更多的一些没有根据的数。于是决定一次性检验在地址是0x6032d0的内存中的更多存储单元。



通过检验，得到以上的这些输出。可以明显看到一个有三个数据域的node结构体，将其第三个数据域用十六进制表示后可以看到：



通过观察发现，第三个数据域储存的是每个node结构体的下一个结构体的地址（除了node6的第三个数据域是空，说明是nullptr）。通过这些可以推断，node是一个链表节点。

struct linkedlist\_node

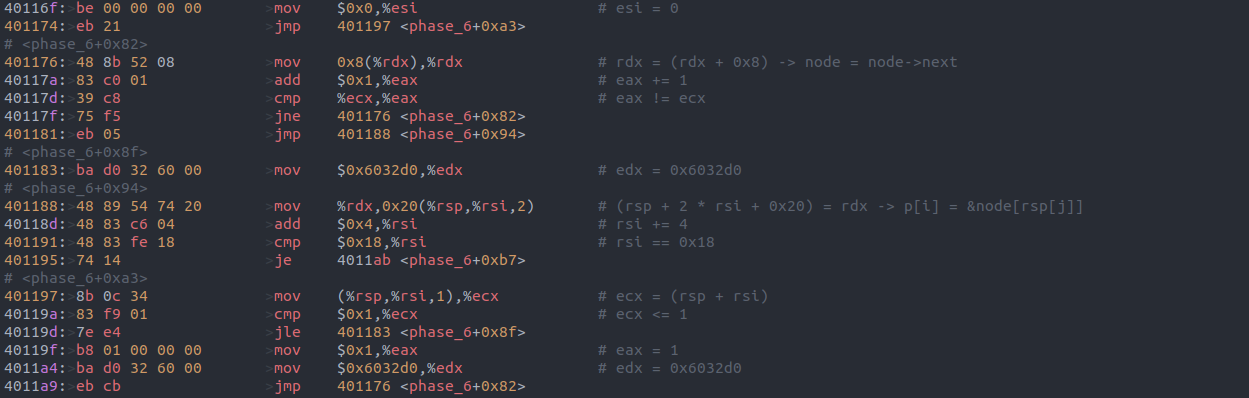
{

    int weight;

    int serial\_number;

    struct linkedlist\_node\* next;

};



通过观察可以将上述指令等价成以下C语言代码：

for (int i = 0; i < 6; i++) {

    int order = six\_numbers[i];

    node\* ptr;

    if (order <= 1) {

        ptr = node1;

    }

    else {

        ptr = node1;

        for (int j = 1; j <= order; j++) {

            ptr = ptr->next;

        }

    }

    memory[i] = ptr;

}

也就是说，以上的指令是根据之前的6个数字的大小作为一个偏移量，去取得对应的node节点，然后将这些node节点依次存放在内存中(%rsp+0x20到%rsp+0x50)。

图片包含 文本

描述已自动生成

通过观察可以将上述指令等价成以下C语言代码：

for (int i = 1; i < 6; i++) {

    memory[i - 1]->next = memory[i];

}

也就是说，以上的指令是将存放在内存中的node节点依次通过给它们的next数据域赋值来实现链表的重新连接，此时的链表的节点的顺序与初始时的不同，这里是顺序是根据之前的6个经过处理的数字进行排列的。

文本

描述已自动生成

通过观察可以将上述指令等价成以下C语言代码：

node\* ptr = memory[0];

memory[5]->next = NULL;

while (int i = 5; i >= 0; i--) {

    if (ptr->weight < ptr->next->weight) {

        explode\_bomb();

    }

    ptr = ptr->next;

}

也就是说，以上的指令是在遍历链表，并且检查链表节点的weight数据域是不是递减的，如果不是，则失败。

由此，我们可以得出Phase 6答案的总体思路：

通过输入特定的6个数字，这6个数字会让7与它们各自作差，得到处理后的6个数字。

将这经过处理的6个数字作为偏移量。按照这6个偏移量的顺序取得链表节点中的各个节点是按照第一个数据域的大小进行降序排列的。所以我们应该根据6链表节点的第一个数据域的值来确定6个数字的具体值。

以下是6个链表节点按照第一个数据域降序排序的结果：

<node3> 924 3

<node4> 691 4

<node5> 477 5

<node6> 443 6

<node1> 332 1

<node2> 168 2

根据以上结果知道，我们需要依次取得第3、4、5、6、1、2个链表节点。又因为要7与初始的6个数字作差，综上，我们可以得出Phase 6的答案为：4 3 2 1 6 5。