

(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:

```
Dump of assembler code for function phase_1:
0x0000000000400ee0 <+0>:      sub    $0x8,%rsp
0x0000000000400ee4 <+4>:      mov    $0x402400,%esi
0x0000000000400ee9 <+9>:      call   0x401338 <strings_not_equal@libc.so.2>
0x0000000000400eee <+14>:     test   %eax,%eax
0x0000000000400ef0 <+16>:     je     0x400ef7 <phase_1+23>
0x0000000000400ef2 <+18>:     call   0x40143a <explode_bomb>
0x0000000000400ef7 <+23>:     add    $0x8,%rsp
0x0000000000400efb <+27>:     ret

End of assembler dump.
(gdb)
```

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：

```
0x0000000000400ef7 <+23>:      add    $0x8,%rsp
0x0000000000400efb <+27>:      ret

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

我们输入这一段话：

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

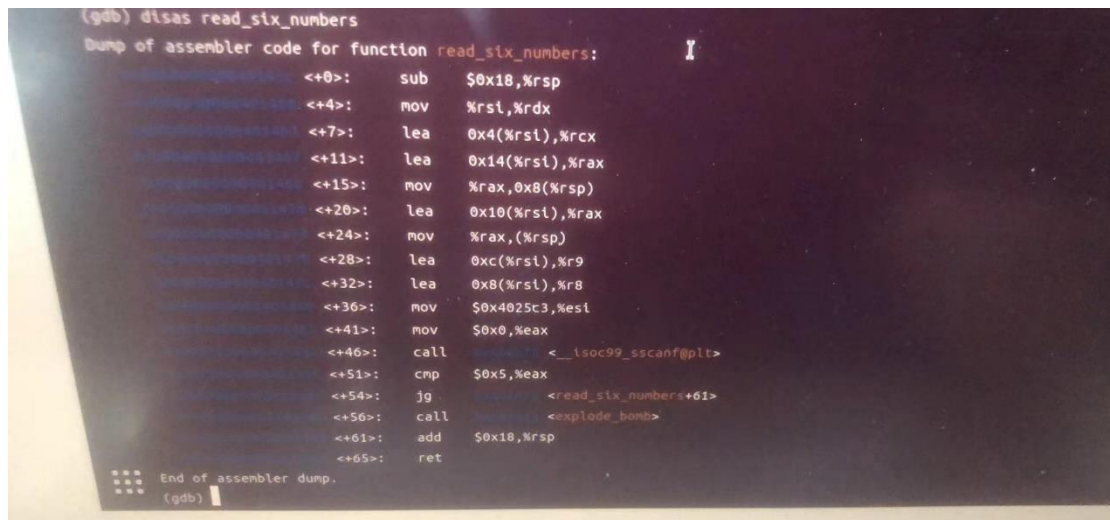
通过。

Phase_2:

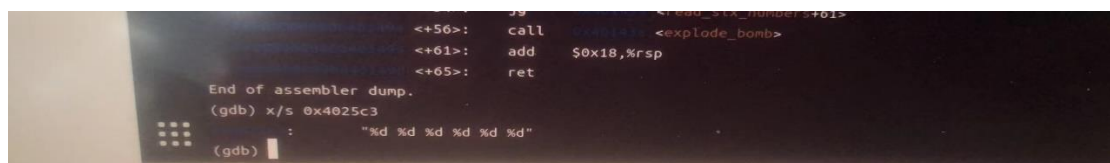
```
Dump of assembler code for function phase_2:
0x0000000000400f00 <+0>:      push   %rbp
0x0000000000400f04 <+4>:      push   %rbp
0x0000000000400f08 <+8>:      sub    $0x2,%rbp
0x0000000000400f0c <+12>:     mov    %rbp,%r13
0x0000000000400f10 <+16>:     call   0x401338 <strings_not_equal@libc.so.2>
0x0000000000400f14 <+20>:     cmpi    $0x1,%r13
0x0000000000400f18 <+24>:     je     0x400f27 <phase_2+31>
0x0000000000400f1c <+28>:     call   0x40143a <explode_bomb>
0x0000000000400f20 <+32>:     jmp     0x400f27 <phase_2+31>
0x0000000000400f24 <+36>:     mov    -0x4(%r13),%eax
0x0000000000400f28 <+40>:     add    %eax,%eax
0x0000000000400f2c <+44>:     cmp    %eax,%r13
0x0000000000400f30 <+48>:     je     0x400f27 <phase_2+31>
0x0000000000400f34 <+52>:     call   0x40143a <explode_bomb>
0x0000000000400f38 <+56>:     add    $0x4,%r13
0x0000000000400f3c <+60>:     cmp    %r13,%r13
0x0000000000400f40 <+64>:     je     0x400f27 <phase_2+31>
0x0000000000400f44 <+68>:     jmp     0x400f27 <phase_2+31>
0x0000000000400f48 <+72>:     lea    0x16(%rsp),%r13
0x0000000000400f4c <+76>:     lea    0x16(%rsp),%r13
0x0000000000400f50 <+80>:     jmp     0x400f27 <phase_2+31>
0x0000000000400f54 <+84>:     add    $0x28,%rsp
0x0000000000400f58 <+88>:     pop    %rbx
0x0000000000400f5c <+92>:     pop    %rbp
0x0000000000400f60 <+96>:     ret

End of assembler dump.
(gdb)
```

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：

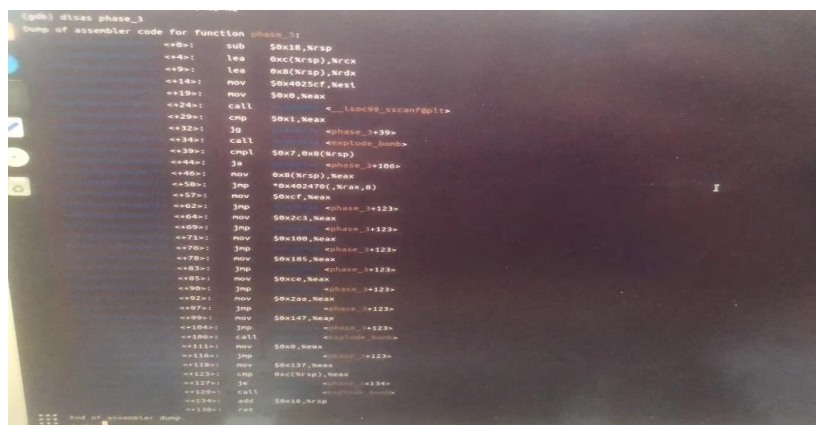


可以看出，除去所有的加载有效地址和 mov 及压栈操作，该函数调用了 __isoc99_sscanf@plt, Sscanf 在 C 语言中可将字符串转化为数字存入数组中(栈顶为起点，这也是为什么将栈顶地址作为参数传进去的原因)，并返回成功转化的个数，而指令(mov 0x4025c3, %esi)，查看 0x4025c3:

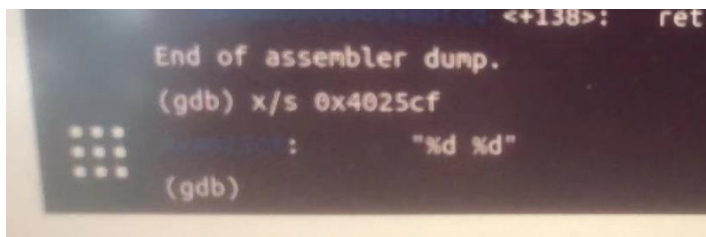


该格式为 sscanf 进行读入，即，将 6 个字符转化为对应的数字，此时可以知道 read_six_numbers 的作用为：读我们输入的六个字符，并转化为数字存在栈上(开头的压栈操作)。此时，我们跳回原函数，原函数接着将 1 和栈顶的第一个元素做对比，即，将 1 和我们输入的第一个数做对比，因此，我们确定第一个数是 1；之后，函数跳到 52，将两个地址给到%rbx 和%rbp,并跳到 27，由于下面有条件判断，且有可能再次跳回 27，因此这可能是一个循环，在第一次循环中，%eax 保存了栈顶值，并自加(乘 2)，将自加的结果与下一个输入的数作比较(%rbx 比%eax 存值的地址大 4，可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：

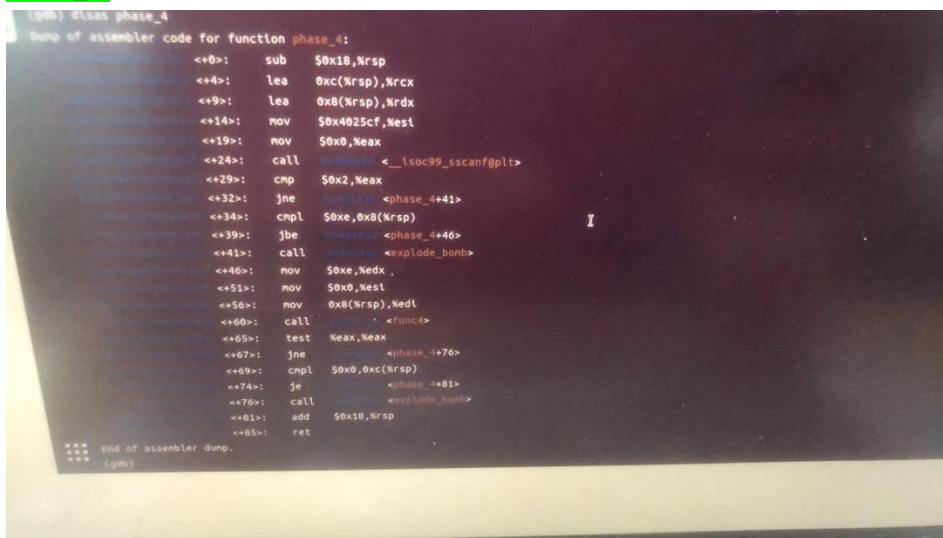


此时可以猜测要输入两个数字，继续读主函数，`sscanf` 返回转化的个数，存到 `%eax` 中，接着与 `1` 对比，大于 `1` 才不会炸，说明我们之前的猜测是有道理的，可能要输入两个数据，继续读，将 `7` 和栈上加 `8` 的位置的数据做对比，大了的话就引爆炸弹，此时，我们知道开头有两个加载有效地址的操作，一个是 `%rsp+c`，另一个是 `%rsp+8`，`c > 8`，因此栈上加 `8` 位置的数应该是第一个，即第一个数要小于等于 `7`，而向下读，`%eax` 保存第一个值，接着出现这个指令：`jmp *0x402470(%rax,8)` 即 `switch` 语句，我们去看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


```
(gdb) disas func4
Dump of assembler code for function func4:
0000000000401000<+0>:  sub    $0x8,%rsp
0000000000401004<+4>:  mov     %edx,%eax
0000000000401006<+6>:  sub     %eax,%eax
0000000000401008<+8>:  mov     %eax,%ecx
000000000040100a<+10>: shr     $0x1f,%ecx
000000000040100c<+12>: add     %ecx,%eax
000000000040100e<+14>: sar     %eax
0000000000401010<+16>: lea     (%rax,%rsi,1),%ecx
0000000000401012<+18>: cmp     %edi,%ecx
0000000000401014<+20>: jle     <func4+36>
0000000000401016<+22>: lea     -0x1(%rcx),%edx
0000000000401018<+24>: call    <func4>
000000000040101a<+26>: add     %eax,%eax
000000000040101c<+28>: jnp     <func4+57>
000000000040101e<+30>: mov     $0x0,%eax
0000000000401020<+32>: cmp     %edi,%ecx
0000000000401022<+34>: jge     <func4+57>
0000000000401024<+36>: lea     0x1(%rcx),%esi
0000000000401026<+38>: call    <func4>
0000000000401028<+40>: lea     0x1(%rax,%rax,1),%eax
000000000040102a<+42>: add     $0x8,%rsp
000000000040102c<+44>: ret
***
End of assembler dump.
(gdb)
```

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:

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
0000000000401000<+0>:  push   %rbx
0000000000401002<+2>:  sub    $0x20,%rsp
0000000000401004<+4>:  mov     %rdi,%rbx
0000000000401006<+6>:  mov     %fs:0x28,%rax
0000000000401008<+8>:  mov     %rax,0x10(%rsp)
000000000040100a<+10>: xor     %eax,%eax
000000000040100c<+12>: call    0x401000 <string_length>
000000000040100e<+14>: cmp     $0x6,%eax
0000000000401010<+16>: je      0x401000 <phase_5+112>
0000000000401012<+18>: call    0x401000 <explode_bomb>
0000000000401014<+20>: jnp     0x401000 <phase_5+112>
0000000000401016<+22>: movzbl  (%rbx,%rax,1),%ecx
0000000000401018<+24>: mov     %cl,(%rsp)
000000000040101a<+26>: mov     (%rsp),%rdx
000000000040101c<+28>: and     $0xf,%edx
000000000040101e<+30>: movzbl  0x4024b0(%rdx),%edx
0000000000401020<+32>: mov     %dl,0x10(%rsp,%rax,1)
0000000000401022<+34>: add     $0x1,%rax
0000000000401024<+36>: cmp     $0x6,%rax
0000000000401026<+38>: jne     0x401000 <phase_5+41>
0000000000401028<+40>: movb    $0x0,0x10(%rsp)
000000000040102a<+42>: mov     $0x40245e,%esi
000000000040102c<+44>: mov     $0x40245e,%esi
--Type <RET> for more, q to quit, c to continue without paging--
000000000040102e<+46>: lea     0x10(%rsp),%rdi
0000000000401030<+48>: call    0x401000 <strings_not_equal>
0000000000401032<+50>: test    %eax,%eax
0000000000401034<+52>: je      0x401000 <phase_5+119>
0000000000401036<+54>: call    0x401000 <explode_bomb>
0000000000401038<+56>: nopl    0x0(%rax,%rax,1)
000000000040103a<+58>: jnp     0x401000 <phase_5+119>
000000000040103c<+60>: mov     $0x0,%eax
000000000040103e<+62>: jnp     0x401000 <phase_5+41>
0000000000401040<+64>: mov     0x10(%rsp),%rax
0000000000401042<+66>: xor     %fs:0x28,%rax
0000000000401044<+68>: je      0x401000 <phase_5+140>
0000000000401046<+70>: call    0x401000 <__stack_chk_fail@plt>
0000000000401048<+72>: add     $0x20,%rsp
000000000040104a<+74>: pop     %rbx
000000000040104c<+76>: ret
***
End of assembler dump.
(gdb)
```

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:

```

0x0010012 <phase_5> push    rbp
0x0010013 <phase_5+1> sub     $0x20, %rsp
0x0010017 <phase_5+5> mov     rdi, %rbx
0x001001a <phase_5+8> mov     %fs:0x28, %rax
0x001001f <phase_5+17> mov     %rax, 0x18(%rsp)
0x0010022 <phase_5+22> xor     %eax, %eax
0x0010024 <phase_5+24> call    0x401110 <string_length>
0x0010029 <phase_5+29> cmp     $0x6, %eax
0x0010032 <phase_5+32> je      0x00100d2 <phase_5+112>
0x0010034 <phase_5+34> call    0x401414 <explode_bomb>
0x0010039 <phase_5+39> jmp     0x0010012 <phase_5+12>
> 0x0010040 <phase_5+41> movzbl  (%rbx,%rax,1), %ecx
0x001004f <phase_5+45> mov     %cl, (%rsp)
0x0010052 <phase_5+48> mov     (%rsp), %rdx
0x0010054 <phase_5+52> and     $0xf, %edx
0x0010059 <phase_5+55> movzbl  0x4024b0(%rdx), %edx
0x0010064 <phase_5+62> mov     %dl, 0x10(%rsp,%rax,1)
0x0010064 <phase_5+66> add     $0x1, %rax
0x0010068 <phase_5+70> cmp     $0x6, %rax
0x001006c <phase_5+74> jne     0x0010040 <phase_5+41>
0x001006e <phase_5+76> movb    $0x0, 0x16(%rsp)
0x0010073 <phase_5+81> mov     $0x40245e, %esi
0x0010078 <phase_5+86> lea     0x10(%rsp), %rdi
0x001007d <phase_5+91> call    0x401130 <strings_not_equal>
0x0010082 <phase_5+96> test    %eax, %eax
0x0010084 <phase_5+98> je      0x0010040 <phase_5+41>
0x0010086 <phase_5+100>

```

```

Multi-Thread Thread 0x7fffffa7 in: phase_5
Run till exit from #0 0x00000000001110 in string_length ()
0x00000000001110 in phase_5 ()
(gdb) stepi
0x00000000001110 in phase_5 ()
0x00000000001110 in phase_5 ()
0x00000000001110 in phase_5 ()
0x00000000001110 in phase_5 ()
(gdb) p/x %rbx
A syntax error in expression, near '%rbx'.
(gdb) p/x $rbx
$1 = 0x0038c0
(gdb) x/s 0x0038c0
0x0038c0 <input_string+320>: "abcdef"
(gdb)

```

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:

```

Multi-Thread Thread 0x7fffffa7 in: phase_5
(gdb) x/s 0x4024b0
0x4024b0 <array.3449>: "madulernsfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
(gdb)

```

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:

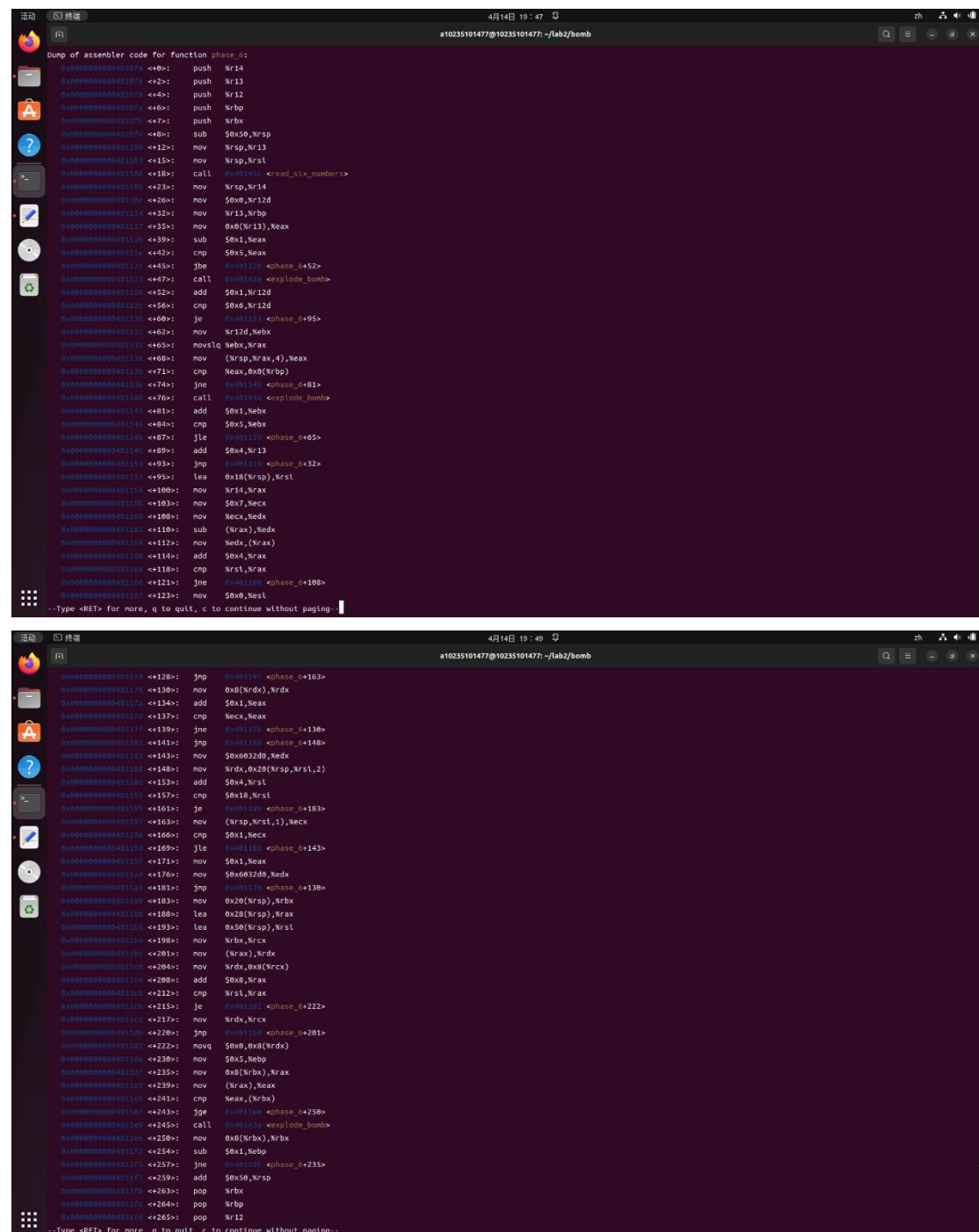
```

(gdb) x/s 0x40245e
0x40245e <array.3449>: "flyers"
(gdb)

```

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，此时已与炸弹无关,故此处加括号)故拆弹的密码需满足：按顺序各字符 ASCII 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:



```
Dump of assembler code for function phase_01:
00401074 <+0>: push    r14
00401076 <+2>: push    r13
00401078 <+4>: push    r12
0040107a <+6>: push    rbp
0040107c <+8>: push    rdx
0040107e <+10>: sub     $0x58, rsp
00401080 <+12>: mov     rsp, r13
00401082 <+14>: mov     rbp, r13
00401084 <+16>: call    00401085 <read_six_numbers>
00401086 <+18>: mov     rsp, r14
00401088 <+20>: mov     $0x0, r12d
0040108a <+22>: mov     r13, rbp
0040108c <+24>: mov     0x0(r13), rax
0040108e <+26>: sub     $0x1, rax
00401090 <+28>: cmp     $0x5, rax
00401092 <+30>: jbe     00401128 <phase_0+52>
00401094 <+32>: call    00401095 <explode_bomb>
00401096 <+34>: add     $0x1, r12d
00401098 <+36>: cmp     $0x6, r12d
0040109a <+38>: je      00401128 <phase_0+95>
0040109c <+40>: mov     r12d, rdx
0040109e <+42>: movslq  rdx, rax
004010a0 <+44>: mov     (rsp, rax, 4), rax
004010a2 <+46>: cmp     rax, 0x0(rbp)
004010a4 <+48>: jne     00401128 <phase_0+81>
004010a6 <+50>: call    00401095 <explode_bomb>
004010a8 <+52>: add     $0x1, rdx
004010aa <+54>: cmp     $0x5, rdx
004010ac <+56>: jle     00401128 <phase_0+65>
004010ae <+58>: add     $0x4, r13
004010b0 <+60>: jmp     00401114 <phase_0+32>
004010b2 <+62>: lea     0x18(rsp), r14
004010b4 <+64>: mov     r14, rax
004010b6 <+66>: mov     $0x7, rdx
004010b8 <+68>: mov     rdx, rdx
004010ba <+70>: sub     (rax), rdx
004010bc <+72>: mov     rdx, (rax)
004010be <+74>: add     $0x4, rax
004010c0 <+76>: cmp     r14, rax
004010c2 <+78>: jne     00401128 <phase_0+108>
004010c4 <+80>: mov     $0x0, rax
004010c6 <+82>: jmp     00401128 <phase_0+103>

--Type <RET> for more, q to quit, c to continue without paging.

00401174 <+128>: jmp     00401197 <phase_0+163>
00401176 <+130>: mov     0x0(rdx), rdx
00401178 <+132>: add     $0x1, rax
0040117a <+134>: cmp     rax, rax
0040117c <+136>: jne     00401176 <phase_0+138>
0040117e <+138>: jmp     00401180 <phase_0+148>
00401180 <+140>: mov     $0x68320, rdx
00401182 <+142>: mov     rdx, 0x20(rsp, r14, 2)
00401184 <+144>: add     $0x4, r14
00401186 <+146>: cmp     $0x18, r14
00401188 <+148>: je      00401180 <phase_0+183>
0040118a <+150>: mov     (rsp, r14, 1), rax
0040118c <+152>: cmp     $0x1, rax
0040118e <+154>: jle     00401180 <phase_0+143>
00401190 <+156>: mov     $0x1, rax
00401192 <+158>: mov     $0x68320, rdx
00401194 <+160>: jmp     00401176 <phase_0+138>
00401196 <+162>: mov     0x20(rsp), rdx
00401198 <+164>: lea     0x20(rsp), rax
0040119a <+166>: lea     0x50(rsp), r14
0040119c <+168>: mov     r14, rax
0040119e <+170>: mov     (rax), rdx
004011a0 <+172>: mov     rdx, 0x0(rax)
004011a2 <+174>: add     $0x8, rax
004011a4 <+176>: cmp     r14, rax
004011a6 <+178>: je      00401180 <phase_0+222>
004011a8 <+180>: mov     rdx, rax
004011aa <+182>: jmp     00401180 <phase_0+201>
004011ac <+184>: movq    $0x0, 0x0(rdx)
004011ae <+186>: mov     $0x5, rbp
004011b0 <+188>: mov     0x0(rbp), rax
004011b2 <+190>: mov     (rax), rax
004011b4 <+192>: cmp     rax, (rbp)
004011b6 <+194>: jge     00401180 <phase_0+250>
004011b8 <+196>: call    00401095 <explode_bomb>
004011ba <+198>: mov     0x0(rbp), rbp
004011bc <+200>: sub     $0x1, rbp
004011be <+202>: jne     00401180 <phase_0+235>
004011c0 <+204>: add     $0x58, rsp
004011c2 <+206>: pop     rbp
004011c4 <+208>: pop     r12
004011c6 <+210>: pop     r12

--Type <RET> for more, q to quit, c to continue without paging--
```

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；

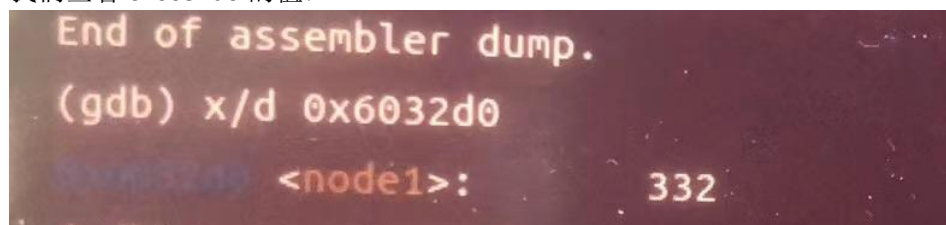
第四段：123~181 行；第五段：剩下所有。

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；

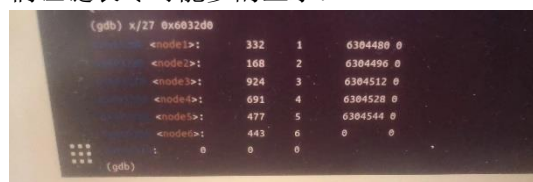
第二段：`%r14` 指向数组的第一个值，接着将 0 赋给 `%r12`，再将 `%r13` 的值赋给 `%rbp`，接着第 35 行，`%r13` 指向栈顶(第 12 行)，将 `%r13` 指向的值赋给 `eax`，若 `%eax-1` 小于等于 5，则，`%r12++`，此时 `%r12` 为 1，小于 6，继续向下读，接下来三句话将数组的第二个元素赋给 `%eax`，将该元素与 `0x0(%rbp)`，第一个元素作比较，一样的话就引爆炸弹，接着 `%ebx++`，重复 65~87 行 5 次，此时可以推出，第一个数与后面五个数都不一样。走出这个循环，`%r13+4`，回到 32 行，可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。

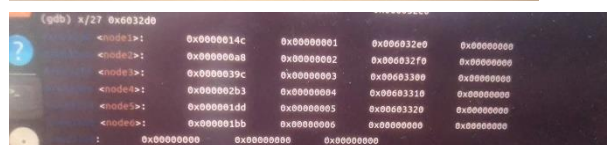
第四段：这一段也是对数组进行处理，只不过处理是有条件的，对于此时数组中的每个数，若该数小于等于一，则将位于内存 `0x6032d0` 的数据传给一个新数组中的对应位置(`0x20(%rsp)` 作为新数组的开头，每个元素为八个字节)，如果不是，更新 `rdx`，且更新的 `rdx` 必须满足这个条件：更新次数为这个数组对应数的值。接着让 `rdx` 更新后指向的值存入数组中，接下来，我们查看 `0x6032d0` 的值：



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)



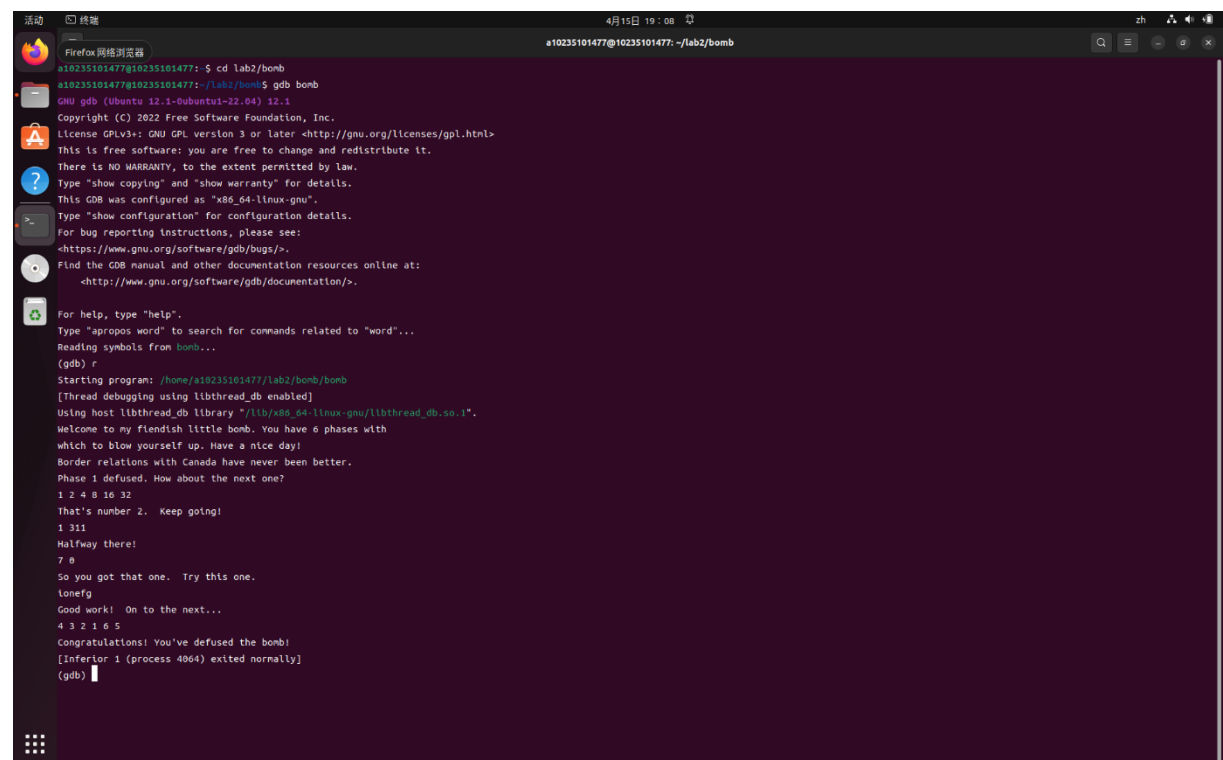
(16 进制)

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 $A_i, 0 < i < 7$)，`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： $7-A_1=3$ ， $7-A_2=4$ ， $7-A_3=5$ ， $7-A_4=6$ ， $7-A_5=1$ ， $7-A_6=2$ ，故密码为：4，3，2，1，6，5。

完整答题截图：



```
活动 终端
a10235101477@10235101477: ~/lab2/bomb
a10235101477@10235101477: $ cd lab2/bomb
a10235101477@10235101477: ~/lab2/bomb$ gdb bomb
GNU gdb (Ubuntu 12.1-0ubuntu1-22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) r
Starting program: /home/a10235101477/lab2/bomb/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
1 3 11
Halfway there!
7 8
So you got that one. Try this one.
Lonefg
Good work! On to the next...
4 3 2 1 6 5
Congratulations! You've defused the bomb!
[Inferior 1 (process 4864) exited normally]
(gdb)
```