

Lab2

Phase 1

```
0x400ee0 <phase_1>      sub    $0x8,%rsp
0x400ee4 <phase_1+4>     mov    $0x402400,%esi
0x400ee9 <phase_1+9>     call   0x401338 <strings_not_equal>
0x400eee <phase_1+14>    test   %eax,%eax
0x400ef0 <phase_1+16>    je     0x400ef7 <phase_1+23>
0x400ef2 <phase_1+18>    call   0x40143a <explode_bomb>
0x400ef7 <phase_1+23>    add    $0x8,%rsp
0x400efb <phase_1+27>    ret
```

`explode_bomb` 在 `0x400ef2` 行，需要通过 `0x400ef0` 行的 `je` 控制语句来跳过。

`je` 语句使用上一行的 `test` 语句，`test` 返回操作数本身，故需要让 `%eax` 为 `0`，`%eax` 是上一个函数 `string_not_equal` 的返回值。猜测该函数是检测两个字符串是否不相等，本题需要让它们相等，其中一个参数通过标准输入，另一个则是由上一行的 `mov $0x402400, %rsi` 传入。

需要输入的字符串与 `0x402400` 处的串相等，使用 `x/s 0x402400` 查看后者字符串，得到结果。

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

Phase1: Border relations with Canada have never been better.

Phase 2

```
0x400efc <phase_2>      push   %rbp
0x400efd <phase_2+1>     push   %rbx
0x400efe <phase_2+2>     sub    $0x28,%rsp
0x400f02 <phase_2+6>     mov    %rsp,%rsi
0x400f05 <phase_2+9>     call   0x40145c <read_six_numbers>
0x400f0a <phase_2+14>    cmpl   $0x1,(%rsp)
0x400f0e <phase_2+18>     je     0x400f30 <phase_2+52>
0x400f10 <phase_2+20>    call   0x40143a <explode_bomb>
0x400f15 <phase_2+25>    jmp    0x400f30 <phase_2+52>
0x400f17 <phase_2+27>    mov    -0x4(%rbx),%eax
0x400f1a <phase_2+30>     add    %eax,%eax
0x400f1c <phase_2+32>     cmp    %eax,(%rbx)
0x400f1e <phase_2+34>     je     0x400f25 <phase_2+41>
0x400f20 <phase_2+36>    call   0x40143a <explode_bomb>
0x400f25 <phase_2+41>    add    $0x4,%rbx
0x400f29 <phase_2+45>     cmp    %rbp,%rbx
0x400f2c <phase_2+48>     jne    0x400f17 <phase_2+27>
0x400f2e <phase_2+50>     jmp    0x400f3c <phase_2+64>
0x400f30 <phase_2+52>     lea    0x4(%rsp),%rbx
0x400f35 <phase_2+57>     lea    0x18(%rsp),%rbp
0x400f3a <phase_2+62>     jmp    0x400f17 <phase_2+27>
0x400f3c <phase_2+64>     add    $0x28,%rsp
0x400f40 <phase_2+68>     pop    %rbx
0x400f41 <phase_2+69>     pop    %rbp
0x400f42 <phase_2+70>     ret
```

在 `0x400f05` 行读入 6 个数字，记作 `a[0]` 至 `a[5]`。检查栈顶 `%rsp` 的值也就是 `a[0]`。若 `a[0] == 1`，`je` 至 `0x400f30` 进行初始化 `%rbx = %rsp + 0x4` `%rbp = %rsp + 0x18`，分别是 `a + 1`

和 `a + 6` 的地址，然后进入 `0x400f17` 与 `0x400f3a` 之间的循环：

1. 将 `*(%rbx - 4)` 赋值给 `%eax` 并翻倍。
2. `%rax` 与 `*(%rbx)` 比较（也就是 `%rbx` 对应数组元素与它的前一项的两倍），若比较结果不相等则引爆炸弹，因此需要比较结果相等。
3. 在每次比较后将 `%rbx += 4`，即循环变量递增。
4. 若 `%rbx` 到达 `%rbp (== %rsp + 0x18)`（即到达数组末尾）时结束循环，否则回到步骤 1。

综上，`phase_2` 需要一个长度为 6 的数组，以 1 开头，且每一项是前一项的两倍

Phase2: 1 2 4 8 16 32

Phase 3

```
0x400f5b <phase_3+24> call 0x400bf0 <__isoc99_sscanf@plt>
0x400f60 <phase_3+29> cmp $0x1,%eax
0x400f63 <phase_3+32> jq 0x400f6a <phase_3+39>
0x400f65 <phase_3+34> call 0x40143a <explode_bomb>
0x400f6a <phase_3+39> cmpl $0x7,0x8(%rsp)
0x400f6f <phase_3+44> ja 0x400fad <phase_3+106>
0x400f71 <phase_3+46> mov 0x8(%rsp),%eax
```

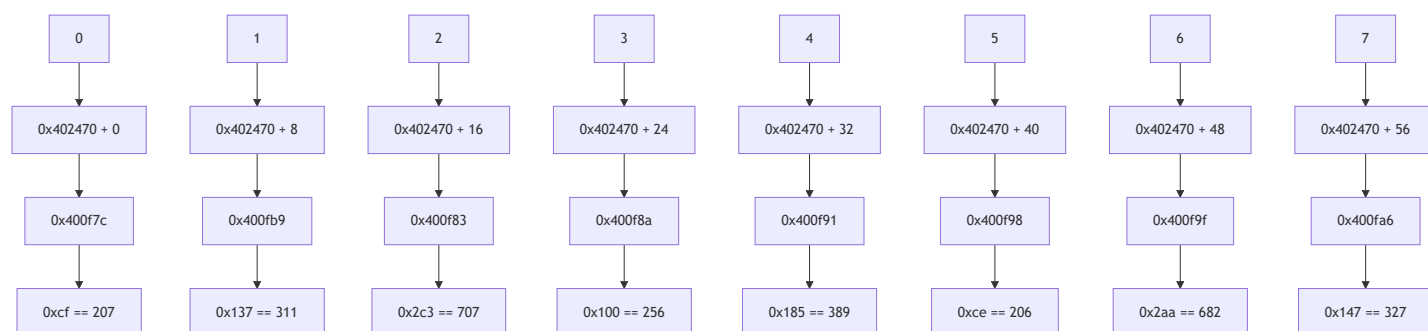
`0x400f5b` 行至 `0x400f71` 行输入至少两个数字，并将第一个数字（经过多次测试得出，这两个数字存放在栈顶下 8 字节与 12 字节处）存入 `%eax`，同时保它是不大于 7 的非负数。

```
0x400f71 <phase_3+46> mov 0x8(%rsp),%eax
0x400f75 <phase_3+50> jmp *0x402470(,%rax,8)
0x400f7c <phase_3+57> mov $0xcf,%eax
0x400f81 <phase_3+62> jmp 0x400fbe <phase_3+123>
0x400f83 <phase_3+64> mov $0x2c3,%eax
0x400f88 <phase_3+69> jmp 0x400fbe <phase_3+123>
0x400f8a <phase_3+71> mov $0x100,%eax
0x400f8f <phase_3+76> jmp 0x400fbe <phase_3+123>
0x400f91 <phase_3+78> mov $0x185,%eax
0x400f96 <phase_3+83> jmp 0x400fbe <phase_3+123>
0x400f98 <phase_3+85> mov $0xce,%eax
0x400f9d <phase_3+90> jmp 0x400fbe <phase_3+123>
0x400f9f <phase_3+92> mov $0x2aa,%eax
0x400fa4 <phase_3+97> jmp 0x400fbe <phase_3+123>
0x400fa6 <phase_3+99> mov $0x147,%eax
0x400fab <phase_3+104> jmp 0x400fbe <phase_3+123>
0x400fad <phase_3+106> call 0x40143a <explode_bomb>
0x400fb2 <phase_3+111> mov $0x0,%eax
0x400fb7 <phase_3+116> jmp 0x400fbe <phase_3+123>
0x400fb9 <phase_3+118> mov $0x137,%eax
0x400fbe <phase_3+123> cmp 0xc(%rsp),%eax
0x400fc2 <phase_3+127> je 0x400fc9 <phase_3+134>
0x400fc4 <phase_3+129> call 0x40143a <explode_bomb>
0x400fc9 <phase_3+134> add $0x18,%rsp
0x400fcd <phase_3+138> ret
```

`0x400f75` 根据输入的第一个数字的值，获取对应地址处（具体为 `0x402470 + 8 * %rax`）存放的另一个值，并间接跳转到该值存储的地址处，可以注意到，`jmp` 间接跳转的目标地址都是其下方某个 `mov` 指令的地址，所有 `mov` 会将 `%eax` 赋为某个值，之后都会到达 `0x400fbe` 处的 `cmp`（包括 `jmp` 至的和恰好顺序执行到的）。

可以总结为 `phase_3` 中生成了一个 `switch-case` 语句的跳转表，不同的 `case` 对应不同的另一个数字，如果输入的两个数字在 `switch-case` 中存在匹配的组合则通关

`0x400fbe` 处如果 `%eax` 与 输入的第二个数字相等则跳过炸弹通关。通过数次的尝试得到的对应关系如下，任意输入其中一对即可



Phase3: 0 207 || 1 311 || 2 707 || 3 256 || 4 389 || 5 206 || 6 682 || 7 327

Phase 4

```

0x40100c <phase_4>    sub    $0x18,%rsp
0x401010 <phase_4+4>   lea     0xc(%rsp),%rcx
0x401015 <phase_4+9>   lea     0x8(%rsp),%rdx
0x40101a <phase_4+14>  mov     $0x4025cf,%esi
0x40101f <phase_4+19>   mov     $0x0,%eax
0x401024 <phase_4+24>  call    0x400bf0 <__isoc99_sscanf@plt>
0x401029 <phase_4+29>   cmp     $0x2,%eax
0x40102c <phase_4+32>   jne     0x401035 <phase_4+41>
0x40102e <phase_4+34>   cmpl    $0xe,0x8(%rsp)
0x401033 <phase_4+39>   jbe     0x40103a <phase_4+46>
0x401035 <phase_4+41>   call    0x40143a <explode_bomb>
0x40103a <phase_4+46>   mov     $0xe,%edx
0x40103f <phase_4+51>   mov     $0x0,%esi
0x401044 <phase_4+56>   mov     0x8(%rsp),%edi
0x401048 <phase_4+60>   call    0x400fce <func4>
0x40104d <phase_4+65>   test    %eax,%eax
0x40104f <phase_4+67>   jne     0x401058 <phase_4+76>
0x401051 <phase_4+69>   cmpl    $0x0,0xc(%rsp)
0x401056 <phase_4+74>   je      0x40105d <phase_4+81>
0x401058 <phase_4+76>   call    0x40143a <explode_bomb>
0x40105d <phase_4+81>   add     $0x18,%rsp
0x401061 <phase_4+85>   ret

```

首先读入恰好两个整数存入 `%rsp + 8` 和 `%rsp + 16`，且保证第一个数是不大于 14 的非负数（负数不能满足无符号比较）。将 `%edx` `%esi` `%edi` 分别初始化为 14 0 `*(%rsp + 8)` 后进入函数 `func4`：

```

0x400fce <func4>      sub     $0x8,%rsp
0x400fd2 <func4+4>      mov     %edx,%eax
0x400fd4 <func4+6>      sub     %esi,%eax
0x400fd6 <func4+8>      mov     %eax,%ecx
0x400fd8 <func4+10>     shr     $0x1f,%ecx
0x400fdb <func4+13>     add     %ecx,%eax
0x400fdd <func4+15>     sar     %eax
0x400fdf <func4+17>     lea     (%rax,%rsi,1),%ecx
0x400fe2 <func4+20>     cmp     %edi,%ecx
0x400fe4 <func4+22>     jle     0x400ff2 <func4+36>
0x400fe6 <func4+24>     lea     -0x1(%rcx),%edx
0x400fe9 <func4+27>     call    0x400fce <func4>
0x400fee <func4+32>     add     %eax,%eax
0x400ff0 <func4+34>     jmp     0x401007 <func4+57>
0x400ff2 <func4+36>     mov     $0x0,%eax
0x400ff7 <func4+41>     cmp     %edi,%ecx
0x400ff9 <func4+43>     jge     0x401007 <func4+57>
0x400ffb <func4+45>     lea     0x1(%rcx),%esi
0x400ffe <func4+48>     call    0x400fce <func4>
0x401003 <func4+53>     lea     0x1(%rax,%rax,1),%eax
0x401007 <func4+57>     add     $0x8,%rsp
0x40100b <func4+61>     ret

```

0x400fd2 至 0x400fdf 进行了如下操作，（使用 c 形式表示，注释内代入数值）：

```
%eax = %edx - %esi          //( == 14 - 0 == 14)
%ecx = %eax                  //( == 14)
%eax = %eax + ((unsigned)%ecx >> 31)  //( == 14 + (14 >> 31) == 14)
%eax = %eax >> 1             //( == 14 >> 1 == 7)
%ecx = %rax + %rsi           //( == 7 + 0 == 7)
```

之后比较 %edi（输入的的第一个数字）与 %ecx (== 7)，如果 7 <= %edi 则跳转至 0x400ff2。

在 0x400ff2 令 %eax = 0 后比较 %ecx (== 7) 与 %edi 如果 7 >= %edi 则跳转至 0x401007。

至此函数已经跳过了其中全部的递归调用直接返回，同时 %eax 恰为所需要的 0，因此只需要使 %edi >= 7 且 %edi <= 7 即可直接通过 func4，而 %edi 即是输入的的第一个数字，因此输入的的第一个数字只需为 7 即可。0x401051 要求第二个数一定为 0。

Phase4: 7 0

Phase 5

```
0x401078 <phase_5+22> xor    %eax,%eax
0x40107a <phase_5+24> call   0x40131b <string_length>
0x40107f <phase_5+29> cmp     $0x6,%eax
0x401082 <phase_5+32> je      0x4010d2 <phase_5+112>
0x401084 <phase_5+34> call   0x40143a <explode_bomb>
0x401089 <phase_5+39> jmp     0x4010d2 <phase_5+112>
0x40108b <phase_5+41> movzbl (%rbx,%rax,1),%ecx
0x40108f <phase_5+45> mov     %cl,(%rsp)
0x401092 <phase_5+48> mov     (%rsp),%rdx
0x401096 <phase_5+52> and     $0xf,%edx
0x401099 <phase_5+55> movzbl 0x4024b0(%rdx),%edx
0x4010a0 <phase_5+62> mov     %dl,0x10(%rsp,%rax,1)
0x4010a4 <phase_5+66> add     $0x1,%rax
0x4010a8 <phase_5+70> cmp     $0x6,%rax
0x4010ac <phase_5+74> jne     0x40108b <phase_5+41>
0x4010ae <phase_5+76> movb    $0x0,0x16(%rsp)
0x4010b3 <phase_5+81> mov     $0x40245e,%esi
0x4010b8 <phase_5+86> lea     0x10(%rsp),%rdi
0x4010bd <phase_5+91> call   0x401338 <strings_not_equal>
0x4010c2 <phase_5+96> test    %eax,%eax
0x4010c4 <phase_5+98> je      0x4010d9 <phase_5+119>
0x4010c6 <phase_5+100> call   0x40143a <explode_bomb>
0x4010cb <phase_5+105> nopl    0x0(%rax,%rax,1)
0x4010d0 <phase_5+110> jmp     0x4010d9 <phase_5+119>
```

查阅了解从 0x40106a 至 0x401073 部分是保证程序鲁棒性的安全措施，不影响解题，故忽略。

0x401078 检测输入的字符串长度（从 string_length 函数的名称可猜测得出），需要为 6。

在 0x401089 处跳转至 0x4010d2 将 %eax 置零再回到 0x40108b 进入 0x40108b 至 0x4010ac 之间的循环，其中 %rax 作为循环变量从 0 至 5，当到达 6 时结束循环

1. 将 (%rbx + %rax) 地址处的变量存入 %ecx，将 %cl（%ecx 的低 8 位）存入栈顶和 %rdx 中，后者只保留低 4 位。
2. 取 (0x4024b0 + %rdx) 地址处的值替换掉 %rdx，再将它的低 8 位存入 (%rsp + %rax + 16)（栈顶下 %rax + 16 字节处）。

4. 其中 `%rbx` 是输入的字符串的地址, `%rax` 是循环变量, `%ecx` `%cl` 和 `%rsp` 作为中间变量。

0x4010ae 在保存的新串后添加结束符 '\0'。

0x4010b3 和 0x4010b8 分别将 0x40245e 的某个串与输入串转换后的新串比较，若相同则 je 至 0x4010e9 结束函数。

要让输入串转变为 "flyers"

由于转换式有对 16 取模操作，转换只需要前 16 个字符 "maduiersnfotvbyl"。其余字符串是在程序执行过程中输入 Ctrl + C 时的提示语，与本题无关。

The diagram illustrates the mapping of hex values to ASCII characters for the word "try". It consists of six columns, each representing a character in the word. Each column has a top box with a hex value and a bottom box with the corresponding ASCII character. Arrows point from the top boxes to the bottom boxes.

Hex Value	ASCII Character
x % 0x10 == 0x9	'r'
x % 0x10 == 0xF	'y'
x % 0x10 == 0xE	' '
x % 0x10 == 0x5	't'
x % 0x10 == 0x6	'r'
x % 0x10 == 0x7	'y'

Phase5: ionefg

Phase 6

Phase 6 Part 1


```

0x401106 <phase_6+18> call 0x40145c <read_six_numbers>
0x40110b <phase_6+23> mov %rsp,%r14
0x40110e <phase_6+26> mov $0x0,%r12d
0x401114 <phase_6+32> mov %r13,%rbp
0x401117 <phase_6+35> mov 0x0(%r13),%eax
0x40111b <phase_6+39> sub $0x1,%eax
0x40111e <phase_6+42> cmp $0x5,%eax
0x401121 <phase_6+45> jbe 0x401128 <phase_6+52>
0x401123 <phase_6+47> call 0x40143a <explode_bomb>
0x401128 <phase_6+52> add $0x1,%r12d
0x40112c <phase_6+56> cmp $0x6,%r12d
0x401130 <phase_6+60> je 0x401153 <phase_6+95>
0x401132 <phase_6+62> mov %r12d,%ebx
0x401135 <phase_6+65> movslq %ebx,%rax
0x401138 <phase_6+68> mov (%rsp,%rax,4),%eax
0x40113b <phase_6+71> cmp %eax,0x0(%rbp)
0x40113e <phase_6+74> jne 0x401145 <phase_6+81>
0x401140 <phase_6+76> call 0x40143a <explode_bomb>
0x401145 <phase_6+81> add $0x1,%ebx
0x401148 <phase_6+84> cmp $0x5,%ebx
0x40114b <phase_6+87> jle 0x401135 <phase_6+65>
0x40114d <phase_6+89> add $0x4,%r13
0x401151 <phase_6+93> jmp 0x401114 <phase_6+32>
0x401153 <phase_6+95> lea 0x18(%rsp),%rsi

```

以上部分首先读入 6 个数字，下文记作 `a[0] ~ a[5]`，做 `%r14 = %rsp` 和 `%r12d = 0` 的初始化后，进入 `0x401114` 至 `0x401151` 的双层循环，其中外层循环的循环变量为 `%r12d`，记作 `i`，从 1 至 5，当到达 6 时结束循环。

内层循环：

- 在 `0x401132` 将 `%rbx` 初始化为外层循环变量 `i` 的值，然后作为内层的循环变量 `j`，进入 `0x401135` 至 `0x40114b` 的内层循环，范围为 `i ~ 5`，到达 6 结束循环，循环体如下：
 - `%rax = j`。
 - `%eax = *(%rsp + %rax * 4)`，其中 `%rax == j`。即计算栈顶下 `j * 4` 字节处的地址中的值，也就是输入的 `a[j]`，存入 `%eax`
 - 比较 `a[j]` 与 `*(%rbp)`，若相同则引爆炸弹，它们不能相同。
 - 循环变量自增 1，若自增后到达 6，结束内层循环，否则回到步骤 1。
- 将 `%r13 += 4`（即指向下一个输入的 `int` 变量，`%r13` 在程序开始时被初始化为 `%rsp`，截图中没展示）后进入外层的下一次循环。
- 由于内层循环中 `%rbp` 保持不变，`j` 从 `i` 至 5，因此会将第 `i` 至第 5 个数字分别与 `*(%rbp)` 比较，保证它们不同。

外层循环，将 `i` 初始化为 0 后执行如下操作：

- 将 `%r13` 存入 `%rbp`，以及它存储的地址处的数字存入 `%eax`。
- 该数字减 1 后与 5 无符号比较，若数字大于 5 则引爆炸弹，因此需要减 1 后的数字无符号比较不大于 5，也即不减少的数字（输入的原数字）在 1 到 6 之间。
 - 若原数小于等于 0，减 1 后的数字的无符号解释显然会大于 5 不符合要求，所以减 1 后的无符号比较可以保证输入为正。
 - 在该步骤完成后 `%eax` 不再作为源操作数，因此它不会影响其它操作，所以在这里仅仅用于检查输入的范围。

3. 将 `i` 自增 1，若到达 6 则结束外层循环，否则将自增后的 `i` 存入 `%rbx`，这是内层循环变量的初始化，进入内层循环（内层循环之前即自增，因此说 `i` 从 1 开始）。

注意到以下事实：

1. 第一次进入内层循环时，`%rbp` 从栈顶也即从 0 开始，而 `%r12d` 从 1 开始，内层循环中的比较是栈顶数字与其下方的每个数字（当然，不超过全部的 6 个数字）比较，不包括自己。保证它们不相同。
2. 其后每一次进入内层循环前，`%r13` 与 `i` 同步增长，其中前者在前一次内层循环结束后自增，后者在下次开始前自增。
3. 在每次外层循环时，都有 `%rbp = %r13`，而 `%rbp` 不在其它地方改变，因此等价于 `%rbp` 与 `i` 的同步增长。
4. 实现了这样的操作，`%rbp` 从栈顶开始，`i` 从它下面一个开始，将 `a[0]` 个数字与第 `a[1...5]` 分别比较，保证互不相同；二者同步自增，将 `a[1]` 数字与 `a[2...5]` 分别比较。以此类推

综上 `0x401106` 至 `0x401151` 实现了输入 6 个数字，范围在 1 至 6 之间，互不相同。

Phase 6 Part 2

```
0x401153 <phase_6+95> lea    0x18(%rsp),%rsi
0x401158 <phase_6+100> mov     %r14,%rax
0x40115b <phase_6+103> mov     $0x7,%ecx
0x401160 <phase_6+108> mov     %ecx,%edx
0x401162 <phase_6+110> sub     (%rax),%edx
0x401164 <phase_6+112> mov     %edx,(%rax)
0x401166 <phase_6+114> add     $0x4,%rax
0x40116a <phase_6+118> cmp     %rsi,%rax
0x40116d <phase_6+121> jne     0x401160 <phase_6+108>
```

将栈顶下 24 字节的地址存入 `%rsi`，并将 `%rax` 初始化为 `%r14` 也即 `%rsp`（`0x40110b` 处将 `%r14` 初始化为 `%rsp` 且未再改变），也就是输入数组首地址 `a`，以及将 `%ecx` 初始化为 7。之后进入 `0x401160` 至 `40116d` 之间的循环，`%rax` 作为循环变量：

执行 `%edx = %ecx (== 7)`，将 7 减去 `a[i]` 并存回到 `a[i]`，`%rax += 4` 指向下一个元素，直至 `%rax` 指向数组末尾。

这一部分实现了将输入的 6 个数字原地对 7 取补的操作，如将 1 转换为 $7 - 1 == 6$ ，4 转换为 $7 - 4 == 3$ ，6 转换为 $7 - 6 == 1$ 等。

Phase 6 Part 3

```

0x40116f <phase_6+123> mov     $0x0,%esi
0x401174 <phase_6+128> jmp     0x401197 <phase_6+163>
0x401176 <phase_6+130> mov     0x8(%rdx),%rdx
0x40117a <phase_6+134> add     $0x1,%eax
0x40117d <phase_6+137> cmp     %ecx,%eax
0x40117f <phase_6+139> jne     0x401176 <phase_6+130>
0x401181 <phase_6+141> jmp     0x401188 <phase_6+148>
0x401183 <phase_6+143> mov     $0x6032d0,%edx
0x401188 <phase_6+148> mov     %rdx,0x20(%rsp,%rsi,2)
0x40118d <phase_6+153> add     $0x4,%rsi
0x401191 <phase_6+157> cmp     $0x18,%rsi
0x401195 <phase_6+161> je      0x4011ab <phase_6+183>
0x401197 <phase_6+163> mov     (%rsp,%rsi,1),%ecx
0x40119a <phase_6+166> cmp     $0x1,%ecx
0x40119d <phase_6+169> jle     0x401183 <phase_6+143>
0x40119f <phase_6+171> mov     $0x1,%eax
0x4011a4 <phase_6+176> mov     $0x6032d0,%edx
0x4011a9 <phase_6+181> jmp     0x401176 <phase_6+130>
0x4011ab <phase_6+183> mov     0x20(%rsp),%rbx

```

进入后首先初始化 `%esi = 0` 后跳转到 `0x401197`。

在此处将栈顶下 `%rsi` 字节处的数字存入 `%rcx`，注意到这一部分 `%rsi` 作为目标操作数只有 `0x40118d` `0x401191` 两行，根据它们的内容可以得出 `%rsi` 表示数组索引，由于变化量是 `4`，下文以 `i` 代指 `%rsi / 4` 表示数组索引。于是这一行的指令可以表示为 `%ecx = a[i]`

之后检查 `%ecx`，若它小于等于 `1`，`jle` 至 `0x401183` 将 `%edx` 赋值为 `0x6032d0`，并将这个指针存入 `0x20 + %rsp + 2 * i` 的地址处。

注意到有且仅有 `0x401188` 这一行对栈顶下 `0x20` 个字节之后的位置有操作，且形式和上文中的数组类似，不妨认为在这个位置有另一个数组，记为 `b`，它的首地址是 `0x20 + %rsp`，而索引 `%rsi` 以 `4` 为变化量，这里的目标操作数比例因子为 `2`，所以数组 `b` 元素以 `8` 字节为变化量。另外，可以发现它的源操作数 `%rdx` 是地址，所以这是一个指针数组。变址与数组 `a` 相同，所以可以使用相同的索引 `i`。

这时重新从 `0x401197` 分析。将 `a[i]` 存入 `%ecx`：

1. 若它小于等于 `1`，将 `0x6032d0` 存入 `b[i]`。
2. 若大于 `1`，令 `%eax` 初始化为 `1`，`%edx` 初始化为 `0x6032d0`，进入 `0x401176` 至 `0x40117f` 的循环，`%eax` 是循环变量
 - i. 将 `%rdx` 改为它存储的指针 `+ 8` 后的指针指向的地址存储的指针。
 - ii. `%eax ++`，若与 `%ecx` 相等结束循环，否则回到步骤 i。
 - iii. 结束循环后，将此时的 `%rdx` 的指针存入 `b[i]`。

可以得出结论，`0x6032d0` 开始有一个链表，每个结点的 `+ 0` 的位置是数据域，`+ 8` 的位置是表示后继结点的地址域。链表结点结构如下：

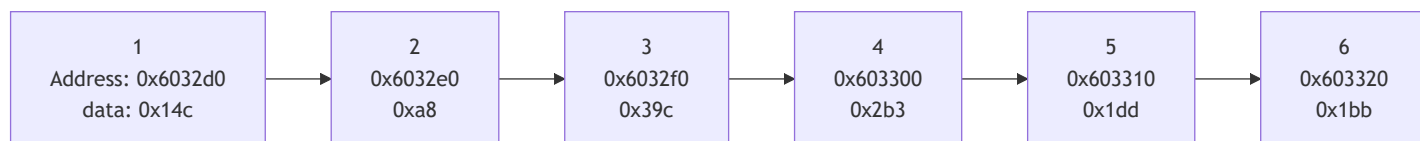
```

typedef struct Node{
    long data;
    Node* next;
};

```


这一部分则是根据 `a` 数组中的值 `a[i]`，取链表中第 `a[i]` 个值，存入 `b[i]` 中。例如，若 `a[i]` 为 1，则取链表中第 1 个值，若 `a[i]` 为 4，则取链表中第 4 个值，将其存入 `b[i]` 中。

使用 `gdb` 调试查看链表内容如下



Phase 6 Part 4

```
0x4011ab <phase_6+183> mov     0x20(%rsp),%rbx
0x4011b0 <phase_6+188> lea     0x28(%rsp),%rax
0x4011b5 <phase_6+193> lea     0x50(%rsp),%rsi
0x4011ba <phase_6+198> mov     %rbx,%rcx
0x4011bd <phase_6+201> mov     (%rax),%rdx
0x4011c0 <phase_6+204> mov     %rdx,0x8(%rcx)
0x4011c4 <phase_6+208> add     $0x8,%rax
0x4011c8 <phase_6+212> cmp     %rsi,%rax
0x4011cb <phase_6+215> je      0x4011d2 <phase_6+222>
0x4011cd <phase_6+217> mov     %rdx,%rcx
0x4011d0 <phase_6+220> jmp     0x4011bd <phase_6+201>
0x4011d2 <phase_6+222> movq    $0x0,0x8(%rdx)
0x4011da <phase_6+230> mov     $0x5,%ebp
0x4011df <phase_6+235> mov     0x8(%rbx),%rax
0x4011e3 <phase_6+239> mov     (%rax),%eax
0x4011e5 <phase_6+241> cmp     %eax,(%rbx)
0x4011e7 <phase_6+243> jge     0x4011ee <phase_6+250>
0x4011e9 <phase_6+245> call    0x40143a <explode_bomb>
0x4011ee <phase_6+250> mov     0x8(%rbx),%rbx
0x4011f2 <phase_6+254> sub     $0x1,%ebp
0x4011f5 <phase_6+257> jne     0x4011df <phase_6+235>
```

`0x4011bd` 至 `0x4011d0` 是一个循环，循环变量 `%rax` 初始化为 `b` 数组第二个元素的地址即 `b + 1`，终止条件为 `%rax` 指向 `b` 数组末尾。同时初始化 `%rbx` 和 `%rcx` 为 `b[0]`（循环中不使用 `%rbx`）：

1. 赋 `%rdx` 为 `*(%rax)` 即 `b[1]`
2. 将 `%rdx` 复制到 `(0x8 + %rcx)` 位置处，
 - i. 此时 `%rcx` 是 `b[0]`，是链表中某个结点的地址。
 - ii. `8 + %rcx` 是这个结点的后继指针，即它指向的下一个结点的地址
 - iii. 将它替换为 `%rdx`，即用 `b[1]` 替换 `b[0]` 结点的后继指针。
3. `%rax += 8`，`%rcx = %rdx`，即让 `%rax` 指向 `b` 数组的下一个元素的所在地址（此时为 `b + 2`），`%rcx` 则是下一个元素本身（`b[1]`），这一步是循环的递增，分别让两个变量在 `b` 数组上递增。
4. 若 `%rax` 指向 `b` 数组末尾，结束循环。

循环结束后，`%rdx` 为 `b[5]`，`0x4011d2` 行保证链表结尾指空

这个循环将链表重排，使链表按照 `b` 数组的顺序排列。

之后，将 `%ebp` 初始化为 5，作为循环变量进入 `0x4011df` 至 `0x4011f5` 之间的循环，此时 `%rbx` 仍为 `b[0]`：

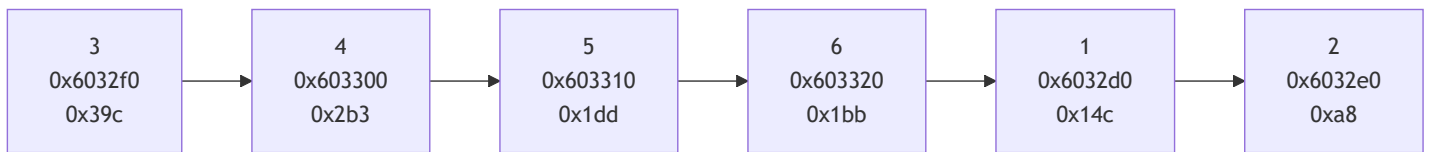
1. `%rbx` 的初始状态是 `b[0]`，是链表中某个结点的地址，`%rbx + 8` 则是这个结点的指针域，`mov` 可以将 `%rbx + 8` 解引用，将内容存入 `%rax`，此时 `%rax` 内就是前一结点的后继结点的地址

2. 再对 `%rax` 自身解引用，因为 `%rax` 内是后继结点的地址，解引用后就是后继结点的 `data`
3. 将其与 `(%rbx)` 比较，`%rbx` 是前一结点的地址，解引用后就是前一结点的 `data`，所以这一步会将相邻两个结点的 `data` 作比较
4. 若 `(%rbx)` 小于 `%eax` 则执行 `explode_bomb`，所以需要前一结点的数据大于或等于后继结点的数据。
5. 将 `*(8 + %rbx)` 存入 `%rbx`，作用类似于步骤 1，将结点更新为后继结点。
6. 重复操作直到 `%ebp` 到达 `0`，此时 `%rbx` 也到达链表结尾，没有后继节点。

综上，这个循环要求链表中的结点的数据按照降序排序

Phase 6 Conclusion

因此只需要获取到链表中的所有结点（并按序编号 `1 ~ 6`）的数据，以降序重排。



使 `a` 数组中为排序后的顺序（的结点编号）即可，即 `3 4 5 6 1 2`。但同时，由于函数会将输入的数组对 `7` 取补，因此需要输入预先对 `7` 取补后的数组。

Phase 6: 4 3 2 1 6 5

Secret Phase

Entrance

在汇编文件中，`phase_6` 函数后有 `func7` 和 `secret_phase`，说明存在隐藏关，首先寻找进入的方法。

直接用编辑器打开汇编文件，使用 `Crtl + F` 查找 `secret_phase`，发现在每次拆弹后的 `phase_defused` 函数中有调用该函数。

```

0x4015c4 <phase_defused>      sub    $0x78,%rsp
0x4015c8 <phase_defused+4>     mov     %fs:0x28,%rax
0x4015d1 <phase_defused+13>    mov     %rax,0x68(%rsp)
0x4015d6 <phase_defused+18>    xor     %eax,%eax
0x4015d8 <phase_defused+20>    cmpl    $0x6,0x202181(%rip)      # 0x603760 <num_input_strings>
0x4015df <phase_defused+27>    jne     0x40163f <phase_defused+123>
0x4015e1 <phase_defused+29>    lea     0x10(%rsp),%r8
0x4015e6 <phase_defused+34>    lea     0xc(%rsp),%rcx
0x4015eb <phase_defused+39>    lea     0x8(%rsp),%rdx
0x4015f0 <phase_defused+44>    mov     $0x402619,%esi
0x4015f5 <phase_defused+49>    mov     $0x603870,%edi
0x4015fa <phase_defused+54>    call    0x400bf0 <__isoc99_sscanf@plt>
0x4015ff <phase_defused+59>    cmp     $0x3,%eax
0x401602 <phase_defused+62>    jne     0x401635 <phase_defused+113>
0x401604 <phase_defused+64>    mov     $0x402622,%esi
0x401609 <phase_defused+69>    lea     0x10(%rsp),%rdi
0x40160e <phase_defused+74>    call    0x401338 <strings_not_equal>
0x401613 <phase_defused+79>    test    %eax,%eax
0x401615 <phase_defused+81>    jne     0x401635 <phase_defused+113>
0x401617 <phase_defused+83>    mov     $0x4024f8,%edi
0x40161c <phase_defused+88>    call    0x400b10 <puts@plt>
0x401621 <phase_defused+93>    mov     $0x402520,%edi
0x401626 <phase_defused+98>    call    0x400b10 <puts@plt>
0x40162b <phase_defused+103>   mov     $0x0,%eax
0x401630 <phase_defused+108>   call    0x401242 <secret_phase>
0x401635 <phase_defused+113>   mov     $0x402558,%edi
0x40163a <phase_defused+118>   call    0x400b10 <puts@plt>
0x40163f <phase_defused+123>   mov     0x68(%rsp),%rax
0x401644 <phase_defused+128>   xor     %fs:0x28,%rax
0x40164d <phase_defused+137>   je      0x401654 <phase_defused+144>
0x40164f <phase_defused+139>   call    0x400b30 <__stack_chk_fail@plt>
0x401654 <phase_defused+144>   add     $0x78,%rsp
0x401658 <phase_defused+148>   ret

```

0x4015d8 行将某寄存器与 6 比较，且有注释标识 num_input_strings，猜测是指输入过的字符串的总个数，如果不等于 6 则直接 jne 到 0x40163f 结束函数，所以必须要总共输入 6 个串之后才可能进入隐藏关。由于每一关对应一个字符串，所以至少要完成前面全部六关。

0x4015f0 行开始，以 0x402619 和 0x603870 作为参数调用标准输入函数 isoc99_sscanf，查看这两个参数。

```

(gdb) x/s 0x402619
0x402619:      "%d %d %s"
(gdb) x/s 0x603870
0x603870 <input_strings+240>:  "7 0"

```

前者为 isoc99_sscanf 函数的格式化参数，表示输入两个整数和一个字符串，后者恰好为 Phase4 的输入，这里会重新读入 Phase4 的输入。

0x4015ff 比较返回值是否为 3，如果不为 3 则 jne 至 0x401635，输出提示语，结束函数。

isoc99_sscanf 的返回值是成功读入的数据个数，因此需要让这个函数读入三个串。前两个分别是 Phase4 本身的解 7 0，第三个输入会作为参数，与 0x402622 处的串进入 strings_not_equal 函数，如果两个串相同，会输出两个提示词，进入 secret_phase。

查看该串

```

(gdb) x/s 0x402622
0x402622:      "DrEvil"

```

所以要在 Phase4 的输入后添加 DrEvil，即

Phase 4: 7 0 DrEvil

Solution

```
0x401242 <secret_phase>      push    %rbx
0x401243 <secret_phase+1>     call    0x40149e <read_line>
0x401248 <secret_phase+6>     mov     $0xa,%edx
0x40124d <secret_phase+11>    mov     $0x0,%esi
0x401252 <secret_phase+16>    mov     %rax,%rdi
0x401255 <secret_phase+19>    call    0x400bd0 <strtol@plt>
0x40125a <secret_phase+24>    mov     %rax,%rbx
0x40125d <secret_phase+27>    lea     -0x1(%rax),%eax
0x401260 <secret_phase+30>    cmp     $0x3e8,%eax
0x401265 <secret_phase+35>    jbe     0x40126c <secret_phase+42>
0x401267 <secret_phase+37>    call    0x40143a <explode_bomb>
0x40126c <secret_phase+42>    mov     %ebx,%esi
0x40126e <secret_phase+44>    mov     $0x6030f0,%edi
0x401273 <secret_phase+49>    call    0x401204 <fun7>
0x401278 <secret_phase+54>    cmp     $0x2,%eax
0x40127b <secret_phase+57>    je      0x401282 <secret_phase+64>
0x40127d <secret_phase+59>    call    0x40143a <explode_bomb>
0x401282 <secret_phase+64>    mov     $0x402438,%edi
0x401287 <secret_phase+69>    call    0x400b10 <puts@plt>
0x40128c <secret_phase+74>    call    0x4015c4 <phase_defused>
0x401291 <secret_phase+79>    pop     %rbx
0x401292 <secret_phase+80>    ret
```

将 `%edx` 初始化为 `10`，`%esi` 为 `0`，并使输入转换成整数，存入 `%rbx`，减 `1` 后存入 `%eax`。将它无符号比较，若不大于 `0x3e8` 则跳过 `explode_bomb`，这里与 `Phase6` 的 `0x40111b` 至 `0x401121` 部分类似，保证输入为正。所以这里保证输入是不大于 `0x3e8` 的正数。

将 `%ebx`（输入的数字）和 `0x6030f0` 作为参数进入 `fun7`，在 `fun7` 中运算后如果返回值 `%eax` 为 `2`，则跳过 `explode_bomb`，完成隐藏关。查看 `0x6030f0` 处的值

```
(gdb) print *(int *)0x6030f0
$2 = 36
```

以下为 `func7`：

```
0x401204 <fun7>               sub     $0x8,%rsp
0x401208 <fun7+4>             test    %rdi,%rdi
0x40120b <fun7+7>             je      0x401238 <fun7+52>
0x40120d <fun7+9>             mov     (%rdi),%edx
0x40120f <fun7+11>            cmp     %esi,%edx
0x401211 <fun7+13>            jle     0x401220 <fun7+28>
0x401213 <fun7+15>            mov     0x8(%rdi),%rdi
0x401217 <fun7+19>            call    0x401204 <fun7>
0x40121c <fun7+24>            add     %eax,%eax
0x40121e <fun7+26>            jmp     0x40123d <fun7+57>
0x401220 <fun7+28>            mov     $0x0,%eax
0x401225 <fun7+33>            cmp     %esi,%edx
0x401227 <fun7+35>            je      0x40123d <fun7+57>
0x401229 <fun7+37>            mov     0x10(%rdi),%rdi
0x40122d <fun7+41>            call    0x401204 <fun7>
0x401232 <fun7+46>            lea     0x1(%rax,%rax,1),%eax
0x401236 <fun7+50>            jmp     0x40123d <fun7+57>
0x401238 <fun7+52>            mov     $0xffffffff,%eax
0x40123d <fun7+57>            add     $0x8,%rsp
0x401241 <fun7+61>            ret
```

如果参数 `%rdi` 为 `NULL` 则 `je` 至 `0x401238` 结束 `fun7`，此时显然不为 `0`。

注意到：

1. 在 `0x40121c` 行会使 `%eax` 加 `%eax`，即 `%eax = %eax * 2`。
2. 在 `0x401220` 行会使 `%eax` 置零，即 `%eax = 0`。

3. 在 `0x401232` 行会计算 `0x1 + %rax + %rax * 1` 存入 `%eax`，即将 `%eax = %eax * 2 + 1`
4. 除置零操作外，其余两个操作都会直接 `jmp` 到函数末尾返回。而置零操作后在 `%edx == %esi` 条件下也会返回。

如果希望以 `2` 作为返回值。可以以 `%eax = 0 --> %eax = %eax * 2 + 1 --> %eax = %eax * 2` 的顺序执行。

由于函数递归调用类似栈操作，所以要先在 `%eax` 翻倍前第一次递归调用，再在翻倍加一前第二次调用，使 `%eax` 置零，再分别返回两次递归，这样即可实现 1. `%eax` 置零 2. 第一次返回后加一 3. 第二次返回后翻倍 4. 完全返回。使 `%eax` 最终为 `2`。

希望到达 `0x401217`，不能满足 `%edx <= %esi`，其中 `%edx` 在 `0x40120d` 行赋值为 `*(%rdi) (== 36)`，所以输入要小于 `36`，之后 `%rdi` 更新为它存储的地址 `+ 8` 的位置的值（这里与 `Phase6` 类似，同样为一个链表）

```
(gdb) print /x *(int *)0x6030f8
$4 = 0x603110
```

这个地址中的值是 `0x8`。

第一次递归调用 `func7`，这次需要到达 `0x401220`。`%edx` 赋值为 `8`，要求 `%esi >= 8`，输入应不小于 `8`。但是如果相等会在 `0x401227` 处跳转到函数末尾，没能使 `%eax` 加一，所以不能相等。

`%rdi` 直接更新为 `+ 16` 的位置存储的地址（`0x603150`，内容为 `0x16`），第二次递归。这次希望直接返回。同样使 `%edx = *(%rdi) // (== 0x16)` 后：

1. 注意到如果 `%edx <= %esi`，可以 `jle` 到 `0x401220` 将 `%eax` 置零。
2. 如果还有 `%edx == %esi` 可以 `je` 到 `0x40123d`，直接返回。
3. 此时的 `%edx` 是 `22`。
4. 同时，如果输入为 `22`，可以同时满足上述 `%rsi <= 36` `%rsi > 8` `%rsi == 22` 的要求。

这时函数就会以预想的位置和顺序递归调用，也会以预想的顺序返回，令 `%eax = 2` 并跳出递归。

secret_phase: 22