宇佐见函钩 编写

时　　间　　2025 年 5 月 10 日

实验 3：拆弹专家

Bomb

1. 实验目的

进一步掌握程序的机器级表示一章的知识。理解程序控制、过程调用的汇编级实现，熟练掌握 汇编语言程序的阅读。

2. 实验内容

程序 bomb 是一个电子炸弹，当该程序运行时，需要按照一定的顺序输入口令，才能阻 止炸弹的引爆。当输入错误的密码时，炸弹将会引爆。此时控制台将会产生如下输出，并结束 程序

```
BOOM!!!
The bomb has blown up.
```

在炸弹程序中，你需要输入多组口令，且每一组口令都正确才能够防止引爆。

目前已知的内容只有炸弹程序的二进制可执行文件 bomb（目标平台为：x86-64）和 bomb 的 main 函数框架代码，见main.c。其他的细节均不会以 c 语言的方式呈现。 你的任务是：利用现有的资源以及相关的工具，猜出炸弹的全部口令，并输入至炸弹程序中，以完成最终的拆弹工作。

3. 实验要求

1）在 Unbuntu18.04LTS 操作系统下，按照实验指导说明书，使用 gdb 和 objdump 等工具，以反向工程方式完成 Bomb 拆弹。

2）需提交：拆弹口令文本文件、电子版实验报告全文。

4. 实验结果

拆弹过程主要使用了 gdb 调试工具对程序进行调试，objdump 工具对 bomb

程序进行反汇编。

Gdb 常用指令：break 设置断点，接函数名或者*地址

b n 在第 n 行设置断点

b 函数名 在某函数起始地址设置断点

x 查看地址中的数据，后面可以接/c(数据为字符串),/d(数据为数字)

disas 查看当前函数的汇编代码

i r 查看寄存器的值

stepi n 运行 n 步(会进入别的函数)

nexti n 运行 n 部(跳过别的函数，只在当前函数)

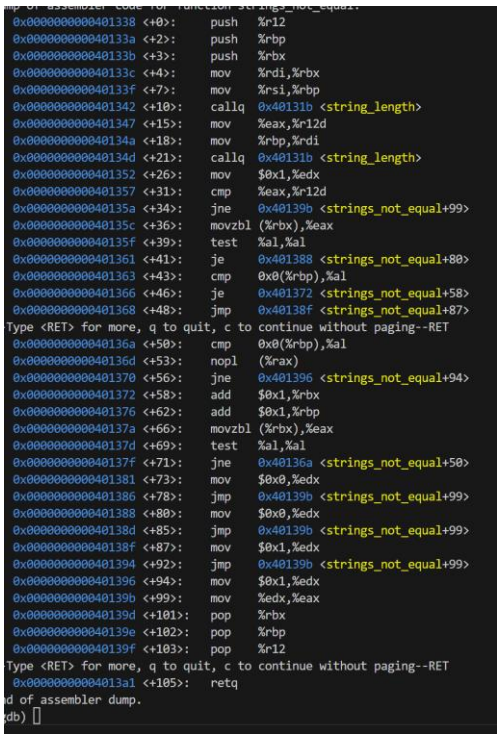
准备工作：使用 objdump -d bomb > bomb.s 将可执行程序汇编代码储存到文件中方便查看。

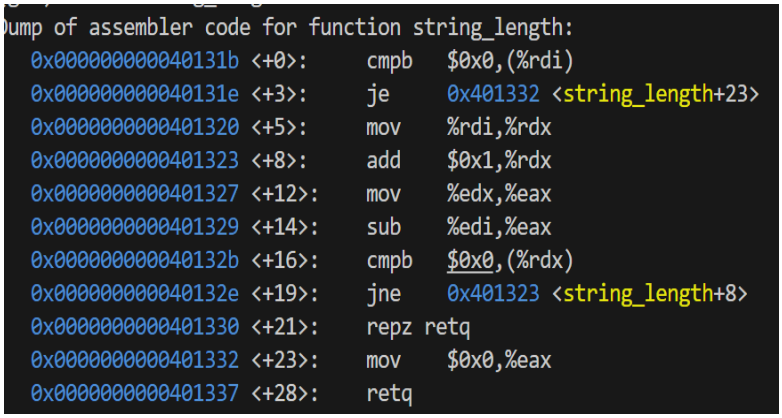
通过浏览 bomb.s 文件可以发现，炸弹的每个关卡都有一个或多个函数组成。我们可以通过使用 disas 指令来分别查看每个函数。

Phase1:disas phase_1 查看函数如下：

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
   0x0000000000400ee0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ee4 <+4>:     mov    $0x402400,%esi
   0x0000000000400ee9 <+9>:     callq  0x401338 <strings_not_equal>
   0x0000000000400eee <+14>:    test   %eax,%eax
   0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
   0x0000000000400ef2 <+18>:    callq  0x40143a <explode_bomb>
   0x0000000000400ef7 <+23>:    add    $0x8,%rsp
   0x0000000000400efb <+27>:    retq
End of assembler dump.
```

我们可以观察到在函数地址+9 处调用了函数<strings_not_equal>





通过追踪 %rdi 可以知道该寄存器存放的是输入字符串的地址

string_length 函数实现：将输入字符串的长度存在寄存器 %rax 中返回

观察 strings_not_equal 函数，我们能够知道它首先得到我们输入字符串长度，然后得到正确答案字符串长度，进行比较，二者不相等则在 %eax 中存 1 返回；若二者长度相等，则逐个比较二者字符串内容，若全部相等则在 %rax 中存 0 返回，若有任意一个不相等则在 %eax 中存 1 返回。现在可以知道 string_not_equal 函数是将地址 0x402400 处的字符串和输入字符串进行比

较。通过 x/s 指令查看 0x402400 处的内容

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

Border relations with Canada have never been better.即为 phase_1 的密码。

Phase_2:

```
Dump of assembler code for function phase_2:
   0x00000000000400efc <+0>:     push   %rbp
   0x00000000000400efd <+1>:     push   %rbx
   0x00000000000400efe <+2>:     sub    $0x28,%rsp
   0x00000000000400f02 <+6>:     mov    %rsp,%rsi
   0x00000000000400f05 <+9>:     callq  0x40145c <read_six_numbers>
   0x00000000000400f0a <+14>:    cmpl   $0x1,(%rsp)
   0x00000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
   0x00000000000400f10 <+20>:    callq  0x40143a <explode_bomb>
   0x00000000000400f15 <+25>:    jmp    0x400f30 <phase_2+52>
   0x00000000000400f17 <+27>:    mov    -0x4(%rbx),%eax
   0x00000000000400f1a <+30>:    add    %eax,%eax
   0x00000000000400f1c <+32>:    cmp    %eax,(%rbx)
   0x00000000000400f1e <+34>:    je     0x400f25 <phase_2+41>
   0x00000000000400f20 <+36>:    callq  0x40143a <explode_bomb>
   0x00000000000400f25 <+41>:    add    $0x4,%rbx
   0x00000000000400f29 <+45>:    cmp    %rbp,%rbx
   0x00000000000400f2c <+48>:    jne    0x400f17 <phase_2+27>
   0x00000000000400f2e <+50>:    jmp    0x400f3c <phase_2+64>
   0x00000000000400f30 <+52>:    lea    0x4(%rsp),%rbx
   0x00000000000400f35 <+57>:    lea    0x18(%rsp),%rbp
   0x00000000000400f3a <+62>:    jmp    0x400f17 <phase_2+27>
   0x00000000000400f3c <+64>:    add    $0x28,%rsp
   0x00000000000400f40 <+68>:    pop    %rbx
   0x00000000000400f41 <+69>:    pop    %rbp
   0x00000000000400f42 <+70>:    retq
```

首先可以看出该函数调用了函数 <read_six_numbers> 可以猜测该函数是要读取六个数字，所以我们可以得知 phase_2 的答案是六个数字。在 phase_2 设置断点，任意输入六个数字后通过 stepi 逐行查看代码。跳转过程中跳转到了 read_six_numbers 中。

```
B+>0x40145c <read_six_numbers>      sub    $0x18,%rsp
   0x401460 <read_six_numbers+4>    mov    %rsi,%rdx
   0x401463 <read_six_numbers+7>    lea    0x4(%rsi),%rcx
   0x401467 <read_six_numbers+11>   lea    0x14(%rsi),%rax
   0x40146b <read_six_numbers+15>   mov    %rax,0x8(%rsp)
   0x401470 <read_six_numbers+20>   lea    0x10(%rsi),%rax
   0x401474 <read_six_numbers+24>   mov    %rax,(%rsp)
   0x401478 <read_six_numbers+28>   lea    0xc(%rsi),%r9
   0x40147c <read_six_numbers+32>   lea    0x8(%rsi),%r8
   0x401480 <read_six_numbers+36>   mov    $0x4025c3,%esi
   0x401485 <read_six_numbers+41>   mov    $0x0,%eax
   0x40148a <read_six_numbers+46>   callq  0x400bf0 <__isoc99_sscanf@plt>
   0x40148f <read_six_numbers+51>   cmp    $0x5,%eax
   0x401492 <read_six_numbers+54>   jg     0x401499 <read_six_numbers+61>
   0x401494 <read_six_numbers+56>   callq  0x40143a <explode_bomb>
   0x401499 <read_six_numbers+61>   add    $0x18,%rsp
   0x40149d <read_six_numbers+65>   retq
```

首先栈顶指针寄存器 %rsp 减去 0x18 ，在内存中开辟了一块 24 字节的栈空间.在第 2 步中的图中我们能看到，调用 read_six_numbers 前先将栈顶指针减去了 0x28 ，然后把栈顶指针值赋给 %rsi.

在 0x401480 处遇到了一个地址 0x4025c3,使用 x/s 查看地址内容为

```
0x4025c3:       "%d %d %d %d %d %d"
```

提示我们要输入六个整形数字，中间用空格隔开。

接下来在 phase_2 函数的 0x400f0a 处，比较输入的第一个数字（储存在%rsp）是否等于“0x1”,.如果是，就跳转到函数地址+52 行处。从 0x400f17 到 0x400f2c 构成了一个循环体，通过循环来判断字符串依次输入的内容。栈顶处

为整型数 1，栈顶指针+4 (整型数占用 4 个字节)指向的下一个整数应该为栈顶的 2 倍，即 2，下一个整数(栈顶指针+8)又是栈顶指针+4 所对应的整数的两倍为 4。我们输入的六个数是逆序入栈，第一个数最后入栈，为栈顶。所以我们输入的字符串为 1 2 4 8 16 32。

Phase3:

```
   0x0000000000400f43 <+0>:     sub    $0x18,%rsp
   0x0000000000400f47 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000400f4c <+9>:     lea    0x8(%rsp),%rdx
   0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
   0x0000000000400f56 <+19>:    mov    $0x0,%eax
   0x0000000000400f5b <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000400f60 <+29>:    cmp    $0x1,%eax
   0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>
   0x0000000000400f65 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
   0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
   0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
   0x0000000000400f75 <+50>:    jmpq   *0x402470(,%rax,8)
   0x0000000000400f7c <+57>:    mov    $0xcf,%eax
   0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f83 <+64>:    mov    $0x2c3,%eax
   0x0000000000400f88 <+69>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f8a <+71>:    mov    $0x100,%eax
   0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f91 <+78>:    mov    $0x185,%eax
   0x0000000000400f96 <+83>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f98 <+85>:    mov    $0xce,%eax
   0x0000000000400f9d <+90>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400f9f <+92>:    mov    $0x2aa,%eax
   0x0000000000400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
   0x0000000000400fa6 <+99>:    mov    $0x147,%eax
   0x0000000000400fab <+104>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fad <+106>:   callq  0x40143a <explode_bomb>
   0x0000000000400fb2 <+111>:   mov    $0x0,%eax
   0x0000000000400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
   0x0000000000400fb9 <+118>:   mov    $0x137,%eax
   0x0000000000400fbe <+123>:   cmp    0xc(%rsp),%eax
   0x0000000000400fc2 <+127>:   je     0x400fc9 <phase_3+134>
   0x0000000000400fc4 <+129>:   callq  0x40143a <explode_bomb>
--Type <RET> for more, q to quit, c to continue without paging--RET
   0x0000000000400fc9 <+134>:   add    $0x18,%rsp
   0x0000000000400fcd <+138>:   retq
End of assembler dump.
```

进入 phase_3 断点，在 0x400f51c 处看到一个内存地址 0x4025cf，使用 x/s 查看得到 0x4025cf: "%d %d",说明 phase3 要输入两个空格隔开的整型数字。但 0x400f60 处的 cmp $0x1,%eax 和其后的两条指令 jg 0x400f6a 、callq 0x40143a 则表明我们输入的数必须超过两个，不然炸弹就被直接引爆。如果使用文件输入的话，在末尾回车换行即可表示第三个数；手动输入的话，输完两个字后按下回车即是第三个输入。0x400f6a 处的 $0x7,0x8(%rsp) 即其后的 ja 0x400fad，表明 0x8(%rsp) 处的值小于或等于 0x7，不妨记为 0xM 的 0x400f75 处的 jmpq *0x402470(,%rax,8) 指令表示直接跳转到 0x402470 + M*8 所存储的地址处

使用命令 x/16x 0x402470 查看从地址 0x402470 处开始的 8 个地址值

```
(gdb) x/16x 0x402470
0x402470:       0x00400f7c      0x00000000      0x00400fb9      0x00000000
0x402480:       0x00400f83      0x00000000      0x00400f8a      0x00000000
0x402490:       0x00400f91      0x00000000      0x00400f98      0x00000000
0x4024a0:       0x00400f9f      0x00000000      0x00400fa6      0x00000000
```

```
400f7c <+57>:    mov    $0xcf,%eax
400f81 <+62>:    jmp    0x400fbe <phase_3+123>
400f83 <+64>:    mov    $0x2c3,%eax
400f88 <+69>:    jmp    0x400fbe <phase_3+123>
400f8a <+71>:    mov    $0x100,%eax
400f8f <+76>:    jmp    0x400fbe <phase_3+123>
400f91 <+78>:    mov    $0x185,%eax
400f96 <+83>:    jmp    0x400fbe <phase_3+123>
400f98 <+85>:    mov    $0xce,%eax
400f9d <+90>:    jmp    0x400fbe <phase_3+123>
400f9f <+92>:    mov    $0x2aa,%eax
400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
400fa6 <+99>:    mov    $0x147,%eax
400fab <+104>:   jmp    0x400fbe <phase_3+123>
400fad <+106>:   callq  0x40143a <explode_bomb>
400fb2 <+111>:   mov    $0x0,%eax
400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
400fb9 <+118>:   mov    $0x137,%eax
400fbe <+123>:   cmp    0xc(%rsp),%eax
400fc2 <+127>:   je     0x400fc9 <phase_3+134>
400fc4 <+129>:   callq  0x40143a <explode_bomb>
400fc9 <+134>:   add    $0x18,%rsp
400fcd <+138>:   retq
```

所以以下跳转部分都为相应数字与%eax 作比较，将几个十六进制数字转换得到对应结果：

第一个数字	第二个数字十六进制	第二个数字十进制
0	cf	207
1	137	311
2	2c3	707
3	100	256
4	185	389
5	ce	206
6	2aa	682
7	147	327

要注意 0~7 的跳转顺序不是和代码中十六进制数字的顺序一一对应的，要参考 0x402470 后所给出的顺序来进行对照！

Phase4：

```
0x000000000040100c <+0>:     sub    $0x18,%rsp
0x0000000000401010 <+4>:     lea    0xc(%rsp),%rcx
0x0000000000401015 <+9>:     lea    0x8(%rsp),%rdx
0x000000000040101a <+14>:    mov    $0x4025cf,%esi
0x000000000040101f <+19>:    mov    $0x0,%eax
0x0000000000401024 <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
0x0000000000401029 <+29>:    cmp    $0x2,%eax
0x000000000040102c <+32>:    jne    0x401035 <phase_4+41>
0x000000000040102e <+34>:    cmpl   $0xe,0x8(%rsp)
0x0000000000401033 <+39>:    jbe    0x40103a <phase_4+46>
0x0000000000401035 <+41>:    callq  0x40143a <explode_bomb>
0x000000000040103a <+46>:    mov    $0xe,%edx
0x000000000040103f <+51>:    mov    $0x0,%esi
0x0000000000401044 <+56>:    mov    0x8(%rsp),%edi
0x0000000000401048 <+60>:    callq  0x400fce <func4>
0x000000000040104d <+65>:    test   %eax,%eax
0x000000000040104f <+67>:    jne    0x401058 <phase_4+76>
0x0000000000401051 <+69>:    cmpl   $0x0,0xc(%rsp)
0x0000000000401056 <+74>:    je     0x40105d <phase_4+81>
0x0000000000401058 <+76>:    callq  0x40143a <explode_bomb>
0x000000000040105d <+81>:    add    $0x18,%rsp
0x0000000000401061 <+85>:    retq
```

Phase4 开头部分和 phsae3 类似(0x40101a 处的 0x4025cf)，可以得知也要传入两个参数以及一个回车。Phase4 调用了函数 func4,此处先查看 fun4 代码。

```
mp of assembler code for function func4:
0x0000000000400fce <+0>:     sub    $0x8,%rsp
0x0000000000400fd2 <+4>:     mov    %edx,%eax
0x0000000000400fd4 <+6>:     sub    %esi,%eax
0x0000000000400fd6 <+8>:     mov    %eax,%ecx
0x0000000000400fd8 <+10>:    shr    $0x1f,%ecx
0x0000000000400fdb <+13>:    add    %ecx,%eax
0x0000000000400fdd <+15>:    sar    %eax
0x0000000000400fdf <+17>:    lea    (%rax,%rsi,1),%ecx
0x0000000000400fe2 <+20>:    cmp    %edi,%ecx
0x0000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
0x0000000000400fe6 <+24>:    lea    -0x1(%rcx),%edx
0x0000000000400fe9 <+27>:    callq  0x400fce <func4>
0x0000000000400fee <+32>:    add    %eax,%eax
0x0000000000400ff0 <+34>:    jmp    0x401007 <func4+57>
0x0000000000400ff2 <+36>:    mov    $0x0,%eax
0x0000000000400ff7 <+41>:    cmp    %edi,%ecx
0x0000000000400ff9 <+43>:    jge    0x401007 <func4+57>
0x0000000000400ffb <+45>:    lea    0x1(%rcx),%esi
0x0000000000400ffe <+48>:    callq  0x400fce <func4>
0x0000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax
0x0000000000401007 <+57>:    add    $0x8,%rsp
0x000000000040100b <+61>:    retq
```

可以看出 func4 在 0x400fe 处又调用了自己,可以推测该函数是一个递归函数。在 0x400fe2 处对%edi 和%ecx 进行比较，若%ecx<=%edi,则跳转，进行下一步计算然后递归，若%ecx>%edi,则进行下一步计算然后进行递归。

分析可知，只有在第一次 cmp %edi,%ecx 时 %ecx<=%edi ，在第二次 cmp %edi,%ecx 时 %ecx >= %edi，即 %ecx = %edi 时函数才能避免递归，正常结束。第一次 cmp %edi,%ecx 时 %ecx <= %edi ，在第二次 cmp %edi,%ecx 时 %ecx < %edi 时，会导致最终 %eax 的值为 1，回到 phase_4 后会引爆炸弹，所以这条路径是不能走的，故首次执行 func4 时，%ecx 的值为 7，%edi(第一个输入的数字)应该小于等于它。将调用 func4

的参数代入，即可求得第一个满足条件的 %edi 值为 7。
若第一次 cmp %edi,%ecx 时 %ecx > %edi，会递归调用 func4，可以得出 6, 3, 1, 0 等都可被接受。
Phase5：

```
Dump of assembler code for function phase_5:
   0x0000000000401062 <+0>:     push   %rbx
   0x0000000000401063 <+1>:     sub    $0x20,%rsp
   0x0000000000401067 <+5>:     mov    %rdi,%rbx
   0x000000000040106a <+8>:     mov    %fs:0x28,%rax
   0x0000000000401073 <+17>:    mov    %rax,0x18(%rsp)
   0x0000000000401078 <+22>:    xor    %eax,%eax
   0x000000000040107a <+24>:    callq  0x40131b <string_length>
   0x000000000040107f <+29>:    cmp    $0x6,%eax
   0x0000000000401082 <+32>:    je     0x4010d2 <phase_5+112>
   0x0000000000401084 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000401089 <+39>:    jmp    0x4010d2 <phase_5+112>
   0x000000000040108b <+41>:    movzbl (%rbx,%rax,1),%ecx
   0x000000000040108f <+45>:    mov    %cl,(%rsp)
   0x0000000000401092 <+48>:    mov    (%rsp),%rdx
   0x0000000000401096 <+52>:    and    $0xf,%edx
   0x0000000000401099 <+55>:    movzbl 0x4024b0(%rdx),%edx
   0x00000000004010a0 <+62>:    mov    %dl,0x10(%rsp,%rax,1)
   0x00000000004010a4 <+66>:    add    $0x1,%rax
   0x00000000004010a8 <+70>:    cmp    $0x6,%rax
   0x00000000004010ac <+74>:    jne    0x40108b <phase_5+41>
   0x00000000004010ae <+76>:    movb   $0x0,0x16(%rsp)
   0x00000000004010b3 <+81>:    mov    $0x40245e,%esi
   0x00000000004010b8 <+86>:    lea    0x10(%rsp),%rdi
   0x00000000004010bd <+91>:    callq  0x401338 <strings_not_equal>
   0x00000000004010c2 <+96>:    test   %eax,%eax
   0x00000000004010c4 <+98>:    je     0x4010d9 <phase_5+119>
   0x00000000004010c6 <+100>:   callq  0x40143a <explode_bomb>
   0x00000000004010cb <+105>:   nopl   0x0(%rax,%rax,1)
   0x00000000004010d0 <+110>:   jmp    0x4010d9 <phase_5+119>
   0x00000000004010d2 <+112>:   mov    $0x0,%eax
   0x00000000004010d7 <+117>:   jmp    0x40108b <phase_5+41>
   0x00000000004010d9 <+119>:   mov    0x18(%rsp),%rax
   0x00000000004010de <+124>:   xor    %fs:0x28,%rax
   0x00000000004010e7 <+133>:   je     0x4010ee <phase_5+140>
   0x00000000004010e9 <+135>:   callq  0x400b30 <__stack_chk_fail@plt>
   0x00000000004010ee <+140>:   add    $0x20,%rsp
--Type <RET> for more, q to quit, c to continue without paging--RET
   0x00000000004010f2 <+144>:   pop    %rbx
   0x00000000004010f3 <+145>:   retq
End of assembler dump.
(gdb)
```

通过前部分的 cmp 和 string_length 不难发现，第五题需要输入一个长度为（0x6）的字符串,6 个字符和一个换行。在 strings_not_equal 处可以看出上方 0x40245e 存放了与答案有关的字符串，x/s 显示得到：

```
(gdb) x/s 0x40245e
0x40245e:       "flyers"
```

向程序中输入这个 flyers，但炸弹被引爆了，说明 flyers 并不是最终答案。
继续看，一开始 %rbx 中存储的是用户输入字符串，逐个取出字符，只保留最低四位后存放在 %rdx。0x401099 处指令 movzbl 0x4024b0(%rdx),%edx 表示从 0x4024b0+ %rdx 处取出一字节数据并零扩展到 4 字节后存储到 %edx 中。该指令后的指令，将 %dl 中的数据存储在 0x10(%rsp,%rax,1) 处(即用户输入字符串)。查看一下 0x4024b0 处内容。得到了如下字样的字符串：

```
0x4024b0 <array.3449>:  "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

用户输入字符只有最低四位(假设为 x)才有意义，用来在 0x4024b0 处挑选第

x 个字符，然后这些挑选出的字符组合成一个字符串，这个字符串应该是 flyers ，这样才能与答案字符串对上。
flyers 的对应数字是 9H FH EH 5H 6H 7H ，从 ASCII 码表 中选择低位满足要求的字符组合。以下是 ascii 码表，即选择 16 进制数字的低位满足对应数字的字符即可。例如 ionefg 或者 yonuvw 等答案均可。

ASCII值	16进制	控制字符	ASCII值	16进制	控制字符
64	40H	@	96	60H	、
65	41H	A	97	61H	a
66	42H	B	98	62H	b
67	43H	C	99	63H	c
68	44H	D	100	64H	d
69	45H	E	101	65H	e
70	46H	F	102	66H	f
71	47H	G	103	67H	g
72	48H	H	104	68H	h
73	49H	I	105	69H	i
74	4AH	J	106	6AH	j
75	4BH	K	107	6BH	k
76	4CH	L	108	6CH	l
77	4DH	M	109	6DH	m
78	4EH	N	110	6EH	n
79	4FH	O	111	6FH	o
80	50H	P	112	70H	p
81	51H	Q	113	71H	q
82	52H	R	114	72H	r
83	53H	X	115	73H	s
84	54H	T	116	74H	t
85	55H	U	117	75H	u
86	56H	V	118	76H	v
87	57H	W	119	77H	w
88	58H	X	120	78H	x
89	59H	Y	121	79H	y
90	5AH	Z	122	7AH	z
91	5BH	[	123	7BH	{
92	5CH	/	124	7CH	|
93	5DH	]	125	7DH	}
94	5EH	^	126	7EH	~
95	5FH	—	127	7FH	DEL

Phase_6:

```
   0x00000000004010f4 <+0>:     push   %r14
   0x00000000004010f6 <+2>:     push   %r13
   0x00000000004010f8 <+4>:     push   %r12
   0x00000000004010fa <+6>:     push   %rbp
   0x00000000004010fb <+7>:     push   %rbx
   0x00000000004010fc <+8>:     sub    $0x50,%rsp
   0x0000000000401100 <+12>:    mov    %rsp,%r13
   0x0000000000401103 <+15>:    mov    %rsp,%rsi
   0x0000000000401106 <+18>:    callq  0x40145c <read_six_numbers>
   0x000000000040110b <+23>:    mov    %rsp,%r14
   0x000000000040110e <+26>:    mov    $0x0,%r12d
   0x0000000000401114 <+32>:    mov    %r13,%rbp
   0x0000000000401117 <+35>:    mov    0x0(%r13),%eax
   0x000000000040111b <+39>:    sub    $0x1,%eax
   0x000000000040111e <+42>:    cmp    $0x5,%eax
   0x0000000000401121 <+45>:    jbe    0x401128 <phase_6+52>
   0x0000000000401123 <+47>:    callq  0x40143a <explode_bomb>
   0x0000000000401128 <+52>:    add    $0x1,%r12d
   0x000000000040112c <+56>:    cmp    $0x6,%r12d
   0x0000000000401130 <+60>:    je     0x401153 <phase_6+95>
   0x0000000000401132 <+62>:    mov    %r12d,%ebx
   0x0000000000401135 <+65>:    movslq %ebx,%rax
   0x0000000000401138 <+68>:    mov    (%rsp,%rax,4),%eax
   0x000000000040113b <+71>:    cmp    %eax,0x0(%rbp)
   0x000000000040113e <+74>:    jne    0x401145 <phase_6+81>
   0x0000000000401140 <+76>:    callq  0x40143a <explode_bomb>
   0x0000000000401145 <+81>:    add    $0x1,%ebx
   0x0000000000401148 <+84>:    cmp    $0x5,%ebx
   0x000000000040114b <+87>:    jle    0x401135 <phase_6+65>
   0x000000000040114d <+89>:    add    $0x4,%r13
   0x0000000000401151 <+93>:    jmp    0x401114 <phase_6+32>
   0x0000000000401153 <+95>:    lea    0x18(%rsp),%rsi
   0x0000000000401158 <+100>:   mov    %r14,%rax
   0x000000000040115b <+103>:   mov    $0x7,%ecx
   0x0000000000401160 <+108>:   mov    %ecx,%edx
   0x0000000000401162 <+110>:   sub    (%rax),%edx
   0x0000000000401164 <+112>:   mov    %edx,(%rax)
   0x0000000000401166 <+114>:   add    $0x4,%rax
   0x000000000040116a <+118>:   cmp    %rsi,%rax
   0x000000000040116d <+121>:   jne    0x401160 <phase_6+108>
   0x000000000040116f <+123>:   mov    $0x0,%esi
   0x0000000000401174 <+128>:   jmp    0x401197 <phase_6+163>
   0x0000000000401176 <+130>:   mov    0x8(%rdx),%rdx
   0x000000000040117a <+134>:   add    $0x1,%eax
   0x000000000040117d <+137>:   cmp    %ecx,%eax
   0x000000000040117f <+139>:   jne    0x401176 <phase_6+130>
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401181 <+141>:   jmp    0x401188 <phase_6+148>
   0x0000000000401183 <+143>:   mov    $0x6032d0,%edx
   0x0000000000401188 <+148>:   mov    %rdx,0x20(%rsp,%rsi,2)
   0x000000000040118d <+153>:   add    $0x4,%rsi
   0x0000000000401191 <+157>:   cmp    $0x18,%rsi
   0x0000000000401195 <+161>:   je     0x4011ab <phase_6+183>
   0x0000000000401197 <+163>:   mov    (%rsp,%rsi,1),%ecx
   0x000000000040119a <+166>:   cmp    $0x1,%ecx
   0x000000000040119d <+169>:   jle    0x401183 <phase_6+143>
   0x000000000040119f <+171>:   mov    $0x1,%eax
   0x00000000004011a4 <+176>:   mov    $0x6032d0,%edx
   0x00000000004011a9 <+181>:   jmp    0x401176 <phase_6+130>
   0x00000000004011ab <+183>:   mov    0x20(%rsp),%rbx
   0x00000000004011b0 <+188>:   lea    0x28(%rsp),%rax
   0x00000000004011b5 <+193>:   lea    0x50(%rsp),%rsi
   0x00000000004011ba <+198>:   mov    %rbx,%rcx
   0x00000000004011bd <+201>:   mov    (%rax),%rdx
   0x00000000004011c0 <+204>:   mov    %rdx,0x8(%rcx)
   0x00000000004011c4 <+208>:   add    $0x8,%rax
   0x00000000004011c8 <+212>:   cmp    %rsi,%rax
   0x00000000004011cb <+215>:   je     0x4011d2 <phase_6+222>
   0x00000000004011cd <+217>:   mov    %rdx,%rcx
   0x00000000004011d0 <+220>:   jmp    0x4011bd <phase_6+201>
   0x00000000004011d2 <+222>:   movq   $0x0,0x8(%rdx)
   0x00000000004011da <+230>:   mov    $0x5,%ebp
   0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
   0x00000000004011e3 <+239>:   mov    (%rax),%eax
   0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
   0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250>
   0x00000000004011e9 <+245>:   callq  0x40143a <explode_bomb>
   0x00000000004011ee <+250>:   mov    0x8(%rbx),%rbx
   0x00000000004011f2 <+254>:   sub    $0x1,%ebp
   0x00000000004011f5 <+257>:   jne    0x4011df <phase_6+235>
   0x00000000004011f7 <+259>:   add    $0x50,%rsp
   0x00000000004011fb <+263>:   pop    %rbx
   0x00000000004011fc <+264>:   pop    %rbp
   0x00000000004011fd <+265>:   pop    %r12
   0x00000000004011ff <+267>:   pop    %r13
   0x0000000000401201 <+269>:   pop    %r14
   0x0000000000401203 <+271>:   retq
End of assembler dump.
(gb) 
```

<-phase6 反汇编得到的代码。

根据代码中的<read_six_numbers>可以知道其答案也是读取六个数字。并将数字储存在%rsp、%rsp+4、%rsp+8、%rsp+12、%rsp+16、%rsp+20 位置.

从 0x401117 开始可以得到%eax-1<=5，跳转，即若第一个数字大于 6 或小于 0（得到 FFF……FFF）则引爆炸弹。0x401138 处开始遍历第二个到第六个数字，要求它们均与第一个数字不相等，不然就引爆炸弹。0x401121 处用的是 jbe，说明这些数字是无符号数。jbe 的全称为（jump if below or equal），即小于等于则跳转。

JBE 指令的转移条件是 CF=1 或 ZF=1。如果比较的无符号数小于或等于另一个数，那么 CF 或 ZF 至少有一个会被置为 1，此时遇到 JBE 指令就会发生跳转。

综上，可以得知六个数字是 1 2 3 4 5 6 的一种排列组合。

满足上面条件后，跳转到 0x401153，到 0x40116d，这部分实现的是"7- 第 x 个数的值后，把该值放在第 x 个数原来的位置，x 从 0 到 6 遍历一遍"，用高级语言就是 a[i] = 7 - a[i]

在 0x4011a4 处可以发现一个地址 0x6032d0.x/s 显示内容，得到：

```
(gdb) x/s 0x6032d0
0x6032d0 <node1>:       "L\001"
```

这样的结果，只得到了一个node（节点），我们通过 x/128x 来查看接下来更多内容：

```
(gdb)  x/128x 0x6032d0
0x6032d0 <node1>:       0x4c    0x01    0x00    0x00    0x01    0x00    0x00    0x00
0x6032d8 <node1+8>:     0xe0    0x32    0x60    0x00    0x00    0x00    0x00    0x00
0x6032e0 <node2>:       0xa8    0x00    0x00    0x00    0x02    0x00    0x00    0x00
0x6032e8 <node2+8>:     0xf0    0x32    0x60    0x00    0x00    0x00    0x00    0x00
0x6032f0 <node3>:       0x9c    0x03    0x00    0x00    0x03    0x00    0x00    0x00
0x6032f8 <node3+8>:     0x00    0x33    0x60    0x00    0x00    0x00    0x00    0x00
0x603300 <node4>:       0xb3    0x02    0x00    0x00    0x04    0x00    0x00    0x00
0x603308 <node4+8>:     0x10    0x33    0x60    0x00    0x00    0x00    0x00    0x00
0x603310 <node5>:       0xdd    0x01    0x00    0x00    0x05    0x00    0x00    0x00
0x603318 <node5+8>:     0x20    0x33    0x60    0x00    0x00    0x00    0x00    0x00
0x603320 <node6>:       0xbb    0x01    0x00    0x00    0x06    0x00    0x00    0x00
0x603328 <node6+8>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x603330:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x603338:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x603340 <host_table>:  0x29    0x26    0x40    0x00    0x00    0x00    0x00    0x00
0x603348 <host_table+8>:        0x43    0x26    0x40    0x00    0x00    0x00    0x00    0x00
```

可以看出，一共有六个节点，每个节点占据 16 个字节内存空间，前 8 个字节存储内容值，后 8 个字节存储地址。而且从方框中的数据能够猜测这应该是个链表(前一个节点保存着后一个节点的地址).

代码在 0x40119a 处判断%ecx 和 1 的关系，若 ecx==1 则跳转至 0x401183,将链表的首地址内容存储到%edx,若 ecx!=1 则继续，

```
0x000000000040119f <+171>:    mov    $0x1,%eax
0x00000000004011a4 <+176>:    mov    $0x6032d0,%edx
0x00000000004011a9 <+181>:    jmp    0x401176 <phase_6+130>
                                        ↓
0x0000000000401176 <+130>:    mov    0x8(%rdx),%rdx
0x000000000040117a <+134>:    add    $0x1,%eax
0x000000000040117d <+137>:    cmp    %ecx,%eax
0x000000000040117d <+137>:    cmp    %ecx,%eax
0x000000000040117f <+139>:    jne    0x401176 <phase_6+130>
(产生了循环)
0x0000000000401181 <+141>:    jmp    0x401188 <phase_6+148>
```

此处的 0x8(%rdx)的内容正好是某个节点存储的后继结点的地址，假设当前存储的数是 m，在这里循环是为了保证%rdx 取到第 m 个节点的地址。

两种情况都是为了取到对应的地址，取址结束后：

```
0x0000000000401188 <+148>:    mov    %rdx,0x20(%rsp,%rsi,2)
0x000000000040118d <+153>:    add    $0x4,%rsi
```

然后将地址存储在 0x20 + %rsp + 2 * %rsi 处，最后将%rsi 和 0x18 进行比较，若相等则跳转至 0x4011ab,若不相等则回到 0x401197 重新循环。

最后得到数字和对应地址在栈中的分布：

4A	
48	数 6 对应地址
44	
40	数 5 对应地址
3C	
38	数 4 对应地址
34	
30	数 3 对应地址

2C	
28	数 2 对应地址
24	
20	数 1 对应地址
1C	
18	
14	数 6
10	数 5
C	数 4
8	数 3
4	数 2
%rsp+0x00	数 1

而且我们有如果 **某处存储的数大小是 m，那么它对应的就是第 m 个节点**

```
   0x00000000004011ab <+183>:   mov    0x20(%rsp),%rbx
   0x00000000004011b0 <+188>:   lea    0x28(%rsp),%rax
   0x00000000004011b5 <+193>:   lea    0x50(%rsp),%rsi
   0x00000000004011ba <+198>:   mov    %rbx,%rcx
   0x00000000004011bd <+201>:   mov    (%rax),%rdx
   0x00000000004011c0 <+204>:   mov    %rdx,0x8(%rcx)
   0x00000000004011c4 <+208>:   add    $0x8,%rax
   0x00000000004011c8 <+212>:   cmp    %rsi,%rax
   0x00000000004011cb <+215>:   je     0x4011d2 <phase_6+222>
   0x00000000004011cd <+217>:   mov    %rdx,%rcx
   0x00000000004011d0 <+220>:   jmp    0x4011bd <phase_6+201>
   0x00000000004011d2 <+222>:   movq   $0x0,0x8(%rdx)

   0x00000000004011da <+230>:   mov    $0x5,%ebp
   0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
   0x00000000004011e3 <+239>:   mov    (%rax),%eax
   0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
   0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250>
   0x00000000004011e9 <+245>:   callq  0x40143a <explode_bomb>
   0x00000000004011ee <+250>:   mov    0x8(%rbx),%rbx
   0x00000000004011f2 <+254>:   sub    $0x1,%ebp
   0x00000000004011f5 <+257>:   jne    0x4011df <phase_6+235>
   0x00000000004011f7 <+259>:   add    $0x50,%rsp
   0x00000000004011fb <+263>:   pop    %rbx
   0x00000000004011fc <+264>:   pop    %rbp
   0x00000000004011fd <+265>:   pop    %r12
   0x00000000004011ff <+267>:   pop    %r13
   0x0000000000401201 <+269>:   pop    %r14
   0x0000000000401203 <+271>:   retq
```

接下来看 0x4011ab 到 0x 4011d 2 处指令，这是把节点链接起来，即第一个数对应的结点的指针域(后 8 字节)存储第二个数对应节点的地址，依此类推，

最后让 数 6 对应节点的指针域指向 Null
0x4011d9 后的指令完成一件事：数 1 对应节点的内容**(取 4 个字节，而不是 8 个字节)**大于数 2 对应节点的内容，数 2 对应节点的内容大于数 3 对应节点的内容，以此类推。
各节点的内容值如下：

	Hex	Dec
Node1	0x0000014c	332
Node2	0x000000a8	168
Node3	0x0000039c	924
Node4	0x000002b3	691
Node5	0x000001dd	477
Node6	0x000001bb	443

根据结论"数 1 对应的节点的内容 > 数 2 对应节点的内容 > ... > 数 6 对应节点的内容"，节点链接情况是：node3 -> node4-> node5-> node6-> node1-> node2
再根据结论"如果 **某处存储的数大小是 m，那么它对应的就是第 m 个节点**"，所以数 1 到数 6 分别是：3、4、5、6、1、2
又由于在第 3 步中用 7 减去输入的数，所以输入的数应该是 4 3 2 1 6 5

5.实验总结及心得体会

（拆弹操作总结，实验中遇到的问题及解决方法等）

拆弹实验很大程度上增强了我们阅读汇编语言的能力，加深了对寄存器，栈等计算机结构的理解。每个不同的寄存器有着比较固定的用途，例如%eax 常用于储存函数的返回值(32 位无符号数),%rax 则储存 64 位无符号数。
通过 phase6 了解了链表结构，每一个节点存储了下一个节点的地址。