# Bomb Lab Report

Danni Ke

## Phase 1

**Assembly Code:**

```
Dump of assembler code for function phase_1:
   0x00000000004010df <+0>:     sub    $0x8,%rsp
   0x00000000004010e3 <+4>:     mov    $0x40243d,%esi
   0x00000000004010e8 <+9>:     callq  0x4012bc <strings_not_equal>
   0x00000000004010ed <+14>:    test   %eax,%eax
   0x00000000004010ef <+16>:    je     0x4010f6 <phase_1+23>
   0x00000000004010f1 <+18>:    callq  0x40147f <explode_bomb>
   0x00000000004010f6 <+23>:    add    $0x8,%rsp
   0x00000000004010fa <+27>:    retq
```

1. First of all, we should get the assemble code by using the command, *disas,* in GDB. In the graph below in the assemble I got from phase_1. It's obvious that the system will call function explode_bomb in 0x4010f1<phase_1+18>, so in order to avoid explode the bomb, I set break point in 0x4017f; it will stop at the moment I meet the explode bomb. Also breakpoint of phase_1 is set so that I can go through those instruction step by step by using *stepi*.

```
Reading symbols from /home/ugrads/dxk5418/311/bomb171/bomb...done.
(gdb) b explode_bomb
Breakpoint 1 at 0x40147f
Breakpoint 2 at 0x401483
(gdb) b phase_1
```

2. Then, run the program and type in random input (here I type in abc), stepping each instruction.

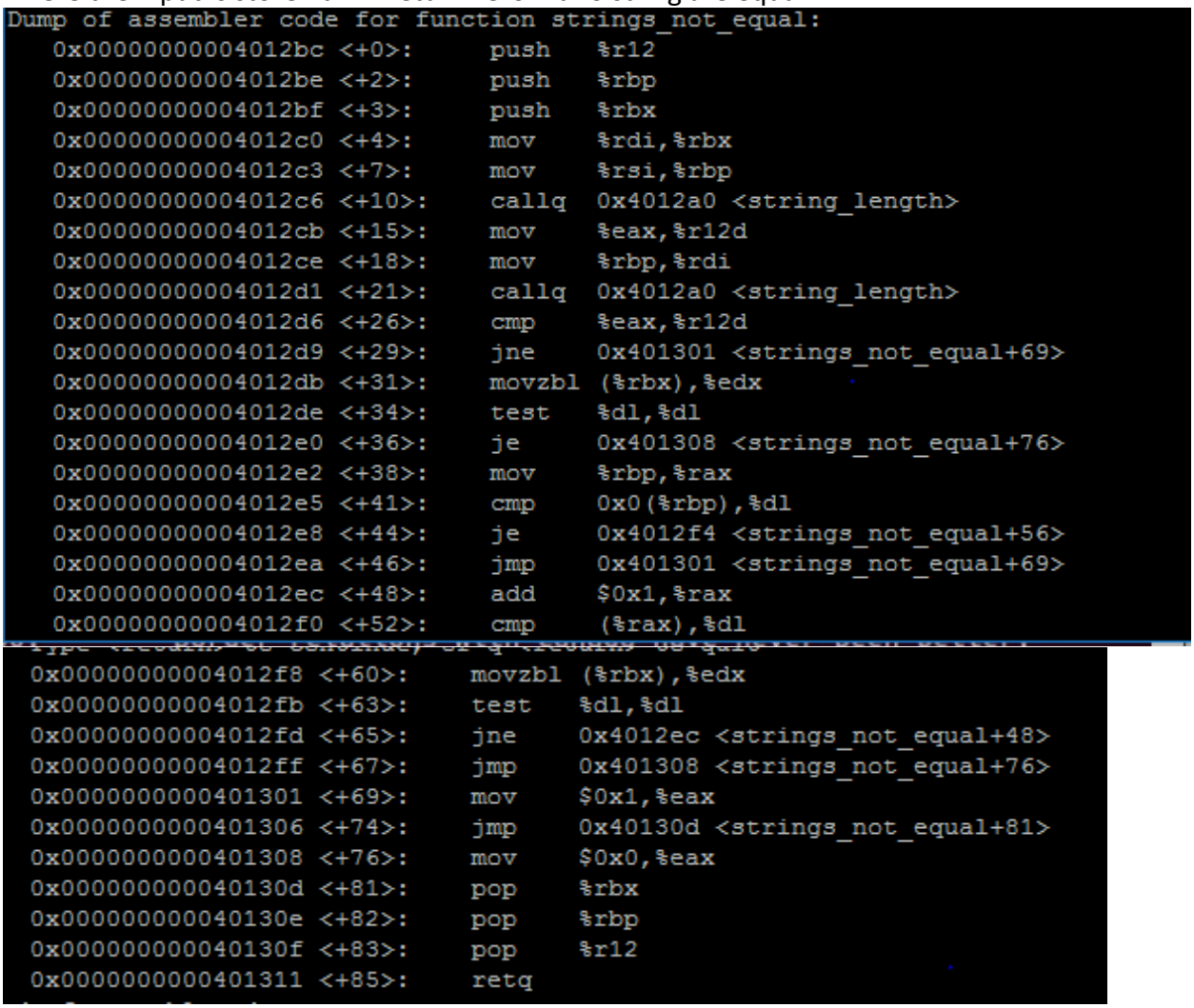
```
Starting program: /home/ugrads/dxk5418/311/bomb171/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
abc
```

```
Dump of assembler code for function phase_1:
   0x00000000004010df <+0>:     sub    $0x8,%rsp
=> 0x00000000004010e3 <+4>:     mov    $0x40243d,%esi
   0x00000000004010e8 <+9>:     callq  0x4012bc <strings_not_equal>
   0x00000000004010ed <+14>:    test   %eax,%eax
   0x00000000004010ef <+16>:    je     0x4010f6 <phase_1+23>
   0x00000000004010f1 <+18>:    callq  0x40147f <explode_bomb>
   0x00000000004010f6 <+23>:    add    $0x8,%rsp
   0x00000000004010fa <+27>:    retq
```

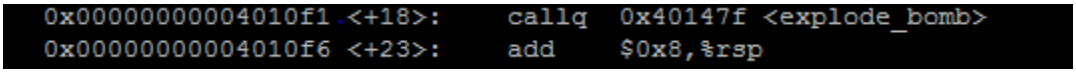
3. In 0x4010e3, content in the address of 0x40243d is moved to the $esi (register). I was curious what is inside. So use print and x/s to check the content inside the $esi and $0x40243d. I thought it should be my input (abc), but it is not.

```
(gdb) print $esi
$1 = 4203581
(gdb) x/s $esi
0x40243d <__dso_handle+389>:     "Public speaking is very easy."
(gdb) x/s 0x40243d
0x40243d <__dso_handle+389>:     "Public speaking is very easy."
```

4. Keeping going, a function <strings_not_equal> is called. I disas the function. The $rsi ($esi) and $rdi is send to this function. So I guess $rdi is our input and it will be compared with the $rsi. Graph below is <strings_not _equal>. I check the register $rdi, found that it's the register where the input is store. It will return zero if two string are equal.




5. Finally, the output of this function is %eax. And in 0x4010ed, it test if the %eax is zero. It jumps to the explode_bomb if it is not zero. So in phase_1, I should type the word stored in $rsi to defuse the bomb.




Phase_1 password: **Public speaking is very easy.**

## Phase 2

**Assembly Code:**

```
Dump of assembler code for function phase_2:
   0x0000000000401036 <+0>:     push   %rbp
   0x0000000000401037 <+1>:     push   %rbx
   0x0000000000401038 <+2>:     sub    $0x28,%rsp
   0x000000000040103c <+6>:     mov    %rsp,%rsi
   0x000000000040103f <+9>:     callq  0x4014b5 <read_six_numbers>
   0x0000000000401044 <+14>:    cmpl   $0x0,(%rsp)
   0x0000000000401048 <+18>:    jne    0x401051 <phase_2+27>
   0x000000000040104a <+20>:    cmpl   $0x1,0x4(%rsp)
   0x000000000040104f <+25>:    je     0x401056 <phase_2+32>
   0x0000000000401051 <+27>:    callq  0x40147f <explode_bomb>
   0x0000000000401056 <+32>:    mov    %rsp,%rbp
   0x0000000000401059 <+35>:    lea    0x8(%rsp),%rbx
   0x000000000040105e <+40>:    add    $0x18,%rbp
   0x0000000000401062 <+44>:    mov    -0x4(%rbx),%eax
   0x0000000000401065 <+47>:    add    -0x8(%rbx),%eax
   0x0000000000401068 <+50>:    cmp    %eax,(%rbx)
   0x000000000040106a <+52>:    je     0x401071 <phase_2+59>
   0x000000000040106c <+54>:    callq  0x40147f <explode_bomb>
   0x0000000000401071 <+59>:    add    $0x4,%rbx
   0x0000000000401075 <+63>:    cmp    %rbp,%rbx
   0x0000000000401078 <+66>:    jne    0x401062 <phase_2+44>
   0x000000000040107a <+68>:    add    $0x28,%rsp
---Type <return> to continue, or q <return> to quit---
   0x000000000040107e <+72>:    pop    %rbx
   0x000000000040107f <+73>:    pop    %rbp
   0x0000000000401080 <+74>:    retq
```

1. Graph below is our assembly code for phase_2, I notice that it creates some space in $rsp in stack, then it calls the function <read_six_number> in 0x40103f. I am guessing that the system will read six number from inputs. I use p/x check the $rsp after I type in 6 random number(after read six number function). It's proved that they were read and stored in $rsp.

```
(gdb) p/x *0x7fffffffdfa0@6
$17 = {0x0, 0x1, 0x1, 0x2, 0x3, 0x4}
```

2. The first output is in $rsp, and it tests if it equals to zero, so the first number is 0 If the first input is not zero, it will jump to the <phase_2+27>, which calls the function explode_bomb. Similarly, it compares the second number with one, if it's not one, it will explode. So the second number is 1.

```
 0x0000000000401044 <+14>:    cmpl   $0x0,(%rsp)
 0x0000000000401048 <+18>:    jne    0x401051 <phase_2+27>
 0x000000000040104a <+20>:    cmpl   $0x1,0x4(%rsp)
 0x000000000040104f <+25>:    je     0x401056 <phase_2+32>
 0x0000000000401051 <+27>:    callq  0x40147f <explode_bomb>
```

3. $rbp is probably the condition for the loop to get the other four numbers, so it is assigned the starting address of those six number and upper bound(0x18 which is 24 in decimal, it's size for six ints, 4 byte per int ). At the same time, $rbx starts from third number (0x8($rsp)). Every

loop it will be the sum of former two elements.

```
0x0000000000401056 <+32>:    mov    %rsp,%rbp
0x0000000000401059 <+35>:    lea    0x8(%rsp),%rbx
0x000000000040105e <+40>:    add    $0x18,%rbp
 0x0000000000401062 <+44>:    mov    -0x4(%rbx),%eax
 0x0000000000401065 <+47>:    add    -0x8(%rbx),%eax
```

4. In every loop, the address $rbx point to will move 4 bytes, which is next number until it reach the $rbp. Therefore, our third number will be 0+1, 1. Forth number will be 1+1, 2. Fifth number will be 1+2, 3. Sixth number will be 2+3, 5. It's actually a Fibonacci sequence.

```
0x0000000000401071 <+59>:    add    $0x4,%rbx
0x0000000000401075 <+63>:    cmp    %rbp,%rbx
0x0000000000401078 <+66>:    jne    0x401062 <phase_2+44>
```

Phase_2 password: **0 1 1 2 3 5**

## Phase 3

**Assembly Code:**

```
Dump of assembler code for function phase_3:
   0x0000000000401158 <+0>:     sub    $0x18,%rsp
   0x000000000040115c <+4>:     lea    0x7(%rsp),%rcx
   0x0000000000401161 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401166 <+14>:    lea    0x8(%rsp),%r8
   0x000000000040116b <+19>:    mov    $0x40245b,%esi
   0x0000000000401170 <+24>:    mov    $0x0,%eax
   0x0000000000401175 <+29>:    callq  0x400ac8 <__isoc99_sscanf@plt>
   0x000000000040117a <+34>:    cmp    $0x2,%eax
   0x000000000040117d <+37>:    jg     0x401184 <phase_3+44>
   0x000000000040117f <+39>:    callq  0x40147f <explode_bomb>
   0x0000000000401184 <+44>:    cmpl   $0x7,0xc(%rsp)
   0x0000000000401189 <+49>:    ja     0x40126d <phase_3+277>
   0x000000000040118f <+55>:    mov    0xc(%rsp),%eax
   0x0000000000401193 <+59>:    jmpq   *0x402470(,%rax,8)
   0x000000000040119a <+66>:    mov    $0x65,%eax
   0x000000000040119f <+71>:    cmpl   $0x54,0x8(%rsp)
   0x00000000004011a4 <+76>:    je     0x401285 <phase_3+301>
   0x00000000004011aa <+82>:    callq  0x40147f <explode_bomb>
   0x00000000004011af <+87>:    mov    $0x65,%eax
   0x00000000004011b4 <+92>:    jmpq   0x401285 <phase_3+301>
   0x00000000004011b9 <+97>:    cmpl   $0x5b,0x8(%rsp)
   0x00000000004011be <+102>:   je     0x401279 <phase_3+289>
---Type <return> to continue, or q <return> to quit---
   0x00000000004011c4 <+108>:   callq  0x40147f <explode_bomb>
   0x00000000004011c9 <+113>:   mov    $0x69,%eax
   0x00000000004011ce <+118>:   jmpq   0x401285 <phase_3+301>
   0x00000000004011d3 <+123>:   cmpl   $0xae,0x8(%rsp)
   0x00000000004011db <+131>:   je     0x401280 <phase_3+296>
   0x00000000004011e1 <+137>:   callq  0x40147f <explode_bomb>
   0x00000000004011e6 <+142>:   mov    $0x67,%eax
   0x00000000004011eb <+147>:   jmpq   0x401285 <phase_3+301>
   0x00000000004011f0 <+152>:   cmpl   $0x15b,0x8(%rsp)
   0x00000000004011f8 <+160>:   je     0x401279 <phase_3+289>
   0x00000000004011fa <+162>:   callq  0x40147f <explode_bomb>
   0x00000000004011ff <+167>:   mov    $0x69,%eax
   0x0000000000401204 <+172>:   jmp    0x401285 <phase_3+301>
   0x0000000000401206 <+174>:   mov    $0x68,%eax
   0x000000000040120b <+179>:   cmpl   $0xa1,0x8(%rsp)
   0x0000000000401213 <+187>:   je     0x401285 <phase_3+301>
   0x0000000000401215 <+189>:   callq  0x40147f <explode_bomb>
   0x000000000040121a <+194>:   mov    $0x68,%eax
   0x000000000040121f <+199>:   jmp    0x401285 <phase_3+301>
   0x0000000000401221 <+201>:   cmpl   $0x95,0x8(%rsp)
   0x0000000000401229 <+209>:   je     0x401280 <phase_3+296>
   0x000000000040122b <+211>:   callq  0x40147f <explode_bomb>
   0x0000000000401230 <+216>:   mov    $0x67,%eax
---Type <return> to continue, or q <return> to quit---
   0x0000000000401235 <+221>:   jmp    0x401285 <phase_3+301>
   0x0000000000401237 <+223>:   mov    $0x77,%eax
   0x000000000040123c <+228>:   cmpl   $0x143,0x8(%rsp)
   0x0000000000401244 <+236>:   je     0x401285 <phase_3+301>
   0x0000000000401246 <+238>:   callq  0x40147f <explode_bomb>
   0x000000000040124b <+243>:   mov    $0x77,%eax
   0x0000000000401250 <+248>:   jmp    0x401285 <phase_3+301>
   0x0000000000401252 <+250>:   mov    $0x62,%eax
   0x0000000000401257 <+255>:   cmpl   $0x2b0,0x8(%rsp)
   0x000000000040125f <+263>:   je     0x401285 <phase_3+301>
   0x0000000000401261 <+265>:   callq  0x40147f <explode_bomb>
```

```
   0x0000000000401266 <+270>:    mov    $0x62,%eax
   0x000000000040126b <+275>:    jmp    0x401285 <phase_3+301>
   0x000000000040126d <+277>:    callq  0x40147f <explode_bomb>
   0x0000000000401272 <+282>:    mov    $0x6c,%eax
   0x0000000000401277 <+287>:    jmp    0x401285 <phase_3+301>
   0x0000000000401279 <+289>:    mov    $0x69,%eax
   0x000000000040127e <+294>:    jmp    0x401285 <phase_3+301>
   0x0000000000401280 <+296>:    mov    $0x67,%eax
   0x0000000000401285 <+301>:    cmp    0x7(%rsp),%al
   0x0000000000401289 <+305>:    je     0x401290 <phase_3+312>
   0x000000000040128b <+307>:    callq  0x40147f <explode_bomb>
   0x0000000000401290 <+312>:    add    $0x18,%rsp
```

1. At the beginning, we have three registers being assigned with different space in $rsp. Then, the $0x40245b is moved to $esi. So I check the $0x40245b first to see the format of input. Also I use x/sb to print out the content inside the $0x40245b, which is “%d %c %d” (decimal char decimal), the format of password. Start debugging right now, set breakpoints in the scan function and other points.

```
   0x000000000040115c <+4>:     lea    0x7(%rsp),%rcx
   0x0000000000401161 <+9>:     lea    0xc(%rsp),%rdx
   0x0000000000401166 <+14>:    lea    0x8(%rsp),%r8
   0x000000000040116b <+19>:    mov    $0x40245b,%esi
0x000000000040116b <+19>:    mov    $0x40245b,%esi
(gdb) x/sb 0x40245b
0x40245b <__dso_handle+419>:     "%d %c %d"
```

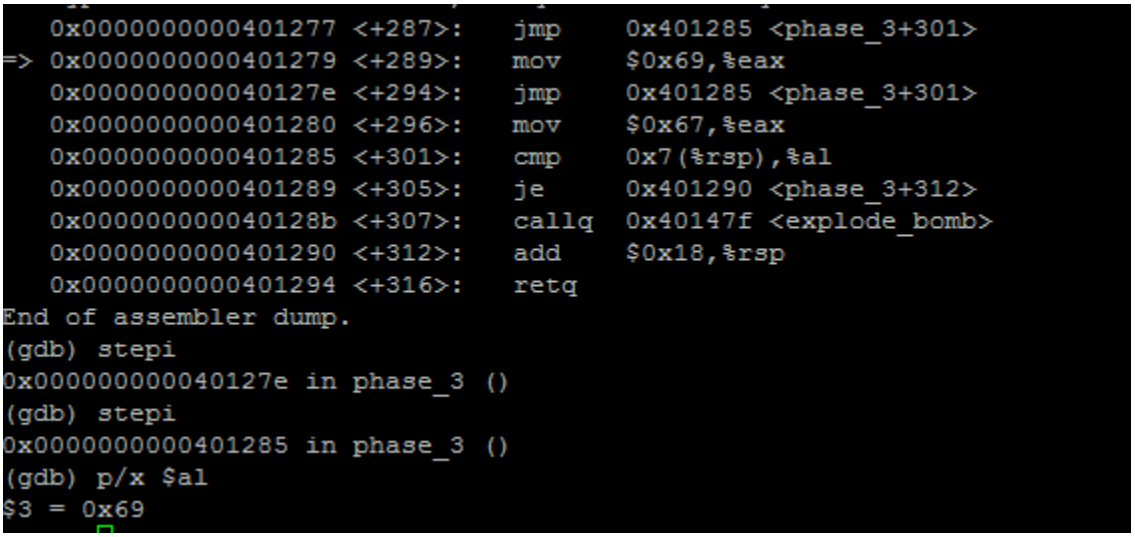
2. I type random combination of the numbers and char, 1 a 2 to see how they are stored. Apparently, they are stored in $rsp. First number is in 0xc.  And the number should be smaller than 0x7 or it will jump to explode_bomb.

```
(gdb) p/x *0x7fffffffdfc0@3
$22 = {0xffffe0d8, 0x61007fff, 0x3}
(gdb) p/x *0x7fffffffdfc0@5
$23 = {0xffffe0d8, 0x61007fff, 0x3, 0x1, 0x0}
(gdb) p/x *0x7fffffffdfc0@7
$24 = {0xffffe0d8, 0x61007fff, 0x3, 0x1, 0x0, 0x0, 0x400e0a}
 0x0000000000401184 <+44>:    cmpl   $0x7,0xc(%rsp)
 0x0000000000401189 <+49>:    ja     0x40126d <phase_3+277>
```

3. As I keep stepi, it jump to graph below, which $rax equals first number which is 1. The final address is 0x402470+ 8*1=0x402478, using p/x to check the actual address. In this test, second number is compared with 0x5b (91). So we have our second number here. If equal, we jump to the <phase_3+289>.

```
  0x0000000000401193 <+59>:     jmpq   *0x402470(,%rax,8)
(gdb) p/x *0x402478
$1 = 0x4011b9
  0x00000000004011b9 <+97>:     cmpl   $0x5b,0x8(%rsp)
  0x00000000004011be <+102>:    je     0x401279 <phase_3+289>
```

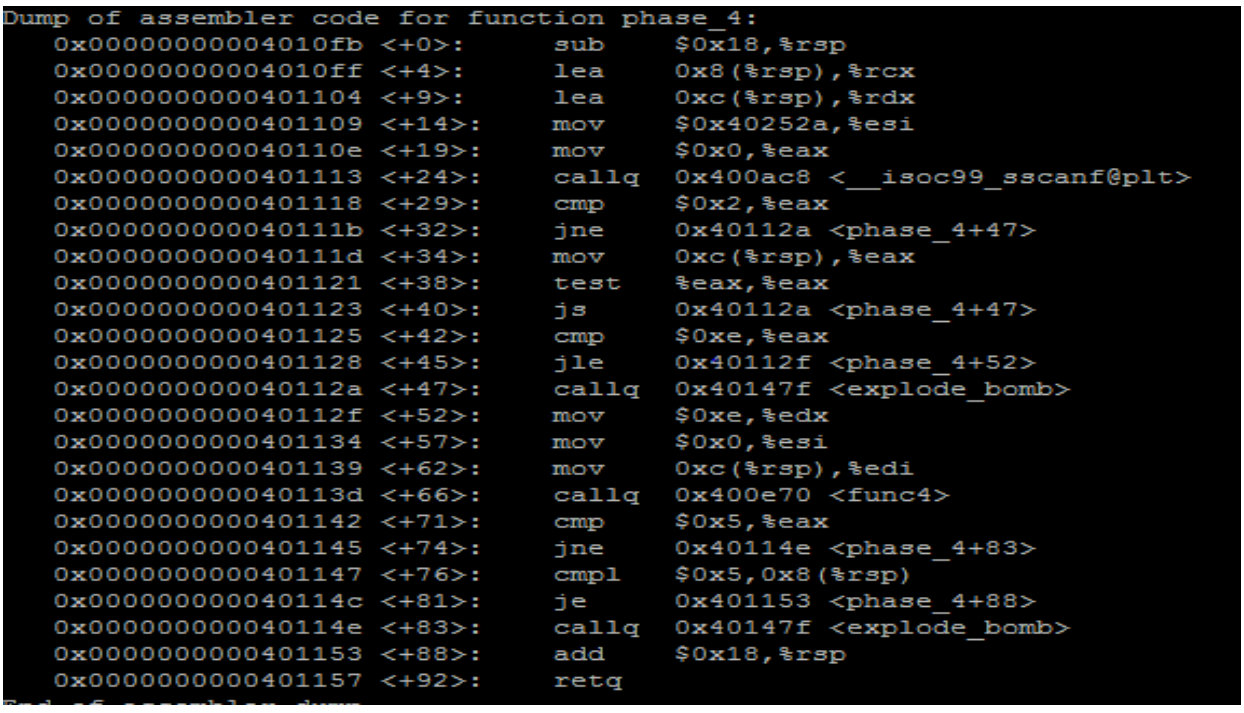
4. When we stepi to the following part, I notice the 0x7($rsp) is compared with $al, so I check the $al, number inside is 0x69, which is i in ASCII Table.




Phase_3 password: **1 i 91**

## Phase 4

**Assembly Code:**




1. According to assembly code above, I guess I need to type in two numbers. I still check the 0x40252a. Result is below, which are two decimal. Based on phased 2, the first number is stored in

0xc($rsp), and second number is stored in 0x8($rsp). Also I notice in 0x401147, second number is compared with 0x5 or the bomb explodes. So the second number should be 5. And the first number is send to the $edi, to the func4, then the return value, $eax, is compared with the 0x5. Therefore, the first number should be certain number that return a number $0x5.

```
(gdb) x/sb 0x40252a
0x40252a:       "%d %d"
```

```
0x0000000000401147 <+76>:    cmpl   $0x5,0x8(%rsp)
0x000000000040114c <+81>:    je     0x401153 <phase_4+88>
0x000000000040114e <+83>:    callq  0x40147f <explode_bomb>
```

```
0x0000000000401139 <+62>:    mov    0xc(%rsp),%edi
0x000000000040113d <+66>:    callq  0x400e70 <func4>
0x0000000000401142 <+71>:    cmp    $0x5,%eax
0x0000000000401145 <+74>:    jne    0x40114e <phase_4+83>
```

2. First number is send to $eax, then test if it's zero, or negative or the bomb explodes. Then it is compared with 0xe. So the first number should be less than 0xe or the bomb explodes. Then the first number will be send to <func4>.

```
0x000000000040111d <+34>:    mov    0xc(%rsp),%eax
0x0000000000401121 <+38>:    test   %eax,%eax
0x0000000000401123 <+40>:    js     0x40112a <phase_4+47>
0x0000000000401125 <+42>:    cmp    $0xe,%eax
0x0000000000401128 <+45>:    jle    0x40112f <phase_4+52>
0x000000000040112a <+47>:    callq  0x40147f <explode_bomb>
```

3. In the Function 4, graph below are three parameters. And Function 4 is a recursive function. However I decided to try all the number that is less than 14 to see which can pass the func4, return the $eax = 5; Finally, I figure out the number is 10.

```
0x000000000040112f <+52>:    mov    $0xe,%edx
0x0000000000401134 <+57>:    mov    $0x0,%esi
0x0000000000401139 <+62>:    mov    0xc(%rsp),%edi
```

.

```
0x0000000000400e70 <+0>:     sub    $0x8,%rsp
0x0000000000400e74 <+4>:     mov    %edx,%eax
0x0000000000400e76 <+6>:     sub    %esi,%eax
0x0000000000400e78 <+8>:     mov    %eax,%ecx
0x0000000000400e7a <+10>:    shr    $0x1f,%ecx
0x0000000000400e7d <+13>:    lea    (%rcx,%rax,1),%eax
0x0000000000400e80 <+16>:    sar    %eax
0x0000000000400e82 <+18>:    lea    (%rax,%rsi,1),%ecx
0x0000000000400e85 <+21>:    cmp    %edi,%ecx
0x0000000000400e87 <+23>:    jle    0x400e95 <func4+37>
0x0000000000400e89 <+25>:    lea    -0x1(%rcx),%edx
0x0000000000400e8c <+28>:    callq  0x400e70 <func4>
0x0000000000400e91 <+33>:    add    %eax,%eax
0x0000000000400e93 <+35>:    jmp    0x400eaa <func4+58>
0x0000000000400e95 <+37>:    mov    $0x0,%eax
0x0000000000400e9a <+42>:    cmp    %edi,%ecx
0x0000000000400e9c <+44>:    jge    0x400eaa <func4+58>
0x0000000000400e9e <+46>:    lea    0x1(%rcx),%esi
0x0000000000400ea1 <+49>:    callq  0x400e70 <func4>
0x0000000000400ea6 <+54>:    lea    0x1(%rax,%rax,1),%eax
0x0000000000400eaa <+58>:    add    $0x8,%rsp
0x0000000000400eae <+62>:    retq
```

I did figure out the corresponding c code, but it really takes time to trace them step by step.

Phase_4 password: **10 5**

## Phase 5

**Assembly Code:**

```
   0x0000000000401081 <+0>:     push   %rbx
   0x0000000000401082 <+1>:     sub    $0x10,%rsp
   0x0000000000401086 <+5>:     mov    %rdi,%rbx
   0x0000000000401089 <+8>:     callq  0x4012a0 <string_length>
   0x000000000040108e <+13>:    cmp    $0x6,%eax
   0x0000000000401091 <+16>:    je     0x401098 <phase_5+23>
   0x0000000000401093 <+18>:    callq  0x40147f <explode_bomb>
   0x0000000000401098 <+23>:    mov    %rsp,%rax
   0x000000000040109b <+26>:    lea    0x6(%rbx),%rsi
   0x000000000040109f <+30>:    mov    $0x4024b0,%edx
   0x00000000004010a4 <+35>:    movsbq (%rbx),%rcx
   0x00000000004010a8 <+39>:    and    $0xf,%ecx
   0x00000000004010ab <+42>:    movzbl (%rdx,%rcx,1),%ecx
   0x00000000004010af <+46>:    mov    %cl,(%rax)
   0x00000000004010b1 <+48>:    add    $0x1,%rbx
   0x00000000004010b5 <+52>:    add    $0x1,%rax
   0x00000000004010b9 <+56>:    cmp    %rsi,%rbx
   0x00000000004010bc <+59>:    jne    0x4010a4 <phase_5+35>
   0x00000000004010be <+61>:    movb   $0x0,0x6(%rsp)
   0x00000000004010c3 <+66>:    mov    %rsp,%rdi
   0x00000000004010c6 <+69>:    mov    $0x402436,%esi
   0x00000000004010cb <+74>:    callq  0x4012bc <strings_not_equal>
---Type <return> to continue, or q <return> to quit---
   0x00000000004010d0 <+79>:    test   %eax,%eax
   0x00000000004010d2 <+81>:    je     0x4010d9 <phase_5+88>
   0x00000000004010d4 <+83>:    callq  0x40147f <explode_bomb>
   0x00000000004010d9 <+88>:    add    $0x10,%rsp
   0x00000000004010dd <+92>:    pop    %rbx
   0x00000000004010de <+93>:    retq
```

1. Firstly, I notice the input format will be string of length 6. I still wanna check the content inside the $rdi, $rdx, which is our input.

```
0x0000000000401086 <+5>:     mov    %rdi,%rbx
0x0000000000401089 <+8>:     callq  0x4012a0 <string_length>
0x000000000040108e <+13>:    cmp    $0x6,%eax
0x0000000000401091 <+16>:    je     0x401098 <phase_5+23>
```

```
(gdb) x/s $rbx
0x603e60 <input_strings+320>:    "ionefg"
(gdb) x/s $rdi
0x603e60 <input_strings+320>:    "ionefg"
```

2. Here is a loop, take in the input and use instruction 0x4010ab to turn it other string, which is kind of like decode. In instruction 0x4010b9, it compares the null with the string in every loop until it becomes empty.

```
 0x00000000004010a4 <+35>:    movsbq (%rbx),%rcx
 0x00000000004010a8 <+39>:    and    $0xf,%ecx
 0x00000000004010ab <+42>:    movzbl (%rdx,%rcx,1),%ecx
 0x00000000004010af <+46>:    mov    %cl,(%rax)
 0x00000000004010b1 <+48>:    add    $0x1,%rbx
 0x00000000004010b5 <+52>:    add    $0x1,%rax
 0x00000000004010b9 <+56>:    cmp    %rsi,%rbx
 0x00000000004010bc <+59>:    jne    0x4010a4 <phase_5+35>
 0x00000000004010be <+61>:    movb   $0x0,0x6(%rsp)
```

3. Finally, it calls a strings_not_equal function to compare the string generated by the step 2 with string inside the 0x402436, which is “flyers”

```
 0x00000000004010c3 <+66>:    mov    %rsp,%rdi
 0x00000000004010c6 <+69>:    mov    $0x402436,%esi
 0x00000000004010cb <+74>:    callq  0x4012bc <strings_not_equal>
--Type <return> to continue, or q <return> to quit---
 0x00000000004010d0 <+79>:    test   %eax,%eax
 0x00000000004010d2 <+81>:    je     0x4010d9 <phase_5+88>
 0x00000000004010d4 <+83>:    callq  0x40147f <explode_bomb>
 0x00000000004010d9 <+88>:    add    $0x10,%rsp
 0x00000000004010dd <+92>:    pop    %rbx
 0x00000000004010de <+93>:    retq
```

```
(gdb) x/s 0x402436
0x402436 <__dso_handle+382>:     "flyers"
```

Phase_5 password: **ionefg**