CSED211 LAB3 report

20210054 정하우

Bomb lab을 하기 위해 gdb로 디버깅을 시작한다.

Disas(disassemble)명령어를 통해 maind을 먼저 본다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000000010aa <+0>:     push   %rbx
   0x00000000000010ab <+1>:     cmp    $0x1,%edi
   0x00000000000010ae <+4>:     je     0x11ac <main+258>
   0x00000000000010b4 <+10>:    mov    %rsi,%rbx
   0x00000000000010b7 <+13>:    cmp    $0x2,%edi
   0x00000000000010ba <+16>:    jne    0x11e1 <main+311>
   0x00000000000010c0 <+22>:    mov    0x8(%rsi),%rdi
   0x00000000000010c4 <+26>:    lea    0x1599(%rip),%rsi        # 0x2664
   0x00000000000010cb <+33>:    callq  0xf10 <fopen@plt>
   0x00000000000010d0 <+38>:    mov    %rax,0x2035b9(%rip)        # 0x204690 <infile>
   0x00000000000010d7 <+45>:    test   %rax,%rax
   0x00000000000010da <+48>:    je     0x11bf <main+277>
   0x00000000000010e0 <+54>:    callq  0x16fa <initialize_bomb>
   0x00000000000010e5 <+59>:    lea    0x15fc(%rip),%rdi        # 0x26e8
   0x00000000000010ec <+66>:    callq  0xe30 <puts@plt>
   0x00000000000010f1 <+71>:    lea    0x1630(%rip),%rdi        # 0x2728
   0x00000000000010f8 <+78>:    callq  0xe30 <puts@plt>
   0x00000000000010fd <+83>:    callq  0x1806 <read_line>
   0x0000000000001102 <+88>:    mov    %rax,%rdi
   0x0000000000001105 <+91>:    callq  0x1204 <phase_1>
   0x000000000000110a <+96>:    callq  0x194a <phase_defused>
   0x000000000000110f <+101>:   lea    0x1642(%rip),%rdi        # 0x2758
   0x0000000000001116 <+108>:   callq  0xe30 <puts@plt>
   0x000000000000111b <+113>:   callq  0x1806 <read_line>
   0x0000000000001120 <+118>:   mov    %rax,%rdi
   0x0000000000001123 <+121>:   callq  0x1224 <phase_2>
   0x0000000000001128 <+126>:   callq  0x194a <phase_defused>
   0x000000000000112d <+131>:   lea    0x1569(%rip),%rdi        # 0x269d
   0x0000000000001134 <+138>:   callq  0xe30 <puts@plt>
   0x0000000000001139 <+143>:   callq  0x1806 <read_line>
   0x000000000000113e <+148>:   mov    %rax,%rdi
   0x0000000000001141 <+151>:   callq  0x128d <phase_3>
   0x0000000000001146 <+156>:   callq  0x194a <phase_defused>
   0x000000000000114b <+161>:   lea    0x1569(%rip),%rdi        # 0x26bb
   0x0000000000001152 <+168>:   callq  0xe30 <puts@plt>
   0x0000000000001157 <+173>:   callq  0x1806 <read_line>
   0x000000000000115c <+178>:   mov    %rax,%rdi
```

R(run)명령어를 이용해서 프로그램을 실행시킨다. 그리고 아무 단어나 치면 폭탄이 터지는 것을 볼 수 있다.

```
(gdb) r
Starting program: /home/std/howru0321/bomb20210054/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
b^H

BOOM!!!
The bomb has blown up.
[Inferior 1 (process 26261) exited with code 010]
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.x86_64
```

<phase_1>

그래서 phase_1에 breakpoint를 두고 그 밑으로는 실행이 안되도록 하였다.

```
(gdb) b phase_1
Breakpoint 1 at 0x555555555204
(gdb) r
Starting program: /home/std/howru0321/bomb20210054/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
hi

Breakpoint 1, 0x0000555555555204 in phase_1 ()
(gdb) disas phase_1
Dump of assembler code for function phase_1:
=> 0x0000555555555204 <+0>:     sub    $0x8,%rsp
   0x0000555555555208 <+4>:     lea    0x15a1(%rip),%rsi        # 0x5555555567b0
   0x000055555555520f <+11>:    callq  0x555555555693 <strings_not_equal>
   0x0000555555555214 <+16>:    test   %eax,%eax
   0x0000555555555216 <+18>:    jne    0x55555555521d <phase_1+25>
   0x0000555555555218 <+20>:    add    $0x8,%rsp
   0x000055555555521c <+24>:    retq
   0x000055555555521d <+25>:    callq  0x55555555579f <explode_bomb>
   0x0000555555555222 <+30>:    jmp    0x555555555218 <phase_1+20>
End of assembler dump.
```

hi라는 단어를 쳐도 폭탄이 터지지 않고 phase_1에 멈춰있는 것을 볼 수 있다.

화살표를 보면 프로그램이 어디에 위치해 있는지 볼 수 있다.

<+11>에서 strings_not_equal 함수를 부른다.

<+16>에서 함수 return값이 들어갈 %eax 레지스터를 test하고

<+18>에서 결과가 같지 않으면

<+25>로 jump한다. (<phase_1+25>)이 부분에서 explode_bomb 함수가 실행되므로 이 함수가 작동되지 않도록 하는게 목표이다.

<+18>에서 두 레지스터의 값을 비교하는 방법은 다음과 같다.

Jne 뜻은 jump if not equal이란 뜻이다. ZF=0일 때 실행되는데,

<+16> test에서 and 연산을 한다. 즉 같으면 1 다르면 0이 return된다. 결국 문자열이 같아야 explode_bomb가 실행되지 않음을 알 수 있다.

```
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
   0x0000555555555693 <+0>:     push   %r12
   0x0000555555555695 <+2>:     push   %rbp
   0x0000555555555696 <+3>:     push   %rbx
   0x0000555555555697 <+4>:     mov    %rdi,%rbx
   0x000055555555569a <+7>:     mov    %rsi,%rbp
   0x000055555555569d <+10>:    callq  0x555555555676 <string_length>
   0x00005555555556a2 <+15>:    mov    %eax,%r12d
   0x00005555555556a5 <+18>:    mov    %rbp,%rdi
   0x00005555555556a8 <+21>:    callq  0x555555555676 <string_length>
   0x00005555555556ad <+26>:    mov    $0x1,%edx
   0x00005555555556b2 <+31>:    cmp    %eax,%r12d
   0x00005555555556b5 <+34>:    je     0x5555555556be <strings_not_equal+43>
   0x00005555555556b7 <+36>:    mov    %edx,%eax
   0x00005555555556b9 <+38>:    pop    %rbx
   0x00005555555556ba <+39>:    pop    %rbp
   0x00005555555556bb <+40>:    pop    %r12
   0x00005555555556bd <+42>:    retq
   0x00005555555556be <+43>:    movzbl (%rbx),%eax
   0x00005555555556c1 <+46>:    test   %al,%al
   0x00005555555556c3 <+48>:    je     0x5555555556ec <strings_not_equal+89>
   0x00005555555556c5 <+50>:    cmp    0x0(%rbp),%al
   0x00005555555556c8 <+53>:    jne    0x5555555556f3 <strings_not_equal+96>
   0x00005555555556ca <+55>:    add    $0x1,%rbx
   0x00005555555556ce <+59>:    add    $0x1,%rbp
   0x00005555555556d2 <+63>:    movzbl (%rbx),%eax
   0x00005555555556d5 <+66>:    test   %al,%al
   0x00005555555556d7 <+68>:    je     0x5555555556e5 <strings_not_equal+82>
   0x00005555555556d9 <+70>:    cmp    %al,0x0(%rbp)
   0x00005555555556dc <+73>:    je     0x5555555556ca <strings_not_equal+55>
   0x00005555555556de <+75>:    mov    $0x1,%edx
   0x00005555555556e3 <+80>:    jmp    0x5555555556b7 <strings_not_equal+36>
   0x00005555555556e5 <+82>:    mov    $0x0,%edx
   0x00005555555556ea <+87>:    jmp    0x5555555556b7 <strings_not_equal+36>
   0x00005555555556ec <+89>:    mov    $0x0,%edx
   0x00005555555556f1 <+94>:    jmp    0x5555555556b7 <strings_not_equal+36>
   0x00005555555556f3 <+96>:    mov    $0x1,%edx
```

Disas strings_not_equal 를 작동시킨 화면이다.

```
End of assembler dump.
(gdb) b strings_not_equal
Breakpoint 2 at 0x555555555693
(gdb) c
Continuing.
```

Strings_not_equal 함수에도 breakpoint를 두고 c(continue) 명령어를 이용해 함수를 디버깅한다.

```
(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
=> 0x0000555555555693 <+0>:     push   %r12
   0x0000555555555695 <+2>:     push   %rbp
   0x0000555555555696 <+3>:     push   %rbx
   0x0000555555555697 <+4>:     mov    %rdi,%rbx
   0x000055555555569a <+7>:     mov    %rsi,%rbp
   0x000055555555569d <+10>:    callq  0x555555555676 <string_length>
   0x00005555555556a2 <+15>:    mov    %eax,%r12d
   0x00005555555556a5 <+18>:    mov    %rbp,%rdi
   0x00005555555556a8 <+21>:    callq  0x555555555676 <string_length>
   0x00005555555556ad <+26>:    mov    $0x1,%edx
   0x00005555555556b2 <+31>:    cmp    %eax,%r12d
   0x00005555555556b5 <+34>:    je     0x5555555556be <strings_not_equal+43>
   0x00005555555556b7 <+36>:    mov    %edx,%eax
   0x00005555555556b9 <+38>:    pop    %rbx
   0x00005555555556ba <+39>:    pop    %rbp
   0x00005555555556bb <+40>:    pop    %r12
   0x00005555555556bd <+42>:    retq
   0x00005555555556be <+43>:    movzbl (%rbx),%eax
   0x00005555555556c1 <+46>:    test   %al,%al
   0x00005555555556c3 <+48>:    je     0x5555555556ec <strings_not_equal+89>
   0x00005555555556c5 <+50>:    cmp    0x0(%rbp),%al
   0x00005555555556c8 <+53>:    jne    0x5555555556f3 <strings_not_equal+96>
   0x00005555555556ca <+55>:    add    $0x1,%rbx
   0x00005555555556ce <+59>:    add    $0x1,%rbp
   0x00005555555556d2 <+63>:    movzbl (%rbx),%eax
   0x00005555555556d5 <+66>:    test   %al,%al
   0x00005555555556d7 <+68>:    je     0x5555555556e5 <strings_not_equal+82>
   0x00005555555556d9 <+70>:    cmp    %al,0x0(%rbp)
   0x00005555555556dc <+73>:    je     0x5555555556ca <strings_not_equal+55>
   0x00005555555556de <+75>:    mov    $0x1,%edx
   0x00005555555556e3 <+80>:    jmp    0x5555555556b7 <strings_not_equal+36>
   0x00005555555556e5 <+82>:    mov    $0x0,%edx
   0x00005555555556ea <+87>:    jmp    0x5555555556b7 <strings_not_equal+36>
   0x00005555555556ec <+89>:    mov    $0x0,%edx
   0x00005555555556f1 <+94>:    jmp    0x5555555556b7 <strings_not_equal+36>
   0x00005555555556f3 <+96>:    mov    $0x1,%edx
   0x00005555555556f8 <+101>:   jmp    0x5555555556b7 <strings_not_equal+36>
End of assembler dump.
```

Rdi 와 rsi 인자에 함수가 들어있는데, I r 명령어를 통해 정보를 볼 수 있다. 여기서 rdi와 rsi의 정보를 보면

```
End of assembler dump.
(gdb) i r
rax            0x5555557586a0   93824994346656
rbx            0x0      0
rcx            0x2      2
rdx            0x5555557586a0   93824994346656
rsi            0x5555555567b0   93824992241584
rdi            0x5555557586a0   93824994346656
rbp            0x0      0x0
rsp            0x7fffffffe488   0x7fffffffe488
r8             0x7ffff7ff7003   140737354100739
r9             0x0      0
r10            0x2      2
r11            0x246    582
r12            0x555555554fa0   93824992235424
r13            0x7fffffffe580   140737488348544
r14            0x0      0
r15            0x0      0
rip            0x555555555693   0x555555555693 <strings_not_equal>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/s $rdi
0x5555557586a0 <input_strings>: "hi"
(gdb) x/s $rsi
0x5555555567b0: "We have to stand with our North Korean allies."
```

rdi에는 처음에 입력한 문자열이, rsi에는 처음보는 문자열이 담겨있다는 것을 알 수 있다.

이제 우리는 phase_1 입력값으로

"We have to stand with our North Korean allies."

를 넣어야 한다는 사실을 알고있다.

D 명령어를 이용해 모든 breakpoint를 삭제하고 r명령어를 통해 프로그램을 실행시킨다.

우리가 알고 있는 정답을 입력값으로 넣으면

```
(gdb) r
Starting program: /home/std/howru0321/bomb20210054/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
We have to stand with our North Korean allies.
Phase 1 defused. How about the next one?
```

Phase_1 이 잘 해결 됨을 볼 수 있다.

<phase_2>

이제 phase_2에 breakpoint를 걸로 프로그램을 실행시켜보자. 할때마다 breakpoint 하지말고

나중에는 시작부터 모든 phase에 breakpoint를 넣어줘도 될 듯 하다. B phase_2는 생략하였다.

```
Phase 1 defused. How about the next one?
hi

Breakpoint 4, 0x0000555555555224 in phase_2 ()
(gdb) disas phase_2
Dump of assembler code for function phase_2:
=> 0x0000555555555224 <+0>:     push   %rbp
   0x0000555555555225 <+1>:     push   %rbx
   0x0000555555555226 <+2>:     sub    $0x28,%rsp
   0x000055555555522a <+6>:     mov    %fs:0x28,%rax
   0x0000555555555233 <+15>:    mov    %rax,0x18(%rsp)
   0x0000555555555238 <+20>:    xor    %eax,%eax
   0x000055555555523a <+22>:    mov    %rsp,%rsi
   0x000055555555523d <+25>:    callq  0x5555555557c5 <read_six_numbers>
   0x0000555555555242 <+30>:    cmpl   $0x1,(%rsp)
   0x0000555555555246 <+34>:    jne    0x555555555251 <phase_2+45>
   0x0000555555555248 <+36>:    mov    %rsp,%rbx
   0x000055555555524b <+39>:    lea    0x14(%rbx),%rbp
   0x000055555555524f <+43>:    jmp    0x555555555261 <phase_2+61>
   0x0000555555555251 <+45>:    callq  0x55555555579f <explode_bomb>
   0x0000555555555256 <+50>:    jmp    0x555555555248 <phase_2+36>
   0x0000555555555258 <+52>:    add    $0x4,%rbx
   0x000055555555525c <+56>:    cmp    %rbp,%rbx
   0x000055555555525f <+59>:    je     0x555555555271 <phase_2+77>
   0x0000555555555261 <+61>:    mov    (%rbx),%eax
   0x0000555555555263 <+63>:    add    %eax,%eax
   0x0000555555555265 <+65>:    cmp    %eax,0x4(%rbx)
   0x0000555555555268 <+68>:    je     0x555555555258 <phase_2+52>
   0x000055555555526a <+70>:    callq  0x55555555579f <explode_bomb>
   0x000055555555526f <+75>:    jmp    0x555555555258 <phase_2+52>
   0x0000555555555271 <+77>:    mov    0x18(%rsp),%rax
   0x0000555555555276 <+82>:    xor    %fs:0x28,%rax
   0x000055555555527f <+91>:    jne    0x555555555288 <phase_2+100>
   0x0000555555555281 <+93>:    add    $0x28,%rsp
   0x0000555555555285 <+97>:    pop    %rbx
   0x0000555555555286 <+98>:    pop    %rbp
   0x0000555555555287 <+99>:    retq
   0x0000555555555288 <+100>:   callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

Read_six_numbers에 breakpoint를 건다. 그리고 disas read_six_numbers를 하자

```
(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x00005555555557c5 <+0>:     sub    $0x8,%rsp
   0x00005555555557c9 <+4>:     mov    %rsi,%rdx
   0x00005555555557cc <+7>:     lea    0x4(%rsi),%rcx
   0x00005555555557d0 <+11>:    lea    0x14(%rsi),%rax
   0x00005555555557d4 <+15>:    push   %rax
   0x00005555555557d5 <+16>:    lea    0x10(%rsi),%rax
   0x00005555555557d9 <+20>:    push   %rax
   0x00005555555557da <+21>:    lea    0xc(%rsi),%r9
   0x00005555555557de <+25>:    lea    0x8(%rsi),%r8
   0x00005555555557e2 <+29>:    lea    0x119a(%rip),%rsi        # 0x555555556983
   0x00005555555557e9 <+36>:    mov    $0x0,%eax
   0x00005555555557ee <+41>:    callq  0x555555554ef0 <__isoc99_sscanf@plt>
   0x00005555555557f3 <+46>:    add    $0x10,%rsp
   0x00005555555557f7 <+50>:    cmp    $0x5,%eax
   0x00005555555557fa <+53>:    jle    0x555555555801 <read_six_numbers+60>
   0x00005555555557fc <+55>:    add    $0x8,%rsp
   0x0000555555555800 <+59>:    retq
   0x0000555555555801 <+60>:    callq  0x55555555579f <explode_bomb>
End of assembler dump.
```

<+36>을 보면 return값을 넣는 %eax에 0(0x0)을 넣는걸 볼 수 있다.

<+50>을 보면 %eax와 5(0x5)를 비교한다.

<+53>에서 만약 5보다 작거나 같으면(jle작거나 같을 때(ZF=1 or SF!=OF) 작동되는 구문이다.)

<+60>에 있는 <explode_bomb> 함수가 실행된다. 즉 입력값이 5개 이하면 폭탄이 터진다는 뜻이다.

```
  0x0000555555555258 <+52>:    add    $0x4,%rbx
  0x000055555555525c <+56>:    cmp    %rbp,%rbx
  0x000055555555525f <+59>:    je     0x555555555271 <phase_2+77>
  0x0000555555555261 <+61>:    mov    (%rbx),%eax
  0x0000555555555263 <+63>:    add    %eax,%eax
  0x0000555555555265 <+65>:    cmp    %eax,0x4(%rbx)
  0x0000555555555268 <+68>:    je     0x555555555258 <phase_2+52>
  0x000055555555526a <+70>:    callq  0x55555555579f <explode_bomb>
  0x000055555555526f <+75>:    jmp    0x555555555258 <phase_2+52>
```

Add 구문을 보면 eax+eax 연산을 한다. 이 뜻은 전 숫자의 두배를 한 6개의 숫자들을 넣으면 된다는 뜻이다.

```
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

1 2 4 8 16 32

를 넣으니 phase_2도 통과했음을 알 수 있다.

<phase_3>

Phase_3는 다음과 같다.

```
Dump of assembler code for function phase_3:
=> 0x000055555555528d <+0>:     sub    $0x18,%rsp
   0x0000555555555291 <+4>:     mov    %fs:0x28,%rax
   0x000055555555529a <+13>:    mov    %rax,0x8(%rsp)
   0x000055555555529f <+18>:    xor    %eax,%eax
   0x00005555555552a1 <+20>:    lea    0x4(%rsp),%rcx
   0x00005555555552a6 <+25>:    mov    %rsp,%rdx
   0x00005555555552a9 <+28>:    lea    0x16df(%rip),%rsi        # 0x55555555698f
   0x00005555555552b0 <+35>:    callq  0x555555554ef0 <__isoc99_sscanf@plt>
   0x00005555555552b5 <+40>:    cmp    $0x1,%eax
   0x00005555555552b8 <+43>:    jle    0x5555555552d3 <phase_3+70>
   0x00005555555552ba <+45>:    cmpl   $0x7,(%rsp)
   0x00005555555552be <+49>:    ja     0x55555555530b <phase_3+126>
   0x00005555555552c0 <+51>:    mov    (%rsp),%eax
   0x00005555555552c3 <+54>:    lea    0x1556(%rip),%rdx        # 0x555555556820
   0x00005555555552ca <+61>:    movslq (%rdx,%rax,4),%rax
   0x00005555555552ce <+65>:    add    %rdx,%rax
   0x00005555555552d1 <+68>:    jmpq   *%rax
   0x00005555555552d3 <+70>:    callq  0x55555555579f <explode_bomb>
   0x00005555555552d8 <+75>:    jmp    0x5555555552ba <phase_3+45>
   0x00005555555552da <+77>:    mov    $0x14b,%eax
   0x00005555555552df <+82>:    jmp    0x55555555531c <phase_3+143>
   0x00005555555552e1 <+84>:    mov    $0xbe,%eax
   0x00005555555552e6 <+89>:    jmp    0x55555555531c <phase_3+143>
   0x00005555555552e8 <+91>:    mov    $0x3d1,%eax
   0x00005555555552ed <+96>:    jmp    0x55555555531c <phase_3+143>
   0x00005555555552ef <+98>:    mov    $0x34b,%eax
   0x00005555555552f4 <+103>:   jmp    0x55555555531c <phase_3+143>
   0x00005555555552f6 <+105>:   mov    $0x275,%eax
   0x00005555555552fb <+110>:   jmp    0x55555555531c <phase_3+143>
   0x00005555555552fd <+112>:   mov    $0x44,%eax
   0x0000555555555302 <+117>:   jmp    0x55555555531c <phase_3+143>
   0x0000555555555304 <+119>:   mov    $0x2b4,%eax
   0x0000555555555309 <+124>:   jmp    0x55555555531c <phase_3+143>
   0x000055555555530b <+126>:   callq  0x55555555579f <explode_bomb>
   0x0000555555555310 <+131>:   mov    $0x0,%eax
   0x0000555555555315 <+136>:   jmp    0x55555555531c <phase_3+143>
   0x0000555555555317 <+138>:   mov    $0x9a,%eax
   0x000055555555531c <+143>:   cmp    %eax,0x4(%rsp)
---Type <return> to continue, or q <return> to quit---<return>
```

Phase_2와 같이 <+28>부분에 lea 가 있으므로 입력값의 형태가 저장되어있다고 보고, ni 명령어를 이용해서 <+28>까지 가서 보니

```
0x00005555555552a6 in phase_3 ()
(gdb) x/s $rip+0x16df
0x555555556985: " %d %d %d %d %d"
```

정수 5개를 입력 받음을 알 수 있었다.

```
0x00005555555552d1 <+68>:    jmpq   *%rax
0x00005555555552d3 <+70>:    callq  0x55555555579f <explode_bomb>
0x00005555555552d8 <+75>:    jmp    0x5555555552ba <phase_3+45>
0x00005555555552da <+77>:    mov    $0x14b,%eax
0x00005555555552df <+82>:    jmp    0x55555555531c <phase_3+143>
0x00005555555552e1 <+84>:    mov    $0xbe,%eax
0x00005555555552e6 <+89>:    jmp    0x55555555531c <phase_3+143>
0x00005555555552e8 <+91>:    mov    $0x3d1,%eax
0x00005555555552ed <+96>:    jmp    0x55555555531c <phase_3+143>
0x00005555555552ef <+98>:    mov    $0x34b,%eax
0x00005555555552f4 <+103>:   jmp    0x55555555531c <phase_3+143>
0x00005555555552f6 <+105>:   mov    $0x275,%eax
0x00005555555552fb <+110>:   jmp    0x55555555531c <phase_3+143>
0x00005555555552fd <+112>:   mov    $0x44,%eax
0x0000555555555302 <+117>:   jmp    0x55555555531c <phase_3+143>
0x0000555555555304 <+119>:   mov    $0x2b4,%eax
0x0000555555555309 <+124>:   jmp    0x55555555531c <phase_3+143>
0x000055555555530b <+126>:   callq  0x55555555579f <explode_bomb>
0x0000555555555310 <+131>:   mov    $0x0,%eax
0x0000555555555315 <+136>:   jmp    0x55555555531c <phase_3+143>
0x0000555555555317 <+138>:   mov    $0x9a,%eax
0x000055555555531c <+143>:   cmp    %eax,0x4(%rsp)
```

이 부분을 보면 <+68> jmpq *%rax 가 있다. 이 말은 첫 정수값 만큼 주소를 이동하라는 뜻인데, 결국 1이면 <+77>로, 2면 <+84>로 가라는 뜻이다. 여기서 eax에 값을 넣어 주는데, 최종적으로 가는 주소인<+143>에서 두번째 정수(0x4(%rsp))값과 eax값을 비교하는 것을 보아 위의 과정에서 eax에 넣어진 숫자들을 비교하는 것이다. 따라서 (첫번째 숫자, 두번째 숫자) 순서쌍의 경우의 수는

(1,331)

(2,190)

(3,997)

(4,843)

(5,629)

(6,68)

(7,692)

가 된다. 뒤의 세 숫자의 대한 조건은 없으므로 아무 숫자나 넣어도 될 듯하다.

```
That's number 2.  Keep going!
1 331 1 1 1

Breakpoint 3, 0x000055555555528d in phase_3 ()
(gdb) c
Continuing.
Halfway there!
```

잘 작동함을 볼 수 있다.

<phase_4>

```
(gdb) x/s $rip+0x15ec
0x55555555696c: "The bomb has blown up."
(gdb) ni
0x0000555555555384 in phase_4 ()
(gdb) ni
0x000055555555538d in phase_4 ()
(gdb) ni
0x0000555555555392 in phase_4 ()
(gdb) x/s $rip+0x15ec
0x55555555697e: " up."
(gdb) ni
0x0000555555555394 in phase_4 ()
(gdb) x/s $rip+0x15ec
0x555555556980: "p."
(gdb) ni
0x0000555555555399 in phase_4 ()
(gdb) x/s $rip+0x15ec
0x555555556985: " %d %d %d %d %d"
(gdb) x/s $rip+0x15eb
0x555555556984: "d %d %d %d %d %d"
(gdb) $x/s $rip+0x15ea
Undefined command: "$x".  Try "help".
(gdb) x/s $rip+0x15ea
0x555555556983: "%d %d %d %d %d %d"
(gdb) x/s $rip+0x15e9
0x555555556982: ""
(gdb) x/s $rip+0x15ed
0x555555556986: "%d %d %d %d %d"
```

위와 같은 논리로 입력값의 개수를 찾을 수 있다. 6개를 입력해야한다. 아마 모든 문제가 6개인걸 암시하는 것 같다.

```
0x00005555555553a3 in phase_4 ()
(gdb) x/d $rsp
0x7fffffffe480: 0
(gdb) ni
0x00005555555553a8 in phase_4 ()
(gdb) x/d $rsp
0x7fffffffe480: 1
```

Ni로 함수를 작동시키고, rsp에 내가 입력한 첫 값이 들어감을 확인한다.(1 1 1 1 1 1입력함)

```
   0x00005555555553ad <+45>:    cmpl   $0xe,(%rsp)
   0x00005555555553b1 <+49>:    jbe    0x5555555553b8 <phase_4+56>
   0x00005555555553b3 <+51>:    callq  0x55555555579f <explode_bomb>
=> 0x00005555555553b8 <+56>:    mov    $0xe,%edx
```

그리고 첫 값이 16이하여야 rsp-0xe 가 1보다 작거나 그 이하가 돼서 jbe를 만족해 <+51>에 있는 <explode_bomb>를 거치지 않고 바로<+56>으로 점프할 수 있다.(1<=16이여서 통과함)

이때 재귀를 돌면 처음에 ecx에 7이 들어간다. 그 밑의 줄이 돌지 않기 위해서는 7보다 작거타 같은 수가 첫 수로 나와야 한다. 계속 재귀를 돌다가 어느 순간에는 첫 입력값과 ecx가 2로 같아지면서 func4가 종료되고, eax는 4보다 작거나 같은 값이 되어서 폭탄이 터지지 않게 된다. 따라서 첫 값은 2가 되어야 한다.

```
0x00000000000013cd <+77>:    jne    0x13d6 <phase_4+86>
0x00000000000013cf <+79>:    cmpl   $0x4,0x4(%rsp)
0x00000000000013d4 <+84>:    je     0x13db <phase_4+91>
```

그리고 $rsp+0x4 즉 두번째 입력값과 비교했을 때 4와 동일하지 않으면 점프를 하지 않아 밑에 폭탄 함수로 가게 되어 폭탄이 터진다. 따라서 두번째 입력값은 4가 되어야 한다.
나머지 4개의 값들에 대한 제약은 없으므로 아무 값이나 입력하면 된다.

따라서 답은 2 4 1 1 1 1 이 된다.

<phase_5>

```
0x00005555555553f9 <+4>:     callq  0x555555555676 <string_length>
0x00005555555553fe <+9>:     cmp    $0x6,%eax
```

String_length 함수를 통해 내가 입력한 문자열의 길이를 리턴받고, 리턴값이 저장되어 있는 eax값과 6과 비교해서

```
0x0000555555555401 <+12>:    jne    0x555555555434 <phase_5+63>
0x0000555555555403 <+14>:    mov    %rbx,%rax
0x0000555555555406 <+17>:    lea    0x6(%rbx),%rdi
0x000055555555540a <+21>:    mov    $0x0,%ecx
0x000055555555540f <+26>:    lea    0x142a(%rip),%rsi        # 0x555555556840 <array.3415>
0x0000555555555416 <+33>:    movzbl (%rax),%edx
0x0000555555555419 <+36>:    and    $0xf,%edx
0x000055555555541c <+39>:    add    (%rsi,%rdx,4),%ecx
0x000055555555541f <+42>:    add    $0x1,%rax
0x0000555555555423 <+46>:    cmp    %rdi,%rax
0x0000555555555426 <+49>:    jne    0x555555555416 <phase_5+33>
0x0000555555555428 <+51>:    cmp    $0x29,%ecx
0x000055555555542b <+54>:    je     0x555555555432 <phase_5+61>
0x000055555555542d <+56>:    callq  0x55555555579f <explode_bomb>
0x0000555555555432 <+61>:    pop    %rbx
0x0000555555555433 <+62>:    retq
0x0000555555555434 <+63>:    callq  0x55555555579f <explode_bomb>
0x0000555555555439 <+68>:    jmp    0x555555555403 <phase_5+14>
nd of assembler dump.
```

6보다 크면 폭탄이 터진다. 즉 문자열크기는 6이하여야함을 알 수 있다.
그리고 각 array에 저장된 값들을 다 더해서 0x29가 나와야한다.

```
0x555555556840 <array.3415>:    0x00000002      0x0000000a      0x00000006      0x00000001
0x555555556850 <array.3415+16>: 0x0000000c      0x00000010      0x00000009      0x00000003
0x555555556860 <array.3415+32>: 0x00000004      0x00000007      0x0000000e      0x00000005
0x555555556870 <array.3415+48>: 0x0000000b      0x00000008      0x0000000f      0x0000000d
0x555555556880: 0x79206f53      0x7420756f      0x6b6e6968      0x756f7920
0x555555556890: 0x6e616320      0x6f747320      0x68742070      0x6f622065
```

배열은 다음과 같고, 7*5+6*1=41=0x29가 나와서 2번째 array값 1개, 9번째 array값 5개를 받으면 폭탄이 해체된다고 생각해서 답을

299999

로 입력했더니 폭탄이 해체되었음을 알 수 있었다.

<phase_6>

```
 0x00005555555554c5 <+138>:   mov    (%rsp,%rsi,4),%ecx
 0x00005555555554c8 <+141>:   mov    $0x1,%eax
 0x00005555555554cd <+146>:   lea    0x202d3c(%rip),%rdx        # 0x555555758210 <node1>
 0x00005555555554d4 <+153>:   cmp    $0x1,%ecx
 0x00005555555554d7 <+156>:   jg     0x5555555554ab <phase_6+112>
 0x00005555555554d9 <+158>:   jmp    0x5555555554b6 <phase_6+123>
```

Node1의 주소를 보자

각 node들의 값들을 볼 수 있다. 그런데 6번째 node가 없으므로 찾아가야한다.

```
0x555555758210 <node1>: 331     1       1433764384      21845
0x555555758220 <node2>: 315     2       1433764400      21845
0x555555758230 <node3>: 398     3       1433764416      21845
0x555555758240 <node4>: 930     4       1433764432      21845
0x555555758250 <node5>: 499     5       1433764112      21845
0x555555758260 <host_table>:    1431661033      21845   1431661059      21845
(gdb)
```

1 331->14b

2 315->13b

3 398->18e

4 930->3a2

5 499->1f3

eax에는 첫 숫자가, 내림차순 기준으로 그 다음 값은 rbx에저장되어 있다. 이 순으로 ni명령어를 통해 보면 6번째 수도 얻을 수 있다고 예측했다. 일단 알고있는 정보를 다 내림차순으로 나열하고 마지막에 6을 넣었다.

4 5 3 1 2 6

```
Breakpoint 5, 0x0000555555555534 in phase_6 ()
(gdb) x/d $eax
0x1be:  Cannot access memory at address 0x1be
(gdb) x/24d $rbx
0x555555758220 <node2>: 315     2       1433764112
```

Ni를 통해 계속 입력받아보면 어느 순간 우리가 보지 못한 값을 찾을 수 있다. 그 값은 0x1be=446이고 우리는 숫자들을 내림차순으로 정렬하여 입력하면 폭탄이 해체됨을 알 수 있다.

6 1be->446

4 5 6 3 1 2

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
We have to stand with our North Korean allies.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
1 331 1 1 1
Halfway there!
2 4 1 1 1 1
So you got that one.  Try this one.
299999
Good work!  On to the next...
4 5 6 3 1 2
Congratulations! You've defused the bomb!
[Inferior 1 (process 29765) exited normally]
```

최종적으로 폭탄이 해체된 모습이다.