

Lab Assignment L2: Defusing a Binary Bomb

20190084 권민재, CSED211

preface

Using [gdb-peda](#) with Intel syntax. The entire assembly codes which is not showing entire on my solution are inserted as an appendix.

phase_1

```
gdb-peda$ disas *phase_1
Dump of assembler code for function phase_1:
0x000055555555204 <+0>: sub    rsp,0x8
0x000055555555208 <+4>: lea    rsi,[rip+0x1691]        # 0x55555555568a0
0x00005555555520f <+11>: call   0x55555555793 <strings_not_equal>
0x000055555555214 <+16>: test   eax,eax
0x000055555555216 <+18>: jne     0x5555555521d <phase_1+25>
0x000055555555218 <+20>: add     rsp,0x8
0x00005555555521c <+24>: ret
0x00005555555521d <+25>: call   0x55555555589f <explode_bomb>
0x000055555555222 <+30>: jmp     0x55555555218 <phase_1+20>
End of assembler dump.
```

페이지 1에서는 단순히 사용자의 입력을 `strings_not_equal` 함수를 이용하여 비교한 뒤, 만약 사용자의 입력과 원하는 문자열이 다를 경우에 폭탄을 터뜨린다. `strings_not_equal` 함수의 첫번째 인자로는 `phase_1` 함수의 첫번째 인자가 그대로 전달되어 사용자의 입력이 전달되며, 비교하고자 하는 문자열은 `rsi`를 통해 전달된다. 그렇기 때문에, 이 문자열을 알아내기 위해서는 우리는 단순히 `0x55555555568a0`이 어떤 문자열인지 알아내거나, `strings_not_equal`이 호출될 때 `rsi`에 어떤 값이 들어있는지 확인하면 된다.

Solution 1. Read String from Address

```
gdb-peda$ x/s 0x55555555568a0
0x55555555568a0: "We have to stand with our North Korean allies."
```

Solution 2. Read RSI

```
gdb-peda$ b *0x5555555520f
Breakpoint 2 at 0x5555555520f
```

```
gdb-peda$ c
```

Continuing.

Welcome to my fiendish little bomb. You have 6 phases with which to blow yourself up. Have a nice day!

1

```
[-----registers-----]
RAX: 0x55555557586a0 --> 0x31 ('1')
RBX: 0x0
RCX: 0x1
RDX: 0x55555557586a0 --> 0x31 ('1')
RSI: 0x55555555568a0 ("We have to stand with our North Korean allies.")
RDI: 0x55555557586a0 --> 0x31 ('1')
RBP: 0x55555555566d0 (<__libc_csu_init>: push    r15)
RSP: 0x7fffffffef230 --> 0x5555555554fa0 (<_start>: xor     ebp,ebp)
RIP: 0x55555555520f (<phase_1+11>: call    0x555555555793 <strings_not_equal>)
R8  : 0x5555555759422 --> 0x0
R9  : 0x7ffff7fcd700 (0x00007ffff7fcd700)
R10 : 0x7ffff7fcd700 (0x00007ffff7fcd700)
R11 : 0x246
R12 : 0x5555555554fa0 (<_start>: xor     ebp,ebp)
R13 : 0x7fffffffef320 --> 0x1
R14 : 0x0
R15 : 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x5555555551ff <main+341>: call    0x5555555554f20 <exit@plt>
0x555555555204 <phase_1>: sub     rsp,0x8
0x555555555208 <phase_1+4>: lea     rsi,[rip+0x1691]          # 0x55555555568a0
=> 0x55555555520f <phase_1+11>: call    0x555555555793 <strings_not_equal>
0x555555555214 <phase_1+16>: test    eax,eax
0x555555555216 <phase_1+18>: jne     0x55555555521d <phase_1+25>
0x555555555218 <phase_1+20>: add     rsp,0x8
0x55555555521c <phase_1+24>: ret
Guessed arguments:
arg[0]: 0x55555557586a0 --> 0x31 ('1')
arg[1]: 0x55555555568a0 ("We have to stand with our North Korean allies.")
[-----stack-----]
0000 | 0x7fffffffef230 --> 0x5555555554fa0 (<_start>: xor     ebp,ebp)
0008 | 0x7fffffffef238 --> 0x555555555510a (<main+96>: call    0x5555555555a4a
<phase_defused>)
0016 | 0x7fffffffef240 --> 0x0
0024 | 0x7fffffffef248 --> 0x7ffff7a2d840 (<__libc_start_main+240>: mov
edi,eax)
0032 | 0x7fffffffef250 --> 0x1
0040 | 0x7fffffffef258 --> 0x7fffffffef328 --> 0x7fffffffef62e
("/home/beta/bomb/bomb")
```

```

0048 | 0x7fffffff260 --> 0x1f7ffcca0
0056 | 0x7fffffff268 --> 0x555555550aa (<main>: push rbx)
[-----]
]
Legend: code, data, rodata, value

Breakpoint 2, 0x00005555555520f in phase_1 ()

```

gdb-peda 를 이용하고 있기 때문에, break가 걸렸을 때 레지스터의 정보를 보여주고, RSI에 `We have to stand with our North Korean allies.` 가 담겨 있는 것을 볼 수 있다. gdb-peda 를 인텔 문법으로 사용할 때에는 해당 문장을 `Guessed arguments` 에서도 확인할 수 있으며, 기본 gdb 를 사용할 때에는 `i r` (info register)을 사용하면 된다.

Answer

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

phase_2

Parsing Input

read_six_numbers

```

# read_six_numbers
gdb-peda$ disas *read_six_numbers
Dump of assembler code for function read_six_numbers:
   0x00005555555558c5 <+0>: sub    rsp,0x8
   0x00005555555558c9 <+4>: mov    rdx,rsi
   0x00005555555558cc <+7>: lea    rcx,[rsi+0x4]
   0x00005555555558d0 <+11>: lea    rax,[rsi+0x14]
   0x00005555555558d4 <+15>: push   rax
   0x00005555555558d5 <+16>: lea    rax,[rsi+0x10]
   0x00005555555558d9 <+20>: push   rax
   0x00005555555558da <+21>: lea    r9,[rsi+0xc]
   0x00005555555558de <+25>: lea    r8,[rsi+0x8]
   0x00005555555558e2 <+29>: lea    rsi,[rip+0x115a]          # 0x555555556a43
   0x00005555555558e9 <+36>: mov    eax,0x0
   0x00005555555558ee <+41>: call   0x55555554ef0 <__isoc99_sscanf@plt>
   0x00005555555558f3 <+46>: add    rsp,0x10
   0x00005555555558f7 <+50>: cmp    eax,0x5
   0x00005555555558fa <+53>: jle    0x55555555901 <read_six_numbers+60>
   0x00005555555558fc <+55>: add    rsp,0x8
   0x0000555555555900 <+59>: ret
   0x0000555555555901 <+60>: call   0x5555555589f <explode_bomb>
End of assembler dump.

```

```
gdb-peda$ x/s 0x555555556a43
0x555555556a43: "%d %d %d %d %d %d"
```

함수 `read_six_numbers` 는 함수 `sscanf` 를 이용하여 사용자의 입력을 숫자로 변환하여 배열에 저장하는 함수이다. 인자로 `"%d %d %d %d %d %d"` , 사용자의 문자열 입력, 그리고 변환된 결과가 저장될 변수가 전해지는 부분에서 알 수 있다. 더불어, 이 함수 안에서 `sscanf` 의 반환값, 즉 정상적으로 입력된 숫자의 개수가 6개보다 작다면 폭탄이 터지므로 유의해야한다.

Check Number is Correct

```
# phase_2
0x000055555555226 <+2>: sub    rsp,0x28
0x00005555555522a <+6>: mov    rax,QWORD PTR fs:0x28
0x000055555555233 <+15>: mov    QWORD PTR [rsp+0x18],rax
0x000055555555238 <+20>: xor    eax,eax
0x00005555555523a <+22>: mov    rsi,rsp
0x00005555555523d <+25>: call   0x5555555558c5 <read_six_numbers>
0x000055555555242 <+30>: cmp    DWORD PTR [rsp],0x0
0x000055555555246 <+34>: js     0x555555555252 <phase_2+46>
0x000055555555248 <+36>: mov    ebx,0x1
0x00005555555524d <+41>: mov    rbp,rsp
0x000055555555250 <+44>: jmp    0x555555555263 <phase_2+63>
0x000055555555252 <+46>: call   0x55555555589f <explode_bomb>
0x000055555555257 <+51>: jmp    0x555555555248 <phase_2+36>
```

처음 `phase_2` 가 불리면, 우선 `read_six_numbers` 를 호출하여 사용자의 문자열 입력을 배열(`rsp+0x18`)에 담는다. 배열에 담은 후에, 배열의 첫번째 값이 0보다 작은지 검사하여 만약 0보다 작다면 `<phase_2+46>` 즉, `explode_bomb` 으로 점프하여 폭탄을 터뜨린다.

이를 통해 페이지 2에서는 우선 사용자의 입력을 파싱하여 올바른 입력이라면 다음 루틴으로 넘어가고, 아니라면 폭탄을 터뜨린다는 것을 알 수 있다.

Main Routine

```
# phase_2
0x000055555555259 <+53>: add    rbx,0x1
0x00005555555525d <+57>: cmp    rbx,0x6
0x000055555555261 <+61>: je     0x555555555276 <phase_2+82>
0x000055555555263 <+63>: mov    eax,ebx
0x000055555555265 <+65>: add    eax,DWORD PTR [rbp+rbx*4-0x4]
0x000055555555269 <+69>: cmp    DWORD PTR [rbp+rbx*4+0x0],eax
0x00005555555526d <+73>: je     0x555555555259 <phase_2+53>
0x00005555555526f <+75>: call   0x55555555589f <explode_bomb>
0x000055555555274 <+80>: jmp    0x555555555259 <phase_2+53>
```

phase_2 <+53> 에서 `rbx` 에 1을 더하고, `phase_2 <+73>`, `phase_2 <+80>` 에서 `phase_2 <+53>` 로 점프한다는 점에서, 일종의 반복문이라고 생각할 수 있다. 이때, `[rbp+rbx*4+0x0]` 는 `rbx` 가 1부터 5일 때까지 `[rbp+rbx*4-0x4]` 에 `rbx` 를 더한 값과 비교당하고 있다. 이때, 값이 다르다면, <+75>로 점프해서 폭탄이 터지게 된다.

Solution

Main Routine에서, 반복문을 통해 우리가 입력해야하는 숫자들은 0으로 시작해서 이전 숫자보다 `rbx` 만큼 더 큰 숫자 6개임을 알 수 있다. 그러므로 우리가 입력해야 하는 수 a_n ($0 \leq n < 6, n \in \mathbb{Z}$)은 아래와 같은 규칙을 따르는 수임을 알 수 있다.

$$a_n = \sum_{k=0}^n k$$

Answer

0 1 3 6 10 15

phase_3

Parsing Input

```
0x0000555555552a6 <+20>: lea    rcx,[rsp+0xf]
0x0000555555552ab <+25>: lea    rdx,[rsp+0x10]
0x0000555555552b0 <+30>: lea    r8,[rsp+0x14]
0x0000555555552b5 <+35>: lea    rsi,[rip+0x163a]          # 0x5555555568f6
0x0000555555552bc <+42>: call   0x555555554ef0 <__isoc99_sscanf@plt>
0x0000555555552c1 <+47>: cmp    eax,0x2
0x0000555555552c4 <+50>: jle    0x555555552e5 <phase_3+83>
0x0000555555552c6 <+52>: cmp    DWORD PTR [rsp+0x10],0x7
0x0000555555552cb <+57>: ja     0x555555553da <phase_3+328>
0x0000555555552d1 <+63>: mov    eax,DWORD PTR [rsp+0x10]
0x0000555555552d5 <+67>: lea    rdx,[rip+0x1634]          # 0x555555556910
0x0000555555552dc <+74>: movsxd rax,DWORD PTR [rdx+rax*4]
0x0000555555552e0 <+78>: add    rax,rdx
0x0000555555552e3 <+81>: jmp    rax
0x0000555555552e5 <+83>: call   0x55555555589f <explode_bomb>
```

```
gdb-peda$ x/s 0x5555555568f6
0x5555555568f6: "%d %c %d"
```

phase_3에서는 우선 `sscanf`를 이용하여 사용자의 입력을 파싱한다. `0x5555555568f6`가 `"%d %c %d"`이라는 점에서, 정수, 문자, 정수를 입력받한다는 사실을 알 수 있으며, 이들은 `rsp+0xf`, `rsp+0xf`, `rsp+0xf`에 저장된다. 이렇게 입력을 받고 난 이후, 만약 `sscanf`의 반환값, 즉 정상적으로 입력받은 인자가 2보다 작게 된다면 <+83>으로 점프하여 폭탄이 터지게 되는 것을 알 수 있다. 이를 통해 페이지 2에서는 정수, 문자, 정수의 입력을 받아야 한다는 사실을 알 수 있다.

Switch

```

0x0000555555552c6 <+52>: cmp    DWORD PTR [rsp+0x10],0x7
0x0000555555552cb <+57>: ja     0x555555553da <phase_3+328>
0x0000555555552d1 <+63>: mov    eax,DWORD PTR [rsp+0x10]
0x0000555555552d5 <+67>: lea    rdx,[rip+0x1634]          # 0x555555556910
0x0000555555552dc <+74>: movsxd rax,DWORD PTR [rdx+rax*4]
0x0000555555552e0 <+78>: add    rax,rdx
0x0000555555552e3 <+81>: jmp    rax
0x0000555555552e5 <+83>: call   0x5555555589f <explode_bomb>
0x0000555555552ea <+88>: jmp    0x555555552c6 <phase_3+52>

```

```

gdb-peda$ x/8wx 0x555555556910
0x555555556910: 0xffffe9dc 0xffffe9fe 0xffffea20 0xffffea42
0x555555556920: 0xffffea5e 0xffffea79 0xffffea94 0xffffeaaf

```

<+81>을 통해, `rax`로 점프하는, 즉 점프 테이블을 가지는 switch-case 구조임을 알 수 있다. <+52>, <+57>을 통해 이 스위치 문은 7개의 케이스를 가지고 있음을 추론할 수 있으며, 각 케이스는 사용자로부터 입력받은 첫번째 숫자에 의해 분기되며, 이는 각각 0~6에 대응된다.

Case

```

0x0000555555552ec <+90>: mov    eax,0x67
0x0000555555552f1 <+95>: cmp    DWORD PTR [rsp+0x14],0x386
0x0000555555552f9 <+103>: je     0x555555553e4 <phase_3+338>
0x0000555555552ff <+109>: call   0x5555555589f <explode_bomb>
0x000055555555304 <+114>: mov    eax,0x67
0x000055555555309 <+119>: jmp    0x555555553e4 <phase_3+338>

```

우리는 여기서 사용자가 첫번째로 0을 입력했을 때의 케이스에 대해 분석할 것이다. 나머지 케이스는 구조는 동일하고 숫자만 다르기 때문에 생략할 것이다. 각 케이스에서, `eax`에 특정한 숫자 A를 넣은 다음, `DWORD PTR [rsp+0x14]` 사용자가 세번째로 넣은 숫자가 특정 숫자 B와 다르다면 폭탄을 터뜨린다. 아니면 <+338>로 점프한다. 이 케이스 0에서 A는 `0x67`, B는 `0x386`이다.

Check Character

```

0x0000555555553e4 <+338>: cmp    BYTE PTR [rsp+0xf],al
0x0000555555553e8 <+342>: je     0x555555553ef <phase_3+349>
0x0000555555553ea <+344>: call   0x5555555589f <explode_bomb>

```

위 케이스에서 `eax`에 특정 숫자를 저장하였는데, 이는 이 곳에서 쓰이게 된다. `eax`의 한 바이트와 사용자가 두번째에 입력한 글자가 다를 경우에는 폭탄이 터지게 된다.

Solution

위의 분석을 바탕으로, 이 페이지에는 여러개의 답이 존재하는 것을 알 수 있다. 이 중에서 가능한, 처음에 0을 선택했을 때 할 수 있는 하나의 답은 아래와 같다.

Answer

0 g 902

phase_4

Parsing Input

```
0x000055555555456 <+20>: mov    rcx, rsp
0x000055555555459 <+23>: lea    rdx, [rsp+0x4]
0x00005555555545e <+28>: lea    rsi, [rip+0x15ea]          # 0x5555555556a4f
0x000055555555465 <+35>: call   0x555555554ef0 <__isoc99_sscanf@plt>
0x00005555555546a <+40>: cmp    eax, 0x2
0x00005555555546d <+43>: jne    0x5555555547a <phase_4+56>
0x00005555555546f <+45>: mov    eax, DWORD PTR [rsp]
0x000055555555472 <+48>: sub    eax, 0x2
0x000055555555475 <+51>: cmp    eax, 0x2
0x000055555555478 <+54>: jbe    0x5555555547f <phase_4+61>
0x00005555555547a <+56>: call   0x55555555589f <explode_bomb>
```

```
gdb-peda$ x/s 0x5555555556a4f
0x5555555556a4f: "%d %d"
```

페이지 4에서도 sscanf를 이용해서 사용자의 입력을 파싱한다. 정수를 2개 입력받는 모습을 볼 수 있다. 이때, 정상적으로 입력받은 정수의 개수가 2개가 아니라면 폭탄이 터지며, 나중에 입력받은 숫자가 4보다 클 경우에도 폭탄이 터진다.

Calling func4

```
0x00005555555547f <+61>: mov    esi, DWORD PTR [rsp]
0x000055555555482 <+64>: mov    edi, 0x7
0x000055555555487 <+69>: call   0x555555555409 <func4>
0x00005555555548c <+74>: cmp    DWORD PTR [rsp+0x4], eax
0x000055555555490 <+78>: je     0x555555555497 <phase_4+85>
0x000055555555492 <+80>: call   0x55555555589f <explode_bomb>
```

사용자의 입력을 파싱 한 후에, 7과 사용자가 뒤에 입력한 정수를 인자로 func4를 호출하고, 그 결과가 사용자가 처음에 입력한 정수와 다르다면 폭탄이 터지는 것을 볼 수 있다.

Solution

```
gdb-peda$ disas *func4
Dump of assembler code for function func4:
0x000055555555409 <+0>: mov    eax, 0x0
0x00005555555540e <+5>: test   edi, edi
```

```

0x000055555555410 <+7>: jle      0x55555555419 <func4+16>
0x000055555555412 <+9>: mov      eax,esi
0x000055555555414 <+11>: cmp      edi,0x1
0x000055555555417 <+14>: jne      0x5555555541b <func4+18>
0x000055555555419 <+16>: repz ret
0x00005555555541b <+18>: push     r12
0x00005555555541d <+20>: push     rbp
0x00005555555541e <+21>: push     rbx
0x00005555555541f <+22>: mov      r12d,esi
0x000055555555422 <+25>: mov      ebx,edi
0x000055555555424 <+27>: lea      edi,[rdi-0x1]
0x000055555555427 <+30>: call     0x55555555409 <func4>
0x00005555555542c <+35>: lea      ebp,[rax+r12*1]
0x000055555555430 <+39>: lea      edi,[rbx-0x2]
0x000055555555433 <+42>: mov      esi,r12d
0x000055555555436 <+45>: call     0x55555555409 <func4>
0x00005555555543b <+50>: add      eax,ebp
0x00005555555543d <+52>: pop      rbx
0x00005555555543e <+53>: pop      rbp
0x00005555555543f <+54>: pop      r12
0x000055555555441 <+56>: ret

```

End of assembler dump.

func4를 디스어셈블해보면 재귀 호출을 이용하여 매우 복잡한 루틴을 가지고 있는 것을 볼 수 있다. 하지만 func4를 우리가 분석할 필요는 없다. 왜냐하면 우리는 굳이 분석해보지 않아도 입력에 따른 func4의 반환값을 확인할 수 있고, 그 과정은 아래와 같다.

```

gdb-peda$ c
Continuing.
Halfway there!
2 2

[-----registers-----]
RAX: 0x42 ('B')
RBX: 0x0
RCX: 0x20 (' ')
RDX: 0x7fffffffef220 --> 0x200000002
RSI: 0x2
RDI: 0x1
RBP: 0x5555555566d0 (<__libc_csu_init>: push r15)
RSP: 0x7fffffffef220 --> 0x200000002
RIP: 0x5555555548c (<phase_4+74>: cmp DWORD PTR [rsp+0x4],eax)
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x7ffff7b846a0 --> 0x2000200020002
R12: 0x555555554fa0 (<_start>: xor ebp,ebp)

```



```

R13: 0x7fffffff320 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-----code-----]
    0x5555555547f <phase_4+61>: mov     esi,DWORD PTR [rsp]
    0x55555555482 <phase_4+64>: mov     edi,0x7
    0x55555555487 <phase_4+69>: call    0x55555555409 <func4>
=> 0x5555555548c <phase_4+74>: cmp     DWORD PTR [rsp+0x4],eax
    0x55555555490 <phase_4+78>: je      0x55555555497 <phase_4+85>
    0x55555555492 <phase_4+80>: call    0x5555555589f <explode_bomb>
    0x55555555497 <phase_4+85>: mov     rax,QWORD PTR [rsp+0x8]
    0x5555555549c <phase_4+90>: xor     rax,QWORD PTR fs:0x28
[-----stack-----]
0000 | 0x7fffffff220 --> 0x200000002
0008 | 0x7fffffff228 --> 0xa6ef4d90ffdb9b00
0016 | 0x7fffffff230 --> 0x555555554fa0 (<_start>: xor     ebp,ebp)
0024 | 0x7fffffff238 --> 0x555555555164 (<main+186>: call    0x55555555a4a
<phase_defused>)
0032 | 0x7fffffff240 --> 0x0
0040 | 0x7fffffff248 --> 0x7ffff7a2d840 (<__libc_start_main+240>: mov
edi,eax)
0048 | 0x7fffffff250 --> 0x1
0056 | 0x7fffffff258 --> 0x7fffffff328 --> 0x7fffffff62e
("/home/beta/bomb/bomb")
[-----]
]
Legend: code, data, rodata, value

Breakpoint 6, 0x00005555555548c in phase_4 ()
gdb-peda$

```

위의 과정에서, 두번째 숫자로 2를 입력 했을 때 `func4` 는 `0x42` 를 반환한다는 사실을 `eax` 를 통해 볼 수 있다. 그래서 답은 아래와 같다.

Answer

66 2

phase_5

Parsing Input

string_length

```
gdb-peda$ disas *string_length
Dump of assembler code for function string_length:
   0x000055555555776 <+0>: cmp     BYTE PTR [rdi],0x0
   0x000055555555779 <+3>: je      0x5555555578d <string_length+23>
   0x00005555555577b <+5>: mov     rdx,rdi
   0x00005555555577e <+8>: add     rdx,0x1
   0x000055555555782 <+12>: mov     eax,edx
   0x000055555555784 <+14>: sub     eax,edi
   0x000055555555786 <+16>: cmp     BYTE PTR [rdx],0x0
   0x000055555555789 <+19>: jne     0x5555555577e <string_length+8>
   0x00005555555578b <+21>: repz ret
   0x00005555555578d <+23>: mov     eax,0x0
   0x000055555555792 <+28>: ret
End of assembler dump.
```

이 함수에서는 우선 첫 문자가 null 문자라면 0을 반환한다. 아니라면 반복문을 돌면서 null 문자가 나오지 않을 때까지 eax를 업데이트 하고, 그 값을 반환한다. 이를 통해 문자열의 길이를 알 수 있다.

Check Length of Input

```
0x0000555555554c9 <+24>: call    0x55555555776 <string_length>
0x0000555555554ce <+29>: cmp     eax,0x6
0x0000555555554d1 <+32>: jne     0x55555555528 <phase_5+119>
... (생략)
0x000055555555528 <+119>: call    0x5555555589f <explode_bomb>
```

phase_5에서는 `string_length`를 우선 호출하여 입력을 정제하는데, 이때 `string_length`의 인자는 phase_5의 첫번째 인자와 같다. 이를 통해 사용자의 입력의 길이를 확인하고, 만약 그 길이가 6이 아니면 폭탄을 터뜨린다.

Set String

```
0x0000555555554d3 <+34>: mov     eax,0x0
0x0000555555554d8 <+39>: lea     rcx,[rip+0x1451]          # 0x555555556930
<array>
0x0000555555554df <+46>: movzx   edx,BYTE PTR [rbx+rax*1]
0x0000555555554e3 <+50>: and     edx,0xf
0x0000555555554e6 <+53>: movzx   edx,BYTE PTR [rcx+rdx*1]
0x0000555555554ea <+57>: mov     BYTE PTR [rsp+rax*1+0x1],dl
0x0000555555554ee <+61>: add     rax,0x1
0x0000555555554f2 <+65>: cmp     rax,0x6
0x0000555555554f6 <+69>: jne     0x555555554df <phase_5+46>
```

```
gdb-peda$ x/16bx 0x555555556930
0x555555556930 <array>: 0x6d 0x61 0x64 0x75 0x69 0x65 0x72 0x73
0x555555556938 <array+8>: 0x6e 0x66 0x6f 0x74 0x76 0x62 0x79 0x6c
```

`rax`가 60이 아닐때까지 `rax`에 1이 더해지면서 `<+46>`으로 점프하는 것을 보았을 때, 이것은 반복문이라고 할 수 있다. 이 반복문에서는 사용자가 입력한 문자열을 한 문자씩 순회하면서 이것을 0xF와 and 연산을 통해 마스킹 한 값을 인덱스로 `0x555555556930`에 존재하는 배열로 부터 1바이트의 값을 가져오고, 이것을 스택에 존재하는 배열의 값으로 업데이트 하는 작업을 진행한다.

Compare String

```
0x00005555555554f8 <+71>: mov     BYTE PTR [rsp+0x7],0x0
0x00005555555554fd <+76>: lea     rdi,[rsp+0x1]
0x0000555555555502 <+81>: lea     rsi,[rip+0x13f6]          # 0x55555555568ff
0x0000555555555509 <+88>: call    0x555555555793 <strings_not_equal>
0x000055555555550e <+93>: test    eax,eax
0x0000555555555510 <+95>: jne     0x55555555552f <phase_5+126>
```

```
gdb-peda$ x/s 0x5555555568ff
0x5555555568ff: "oilers"
```

문자열이 세팅된 이후에, `strings_not_equal`을 이용하여 문자열이 `0x5555555568ff`에 존재하는 문자열인 `oilers`와 동일하지 비교한다. 만약 다르다면, 폭탄을 터뜨린다.

Solution

우선 `0x555555556930` 배열에 담긴 문자를 인덱스로 나타내면 아래와 같이 표현할 수 있다.

```
00: 'm', 01: 'a', 02: 'd', 03: 'u', 04: 'i', 05: 'e', 06: 'r', 07: 's', 08:
    'n', 09: 'f', 10: 'o', 11: 't', 12: 'v', 13: 'b', 14: 'y', 15: 'l'
```

이때, `oilers`의 글자는 각각 `0xA, 0x4, 0xF, 0x5, 0x6, 0x7`에 해당하고, 사용자가 입력한 문자열의 각 글자에서 1바이트만 가져오기 때문에 사용자는 `0x6a, 0x64, 0x6F, 0x65, 0x66, 0x67`에 해당하는 글자들을 순서대로 입력하여 폭탄을 해체할 수 있다.

Answer

jdoefg

phase_6

Parsing Input

```
0x0000555555555555 <+26>: mov    r12, rsp
0x0000555555555558 <+29>: mov    rsi, r12
0x000055555555555b <+32>: call   0x5555555558c5 <read_six_numbers>
```

페이지 6이 시작될 때, 우선 사용자의 입력을 read_six_numbers를 이용하여 6개의 정수로 변환해서 배열 [rsp+0]에 저장한다.

Check Unique Number

반복문이 이중으로 존재하기 때문에, 메인 루틴과 서브 루틴 두 부분으로 나눠서 생각해볼 것이다.

Main Routine

```
0x0000555555555560 <+37>: mov    r13d, 0x0
0x0000555555555566 <+43>: jmp    0x55555555558d <phase_6+82>
0x0000555555555568 <+45>: call   0x555555555589f <explode_bomb>
0x000055555555556d <+50>: jmp    0x55555555559c <phase_6+97>
# ... (Subroutine)
0x0000555555555589 <+78>: add    r12, 0x4
0x000055555555558d <+82>: mov    rbp, r12
0x0000555555555590 <+85>: mov    eax, DWORD PTR [r12]
0x0000555555555594 <+89>: sub    eax, 0x1
0x0000555555555597 <+92>: cmp    eax, 0x5
0x000055555555559a <+95>: ja     0x555555555568 <phase_6+45>
0x000055555555559c <+97>: add    r13d, 0x1
0x00005555555555a0 <+101>: cmp    r13d, 0x6
0x00005555555555a4 <+105>: je     0x5555555555db <phase_6+160>
0x00005555555555a6 <+107>: mov    ebx, r13d
0x00005555555555a9 <+110>: jmp    0x555555555577 <phase_6+60>
```

사용자가 입력한 숫자에 대해 반복문을 돌면서, [r12]가 6보다 크다면 폭탄을 터트린다. 그리고 r13d는 반복 카운터로서 작용하고 있는데, 이것이 6이 되었을 때 반복문을 탈출하게 된다. 이후 ebx에 r13d의 값을 저장하고 서브루틴에 들어간다. 서브루틴이 종료된 이후에는 r12에 4가 더해지는, 즉 배열에서 다음 인덱스의 주소가 들어가게 되고, 이것은 rbp에 전달되게 된다.

Subroutine

```
0x000055555555556f <+52>: add    ebx, 0x1
0x0000555555555572 <+55>: cmp    ebx, 0x5
0x0000555555555575 <+58>: jg     0x555555555589 <phase_6+78>
0x0000555555555577 <+60>: movsxd rax, ebx
0x000055555555557a <+63>: mov    eax, DWORD PTR [rsp+rax*4]
0x000055555555557d <+66>: cmp    DWORD PTR [rbp+0x0], eax
0x0000555555555580 <+69>: jne    0x55555555556f <phase_6+52>
0x0000555555555582 <+71>: call   0x555555555589f <explode_bomb>
0x0000555555555587 <+76>: jmp    0x55555555556f <phase_6+52>
```

서브 루틴에서는 사용자가 입력한 숫자 배열에서, 현재 지정된 숫자에 대해 같은 숫자가 있는지 검사하고, 만약 있다면 폭탄을 터트린다.

메인 루틴과 서브루틴을 종합해보았을 때, 이 부분은 사용자가 입력한 숫자 배열에 대해 중복된 숫자가 있는지 확인하는 부분이라고 말할 수 있고, 만약 중복된 숫자가 있다면 폭탄이 터지게 된다.

Set Node Array

```
0x000055555555cd <+146>: lea    rdx,[rip+0x202c3c]    # 0x555555758210
<node1>
```

```
gdb-peda$ x/10gx 0x555555758210
0x555555758210 <node1>: 0x000000010000028a  0x0000555555758220
0x555555758220 <node2>: 0x00000002000002da  0x0000555555758230
0x555555758230 <node3>: 0x0000000300000053  0x0000555555758240
0x555555758240 <node4>: 0x0000000400000218  0x0000555555758250
0x555555758250 <node5>: 0x000000050000028e  0x0000555555758110
gdb-peda$ x/2gx 0x0000555555758110
0x555555758110 <node6>: 0x00000006000002d2  0x0000000000000000
```

gdb 상에서, 0x555555758210 라는 주소에 대해 node1 이라는 심볼이 있는 것을 확인 할 수 있었고, 이를 바탕으로 우선 해당 주소의 데이터를 x/10gx 0x555555758210 와 같은 명령어로 확인해보았을 때, 아래와 같은 링크드 리스트로 구성되어 있는 것을 확인할 수 있었다.

링크드 리스트가 존재한다는 사실을 바탕으로, 이 부분도 메인루틴과 서브루틴으로 나누어서 분석을 진행할 것이다.

Main Routine

```
# ...(subroutine)
0x000055555555b6 <+123>: mov    QWORD PTR [rsp+rsi*8+0x20],rdx
0x000055555555bb <+128>: add    rsi,0x1
0x000055555555bf <+132>: cmp    rsi,0x6
0x000055555555c3 <+136>: je     0x55555555e2 <phase_6+167>
0x000055555555c5 <+138>: mov    ecx,DWORD PTR [rsp+rsi*4]
0x000055555555c8 <+141>: mov    eax,0x1
0x000055555555cd <+146>: lea    rdx,[rip+0x202c3c]    # 0x555555758210
<node1>
0x000055555555d4 <+153>: cmp    ecx,0x1
0x000055555555d7 <+156>: jg     0x55555555ab <phase_6+112>
0x000055555555d9 <+158>: jmp    0x55555555b6 <phase_6+123>
0x000055555555db <+160>: mov    esi,0x0
0x000055555555e0 <+165>: jmp    0x55555555c5 <phase_6+138>
```

여기서 메인 루틴은 <+160> 부터 시작된다. ecx에 사용자가 입력한 숫자가 반복적으로 들어가며, eax에 1이 세팅되고, rdx에 이 세팅된다. 이때, ecx가 1보다 크다면, 즉 사용자가 입력한 숫자가 이 반복 세포에서 1보다 크다면 서브 루틴에 진입하고, 아니라면 <+123>으로 점프하게 된다. 여기서 rdx가 스택에 존재하는 node의 배열에 할당되며, rsi에 1이 더해진다. 만약 rsi가 6이 된다면 이 루틴은 종료된다.

Subroutine

```
0x00005555555555ab <+112>: mov    rdx,QWORD PTR [rdx+0x8]
0x00005555555555af <+116>: add    eax,0x1
0x00005555555555b2 <+119>: cmp    eax,ecx
0x00005555555555b4 <+121>: jne    0x5555555555ab <phase_6+112>
```

이 서브루틴에서는 반복적으로 rdx에 다음 노드의 주소를 할당하면서 ecx, 즉 사용자가 입력한 숫자와 반복한 횟수가 같아질 때 까지 반복하고 탈출한다.

메인 루틴과 서브루틴의 분석을 종합해봤을 때, 이 루틴은 스택에 존재하는 사용자가 입력한 정수만큼의 깊이에 존재하는 node를 순서대로 node의 배열에 넣고자 하는 작업이다.

Relocation Nodes

```
0x00005555555555e2 <+167>: mov    rbx,QWORD PTR [rsp+0x20]
0x00005555555555e7 <+172>: mov    rax,QWORD PTR [rsp+0x28]
0x00005555555555ec <+177>: mov    QWORD PTR [rbx+0x8],rax
0x00005555555555f0 <+181>: mov    rdx,QWORD PTR [rsp+0x30]
0x00005555555555f5 <+186>: mov    QWORD PTR [rax+0x8],rdx
0x00005555555555f9 <+190>: mov    rax,QWORD PTR [rsp+0x38]
0x00005555555555fe <+195>: mov    QWORD PTR [rdx+0x8],rax
0x0000555555555602 <+199>: mov    rdx,QWORD PTR [rsp+0x40]
0x0000555555555607 <+204>: mov    QWORD PTR [rax+0x8],rdx
0x000055555555560b <+208>: mov    rax,QWORD PTR [rsp+0x48]
0x0000555555555610 <+213>: mov    QWORD PTR [rdx+0x8],rax
0x0000555555555614 <+217>: mov    QWORD PTR [rax+0x8],0x0
0x000055555555561c <+225>: mov    ebp,0x5
0x0000555555555621 <+230>: jmp    0x55555555562c <phase_6+241>
```

스택에 있는 node 배열에 존재하는 node의 순서대로 이 함수의 지역 변수가 될 수 있는 레지스터에 순서대로 node의 주소를 복사하고, 지역 변수의 순서대로 각 node의 next를 갱신한다.

Check Nodes

```
0x0000555555555623 <+232>: mov    rbx,QWORD PTR [rbx+0x8]
0x0000555555555627 <+236>: sub    ebp,0x1
0x000055555555562a <+239>: je     0x55555555563d <phase_6+258>
0x000055555555562c <+241>: mov    rax,QWORD PTR [rbx+0x8]
0x0000555555555630 <+245>: mov    eax,DWORD PTR [rax]
0x0000555555555632 <+247>: cmp    DWORD PTR [rbx],eax
0x0000555555555634 <+249>: jge    0x555555555623 <phase_6+232>
0x0000555555555636 <+251>: call   0x55555555589f <explode_bomb>
0x000055555555563b <+256>: jmp    0x555555555623 <phase_6+232>
```

지역 변수의 첫 노드부터 탐색을 시작해서 크기가 큰 것 부터 작은 것 까지 차례대로 정렬되어 있는지 확인하고, 그렇지 않다면 폭탄이 터진다.

Solution

위의 분석을 요약해보면 사용자는 1에서 6까지의 숫자를 중복되지 않게 입력해야되고, 링크드 리스트에서 각 숫자 만큼의 깊이에 있는 노드가 사용자가 입력한 순서대로 재배치되게 된다. 이때, 이 재배치된 노드가 깊이가 얕을 수록 더 큰 값을 가지도록 숫자를 입력하는 것이 목표이다.

원래 존재하던 노드의 정보는 아래와 같다.

```
gdb-peda$ x/10gx 0x555555758210
0x555555758210 <node1>: 0x000000010000028a 0x0000555555758220
0x555555758220 <node2>: 0x00000002000002da 0x0000555555758230
0x555555758230 <node3>: 0x0000000300000053 0x0000555555758240
0x555555758240 <node4>: 0x0000000400000218 0x0000555555758250
0x555555758250 <node5>: 0x000000050000028e 0x0000555555758110
gdb-peda$ x/2gx 0x0000555555758110
0x555555758110 <node6>: 0x00000006000002d2 0x0000000000000000
```

이를 크기 순으로 다시 배치하게 되면 2, 6, 5, 1, 4, 3이 되어야 하기 때문에 답은 아래와 같다.

Answer

2 6 5 1 4 3

Secret Phase

phase_defused: Secret Entrance

```
gdb-peda$ disas *phase_defused
Dump of assembler code for function phase_defused:
   0x000055555555a4a <+0>: sub    rsp,0x78
   0x000055555555a4e <+4>: mov    rax,QWORD PTR fs:0x28
   0x000055555555a57 <+13>: mov    QWORD PTR [rsp+0x68],rax
   0x000055555555a5c <+18>: xor    eax,eax
   0x000055555555a5e <+20>: cmp    DWORD PTR [rip+0x202c27],0x6          #
0x55555575868c <num_input_strings>
   0x000055555555a65 <+27>: je     0x55555555a7c <phase_defused+50>
   0x000055555555a67 <+29>: mov    rax,QWORD PTR [rsp+0x68]
   0x000055555555a6c <+34>: xor    rax,QWORD PTR fs:0x28
   0x000055555555a75 <+43>: jne    0x55555555aea <phase_defused+160>
   0x000055555555a77 <+45>: add    rsp,0x78
   0x000055555555a7b <+49>: ret
   0x000055555555a7c <+50>: lea    rcx,[rsp+0xc]
   0x000055555555a81 <+55>: lea    rdx,[rsp+0x8]
   0x000055555555a86 <+60>: lea    r8,[rsp+0x10]
   0x000055555555a8b <+65>: lea    rsi,[rip+0x1007]          # 0x555555556a99
   0x000055555555a92 <+72>: lea    rdi,[rip+0x202cf7]       # 0x555555758790
<input_strings+240>
```

```

0x000055555555a99 <+79>: call 0x555555554ef0 <__isoc99_sscanf@plt>
0x000055555555a9e <+84>: cmp  eax,0x3
0x000055555555aa1 <+87>: je   0x555555555ab1 <phase_defused+103>
0x000055555555aa3 <+89>: lea  rdi,[rip+0xf2e]      # 0x55555555569d8
0x000055555555aaa <+96>: call 0x555555554e30 <puts@plt>
0x000055555555aaf <+101>: jmp  0x555555555a67 <phase_defused+29>
0x000055555555ab1 <+103>: lea  rdi,[rsp+0x10]
0x000055555555ab6 <+108>: lea  rsi,[rip+0xfe5]      # 0x5555555556aa2
0x000055555555abd <+115>: call 0x555555555793 <strings_not_equal>
0x000055555555ac2 <+120>: test eax,eax
0x000055555555ac4 <+122>: jne  0x555555555aa3 <phase_defused+89>
0x000055555555ac6 <+124>: lea  rdi,[rip+0xeab]      # 0x5555555556978
0x000055555555acd <+131>: call 0x555555554e30 <puts@plt>
0x000055555555ad2 <+136>: lea  rdi,[rip+0xec7]      # 0x55555555569a0
0x000055555555ad9 <+143>: call 0x555555554e30 <puts@plt>
0x000055555555ade <+148>: mov  eax,0x0
0x000055555555ae3 <+153>: call 0x55555555569c <secret_phase>
0x000055555555ae8 <+158>: jmp  0x555555555aa3 <phase_defused+89>
0x000055555555aea <+160>: call 0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.

```

```

gdb-peda$ x/s 0x55555556a99
0x55555556a99: "%d %d %s"

```

```

gdb-peda$ x/s 0x555555758790
0x555555758790 <input_strings+240>: "66 2"

```

이 함수를 뜯어보면 뜬금없이 sscanf를 통해 입력을 처리하고 있는 모습을 볼 수 있다. 해당 sscanf가 처리하고 있는 포맷 스트링을 뜯어보면 "%d %d %s"와 같고, 입력을 받아오는 곳의 주소를 살펴보면 "66 2"라는 데이터가 있음을 알 수 있다. 이것은 페이지 4의 정답이며, 이를 통해 페이지 4의 정답을 입력할 때 추가적으로 문자열을 받고 있는 것을 알 수 있다.

```

gdb-peda$ x/s 0x55555556aa2
0x55555556aa2: "DrEvil"

```

이때, strings_not_equal에서 비교하고 있는 문자열의 주소인 0x55555556aa2를 확인해보면, 그것이 "DrEvil"임을 알 수 있고, 페이지 4에서 아래와 같은 답안을 입력하면 페이지 6이 끝나고 secret_phase에 들어갈 수 있다. 이것은 num_input_strings를 통해 현재까지 입력된 답안의 개수를 체크하고 있기 때문이다.

Answer

Input 66 2 DrEvil when phase 4.

secret_phase

```

gdb-peda$ disas *secret_phase

```


Dump of assembler code for function secret_phase:

```
0x000055555555569c <+0>: push    rbx
0x000055555555569d <+1>: call   0x555555555906 <read_line>
0x00005555555556a2 <+6>: mov     edx,0xa
0x00005555555556a7 <+11>: mov     esi,0x0
0x00005555555556ac <+16>: mov     rdi,rax
0x00005555555556af <+19>: call   0x555555554ed0 <strtol@plt>
0x00005555555556b4 <+24>: mov     rbx,rax
0x00005555555556b7 <+27>: lea     eax,[rax-0x1]
0x00005555555556ba <+30>: cmp     eax,0x3e8
0x00005555555556bf <+35>: ja      0x5555555556ec <secret_phase+80>
0x00005555555556c1 <+37>: mov     esi,ebx
0x00005555555556c3 <+39>: lea     rdi,[rip+0x202a66]          # 0x5555555758130
<n1>
0x00005555555556ca <+46>: call   0x55555555565d <fun7>
0x00005555555556cf <+51>: cmp     eax,0x5
0x00005555555556d2 <+54>: je      0x5555555556d9 <secret_phase+61>
0x00005555555556d4 <+56>: call   0x55555555589f <explode_bomb>
0x00005555555556d9 <+61>: lea     rdi,[rip+0x11f0]          # 0x55555555568d0
0x00005555555556e0 <+68>: call   0x555555554e30 <puts@plt>
0x00005555555556e5 <+73>: call   0x555555555a4a <phase_defused>
0x00005555555556ea <+78>: pop     rbx
0x00005555555556eb <+79>: ret
0x00005555555556ec <+80>: call   0x55555555589f <explode_bomb>
0x00005555555556f1 <+85>: jmp     0x5555555556c1 <secret_phase+37>
```

End of assembler dump.

secret_phase의 기본적인 구조는 단순하다. read_line을 통해 숫자를 입력받은 다음, strtol을 이용하여 해당 문자열을 십진수 숫자로 변환한다. 그리고 만약 그 숫자가 1001 보다 큰 경우에는 폭탄을 터뜨린다. 그리고, n1의 주소와 사용자가 입력한 숫자를 인자로 가지고 fun7을 실행시킨 결과가 5가 되도록 하면 secret phase를 처리할 수 있다. 그러면 fun7을 살펴보자.

```
gdb-peda$ p n1
$15 = 0x24
```

fun7

gdb-peda\$ disas *fun7

Dump of assembler code for function fun7:

```
0x000055555555565d <+0>: test    rdi,rdi
0x0000555555555660 <+3>: je      0x555555555696 <fun7+57>
0x0000555555555662 <+5>: sub     rsp,0x8
0x0000555555555666 <+9>: mov     edx,DWORD PTR [rdi]
0x0000555555555668 <+11>: cmp     edx,esi
0x000055555555566a <+13>: jg      0x55555555567a <fun7+29>
0x000055555555566c <+15>: mov     eax,0x0
0x0000555555555671 <+20>: cmp     edx,esi
```

```

0x000055555555673 <+22>: jne    0x55555555687 <fun7+42>
0x000055555555675 <+24>: add    rsp,0x8
0x000055555555679 <+28>: ret
0x00005555555567a <+29>: mov    rdi,QWORD PTR [rdi+0x8]
0x00005555555567e <+33>: call   0x5555555565d <fun7>
0x000055555555683 <+38>: add    eax,eax
0x000055555555685 <+40>: jmp    0x55555555675 <fun7+24>
0x000055555555687 <+42>: mov    rdi,QWORD PTR [rdi+0x10]
0x00005555555568b <+46>: call   0x5555555565d <fun7>
0x000055555555690 <+51>: lea    eax,[rax+rax*1+0x1]
0x000055555555694 <+55>: jmp    0x55555555675 <fun7+24>
0x000055555555696 <+57>: mov    eax,0xffffffff
0x00005555555569b <+62>: ret

```

End of assembler dump.

우선, 입력된 `rdi` 가 없는 값이라면 함수는 종료된다. 이후, `DWORD PTR [rdi]` 가 `esi` (사용자가 입력한 값)보다 크다면, `rdi` 를 `QWORD PTR [rdi+0x8]` 로 수정해서 함수를 재귀 호출한다. 해당 호출 결과를 2배한 값을 반환한다. 만약 `DWORD PTR [rdi]` 가 사용자가 입력한 값과 다르다면, `rdi` 를 `QWORD PTR [rdi+0x10]` 로 수정해서 함수를 재귀 호출한다. 해당 호출 결과에 2배해서 1을 더한 값을 반환한다. 이 외의 경우 0을 반환한다.

이때 우리는 `fun7`의 첫번째 인자의 자료형이 구조체일 것이라고 추론해볼 수 있다. 처음 `DWORD PTR`에 값을 가지고, `rdi+0x8`와 `rdi+0x10`에 각각 다른 구조체 변수의 주소를 가지고 있는 형식이라고 추론 가능하다. 이 점을 바탕으로, `n1`을 시작으로 어떻게 연결된 변수가 존재하는지 확인해 볼 수 있다.

```

gdb-peda$ i ad n1
Symbol "n1" is at 0x555555758130 in a file compiled without debugging.

```

```

gdb-peda$ x/3gx 0x555555758130
0x555555758130 <n1>:  0x0000000000000024  0x0000555555758150
0x555555758140 <n1+16>: 0x0000555555758170

gdb-peda$ x/3gx 0x0000555555758150
0x555555758150 <n21>: 0x0000000000000008  0x00005555557581d0
0x555555758160 <n21+16>: 0x0000555555758190

gdb-peda$ x/3gx 0x0000555555758170
0x555555758170 <n22>: 0x0000000000000032  0x00005555557581b0
0x555555758180 <n22+16>: 0x00005555557581f0

gdb-peda$ x/3gx 0x00005555557581d0
0x5555557581d0 <n31>: 0x0000000000000006  0x0000555555758030
0x5555557581e0 <n31+16>: 0x0000555555758090

gdb-peda$ x/3gx 0x0000555555758190
0x555555758190 <n32>: 0x0000000000000016  0x00005555557580b0
0x5555557581a0 <n32+16>: 0x0000555555758070

```

```

gdb-peda$ x/3gx 0x00005555557581b0
0x5555557581b0 <n33>: 0x000000000000002d 0x0000555555758010
0x5555557581c0 <n33+16>: 0x00005555557580d0

gdb-peda$ x/3gx 0x00005555557581f0
0x5555557581f0 <n34>: 0x000000000000006b 0x0000555555758050
0x555555758200 <n34+16>: 0x00005555557580f0

gdb-peda$ x/3gx 0x0000555555758030
0x555555758030 <n41>: 0x0000000000000001 0x0000000000000000
0x555555758040 <n41+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x0000555555758090
0x555555758090 <n42>: 0x0000000000000007 0x0000000000000000
0x5555557580a0 <n42+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x00005555557580b0
0x5555557580b0 <n43>: 0x0000000000000014 0x0000000000000000
0x5555557580c0 <n43+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x0000555555758070
0x555555758070 <n44>: 0x0000000000000023 0x0000000000000000
0x555555758080 <n44+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x0000555555758010
0x555555758010 <n45>: 0x0000000000000028 0x0000000000000000
0x555555758020 <n45+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x00005555557580d0
0x5555557580d0 <n46>: 0x000000000000002f 0x0000000000000000
0x5555557580e0 <n46+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x0000555555758050
0x555555758050 <n47>: 0x0000000000000063 0x0000000000000000
0x555555758060 <n47+16>: 0x0000000000000000

gdb-peda$ x/3gx 0x00005555557580f0
0x5555557580f0 <n48>: 0x000000000000003e9 0x0000000000000000
0x555555758100 <n48+16>: 0x0000000000000000

```

위와 같이 연결된 변수들이 존재하는 것을 확인 할 수 있었다. DWORD PTR를 사용하는데, 구조체의 멤버 중에 값이 8 바이트를 차지하는 것은 메모리 align을 맞추기 위함이라고 해석할 수 있다.

Solution 1.

위와 같은 구조에서 5를 만들기 위해서는, $(((0 \times 2) + 1) \times 2) \times 2 + 1$ 과 같은 과정을 통해 만들 수 있을 것이다. 해당 과정을 거치기 위해서 우리가 입력해야 하는 수는 0x24보다 크면서, 0x32보다는 작고, 0x2d 보다는 크면서 0x2f와 동일해야 하는 값이 되어야 한다. 그래서 우리가 입력해야 하는 값은 47이다.

Solution 2.

입력된 수와 구조체의 값이 같으면 0을 반환하고, 입력된 숫자보다 작으면 재귀적으로 구한 결과에 2를 곱한 수를 반환하고, 아니라면 재귀적으로 구한 결과에 2를 곱해서 1을 더한 결과를 반환하는 이러한 구조체 구조는, 이진탐색 트리와 같은 것이며, 이때 반환되는 값은 입력된 수와 같은 값을 가지는 노드가 자신의 층에서 n 번째로 존재할 때 n 이다. ($0 \leq n, n \in \mathbb{Z}$) 우리는 위에서 모든 노드를 구했고, 노드의 변수 이름에는 $n_{(\text{깊이})(\text{인덱스})}$ 라는 규칙이 존재한다. 우리가 원하는 반환값은 5이기 때문에, 4번째 깊이에서 여섯번째 노드인 n_{46} 의 값, 0x2f가 답이다.

Solution

47

Final Answer

solution.txt

```
We have to stand with our North Korean allies.  
0 1 3 6 10 15  
0 g 902  
66 2 DrEvil  
jdoefg  
2 6 5 1 4 3  
47
```

Appendix

Assembly Codes

phase_1

```

gdb-peda$ disas *phase_1
Dump of assembler code for function phase_1:
   0x000055555555204 <+0>: sub    rsp,0x8
   0x000055555555208 <+4>: lea    rsi,[rip+0x1691]          # 0x5555555568a0
   0x00005555555520f <+11>: call   0x55555555793 <strings_not_equal>
   0x000055555555214 <+16>: test   eax,eax
   0x000055555555216 <+18>: jne    0x5555555521d <phase_1+25>
   0x000055555555218 <+20>: add    rsp,0x8
   0x00005555555521c <+24>: ret
   0x00005555555521d <+25>: call   0x5555555589f <explode_bomb>
   0x000055555555222 <+30>: jmp    0x55555555218 <phase_1+20>

End of assembler dump.

```

phase_2

```

gdb-peda$ disas *phase_2
Dump of assembler code for function phase_2:
   0x000055555555224 <+0>: push    rbp
   0x000055555555225 <+1>: push    rbx
   0x000055555555226 <+2>: sub     rsp,0x28
   0x00005555555522a <+6>: mov     rax,QWORD PTR fs:0x28
   0x000055555555233 <+15>: mov     QWORD PTR [rsp+0x18],rax
   0x000055555555238 <+20>: xor     eax,eax
   0x00005555555523a <+22>: mov     rsi,rsp
   0x00005555555523d <+25>: call    0x555555558c5 <read_six_numbers>
   0x000055555555242 <+30>: cmp     DWORD PTR [rsp],0x0
   0x000055555555246 <+34>: js      0x55555555252 <phase_2+46>
   0x000055555555248 <+36>: mov     ebx,0x1
   0x00005555555524d <+41>: mov     rbp,rsp
   0x000055555555250 <+44>: jmp     0x55555555263 <phase_2+63>
   0x000055555555252 <+46>: call    0x5555555589f <explode_bomb>
   0x000055555555257 <+51>: jmp     0x55555555248 <phase_2+36>
   0x000055555555259 <+53>: add     rbx,0x1
   0x00005555555525d <+57>: cmp     rbx,0x6
   0x000055555555261 <+61>: je      0x55555555276 <phase_2+82>
   0x000055555555263 <+63>: mov     eax,ebx
   0x000055555555265 <+65>: add     eax,DWORD PTR [rbp+rbx*4-0x4]
   0x000055555555269 <+69>: cmp     DWORD PTR [rbp+rbx*4+0x0],eax
   0x00005555555526d <+73>: je      0x55555555259 <phase_2+53>
   0x00005555555526f <+75>: call    0x5555555589f <explode_bomb>
   0x000055555555274 <+80>: jmp     0x55555555259 <phase_2+53>
   0x000055555555276 <+82>: mov     rax,QWORD PTR [rsp+0x18]
   0x00005555555527b <+87>: xor     rax,QWORD PTR fs:0x28
   0x000055555555284 <+96>: jne     0x5555555528d <phase_2+105>
   0x000055555555286 <+98>: add     rsp,0x28
   0x00005555555528a <+102>: pop     rbx
   0x00005555555528b <+103>: pop     rbp

```

```
0x00005555555528c <+104>: ret
0x00005555555528d <+105>: call 0x55555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

phase_3

```
gdb-peda$ disas *phase_3
```

Dump of assembler code for function phase_3:

```
0x000055555555292 <+0>: sub    rsp,0x28
0x000055555555296 <+4>: mov    rax,QWORD PTR fs:0x28
0x00005555555529f <+13>: mov    QWORD PTR [rsp+0x18],rax
0x0000555555552a4 <+18>: xor    eax,eax
0x0000555555552a6 <+20>: lea    rcx,[rsp+0xf]
0x0000555555552ab <+25>: lea    rdx,[rsp+0x10]
0x0000555555552b0 <+30>: lea    r8,[rsp+0x14]
0x0000555555552b5 <+35>: lea    rsi,[rip+0x163a]          # 0x5555555568f6
0x0000555555552bc <+42>: call   0x55555554ef0 <__isoc99_sscanf@plt>
0x0000555555552c1 <+47>: cmp    eax,0x2
0x0000555555552c4 <+50>: jle     0x555555552e5 <phase_3+83>
0x0000555555552c6 <+52>: cmp    DWORD PTR [rsp+0x10],0x7
0x0000555555552cb <+57>: ja      0x555555553da <phase_3+328>
0x0000555555552d1 <+63>: mov    eax,DWORD PTR [rsp+0x10]
0x0000555555552d5 <+67>: lea    rdx,[rip+0x1634]          # 0x555555556910
0x0000555555552dc <+74>: movsxd rax,DWORD PTR [rdx+rax*4]
0x0000555555552e0 <+78>: add    rax,rdx
0x0000555555552e3 <+81>: jmp     rax
0x0000555555552e5 <+83>: call   0x5555555589f <explode_bomb>
0x0000555555552ea <+88>: jmp     0x555555552c6 <phase_3+52>
0x0000555555552ec <+90>: mov    eax,0x67
0x0000555555552f1 <+95>: cmp    DWORD PTR [rsp+0x14],0x386
0x0000555555552f9 <+103>: je      0x555555553e4 <phase_3+338>
0x0000555555552ff <+109>: call   0x5555555589f <explode_bomb>
0x000055555555304 <+114>: mov    eax,0x67
0x000055555555309 <+119>: jmp     0x555555553e4 <phase_3+338>
0x00005555555530e <+124>: mov    eax,0x68
0x000055555555313 <+129>: cmp    DWORD PTR [rsp+0x14],0x2cf
0x00005555555531b <+137>: je      0x555555553e4 <phase_3+338>
0x000055555555321 <+143>: call   0x5555555589f <explode_bomb>
0x000055555555326 <+148>: mov    eax,0x68
0x00005555555532b <+153>: jmp     0x555555553e4 <phase_3+338>
0x000055555555330 <+158>: mov    eax,0x74
0x000055555555335 <+163>: cmp    DWORD PTR [rsp+0x14],0x244
0x00005555555533d <+171>: je      0x555555553e4 <phase_3+338>
0x000055555555343 <+177>: call   0x5555555589f <explode_bomb>
0x000055555555348 <+182>: mov    eax,0x74
0x00005555555534d <+187>: jmp     0x555555553e4 <phase_3+338>
0x000055555555352 <+192>: mov    eax,0x67
```

```

0x000055555555357 <+197>: cmp     DWORD PTR [rsp+0x14],0x76
0x00005555555535c <+202>: je      0x555555553e4 <phase_3+338>
0x000055555555362 <+208>: call   0x55555555589f <explode_bomb>
0x000055555555367 <+213>: mov     eax,0x67
0x00005555555536c <+218>: jmp     0x555555553e4 <phase_3+338>
0x00005555555536e <+220>: mov     eax,0x6a
0x000055555555373 <+225>: cmp     DWORD PTR [rsp+0x14],0x38b
0x00005555555537b <+233>: je      0x555555553e4 <phase_3+338>
0x00005555555537d <+235>: call   0x55555555589f <explode_bomb>
0x000055555555382 <+240>: mov     eax,0x6a
0x000055555555387 <+245>: jmp     0x555555553e4 <phase_3+338>
0x000055555555389 <+247>: mov     eax,0x6b
0x00005555555538e <+252>: cmp     DWORD PTR [rsp+0x14],0xfd
0x000055555555396 <+260>: je      0x555555553e4 <phase_3+338>
0x000055555555398 <+262>: call   0x55555555589f <explode_bomb>
0x00005555555539d <+267>: mov     eax,0x6b
0x0000555555553a2 <+272>: jmp     0x555555553e4 <phase_3+338>
0x0000555555553a4 <+274>: mov     eax,0x6e
0x0000555555553a9 <+279>: cmp     DWORD PTR [rsp+0x14],0x126
0x0000555555553b1 <+287>: je      0x555555553e4 <phase_3+338>
0x0000555555553b3 <+289>: call   0x55555555589f <explode_bomb>
0x0000555555553b8 <+294>: mov     eax,0x6e
0x0000555555553bd <+299>: jmp     0x555555553e4 <phase_3+338>
0x0000555555553bf <+301>: mov     eax,0x64
0x0000555555553c4 <+306>: cmp     DWORD PTR [rsp+0x14],0x3bc
0x0000555555553cc <+314>: je      0x555555553e4 <phase_3+338>
0x0000555555553ce <+316>: call   0x55555555589f <explode_bomb>
0x0000555555553d3 <+321>: mov     eax,0x64
0x0000555555553d8 <+326>: jmp     0x555555553e4 <phase_3+338>
0x0000555555553da <+328>: call   0x55555555589f <explode_bomb>
0x0000555555553df <+333>: mov     eax,0x72
0x0000555555553e4 <+338>: cmp     BYTE PTR [rsp+0xf],al
0x0000555555553e8 <+342>: je      0x555555553ef <phase_3+349>
0x0000555555553ea <+344>: call   0x55555555589f <explode_bomb>
0x0000555555553ef <+349>: mov     rax,QWORD PTR [rsp+0x18]
0x0000555555553f4 <+354>: xor     rax,QWORD PTR fs:0x28
0x0000555555553fd <+363>: jne     0x55555555404 <phase_3+370>
0x0000555555553ff <+365>: add     rsp,0x28
0x000055555555403 <+369>: ret
0x000055555555404 <+370>: call   0x555555554e50 <__stack_chk_fail@plt>

```

End of assembler dump.

phase_4

```
gdb-peda$ disas *phase_4
```

Dump of assembler code for function phase_4:

```
0x000055555555442 <+0>: sub    rsp,0x18
0x000055555555446 <+4>: mov    rax,QWORD PTR fs:0x28
0x00005555555544f <+13>: mov    QWORD PTR [rsp+0x8],rax
0x000055555555454 <+18>: xor    eax,eax
0x000055555555456 <+20>: mov    rcx,rsp
0x000055555555459 <+23>: lea    rdx,[rsp+0x4]
0x00005555555545e <+28>: lea    rsi,[rip+0x15ea]          # 0x555555556a4f
0x000055555555465 <+35>: call   0x555555554ef0 <__isoc99_sscanf@plt>
0x00005555555546a <+40>: cmp    eax,0x2
0x00005555555546d <+43>: jne    0x5555555547a <phase_4+56>
0x00005555555546f <+45>: mov    eax,DWORD PTR [rsp]
0x000055555555472 <+48>: sub    eax,0x2
0x000055555555475 <+51>: cmp    eax,0x2
0x000055555555478 <+54>: jbe    0x5555555547f <phase_4+61>
0x00005555555547a <+56>: call   0x55555555589f <explode_bomb>
0x00005555555547f <+61>: mov    esi,DWORD PTR [rsp]
0x000055555555482 <+64>: mov    edi,0x7
0x000055555555487 <+69>: call   0x555555555409 <func4>
0x00005555555548c <+74>: cmp    DWORD PTR [rsp+0x4],eax
0x000055555555490 <+78>: je     0x555555555497 <phase_4+85>
0x000055555555492 <+80>: call   0x55555555589f <explode_bomb>
0x000055555555497 <+85>: mov    rax,QWORD PTR [rsp+0x8]
0x00005555555549c <+90>: xor    rax,QWORD PTR fs:0x28
0x0000555555554a5 <+99>: jne    0x5555555554ac <phase_4+106>
0x0000555555554a7 <+101>: add    rsp,0x18
0x0000555555554ab <+105>: ret
0x0000555555554ac <+106>: call   0x555555554e50 <__stack_chk_fail@plt>
```

End of assembler dump.

phase_5

```
gdb-peda$ disas *phase_5
```

Dump of assembler code for function phase_5:

```
0x0000555555554b1 <+0>: push   rbx
0x0000555555554b2 <+1>: sub    rsp,0x10
0x0000555555554b6 <+5>: mov    rbx,rdi
0x0000555555554b9 <+8>: mov    rax,QWORD PTR fs:0x28
0x0000555555554c2 <+17>: mov    QWORD PTR [rsp+0x8],rax
0x0000555555554c7 <+22>: xor    eax,eax
0x0000555555554c9 <+24>: call   0x555555555776 <string_length>
0x0000555555554ce <+29>: cmp    eax,0x6
0x0000555555554d1 <+32>: jne    0x555555555528 <phase_5+119>
```



```

0x0000555555554d3 <+34>: mov    eax,0x0
0x0000555555554d8 <+39>: lea    rcx,[rip+0x1451]          # 0x555555556930
<array>
0x0000555555554df <+46>: movzx  edx,BYTE PTR [rbx+rax*1]
0x0000555555554e3 <+50>: and    edx,0xf
0x0000555555554e6 <+53>: movzx  edx,BYTE PTR [rcx+rdx*1]
0x0000555555554ea <+57>: mov    BYTE PTR [rsp+rax*1+0x1],dl
0x0000555555554ee <+61>: add    rax,0x1
0x0000555555554f2 <+65>: cmp    rax,0x6
0x0000555555554f6 <+69>: jne    0x555555554df <phase_5+46>
0x0000555555554f8 <+71>: mov    BYTE PTR [rsp+0x7],0x0
0x0000555555554fd <+76>: lea    rdi,[rsp+0x1]
0x000055555555502 <+81>: lea    rsi,[rip+0x13f6]          # 0x5555555568ff
0x000055555555509 <+88>: call   0x55555555793 <strings_not_equal>
0x00005555555550e <+93>: test   eax,eax
0x000055555555510 <+95>: jne    0x5555555552f <phase_5+126>
0x000055555555512 <+97>: mov    rax,QWORD PTR [rsp+0x8]
0x000055555555517 <+102>: xor    rax,QWORD PTR fs:0x28
0x000055555555520 <+111>: jne    0x55555555536 <phase_5+133>
0x000055555555522 <+113>: add    rsp,0x10
0x000055555555526 <+117>: pop    rbx
0x000055555555527 <+118>: ret
0x000055555555528 <+119>: call   0x5555555589f <explode_bomb>
0x00005555555552d <+124>: jmp    0x555555554d3 <phase_5+34>
0x00005555555552f <+126>: call   0x5555555589f <explode_bomb>
0x000055555555534 <+131>: jmp    0x55555555512 <phase_5+97>
0x000055555555536 <+133>: call   0x555555554e0 <__stack_chk_fail@plt>
End of assembler dump.

```

phase_6

```

gdb-peda$ disas *phase_6
Dump of assembler code for function phase_6:
0x00005555555553b <+0>: push    r13
0x00005555555553d <+2>: push    r12
0x00005555555553f <+4>: push    rbp
0x000055555555540 <+5>: push    rbx
0x000055555555541 <+6>: sub     rsp,0x68
0x000055555555545 <+10>: mov     rax,QWORD PTR fs:0x28
0x00005555555554e <+19>: mov     QWORD PTR [rsp+0x58],rax
0x000055555555553 <+24>: xor     eax,eax
0x000055555555555 <+26>: mov     r12,rsp
0x000055555555558 <+29>: mov     rsi,r12
0x00005555555555b <+32>: call    0x555555558c5 <read_six_numbers>
0x000055555555560 <+37>: mov     r13d,0x0
0x000055555555566 <+43>: jmp     0x5555555558d <phase_6+82>
0x000055555555568 <+45>: call    0x5555555589f <explode_bomb>
0x00005555555556d <+50>: jmp     0x5555555559c <phase_6+97>

```

```

0x000055555555556f <+52>: add    ebx,0x1
0x0000555555555572 <+55>: cmp    ebx,0x5
0x0000555555555575 <+58>: jg     0x555555555589 <phase_6+78>
0x0000555555555577 <+60>: movsxd rax,ebx
0x000055555555557a <+63>: mov    eax,DWORD PTR [rsp+rax*4]
0x000055555555557d <+66>: cmp    DWORD PTR [rbp+0x0],eax
0x0000555555555580 <+69>: jne    0x55555555556f <phase_6+52>
0x0000555555555582 <+71>: call   0x555555555589f <explode_bomb>
0x0000555555555587 <+76>: jmp    0x55555555556f <phase_6+52>
0x0000555555555589 <+78>: add    r12,0x4
0x000055555555558d <+82>: mov    rbp,r12
0x0000555555555590 <+85>: mov    eax,DWORD PTR [r12]
0x0000555555555594 <+89>: sub    eax,0x1
0x0000555555555597 <+92>: cmp    eax,0x5
0x000055555555559a <+95>: ja     0x555555555568 <phase_6+45>
0x000055555555559c <+97>: add    r13d,0x1
0x00005555555555a0 <+101>: cmp    r13d,0x6
0x00005555555555a4 <+105>: je     0x5555555555db <phase_6+160>
0x00005555555555a6 <+107>: mov    ebx,r13d
0x00005555555555a9 <+110>: jmp    0x555555555577 <phase_6+60>
0x00005555555555ab <+112>: mov    rdx,QWORD PTR [rdx+0x8]
0x00005555555555af <+116>: add    eax,0x1
0x00005555555555b2 <+119>: cmp    eax,ecx
0x00005555555555b4 <+121>: jne    0x5555555555ab <phase_6+112>
0x00005555555555b6 <+123>: mov    QWORD PTR [rsp+rsi*8+0x20],rdx
0x00005555555555bb <+128>: add    rsi,0x1
0x00005555555555bf <+132>: cmp    rsi,0x6
0x00005555555555c3 <+136>: je     0x5555555555e2 <phase_6+167>
0x00005555555555c5 <+138>: mov    ecx,DWORD PTR [rsp+rsi*4]
0x00005555555555c8 <+141>: mov    eax,0x1
0x00005555555555cd <+146>: lea    rdx,[rip+0x202c3c]          # 0x555555758210
<node1>
0x00005555555555d4 <+153>: cmp    ecx,0x1
0x00005555555555d7 <+156>: jg     0x5555555555ab <phase_6+112>
0x00005555555555d9 <+158>: jmp    0x5555555555b6 <phase_6+123>
0x00005555555555db <+160>: mov    esi,0x0
0x00005555555555e0 <+165>: jmp    0x5555555555c5 <phase_6+138>
0x00005555555555e2 <+167>: mov    rbx,QWORD PTR [rsp+0x20]
0x00005555555555e7 <+172>: mov    rax,QWORD PTR [rsp+0x28]
0x00005555555555ec <+177>: mov    QWORD PTR [rbx+0x8],rax
0x00005555555555f0 <+181>: mov    rdx,QWORD PTR [rsp+0x30]
0x00005555555555f5 <+186>: mov    QWORD PTR [rax+0x8],rdx
0x00005555555555f9 <+190>: mov    rax,QWORD PTR [rsp+0x38]
0x00005555555555fe <+195>: mov    QWORD PTR [rdx+0x8],rax
0x0000555555555602 <+199>: mov    rdx,QWORD PTR [rsp+0x40]
0x0000555555555607 <+204>: mov    QWORD PTR [rax+0x8],rdx
0x000055555555560b <+208>: mov    rax,QWORD PTR [rsp+0x48]
0x0000555555555610 <+213>: mov    QWORD PTR [rdx+0x8],rax
0x0000555555555614 <+217>: mov    QWORD PTR [rax+0x8],0x0

```

```

0x000055555555561c <+225>: mov     ebp,0x5
0x0000555555555621 <+230>: jmp     0x55555555562c <phase_6+241>
0x0000555555555623 <+232>: mov     rbx,QWORD PTR [rbx+0x8]
0x0000555555555627 <+236>: sub     ebp,0x1
0x000055555555562a <+239>: je      0x55555555563d <phase_6+258>
0x000055555555562c <+241>: mov     rax,QWORD PTR [rbx+0x8]
0x0000555555555630 <+245>: mov     eax,DWORD PTR [rax]
0x0000555555555632 <+247>: cmp     DWORD PTR [rbx],eax
0x0000555555555634 <+249>: jge     0x555555555623 <phase_6+232>
0x0000555555555636 <+251>: call    0x55555555589f <explode_bomb>
0x000055555555563b <+256>: jmp     0x555555555623 <phase_6+232>
0x000055555555563d <+258>: mov     rax,QWORD PTR [rsp+0x58]
0x0000555555555642 <+263>: xor     rax,QWORD PTR fs:0x28
0x000055555555564b <+272>: jne     0x555555555658 <phase_6+285>
0x000055555555564d <+274>: add     rsp,0x68
0x0000555555555651 <+278>: pop     rbx
0x0000555555555652 <+279>: pop     rbp
0x0000555555555653 <+280>: pop     r12
0x0000555555555655 <+282>: pop     r13
0x0000555555555657 <+284>: ret
0x0000555555555658 <+285>: call    0x555555554e50 <__stack_chk_fail@plt>

```

End of assembler dump.