Lab Assignment L2: Defusing a Binary Bomb

20190084 권민재, CSED211

preface

Using gdb-peda with AT&T syntax. In fact, please note that Intel syntax expression may remain because the report written with Intel syntax was modified to AT&T 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:
   0x0000555555555204 <+0>:  sub    $0x8,%rsp
   0x0000555555555208 <+4>:  lea    0x1691(%rip),%rsi        # 0x5555555568a0
   0x000055555555520f <+11>:  callq  0x555555555793 <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  0x55555555589f <explode_bomb>
   0x0000555555555222 <+30>:  jmp    0x555555555218 <phase_1+20>
End of assembler dump.
```

페이즈 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 rsi를 통해 전달된다. 그렇기 때문에, 이 문자열을 알아내기 위해서는 우리는 단순히 0x5555555568a0이 어떤 문자열인지 알아내거나, strings_not_equal이 호출될 때 rsi에 어떤 값이 들어있는지 확인하면 된다.

Solution 1. Read String from Address

```
gdb-peda$ x/s 0x5555555568a0
0x5555555568a0: "We have to stand with our North Korean allies."
```

Solution 2. Read RSI

```
gdb-peda$ b *0x55555555520f
Breakpoint 2 at 0x55555555520f
```

```
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: 0x5555557586a0 --> 0x31 ('1')
RBX: 0x0
RCX: 0x1
RDX: 0x5555557586a0 --> 0x31 ('1')
RSI: 0x5555555568a0 ("We have to stand with our North Korean allies.")
RDI: 0x5555557586a0 --> 0x31 ('1')
RBP: 0x5555555566d0 (<__libc_csu_init>: push   %r15)
RSP: 0x7fffffffe1e0 --> 0x555555554fa0 (<_start>: xor    %ebp,%ebp)
RIP: 0x55555555520f (<phase_1+11>:  callq  0x555555555793 <strings_not_equal>)
R8 : 0x555555759422 --> 0x0
R9 : 0x7ffff7fcd700 (0x00007ffff7fcd700)
R10: 0x7ffff7fcd700 (0x00007ffff7fcd700)
R11: 0x246
R12: 0x555555554fa0 (<_start>:  xor    %ebp,%ebp)
R13: 0x7fffffffe2d0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-----------------------------------code-----------------------------------
]
   0x5555555551ff <main+341>: callq  0x555555554f20 <exit@plt>
   0x555555555204 <phase_1>:  sub    $0x8,%rsp
   0x555555555208 <phase_1+4>:
    lea    0x1691(%rip),%rsi        # 0x5555555568a0
=> 0x55555555520f <phase_1+11>: callq  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    $0x8,%rsp
   0x55555555521c <phase_1+24>: retq
No argument
[-----------------------------------stack-----------------------------------
]
0000| 0x7fffffffe1e0 --> 0x555555554fa0 (<_start>:  xor    %ebp,%ebp)
0008| 0x7fffffffe1e8 --> 0x55555555510a (<main+96>: callq  0x555555555a4a
<phase_defused>)
0016| 0x7fffffffe1f0 --> 0x0
0024| 0x7fffffffe1f8 --> 0x7ffff7a2d840 (<__libc_start_main+240>: mov
 %eax,%edi)
0032| 0x7fffffffe200 --> 0x1
0040| 0x7fffffffe208 --> 0x7fffffffe2d8 --> 0x7fffffffe5e6
("/home/beta/bomb/bomb")
0048| 0x7fffffffe210 --> 0x1f7ffcca0
```

```
0056| 0x7fffffffe218 --> 0x5555555550aa (<main>:  push   %rbx)
[-------------------------------------------------------------------------------
]
Legend: code, data, rodata, value

Breakpoint 2, 0x000055555555520f 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. 이는 `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    $0x8,%rsp
   0x00005555555558c9 <+4>: mov    %rsi,%rdx
   0x00005555555558cc <+7>: lea    0x4(%rsi),%rcx
   0x00005555555558d0 <+11>:  lea    0x14(%rsi),%rax
   0x00005555555558d4 <+15>:  push   %rax
   0x00005555555558d5 <+16>:  lea    0x10(%rsi),%rax
   0x00005555555558d9 <+20>:  push   %rax
   0x00005555555558da <+21>:  lea    0xc(%rsi),%r9
   0x00005555555558de <+25>:  lea    0x8(%rsi),%r8
   0x00005555555558e2 <+29>:  lea    0x115a(%rip),%rsi        # 0x555555556a43
   0x00005555555558e9 <+36>:  mov    $0x0,%eax
   0x00005555555558ee <+41>:  callq  0x555555554ef0 <__isoc99_sscanf@plt>
   0x00005555555558f3 <+46>:  add    $0x10,%rsp
   0x00005555555558f7 <+50>:  cmp    $0x5,%eax
   0x00005555555558fa <+53>:  jle    0x555555555901 <read_six_numbers+60>
   0x00005555555558fc <+55>:  add    $0x8,%rsp
   0x0000555555555900 <+59>:  retq
   0x0000555555555901 <+60>:  callq  0x55555555589f <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
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  0x5555555558c5 <read_six_numbers>
0x0000555555555242 <+30>: cmpl   $0x0,(%rsp)
0x0000555555555246 <+34>: js     0x555555555252 <phase_2+46>
0x0000555555555248 <+36>: mov    $0x1,%ebx
0x000055555555524d <+41>: mov    %rsp,%rbp
0x0000555555555250 <+44>: jmp    0x555555555263 <phase_2+63>
0x0000555555555252 <+46>: callq  0x55555555589f <explode_bomb>
0x0000555555555257 <+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 페이즈 2에서는 우선 사용자의 입력을 파싱하여 올바른 입력이라면 다음 루틴으로 넘어가고, 아니라면 폭탄을 터트린다는 것을 알 수 있다.

Main Routine

```
# phase_2
0x0000555555555259 <+53>: add    $0x1,%rbx
0x000055555555525d <+57>: cmp    $0x6,%rbx
0x0000555555555261 <+61>: je     0x555555555276 <phase_2+82>
0x0000555555555263 <+63>: mov    %ebx,%eax
0x0000555555555265 <+65>: add    -0x4(%rbp,%rbx,4),%eax
0x0000555555555269 <+69>: cmp    %eax,0x0(%rbp,%rbx,4)
0x000055555555526d <+73>: je     0x555555555259 <phase_2+53>
0x000055555555526f <+75>: callq  0x55555555589f <explode_bomb>
0x0000555555555274 <+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

```
0x00005555555552a6 <+20>: lea    0xf(%rsp),%rcx
0x00005555555552ab <+25>: lea    0x10(%rsp),%rdx
0x00005555555552b0 <+30>: lea    0x14(%rsp),%r8
0x00005555555552b5 <+35>: lea    0x163a(%rip),%rsi        # 0x5555555568f6
0x00005555555552bc <+42>: callq  0x555555554ef0 <__isoc99_sscanf@plt>
0x00005555555552c1 <+47>: cmp    $0x2,%eax
0x00005555555552c4 <+50>: jle    0x5555555552e5 <phase_3+83>
0x00005555555552c6 <+52>: cmpl   $0x7,0x10(%rsp)
0x00005555555552cb <+57>: ja     0x5555555553da <phase_3+328>
0x00005555555552d1 <+63>: mov    0x10(%rsp),%eax
0x00005555555552d5 <+67>: lea    0x1634(%rip),%rdx        # 0x555555556910
0x00005555555552dc <+74>: movslq (%rdx,%rax,4),%rax
0x00005555555552e0 <+78>: add    %rdx,%rax
0x00005555555552e3 <+81>: jmpq   *%rax
0x00005555555552e5 <+83>: callq  0x55555555589f <explode_bomb>```
```r
gdb-peda$ x/s 0x5555555568f6
0x5555555568f6: "%d %c %d"
```

`phase_3`에서는 우선 `sscanf`를 이용하여 사용자의 입력을 파싱한다. `0x5555555568f6`가 `"%d %c %d"`이라는 점에서, 정수, 문자, 정수를 입력받는다는 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 페이즈 2에서는 정수, 문자, 정수의 입력을 받아야 한다는 사실을 알 수 있다.

Switch
```

```
0x00005555555552c6 <+52>: cmpl   $0x7,0x10(%rsp)
0x00005555555552cb <+57>: ja     0x5555555553da <phase_3+328>
0x00005555555552d1 <+63>: mov    0x10(%rsp),%eax
0x00005555555552d5 <+67>: lea    0x1634(%rip),%rdx        # 0x555555556910
0x00005555555552dc <+74>: movslq (%rdx,%rax,4),%rax
0x00005555555552e0 <+78>: add    %rdx,%rax
0x00005555555552e3 <+81>: jmpq   *%rax
0x00005555555552e5 <+83>: callq  0x55555555589f <explode_bomb>
0x00005555555552ea <+88>: jmp    0x5555555552c6 <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

```
0x00005555555552ec <+90>: mov    $0x67,%eax
0x00005555555552f1 <+95>: cmpl   $0x386,0x14(%rsp)
0x00005555555552f9 <+103>:  je     0x5555555553e4 <phase_3+338>
0x00005555555552ff <+109>:  callq  0x55555555589f <explode_bomb>
0x0000555555555304 <+114>:  mov    $0x67,%eax
0x0000555555555309 <+119>:  jmpq   0x5555555553e4 <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. 아니라면 `<+338>`로 점프한다. 이 케이스 0에서 A는 `0x67`, B는 `0x386`이다.

Check Character

```
0x00005555555553e4 <+338>:  cmp    %al,0xf(%rsp)
0x00005555555553e8 <+342>:  je     0x5555555553ef <phase_3+349>
0x00005555555553ea <+344>:  callq  0x55555555589f <explode_bomb>
```

위 케이스에서 eax에 특정 숫자를 저장하였는데, 이는 이 곳에서 쓰이게 된다. eax의 한 바이트와 사용자가 두번째에 입력한 글자가 다를 경우에는 폭탄이 터지게 된다.

Solution

위의 분석을 바탕으로, 이 페이즈에는 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

```
0x0000555555555456 <+20>: mov    %rsp,%rcx
0x0000555555555459 <+23>: lea    0x4(%rsp),%rdx
0x000055555555545e <+28>: lea    0x15ea(%rip),%rsi        # 0x555555556a4f
0x0000555555555465 <+35>: callq  0x555555554ef0 <__isoc99_sscanf@plt>
0x000055555555546a <+40>: cmp    $0x2,%eax
0x000055555555546d <+43>: jne    0x55555555547a <phase_4+56>
0x000055555555546f <+45>: mov    (%rsp),%eax
0x0000555555555472 <+48>: sub    $0x2,%eax
0x0000555555555475 <+51>: cmp    $0x2,%eax
0x0000555555555478 <+54>: jbe    0x55555555547f <phase_4+61>
0x000055555555547a <+56>: callq  0x55555555589f <explode_bomb>
```

```
gdb-peda$ x/s 0x555555556a4f
0x555555556a4f: "%d %d"
```

페이즈 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`

```
0x000055555555547f <+61>: mov    (%rsp),%esi
0x0000555555555482 <+64>: mov    $0x7,%edi
0x0000555555555487 <+69>: callq  0x555555555409 <func4>
0x000055555555548c <+74>: cmp    %eax,0x4(%rsp)
0x0000555555555490 <+78>: je     0x555555555497 <phase_4+85>
0x0000555555555492 <+80>: callq  0x55555555589f <explode_bomb>
```

사용자의 입력을 파싱 한 후에, 7과 사용자가 뒤에 입력한 정수를 인자로 `func4`를 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:
   0x0000555555555409 <+0>: mov    $0x0,%eax
   0x000055555555540e <+5>: test   %edi,%edi
```

```
   0x0000555555555410 <+7>: jle    0x555555555419 <func4+16>
   0x0000555555555412 <+9>: mov    %esi,%eax
   0x0000555555555414 <+11>:  cmp    $0x1,%edi
   0x0000555555555417 <+14>:  jne    0x55555555541b <func4+18>
   0x0000555555555419 <+16>:  repz retq
   0x000055555555541b <+18>:  push   %r12
   0x000055555555541d <+20>:  push   %rbp
   0x000055555555541e <+21>:  push   %rbx
   0x000055555555541f <+22>:  mov    %esi,%r12d
   0x0000555555555422 <+25>:  mov    %edi,%ebx
   0x0000555555555424 <+27>:  lea    -0x1(%rdi),%edi
   0x0000555555555427 <+30>:  callq  0x555555555409 <func4>
   0x000055555555542c <+35>:  lea    (%rax,%r12,1),%ebp
   0x0000555555555430 <+39>:  lea    -0x2(%rbx),%edi
   0x0000555555555433 <+42>:  mov    %r12d,%esi
   0x0000555555555436 <+45>:  callq  0x555555555409 <func4>
   0x000055555555543b <+50>:  add    %ebp,%eax
   0x000055555555543d <+52>:  pop    %rbx
   0x000055555555543e <+53>:  pop    %rbp
   0x000055555555543f <+54>:  pop    %r12
   0x0000555555555441 <+56>:  retq
End of assembler dump.
```

`func4`를 디스어셈블해보면 재귀 호출을 이용하여 매우 복잡한 루틴을 가지고 있는 것을 볼 수 있다. 하지만 `func4`를 우리가 분석할 필요는 없다. 왜냐하면 우리는 굳이 분석해보지 않아도 입력에 따른 `func4`의 반환값을 확인할 수 있고, 그 과정은 아래와 같다.

```
gdb-peda$ c
Continuing.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2.  Keep going!
0 g 902
Halfway there!
2 2
[----------------------------------registers-----------------------------------
]
RAX: 0x42 ('B')
RBX: 0x0
RCX: 0x20 (' ')
RDX: 0x7fffffffe1d0 --> 0x4200000002
RSI: 0x2
RDI: 0x1
RBP: 0x5555555566d0 (<__libc_csu_init>: push   %r15)
RSP: 0x7fffffffe1d0 --> 0x4200000002
RIP: 0x55555555548c (<phase_4+74>:  cmp    %eax,0x4(%rsp))
R8 : 0x0
R9 : 0x0
```

```
R10: 0x0
R11: 0x7ffff7b846a0 --> 0x2000200020002
R12: 0x555555554fa0 (<_start>:  xor    %ebp,%ebp)
R13: 0x7fffffffe2d0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------
]
   0x55555555547f <phase_4+61>: mov    (%rsp),%esi
   0x555555555482 <phase_4+64>: mov    $0x7,%edi
   0x555555555487 <phase_4+69>: callq  0x555555555409 <func4>
=> 0x55555555548c <phase_4+74>: cmp    %eax,0x4(%rsp)
   0x555555555490 <phase_4+78>: je     0x555555555497 <phase_4+85>
   0x555555555492 <phase_4+80>: callq  0x55555555589f <explode_bomb>
   0x555555555497 <phase_4+85>: mov    0x8(%rsp),%rax
   0x55555555549c <phase_4+90>: xor    %fs:0x28,%rax
[-------------------------------------stack-------------------------------------
]
0000| 0x7fffffffe1d0 --> 0x4200000002
0008| 0x7fffffffe1d8 --> 0xadbdba5642b56a00
0016| 0x7fffffffe1e0 --> 0x555555554fa0 (<_start>:  xor    %ebp,%ebp)
0024| 0x7fffffffe1e8 --> 0x555555555164 (<main+186>:  callq  0x555555555a4a
<phase_defused>)
0032| 0x7fffffffe1f0 --> 0x0
0040| 0x7fffffffe1f8 --> 0x7ffff7a2d840 (<__libc_start_main+240>: mov
%eax,%edi)
0048| 0x7fffffffe200 --> 0x1
0056| 0x7fffffffe208 --> 0x7fffffffe2d8 --> 0x7fffffffe5e6
("/home/beta/bomb/bomb")
[------------------------------------------------------------------------------
]
Legend: code, data, rodata, value

Breakpoint 3, 0x000055555555548c in phase_4 ()
```

위의 과정에서, 두번째 숫자로 2를 입력 했을 때 `func4`는 `0x42`를 반환한다는 사실을 `rax`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:
   0x0000555555555776 <+0>: cmpb   $0x0,(%rdi)
   0x0000555555555779 <+3>: je     0x55555555578d <string_length+23>
   0x000055555555577b <+5>: mov    %rdi,%rdx
   0x000055555555577e <+8>: add    $0x1,%rdx
   0x0000555555555782 <+12>:  mov    %edx,%eax
   0x0000555555555784 <+14>:  sub    %edi,%eax
   0x0000555555555786 <+16>:  cmpb   $0x0,(%rdx)
   0x0000555555555789 <+19>:  jne    0x55555555577e <string_length+8>
   0x000055555555578b <+21>:  repz retq
   0x000055555555578d <+23>:  mov    $0x0,%eax
   0x0000555555555792 <+28>:  retq
End of assembler dump.
```

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

Check Length of Input

```
0x00005555555554c9 <+24>: callq  0x555555555776 <string_length>
0x00005555555554ce <+29>: cmp    $0x6,%eax
0x00005555555554d1 <+32>: jne    0x555555555528 <phase_5+119>
... (생략)
0x0000555555555528 <+119>:  callq  0x55555555589f <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 아니라면 폭탄을 터뜨린다.

Set String

```
0x00005555555554d3 <+34>: mov    $0x0,%eax
0x00005555555554d8 <+39>: lea    0x1451(%rip),%rcx        # 0x555555556930
<array>
0x00005555555554df <+46>: movzbl (%rbx,%rax,1),%edx
0x00005555555554e3 <+50>: and    $0xf,%edx
0x00005555555554e6 <+53>: movzbl (%rcx,%rdx,1),%edx
0x00005555555554ea <+57>: mov    %dl,0x1(%rsp,%rax,1)
0x00005555555554ee <+61>: add    $0x1,%rax
0x00005555555554f2 <+65>: cmp    $0x6,%rax
0x00005555555554f6 <+69>: jne    0x5555555554df <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 6이 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

```
0x0000555555555f8 <+71>: movb   $0x0,0x7(%rsp)
0x00005555555554fd <+76>: lea    0x1(%rsp),%rdi
0x0000555555555502 <+81>: lea    0x13f6(%rip),%rsi        # 0x5555555568ff
0x0000555555555509 <+88>: callq  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 동일한지 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    %rsp,%r12
0x0000555555555558 <+29>: mov    %r12,%rsi
0x000055555555555b <+32>: callq  0x5555555558c5 <read_six_numbers>
```

페이즈 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    $0x0,%r13d
0x0000555555555566 <+43>: jmp    0x55555555558d <phase_6+82>
0x0000555555555568 <+45>: callq  0x55555555589f <explode_bomb>
0x000055555555556d <+50>: jmp    0x55555555559c <phase_6+97>
# ... (Subroutine)
0x0000555555555589 <+78>: add    $0x4,%r12
0x000055555555558d <+82>: mov    %r12,%rbp
0x0000555555555590 <+85>: mov    (%r12),%eax
0x0000555555555594 <+89>: sub    $0x1,%eax
0x0000555555555597 <+92>: cmp    $0x5,%eax
0x000055555555559a <+95>: ja     0x555555555568 <phase_6+45>
0x000055555555559c <+97>: add    $0x1,%r13d
0x00005555555555a0 <+101>:  cmp    $0x6,%r13d
0x00005555555555a4 <+105>:  je     0x5555555555db <phase_6+160>
0x00005555555555a6 <+107>:  mov    %r13d,%ebx
0x00005555555555a9 <+110>:  jmp    0x555555555577 <phase_6+60>
```

사용자가 입력한 숫자에 대해 반복문을 돌면서, `[r12]`가 6보다 크다면 폭탄을 터트린다. 그리고 `r13d`는 반복 카운터로서 작용하고 있는데, 이것이 6이 되었을 때 반복문을 탈출하게 된다. 이후 `ebx`에 `r13d`의 값을 저장하고 서브루틴에 들어간다. 서브루틴이 종료된 이후에는 `r12`에 4가 더해지는, 즉 배열에서 다음 인덱스의 주소가 들어가게 되고, 이것은 rbp에 전달되게 된다.

Subroutine

```
0x000055555555556f <+52>: add    $0x1,%ebx
0x0000555555555572 <+55>: cmp    $0x5,%ebx
0x0000555555555575 <+58>: jg     0x555555555589 <phase_6+78>
0x0000555555555577 <+60>: movslq %ebx,%rax
0x000055555555557a <+63>: mov    (%rsp,%rax,4),%eax
0x000055555555557d <+66>: cmp    %eax,0x0(%rbp)
0x0000555555555580 <+69>: jne    0x55555555556f <phase_6+52>
0x0000555555555582 <+71>: callq  0x55555555589f <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

```
0x00005555555555cd <+146>:  lea    0x202c3c(%rip),%rdx        # 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)
0x00005555555555b6 <+123>:  mov    %rdx,0x20(%rsp,%rsi,8)
0x00005555555555bb <+128>:  add    $0x1,%rsi
0x00005555555555bf <+132>:  cmp    $0x6,%rsi
0x00005555555555c3 <+136>:  je     0x5555555555e2 <phase_6+167>
0x00005555555555c5 <+138>:  mov    (%rsp,%rsi,4),%ecx
0x00005555555555c8 <+141>:  mov    $0x1,%eax
0x00005555555555cd <+146>:  lea    0x202c3c(%rip),%rdx        # 0x555555758210
<node1>
0x00005555555555d4 <+153>:  cmp    $0x1,%ecx
0x00005555555555d7 <+156>:  jg     0x5555555555ab <phase_6+112>
0x00005555555555d9 <+158>:  jmp    0x5555555555b6 <phase_6+123>
0x00005555555555db <+160>:  mov    $0x0,%esi
0x00005555555555e0 <+165>:  jmp    0x5555555555c5 <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    0x8(%rdx),%rdx
0x00005555555555af <+116>:  add    $0x1,%eax
0x00005555555555b2 <+119>:  cmp    %ecx,%eax
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    0x20(%rsp),%rbx
0x00005555555555e7 <+172>:  mov    0x28(%rsp),%rax
0x00005555555555ec <+177>:  mov    %rax,0x8(%rbx)
0x00005555555555f0 <+181>:  mov    0x30(%rsp),%rdx
0x00005555555555f5 <+186>:  mov    %rdx,0x8(%rax)
0x00005555555555f9 <+190>:  mov    0x38(%rsp),%rax
0x00005555555555fe <+195>:  mov    %rax,0x8(%rdx)
0x0000555555555602 <+199>:  mov    0x40(%rsp),%rdx
0x0000555555555607 <+204>:  mov    %rdx,0x8(%rax)
0x000055555555560b <+208>:  mov    0x48(%rsp),%rax
0x0000555555555610 <+213>:  mov    %rax,0x8(%rdx)
0x0000555555555614 <+217>:  movq   $0x0,0x8(%rax)
0x000055555555561c <+225>:  mov    $0x5,%ebp
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    0x8(%rbx),%rbx
0x0000555555555627 <+236>:  sub    $0x1,%ebp
0x000055555555562a <+239>:  je     0x55555555563d <phase_6+258>
0x000055555555562c <+241>:  mov    0x8(%rbx),%rax
0x0000555555555630 <+245>:  mov    (%rax),%eax
0x0000555555555632 <+247>:  cmp    %eax,(%rbx)
0x0000555555555634 <+249>:  jge    0x555555555623 <phase_6+232>
0x0000555555555636 <+251>:  callq  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:
   0x0000555555555a4a <+0>: sub    $0x78,%rsp
   0x0000555555555a4e <+4>: mov    %fs:0x28,%rax
   0x0000555555555a57 <+13>:  mov    %rax,0x68(%rsp)
   0x0000555555555a5c <+18>:  xor    %eax,%eax
   0x0000555555555a5e <+20>:  cmpl   $0x6,0x202c27(%rip)        #
0x55555575868c <num_input_strings>
   0x0000555555555a65 <+27>:  je     0x555555555a7c <phase_defused+50>
   0x0000555555555a67 <+29>:  mov    0x68(%rsp),%rax
   0x0000555555555a6c <+34>:  xor    %fs:0x28,%rax
   0x0000555555555a75 <+43>:  jne    0x555555555aea <phase_defused+160>
   0x0000555555555a77 <+45>:  add    $0x78,%rsp
   0x0000555555555a7b <+49>:  retq
   0x0000555555555a7c <+50>:  lea    0xc(%rsp),%rcx
   0x0000555555555a81 <+55>:  lea    0x8(%rsp),%rdx
   0x0000555555555a86 <+60>:  lea    0x10(%rsp),%r8
   0x0000555555555a8b <+65>:  lea    0x1007(%rip),%rsi        # 0x555555556a99
   0x0000555555555a92 <+72>:  lea    0x202cf7(%rip),%rdi        #
0x555555758790 <input_strings+240>
```

```
   0x0000555555555a99 <+79>:  callq  0x555555554ef0 <__isoc99_sscanf@plt>
   0x0000555555555a9e <+84>:  cmp    $0x3,%eax
   0x0000555555555aa1 <+87>:  je     0x555555555ab1 <phase_defused+103>
   0x0000555555555aa3 <+89>:  lea    0xf2e(%rip),%rdi        # 0x5555555569d8
   0x0000555555555aaa <+96>:  callq  0x555555554e30 <puts@plt>
   0x0000555555555aaf <+101>: jmp    0x555555555a67 <phase_defused+29>
   0x0000555555555ab1 <+103>: lea    0x10(%rsp),%rdi
   0x0000555555555ab6 <+108>: lea    0xfe5(%rip),%rsi        # 0x555555556aa2
   0x0000555555555abd <+115>: callq  0x555555555793 <strings_not_equal>
   0x0000555555555ac2 <+120>: test   %eax,%eax
   0x0000555555555ac4 <+122>: jne    0x555555555aa3 <phase_defused+89>
   0x0000555555555ac6 <+124>: lea    0xeab(%rip),%rdi        # 0x555555556978
   0x0000555555555acd <+131>: callq  0x555555554e30 <puts@plt>
   0x0000555555555ad2 <+136>: lea    0xec7(%rip),%rdi        # 0x5555555569a0
   0x0000555555555ad9 <+143>: callq  0x555555554e30 <puts@plt>
   0x0000555555555ade <+148>: mov    $0x0,%eax
   0x0000555555555ae3 <+153>: callq  0x55555555569c <secret_phase>
   0x0000555555555ae8 <+158>: jmp    0x555555555aa3 <phase_defused+89>
   0x0000555555555aea <+160>: callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

```
gdb-peda$ x/s 0x555555556a99
0x555555556a99: "%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 페이즈 4의 정답이며, 이를 통해 페이즈 4의 정답을 입력할 때 추가적으로 문자열을 받고 있는 것을 알 수 있다.

```
gdb-peda$ x/s 0x555555556aa2
0x555555556aa2: "DrEvil"
```

이때, `strings_not_equal`에서 비교하고 있는 문자열의 주소인 0x555555556aa2를 확인해보면, 그것이 `"DrEvil"`임을 알 수 있고, 페이즈 4에서 아래와 같은 답안을 입력하면 페이즈 6이 끝나고 secret_phase에 들어갈 수 있다. 이것은 `num_input_strings`를 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:
   0x00005555555556c <+0>: push   %rbx
   0x00005555555556d <+1>: callq  0x555555555906 <read_line>
   0x00005555555556a2 <+6>: mov    $0xa,%edx
   0x00005555555556a7 <+11>:  mov    $0x0,%esi
   0x00005555555556ac <+16>:  mov    %rax,%rdi
   0x00005555555556af <+19>:  callq  0x555555554ed0 <strtol@plt>
   0x00005555555556b4 <+24>:  mov    %rax,%rbx
   0x00005555555556b7 <+27>:  lea    -0x1(%rax),%eax
   0x00005555555556ba <+30>:  cmp    $0x3e8,%eax
   0x00005555555556bf <+35>:  ja     0x5555555556ec <secret_phase+80>
   0x00005555555556c1 <+37>:  mov    %ebx,%esi
   0x00005555555556c3 <+39>:  lea    0x202a66(%rip),%rdi        #
0x555555758130 <n1>
   0x00005555555556ca <+46>:  callq  0x55555555565d <fun7>
   0x00005555555556cf <+51>:  cmp    $0x5,%eax
   0x00005555555556d2 <+54>:  je     0x5555555556d9 <secret_phase+61>
   0x00005555555556d4 <+56>:  callq  0x55555555589f <explode_bomb>
   0x00005555555556d9 <+61>:  lea    0x11f0(%rip),%rdi        # 0x5555555568d0
   0x00005555555556e0 <+68>:  callq  0x555555554e30 <puts@plt>
   0x00005555555556e5 <+73>:  callq  0x555555555a4a <phase_defused>
   0x00005555555556ea <+78>:  pop    %rbx
   0x00005555555556eb <+79>:  retq
   0x00005555555556ec <+80>:  callq  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, `strtol`을 이용하여 해당 문자열을 십진수 숫자로 변환한다. 그리고 만약 그 숫자가 1001 보다 큰 경우에는 폭탄을 터뜨린다. 그리고, `n1`의 주소와 사용자가 입력한 숫자를 인자로 가지고 `fun7`을 실행시킨 결과가 5가 되도록 하면 secret phase를 처리할 수 있다. 그러면 `fun7`을 살펴보자.

fun7

```
gdb-peda$ disas *fun7
Dump of assembler code for function fun7:
   0x00005555555555d <+0>: test   %rdi,%rdi
   0x0000555555555660 <+3>: je     0x555555555696 <fun7+57>
   0x0000555555555662 <+5>: sub    $0x8,%rsp
   0x0000555555555666 <+9>: mov    (%rdi),%edx
   0x0000555555555668 <+11>:  cmp    %esi,%edx
   0x000055555555566a <+13>:  jg     0x55555555567a <fun7+29>
   0x000055555555566c <+15>:  mov    $0x0,%eax
   0x0000555555555671 <+20>:  cmp    %esi,%edx
   0x0000555555555673 <+22>:  jne    0x555555555687 <fun7+42>
   0x0000555555555675 <+24>:  add    $0x8,%rsp
   0x0000555555555679 <+28>:  retq
   0x000055555555567a <+29>:  mov    0x8(%rdi),%rdi
```

```
   0x000055555555567e <+33>:  callq  0x55555555565d <fun7>
   0x0000555555555683 <+38>:  add    %eax,%eax
   0x0000555555555685 <+40>:  jmp    0x555555555675 <fun7+24>
   0x0000555555555687 <+42>:  mov    0x10(%rdi),%rdi
   0x000055555555568b <+46>:  callq  0x55555555565d <fun7>
   0x0000555555555690 <+51>:  lea    0x1(%rax,%rax,1),%eax
   0x0000555555555694 <+55>:  jmp    0x555555555675 <fun7+24>
   0x0000555555555696 <+57>:  mov    $0xffffffff,%eax
   0x000055555555569b <+62>:  retq
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 fun7의 첫번째 인자의 자료형이 구조체일 것이라고 추론해볼 수 있다. 처음 DWORD PTR에 값을 가지고, rdi+0x8와 rdi+0x10에 각각 다른 구조체 변수의 주소를 가지고 있는 형식이라고 추론 가능하다. 이 점을 바탕으로, n1을 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>: 0x00000000000003e9  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 n번째로 존재할 때 n이다. $(0 \leq n, n \in \mathbb{Z})$ 우리는 위에서 모든 노드를 구했고, 노드의 변수 이름에는 $n_{(깊이)(인덱스)}$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:
   0x0000555555555204 <+0>:  sub    $0x8,%rsp
   0x0000555555555208 <+4>:  lea    0x1691(%rip),%rsi        # 0x5555555568a0
   0x000055555555520f <+11>:  callq  0x555555555793 <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  0x55555555589f <explode_bomb>
   0x0000555555555222 <+30>:  jmp    0x555555555218 <phase_1+20>
End of assembler dump.
```

phase_2

```
gdb-peda$ 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  0x5555555558c5 <read_six_numbers>
   0x0000555555555242 <+30>:  cmpl   $0x0,(%rsp)
   0x0000555555555246 <+34>:  js     0x555555555252 <phase_2+46>
   0x0000555555555248 <+36>:  mov    $0x1,%ebx
   0x000055555555524d <+41>:  mov    %rsp,%rbp
   0x0000555555555250 <+44>:  jmp    0x555555555263 <phase_2+63>
   0x0000555555555252 <+46>:  callq  0x55555555589f <explode_bomb>
   0x0000555555555257 <+51>:  jmp    0x555555555248 <phase_2+36>
   0x0000555555555259 <+53>:  add    $0x1,%rbx
   0x000055555555525d <+57>:  cmp    $0x6,%rbx
   0x0000555555555261 <+61>:  je     0x555555555276 <phase_2+82>
   0x0000555555555263 <+63>:  mov    %ebx,%eax
   0x0000555555555265 <+65>:  add    -0x4(%rbp,%rbx,4),%eax
   0x0000555555555269 <+69>:  cmp    %eax,0x0(%rbp,%rbx,4)
   0x000055555555526d <+73>:  je     0x555555555259 <phase_2+53>
   0x000055555555526f <+75>:  callq  0x55555555589f <explode_bomb>
   0x0000555555555274 <+80>:  jmp    0x555555555259 <phase_2+53>
   0x0000555555555276 <+82>:  mov    0x18(%rsp),%rax
   0x000055555555527b <+87>:  xor    %fs:0x28,%rax
   0x0000555555555284 <+96>:  jne    0x55555555528d <phase_2+105>
   0x0000555555555286 <+98>:  add    $0x28,%rsp
   0x000055555555528a <+102>: pop    %rbx
   0x000055555555528b <+103>: pop    %rbp
   0x000055555555528c <+104>: retq
   0x000055555555528d <+105>: callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

phase_3

```
gdb-peda$ disas *phase_3
Dump of assembler code for function phase_3:
   0x0000555555555292 <+0>: sub    $0x28,%rsp
   0x0000555555555296 <+4>: mov    %fs:0x28,%rax
   0x000055555555529f <+13>:  mov    %rax,0x18(%rsp)
   0x00005555555552a4 <+18>:  xor    %eax,%eax
   0x00005555555552a6 <+20>:  lea    0xf(%rsp),%rcx
```

```
0x00005555555552ab <+25>:  lea    0x10(%rsp),%rdx
0x00005555555552b0 <+30>:  lea    0x14(%rsp),%r8
0x00005555555552b5 <+35>:  lea    0x163a(%rip),%rsi        # 0x5555555568f6
0x00005555555552bc <+42>:  callq  0x555555554ef0 <__isoc99_sscanf@plt>
0x00005555555552c1 <+47>:  cmp    $0x2,%eax
0x00005555555552c4 <+50>:  jle    0x5555555552e5 <phase_3+83>
0x00005555555552c6 <+52>:  cmpl   $0x7,0x10(%rsp)
0x00005555555552cb <+57>:  ja     0x5555555553da <phase_3+328>
0x00005555555552d1 <+63>:  mov    0x10(%rsp),%eax
0x00005555555552d5 <+67>:  lea    0x1634(%rip),%rdx        # 0x555555556910
0x00005555555552dc <+74>:  movslq (%rdx,%rax,4),%rax
0x00005555555552e0 <+78>:  add    %rdx,%rax
0x00005555555552e3 <+81>:  jmpq   *%rax
0x00005555555552e5 <+83>:  callq  0x55555555589f <explode_bomb>
0x00005555555552ea <+88>:  jmp    0x5555555552c6 <phase_3+52>
0x00005555555552ec <+90>:  mov    $0x67,%eax
0x00005555555552f1 <+95>:  cmpl   $0x386,0x14(%rsp)
0x00005555555552f9 <+103>: je     0x5555555553e4 <phase_3+338>
0x00005555555552ff <+109>: callq  0x55555555589f <explode_bomb>
0x0000555555555304 <+114>: mov    $0x67,%eax
0x0000555555555309 <+119>: jmpq   0x5555555553e4 <phase_3+338>
0x000055555555530e <+124>: mov    $0x68,%eax
0x0000555555555313 <+129>: cmpl   $0x2cf,0x14(%rsp)
0x000055555555531b <+137>: je     0x5555555553e4 <phase_3+338>
0x0000555555555321 <+143>: callq  0x55555555589f <explode_bomb>
0x0000555555555326 <+148>: mov    $0x68,%eax
0x000055555555532b <+153>: jmpq   0x5555555553e4 <phase_3+338>
0x0000555555555330 <+158>: mov    $0x74,%eax
0x0000555555555335 <+163>: cmpl   $0x244,0x14(%rsp)
0x000055555555533d <+171>: je     0x5555555553e4 <phase_3+338>
0x0000555555555343 <+177>: callq  0x55555555589f <explode_bomb>
0x0000555555555348 <+182>: mov    $0x74,%eax
0x000055555555534d <+187>: jmpq   0x5555555553e4 <phase_3+338>
0x0000555555555352 <+192>: mov    $0x67,%eax
0x0000555555555357 <+197>: cmpl   $0x76,0x14(%rsp)
0x000055555555535c <+202>: je     0x5555555553e4 <phase_3+338>
0x0000555555555362 <+208>: callq  0x55555555589f <explode_bomb>
0x0000555555555367 <+213>: mov    $0x67,%eax
0x000055555555536c <+218>: jmp    0x5555555553e4 <phase_3+338>
0x000055555555536e <+220>: mov    $0x6a,%eax
0x0000555555555373 <+225>: cmpl   $0x38b,0x14(%rsp)
0x000055555555537b <+233>: je     0x5555555553e4 <phase_3+338>
0x000055555555537d <+235>: callq  0x55555555589f <explode_bomb>
0x0000555555555382 <+240>: mov    $0x6a,%eax
0x0000555555555387 <+245>: jmp    0x5555555553e4 <phase_3+338>
0x0000555555555389 <+247>: mov    $0x6b,%eax
0x000055555555538e <+252>: cmpl   $0xfd,0x14(%rsp)
0x0000555555555396 <+260>: je     0x5555555553e4 <phase_3+338>
0x0000555555555398 <+262>: callq  0x55555555589f <explode_bomb>
```

```
   0x00005555555553d <+267>: mov    $0x6b,%eax
   0x00005555555553a2 <+272>: jmp    0x5555555553e4 <phase_3+338>
   0x00005555555553a4 <+274>: mov    $0x6e,%eax
   0x00005555555553a9 <+279>: cmpl   $0x126,0x14(%rsp)
   0x00005555555553b1 <+287>: je     0x5555555553e4 <phase_3+338>
   0x00005555555553b3 <+289>: callq  0x55555555589f <explode_bomb>
   0x00005555555553b8 <+294>: mov    $0x6e,%eax
   0x00005555555553bd <+299>: jmp    0x5555555553e4 <phase_3+338>
   0x00005555555553bf <+301>: mov    $0x64,%eax
   0x00005555555553c4 <+306>: cmpl   $0x3bc,0x14(%rsp)
   0x00005555555553cc <+314>: je     0x5555555553e4 <phase_3+338>
   0x00005555555553ce <+316>: callq  0x55555555589f <explode_bomb>
   0x00005555555553d3 <+321>: mov    $0x64,%eax
   0x00005555555553d8 <+326>: jmp    0x5555555553e4 <phase_3+338>
   0x00005555555553da <+328>: callq  0x55555555589f <explode_bomb>
   0x00005555555553df <+333>: mov    $0x72,%eax
   0x00005555555553e4 <+338>: cmp    %al,0xf(%rsp)
   0x00005555555553e8 <+342>: je     0x5555555553ef <phase_3+349>
   0x00005555555553ea <+344>: callq  0x55555555589f <explode_bomb>
   0x00005555555553ef <+349>: mov    0x18(%rsp),%rax
   0x00005555555553f4 <+354>: xor    %fs:0x28,%rax
   0x00005555555553fd <+363>: jne    0x555555555404 <phase_3+370>
   0x00005555555553ff <+365>: add    $0x28,%rsp
   0x0000555555555403 <+369>: retq
   0x0000555555555404 <+370>: callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

phase_4

```
gdb-peda$ disas *phase_4
Dump of assembler code for function phase_4:
   0x0000555555555442 <+0>: sub    $0x18,%rsp
   0x0000555555555446 <+4>: mov    %fs:0x28,%rax
   0x000055555555544f <+13>:  mov    %rax,0x8(%rsp)
   0x0000555555555454 <+18>:  xor    %eax,%eax
   0x0000555555555456 <+20>:  mov    %rsp,%rcx
   0x0000555555555459 <+23>:  lea    0x4(%rsp),%rdx
   0x000055555555545e <+28>:  lea    0x15ea(%rip),%rsi        # 0x555555556a4f
   0x0000555555555465 <+35>:  callq  0x555555554ef0 <__isoc99_sscanf@plt>
   0x000055555555546a <+40>:  cmp    $0x2,%eax
   0x000055555555546d <+43>:  jne    0x55555555547a <phase_4+56>
   0x000055555555546f <+45>:  mov    (%rsp),%eax
   0x0000555555555472 <+48>:  sub    $0x2,%eax
   0x0000555555555475 <+51>:  cmp    $0x2,%eax
   0x0000555555555478 <+54>:  jbe    0x55555555547f <phase_4+61>
   0x000055555555547a <+56>:  callq  0x55555555589f <explode_bomb>
   0x000055555555547f <+61>:  mov    (%rsp),%esi
   0x0000555555555482 <+64>:  mov    $0x7,%edi
```

```
   0x0000555555555487 <+69>:  callq  0x555555555409 <func4>
   0x000055555555548c <+74>:  cmp    %eax,0x4(%rsp)
   0x0000555555555490 <+78>:  je     0x555555555497 <phase_4+85>
   0x0000555555555492 <+80>:  callq  0x55555555589f <explode_bomb>
   0x0000555555555497 <+85>:  mov    0x8(%rsp),%rax
   0x000055555555549c <+90>:  xor    %fs:0x28,%rax
   0x00005555555554a5 <+99>:  jne    0x5555555554ac <phase_4+106>
   0x00005555555554a7 <+101>: add    $0x18,%rsp
   0x00005555555554ab <+105>: retq
   0x00005555555554ac <+106>: callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

phase_5

```
gdb-peda$ disas *phase_5
Dump of assembler code for function phase_5:
   0x00005555555554b1 <+0>: push   %rbx
   0x00005555555554b2 <+1>: sub    $0x10,%rsp
   0x00005555555554b6 <+5>: mov    %rdi,%rbx
   0x00005555555554b9 <+8>: mov    %fs:0x28,%rax
   0x00005555555554c2 <+17>:  mov    %rax,0x8(%rsp)
   0x00005555555554c7 <+22>:  xor    %eax,%eax
   0x00005555555554c9 <+24>:  callq  0x555555555776 <string_length>
   0x00005555555554ce <+29>:  cmp    $0x6,%eax
   0x00005555555554d1 <+32>:  jne    0x555555555528 <phase_5+119>
   0x00005555555554d3 <+34>:  mov    $0x0,%eax
   0x00005555555554d8 <+39>:  lea    0x1451(%rip),%rcx        # 0x555555556930
<array>
   0x00005555555554df <+46>:  movzbl (%rbx,%rax,1),%edx
   0x00005555555554e3 <+50>:  and    $0xf,%edx
   0x00005555555554e6 <+53>:  movzbl (%rcx,%rdx,1),%edx
   0x00005555555554ea <+57>:  mov    %dl,0x1(%rsp,%rax,1)
   0x00005555555554ee <+61>:  add    $0x1,%rax
   0x00005555555554f2 <+65>:  cmp    $0x6,%rax
   0x00005555555554f6 <+69>:  jne    0x5555555554df <phase_5+46>
   0x00005555555554f8 <+71>:  movb   $0x0,0x7(%rsp)
   0x00005555555554fd <+76>:  lea    0x1(%rsp),%rdi
   0x0000555555555502 <+81>:  lea    0x13f6(%rip),%rsi        # 0x5555555568ff
   0x0000555555555509 <+88>:  callq  0x555555555793 <strings_not_equal>
   0x000055555555550e <+93>:  test   %eax,%eax
   0x0000555555555510 <+95>:  jne    0x55555555552f <phase_5+126>
   0x0000555555555512 <+97>:  mov    0x8(%rsp),%rax
   0x0000555555555517 <+102>: xor    %fs:0x28,%rax
   0x0000555555555520 <+111>: jne    0x555555555536 <phase_5+133>
   0x0000555555555522 <+113>: add    $0x10,%rsp
   0x0000555555555526 <+117>: pop    %rbx
   0x0000555555555527 <+118>: retq
   0x0000555555555528 <+119>: callq  0x55555555589f <explode_bomb>
```

```
   0x000055555555552d <+124>: jmp    0x5555555554d3 <phase_5+34>
   0x000055555555552f <+126>: callq  0x55555555589f <explode_bomb>
   0x0000555555555534 <+131>: jmp    0x555555555512 <phase_5+97>
   0x0000555555555536 <+133>: callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```

phase_6

```
gdb-peda$ disas *phase_6
Dump of assembler code for function phase_6:
   0x000055555555553b <+0>: push   %r13
   0x000055555555553d <+2>: push   %r12
   0x000055555555553f <+4>: push   %rbp
   0x0000555555555540 <+5>: push   %rbx
   0x0000555555555541 <+6>: sub    $0x68,%rsp
   0x0000555555555545 <+10>:  mov    %fs:0x28,%rax
   0x000055555555554e <+19>:  mov    %rax,0x58(%rsp)
   0x0000555555555553 <+24>:  xor    %eax,%eax
   0x0000555555555555 <+26>:  mov    %rsp,%r12
   0x0000555555555558 <+29>:  mov    %r12,%rsi
   0x000055555555555b <+32>:  callq  0x5555555558c5 <read_six_numbers>
   0x0000555555555560 <+37>:  mov    $0x0,%r13d
   0x0000555555555566 <+43>:  jmp    0x55555555558d <phase_6+82>
   0x0000555555555568 <+45>:  callq  0x55555555589f <explode_bomb>
   0x000055555555556d <+50>:  jmp    0x55555555559c <phase_6+97>
   0x000055555555556f <+52>:  add    $0x1,%ebx
   0x0000555555555572 <+55>:  cmp    $0x5,%ebx
   0x0000555555555575 <+58>:  jg     0x555555555589 <phase_6+78>
   0x0000555555555577 <+60>:  movslq %ebx,%rax
   0x000055555555557a <+63>:  mov    (%rsp,%rax,4),%eax
   0x000055555555557d <+66>:  cmp    %eax,0x0(%rbp)
   0x0000555555555580 <+69>:  jne    0x55555555556f <phase_6+52>
   0x0000555555555582 <+71>:  callq  0x55555555589f <explode_bomb>
   0x0000555555555587 <+76>:  jmp    0x55555555556f <phase_6+52>
   0x0000555555555589 <+78>:  add    $0x4,%r12
   0x000055555555558d <+82>:  mov    %r12,%rbp
   0x0000555555555590 <+85>:  mov    (%r12),%eax
   0x0000555555555594 <+89>:  sub    $0x1,%eax
   0x0000555555555597 <+92>:  cmp    $0x5,%eax
   0x000055555555559a <+95>:  ja     0x555555555568 <phase_6+45>
   0x000055555555559c <+97>:  add    $0x1,%r13d
   0x00005555555555a0 <+101>: cmp    $0x6,%r13d
   0x00005555555555a4 <+105>: je     0x5555555555db <phase_6+160>
   0x00005555555555a6 <+107>: mov    %r13d,%ebx
   0x00005555555555a9 <+110>: jmp    0x555555555577 <phase_6+60>
   0x00005555555555ab <+112>: mov    0x8(%rdx),%rdx
   0x00005555555555af <+116>: add    $0x1,%eax
   0x00005555555555b2 <+119>: cmp    %ecx,%eax
```

```
   0x00005555555555b4 <+121>: jne    0x5555555555ab <phase_6+112>
   0x00005555555555b6 <+123>: mov    %rdx,0x20(%rsp,%rsi,8)
   0x00005555555555bb <+128>: add    $0x1,%rsi
   0x00005555555555bf <+132>: cmp    $0x6,%rsi
   0x00005555555555c3 <+136>: je     0x5555555555e2 <phase_6+167>
   0x00005555555555c5 <+138>: mov    (%rsp,%rsi,4),%ecx
   0x00005555555555c8 <+141>: mov    $0x1,%eax
   0x00005555555555cd <+146>: lea    0x202c3c(%rip),%rdx        #
0x555555758210 <node1>
   0x00005555555555d4 <+153>: cmp    $0x1,%ecx
   0x00005555555555d7 <+156>: jg     0x5555555555ab <phase_6+112>
   0x00005555555555d9 <+158>: jmp    0x5555555555b6 <phase_6+123>
   0x00005555555555db <+160>: mov    $0x0,%esi
   0x00005555555555e0 <+165>: jmp    0x5555555555c5 <phase_6+138>
   0x00005555555555e2 <+167>: mov    0x20(%rsp),%rbx
   0x00005555555555e7 <+172>: mov    0x28(%rsp),%rax
   0x00005555555555ec <+177>: mov    %rax,0x8(%rbx)
   0x00005555555555f0 <+181>: mov    0x30(%rsp),%rdx
   0x00005555555555f5 <+186>: mov    %rdx,0x8(%rax)
   0x00005555555555f9 <+190>: mov    0x38(%rsp),%rax
   0x00005555555555fe <+195>: mov    %rax,0x8(%rdx)
   0x0000555555555602 <+199>: mov    0x40(%rsp),%rdx
   0x0000555555555607 <+204>: mov    %rdx,0x8(%rax)
   0x000055555555560b <+208>: mov    0x48(%rsp),%rax
   0x0000555555555610 <+213>: mov    %rax,0x8(%rdx)
   0x0000555555555614 <+217>: movq   $0x0,0x8(%rax)
   0x000055555555561c <+225>: mov    $0x5,%ebp
   0x0000555555555621 <+230>: jmp    0x55555555562c <phase_6+241>
   0x0000555555555623 <+232>: mov    0x8(%rbx),%rbx
   0x0000555555555627 <+236>: sub    $0x1,%ebp
   0x000055555555562a <+239>: je     0x55555555563d <phase_6+258>
   0x000055555555562c <+241>: mov    0x8(%rbx),%rax
   0x0000555555555630 <+245>: mov    (%rax),%eax
   0x0000555555555632 <+247>: cmp    %eax,(%rbx)
   0x0000555555555634 <+249>: jge    0x555555555623 <phase_6+232>
   0x0000555555555636 <+251>: callq  0x55555555589f <explode_bomb>
   0x000055555555563b <+256>: jmp    0x555555555623 <phase_6+232>
   0x000055555555563d <+258>: mov    0x58(%rsp),%rax
   0x0000555555555642 <+263>: xor    %fs:0x28,%rax
   0x000055555555564b <+272>: jne    0x555555555658 <phase_6+285>
   0x000055555555564d <+274>: add    $0x68,%rsp
   0x0000555555555651 <+278>: pop    %rbx
   0x0000555555555652 <+279>: pop    %rbp
   0x0000555555555653 <+280>: pop    %r12
   0x0000555555555655 <+282>: pop    %r13
   0x0000555555555657 <+284>: retq
   0x0000555555555658 <+285>: callq  0x555555554e50 <__stack_chk_fail@plt>
End of assembler dump.
```