

CSED 211 BOMBLAB REPORT

20230024 문요준

Phase 1 (답: I can see Russia from my house!)

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x0000000000400ef0 <+0>: sub $0x8,%rsp
0x0000000000400ef4 <+4>: mov $0x402620,%esi
0x0000000000400ef9 <+9>: callq 0x4013ae <strings_not_equal>
0x0000000000400efe <+14>: test %eax,%eax
0x0000000000400f00 <+16>: je 0x400f07 <phase_1+23>
0x0000000000400f02 <+18>: callq 0x401614 <explode_bomb>
0x0000000000400f07 <+23>: add $0x8,%rsp
0x0000000000400f0b <+27>: retq
End of assembler dump.

(gdb) disas strings_not_equal
Dump of assembler code for function strings_not_equal:
0x000000000040130e <+0>: push %r12
0x0000000000401310 <+2>: push %rbp
0x0000000000401311 <+3>: push %rbx
0x0000000000401312 <+4>: mov %rdi,%rbx
0x0000000000401315 <+7>: mov %rsi,%rbp
0x0000000000401318 <+10>: callq 0x401391 <string_length>
0x000000000040131d <+15>: mov %eax,%r12d
0x000000000040131e <+16>: mov %rbp,%rdi
0x000000000040131f <+17>: callq 0x401391 <string_length>
0x0000000000401320 <+18>: mov $0x1,%edx
0x0000000000401321 <+19>: cmp %eax,%r12d
0x0000000000401322 <+20>: jne 0x401410 <strings_not_equal+98>
0x0000000000401323 <+21>: movzbl (%rbx),%eax
0x0000000000401324 <+22>: test %al,%al
0x0000000000401325 <+23>: je 0x4013fd <strings_not_equal+79>
0x0000000000401326 <+24>: cmp 0x0(%rbp),%al
0x0000000000401327 <+25>: je 0x4013e7 <strings_not_equal+57>
0x0000000000401328 <+26>: xchg %ax,%ax
0x0000000000401329 <+27>: jmp 0x401404 <strings_not_equal+86>
0x000000000040132a <+28>: cmp 0x0(%rbp),%al
0x000000000040132b <+29>: jne 0x40140b <strings_not_equal+93>
0x000000000040132c <+30>: add $0x1,%rbx
0x000000000040132d <+31>: add $0x1,%rbp
0x000000000040132e <+32>: orq %rax,%rax
0x000000000040132f <+33>: movzbl (%rbx),%eax
0x0000000000401330 <+34>: test %al,%al
0x0000000000401331 <+35>: jne 0x4013e2 <strings_not_equal+52>
0x0000000000401332 <+36>: mov $0x0,%edx
0x0000000000401333 <+37>: jmp 0x401410 <strings_not_equal+98>
0x0000000000401334 <+38>: mov $0x0,%edx
0x0000000000401335 <+39>: jmp 0x401410 <strings_not_equal+98>
0x0000000000401336 <+40>: mov $0x1,%edx
0x0000000000401337 <+41>: jmp 0x401410 <strings_not_equal+98>
0x0000000000401338 <+42>: mov $0x1,%edx
0x0000000000401339 <+43>: jmp 0x401410 <strings_not_equal+98>
0x000000000040133a <+44>: mov $0x1,%edx
0x000000000040133b <+45>: jmp 0x401410 <strings_not_equal+98>
0x000000000040133c <+46>: mov $0x1,%edx
0x000000000040133d <+47>: jmp 0x401410 <strings_not_equal+98>
0x000000000040133e <+48>: mov $0x1,%edx
0x000000000040133f <+49>: jmp 0x401410 <strings_not_equal+98>
0x0000000000401340 <+50>: mov $0x1,%edx
0x0000000000401341 <+51>: jmp 0x401410 <strings_not_equal+98>
0x0000000000401342 <+52>: pop %rbx
0x0000000000401343 <+53>: pop %rbp
0x0000000000401344 <+54>: pop %r12
0x0000000000401345 <+55>: retq
End of assembler dump.

(gdb) disas string_length
Dump of assembler code for function string_length:
0x0000000000401391 <+0>: cmpb $0x0,(%rdi)
0x0000000000401392 <+1>: je 0x4013a8 <string_length+23>
0x0000000000401393 <+2>: mov %rdi,%rdx
0x0000000000401394 <+3>: add $0x1,%rdx
0x0000000000401395 <+4>: mov %rdi,%eax
0x0000000000401396 <+5>: sub %rdi,%eax
0x0000000000401397 <+6>: cmpb $0x0,(%rdi)
0x0000000000401398 <+7>: je 0x4013a8 <string_length+23>
0x0000000000401399 <+8>: mov %rdi,%eax
0x000000000040139a <+9>: add $0x1,%rdx
0x000000000040139b <+10>: mov %rdi,%eax
0x000000000040139c <+11>: sub %rdi,%eax
0x000000000040139d <+12>: cmpb $0x0,(%rdi)
0x000000000040139e <+13>: je 0x4013a8 <string_length+23>
0x000000000040139f <+14>: mov %rdi,%eax
0x00000000004013a0 <+15>: add $0x1,%rdx
0x00000000004013a1 <+16>: mov %rdi,%eax
0x00000000004013a2 <+17>: sub %rdi,%eax
0x00000000004013a3 <+18>: cmpb $0x0,(%rdi)
0x00000000004013a4 <+19>: je 0x4013a8 <string_length+23>
0x00000000004013a5 <+20>: repz retq
0x00000000004013a6 <+21>: mov $0x0,%eax
0x00000000004013a7 <+22>: retq
End of assembler dump.
```

<phase_1>

<+0>에서 스택 포인터를 8바이트 줄이고 \$0x402620 주소를 esi 레지스터에 저장한다. 그리고 strings_not_equal 함수를 호출한다.

<strings_not_equal>

함수 strings_not_equal에서는 두 인자(rdi, rsi)를 받아 각각 rbx, rbp 레지스터에 할당한다. 그리고 string_length 함수를 호출한다.

<string_length>

rdi 레지스터가 가리키는 메모리 주소 값을 NULL값과 비교해서 길이가 0인지 체크하고 0이 아니면 포인터(rdx 레지스터)에 1을 더하고 문자열의 다음 문자로 이동하면서(<+8>에서) 최종적으로 시작 주소와 끝 주소의 차이를 계산(<+14>에서), 문자열의 길이를 반환하는 함수임을 알 수 있다. 다시 strings_not_equal로 돌아가면, string_length를 <+21>

에서 한 번 더 호출하여 문자열의 길이를 알아냄을 알 수 있는데, 여기서 즉 두 문자열 (입력 문자열, 정답 문자열)의 길이를 비교함(<+31>에서) 알 수 있다.

맨처음에 문자열 인자를 두 개 받았으므로 이 인자 레지스터의 값을 확인해보면

```
(gdb) x/s $rbx
0x6047c0 <input_strings>: "dafsdaf"
(gdb) x/s $rbp
0x402620: "I can see Russia from my house!"
```

입력한 문자열과 정답 문자열이 출력되는 것을 확인할 수 있다.

따라서 올바른 입력값은 "I can see Russia from my house!" 이다.

Phase 2 (답: 1 2 4 8 16 32)

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
0x00000000400f0c <+0>: push %rbp
0x00000000400f0d <+1>: push %rbx
0x00000000400f0e <+2>: sub $0x28,%rsp (스택 40바이트 감소)
0x00000000400f12 <+6>: mov %rsp,%rsi (rsp 값을 rsi에 저장)
0x00000000400f15 <+9>: callq 0x4016fe <read_six_numbers>
0x00000000400f1a <+14>: cmpl $0x1, (%rsp) (스택의 저장된 정답과 숫자가 1인지 비교)
0x00000000400f1e <+18>: je 0x400f40 <phase_2+52> (이 아니면 폭발 명령. <+22>로 이동)
0x00000000400f20 <+20>: callq 0x401614 <explode_bomb>
0x00000000400f25 <+25>: jmp 0x400f40 <phase_2+52>
0x00000000400f27 <+27>: mov -0x4(%rbx),%eax (ebx에 rbx-4 (현재 숫자 이전) 저장)
0x00000000400f2a <+30>: add %eax,%eax (eax를 2배로 만든다)
0x00000000400f2c <+32>: cmp %eax, (%rbx) (eax (숫자 2배)와 rbx 비교)
0x00000000400f2e <+34>: je 0x400f35 <phase_2+41> (값이 같으면 41로 이동)
0x00000000400f30 <+36>: callq 0x401614 <explode_bomb>
0x00000000400f35 <+41>: add $0x4,%rbx (rbx에 다음 숫자 할당)
0x00000000400f39 <+45>: cmp %rbp,%rbx (현재 숫자와 (첫 번째 숫자)인지 비교)
0x00000000400f3c <+48>: jne 0x400f27 <phase_2+27>
0x00000000400f3e <+50>: jmp 0x400f4c <phase_2+64>
0x00000000400f40 <+52>: lea 0x4(%rsp),%rbx (rsp(현재 위치) 다음 숫자 저장)
0x00000000400f45 <+57>: lea 0x18(%rsp),%rbp (rsp에 24바이트 → 정답 저장)
0x00000000400f4a <+62>: jmp 0x400f27 <phase_2+27> (입력 숫자 (현재 번째 숫자) 저장)
0x00000000400f4c <+64>: add $0x28,%rsp (스택 40바이트 복원)
0x00000000400f50 <+68>: pop %rbx
---Type <return> to continue, or q <return> to quit---
0x00000000400f51 <+69>: pop %rbp
0x00000000400f52 <+70>: retq
End of assembler dump.

(gdb) disas read_six_numbers
Dump of assembler code for function read_six_numbers:
0x000000004016fe <+0>: sub $0x18,%rsp (스택에 24바이트 할당)
0x00000000401702 <+4>: mov %rsi,%rdx (rdx에 rsi (첫 번째 숫자)
0x00000000401705 <+7>: lea 0x4(%rsi),%rcx (rcx에 rsi의 4바이트 주소 → 두 번째 숫자 주소)
0x00000000401709 <+11>: lea 0x14(%rsi),%rax (rax에 rsi+20 주소 → 여섯 번째 숫자 주소)
0x0000000040170d <+15>: mov %rax,0x8(%rsp) (여섯 번째 주소 스택에 저장)
0x00000000401712 <+20>: lea 0x10(%rsi),%rax (rax에 rsi+16 주소 → 다섯 번째 숫자 주소)
0x00000000401716 <+24>: mov %rax, (%rsp) (다섯 번째 주소 스택에 저장)
0x0000000040171a <+28>: lea 0xc(%rsi),%r9 (r9에 rsi+12 주소 → 네 번째 숫자 주소)
0x0000000040171e <+32>: lea 0x8(%rsi),%r8 (r8에 rsi+8 주소 → 세 번째 숫자 주소)
0x00000000401722 <+36>: mov $0x4033e8,%esi (esi에 'I can see Russia from my house!' 문자열 저장)
0x00000000401727 <+41>: mov $0x0,%eax (eax에 0 저장)
0x0000000040172c <+46>: callq 0x400c30 <__isoc99_sscanf@plt>
0x00000000401731 <+51>: cmp $0x5,%eax (eax에 5가 계속 확인)
0x00000000401734 <+54>: jg 0x40173b <read_six_numbers+61> (5보다 많거나
0x00000000401736 <+56>: callq 0x401614 <explode_bomb> (올바르지 않으면 폭발)
0x0000000040173b <+61>: add $0x18,%rsp
0x0000000040173f <+65>: nop
0x00000000401740 <+66>: retq
End of assembler dump.
```

<phase_2>

처음에 스택에 메모리를 확보하고 스택에 저장된 데이터를 rsi 레지스터로 접근하도록 설정함을 알 수 있다. 그리고 read_six_numbers를 호출한다.

<read_six_numbers>

<+0>에서 스택에 24바이트를 확보함, 함수의 이름에서 정수(4바이트) 6개를 스택에 할당하려는 것으로 유추할 수 있다. 이후의 코드에서는 lea 명령어를 이용해 주소 연산 후 n 번째 정수를 담은 주소들을 각각 레지스터에 저장함을 알 수 있고, <+51>에서 sscanf 함수로 읽은 정수의 개수를 확인한 다음 6개의 숫자를 모두 읽었다면 폭탄을 터뜨리지 않고 phase_2로 복귀한다.

1 2 3 4 5 6를 입력 후 확인해보면, 값이 스택에 잘 저장되어 있음을 확인할 수 있다.

```
(gdb) x/d $rsp
0x7fffffffefc0: 1
(gdb) x/d $rsp+4
0x7fffffffefc4: 2
(gdb) x/d $rsp+8
0x7fffffffefc8: 3
(gdb) x/d $rsp+12
0x7fffffffefcc: 4
(gdb) x/d $rsp+16
0x7fffffffefd0: 5
(gdb) x/d $rsp+20
0x7fffffffefd4: 6
```

phase_2 <+14>에서 스택 첫 번째 숫자가 1인지 확인하고 그렇지 않으면 폭탄을 터뜨리므로, 첫 번째 숫자는 1이어야 한다. <+52>에서 rbx에 그 다음 두 번째 숫자(rsp에서 4바이트 떨어진)를 저장, <+57>에서 rbp에 여섯 번째 숫자를 저장하고 <+48>에서 <+27>로 돌아가서 eax에 rbx 4바이트 전, 즉 다시 첫 번째 숫자를 할당한다. 이때 eax에 eax를 더해 eax를 두배로 만들고 이를 rbx와 비교함을 알 수 있는데, 두 번째 숫자가 첫 번째 숫자의 2배가 되어야 함을 의미한다. 다시 <+41>에서 rbx에 그 다음 숫자를 저장하고 <+45>에서 rbx와 rbp가 같은지 확인(루프 종료 기준, 마지막 숫자인지)하고 다시 <+27>로 돌아가 반복하고 있음을 알 수 있다. 즉, n+1번째 숫자는 n번째 숫자의 두 배임을 확인하는 로직이라고 유추할 수 있다.

따라서 올바른 입력값은 1 2 4 8 16 32이다.

<phase_3>

<+0>에서 스택 포인터를 24바이트 줄이고, <+14>까지 rdx, rcs, r8에 각각 rsp부터 8바이트, 7바이트, 12바이트 떨어진 메모리 주소 3개를 할당하고 있다. <+19>에서 0x402666가 가리키는 값을 출력한 결과

```
(gdb) x/s 0x402666
0x402666: "%d %c %d"
```

"%d %c %d"로, 밑의 sscanf 함수에서 int char int 로 이루어진 입력을 받는 것을 유추할 수 있다. sscanf는 입력값의 개수를 반환하기에, 이 개수는 eax에 저장되어있고 <+34>에서 eax가 2보다 커야 폭탄이 터지지 않기 때문에 int char int 세 개의 입력(1 'a' 10)을 넣은 후 추론을 진행했다. 스택을 출력해본 결과

```
(gdb) x/d $rsp+12
0x7fffffffefec: 1
(gdb) x/d $rsp+8
0x7fffffffef8: 10
(gdb) x/s $rsp+7
0x7fffffffef7: "a\n"
```

값이 잘 들어간 것을 확인할 수 있다. <+44>에서 rsp+12, 즉 첫 번째 정수가 7보다 크면 폭탄이 터지므로 첫 번째 정수는 7보다 같거나 작아야 한다. <+55>에서 eax에 첫 번째 정수를 저장한 다음, <+59>에서 rax값에 8을 곱하고 0x402680에 더한 주소로 점프하는 것을 알 수 있는데, 점프 테이블을 출력하면 다음과 같다.

```
(gdb) x/10gx 0x402680
0x402680: 0x0000000000400f95 0x0000000000400fb7
0x402690: 0x0000000000400fd9 0x0000000000400ffb
0x4026a0: 0x000000000040101a 0x0000000000401035
0x4026b0: 0x0000000000401050 0x000000000040106b
0x4026c0 <array.3162>: 0x0000000a00000002 0x0000000100000006
```

주소는 400f95 400fb7 400fd9 ~ 40106b 순서로 <+66>, <+100>, <+134>, <+168>, <+199>, <+226>, <+253>, <+280>에 대응되며, case문이라고 유추할 수 있다.

각 case에서 특정 숫자와 입력한 숫자(rsp+8)를 비교한 후, 같으면 <+317>로 이동, 아니면 폭탄이 터지도록 되어있다. 각 case 별로 숫자 2개를 순서쌍으로 적었을 때 (0,616), (1, 655), (2,895), (3,639), (4,194), (5,339), (6,614), (7,977)이다.

<+317>로 이동하면 입력한 문자(rsp+7)과 al에 담긴 문자를 비교함을 알 수 있다. al 레지스터는 eax의 하위 비트를 저장하므로, eax 레지스터에 담긴 값을 알면 유추할 수 있다. 각각의 케이스에서 순서대로 0x74, 0x65, 0x62, 0x7a, 0x78, 0x61, 0x62, 0x6d가 할당되는데 아스키코드 변환 시 t, e, b, z, x, a, b, m에 해당한다. 따라서 정답은 (0,t,616), (1,e, 655), (2,b,895), (3,z,639), (4,x,194), (5,a,339), (6,b,614), (7,m,977) 중 하나이다.

Phase 4 (답: 10 5)

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
0x00000000004010de <+0>: sub    $0x18,%rsp
0x00000000004010e2 <+4>: lea    0x8(%rsp),%rcx
0x00000000004010e7 <+9>: lea    0xc(%rsp),%rdx
0x00000000004010ec <+14>: mov    $0x4033f4,%esi
0x00000000004010f1 <+19>: mov    $0x0,%eax
0x00000000004010f6 <+24>: callq 0x400c30<...sscanf@plt>
0x00000000004010fb <+29>: cmp    $0x2,%eax
0x00000000004010fe <+32>: jne    0x401107<phase_4+1>
0x0000000000401100 <+34>: cmpl   $0xc,%esi
0x0000000000401105 <+39>: jbe    0x40110c<phase_4+4>
0x0000000000401107 <+41>: callq 0x401614<explode_bomb>
0x000000000040110c <+46>: mov    $0x0,%edi
0x0000000000401111 <+51>: mov    0xc(%rsp),%edi
0x0000000000401116 <+56>: callq 0x4010a0<func4>
0x000000000040111f <+65>: cmp    $0x5,%eax
0x0000000000401122 <+68>: jne    0x40112b<phase_4+7>
0x0000000000401124 <+70>: cmpl   $0x5,0x8(%rsp)
0x0000000000401129 <+75>: je     0x401130<phase_4+8>
0x000000000040112b <+77>: callq 0x401614<explode_bomb>
0x0000000000401130 <+82>: add    $0x18,%rsp
0x0000000000401134 <+86>: retq
End of assembler dump.
```

func4(10, 0, 14)

$$eax = edx - esi = 14 - 0 = 14$$

$$ecx = eax / 2 = 7$$

$$ecx = rcx + esi = 7 + 0 = 7$$

$$edx > ecx, \quad edx = ecx + 1 = 8$$

func4(10, 8, 14)

$$eax = edx - esi = 14 - 8 = 6$$

$$ecx = 6 / 2 = 3$$

$$ecx = rcx + esi = 3 + 8 = 11$$

$$edx < ecx, \quad \text{if } 0, \quad edx = ecx + 1 = 10$$

func4(10, 8, 10)

$$eax = edx - esi = 10 - 8 = 2$$

$$ecx = 2 / 2 = 1$$

$$ecx = rcx + esi = 1 + 8 = 9$$

$$edx > ecx, \quad \text{if } 1, \quad edx = ecx + 1 = 10$$

func4(10, 10, 10)

$$eax = edx - esi = 10 - 10 = 0$$

$$ecx = 0 / 2 = 0$$

$$ecx = rcx + esi = 0 + 10 = 10$$

$$10 \neq 0 \rightarrow 0 \text{ 번째 반복}$$

func4(10, 10, 10)

$$\text{if } (5) \text{ times } \text{if } (rcx, rcx, 1), \text{ call } \text{func4}(rcx, rcx, 1)$$

$$\text{if } (5) \text{ times } \text{if } (rcx, rcx, 1), \text{ call } \text{func4}(rcx, rcx, 1)$$

$$\text{if } (5) \text{ times } \text{if } (rcx, rcx, 1), \text{ call } \text{func4}(rcx, rcx, 1)$$

$$\text{if } (5) \text{ times } \text{if } (rcx, rcx, 1), \text{ call } \text{func4}(rcx, rcx, 1)$$

$$\text{if } (5) \text{ times } \text{if } (rcx, rcx, 1), \text{ call } \text{func4}(rcx, rcx, 1)$$

<phase_4>

<+0>에서 스택 포인터를 24바이트 줄이고, <+9>까지 rcx, rdx에 각각 rsp부터 8바이트, 12바이트 떨어진 메모리 주소 2개를 할당하고 있다. <+14>에서 0x4033f4가 가리키는 값을 출력한 결과

```
(gdb) x/s 0x4033f4
0x4033f4: "%d %d"
```

sscanf에서 정수 두 개를 입력받는다라는 것을 알 수 있다.

임의로 정수 9와 6을 입력하였고 rsp+12에 첫 번째 정수, rsp+8에 두 번째 정수가 들어감을 알 수 있다.

```
(gdb) x/d $rsp+12
0x7fffffffef1bc: 9
(gdb) x/d $rsp+8
0x7fffffffef1b8: 6
```

sscanf는 입력값의 개수를 반환하기 위해, 이 개수는 eax에 저장되어 있고 <+29>에서 정수 2와 비교한다. 이때 $eax \neq 2$ 이면 폭탄이 터지므로, 정수 2개를 입력하는 것이 맞다는 것을 알 수 있다. <+34>에서 14와 rsp+12에 담긴 정수를 비교하는데, 첫 번째 입력값이 14 이하일 경우 <+46>으로 이동하고 그렇지 않은 경우 폭탄이 터지는 것을 알 수 있다. <+46>, <+51>, <+56>에서는 edx에 14(상한값), esi에 0, edi에 rsp+12를 할당하고, func4를

호출한다. func4를 disassemble하면 다음과 같다.

<func4>

<+4>에서 eax에 상한값 14를 할당하고 <+6>에서 $eax = 14 - 0 = 14$ 가 다시 할당된다. 그리고 <+8>에서 ecx에 eax의 값이 할당된다. <+10>에서는 ecx의 값을 오른쪽으로 31비트 시프트하는데, 최상위 비트(부호 비트)만 남게 되며 따라서 $ecx = 0$ 이다. <+13>에서는 $eax = eax + ecx$, eax는 그대로이다. <+15>에서 sar은 오른쪽으로 1비트 산술 시프트를 수행하는데, 최종적으로 2로 나누는 효과를 가지므로 $eax = eax / 2 = 7$ 이다. <+17>에서 $ecx = rax + rsi = 7 + 0 = 7$ 이 할당된다. <+20>에서 edi(rsp+12, 첫 번째 정수)와 ecx(직전 계산으로 7이 저장되어 있음)을 비교하고 만약 edi와 ecx가 같으면 jge, jle를 모두 충족, <+36>, <+57>, <+61>로 이동해 함수가 반환된다. 현재는 같지 않으므로, <+45>에서 lea 연산으로 $esi = rcx + 1 = 7 + 1 = 8$ 이 계산된다. 그리고 <+48>에서 func4를 다시 호출하는 재귀함수임을 알 수 있다. 여기서 func4가 상한값을 줄이거나 하한값을 늘리는 방법으로 범위를 반씩 좁혀가며 목표값을 찾는다는 것에서 이진 탐색 알고리즘임을 유추할 수 있다.

$eax = edx - esi$, $ecx = (eax / 2) + esi$, edi ecx 비교 후 edi가 더 클 때: $esi = ecx + 1$ | ecx가 더 클 때: $edx = ecx - 1 \rightarrow$ 이 과정이 func4 내의 알고리즘이다.

eax	반환값
edi	첫 번째 입력 값,
edx	상한값
ecx	$eax / 2 + esi$
esi	하한값

값이 변하며, 변한 값이 다음 재귀함수 계산에 쓰이는 edi, esi, edx를 인자로 func4를 $func4(edi, esi, edx)$ 라 하자. 입력값을 찾는 것이 목표이므로, 역계산을 위해 반환값을 구하기 위해 func4에 돌아가면, <+65>에서 eax과 5와 비교하는 것으로, func4 반환값이 5이어야 함을 유추할 수 있다. 그리고 <+70>에서 phase_4 입력값이 5이어야함을 알 수 있다. func4에 14부터 숫자를 넣어 계산을 해본 결과, 10을 넣었을 때

첫 번째 호출: $func4(10, 0, 14)$

$eax = edx - esi = 14 - 0 = 14$

$eax = (eax / 2) + esi = (14 / 2) + 0 = 7$

$edi(10) > eax(7)$ 이므로, $esi = ecx + 1 = 8$

재귀 호출: $func4(10, 8, 14) \rightarrow func4(10, 8, 10) \rightarrow func4(10, 10, 10)$, $func4(10, 10, 10)$ 에서 0이 반환되었으며 세 번째 호출, 두 번째 호출, 첫 번째 호출로 돌아갔을 때 5를 반환함을 확인할 수 있었다. 따라서 입력값이 10이어야 5가 반환된다. 따라서 올바른 입력값은 10이다.

Phase 5 (답: pqryxv (순서 상관 X, 하위 4비트가 0,1,2,9,8,6인 문자))

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
0x0000000000401135 <+0>:      push    %rbx
0x0000000000401136 <+1>:      mov     %rdi,%rbx  rbx에 rdi 값 저장
0x0000000000401139 <+4>:      callq  0x401391 <string_length>
0x000000000040113e <+9>:      cmp     $0x6,%eax  edx(문자열)과 6 비교
0x0000000000401141 <+12>:     je      0x401148 <phase_5+19>  같으면 <+19>로 이동, 아니면
0x0000000000401143 <+14>:     callq  0x401614 <explode_bomb>  폭탄
0x0000000000401148 <+19>:     mov     $0x0,%eax  eax에 0 저장
0x000000000040114d <+24>:     mov     $0x0,%edx  edx에 0 저장
0x0000000000401152 <+29>:     movzbl (%rbx,%rax,1),%ecx  rbx+rax의 주소에서 1바이트를 읽어 ecx에 저장
0x0000000000401156 <+33>:     and     $0xf,%ecx  ecx를 하위 4비트만 남김  → rax는 인덱스 역할, 인덱스에 해당하는 문자를
0x0000000000401159 <+36>:     add     0x4026c0(,%rcx,4),%edx  rcx(ecx)의 값을  불러옴.
0x0000000000401160 <+43>:     add     $0x1,%rax  rax(인덱스) += 1  인덱스로 주소(배열)에서 값을 읽어 edx에
0x0000000000401164 <+47>:     cmp     $0x6,%rax  rax과 6 비교  더함.
0x0000000000401168 <+51>:     jne     0x401152 <phase_5+29>  6이 아니면 루프 반복 6이랑 다름으로
0x000000000040116a <+53>:     cmp     $0x26,%edx  edx와 38 비교
0x000000000040116d <+56>:     je      0x401174 <phase_5+63>  이수면 폭탄 맞으면 다음으로 넘어감.
0x000000000040116f <+58>:     callq  0x401614 <explode_bomb>
0x0000000000401174 <+63>:     pop     %rbx
0x0000000000401175 <+64>:     retq

End of assembler dump.
```

<phase_5>

<+0>, <+1>에서 rbx의 값을 스택에 저장, 인자 rdi의 값을 rbx로 복사한다. 그리고 <+4>에서 string_length 함수를 호출해서 반환값 eax를 <+9>에서 6과 비교한다. 문자열의 길이가 6이 아니면 폭탄이 터지므로, 입력값의 문자열 길이는 6이어야함을 알 수 있다. <+29>에서는 문자열의 인덱스 숫자(rax)에 위치한 문자를 문자열에서 읽어와서 (메모리 주소를 rbx + rax로 계산) ecx에 저장한다. <+33>에서 0xf는 하위 4비트가 1111로, 다른 값과 and 연산을 수행할 시 다른 값의 하위 4비트만 남게 된다. 따라서 ecx에는 ecx의 하위 4비트만 남는다. <+36>에서는 기본 주소 0x4026c0에 rcx(ecx) * 4 만큼을 더한 주소가 가리키는 값을 edx에 더하게 되는데, 기본 주소가 가리키는 것이 배열임을 유추할 수 있다. <+43>에서 rax에 1을 더하고 <+47>에서 rax가 6인지 확인하기 때문에 6번 반복하는 반복문인 것을 알 수 있으며, <+53>에서 edx가 38이어야 폭탄이 터지지 않기 때문에 배열을 인덱싱한 값 6개를 모두 더한 값이 38이어야함을 알 수 있다.

<0x4026c0>

```
(gdb) x/64xb 0x4026c0
0x4026c0 <array.3162>: 0x02 0x00 0x00 0x00 0x0a 0x00 0x00 0x00
0x4026c8 <array.3162+8>: 0x06 0x00 0x00 0x00 0x01 0x00 0x00 0x00
0x4026d0 <array.3162+16>: 0x0c 0x00 0x00 0x00 0x18 0x00 0x00 0x00
0x4026d8 <array.3162+24>: 0x09 0x00 0x00 0x00 0x03 0x00 0x00 0x00
0x4026e0 <array.3162+32>: 0x04 0x00 0x00 0x00 0x07 0x00 0x00 0x00
0x4026e8 <array.3162+40>: 0x0e 0x00 0x00 0x00 0x05 0x00 0x00 0x00
0x4026f0 <array.3162+48>: 0x0b 0x00 0x00 0x00 0x08 0x00 0x00 0x00
0x4026f8 <array.3162+56>: 0x0f 0x00 0x00 0x00 0x0d 0x00 0x00 0x00
```

index	value	index	value	index	value	index	value
0	2	4	12	8	4	12	11
1	10	5	16	9	7	13	8
2	6	6	9	10	14	14	15
3	1	7	3	11	5	15	13

배열에서 value 6개가 2, 10, 6, 7, 4, 9가 뽑혀야 합이 38이 되며, 해당 인덱스는 0, 1, 2, 9, 8, 6이다. 하위 4비트가 0,1,2,9,8,6인 문자는 p(112),q(113),r(114),y(121),x(120),v(118)이 있으며 문자의 순서와 상관없이 p, q, r, y, x, v 이 모두 하나씩 들어간 문자열이 정답이다.

Phase 6 (정답: 2 4 5 3 6 1)

phase_6를 disassemble하면 다음과 같다.

<phase_6>

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
0x0000000000401176 <+0>: push %r14
0x0000000000401178 <+2>: push %r13
0x000000000040117a <+4>: push %r12
0x000000000040117c <+6>: push %rbp
0x000000000040117d <+8>: push %rbx
0x000000000040117e <+10>: sub $0x50,%rsp
0x0000000000401182 <+12>: lea 0x30(%rsp),%r13
0x0000000000401187 <+17>: mov %r13,%rsi
0x000000000040118a <+20>: callq 0x4016fe <read_six_numbers>
0x000000000040118f <+25>: mov %r13,%r14
0x0000000000401192 <+28>: mov $0x0,%r12d
0x0000000000401198 <+34>: mov %r13,%rbp
0x000000000040119b <+37>: mov 0x0(%r13),%eax
0x000000000040119f <+41>: sub $0x1,%eax
0x00000000004011a2 <+44>: cmp $0x5,%eax
0x00000000004011a5 <+47>: jbe 0x4011ac <phase_6+54>
0x00000000004011a7 <+49>: callq 0x401614 <explode_bomb>
0x00000000004011aa <+54>: add $0x1,%r12d
0x00000000004011b0 <+58>: cmp $0x6,%r12d
0x00000000004011b4 <+62>: jg je 0x4011d8 <phase_6+98>
0x00000000004011b6 <+64>: mov %r12d,%ebx
0x00000000004011b9 <+67>: movslq %ebx,%rax
0x00000000004011bc <+70>: mov 0x30(%rsp,%rax,4),%eax
0x00000000004011c0 <+74>: cmp %eax,0x0(%rbp)
0x00000000004011c3 <+77>: jne 0x4011ca <phase_6+84>
0x00000000004011c5 <+79>: callq 0x401614 <explode_bomb>
0x00000000004011c8 <+84>: add $0x1,%ebx
0x00000000004011cd <+87>: cmp $0x5,%ebx
0x00000000004011d0 <+90>: jle 0x4011b9 <phase_6+67>
0x00000000004011d2 <+92>: add $0x4,%r13
0x00000000004011d6 <+96>: jmp 0x401198 <phase_6+34>
0x00000000004011d8 <+98>: lea 0x48(%rsp),%rsi
0x00000000004011dd <+103>: mov %r14,%rax
0x00000000004011e0 <+106>: mov $0x7,%ecx
0x00000000004011e5 <+111>: mov %ecx,%edx
0x00000000004011e7 <+113>: sub 0x1(%rax),%edx
0x00000000004011e9 <+115>: mov %edx,%rax
0x00000000004011eb <+117>: add $0x4,%rax
0x00000000004011ef <+121>: cmp %rsi,%rax
0x00000000004011f2 <+124>: jne 0x4011e5 <phase_6+111>
0x00000000004011f4 <+126>: mov $0x0,%rsi
0x00000000004011f9 <+131>: jmp 0x40121b <phase_6+165>
0x00000000004011fb <+133>: mov 0x8(%rdx),%rdx
0x00000000004011ff <+137>: add $0x1,%eax
0x0000000000401202 <+140>: cmp %ecx,%eax
0x0000000000401204 <+142>: jne 0x4011fb <phase_6+133>
0x0000000000401206 <+144>: jmp 0x4012d0 <phase_6+151>

---Type <return> to continue, or q <return> to quit---
0x0000000000401208 <+146>: mov $0x6042f0,%edx
0x000000000040120d <+151>: mov %rdx,(%rsp,%rsi,2)
0x0000000000401211 <+155>: add $0x4,%rsi
0x0000000000401215 <+159>: cmp $0x18,%rsi
0x0000000000401219 <+163>: je 0x401230 <phase_6+186>
0x000000000040121b <+165>: mov 0x30(%rsp,%rsi,1),%ecx
0x000000000040121f <+169>: cmp $0x1,%ecx
0x0000000000401222 <+172>: jle 0x401208 <phase_6+146>
0x0000000000401224 <+174>: mov $0x1,%eax
0x0000000000401229 <+179>: mov $0x6042f0,%edx
0x000000000040122e <+184>: jmp 0x4011fb <phase_6+133>
0x0000000000401230 <+186>: mov (%rsp),%rbx
0x0000000000401234 <+190>: lea 0x8(%rsp),%rax
0x0000000000401239 <+195>: lea 0x30(%rsp),%rsi
0x000000000040123e <+200>: mov %rbx,%rcx
0x0000000000401241 <+203>: mov (%rax),%rdx
0x0000000000401244 <+206>: mov %rdx,0x8(%rcx)
0x0000000000401248 <+210>: add $0x8,%rax
0x000000000040124c <+214>: cmp %rsi,%rax
0x000000000040124f <+217>: je 0x401256 <phase_6+224>
0x0000000000401251 <+219>: mov %rdx,%rcx
0x0000000000401254 <+222>: jmp 0x401241 <phase_6+203>
0x0000000000401256 <+224>: movq $0x0,0x8(%rdx)
0x000000000040125e <+232>: mov $0x5,%ebp
0x0000000000401263 <+237>: mov 0x8(%rbx),%rax
0x0000000000401267 <+241>: mov (%rax),%eax
0x0000000000401269 <+243>: cmp %eax,(%rbx)
0x000000000040126b <+245>: jge 0x401272 <phase_6+252>
0x000000000040126d <+247>: callq 0x401614 <explode_bomb>
0x0000000000401272 <+252>: mov 0x8(%rbx),%rbx
0x0000000000401276 <+256>: sub $0x1,%ebp
0x0000000000401279 <+259>: jne 0x401263 <phase_6+237>
0x000000000040127b <+261>: add $0x50,%rsp
0x000000000040127f <+265>: pop %rbx
0x0000000000401280 <+266>: pop %rbp
0x0000000000401281 <+267>: pop %r12
0x0000000000401283 <+269>: pop %r13
0x0000000000401285 <+271>: pop %r14
0x0000000000401287 <+273>: retq

---Type <return> to continue, or q <return> to quit---
End of assembler dump.
```

<+8>에서 스택에서 80바이트를 확보하고 read_six_numbers 함수로 사용자로부터 6개의 숫자를 읽어옴을 알 수 있다. <+34>에서 입력 받은 첫 번째 숫자를 eax로 복사하고 이 숫자가 <+41> ~ <+47>에서 이 숫자가 1에서 6사이에 있는지 확인한다. r12d는 1씩 더하면서 6과 비교한다는 점에서 루프 인덱스임을 유추할 수 있다. 이후, <+64>, <+67>에서 루프 인덱스 r12d의 값을 ebx, rax로 복사한 다음 <+70>에서 rsp에서 해당 루프 인덱스 만큼 떨어진 숫자를 eax에 복사한다. <+74>에서는 eax에 저장된 숫자와 rbp가 가리키는 메모리 주소의 값을 비교하는데, 두 값이 같으면 폭탄이 터지는 것으로 보아 루프를 돌며 6개의 숫자 중 같은 두 개의 숫자가 있는지 확인함을 알 수 있다. <+92>에서 r13에 4바이트를 더해 배열 내의 숫자를 다음 숫자로 변경하고 다시 이전 코드로 복귀하는 것으로 보아, <+96>까지 6개의 숫자가 1~6까지인지, 중복된 숫자가 있는지 확인함을 알 수

있다. <+98>에서 rsp에서 72바이트 떨어진 주소를 rsi에 저장하고 r14(입력된 숫자가 저장된 메모리 주소)를 rax에 복사하고 edx에서 rax가 가리키는 메모리 주소의 값을 빼서 edx에 7-입력값 의 값을 저장한다. rax에 edx의 값을 다시 저장하고 rax에 주소값 4를 더 함으로써 반복문으로 력한 값을 모두 7-입력값을 만듦을 확인할 수 있다. 이후 0x6042f0 주소를 edx에 저장하는데, 0x6042f0이 가리키는 값을 출력하면 다음과 같다.

<0x6042f0>

```
(gdb) x/32d 0x6042f0
0x6042f0 <node1>:    369    1    6308608 0
0x604300 <node2>:    758    2    6308624 0
0x604310 <node3>:    934    3    6308640 0
0x604320 <node4>:    492    4    6308656 0
0x604330 <node5>:    970    5    6308672 0
0x604340 <node6>:    281    6    0      0
```

인덱스와 함께 노드 값이 저장되어있음을 알 수 있다. 링크드 리스트의 첫 번째 주소를 edx에 저장하고 rsp에서 rsi * 2만큼 떨어진 위치에 rdx를 저장한다. <+155>에서 rsi를 4만큼 증가시키고 <+159>에서 24과 비교하는 것으로 보아 노드 인덱스로 쓰임을 알 수 있다. <+165>에서는 %rsp + 0x30 + %rsi로 주소 계산 후(스택에 저장된 사용자 입력값 중 노드 인덱스 rsi에 따라 값을 참조함) 계산된 메모리 주소에서 값을 읽어 ecx에 로드한다. 즉, 노드 인덱스 번호에 해당하는 순서에 위치한 사용자 입력값을 ecx에 로드하는 것이다. <+169>에서 만약 ecx가 1보다 크다면 <+203>, <+206>에서 0x6042f0 + 8*ecx 값을 저장한다. 즉, 사용자가 입력한 값을 7-입력값으로 변형 후, 이 변형된 입력값이 노드의 인덱스를 구하는데 사용되면서 노드들의 순서를 결정하고, 이 노드 인덱스에 따라 노드들이 다음 노드로 연결되면서 리스트에서 노드들이 배치되는 것이다. (ex: 위의 노드 표에서 사용자가 431265로 입력했다면 934->492->281->970->369->758로 배치) 반복문으로 노드를 모두 연결한 후, <+243>에서 rbx와 rbx+8=eax 값을 비교하는데, rbx가 eax보다 작으면 폭탄이 터지기에 rbx가 eax보다 크거나 같아야 함을 알 수 있다. 즉, 내림차순이어야 하는데 노드들의 숫자가 내림차순으로 배열되려면 970->934->758->492->369->281 이어야하며 인덱스 상으로 5->3->2->4->1->6이다. 7-입력값이 이 인덱스와 같으므로, 입력값은 2 4 5 3 6 1 이 정답이다.