x86 어셈블리 (2) + 퀴즈

함수/프로시저

어셈블리에서도 라벨로 특정 구역을 표시하고 call을 이용해 함수 사용

함수 : 서브루틴, 프로시저, 함수로 혼용되어 불린다.

스택 관련 명령어

스택이란?

자료 구조의 한 종류로, LIFO(후입선출) 방식으로 동작한다. 가장 나중에 삽입된 데이터가 가장 먼저 꺼내짐.

메모리 주소가 낮은 방향으로 확장한다. (아래로 자란다.) 데이터를 추가할 때마다 메모리 주소가 감소하며, 데이터를 제거하면 다시증가함.

스택 포인터 레지스터 : `rsp`

→ 스택의 가장 위를 가리키며, push 명령으롤 실행하면 rsp가 감소하면서 스택에 값이 저장됨

→ pop 실행하면 rsp 값이 증가하면서 스택에 저장되어 있던 값 꺼낼 수 있음.

push

스택에 값을 저장하는 명령어. rsp가 감소되며 스택에 값이 저장된다.

```
push val
```

val 값을 스택에 저장.

내부적으로 수행되는 연산 :

```
rsp -= 8
[rsp] = val
```

예제

```
[Register]
rsp = 0x7fffffffc400

[Stack]
0x7fffffffc400 | 0x0  ← rsp
0x7fffffffc408 | 0x0

[Code]
push 0x31337
```

결과

```
[Register]
rsp = 0x7fffffffc3f8

[Stack]
0x7fffffffc3f8 | 0x31337 ← rsp
0x7fffffffc400 | 0x0
0x7fffffffc408 | 0x0
```

rsp가 감소함. (아래로 자란다.)

pop

스택에서 값을 꺼내는 명령어. pop을 실행하면 rsp 가 증가하면서 스택에 저장되어 있던 값을 꺼낼 수 있다.

```
pop reg
```

스택 최상단에 저장된 값을 꺼내서 레지스터인 reg에 대입한다.

내부적으로 수행되는 연산

```
reg = [rsp]
rsp += 8
```

예제

```
[Register]
rax = 0
rsp = 0x7fffffffc3f8

[Stack]
0x7fffffffc3f8 | 0x31337 ← rsp
0x7fffffffc400 | 0x0
0x7fffffffc408 | 0x0

[Code]
pop rax
```

결과

```
[Register]
rax = 0x31337
rsp = 0x7fffffffc400

[Stack]
0x7fffffffc400 | 0x0 ← rsp
0x7fffffffc408 | 0x0
```

데이터가 빠져나가며 rsp 값이 증가. (위로 줄어들었음).

함수 호출 및 반환 관련 명령어

call

```
call addr
```

주소 값 addr에 위치한 프로시저 호출

배후 연산 :

```
push return_address ; 함수 실행이 끝난 뒤 돌아갈 코드의 주소
jmp addr
```

예제

```
[Register]
rip = 0x400000
rsp = 0x7fffffffc400

[Stack]
0x7fffffffc3f8 | 0x0
0x7fffffffc400 | 0x0 ← rsp

[Code]
0x400000 | call 0x401000  ← rip
0x400005 | mov esi, eax
...
0x401000 | push rbp
```

결과

```
[Register]
rip = 0x401000
rsp = 0x7fffffffc3f8

[Stack]
0x7fffffffc3f8 | 0x400005  ← rsp
0x7fffffffc400 | 0x0

[Code]
```

```
0x400000 | call 0x401000
0x400005 | mov esi, eax
...
0x401000 | push rbp  ← rip
```

`call 0x401000` 실행 뒤 스택 포인터 rsp 위치에 반환 주소인 `0x400005` 저장되어 있음, rip 레지스터가 `0x401000` 으로 바뀜

leave

프로시저 반환 전 스택 프레임을 정리하는 명령어

연산

```
mov rsp, rbp
pop rbp
```

예제

```
[Register]
rsp = 0x7fffffffc400
rbp = 0x7fffffffc480

[Stack]
0x7fffffffc400 | 0x0 ← rsp
...
0x7fffffffc480 | 0x7fffffffc500 ← rbp
0x7fffffffc488 | 0x31337

[Code]
leave
```

결과

```
[Register]
rsp = 0x7fffffffc488
rbp = 0x7fffffffc500

[Stack]
0x7fffffffc400 | 0x0
...
0x7fffffffc480 | 0x7fffffffc500
0x7fffffffc488 | 0x31337 ← rsp
...
0x7fffffffc500 | 0x7fffffffc550 ← rbp
```

ret

```
pop ret
```

rip 레지스터의 값을 return address로 변경

예제

```
[Register]
rsp = 0x7fffffffc488
rbp = 0x7fffffffc500

[Stack]
0x7fffffffc400 | 0x0
...
0x7fffffffc480 | 0x7fffffffc500
0x7fffffffc488 | 0x31337 ← rsp
...
0x7fffffffc500 | 0x7fffffffc550 ← rbp
```

결과

```
[Register]
rsp = 0x7fffffffc488
rbp = 0x7fffffffc500

[Stack]
0x7fffffffc400 | 0x0
...
0x7fffffffc480 | 0x7fffffffc500
0x7fffffffc488 | 0x31337 ← rsp
...
0x7fffffffc500 | 0x7fffffffc550 ← rbp
```

시스템 콜

x86 syscall

```
int 0x80

요청하는 시스템 콜 번호 : eax
인자 순서 : ebx → ecx → edx → esi → edi → ebp → .....
```

x86에서는 int 0x80 명령어를 사용해 시스템 콜을 호출한다. 이때 eax 레지스터에 호출하고자 하는 시스템 콜의 번호를 넣는다. 이후 시스템 콜의 반환값 역시 eax 레지스터에 저장된다.

예제 : open syscall 호출 (번호 : 5)

💡 `open FILEHANDLE,MODE[,EXPR]`

```
section .data
    filename db "dreamhack.txt", 0
```

```
    buffer   times 100 db 0

section .text
    global _start

_start:
    mov eax, 5          ; syscall 번호
    mov ebx, filename   ; 이후 인자들 (파일 이름)
    mov ecx, 0          ; 열기 모드 (0)
    int 0x80
```

x64 syscall

```
명령어 : syscall
요청하는 syscall 번호 : rax
인자 순서 : rdi → rsi → rdx → rcx .....
```

syscall 명령어를 사용해 시스템 콜을 호출한다. 호출하고자 하는 syscall 번호는 rax에 저장된다.

예제

```
section .data
    filename db "dreamhack.txt", 0
    buffer   times 100 db 0

section .text
    global _start

_start:
    ; 파일 열기: open("dreamhack.txt", O_RDONLY, 0)
    mov rax, 2       ; syscall 번호
    lea rdi, [filename] ; 이후 인자 (파일 이름)
    mov rsi, 0          ; 읽기 모드
```

```
xor rdx, rdx
syscall
```

퀴즈 (1)

Q1

Q1. 레지스터, 메모리 및 코드가 다음과 같다. Code를 1까지 실행했을 때, rax에 저장된 값은?

```
[Register]
rbx = 0x401A40

==================================

[Memory]
0x401a40 | 0x0000000012345678
0x401a48 | 0x000000000C0FFEE
0x401a50 | 0x00000000DEADBEEF
0x401a58 | 0x00000000CAFEBABE
0x401a60 | 0x0000000087654321

==================================

[Code]
1: mov rax, [rbx+8]
2: lea rax, [rbx+8]
```

정답 : 0xC0FFEE

Q2

Q2. Code를 2까지 실행했을 때, rax에 들어있는 값은?

```
[Register]
rbx = 0x401A40

==================================

[Memory]
0x401a40 | 0x0000000012345678
0x401a48 | 0x000000000C0FFEE
0x401a50 | 0x00000000DEADBEEF
0x401a58 | 0x00000000CAFEBABE
0x401a60 | 0x0000000087654321

==================================

[Code]
1: mov rax, [rbx+8]
2: lea rax, [rbx+8]
```

1번까지 실행했을 때 rax에 c0ffee

2번까지 실행하면 rax에 rbx+8 이 있는 주소를 복사하므로 0x401a48

Q3

Q3. 레지스터, 메모리 및 코드가 다음과 같다. Code를 1까지 실행했을 때, rax에 저장된 값은?

```
[Register]
rax = 0x31337
rbx = 0x555555554000
rcx = 0x2

================================

[Memory]
0x555555554000| 0x0000000000000000
0x555555554008| 0x0000000000000001
0x555555554010| 0x0000000000000003
0x555555554018| 0x0000000000000005
0x555555554020| 0x000000000003133A

================================

[Code]
1: add rax, [rbx+rcx*8]
2: add rcx, 2
3: sub rax, [rbx+rcx*8]
4: inc rax
```

[rbx + rcx*8]

rcx : 0x2이므로 곱하기 팔하면 16, 16을 십진수로??

헐... 여기까지 와서 십육진수 변환 때문에 애먹다니ㅋㅋ

0x10이라고 함, gpt 힘 빌림

암튼 rbx + 0x10

rbx = 0x5555.....10

0x555555554010

인데!! []로 감싸여져 있으니 주소임!

[0x555555554010]의 주소에 써 있는 값을 봐야 하는 거임

거기 있는 값은 0x000....3

따라서 0x31337 + 0x03 = 0x3133A이다.

Q4

```
[Register]
rax = 0x31337
rbx = 0x555555554000
rcx = 0x2

================================

[Memory]
0x555555554000| 0x0000000000000000
0x555555554008| 0x0000000000000001
0x555555554010| 0x0000000000000003
0x555555554018| 0x0000000000000005
0x555555554020| 0x000000000003133A

================================

[Code]
1: add rax, [rbx+rcx*8]
2: add rcx, 2
3: sub rax, [rbx+rcx*8]
4: inc rax
```

이후 rcx에 2 더하면 0x4

sub rax [rbx + rcx*8]

[rbx + rcx*8] = -x555...4000 + 이십사 = 0x... + 0x0000....18

따라서 0x....4018의 주소에 있는 0x00....5를 rax에서 빼보자

코드 1까지 실행했을 때의 rax가 0x3133A기 때문에 0x31335가 될 것임

→ 정답에 없다

🤦

아니 진심 왜지.....

아니 rcx*8하면 32구나

따라서 [rbx + rcx*8]은 + 0x20, [0x555.....4020] 에 있는 3133A

rax 현재 0x3133A, 따라서 0

Q5

```
[Register]
rax = 0x31337
rbx = 0x555555554000
rcx = 0x2

==================================

[Memory]
0x555555554000| 0x0000000000000000
0x555555554008| 0x0000000000000001
0x555555554010| 0x0000000000000003
0x555555554018| 0x0000000000000005
0x555555554020| 0x000000000003133A

==================================

[Code]
1: add rax, [rbx+rcx*8]
2: add rcx, 2
3: sub rax, [rbx+rcx*8]
4: inc rax
```

4까지 실행했을 때 1

and rax, rcx

and 연산은 두 비트가 모두 1이어야 1

Q6

Q6. 레지스터, 메모리 및 코드가 다음과 같다. Code를 1까지 실행했을 때, rax에 저장된 값은?

```
[Register]
rax = 0xffffffff00000000
rbx = 0x00000000ffffffff
rcx = 0x123456789abcdef0

==================================

[Code]
1: and rax, rcx
2: and rbx, rcx
3: or rax, rbx
```

0xffffffff00000000
0x123456789abcdef0

0x00000000000000

→ 오답

흠...

정답은 0x12345678000000이엇음

왜인지는 모르겠다

아 이걸 다 이진수로 바꿔서 계산해야 하는 거였음;;;;

헐... 그... 그렇게까지?ㅋㅋㅋ;;

Q7

and rbx, rcx

걍 파이썬으로 변환해서 풀자

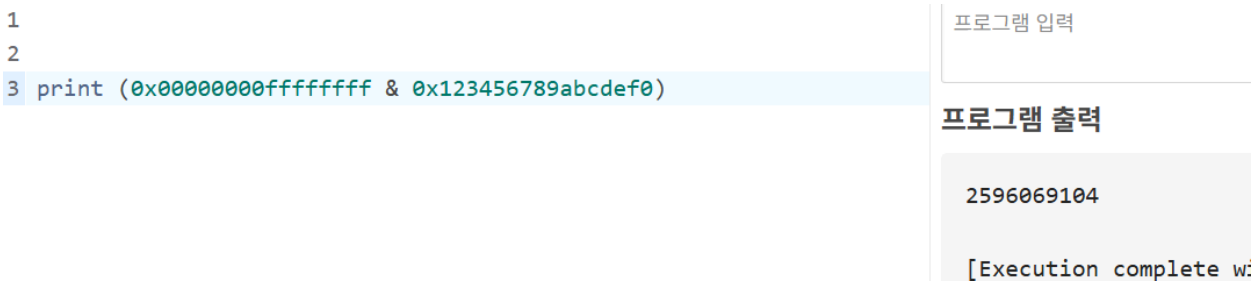
value1 = bin(0x00000000ffffffff)

value2 = bin(0x123456789abcdef0)

print(hex(value1 & value2))

https://www.mycompiler.io/ko/new/python

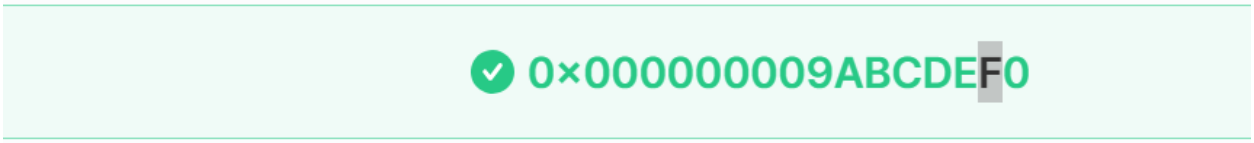
→ 설치 없이 파이썬 실행



위의 코드로 오류 뜨길래 물어보니까 bin()은 인자를 2진수 "문자열"로 반환하는 모듈이라고 한다. 아하... 그래서 문자열로 연산할 수 없다고 오류 뜨길래 그냥 16진수 상태 그대로 연산시켰음.

근데 정답지에 없음🤦🤦

일단 찍어서 4번...



그래서 cluade한테 나 대신 파이썬 짜 달라고 하니까 아래 같은 코드가 나왔다.

```
# 초기 레지스터 값들
rax = 0xffffffff00000000
rbx = 0x00000000ffffffff
rcx = 0x123456789abcdef0

print("초기값:")
print(f"rax = {hex(rax)}")
print(f"rbx = {hex(rbx)}")
print(f"rcx = {hex(rcx)}")
```

```
print()

# 1번: and rax, rcx
rax = rax & rcx
print("1번 실행 후:")
print(f"rax = rax & rcx = {hex(rax)}")
print()

# 2번: and rbx, rcx
rbx = rbx & rcx
print("2번 실행 후:")
print(f"rbx = rbx & rcx = {hex(rbx)}")
print()

# 3번: or rax, rbx (참고용)
final_rax = rax | rbx
print("3번까지 실행하면:")
print(f"rax = rax | rbx = {hex(final_rax)}")
```

이번에는 정답 맞게 나옴.

```
초기값:
rax = 0xffffffff00000000
rbx = 0xffffffff
rcx = 0x123456789abcdef0

1번 실행 후:
rax = rax & rcx = 0x1234567800000000

2번 실행 후:
rbx = rbx & rcx = 0x9abcdef0

3번까지 실행하면:
rax = rax | rbx = 0x123456789abcdef0
```

```
[Execution complete with exit code 0]
```

아니 근데 rbx & rcx 로 변수 따로 저장해서 하는 거나 나처럼 16진수로 바로 계산한 거나 뭐가 다르다고 값이 다르게 나온거지

아니 걍 내가 작성한 초기 코드 출력값이 10진수로 나온 거였다.... hex()해보면 아마도? 같게 나올 거임....

Q9

Q9. 레지스터, 메모리 및 코드가 다음과 같다. Code를 1까지 실행했을 때, rax에 저장된 값은?

```
[Register]
rax = 0x35014541
rbx = 0xdeadbeef

==================================

[Code]
1: xor rax, rbx
2: xor rax, rbx
3: not eax
```

```
print(hex(0x35014541 ^ 0xdeadbeef))
```

프로그램 출력

`0xebacfbae`

Q10

Q9. 레지스터, 메모리 및 코드가 다음과 같다. Code를 1까지 실행했을 때, rax에 저장된 값은?

```
[Register]
rax = 0x35014541
rbx = 0xdeadbeef

==================================

[Code]
1: xor rax, rbx
2: xor rax, rbx
3: not eax
```

프로그램 출력

0x35014541

Q11

Q11. Code를 3까지 실행했을 때, rax에 저장된 값은?

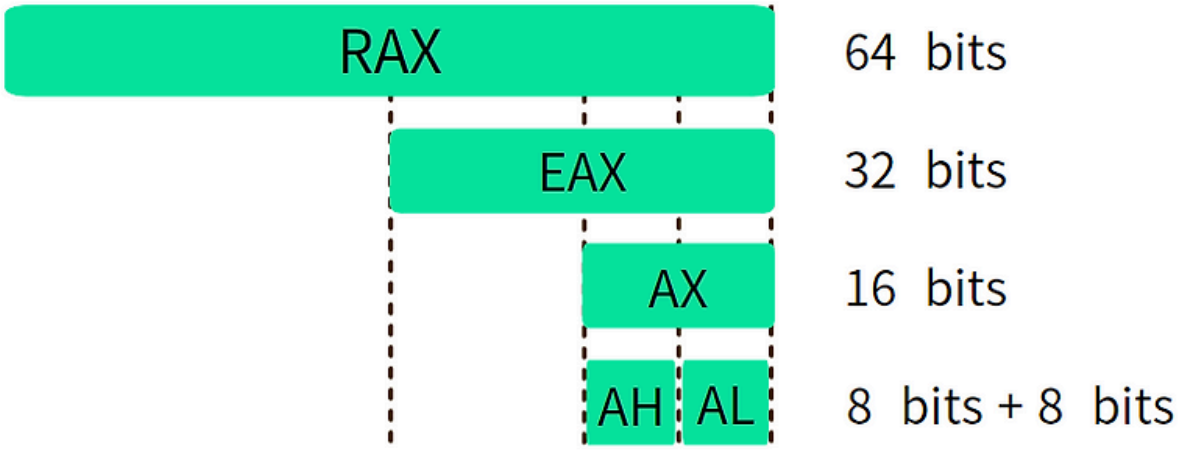
```
[Register]
rax = 0x35014541
rbx = 0xdeadbeef

==================================

[Code]
1: xor rax, rbx
2: xor rax, rbx
3: not eax
```

not eax

eax는 rax의 하위 32비트(로 기억하고 있음)



그러니까 현재 rax `0x35014541`에서 하위 32비트만 반전하겠다는 뜻

~ ← 이거 Not이라길래 사용했는데 결과 자꾸 이상하게 나와서 결국 claude 조언대로 아래 코드 사용함

```
print(hex( 0x35014541 ^ 0x0000FFFF ))
```

프로그램 출력

`0x3501babe`

정답은 cafebabe였다... 어케 하면 그런 답이 나오는 거임

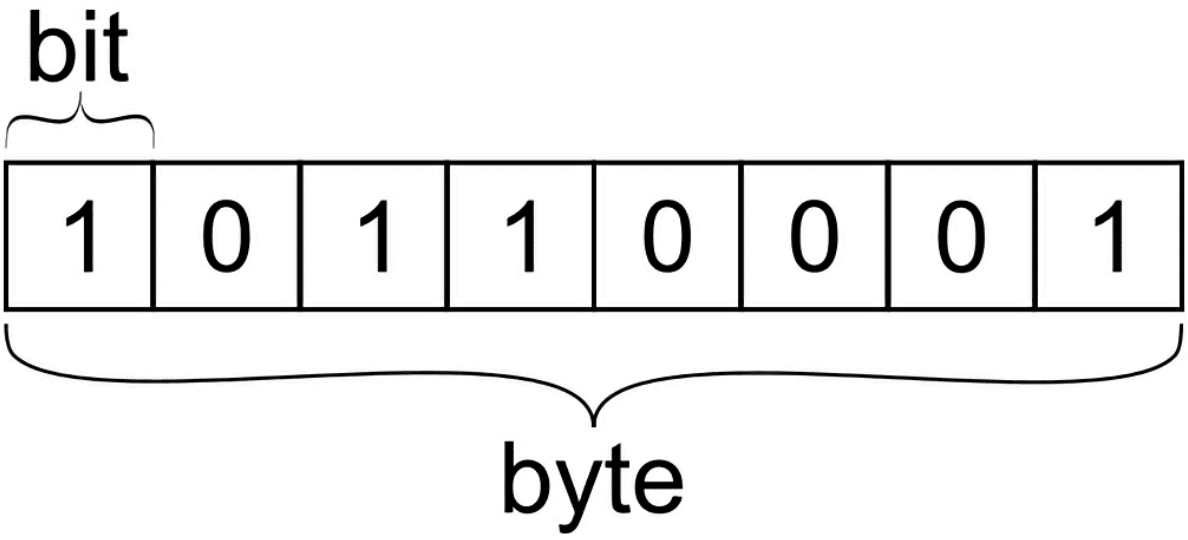
암튼 느낀 점 : 파이썬은 신의 언어다

퀴즈(2)

Q1. end로 점프하면 프로그램이 종료된다고 가정하자. 프로그램이 종료됐을 때, 0x400000 부터 0x400019까지의 데이터를 대응되는 아스키 문자로 변환하면 어느 문자열이 나오는가?

```
[Register]
rcx = 0
rdx = 0
rsi = 0x400000
=======================
[Memory]
0x400000 | 0x67 0x55 0x5c 0x53 0x5f 0x5d 0x55 0x10
0x400008 | 0x44 0x5f 0x10 0x51 0x43 0x43 0x55 0x5d
0x400010 | 0x52 0x5c 0x49 0x10 0x47 0x5f 0x42 0x5c
0x400018 | 0x54 0x11 0x00 0x00 0x00 0x00 0x00 0x00
=======================
[code]
1: mov dl, BYTE PTR[rsi+rcx]
2: xor dl, 0x30
3: mov BYTE PTR[rsi+rcx], dl
4: inc rcx
5: cmp rcx, 0x19
6: jg end
7: jmp 1
```

1. dl에 byte ptr[rsi + rcx] 복사
    a. [0x40000] 의 byte ptr (1바이트) 데이터 값 복사
    b. 1바이트 = 8비트



    1. 0x5567? ;;;
    2. 0x67??
2. xor dl, 0x30
    a.

    `0x57`
3. 0x400000의 데이터 0x57로 교체
4. inc rcx = 1
5. cmp rcx, 0x19
6.

일단 분기문 rcx 랑 0x19 비교해서 rcx 값이 더 크면 end로 점프

헐..... rcx가 더 작음

점프를 안 해서 프로그램 종료가 안 됨;;;

이후 구문 jmp 1이니까 다시 코드 1부터 반복

반복문이구나 이거

rcx가 0x19(25)보다 커질 때까지26회 반복해야함

장난하나

0x67을 0x30과 26번 xor하라고?

```
i=0
value = 0x67
while i<26:
    result = value ^ 0x30
    value = result
    i+=1

print (hex(result))
```

글케 됐다

프로그램 출력

`0x67`

하고 코드 cluade한테 물어보니까 친절하게 설명해줌

```
# 참고: XOR의 특성상 같은 값을 두 번 XOR하면 원래 값으로 돌아옴
print(f"\n참고: 0x67 ^ 0x30 = {hex(0x67 ^ 0x30)}")
print(f"그 결과에 다시 0x30을 XOR: {hex((0x67 ^ 0x30) ^ 0x30)}")
print("26회 반복하면 원래 값으로 돌아갈 것입니다!")
```

짝수번으로 xor 반복하면 원래 값으로 돌아간다고....

그래라....

헥스 덤프 아스키 변환 코드는 다시 ai에게 부탁

```
# 주어진 hex 덤프 데이터
hex_dump = """
0x400000 | 0x67 0x55 0x5c 0x53 0x5f 0x5d 0x55 0x10
0x400008 | 0x44 0x5f 0x10 0x51 0x43 0x43 0x55 0x5d
0x400010 | 0x52 0x5c 0x49 0x10 0x47 0x5f 0x42 0x5c
0x400018 | 0x54 0x11 0x00 0x00 0x00 0x00 0x00 0x00
"""

def hex_dump_to_ascii(hex_dump_text):
    """hex 덤프를 ASCII 문자로 변환"""
```

```
result = ""
ascii_result = ""

print("주소별 변환 결과:")
print("-" * 60)

for line in hex_dump_text.strip().split('\n'):
    if not line.strip():
        continue

    # 주소와 데이터 분리
    parts = line.split(' | ')
    if len(parts) != 2:
        continue

    address = parts[0].strip()
    hex_values = parts[1].strip().split()

    line_ascii = ""
    line_details = []

    for hex_val in hex_values:
        # 0x 제거하고 정수로 변환
        byte_val = int(hex_val, 16)

        # ASCII 문자로 변환 (출력 가능한 문자만)
        if 32 <= byte_val <= 126:  # 출력 가능한 ASCII 범위
            char = chr(byte_val)
            line_ascii += char
            line_details.append(f"{hex_val} → '{char}'")
        else:
            line_ascii += "."  # 출력 불가능한 문자는 .으로 표시
            line_details.append(f"{hex_val} → '.' (non-printable)")

    print(f"{address}: {line_ascii}")
    print(f"          {' '.join(line_details)}")
```

```
        print()

        ascii_result += line_ascii

    return ascii_result

# 변환 실행
ascii_text = hex_dump_to_ascii(hex_dump)

print("=" * 60)
print("전체 ASCII 결과:")
print(f"'{ascii_text}'")
print()

# null 바이트 제거한 버전
clean_ascii = ascii_text.replace('\x00', '').replace('.', '')
print("정리된 ASCII 결과 (null 바이트 및 특수문자 제거):")
print(f"'{clean_ascii}'")

# 간단한 방법으로도 해보기
print("\n" + "=" * 60)
print("간단한 방법으로 한 번에 변환:")

# 모든 hex 값들을 추출
all_hex_values = []
for line in hex_dump.strip().split('\n'):
    if '|' in line:
        hex_part = line.split('|')[1].strip()
        hex_values = hex_part.split()
        all_hex_values.extend(hex_values)

# ASCII로 변환
simple_result = ""
for hex_val in all_hex_values:
    byte_val = int(hex_val, 16)
    if byte_val == 0:  # null 바이트는 건너뛰기
```

```
        break
    elif 32 <= byte_val <= 126:
        simple_result += chr(byte_val)
    else:
        simple_result += "."

print(f"결과: '{simple_result}'")
```

이렇게 긴 코드를 써줌

```
정리된 ASCII 결과 (null 바이트 및 특수문자 제거):
'gU\S_]UD_QCCU]R\IG_B\T'
```

결과가 전혀 flag 같지 않다... 헐.....

어디서 잘못된 거지?

claude에게 재부탁

```
# Assembly 코드 분석 및 시뮬레이션

# 초기 상태
print("=== 초기 상태 ===")
rcx = 0
rdx = 0
rsi = 0x400000
print(f"rcx = {rcx}")
print(f"rdx = {rdx}")
print(f"rsi = {hex(rsi)}")
print()

# 메모리 데이터 (0x400000부터)
memory = {
    0x400000: [0x67, 0x55, 0x5c, 0x53, 0x5f, 0x5d, 0x55, 0x10],
    0x400008: [0x44, 0x5f, 0x10, 0x51, 0x43, 0x43, 0x55, 0x5d],
```

```
    0x400010: [0x52, 0x5c, 0x49, 0x10, 0x47, 0x5f, 0x42, 0x5c],
    0x400018: [0x54, 0x11, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
}

def get_memory_byte(address):
    """주어진 주소의 바이트 값을 반환"""
    base_addr = (address // 8) * 8
    offset = address % 8
    if base_addr in memory:
        return memory[base_addr][offset]
    return 0

print("=== 코드 실행 시뮬레이션 ===")
print("Loop 시작:")
print()

result_string = ""
iteration = 0

while True:
    iteration += 1
    current_address = rsi + rcx

    print(f"반복 {iteration}:")
    print(f"  현재 주소: {hex(current_address)} (rsi + rcx = {hex(rsi)} + {rcx})")

    # 1: mov dl, BYTE PTR[rsi+rcx]
    dl = get_memory_byte(current_address)
    print(f"  1. mov dl, BYTE PTR[{hex(current_address)}] → dl = {hex(dl)}")

    # 2: xor dl, 0x30
    dl = dl ^ 0x30
    print(f"  2. xor dl, 0x30 → dl = {hex(dl)}")

    # 3: mov BYTE PTR[rsi+rcx], dl (메모리에 다시 저장)
    print(f"  3. mov BYTE PTR[{hex(current_address)}], dl → 메모리 업데이트")
```

```
    # ASCII 변환 및 저장
    if 32 <= dl <= 126:  # 출력 가능한 ASCII
        char = chr(dl)
        result_string += char
        print(f"     ASCII 문자: '{char}'")
    else:
        result_string += "."
        print(f"     ASCII 문자: '.' (출력 불가)")

    # 4: inc rcx
    rcx += 1
    print(f"  4. inc rcx → rcx = {rcx}")

    # 5: cmp rcx, 0x19
    print(f"  5. cmp rcx, 0x19 → {rcx} vs 25")

    # 6: jg end (rcx > 25이면 종료)
    if rcx > 0x19:  # 0x19 = 25
        print("  6. jg end → 조건 만족, 루프 종료")
        break
    else:
        print("  6. jg end → 조건 불만족, 루프 계속")
        print("  7. jmp 1 → 루프 처음으로")

    print()

print("\n=== 최종 결과 ===")
print(f"변환된 문자열: '{result_string}'")
print(f"총 반복 횟수: {iteration}")
print()

print("=== 해설 ===")
print("이 코드는 다음과 같은 작업을 수행합니다:")
print("1. 0x400000 주소부터 시작해서 각 바이트를 읽음")
print("2. 각 바이트에 0x30을 XOR 연산")
```

```
print("3. 결과를 다시 메모리에 저장")
print("4. 25개 바이트(0x19 = 25)까지 반복")
print("5. XOR 0x30은 ASCII 숫자와 대문자 사이의 변환에 사용됨")
print("   - 숫자 '0'~'9' (0x30~0x39) ↔ 문자 '@'~'I' (0x00~0x09)")
print("   - 대문자 'A'~'Z' (0x41~0x5A) ↔ 특수문자들")
```

아...! 포인터처럼 덤프 내 데이터 하나씩 돌면서 다 xor하는 거였구나

그러고 보니 루프마다 rcx 값 증가하고, 코드 1에서 rsi+rcx 하니까 다음 데이터로 이동하는 거였을텐데 내가 제대로 안 봐서 놓쳤음;; 그래도 루프문인 것까지는 맞췄으니까, 그건 잘했다고 생각한다....

정답은

```
=== 초기 상태 ===
rcx = 0
rdx = 0
rsi = 0x400000

=== 코드 실행 시뮬레이션 ===
Loop 시작:

반복 1:
  현재 주소: 0x400000 (rsi + rcx = 0x400000 + 0)
  1. mov dl, BYTE PTR[0x400000] → dl = 0x67
  2. xor dl, 0x30 → dl = 0x57
  3. mov BYTE PTR[0x400000], dl → 메모리 업데이트
     ASCII 문자: 'W'
  4. inc rcx → rcx = 1
  5. cmp rcx, 0x19 → 1 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 2:
  현재 주소: 0x400001 (rsi + rcx = 0x400000 + 1)
  1. mov dl, BYTE PTR[0x400001] → dl = 0x55
  2. xor dl, 0x30 → dl = 0x65
```

3. mov BYTE PTR[0x400001], dl → 메모리 업데이트
   ASCII 문자: 'e'
4. inc rcx → rcx = 2
5. cmp rcx, 0x19 → 2 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 3:
현재 주소: 0x400002 (rsi + rcx = 0x400000 + 2)
1. mov dl, BYTE PTR[0x400002] → dl = 0x5c
2. xor dl, 0x30 → dl = 0x6c
3. mov BYTE PTR[0x400002], dl → 메모리 업데이트
   ASCII 문자: 'l'
4. inc rcx → rcx = 3
5. cmp rcx, 0x19 → 3 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 4:
현재 주소: 0x400003 (rsi + rcx = 0x400000 + 3)
1. mov dl, BYTE PTR[0x400003] → dl = 0x53
2. xor dl, 0x30 → dl = 0x63
3. mov BYTE PTR[0x400003], dl → 메모리 업데이트
   ASCII 문자: 'c'
4. inc rcx → rcx = 4
5. cmp rcx, 0x19 → 4 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 5:
현재 주소: 0x400004 (rsi + rcx = 0x400000 + 4)
1. mov dl, BYTE PTR[0x400004] → dl = 0x5f
2. xor dl, 0x30 → dl = 0x6f
3. mov BYTE PTR[0x400004], dl → 메모리 업데이트
   ASCII 문자: 'o'
4. inc rcx → rcx = 5

5. cmp rcx, 0x19 → 5 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 6:
현재 주소: 0x400005 (rsi + rcx = 0x400000 + 5)
1. mov dl, BYTE PTR[0x400005] → dl = 0x5d
2. xor dl, 0x30 → dl = 0x6d
3. mov BYTE PTR[0x400005], dl → 메모리 업데이트
ASCII 문자: 'm'
4. inc rcx → rcx = 6
5. cmp rcx, 0x19 → 6 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 7:
현재 주소: 0x400006 (rsi + rcx = 0x400000 + 6)
1. mov dl, BYTE PTR[0x400006] → dl = 0x55
2. xor dl, 0x30 → dl = 0x65
3. mov BYTE PTR[0x400006], dl → 메모리 업데이트
ASCII 문자: 'e'
4. inc rcx → rcx = 7
5. cmp rcx, 0x19 → 7 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 8:
현재 주소: 0x400007 (rsi + rcx = 0x400000 + 7)
1. mov dl, BYTE PTR[0x400007] → dl = 0x10
2. xor dl, 0x30 → dl = 0x20
3. mov BYTE PTR[0x400007], dl → 메모리 업데이트
ASCII 문자: ' '
4. inc rcx → rcx = 8
5. cmp rcx, 0x19 → 8 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 9:
  현재 주소: 0x400008 (rsi + rcx = 0x400000 + 8)
  1. mov dl, BYTE PTR[0x400008] → dl = 0x44
  2. xor dl, 0x30 → dl = 0x74
  3. mov BYTE PTR[0x400008], dl → 메모리 업데이트
     ASCII 문자: 't'
  4. inc rcx → rcx = 9
  5. cmp rcx, 0x19 → 9 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 10:
  현재 주소: 0x400009 (rsi + rcx = 0x400000 + 9)
  1. mov dl, BYTE PTR[0x400009] → dl = 0x5f
  2. xor dl, 0x30 → dl = 0x6f
  3. mov BYTE PTR[0x400009], dl → 메모리 업데이트
     ASCII 문자: 'o'
  4. inc rcx → rcx = 10
  5. cmp rcx, 0x19 → 10 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 11:
  현재 주소: 0x40000a (rsi + rcx = 0x400000 + 10)
  1. mov dl, BYTE PTR[0x40000a] → dl = 0x10
  2. xor dl, 0x30 → dl = 0x20
  3. mov BYTE PTR[0x40000a], dl → 메모리 업데이트
     ASCII 문자: ' '
  4. inc rcx → rcx = 11
  5. cmp rcx, 0x19 → 11 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 12:
  현재 주소: 0x40000b (rsi + rcx = 0x400000 + 11)

1. mov dl, BYTE PTR[0x40000b] → dl = 0x51
2. xor dl, 0x30 → dl = 0x61
3. mov BYTE PTR[0x40000b], dl → 메모리 업데이트
   ASCII 문자: 'a'
4. inc rcx → rcx = 12
5. cmp rcx, 0x19 → 12 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 13:
현재 주소: 0x40000c (rsi + rcx = 0x400000 + 12)
1. mov dl, BYTE PTR[0x40000c] → dl = 0x43
2. xor dl, 0x30 → dl = 0x73
3. mov BYTE PTR[0x40000c], dl → 메모리 업데이트
   ASCII 문자: 's'
4. inc rcx → rcx = 13
5. cmp rcx, 0x19 → 13 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 14:
현재 주소: 0x40000d (rsi + rcx = 0x400000 + 13)
1. mov dl, BYTE PTR[0x40000d] → dl = 0x43
2. xor dl, 0x30 → dl = 0x73
3. mov BYTE PTR[0x40000d], dl → 메모리 업데이트
   ASCII 문자: 's'
4. inc rcx → rcx = 14
5. cmp rcx, 0x19 → 14 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 15:
현재 주소: 0x40000e (rsi + rcx = 0x400000 + 14)
1. mov dl, BYTE PTR[0x40000e] → dl = 0x55
2. xor dl, 0x30 → dl = 0x65
3. mov BYTE PTR[0x40000e], dl → 메모리 업데이트

```
    ASCII 문자: 'e'
  4. inc rcx → rcx = 15
  5. cmp rcx, 0x19 → 15 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 16:
  현재 주소: 0x40000f (rsi + rcx = 0x400000 + 15)
  1. mov dl, BYTE PTR[0x40000f] → dl = 0x5d
  2. xor dl, 0x30 → dl = 0x6d
  3. mov BYTE PTR[0x40000f], dl → 메모리 업데이트
     ASCII 문자: 'm'
  4. inc rcx → rcx = 16
  5. cmp rcx, 0x19 → 16 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 17:
  현재 주소: 0x400010 (rsi + rcx = 0x400000 + 16)
  1. mov dl, BYTE PTR[0x400010] → dl = 0x52
  2. xor dl, 0x30 → dl = 0x62
  3. mov BYTE PTR[0x400010], dl → 메모리 업데이트
     ASCII 문자: 'b'
  4. inc rcx → rcx = 17
  5. cmp rcx, 0x19 → 17 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 18:
  현재 주소: 0x400011 (rsi + rcx = 0x400000 + 17)
  1. mov dl, BYTE PTR[0x400011] → dl = 0x5c
  2. xor dl, 0x30 → dl = 0x6c
  3. mov BYTE PTR[0x400011], dl → 메모리 업데이트
     ASCII 문자: 'l'
  4. inc rcx → rcx = 18
  5. cmp rcx, 0x19 → 18 vs 25
```

6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 19:
현재 주소: 0x400012 (rsi + rcx = 0x400000 + 18)
1. mov dl, BYTE PTR[0x400012] → dl = 0x49
2. xor dl, 0x30 → dl = 0x79
3. mov BYTE PTR[0x400012], dl → 메모리 업데이트
ASCII 문자: 'y'
4. inc rcx → rcx = 19
5. cmp rcx, 0x19 → 19 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 20:
현재 주소: 0x400013 (rsi + rcx = 0x400000 + 19)
1. mov dl, BYTE PTR[0x400013] → dl = 0x10
2. xor dl, 0x30 → dl = 0x20
3. mov BYTE PTR[0x400013], dl → 메모리 업데이트
ASCII 문자: ' '
4. inc rcx → rcx = 20
5. cmp rcx, 0x19 → 20 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 21:
현재 주소: 0x400014 (rsi + rcx = 0x400000 + 20)
1. mov dl, BYTE PTR[0x400014] → dl = 0x47
2. xor dl, 0x30 → dl = 0x77
3. mov BYTE PTR[0x400014], dl → 메모리 업데이트
ASCII 문자: 'w'
4. inc rcx → rcx = 21
5. cmp rcx, 0x19 → 21 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 22:
현재 주소: 0x400015 (rsi + rcx = 0x400000 + 21)
1. mov dl, BYTE PTR[0x400015] → dl = 0x5f
2. xor dl, 0x30 → dl = 0x6f
3. mov BYTE PTR[0x400015], dl → 메모리 업데이트
ASCII 문자: 'o'
4. inc rcx → rcx = 22
5. cmp rcx, 0x19 → 22 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 23:
현재 주소: 0x400016 (rsi + rcx = 0x400000 + 22)
1. mov dl, BYTE PTR[0x400016] → dl = 0x42
2. xor dl, 0x30 → dl = 0x72
3. mov BYTE PTR[0x400016], dl → 메모리 업데이트
ASCII 문자: 'r'
4. inc rcx → rcx = 23
5. cmp rcx, 0x19 → 23 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 24:
현재 주소: 0x400017 (rsi + rcx = 0x400000 + 23)
1. mov dl, BYTE PTR[0x400017] → dl = 0x5c
2. xor dl, 0x30 → dl = 0x6c
3. mov BYTE PTR[0x400017], dl → 메모리 업데이트
ASCII 문자: 'l'
4. inc rcx → rcx = 24
5. cmp rcx, 0x19 → 24 vs 25
6. jg end → 조건 불만족, 루프 계속
7. jmp 1 → 루프 처음으로

반복 25:
현재 주소: 0x400018 (rsi + rcx = 0x400000 + 24)
1. mov dl, BYTE PTR[0x400018] → dl = 0x54

```
  2. xor dl, 0x30 → dl = 0x64
  3. mov BYTE PTR[0x400018], dl → 메모리 업데이트
     ASCII 문자: 'd'
  4. inc rcx → rcx = 25
  5. cmp rcx, 0x19 → 25 vs 25
  6. jg end → 조건 불만족, 루프 계속
  7. jmp 1 → 루프 처음으로

반복 26:
  현재 주소: 0x400019 (rsi + rcx = 0x400000 + 25)
  1. mov dl, BYTE PTR[0x400019] → dl = 0x11
  2. xor dl, 0x30 → dl = 0x21
  3. mov BYTE PTR[0x400019], dl → 메모리 업데이트
     ASCII 문자: '!'
  4. inc rcx → rcx = 26
  5. cmp rcx, 0x19 → 26 vs 25
  6. jg end → 조건 만족, 루프 종료

=== 최종 결과 ===
변환된 문자열: 'Welcome to assembly world!'
총 반복 횟수: 26

=== 해설 ===
이 코드는 다음과 같은 작업을 수행합니다:
1. 0x400000 주소부터 시작해서 각 바이트를 읽음
2. 각 바이트에 0x30을 XOR 연산
3. 결과를 다시 메모리에 저장
4. 25개 바이트(0x19 = 25)까지 반복
5. XOR 0x30은 ASCII 숫자와 대문자 사이의 변환에 사용됨
   - 숫자 '0'~'9' (0x30~0x39) ↔ 문자 '@'~'I' (0x00~0x09)
   - 대문자 'A'~'Z' (0x41~0x5A) ↔ 특수문자들

[Execution complete with exit code 0]
```

퀴즈 3

Q1. 다음 어셈블리 코드를 실행했을 때 출력되는 결과로 올바른 것은?

```
[Code]
main:
    push rbp
    mov rbp, rsp
    mov esi, 0xf
    mov rdi, 0x400500
    call 0x400497 <write_n>
    mov eax, 0x0
    pop rbp
    ret

write_n:
    push rbp
    mov rbp, rsp
    mov QWORD PTR [rbp-0x8],rdi
    mov DWORD PTR [rbp-0xc],esi
    xor rdx, rdx
    mov edx, DWORD PTR [rbp-0xc]
    mov rsi,QWORD PTR [rbp-0x8]
    mov rdi, 0x1
    mov rax, 0x1
    syscall
    pop rbp
    ret

==================================
[Memory]
0x400500 | 0x3037207964343372
0x400508 | 0x003f367562336420
```

esi : 0xf

rdi : 0x400500

write_n 호출

[rbp-0x8]에 rdi값 8바이트 복사

rbp는 `0x400500`..? 일 것으로 추정

`0x3037207964343372` → `0x3037400500343372`

아니지 ㅁㅊ 0x400508이겠지?? 스택은 아래로 자라니까 제일 큰 값이 베이스일 것 가틈(그리고 안 그러면 빼기 연산이 안 됨)

다시 정정하면

`0x400508 | 0x003f367562336420` → `0x400508 | 0x4005007562336420`

[rbp-0xc], esi

c면 아마 12니까

400508 -12 = 4004946이니까... 현재 제시된 메모리 주소를 넘어감

미치겠네 단위를 못 봐갖고

아니 쨌든... esi 값을 이번엔 4바이트 복사

그럼 그냥....

`0x400500 | 0x3037207964343372` → `0x400500 | 0x000f207964343372`

이렇다고 치자...

이후 xor rdx, rdx → rdx 0으로 만들기

mov edx, dword prt [rbp-0xc]

edx에 `0x000f` 복사

mov rsi, qword ptr [rbx-0x8]

rsi 에 `0x400500` 복사.....

아니 암튼 syscall write_n에 0xf, 0x400500을 인자로 전달해서 호출하는 문 아님??

ssize_t write(int fildes, const void *buf, size_t nbyte);

0xf는 파일 디스크립터, 400500은 버프 시작점? 가리키는 거라 하는 거라 보면 말 되는 것 같은데

그니까 0xf번 파일의 400500번지부터 데이터를 쓰겠다는 거잖아

글케 인자로 줘서 write_n 호출한 뒤

rdi에 넣어놓은 버퍼 시작점을 메모리에 옮기고(왜지)

파일 디크립터 번호도 메모리에 옮기고(왜지)

rdx 0으로 비워두고(ㄹㅇ왜지? 시스템콜할 때 쓰려나봄)

다시 edx rsi 에 아까 그것들 복사하고

그럼 결국 여기서 어떤 쓰기 행위가 이루어진 거임?? 어디에서 뭘 쓴 거지

rdi rax에 0x1 복사하는 건 헥스 덤프에는 쓰기가 안 이루어지잖아

뭐한거지 진짜

```
2
3
4 print(bytes.fromhex("3037207964343372").decode('ascii'))
5 print(bytes.fromhex("000f207964343372").decode('ascii'))
6 print(bytes.fromhex("003f367562336420").decode('ascii'))
7 print(bytes.fromhex("4005007562336420").decode('ascii'))
```

프로그램 출력

```
07 yd43r
☼ yd43r
?6ub3d
@ub3d
```

일단 파이썬으로 bytes.fromhex() → 2진수 변환, .decode('ascii') → 아스키 변환함

원본 헥스덤프와 내가 변환한 줄을 차례로 디코딩했다

그 결과 내가 변환한 줄은 아예 의미가 상실되었고, 원본 헥스 덤프를 풀어보니 r34dy 70 d3bu6?가 리틀 엔디안 식으로 뒤집어져서 저장된 듯

정답임

그치만 이건 다른 사람 라업을 좀 봐야 할 것 같다....

다른 사람 라업 참조

syscall	rax	arg0 (rdi)	arg1 (rsi)	arg2 (rdx)
write	0x01	unsigned int fd * 파일 디스크립터 지정	const char *buf * 해당 메모리에 위치한 값을 가리킨다.	size_t count * 크기

아... 인자로 어느 레지스터가 순서대로 들어가는지 잘 봐야겠구나(ㅋㅋㅋㅋ)

그래도 여튼 syscall 인자일 거라는 데까지는 맞았네요

예상한대로 rax 0x1은 syscall 번호가 맞았으며, rdi는 파일 디스크립터, rsi 는 버퍼의 메모리 위치, rdx는 사이즈(카운트). 여기서 rdx가 0으로 초기화되었다가 edx가 0xf가 되었기 때문에, 400500에 있는 데이터를 15바이트만큼 출력해야 함.

결과적으로는 출력 함수가 맞다. 끝!