5월 Space WAR 라이트업

2023111748 노희민

1.문제

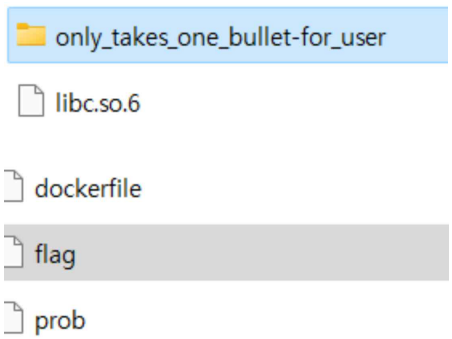
제목: only_takes_one_bullet

종목: 포너블

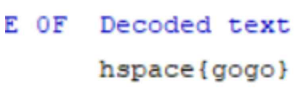
설명: very easy pwnable you can solve it

2.풀이

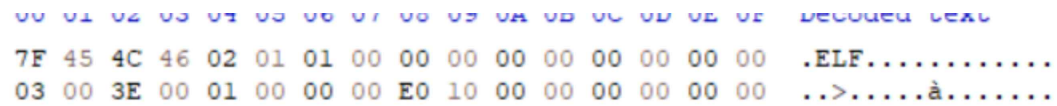
다운로드 받은 파일



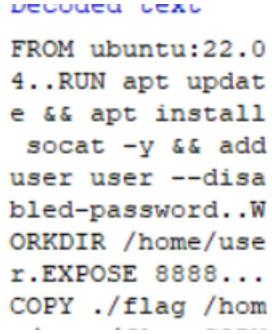
그 중 flag는 디코딩 하니 다음과 같은 문구가 나왔다.



물론 정답은 아니다.



prob와 libc.so.6은 .ELF라는 동일한 파일 시그니처를 가지고 있다.



dockerfile엔 우분투 실행 로그가 들어있다.

전문:

(도커명령어)

FROM ubuntu:22.04

RUN apt update && apt install socat -y && adduser user --disabled-password

//컨테이너에서 실행할 명령어

WORKDIR /home/user

EXPOSE 8888

COPY ./flag /home/user/flag

COPY ./prob /home/user/prob

RUN chown -R root:user /home/user && ₩

//파일, 디렉터리 소유자 변경

chmod -R 750 /home/user && ₩

//파일 모드 (읽기 모드로) 변경

chmod 744 /home/user/flag

USER user

//이후 따르는 명령 실행할 사용자 지정(도커 파일 명령어)

EXPOSE 8888

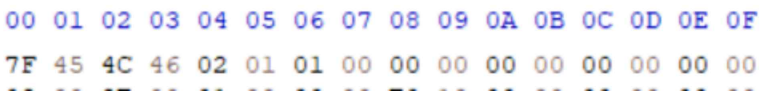
//포트번호

ENTRYPOINT ["socat", "TCP-LISTEN:8888,reuseaddr,fork", "EXEC:./prob,pty"]

//명령어 실행

도커란 호스트 운영체제의 커널을 공유하는 가상 머신을 의미하는 것 같다. 호스트 운영체제 위에 가상화된 하드웨어를 생성하지 않는다는 점에서 기존 가상머신보다 경량화되었다는 차이점이 있다.

찾아보니 7F 45 4C 46 02는 유닉스 elf 파일의 시그니처라고 한다. 리눅스가 유닉스 계열의 운영체제이니 리눅스 환경에서도 실행이 가능할지도 모르겠다.



prob 파일의 파일 헤더를 분석해 보자면 이렇다.

7F 45 4C 46: 매직 넘버

02 : 64비트

01 : 리틀 인디언

01: 버전(고정)

00: 운영체제 - 03 리눅스, 09 freeBSD. 00으로 설정되어도 문제는 없다고 한다.

리눅스에 elf 파일 정보를 간편하게 읽을 수 있는 명령어가 있다고 한다. wsl 툴을 이용해 readelf 명령어를 실행해보겠다.

```
r_user# readelf -h prob
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x10e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14200 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
```

prob 파일의 헤더 정보.

readelf -a (elf의 모든 정보 조회)

libc.so 파일에서의 실행결과.

```
readelf -h libc.so.6
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x29f50
  Start of program headers:          64 (bytes into file)
  Start of section headers:          2212080 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         14
  Size of section headers:           64 (bytes)
  Number of section headers:         66
  Section header string table index: 65
```

파일 헤더 정보.

타입: DYN (Shared object file)

Entry point address: 0x29f50

Section header string table index: 65

```
Relocation section '.rela.plt' at offset 0x27ad0 contains 54 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000219028  079300000007 R_X86_64_JUMP_SLO 00000000000a5740 realloc@@GLIBC_2.2.5 + 0
000000219038  000100000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_exception_create@GLIBC_PRIVATE + 0
000000219050  0b7000000007 R_X86_64_JUMP_SLO 00000000000a6520 calloc@@GLIBC_2.2.5 + 0
000000219088  000300000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_find_dso_for_[...]@GLIBC_PRIVATE + 0
0000002190c8  000500000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_deallocate_tls@GLIBC_PRIVATE + 0
0000002190d0  000600000007 R_X86_64_JUMP_SLO 0000000000000000 __tls_get_addr@GLIBC_2.3 + 0
0000002190f0  000900000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_fatal_printf@GLIBC_PRIVATE + 0
000000219138  000a00000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_audit_symbind_alt@GLIBC_PRIVATE + 0
000000219168  000b00000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_rtld_di_serinfo@GLIBC_PRIVATE + 0
000000219170  000c00000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_allocate_tls@GLIBC_PRIVATE + 0
000000219178  000d00000007 R_X86_64_JUMP_SLO 0000000000000000 __tunable_get_val@GLIBC_PRIVATE + 0
0000002191a0  000e00000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_allocate_tls_init@GLIBC_PRIVATE + 0
0000002191a8  001000000007 R_X86_64_JUMP_SLO 0000000000000000 __nptl_change_sta[...]@GLIBC_PRIVATE + 0
0000002191b8  001100000007 R_X86_64_JUMP_SLO 0000000000000000 _dl_audit_preinit@GLIBC_PRIVATE + 0
0000002191c0  000000000025 R_X86_64_IRELATIV                    a8720
0000002191b0  000000000025 R_X86_64_IRELATIV                    a89e0
```

objdump 명령어를 사용해 각 섹션을 어셈블리어로 디스어셈블 해봤다.

-D 명령어로 파일 전체를 디스어셈블 했더니 다음과 같이 상위 섹션이 끊겨 보이지 않는 오류가 발생했다. (섹션 헤더 시작점: 2212080) 섹션별로 나눠 디스어셈블을 다시 시도했다.

```
214532:       00 00                   add    %al,(%rax)
214534:       6f                      outsl  %ds:(%rsi),(%dx)
214535:       02 00                   add    (%rax),%al
      ...
21453f:       00 cb                   add    %cl,%bl
```

objdump -h libc.so.6 명령어로 섹션 헤더만 뽑아보자.

```
Sections:
Idx Name          Size      VMA
  0 .note.gnu.property 00000030  00000
                  CONTENTS, ALLOC, LOA
  1 .note.gnu.build-id 00000024  00000
                  CONTENTS, ALLOC, LOA
  2 .note.ABI-tag 00000020  0000000000
                  CONTENTS, ALLOC, LOA
  3 .gnu.hash     00004704  0000000000
                  CONTENTS, ALLOC, LOA
  4 .dynsym       00011b80  0000000000
                  CONTENTS, ALLOC, LOA
  5 .dynstr       00007f15  0000000000
                  CONTENTS, ALLOC, LOA
  6 .gnu.version  000017a0  0000000000
                  CONTENTS, ALLOC, LOA
  7 .gnu.version_d 00000524  000000000
                  CONTENTS, ALLOC, LOA
  8 .gnu.version_r 00000040  000000000
                  CONTENTS, ALLOC, LOA
  9 .rela.dyn     00007860  0000000000
                  CONTENTS, ALLOC, LOA
 10 .rela.plt     00000510  0000000000
                  CONTENTS, ALLOC, LOA
 11 .plt          00000370  0000000000
                  CONTENTS, ALLOC, LOA
```

섹션 헤더들.

우선 elf 파일에서 사용 가능한 함수들이 모여 있다는 .text부터 보자.

<__nss_database_lookup@GLIBC_2.2.5+0x39d29>

(2.2.5 뒤의 숫자는 가변적)

vmovdqa64 0x80(%rdi),%ymm17

1b1f77: 62 e1 75 20 da 57 05 vpminub 0xa0(%rdi),%ymm17,%ymm18

```
1b1f7e:        62 e1 fd 28 6f 5f 06      vmovdqa64 0xc0(%rdi),%ymm19
1b1f85:        62 e1 65 20 da 67 07      vpminub 0xe0(%rdi),%ymm19,%ymm20
1b1f8c:        62 b3 7d 20 3f c2 00      vpcmpeqb %ymm18,%ymm16,%k0
1b1f93:        62 b3 7d 20 3f cc 00      vpcmpeqb %ymm20,%ymm16,%k1
```

여긴 처음 보는 명령어들이다.

별다른 힌트를 얻을 수 없었기에 prob elf 파일을 먼저 디스어셈블 해보기로 했다 .text 섹션을 디스어셈블 한 결과 메인함수 진입점을 찾을 수 있었다.

```
0000000000010e0 <_start>:
    10e0:       f3 0f 1e fa             endbr64
    10e4:       31 ed                   xor    %ebp,%ebp
    10e6:       49 89 d1                mov    %rdx,%r9
    10e9:       5e                      pop    %rsi
    10ea:       48 89 e2                mov    %rsp,%rdx
    10ed:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
    10f1:       50                      push   %rax
    10f2:       54                      push   %rsp
    10f3:       45 31 c0                xor    %r8d,%r8d
    10f6:       31 c9                   xor    %ecx,%ecx
    10f8:       48 8d 3d ca 00 00 00    lea    0xca(%rip),%rdi        # 11c9 <main>
    10ff:       ff 15 d3 2e 00 00       call   *0x2ed3(%rip)        # 3fd8 <__libc_start_main@GLIBC_2.34>
```

#11c9 -> main

#3fd8-> libc_start_main@GLIBC

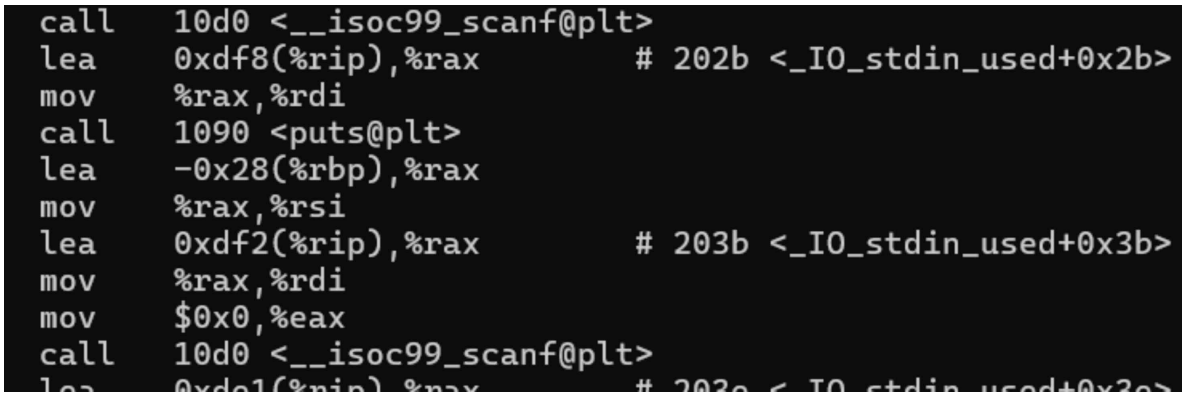
이 위치로 어떻게 이동하지?

```
15 .text         000001e1  00000000000010e0  00000000000010e0  000010e0  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
16 .fini         0000000d  00000000000012c4  00000000000012c4  000012c4  2**2
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
```

메인함수의 위치가 11c9이기 때문에 이 둘 사이에 있을 것 같다.

```
00000000000011c9 <main>:
    11c9:       f3 0f 1e fa             endbr64
    11cd:       55                      push   %rbp
    11ce:       48 89 e5                mov    %rsp,%rbp
    11d1:       48 83 ec 30             sub    $0x30,%rsp
    11d5:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    11dc:       00 00
```

.text 섹션에 있었다.



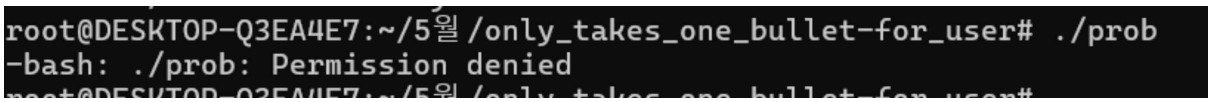
무언가 값을 입력받는 모습.

사실 어셈블리어가 무슨 뜻인지는 전혀 모르겠다. 그래도 조금이나마 짐작해 보자면...

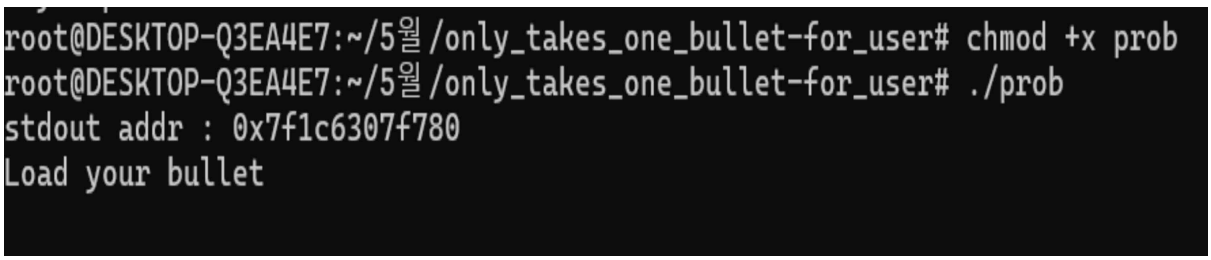
아...이거 올리디버거 같은 걸로 볼 순 없나?

검색을 통해 우분투로 elf 파일을 실행할 수 있다는 사실을 알아냈다.

우선 문제가 든 폴더를 통째로 우분투의 root 폴더로 옮긴 뒤 prob 실행 명령어를 입력했다.



권한이 거절되었다는 메시지가 나온다.



chmod 명령어를 사용해 파일 권한을 변경해줬다. (+x : 실행 권한 추가.)

그러자 Load your Bullet이라는 문구가 나온다. 이후 아까 본 scanf 함수를 통해 내 답을 입력받으려는 것 같다.

위의 stdout addr : 0x7f1c6307f780은 무슨 뜻이지?

stdout; 표준 출력 스트림. 표준 출력 장치에 대한 포인터.

단순히 Load your bullet이라는 출력 문구의 주소를 가리키는 포인터인듯. 그런데 이걸 굳이 출력 가능하게 설정한 이유는 뭐지?

```
Load your bullet
AAAAAAA
Aim your target
```

일단 아무 문자나 입력해봤다.

```
Load your bullet
AAAAAAA
Aim your target
AAAAAAA
What's your cool monolog?
AAAAAAA
Trigger!
Segmentation fault
```

세번 오답을 입력하자 파일이 실행 종료된다.

문제 제목: only_takes_one_bullet

이것과 관련이 있어보이는 출력문들이다.

밑의 Segmentation fault라는 건 건드리지 말아야 할 영역을 건드렸을 때 나오는 에러문구라는데, 입력값 글자수가 파일이 받아들일 수 있는 데이터 영역을 넘어선건지 뭔지.

```
root@DESKTOP-Q3EA4E7:~/5월# ./libc.so.6
-bash: ./libc.so.6: Permission denied
root@DESKTOP-Q3EA4E7:~/5월# chmod +x libc.so.6
root@DESKTOP-Q3EA4E7:~/5월# ./libc.so.6
GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.5) stable release version 2.35.
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 11.4.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
For bug reporting instructions, please see:
<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>.
root@DESKTOP-Q3EA4E7:~/5월#
```

기왕 이렇게 된 거 또 다른 elf 파일인 libc.so.6도 같은 과정으로 실행해봤다.

다.

GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.5) stable release version 2.35.

Copyright (C) 2022 Free Software Foundation, Inc.

This is free software; see the source for copying conditions.

There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A

PARTICULAR PURPOSE.

Compiled by GNU CC version 11.4.0.

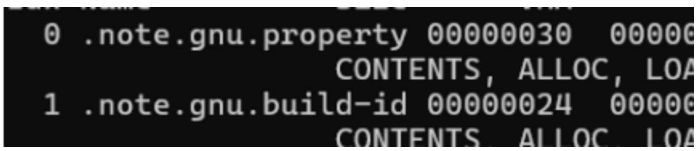
libc ABIs: UNIQUE IFUNC ABSOLUTE

For bug reporting instructions, please see:

<https://bugs.launchpad.net/ubuntu/+source/glibc/+bugs>

!

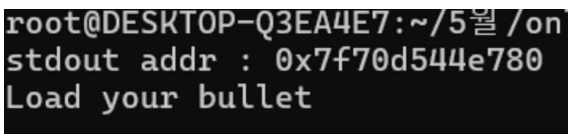
아까 같은 파일의 섹션 헤더에 gnu 라는 단어가 있었다.



GNU C Library (Ubuntu GLIBC 2.35-0ubuntu3.5) stable release version 2.35.

C표준 라이브러리.

다시 prob.



음? 한번 더 실행하니까 stdout addr 주소가 바뀌었다. 변수를 사용하는 것 같은데. 이렇게 되면 저 주소 자체를 추적하는 건 의미가 없지 않을까? 아무튼 다시

메인함수로 돌아가서 입력값을 판단하는 함수 이전의 비교 함수 등을 찾아 내용을 확인하는 게 정석적이긴 하다. 그런데 웬만하면 그 방법을 쓰고 싶지 않아 다른 경로가 없나 이것저것 시도 중이다. 가장 큰 이유는 그 정석적인 방법이 제일 어렵기 때문이다. 말도 못 하게 까다롭다. 그러니 일단 저 stdout addr 주소로 가보자.

주소를 못 찾겠다. 그냥 정석법으로 가보자.

stdout addr : %p

�Load your bullet�%lx�Aim your target�%p�What's your cool monolog?�Trigger!

hxd로 디코딩한 코드.

파일 오프셋(주소):

```
00002010
00002020
00002030
00002040
00002050
```

```
74 05                   je     12bf <main+0xf6>
e8 e1 fd ff ff          call   10a0 <__stack_chk_fail@plt>
c9                      leave
c3                      ret
```

메인코드 맨 끝의 이 부분.

```
12a1:       e8 ea fd ff ff          call   1090 <puts@plt>
12a6:       b8 00 00 00 00          mov    $0x0,%eax
12ab:       48 8b 55 f8             mov    -0x8(%rbp),%rdx
12af:       64 48 2b 14 25 28 00    sub    %fs:0x28,%rdx
12b6:       00 00
```

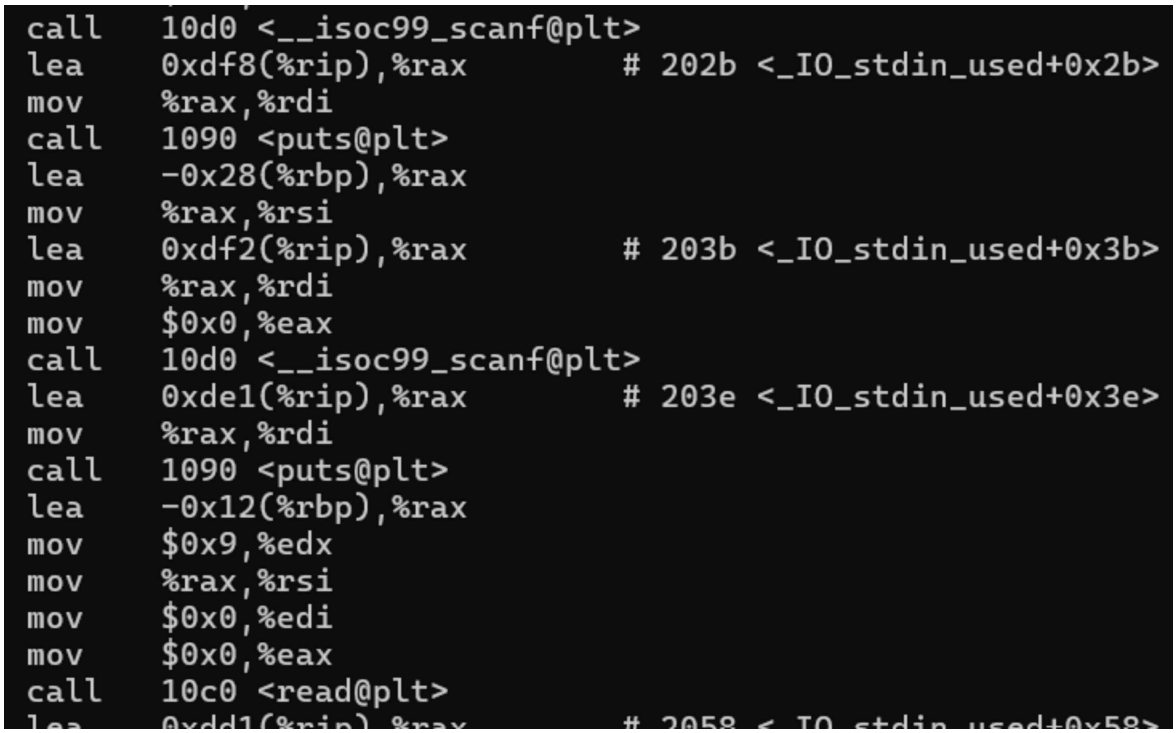
eax 레지스터의 데이터를 0x0에 넣고, rdx 레지스터의 데이터를 rbp 레지스터에 넣음. 이후 0x28 에서 rdx레지스터 데이터 값을 뺄셈한 결과를 다시 ox28에 저장.

이후 je 명령어; 두 피연산자가 같을 시 call로 leave로 점프. 다르면 fail 함수 호출.

대강 이런 흐름인가? 그런데 cmp 명령어가 없는데 je를 어떻게 실행한다는 건지 이해가 안 간다.

fs는 레지스터 이름이라고. 흠.

lea; 좌변 레지스터에 우변 주소값 저장



?

뭘 하는 건지 전혀 이해하질 못 하겠다.

미해결