



EDDI

Electronic Design  
Development Institute

---

# 에디로봇아카데미 임베디드 마스터 Lv1 과정

제 5기

2023. 05. 31

어광선

# CONTENTS

1. GDB란
2. GDB 명령어
3. function 예제 디버깅
4. function 메모리 맵

GNU 소프트웨어 시스템을 위한 기본 디버거.

개발자 : 리처드 스톨만

(GNU? (General Public License) 운영 체제의 하나이자 컴퓨터 소프트웨어의 모음)  
다양한 유닉스 기반의 시스템에서 동작하는 이식성있는 디버거이다. C, C++, 포트란  
등 여러 프로그래밍 언어를 지원한다.

임베디드 시스템을 디버깅할 때 사용되는 **원격 모드**를 지원한다. GDB는 GDB 프로  
토콜을 알고 있는 원격지의 'stub'와 직렬 포트 혹은 TCP/IP를 통해 통신할 수 있다.  
리눅스 커널에 사용되는 소스 수준의 디버거인 KGDB에서도 사용된다.

일반적으로 실행파일을 GDB를 이용하여 디버깅하기 위해서 컴파일 옵션에 -g 옵션  
을 추가하여 컴파일 해줘야한다. (ex : gcc -o test test.c -g)

# GDB 명령어

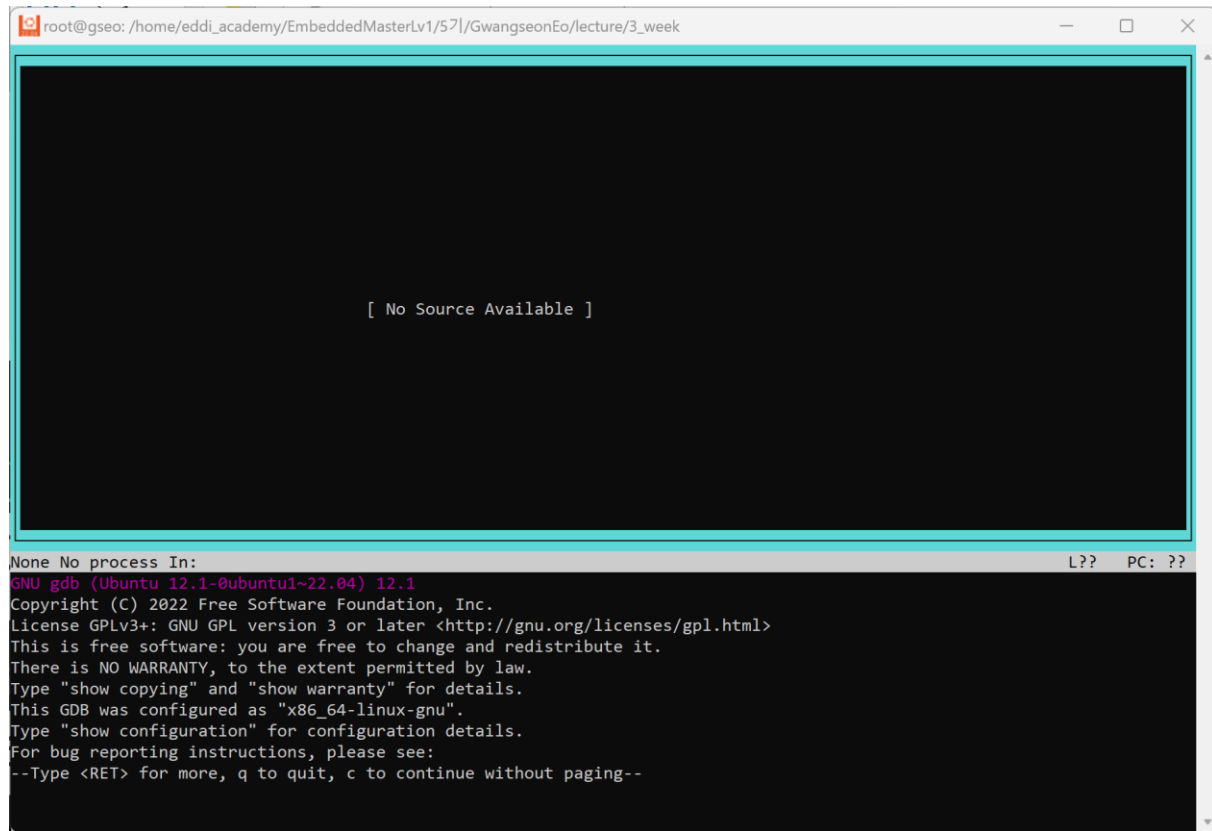
```
gseo@DESKTOP-A6CM3FL:~/test$ ls -l
total 16
-rwxr-xr-x 1 gseo gseo 10944 May 22 09:36 function
-rw-r--r-- 1 gseo gseo  180 May 22 09:36 function.c
```

compile option에 `-g` 옵션을 추가하고, 실행파일 하나 만든다. (예: function)

```
function function.c
gseo@DESKTOP-A6CM3FL:~/test$ gdb -tui function
```

`gdb -tui function` 명령어를 입력한다. (`-tui` 옵션을 사용하면 source에서 어느 라인에서 디버깅 되는지 실시간으로 알 수 있다.)

# GDB 명령어



```
root@gseo: /home/eddi_academy/EmbeddedMasterLv1/57/GwangseonEo/lecture/3_week

[ No Source Available ]

None No process In: L?? PC: ??
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
--Type <RET> for more, q to quit, c to continue without paging--
```

왼쪽 결과는 gdb -  
tui fucntion 명령어  
를 입력한 결과이다.

# function 예제 디버깅

b main : main문을 breakpoint 지정한다.  
b \*(주소) : 주소로 breakpoint를 지정한다.  
si : 명령어를 하나씩 수행한다.  
p/x \$(레지스터) : 레지스터의 값을 Read한다.  
x/gx (주소) : 주소의 값을 Read한다.  
q : gdb를 종료한다.

# function 메모리 맵

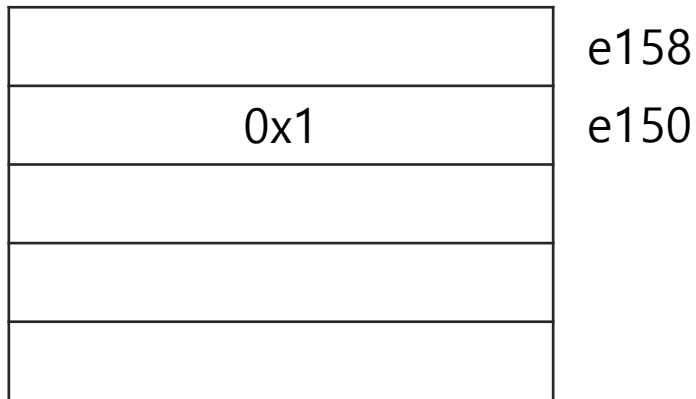
Dump of assembler code for function `main`:

```
0x00005555555515b <+0>:    endbr64
=> 0x00005555555515f <+4>:    push   %rbp
0x000055555555160 <+5>:    mov    %rsp,%rbp
0x000055555555163 <+8>:    sub    $0x10,%rsp
0x000055555555167 <+12>:   movl   $0x3,-0x8(%rbp)
0x00005555555516e <+19>:   mov    -0x8(%rbp),%eax
```

함수가 시작할 때마다 새로운 스택 프레임이 형성되는데, 이전의 어떤 상태로 돌아가기 위해 보존하거 저장하기 위해 rbp 값을 push 한다.

```
(gdb) p/x $rbp
$1 = 0x1
```

push %rbp 명령어를 실행하기 전 rbp 값을 보면, 0x1로 되어있다. push %rbp를 수행하면 rbp가 메모리 어떤 공간 안에 push된다.



rbp	0x1	0x1
rsp	0x7fffffffef158	0x7fffffffef158

push %rbp 수행 전

rbp	0x1	0x1
rsp	0x7fffffffef150	0x7fffffffef150

push %rbp 수행 후

# function 메모리 맵

```
0x00005555555515f <+4>:  push  %rbp
=> 0x000055555555160 <+5>:  mov   %rsp,%rbp
0x000055555555163 <+8>:  sub   $0x10,%rsp
0x000055555555167 <+12>: movl  $0x3,-0x8(%rbp)
0x00005555555516e <+19>: mov   -0x8(%rbp),%eax
```

새로운 스택 프레임을 형성하기 위한, 새로운 시작 위치를 rbp에 지정한다. rsp의 현 주소로 rbp가 위치하게 된다.

rbp	0x1	0x1
rsp	0x7fffffffef150	0x7fffffffef150

mov %rsp, %rbp 수행 전

rbp	0x7fffffffef150	0x7fffffffef150
rsp	0x7fffffffef150	0x7fffffffef150

mov %rsp, %rbp 수행 후

	E158
0x1	e150

rbp rsp



# function 메모리 맵

```
=> 0x000055555555163 <+8>:  sub    $0x10,%rsp
    0x000055555555167 <+12>: movl    $0x3,-0x8(%rbp)
    0x00005555555516e <+19>:  mov    -0x8(%rbp),%eax
    0x000055555555171 <+22>:  mov    %eax,%edi
```

코드에 있는 `int num=3;` 을 저장할 공간을 만들어 주기 위해 `rsp` 주소값에서 `0x10`을 뺀다.

```
(gdb) l
4      {
5          return num * 2;
6      }
7
8  int main(void)
9  {
10     int num = 3;
11     int result = multiply_two(num);
12     printf("result = %d\n", result);
13     return 0;
```

`int`는 4byte 공간을 할당해야 한다. 64bit 시스템은 포인터의 크기인 8byte 마다 메모리 공간이 할당된다. `rsp` 주소에서 `0x10(16)`을 빼서 지역 변수가 들어갈 공간을 만든다.

rbp		e158	rsp ↓ rsp
	0x1	e150	
		e148	
		e140	

# function 메모리 맵

```
0x000055555555163 <+8>:    sub    $0x10,%rsp  
=> 0x000055555555167 <+12>:   movl    $0x3,-0x8(%rbp)  
0x00005555555516e <+19>:   mov     -0x8(%rbp),%eax  
0x000055555555171 <+22>:   mov     %eax,%edi
```

movl : 32bit를 다룬다는 의미.  
0x3(3)을 rbp에서 0x8 뺀 주소 공간안에 대입  
한다.  $0xe150 - 0x8 = 0xe148$

```
(gdb) x/gx 0x7fffffffef148  
0x7fffffffef148: 0x0000555500000003
```

rbp		e158	rsp
	0x1	e150	
	0x3	e148	
		e140	

```
0x000055555555167 <+12>:   movl    $0x3,-0x8(%rbp)  
=> 0x00005555555516e <+19>:   mov     -0x8(%rbp),%eax  
0x000055555555171 <+22>:   mov     %eax,%edi
```

rbp에서  $-0x8$  뺀 주소에 들어간 값을 eax 레지  
스터에 대입한다.  $eax = 0x3$

```
(gdb) p/x $rax  
$1 = 0x3
```

# function 메모리 맵

```
0x00005555555516e <+19>:  mov    -0x8(%rbp),%eax  
=> 0x000055555555171 <+22>:  mov    %eax,%edi  
0x000055555555173 <+24>:  call   0x55555555149 <multiply_two>  
0x000055555555178 <+29>:  mov    %eax,-0x4(%rbp)  
0x00005555555517b <+32>:  mov    -0x4(%rbp),%eax
```

eax에 있는 값을 edi에 대입한다. edi 레지스터에는 0x3이 저장된다.

rbp

	e158
0x1	e150
0x3	e148
	e140

rsp

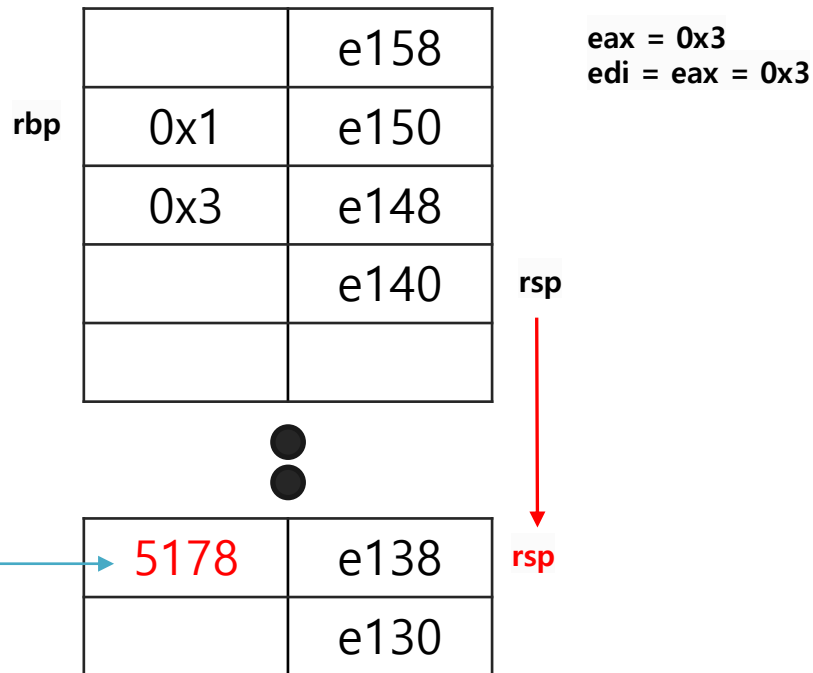
eax = 0x3

edi = eax = 0x3

# function 메모리 맵

```
0x00005555555516e <+19>: mov -0x8(%rbp),%eax
0x000055555555171 <+22>: mov %eax,%edi
=> 0x000055555555173 <+24>: call 0x55555555149 <multiply_two>
0x000055555555178 <+29>: mov %eax,-0x4(%rbp)
0x00005555555517b <+32>: mov -0x4(%rbp),%eax
0x00005555555517e <+35>: mov %eax,%esi
```

multiply\_two 함수를 call 한다. 함수를 call 하기 위해 새로운 스택 프레임을 형성해야 한다. call 명령어를 수행하면 rsp 값이 이동한다.



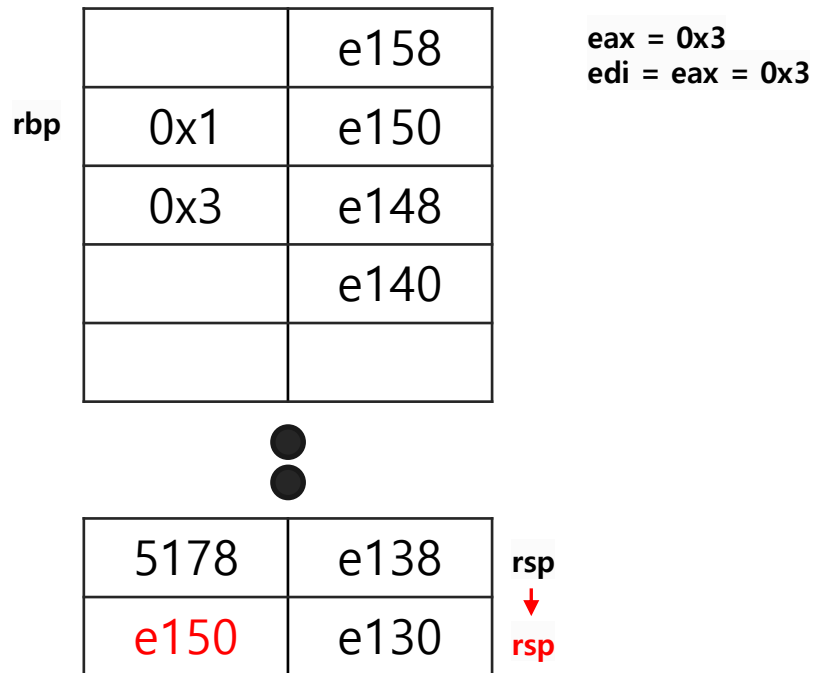
multiply\_two 함수를 끝내고, 다음 명령어를 수행해야 하기 위해 복귀할 주소를 저장한다.

# function 메모리 맵

```
Dump of assembler code for function multiply_two:
=> 0x000055555555149 <+0>:    endbr64
    0x00005555555514d <+4>:    push   %rbp
    0x00005555555514e <+5>:    mov    %rsp,%rbp
    0x000055555555151 <+8>:    mov    %edi,-0x4(%rbp)
```

```
Dump of assembler code for function multiply_two:
    0x000055555555149 <+0>:    endbr64
=> 0x00005555555514d <+4>:    push   %rbp
    0x00005555555514e <+5>:    mov    %rsp,%rbp
    0x000055555555151 <+8>:    mov    %edi,-0x4(%rbp)
```

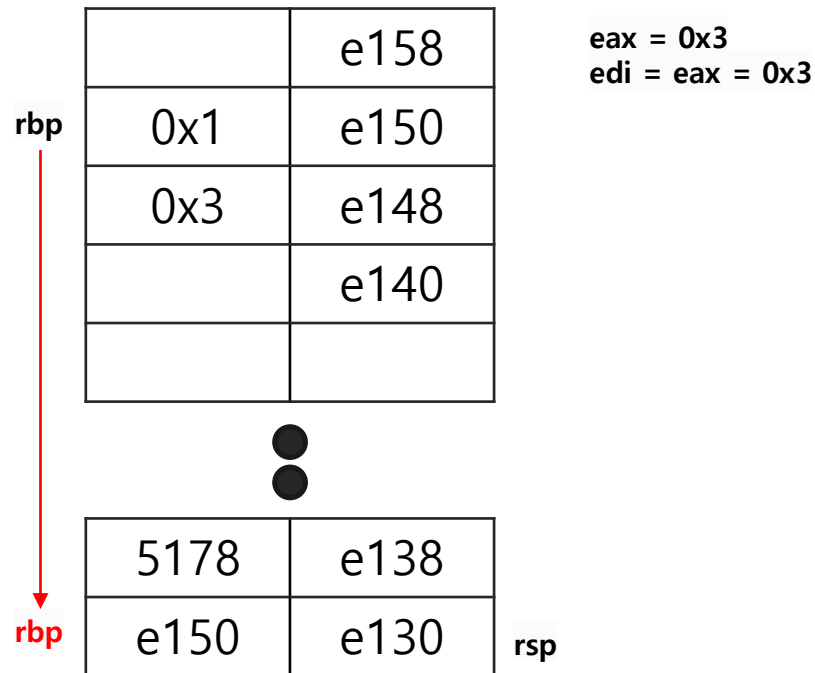
push %rbp를 수행하면, rsp가 다음 메모리 주소로 이동하고(8byte), 현재 rbp의 값이 rsp 메모리 주소 공간 안에 저장된다.



# function 메모리 맵

```
0x00005555555514d <+4>:    push    %rbp
0x00005555555514e <+5>:    mov     %rsp,%rbp
=> 0x000055555555151 <+8>:    mov     %edi,-0x4(%rbp)
0x000055555555154 <+11>:   mov     -0x4(%rbp),%eax
```

mov %rsp, %rbp를 수행하면, rbp가 가리키는 메모리 주소가 rsp와 동일하게 된다.



# function 메모리 맵

```
0x00005555555514d <+4>:    push    %rbp
0x00005555555514e <+5>:    mov     %rsp,%rbp
=> 0x000055555555151 <+8>:    mov     %edi,-0x4(%rbp)
0x000055555555154 <+11>:   mov     -0x4(%rbp),%eax
```

edi = 0x3 이고, 현재 rbp가 가리키고 있는 주소에서 0x4를 뺀 주소에 edi 값을 저장한다.

```
(gdb) x/gx 0x7fffffffef12c
0x7fffffffef12c: 0xfffff15000000003
```

	e158
0x1	e150
0x3	e148
	e140



5178	e138	
e150	e130	
0x3	e12c	
	e128	

rbp

rsp

eax = 0x3  
edi = eax = 0x3

# function 메모리 맵

```
0x000055555555151 <+8>:  mov    %edi,-0x4(%rbp)
=> 0x000055555555154 <+11>: mov    -0x4(%rbp),%eax
0x000055555555157 <+14>:  add    %eax,%eax
0x000055555555159 <+16>:  pop    %rbp
0x00005555555515a <+17>:  ret
```

rbp가 가리키는 주소에서 0x4뺀 주소에 있는 값(0x3)을 eax에 저장한다.  
-> 이미 저장되어 있는데.. 또?

```
(gdb) p/x $rax
$2 = 0x3
```

	e158
0x1	e150
0x3	e148
	e140



	5178	e138	
	e150	e130	
	0x3	e12c	
		e128	

rbp

rsp

eax = 0x3  
edi = eax = 0x3



# function 메모리 맵

```
0x000055555555154 <+11>: mov    -0x4(%rbp),%eax  
=> 0x000055555555157 <+14>: add    %eax,%eax  
0x000055555555159 <+16>: pop    %rbp  
0x00005555555515a <+17>: ret
```

eax에 eax를 더한다.  $0x3 + 0x3 = 0x6$

```
(gdb) p/x $rax  
$3 = 0x6
```

	e158
0x1	e150
0x3	e148
	e140



	5178	e138	
	e150	e130	
	0x3	e12c	
		e128	

rbp

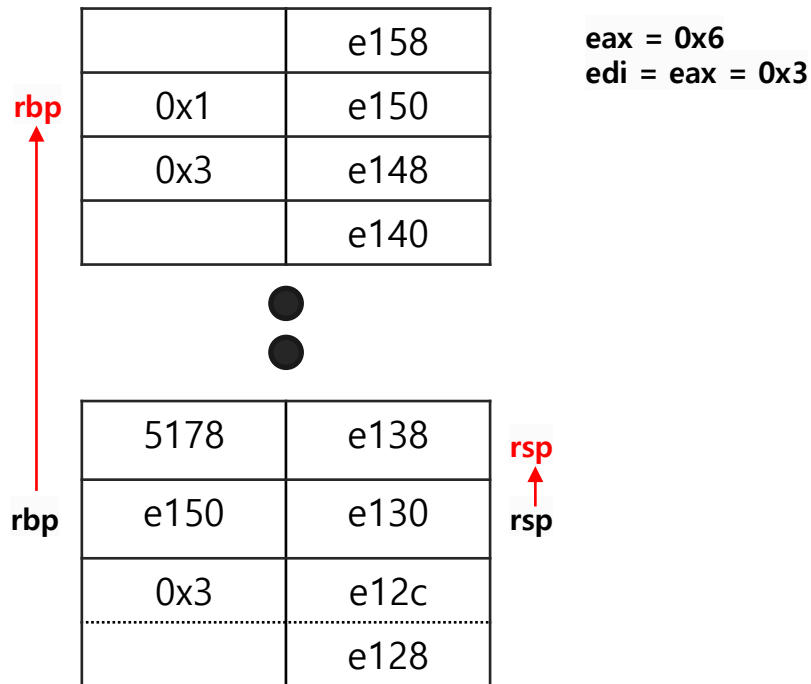
rsp

**eax = 0x6**  
**edi = eax = 0x3**

# function 메모리 맵

```
0x000055555555157 <+14>: add    %eax,%eax  
=> 0x000055555555159 <+16>: pop    %rbp  
0x00005555555515a <+17>: ret
```

multiply\_two 함수 안의 명령어들을 끝내면  
pop %rbp를 수행하면 rbp에 현재 rsp에 있는  
값 (e150)을 넣어주고, rsp를 +0x8 해준다.



# function 메모리 맵

```
0x000055555555157 <+14>: add    %eax,%eax
0x000055555555159 <+16>: pop    %rbp
=> 0x00005555555515a <+17>: ret
```

ret는 이전 함수로 돌아가는 명령어  
실제 구성은 아래와 같다.

pop rip

jmp rip

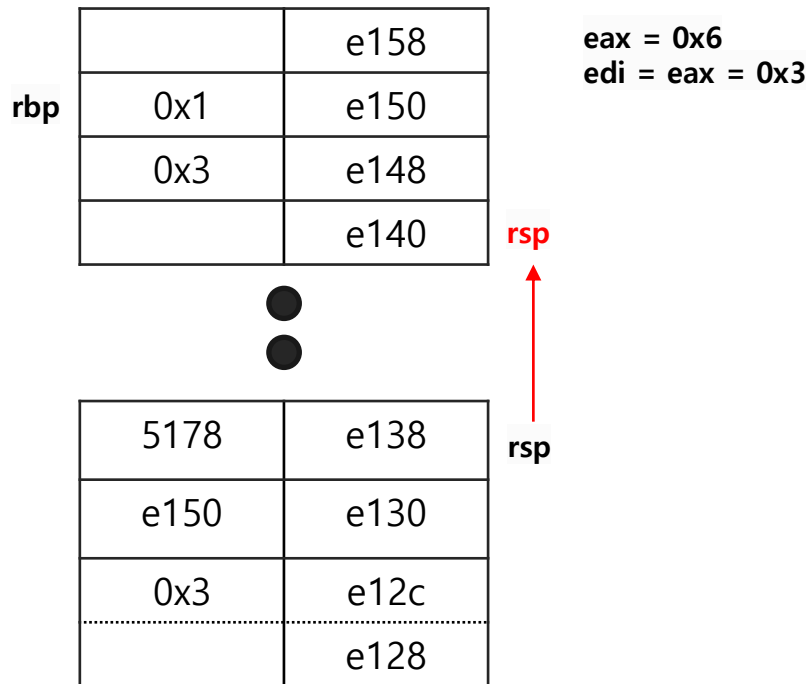
현재 rsp에 있는 값을 rip에 넣은 뒤 해당 위치  
로 점프 한다.

```
(gdb) p/x $rip
$4 = 0x5555555515a
```

ret 수행 전

```
(gdb) p/x $rip
$6 = 0x55555555178
```

ret 수행 후



# function 메모리 맵

```
=> 0x000055555555178 <+29>: mov    %eax,-0x4(%rbp)
0x00005555555517b <+32>: mov    -0x4(%rbp),%eax
0x00005555555517e <+35>: mov    %eax,%esi
0x000055555555180 <+37>: lea    0xe7d(%rip),%rax    # 0x55555556004
```

eax에 있는 값(0x6)을 rbp 주소 -0x4에 저장한다. e14c에 저장.

```
0x000055555555178 <+29>: mov    %eax,-0x4(%rbp)
=> 0x00005555555517b <+32>: mov    -0x4(%rbp),%eax
0x00005555555517e <+35>: mov    %eax,%esi
0x000055555555180 <+37>: lea    0xe7d(%rip),%rax    # 0x55555556004
```

rbp가 가리키는 주소에서 0x4를 뺀 메모리주소 공간 안의 값을 eax에 저장한다.  
eax = 0x6

```
0x00005555555517b <+32>: mov    -0x4(%rbp),%eax
=> 0x00005555555517e <+35>: mov    %eax,%esi
0x000055555555180 <+37>: lea    0xe7d(%rip),%rax    # 0x55555556004
0x000055555555187 <+44>: mov    %rax,%rdi
```

eax 값을 esi에 저장한다.

	e158
rbp	0x1
	e150
	0x6
	e14c
	0x3
	e148
	e140

rsp

eax = 0x6  
edi = eax = 0x3

# function 메모리 맵

```
0x00005555555517e <+35>: mov    %eax,%esi  
=> 0x000055555555180 <+37>: lea    0xe7d(%rip),%rax    # 0x555555556004  
0x000055555555187 <+44>: mov    %rax,%rdi  
0x00005555555518a <+47>: mov    $0x0,%eax
```

lea 0xe7d(%rip), %rax

좌변의 주소값을 우변의 주소값에 저장.

```
0x000055555555180 <+37>: lea    0xe7d(%rip),%rax    # 0x555555556004  
=> 0x000055555555187 <+44>: mov    %rax,%rdi  
0x00005555555518a <+47>: mov    $0x0,%eax  
0x00005555555518f <+52>: call   0x55555555050 <printf@plt>
```

mov %rax, %rdi 를 통해 rax값을 rdi에 저장

rbp		e158	rsp
	0x1	e150	
	0x6	e14c	
	0x3	e148	
		e140	

rbp		e158	rsp	eax = 0x6004 edi = 0x6004
	0x1	e150		
	0x6	e14c		
	0x3	e148		
		e140		

# function 메모리 맵

```
0x000055555555187 <+44>: mov %rax,%rdi  
=> 0x00005555555518a <+47>: mov $0x0,%eax  
0x00005555555518f <+52>: call 0x555555555050 <printf@plt>  
0x000055555555194 <+57>: mov $0x0,%eax
```

0x0을 eax에 저장한다.

```
0x00005555555518a <+47>: mov $0x0,%eax  
=> 0x00005555555518f <+52>: call 0x555555555050 <printf@plt>  
0x000055555555194 <+57>: mov $0x0,%eax  
0x000055555555199 <+62>: leave  
0x00005555555519a <+63>: ret
```

printf 함수를 call 한다.

rbp

	e158
0x1	e150
0x6	e14c
0x3	e148
	e140

rsp

eax = 0x0  
edi = 0x6004

# function 메모리 맵

```
0x00005555555518f <+52>: call 0x555555555050 <printf@plt>
=> 0x000055555555194 <+57>: mov $0x0,%eax
0x000055555555199 <+62>: leave
0x00005555555519a <+63>: ret
```

eax에 0x0을 넣는다.

```
rbp 0x1 0x1
rsp 0x7fffffff158 0x7fffffff158
```

rbp

	e158
0x1	e150
0x6	e14c
0x3	e148
	e140

rsp



rsp

eax = 0x0  
edi = 0xdbe0

```
0x000055555555194 <+57>: mov $0x0,%eax
=> 0x000055555555199 <+62>: leave
0x00005555555519a <+63>: ret
```

이전 함수로 돌아가기 전, 현재 함수의 스택을 정리하는 명령어.

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