

시스템 해킹 입문하기 1회

2019. 03. 19.

활동 계획과 사전지식





INDEX



001/ 시스템 해킹을 알아보자

002/ 컴파일이란?

003/ 어셈블리와 디버깅

004/ 스택에서는 무슨 일이?

005/ Buffer Over Flow(BOF)



지피지기면 '백전백승' 이니라~@@

시스템 해킹을 알아보자

- 시스템 해킹이 뭘까?
- 대략적인 커리큘럼



소프트웨어나 운영체제등의 취약점을 공격하는 해킹 분야 시스템 해킹이 뭘까?



⇒ 소프트웨어는 대부분 사람이 만든다.

⇒ 사람이 만들다 보니 **치명적인 실수**가 있을 수 있다.

⇒ **개발자가 의도하지 않은 부분(취약점)** 을 공격해 악의적인 행동을 하는 것이 시스템 해킹!

※악의적인 행동? -> root 탈취 후 백도어 설치, 로그 삭제, 관리 시스템 무력화 등등.. 무궁무진



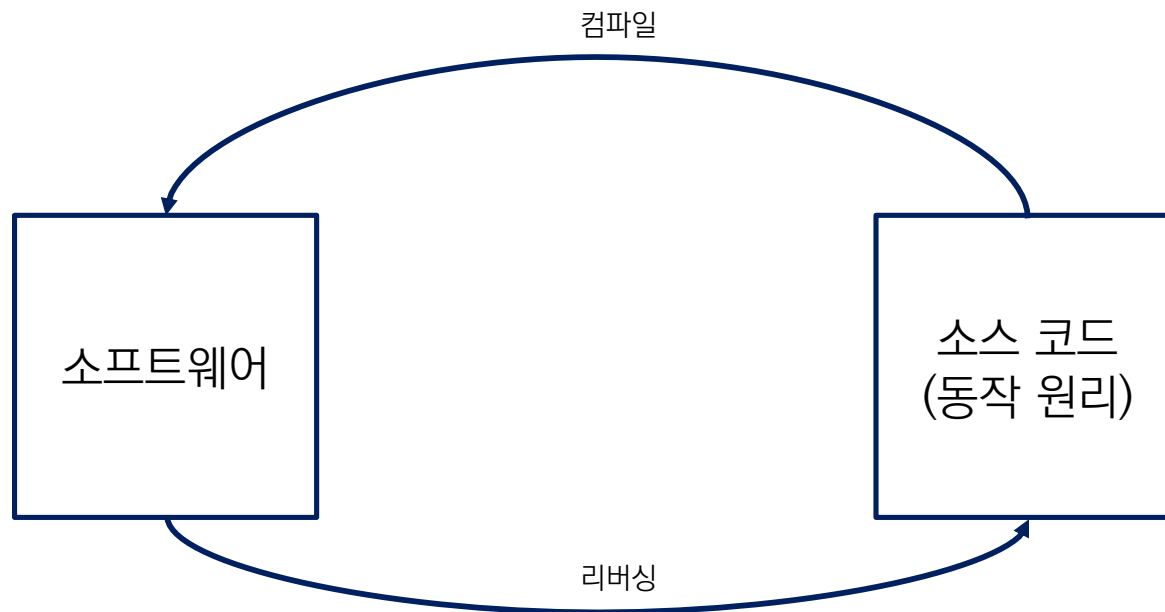
나쁘게 말하면 남의 집 문 열고 들어가서 도둑질,,

그래서 우리는..

대략적인 커리큘럼



취약점을 알려면 동작원리를 알아야겠지? → 동작원리 분석은 리버싱!!





그래서 우리는..

대략적인 커리큘럼

리버싱의 기초(공포의 어셈블리, 스택, 함수 호출 등등..)

메모리 공격(BOF, UAF, 기회가 된다면 끝판왕 ROP도)

메모리 보호 기법(canary, NX, PIE...)

조금 욕심 부려서 셸코드 작성도 $\pi\pi$

걱정 마세요 어차피 이거 다 못해요 ㅎ

컴파일러는 소스코드를 외계어로 바꾼다;;

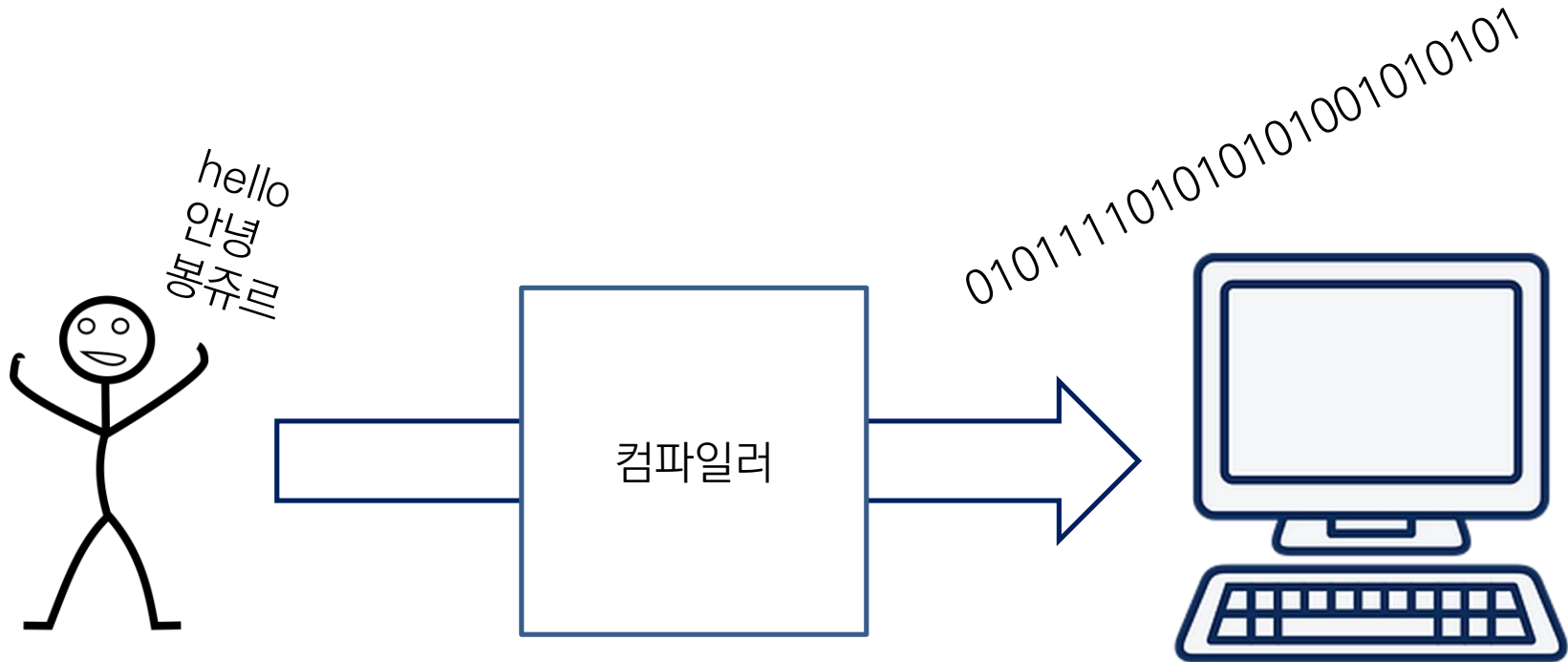
컴파일이란?

- 컴파일, 복습하자
- (실습) hello, GCC!



컴퓨터는 이진수만 알아먹는다.. 이진수로 말할 줄 아는 분..?

컴파일, 복습하자



컴파일러는 자연어(인간의 언어)를 기계어(이진수)로 바꾸어 전달해 주는 통역사.

컴퓨터는 이진수만 알아먹는다.. 이진수로 말할 줄 아는 분..?

컴파일, 복습하자



우리는 컴파일러중에 gcc(GNU C Compiler)를 쓸거예요.

앞으로 질리도록 볼 친구이니 친해집시다 ㅎㅎ

```
minibee@argos-edu:~/C$ vim test.c
minibee@argos-edu:~/C$ gcc -o test test.c
minibee@argos-edu:~/C$ ls -l
total 16
-rwxrwxr-x 1 minibee minibee 8304 Mar 17 16:15 test
-rw-rw-r-- 1 minibee minibee 66 Mar 17 16:15 test.c
minibee@argos-edu:~/C$
```

명령어 -> gcc -o <출력할 이름> <소스코드 파일>

직접 작성한 C 소스코드를 컴파일하고 실행해보자~ (실습) hello, GCC!



- 1) “vim hello.c” 입력 -> i 눌러서 편집모드
- 2) 아무 문자열이나 출력하는 C 코드 작성
- 3) ESC(커맨드 모드) -> :wq 입력 (저장하고 나가기)
- 4) gcc -o hello hello.c (컴파일 하기)
- 5) ./hello (실행파일 실행)

※도움이 필요할 때 튜터를 불러주세요

가차없이 튜터 소환 @_@

```
minibeef@argos-edu:~/C$ vim hello.c
```

```
#include <stdio.h>

int main()
{
    puts("hello, GCC!");
    return 0;
}
```

```
~
~
:wq
```

```
minibeef@argos-edu:~/C$ gcc -o hello hello.c
minibeef@argos-edu:~/C$ ./hello
hello, GCC!
minibeef@argos-edu:~/C$
```

본격적으로 리버싱을 해봅시다 ㅎㅎㅎㅎ

어셈블리와 디버깅

- 리버싱을 하는데 레지스터를 모른다고?
- 기초 어셈블리/디버깅
- (실습) hello reversing





어셈은 레지스터로 되어 있다아.

리버싱을 하는데 레지스터를 모른다고?

어셈블리어는 기계어와 일대일 대응 관계예요,,
그래서 소스코드 전부를 알 수는 없어도 프로그램의 흐름을 캐치할 수는 있죠
리버싱에서는 어셈블리가 필수입니다!!!

2진수 기계어 : 001000 00001 00000 00000000000001010

16진수 기계어 : 08 01 00 000A

어셈블리 : `addi $0, $1, 10` // 뜻 : 1번 레지스터에 이진수 10을 더한 값을 0번 레지스터에 저장!

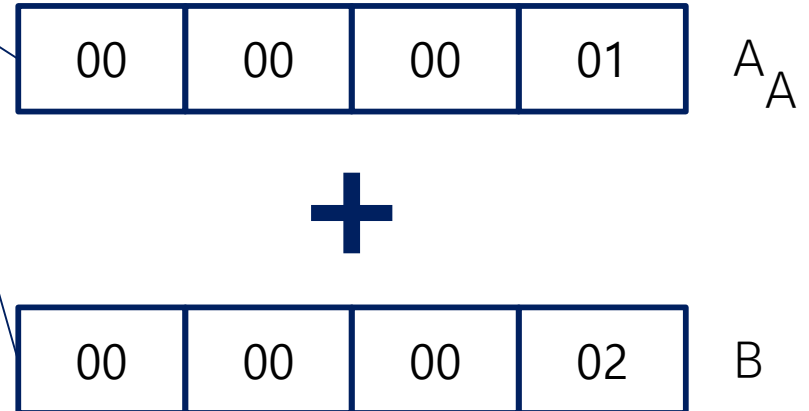
=> 그런데 레지스터가 뭐지..?

어셈블리 레지스터로 되어 있다.

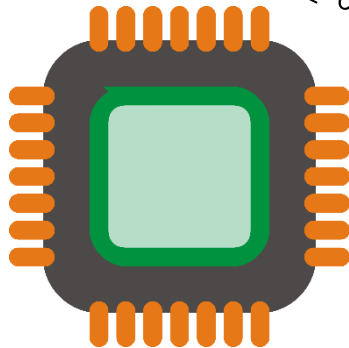
리버싱을 하는데 레지스터를 모른다고?

- ⇒ 프로세서가 연산을 하는 데에는 저장소가 필요해요.
- ⇒ 그래서 생긴 것이 바로 레지스터~

레지스터



1 더하기 2를 해야하는데
1이랑 2는 어디서 가져오냐 ???



<cpu 프로세서>

어셈블리 코드 : add A, B

즉, $A = A + B$ 랑 같은 소리!!

어셈블리 레지스터로 되어 있다.

리버싱을 하는데 레지스터를 모른다고?

[표 2-4] 레지스터의 종류와 용도

범주	80386 레지스터	이름	비트	용도
범용 레지스터 (General Register)	EAX	누산기 (Accumulator)	32	주로 산술 연산에 사용(함수의 결과 값 저장)
	EBX	베이스 레지스터 (Base Register)	32	특정 주소 한 인덱스
	ECX	카운트 레지스터 (Count Register)	32	반복적으로 프의 반복 수 기억)
	EDX	데이터 레지스터 (Data Register)	32	일반 자료
세그먼트 레지스터 (Segment Register)	CS	코드 세그먼트 레지스터 (Code Segment Register)	16	실행될 기 지정
	DS	데이터 세그먼트 레지스터 (Data Segment Register)	16	프로그램 영역의 매
	SS	스택 세그먼트 레지스터 (Stack Segment Register)	16	프로그램 나 사용자 subroutin
	ES, FS, GS	엑스트라 세그먼트 레지스터 (Extra Segment Register)	16	문자 연산 용되는 여

레지스터의 종류와 기능

■ [표 2-4] 레지스터의 종류와 용도

범주	80386 레지스터	이름	비트	용도
포인터 레지스터 (Pointer Register)	EBP	베이스 포인터 (Base Pointer)	32	SS 레지스터와 함께 사용되어 <u>스택</u> 내의 변수 값을 읽는 데 사용
	ESP	스택 포인터 (Stack Pointer)	32	SS 레지스터와 함께 사용되며 <u>스택의</u> 가장 끝 주소를 가리킴
	EIP	명령 포인터 (Instruction Pointer)	32	다음 명령어의 오프셋(Offset, 상대 위치 주소)을 저장하며 CS 레지스터와 합쳐져 다음에 수행될 명령의 주소 형성
인덱스 레지스터 (Index Register)	EDI	목적지 인덱스 (Destination Index)	32	목적지 주소에 대한 값 저장
	ESI	출발지 인덱스 (Source Index)	32	출발지 주소에 대한 값 저장
플래그 레지스터	EFLAGS	플래그 레지스터 (Flag Register)	32	연산 결과 및 시스템 상태와 관련된 여러 가지 플래그 값 저장

레지스터 종류는 소름 돋게 많으니 그때그때 필요할 때 외우는 걸로 해요

기초 어셈블리



```
0x000000000000006ed <+0>:    push    rbp
0x000000000000006ee <+1>:    mov     rbp, rsp
0x000000000000006f1 <+4>:    sub     rsp, 0x10
0x000000000000006f5 <+8>:    lea     rax, [rbp-0x10]
0x000000000000006f9 <+12>:   mov     rdi, rax
0x000000000000006fc <+15>:   mov     eax, 0x0
0x00000000000000701 <+20>:   call    0x5b0 <gets@plt>
0x00000000000000706 <+25>:   lea     rax, [rbp-0x10]
0x0000000000000070a <+29>:   mov     rsi, rax
0x0000000000000070d <+32>:   lea     rdi, [rip+0xa8]      # 0x7bc
0x00000000000000714 <+39>:   mov     eax, 0x0
0x00000000000000719 <+44>:   call    0x5a0 <printf@plt>
0x0000000000000071e <+49>:   mov     eax, 0x0
0x00000000000000723 <+54>:   leave
0x00000000000000724 <+55>:   ret
```

어셈블리는 이렇게 생겼어요, 벌써부터 신나지 않나요..!

프로그램을 까 재껴보자

기초 어셈블리



```
mov <목표> <소스>  
add <목표> <소스>  
push <대상>
```

·
·
·

명령어 <인자> <인자> ...

감이 잡히시나요? 별로 복잡하진 않아요
자세한건 실전에서 ㄱ ㄱ



디버깅?

- ⇒ 디버거에 프로그램을 넣으면 그 프로그램의 어셈블리 코드를 보여준다
- ⇒ 리버스 엔지니어링의 영원한 동반자

우리는 교육서버에 미리 설치되어있는 gdb라는 디버거를 사용할 것이다@@ 사용법은

`gdb <프로그램 이름>`

```
hello hello.c test test.c
minibeef@argos-edu:~/C$ gdb hello
```

설마 아까 만든 hello.. 지우진 않았겠지?

(실습 1) hello reversing



- 1) gdb hello (hello파일을 대상으로 디버깅 하겠다.)
- 2) Set disassembly-flavor intel (어셈블리 문법을 이쁘게)
- 3) disas main (hello의 main함수를 디버깅)

```
(gdb) set disassembly-flavor intel
(gdb) disas main
Dump of assembler code for function main:
   0x000000000000063a <+0>:    push    rbp
   0x000000000000063b <+1>:    mov     rbp, rsp
   0x000000000000063e <+4>:    lea     rdi, [rip+0x9f]          # 0x6e4
   0x0000000000000645 <+11>:   call    0x510 <puts@plt>
   0x000000000000064a <+16>:   mov     eax, 0x0
   0x000000000000064f <+21>:   pop     rbp
   0x0000000000000650 <+22>:   ret
End of assembler dump.
(gdb) █
```



설마 아까 만든 hello.. 지우진 않았겠지?

(실습 1) hello reversing

```
0x0000000000000063e <+4>:    lea    rdi,[rip+0x9f]    # 0x6e4
0x00000000000000645 <+11>:    call   0x510 <puts@plt>
```

lea는 오른쪽에 있는 친구의 주소를 왼쪽으로 복사하는 명령이에요.

오른쪽에 있는 친구는 0x6e4로, 문자열로 변환하면 이렇게 되어요.

```
(gdb) x/s 0x6e4
0x6e4: "hello, GCC!"
(gdb) █
```

(x/s <16진수>를 하면 16진수의 문자열을 볼 수 있다!)
-google에 >>gdb x 명령어 검색<<

rdi에 있던 문자열을 puts 함수가 출력했다..
아하, rdi는 함수의 인자로 이용되는 레지스터군!

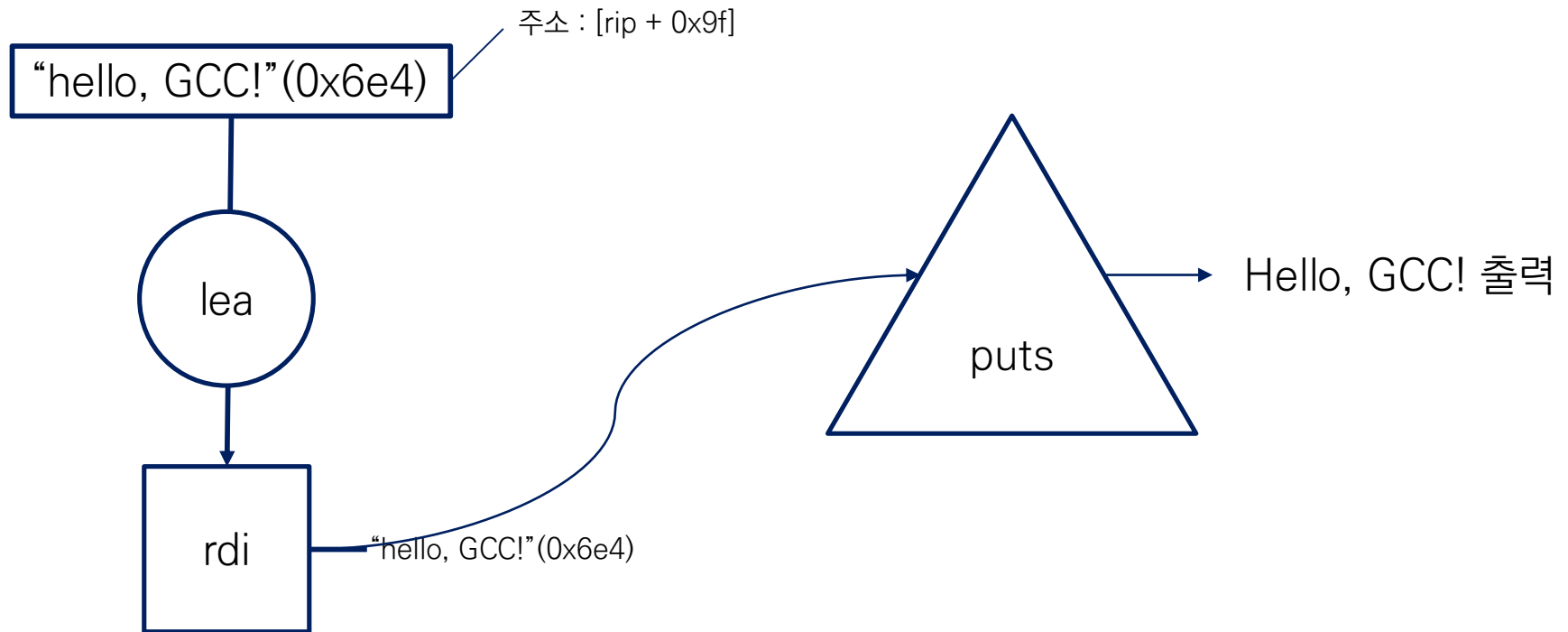
덧붙여 rsi도 함수 인자 전달 레지스터예요



설마 아까 만든 hello.. 지우진 않았겠지?

(실습 1) hello reversing

```
0x0000000000000063e <+4>:    lea    rdi,[rip+0x9f]    # 0x6e4
0x00000000000000645 <+11>:    call   0x510 <puts@plt>
```





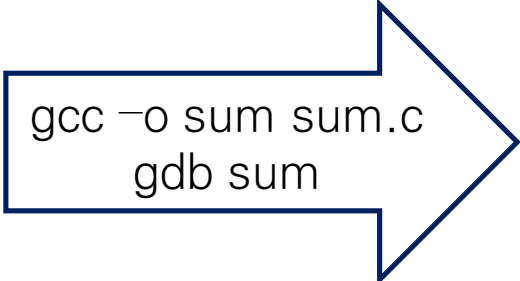
설마 아까 만든 hello.. 지우진 않았겠지?
(실습 2) sum reversing

```
#include <stdio.h>

int main()
{
    int a, b;
    a = 4;
    b = 5;

    printf("%d", a+b);

    return 0;
}
```



```
gcc -o sum sum.c
gdb sum
```

〈다음과 같이 sum.c를 작성하고
컴파일 해봅시다 ㅎㅎ..〉



설마 아까 만든 hello.. 지우진 않았겠지?

(실습 2) sum reversing

```
0x0000000000000064a <+0>:    push    rbp
0x0000000000000064b <+1>:    mov     rbp, rsp
0x0000000000000064e <+4>:    sub     rsp, 0x10
0x00000000000000652 <+8>:    mov     DWORD PTR [rbp-0x8], 0x4
0x00000000000000659 <+15>:   mov     DWORD PTR [rbp-0x4], 0x5
0x00000000000000660 <+22>:   mov     edx, DWORD PTR [rbp-0x8]
0x00000000000000663 <+25>:   mov     eax, DWORD PTR [rbp-0x4]
0x00000000000000666 <+28>:   add     eax, edx
0x00000000000000668 <+30>:   mov     esi, eax
0x0000000000000066a <+32>:   lea     rdi, [rip+0xa3]          # 0x714
0x00000000000000671 <+39>:   mov     eax, 0x0
0x00000000000000676 <+44>:   call    0x520 <printf@plt>
0x0000000000000067b <+49>:   mov     eax, 0x0
0x00000000000000680 <+54>:   leave
0x00000000000000681 <+55>:   ret
```

1) 스택에 4 저장, 5 저장

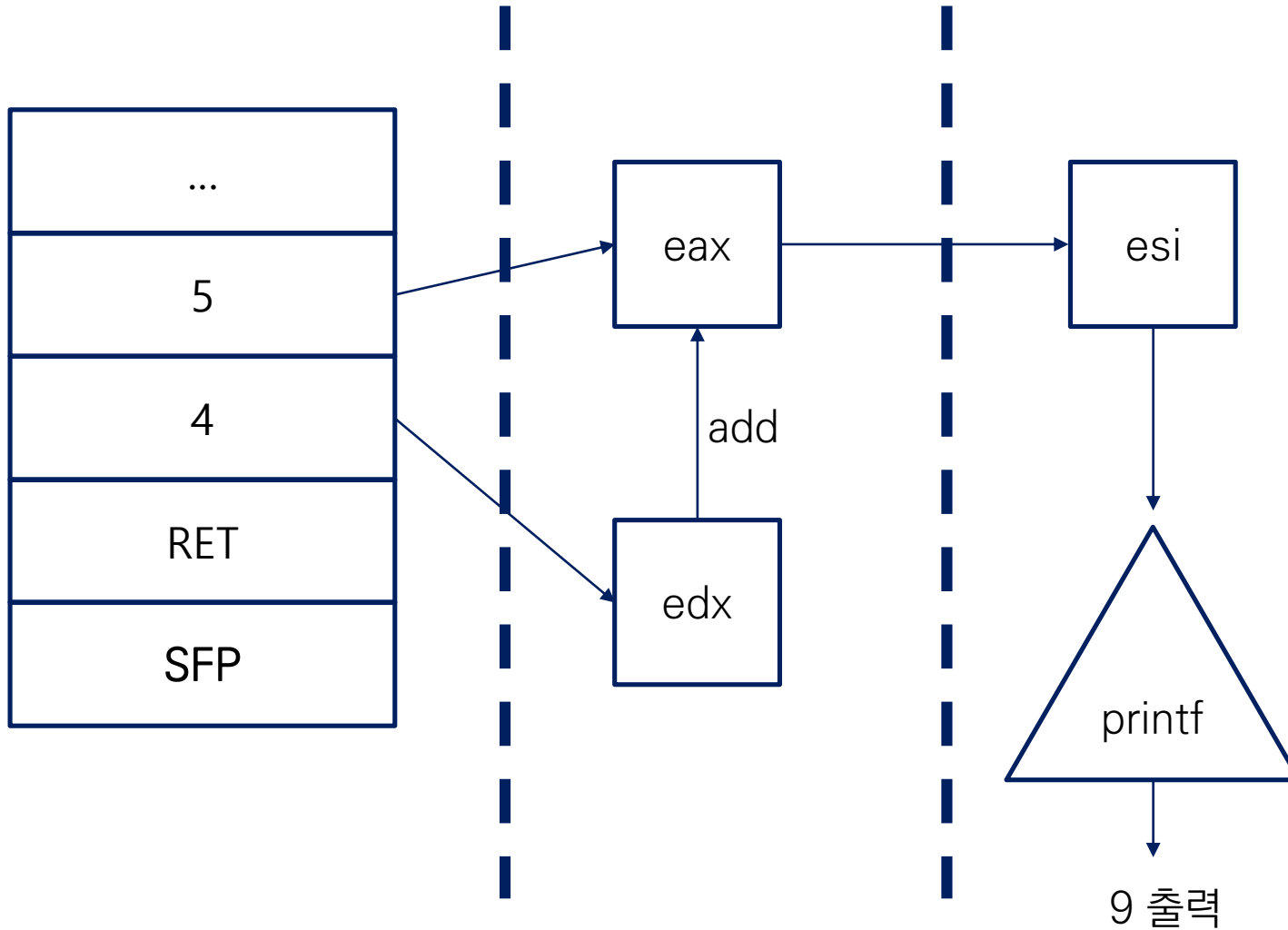
2) 각각 edx, eax에 옮김

3) add eax, edx (eax = eax + edx)

4) 인자에 eax(4+5) 저장

설마 아까 만든 hello.. 지우진 않았겠지?

(실습 2) sum reversing



Q & A

Thank You for Listening

