

# Pwn-Basics 2025

CPU, 메모리, 레지스터 동작 원리부터 어셈블리 언어 심층 이해까지, 시스템 취약점 분석 및 악스플로잇의 핵심 기초 개념을 다룬다.

# 프로그래밍 언어의 계층 구조

프로그래밍 언어는 인간과 기계 사이의 거리에 따라 여러 단계로 나뉜다. 각 레벨은 추상화 정도와 하드웨어 제어 수준이 다르다.



## Machine Code (기계어)

CPU가 직접 실행하는 0과 1의 바이트 시퀀스이다. 가장 낮은 수준의 언어로, 사람이 직접 읽고 쓰기는 거의 불가능하다.



## Low-Level (저수준 언어)

Assembly 언어가 대표적이다. CPU 명령어와 레지스터, 메모리를 직접 제어하며 기계어와 1:1 대응된다.



## Mid-Level (중간 수준 언어)

C, C++ 등이 해당한다. 인간이 읽기 쉬운 문법이지만 포인터와 메모리를 직접 제어할 수 있다.



## High-Level (고수준 언어)

Python, Java 등이 대표적이다. 메모리와 저수준 세부 처리를 대부분 숨겨 개발자가 로직에만 집중할 수 있다.

# C 프로그램의 컴파일 과정

C 소스 코드가 실행 가능한 프로그램이 되기까지는 여러 단계의 변환 과정을 거친다. 각 단계마다 다른 도구가 사용된다.



- 로더(Loader)가 최종 실행 파일을 메모리에 적재하면, CPU가 기계어를 직접 실행하게 된다.

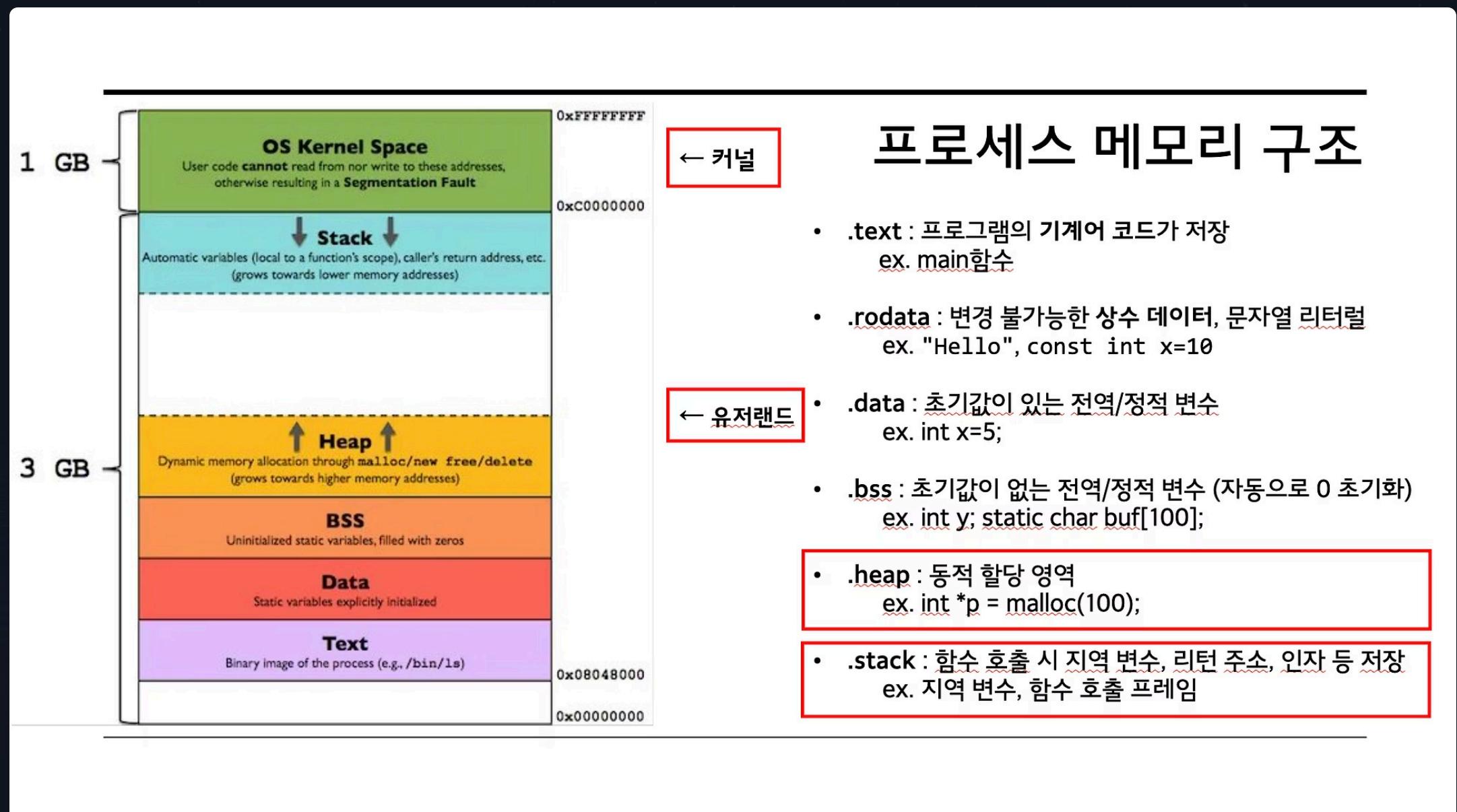
# 메모리 구조

이렇게 생성된 실행파일(바이너리)은 특정한 메모리 구조로 이루어져있음.

커널 - 운영 체제(OS)의 가장 핵심적인 부분으로, 컴퓨터의 하드웨어 자원(CPU, 메모리, 입출력 장치)과 응용 프로그램 사이에서 다리 역할을 하며 이 모든 것을 관리하고 제어하는 프로그램(대가리)

유저랜드 - 사용자가 직접 접근하여 프로그램을 실행하는 '사용자 공간(User Space)' (따끼리)

여기서 우리가 공부할 것은 유저랜드임.



# 예시

```
#include <stdio.h>

char name[64];      //함수 밖에서 초기값 없이 선언된 name 배열은 bss에 위치
int b = 4;          //함수 밖에서 초기값 4로 선언된 b는 .data 영역에 위치

int main(void) {
    int a = 10;       //함수 내에서 선언된 a는 지역변수로 .stack 영역에 위치
    int *p = malloc(20); //malloc(20)으로 동적 할당된 영역은 .heap 영역에 위치

    printf("Hello World!"); //여기서 printf()의 인자인 "Hello World"은 .rodata 영역에 위치
}

}
```

# stack

"유저랜드를 공부한다" 할때는 크게 스택과 힙으로 나뉨

스택이란? 각 함수들이 호출될때 임시로 데이터를 쌓아두는 공간

핵심 특징 (일단 외워만 두셈)

- **LIFO** (Last In, First Out) - 마지막에 넣은 게 먼저 나옴
- 높은 주소 → 낮은 주소로 자람
- 함수 끝나면 자동으로 정리됨

# 메모리와 레지스터의 관계

## 메모리란?

프로그램이 실행될 때 데이터를 임시로 저장하는 공간이다. CPU가 계산하려면 데이터를 어딘가에서 꺼내오고, 결과를 다시 저장해야 하는데 그때 사용하는 저장 창고가 바로 메모리이다.

**메모리 주소**는 각 데이터가 저장된 위치를 나타내는 고유한 번호이다.

## 레지스터란?

CPU가 계산할 때 직접 사용하는 초고속 임시 저장 공간이다. CPU는 대부분의 명령을 레지스터 안의 값으로 연산한다.

메모리에 데이터를 저장하거나 함수의 인자로 전달할 때 레지스터를 통해 이루어진다.

쉽게 비유하면: 메모리는 집, 메모리 주소는 집주소, 레지스터는 배달부이다.

# 함수 호출 시 레지스터의 역할

C 코드에서 함수를 호출할 때 한줄로 단순하게 적어주지만, 실제로는 레지스터를 통해 값이 전달된다.

## C 코드 예시

```
int add(int x, int y) {  
    return x + y;  
}  
  
// 함수 호출  
add(1, 2);
```

## 어셈블리 코드 (실제 동작)

```
mov rdi, 1  
mov rsi, 2  
call add
```

첫 번째 인자는 **rdi**, 두 번째 인자는 **rsi** 레지스터를 통해 전달된다.

# 주요 범용 레지스터

x86-64 아키텍처에서 자주 사용되는 레지스터들의 역할과 용도를 정리한다.

## RAX (Accumulator)



연산 결과와 함수의 반환값을 저장한다.

하위 접근: EAX, AX, AH, AL

## RBX (Base)



데이터 보관 및 베이스 주소로 사용된다.

하위 접근: EBX, BX, BH, BL

## RCX (Counter)



반복문과 루프, 문자열 연산에 사용되며 4번째 인자를 전달한다.

하위 접근: ECX, CX, CH, CL

## RDX (Data)



곱셈·나눗셈 보조 및 3번째 인자를 전달한다.

하위 접근: EDX, DX, DH, DL

## RSI (Source Index)



메모리 복사 시 소스 주소, 2번째 인자를 전달한다.

하위 접근: ESI, SI, SIL

## RDI (Destination Index)



메모리 복사 시 목적지 주소, 1번째 인자를 전달한다.

하위 접근: EDI, DI, DIL

# 제어 흐름 레지스터

## RBP (Base Pointer)

함수의 스택 프레임 기준 주소를 저장한다.  
이전 함수의 RBP 값이 여기에 저장된다.

## RSP (Stack Pointer)

스택의 현재 최상단 위치를 가리킨다.  
push/pop 명령이 실행될 때마다 자동으로  
변경된다.

## RIP (Instruction Pointer)

다음에 실행할 명령어의 주소를 저장한다. 프  
로그램의 실행 흐름을 제어한다.

- ▣ **핵심 요약:** RIP, RSP가 프로그램의 실행 흐름을 제어하고, RAX~RDI는 데이터 연산을 담당하며, RBP는 함수 스택의 기준점을 유지한다.

# 플래그 레지스터 (RFLAGS)

CPU가 연산을 수행한 후, 그 결과의 상태를 0과 1로 저장하는 특수한 레지스터이다. 조건 분기문에서 매우 중요한 역할을 한다.

## ZF (Zero Flag)

연산 결과가 0일 때 1로 설정된다.

예: `cmp eax, eax` → ZF=1

## CF (Carry Flag)

자리올림 또는 언더플로가 발생하면 1로 설정된다.

예: `0xFF + 1` → CF=1

## SF (Sign Flag)

연산 결과가 음수일 때 1로 설정된다.

예: `sub eax, 10` (결과가 음수)

## OF (Overflow Flag)

부호 있는 연산에서 오버플로가 발생하면 1로 설정된다.

예: `127 + 1` → OF=1

# 어셈블리 기본 명령어



## mov - 데이터 복사

mov rax, 8 → RAX에 8을 저장한다

mov [rbp-8], rax → 스택에 RAX 값을 저장한다



## add - 덧셈 연산

add rax, 5 → RAX = RAX + 5



## sub - 뺄셈 연산

sub rax, 2 → RAX = RAX - 2



## cmp - 값 비교

cmp rax, 5 → RAX와 5를 비교하여 플래그를 설정한다



## je/jne - 조건 분기

je label → ZF=1이면 label로 점프한다

jne label → ZF=0이면 label로 점프한다



## call/ret - 함수 호출

call func → 함수 호출 및 복귀 주소를 저장한다

ret → 함수에서 복귀한다



## push/pop - 스택 조작

push rax → 스택에 RAX를 저장한다

pop rbx → 스택에서 값을 꺼내 RBX에 저장한다

# 실전 연습 퀴즈

지금까지 배운 내용을 바탕으로 레지스터 값과 플래그 상태를 추적해 본다.

## 퀴즈 1: 레지스터 값 추적

```
mov rax, 5  
mov rbx, 3  
add rax, rbx  
sub rbx, 2  
mov rcx, rax  
sub rcx, rbx
```

질문:

- 각 명령 후 RAX, RBX, RCX의 값은?
- 최종 결과: RAX=8, RBX=1, RCX=7

## 퀴즈 2: 플래그와 분기

```
mov rax, 8  
mov rbx, 3  
cmp rax, rbx  
je 0x404000  
  
mov rcx, 3  
cmp rbx, rcx  
je 0x404040
```

질문:

- 첫 번째 cmp 후: SF=0, ZF=0 ( $8>3$ )
- 첫 번째 je: 거짓 (실행 안 됨)
- 두 번째 cmp 후: SF=0, ZF=1 ( $3=3$ )
- 두 번째 je: 참 (점프 실행)

# System V AMD64 호출 규약

리눅스 64비트 환경에서 함수를 호출할 때, 인자(Argument)와 리턴값을 효율적으로 전달하기 위한 표준 규칙이다. 이 규약은 함수 간의 데이터 교환 방식을 정의하여 프로그램의 일관성과 호환성을 보장한다.

## 함수 인자 전달 규칙

함수 인자는 특정 레지스터를 통해 전달되며, 레지스터가 부족할 경우 스택을 사용한다.

순서	인자	할당 레지스터 / 위치
1번째	첫 번째 인자	RDI
2번째	두 번째 인자	RSI
3번째	세 번째 인자	RDX
4번째	네 번째 인자	RCX
5번째	다섯 번째 인자	R8
6번째	여섯 번째 인자	R9
7번째~	나머지 인자	스택(Stack)에 push

즉, 함수의 인자가 많을수록 앞쪽 6개까지는 레지스터를 통해 전달되며, 그 이후의 인자들은 스택에 순서대로 쌓이게 된다.

# 함수 리턴값 전달

함수가 실행을 마치고 호출자(Caller)에게 결과값을 돌려줄 때, 이 리턴값은 특정 레지스터를 통해 전달된다.

System V AMD64 규약에서는 주로 RAX 레지스터를 사용한다.

## 리턴값 할당 레지스터

리턴값 타입	할당 레지스터
정수 / 포인터	RAX

함수가 어떤 값을 반환하든, 최종적으로는 RAX에 해당 값이 저장되어 호출자에게 전달된다.

## 실전 연습 퀴즈: 리턴값 추적

```
int add(int x, int y) {  
    return x + y;  
}  
  
// 다음 어셈블리 코드 실행 후 RAX의 값은?  
mov rax, 2  
mov rsi, 3  
mov rdi, rax // RDI = 2  
call add // add(2, 3) 실행, RAX = 5  
mov rcx, rax // RCX = 5
```

### 질문:

위 어셈블리 코드 실행 후,  
mov rcx, rax 시점에서 RAX에 저장되어 있는 값은 무엇인가?

# 함수 호출 예시: puts()

실제로 puts()와 같은 함수를 호출할 때 인자가 어떻게 레지스터를 통해 전달되는지 구체적인 C 코드와 어셈블리 코드를 비교하여 살펴본다.

## C 코드

```
#include <stdio.h>

int main() {
    puts("Hello!");
    return 0;
}
```

puts() 함수는 전달된 문자열을 화면에 출력한다.

## 어셈블리 코드

```
lea rdi, [rip + .LC0]
call puts@PLT
```

.LC0은 컴파일러가 생성한 "Hello!" 문자열의 주소를 나타낸다.

## 인자 전달 상세

인자	전달 레지스터
"Hello!" 문자열 주소	RDI

- ❑ lea (Load Effective Address) 명령은 문자열 "Hello!"가 저장된 메모리 주소(.LC0)를 RDI 레지스터에 로드한다. 이는 System V AMD64 호출 규약에 따라 첫 번째 인자를 RDI로 전달하는 과정이다.

# 함수 호출 예시: scanf()

scanf() 함수를 호출할 때 입력 포맷과 버퍼 주소가 어떻게 레지스터를 통해 전달되는지 살펴본다.

## C 코드

```
char buf[100];  
scanf("%s", buf);
```

사용자로부터 문자열을 입력받아 buf 배열에 저장한다.

## 어셈블리 코드

```
lea rdi, [rip + .LC_fmt]  
lea rsi, [rbp - 0x64]  
call scanf@PLT
```

.LC\_fmt는 컴파일러가 생성한 "%s" 문자열의 주소를 나타낸다. rbp - 0x64는 buf 배열의 시작 주소를 나타낸다.

## 인자 전달 상세

인자	전달 레지스터
"%s" (입력 포맷)	RDI
buf (입력 받을 주소)	RSI

- ❑ lea (Load Effective Address) 명령을 사용하여 "%s" 포맷 문자열의 주소와 buf 배열의 시작 주소를 각각 RDI와 RSI 레지스터에 로드한다. 이는 System V AMD64 호출 규약에 따라 첫 번째 인자는 RDI, 두 번째 인자는 RSI로 전달되어야 하기 때문이다.

# 함수 호출 예시: read()

운영체제 커널이 제공하는 `read()` 함수를 호출할 때, 파일 디스크립터, 버퍼 주소, 읽을 크기 등 인자들이 어떻게 레지스터를 통해 전달되는지 살펴본다.

## C 코드

```
read(0, buf, 0x100);
```

표준 입력(0)으로부터 0x100 바이트를 `buf` 배열에 읽어들인다.

## 어셈블리 코드

```
mov edi, 0          ; fd = 0 (표준 입력)
lea rsi, [rbp - 0x100] ; buf 주소
mov edx, 0x100      ; 읽을 크기
call read@PLT
```

`fd=0`은 표준 입력(`Stdin`)을 의미한다. `rbp - 0x100`은 `buf` 배열이 스택에 할당된 주소를 나타낸다.

## 인자 전달 상세

인자	전달 레지스터
파일 디스크립터 (0)	RDI
<code>buf</code> (버퍼 주소)	RSI
읽을 길이 (0x100)	RDX

- read() 함수 호출 시 System V AMD64 규약에 따라 첫 번째 인자는 RDI, 두 번째는 RSI, 세 번째는 RDX 레지스터를 통해 전달된다.

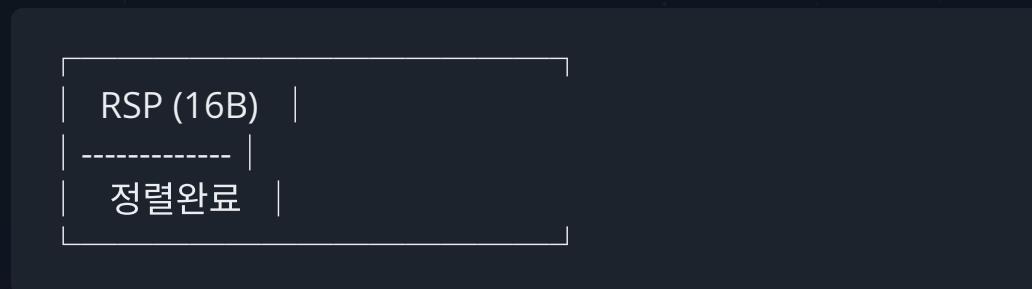
# 스택 정렬(Alignment) - 1

함수를 호출할 때 스택은 항상 16바이트 단위로 정렬되어야 한다. 이는 특정 CPU 명령어들이 최적의 성능을 발휘하기 위해 데이터를 16바이트 경계에 맞춰 접근하기 때문이다.

간단히 말하면 `call` 명령을 실행할 직전의 `RSP % 16 == 0` 이어야 한다.

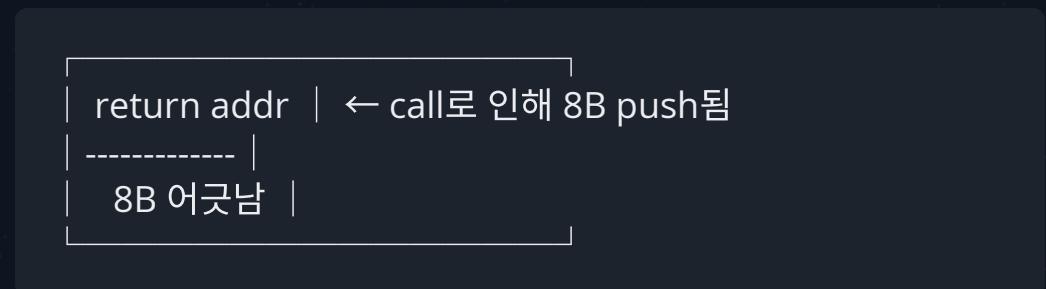
하지만 `call` 명령은 함수의 복귀 주소(8바이트)를 스택에 자동으로 푸시하므로, 스택의 정렬이 깨질 수 있다. 이를 방지하기 위해 호출 전에 스택을 미리 정렬하는 과정이 필요하다.

**Before call:**



함수 호출 전, RSP는 16바이트 경계에 맞춰져 있다.

**After call:**



`call` 명령이 8바이트 복귀 주소를 푸시하여 스택 정렬이 8바이트 어긋나게 된다.

- ▣ **핵심:** 스택 정렬은 함수 호출의 안정성과 성능을 위해 필수적이다. `call` 명령으로 인한 정렬 깨짐을 해결하기 위해 프로그래머는 추가적인 정렬 작업을 수행해야 한다.

# 스택 정렬 (Alignment) - 2

`call` 명령으로 인해 스택 정렬이 8바이트 어긋나는 문제를 해결하기 위해, 함수 호출 전에 RSP 값을 8바이트 감소시켜 16바이트 정렬을 맞춘다.

함수 호출이 끝나면 다시 RSP 값을 8바이트 증가시켜 원래 상태로 복원한다.



## 1. 함수 호출 전

RSP는 16바이트 경계에 정렬되어 있다.

`RSP % 16 == 0`

## 2. 정렬 맞추기

`sub rsp, 8` 명령으로 RSP를 8바이트 감소시킨다.

`sub rsp, 8`



## 3. 함수 호출

`call func` 실행 시 8바이트 복귀 주소가 푸시되어 RSP가 다시 16바이트 정렬된다.

`call func`

## 4. 함수 복귀 후

`ret` 명령으로 복귀 주소 팝, 이후 `add rsp, 8`로 RSP를 복원한다.

`add rsp, 8`

- ▣ **핵심:** 이러한 스택 정렬 과정은 `printf`와 같은 가변 인자 함수나 **SSE** 명령어와 같이 엄격한 16바이트 스택 정렬을 요구하는 함수를 호출할 때 필수적이다. 이를 지키지 않으면 **세그먼트 폴트 (Segment Fault)**가 발생할 수 있다.

# 파이널 퀴즈 : C 코드 vs 어셈블리

puts()와 read() 함수를 사용하는 간단한 C 프로그램이다.

왼쪽의 소스코드와 오른쪽의 컴파일 후 어셈블리어 코드를 비교하면서 각 줄을 해석해보고 스택 정렬 및 함수 호출 규약을 적용하는 방식을 살펴본다.

## C 코드

```
#include <stdio.h>
#include <unistd.h>

int main() {
    char buf[0x100];

    puts("입력하세요:");
    read(0, buf, 0x100);

    puts(buf);
    return 0;
}
```

## 어셈블리 코드 (gcc -O0 기준)

```
main:
    push rbp
    mov rbp, rsp
    sub rsp, 0x110

    lea rdi, [rip + .LC0]
    call puts@PLT

    mov edi, 0
    lea rsi, [rbp - 0x100]
    mov edx, 0x100
    call read@PLT

    lea rdi, [rbp - 0x100]
    call puts@PLT

    mov eax, 0
    leave
    ret
```

- 이 예제는 System V AMD64 호출 규약에 따라 함수 인자가 레지스터(RDI, RSI, RDX)를 통해 전달되는 방식과, read()와 같이 스택 정렬이 필요한 함수의 처리 과정을 명확하게 보여준다. 이를 지키지 않으면 세그먼트 폴트 (Segment Fault)가 발생할 수 있다.

# 정답

앞서 살펴본 C 코드 예제를 기반으로, 각 C 소스 코드 라인이 어셈블리어로 어떻게 번역되고 System V AMD64 호출 규약에 따라 인자가 어떻게 전달되는지 상세하게 비교한다.

puts("입력하세요:");

```
lea rdi, [rip + .LC0]
call puts@PLT
```

puts의 첫 번째 인자("입력하세요:" 문자열 주소)가 RDI 레지스터로 전달된다.

read(0, buf, 0x100);

```
mov edi,0
lea rsi,[rbp-0x100]
mov edx,0x100
call read@PLT
```

read의 인자들 (파일 디스크립터, 버퍼 주소, 읽을 크기)이 각각 RDI, RSI, RDX 레지스터로 전달된다.

puts(buf);

```
lea rdi,[rbp-0x100]
call puts@PLT
```

puts의 인자(입력받은 buf의 주소)가 RDI 레지스터로 전달된다.

return 0;

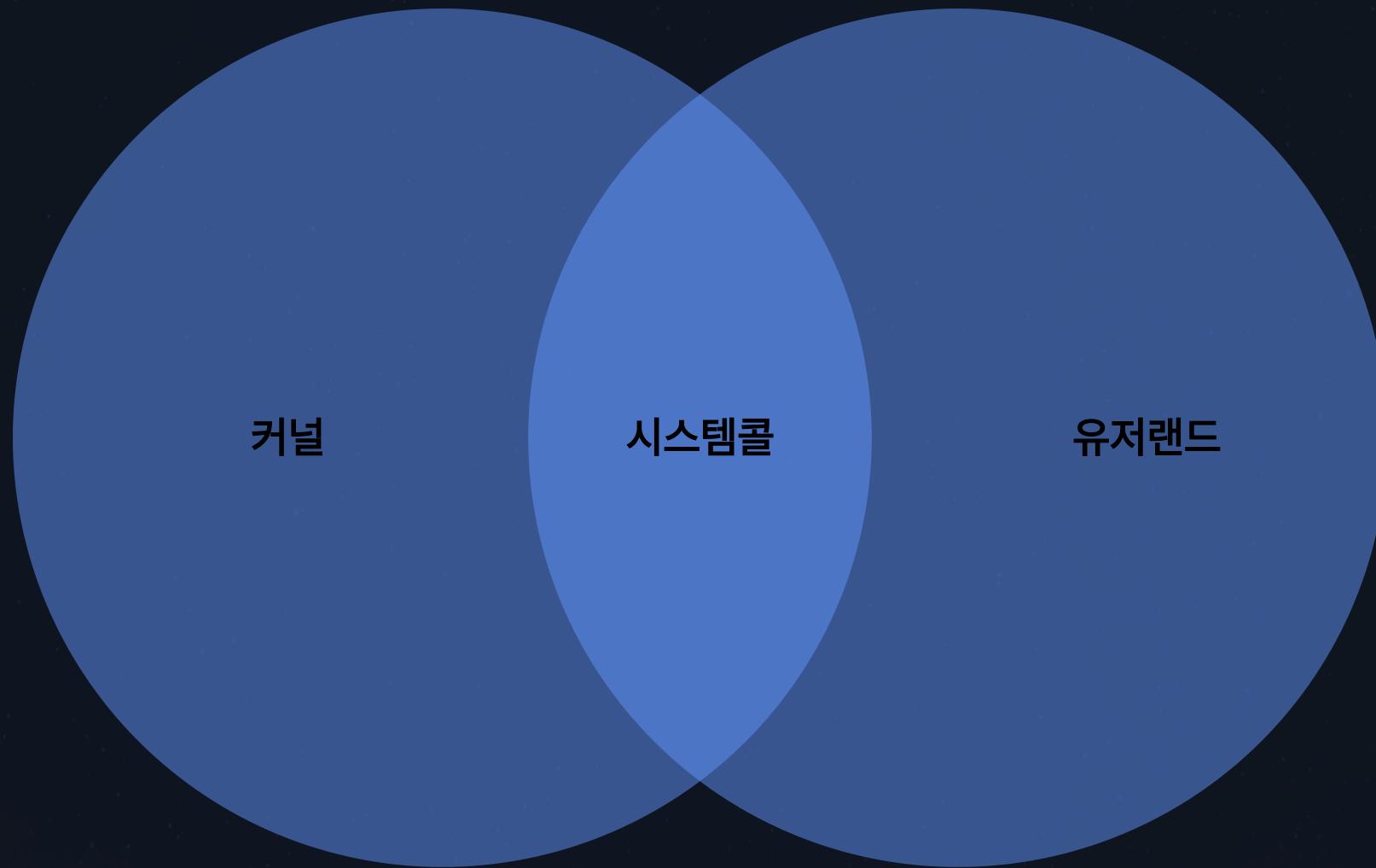
```
mov eax,0
leave
ret
```

함수의 리턴값 0이 EAX(RAX의 하위 32비트) 레지스터로 전달된다.

- 바이너리 분석에 있어서 어셈블리어 해석 능력은 매우 중요하기 때문에, 초반에 어셈블리어에 빠르게 익숙해지기 위해서는 gdb를 통해 수시로 실행흐름을 해석해보는 습관을 기르는 것이 중요하다는 게 필자의 생각이다. (복잡하지 않은 문제들은 gdb만으로 풀어본다)

# 메모리 구조

프로세스 메모리는 크게 커널 영역과 유저랜드 영역으로 나뉘고, 각기 다른 역할과 접근 권한을 가진다.



## 커널 (Kernel)

- 하드웨어·프로세스·권한 제어를 담당한다.
- 운영체제 내부에 존재하며 유저 프로세스와 분리된다.
- 높은 접근 권한을 가진다.

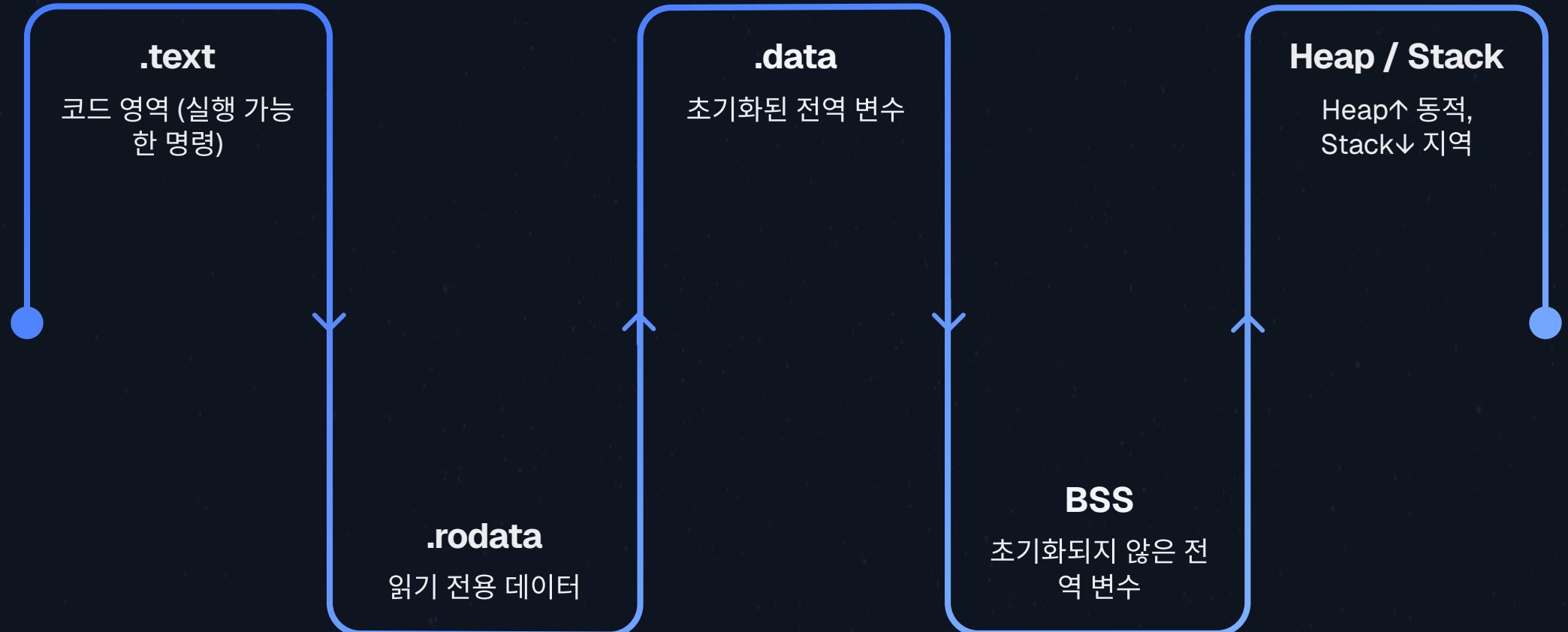
## 유저랜드 (Userland)

- 우리가 실행하는 프로그램이 동작하는 공간이다.
- 권한이 제한되어 있으며 안전장치가 존재한다.
- 직접적인 하드웨어 접근이 불가하다.

핵심은 유저 프로세스는 커널 자원에 바로 접근할 수 없으며, 반드시 시스템콜(System Call)을 통해서만 접근이 가능하다는 점이다.

# 메모리 구조 상세

프로세스의 가상 메모리 공간은 효율적인 관리와 보안을 위해 여러 구역으로 나뉘어 사용된다. 각 구역은 특정 용도의 데이터를 저장하며, 주소 공간은 대체로 낮은 주소부터 높은 주소로 증가하는 방식으로 구성된다. 아래 그림은 일반적인 리눅스 기반 시스템의 메모리 레이아웃을 나타낸다.



## Stack (스택)

지역 변수, 함수 호출 시 복귀 주소 및 인자, 스택 프레임 등이 저장된다. **높은 주소부터 낮은 주소로** 동적으로 할당되며 LIFO(Last-In, First-Out) 구조를 가진다.

## Heap (힙)

프로그램 실행 중에 `malloc()` 또는 `new`와 같은 함수로 동적으로 할당되는 메모리 영역이다. **낮은 주소에서 높은 주소로** 확장된다.

## BSS

초기화되지 않은 전역 변수나 정적 변수가 저장되는 영역이다. 프로그램 시작 시 0으로 초기화된다.

## Data

초기화된 전역 변수나 정적 변수가 저장되는 영역이다. 프로그램 시작 시 지정된 값으로 초기화된다.

## .rodata

읽기 전용 데이터가 저장되는 영역이다. 문자열 리터럴, 상수 등 프로그램 실행 중 변경되지 않는 데이터가 여기에 포함된다.

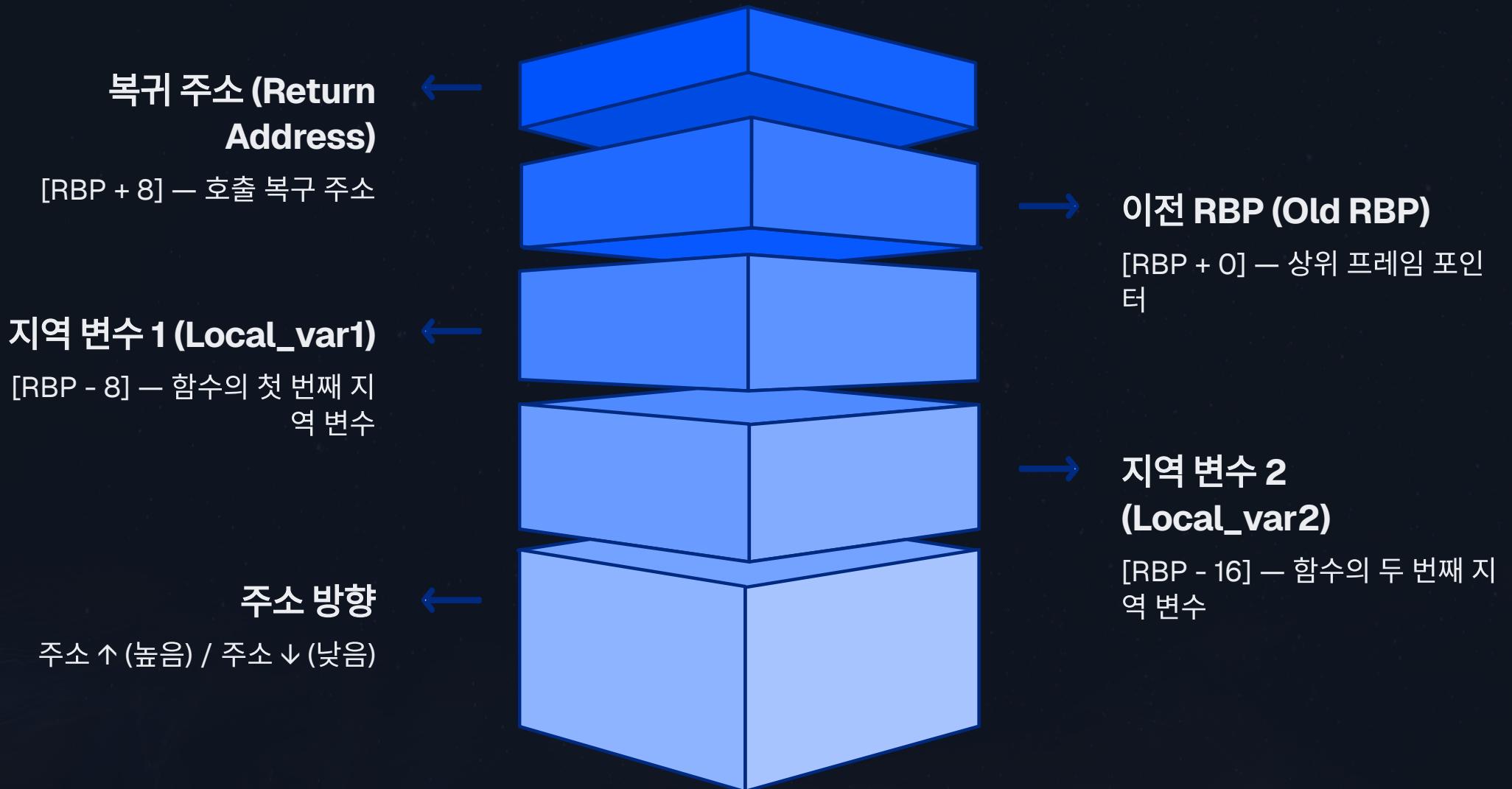
## .text

프로그램의 실행 가능한 기계어 코드가 저장되는 영역이다. 읽기 전용으로 설정되어 코드의 무단 변경을 방지한다.

- 이러한 메모리 구조는 바이너리를 분석할 때 기본으로 알아야 할 요소 중 하나이다.

# 스택 프레임

각 함수는 고유한 스택 프레임을 가지며, 이는 함수가 호출될 때 스택에 할당되는 메모리 영역이다. 스택 프레임은 지역 변수, 저장된 레지스터 값, 그리고 함수가 리턴될 때 실행을 재개할 복귀 주소 등 함수 실행에 필요한 모든 정보를 저장한다.



스택 프레임 구조를 이해하는 것은 함수의 실행 흐름과 지역 변수가 어떻게 관리되는지 파악하는 데 필수적이다. 특히, 복귀 주소(Return Address)가 저장된 위치는 매우 중요한데, 만약 이 값이 조작된다면 프로그램의 제어 흐름이 공격자가 원하는 방향으로 바뀔 수 있기 때문이다.

# 지역 변수 할당 예시

함수 내에서 선언된 지역 변수는 스택 프레임 내에 할당된다. 다음 예시는 `char name[32]` 배열이 스택에 어떻게 공간을 확보하는지 C 코드와 어셈블리 코드를 통해 자세히 살펴본다.

## C 코드

```
int main() {
    char name[32];
    read(0, name, 32);
    return 0;
}
```

함수 내부에 32바이트 크기의 `name` 배열이 지역 변수로 선언되었다.

## 어셈블리 코드

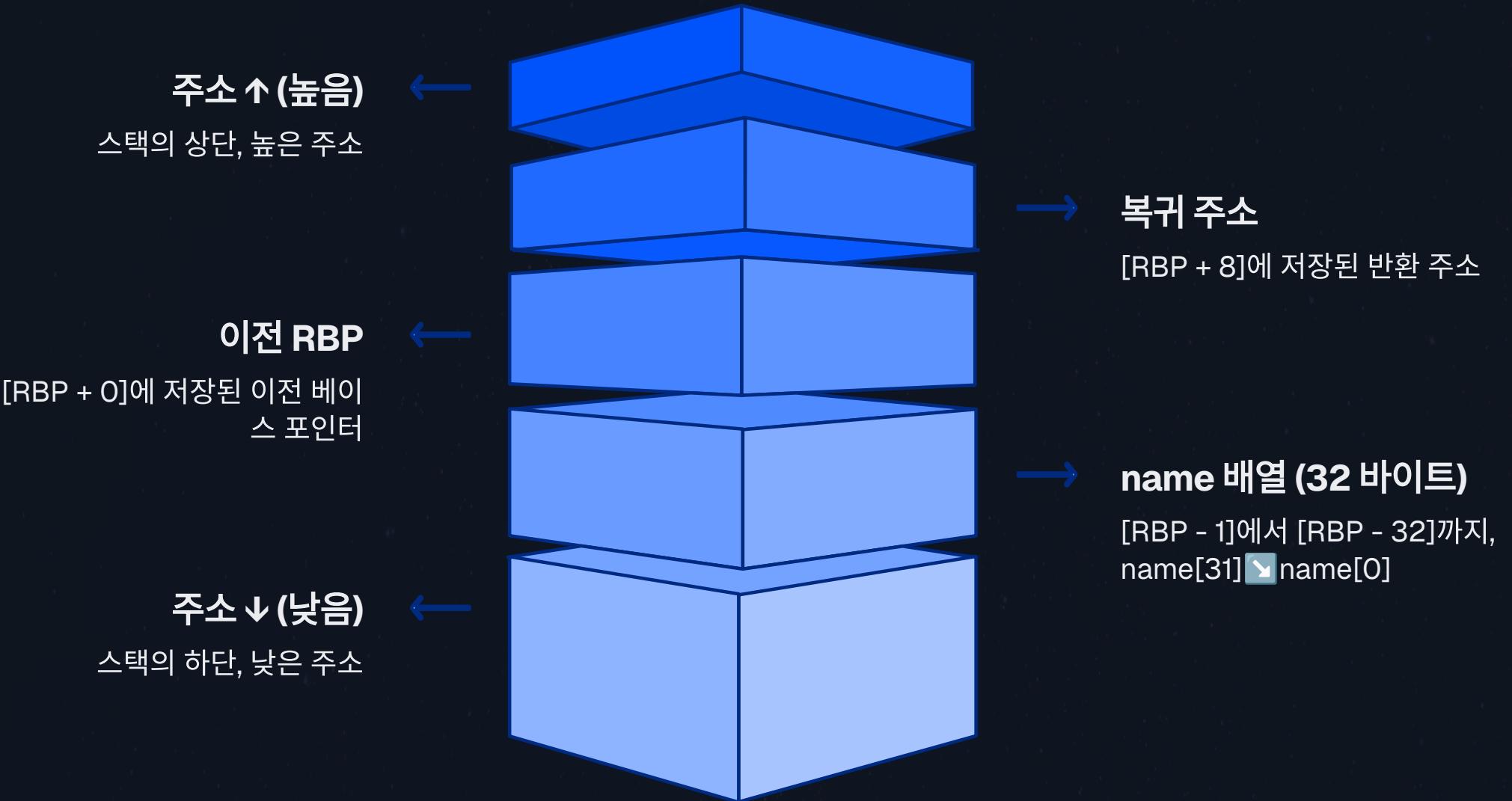
```
main:
    push rbp          ; (1) 이전 RBP 저장
    mov  rbp, rsp     ; (2) 새 스택 프레임의 기준점 설정
    (RBP = RSP)
    sub  rsp, 0x20    ; (3) 지역 변수 name[32]를 위한 32바이트 공간 확보
    ; ... read() 호출 관련 코드 ...
    leave            ; RBP를 RSP로, pop RBP
    ret
```

`sub rsp, 0x20` 명령을 통해 `name` 배열을 위한 32바이트(0x20) 공간이 스택에 할당된다. 이 공간은 `RBP`를 기준으로 낮은 주소에 위치한다.

- 지역 변수는 `RBP`를 기준으로 낮은 주소(음수 오프셋)에 할당되며, `sub rsp, [크기]` 명령으로 스택 포인터(`RSP`)를 감소시켜 공간을 확보한다. 배열과 같이 여러 바이트를 차지하는 변수는 연속된 메모리 공간을 사용한다.

# 스택 프레임 구조 상세 (char name[32] 예시)

char name[32]와 같이 지역 변수가 선언될 때, 스택 프레임 내에서 해당 변수가 어떻게 할당되는지 그 상세 구조를 이해하는 것은 매우 중요하다. 아래 그림은 함수 호출 시 스택에 쌓이는 복귀 주소, 이전 RBP 값, 그리고 지역 변수 name[32]가 차지하는 공간을 수직적으로 보여준다.



## RBP (Base Pointer) 기준

- [RBP + 8]: 함수가 반환될 주소 (Return Address)
- [RBP + 0]: 호출한 함수의 RBP 값 (Old RBP)

## 지역 변수 name[32]

- [RBP - 1]: name 배열의 가장 높은 주소 (name[31])
- [RBP - 32]: name 배열의 가장 낮은 주소 (name[0])

□ 지역 변수는 RBP를 기준으로 낮은 주소(음수 오프셋)에 할당되며, 배열은 연속된 메모리 공간을 차지한다. 버퍼 오버플로우와 같은 취약점은 이러한 배열의 경계를 넘어 데이터를 조작할 때 발생할 수 있다.

이외에도 함수 내에 추가 지역 변수가 선언되면, 이들은 RBP를 기준으로 메모리 정렬 규칙에 따라 순차적으로 할당된다. 예를 들어 int형 변수나 다른 배열이 추가되면 적절한 바이트 경계에 맞춰 스택 프레임 내에 배치된다.

# ORW (open → read → write)

`open()`, `read()`, `write()`는 운영체제에서 파일을 다루는 데 사용되는 핵심 시스템 호출 함수들이다. 이 세 함수는 파일 I/O의 가장 기본적인 패턴을 구성하며, 시스템 프로그래밍과 익스플로잇 개발에서 자주 활용된다.



**open()**

파일 또는 디바이스에 대한 파일 디스크립터를 얻는다.



**read()**

열린 파일 디스크립터에서 지정된 크기 만큼 데이터를 읽는다.



**write()**

열린 파일 디스크립터로 지정된 크기만큼 데이터를 쓴다.

이 ORW 패턴을 이해하는 것은 파일 접근 권한, 데이터 처리, 그리고 저수준 I/O 제어 메커니즘을 파악하는 데 필수적이다.

# ORW 기본

open(), read(), write() 시스템 호출을 깊이 이해하기 위해 필요한 핵심 개념들을 먼저 살펴보자.

## 파일 디스크립터 (fd)

커널이 파일, 소켓, 터미널 등을 식별하기 위해 부여하는 작은 정수. (0=stdin, 1=stdout, 2=stderr는 미리 예약됨)

## 권한/권한 상승

프로그램이 파일을 열기 위해서는 해당 파일에 대한 읽기/쓰기 권한이 필요하다. CTF에서는 setuid 같은 특수 권한으로 플래그 파일을 읽는 경우가 많다.

## 절대경로 vs 상대경로

"/flag.txt"는 루트 디렉토리 기준 절대경로이며, "./flag.txt"와 같은 상대경로는 현재 작업 디렉토리에 따라 경로가 달라진다.

이제 각 시스템 호출의 역할과 사용법을 상세히 알아보자.



### open() — 파일 열기

- 역할:** 운영체제에게 특정 파일을 열어 달라고 요청.
- 반환값:** 성공 시 음이 아닌 파일 디스크립터(예: 3, 4)를, 실패 시 -1을 반환.
- 예시:** `int fd = open("/flag.txt", O_RDONLY);` (읽기 전용으로 열기)
- 의미:** 반환된 fd는 이후 read(), write()에서 해당 파일을 지정하는 식별자로 사용됨.



### read() — 파일에서 바이트 읽기

- 역할:** `read(fd, buf, n)`은 fd에서 최대 n바이트를 읽어 buf에 저장하고, 읽은 바이트 수를 반환.
- 반환값:** 읽은 바이트 수 (0이면 파일의 끝(EOF)), 실패 시 -1을 반환.
- 예시:** `char buf[128]; ssize_t r = read(fd, buf, sizeof(buf));`



### write() — 출력(또는 다른 fd로 쓰기)

- 역할:** `write(fd_out, buf, n)`은 buf의 데이터를 fd\_out에 n바이트만큼 출력.
- 일반적 사용:** fd\_out이 1이면 표준출력(stdout)으로 데이터를 보내 터미널에 출력됩니다.
- 반환값:** 실제로 쓴 바이트 수를 반환.
- 예시:** `write(1, buf, r);` (읽은 r 바이트 데이터를 터미널에 출력)

# ORW 요약: 플래그 읽기

`open() → read() → write()`는 시스템 프로그래밍에서 파일 I/O의 가장 기본적인 패턴이다. 이 패턴은 특히 CTF(Capture The Flag)에서 플래그 파일을 읽고 출력하는 데 핵심적으로 사용된다.



## Open

`open("/flag.txt", O_RDONLY);`  
파일을 열고 파일 디스크립터를 얻는다.



## Read

`read(fd, buf, sizeof(buf));`  
열린 파일에서 지정된 크기만큼 데이터를 버퍼로 읽는다.



## Write

`write(1, buf, n);`  
읽은 데이터를 표준 출력(`stdout`)으로 내보내 화면에 출력한다.



## Close

`close(fd);`  
사용했던 파일 디스크립터를 닫아 자원을 해제한다.

이 4단계가 플래그를 읽고 보여주는 핵심 과정이다. CTF 바이너리에서 이런 코드를 찾아내거나 직접 삽입하여 권한이 있는 상태에서 실행하면, 플래그가 화면에 출력될 것이다.

## 간단한 ORW C 코드 예시 (학습용)

```
#include <fcntl.h>
#include <unistd.h>

int main(void) {
    int fd = open("/flag.txt", O_RDONLY);
    if (fd < 0) return 1; // 파일 열기 실패 시 종료
    char buf[256];
    ssize_t n = read(fd, buf, sizeof(buf));
    if (n > 0) write(1, buf, n); // 읽은 데이터가 있다면 stdout으로 출력
    close(fd);
    return 0;
}
```

이 코드는 `/flag.txt` 파일을 열고, 그 내용을 읽어 `buf`에 저장한 다음, 다시 `stdout`으로 출력하고 파일을 닫는 정상적인 과정을 보여준다.

# system() vs execve() – 쉘을 만드는 방법들

CTF에서 "쉘 따기"는 인터랙티브하게 명령을 실행할 수 있는 환경을 만드는 것을 의미한다. 이를 구현하는 대표적인 두 가지 방식인 system()과 execve()에 대해 살펴보자.

## system() (간단한 래퍼)

선언: int system(const char \*command);

- 동작: 내부적으로 새로운 프로세스를 생성하고, 해당 프로세스에서 `/bin/sh -c "command"`를 실행한다.
- 특징: 사용법이 간단하지만 입출력 제어에 추가 작업이 필요하다. 입력 문자열에 대한 검증이 없을 경우 **명령 주입(Command Injection)**의 위험이 있어 보안에 취약할 수 있다.

## execve() (저수준 직접 실행)

선언: int execve(const char \*pathname, char \*const argv[], char \*const envp[]);

- 동작: 현재 프로세스의 코드를 새로운 프로그램으로 완전히 덮어씌운다. 일반적으로 fork()와 함께 사용해 새 쉘 프로세스를 생성한다.
- 예시: `char *argv[] = {"./bin/sh", NULL}; execve("./bin/sh", argv, NULL);`
- 특징: 세밀한 제어가 가능하며, 인터랙티브 쉘을 만들 때 `/bin/sh`를 직접 실행함으로써 해당 프로세스를 쉘로 전환한다.

요약하면, system()은 편리하지만 제한적인 반면, execve()는 저수준 제어를 통해 더 강력한 기능을 제공한다. Pwn 문제에서는 취약점을 이용해 이 두 함수를 호출함으로써 시스템 명령 실행 권한을 얻는 것이 핵심 목표가 될 수 있다.

# 쉘 따기

포너블(Pwnable) 문제 해결에서 "쉘 따기"는 핵심 목표이며, `system()` 함수 호출은 이를 달성하는 중요한 경로가 될 수 있다. 바이너리 분석 시 `system()` 함수를 발견하면 이를 통해 플래그를 얻을 수 있는 기회로 작용한다.



## 1. 직접 플래그 파일 읽기

바이너리 자체에 `flag.txt` 파일을 `open() -> read() -> write()` 하는 로직이 존재하여, 이 부분을 트리거하면 플래그를 직접 얻을 수 있다.



## 2. `system("/bin/sh")`으로 쉘 획득

`system("/bin/sh")`과 같은 함수를 호출하여 대화형 쉘을 얻은 후, 해당 쉘에서 직접 `flag.txt`를 읽어야 하는 상황이다.

## `system("/bin/sh")` 호출 예시

다음은 `system("/bin/sh")`을 호출하는 간단한 C 코드와 그에 해당하는 어셈블리 코드를 통해 `system()` 함수의 호출 과정을 이해하는 데 도움을 준다.

### C 코드

```
#include <stdio.h>

int main(void) {
    system("/bin/sh");
    return 0;
}
```

`system("/bin/sh")`을 호출하여 쉘을 실행하는 소스코드이다.

아래는 위 소스코드의 어셈블리 코드로 각 줄을 해석해보며 눈에 잘 익혀놓자.

### 어셈블리 코드 (pwndbg> disas main)

```
0x0000000000401136 <+0>: endbr64
0x000000000040113a <+4>: push rbp
0x000000000040113b <+5>: mov rbp,rs
0x000000000040113e <+8>: lea rax,[rip+0xebf] ; # 0x402004
0x0000000000401145 <+15>: mov rdi,rax
0x0000000000401148 <+18>: mov eax,0x0
0x000000000040114d <+23>: call 0x401040 <system@plt>;
0x0000000000401152 <+28>: mov eax,0x0
0x0000000000401157 <+33>: pop rbp
0x0000000000401158 <+34>: ret
```

- 바이너리를 분석할 때 `system()` 함수 호출이 보이는 부분은 매우 중요하게 살펴봐야 한다. 이는 해당 함수를 제어할 수 있다면 시스템 명령 실행 권한을 얻을 수 있음을 의미한다.