StackFrame2

- Low Level -

안태진(taejin@codecure.smuc.ac.kr)

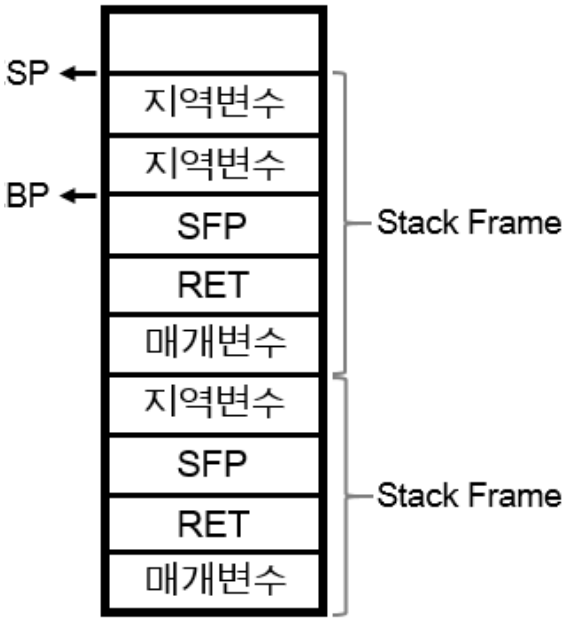
상명대학교 보안동아리 CodeCure

목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

StackFrame

- 스택 프레임(Stack Frame)
  - 함수 호출 시 사용되는 스택 공간
  - SP(Stack Pointer)
    - 스택 프레임의 가장 최상부, 현재위치
  - BP(Base Pointer)
    - 스택 프레임이 시작된 주소
  - RET
    - 복귀할 함수의 주소
  - SFP(Saved Frame Pointer)
    - EBP 복귀 주소 저장

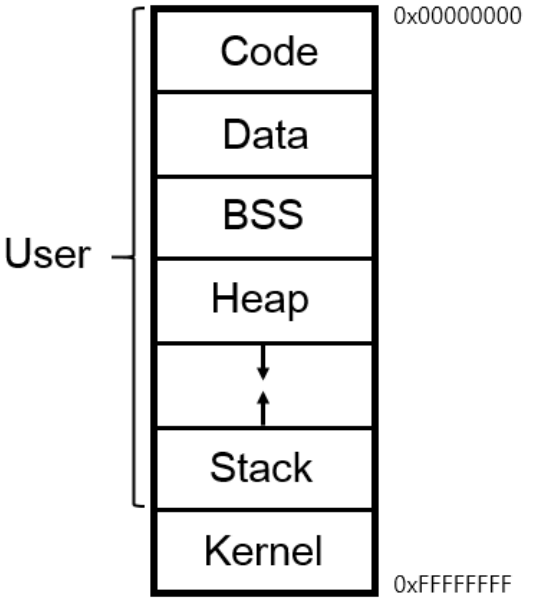


목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

StackFrame

- 메모리 구조 (Memory Structure)
  - Code Segment
    - 프로그래머가 작성한 코드가 저장되는 영역
  - Data Segment
    - 전역 변수, 정적 변수가 저장되는 영역
  - Heap Segment
    - 동적 할당한 공간이 저장되는 영역
  - Stack Segment
    - 함수가 저장되는 영역
    - 높은 주소에서 낮은 주소로 커짐



목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

StackFrame

- Prologue (프롤로그)
  - 함수가 호출자 (Caller)에 의해 가장 먼저 불렸을 때 이전의 함수로 돌아가기위해,현재의 함수의 공간을 이전의 스택과 구분하기 위해 수행하는 과정
  - BP를 스택에 PUSH; BP를 SP의 위치로 옮김
  - Code

```
push    rbp
mov     rbp,rsp
```

StackFrame

- Epilogue (에필로그)
  - 자신을 부른 호출자로 다시 돌아가기 위해 현재의 스택 공간을 청산하고, 호출 되기 전의 상태로 돌아가는 준비를 하는 것
  - SP를 BP의 위치로 옮김; BP에 스택에서 POP; RET
  - Code

```
leave
ret
```

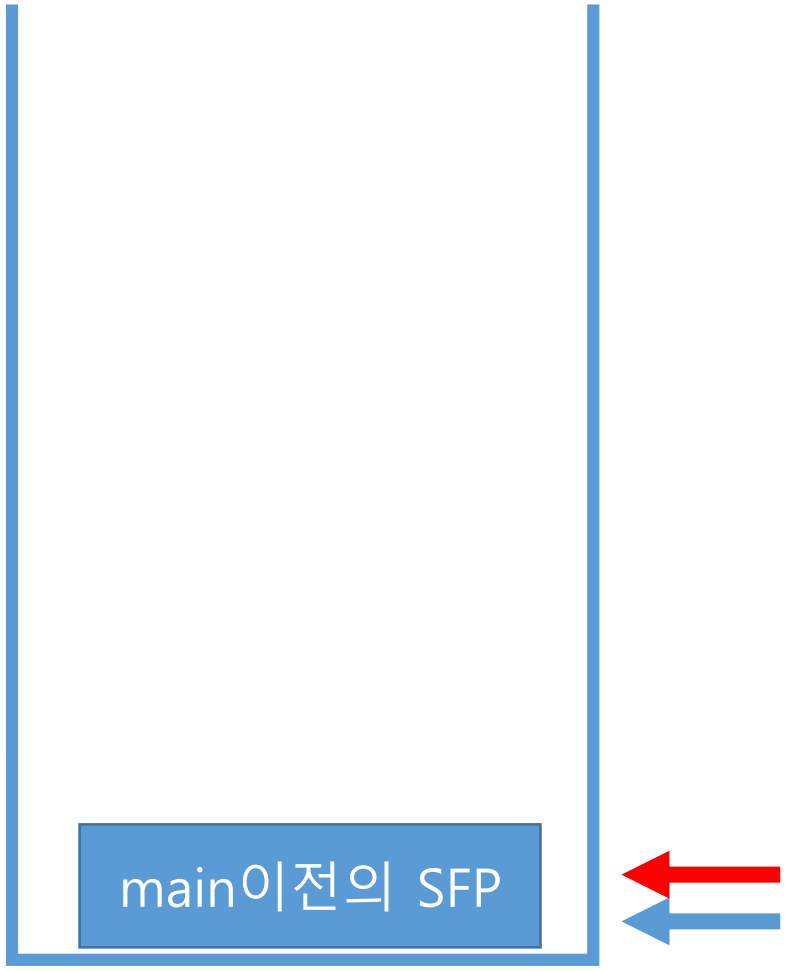
    - leave => mov rsp, rbp; pop rbp
    - ret => pop rip

목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

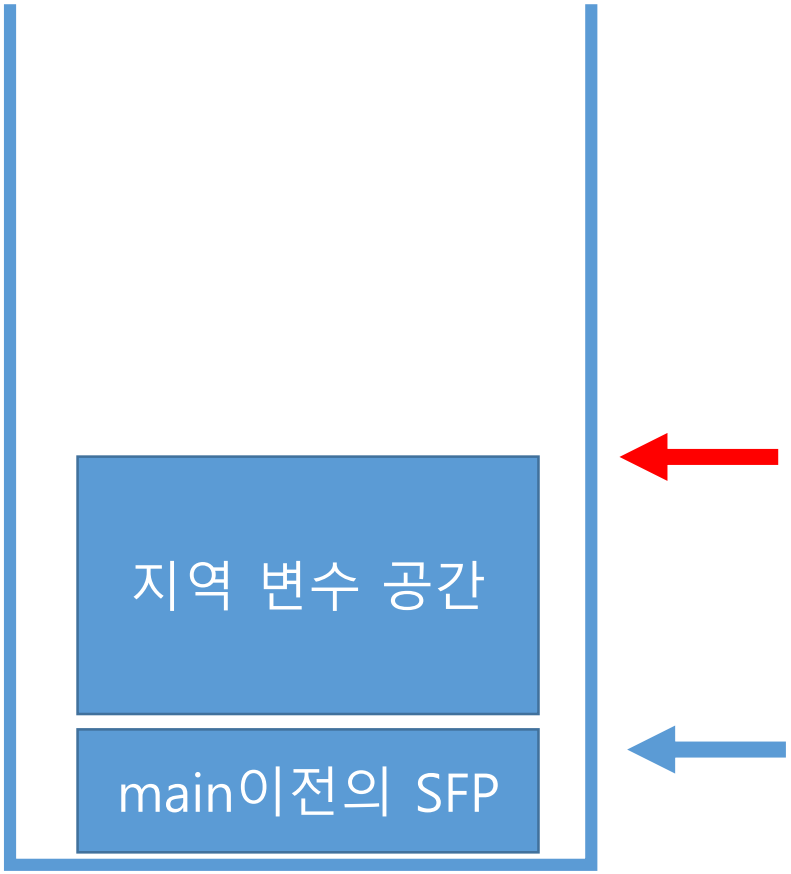
StackFrame

- 동작 과정 (1/9)
  - main 함수 프롤로그 (파랑: BP, 빨강: SP)



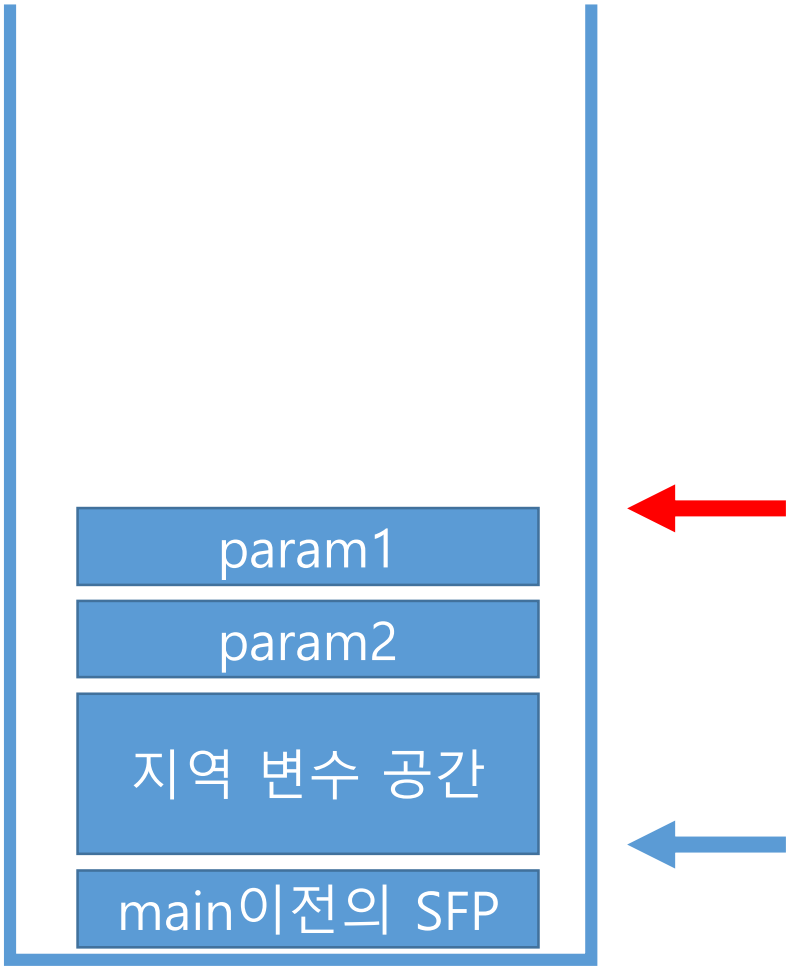
StackFrame

- 동작 과정 (2/9)
  - main 지역 변수 공간 할당
    - BP – (all size of variables)



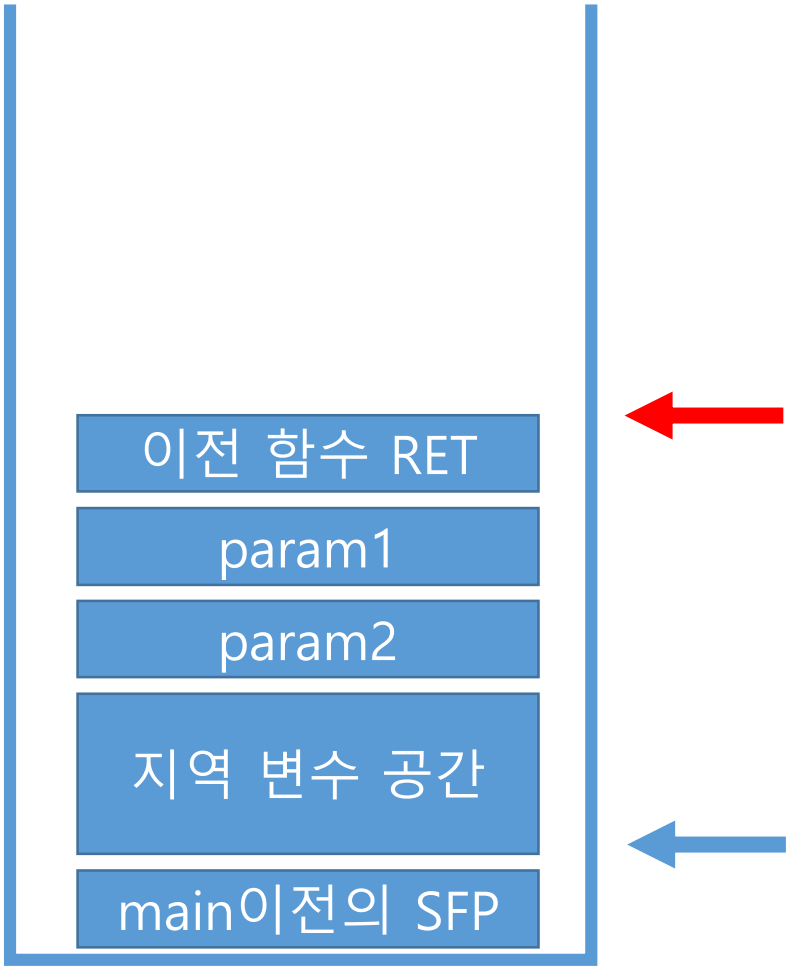
StackFrame

- 동작 과정 (3/9)
  - 매개변수 입력



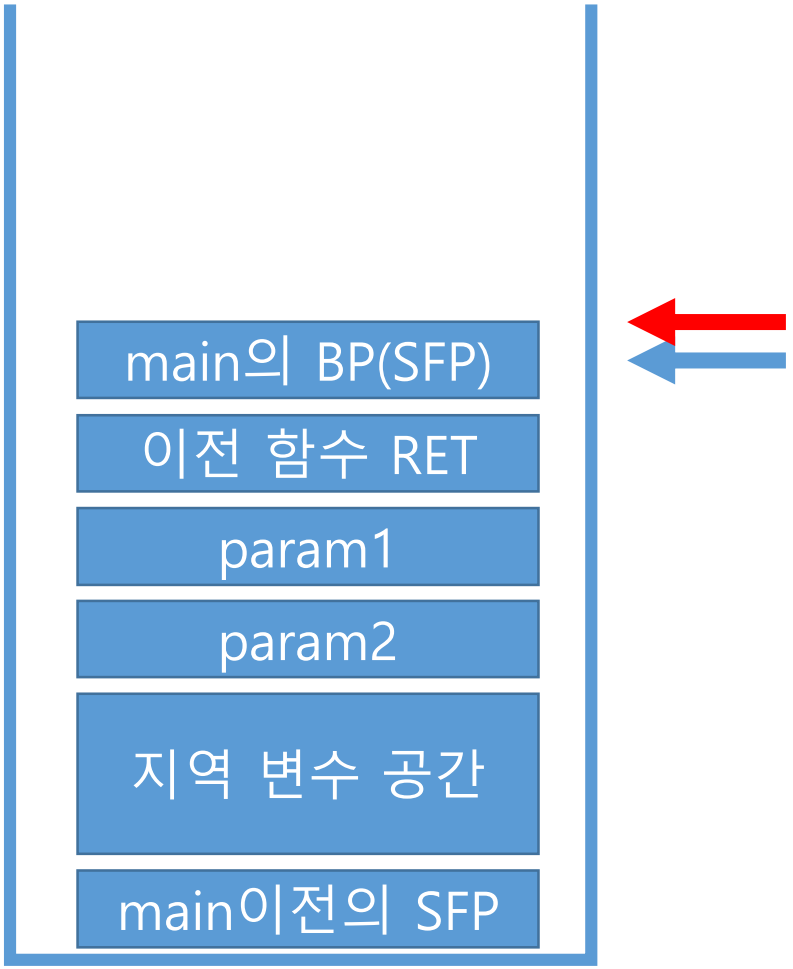
StackFrame

- 동작 과정 (4/9)
  - 함수 호출



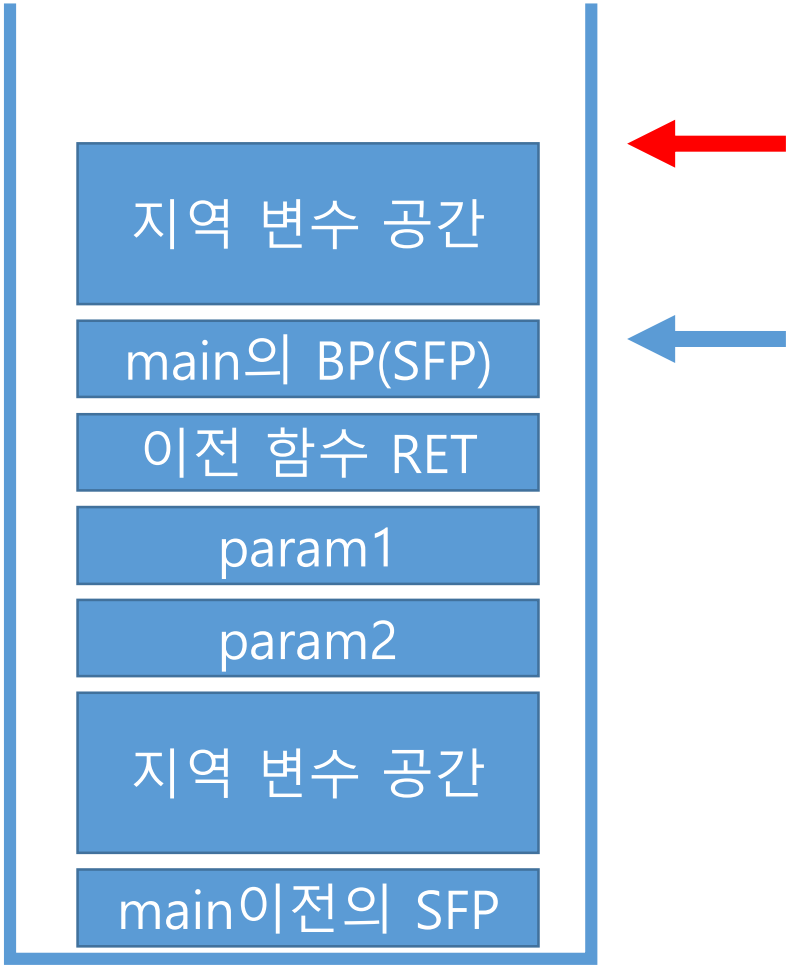
StackFrame

- 동작 과정 (5/9)
  - 함수 프롤로그



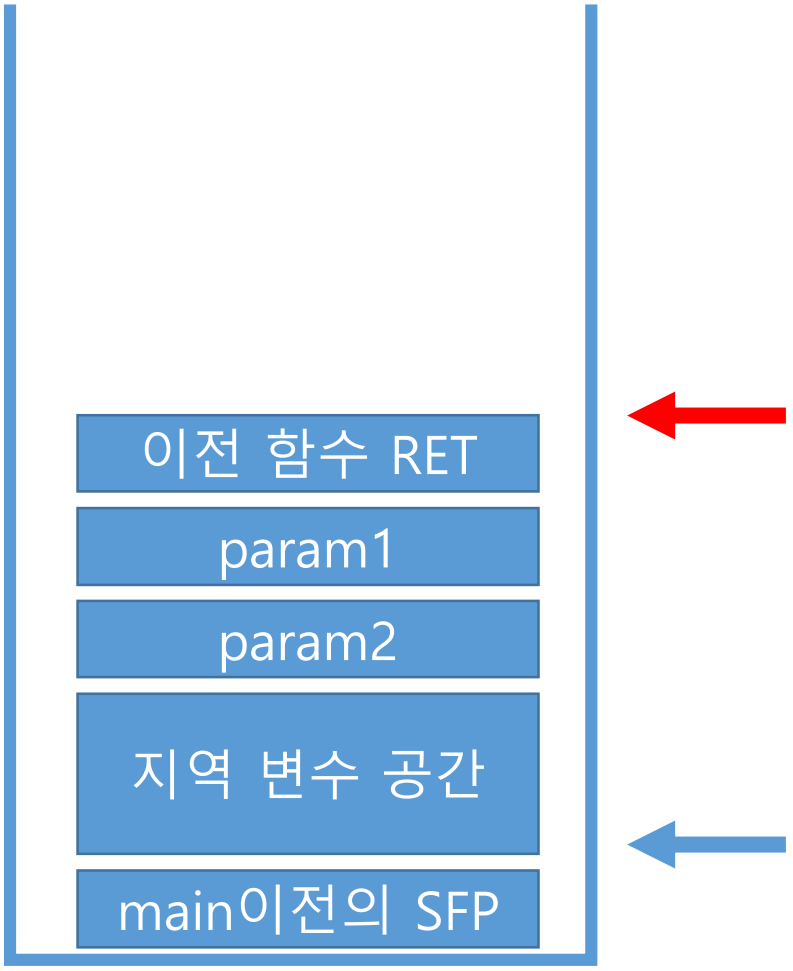
StackFrame

- 동작 과정 (6/9)
  - 함수 지역 변수 공간 할당



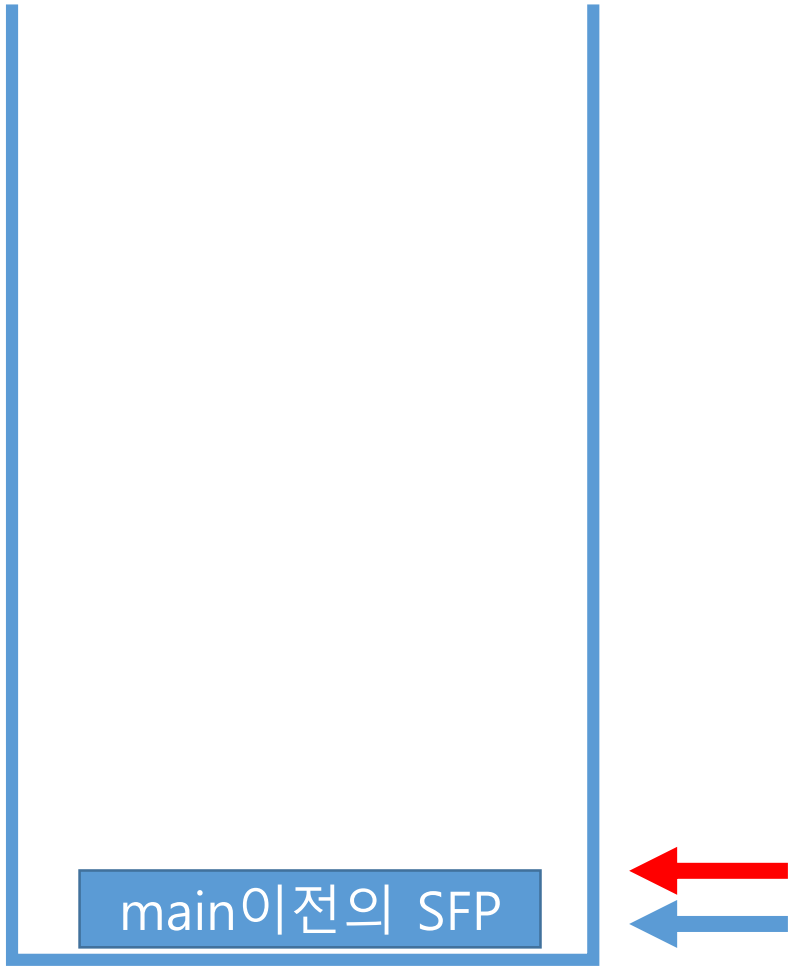
StackFrame

- 동작 과정 (7/9)
  - 함수 에필로그



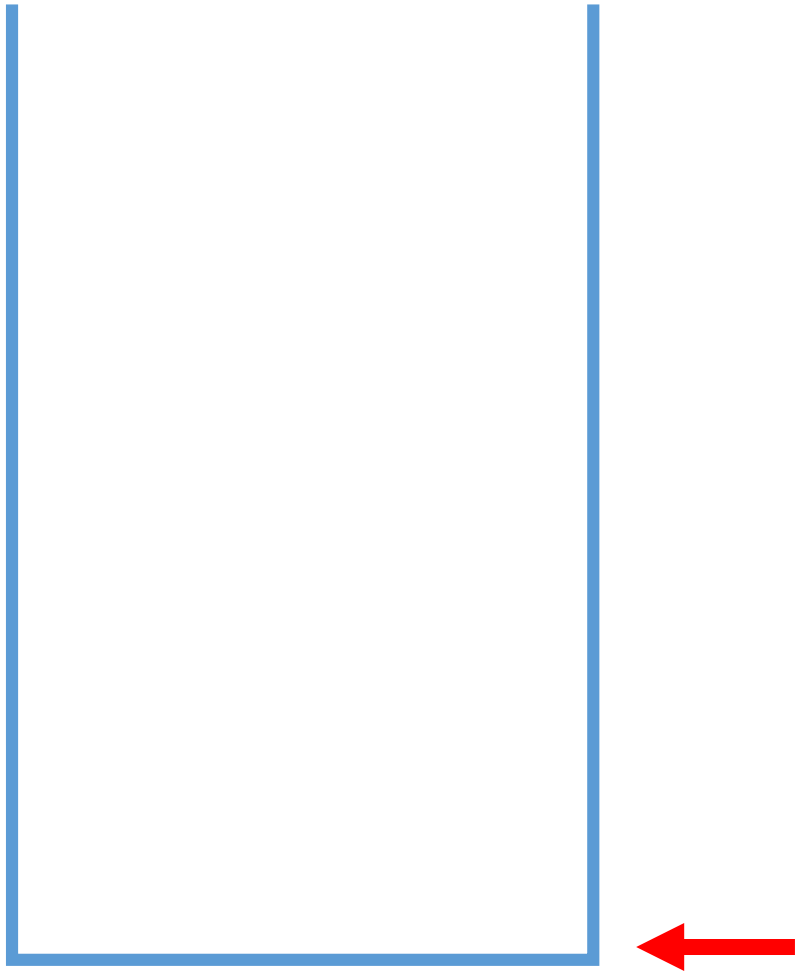
StackFrame

- 동작 과정 (8/9)
  - main 에필로그



StackFrame

- 동작 과정 (9/9)
  - 마지막 상태



목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

ASM Inst.

- Registers (1/2)
  - RSP (Register Stack Pointer)
    - 64bit에서의 Stack Pointer 즉, 8bytes 데이터
    - ESP (Extend Stack Pointer)는 32bit, 4bytes
  - RBP (Register Base Pointer)
    - 64bit에서의 Base Pointer 즉, 8bytes 데이터
    - EBP (Extend Stack Pointer)는 32bit, 4bytes
  - RIP (Register Instruction Pointer)
    - 64bit에서의 Instruction Pointer
    - 명령어를 가리키는 포인터, 이 포인터가 가리키는 명령어를 실행

ASM Inst.

- Registers (2/2)
  - RAX (Register Accumulator Register)
    - 산술 연산에 사용
    - return 값을 저장할 때도 사용
    - EAX (Extend Accumulator Register)
  - 이 외에 RBX, RCX, RDX 등등도 존재, 하지만 단순한 값을 저장하는 임시 변수라고 생각해도 무방

목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

ASM Inst.

- Instructions (1/3)
  - push [opr1]
    - 스택에 opr1을 넣는 명령어
    - push 이후엔 sp가 운영체제 bit 만큼 -
  - pop [opr1]
    - 스택에서 데이터를 인출하여 opr1에 저장
    - pop 이후에 sp가 운영체제 bit 만큼 +
  - mov [opr1] [opr2]
    - opr1에 opr2의 값을 저장
    - C언어의 opr1 = opr2와 비슷

ASM Inst.

- Instructions (2/3)
  - add, sub [opr1] [opr2]
    - opr1 에서 opr2를 빼거나 더해 opr1에 저장
    - opr1 = opr1 +(-) opr2
  - call [func addr]
    - 함수를 호출하는 명령어
    - call 이후에 IP는 call한 함수의 시작 주소를 가리키며, call 다음 명령어의 주소가 스택에 push 된다
  - leave
    - 함수를 떠나는 명령어
    - Epilogue에 주로 쓰이며 mov rsp, rbp; pop rbp와 같은 일을 하는 명령어

ASM Inst.

- Instructions (3/3)
  - ret
    - 함수의 수행이 모두 종료된 뒤, 호출 함수로 돌아가는 명령어
    - pop rip 와 같은 일을 함
      - pop rip를 쓰지 않는 이유
        - rip를 어셈블리어 수준에서 바꿀 수 있으면 함수의 흐름을 쉽게 변경 가능

        => 또 다른 취약점 발생 가능
  - nop
    - no oprerator
    - 말 그대로 연산자가 없는 것으로, 이 명령어를 만나면 단순히 지나감

목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

CallingConvention

- 호출 규약 (Calling Convention)
  - 함수 호출 시 여러가지 문제 처리에 대한 규약
    - 함수를 호출 할 때 매개변수는 어떻게 저장할 것인가?
    - 함수가 끝난 뒤 Stack은 누가 이전의 상태로 정리하는가?
  - cdecl, stdcall, fastcall, x64 Calling Convention 등등
    - 많이 쓰이는 cdecl, x64를 알아볼 예정

CallingConvention

- cdecl (C declaration) (1/2)
  - C언어에서 만든 호출 규약
  - Parameter (매개변수)
    - 함수 호출 전에 모든 매개변수들을 스택에 push
    - 함수 내에서 매개변수는 BP + (size)하여 접근
  - Stack 정리
    - 함수 호출자가 직접 Stack 정리

CallingConvention

- cdecl (C declaration) (2/2)

```c
/* cdecl_CallingConvention.c */
#include <stdio.h>

int Sum4(int num1, int num2, int num3, int num4) {
    int w = num1, x = num2, y = num3, z = num4;
    int sum = num1 + num2 + num3 + num4;

    return sum;
}

int main(void) {
    int sum = Sum4(10, 20, 30, 40);
    printf("%d\n", sum);

    return 0;
}
```

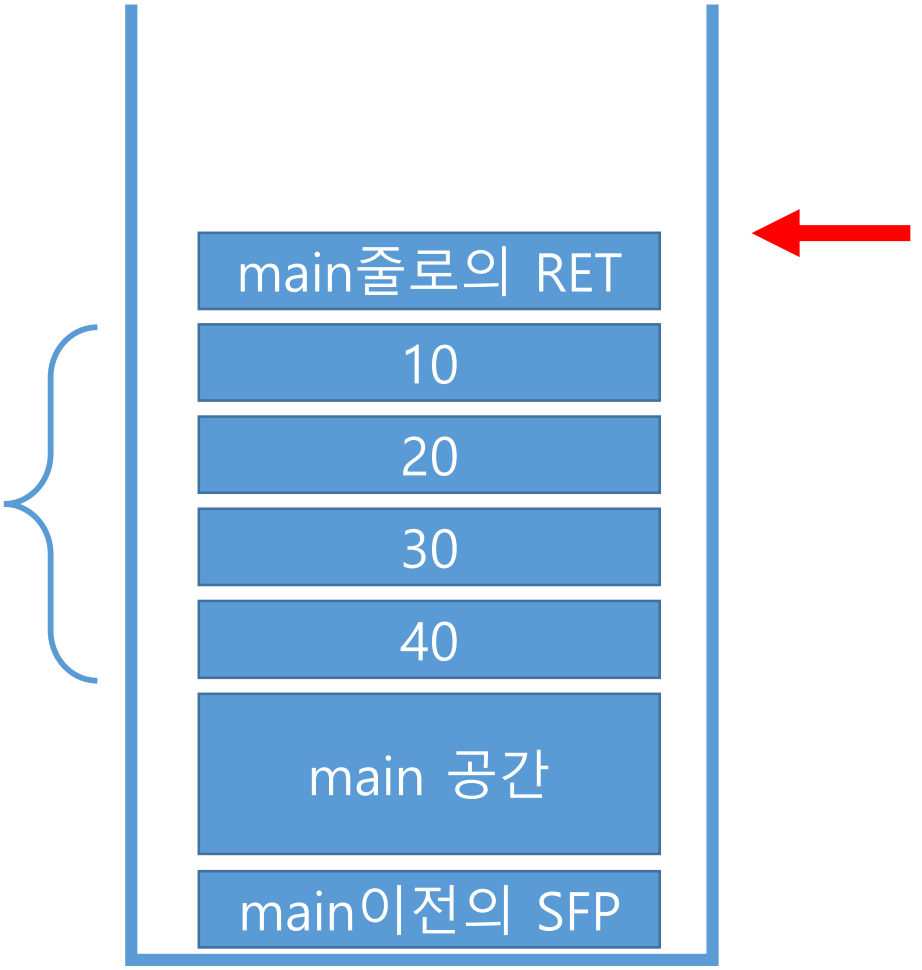
매개변수 push

```
push        28h   40
push        1Eh   30
push        14h   20
push        0Ah   10
call        _Sum4 (0B81221h)
```

Stack 정리

```
call        _Sum4 (0B81221h)
add         esp,10h   16
```

16 = 4 * 4



CallingConvention

- x64 Calling Convention (1/2)
  - 64bit에서 많이 쓰이는 호출 규약
  - Parameter (매개변수)
    - 6개는 레지스터에 저장, 나머지는 스택에 저장
      - edi, esi, edx, ecx, r8d, r9d
      - 스택 저장 데이터는 cdecl과 같은 방식으로 데이터 인출
  - Stack 정리
    - cdecl과 동일

CallingConvention

- x64 Calling Convention (2/2)

매개변수 저장

```
push    0x8
push    0x7
mov     r9d,0x6
mov     r8d,0x5
mov     ecx,0x4
mov     edx,0x3
mov     esi,0x2
mov     edi,0x1
call    0x691 <Sum8>
```

지역 변수에 다시 저장

```
DWORD PTR [rbp-0x14],edi
DWORD PTR [rbp-0x18],esi
DWORD PTR [rbp-0x1c],edx
DWORD PTR [rbp-0x20],ecx
DWORD PTR [rbp-0x24],r8d
DWORD PTR [rbp-0x28],r9d
```

push한 데이터에 접근

```
mov     eax,DWORD PTR [rbp+0x10]
add     DWORD PTR [rbp-0x4],eax
mov     eax,DWORD PTR [rbp+0x18]
```

```c
/* x64CallingConvention.c */
#include <stdio.h>

int Sum8(int num1, int num2, int num3, int num4, int num5, int num6, int num7, int num8);

void main(void) {
    printf("%d\n",
    Sum8(1, 2, 3, 4, 5, 6, 7, 8));
}

int Sum8(int num1, int num2, int num3, int num4, int num5, int num6, int num7, int num8) {
    int sum = 0;

    sum += num1;
    sum += num2;
    sum += num3;
    sum += num4;
    sum += num5;
    sum += num6;
    sum += num7;
    sum += num8;

    return sum;
}
```

목차

- StackFrame
  - Introduction
  - Memory Structure
  - Prologue, Epilogue
  - Process
- ASM Inst. (Intel)
  - Registers
  - Instructions
- Calling Convention
- GDB Inst. (peda)

GDB Inst.

- GDB (The GNU Debugger)
  - Linux에서 지원하는 디버거
  - peda (Python Exploit Development Assistance for GDB)
    - python으로 만든 GDB 향상 프로그램
    - 디버깅하기 쉽도록 알록달록하고 다양한 기능과 정보 지원

```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000000064a <+0>:     push   %rbp
   0x000000000000064b <+1>:     mov    %rsp,%rbp
   0x000000000000064e <+4>:     sub    $0x10,%rsp
   0x0000000000000652 <+8>:     movl   $0xa,-0xc(%rbp)
   0x0000000000000659 <+15>:    movl   $0x14,-0x8(%rbp)
   0x0000000000000660 <+22>:    mov    -0x8(%rbp),%edx
   0x0000000000000663 <+25>:    mov    -0xc(%rbp),%eax
   0x0000000000000666 <+28>:    mov    %edx,%esi
   0x0000000000000668 <+30>:    mov    %eax,%edi
   0x000000000000066a <+32>:    callq  0x68f <Add>
   0x000000000000066f <+37>:    mov    %eax,-0x4(%rbp)
   0x0000000000000672 <+40>:    mov    -0x4(%rbp),%eax
   0x0000000000000675 <+43>:    mov    %eax,%esi
   0x0000000000000677 <+45>:    lea    0xb6(%rip),%rdi        # 0x734
   0x000000000000067e <+52>:    mov    $0x0,%eax
   0x0000000000000683 <+57>:    callq  0x520 <printf@plt>
   0x0000000000000688 <+62>:    mov    $0x0,%eax
   0x000000000000068d <+67>:    leaveq
   0x000000000000068e <+68>:    retq
End of assembler dump.
(gdb) b main
Breakpoint 1 at 0x64e
(gdb) r
Starting program: /home/jin-book/sfp1

Breakpoint 1, 0x000000000800064e in main ()
(gdb)
```

```
gdb-peda$ r
Starting program: /home/jin-book/sfp1
[----------------------------------registers-----------------------------------]
RAX: 0x800064a --> 0x10ec8348e5894855
RBX: 0x0
RCX: 0x80006b0 --> 0x41d7894956415741
RDX: 0x7fffffee358 --> 0x7fffffee57b ("LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...)
RSI: 0x7fffffee348 --> 0x7fffffee567 ("/home/jin-book/sfp1")
RDI: 0x1
RBP: 0x7fffffee260 --> 0x80006b0 --> 0x41d7894956415741
RSP: 0x7fffffee260 --> 0x80006b0 --> 0x41d7894956415741
RIP: 0x800064e --> 0xaf445c710ec8348
R8 : 0x7fffff3ecd80 --> 0x0
R9 : 0x7fffff3ecd80 --> 0x0
R10: 0x2
R11: 0x3
R12: 0x8000540 --> 0x89485ed18949ed31
R13: 0x7fffffee340 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8000645 <frame_dummy+5>:   jmp    0x80005b0 <register_tm_clones>
   0x800064a <main>:    push   rbp
   0x800064b <main+1>:  mov    rbp,rsp
=> 0x800064e <main+4>:  sub    rsp,0x10
   0x8000652 <main+8>:  mov    DWORD PTR [rbp-0xc],0xa
   0x8000659 <main+15>: mov    DWORD PTR [rbp-0x8],0x14
   0x8000660 <main+22>: mov    edx,DWORD PTR [rbp-0x8]
   0x8000663 <main+25>: mov    eax,DWORD PTR [rbp-0xc]
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffee260 --> 0x80006b0 --> 0x41d7894956415741
0008| 0x7fffffee268 --> 0x7ffff021b97 (<__libc_start_main+231>:       mov    edi,eax)
0016| 0x7fffffee270 --> 0x1
0024| 0x7fffffee278 --> 0x7fffffee348 --> 0x7fffffee567 ("/home/jin-book/sfp1")
0032| 0x7fffffee280 --> 0x100008000
0040| 0x7fffffee288 --> 0x800064a --> 0x10ec8348e5894855
0048| 0x7fffffee290 --> 0x0
0056| 0x7fffffee298 --> 0xc57ab5a7fb6dd36e
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x000000000800064e in main ()
```

GDB Inst.

- 기본 명령어
  - pdisas (peda disassembly)
    - 어셈블리 코드를 보여줌 (peda 방식)
  - break(b) [position]
    - 브레이크 포인트를 검
  - print(p) / [format] [variable]
    - 변수의 값을 format 형태로 출력
      - e.g., p/d num1, p/x hexa1
  - x/[num][bytes][system] [variable]
    - [variable] 주소로 들어가 값 출력
      - e.g., x/10bu $rbp

Bytes	System
b -> 1byte (Bytes)	**t** -> 2진법 (Binary System)
h -> 2bytes (Half word)	**o** -> 8진법 (Octal System)
w -> 4bytes (Word)	**x** -> 16진법 (Hexadecimal System)
g -> 8bytes (Giant)	**u** -> 10진법 (Decimal System)
s -> String	

GDB Inst.

- Execute
  - run(r)
    - 프로그램 실행 (Break 포인트까지)
  - continue(c)
    - 프로그램 마저 실행 (다음 Break 포인트까지)
  - step(s)
    - 어셈블리어 한 줄 실행, 만약 함수면 그 함수 안으로 들어감
  - next(n)
    - 어셈블리어 한 줄 실행, 만약 함수면 함수 실행

감사합니다!