

Whols 리버싱 스터디

Mungsul

계획

1주차 (1월 8 - 10일) : Intel x86 어셈블리, 스택, 콜링 컨벤션, 디버거 다뤄보기(IDA 7.0 free, ollydbg)

2주차 (1월 15 - 17일) : PE 포맷, .NET, APK 분석

* 18-2 학기 기존 내용.

3주차 (1월 22 - 24일) : DLL Injection, API 후킹, 안티 디버깅, 안티 VM, 패킹

4주차 (1월 29 - 31일) : 악성코드 분석 (1)

- 설날 주간

5주차 (2월 12 - 14일) : 악성코드 분석 (2)

6주차 (2월 19 - 21일) : 악성코드 분석 (3)

- 새내기 배움터 주간


- 개강

- 이론 12회 (오프라인)
- 일 과제(6회) : 매주 화요일 출제 수요일 자정까지 제출
- 주 과제(6회) : 매주 목요일 출제, 그 다음주 월요일 자정까지 제출

리버싱이 뭔가요 ?

저작권 기술 용어사전

역공학

[Reverse Engineering ]

완성된 제품을 상세하게 분석하여 그 기본적인 설계 내용을 추적하는 것을 의미한다. 소프트웨어에서는 기계어 코드로부터 원시 소스코드로 변환하는 기술을 특화하는 것을 의미한다. 제품의 통상적인 공정을 역으로 추적한다는 의미에서 역공학이라고 부른다. 제품이 어떻게 작동하는지 분석하여 프로그램이나 보안 메커니즘을 어떻게 작동시키는지 알아내기 위하여 사용된다.

리버싱이 뭔가요 ?

Reverse Engineering

리버스 엔지니어링, 역공학(逆工學)

① 장치 또는 시스템의 기술적인 원리를 그 구조분석을 통해 발견하는 과정이다. 이것은 종종 대상(기계 장치, 전자 부품, 소프트웨어 프로그램 등)을 조각내서 분석하는 것을 포함한다. 그리고 유지 보수를 위해 또는 같은 기능을 하는 새 장치를 원본의 일부를 이용하지 않고 만들기 위해 대상의 세부적인 작동을 분석하는 것을 포함한다.

② 소프트웨어 공학의 한 분야로, 이미 만들어진 시스템을 역으로 추적하여 처음의 문서나 설계기법 등의 자료를 얻어내는 일을 말한다. 전통적인 공학인 순공학(Forward Engineering)은 개념으로부터 실물을 얻어내는 과정이라면 역공학은 그와는 반대로 실물로부터 개념을 얻어내는 과정이라 할 수 있다. 이것은 시스템을 이해하여 적절히 변경하는 소프트웨어 유지보수 과정의 일부이다.

보통 문서 분실, 상품 분석, 안전 검사 등의 이유로 역공학을 수행한다. 소프트웨어에 대한 역공학 자체는 위법 행위가 아니지만, 이러한 수법을 사용해서 개발한 제품은 지적 재산을 침해할 위험성이 있다. 따라서 역공학 분석과 같이 악의적인 공격들로부터 소프트웨어의 주요 알고리즘 및 자료구조 등의 지적재산을 보호하기 위한 연구가 이루어지고 있다.

이러한 공격으로부터 소프트웨어를 보호하는 기법은 암호화(Encryption), 워터마킹(Watermarking), 변조 방지(Modulation Prevention), 서버 측 실행(server-side execution), 본래 코드 신뢰(Trusted native code), 자가 확인(Self-checking), 바이너리 변경(Binary modification), 흐리기(obfuscation)가 있다. 그 중 대표적으로 사용하는 역공학 방지 기법이 흐리기이다.

리버싱이 뭔가요 ?

완성된 제품을 상세하게 분석하여 그 기본적인 설계 내용을 추적하는 것을 의미한다.

소프트웨어에서는 기계어 코드로부터 원시 소스코드로 변환하는 기술을 특화하는 것을 의미한다.

제품의 통상적인 공정을 역으로 추적한다는 의미에서 역공학이라고 부른다.

소프트웨어 공학의 한 분야로, 이미 만들어진 시스템을 역으로 추적하여 처음의 문서나 설계기법 등의 자료를 얻어내는 일을 말한다.

역공학은 그와는 반대로 실물로부터 개념을 얻어내는 과정이라 할 수 있다.

소프트웨어에 대한 역공학 자체는 위법 행위가 아니지만, 이러한 수법을 사용해서 개발한 제품은 지적 재산을 침해할 위험성이 있다.

리버싱이 뭔가요 ?

- 역으로 추적?
- 기계어 코드로부터 소스코드로 변환?
- 제품이 어떻게 작동하는지 분석?
- 실물로부터 개념을 얻어내는 과정?
- 자체로는 위법이 아님?

리버싱이 뭔가요 ?

- 우리가 만들어내는 제품 ?
- 소스코드 -> 기계어 코드 ?

리버싱이 뭔가요 ?

- 우리가 만들어내는 제품은 **프로그램**
- 프로그램의 형태는 다양하게 존재할 수 있다.
EXE, ELF, APK 등등..
- 프로그램은 대부분 **컴파일 과정**을 거친다.

리버싱이 뭔가요 ?

- 컴파일 과정

Preprocess : 매크로 처리.

Compile : 소스코드를 어셈블리어로 변환

Assemble : 어셈블리어를 기계어(오브젝트 코드)로 변환

편의상 이 과정 까지를 Compile이라 부른다.

Link : 만들어진 오브젝트 코드들을 실행 가능 파일로 묶는다.

리버싱이 뭔가요 ?

- 기계어 (machine code)

기계 처리장치가 이해할 수 있는 코드.

사람이 이해하기 많이 힘들

00000400	55 8B EC 51 C7 45 FC 00 00 00 00 C7 45 FC 00 00	U<ìQÇEü....ÇEü..
00000410	00 00 EB 09 8B 45 FC 83 C0 01 89 45 FC 8B 4D 0C	..ë.<EüfÄ.‰Eü<M.
00000420	51 E8 BA 77 00 00 83 C4 04 39 45 FC 7D 1C 8B 55	Qè°w...fÄ.9Eü}.<U
00000430	0C 03 55 FC 0F BE 02 8B 4D 08 03 4D FC 0F BE 11	..Uü.‰.<M..Mü.‰.
00000440	3B C2 74 04 33 C0 EB 07 EB CA B8 01 00 00 00 8B	;Ät.3Äë.ëÊ,....<

- 어셈블리어 (assembly code)

기계어와 1:1로 매칭되는 저급언어

사람이 어느정도 이해할 수 있음.

0:	55	push	ebp
1:	8b ec	mov	ebp,esp
3:	51	push	ecx
4:	c7 45 fc 00 00 00 00	mov	DWORD PTR [ebp-0x4],0x0
b:	c7 45 fc 00 00 00 00	mov	DWORD PTR [ebp-0x4],0x0
12:	eb 09	jmp	0x1d
14:	8b 45 fc	mov	eax,DWORD PTR [ebp-0x4]
17:	83 c0 01	add	eax,0x1
1a:	89 45 fc	mov	DWORD PTR [ebp-0x4],eax
1d:	8b 4d 0c	mov	ecx,DWORD PTR [ebp+0xc]
20:	51	push	ecx
21:	e8 ba 77 00 00	call	0x77e0
26:	83 c4 04	add	esp,0x4
29:	39 45 fc	cmp	DWORD PTR [ebp-0x4],eax
2c:	7d 1c	jge	0x4a
2e:	8b 55 0c	mov	edx,DWORD PTR [ebp+0xc]
31:	03 55 fc	add	edx,DWORD PTR [ebp-0x4]
34:	0f be 02	movsx	eax,BYTE PTR [edx]
37:	8b 4d 08	mov	ecx,DWORD PTR [ebp+0x8]
3a:	03 4d fc	add	ecx,DWORD PTR [ebp-0x4]

리버싱이 뭔가요 ?

즉, 리버싱은 소스코드 없이 프로그램만 주어진 채로
이 프로그램이 어떤 동작을 하는지에 대해 분석하는 기술을 말한다.

컴파일 과정은 실제로 소스코드 -> 어셈블리어 -> 기계어 로 변환이 되는데,
기계어와 어셈블리어는 1:1 매칭이 되기 때문에 만들어진 프로그램의 기계어 코드를
어셈블리어로 해석하는 것이 가능하다.

프로그램을 분석하기 위해 아키텍처에 따른 어셈블리어 해석과 해당 운영체제 시스템 메커니즘
을 이해하는 것이 필수적이다.

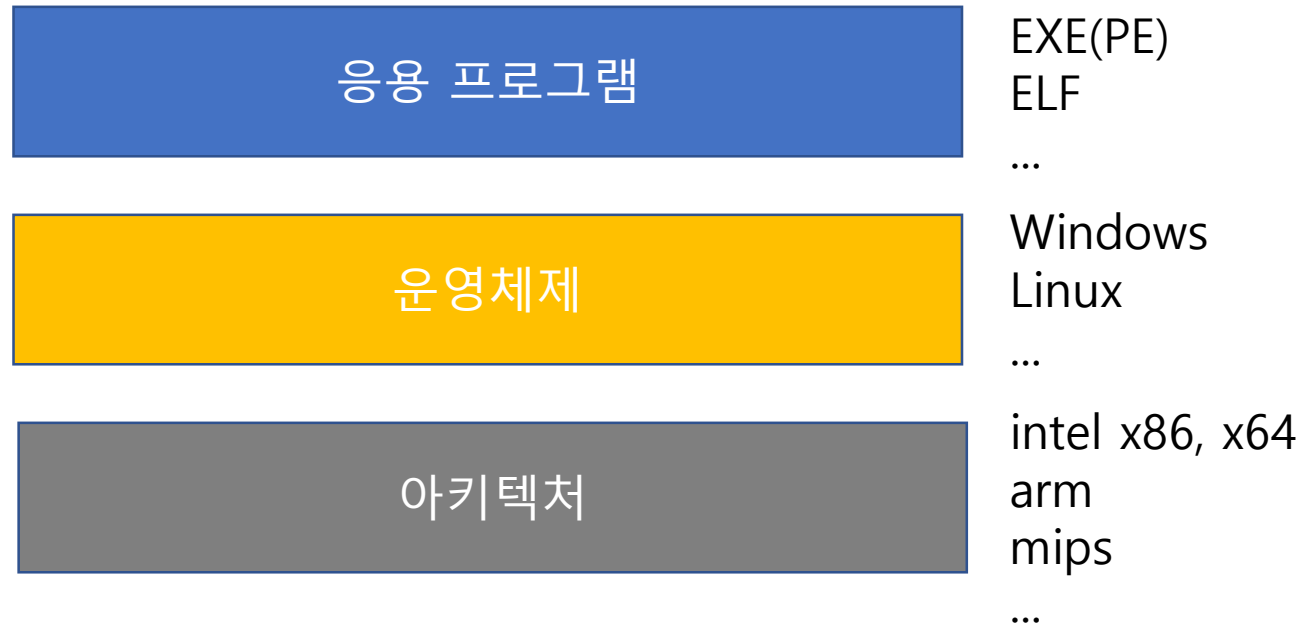
아키텍처 (Architecture)

- CPU 디자인을 말함.
- CPU 종류 마다 처리하게 되는 기계어의 형식이 다름.
ex) Intel x86, x64, arm, mips 등등..
- 보통 PC는 대부분 intel x86을 쓰고 있다고 보면 됨.
- 작은 장치들은 arm이나 mips를 많이 씀.

운영체제 (OS)

- 유저에게는 응용 프로그램을 실행할 수 있는 환경을 제공.
- 하드웨어(시스템)에게는 효율적으로 자원을 관리할 수 있게 함.
- 운영체제마다 지원하는 응용 프로그램이 다름.
ex) Windows : EXE(PE), Linux : ELF, android : APK, iOS : ipa 등..

전체 구조



기본 배경

- 현재까지의 컴퓨터는 거의 모두 프로그램 내장 방식
- 프로그램을 실행시키면 메모리에 올라감
- 메모리에 올려진 프로그램을 CPU가 실행시킴.
- 즉, 우리가 커버할 내용은 프로그램 구조, 메모리, CPU 등이 됨.

Intel x86

- 인텔이 개발한 CPU, 8086, 80186, 80286, 386, 486 등이 있음
- 제일 널리 쓰임.
- 요새는 64bit cpu 및 OS가 많이 쓰이는데, x64 혹은 amd64로 부름, 어셈블리는 몇개가 달라지고 거의 비슷.

어셈블리어

- CPU가 이해하는 기계어와 1:1로 매칭되는 저급언어
- 사람도 어느정도 이해는 함.
- C, Java와 같은 언어는 고급 언어에 낀.

어셈블리어

```
:0040D270 sub_40D270      proc near      ;
:0040D270                                     ;
:0040D270
:0040D270 var_40      = byte ptr -40h
:0040D270 arg_0      = dword ptr  8
:0040D270
:0040D270          push    ebp
:0040D271          mov     ebp, esp
:0040D273          sub     esp, 40h
:0040D276          push    ebx
:0040D277          push    esi
:0040D278          push    edi
:0040D279          lea     edi, [ebp+var_40]
:0040D27C          mov     ecx, 10h
:0040D281          mov     eax, 0CCCCCCCCh
:0040D286          rep stosd
:0040D288
:0040D288 loc_40D288:                                     ;
:0040D288          mov     eax, 1
:0040D28D          test    eax, eax
:0040D28F          jz      short loc_40D2AC
:0040D291          call    _rand
:0040D296          mov     [ebp+arg_0], eax
:0040D299          cmp     [ebp+arg_0], 4
:0040D29D          jge     short loc_40D2AA
:0040D29F          cmp     [ebp+arg_0], 0
:0040D2A3          jle     short loc_40D2AA
:0040D2A5          mov     eax, [ebp+arg_0]
:0040D2A8          jmp     short loc_40D2AC
:0040D2AA          ;
```

이런 식으로 생겨먹었다.

어셈블리어

- CPU 아키텍처마다 어셈블리어가 다릅니다.
- 세상에 많은 CPU 아키텍처가 만들어져 있습니다.
- 즉, 세상에는 많은 어셈블리어가 있습니다.
- x86, x64, arm, mips 등등..

어셈블리어

- 다 알아야하나요?
- 네. 결국에는. 하지만 처음에 배우는 한 종류가 중요합니다.
- 그리고 필요할 때, 배우면 됩니다.

어셈블리어

- Assembly
 - 저급언어
 - 기호로 이루어져 있어 되게 직관적임.
 - 문법이 2개임! (Intel, AT&T)
- 일반적인 형태 => 명령 오퍼랜드1, 오퍼랜드2
오퍼랜드 : 연산의 대상, 레지스터나 숫자가 올 수 있다.

ex1 AT&T) `mov $0x1, %eax`

ex2 Intel) `mov eax, 0x1`

이 문서는 intel 기준으로 설명하겠음.

AT&T 문법

```
0x0804846d <+0>:    push    %ebp
0x0804846e <+1>:    mov     %esp,%ebp
0x08048470 <+3>:    and     $0xffffffff0,%esp
0x08048473 <+6>:    sub     $0x20,%esp
0x08048476 <+9>:    lea     0x18(%esp),%eax
0x0804847a <+13>:   mov     %eax,0x8(%esp)
0x0804847e <+17>:   lea     0x1c(%esp),%eax
0x08048482 <+21>:   mov     %eax,0x4(%esp)
0x08048486 <+25>:   movl    $0x8048540, (%esp)
0x0804848d <+32>:   call    0x8048360 <__isoc99_scanf@plt>
0x08048492 <+37>:   mov     0x1c(%esp),%edx
0x08048496 <+41>:   mov     0x18(%esp),%eax
0x0804849a <+45>:   cmp     %eax,%edx
0x0804849c <+47>:   jle     0x80484aa <main+61>
0x0804849e <+49>:   movl    $0x8048546, (%esp)
0x080484a5 <+56>:   call    0x8048330 <puts@plt>
0x080484aa <+61>:   leave
0x080484ab <+62>:   ret
```

명령 source destination

Intel 문법

```
0x0804846d <+0>:    push    ebp
0x0804846e <+1>:    mov     ebp,esp
0x08048470 <+3>:    and     esp,0xffffffff
0x08048473 <+6>:    sub     esp,0x20
0x08048476 <+9>:    lea     eax,[esp+0x18]
0x0804847a <+13>:   mov     DWORD PTR [esp+0x8],eax
0x0804847e <+17>:   lea     eax,[esp+0x1c]
0x08048482 <+21>:   mov     DWORD PTR [esp+0x4],eax
0x08048486 <+25>:   mov     DWORD PTR [esp],0x8048540
0x0804848d <+32>:   call    0x8048360 <__isoc99_scanf@plt>
0x08048492 <+37>:   mov     edx,DWORD PTR [esp+0x1c]
0x08048496 <+41>:   mov     eax,DWORD PTR [esp+0x18]
0x0804849a <+45>:   cmp     edx,eax
0x0804849c <+47>:   jle     0x80484aa <main+61>
0x0804849e <+49>:   mov     DWORD PTR [esp],0x8048546
0x080484a5 <+56>:   call    0x8048330 <puts@plt>
0x080484aa <+61>:   leave
0x080484ab <+62>:   ret
```

명령 destination source

레지스터

- 값을 담아놓는 공간 => 다기능 변수라고 보면 됨
- CPU에 존재함
- 여러가지 레지스터가 존재함

범용 레지스터(x86 기준)

- `eax` : accumulator register // 함수의 return 값이 저장.
- `ebx` : base
- `ecx` : count
- `edx` : data register
- `esi` : source index
- `edi` : destination index
- `esp` : stack pointer // 스택 프레임 포인터.
- `ebp` : base pointer // 스택 프레임의 기준
- `eip` : instruction pointer // 실행할 코드의 주소.

레지스터(x86 기준)

```
EAX 00000004  
EBX 00209000 TIB[000020C4]:00209000  
ECX 00424A68 .data:stru_424A68  
EDX 00424A68 .data:stru_424A68  
ESI 00401350 start  
EDI 0019FF40 debug007:0019FF40  
EBP 0019FF40 debug007:0019FF40  
ESP 0019FED8 debug007:0019FED8  
EIP 0040D270 sub_40D270  
EFL 00000212
```

이렇게 생겨먹었다.

어셈블리 명령

- mov : source를 destination에 대입 한다.
mov destination, source

ex1) mov eax, 0x1

ex2) mov eax, ebx

ex3) mov DWORD PTR [ebp+4], eax

어셈블리 명령

- lea : source의 주소를 destination에 대입
lea destination, source

ex1) lea ebx, [eax]

어셈블리 명령

- add : destination에 source를 더한다.
add destination, source
ex) add esp, 0x4
- sub : destination에서 source를 뺀다.
sub destination, source
ex) sub esp, 0x20

어셈블리 명령

- inc : 레지스터의 값을 1 올린다.
- dec : 레지스터의 값을 1 내린다.

ex1) inc eax

ex2) dec ebx

어셈블리 명령

- xor : source와 destination을 xor 한 뒤 destination에 담는다.
xor destination, source
ex) xor ebx, eax
- and : source와 destination을 and 한 후 destination에 담는다.
and destination, source
ex) and esp, 0xffffffff20

어셈블리 명령

- or : source와 destination을 or 한 뒤 destination에 담는다.
or destination, source
- cmp : destination과 source를 비교함
destination이 source보다 ~할 때, 조건 성립
cmp destination, source
- jmp : 지정한 주소로 eip 지정.
jmp 0x08048841

어셈블리 명령

- je(jump equal) : 두 값이 같으면 jump
- jb(jump below) : destination이 더 작으면 jump (unsigned)
- ja(jump above) : destination 이 더 크면 jump (unsigned)
- jl(jump less) : destination이 더 작으면 jump (signed)
- jg(jump greater) : destination 이 더 크면 jump (signed)
- jne(jump not equal) : 두 값이 같지 않으면 jump

어셈블리 명령

- call : 현재 EIP를 Memory에 저장 후 지정한 주소로 jmp

ex) call 0x8048360

어셈블리 명령

- push : esp를 감소하고 지정한 데이터를 esp가 가리키고 있는 곳에 저장

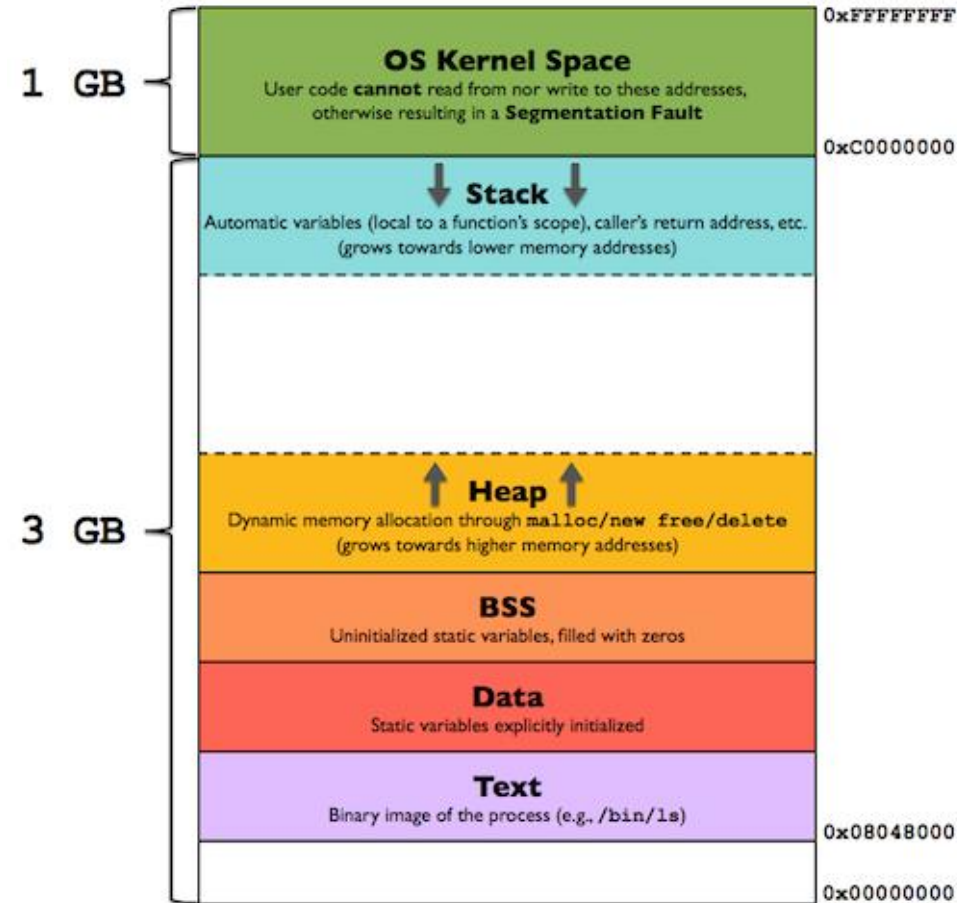
ex) push ebp

- pop : 현재 esp가 가리키고 있는 값을 해당 레지스터로 복원

ex) pop ebp

- ret : 현재 esp가 가리키고 있는 값을 eip로 바꾼다.

Memory



프로그램 실행 시, 메모리 레이아웃.

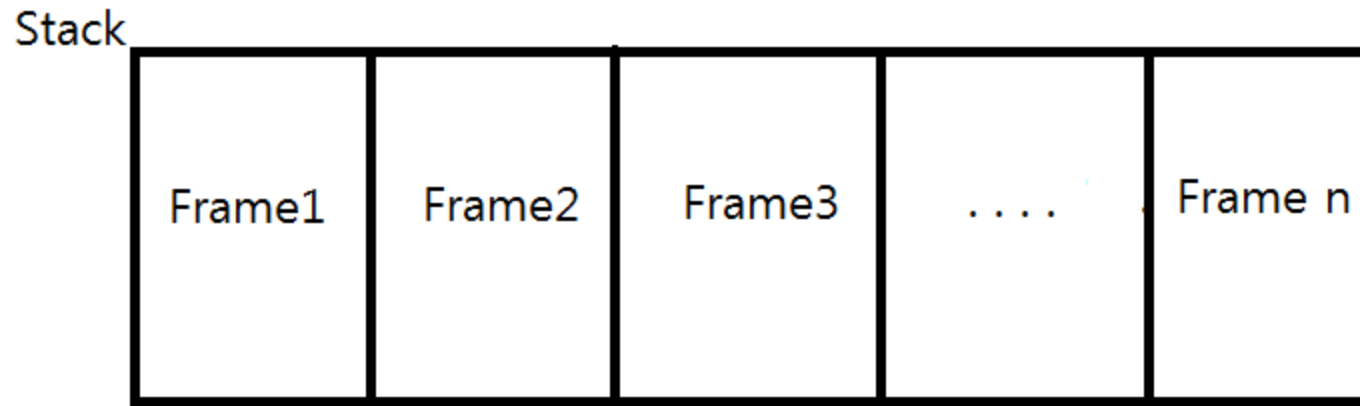
Memory

- 스택 : 함수들이 사용할 공간. 지역변수들이 위치함.
- 힙 : 동적할당을 위한 공간 malloc() 함수같은 것으로 할당 가능
- 데이터 : 문자열이나 정적 변수들이 존재.
- BSS : 전역 변수들 존재.
- Text(Code) : 실제 프로그램의 코드들이 존재함 (opcode들)

Memory

- stack frame

- 각 함수마다 사용할 공간을 할당한 것.
- 함수가 끝나면 메모리에 저장해놓은 복귀주소(Return Address)를 참조하여 이전 함수로 돌아감.



스택 프레임 예시

스택 프레임 어게인

- push : $esp -= 4$ 한 뒤 지정한 값을 스택에 저장
- pop : esp 가 가리키고 있는 값을 뽑아와 지정한 레지스터에 넣고 $esp += 4$
- ret : 현재 esp 가 가리키고 있는 값을 eip 로 저장하고 $esp += 4$ 마치 pop eip 와 같은 역할을 함.
- call : push eip ; jmp [주소] 와 같은 역할을 함

직접 분석해보자.

```
#include<stdio.h>
int add(int x, int y);
int main()
{
    int a,b;
    a = 10;
    b = 20;
    printf("%d\n",add(a,b));
}
int add(int x, int y)
{
    return x+y;
}
```

이런 식으로 소스를 짜고 컴파일을 한다!

main 함수

```
(gdb) disas main
Dump of assembler code for function main:
0x0804840b <+0>:    push    ebp
0x0804840c <+1>:    mov     ebp,esp
0x0804840e <+3>:    sub     esp,0x8
0x08048411 <+6>:    mov     DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>:   mov     DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>:   push    DWORD PTR [ebp-0x8]
0x08048422 <+23>:   push    DWORD PTR [ebp-0x4]
0x08048425 <+26>:   call    0x8048442 <add>
0x0804842a <+31>:   add     esp,0x8
0x0804842d <+34>:   push    eax
0x0804842e <+35>:   push    0x80484d0
0x08048433 <+40>:   call    0x80482e0 <printf@plt>
0x08048438 <+45>:   add     esp,0x8
0x0804843b <+48>:   mov     eax,0x0
0x08048440 <+53>:   leave
0x08048441 <+54>:   ret
End of assembler dump.
```

call add를 볼 수 있다.

leave : mov esp, ebp
pop ebp
와 같은 수행을 함.

add 함수

```
(gdb) disas add
Dump of assembler code for function add:
0x08048442 <+0>:    push    ebp
0x08048443 <+1>:    mov     ebp,esp
0x08048445 <+3>:    mov     edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:    mov     eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:    add     eax,edx
0x0804844d <+11>:   pop     ebp
0x0804844e <+12>:   ret
End of assembler dump.
(gdb) █
```

이 부분을 함수 프로로그라고 부른다.

이 부분을 함수 에필로그라고 부른다.

EIP → **<main>**

```

0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

<add>

```

0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

0xbffffff0c ESP



register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffff0c
ebp	?
eip	0x0804840b

```

EIP → <main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

0xbffff0c



register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffff08
ebp	?
eip	0x0804840c

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
EIP → 0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EBP ESP

0xbffffff0c



register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffff08
ebp	0xbffffff08
eip	0x0804840e

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
EIP → 0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

EBP

0xbffffff0c



register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffff00
ebp	0xbffffff08
eip	0x08048411

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
EIP → 0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

EBP

0xbffffff0c



register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffff00
ebp	0xbffffff08
eip	0x08048418

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
EIP → 0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

EBP
0xbffffff0c

ESP



```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffff00
ebp	0xbffffff08
eip	0x0804841f


```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
EIP → 0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

EBP

0xbffff0c

	0x14
	0x14
	0xa
	이전 함수의 EBP

Low

High

register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffefc
ebp	0xbffff08
eip	0x08048422

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
EIP → 0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

EBP

0xbffff0c

0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffef8
ebp	0xbffff08
eip	0x08048425

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

EIP → <add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

EBP

0xbffffff0c

0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffef4
ebp	0xbffffff08
eip	0x08048442

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EIP →

ESP

EBP

0xbffffff0c

0xbffffff08
0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffef0
ebp	0xbffffff08
eip	0x08048443

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
EIP → 0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EBP ESP

0xbffffff0c

0xbffffff08
0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	?
ebx	?
ecx	?
edx	?
esp	0xbffffff0
ebp	0xbffffff0
eip	0x08048443

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EIP →

EBP ESP

0xbffffff0c

0xbffffff08
0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	?
ebx	?
ecx	?
edx	0xa
esp	0xbffffff0
ebp	0xbffffff0
eip	0x08048448

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EIP →

EBP ESP

0xbffffff0c

0xbffffff08
0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	0x14
ebx	?
ecx	?
edx	0xa
esp	0xbffffff0
ebp	0xbffffff0
eip	0x0804844b

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EIP →

EBP ESP

0xbffffff0c

0xbffffff08
0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	0x1e
ebx	?
ecx	?
edx	0xa
esp	0xbffffff0
ebp	0xbffffff0
eip	0x0804844d


```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

```

```

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

EIP →

ESP

EBP

0xbffffff0c

0xbffffff08
0x0804842a
0xa
0x14
0x14
0xa
이전 함수의 EBP

Low

High

register	value
eax	0x1e
ebx	?
ecx	?
edx	0xa
esp	0xbffffef0
ebp	0xbffffff08
eip	0x0804844e

```

<main>
0x0804840b <+0>:  push  ebp
0x0804840c <+1>:  mov   ebp,esp
0x0804840e <+3>:  sub   esp,0x8
0x08048411 <+6>:  mov   DWORD PTR [ebp-0x4],0xa
0x08048418 <+13>: mov   DWORD PTR [ebp-0x8],0x14
0x0804841f <+20>: push  DWORD PTR [ebp-0x8]
0x08048422 <+23>: push  DWORD PTR [ebp-0x4]
0x08048425 <+26>: call  0x8048442 <add>
EIP → 0x0804842a <+31>: add   esp,0x8
0x0804842d <+34>: push  eax
0x0804842e <+35>: push  0x80484d0
0x08048433 <+40>: call  0x80482e0 <printf@plt>
0x08048438 <+45>: add   esp,0x8
0x0804843b <+48>: mov   eax,0x0
0x08048440 <+53>: leave
0x08048441 <+54>: ret

<add>
0x08048442 <+0>:  push  ebp
0x08048443 <+1>:  mov   ebp,esp
0x08048445 <+3>:  mov   edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:  mov   eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:  add   eax,edx
0x0804844d <+11>: pop   ebp
0x0804844e <+12>: ret

```

ESP

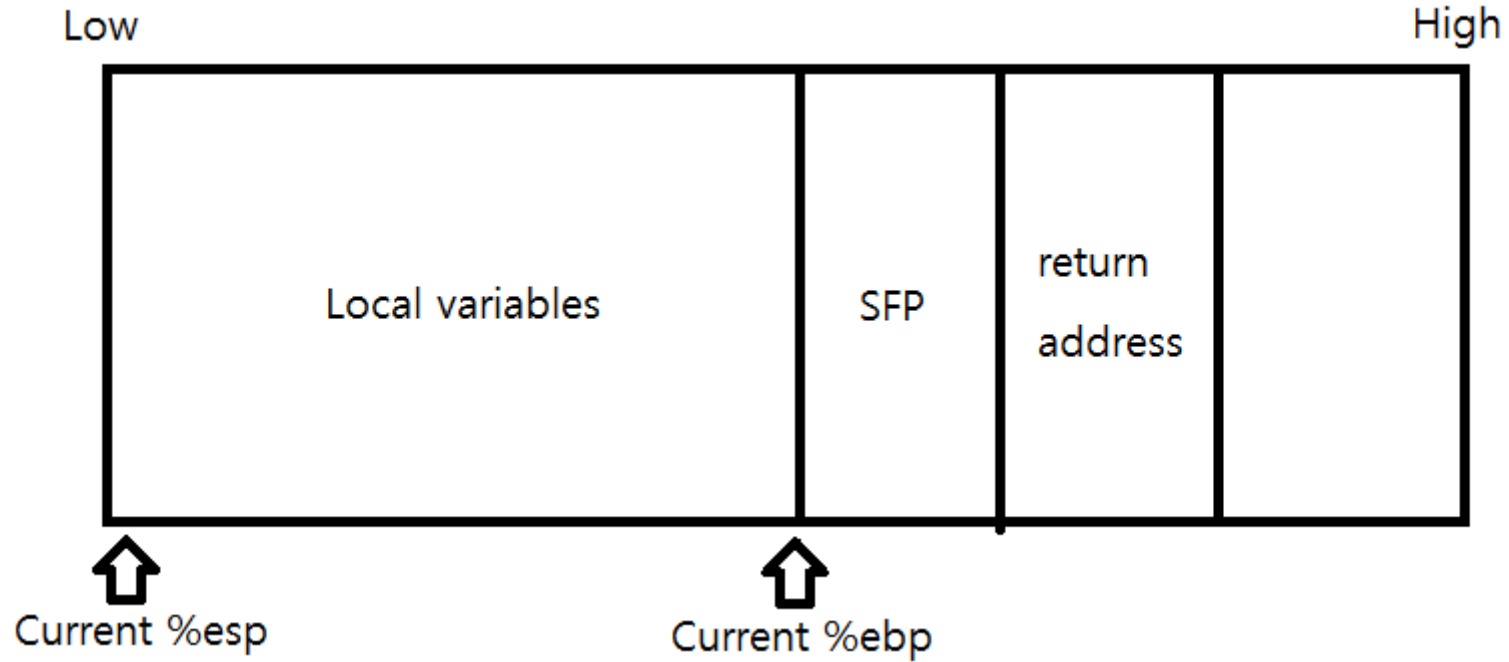
EBP

0xbffffff0c

0xbffffff08	Low
0x0804842a	
0xa	
0x14	
0x14	
0xa	
이전 함수의 EBP	
	High

register	value
eax	0x1e
ebx	?
ecx	?
edx	0xa
esp	0xbffffef0
ebp	0xbffffff08
eip	0x0804842a

SFP의 정체는?



이전 함수의 EBP 또한 스택에 저장해 둔 것이었다.

Little Endian

```
(gdb) r abcdefghijklmnopqrstuvwxyz
Starting program: /tmp/dd/a abcdefghijklmnopqrstuvwxyz

Breakpoint 1, 0x08048444 in main ()
(gdb) ni
0x08048449 in main ()
(gdb) x/20wx $eax
0xffffdc50: 0x64636261 0x68676665 0x6c6b6a69 0x706f6e6d
0xffffdc60: 0x74737271 0x78777675 0x00007a79 0x7e3f4b3
0xffffdc70: 0x00000002 0xffffdd04 0xffffdd10 0xf7fcd000
0xffffdc80: 0x00000000 0xffffdd1c 0xffffdd10 0x00000000
0xffffdc90: 0x0804822c 0xf7fc8ff4 0x00000000 0x00000000
```

메모리에 4바이트씩 거꾸로 배치됨!

Hand-lays

IDA pro 플러그인 중 Hex-rays 라는 강력한 플러그인이 있음.

```
1 void __cdecl main_0()
2 {
3     time_t v0; // eax@2
4     int v1; // eax@7
5     int v2; // eax@7
6     char v3; // [sp+Ch] [bp-54h]@1
7     int v4; // [sp+4Ch] [bp-14h]@1
8     int v5; // [sp+50h] [bp-10h]@7
9     int v6; // [sp+54h] [bp-Ch]@1
10    int v7; // [sp+58h] [bp-8h]@1
11    int v8; // [sp+5Ch] [bp-4h]@2
12
13    memset(&v3, 0xCCu, 0x54u);
14    v7 = 4;
15    v6 = 0;
16    v4 = 0;
17    printf("10000승을 달성하세요 단 패는 없습니다\n");
18    while ( 1 )
19    {
20        while ( 1 )
21        {
22            v0 = time(0);
23            sub_4010C0(v0);
24            printf("바위는 1 가위는 2 보는 3 전적 999 : ");
25            scanf("%d", &v8);
26            if ( v8 != 999 )
27                break;
28            printf("%d승 %d패\n", v6, v4);
29            sub_401023(v6);
30        }
31        if ( v8 <= 4 || v8 == 999 )
32        {
33            printf("당신은 ");
34            sub_401019(v8);
35            printf("컴퓨터는 ");
36            v1 = sub_40101E(v7);
37            sub_401019(v1);
38            v2 = sub_40101E(v7);
39            v5 = sub_40100F(v8, v2);
40            if ( v5 == 2 )
```

Hand-lays

처음부터 무분별한 hex-rays 남용은 실력 저하를 불러 일으킨다.

그러므로 어셈블리 단에서 C코드 단으로 복구 시키는 Hand-lays를 선행하는 것이 큰 도움이 된다.

Hand-lays

```
#include<stdio.h>
int add(int x, int y);
int main()
{
    int a,b;
    a = 10;
    b = 20;
    printf("%d\n",add(a,b));
}
int add(int x, int y)
{
    return x+y;
}
```

예시 프로그램.

Hand-lays

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804840b <+0>:    push    ebp
   0x0804840c <+1>:    mov     ebp,esp
   0x0804840e <+3>:    sub     esp,0x8
   0x08048411 <+6>:    mov     DWORD PTR [ebp-0x4],0xa
   0x08048418 <+13>:   mov     DWORD PTR [ebp-0x8],0x14
   0x0804841f <+20>:   push    DWORD PTR [ebp-0x8]
   0x08048422 <+23>:   push    DWORD PTR [ebp-0x4]
   0x08048425 <+26>:   call    0x08048442 <add>
   0x0804842a <+31>:   add     esp,0x8
   0x0804842d <+34>:   push    eax
   0x0804842e <+35>:   push    0x080484d0
   0x08048433 <+40>:   call    0x080482e0 <printf@plt>
   0x08048438 <+45>:   add     esp,0x8
   0x0804843b <+48>:   mov     eax,0x0
   0x08048440 <+53>:   leave
   0x08048441 <+54>:   ret
End of assembler dump.
```

```
(gdb) disas add
Dump of assembler code for function add:
   0x08048442 <+0>:    push    ebp
   0x08048443 <+1>:    mov     ebp,esp
   0x08048445 <+3>:    mov     edx,DWORD PTR [ebp+0x8]
   0x08048448 <+6>:    mov     eax,DWORD PTR [ebp+0xc]
   0x0804844b <+9>:    add     eax,edx
   0x0804844d <+11>:   pop     ebp
   0x0804844e <+12>:   ret
End of assembler dump.
(gdb) █
```


Hand-lays

- Rule

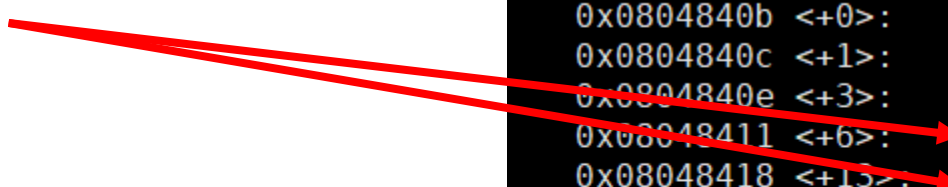
지역변수 할당.

함수 호출 전 인자 정리

매개변수 전달 받는 방식

함수 종료 전 리턴 값 지정

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804840b <+0>:    push    ebp
   0x0804840c <+1>:    mov     ebp,esp
   0x0804840e <+3>:    sub     esp,0x8
   0x08048411 <+6>:    mov     DWORD PTR [ebp-0x4],0xa
   0x08048418 <+15>:   mov     DWORD PTR [ebp-0x8],0x14
   0x0804841f <+20>:   push    DWORD PTR [ebp-0x8]
   0x08048422 <+23>:   push    DWORD PTR [ebp-0x4]
   0x08048425 <+26>:   call    0x8048442 <add>
   0x0804842a <+31>:   add     esp,0x8
   0x0804842d <+34>:   push    eax
   0x0804842e <+35>:   push    0x80484d0
   0x08048433 <+40>:   call    0x80482e0 <printf@plt>
   0x08048438 <+45>:   add     esp,0x8
   0x0804843b <+48>:   mov     eax,0x0
   0x08048440 <+53>:   leave
   0x08048441 <+54>:   ret
End of assembler dump.
```



Hand-lays

- Rule

지역변수 할당.

함수 호출 전 인자 정리

매개변수 전달 받는 방식

함수 종료 전 리턴 값 지정

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804840b <+0>:    push    ebp
   0x0804840c <+1>:    mov     ebp,esp
   0x0804840e <+3>:    sub     esp,0x8
   0x08048411 <+6>:    mov     DWORD PTR [ebp-0x4],0xa
   0x08048418 <+13>:   mov     DWORD PTR [ebp-0x8],0x14
   0x0804841f <+20>:   push    DWORD PTR [ebp-0x8]
   0x08048422 <+23>:   push    DWORD PTR [ebp-0x4]
   0x08048425 <+26>:   call    0x8048442 <add>
   0x0804842a <+31>:   add     esp,0x8
   0x0804842d <+34>:   push    eax
   0x0804842e <+35>:   push    0x80484d0
   0x08048433 <+40>:   call    0x80482e0 <printf@plt>
   0x08048438 <+45>:   add     esp,0x8
   0x0804843b <+48>:   mov     eax,0x0
   0x08048440 <+53>:   leave
   0x08048441 <+54>:   ret
End of assembler dump.
```

Hand-lays

- Rule

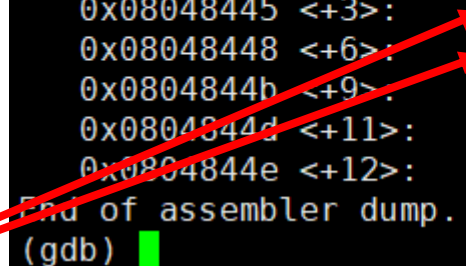
지역변수 할당.

함수 호출 전 인자 정리

매개변수 전달 받는 방식

함수 종료 전 리턴 값 지정

```
(gdb) disas add
Dump of assembler code for function add:
0x08048442 <+0>:    push    ebp
0x08048443 <+1>:    mov     ebp,esp
0x08048445 <+3>:    mov     edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:    mov     eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:    add     eax,edx
0x0804844d <+11>:   pop     ebp
0x0804844e <+12>:   ret
End of assembler dump.
(gdb) █
```



Hand-lays

- Rule

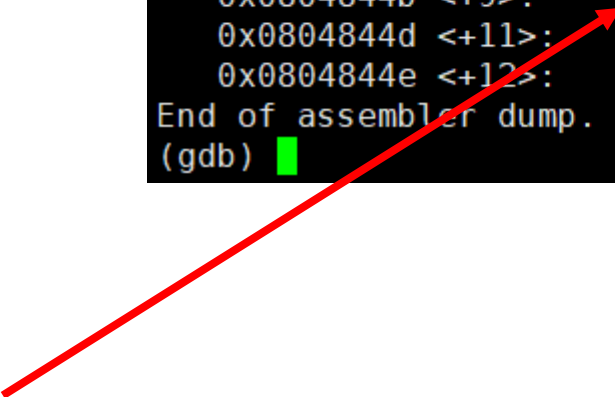
지역변수 할당.

함수 호출 전 인자 정리

매개변수 전달 받는 방식

함수 종료 전 리턴 값 지정

```
(gdb) disas add
Dump of assembler code for function add:
0x08048442 <+0>:    push    ebp
0x08048443 <+1>:    mov     ebp,esp
0x08048445 <+3>:    mov     edx,DWORD PTR [ebp+0x8]
0x08048448 <+6>:    mov     eax,DWORD PTR [ebp+0xc]
0x0804844b <+9>:    add     eax,edx
0x0804844d <+11>:   pop     ebp
0x0804844e <+12>:   ret
End of assembler dump.
(gdb) █
```



Hand-lays

- Question

반복문 처리

분기문 처리

포인터 처리

구조체 처리

과제로 나가게 될 것

과제 - 반복문 처리

(gdb) disas main

Dump of assembler code for function main:

```
0x080483db <+0>: push    ebp
0x080483dc <+1>: mov     ebp,esp
0x080483de <+3>: sub     esp,0x10
0x080483e1 <+6>: mov     DWORD PTR [ebp-0x8],0x0
0x080483e8 <+13>: mov     DWORD PTR [ebp-0x4],0x0
0x080483ef <+20>: mov     DWORD PTR [ebp-0x8],0x0
0x080483f6 <+27>: jmp     0x8048402 <main+39>
0x080483f8 <+29>: mov     eax,DWORD PTR [ebp-0x8]
0x080483fb <+32>: add     DWORD PTR [ebp-0x4],eax
0x080483fe <+35>: add     DWORD PTR [ebp-0x8],0x1
0x08048402 <+39>: cmp     DWORD PTR [ebp-0x8],0x9
0x08048406 <+43>: jle     0x80483f8 <main+29>
0x08048408 <+45>: mov     eax,0x0
0x0804840d <+50>:         leave
0x0804840e <+51>: ret
```

End of assembler dump.

(gdb)

과제 - 분기문 처리

(gdb) disas main

Dump of assembler code for function main:

```
0x0804845b <+0>:    push    ebp
0x0804845c <+1>:    mov     ebp,esp
0x0804845e <+3>:    sub     esp,0x4
0x08048461 <+6>:    mov     DWORD PTR [ebp-0x4],0x0
0x08048468 <+13>:   lea     eax,[ebp-0x4]
0x0804846b <+16>:   push    eax
0x0804846c <+17>:   push    0x8048530
0x08048471 <+22>:   call   0x8048340 <__isoc99_scanf@plt>
0x08048476 <+27>:   add     esp,0x8
0x08048479 <+30>:   mov     eax,DWORD PTR [ebp-0x4]
0x0804847c <+33>:   cmp     eax,0x10
0x0804847f <+36>:   jne     0x8048490 <main+53>
0x08048481 <+38>:   push    0x8048533
0x08048486 <+43>:   call   0x8048320 <puts@plt>
0x0804848b <+48>:   add     esp,0x4
0x0804848e <+51>:   jmp     0x804849d <main+66>
0x08048490 <+53>:   push    0x804853b
0x08048495 <+58>:   call   0x8048320 <puts@plt>
0x0804849a <+63>:   add     esp,0x4
0x0804849d <+66>:   mov     eax,0x0
0x080484a2 <+71>:   leave
0x080484a3 <+72>:   ret
```

End of assembler dump.

(gdb) x/s 0x8048530

0x8048530: "%d"

(gdb) x/s 0x8048533

0x8048533: "Correct"

(gdb) x/s 0x804853b

0x804853b: "Wrong"

(gdb)

hint:

__isoc99_scanf@plt => scanf() 함수

puts@plt => puts() 함수

과제 - 포인터 처리

(gdb) disas main

Dump of assembler code for function main:

```
0x080483db <+0>: push    ebp
0x080483dc <+1>: mov     ebp,esp
0x080483de <+3>: sub     esp,0x10
0x080483e1 <+6>: mov     DWORD PTR [ebp-0x8],0x7a69
0x080483e8 <+13>: mov     DWORD PTR [ebp-0x4],0x0
0x080483ef <+20>: lea     eax,[ebp-0x8]
0x080483f2 <+23>: mov     DWORD PTR [ebp-0x4],eax
0x080483f5 <+26>: mov     eax,DWORD PTR [ebp-0x4]
0x080483f8 <+29>: mov     eax,DWORD PTR [eax]
0x080483fa <+31>: lea     edx,[eax+0xa]
0x080483fd <+34>: mov     eax,DWORD PTR [ebp-0x4]
0x08048400 <+37>: mov     DWORD PTR [eax],edx
0x08048402 <+39>: mov     eax,0x0
0x08048407 <+44>: leave
0x08048408 <+45>: ret
```

End of assembler dump.

(gdb)

과제 - 구조체 처리

(gdb) disas main

Dump of assembler code for function main:

```
0x080483db <+0>: push    ebp
0x080483dc <+1>: mov     ebp,esp
0x080483de <+3>: sub     esp,0x20
0x080483e1 <+6>: mov     DWORD PTR [ebp-0x14],0x7b
0x080483e8 <+13>: lea     eax,[ebp-0x14]
0x080483eb <+16>: add     eax,0x4
0x080483ee <+19>: mov     DWORD PTR [eax],0x64636261
0x080483f4 <+25>: mov     WORD PTR [eax+0x4],0x65
0x080483fa <+31>: mov     eax,0x0
0x080483ff <+36>: leave
0x08048400 <+37>: ret
```

End of assembler dump.

(gdb)