리버스 엔지니어링 기본

01. 리버스 엔지니어링만을 위한 어셈블리

keyword : C vs Assembly, opcode, operand, register, little endian, stack, func call, return addr

* 어셈블리의 기본 구조

  - 어셈블리는 한 번에 한 가지 동작밖에 하지 못한다.

* 어셈블리의 명령 포맷

  - x86 CPU의 기본 구조인 IA-32를 기본 플랫폼으로 설명

  - IA-32의 기본 형태 : "명령어 + 인자"

    > 명령어 : opcode

    > 인자 : operand

      # operand가 2개인 경우 앞은 destination, 뒤는 source

* 레지스터

  - 실제 변수와는 개념 자체가 완전히 다르지만, 접근을 위해 CPU가 사용하는 변수로 생각

  - EAX : 산술 계산을 하며, 리턴값을 전달, Accumulator

  - EDX : EAX와 역할은 같되, 리턴 값의 용도로는 사용되지 않음, Data

  - ECX : 루프문을 수행할 때 카운팅하는 역할, 미리 루프를 돌 값을 넣어 두고 감고시키며
    루프 카운터가 0이 될 때까지 카운팅, Count

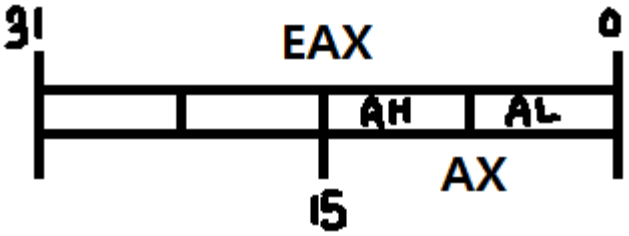
  - EBX : 레지스터가 하나쯤 더 필요하거나 공간이 필요할 때 사용

  - ESI : 문자열이나 각종 반복 데이터를 처리 또는 메모리를 옮기는 데 사용, Source Index

  - ESD : Destination Index

    > ESI에서 메모리를 읽어 EDI로 복사한다고 생각

- EAX 등의 레지스터는 32비트, 즉 4바이트의 크기



- AX는 16비트, 2바이트

- AH와 AL은 각각 8비트, 1바이트

1	2
EAX 78563412 ECX 004070C0 Ma. 004070C0 EDX 00000003 EBX 7FFDD000 ESP 0013FF38 EBP 0013FFC0 ESI FFFFFFFF EDI 7C940228 ntdll. 7C940228 EIP 00401005 Ma. 00401005	EAX **78568312** ECX 004070C0 Ma. 004070C0 EDX 00000003 EBX 7FFDD000 ESP 0013FF38 EBP 0013FFC0 ESI **00401020 Ma. 00401020** EDI 7C940228 ntdll. 7C940228 EIP **0040100D Ma. 0040100D**
3	4
EAX 78568383 ECX 004070C0 Ma. 004070C0 EDX 00000003 EBX 7FFDD000 ESP 0013FF38 EBP 0013FFC0 ESI 00401020 Ma. 00401020 EDI 7C940228 ntdll. 7C940228 EIP **00401010 Ma. 00401010**	**EAX 7856EC83** ECX 004070C0 Ma. 004070C0 EDX 00000003 EBX 7FFDD000 ESP 0013FF38 EBP 0013FFC0 ESI 00401020 Ma. 00401020 EDI 7C940228 ntdll. 7C940228 EIP **00401014 Ma. 00401014**

mov ah, byte ptr ds:[esi]

mov al, byte ptr ds:[esi]

mov ax, word ptr:[esi]

현재 ESI에 담긴 0x401020 번지에는 83 EC 40 56 값이 담겨있다.

- esi : 0x401020 - 83 EC 40 56

  > mov ah, byte ptr ds:[esi]

    # EAX를 1바이트 단위로 잘라서 4바이트로 생각해 볼 때 두 번째 바이트인 0x34 값이 0x83으로 바뀜, ah에 해당하는 위치에 새로운 값이 들어감

  > mov al, byte ptr ds:[esi]

    # al에 해당하는 위치인 첫 번째 바이트 값 0x12가 0x83으로 바뀜

  > mov ax, word ptr:[esi]

    # ax에 해당하는 2바이트가 0x8383에서 0xEC83으로 바뀜

32비트	16비트	상위 8비트 High	하위 8비트 Low
EAX	AX	AH	AL
EDX	DX	DH	DL
ECX	CX	CH	CL
EBX	BX	BH	BL

* 엔디언

  - Endian : 바이트 저장 순서

  - DWORD 0x12345678 : DWORD 4바이트 값

    > Big endian : 12 34 56 78

    > Little endian : 78 56 34 12 (인텔 CPU에서 채택한 방법, 포인터 컨트롤 속도와도 연관)

int add(int a, int b) {
  return a + b;
}
	mov eax, dword ptr ss:[esp+8]
	mov ecx, dword ptr ss:[esp+4]
	add eax, ecx
	retn

  - 첫 번째 파라미터 a는 [esp+4]

  - 두 번째 파라미터 b는 [esp+8]

* 어셈블리 명령어

- PUSH, POP

  > 스택 명령어

  > PUSHAD, POPAD 모든 레지스터를 PUSH, POP

  > push eax / push 1 / push edx ...

- MOV

  > 값을 넣는 명령어

  > mov eax, 1 / mov ebx ecx

- LEA

  > 주소를 가져오는 명령어 (mov는 값)

  > LEA는 가져올 src 오퍼랜드가 주소라는 의미로 대부분 []로 둘러싸여 있다.

- esi 0x401000, *esi 5640EC83

lea eax, dword ptr ds:[esi]	mov eax, dword ptr ds:[esi]
eax 0x401000	eax 5640EC83

- ADD

  > src에서 dest로 값을 더함

- SUB

  > src에서 dest로 값을 뺌

- INT

  > 인터럽트를 일으키는 명령어

  > 가장 많이 만나는 것은 INT 3으로 옵코드가 0xCC인 DebugBreak()

- CALL

  > 함수를 호출하는 명령어, 오퍼랜드로 번지가 붙음

  > 해당 번지를 호출하고 작업이 끝나면 CALL 다음 번지로 되돌아 옴

- INC, DEC

  > INC : i++; 이고 DEC는 i--;라고 생각

- AND, OR, XOR

  > dest와 src를 연산

  > XOR은 dest와 src를 동일한 오퍼랜드로 처리 가능함 (변수를 0으로 초기화하는 효과)

- NOP

  > NOP 아무것도 하지 말라는 명령어

- CMP, JMP

  > 비교해서 점프하는 명령어

* Stack : LIFO

1. 함수 호출 시 파라미터가 들어가는 방향
2. 리턴 주소
3. 지역 변수 사용

- 함수 안에서 스택을 사용하게 되면 보통 다음과 같은 코드가 함수의 엔트리 포인트에 생성

push ebp // ebp 레지스터를 스택에 넣는다
mov ebp, esp // esp의 값을 ebp에 넣는다. 이 함수에서 지역변수 ebp는 얼마든지 계산 가능 (ebp를 기준으로 오프셋을 더하고 빼는 작업으로 스택을 처리할 수 있게 됨)
sub esp, 50h // esp에서 50h만큼을 뺌, 스택은 LIFO 특성으로 인해 아래로 자란다.
(특정 값만큼 뺀다는 것은 그만큼 스택을 사용, 즉 50h만큼 지역 변수를 사용하겠다고 해석)

ebp가 현재 함수에서 스택의 맨 위가 되었고, 첫 번째 번지가 되었다. 그리고 사이즈를 빼가며 자리를 확보하고 있으므로 결국 지역변수는 "-"마이너스의 형태로 계산이 가능하다. 4바이트 단위로 움직이는 변수라고 가정했을 때 ebp-4라면 첫 번째 지역변수가 될 것이고, ebp-x 형태로 변수를 계산할 수 있다.

* 함수의 호출

- DWORD 타입으로 3개의 인자를 받는 함수 타입

DWORD HelloFunc(DWORD dwParam1, DWORD dwParam2, DWORD dwParam3)

- 이 함수를 다음과 같이 호출

```
main() {
    DWORD dwRet = HelloFunc(0x37, 0x38, 0x39);
    if (dwRet)
    // ...
}
```

```
push 39h
push 38h
push 37h
call 401300h
```

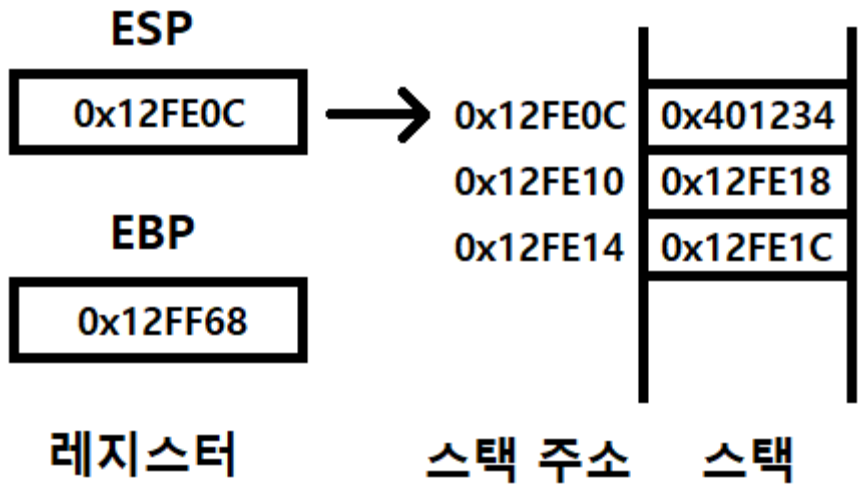
함수의 인자는 스택에 값을 LIFO 순서대로 넣기 때문에 실제 소스 코드에서 호출한 것과는 반대로 들어간다. call 401300h 안으로 들어가서 생각해 보면 지역 변수를 봤을 때는 ebp-x 등과 같이 마이너스로 스택에 보관된 변수를 사용했는데, 파라미터를 push로 넣어 놓았기 때문에 이 값들에 접근하려면 ebp에서 오프셋을 더하는 방식으로 계산해야 한다.

즉 파라미터는 ebp+x 형태로 계산할 수 있다.

첫 번째 인자 37h : ebp+0x8, 두 번째 인자 38h : ebp+0xc, 세 번째 인자 39h : ebp+0x10

- 리턴 주소

> ebp+0x4 : 함수가 끝나고 돌아갈 리턴 주소가 담김



02. C 문법과 디스어셈블링

keyword : 함수의 기본 구조, 함수의 호출 규약, 조건문, 반복문, 구조체와 API 호출

* 함수의 기본 구조

int sum(int a, int b) {
 	
int c = a + b;
 	
return c;
 	
}
	push ebp // ebp : 스택 베이스 포인터, 베이스 주소 스택에 백업
	mov ebp, esp // 현재의 스택 포인터인 esp를 ebp로 바꿈
	push ecx
	mov eax, [ebp+arg_0]
	add eax, [ebp+arg_4]
	mov [ebp+var_4], eax
	mov eax, [ebp+var_4]
	mov esp, ebp
	pop ebp
	retn

- 스택을 사용하지 않아도 될 만큼 간단한 함수 제외, 함수의 처음과 끝은 push ebp, pop ebp

* 함수의 호출 규약

- 리버스 엔지니어링을 하기 위해 디버거를 켜고 바이너리를 올려 놓았다. 가장 먼저 뭘 생각?

  > 지금 보는 이 함수의 역할은 무엇이고 파라미터는 이런 구조로 넘어가는구나!

- calling convention

  > call 문을 보고 이 함수의 인자가 몇 개이고 어떤 용도로 쓰이는지를 분석하기 위함

1. __cdecl

```
int __cdecl sum(int a, int b) {
	int c = a + b;
	return c;
}
```

```
int main(int argc, char* argv[]) {
	sum(1, 2);
	return 0;
}


sum:
push ebp
mov ebp, esp
push ecx
mov eax, [ebp+arg_0]
add eax, [ebp+arg_4]
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp
retn

main:
push 2
push 1
call calling.00401000
add esp, 8
```

1. __cdecl : add esp, 8 - 함수를 호출한 곳에서 스택을 보정하는 방식
2. 파라미터는 2개 : push 문이 2개이고 8만큼 보정, 4바이트 파라미터가 두 개
3. 리턴 값이 숫자 : 함수의 맨 마지막 부분에 eax에 들어가는 값이 숫자, 리턴 값은 주소 X

2. __stdcall

```
int __stdcalll sum(int a, int b) {
	int c = a + b;
	return c;
}
```

```
int main(int argc, char* argv[]) {
	sum(1, 2);
	return 0;
}

sum:
push ebp
mov ebp, esp
push ecx
mov eax, [ebp+arg_0]
add eax, [ebp+arg_4]
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp
retn 8

main:
push 2
push 1
call calling.00401000
```

1. __stdcalll : retn 8 - 함수안에서 스택을 보정하는 방식
2. Win32 API는 __stdcall 방식을 이용

```
call USER32.MessageBoxExA // call 후에 add esp, x가 보이지 않음
pop ebp
retn 10 // 16진수
-> __stdcall 방식이자 파라미터가 없음
```

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTP lpCaption, UINT uType);
-> 10(16진수) -> 16(10진수)/4 = 4 : 인자가 4개인 것은 알 수 있음.
```

3. __fastcall

```
int __fastcall sum(int a, int b) {
	int c = a + b;
	return c;
}
```

```
int main(int argc, char* argv[]) {
	sum(1, 2);
	return 0;
}
```

```
sum:
push ebp
mov ebp, esp
sub esp, 0Ch
mov [ebp+var_C], edx
mov [ebp+var_8], ecx
mov eax, [ebp+var_8]
add eax, [ebp+var_C]
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp
retn

main:
push ebp
mov ebp, esp
mov edx, 2
mov ecx, 1
call sub_401000
xor eax, eax
pop ebp
retn
```

1. sub esp, 0Ch로 스택 공간을 확보하고 edx 레시스터를 사용
2. 함수의 파라미터가 2개 이하일 경우, 인자를 push로 넣지 않고 ecx와 edx 레지스터를 사용
3. __fastcall : 인자가 2개 이하면서 빈번히 사용되는 함수에 많이 쓰임

4. __thiscall

```
Class CTemp {
      public:
            int MemberFunc(int a, int b);
};
```

```
mov eax, dword ptr [ebp-14h]
push eax
mov edx, dword ptr [ebp-10h]
push edx
lea ecx, [ebp-4]
call 402000
```

1. __thiscall : C++ 클래스에서 이용되는 방법
2. 현재 객체의 포인터를 ecx에 전달 (this pointer)
3. 모양은 완전히 동일한 클래스더라도 실제 오브젝트 입장에서 생각해보면 서로 다른 메모리 번지에 존재. 그리고 그것을 각각 구분하기 위해서는 현재 자신이 어떤 객체를 이용하고 있는지 구분해줄 값이 필요. -> this pointer
4. 해당 클래스에서 사용하고 있는 멤버 변수나 각종 값은 ecx 포인터에 오프셋 번지를 더함
   ex> ecx+x, ecx+y, ecx+z
5. 인자 전달 방법이나 스택 처리 방법은 __stdcall과 동일