

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

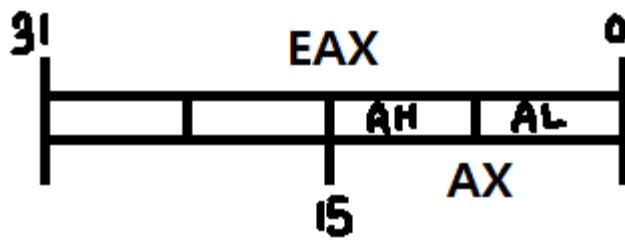
- EBX : 레지스터가 하나쯤 더 필요하거나 공간이 필요할 때 사용

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

- EDI : 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 바뀐, ah에 해당하는 위치에 새로운 값이 들어감

> mov al, byte ptr ds:[esi]

al에 해당하는 위치인 첫 번째 바이트 값 0x12가 0x83으로 바뀐

> mov ax, word ptr:[esi]

ax에 해당하는 2바이트가 0x8383에서 0xEC83으로 바뀐

| 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 속도에도 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 오프코드가 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

