리버싱 이 정도는 알아야지

Chapter 03. 리버싱을 위한 준비

http://securityfactory.tistory.com

목 차

CHAPTER 03 리버싱을 위한 준비 ---------- 3

1. **디버거 화면 구성** ---------- 4
2. **IA-32 INSTRUCTION** ---------- 10
   2.1. IA-32 어셈블리 ---------- *10*
   2.2. IA-32 레지스터 ---------- *15*
3. **어셈블리 맛보기** ---------- 22
4. **STARTUP 코드 이해하기** ---------- 25

Chapter 03

리버싱을 위한 준비

SECURITY FACTORY

1. 디버거 화면 구성

PE 파일이 실행되는 과정을 크게 나누면, 다음 두 단계로 구분할 수 있지 않을까 싶습니다.

- 실행파일의 데이터가 메모리에 적재된다.
- 적재된 코드의 연산이 이루어진다.

실행파일 데이터의 메모리 적재는 PE 로더에서 담당합니다. 파일의 헤더 정보를 참고해서 수행하죠. 그러고 나서 코드가 실행되는데, 이는 CPU 의 연산을 통해서 이루어집니다.

앞서 'Intro. 리버싱 시작하기'에서 실행파일을 설명할 때, EXE 와 DLL 이 주체적으로 동작하는 것처럼 묘사했습니다. 그런데 사실 이러한 동작의 핵심 주체는 CPU 입니다. EXE 와 DLL 은 그냥 꼭두각시였던 겁니다. CPU 는 EXE 와 DLL 을 포함한 운영체제 실행 모듈의 코드 연산을 모두 담당하고 있습니다. 그래서 CPU 를 컴퓨터의 두뇌라고도 합니다.

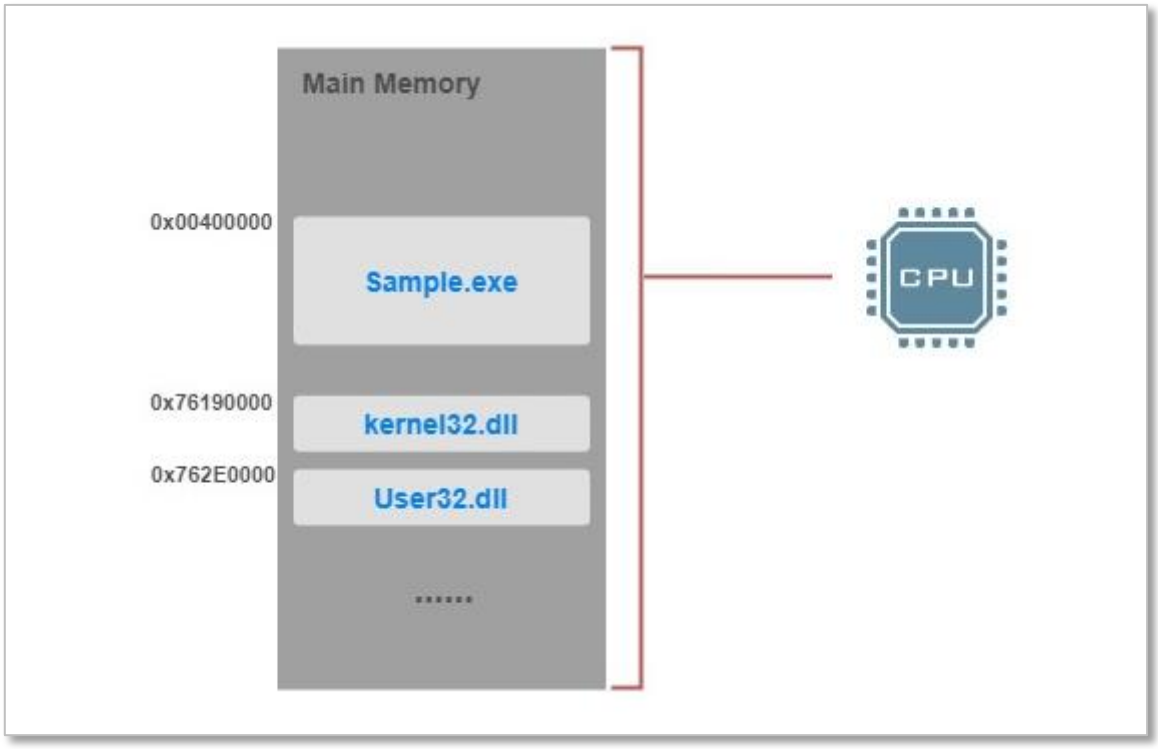


그림 1-1 코드 연산 주체 확인

음, 우리가 직접 한번 CPU가 되어 볼까요? 코드와 데이터는 준비해뒀습니다.

섹 션	주 소	어셈블리 코드

CODE	00401000 00401007 00401009	CMP DWORD PTR DS:[402000], 0 JE SHORT 00401000 ……
DATA	00402000	00 00 00 00

표 1-1 디버깅 연습

첫 번째 코드는 '0x00402000 주소에 있는 4바이트 값을 0x0과 비교해라.' 입니다. 0x00402000 주소에는 0x00000000이 기록되어 있습니다. 이 값과 0x0을 비교해야 합니다. 두 값이 일치하네요. (실제 이러한 연산은 CPU의 ALU에서 담당합니다.)

다음 코드로 넘어가겠습니다. 'JE 주소값'은 조건 점프 명령어입니다. 앞선 비교 결과가 일치할 경우, 'JE (Jump Equal)' 명령에 의해 코드 흐름이 '주소값'으로 이동하게 됩니다. 우리는 두 값이 일치한다는 사실을 기억하고 있습니다. 그래서 코드 흐름이 0x00401000으로 이동하게 된다는 사실도 쉽게 알 수 있습니다.

그런데 CPU는 두 값이 일치한다는 사실을 어떻게 기억할 수 있을까요? 다행히 CPU에도 어떤 결과를 기억할 수 있도록 임시 저장소가 마련되어 있습니다. 이를 CPU 레지스터라고 합니다.

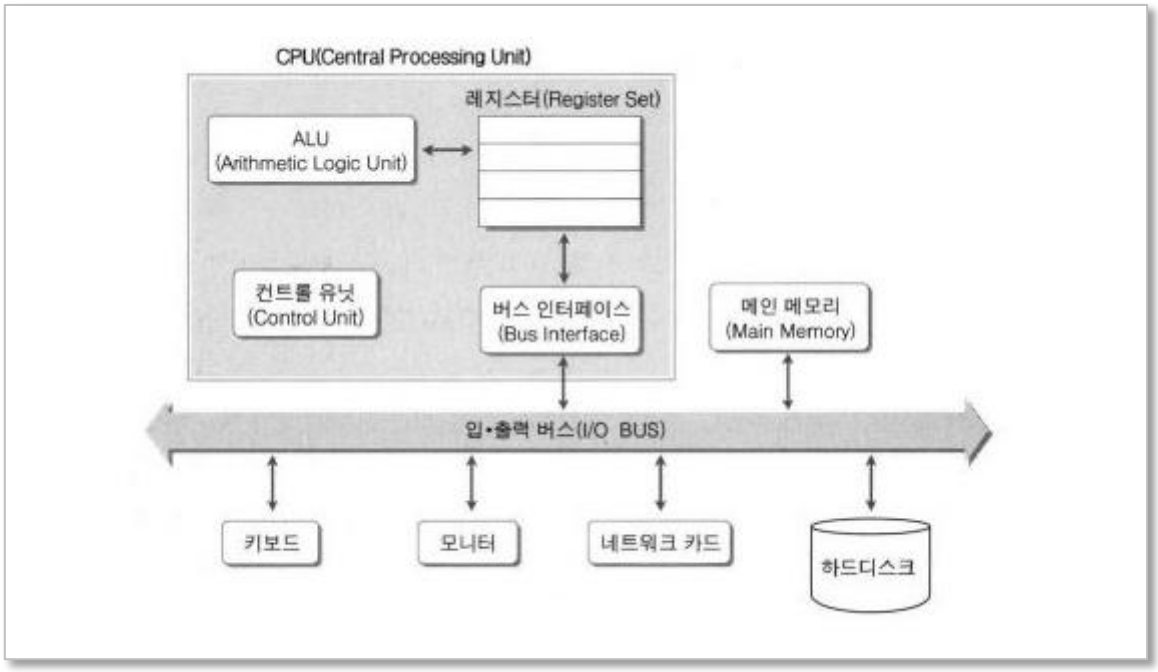


그림 1-2 컴퓨터 하드웨어의 구성[1]
출처: 뇌를 자극하는 윈도우 시스템 프로그래밍, 윤성우

[1] 뇌를 자극하는 윈도우 시스템 프로그래밍, 윤성우, 인사이트, 2012년 9월(1판 1쇄), p53

이렇게 보니 다음 세 가지 조건만 갖춰지면 직접 디버깅을 할 수 있을 것 같지 않나요?

- 코드와 데이터
- 코드를 연산할 수 있는 두뇌
- 코드 연산 결과를 임시로 저장할 수 있는 저장소

진짜 가능한지 확인해봅시다.

※ Note

블로그: http://bitly.kr/udWs
페이스북: http://bitly.kr/OrHQ

[그림 1-4]는 우리가 분석하면서 주력으로 사용하게 될 OllyDBG 입니다. OllyDBG의 기본 화면 구성이 어떻게 되어 있는지 살펴보도록 하죠.

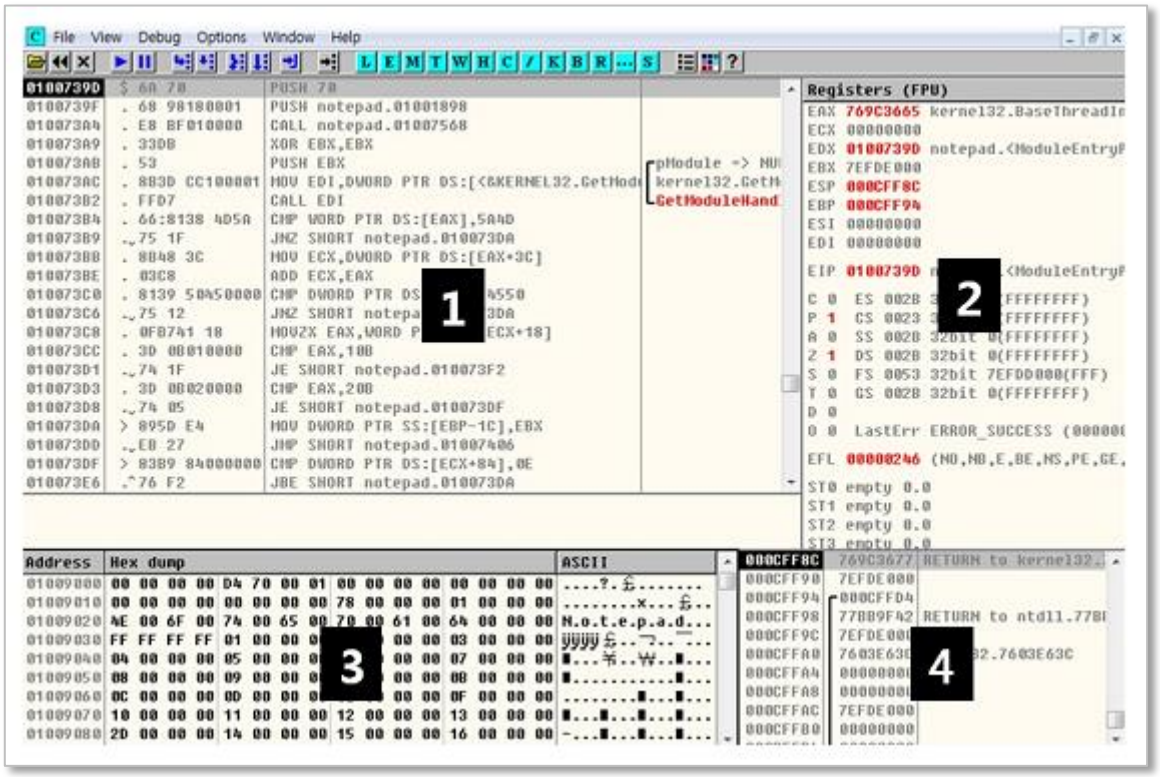


그림 1-4 OllyDBG 기본 화면 구성

① Assembled code

OllyDBG는 기계어를 보기 쉽게 어셈블리어로 번역해서 'Assembled code' 영역에 띄워주고, 어셈블리 코드를 실행 또는 중지 할 수 있도록 환경을 제공합니다. [그림 1-5]는 순서대로 "메모리 주소 | 기계어 코드 | 어셈블리로 번역된 코드"입니다.

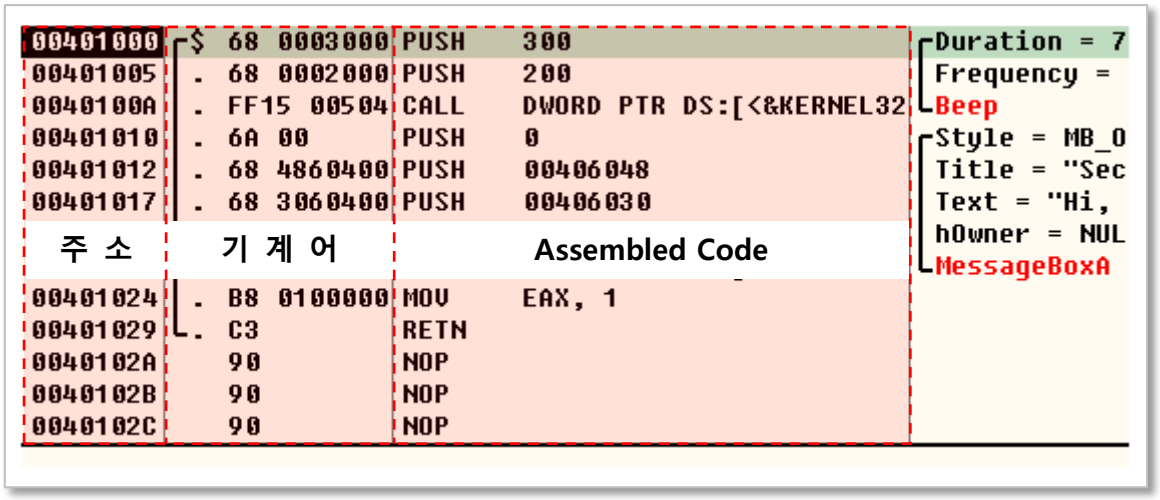


그림 1-5 Assembled code 영역

② 레지스터

레지스터는 CPU 내부에 존재하는 다목적 저장 공간입니다. 코드 연산 과정에서 필요한 값들을 임시로 저장하는 저장소라고 보면 됩니다. 용도에 따라 범용 레지스터, 세그먼트 레지스터, 플래그 레지스터 등으로 나누어져 있습니다. 자세한 설명은 '2.2 IA-32 레지스터'에서 하겠습니다.

③ Memory dump

'Memory dump'는 메인 메모리에서 할당된 공간 중, 사용자가 원하는 영역의 데이터를 확인할 수 있도록 환경을 제공합니다. 프로세스가 읽고 쓰는 값들을 확인 및 수정할 수 있습니다.

④ Stack

스택 역시 임시 저장 공간입니다. 데이터를 일시적으로 겹쳐 쌓아 두었다가 필요할 때에 꺼내서 사용할 수 있습니다. 이렇게 적고 보니 레지스터와 구분이 잘 안되네요.

우선 스택은 레지스터와 달리 요소의 개수가 제한적이지 않고 길이가 가변적입니다. 그리고 저장 공간의 위치도 다릅니다. 레지스터는 CPU에 존재하는 반면 스택은 메인 메모리에 위치하죠. 레지스터가 머리 속 기억공간이라면, 스택은 들고 다니는 수첩 정도가 되지 않을까 싶습니다. 그래서 데이터를 처리하는 속도에서도 차이가 납니다.

사실 이러한 차이를 안다고 해서 리버싱하는데 크게 도움이 되진 않습니다. 지금 단계에서는 'Win32 API를 호출할 때, 필요한 인자 정보를 전달하는 용도로 쓰인다.' 정도만 알아도 충분합니다. (32비트 이하에서만 해당되니 유의하시기 바랍니다.)

※ 참고
Windows 환경에서는 WinDBG, OllyDBG, Immunity DBG, x64 DBG, IDA Pro등 다양한 디버거들이 존재합니다. 이러한 디버거들은 저마다 장단점이 있기 때문에 어떤 것이 좋다 나쁘다를 판단하기 어렵습니다. 디버거의 선택은 분석가의 성향과 상황에 따라서 달라질 수 있습니다. 일례로 필자는 OllyDBG를 주력으로 사용합니다. 그런데 OllyDBG는 서비스 분석이나 64비트 환경에서는 재대로 분석을 못해줍니다. 이럴 땐, Immunity DBG와 x64 DBG를 사용합니다. 상황에 따라서 OllyDBG와 IDA Pro를 같이 쓰기도 합니다. 이렇게 다양한 디버거들을 사용해보고 적절히 이용할 줄 아는 것도 중요하지만 그건 분석이 어느 정도 익숙해졌을 때의 문제입니다. 지금 우리는 가장 친숙한 디버거를 선택해서 분석에 익숙해지는 것이 중요합니다.

2. IA-32 Instruction

2.1. IA-32 어셈블리

우리는 분석을 하기 위해서 다음 세가지가 필요하다는 사실을 알았습니다.

- 코드와 데이터
- 코드를 연산할 수 있는 두뇌
- 코드 연산 결과를 임시로 저장할 수 있는 저장소

알다시피 “코드와 데이터”, “코드 연산 결과를 임시로 저장할 수 있는 저장소”는 디버거에서 제공 해줍니다. 그런데 디버거가 인공지능은 아니기 때문에 코드 동작에 대한 설명은 따로 해주지 않습니다. 분석가의 역할이 여기에 있습니다.

이번 시간에는 기계어와 어셈블리어에 대해서 알아보겠습니다.

※ 참고
우리가 기계어와 어셈블리어를 공부할 때, 운영체제와 연관 지어서 생각하는 오류를 범하기도 합니다. 이는 아주 잘못된 생각입니다. 기계어는 CPU에 종속적입니다. 대표적인 CPU 제조사에는 Intel, AMD, ARM가 있습니다. 그 중에서 우리는 인텔 CPU 환경에서 사용하는 기계어를 알면 됩니다. (이를 IA-32 Instruction이라고 합니다.)

인텔 32비트 CPU에서 사용하는 기계어의 체계가 어떻게 되는지 보겠습니다. 이걸 알면 기계어에서 어셈블리어로의 변환을 마음껏 할 수 있습니다. 예를 들어보겠습니다. 다음과 같은 기계어 코드가 있습니다.

```
68 30 70 40 00 E8 06 00 00 00 ……
```

여기서 ‘0x68’은 PUSH 명령을 의미하고, 뒤에 붙은 4byte 값을 상수로 가집니다. 그래서 ‘68’

을 포함한 '30 70 40 00'이 하나의 명령코드가 됩니다. 'PUSH 0x00407040'입니다

기계어	어셈블리어
68 30 70 40 00	PUSH 0x00407030
E8 06 00 00 00 ……	……

표 2-1 기계어 예시

이걸 어떻게 알았을까요? 그 방법을 아는 것이 IA-32 Instruction을 공부하는 목적이죠. 그런데 사실 IA-32 Instruction을 깊이 있게 공부한다고 해서 리버싱에 크게 도움이 되진 않습니다. 분석을 한답시고 이 많은 기계어를 일일이 어셈블리어로 변환하는 일은 없으니까요. 물론 원리를 아는 것은 중요하고 매우 칭찬합니다만, 지금 우리에겐 그보다 어셈블리어에 익숙해지는 것이 더 중요하지 않을까 싶습니다. (변환은 디버거가 알아서 해줍니다.)

그래서 아주 간단하게만 살펴보려고 합니다. IA-32 Instruction은 크게 Opcode와 Operand로 구성되어 있습니다. Opcode가 코드 행위의 주체라면 Operand는 행위에 필요한 부가적인 요소라고 할 수 있습니다.

그림 2-1 IA-32 Instruction 기본 구조

각 항목을 세부적으로 나누면 다음과 같습니다. 총 6개의 항목으로 구성되어 있습니다. 여기서 Opcode는 반드시 존재해야 하고, 나머지 항목은 옵션으로 들어갑니다.

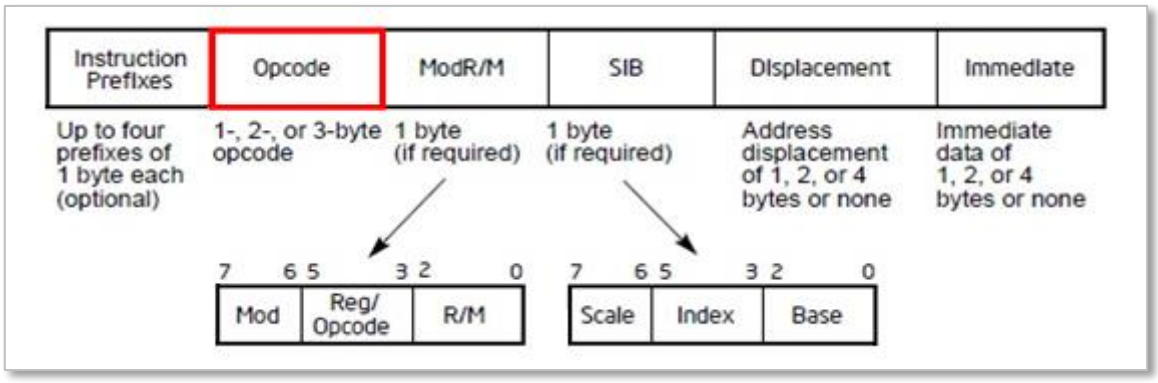


그림 2-2 Intel 64 and IA-32 Architectures Instruction Format

Opcode를 제외한 나머지 항목들은 Operand를 구성하는데 중요한 역할을 합니다. Operand의 타입과 크기를 결정하는 요소가 되죠. 그래서 각 항목들의 의미를 파악하고, 어떻게 구성되는지 알아야 합니다. 그런데

이게 좀 복잡하거든요. 더 깊이 있게 들어가진 않겠습니다.

그보다 직접 한번 변환을 해봅시다. 다음과 같은 기계어 코드가 있습니다. 먼저 0x68이 무슨 값인지 알아야겠네요.

```
68 30 70 40 00 E8 06 00 00 00 ……
```

Opcode 표를 봅시다. 여기서 0x68이 무슨 명령인지 찾아보겠습니다.

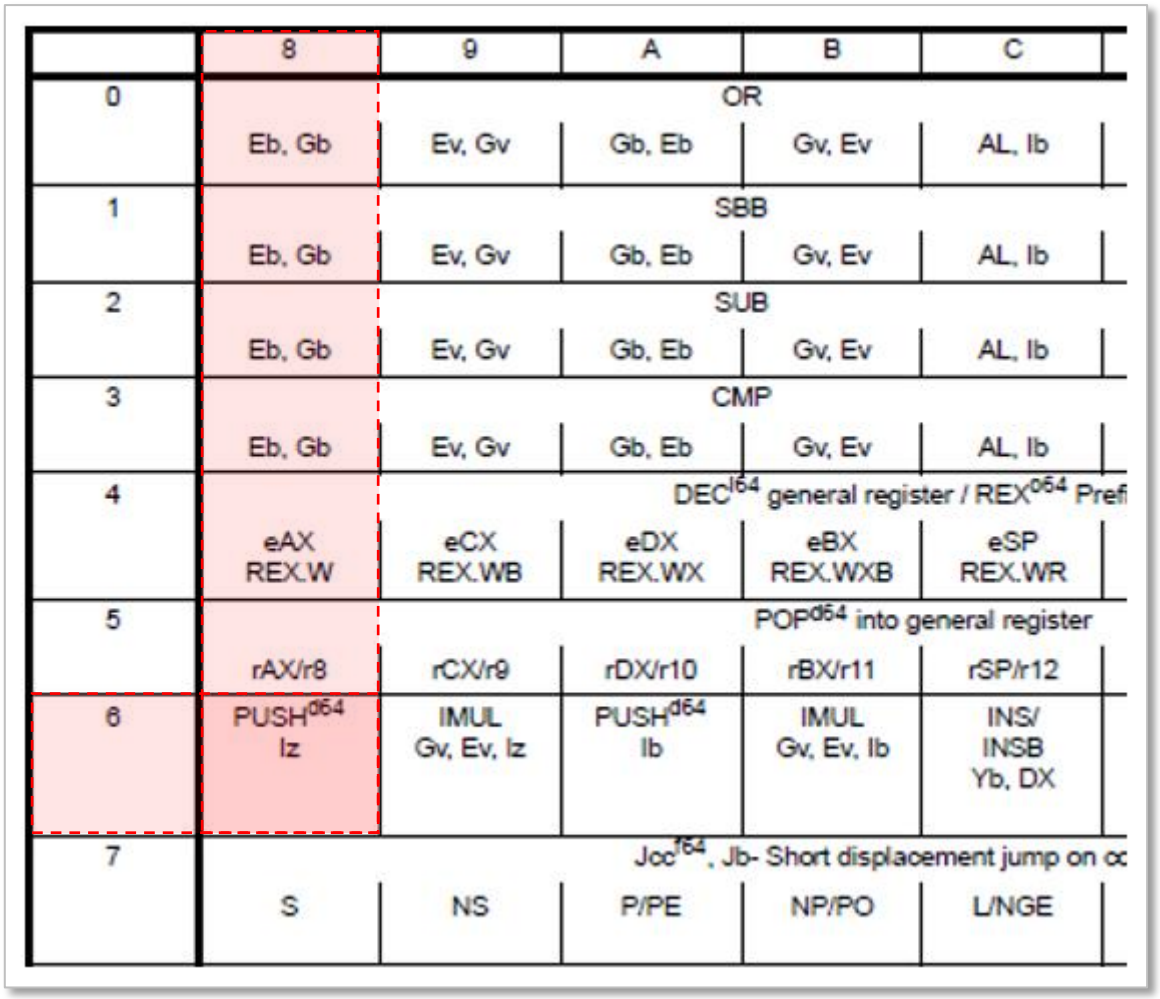
OPCODE MAP

Table A-2. One-byte Opcode Map: (00H — F7H) *

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES^{i64}	POP ES^{i64}
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS^{i64}	POP SS^{i64}
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA^{i64}
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA^{i64}
4	INC^{i64} general register / REX^{o64} Prefixes eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH^{d64} general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, Jb - Short-displacement jump on condition O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1^{1A} Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	TEST Eb, Gb	Ev, Gv	XCHG Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15

그림 2-3 One-byte Opcode Map

0x68은 PUSH 명령어이고, 오퍼랜드로 Iz 타입 값을 가집니다.



	8	9	A	B	C
0	OR				
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib
1	SBB				
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib
2	SUB				
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib
3	CMP				
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib
4	DEC[i64] general register / REX[o64] Prefi				
	eAX REX.W	eCX REX.WB	eDX REX.WX	eBX REX.WXB	eSP REX.WR
5	POP[d64] into general register				
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12
6	PUSH[d64] Iz	IMUL Gv, Ev, Iz	PUSH[d64] Ib	IMUL Gv, Ev, Ib	INS/ INSB Yb, DX
7	Jcc[f64], Jb- Short displacement jump on cc				
	S	NS	P/PE	NP/PO	L/NGE

그림 2-4 "0x68" 명령어 확인

이제 우리는 Operand 타입표를 보고, 그 특성을 파악해야 합니다. Iz는 4바이트 상수 값을 의미하네요.

※ Operand Type

Addressing Method

I	Immediate data: the operand value is encoded in subsequent bytes of the instruction. → **상수**

Operand Type

z	Word for 16-bit operand-size or doubleword for 32 or 64-bit operand-size. → 32Bit: **DWORD**

그 결과 0x68은 PUSH 명령어가 되고, 따라오는 4byte 값은 오퍼랜드가 됩니다.

```
68 30 70 40 00          → PUSH 0x00407030
```

우선 IA-32 Instruction 체계가 이렇다는 정도만 알고 넘어가겠습니다. 이보다 우리는 IA-32 어셈블리어를 알고, 익숙해지는 것이 중요합니다. 그런데 어셈블리어도 생각보다 많기 때문에 한번에 숙지하는 것은 어렵습니다. 그렇다고 외울 필요는 없습니다. 동작 과정에서 일어나는 변화를 보면서 익히는 것과 쓰임새도 모른채 무작정 외우는 것은 분명 차이가 있겠죠? 어셈블리어 하나하나에 집착하지 마세요. 예를 들어 MOV 명령어는 “값을 옮기는 명령어구나!" 정도만 알아도 됩니다. 꾸준히 분석하면서 자연스럽게 체득하는 것이 리버싱을 오랫동안 재미있게 할 수 있는 방법입니다.

다음은 리버싱을 하면서 자주 접하는 명령어입니다. 언급했듯이 편하게 보면서 어떤 명령어가 어떤 쓰임새를 가지는지 정도만 이해하기 바랍니다.

명령어	설 명
PUSHAD	8개의 범용 레지스터의 값을 스택에 저장한다.
POPAD	PUSHAD 명령에 의해서 스택에 저장된 값들을 다시 레지스터에 입력한다.
PUSH A	A 값을 스택에 넣는다.
POP 레지스터	스택에서 값을 꺼내서 레지스터에 넣는다.
INC A	A 값을 1 증가시킨다.
DEC A	A 값을 1 감소시킨다.

표 2-2 자주 사용하는 어셈블리

명령어	설 명
ADD A B	A 값과 B 값을 더해서, 그 결과를 A에 저장한다.
SUB A B	A 값에서 B 값을 빼고, 그 결과를 A에 저장한다.
IML A B	A 값과 B 값을 곱하고, 그 결과를 A에 저장한다.
LEA A B	B 값을 A로 옮긴다.
MOV A B	B 값을 A로 옮긴다.
XCHG A B	A 값과 B 값을 바꾼다.

표 2-3 자주 사용하는 어셈블리

명령어	설 명
OR A B	A 값과 B 값을 OR 연산한다.
XOR A B	A 값과 B 값을 XOR 연산한다. 주로 레지스터를 초기화할 때 많이 사용한다.
AND A B	A 값과 B 값을 AND 연산한다.

TEST A B	A 값과 B 값을 AND 연산한다. (함수 호출 리턴 값을 확인하는 용도로 많이 사용된다.)
연산 결과 값이 A에 저장되지 않지만 ZF 플래그를 설정에 영향을 준다. 연산 결과가 0이면 ZF 가 1이 되고 연산 결과가 0이 아니면 ZF 는 0이 된다.	
CMP A B	비교구문으로 A와 B의 값이 같은지 판단한다.
같을 경우 ZF는 1이 되고, 다를 경우 0이 된다.	

표 2-4 자주 사용하는 어셈블리

명령어		설 명
JMP	**Address**	해당 주소로 무조건 이동한다.
JZ(Jump if Zero)	**Address**	연산 결과가 0이면(ZF=1) 이동하고, 아니면(ZF=0) 다음 명령을 실행한다.
ZE(Jump if Equal)	**Address**	
JNZ(Jump if Not Zero)	**Address**	연산 결과가 0이 아니면(ZF=0) 이동하고, 0이면 (ZF=1) 다음 명령을 실행한다.
JNE(Jump if Not Equal)	**Address**	

표 2-5 자주 사용하는 어셈블리

명령어	설 명
MOVE DWORD PTR DS:[Address], EAX	Address부터 4Byte 값을 EAX로 복사한다.
CALL DWORD PTR DS:[Address]	Address부터 4Byte 주소 값을 호출한다

표 2-6 자주 사용하는 어셈블리

2.2. IA-32 레지스터

앞서 설명한 것처럼 레지스터는 코드 연산 과정에서 필요한 값들을 임시로 저장하는 저장소입니다. 범용 레지스터, 명령 포인터 레지스터, 상태 플래그 레지스터, 세그먼트 레지스터로 구성되어 있습니다.

※ 참고
IA-32는 지원하는 기능도 많고 그만큼 레지스터도 다양합니다. 그 중에서 우리가 공부하는 레지스터의 정식 명칭은 IA-32의 Basic program execution registers입니다.

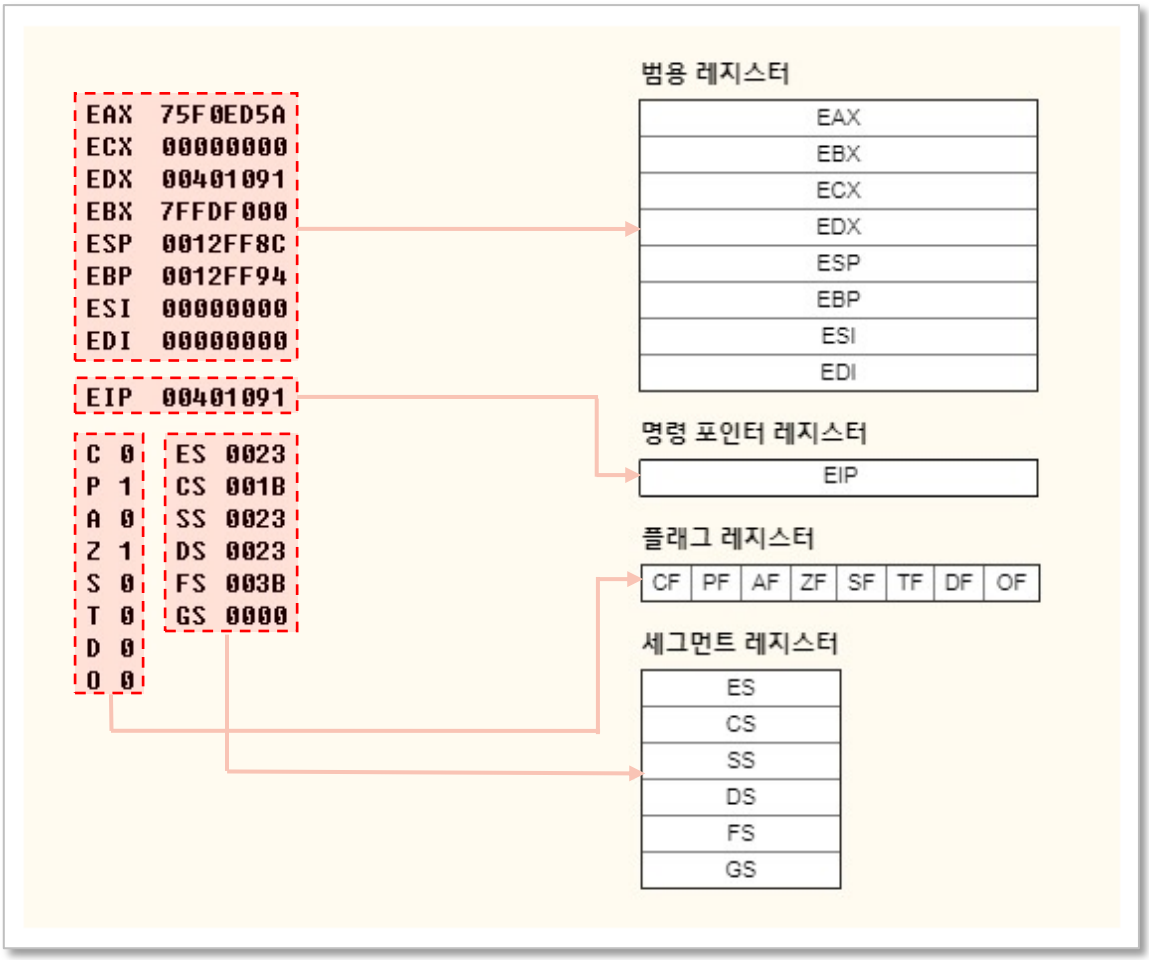


그림 2-5 IA-32 레지스터

이들 레지스터가 주로 어떤 용도로 사용되는지 가볍게 보고 넘어가겠습니다.

2.2.1 범용 레지스터

이름처럼 범용적으로 사용되는 레지스터입니다. 용도가 한정적이지 않고, 다양한 목적으로 쓰이는 저장소라고 보면 됩니다. IA-32에서 각 범용 레지스터들의 크기는 32bit(4byte)입니다.

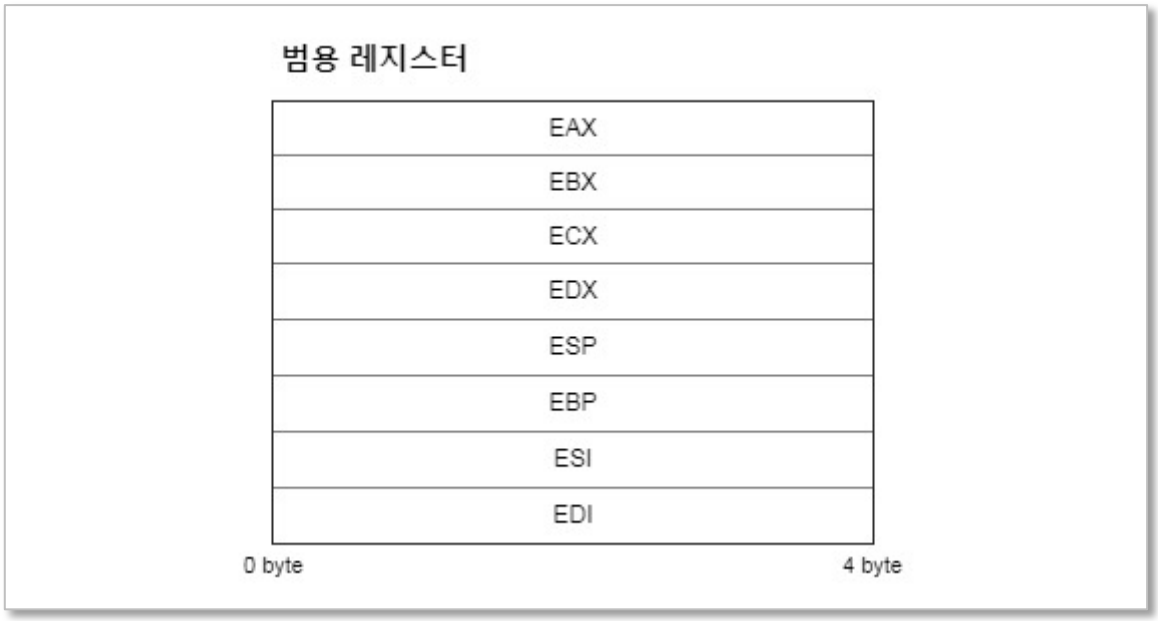


그림 2-6 범용 레지스터

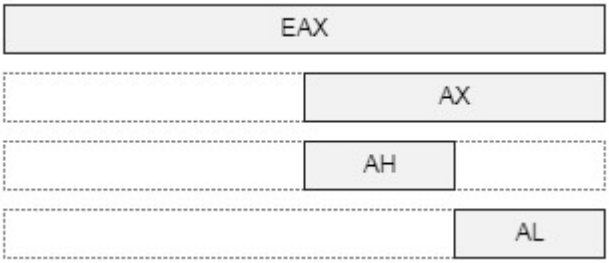
이 중에서 주 용도가 어느 정도 정해져 있는 레지스터들이 있습니다. 이 정도만 알아도 충분합니다. (단, ESP와 EBP를 제외하고는 다른 용도로도 충분히 사용될 수 있습니다.)

레지스터	용 도
EAX **(Extended Accumulator Register)**	함수 호출에 대한 결과 값이 저장된다.
ECX **(Extended Counter Register)**	코드의 반복이 필요할 경우, 반복 카운터 값이 저장된다.
ESI **(Extended Source Index)**	데이터를 복사하거나 조작할 때, 원본 데이터의 위치 주소가 저장된다.
EDI **(Extended Destination Index)**	데이터를 복사할 때 조작할 때, 목적지의 주소가 저장된다.
ESP **(Extended Stack Pointer)**	하나의 스택 프레임의 끝 지점 주소가 저장된다. PUSH, POP 명령어에 따라서 ESP의 값이 4byte씩 변한다.
EBP **(Extended Base Pointer)**	하나의 스택 프레임의 시작 지점 주소가 저장된다.

표 2-7 범용 레지스터의 주 용도

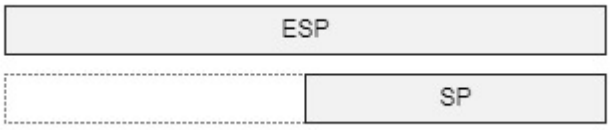
※ 참고

IA-32에서 범용 레지스터는 '확장되었다(Extended)'는 의미로 이름의 첫 글자에 'E'가 붙습니다. 16bit 레지스터인 AX, BX, CX, DX 등이 32bit 환경이 되면서 확장된 것입니다. 그리고 용도에 따라 16bit 레지스터를 기준으로 상위 8bit, 하위 8bit로 나누어 사용하기도 합니다. 예를 들어 EAX 레지스터는 32bit, AX는 16bit, AH는 상위 8bit, AL은 하위 8bit를 의미합니다.



32bit (4byte)	16bit (2byte)	상위8bit (1byte)	하위8bit (1byte)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

ESI, EDI, EBP, ESP 레지스터는 32bit와 16bit로 크기로 사용할 수 있습니다.



32bit	16bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

2.2.2 명령 포인터 레지스터

EIP(Extended Instruction Pointer)는 말 그대로 다음에 실행해야 할 명령어의 메모리 주소를 저장하는 레지스터입니다. CPU가 EIP 레지스터 값을 보고 다음에 실행해야 할 명령어를 아는 겁니다.

Sample 01.exe를 OllyDBG에 올려보세요. 코드의 시작 주소와 EIP 레지스터에 기록되어 있는 주소 정보가 일치하죠?

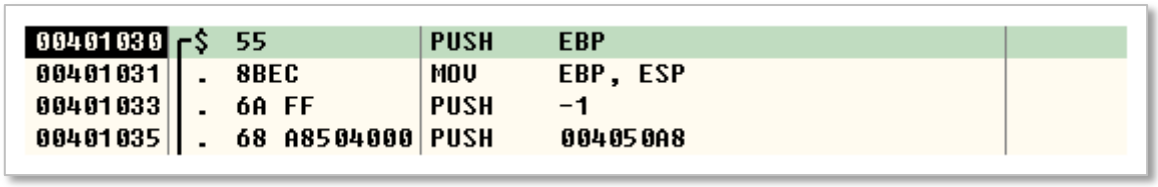


그림 2-7 실행코드 위치 확인

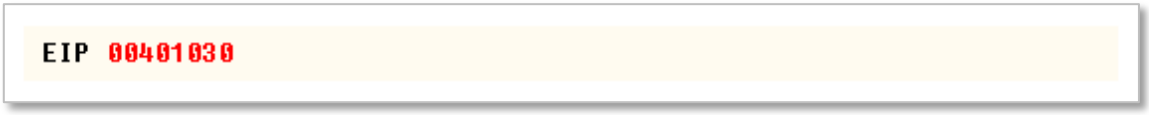


그림 2-8 EIP 레지스터에 기록되어 있는 실행코드 위치 확인

코드를 한 줄씩 실행시켜봐도 같습니다. 코드 실행에 맞춰서 EIP 값도 바뀝니다. 이러한 용도로 사용화려고 만든 레지스터입니다.

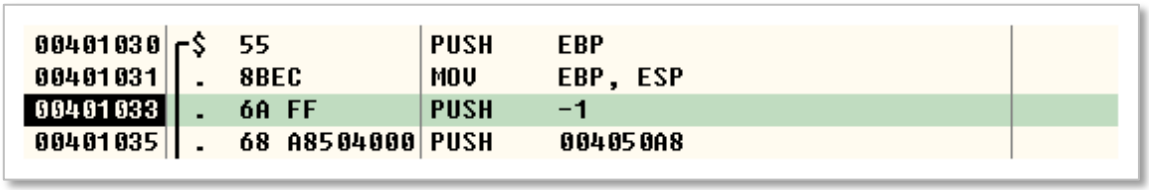


그림 2-9 실행코드 이동

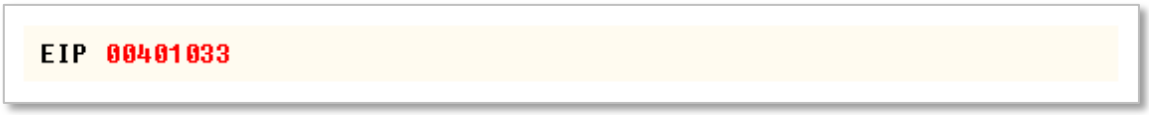


그림 2-10 EIP 레지스터에서 이동된 주소 확인

2.2.3 플래그 레지스터

플래그 레지스터는 행위에 대한 상태와 처리 결과를 나타냅니다. 총 8개로 구성되어 있고, 각각의 크기가 1bit이기 때문에 0 또는 1의 값을 가집니다.

그림 2-11 플래그 레지스터

다음은 주요 플래그 레지스터의 용도입니다. (분석하면서 플래그 레지스터를 신경 쓰는 일이 잘 없습니다. 그나마 본다고 해도 ZF 레지스터 정도가 아닐까 싶습니다.)

레지스터	용 도
CF (Carry Flag)	부호 없는 수의 연산결과, Overflow가 발생했을 때 1로 설정된다.
PF (Parity Flag)	코드 연산 결과, 최하위 1byte 에서 1의 값을 가지는 bit의 수가 짝수일 경우 1, 홀수일 경우 0으로 설정된다.
ZF (Zero Flag)	산술이나 비교 연산의 결과를 나타낸다. 연산 결과가 0일 경우 1, 0이 아닐 경우 0으로 설정된다.
SF (Sign Flag)	산술 연산의 결과 값에 대한 부호를 나타낸다. (양수 = 0, 음수 = 1)
TF (Trap Flag)	Single-step mode의 프로세서 연산을 허용한다. 디버거 프로그램의 경우 TF 플래그를 설정해서 한번에 하나의 명령어만 실행시키도록 만들고 레지스터와 메모리 상에서 그 영향을 조사할 수 있게 한다.
OF (Overflow Flag)	부호 있는 수의 연산결과, Overflow가 발생했을 때 1로 설정된다.

표 2-8 플래그 레지스터의 용도

앞서 “행위에 대한 상태와 처리 결과를 나타낸다.”라고 했는데 무슨 말인지 잘 모르겠네요. 설명이 필요해 보입니다. 두 단계로 나눠서 살펴보겠습니다.

※ Note

블로그: http://bitly.kr/udWs
페이스북: http://bitly.kr/OrHQ

2.2.4 세그먼트 레지스터

세그먼트 레지스터는 '이걸 굳이 포함시켜야 할까?'를 고민할 정도로, 지금 우리에게는 중요하지 않습니다. 무슨 말인지 모르겠다면 쿨하게 넘어갑시다. 'FS 레지스터에는 TEB 주소가 지정되어 있다.'는 정도만 기억하기 바랍니다. (자세한 내용은 나중에 STEP 03에서 확인하겠습니다.)

프로그램에 정의 된 특정 영역(코드, 데이터, 스택 등)을 세그먼트라고 합니다. 그리고 세그먼트 레지스터에는 해당 세그먼트에 대한 오프셋이 저장됩니다. 세그먼트 레지스터는 총 6개이고, 각 크기는 16bit입니다.

레지스터	용 도
CS (Code Segment)	실행 가능한 명령어가 존재하는 세그먼트의 오프셋이 저장된다.
DS (Data Segment)	프로그램에서 사용되는 데이터가 존재하는 세그먼트의 오프셋이 저장된다.

SS (Stack Segment)	스택이 존재하는 세그먼트의 오프셋이 저장된다.
ES (Extra Segment)	추가 레지스터이다. 주로 문자 데이터의 주소를 지정하는데 사용한다.
FS (Data Segment):	추가 레지스터이다. TEB(Thread Environment Block) 주소를 지정하는데 사용한다.

표 2-9 세그먼트 레지스터의 용도

3. 어셈블리 맛보기

백날 보는 것보다 한번 해보는게 더 낫겠죠? 그래서 'Assembly.exe'를 준비했습니다. 코드를 한 줄씩 실행하면서 어떻게 동작하는지 살펴보고, 변화를 파악해 봅시다.

```
00401000 XOR      EAX, EAX
00401002 MOV      EBX, 2000
00401007 ADD      EBX, EDX                    Assembly.<ModuleEntryP
00401009 CALL     DWORD PTR DS:[EBX]
0040100B RETN
0040100C DB       00
0040100D DB       00
0040100E DB       00
0040100F DB       00
00401010 PUSH     16
00401012 POP      ECX                         kernel32.76CBED6C
00401013 DEC      ECX
00401014 MOV      AL, BYTE PTR DS:[EBX+10]
00401017 XOR      AL, 0FF
00401019 MOV      BYTE PTR DS:[EBX+10], AL
0040101C INC      EBX
0040101D CMP      ECX, 0
00401020 JNZ      SHORT 00401013
00401022 PUSH     00403010                    FileName = "셔, 슌뭴뻤
00401027 CALL     <JMP.&kernel32.DeleteFileA> DeleteFileA
0040102C RETN
0040102D DB       00
```

그림 3-1 'Assembly.exe' 코드

※ Note

블로그: http://bitly.kr/udWs
페이스북: http://bitly.kr/OrHQ

4. StartUp 코드 이해하기

StartUp 코드는 실행파일이 만들어지는 과정에서 컴파일러가 집어넣는 코드입니다. 파일이 동작하는데 필요한 설정코드로 구성되어 있습니다. 그런데 이 코드가 main( ) 함수의 앞 단에 위치하고 있습니다. 우리의 1차 목적은 분석을 통해서 실행파일의 기능을 파악하는 것이죠. 그렇기 때문에 StartUp 코드는 빨리 건너뛰고, main( )을 찾아갈 줄 알아야 합니다.

무슨 말인지 확인해 봅시다. 다음은 Sample 01.exe 파일의 소스코드입니다.

```
#include <windows.h>

int main( )
{
        Beep(0x200, 0x300);

        MessageBoxA(NULL, "Hi, Have a nice day!", "SecurityFactory", 0);

        return 0;
}
```

표 4-1 Sample 01.exe 소스코드

Sample 01.exe를 OllyDBG에 올려보겠습니다. Beep( )와 MessageBoxA( ) 호출 코드가 바로 나올 줄 알았는데 안보이네요. 여기가 StartUp 코드입니다.

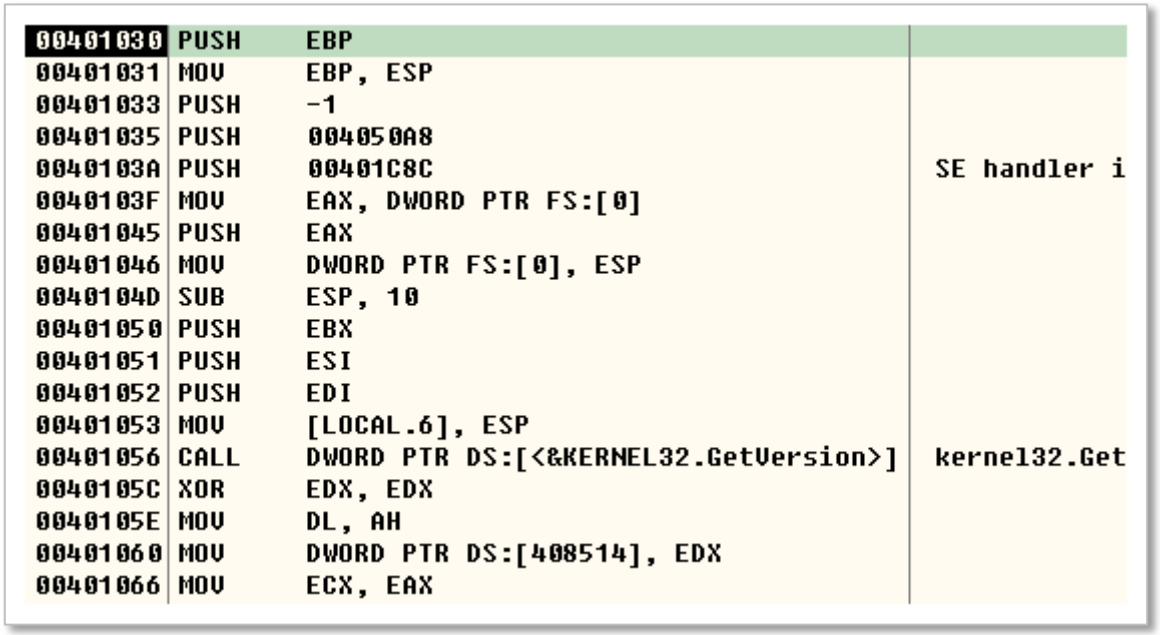


그림 4-1 Sample 01.exe의 StartUp 코드 시작지점

그리고 코드를 따라가다 보면, main( ) 함수 호출 지점이 나옵니다.

```
004010D2 PUSH    EAX
004010D3 PUSH    DWORD PTR DS:[40851C]
004010D9 PUSH    DWORD PTR DS:[408518]
004010DF CALL    00401000                -> main( ) 함수 호출
004010E4 ADD     ESP, 0C
004010E7 MOV     [LOCAL.7], EAX
```

그림 4-2 main( ) 호출 코드

'F7' 키를 사용해서 내부로 들어가보면, 소스로 확인했던 main( ) 함수 코드가 있습니다.

```
00401000 PUSH    300                    ┌Duration = 768. ms
00401005 PUSH    200                    │Frequency = 200 (512.)
0040100A CALL    DWORD PTR DS:[<&KERN   └Beep
00401010 PUSH    0                      ┌Style = MB_OK|MB_APPLMODAL
00401012 PUSH    00406048               │Title = "SecurityFactory"
00401017 PUSH    00406030               │Text = "Hi, Have a nice day!"
0040101C PUSH    0                      │hOwner = NULL
0040101E CALL    DWORD PTR DS:[<&USER   └MessageBoxA
00401024 MOV     EAX, 1
00401029 RETN
```

그림 4-3 main( ) 함수 코드

이 작업을 원활하게 할 수 있어야 합니다. 그 실력을 쌓는게 목적입니다. 그런데 Stub 코드에서 main( ) 함수 호출 지점을 찾는 것은 많은 훈련이 필요합니다. 컴파일 환경(개발 도구 및 버전)에 따라 모양이 다르기 때문에 다양한 파일을 많이 분석해보는 것이 중요하죠. 그렇다고 아무런 정보 없이 무작정 찾아 들어가면서 확인하는 것은 매우 비효율적이겠죠? 분석 능력을 쌓는데도 별 도움이 되지 않을 겁니다. 그럼 어떻게 접근해야 할까요?

Stub 코드의 모양이 컴파일 환경에 따라 달라진다고 하지만 '프로그램이 동작하는데 필요한 정보를 얻기 위한 코드'라는 것은 변하지 않습니다. 이러한 기능에 필요한 코드를 안다면 많은 도움이 될 것입니다. 다음은 가장 기본적인 형태라고 할 수 있는 Visual Studio 6.0의 C++ 환경에서 컴파일 된 실행파일의 Stub 코드입니다.

SE Handler 설치

Getversion() 호출

_heap_init() 호출
〉내부에서 HeapCreate() 호출
〉Heap 영역을 초기화한다.

_ioinit() 호출
〉내부에서 ExitProcess() 호출
〉Heap 영역 초기화에 실패하면 실행된다.

GetStartupInfoA() 호출

GetCommandLineA() 호출
〉프로세스를 생성할 때, 전달된 명령 문자열을 획득한다.

_crtGetEnvironmentStrings() 호출
〉내부에서 GetEnvironmentStrings() 호출
〉현재 프로세스의 환경 변수를 획득한다.

GetModuleHandleA() 호출
〉현재 프로세스의 ImageBase 주소 획득한다.

main() 호출
〉main() 함수 코드 시작

exit() 호출
〉프로세스가 종료된다.

표 4-2 Visual Studio 6.0 에서 작성된 실행파일의 Stub 코드

Sample 01.exe도 Visual Studio 6.0의 C++ 환경에서 컴파일 했기 때문에 Stub 코드 구성이 동일합니다. 이를 토대로 다시 main() 호출 지점을 찾아가보겠습니다.

```
00401030 PUSH    EBP
00401031 MOV     EBP, ESP
00401033 PUSH    -1
00401035 PUSH    004050A8
0040103A PUSH    00401C8C
0040103F MOV     EAX, DWORD PTR FS:[0]
00401045 PUSH    EAX
00401046 MOV     DWORD PTR FS:[0], ESP
0040104D SUB     ESP, 10
00401050 PUSH    EBX
00401051 PUSH    ESI
00401052 PUSH    EDI
00401053 MOV     [LOCAL.6], ESP
00401056 CALL    DWORD PTR DS:[<&KERNEL32.GetVers kernel32.GetVersio
0040105C XOR     EDX, EDX

                 ......

004010CD MOV     DWORD PTR DS:[408528], EAX
004010D2 PUSH    EAX
004010D3 PUSH    DWORD PTR DS:[40851C]
004010D9 PUSH    DWORD PTR DS:[408518]
004010DF CALL    00401000
004010E4 ADD     ESP, 0C
004010E7 MOV     [LOCAL.7], EAX
004010EA PUSH    EAX
004010EB CALL    00401185
004010F0 MOV     EAX, [LOCAL.5]
004010F3 MOV     ECX, DWORD PTR DS:[EAX]
004010F5 MOV     ECX, DWORD PTR DS:[ECX]
004010F7 MOV     [LOCAL.8], ECX
004010FA PUSH    EAX
004010FB PUSH    ECX
004010FC CALL    0040125A
00401101 POP     ECX
00401102 POP     ECX
00401103 RETN
```

그림 4-4 Sample 01.exe의 StartUp 코드

사실 Sample 01.exe와 같이 행위의 특징이 바로 드러나는 파일은 메인함수를 찾기가 너무 쉽습니다. 코드를 하나씩 실행하다 보면 눈에 보이는 이벤트가 발생하는데, 그 부분이 메인함수 호출 지점입니다. 그리고 많은 파일들의 메인함수를 이런 식으로 찾을 수 있습니다.

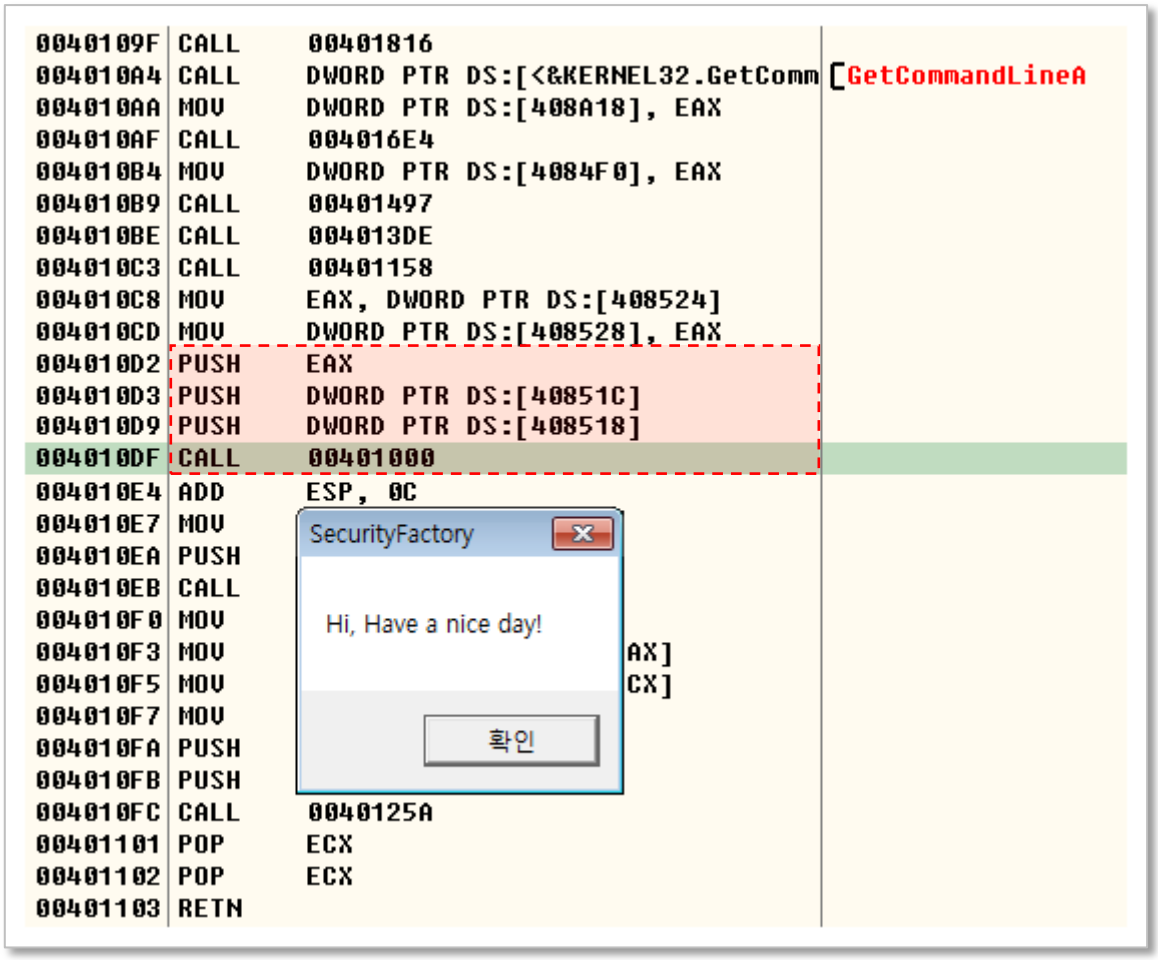


그림 4-5 main() 호출 지점

그렇다고 여기서 끝낼 순 없겠죠? 우리는 어떤 상황에서도 메인함수를 잘 찾을 수 있어야 하기에, 이벤트가 확인되지 않는다고 가정하겠습니다.

※ Note

블로그: http://bitly.kr/udWs
페이스북: http://bitly.kr/OrHQ

매번 강조하지만 분석에 정해진 방법은 없습니다. 다양한 상황에서 그에 맞는 방법을 찾고 활용하는 겁니다. 그렇다고 너무 걱정하진 마세요. 분석을 조금 하다 보면, '여기가 메인이겠구나!' 감이 옵니다.

리버싱 이 정도는 알아야지

발행일 | 2018 년 09 월
발행자 | SecurityFactory
페북 주소 | http://bitly.kr/OrHQ
이메일 | itseeyou@naver.com


SECURITY FACTORY
http://securityfactory.tistory.com