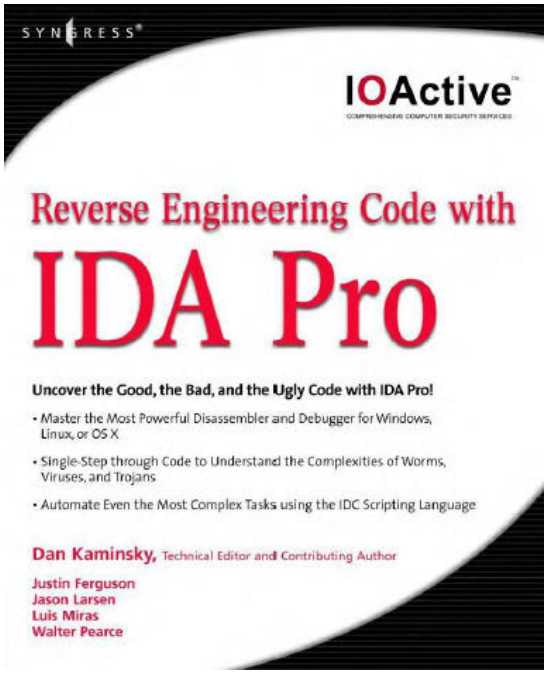
Reverse Engineering Code with IDA Pro



By Dan Kaminsky, Justin Ferguson, Jason Larsen, Luis Miras, Walter Pearce

정리: vangelis(securityproof@gmail.com)

이 글은 ***Reverse Engineering Code with IDA Pro***(2008년 출판)라는 책을 개인적으로 공부하면서 정리한 것입니다. 목적이 책 소개가 아니라 공부이므로 글의 내용은 형식에 국한되지 않고 경우에 따라 책의 내용 일부를 편역하거나 또는 철저하게 요약해서 올리고, 경우에 따라 필자가 내용을 추가할 것입니다. 내용에 오류 및 오타가 있다면 지적 부탁 드립니다.

2장 Assembly and Reversing Engineering Basics

개관[1]

이 장에서는 리버싱 엔지니어링을 위해 필요한 어셈블리어, Intel 아키텍처 프로세서, 명령어 등 기본적인 요소들을 소개하고 있다. 이 책은 32-bit Intel architecture(IA-32) 어셈블리어에 초점을 맞추고 있으며, 운영체계는 Windows와 Linux를 다루고 있다. 독자들은 IA-32 어셈블리어와 C/C++에 대한 이해도가 필요하다. 그렇다고 해서 이 장에서 관련 항목을 깊게 다루지는 않고 있다.

어셈블리어와 IA-32 프로세서[2]

어셈블러(assembler)들은 다른 문장 구조(syntax)를 가지고 있다. 가장 대표적인 Intel과 AT&T 문장 구조인데, 이 책에서는 Intel 문장 구조를 사용하고 있다. 이것은 IDA에 의해 사용되는 것이 Intel 문장 구조이기 때문인 것도 있고, 각종 문서나 white paper에서도 더 많이 사용되기 때문이다.

Intel과 AT&T 문장 구조의 차이점에 대해 다음 예를 보자. 다음은 간단한 "Hello, World" 프로그램이다. 첫 번째 것은 Red Hat 리눅스에서 gcc로 컴파일하고, gdb를 통해 디스어셈블링 한 것이다. Gdb를 통해 나오는 결과는 AT&T문장 구조이다.

```
(gdb) disas main
Dump of assembler code for function main:
0x8048328 <main>:      push   %ebp
0x8048329 <main+1>:    mov    %esp,%ebp
0x804832b <main+3>:    sub    $0x8,%esp
0x804832e <main+6>:    and    $0xfffffff0,%esp
0x8048331 <main+9>:    mov    $0x0,%eax
0x8048336 <main+14>:   sub    %eax,%esp
0x8048338 <main+16>:   sub    $0xc,%esp
0x804833b <main+19>:   push   $0x8048398
0x8048340 <main+24>:   call   0x8048268 <printf>
0x8048345 <main+29>:   add    $0x10,%esp
0x8048348 <main+32>:   mov    $0x0,%eax
```

---

[1] 2장에서는 시스템 해킹을 공부한 사람이라면 많이 접했던 분야이다. 그래서 아주 간단하게 정리를 할 생각이다.

[2] 이 섹션에서 "Assembly refers to the use of instruction mnemonics..."라는 부분이 나온다. 여기서 'mnemonics'는 같은 기능을 가진 instruction opcode 클래스의 예약된 이름이다. "mov eax, 0"라는 instruction의 포맷을 살펴보면, mov가 mnemocic이고, eax와 0은 operand가 되는 아규먼트들이다. 몇 개의 오퍼랜드를 가질 것인가는 opcode에 따라 결정되며, 오퍼랜드의 수는 0 ~3까지이다.

```
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
0x804834f <main+39>:    nop
End of assembler dump.
(gdb)
```

다음은 Dev-C++에서 컴파일 한 것을 IDA Pro로 본 것이다.

```
;
; +-------------------------------------------------------------------------+
; |    This file is generated by The Interactive Disassembler (IDA)       |
; |    Copyright (c) 2007 by DataRescue sa/nv, <ida@datarescue.com>       |
; | Licensed to: Mach EDV Dienstleistungen, Jan Mach, 1 user, adv, 11/2007|
; +-------------------------------------------------------------------------+
;
; Input MD5   : 88FEF2313AAAB6E089DCD55EF42C7804

; File Name   : C:\Documents and Settings\free2\바탕 화면\test.exe
; Format      : Portable executable for 80386 (PE)
; Imagebase   : 400000
; Section 1. (virtual address 00001000)
; Virtual size                : 000008D4 (   2260.)
; Section size in file        : 00000A00 (   2560.)
; Offset to raw data for section: 00000400
; Flags 60000060: Text Data Executable Readable
; Alignment     : default
; OS type        :  MS Windows
; Application type:  Executable 32bit

.686p
.mmx
.model flat
.intel_syntax noprefix

- 중략 -

push    ebp
mov     ebp, esp
sub     esp, 8
```

```
and     esp, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    ___chkstk
call    ___main
mov     [esp+8+var_8], offset aHello ; "Hello"
call    printf
mov     eax, 0
leave
retn
_main endp
```

- 중략 -

결과를 보면 파란색으로 표시한 **.intel_syntax noprefix** 부분을 보면 Intel 문장 구조를 사용하고 있음을 알 수 있다. printf() 함수가 호출되는 부분부터 두 문장 구조를 비교해보자.

[AT&T 문장구조]

```
0x8048340 <main+24>:    call   0x8048268 <printf>
0x8048345 <main+29>:    add    $0x10,%esp
0x8048348 <main+32>:    mov    $0x0,%eax  //책에서는 xor %eax, %eax
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
```

[Intel 문장구조]

```
mov     [esp+8+var_8], offset aHello ; "Hello"
call    printf
mov     eax, 0   //책에서는 xor eax, eax
leave
retn
```

책에서는 "xor %eax, %eax"를 사용하고 있다. 이 연산 결과는 잘 알겠지만 eax 레지스터의 값

이 0이 된다. xor를 사용하고, 두 오퍼랜드가 같으면 그 오퍼랜드의 값은 0이 된다. 그리고 "mov $0x0,%eax"의 결과를 보면 eax 레지스터에 0의 값을 복사한다. 결과는 당연 eax 레지스터의 값은 0이 된다. 책에 나오는 것과 필자의 테스트에서 나온 결과는 같은 것이다.

우리가 주목할 것은 위의 두 결과를 보면 source operand와 destination operand의 위치가 다르다는 것이다. 정리하면 다음과 같다.

* AT&T 문장구조 – **명령어 source operand, destination operand**

```
mov     $0x0,       %eax
```

* Intel 문장구조 – **명령어 destination operand, source operand**

```
mov     eax,        0
```

다음 부분은 C언어에서는 "return 0;"을 나타낸다.

```
mov    eax, 0   // mov    $0x0,%eax
leave
retn            // ret
```

프로세서는 'return 0;'이라는 것을 이해할 수 없다. 그런 의미에서 어셈블리어는 인간이 읽을 수 있는 마지막 코드 층이라고 할 수 있다. 컴파일링 후에 코드를 어셈블링하면 'opcode'를 출력하는데, opcode는 어셈블리어 명령을 이진수 형태로 나타낸 것 또는 개별 명령을 실행하기 위해 필요한 on-off의 연속(sequence)이다. 보통 opcode는 이진수 보다는 16진수로 나타낸다.

```
0x8048348 <main+32>:    mov    $0x0,%eax
0x804834d <main+37>:    leave
0x804834e <main+38>:    ret
0x804834f <main+39>:    nop
End of assembler dump.
(gdb) x/8b 0x8048348
0x8048348 <main+32>:    0xb8    0x00    0x00    0x00    0x00    0xc9    0xc3    0x90
(gdb)
```

이것을 objdump를 이용해 좀더 알기 쉽게 확인할 수 있다.

```
8048348:      b8 00 00 00 00       mov    $0x0,%eax
804834d:      c9                   leave
804834e:      c3                   ret
804834f:      90                   nop
```

하지만 결국 컴퓨터의 프로세서에 전달되는 것은 1과 0의 연속이다. Opcode에 대한 좀더 자세한 자료는 ***Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A***[3]의 **section 2.1.2**를 참고하길 바란다.[4] 이 책은 http://www.intel.com/products/processor/manuals/에서 pdf 파일로 구할 수 있으며, 책으로 받으려면 http://www.intel.com/design/literature.htm를 참고하면 된다. 참고로 이 책들은 무료이다. 신청하면 늦어도 2주 안으로 미국에서 한국으로 책이 배달된다. 필자의 경우 매뉴얼 페이지에 있는 6 종류의 책을 신청하여 2개의 박스로 받았던 적이 있다.

다음은 **섹션 2.1.2**의 원문 전체를 첨부한 것이다.

****

2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte, or
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet)

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix for SSE/SSE2/SSE3 instructions (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes, or
- A mandatory prefix (66H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet)

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

****

[3] http://download.intel.com/design/processor/manuals/253666.pdf

[4] 어떤 책을 공부하다 보면 각종 책이나 참고 자료들이 제시된다. 공부과정에서 그 참고 자료들도 같이 구해서 본다면 공부하고 있는 책 한 권을 제대로 이해할 수 있을 뿐만 아니라 더 많은 것을 이해할 수 있게 될 것이다.

opcode에 대한 더 자세한 내용은 위의 첨부한 원문 끝 줄에 나오듯 Appendix A와 B에 나와 있다. 그리고 ***Reversing Engineering Code with IDA Pro***라는 이 책을 잘 이해하기 위해서는 ***Intel® 64 and IA-32 Architectures Software Developer's Manual*** Volume 2*A, 2B* 두 권이 늘 필요할 것이다. 이 두 권은 instruction set reference에 대한 것이다.

Tools & Traps…

Shellcode는 보통 C언어에서 문자열 배열로 저장되는 일련의 opcode이다. 'shellcode'라는 용어를 사용하게 된 것은 그 일련의 opcode가 쉘(/bin/sh 또는 cmd.exe)을 실행하는데 필요한 명령어(instruction)들이기 때문이다.

C언어에서 쉘코드를 사용하려면 다음과 같이 한다는 것을 이 글의 독자들은 다 알고 있을 것이다.

unsigned char shellcode[] = "\xc9\x31\xc0\xc9";

쉘코드는 해킹에서 필수적인 요소이므로 관련 지식을 반드시 알아둘 필요가 있다. 쉘코드를 만드는 방법에 대해서는 국내외에 많은 자료들이 있으므로 그 자료들을 참고하길 바란다.

To be continued…