# Reverse Engineering For Beginners



Univ.Chosun & HackerLogin & CERT : Choi Young Mo
Email : mdjdwjm3@nate.com

# - 목 차 -

# 1. RCE란?

일반적으로 리버싱하면 Reverse Engineering이라고만 알고 있지만, 의미적인 Full name을 이야기 하자면 RCE(Reverse Code Engineering)가 된다. RCE는 여러 분야에서 사용되는 용어로써 인공물로부터 설계의 사상이나 지식을 추출해 내는 행위를 말하는 것으로 지금부터 다룰 내용은 소프트웨어에 대한 RCE에 대해서 설명할 것이다. 쉽게말하면 소스 코드를 컴파일하면 컴퓨터가 이해하는 바이너리 코드가 생성되는데 이렇게 생성된 바이너리 코드로부터 소스 코드를 유출해내는 기술을 말한다. RCE에서 가장 많이 사용되는 방식은 첫 번째로 이야기 했던 바이너리를 디스어셈블 하여 코드를 얻어내는 것이다. 이것을 하기 위해서는 먼저 인텔 어셈블리를 배워야 하고, 물론 C언어도 알아야 된다.

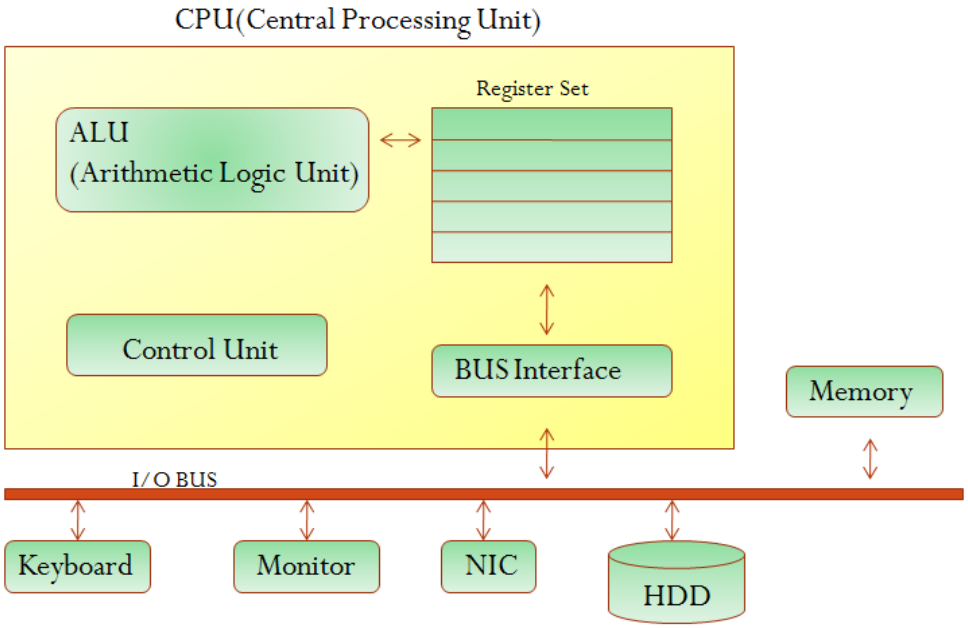
내가 공부할 때 참고로 했던 문헌에서 이런 글이 있어 적어본다. RCE을 잘 하려면 중학교 수학책에 나오는 인수분해를 잘해야 한다. 물론 사실 인수분해를 잘할 필요는 없다. 어떻게 하면 인수분해를 잘할 수 있는지를 아는 것이 중요하다. 같은 과정을 RCE에 적용하면 득도의 경지에 도달할 수 있다고 감히 말할 수 있다. 인수 분해란 나열된 수식을 정리해서 간단한 인수의 곱으로 바꾸는 것을 말한다. x^2 + 2xy + y^2과 같은 수식을 (x+y)^2으로 나타내는 것을 말한다. 처음 인수 분해를 배우는 학생들은 당황한다. 늘 해왔던 것과는 반대로 진행되기 때문이다.

하지만 신기한 사실은 두, 세 시간의 수업 후에는 대부분의 학생들이 능숙하게 인수 분해를 해낸다는 점이다. 이는 RCE과는 대조적인 차이라 할 수 있다. 어떻게 그렇게 됐을까? 답은 간단하다 10가지 안팎의 공식에 있다. 교과서에는 인수 분해에 대한 설명과 함께 암기해야 할 필수 공식이 나온다. 이 공식들은 인수 분해에 자주 사용되는 것들로 이것들을 암기하고 나면 대부분의 역행 패턴을 쉽게 유추할 수 있는 것이다.

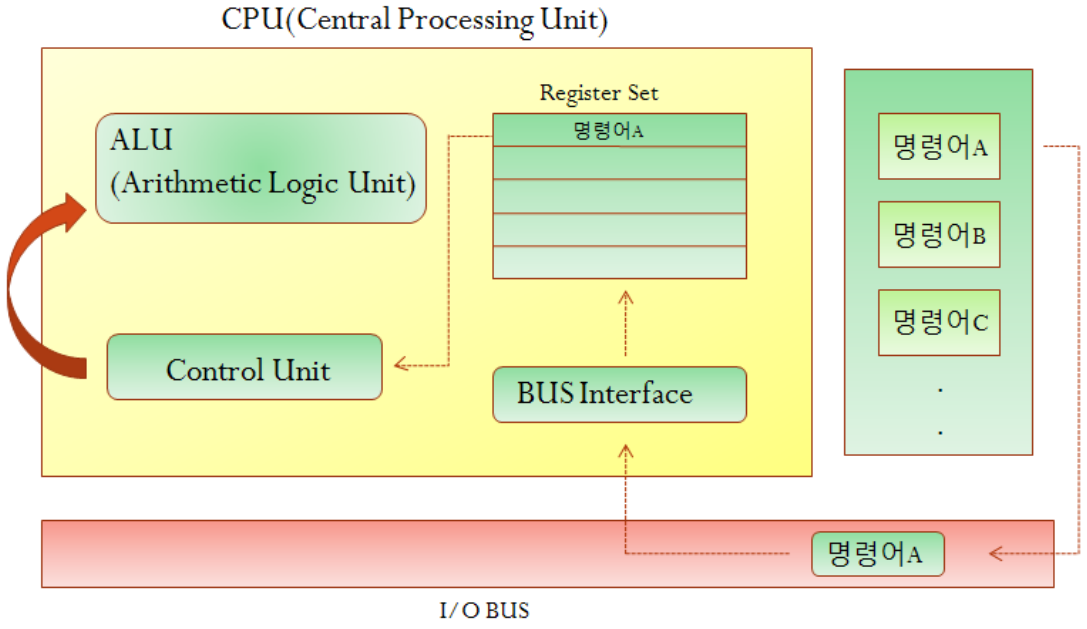
RCE도 마찬가지다. 개발자들이 프로그램을 작성하면서 사용하는 코드는 한정적이고, 여기에는 일정한 패턴이 존재한다. 이렇게 자주 사용하는 코드들의 변환 결과와 패턴을 모두 암기하고 있다면 변환 결과로부터 원 소스를 유추해 내는 과정은 그리 어렵지 않다. 실제로 유능한 Reverser들은 어셈블리 코드를 보면 바로 C 코드가 떠오를 정도로 기계적이라고 한다(보통 중국 애들). 개발자들이 많이 사용하는 코드도 그 결과를 어떻게 보는지도 모르겠다고 생각할 수도 있다. 하지만 걱정할 필요가 없다. 우리보다 먼저 그 길을 간 사람들이 우리를 위해서 잘 정리해 둔 결과물이 있기 때문이다.

# 2. RCE 기초

## (1) CPU 동작방식의 이해



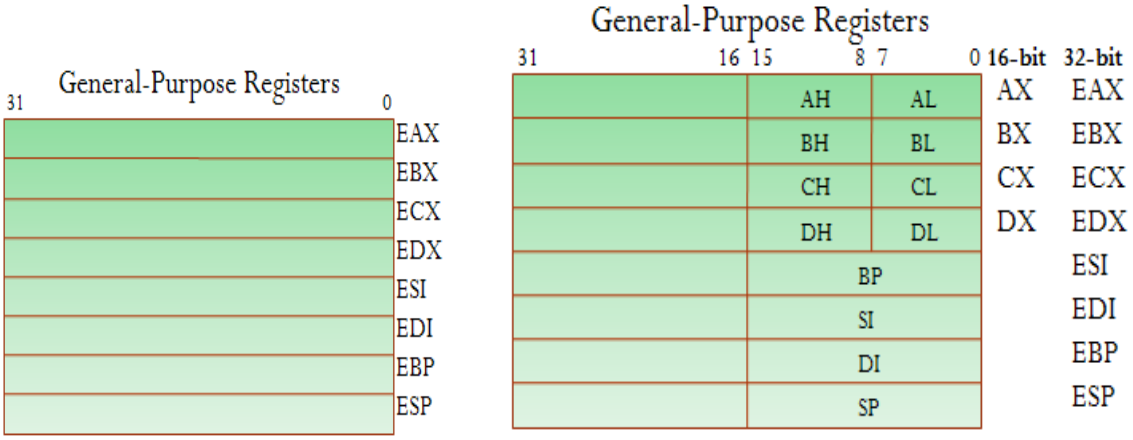

CPU는 소프트웨어에서 명령의 실행이 이루어지는 컴퓨터의 일부분 혹은 그 기능을 내장한 칩을 말하며, CPU는 기계어로 쓰여진 컴퓨터 프로그램의 명령어를 해석하여 실행한다. 프로그램에 따라 외부에서 정보를 입력, 기억, 연산하고, 외부로 출력한다. CPU의 기본구성은 레지스터, 산술논리연산장치(ALU:arithmetic logic unit), 제어부(control unit)와 내부 버스 등이 있다. 프로그램이 실행되면 실행 프로그램이 메모리에 적재되고 메모리의 내용을 CPU에서 가져와 레지스터에 저장한 후 Control Unit을 통해 ALU에 전달된 후에 ALU에서 산술 연산을 하게 된다. ALU는 논리연산까지도 하게 되는데 중앙처리장치의 기본설계 블록이다. 우리의 뇌에 해당하는 부분이다.




Assembly언어로 이루어진 명령어들을 Memory에서 CPU로 LOAD와 STORE를 반복하면서 register에 저장된다. 이때 Control Unit은 명령어를 실행(Excution)가능하도록 Decoding 해주는 역할을 한다.

## (2) CPU Register

간단하게 CPU Register의 구조를 알아보도록 하자.



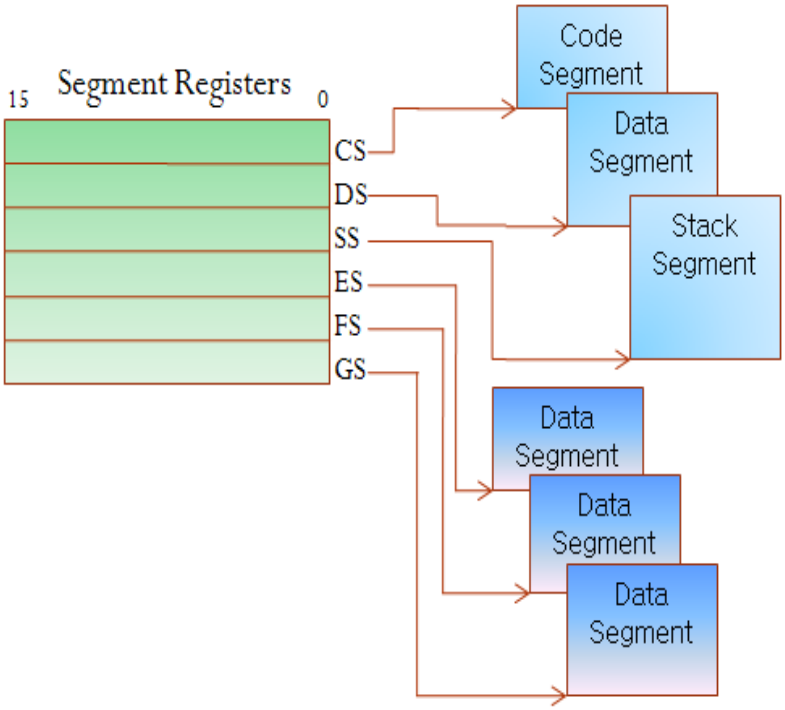

위의 그림은 범용레지스터를 도식해 놓은 것이다. 기본적으로 32비트(4바이트)의 크기를 갖고 있으며 16비트나 8비트 형태로 나누어서 사용되기도 한다. 범용 레지스터는 논리, 산술 연산에 사용되는 오퍼랜드(피연산자)나 주소계산을 위한 오퍼랜드, 메모리 포인터에 사용된다. 레지스터의 종류는 크게 General-Purpose, Segment, EFLAGS, EIP Register 이렇게 네가지로 구분된다.

범용으로 사용되는 레지스터의 역할은 다음과 같다.

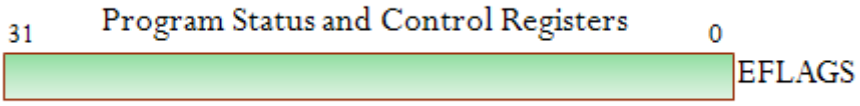
·**EAX(Extended Accumulator Register)** - 산술연산에 사용(함수의 결과값을 저장)
·**EBX(Extended Base Register)** - DS 세그먼터에 데이터를 가리키는 역할
(주소지정을 확대하기 위한 인덱스로 사용)
·**ECX(Extended Counter Register)** - 루프의 반복 횟수나 좌우방향 시프트 비트 수 기억
·**EDX(Extended Data Register)** - 입출력 동작에서 사용
·**ESI(Extended Source Index)** - 출발지 인덱스에 대한 값 저장
·**EDI(Extended Destination Index)** - 다음 목적지 주소에 대한 값 저장
·**ESP(Extended Stack Pointer)** - Stack 포인터, Stack의 TOP을 가리킴
·**EBP(Extended Base Pointer)** - Stack 내의 변수값을 읽는데 사용, Stack 프레임의 시작점을 가리킴

이중에 잘알아두어야 할 것은 ESP, EBP이다. ESP는 함수를 사용시에 하나하나의 스택포인터로 사용되며, EBP는 스택내의 가장 바닥에 해당되는 레지스터를 가리키는 포인터이므로 항상 가장먼저 알아보고 알아두어야 하는 레지스터이다.

아래는 Segment Register 이다. 16비트의 크기를 가지고 있으며, 세그먼터 선택자는 메모리에서 세그먼트를 확인하는 특별한 포인터의 역할을 한다. 각 세그먼트들은 주소의 시작점을 가리키고 있다.



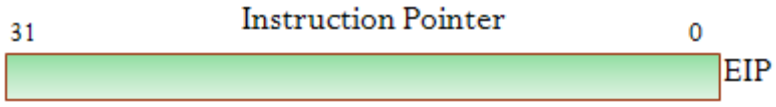

- CS : 코드 세그먼트의 시작 주소를 저장
  - 프로그램에서 명시적으로 로드 될 수는 없으나 프로그램 제어를 변경하는 명령 또는 내부 프로세서 조작에 의해 묵시적으로 로드 될 수 있다.
- SS : 스택으로 사용할 메모리 영역의 시작 주소를 저장
  - 명시적으로 로드 될 수 있기 때문에 프로그램에서 이 레지스터를 통하여 여러개의 스택을 셋업하여 그들 간을 상호 전환할 수 있다.
- DS : 데이터 세그먼터의 시작주소를 저장
- ES : 여분의 데이터 세그먼트용, 문자열 처리 명령시 목적지 데이터 저장
- FS / GS : 여분의 세그먼트용, IA32 프로세서에서만 사용이 가능하다.




위의 EFLAGS Register는 상태 플러그, 제어 플래그, 시스템 플래그 그룹으로 구성되어 있으며 1, 3, 5, 15, 22~31 비트는 이미 예약되어 있으므로 사용이 불가능하다. EFLAGS Register의 내

용은 비트 조작 명령(BT, BTS, BTR, BTC)을 사용하여 플래그를 검사 및 변경할 수 있다. 이 레지스터 안에 들어있는 플래그들 중에 RCE에 필요한 플래그는 0, 6, 11번 비트로 CF, ZF, OF이다. 이 플래그들은 Status Flags로 산술연산을 하는 명령의 결과값을 가리킨다.

- CF(Carry flag) : 부호 없는 정수끼리의 산술 연산 후 오버플로우가 발생했는지 확인
  - 0 : 오버플로우가 발생하지 않은 경우
  - 1 : 오버플로우가 발생한 경우
- ZF(Zero flag) : 산술연산 또는 비교동작의 결과를 나타냄
  - 0 : 결과가 0이 아닌 경우
  - 1 : 결과가 0인 경우
- OF(Overflow flag) : 부호있는 수끼리의 산술 연산 후 오버 플로우가 발생했는지 확인
  - 0 : 오버 플로우가 발생하지 않은 경우
  - 1 : 오버 플로우가 발생한 경우(너무 큰 정수나 너무 작은 음수인 경우)




EIP Register는 현재 프로세서가 실행하고 있는 명령 바로 다음에 실행할 명령어의 오프셋을 저장하게 된다. 수행한 명령어의 길이만큼 증가된 **EIP는 Memory내의 다음 실행할 명령어를 가리킨다.(중요!)** 실제 모드로 동작할 때는 16비트 EIP로, 보호 모드로 동작할 때는 32비트 EIP로 동작하게 된다. EIP는 소프트웨어에 의해서 조작이 불가능 하고 CALL, JMP, RET와 같은 control-transfer 명령에 의해서만 영향을 받는다. 여기서 CALL과 JMP를 잠깐 비교해보면 다음과 같다.

◎ EIP 백업 차이
- CALL : EIP 값을 변경 전에 Stack에 백업 후에 변경한다.
- JMP : EIP 값을 변경 시에 백업 없이 바로 변경한다.

◎ 조건 분기
- CALL : 조건 분기가 가능한 명령어 형태가 존재하지 않는다.
- JMP : 조건 분기가 강한 명령어 형태가 존재한다. 즉, 조건을 줄 수 있는냐 없느냐의 차이를 보인다.

## (3) Assembly 언어 맛보기

어셈블리어의 명령어는 아래와 같이 크게 6가지로 나눌 수 있는데,

◉ Arithmetic Instruction
◉ Data Transfer Instruction
◉ Logical Instruction
◉ String Instruction
◉ Control Transfer Instruction
◉ Processor Control Instruction

| Arithmetic instruction | | |
|---|---|---|
| 명령어 | 설 | 명 |
| ADD | Add | 캐리를 포함하지 않은 덧셈 |
| SBB | Subtract with Borrow | 캐리를 포함한 뺄셈 |
| DEC | Decrement | 오퍼랜드 내용을 1 감소 |
| NEG | Change Sign | 오퍼랜드의 2의 보수, 즉 부호 반전 |
| CMP | Compare | 두 개의 오퍼랜드를 비교한다 |
| ADC | Add with Carry | 캐리를 포함한 덧셈 |
| INC | Increment | 오퍼랜드 내용을 1 증가 |
| AAA | ASCII adjust for Add | 덧셈 결과 AL값을 UNPACK 10진수로 보정 |
| DAA | Decimal adjust for Add | 덧셈 결과의 AL값을 PACK 10진수로 보정 |
| SUB | Subtract | 캐리를 포함하지 않은 뺄셈 |
| AAS | ASCII adjust for Subtract | 뺄셈 결과 AL값을 UNPACK 10진수로 보정 |
| DAS | Decimal adjust for Subtract | 뺄셈 결과의 AL값을 PACK 10진수로 보정 |
| MUL | Multiply (Unsigned) | AX와 오퍼랜드를 곱셈하여 결과를 AX 또는 DX:AX에 저장 |
| IMUL | Integer Multiply (Signed) | 부호화된 곱셈 |
| AAM | ASCII adjust for Multiply | 곱셈 결과 AX값을 UNPACK 10진수로 보정 |
| DIV | Divide (Unsigned) | AX 또는 DX:AX내용을 오퍼랜드로 나눔, 몫은 AL, AX 나머지는 AH, DX로 저장 |
| IDIV | Integer Divide (Signed) | 부호화된 나눗셈 |
| AAD | ASCII adjust for Divide | 나눗셈 결과 AX값을 UNPACK 10진수로 보정 |
| CBW | Convert byte to word | AL의 바이트 데이터를 부호 비트를 포함하여 AX 워드로 확장 |
| CWD | Convert word to double word | AX의 워드 데이터를 부호를 포함하여 DX:AX의 더블 워드로 변환 |

| Data Transfer Instruction | | |
|---|---|---|
| 명령어 | 설 | 명 |
| MOV | Move | 데이터 이동(전송) |
| PUSH | Push | 오퍼랜드의 내용을 스택에 쌓는다 |
| POP | Pop | 스택으로부터 값을 뽑아낸다. |
| XCHG | Exchange Register/memory with Register | 첫 번째 오퍼랜드와 두 번째 오퍼랜드 교환 |
| IN | Input from AL/AX to Fixed port | 오퍼랜드로 지시된 포트로부터 AX에 데이터 입력 |
| OUT | Output from AL/AX to Fixed port | 오퍼랜드가 지시한 포트로 AX의 데이터 출력 |
| XLAT | Translate byte to AL | BX:AL이 지시한 데이블의 내용을 AL로 로드 |
| LEA | Load Effective Address to Register | 메모리의 오프셋값을 레지스터로 로드 |
| LDS | Load Pointer to DS | REG←(MEM), DS←(MEM+2) |
| LES | Load Pointer ti ES | REG←(MEM), ES←(MEM+2) |
| LAHF | Load AH with Flags | 플래그의 내용을 AH의 특정 비트로 로드 |
| SAHF | Store AH into Flags | AH의 특정 비트가 플래그 레지스터로 전송 |
| PUSHF | Push Flags | 플래그 레지스터의 내용을 스택에 쌓음 |
| POPF | Pop Flags | 스택으로부터 플래그 레지스터로 뽑음 |

| Control Transfer Instruction | | | |
|---|---|---|---|
| 명령어 | 설 | 명 | 플래그의 변화 |
| CALL | Call | 프로시저 호출 | |
| JMP | Unconditional Jump | 무조건 분기 | |
| RET | Return from CALL | CALL로 스택에 PUSH된 주소로 복귀 | |
| JE/JZ | Jump on Equal / Zero | 결과가 0이면 분기 | ZP=1 |
| JL/JNGE | Jump on Less /not Greater or Equal | 결과가 작으면 분기 (부호화된 수) | SP != OP |
| JB/JNAE | Jump on Below /not Above or Equal | 결과가 작으면 분기 (부호화 안 된 수) | CP=1 |
| JBE/JNA | Jump on Below or Equal /not Above | 결과가 작거나 같으면 분기 (부호화 안 된 수) | CP=1 or ZP=1 |
| JP/JPE | Jump on Parity / Parity Even | 패리티 플레그가 1이면 분기 | PP=1 |
| JO | Jump on Overflow | 오버플로가 발생하면 분기 | OP=1 |
| JS | Jump on Sign | 부호 플레그가 1이면 분기 | SP=1 |
| JNE/JNZ | Jump on not Equal / not Zero | 결과가 0이 아니면 분기 | ZP=0 |
| JNL/JGE | Jump on not Less /Greater or Equal | 결과가 크거나 같으면 분기 (부호화된 수) | SP=OP |
| JNLE/JG | Jump on not Less or Equal /Greater | 결과가 크면 분기 (부호화된 수) | ZP=0 and SP=OP |
| JNB/JAE | Jump on not Below /Above or Equal | 결과가 크거나 같으면 분기 (부호화 안 된 수) | CP=0 |
| JNBE/JA | Jump on not Below or Equal /Above | 결과가 크면 분기 (부호화 안 된 수) | CP=0 and ZP=0 |
| JNP/JPO | Jump on not Parity / Parity odd | 패리티 플레그가 0이면 분기 | PP=0 |
| JNO | Jump on not Overflow | 오버플로우가 아닌 경우 분기 | OP=0 |
| JNS | Jump on not Sign | 부호 플레그가 0이면 분기 | SP=0 |
| LOOP | Loop CX times | CX를 1감소하면서 0이 될 때까지 지정된 라벨로 분기 | |
| LOOPZ/LOOPE | Loop while Zero / Equal | 제로 플레그가 1이고 CX≠0이면지정된 라벨로 분기 | ZP=1 |
| LOOPNZ/LOOPNE | Loop while not Zero / not Equal | 제로 플레그가 0이고 CX≠0이면지정된 라벨로 분기 | ZP=0 |
| JCXZ | Jump on CX Zero | CX가 0이면 분기 | CX=0 |
| INT | Interrupt | 인터럽트 실행 | |
| INTO | Interrupt on Overflow | 오버플로우가 발생하면 인터럽트 실행 | |
| IRET | Interrupt Return | 인터럽트 복귀 (리턴) | |

| Logical Instruction | | |
|---|---|---|
| 명령어 | 설 | 명 |
| NOT | Invert | 오퍼랜드의 1의 보수, 즉 비트 반전 |
| SHL/SAL | Shift logical / arithmetic Left | 왼쪽으로 오퍼랜드만큼 자리 이동 (최하위 비트는 0) |
| SHR | Shift logical Right | 오른쪽으로 오퍼랜드만큼 자리 이동 (최상위 비트 0) |
| SAR | Shift arithmetic Right | 오른쪽 자리이동, 최상위 비트는 유지 |
| ROL | Rotate Left | 왼쪽으로 오퍼랜드만큼 회전 이동 |
| ROR | Rotate Right | 오른쪽으로 오퍼랜드만큼 회전 이동 |
| RCL | Rotate through Carry Left | 캐리를 포함하여 왼쪽으로 오퍼랜드만큼 회전 이동 |
| RCR | Rotate through Carry Right | 캐리를 포함하여 오른쪽으로 오퍼랜드만큼 회전 이동 |
| AND | And | 논리 AND |
| TEST | And function to Flags, no result | 첫 번째 오퍼랜드와 두 번째 오퍼랜드를 AND하여 그 결과로 플래그 세트 |
| OR | Or | 논리 OR |
| XOR | Exclusive Or | 배타 논리 합 (OR) |

| String Manipulation Instruction | | |
|---|---|---|
| 명령어 | 설 | 명 |
| REP | Repeat | REP 뒤에 오는 스트링 명령을 CX가 00이 될 때까지 반복 |
| MOVS | Move String | DS:SI가 지시한 메모리 데이터를 ES:DI가지시한 메모리로 전송 |
| CMPS | Compare String | DS:SI와 ES:DI의 내용을 비교하고 결과에 따라 플래그 설정 |
| SCAS | Scan String | AL 또는 AX와 ES:DI가 지시한 메모리 내용 비교하고 결과에 따라 플래그 설정 |
| LODS | Load String | SI 내용을 AL 또는 AX로 로드 |
| STOS | Store String | AL 또는 AX를 ES:DI가 지시하는 메모리에 저장 |

| Processor Control Instruction | | |
|---|---|---|
| 명령어 | 설 | 명 |
| CLC | Clear Carry | 캐리 플레그 클리어 |
| CMC | Complement Carry | 캐리 플레그를 반전 |
| CLD | Clear Direction | 디렉션 플레그를 클리어 |
| CLI | Clear Interrupt | 인터럽트 플레그를 클리어 |
| HLT | Halt | 정지 |
| LOCK | Bus Lock prefix | |
| STC | Set Carry | 캐리 플레그 셋 |
| NOP | No operation | |
| STD | Set Direction | 디렉션 플레그 셋 |
| STI | Set Interrupt | 인터럽트 인에이블 플레그 셋 |
| WAIT | Wait | 프로세서를 일지 정지 상태로 한다 |
| ESC | Escape to External device | 이스케이프 명령 |

위는 어셈블리어의 명령어들을 간단히 모아서 표로 만들어 보았고 표에서 초록색바탕이 된 부분은 기본적으로 알아두어야 하며 많이 쓰이는 명령어 들이다. 간단한 팁을 몇 개 말해 보자면 코드를 살펴보다가 CMP와 TEST구문이 같이 보이면 JMP와 연계가 많이 된다는 것을 알수 있고, AND명령은 결과값을 저장하는 반면에 TEST명령어는 결과값을 저장하지 않는다. 그리고 프로그래머가 PUSH 명령을 사용하는 경우는 argument를 전달시(EBP +N), 지역변수를 할당할시(EBP -N) 그리고 마지막으로 백업할 때이므로 잘 살펴봐야한다.

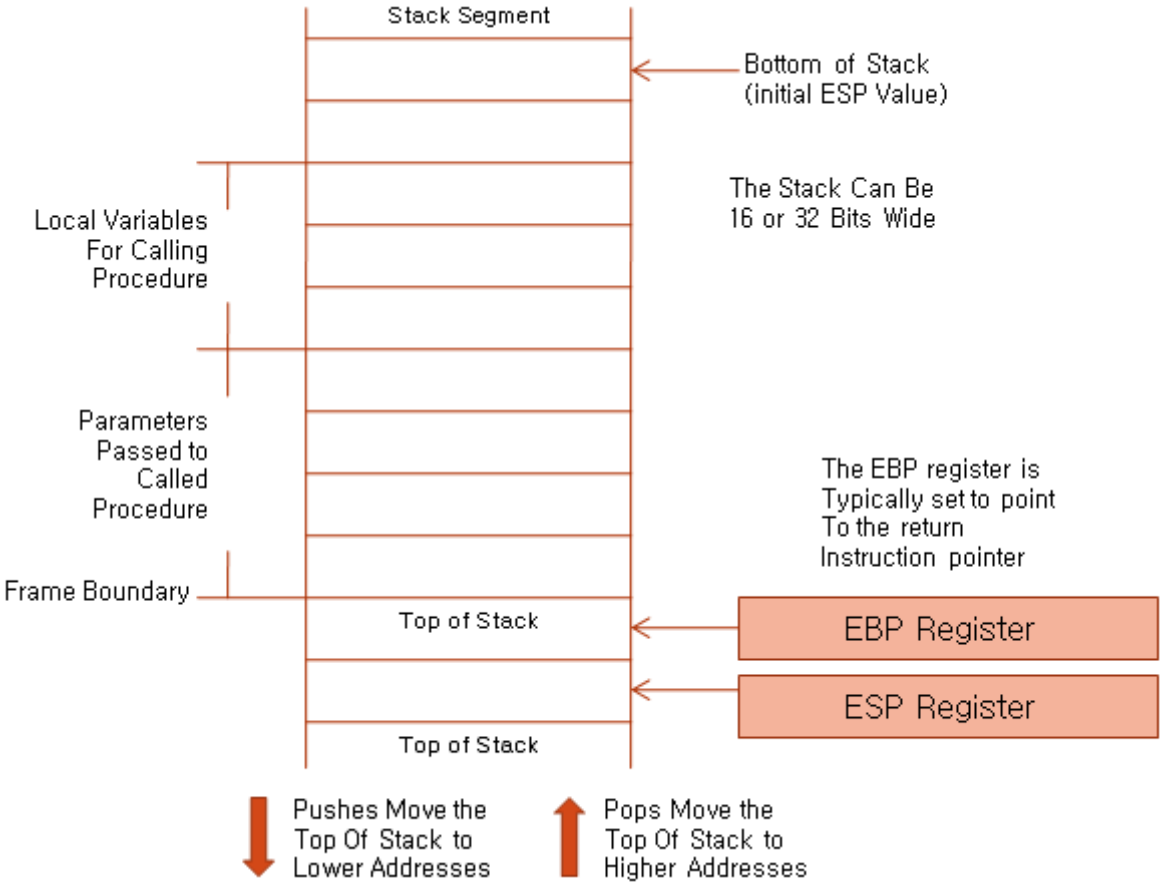
## (4) STACK의 구조

스택은 한 쪽 끝에서만 데이터를 넣거나 뺄 수 있는 선형 구조로 되어있다. 스택의 공간에 지역변수를 할당하게 되며 함수가 실행 시에 동작한다. 자료를 넣는 것을 PUSH라 하며 자료를 꺼내는 것을 POP이라고 하는데 이 때 꺼내어 지는 데이터는 가장 최근에 넣은 데이터가 나오게 된다. LIFO(Last In First Out)의 형태를 갖는 구조이다.

스택을 구현 방법에 따라 네 가지로 구분하면 다음과 같다.

◉ Full Stack : TOP이 마지막으로 PUSH된 데이터를 가리킴
◉ Empty Stack : TOP이 다음 데이터가 들어올 곳을 가리킴
◉ Ascending Stack : 낮은 메모리 주소에서 시작하여 높은 메모리 주소 방향으로 저장
◉ Descending Stack: 높은 메모리 주소에서 시작하여 낮은 메모리 주소 방향으로 저장

우리가 흔히 사용하는 인텔의 메모리는 Full Descendign Stack의 형태를 가지고 있음을 알아야 한다. 즉, 스택의 TOP이 마지막으로 들어온 데이터를 가리키고 있으며 높은 메모리 주소에서 시작하여 낮은 메모리 주소 방향으로 자란다는 뜻이 된다.

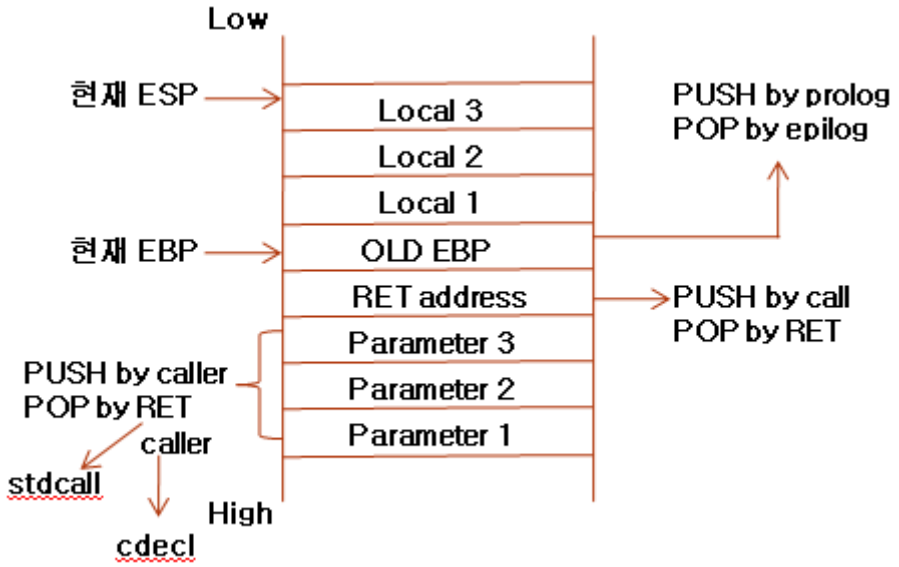



EBP는 스택 프레임의 시작점을 가리키고 함수 내에서 ESP를 통하여 스택의 크기를 늘리고 줄일 때 기존의 ESP값을 백업한다. ESP는 스택의 TOP을 가리키는데 스택 포인터 이동시에 사용된다. EIP는 다음에 실행할 명령의 오프셋(주소값)을 가지고 있는데 보통 함수 호출 후 복귀 주소를 나타내는 레지스터로써 소프트웨어에 의해 조작될 수는 없다고 앞서 말한바 있다.

## (5) Calling Convention

함수의 호출규약은 Argument 전달 방법과 Argument 전달 순서, Stack Clearing 방법 그리고 Return Value하는 방법이 있다. 각 함수는 실행할 함수의 코드 위치를 갖고 있는 포인터나 그 값을 가져올 수 있는 함수의 이름, 현재 처리하고 있는 함수의 정보를 저장할 공간, 함수의 실행이 끝난 뒤에 리턴 값을 돌려받을 공간, 함수의 실행에 필요한 인자를 넘겨줄 공간 등이 필요하다. 이런 것들을 쉽게 마련하고자 함수 호출 규약을 정해둔 것이라고 생각하면 되겠다.

주요 함수 호출 규약에는 __stdcall, __cdecl, __fastcall 이렇게 3가지가 있다. stdcall 호출 규약의 경우 호출된 함수(callee)가 스택을 정리하기 때문에 호출하는 함수(Caller)와 Callee모두 파라미터의 크기를 알고 있어야 정상적인 처리가 가능하지만 cdecl호출 규약의 경우 Caller가 스택을 정리하기 때문에 Callee는 파라미터의 크기를 정확히 몰라도 되는 장점이 있다.

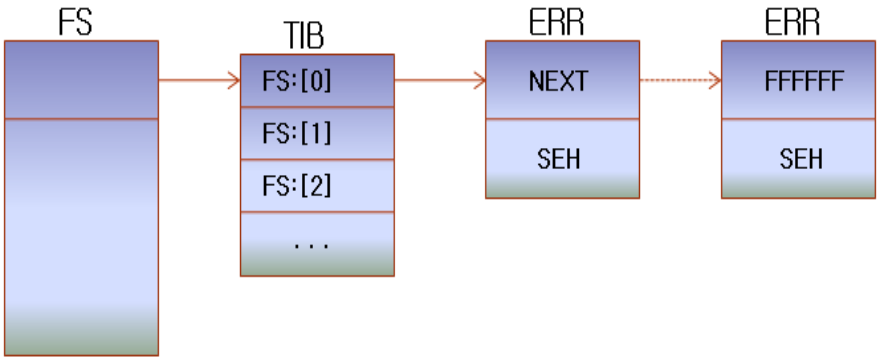



함수의 프롤로그와 에필로그의 형식을 나타내는 명령어들의 조합은 다음과 같은 순서를 많이 띄게 된다.

◉ 프롤로그 : PUSH EBP
MOV ESP. EBP
◉ 에필로그 : MOV ESP. EBP
POP EBP

### (6) SEH(Structured Exception Handling)

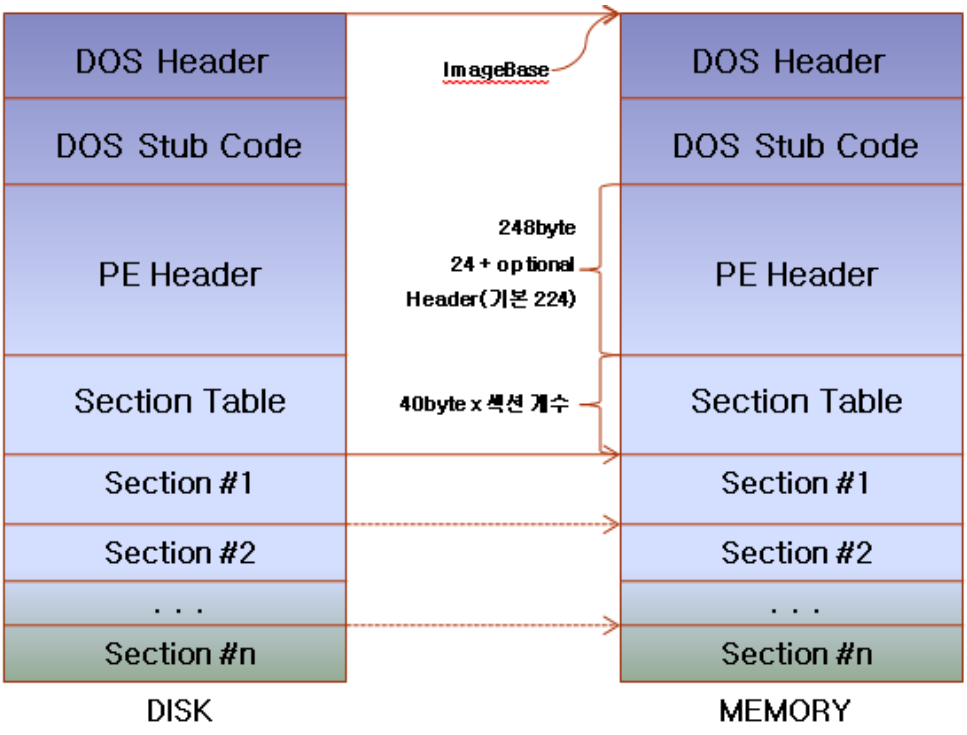
SEH(Structured Exception Handling)은 윈도우에서 제공하는 예외 처림 방식인데 예외를 발생시키는 상황은 여러 가지가 있지만 대표적으로 0으로 나누거나 숫자형식의 오버플로우 라든지, 잘못된 메모리 주소에 대한 읽기나 쓰기를 시도했을 때이다. 예외 발생시에 TIB에서 가리키고 있는 주소에서 예외를 처리하는데 TIB는 쓰레드에 대한 중요한 정보를 가지고 있는 구조체이다. 그 때문에 이 구조체의 구조는 아래와 같이 추상해 볼 수 있다.




# 3. PE 파일구조

### (1) PE 파일포맷

PE 파일 형식(Portable Executable File Format)이란 파일(File)에 담겨 다른 곳에 옮겨져도 실행 시킬 수 있도록 규정한 형식이란 뜻이다. Win32의 기본적인 파일 형식이며 윈도우 운영체제에서 실행되는 프로그램이 모두 PE파일의 형식을 가지고 있다. EXE파일이 PE파일의 대표적인 예이다. 그리고 동적 링크 라이브러리인 DLL파일도 PE 파일의 형식을 가지고 있다. 그러므로 PE 파일의 구조를 알아야만 RCE를 능숙하게 할 수 있는 것이다.

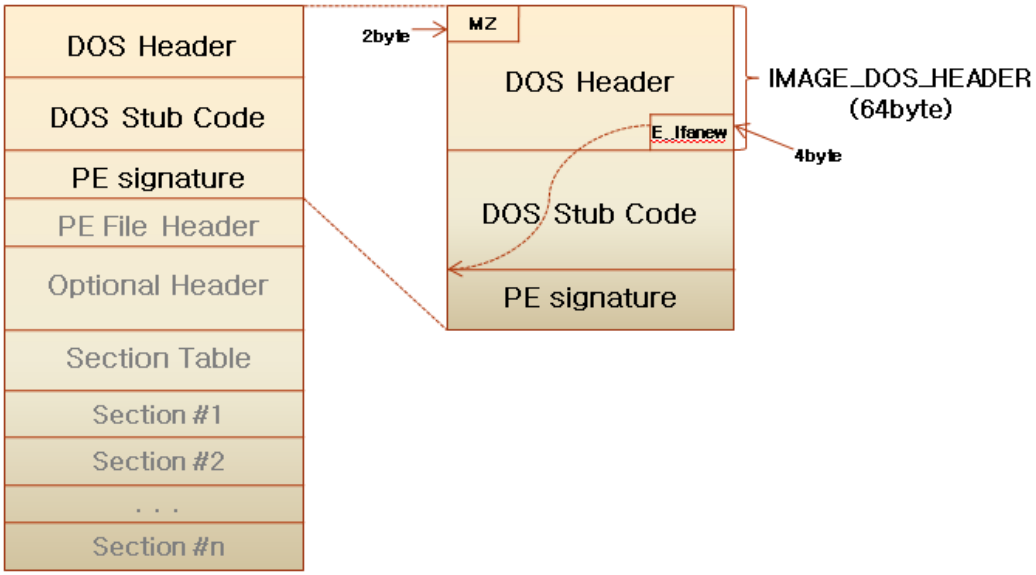


**"PE 파일은 디스크에서의 모습과 메모리에서의 로드된 모습이 거의 같다"**

디스크 상에서 PE파일은 DOS Header 로 시작하고 메모리상에서는 ImageBase에서 시작하기 때문에 DOS Header는 ImageBase를 찾으면 찾을 수가 있다.

## (2) DOS Header 및 DOS Stub Code

DOS Header는 DOS 와의 호환을 위해 사용되는 Header이다. PE 파일은 DOS Header로 시작하고 DOS Header 는 항상 64byte의 크기를 갖는다. 정확히 알아야 할 점은 가장 처음 2byte를 차지하고 있는 e_magic과 마지막 4byte를 차지하는 e_lfanew이다. e_lfanew는 PE Header의 시작점을 가리키는 오프셋 값이므로 반드시 찾아야 한다.




## (3) PE File, Optional Header

PE Header는 PE 파일의 정보를 가지고 있다. 섹션의 개수나 파일의 속성, optional header의 크기를 비롯하여 ImageBase, EntryPoint 주소, 메모리상에 각 섹션이 차지하는 크기, 디스크상에서의 섹션 정렬, PE파일의 총 사이즈, 디스크 상에서의 헤더의 총 사이즈 등과 같은 정보를 모두 담고 있다. File Header에서 꼭 알아두어야 할 4가지의 필드는 다음과 같다.

◉ Machine : CPU ID를 나타낸다. IA32(Intel Architecture 32)의 경우 0x14C가 되고 IA64일 경우 0x200이 되어야 한다.

◉ NumberOfSections : 섹션의 개수를 의미한다.

◉ SizeOfOptionalHeader : 파일 헤더 뒤에 오는 optional 헤더의 크기를 나타낸다. 기본적으로 224byte의 사이즈를 갖지만 data directory 가 생략될 경우 96byte의 사이즈를 갖게 된다.

◉ Characteristics : 파일의 속성을 나타낸다. 일반적인 실행 파일의 경우 0x10F의 값을 가진다.

Optional Header는 PE 파일에서 가장 중요한 부분이라고 할 수 있다. 명칭은 옵션이지만 절대 옵션의 성격을 갖고 있지 않고 프로그램이 메모리상에 로드되었을 때 시작할 주소라든지 메모리상에서의 정렬 단위와 같이 중요한 정보들을 다수 포함하고 있다. optional header는 30개의 필드와 1개의 데이터 디렉토리를 가지고 있다. 하지만 모든 필드를 알아야 할 필요는 마찬가지로 없다. 중요한 몇 가지의 필드를 알아보면 다음과 같다.

◉ Magic : optional 헤더의 시작위치에 존재하는 필드로 optional 헤더를 구분하는 Signature 로 사용됨. 0x10B로 고정되어 있다.

◉ AddressOfEntryPoint : EntryPoint는 PE파일이 메모리에 로드된 후 맨 처음으로 실행되는 코드의 주소를 가지고 있다. 이 부분에 지정된 주소값은 가상 주소가 아닌 RVA값이다. 즉, ImageBase에서부터의 오프셋 값이다.

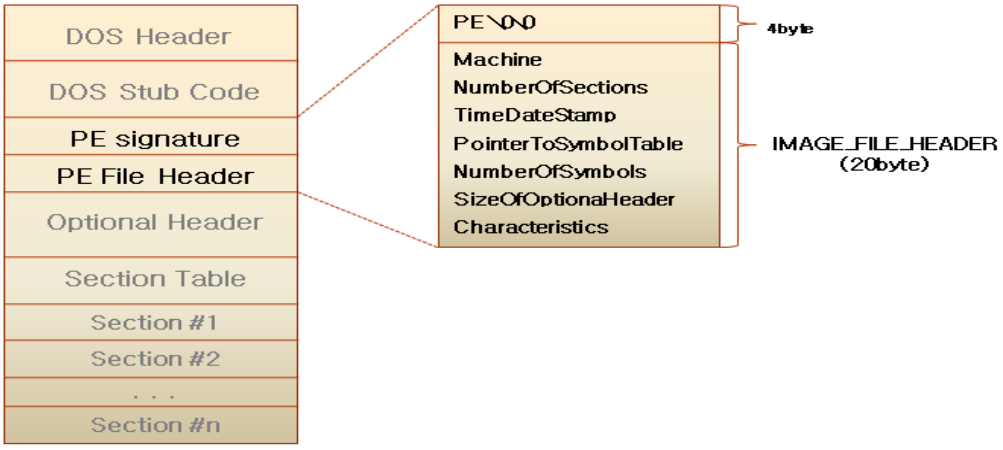
◉ ImageBase : PE파일이 로더에 의해서 로드되는 위치이다.

◉ SectionAlignment :각 섹션이 메모리상에서 차지하는 최소 단위이다. 각 섹션의 시작주소는 언제나 여기의 지정된 값의 배수가 되어야 한다.

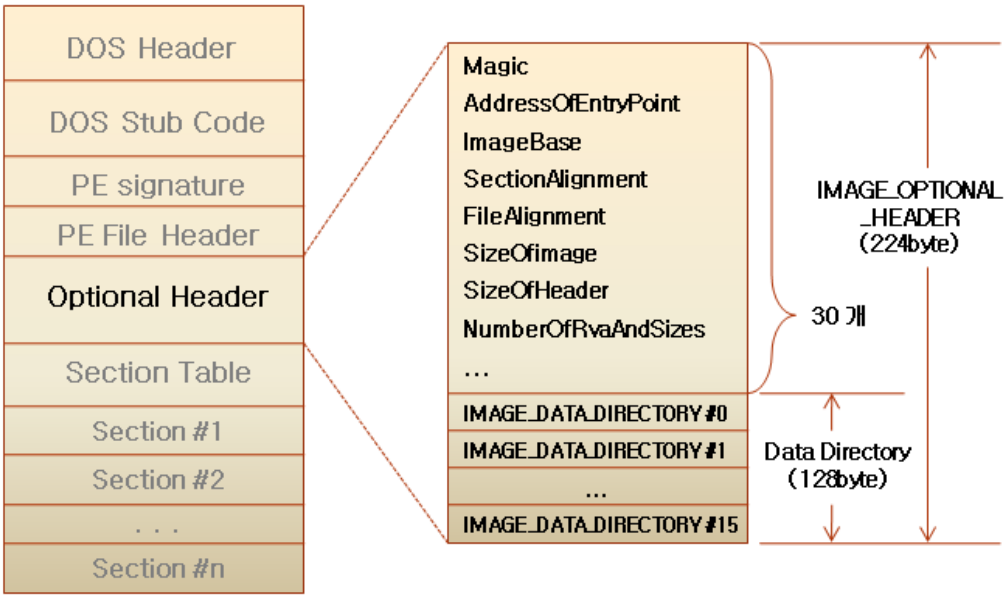
◉ FileAlignment : 디스크 상에서 섹션이 차지하는 최소 단위. 각 섹션의 시작주소는 이곳의 지정된 값의 배수가 되어야 한다.

◉ SizeOfImage :메모리상에 로드된 PE 파일의 총 사이즈이고 SecrionAlignment의 배수가 되어야 한다.

◉ SizeOfHeader : 디스크 상에서의 헤더의 총 사이즈이고 FileAlignment의 배수가 되어야 한다.

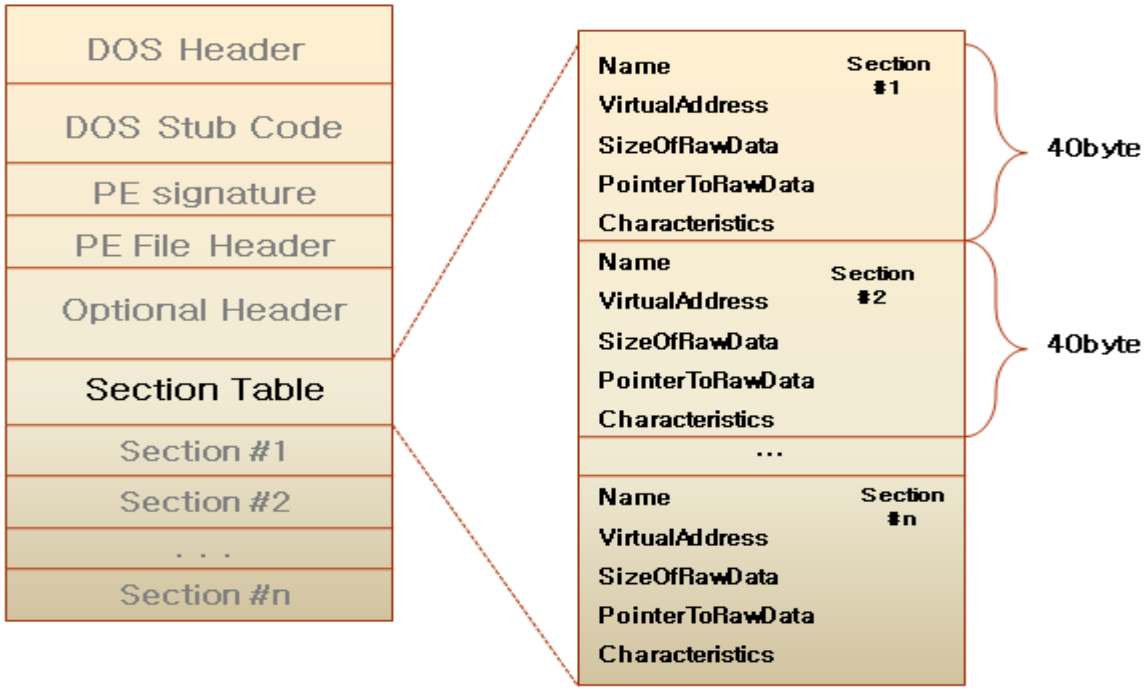



[ PE File Header ]




[ Optional Header ]

## (4) Section Table



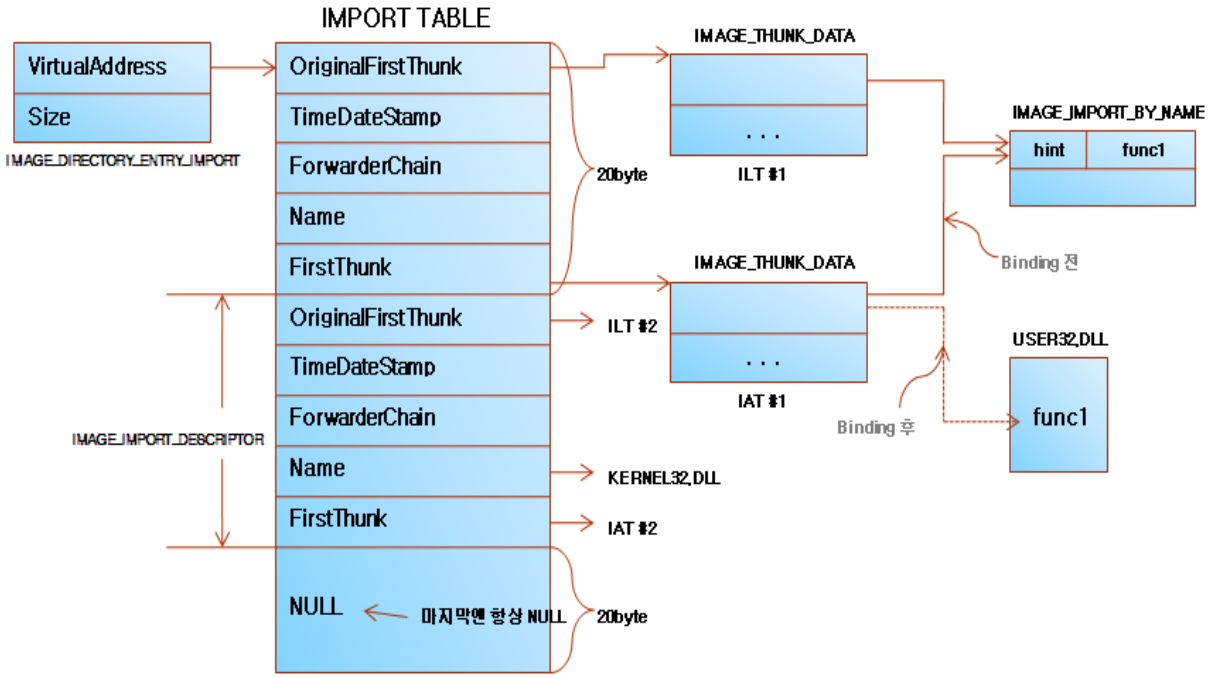

Section Table은 IMAGE_SECTION_HEADER 타입의 엘리먼트로 구성된 배열이다. 섹션 헤더는 로더가 각 섹션을 메모리에 로드하고 속성을 설정하는데 필요한 정보들을 가지고 있다. 섹션은 동일한 성질의 데이터가 저장되어 있는 영역으로 윈도우에서 사용하는 메모리 프로텍션 매커니즘과 연관이 있는데 윈도우의 경우 메모리 프로텍션의 최소 단위가 페이지이고 페이지 단위로 여러 속성을 설정해두고 속성에 위배되는 행동 시에는 access violation을 발생시켜서 메모리를 보호한다. 이것은 성질이 다른 데이터들은 하나의 페이지에 담을 수 없다는 것을 의미한다. 그렇다보니 프로그램에 포함된 데이터들 중 읽기와 실행이 가능해야 하는 데이터인 실행코드와 읽고 쓰기가 가능한 데이터, 읽기만 가능한 데이터들을 별도의 페이지에 두어야 하는데 로더의 입장에서는 이를 구분할 방법이 없으므로 섹션이라는 개념을 두어 실행 파일 생성 단계에서 구분해 놓도록 하는 것이다. 하나의 섹션은 8byte로 이루어진 섹션의 이름과 섹션이 로드될 가상주소(RVA), 파일 상에서의 섹션의 사이즈, 파일 상에서의 섹션의 시작위치, 섹션의 속성 값으로 이루어진다.

## (5) Import and Export Table

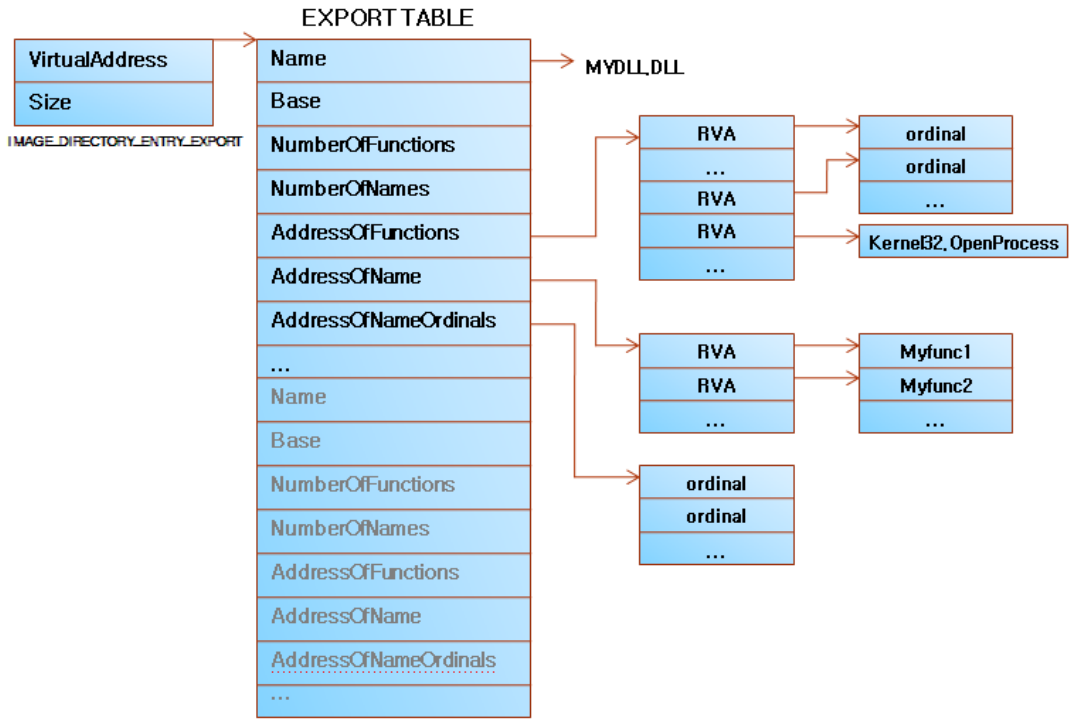
Import Table에는 PE파일이 실행될 때 외부로부터 가져와서 사용하는 함수들(DLL)의 목록이 들어있다. 아래의 그림에서는 USER32.DLL 과 KERNEL32.DLL을 IMPORT한 모습을 예로 한 것이다. DLL들은 함수를 Export하고 EXE가 그 DLL로부터 함수를 Import 한다. Export된 함수를 가져온다는 것은 결국 해당 DLL과 사용하는 함수에 대한 정보를 어딘가에 저장한다는 것을 의미하고 사용하고자 하는 Export함수들과 그 DLL에 대한 정보를 저장하고 있는 곳이 Import Table이다. 일반적으로 PE파일의 섹션 테이블에는 .idata라는 이름으로 지정된다.

◉ OriginalFirstThunk : ILT(Import Lookup Table)을 가리키는 RVA값이다. 앞의 그림에서 보이는 것처럼 ILT는 IMAGE_THUNK_DATA로 구성된 배열이다. IMAGE_THUNK_DATA는 상황에 따라서 IMAGE_IMPORT_BY_NAME을 가리키기도 하고 함수의 주소를 가리키기도 하며 일종의 일련번호로 사용되거나 포워더로 사용되기도 한다.

◉ TimeDateStamp : 바인딩 전에는 0으로 설정되며 바인딩 후에는 -1로 설정된다.
◉ ForwarderChain : 바인딩 전에는 0으로 설정되며 바인딩 후에는 -1로 설정된다.
◉ Name : Import한 DLL의 이름을 가리키는 RVA값이다.
◉ FirstThunk : Import Address Table의 RVA주소값을 가지고 있다.
쉽게 말해서 ILT에서 Import할 함수의 이름을 확인하고 IAT에서 확인된 이름이 있는 곳으로 바꿔면서 함수의 실제 위치를 가리키게 된다. 그래서 바인딩 후에 가리키는 곳이 서로가 다르게 된다. IAT를 복구한다는 말이 여기에서 나오는 것이다.
◉ ILT : 함수의 코드는 없다. 이름만 존재한다.
◉ IAT : 실제 함수가 있는 곳의 주소가 존재한다.



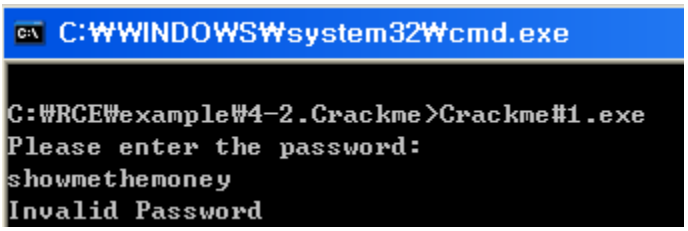

Import Table이 외부로부터 가져오는 함수들의 목록이 들어있다면 Export Table은 다른 프로그램을 위한 기능을 제공하기 위해 노출하는 함수들의 목록이 들어있다. Export Table은 Data Directory 중에서 0번째 Index인 IMAGE_DIRECTORY_ENTRY_EXPORT에서 Virtual Address와 Size를 확인할 수 있다. Export Table은 Import 테이블에 비해 비교적 직관적이고 간단한 구조로 되어있다.
◉ Name : DLL 이름을 나타내는 ASCII 문자열의 위치를 지시하는 RVA값
◉ Base : Export된 함수들에 대한 서수의 시작번호
◉ NumberOfFunctions : AddressOfFunctions가 가리키는 RVA 배열의 개수
◉ NumberOfNames : AddressOfNames가 가리키는 RVA 배열의 개수
◉ AddressOfFunctions : 함수의 실제 주소가 담긴 배열(RVA값)
◉ AddressOfNames : 함수의 심볼을 나타내는 문자열 배열(RVA값)
◉ AddressOfNameOrdinals : 함수의 서수 값의 배열 위치, 실제 해당 함수의 서수는 Base 값을 더해야 한다.
Export Table은 내보내지는 DLL이 가지고 있는 함수들의 이름과 위치(RVA값), 그리고 서수를 가지고 있다.




# 4. 분석의 기초

## (1) CrackMe 분석실습

CrackMe #1




실행 시키게 되면 패스워드를 요구하고 패스워드가 일치하지 않기 때문에 “Invalid Password"라는 메시지를 출력하면서 프로그램이 종료된다. 하지만 이것만으로도 많은 수확이 있었음을 알아야 한다. 우리는 어떠한 입력을 했고 “Invalid Password"라는 문자열이 입력에 반응하여 출력되었기 때문이다. 이제 디버거를 통하여 프로그램을 실행시킨다. 스트링값을 검색하기 위해 Serrch for -> All reference text strings를 클릭하면 다음과 같이 스트링값들이 모두 뜨게 된다.

R Text strings referenced in Crackme#:.text

| Address | Disassembly | Text string |
|---|---|---|
| 00401022 | PUSH Crackme#.00407030 | ASCII "Please enter the password:" |
| 0040108C | MOV DWORD PTR SS:[EBP-20],0 | (Initial CPU selection) |
| 004010C9 | PUSH Crackme#.0040704C | ASCII "Invalid Password" |
| 004010E0 | PUSH Crackme#.00407060 | ASCII "The password is %s" |
| 004023C3 | MOV ESI,Crackme#.00407D6C | ASCII "C:\RCE\example\4-2.Crackme\Crackme#1.exe" |
| 00402C7A | PUSH Crackme#.00406430 | ASCII "<program name unknown>" |
| 00402CBC | PUSH Crackme#.0040642C | ASCII "..." |
| 00402CD0 | PUSH Crackme#.00406410 | ASCII "Runtime Error!Program: " |
| 00402CEE | PUSH Crackme#.0040640C | ASCII "" |
| 00402D16 | PUSH Crackme#.004063E4 | ASCII "Microsoft Visual C++ Runtime Library" |
| 0040473A | PUSH Crackme#.00406478 | ASCII "user32.dll" |
| 00404751 | PUSH Crackme#.0040646C | ASCII "MessageBoxA" |
| 00404762 | PUSH Crackme#.0040645C | ASCII "GetActiveWindow" |
| 0040476A | PUSH Crackme#.00406448 | ASCII "GetLastActivePopup" |

여기서 어떤것들이 나오겠구나~라는 것을 파악하고 넘어가야 한다. “Invalid Password”라는 스트링에서 더블클릭하면 해당 코드로 이동하게 된다.

CPU - main thread, module Crackme#

| Address | Hex dump | Disassembly | Comment |
|---|---|---|---|
| 0040108C | C745 E0 00000( | MOV DWORD PTR SS:[EBP-20],0 | |
| 00401093 | C745 DC 0300( | MOV DWORD PTR SS:[EBP-24],3 | |
| 0040109A | EB 12 | JMP SHORT Crackme#.004010AE | |
| 0040109C | 8B4D E0 | MOV ECX,DWORD PTR SS:[EBP-20] | |
| 0040109F | 83C1 04 | ADD ECX,4 | |
| 004010A2 | 894D E0 | MOV DWORD PTR SS:[EBP-20],ECX | |
| 004010A5 | 8B55 DC | MOV EDX,DWORD PTR SS:[EBP-24] | |
| 004010A8 | 83EA 01 | SUB EDX,1 | |
| 004010AB | 8955 DC | MOV DWORD PTR SS:[EBP-24],EDX | |
| 004010AE | 837D E0 0D | CMP DWORD PTR SS:[EBP-20],0D | |
| 004010B2 | 73 28 | JNB SHORT Crackme#.004010DC | |
| 004010B4 | 8B45 E0 | MOV EAX,DWORD PTR SS:[EBP-20] | |
| 004010B7 | C1E8 02 | SHR EAX,2 | |
| 004010BA | 8B4D F8 | MOV ECX,DWORD PTR SS:[EBP-8] | |
| 004010BD | 8B55 DC | MOV EDX,DWORD PTR SS:[EBP-24] | |
| 004010C0 | 8B0481 | MOV EAX,DWORD PTR DS:[ECX+EAX*4] | |
| 004010C3 | 3B4495 E8 | CMP EAX,DWORD PTR SS:[EBP+EDX*4-18] | |
| 004010C7 | 74 11 | JE SHORT Crackme#.004010DA | |
| 004010C9 | 68 4C704000 | PUSH Crackme#.0040704C | ASCII "Invalid Password" |
| 004010CE | E8 20000000 | CALL Crackme#.004010F3 | |
| 004010D3 | 83C4 04 | ADD ESP,4 | |
| 004010D6 | 33C0 | XOR EAX,EAX | |
| 004010D8 | EB 15 | JMP SHORT Crackme#.004010EF | |
| 004010DA | EB C0 | JMP SHORT Crackme#.0040109C | |
| 004010DC | 8D4D CC | LEA ECX,DWORD PTR SS:[EBP-34] | |
| 004010DF | 51 | PUSH ECX | |
| 004010E0 | 68 60704000 | PUSH Crackme#.00407060 | ASCII "The password is %s" |
| 004010E5 | E8 09000000 | CALL Crackme#.004010F3 | |
| 004010EA | 83C4 08 | ADD ESP,8 | |
| 004010ED | 33C0 | XOR EAX,EAX | |
| 004010EF | 8BE5 | MOV ESP,EBP | |
| 004010F1 | 5D | POP EBP | |
| 004010F2 | C3 | RETN | |

“invalid Password”라는 메시지 위쪽에 비교하는 부분이 있고 같을 경우 004010DA로 분기하는 것을 볼 수 있다. 004010DA는 다시 0040109C로 분기하기 때문에 아래쪽에 보이는 “The password is %s” 부분으로는 이동하지 않는다. 조금 더 위쪽을 보게되면 JNB에 의해서 분기하는 코드가 보이는데 조건에 만족하면 004010DC로 분기하여 “The password is %s”라는 메시지를 출력하게 된다. 여기까지해서 “The password is %s”이런 메시지가 출력된다는 것은 이곳보다 위쪽에서 우리가 입력한 패스워드와 프로그램이 요구하는 패스워드를 비교하는 부분이 있을 것이라는 것을 알 수 있다. 처음으로 가서 “Please enter the password:”처럼 입력이 들어오는 부분을 살펴보면 다음과 같다.

| Address | Hex dump | Disassembly | Comment |
|---|---|---|---|
| 00401000 | $ 55 | PUSH EBP | |
| 00401001 | 8BEC | MOV EBP,ESP | |
| 00401003 | 83EC 34 | SUB ESP,34 | |
| 00401006 | C745 E8 670A( | MOV DWORD PTR SS:[EBP-18],0A67 | |
| 0040100D | C745 EC 7070( | MOV DWORD PTR SS:[EBP-14],6E697070 | |
| 00401014 | C745 F0 7274( | MOV DWORD PTR SS:[EBP-10],69727472 | |
| 0040101B | C745 F4 706F( | MOV DWORD PTR SS:[EBP-C],65776F70 | |
| 00401022 | 68 30704000 | PUSH Crackme#.00407030 | ASCII "Please enter the password:" |
| 00401027 | E8 7C010000 | CALL Crackme#.004011A8 | |
| 0040102C | 83C4 04 | ADD ESP,4 | |
| 0040102F | 6A 10 | PUSH 10 | |
| 00401031 | 6A 00 | PUSH 0 | |
| 00401033 | 8D45 CC | LEA EAX,DWORD PTR SS:[EBP-34] | |
| 00401036 | 50 | PUSH EAX | |
| 00401037 | E8 14010000 | CALL Crackme#.00401150 | |

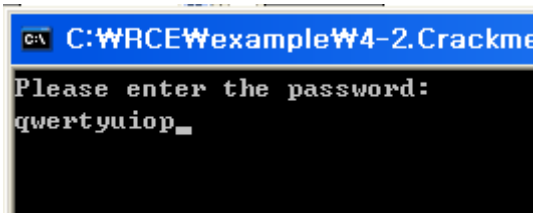
PUSH Crackme#.00407030은 “Please enter the password:”라는 출력할 메시지를 스택에 넣어둔 후 바로 아래에 있는 CALL에 의해서 화면에 출력이 된다. 그러므로 항상 CALL에 의해서 함수 호출이 끝나면 스택을 잘 보아야한다. 함수 내부에서 어떠한 연산 작업 후 스택에 데이터들이 저장되기 때문이다. 그리고 이 데이터 값들에 의해서 다음 코드에 영향을 미치게 되기 때문이다. 현재 스택의 값들은 다음과 같다.

```
Registers (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C93E4F4 ntdll.KiFastSystemCallRet
EBX 7FFDD000
ESP 0012FFC4
EBP 0012FFF0
ESI FFFFFFFF
EDI 7C940208 ntdll.7C940208
EIP 00401220 Crackme#.<ModuleEntryPoint>
```

스택의 값들을 확인하고 그밑의 코드들을 하나하나 살펴보도록 하겠다.

```
0040102C   |.  ADD ESP,4                          //Stack Clear
0040102F   |.  PUSH 10                            //Stack에 10을 저장
00401031   |.  PUSH 0                             //Stack에 0을 저장
00401033   |.  LEA EAX,DWORD PTR SS:[EBP-34]      //EBP-34의 위치(주소)를 EAX에 저장
00401036   |.  PUSH EAX                           //EAX의 값을 Stack에 저장
00401037   |.  CALL Crackme#.00401150             //00401150의 코드를 호출
0040103C   |.  ADD ESP,0C                         //Stack Clear
0040103F   |.  MOV DWORD PTR SS:[EBP-1C],0        //EBP-1C의 위치에 0을 저장
00401046   |.  MOV DWORD PTR SS:[EBP-20],0        //EBP-20의 위치에 0을 저장
0040104D   |.  MOV DWORD PTR SS:[EBP-24],0        //EBP-24의 위치에 0을 저장
```

위의 부분들은 단순히 우리가 입력한 패스워드와 프로그램이 가지고 있는 패스워드를 비교하는 연산을 위해서 공간을 만드는 역할을 하는 코드이다. 다음 코드로 진행하면 패스워드를 입력하는 부분이 나온다. 임의의 패스워드를 입력하고 다음 코드를 살펴보자.

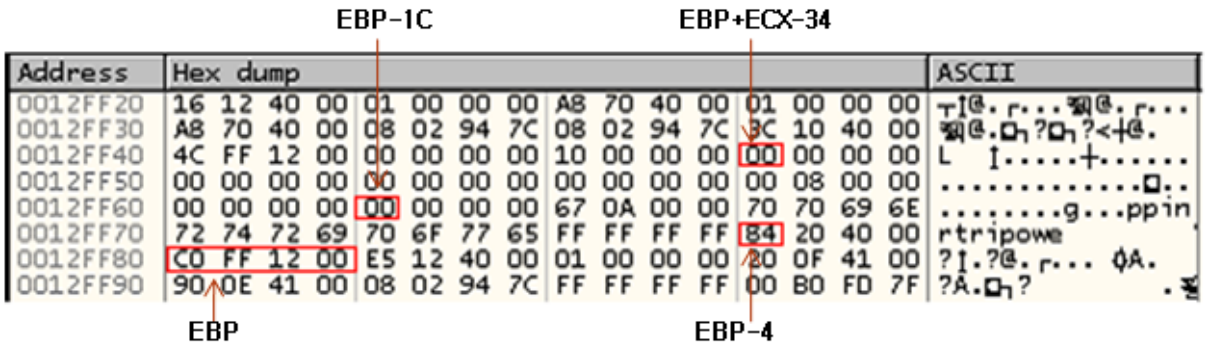



00401054에서 00401140에 위치한 함수를 호출한다. 이 함수는 사용자의 입력값을 받아들이는 함수이다. 그리고 함수를 호출하고 연산을 수행한 결과를 EAX에 저장한다. 그다음 나오는 루프문의 코드가 어떤 역할을 하는지 살펴보면 다음과 같다.

```
Address   Hex dump          Disassembly
0040103F  .  C745 E4 00000( MOV DWORD PTR SS:[EBP-1C],0
00401046  .  C745 E0 00000( MOV DWORD PTR SS:[EBP-20],0
0040104D  .  C745 DC 00000( MOV DWORD PTR SS:[EBP-24],0
00401054  >  E8 E7000000    CALL Crackme#.00401140
00401059  .  8845 FC        MOV BYTE PTR SS:[EBP-4],AL
0040105C  .  8B4D E4        MOV ECX,DWORD PTR SS:[EBP-1C]
0040105F  .  8A55 FC        MOV DL,BYTE PTR SS:[EBP-4]
00401062  .  88540D CC      MOV BYTE PTR SS:[EBP+ECX-34],DL
00401066  .  8B45 E4        MOV EAX,DWORD PTR SS:[EBP-1C]
00401069  .  83C0 01        ADD EAX,1
0040106C  .  8945 E4        MOV DWORD PTR SS:[EBP-1C],EAX
0040106F  .  0FBE4D FC      MOVSX ECX,BYTE PTR SS:[EBP-4]
00401073  .  83F9 0A        CMP ECX,0A
00401076  .v 74 0E          JE SHORT Crackme#.00401086
00401078  .  0FBE55 FC      MOVSX EDX,BYTE PTR SS:[EBP-4]
0040107C  .  85D2           TEST EDX,EDX
0040107E  .v 74 06          JE SHORT Crackme#.00401086
00401080  .  837D E4 10     CMP DWORD PTR SS:[EBP-1C],10
00401084  .^ 72 CE          JB SHORT Crackme#.00401054
00401086  >  8D45 CC        LEA EAX,DWORD PTR SS:[EBP-34]
00401089  .  8945 F8        MOV DWORD PTR SS:[EBP-8],EAX
```

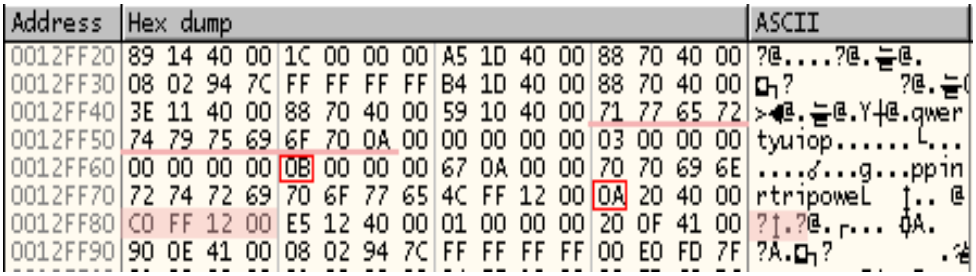
```
00401054  CALL Crackme#.00401140            //00401140 호출
00401059  MOV BYTE PTR SS:[EBP-4],AL       //AL에 있는 값을 EBP-4에 1바이트만 저장
0040105C  MOV ECX,DWORD PTR SS:[EBP-1C]     //EBP-1C에 있는 값을 ECX에 저장
0040105F  MOV DL,BYTE PTR SS:[EBP-4]       //EBP-4에 있는 값을 DL에 저장
00401062  MOV BYTE PTR SS:[EBP+ECX-34],DL //DL에 있는 값을 EBP+ECX-34에 저장
00401066  MOV EAX,DWORD PTR SS:[EBP-1C]  //EBP-1C에 있는 값을 EAX에 저장
00401069  ADD EAX,1                         //EAX값을 1증가
0040106C  MOV DWORD PTR SS:[EBP-1C],EAX  //EAX에 있는 값을 EBP-1C에 저장
0040106F  MOVSX ECX,BYTE PTR SS:[EBP-4]   //EBP-4에 있는 값을 ECX에 저장
00401073  CMP ECX,0A                        //ECX와 0x0A를 비교
00401076  JE SHORT Crackme#.00401086        //같으면 00401086으로 분기
00401078  MOVSX EDX,BYTE PTR SS:[EBP-4]   //EBP-4에 있는 값을 EDX에 저장
0040107C  TEST EDX,EDX                      //EDX와 EDX를 AND연산
0040107E  JE SHORT Crackme#.00401086        //같으면 00401086으로 분기
00401080  CMP DWORD PTR SS:[EBP-1C],10     //EBP-1C에 있는 값과 0x10을 비교
00401084  JB SHORT Crackme#.00401054   //EBP-1C의 값이 0x10보다 적으면 00401054로 분기
```

이렇게 계속 해서 루프를 돌면서 뭔가를 계산하고 있다. 4번 정도 루프를 돈 후 헥사 값이 있는 부분을 확인해 보도록 하자.

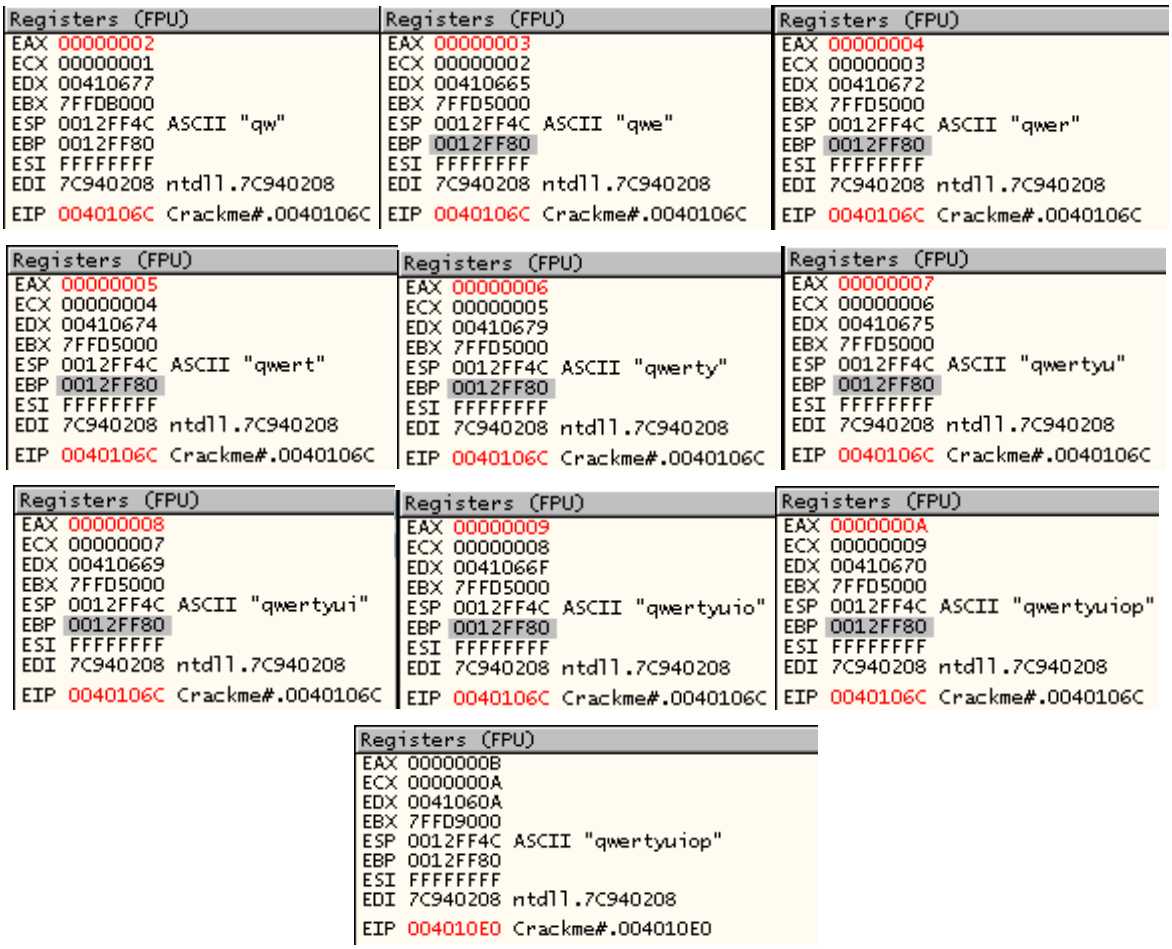



CALL 00401140에 의해 우리가 입력한 값을 한 글자씩 가져와서 EAX에 저장하고 글자 수를 세고 있다. 입력한 글자의 개수는 [EBP-1C]에 저장이 되고 현재까지 읽어 들인 글자들은 [EBP+ECX-34]에 저장이 된다. 그리고 현재 읽어 들인 글자는 [EBP-4]에 저장이 된다. 코드에서 보면 BYTE PTR SS라는 부분이 자주 나오는데 이것이 바로 한 글자씩 읽어온다는 증거이다. 그리고 ECX에는 위의 코드에서처럼 [EBP-4]에 있는 값을 저장한 후 0x0A와 비교한다. [EBP-4]는 현재 읽어 들인 글자하나를 의미하며 LF(Line Feed)인 0x0A와 사용자가 입력한 글자의 끝을 찾기 위해 사용된 비교 구문이다.

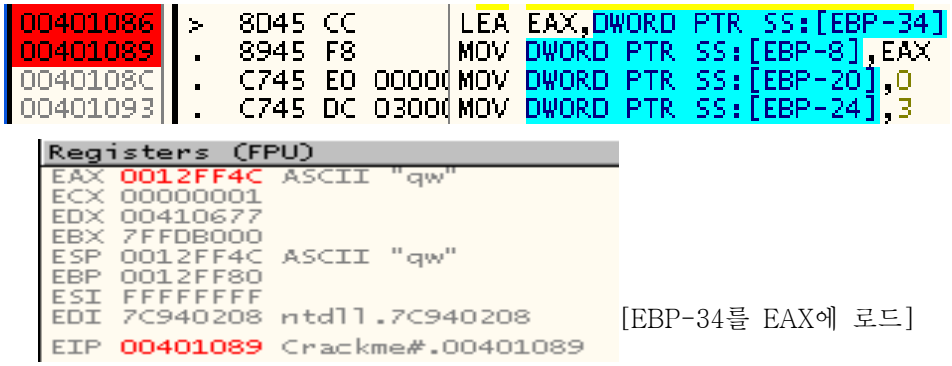
위쪽 코드에서 맨 아래의 두줄은 [EBP-1C]에 있는 값과 0x10을 비교하는 부분인데 [EBP-1C]에는 현재까지 읽어 들인 글자의 개수를 나타내므로 사용자가 입력한 값의 길이가 16글자(0x10)보다 작은지 확인하는 부분이다. 16글자 보다 많은 경우는 00401054로 분기하지 않고 루프문을 빠져 나오게 된다. 즉, 입력한 글자를 16글자까지만 인식을 하게 된다. 다음은 루프를 빠져 나온 후의 레지스터에 저장된 값과 헥사코드 정보에 대한 부분이다.



브레이크 포인트를 00401086에 걸고 루프를 계속 돌리면 입력받은 패스워드를 한글자씩 가져와서 비교하는 것을 눈으로 확인할 수 있다.




여기까지 알아낸 Crackme#1을 풀기 위한 조건은 16글자 이하라는 것과 EAX에 사용자가 입력한 글자 수를 저장해두었다는 것이다. 물론 EAX에 있는 값은 어디론가 옮겨져 사용될 것이다. EAX에 계속해서 저장해둘 수만은 없기 때문이다.




[EBP-34를 EAX에 로드]

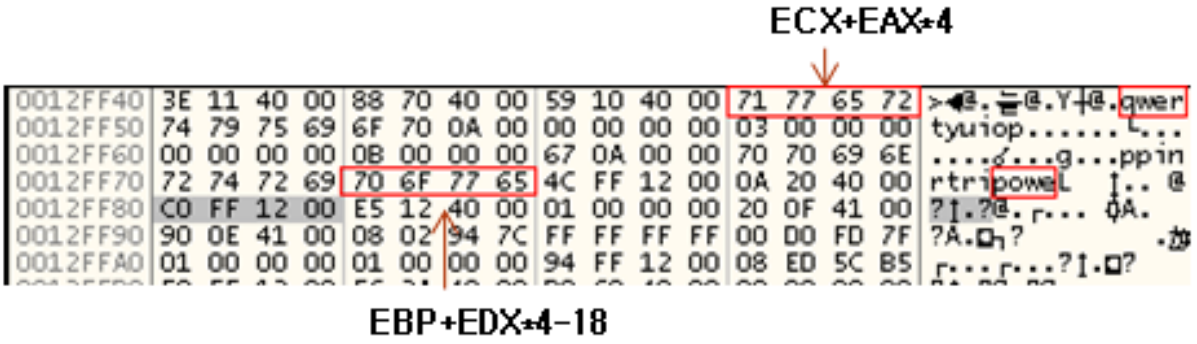
EBP-34의 주소값을 EAX에 로드하고 EAX에 있는 값을 EBP-8의 위치로 옮긴다. 그리고 EBP-20과 EBP-24는 각각 0과 3이라는 값을 저장한다. 아마도 다음 작업을 위한 준비 단계일 것이다. EBP-34의 주소값을 확인해보니 0012FF4C이고 이 주소에는 사용자가 입력했던 값이 저장되어 있다. 다음에 나오는 코드는 004010AE로 분기하는 코드이다.

```
Address   Hex dump          Disassembly
00401093  .  C745 DC 03000( MOV DWORD PTR SS:[EBP-24],3
0040109A  .v EB 12          JMP SHORT Crackme#.004010AE
0040109C  >  8B4D E0        MOV ECX,DWORD PTR SS:[EBP-20]
0040109F  .  83C1 04        ADD ECX,4
004010A2  .  894D E0        MOV DWORD PTR SS:[EBP-20],ECX
004010A5  .  8B55 DC        MOV EDX,DWORD PTR SS:[EBP-24]
004010A8  .  83EA 01        SUB EDX,1
004010AB  .  8955 DC        MOV DWORD PTR SS:[EBP-24],EDX
004010AE  >  837D E0 0D     CMP DWORD PTR SS:[EBP-20],0D
004010B2  .v 73 28          JNB SHORT Crackme#.004010DC
004010B4  .  8B45 E0        MOV EAX,DWORD PTR SS:[EBP-20]
```

[EBP-20]에 있는 값과 0x0D를 비교하여 [EBP-20]의 값이 0x0D보다 작지 않으면 004010DC로 분기한다. 하지만 앞서 나온 코드에서 [EBP-20]에 0을 저장했으므로 분기하지 않고 다음 코드를 실행하게 된다.

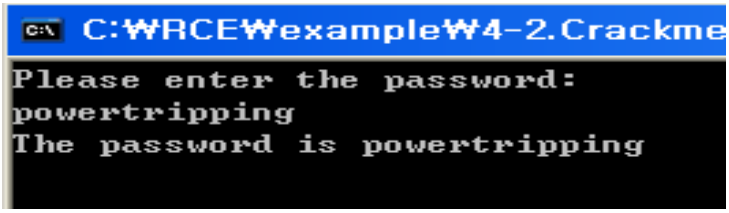
```
Address   Hex dump          Disassembly                          Comment
004010AE  >  837D E0 0D     CMP DWORD PTR SS:[EBP-20],0D
004010B2  .v 73 28          JNB SHORT Crackme#.004010DC
004010B4  .  8B45 E0        MOV EAX,DWORD PTR SS:[EBP-20]
004010B7  .  C1E8 02        SHR EAX,2
004010BA  .  8B4D F8        MOV ECX,DWORD PTR SS:[EBP-8]
004010BD  .  8B55 DC        MOV EDX,DWORD PTR SS:[EBP-24]
004010C0  .  8B0481         MOV EAX,DWORD PTR DS:[ECX+EAX*4]
004010C3  .  3B4495 E8      CMP EAX,DWORD PTR SS:[EBP+EDX*4-18]
004010C7  .v 74 11          JE SHORT Crackme#.004010DA
004010C9  .  68 4C704000    PUSH Crackme#.0040704C               ASCII "Invalid Password"
004010CE  .  E8 20000000    CALL Crackme#.004010F3
004010D3  .  83C4 04        ADD ESP,4
004010D6  .  33C0           XOR EAX,EAX
004010D8  .v EB 15          JMP SHORT Crackme#.004010EF
004010DA  >^ EB C0          JMP SHORT Crackme#.0040109C
004010DC  >  8D4D CC        LEA ECX,DWORD PTR SS:[EBP-34]
004010DF  .  51             PUSH ECX
004010E0  .  68 60704000    PUSH Crackme#.00407060               ASCII "The password is %s"
004010E5  .  E8 09000000    CALL Crackme#.004010F3
004010EA  .  83C4 08        ADD ESP,8
004010ED  .  33C0           XOR EAX,EAX
004010EF  >  8BE5           MOV ESP,EBP
004010F1  .  5D             POP EBP
004010F2  L. C3             RETN
```

사용자가 입력한 패스워드와 프로그램이 가지고 있는 패스워드의 처음 4글자를 비교하는 부분이다. 헥사코드 부분을 보도록 하자.




004010B4에서부터 004010C3까지의 코드가 처음 실행될 때 ECX의 값은 사용자가 입력한 값이 저장되어 있는 주소를 가지고 있으며 EAX는 0을 가지고 있다. 그리고 EDX는 EBP-24에 있는 값을 가지고 있는데 이전 코드에서 3이라는 값을 저장해두었다. 결과적으로 EBP+EDX*4-18을 해보면 0x0013FF74가 된다. 따라서 qwer과 powe를 비교한다. 그리고 다음 코드에서 같으면 004010DA로 분기하고 그렇지 않으면 004010C9을 실행한다. 004010C9에서는 “Incalid Password”라는 메시지를 출력하면서 종료되는 루틴이고 004010DA는 다시 0040109C로 분기하여 다음 값을 비교하는 코드이다. 이러한 패턴으로 패스워드는 4글자씩 분할되어 거꾸로 저장이 되어있고 비교할 때 뒤에서부터 4글자씩 가져와서 사용자가 입력한 값과 비교를 한다는 것을 알 수 있다. 알아낸 패스워드를 입력하게 되면 다

음과 같은 화면을 출력하면서 끝이나게 된다.

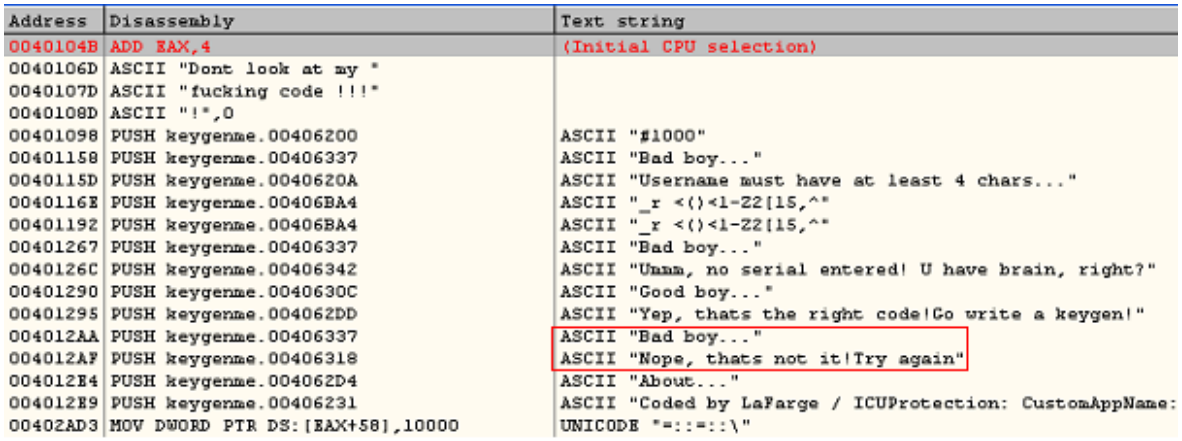



## (2) KeygenMe 분석실습

Keygenme#1

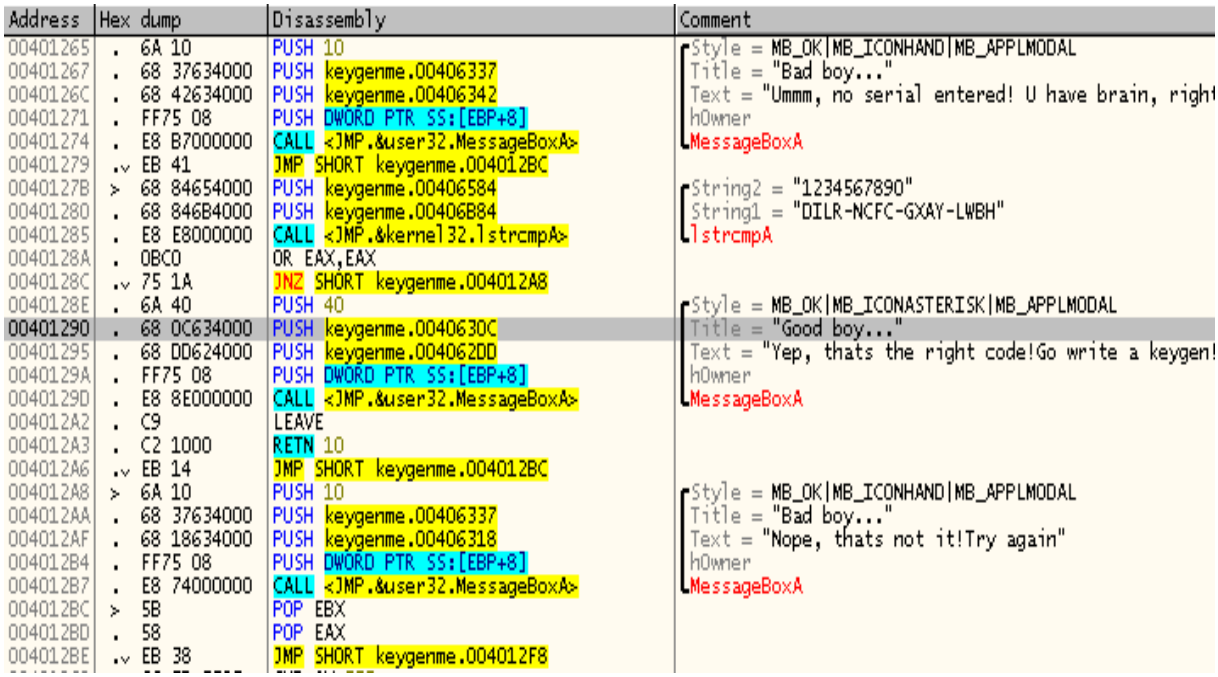
킷값을 만들어 내는 Keygenme에 대해서 분석해보도록 하겠다. Keygenme는 말 그대로 키를 생성하는 프로그램으로 직접 C나 C++과 같은 프로그래밍 언어를 이용하여 작성할 수 있고 내부 키젠이라고 해서 프로그램 내부에 키 값을 생성하는 루틴을 찾아서 사용자가 잘못 입력하면 정상적인 킷값을 보여줄 수 있도록 프로그램을 패치시키는 방법을 사용하기도 한다. 이제 프로그램을 실행시키고 사용자 이름과 인증코드를 임의로 입력하면 다음과 같다.


위와 같이 정상적인 인증코드를 입력하지 않으면 “Nope, that's not it! Try again”이라는 메시지를 출력한다. 디버거를 통해서 파일을 오픈해보겠다. 먼저 위에 보이는 에러 메시지가 있는 곳을 찾아가 보는게 순서일 것이다.



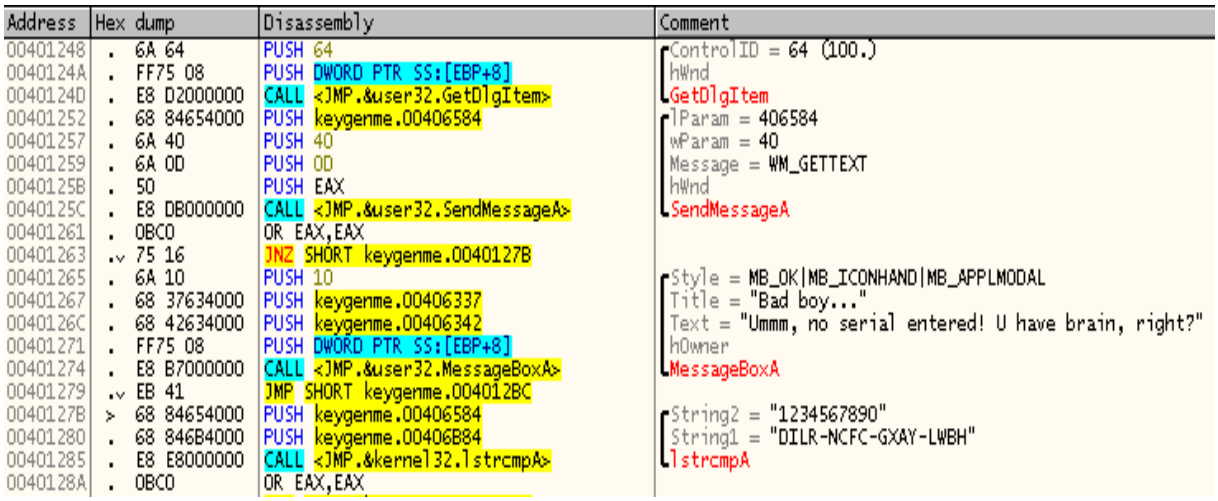
| Address | Disassembly | Text string |
|---|---|---|
| 0040104B | ADD EAX,4 | (Initial CPU selection) |
| 0040106D | ASCII "Dont look at my " | |
| 0040107D | ASCII "fucking code !!!" | |
| 0040108D | ASCII "!",0 | |
| 00401098 | PUSH keygenme.00406200 | ASCII "$1000" |
| 00401158 | PUSH keygenme.00406337 | ASCII "Bad boy..." |
| 0040115D | PUSH keygenme.0040620A | ASCII "Username must have at least 4 chars..." |
| 0040116E | PUSH keygenme.00406BA4 | ASCII "_r <()<1-Z2[15,^" |
| 00401192 | PUSH keygenme.00406BA4 | ASCII "_r <()<1-Z2[15,^" |
| 00401267 | PUSH keygenme.00406337 | ASCII "Bad boy..." |
| 0040126C | PUSH keygenme.00406342 | ASCII "Umm, no serial entered! U have brain, right?" |
| 00401290 | PUSH keygenme.0040630C | ASCII "Good boy..." |
| 00401295 | PUSH keygenme.004062DD | ASCII "Yep, thats the right code!Go write a keygen!" |
| 004012AA | PUSH keygenme.00406337 | ASCII "Bad boy..." |
| 004012AF | PUSH keygenme.00406318 | ASCII "Nope, thats not it!Try again" |
| 004012E4 | PUSH keygenme.004062D4 | ASCII "About..." |
| 004012E9 | PUSH keygenme.00406231 | ASCII "Coded by LaFarge / ICUProtection: CustomAppName: |
| 00402AD3 | MOV DWORD PTR DS:[EAX+58],10000 | UNICODE "=:::=::\" |

우리가 보았던 메시지는 Bad boy라는 곳의 메시지이고 그 위쪽에 Good boy는 성공 메시지일 것이라는 추측도 가능하다. 성공 메시지를 더블클릭하여 해당 코드로 이동하겠다.



```
Address   Hex dump     Disassembly                 Comment
00401265  .  6A 10     PUSH 10                     Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401267  .  68 37634000 PUSH keygenme.00406337    Title = "Bad boy..."
0040126C  .  68 42634000 PUSH keygenme.00406342    Text = "Ummm, no serial entered! U have brain, right
00401271  .  FF75 08   PUSH DWORD PTR SS:[EBP+8]   hOwner
00401274  .  E8 B7000000 CALL <JMP.&user32.MessageBoxA>  MessageBoxA
00401279  .v EB 41     JMP SHORT keygenme.004012BC
0040127B  >  68 84654000 PUSH keygenme.00406584    String2 = "1234567890"
00401280  .  68 846B4000 PUSH keygenme.00406B84    String1 = "DILR-NCFC-GXAY-LWBH"
00401285  .  E8 E8000000 CALL <JMP.&kernel32.lstrcmpA>  lstrcmpA
0040128A  .  0BC0      OR EAX,EAX
0040128C  .v 75 1A     JNZ SHORT keygenme.004012A8
0040128E  .  6A 40     PUSH 40                     Style = MB_OK|MB_ICONASTERISK|MB_APPLMODAL
00401290  .  68 0C634000 PUSH keygenme.0040630C    Title = "Good boy..."
00401295  .  68 DD624000 PUSH keygenme.004062DD    Text = "Yep, thats the right code!Go write a keygen!
0040129A  .  FF75 08   PUSH DWORD PTR SS:[EBP+8]   hOwner
0040129D  .  E8 8E000000 CALL <JMP.&user32.MessageBoxA>  MessageBoxA
004012A2  .  C9        LEAVE
004012A3  .  C2 1000   RETN 10
004012A6  .v EB 14     JMP SHORT keygenme.004012BC
004012A8  >  6A 10     PUSH 10                     Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
004012AA  .  68 37634000 PUSH keygenme.00406337    Title = "Bad boy..."
004012AF  .  68 18634000 PUSH keygenme.00406318    Text = "Nope, thats not it!Try again"
004012B4  .  FF75 08   PUSH DWORD PTR SS:[EBP+8]   hOwner
004012B7  .  E8 74000000 CALL <JMP.&user32.MessageBoxA>  MessageBoxA
004012BC  >  5B        POP EBX
004012BD  .  58        POP EAX
004012BE  .v EB 38     JMP SHORT keygenme.004012F8
```

여기에서 성공 메시지가 있는 코드로 분기할 것인지 에러 메시지가 있는 코드로 분기할 것인지를 결정하는 부분은 0040128A에 있는 OR EAX, EAX이다. 그리고 그 위쪽으로 lstcmpA라는 함수가 보이고 이 함수에 의해서 우리가 입력한 값과 프로그램이 가지고 있는 값을 비교할 것이라고 예상할 수 있다. 그럼 입력한 값은 어디에 저장되어 있으며 프로그램이 가지고 있는 값은 어디에 저장되어 있는지 찾아보도록 하겠다. 위의 코드에서 조금 더 위로 올라가면 다음과 같은 코드를 찾을 수 있다.



```
Address   Hex dump     Disassembly                 Comment
00401248  .  6A 64     PUSH 64                     ControlID = 64 (100.)
0040124A  .  FF75 08   PUSH DWORD PTR SS:[EBP+8]   hWnd
0040124D  .  E8 D2000000 CALL <JMP.&user32.GetDlgItem>  GetDlgItem
00401252  .  68 84654000 PUSH keygenme.00406584    lParam = 406584
00401257  .  6A 40     PUSH 40                     wParam = 40
00401259  .  6A 0D     PUSH 0D                     Message = WM_GETTEXT
0040125B  .  50        PUSH EAX                    hWnd
0040125C  .  E8 DB000000 CALL <JMP.&user32.SendMessageA>  SendMessageA
00401261  .  0BC0      OR EAX,EAX
00401263  .v 75 16     JNZ SHORT keygenme.0040127B
00401265  .  6A 10     PUSH 10                     Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
00401267  .  68 37634000 PUSH keygenme.00406337    Title = "Bad boy..."
0040126C  .  68 42634000 PUSH keygenme.00406342    Text = "Ummm, no serial entered! U have brain, right?"
00401271  .  FF75 08   PUSH DWORD PTR SS:[EBP+8]   hOwner
00401274  .  E8 B7000000 CALL <JMP.&user32.MessageBoxA>  MessageBoxA
00401279  .v EB 41     JMP SHORT keygenme.004012BC
0040127B  >  68 84654000 PUSH keygenme.00406584    String2 = "1234567890"
00401280  .  68 846B4000 PUSH keygenme.00406B84    String1 = "DILR-NCFC-GXAY-LWBH"
00401285  .  E8 E8000000 CALL <JMP.&kernel32.lstrcmpA>  lstrcmpA
0040128A  .  0BC0      OR EAX,EAX
```

00401248에 BreakPoint를 걸고 Run을 시킨 후에 임의의 사용자 이름과 인증 코드를 입력하고 Check It을 누르면 00401248에서 멈춘다. 이때 F8을 이용하여 한줄 씩 실행다가가보면 00401252주소에 있는 코드를 실행한 후에 스택에 다음과 같은 값이 올라옴을 알 수가 있다.

```
Address   Value      Comment
0012FA5C  00406584   ASCII "DILRNCFCGXAYLWBH"
0012FA60  0012FA6C
0012FA64  00000000
```

```
Address   Hex dump                                          ASCII
00406584  44 49 4C 52 4E 43 46 43 47 58 41 59 4C 57 42 48  DILRNCFCGXAYLWBH
00406594  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004065A4  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

이것을 인증코드라고 쉽게 예상할 수 있다. 그리고 어디선가 이 값을 만들어 냈을 것이다. 일단 앞에서 진행했던 코드들을 계속 실행해 보겠다.

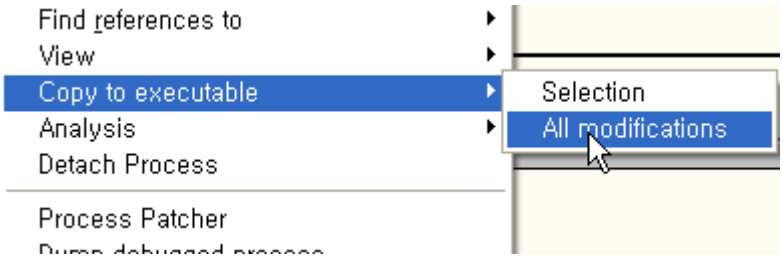
```
Address   Hex dump    Disassembly                          Comment
00401274  .  E8 B7000000  CALL <JMP.&user32.MessageBoxA>    MessageBoxA
00401279  .v EB 41        JMP SHORT keygenme.004012BC
0040127B  >  68 84654000  PUSH keygenme.00406584            String2 = "1234567890"
00401280  .  68 846B4000  PUSH keygenme.00406B84            String1 = "DILR-NCFC-GXAY-LWBH"
00401285  .  E8 E8000000  CALL <JMP.&kernel32.lstrcmpA>    lstrcmpA
```

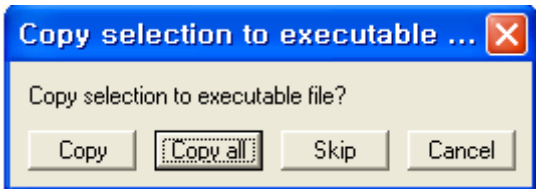
위를 보면 예상이 정확히 맞았음을 알 수 있다. 00401285에서 스트링을 비교하는 lstcmp함수를 호출하는데 인자값으로 사용자가 입력한 “1234567890”과 스택에서 보았던 “DILR-NCFC-GXAV-LWBH" 를 받아 들여서 비교하고 있는 것을 볼 수 있다. 그리고 OR EAX, EAX에 의해서 두 개의 인자 값이 같은지 확인하고 같다면 0040128E를 실행할 것이고 다르면 004012A8을 실행할 것이다. 이제 잘못된 인증 코드를 입력하더라도 올바른 킷값을 알아낼 수 있도록 키 값 자체를 메시지박스로 띄워 주도록 하겠다. 우선 기억을 더듬어서 처음 잘못된 인증 코드를 입력했을 때 Bad boy 메시지가 뜬것처럼 그 메시지 대신에 올바른 키 값을 출력한다면 큰 문제가 없을거라고 생각된다. 이제 고쳐야 할 Bad boy의 MessageBoxA함수는 4개의 인자를 받아들인다. style, Title, Text, h0wner 중에서 Title에 해당하는 00406337과 Text에 해당하는 00406318의 문자열들을 출력한다. 그럼 00406318에 있는 문자열을 키 값을 가진 주소로 변경하면 에러 메시지 대신 키 값을 간단하게 출력할 수 있을 것이다.

따라서 PUSH keygenme, 00406318 을 PUSH keygenme, 00406B84로 바꾸기만 하면 되는 것이다.

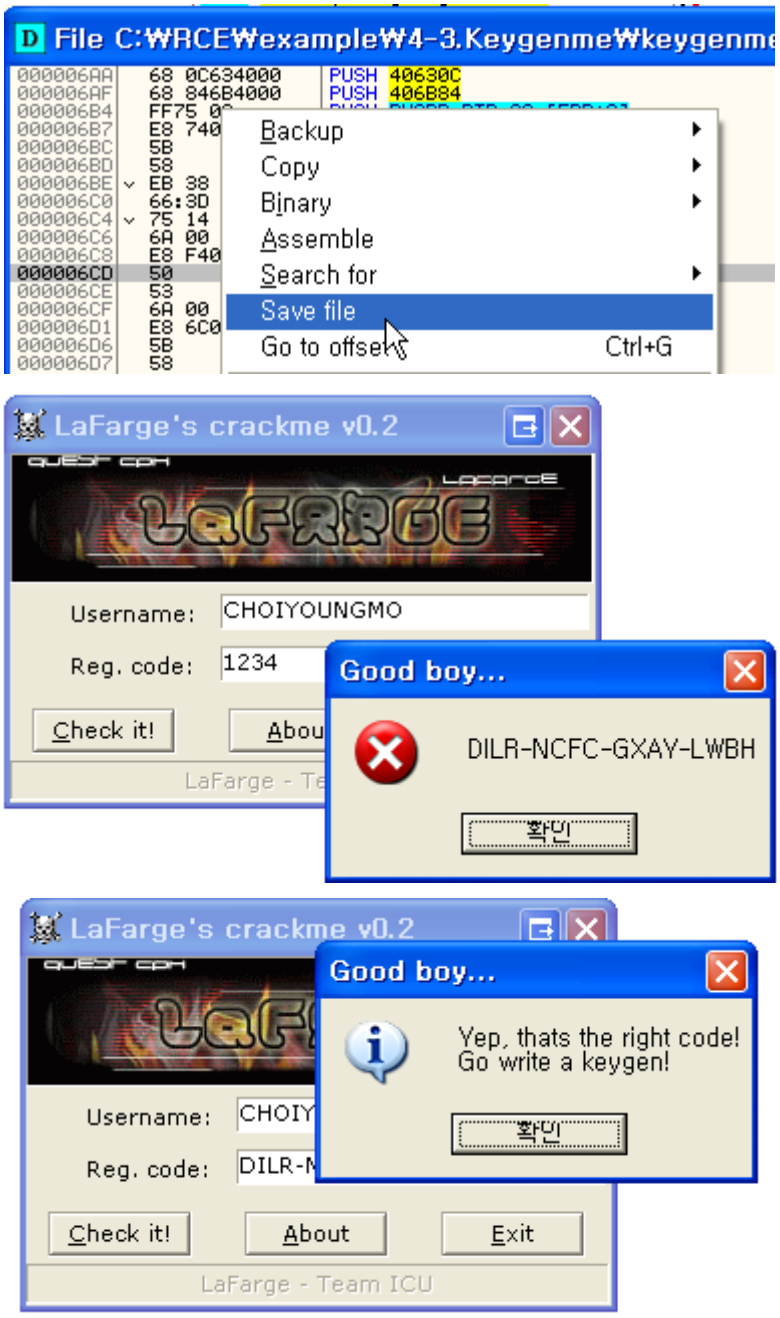
```
004012A3  .  C2 1000      RETN 10
004012A6  .v EB 14        JMP SHORT keygenme.004012BC
004012A8  >  6A 10        PUSH 10                          Style = MB_OK|MB_ICONHAND|MB_APPLMODAL
004012AA     68 0C634000  PUSH keygenme.0040630C            ASCII "Good boy..."
004012AF     68 846B4000  PUSH keygenme.00406B84            ASCII "DILR-NCFC-GXAY-LWBH"
004012B4  .  FF75 08      PUSH DWORD PTR SS:[EBP+8]        hOwner
004012B7  .  E8 74000000  CALL <JMP.&user32.MessageBoxA>   MessageBoxA
```

위와 같이 코드를 수정하고 아래처럼 Copy to executable에서 Allmodificaions를 선택하고 Copy all 하면 창이 새로 하나 뜬다.



그것을 Save file을 눌러서 다른 이름으로 저장한 후에 실행시켜서 아무값이나 입력하게 되면 원하는 키 값을 얻을수 있을 것이다.




이런 방법으로 내부 키젠을 생성할 수도 있고 또는 프로그램 내에서 키 값을 생성하는 알고리즘을 이용하여 별도의 키젠을 만들 수도 있다. 이상으로 초보자들을 위한 RCE문서는 여기까지 쓰도록 하겠다. 궁금한 점이나 더 많은 정보를 원한다면 메일로 문의하길 바란다.

# 5. 참고 문헌

[1]. http://www.jiniya.net/tt/524
[2]. 위키백과사전
[3]. http://dual5651.hacktizen.com/new/87
[4]. http://tuts4you.com
[5]. KUCIS Reverse Engineering