기 술 문 서

Reverse Engineering in the C++ Environment

정지훈
binoopang@is119.jnu.ac.kr

Abstract

C++로 작성된 프로그램이 어셈블리언어로 번역되었을 때 기본적인 문법에 대한 것들은 C언어와 비슷하지만 C++만이 가지는 특징은 리버스엔지니어링을 상당히 힘들게 만듭니다. C++에는 C언어가 가지고 있지 않은 객체지향이라는 언어적 특성이 추가 되었고, 여러 문법이 추가되었습니다.

우리가 어떤 프로그램을 분석할 때 이것이 C++ 언어를 사용하여 만든 프로그램인지 아닌지 알 수 있어야 하며, 만약 C++로 작성되었다면 C++의 특성을 염두해 두고 분석하는 것이 도움이 됩니다. 예로 구조체와 클래스차이를 알고 실제 분석 과정에서 이것이 클래스인지 구조체인지 구분할 수 있다면 분석하는 과정이 훨씬 수월 해 질것입니다. 더 나아가 클래스의 멤버함수가 어떻게 호출 되는지 그리고 이것이 가상함수인지 아닌지 구분하는 것도 도움이 될 수 있습니다.

윈도우에서 C++언어를 가장 잘 활용한 MFC의 경우 위에서 설명한 특징이 잘 반영되어 있기 때문에 C++의 특성을 잘 알게되면 MFC를 분석하는데 도움이 될 수 있습니다. MFC는 마이크로소프트에서 제공하는 클래스들을 사용하여 프로그래밍을 하는 것인데, 이 구조가 상당히 복잡합니다. 만약 MFC를 전혀 모르는데 MFC로 작성된 프로그램을 분석한다면 WinAPI로 작성된 매우 복잡한 프로그램을 분석하는 기분이 들 것입니다.

당연히 이 문서를 읽는데에도 MFC를 어느정도 알 고 있다면 읽기 수월할 것입니다. MFC의 구조를 알고 분석하는 것과 모르는 상태에서 분석하는 것은 큰 차이가 있을 것입니다.

Content

1. 목적

1.1. C++ 환경을 공부하는 이유

이 기술문서를 통해서 리버스엔지니어링에 대해서 좀 더 깊은 지식과 경험을 쌓을 수 있을 것이라 기대합니다. C와 C++은 서로 닮은 언어이고, 리버스엔지니어링의 대상의 프로그램들은 대부분 위 언어들 중 하나이기 때문에 떼어놓기 힘듭니다.

이 문서를 통하여 C++이 C언에 비해서 가지는 큰 특징들을 체계적으로 알아보고 이를 활용하여 MFC를 리버싱 하는것이 목적입니다. 사실 문서를 쓰는 본인도 C++과 리버스엔지니어링에 대해서 잘 모르기 때문에 공부한다는 생각으로 정리하고자 합니다.

1.2. 실습환경

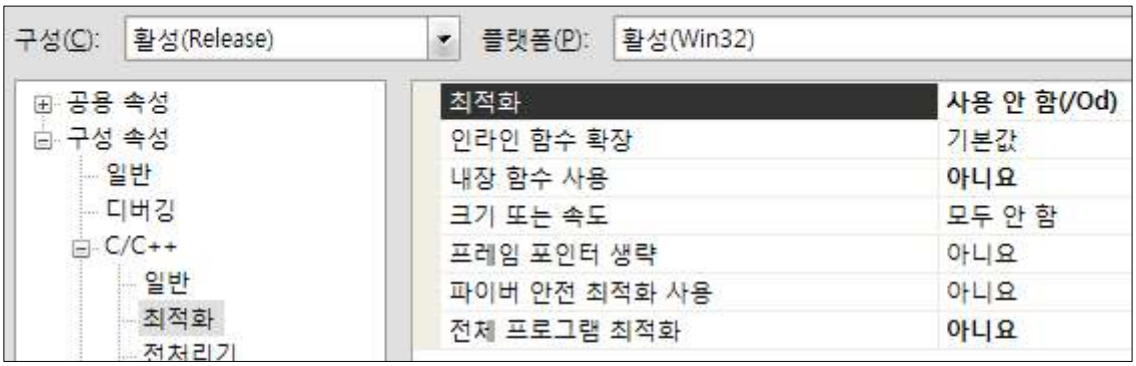
1.2.1. 사용한 도구들

이 기술문서에서 디스어셈블링과 컴파일, 디버깅등에 사용된 도구들은 아래와 같습니다.

- 운영체제 : Windows XP ServicePack 3
- 컴파일러 : VisualStudio2008 (VC9.0)
- 디스어셈블러 : IDA Pro 5.2(Hexray1.0), OllyDBG 1.3
- 디버거 : OllyDBG 1.3

1.2.2. 컴파일 옵션

소스코드를 컴파일할 때 코드수준에서 자세한 분석을 위하여 최적화를 하지 않았습니다.



[그림 2] 최적화 옵션

2. 어셈블리 수준에서 C++ 언어의 특징

2.1. 간단한 예제를 통한 C++ Disassembling

2.1.1. 소스코드

객체지향 언어인 C++이 가지는 가장 큰 특징은 역시 클래스입니다. 물론 C언어에서도 클래스를 구조체를 통하여 흉내낼 수 있지만 그럴 바엔 C++언어를 사용하여 프로그래밍하는 것이 더 빠를 것입니다.

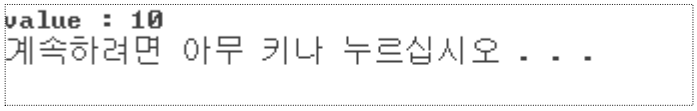
다음 소스코드는 클래스를 사용하는 간단한 예를 보입니다.

```
#include "cpp_reverse.h"
#include <stdio.h>

class CBasic_class
{
public:
        CBasic_class();
        void print_value();

private:
        int m_value
        char buffer[0x80];
};

int main()
{
        CBasic_class basic
        basic.print_value();
        return 0;
}

CBasic_class::CBasic_class()
{
        memset(buffer, 'a', 0x80);
        m_value=10;
}

void CBasic_class::print_value()
{
        printf("value : %d₩n", m_value);
}
```

[소스코드 1] 간단한 c++ 클래스 예제

(소스코드 1)은 간단한 C++ 클래스 예제입니다. 클래스는 생성자와 보호된 변수를 출력해

주는 멤버 함수를 가지고 있습니다. main()에서는 이 클래스 객체를 하나 생성한 뒤 print_value() 멤버변수를 호출합니다. 결과는 쉽게 예상할 수 있을 것입니다.



[그림 3] (소스코드 1)의 실행결과

생성자에서 m_value값을 10으로 초기화 하였기 때문에 print_value()는 m_value가 가지고 있는 10이라는 값을 출력하였습니다.

2.1.2. 디어셈블링

이제 (소스코드 1)로 컴파일한 프로그램에서 main() 부분을 디어셈블링 해보겠습니다. 주로 사용하는 디스어셈블러로써 올리디버거와 IDA pro가 있는데 여기서는 둘 다 사용해 보겠습니다.

- IDA를 사용한 Disassembling 결과

```
; int __cdecl main(int argc, const char **argv, const char *envp)
_main           proc near               ; CODE XREF: ___tmainCRTStartup+10A↓p

var_90          = byte ptr -90h
var_4           = dword ptr -4
argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                sub     esp, 90h
                mov     eax, __security_cookie
                xor     eax, ebp
                mov     [ebp+var_4], eax
                lea     ecx, [ebp+var_90]
                call    ??0CBasic_class@@QAE@XZ ; CBasic_class::CBasic_class(void)
                lea     ecx, [ebp+var_90]
                call    ?print_value@CBasic_class@@QAEXXZ ; CBasic_class::print_value(void)
                xor     eax, eax
                mov     ecx, [ebp+var_4]
                xor     ecx, ebp
                call    @__security_check_cookie@4 ; __security_check_cookie(x)
                mov     esp, ebp
                pop     ebp
                retn
_main           endp
```

[그림 4] IDA를 사용한 Disassembling 결과

IDA pro가 디버거는 몰라도 디스어셈블러는 최고라는 것에 대해서 불만 가질 사람은 거의 없을 것 같습니다. (그림 4)을 보면 상당히 보기 좋게 어셈블리언어로 번역된 것을 확인할 수 있습니다.

컴파일러에 따라서 __security_cookie부분은 있을 수도 있고 없을 수도 있습니다. 이것은 Buffer Overflow와 같은 취약점을 막기 위하여 도입된 것으로 VisualStudio2008에서 컴파일

할 경우 위와 같이 존재하는 것을 확인할 수 있었습니다. 위의 코드에서 실제 우리가 관심을 가져야 할 부분은 다음과 같습니다.

```
; int __cdecl main(int argc, const char **argv, const char *envp)
_main           proc near               ; CODE XREF: ___tmainCRTStartup+10A↓p

var_90          = byte ptr -90h
var_4           = dword ptr -4
argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                sub     esp, 90h
                mov     eax, __security_cookie
                xor     eax, ebp
                mov     [ebp+var_4], eax
                lea     ecx, [ebp+var_90]
                call    ??0CBasic_class@@QAE@XZ ; CBasic_class::CBasic_class(void)
                lea     ecx, [ebp+var_90]
                call    ?print_value@CBasic_class@@QAEXXZ ; CBasic_class::print_value(void)
                xor     eax, eax
                mov     ecx, [ebp+var_4]
                xor     ecx, ebp
                call    @__security_check_cookie@4 ; __security_check_cookie(x)
                mov     esp, ebp
                pop     ebp
                retn
_main           endp
```

(그림 5) 클래스와 관련된 주요 루틴

위에서 블록으로 표시한 부분은 클래스와 관련된 부분이기 때문에 중요합니다. 아래에서 자세히 설명하겠지만 클래스를 위한 공간을 할당하고 멤버 함수를 호출하는 부분입니다.

- 올리디버거를 사용한 Disassembling

아래는 올리디버거를 사용하여 디어셈블링한 결과입니다.

```
00401000  $  55              PUSH EBP
00401001  .  8BEC            MOV EBP,ESP
00401003  .  81EC 90000000   SUB ESP,90
00401009  .  A1 00304000     MOV EAX,DWORD PTR DS:[403000]
0040100E  .  33C5            XOR EAX,EBP
00401010  .  8945 FC         MOV [LOCAL.1],EAX
00401013  .  8D8D 70FFFFFF   LEA ECX,[LOCAL.36]
00401019  .  E8 22000000     CALL cpp_reve.00401040
0040101E  .  8D8D 70FFFFFF   LEA ECX,[LOCAL.36]
00401024  .  E8 37000000     CALL cpp_reve.00401060
00401029  .  33C0            XOR EAX,EAX
0040102B  .  8B4D FC         MOV ECX,[LOCAL.1]
0040102E  .  33CD            XOR ECX,EBP
00401030  .  E8 4A000000     CALL cpp_reve.0040107F
00401035  .  8BE5            MOV ESP,EBP
00401037  .  5D              POP EBP                    cpp_reve.004011E8
00401038  .  C3              RETN
```

[그림 6] 올리디버거를 사용한 Disassembling 결과

올리디버거는 디버거로서는 최고의 역할을 수행하지만 디스어셈블러로는 IDA보다는 그 역할이 부족한것 같습니다. (그림 6)은 (그림 4)와 동일한 부분의 디어셈블링 결과 인데 (그림 6)을 봐서는 이것이 과연 C++언어로 작성된 것인지 C로 작성된 것인지 구분하기가 쉬운일이 아닙니다.

2.2. Class의 생성과 구조

2.2.1. 구조체 변수가 생성되는 방법

클래스가 객체가 생성되는 과정을 보기 전에 이와 유사한 구조체 변수가 할당되는 순서를 확인해 보겠습니다.

```
#include <stdio.h>

struct Number
{
        int m_nOne
        int m_nTwo
        int m_nThree
        char buffer[0x32];
};

int main()
{
        struct Number numbers

        numbers.m_nOne = 1;
        printf("m_nOne : %dWn", numbers.m_nOne);

        return 0;
}
```

[소스코드 2] 간단한 구조체 예제

(소스코드 2)는 간단한 구조체를 선언하고 멤버변수에 값을 대입한 후 출력까지 하는 예제입니다. 매우 간단합니다. 이제 이것을 IDA를 사용하여 디어셈블링 해보겠습니다.

```
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
argc            = dword ptr  8
argv            = dword ptr  0Ch
envp            = dword ptr  10h

                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                mov     [ebp+var_C], 1
                mov     [ebp+var_8], 2
                mov     [ebp+var_4], 3
                xor     eax, eax
                mov     esp, ebp
                pop     ebp
                retn
_main           endp
```

[그림 7] 구조체 변수의 생성과 값 할당

(그림 7)의 Disassembling결과에서 가장먼저 확인해야 할 것은 스택의 확장 부분입니다. 스택의 확장은 곧 지역 변수를 선언하는 것과 같은 의미를 가집니다. 때문에 위 그림처럼 0xc만큼 스택을 확장하는 것은 0xc크기의 구조체 변수를 선언하는 것과 일치합니다.

실제로 (소스코드 2)에서 확인할 수 있듯이 구조체는 3개의 int형 변수를 가지기 때문에 전체 크기는 12 즉 0xc입니다. 지역변수의 할당이 완료되면 ebp를 기준으로 오프셋을 참조하여 정해진 위치에 값을 넣는 것을 볼 수 있습니다. 이것은 (소스코드 2)에서 멤버변수에 값을 할당하는 것을 말합니다.

결론적으로 구조체를 사용하기 위해서는 먼저 구조체 전체 크기에 해당하는 공간을 할당한 후에 오프셋을 기준으로 값을 대입합니다.

2.2.2. 클래스 객체(인스턴스)가 생성되는 방법

- (소스코드 1)의 클래스 생성

사실 클래스와 구조체는 매우 유사한 부분이 많습니다. 어셈블리 수준에서 확인하면 거의 다를 것이 없습니다.

```
mov     ebp, esp
sub     esp, 90h
mov     eax, __security_cookie
xor     eax, ebp
mov     [ebp+var_4], eax
lea     ecx, [ebp+var_90]
call    ??0CBasic_class@@QAE@XZ ; CBasic_class::CBasic_class(void)
lea     ecx, [ebp+var_90]
call    ?print_value@CBasic_class@@QAEXXZ ; CBasic_class::print_value(void)
xor     eax, eax
mov     ecx, [ebp+var_4]
```

[그림 8] 클래스 객체의 할당

구조체와 마찬가지로 클래스도 클래스 크기만큼의 공간을 확보합니다. 여기서 클래스의 크기란 (소스코드 1)의 경우 변수만을 의미합니다. 클래스 객체에는 가상 함수의 특수한 경우를 제외하고 멤버함수는 오프셋을 기반으로 호출하기 때문에, 객체에는 함수의 주소와 같은 정보는 담겨져 있지 않습니다.

위 설명은 클래스 객체를 정적으로 사용한 경우입니다. 그렇다면 만약 클래스를 동적으로 생성할 경우에는 어떨지 보겠습니다.

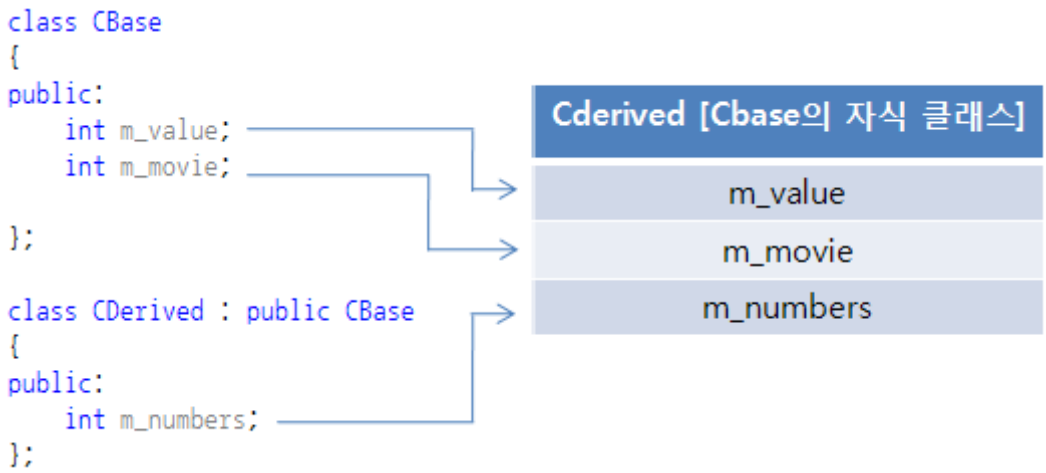
```
push    84h             ; unsigned int
call    ??2@YAPAXI@Z    ; operator new(uint)
add     esp, 4
mov     [ebp+var_18], eax
mov     [ebp+var_4], 0
cmp     [ebp+var_18], 0
jz      short loc_401056
mov     ecx, [ebp+var_18]
call    ??0CBasic_class@@QAE@XZ ; CBasic_class::CBasic_class(void)
mov     [ebp+var_20], eax
jmp     short loc_40105D
```

[그림 9] 동적 클래스 객체 생성의 경우

new 연산자를 사용하여 클래스 객체를 생성하는 경우입니다. 스택에 클래스 공간을 확보했던 것과는 달리 첫 번째 블록에서 클래스 크기를 new의 인자로 넣고 할당하는 것을 확인할 수 있습니다. 그리고 나머지의 경우 정적인 클래스 선언과 멤버함수 호출하는 방법에 있어서는 동일하고 함수 호출에 대한 특성은 아래에서 다시 보겠습니다.

- 상속받은 클래스의 경우?

일반적으로 상속받지 않은 클래스는 위에서 알아본 것 처럼 일반 구조체와 동일하게 생성되는 것을 보았습니다. 하지만 객체지향 프로그래밍을 할 때 우리는 클래스를 주로 상속해서 많이 사용합니다. 이렇게 되면 상속받은 클래스는 부모 클래스의 속성을 물려받게 됩니다. 어셈블리 수준에서도 마찬가지입니다. 만약 int형 변수 2개를 가진 부모 클래스를 물려받은 자식 클래스에 int형 변수 1개를 더 추가하였다면 자식 클래스는 총 int형 변수 3개를 가진것이므로 (다른 속성이 없다면)총 12바이트의 크기를 가지게 됩니다.



[그림 10] 상속받은 클래스의 메모리 구조

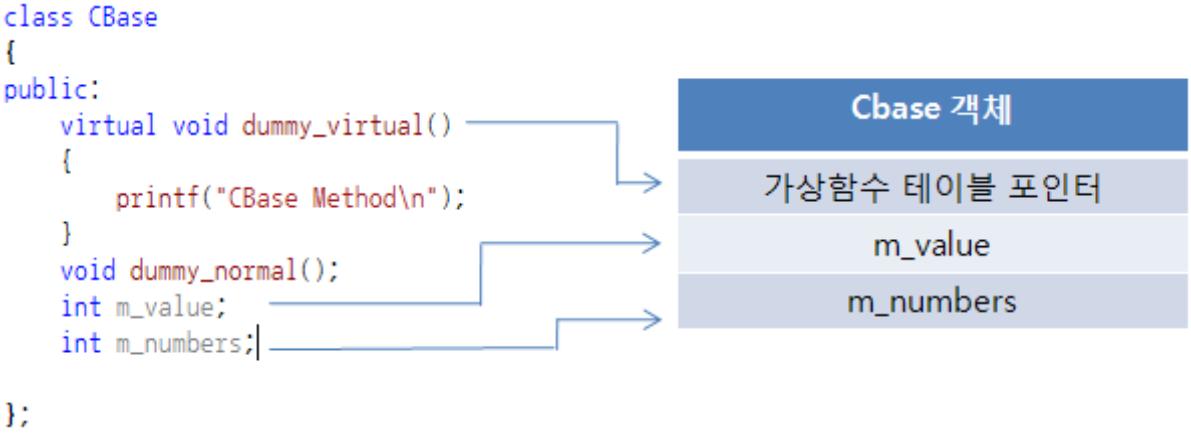
(그림 10)와 같은 메모리 구조를 가지게 되는데, 만약 CDerived를 상속받은 클래스가 또 있고, 그 클래스가 변수를 추가했다면 (그림 10)보다 더 복잡해 질 것입니다. 단순히 클래스상속에 대해서 알고 있다면 쉽게 이해할 수 있을 것입니다.

2.2.3. 클래스의 구조

2.2.2.에서 객체가 생성될 때 객체의 크기는 변수의 크기라고 하였습니다. 또한 객체에는 (가상함수를 제외한)멤버함수에 대한 정보가 없다고 언급하였습니다. 이 말은 곧 객체는 저장공간과 참조를 위한 공간임을 의미합니다.

실제로 클래스 멤버함수를 호출할 때 객체를 참조하여 호출하지 않습니다. 다만 호출할 때 객체의 주소를 넘겨줍니다. 그렇다면 어떻게 멤버함수의 주소를 알 수 있을까요? 이것은 간단하게 컴파일할 때 주소가 결정되기 때문에 간접적인 주소 계산이 필요가 없는 것입니다. 결국 클래스의 멤버함수가 아닌 일반 함수와 똑같습니다.

하지만 클래스 멤버함수가 가상함수라면 이야기가 조금 달라집니다. 위의 내용에 가상함수 테이블의 포인터가 들어가게 됩니다. 가상함수에 대해서는 아래에서 다시 설명하겠지만 다형성에 있어서 매우 중요한 부분입니다. 가상함수가 사용되면 필연적으로 가상함수 테이블이 생성되고 이 포인터는 객체에 저장이 됩니다.



[그림 11] 가상함수가 존재할 때의 객체 구조

어셈블리 수준에서 가상함수에 대해서는 다음 장에서 다시 설명하도록 하겠습니다.

2.3. Class 멤버 함수들의 특징

2.3.1. 클래스 함수들의 독특한 호출 방법

2.2.까지 보았을 때 구조체와 클래스는 서로 다른점이 별로 없어 보입니다. 이 상황에서 클래스를 분별하기 위하여 가장 큰 다른 점을 꼽으라면 역시 멤버함수 호출부분입니다.

C++에서 멤버함수를 호출 할때는 항상 클래스 객체 포인터인 this 변수가 따라갑니다. 하지만 이 포인터는 실제로 멤버함수를 호출할 때 인자로 넣어주지 않습니다. 그 이유는 항상 멤버함수를 호출 할때 this를 인자로 넘겨줘야 하므로, 처음부터 C++설계 할 때 묵시적으로 this를 넘기기로 약속한 것입니다. 이때 this를 스택을 사용하여 넘기는 게 아니라 레지스터를 사용하여 넘깁니다.

이러한 호출규약을 **__thiscall**[1]이라고 하는데, 올리디버거로 분석할 때는 호출규약까지 보여주지 않지만 IDA를 사용하면 호출규약까지 보여주므로 쉽게 확인이 가능합니다.

```
.text:00401070
.text:00401070 ; public: __thiscall CBase::CBase(void)
.text:00401070 ??0CBase@@QAE@XZ proc near              ; CODE XREF: _main+1C1p
.text:00401070
.text:00401070 var_4           = dword ptr -4
```

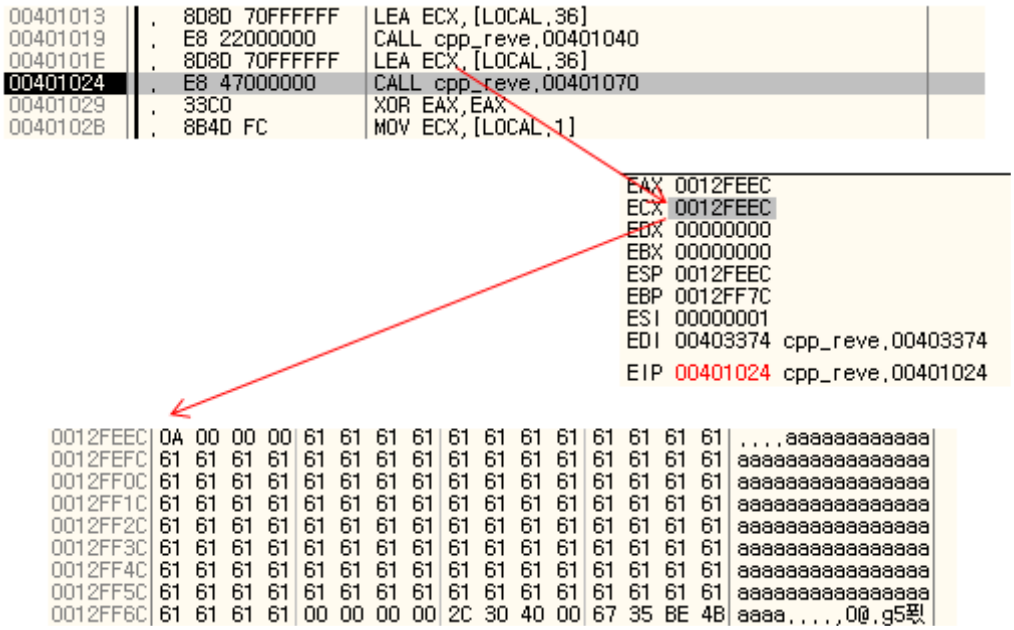
[그림 12] 생성자의 함수 타입 정보

(그림 12)와 같이 생성자의 __thiscall 이라는 호출규약을 확인할 수 있습니다.

1) __thiscall : this 인자가 같이 전달되는 것을 제외하고 이 호출규약은 __cdecl과 동일합니다.

(그림 13)에서 call 명령어가 나오기 전에 lea 명령어를 사용하여 ecx 레지스터에 주소를 저장하는 것을 확인할 수 있는데 ecx에는 객체 포인터가 들어갑니다.

올리디버거를 통하여 확인해 보겠습니다.



[그림 13] 클래스 객체 포인터를 가지고 있는 ecx 레지스터

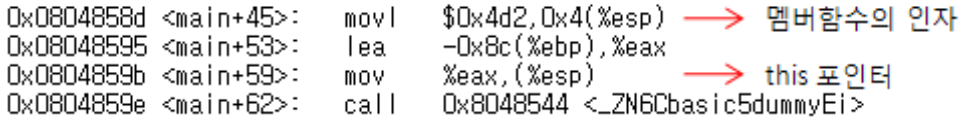
(그림 13)을 통하여 함수 멤버함수를 호출하기 전에 ecx 레지스터로 객체 포인터가 저장됨을 알았습니다. 물론 모든 컴파일러가 레지스터를 사용하는 것은 아닙니다. gcc컴파일러의 경우 레지스터가 아닌 스택을 사용하는데 가장 상위 스택, 즉 첫 번째 인자로 객체 포인터를 전달합니다.

```
void Cbasic::dummy(int number)
{
    printf("number : %d₩n", number);
}
```

[소스코드 3] gcc로 컴파일 될 멤버함수

(소스코드 3)은 gcc로 컴파일 할 멤버함수입니다. 인자가 하나기 때문에 VC컴파일러를 사용할 경우 스택에 인자가 하나만 전달 되겠지만 gcc의 경우 스택에 인자를 2개 넘깁니다.

하지만 객체포인터가 매번 저장되는 모양은 비슷하기 때문에 쉽게 멤버 함수라는 것을 식별할 수 있습니다.



[그림 15] gcc로 컴파일 한 경우 this 포인터 전달 방법

(그림 15)에서 볼 수 있듯이 객체 포인터는 항상 최상위 스택을 사용하여 전달하고 있습니다.

2.3.2. Class 객체 생성자(Constructor)와 소멸자(파괴자)(Destructor)

Class의 생성자와 소멸자는 선택 가능한 함수입니다. 즉 사용해도 되고 안해도 된다는 이야기입니다. 보통의 객체지향 프로그래밍에서 생성자와 소멸자는 초기화 작업과 정리 작업을 주로 맡아서 역할 하기 때문에 자주 쓰인다고 봐야 할 것입니다.

- 생성자의 특징

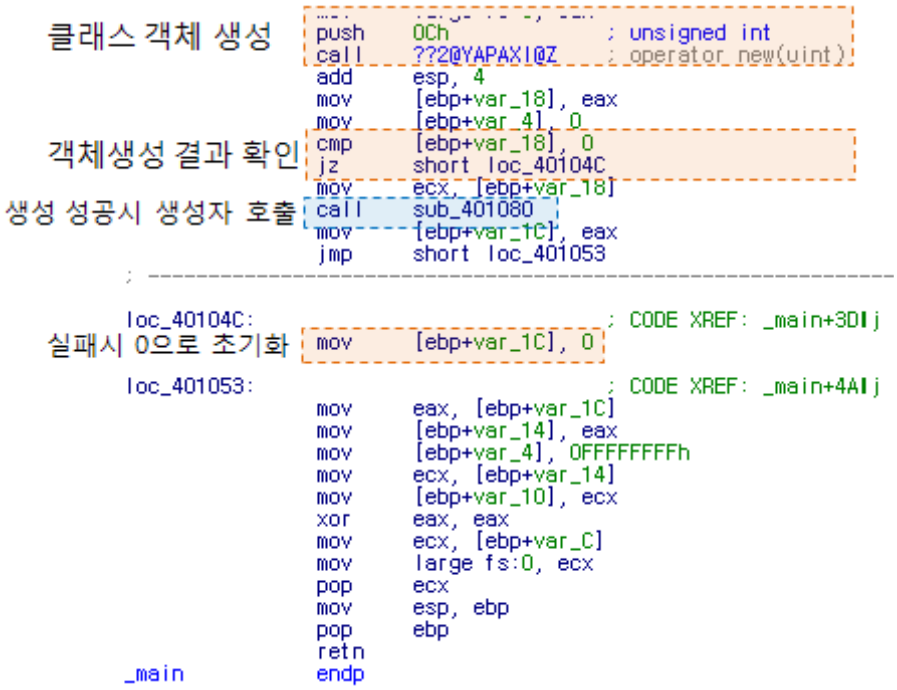
생성자는 클래스의 함수중 (생성자가 존재한다면) 가장 먼저 호출 되는 함수입니다. 이 생성자의 가장 큰 특징은 아마도 리턴이 없는 것과 자동으로 호출된다는 정도 일 것입니다. 하지만 이런 특징들만을 가지고 어셈블리 수준에서 이것이 생성자라고 판단하기는 힘들어 보입니다. 위에서 열거한 특징이 오직 생성자만의 특징은 아닐 것이기 때문입니다.

하지만 생성자만의 독특한 특징이 한 가지 더 존재합니다. 그것은 생성자가 예외를 일으키지 않는다는 것입니다. 그리고 이 특징은 클래스 객체가 동적으로 생성되었을 때 생성자를 쉽게 식별할 수 있게 해줍니다.

```
#include <stdio.h>

class CBase
{
public:
        CBase(){
                m_value = 5;
        }
        virtual void dummy_virtual()  // 가상함수
        {
                printf("CBase MethodWn");
        }
        void dummy_normal();                    // 일반멤버함수
        int m_value
        int m_numbers

};

int main()
{
        CBase *pBase
        pBase = new CBase

        return 0;
}
```

[소스코드 4] 생성자를 가지는 객체 생성

(소스코드 4)는 생성자를 가지는 클래스의 객체를 동적으로 생성하는 프로그램입니다. 위 프로그램을 디어셈블링해서 생성자가 호출되는 특징을 살펴 보겠습니다.



[그림 16] 생성자 호출 과정

생성자의 호출은 위와 같습니다. 다시 정리하면 다음과 같습니다.

- new를 사용하여 클래스 객체 동적할당을 시도한다.
- new의 반환 값[객체의 포인터]이 eax레지스터에 저장되고 그 값이 NULL인지 비교한다.
- 만약 NULL이라면 생성자 호출을 하지 않고 객체 포인터를 초기화 한다
- 만약 NULL이 아니라면 생성자를 호출한다.

위에서 생성자는 예외처리를 하지 않는다고 했습니다. 따라서 생성자에 진입하기 전에 미리 예외가 일어날 상황[객체가 성공적으로 생성되었는지]을 확인하고, 생성자를 호출하는 것입니다. 따라서 위와 같은 루틴을 확인한다면 생성자라는 것을 추측할 수 있습니다.

- 소멸자의 특징

소멸자도 생성자와 마찬가지로 선택가능한 함수입니다. 만약 소멸자가 존재한다면 클래스의 함수들 중에 가장 마지막으로 호출되는 함수일 것입니다.

생성자에서 그랬던것 처럼 소멸자의 특징에 대해서 생각해 봐야합니다. 소멸자도 자동으로 호출되는 함수이며 리턴 타입이 없습니다. 하지만 역시 이 특징으로는 무언가 부족합니다. 좀더 이게 소멸자 이겠구나 하는 확실한 특징을 찾아야 합니다. 이 특징은 생성자의 특

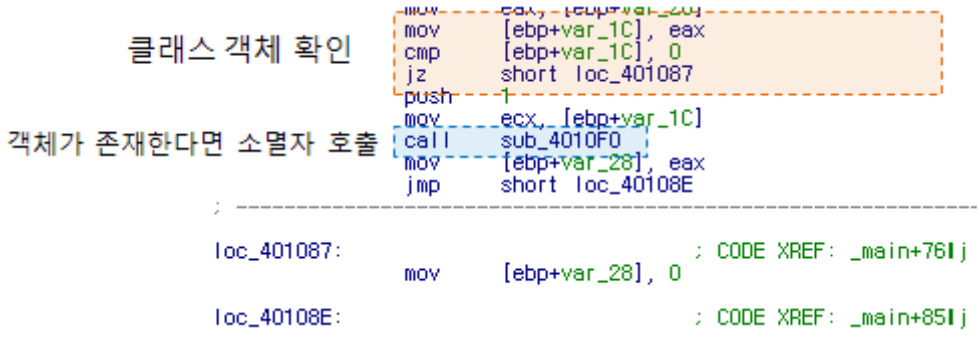
징과 비슷합니다.

생성자가 객체가 생성되고 그것이 확인되면 생성자를 호출 했던 것 처럼 소멸자는 객체가 확실히 생성되어 있는 상태라면 호출됩니다. 다음 소스코드를 확인 해 보겠습니다.

```
#include <stdio.h>

class CBase
{
public:
        CBase(){
                m_value = 5;
        }
        ~CBase(){
                m_value = 0;
        }
        virtual void dummy_virtual()  // 가상함수
        {
                printf("CBase Method₩n");
        }
        void dummy_normal();                            // 일반멤버함수
        int m_value
        int m_numbers

};

int main()
{
        CBase *pBase
        pBase = new CBase
        delete pBase

        return 0;
}
```

[소스코드 5] 소멸자를 가지는 객체 생성

위 (소스코드 5)에서 객체를 생성한 뒤 바로 delete를 사용해서 객체를 해제 하였습니다. 당연히 delete를 사용하지 않으면 소멸자는 호출되지 않을 것입니다.

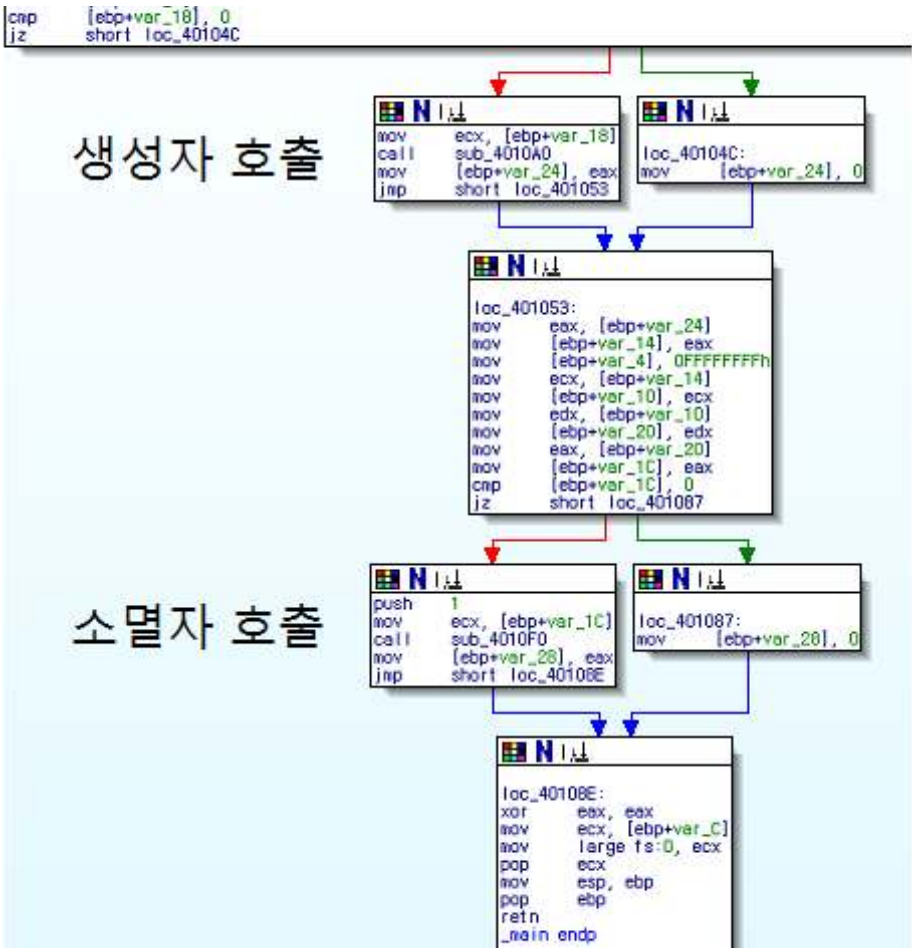


[그림 17] 소멸자 호출 과정

다시 정리하면 다음과 같습니다.

◦ 생성된 객체의 포인터를 검사하여 NULL인지 비교한다.
◦ 만약 존재한다면 소멸자를 호출한다.
◦ 존재하지 않는다면 소멸자 호출을 건너뛴다.

생성자와 매우 유사한 것을 확인할 수 있습니다. 아래 그림은 IDA를 사용하여 코드블럭의 흐름을 그래프로 그린 것입니다.



[그림 18] 생성자와 소멸자 호출 코드블럭

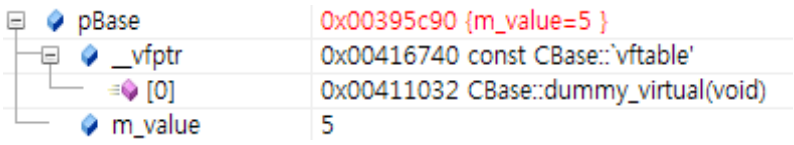
다른 점이라면 생성자는 바로위에 new 연산자가 나와야합니다. 또한 NULL 포인터 검사가 위 아래로 모두 존재해야 합니다. 따라서 객체에 생성자 또는 소멸자 한 가지만 있을 때에도 구분이 가능합니다.

2.3.3. 가상함수(Virtual Function)의 특징

가상함수는 매우 독특한 함수인 것을 알고 계실 것입니다. 이 함수는 정적타입이 아닌 포인터가 실제로 가리키고 있는 동적 타입에 의해서 멤버 함수를 호출합니다. 때문에 프로그램 실행 중에 결정되는 함수라고 정의합니다. 가상함수의 이론에 대해서는 모두 알고 있다고 가정하고 이 가상함수가 어셈블리언어 수준에서 어떻게 구현되는지가 중요합니다.

- 가상함수 테이블 포인터의 위치

먼저 가상함수 테이블이 어디에 있는지 살펴보겠습니다. 클래스의 구조에서 가상함수가 한 개라도 사용되면 클래스에는 가상함수 테이블 포인터가 저장된다고 언급하였습니다. 그렇다면 실제로 이 포인터를 먼저 확인해 보겠습니다. (소스코드 5)를 컴파일 하고 비쥬얼스튜디오 디버거를 사용하여 가상함수 테이블을 확인하겠습니다.



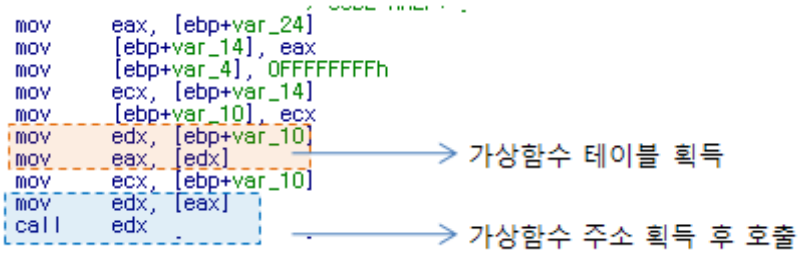
[그림 19] 가상함수가 존재할 때의 클래스 구조

함수를 제외한 변수는 m_value하나 뿐이므로 가상함수가 없었다면 m_value에 해당하는 int형 공간만 있어야 하지만 가상함수가 사용되었기 때문에 (그림 19)의 _vfptr이라는 가상함수 테이블 포인터가 들어가 있습니다. 그리고 그 포인터를 따라가면 실제 클래스에 존재하는 가상함수 목록을 확인할 수 있는데 여기서는 하나의 함수만 존재합니다. 만약 여러개의 가상함수가 존재했다면 여러개의 함수의 주소가 테이블에 기록되어져 있을 것입니다.

결국 가상함수 테이블 주소는 객체에 저장되어 있습니다. 이는 가상함수를 호출할 때 객체를 통해서 주소를 얻어야 하기 때문에 당연한 것입니다.

- 가상함수가 호출되는 과정

가상함수 테이블 포인터의 위치를 알고 있으니 이제 가상함수가 실제로 어떤 과정을 거쳐서 호출되는지 알아보겠습니다.



[그림 20] 가상함수 호출

(그림 20)은 가상함수가 호출되는 과정을 보인 것입니다. VC로 컴파일 했을 경우 가상함수는 모두 edx레지스터에 가상함수 주소를 받아와서 호출하고 있었습니다. 물론 클래스 함수들이 호출될 때의 특징인 ecx레지스터에 객체 주소가 들어가는 것은 일치합니다.
호출과정을 정리하면 다음과 같습니다.

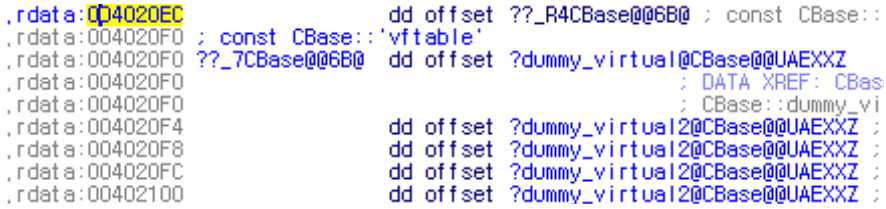
- 객체 주소를 edx레지스터에 저장한다.
- 객체에 저장된 가상함수 테이블 포인터를 eax레지스터로 가져온다.
- 객체 주소를 ecx레지스터에 넣는다.
- 가상함수 테이블 포인터가 저장된 eax레지스터로부터 가상함수 주소를 가져와 edx레지스터에 넣는다.
- edx레지스터에 저장된 주소를 호출한다.

어셈블리 줄 단위로 정리해서 다소 복잡해 보이지만 사실 간단한 과정입니다. 중요한 것은 가상함수 테이블을 가져올 때 객체로부터 그 주소를 가져온다는 사실이고, 테이블로부터 가상함수주소를 레지스터에 저장하고 호출한다는 것입니다.
가상함수의 묘미인 객체가 다른 타입의 클래스를 가리킬 땐 어떨까요? 당연히 다른 객체의 포인터를 가지기 때문에 해당 객체의 가상함수 테이블을 얻게 될 것이고 해당 객체의 가상함수 포인터를 얻을 수 있습니다.
마지막으로 가상함수 테이블의 위치공간을 아는 것은 역 분석 과정에서 해당 함수가 가상함수라는 것을 쉽게 알 수 있게 해 줍니다.

```
class CBase
{
public:
        virtual void dummy_virtual()  // 가상함수
        {
                printf("CBase Method₩n");
        }
        virtual void dummy_virtual2();
        virtual void dummy_virtual3();
        virtual void dummy_virtual4();
        virtual void dummy_virtual5();
};
```

[소스코드 6] 여러 가상함수를 가지는 클래스

(소스코드 6)과 같이 가상함수를 여러 개 사용하는 클래스를 예로 들었을 때, 실제 가상함수 주소를 가지고 있는 테이블은 .rdata에 저장되어 있습니다. 따라서 (그림 19)와 같이 레지스터에 함수 포인터를 가져오는데 .rdata에서 가져온다면 이는 가상함수라고 생각할 수 있습니다.



[그림 21] 가상함수 테이블

이처럼 가상함수 테이블은 .rdata 주소공간에 있으며 고정된 주소를 가지고 있습니다. 따라서 쉽게 어떤 클래스의 모든 가상함수 목록을 확인할 수 있으며 그들의 주소 또한 알 수 있습니다.

3. MFC 리버스엔지니어링

3.1. MFC로 작성된 프로그램의 특징

3.1.1. MFC 역분석의 어려운 점

MFC(Microsoft Foundation Classes)로 만들어진 GUI프로그램들은 객체지향 언어로 설계되어 만들어 진 만큼 C++의 특징을 그대로 가지고 있으며, 콘솔프로그램보다 매우 복잡한 구조를 가집니다.. 또한 메시지를 사용한 이벤트 드리븐(Driven)방식이기 때문에 이벤트를 처리하는 프로시져들이 모여있습니다.

기타 언어로 만들어진 프로그램들도 마찬가지 이지만 GUI방식의 프로그램을 역분석할 때 가장 어려운 점은 원하는 코드를 찾기가 어렵다는 것입니다. 이것은 프로그램이 Graphic Interface를 유지하기 위한 코드와 이벤트를 처리하기 위한 많은 루틴들이 뒤섞여 있기 때문일 것입니다.

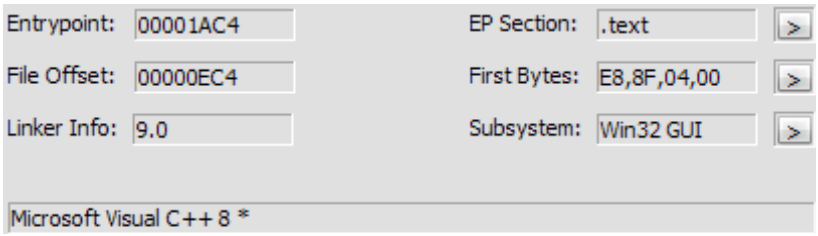
3.1.2. 관심 가질 필요가 있는 부분

우리가 MFC를 분석한다고 해서 MFC로 작성된 프로그램의 모든 루틴에 관심을 가지는 것은 아닐 것입니다. 특히 우리가 관심가져야 할 부분은 아마도 프로그램이 초기화되면서 어떤 정보들이 세팅되는지와, 특정 버튼을 눌렀을 때 해당 이벤트를 처리하는 함수들의 위치를 아는 것일 것입니다.

사실 이것들만 분석하면 MFC로 작성된 프로그램을 이해하는데 큰 어려움이 없을 것입니다.

3.1.3. MFC로 작성된 프로그램의 식별 방법

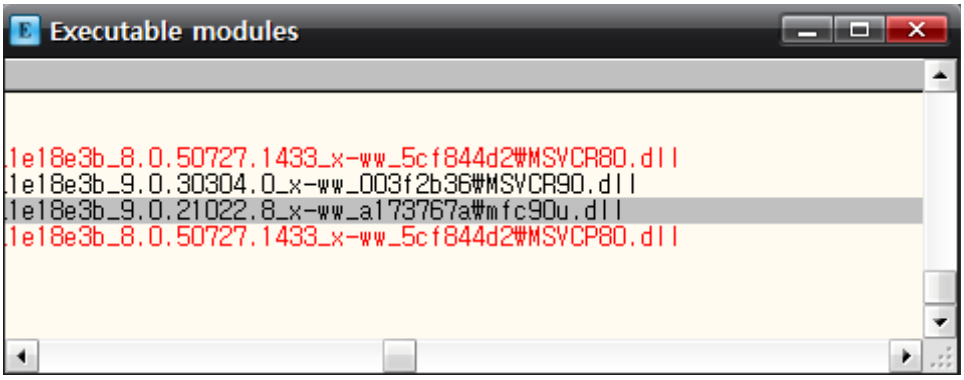
MFC로 작성된 프로그램은 겉으로 봤을 땐 WinAPI만으로 작성된 GUI프로그램이나 델파이로 작성된 프로그램과 큰 차이가 없습니다. PE파일을 시그니쳐 기반으로 어떤 언으로 개발되었는지 알 수 있게 해주는 프로그램이 있습니다. 대표적으로 PEiD가 있는데, MFC로 작성한 프로그램을 불러와 보겠습니다.



[그림 22] PEiD 분석 결과

PEiD를 사용하여 파일을 검사한 결과 Microsoft Visual C++ 8을 사용하여 작성되었다는 결과를 보여주었습니다. 틀린말은 아니지만, 결정적으로 MFC라는 말은 없습니다.

사실 이런 방법보다 간단하게 디버거로 불러와서 mfc 관련 라이브러리를 로드하는지 안하는지를 사용하여 판단하면 간단합니다.



[그림 23] 로드된 mfc90u.dll

Visual Studio 2008을 사용하여 MFC 프로그램을 컴파일하면 기본적으로 mfc90u.dll이 로드됩니다. 이런식으로 라이브러리를 확인하여 이 프로그램이 MFC를 사용하여 작성된 프로그램인지 식별 가능합니다.

3.2. MFC 초기화 함수 분석

3.2.1. App::InitInstance() 함수

InitInstance()는 CWinAPP 클래스의 멤버함수입니다. 보통 우리는 이 함수를 오버라이딩해서 코드를 추가해서 사용합니다. 실제로 이 함수가 사용하는 예에 대해서 최호성씨가 쓴 “Windows Programming"이라는 책에는 다음과 같은 내용이 있습니다.

> **App::InitInstance()**
>
> MFC에서 이 함수의 역할은 매우 중요하다. 응용 프로그램이 초기화되는 부분이며, 이와 관련한 코드들이 집결되는 곳이다. 그러므로 여러분들은 이 부분에 코딩할 일이 상당히 많이 있다. 다음의 것들이 아주 대표적인 예이다.
>
> ◦ 응용프로그램의 설정 정보 로딩(예 : 윈도우 크기, 옵션, 스타일등)
> ◦ 스플래시 윈도우 초기화(일반적으로 출력은 CMainFrame의 OnCreate()함수
> ◦ 응용프로그램의 중복실행 방지
> ◦ 트라이얼(Trial) 버전의 기간 검사 루틴
> ◦ 운영체제의 버전을 확인하고 프로그램 실행 허용 여부 결정(예 : 응용 프로그램을 Windows 2000에서만 실행되도록 하는 경우)
> ◦ 프로그램 사용자 인증 부분(예 : 관리자만 프로그램을 실행하도록 하고 싶은 경우
> ◦ 프로그램 실행시 인자로 전달되는 매개 변수 처리

물론 여기서 언급되지 않았더라도 프로그램이 실행되기 전에 해야 하는 일들은 모두 여기서 하는것이 좋다.

언급된 것 처럼 InitInstance()에 모든 초기화를 넣지 않아도 상관은 없지만 될 수 있으면 이 함수에 코드를 추가하는 것이 일반적이고 권장사항이라는 것입니다.

따라서 역분석하는 입장에서 이 함수는 중요할 수 밖에 없습니다. 이 함수에 개발자가 정의한 주요 초기화는 대부분 존재 할 것이기 때문입니다.

- InitInstance()에 대해서 좀 더 자세히

C++ 소스코드에서 InitInstance()함수를 살펴보겠습니다.

```
class CMFC_ReversingApp : public CWinApp
{
public:
    CMFC_ReversingApp();

// 재정의입니다.
    public:
    virtual BOOL InitInstance();

// 구현입니다.

    DECLARE_MESSAGE_MAP()
};
```

[소스코드 8] CWinApp를 상속받은 CMFC_ReversingApp

CMFC_ReversingApp는 CWinApp를 상속받고 있습니다. 그리고 멤버 함수를 보면 InitInstance()를 재정의하고 있습니다.

실제 InitInstance()를 정의하는 부분(몸체)은 다른곳에 있겠지만 이정도만 알면 충분할 것 같습니다. 이 함수가 가상함수라는 사실을 알았다면 이 함수의 주소는 .rdata에 존재한다는 사실을 알 수 있습니다. 이에대해서는 2장의 가상함수 부분에서 자세히 다뤘습니다.

```
.rdata:004034C8    dd offset CCmdTarget::GetInterfaceHook(void cons
.rdata:004034CC    dd offset CCmdTarget::GetExtraConnectionPoints(C
.rdata:004034D0    dd offset CCmdTarget::GetConnectionHook(_GUID cc
.rdata:004034D4    dd offset CMFC_ReversingApp::InitInstance(void)
.rdata:004034D8    dd offset CWinApp::Run(void)
.rdata:004034DC    dd offset CWinThread::PreTranslateMessage(tagMSG
```

[그림 25] CMFC_ReversingApp의 가상함수 테이블

(소스코드 8)에서 볼 수 있듯이 클래스에는 한 개의 가상함수만이 존재하기 때문에 이 클래

스의 가상함수 테이블에도 하나의 엔트리 즉 InitInstance()만 존재할 것입니다. (그림 28)에서 확인할 수 있듯이 말입니다. 물론 그렇다고 해서 InitInstance()함수만 프로그램 실행 중에 실행되냐? 그건 아닙니다. 단지 우리가 이 함수를 오버라이딩 해서 사용했기 때문에 이 함수를 CMFC_ReversingApp 클래스에서 확인할 수 있는 것이지 다른 함수들은 CWinApp에 모두 멤버함수로 존재하고 있으며 실제로 실행 중에 호출됩니다.
하지만 오버라이딩 되지 않았다면 우리의 관심 밖일 것입니다. 그것은 자신이 마이크로소프트에서 미리 정의한 일을 수행할 뿐 그 어떤 다른 일도 하지 않을 것이기 때문입니다.

- 누가 호출할까?

사실 가장 궁금한 것은 이것 이었습니다. 과연 누가 이 함수를 호출할까요? 우리가 MFC를 코딩할 때 직접 InitInstance()함수를 호출하도록 코딩하지 않습니다. 단지 실행과정에서 누군가가 자동으로 이 함수를 호출하는 것입니다.

ELF[2] 형식을 생각해보면 __libc_start_main()이라는 함수가 main()보다 먼저 실행되어서 스택이나 환경변수를 초기화 한 후에 main()을 실행합니다. 그리고 __libc_start_main()은 libc.so이라는 C 라이브러리에 존재합니다.

아마 MFC도 이와 비슷한 구조일 것이라는 생각이 듭니다. 다음은 "Ivor Horton의 Beginning Visual C++.NET 2008"이라는 책의 한 부분을 인용한 것입니다.

> InitInstance(void)
>
> 이것은 기반 클래스 CWinApp에 정의된 가상 함수를 재정의한 것이며, 이전에 말했듯이 **MFC 라이브러리가 자동으로 제공해주는 WinMain()함수에 의해 호출된다.**

매우 중요한 말이 아닐 수 없습니다. 먼저 이 함수는 가상함수라는 것을 다시 말하고 있고, 가장 중요한 것은 MFC 라이브러리가 자동으로 제공해주는 WinMain()함수에 의해 호출된다는 것입니다. 다시말하면 InitInstance()는 WinMain()이라는 함수가 호출하는데, WinMain()이라는 함수는 우리가 직접 코딩한것이 아니라 MFC 라이브러리가 제공한다는 것입니다. 결론적으로 InitInstance()는 MFC 라이브러리가 호출하는 것입니다. 기타 다른 프로그램들도 대부분 이와 같은 방식을 통해서 호출됩니다.

사실 초기화 함수가 이 함수 외에도 여러 가지 함수가 존재하지만 가장 처음 분석할 때 초기화에는 이 함수에 집중하는 것이 좋습니다.

2) ELF : Executable and Linkable Format으로 리눅스, *nix에서 사용하는 실행파일, 라이브러리등에 대한 형식

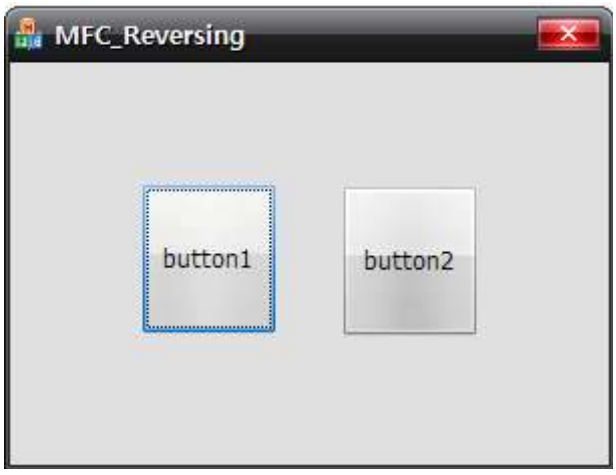
3.3. MFC 메시지 핸들러 찾기

MFC로 작성된 프로그램을 분석하다보면 특정 버튼을 눌렀을 때 이를 처리하는 프로시져를 찾는 일을 많이 하게 됩니다. 만약 리버스엔지니어링 경험이 있다면 단순히 WinAPI로 작성된 프로그램을 분석하는 것 처럼 특정 API에 브레이크를 걸고 실행해서 브레이크가 걸렸을 때 Step over 하는 형식으로 프로시져를 검사하는 방법을 사용 할 것입니다. 물론 이 방법도 괜찮습니다. 아니 때로는 매우 빠르게 프로시져를 찾게 해주니 좋은 방법이라고 까지 말할 수 있을 것 같습니다.

하지만 위 방법은 어느 정도 운이 작용합니다. 이 방법으로 매우 빠르게 찾을 수 도 있을테지만 또 어쩌다가는 못 찾을 수도 있습니다. MFC에서는 이방법이 매우 세련된 방법이라고 말 할 수 없습니다.

3.3.1. 코드 추적 방법(Code Tracing Method)

우리가 특정 버튼을 눌렀을 때 메시지박스를 띄워주는 프로그램을 아래와 같은 모양으로 만들었다고 가정해 보겠습니다.



[그림 26] 예제 프로그램

이제 하고자 하는 일은 button1을 눌렀을 때 발생하는 이벤트를 처리하는 함수를 찾는 일입니다. 그리고 먼저 코드 추적 방법을 사용해 보겠습니다.

- 코드 트레이싱?

먼저 이 방법을 알게된 계기는 2009 코드게이트 세미나에서 리버스엔지니어링을 쉽게하기라는 주제의 세미나에서 소개한 Hit Trace입니다. Hit Trace는 추적하고자 하는 범위를 설정하고 명령어가 실행될 때 마다 주소를 로깅해주는 올리디버거의 플러그인입니다. 그런데 이 플러그인 보다는 Olly Hits Snake가 원하는 코드를 쉽게 찾을 수 있도록 도와주는 것 같아서 주로 이걸 사용하고 있습니다.

이 플러그인은 Hit trace처럼 먼저 범위를 설정하고 원하는 코드를 찾고자 하는 이벤트가 발생하기 전까지 코드 추적을 합니다. 이벤트를 발생시키기 전(그림 30을 예로 버튼을 누르기전)에 한번 추적한 코드를 저장하고, 버튼을 누른 다음 다시 저장합니다. 이렇게 되면 아래와 같은 결과가 될 것입니다.

[그림 27] Olly Hits Snake 원리

두 로그의 차이는 버튼을 눌렀을 때 발생한 이벤트에 대한 코드가 있냐 없냐의 차이 정도입니다. 따라서 일치하는 코드는 제외하고 새로운 코드만 남겨두면 버튼을 눌렀을 때 코드만 볼 수 있는 것입니다.

이 방법은 프로그램이 비교적 단순하거나, 이벤트가 명확하게 구분 가능할 때는 확실히 좋은 효과를 볼 수 있습니다. 하지만 다소 복잡하고 이벤트가 연속적으로 발생한다면 이 방법으로 코드를 찾는다 해도 찾은 결과에서 다시 분석을 해야합니다.

즉 좀더 체계적인 방법이 필요합니다.

3.3.2. 메시지 맵을 사용한 핸들러 찾기

메시지 맵이란 특정 메시지가 발생하였을 때 이 메시지를 처리할 수 있는 핸들러를 지정하는 일종의 테이블입니다. MFC에서 버튼이나 메뉴를 클릭하였을 때 발생하는 모든 이벤트들은 이러한 메시지 맵에 기록되어져 있습니다.

- 메시지 맵

메시지 맵에 대해서 알아보겠습니다. 먼저 메시지 맵은 구조체 배열입니다. 어떤 메시지가 발생하였을 때 이 메시지를 어떻게 처리할 것인지에 대한 정보가 담긴 것입니다. 그 모양은 다음과 같습니다.

```
struct AFX_MSGMAP_ENTRY
{
    UINT nMessage;  // 윈도우 메시지
    UINT nCode;     // 제어 코드 또는 WM_NOTIFY 코드
    UINT nID;       // control ID (또는 윈도우 메시지는 0)
    UINT nLastID;   // control id의 범위를 정의하기 위한 엔트리로 사용
    UINT nSig;      // 액션, 메시지 타입 또는 메시지 번호의 포인터
    AFX_PMSG pfn;   // 호출 루틴 또는 특별한 값
};
```

[소스코드 10] AFX_MSGMAP_ENTRY 구조체

(소스코드 10)에서 가장 중요하게 확인할 부분은 가장 마지막 부분의 pfn입니다. 여기에 우리가 원하는 핸들러의 주소가 적힙니다. 그리고 nID와 nLastID에는 리소스 아이디가 적힙니다.

예로 (그림 26)의 메시지 맵을 확인해 보도록 하겠습니다.

```
BEGIN_MESSAGE_MAP(CMFC_ReversingDlg, CDialog)
        ON_WM_SYSCOMMAND()
        ON_WM_PAINT()
        ON_WM_QUERYDRAGICON()
        //}}AFX_MSG_MAP
        ON_BN_CLICKED(IDC_BUTTON_1, &CMFC_ReversingDlg::OnBnClickedButton1)
        ON_BN_CLICKED(IDC_BUTTON_2, &CMFC_ReversingDlg::OnBnClickedButton2)
END_MESSAGE_MAP()
```

[소스코드 11] 그림 30의 메시지 맵

위 소스코드는 (그림 26)의 메시지 맵입니다. 눈여겨 볼 부분은 ON_BN_CLICKED 메시지 부분입니다. 이 메시지는 버튼이 클릭 되었을 때 발생하는 메시지입니다. 그런데 우리가 만든 (그림 26)의 프로그램은 버튼이 2개이죠. 따라서 이 메시지도 두 개로 구분 되어져야 합니다.

이 때 구분짓는 방법으로 리소스 ID를 사용합니다. 즉 이런 것입니다.

```
if(MSG = ON_BN_CLICKED)
{
        switch(resource_ID)
        {
                case IDC_BUTTON_1:
                        do_something();
                        break
                case IDC_BUTTON_2:
                        do_something();
                        break
        }
}
```

[소스코드 12] 메시지에 대한 핸들러 찾아가기

(소스코드 12)는 의사코드 입니다. 메시지 맵은 사실 저런 내용을 간단히 표현한 것입니다. 먼저 어떤 메시지인지 검사를 하고, 어떤 리소스에서 발생한 것인지 확인합니다. 다음 해당 리소스의 핸들러를 호출합니다.

그렇다면 리소스 실제 어떤 값을 가지고 있을까요?

```
#define IDM_ABOUTBOX                    0x0010
#define IDD_ABOUTBOX                    100
#define IDS_ABOUTBOX                    101
#define IDD_MFC_REVERSING_DIALOG        102
#define IDR_MAINFRAME                   128
#define IDC_BUTTON_1                    1001
#define IDC_BUTTON_2                    1002
```

[소스코드 13] 리소스 아이디 정의

(소스코드 13)은 리소스파일에 저장된 #define부분입니다. 여기서 실제 리소스 아이디를 확인할 수 있습니다. IDC_BUTTON_1과 IDC_BUTTON_2는 각각 1001, 1002라는 것을 알았습니다.

(소스코드 11)을 다시 확인하면 리소스 아이디에 해당하는 핸들러 함수 주소를 넣어주는 것을 확인할 수 있습니다. 결론적으로 어떤 메시지가 발생했을 경우 리소스 아이디를 확인해서 해당하는 핸들러 함수를 가져와 호출하는 것입니다.

- 리소스 아이디 찾기

(소스코드 13)은 실제 소스를 가지고 있기 때문에 쉽게 확인할 수 있었습니다. 소스가 없는 컴파일된 프로그램의 경우 이 아이디를 어떻게 확인할 수 있을까요? MFC 리소스의 경우

쉽게 확인이 가능합니다. 리소스 해커나 PE Explorer를 사용하면 쉽습니다.
다음은 PE Explorer의 리소스 에디터를 사용하여 다이얼로그를 분석한 결과입니다.

```
102 DIALOGEX 0, 0, 170, 114, 0
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUPWINDOW | WS_VISIBLE
EXSTYLE WS_EX_APPWINDOW
CAPTION "MFC_Reversing"
LANGUAGE LANG_KOREAN, SUBLANG_KOREAN
FONT 9, "MS Shell Dlg"
{
  PUSHBUTTON    "button1", 1001, 37, 34, 39, 43
  PUSHBUTTON    "button2", 1002, 94, 35, 39, 43
}
```

[그림 29] PE Explorer로 확인한 다이얼로그 리소스

(그림 29)의 PUSHBUTTON 부분을 통하여 각 버튼의 리소스 아이디를 확인할 수 있습니다. 각각 1001과 1002라고 표기가 되어 있습니다. 이런식으로 리소스는 쉽게 획득할 수 있습니다.

- 실제 핸들러 찾기

우리는 위에서 메시지 핸들러가 어떻게 저장되는지 확인하였고, 같은 메시지일 때 리소스 아이디를 통해서 식별한다는 것을 알게 되었습니다. 또한 리소스 아이디는 리소스 해커나 PE Explorer를 사용하면 쉽게 분석이 가능하다는 것도 알고 있습니다.
이제 디스어셈블러를 사용하여 프로그램의 핸들러를 찾아보도록 하겠습니다. 먼저 우리가 해야할 일은 메시지맵 구조체 배열의 위치를 찾는 일입니다. 염두해 둘 것은 메시지맵 구조체 배열은 .rdata 섹션에 위치해 있습니다.

```
dd offset ?GetThisMessageMap@CDialog@@KGPBUAFX_MSGMAP@@XZ
                        ; DATA XREF: CObject::Dump(CDumpContext &)Io
                        ; CDialog::GetThisMessageMap(void)
dd offset unk_403544
align 8
db  12h                 ; DATA XREF: .rdata:004035FC↓o
db    1
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db    0
db  1Eh
db    0
db    0
db    0
dd offset ?OnSysCommand@CMFC_ReversingDlg@@IAEXIJ@Z ; CMFC_ReversingDlg::OnSysCommand
```

[그림 30] 메시지맵의 시작 부분

메시지 맵은 (그림 30)와 같은 형식으로 시작합니다. .rdata에서 messagemap 주소로 시작하는 부분이 있으면 다음에는 메시지맵 구조체 배열들이 등장합니다. 하지만 IDA로 확인하니 구조체가 눈에 잘 들어오지 않습니다. IDA에 구조체를 보기좋게 정리해주는 플러그인이 있다고 들었는데 아직 찾질 못하였습니다.

대신 올리디버거를 사용하겠습니다. IDA를 사용해서 찾은 주소를 올리디버거의 Address with Ascii Dump를 사용해서 확인하겠습니다.

```
00403560  00403544  D5@.   MFC_Reve.00403544
00403564  00000000  ....
00403568  00000112  ↕r..
0040356C  00000000  ....
00403570  00000000  ....
00403574  00000000  ....
00403578  0000001E  ...▼
0040357C  004012E0  ?@.   MFC_Reve.004012E0
00403580  0000000F  ☼...
00403584  00000000  ....
00403588  00000000  ....
0040358C  00000000  ....
00403590  00000013  !...
00403594  00401380  ▒@.   MFC_Reve.00401380
00403598  00000037  7...
0040359C  00000000  ....
004035A0  00000000  ....
004035A4  00000000  ....
004035A8  00000028  (...
004035AC  00401450  P@.   MFC_Reve.00401450
004035B0  00000111  ◄r..
004035B4  00000000  ....
004035B8  000003E9  ?..
004035BC  000003E9  ?..
004035C0  00000039  9...
004035C4  00401460  `@.   MFC_Reve.00401460
004035C8  00000111  ◄r..
004035CC  00000000  ....
004035D0  000003EA  ?..
004035D4  000003EA  ?..
004035D8  00000039  9...
004035DC  00401470  p@.   MFC_Reve.00401470
```

[그림 31] 메시지 맵 구조체 배열

(그림 31)에서 일정한 패턴이 있는 것을 확인할 수 있습니다. 구조체 배열이므로 당연한 결과입니다. 저 구조체 배열 중 우리가 원하는 구조체를 찾아야 합니다. 어떻게 찾을까요? 네, 바로 리소스 아이디입니다. 우리는 첫 번째 버튼의 리소스 아이디가 1001이라는 것을 알고 있습니다 그리고 이것은 16진수로 0x3e9입니다.

```
                    struct AFX_MSGMAP_ENTRY
                    {
004035B0|00000111-------> UINT nMessage;  // 윈도우 메시지
004035B4|00000000-------> UINT nCode;     // 제어 코드 또는 WM_NOTIFY 코드
004035B8|000003E9-------> UINT nID;       // control ID (또는 윈도우 메시지는 0)
004035BC|000003E9-------> UINT nLastID;   // control id의 범위를 정의하기 위한 엔트리로
004035C0|00000039-------> UINT nSig;      // 액션, 메시지 타입 또는 메시지 번호의 포인터
004035C4|00401460-------> AFX_PMSG pfn;   // 호출 루틴 또는 특별한 값
                    };
```

[그림 32] 구조체 배열과 매핑

(그림 31)에서 리소스 아이디 0x3e9가 들어간 구조체는 (그림 32)입니다. 그리고 각 원소를 실제 구조체에 매핑시켜보면 이해하기 편할 것입니다. 게다가 우리는 가장 중요한 함수 주소를 알게 되었습니다. button1의 핸들러 주소는 0x401460입니다. IDA를 사용하여 그 주소로 이동해 보겠습니다.

```
00401460 ; public: void __thiscall CMFC_ReversingDlg::OnBnClickedButton1(void)
00401460 ?OnBnClickedButton1@CMFC_ReversingDlg@@QAEXXZ proc near
00401460                                         ; DATA XREF: .rdata:004035C4↓o
00401460                push    0
00401462                push    0
00401464                push    offset aButton_1Clicke ; "Button_1 clicked"
00401469                call    ?AfxMessageBox@@YGHPB_WII@Z ; AfxMessageBox(wchar_t const *,uint,uint)
0040146E                retn
0040146E ?OnBnClickedButton1@CMFC_ReversingDlg@@QAEXXZ endp
```

[그림 33] button1의 핸들러 함수

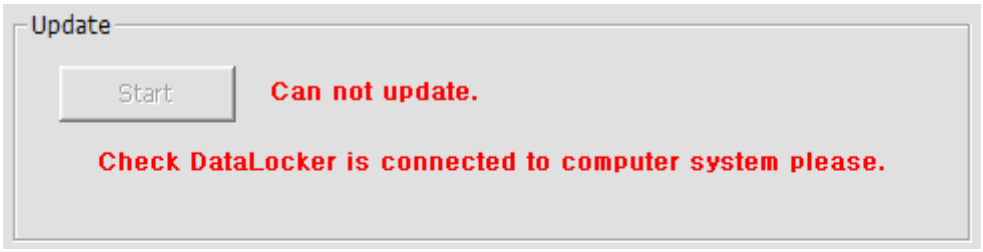
찾아낸 주소로 이동해 보니 AfxMessageBox() API를 사용해서 "Button_1 clicked" 라는 문자열을 메시지박스로 출력하는 것을 확인할 수 있습니다. 이로써 실제 버튼의 핸들러를 찾을 수 있었습니다.

- 쉽게 핸들러 찾기

핸들러의 주소가 어디에 저장이 되고 어떻게 호출되는지 알 수 있었습니다. 위의 방법 말고 쉽게 찾을 수 있는 방법을 생각해 보면 간단히 아래와 같이 정리될 수 있을 것 같습니다. (사실 동일한 내용입니다)

- 메시지맵은 .rdata에 존재하므로 .rdata만 탐색의 대상이 된다.
- 구조체에서 nID와 nLastID는 동일한 리소스 아이디로 설정된다.

위 두 가지만 가지고 쉽게 찾을 수 있습니다. 예를 위해서 Datalocker 사의 펌웨어 업그레이드 프로그램을 가지고 실험을 해 보겠습니다.



[그림 34] 펌웨어 업그레이드 프로그램

프로그램을 시작하면 대화상자에 Start라는 버튼이 있으며 실제 펌웨어를 시작하는 역할을 합니다. 먼저 리소스 아이디를 PE Explorer를 사용하여 알아냅니다.

```
PUSHBUTTON    "종료", 1, 230, 205, 50, 16
DEFPUSHBUTTON    "업데이트", 1000, 20, 146, 50, 16, WS_DISABLED
GROUPBOX    "ezSECU 정보", 1011, 7, 7, 273, 118
RTEXT    "칩셋", 1002, 10, 23, 51, 8
EDITTEXT    "", 1001, 67, 21, 122, 14, ES_AUTOHSCROLL | ES_READONLY
RTEXT    "디스크명", 1003, 10, 42, 51, 8
EDITTEXT    "", 1004, 67, 39, 190, 14, ES_AUTOHSCROLL | ES_READONLY
RTEXT    "드라이브명", 1005, 10, 60, 51, 8
```

[그림 35] 프로그램 리소스 확인

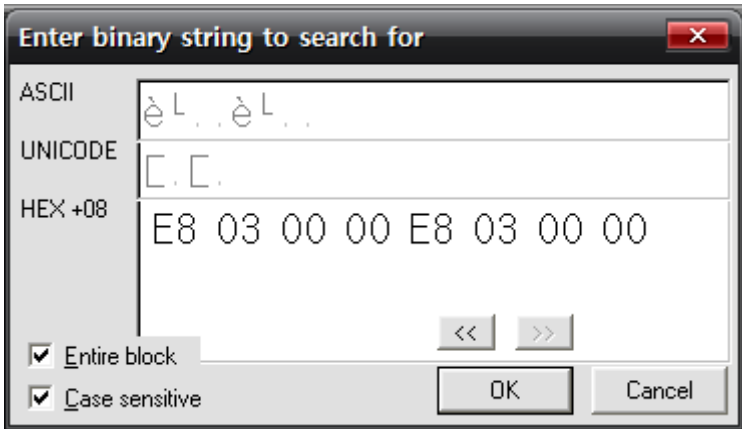
리소스 확인 결과 버튼의 ID는 1000인 것을 확인하였고 이는 16진수로 0x3e8입니다. 그렇다면 이 정보가 실제 구조체에 대입될 때 어떻게 대입될까요? nID와 nLastID는 바로 인접하는 4바이트씩 메모리 공간을 차지하기 때문에 Little Endian 방식으로 e8 03 00 00 e8 03 00 00 과 같은 식으로 저장 될 것입니다. 그리고 이 패턴은 다른곳에 있지않고 해당 프로그램의 .rdata 영역에 있을 것입니다. 따라서 동일한 다른 패턴이 검색될 확률은 매우 낮습니다.

올리디버거에서 Alt+M을 선택하여 프로그램의 .rdata로 이동합니다.

00400000	00001000	DataLock		PE header	Imag	R	RWE
00401000	00003000	DataLock	.text	code	Imag	R	RWE
00404000	00002000	DataLock	.rdata	imports	Imag	R	RWE
00406000	00013000	DataLock	.data	data	Imag	R	RWE
00419000	00006000	DataLock	.rsrc	resources	Imag	R	RWE

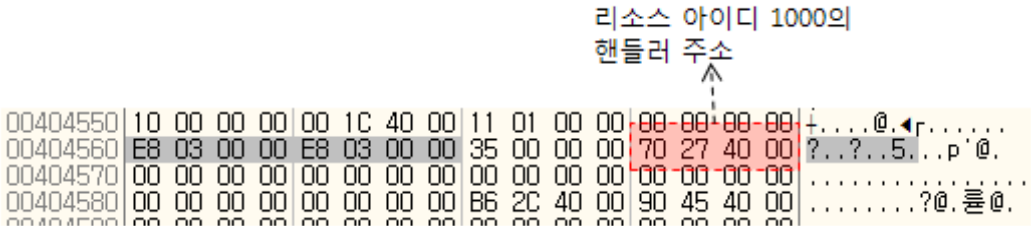
[그림 36] .rdata 영역

(그림 36)의 .rdata 부분을 더블클릭하면 이동이 됩니다. 이동이 되면 가장 첫 바이트위치에서 search for -> binary string을 선택합니다.



[그림 37] 리소스아이디 입력

(그림 37)과 같이 리소스아이디를 4바이트씩 2번 Little Endian 방식으로 써준 다음 OK를 누릅니다.



[그림 38] 검색 결과

검색결과는 매우 빠르게 나올 것입니다. 검색결과의 4바이트 떨어진 곳에 실제 버튼의 핸들러 주소가 적혀 있습니다. IDA를 사용해서 얻어낸 핸들러 주소로 이동해 보겠습니다.

```
.text:00402769 ; ---------------------------------------------------------
.text:0040276C                 align 10h
.text:00402770                 push    0
.text:00402772                 push    0
.text:00402774                 push    ecx
.text:00402775                 push    offset sub_402760
.text:0040277A                 push    0
.text:0040277C                 push    0
.text:0040277E                 call    ds:CreateThread
.text:00402784                 retn
.text:00402784 ; ---------------------------------------------------------
.text:00402785                 align 10h
```

[그림 39] 버튼의 핸들러 함수

(그림 39)는 어떤 함수 주소를 넣고 쓰레드를 생성하는 루틴입니다. 위에서 push한 함수 주소로 따라가면 실제 펌웨어를 업그레이드 하는 루틴들이 있습니다.

이처럼 쉽게 핸들러를 찾는 방법에 대해서 살펴보았습니다. 사실 위에서 살펴본것과 동일한 내용이지만 메시지맵 찾는 부분이 큰 프로그램에서는 다소 복잡할 수 있기 때문에 이와 같은 방법을 생각해 보았습니다.

4. 마치며

사실 좀 더 많은 내용을 다뤄보고 싶었는데, 많이 부족하다는 것을 느꼈습니다. 리버스엔지니어링은 하면 할수록 새로운 내용들이 많이 나오고, MFC 리버싱과 같이 MFC의 구조적인 동작 방식을 알면 좀더 쉽게 접근할 수 있는걸 알게 됩니다.

아직 많이 부족한 MFC 리버스엔지니어링 부분은 좀더 연구를 해야할 필요가 있습니다. 사실 실행중에 조건 브레이크를 사용해서 핸들러 찾는 방법에 대해서 쓰고 싶었는데 아직 완벽한 솔루션을 찾지 못하였습니다. 좀 더 연구를 해서 체계적이고 세련된 방법론을 찾아 내도록 하겠습니다.

참고문헌

[1] 엘다드 에일람, "리버싱 - 리버스엔지니어링 비밀을 파헤치다", WILEY, May 2009

[2] 크리스카스퍼스키, "HACKER DISASSEMBLING UNCOVERED", alist, 2003

[3] Externalist, "Reversing MFC Applications"

[4] window32, "MFC,DLL 모듈 리버싱 “, 마이크로소프트웨어 2008년 8월호, August 2008

[5] 최호성, "Windows Programming 기초편", FREELEC, MAY 2006

[6] Ivor Horton, "Beginning Visual C++.NET 2008, WROX, 2008

[7] "메시지맵과 메시지 라우팅의 원리", http://microdev.tistory.com/51

APPENDIX.A Disassembler & Debugger Tips

사실 이걸 따로 말할 필요가 있을지 모르겠지만 알아두면 좋은 사항입니다. 먼저 MFC를 분석할 때 디스어셈블러로는 IDA가 확실히 좋습니다. IDA는 최대한의 디버깅정보를 활용하여 체계화된 분석결과를 보여주기 때문에 MFC와 같이 복잡한 프로그램을 분석하는데 많은 도움이 됩니다.

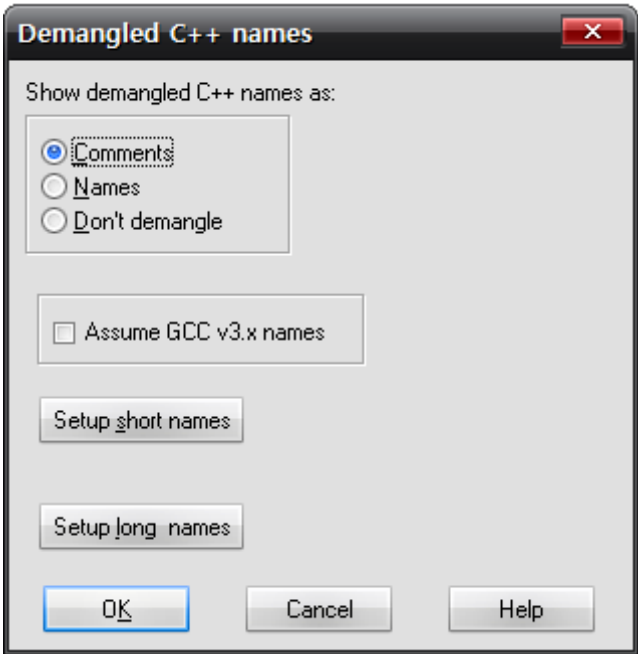
a.1. Demangled C++ name 위치 변경

IDA의 장점은 pdb 파일을 활용할 수 있다는 것입니다. MFC로 작성된 프로그램을 IDA로 로드하면 자동으로 pdb파일을 사용하여 MFC 함수에 라벨을 붙여줍니다. 이런건 역분석 과정에서 가독성을 매우 높여줍니다. 그런데 IDA는 기본적으로 Demangled 된 C++ 함수 이름을 실제 함수 라벨에 붙이지 않고 주석으로 달아놓습니다.

```
mov     edi, edi
push    ebp
mov     ebp, esp
call    ?AfxGetModuleState@@YGPAVAFX_MODULE_STATE@@XZ ; AfxGetModuleState(void)
mov     cl, [ebp+arg_0]
mov     [eax+14h], cl
```

[그림 40] 주석으로 표기된 Demangled 함수 이름

물론 주석으로 실제 함수의 이름을 확인할 수 있기 때문에 큰 문제될 건 없지만, 이게 맘에 안들어서 역분석에 스트레스를 받는다면 주석이 아니라 함수이름으로 대체가능합니다.



[그림 41] Demangled C++ names 메뉴

옵션에 Demangled C++ names 라는 항목이 있습니다. 클릭하면 (그림 41)와 같은 대화상자가 나타납니다. 항목 중 Show demangled C++ names as: 의 라디오 버튼을 Names로 바꾸면 함수 이름에 Demangled된 함수가 표기됩니다.

```
push    ebp
mov     ebp, esp
call    AfxGetModuleState(void)
mov     cl, [ebp+arg_0]
mov     [eax+14h], cl
```

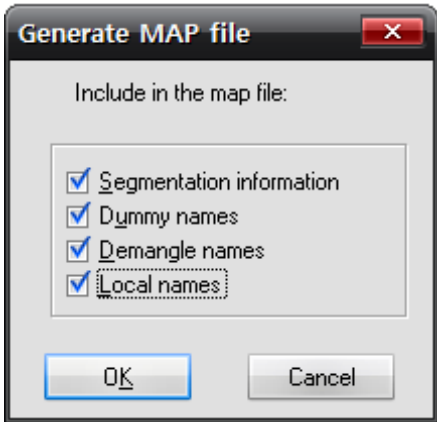
[그림 42] Demangled name 위치 변경

코드가 더 깔끔해 보여서 좋습니다. 물론 이전게 더편하다면 이전걸 쓰셔도 됩니다. 편한 걸 골라쓰는 거죠.

a.2. MapConv를 사용한 올리디버거에서 심볼정보 활용

IDA는 위에서 보았듯이 디버깅정보를 매우 잘 활용합니다. 그렇지만 올리디버거는 함수 이름표기와 같은 부분에 있어서 약한모습을 보입니다. MapConv라는 올리디버거 플러그인을 사용하면 IDA에서 생성한 Map파일을 올리디버거에서 사용할 수 있기 때문에 올리디버거에서도 풍부한 심볼과 함께 디버깅이 가능합니다.

먼저 IDA에서 분석을 마치면(Loading때 자동으로 수행되는 분석) File -> Produce File -> Create Map File 을 선택합니다. 생성될 파일 경로를 지정하고 나면 다음과 같은 대화상자가 나타날 것입니다.



[그림 43] MAP파일 옵션

어떤 정보를 포함시킬 것인지에 대한 설정입니다. 그냥 위와 같이 다하셔도 됩니다. 파일 생성이 완료되면 이제 올리디버거로 넘어옵니다.

```
PUSH 14
PUSH MFC_Reve.00403D20
CALL MFC_Reve.00401E8C
PUSH DWORD PTR DS:[40553C]
MOV ESI,DWORD PTR DS:[<&MSVCR90._decode_pointer>]
CALL ESI
POP ECX
MOV [LOCAL.7],EAX
CMP EAX,-1
JNZ SHORT MFC_Reve.00401AEC
PUSH [ARG.1]
CALL DWORD PTR DS:[<&MSVCR90._onexit>]
```

[그림 44] 디스어셈블링 결과

기본적으로 (그림 44)정도 수준의 디스어셈블링 결과를 보여줍니다. 이제 플러그인의 MApConv-> Replace Lavel과 Replace Comment를 선택합니다. IDA로 생성한 Map파일을 선택하면 아래와 같이 디버깅정보가 더 늘어난 것을 확인할 수 있습니다.

```
PUSH 14                                                    __onexit
PUSH MFC_Reve.00403D20
CALL <MFC_Reve.__SEH_prolog4>
PUSH DWORD PTR DS:[40553C]
MOV ESI,DWORD PTR DS:[<&MSVCR90._decode_pointer>]          MSVCR90._decode_pointer
CALL ESI                                                   <&MSVCR90._decode_pointer>
POP ECX                                                    kernel32.7C7E7077
MOV [LOCAL.7],EAX
CMP EAX,-1
JNZ SHORT <MFC_Reve.loc_401AEC>
PUSH [ARG.1]                                               func = <MFC_Reve._wWinMainC
CALL DWORD PTR DS:[<&MSVCR90._onexit>]                     _onexit
```

[그림 45] IDA에서 생성한 MAP파일 적용

(그림 44)와 (그림 45)만을 봐서는 큰 변화를 알 수 없지만, 사실 많은 부분이 보기좋게 바뀝니다. 디버깅을 할 때에 도움이 되니 알아두면 좋습니다.

a.3. OllyDBG에서 .lib파일 사용하기

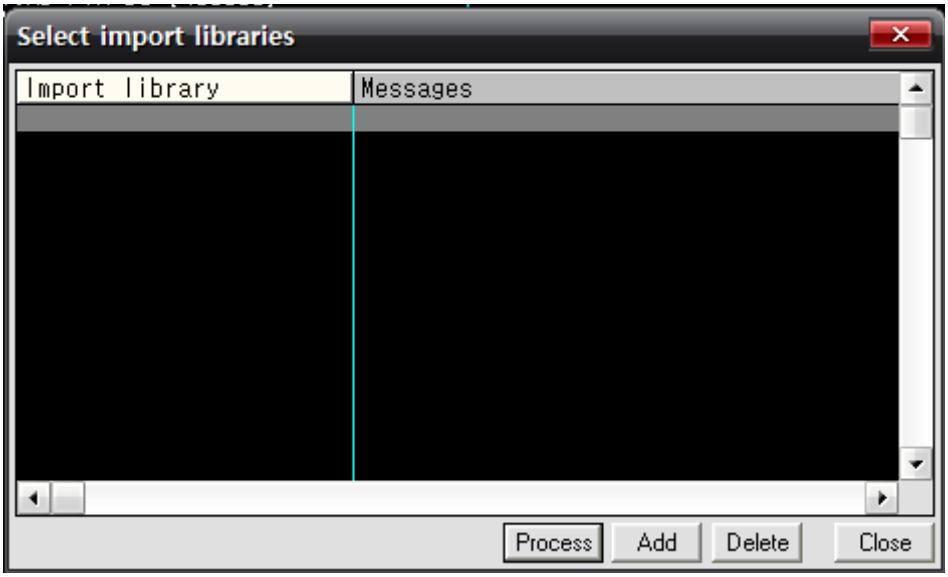
올리디버거에서 MFC나 기타 라이브러리를 사용하여 디버깅을 할 때 MFC의 심볼정보가 해석되어져 있지않아 분석하기 어려울 때가 있습니다. Names 윈도우를 보면 아래와 같이 나오는 경우입니다.

Address	Section	Type	Name
0040321C	.rdata	Import	mfc90u.#1048
004032AC	.rdata	Import	mfc90u.#1098
004032BC	.rdata	Import	mfc90u.#1108
004032B8	.rdata	Import	mfc90u.#1137
004032A8	.rdata	Import	mfc90u.#1183
004032F0	.rdata	Import	mfc90u.#1272
00403128	.rdata	Import	mfc90u.#1442

[그림 46] MFC 심볼정보가 없는 경우

이 때 MFC dll의 lib파일을 사용하면 심볼 정보를 추가시킬 수 있습니다. 이를 위해서 당연히 lib 파일이 존재해야 합니다. 이 파일은 Visual Studio를 설치하면 같이 설치가 됩니

다. 만약 없다면 따로 구해야합니다. (그림 46)의 경우 mfc90u.dll이 사용되었는데 lib파일 이름은 mfc90u.lib가 됩니다. 올리디버거의 Debug->Select Import Libraries를 선택하면 다음과 같은 화면을 볼 수 있습니다.



[그림 47] lib파일 추가

Add 버튼을 선택해서 .lib파일을 추가해 줍니다. 다음 Process 버튼을 클릭하면 심볼정보가 추가됩니다. 여기까지 끝나면 올리디버거를 재시작한 뒤 다시 MFC프로그램을 불러오고 똑같이 Names 창을 열어보면 아래와 같이 심볼정보가 포함된 것을 확인할 수 있습니다.

Address	Section	Type	Name
0040321C	.rdata	Import	mfc90u.#1048_CWinApp::AddToRecentFile
004032AC	.rdata	Import	mfc90u.#1098_AfxEnableControlContaine
004032BC	.rdata	Import	mfc90u.#1108_AfxFindResourceHandle
004032B8	.rdata	Import	mfc90u.#1137_AfxGetModuleState
004032A8	.rdata	Import	mfc90u.#1183_AfxMessageBox
004032F0	.rdata	Import	mfc90u.#1272_AfxWinMain
00403128	.rdata	Import	mfc90u.#1442_CWnd::CalcWindowRect
004031D8	.rdata	Import	mfc90u.#1492_CDialog::CheckAutoCenter
00403148	.rdata	Import	mfc90u.#1641_CWnd::ContinueModal
004031F0	.rdata	Import	mfc90u.#1675_CDialog::Create
00403114	.rdata	Import	mfc90u.#1727_CWnd::Create
004031AC	.rdata	Import	mfc90u.#1728_CWnd::CreateAccessiblePr
004031E0	.rdata	Import	mfc90u.#1751_CWnd::CreateControlConta
004031E4	.rdata	Import	mfc90u.#1754_CWnd::CreateControlSite
00403118	.rdata	Import	mfc90u.#1791_CWnd::CreateEx

[그림 48] 심볼정보가 추가된 dll

심볼정보가 있냐 없냐의 차이는 실제로 분석하다보면 엄청난 차이를 보입니다. 가능하다면 항상 심볼 정보를 추가시켜 놓고 분석하는 것이 좋습니다.

APPENDIX.B Tool Review

b.1. OllyDRX

OllyDRX는 OllyDBG를 커스터마이징한 프로그램입니다. 사실 OllyDRX를 말하지 않더라도 OllyDBG를 주로 사용하는 사람들은 여러 가지 플러그인들을 사용해서 사용하기 편하고 기능을 추가하는 방식으로 커스터마이징합니다. 그런데 OllyDRX는 그 커스터마이징의 정도가 매우 높은 프로그램입니다.

[그림 49] 상단 인터페이스

프로그램을 실행시키면 전체적으로 OllyDBG와 동일한 인터페이스를 가지고 있기 때문에 OllyDBG에 익숙하다면 사용에 지장은 없을 것 같습니다.

크게 다른점이라면 (그림 49)의 상단 인터페이스인데 여기에 주로 사용하는 프로그램들이 나열되어 있습니다.(사실 이것은 OllyDRX의 새로운 기능이 아닙니다. OllyDBG의 TBAR Manager 플러그인을 사용합니다.)

보통 저런 작은 프로그램들은 윈도우 시작메뉴의 빠른 실행 아이콘으로 등록해서 사용하는데 위와 같이 정렬 되어 있으니 상당히 편리해 보였습니다. 특히 마지막 pdf는 ASCII 테이블을 보여주기 위한 아이콘인데 상당히 아이디어가 좋아 보입니다.

위 프로그램 외에 TOOLS 폴더에 들어가면 언패커를 포함하여 19개 프로그램들이 포함되어져 있습니다. 또한 스크립트 폴더에 들어가면 600개가 넘는 스크립트들이 포함되어져 있습니다.

결론적으로 OllyDRX는 리버싱 툴들을 통합해 놓은 프로그램입니다. 어떤 프로그램을 분석하기 위해서 편리하게 하기 위해서 만들어 놓은것인데, 사용자가 만약 이 툴에 좀더 편리함을 느낀다면 이 디버거를 사용하는 것도 좋을 것 같습니다.