윈도우 프로그래머를 위한 PE 포맷 가이드
코드 패칭

목차

저작권

소개

코드 패칭이란 이미 만들어진 프로그램의 바이너리 코드를 수정하는 작업을 말한다. 이 기술은 해킹에서부터 최신 업데이트 기술에 이르기까지 광범위하게 사용되는 방법이다. 이번 시간에는 이런 코드 패칭의 기본 원리에 대해서 살펴보고, 정적 코드 패칭 작업을 실습한다. 또한 이 과정에서 코드를 추가하기 위해서 PE 포맷의 패딩된 공간을 사용하는 방법과 새로운 섹션을 추가하는 방법에 대해서 설명한다.

연재 가이드

운영체제: 윈도우 2000/XP
개발도구: Visual Studio 2005

기초지식: C/C++, Win32 API, Assembly
응용분야: 보안 프로그램

연재 순서

2007. 08. 실행파일 속으로
2007. 09. DLL 로딩하기
2007. 10. 실행 파일 생성기의 원리
2007. 11 코드 패칭
2007. 12 바이러스

필자소개

신영진 pop@jiniya.net, http://www.jiniya.net
웰비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

필자 메모

뿌리깊은 나무는 바람에 흔들리지 않으니 꽃 좋고 열매가 풍성할 것이요,
샘이 깊은 물은 가뭄에 마르지 않으니 내가 되어 바다에 이른다.

하루가 멀다 하고 새로운 기술이 쏟아지는 요즘이다. 수많은 베타 꼬리표를 달고 있는 그러한 신기술을 보고 있노라면 개발자의 길이 쉽지만은 않다는 생각이 절실히 든다. 이런 때일수록 용비어천가의 한 구절과 같이 뿌리를 튼튼히 하는 일에 신경을 써야 할 것이다. 그렇다면 개발자에게 뿌리가 되는 지식은 무엇일까? 필자는 컴퓨터 그 자체라고 생각한다. 플랫폼과 개발 기술을 뛰어넘어서 결국 개발자는 컴퓨터가 실행할 무엇인가를 만들어 내는 사람이기 때문이다.

이번 시간에 우리는 짙은 화장과 화려한 옷차림으로 자신을 가리고 있는 컴퓨터의 알몸을 살펴볼 것이다. 그 과정에서 배운 지식들은 분명 컴퓨터의 원시적인 동작 원리를 이해하는데 큰 도움을 줄 것이다. 여러분이 준비해야 할 것은 처음 누드 사진을 보는 사춘기 소년과 같은 호기심과 열정이 전부다.

Introduction

개발자라면 누구나 한번쯤은 해적판 소프트웨어나 쉐어웨어의 광고 창을 제거하는 방법에 대해서 호기심을 가져본 적이 있을 것이다. 내지는 그러한 것들을 배우기 위해서 개발이란 직업 세계로 뛰어든 사람도 있을 것이다. 또는 오래돼서 소스 코드가 존재하지 않는 프로그램을 수정하는 일을 맡아서 난감했던 경험이나 실행중인 프로그램을 중단하지 않고 업데이트 하는 일에 대해서 고민해본 적이 한번쯤은 있을 것이다. 이와 같은 모든 일에 두루 사용되는 기술이 패칭이다.

패칭이란 기록된 내용을 변경하는 것을 말한다. 결국 컴퓨터의 모든 데이터는 파일에서 읽혀져서 메모리에 로드되어 실행된다는 동일한 절차를 거친다. 따라서 파일이나 메모리의 기록된 내용을 변경한다면 실행되는 결과도 변경할 수 있다는 의미가 된다.

패칭은 크게 동적 패칭과 정적 패칭으로 나눌 수 있다. 동적 패칭은 실행 되어 있는 프로그램의 코드를 메모리 상에서 직접 수정하는 것이고, 정적 패칭은 실행 되기 전 파일의 내용을 고쳐서 다음 번 실행 때부터 변경된 내용이 적용되도록 하는 것이다. 여기서는 정적 패칭에 대한 내용만 다루도록 한다.

이번 시간에 진행되는 내용은 모두 컴파일된 바이너리 코드를 대상으로 한다. 필자가 사용한 Visual Studio 2005에 포함된 C++ 컴파일러가 아닌 다른 C++ 컴파일러로 컴파일된 바이너리 파일을 가지고 본다면 내용과 틀린 부분이 많을 것이다. 따라서 각 과정을 실습해보고 싶다면 Visual Studio 2005를 설치해서 사용하도록 하자. 마이크로소프트 홈페이지에서 Visual Studio 2005 Express 버전을 무료로 다운로드 받을 수 있다.

워밍업

패칭의 이론적인 부분을 살펴보기에 앞서 우리가 하려는 것이 무엇인지를 보여줄 수 있는 간단한 단계를 한번 수행해 보자. 한번도 실행 파일의 코드나 데이터를 수정해 보지 않았던 독자라면 이 과정을 컴퓨터 앞에서 따라 해보는 것도 꽤나 재미난 일이 될 것이다. 실습에 앞서서 아직 애용하고 있는 헥사 에디터나 디스어셈블러가 없다면 본문에 사용된 것들을 다운로드 받아서 설치해 두도록 한다. 본문에 사용된 헥사 에디터인 frhed는 http://www.rs.e-technik.tu-darmstadt.de/applets/frhed-v1.1.zip에서 무료로 다운로드 받을 수 있고, 디스어셈블러인 IDA는 http://www.datarescue.be/idafreeware/freeida43.exe에서 무료 버전인 4.3을 다운로드 받을 수 있다.

<리스트 1>에 우리가 패칭할 프로그램의 전체 코드가 나와있다. 패스워드를 입력 받고 그것을 체크하는 프로그램이다. 이 프로그램을 Visual C++에 입력하고 컴파일한다. 컴파일은 릴리즈 모드로 한다.

리스트 1 패칭을 수행할 간단한 샘플 프로그램

```
#include <windows.h>

void Print(PTSTR buf)
{
    int len = lstrlen(buf);
    WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE)
                , buf
                , len
                , NULL
```

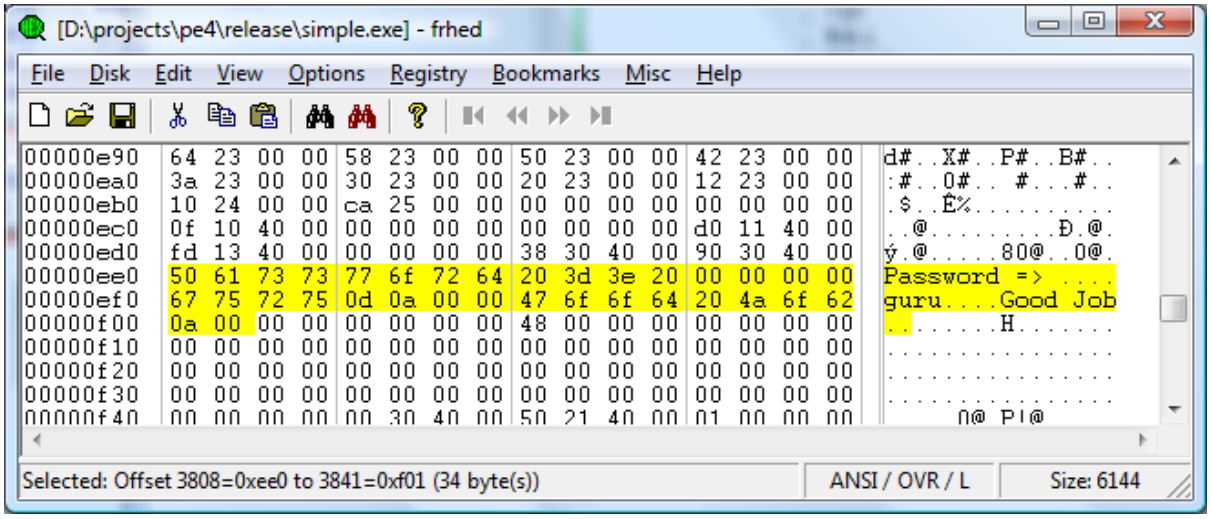
```
                , NULL);
}

int _tmain(int argc, _TCHAR* argv[])
{
    TCHAR buf[80] = {0,};
    Print(_T("Password => "));

    DWORD readed;
    ReadConsole(GetStdHandle(STD_INPUT_HANDLE)
                , buf
                , 80
                , &readed
                , NULL);
    if(lstrcmp(buf, _T("guru\r\n")) != 0)
        return 0;

    Print(_T("Good Job\n"));
    return 0;
}
```

컴파일이 끝났다면 simple.exe가 생성되었을 것이다. 간단하게 한번 실행해서 프로그램의 기능을 살펴보자. 'guru'를 입력하면 'Good Job'이 출력되고, 다른 값을 입력하면 조용히 종료된다. 기능을 한번쯤 테스트 해보았다면 이제 simple.exe를 헥사 에디터로 열어보자. 찾기 메뉴를 통해서 'Password =>'를 찾아보자. 아마 <화면 1>과 같은 부분이 발견될 것이다. 'Password =>'뒤로 'guru'와 'Good Job'이 있는 것이 보인다. 여기서 'guru'를 'chobo'로 변경해 보자. 주의해야 할 점은 'guru' 다음에 있는 0x0d, 0x0a도 같이 붙여주어야 한다는 점이다. 수정하고 실행해 보자. 아마 정상적으로 수정되었다면 이제부터는 'guru'가 아니라 'chobo'를 입력해야 'Good Job' 메시지가 출력될 것이다. 이와 같이 코드가 아닌 데이터를 변형 시키는 것을 데이터 패칭이라 한다.



화면 1 헥사 에디터로 simple.exe를 열어본 화면

이제는 조금 더 재미있는 일을 해보도록 하자. 헥사 에디터가 아닌 디스어셈블러로 simple.exe를 열어보자. 디스어셈블 작업이 완료되면 <화면 2>와 같은 부분이 표시된다. 여기서 가장 중요한 부분은 네모로 표시된 부분이다. 입력 받은 패스워드가 'guru'와 같은지 검사하는 부분이다. lstrcmpA를 호출한 다음을 보면 'jz short loc_401873'이란 부분이 보인다. 그 부분이 패스워드가 일치하면 이동하는 곳이다. 헥사 뷰로 이동해서 그 부분의 코드를 보면 0x74, 0x04인 것을 볼 수 있다. 0x74는 jz의 명령어 코드이고, 0x04는 다음 명령어가 실행될 지점(EIP)에서 4를 더한 곳으로 점프한다는 것을 나타낸다. 자 그렇다면 여기서 'guru'가 아닌 다른 패스워드를 입력해도 'Good Job'이 출력되도록 만들려면 코드를 어떻게 고치면 될까?

```
68 E0 20 40 00                  push    offset Buffer   ; "Password => "
E8 A2 FF FF FF                  call    sub_4017E0
83 C4 04                        add     esp, 4
6A 00                           push    0               ; lpReserved
8D 4D A4                        lea     ecx, [ebp+NumberOfCharsRead]
51                              push    ecx             ; lpNumberOfCharsRead
6A 50                           push    50h             ; nNumberOfCharsToRead
8D 55 A8                        lea     edx, [ebp+String1]
52                              push    edx             ; lpBuffer
6A F6                           push    0FFFFFFF6h      ; nStdHandle
FF 15 08 20 40 00               call    ds:GetStdHandle
50                              push    eax             ; hConsoleInput
FF 15 0C 20 40 00               call    ds:ReadConsoleA
68 F0 20 40 00                  push    offset String2  ; "guru₩r₩n"
8D 45 A8                        lea     eax, [ebp+String1]
50                              push    eax             ; lpString1
FF 15 10 20 40 00               call    ds:lstrcmpA
85 C0                           test    eax, eax
74 04                           jz      short loc_401873
33 C0                           xor     eax, eax
EB 0F                           jmp     short loc_401882

                loc_401873:                             ; CODE XREF: _main+5D↑j
68 F8 20 40 00                  push    offset aGoodJob ; "Good Job₩n"
E8 63 FF FF FF                  call    sub_4017E0
83 C4 04                        add     esp, 4
33 C0                           xor     eax, eax

                loc_401882:                             ; CODE XREF: _main+61↑j
8B 4D FC                        mov     ecx, [ebp+var_4]
33 CD                           xor     ecx, ebp
E8 74 F7 FF FF                  call    sub_401000
```

화면 2 디스어셈블러로 simple.exe의 코드를 분석한 화면

생각한 답대로 고쳐보고 프로그램을 실행시켜서 테스트 해보자. 예상했던 결과가 나온다면 그것이 정답이다. 필자가 생각한 두 가지 방법은 jz를 jmp로 바꾸는 방법과 jmp 뒤에 있는 loc_401882를 loc_491763으로 고치는 방법이다. jz를 jmp로 바꾸려면 0x74, 0x04를 0xeb(jmp 명령어 코드), 0x04로 바꾸면 된다. 두 번째 방법을 하려면 0xeb, 0x0f를 0xeb, 0x00으로 고치면 된다. 이런 식으로 어셈블리 코드를 직접 수정하는 것을 코드 패칭이라고 부른다.

다음 내용으로 넘어가기 전에 여기서 간단한 문제를 하나 풀어보자. 앞서 데이터 패칭을 수행할 때 우리는 'guru'를 'chobo'로 변경했다. 변경해 보았다면 알겠지만 'guru'를 'chobo'로 변경하기 위한 충분한 공간이 있기에 가능한 일이었다. 만약 'guru'를 'I love you so much'로 변경하려면 어떻

게 해야 할까? 나머지 부분을 읽기 전에 한번씩 고민해 보도록 하자.

NOP

어셈블리 언어에서 NOP는 아무 일도 하지 않고 CPU 클럭을 소모하는 명령어다. No operation의 줄인 말이라고 생각하면 쉽다. '아무 일도 하지 않고 CPU 클럭을 소모하는 명령어를 왜 만들어 두었을까?'라고 궁금해 하는 독자가 많을 것이다. 사실 실제로 이 NOP을 직접적으로 사용해야 하는 일은 별로 없다. 물론 간간히 임베디드 시스템에서는 delay를 구현하기 위해서 의도적으로 사용하는 경우가 있지만, 우리가 사용하고 있는 복잡한 컴퓨터, 그 위에서도 복잡한 윈도우, 그 위에서도 잘 추상화된 유저 모드 프로그램을 작성하는 입장에서는 사용할 일이 거의 없다고 할 수 있다. 하지만 이렇게 쓸모 없어 보이는 NOP 명령어도 패칭을 할 때에는 매우 요긴한 녀석으로 변한다.

NOP이 패칭에 유용한 이유는 두 가지 측면이다. 하나는 NOP이 아무 일도 하지 않는다는 것이고, 다른 하나는 NOP의 명령어 코드는 0x90으로 한 바이트만 차지한다는 점이다. 즉, NOP을 코드를 지우는 용도로 쓸 수 있다는 의미다.

앞서 패칭을 수행했던 <화면 2>의 코드를 다시 살펴보자. 우리는 앞서 'jz short loc_401873'을 두 가지 형태로 변경했다. 하지만 코드의 흐름만 조금 살펴보면 그렇게 복잡하게 하지 않더라도 단순히 jz만 수행하지 않는다면 비교가 발생하지 않는다는 것을 알 수 있다. 따라서 jz부터 jmp까지 6바이트를 모두 NOP(0x90)으로 채우면 우리가 앞서 패칭한 것과 동일한 결과를 얻을 수 있다.

물론 아무 일도 하지 않는 코드는 얼마든지 쉽게 만들 수 있다. 'move eax, eax', 'push eax; pop eax', 'add eax, 3; sub eax, 3'등과 같은 단순한 코드에서 아주 복잡한 코드까지 얼마든지 만들 수 있다. 하지만 이런 코드는 대부분 2바이트 이상이기 때문에 정확하게 남는 코드 부분을 채우기에는 적당하지 않다.

<화면 2>의 박스 친 부분 위쪽을 보면 ReadConsoleA를 호출하는 부분이 있다. 그 부분의 바이트 코드를 읽어보면, 0xff, 0x15, 0x0c, 0x20, 0x40, 0x00이다. 이 다섯 바이트의 코드를 0xe8, 0x12, 0x34, 0x56의 4바이트 코드로 변경하고 싶다고 하자. 앞 부분부터 네 바이트를 채우고 나면 한 바이트가 남는다. 그 부분을 그대로 놔두면 명령어 해석 순서가 틀어져서 완전히 새로운 코드가 된다. 따라서 그 부분을 무엇인가로 채울 필요가 있다. 무엇이 가장 적당할까? 바로 NOP이다.

표 1 NOP을 통한 실행 파일 변형

어셈블리어로 A, B, C라는 명령어를 순차적으로 실행하는 코드를 생각해보자. 앞서 설명했던 NOP의 특성을 생각해 본다면 A, B, C와 A, NOP, B, C는 동일한 과정을 수행함을 알 수 있다. 마찬가지로 A, B, NOP, C, A, NOP, B, NOP, C등도 똑 같은 일을 한다. 이와 같은 특징을 이용하면 하나의 코드에서 동일한 일을 수행하는 다양한 종류의 코드를 생성해 내는 것이 가능하다. 과거 바이러스는 생존률을 높이기 위해서 이러한 특성이 많이 사용했다. 물

론 요즘의 백신은 이렇게 변형된 것들도 변종으로 모두 검출해낸다.

완전 함수

특정 프로그램의 함수를 다른 프로그램에 그대로 붙여 넣으면 정상적으로 동작 할까? 여기서 그대로 붙여 넣는다는 말은 컴파일된 함수의 바이너리 코드를 그대로 복사한다는 말이다. 이해를 쉽게 하기 위해서 앞서 작성했던 simple.exe의 Print 함수를 살펴보도록 하자. Print 함수의 어셈블리 코드가 <화면 3>에 나와있다. 0x55, 0x8b, 0xec로 시작하는 이 함수의 바이트를 다른 프로그램의 일정 영역으로 복사하고 이것을 호출하면 어떻게 될까?

```
                ; int __cdecl sub_4017E0(void *lpBuffer)
                sub_4017E0      proc near               ; CODE XREF: _main+36↓p
                                                        ; _main+57↓p ...

                nNumberOfCharsToWrite= dword ptr -4
                lpBuffer        = dword ptr  8

* 55                             push    ebp
* 8B EC                         mov     ebp, esp
* 51                             push    ecx
* 8B 45 08                      mov     eax, [ebp+lpBuffer]
* 50                             push    eax             ; lpString
* FF 15 00 20 40 00              call    ds:lstrlenA
* 89 45 FC                      mov     [ebp+nNumberOfCharsToWrite], eax
* 6A 00                         push    0               ; lpReserved
* 6A 00                         push    0               ; lpNumberOfCharsWritten
* 8B 4D FC                      mov     ecx, [ebp+nNumberOfCharsToWrite]
* 51                             push    ecx             ; nNumberOfCharsToWrite
* 8B 55 08                      mov     edx, [ebp+lpBuffer]
* 52                             push    edx             ; lpBuffer
* 6A F5                         push    0FFFFFFF5h      ; nStdHandle
* FF 15 08 20 40 00              call    ds:GetStdHandle
* 50                             push    eax             ; hConsoleOutput
* FF 15 04 20 40 00              call    ds:WriteConsoleA
* 8B E5                         mov     esp, ebp
* 5D                             pop     ebp
* C3                             retn
                sub_4017E0      endp
```

화면 3 Print 함수의 어셈블리 코드

실행을 해보면 잘못된 연산 오류가 발생한다. 왜냐하면 Print 함수가 다른 함수에 의존적이기 때문이다. Print는 lstrlenA, GetStdHandle, WriteConsoleA라는 세 가지 함수를 사용한다. 이들 함수의 주소가 복사하는 프로그램에서도 똑 같은 위치라고 보장할 수 없기 때문이다.

보통 개발자들이 작성하는 함수는 전역 변수, 정적 지역 변수, DLL 함수, 라이브러리 함수 등에 의존적이다. 그렇기 때문에 그런 함수들은 불완전하다. 다른 프로그램으로 복사했을 때 바로 사용할 없다는 말이다. 반면에 'int plus(int a, int b) { return a+b; }'과 같은 함수는 그런 의존적인 요소가 전혀 없다. 따라서 plus는 그 자체로 완전한 함수라 할 수 있다. 다른 프로그램에 바이너리 코드를 붙여 넣고 호출해도 정상적으로 동작한다.

처음으로 다른 프로그램에 코드를 추가하는 작업을 하는 사람들은 보통 십중팔구 잘못된 연산 오

류를 만나기 마련이다. 바로 이 완전함수의 원리를 이해하지 못했기 때문이다. 함수를 다른 프로그램으로 이식시키기 위해서는 함수가 의존하고 있는 모든 요소를 제거해야 한다. 제거한다는 말은 각각의 주소를 적절하게 변경해 주어야 한다는 것이다. 하지만 이런 작업은 생각보다 간단하지 않다. 따라서 다른 프로그램에 이식할 함수라면 처음부터 의존 요소가 없도록 만드는 것이 좋다.

Dll 함수 호출

지난 시간에 우리는 DLL 로더를 제작하면서도 실제로 DLL의 함수가 어떻게 호출되는지에 대해서는 살펴보지 않았다. 단지 IAT에 DLL 함수 번지를 채워주면 DLL 함수 호출이 정상적으로 이루어진다고만 설명했다. 그렇다면 도대체 어떤 과정을 거치셔 단지 IAT에 주소만 채운 것으로 DLL 함수 호출이 이루어지는 살펴보자.

<화면 3>에서 kernel32.dll의 함수인 lstrlenA를 호출하는 부분을 살펴보자. 옆에 있는 바이트 코드를 살펴보면 0xff, 0x15, 0x00, 0x20, 0x40, 0x00이라 되어 있다. 0xff는 call 명령어 코드다. 다음에 이어서 나오는 0x15는 modr/m 바이트라고 불리는 것으로 명령어의 형태와 오퍼랜드의 종류를 규정하는 역할을 한다. 0x15는 이 call 명령어 뒤에 나오는 오퍼랜드가 메모리 주소이고, 그곳에 기록된 값으로 간접 호출한다는 것을 나타낸다. 나머지 네 바이트는 주소를 나타낸다. 읽어보면 0x00402000이 된다. 만약 0x00402000에 0x1234가 기록되어 있다면 실제 위의 call 명령은 0x1234로 이동하는 것이 된다.

위의 설명은 잠시 기억 속에서 지우고, 파일의 임포트 테이블을 한번 살펴보도록 하자. simple.exe의 PE헤더에서 임포트 테이블의 위치를 찾아보면 0x21c4라고 되어 있다. 이는 RVA이기 때문에 실제 파일 오프셋으로 변환해 보면 0xfc4(0x21c4 – 0x2000 + 0xe00)가 된다. 그 부분에 대한 헥사 덤프가 <화면 4>에 나와있다. 반전된 부분이 IMAGE_IMPORT_DESCRIPTOR 구조체를 나타낸다.

```
0fb0  d8 ff ff ff 00 00 00 00 fe ff ff ff 29 16 40 00  Øÿÿÿ....þÿÿÿ).@.
0fc0  3d 16 40 00 00 22 00 00 00 00 00 00 00 00 00 00  =.@.."..........
0fd0  04 23 00 00 00 20 00 00 4c 22 00 00 00 00 00 00  .#... ..L"......
0fe0  00 00 00 00 24 24 00 00 4c 20 00 00 00 00 00 00  ....$$..L ......
```

화면 4 임포트 테이블의 시작 부분

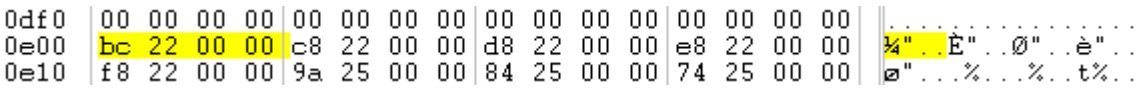
Name 필드의 값을 읽어 보면 0x2304다. 이 또한 RVA이기 때문에 파일 오프셋으로 변환해야 한다. 변환하면 0x1104이 되고, <화면 5>에 그 부분에 대한 헥사 덤프가 나와 있다. 반전된 부분을 읽어보면 지금 우리가 보고 있는 부분이 kernel32.dll에 대한 부분이라는 것을 알 수 있다.

```
10f0  6e 73 6f 6c 65 41 00 00 c0 03 6c 73 74 72 63 6d  nsoleA..À.lstrcm
1100  70 41 00 00 4b 45 52 4e 45 4c 33 32 2e 64 6c 6c  pA..KERNEL32.dll
1110  00 00 1d 01 5f 61 6d 73 67 5f 65 78 69 74 00 00  ...._amsg_exit..
```

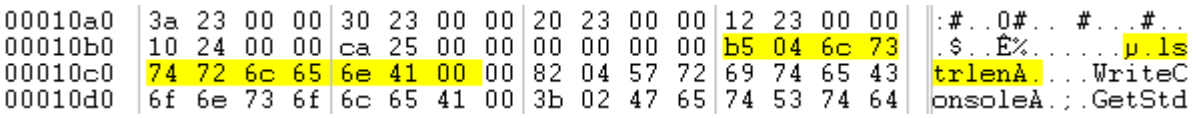
화면 5 임포트 테이블의 Name 필드가 가리키는 부분

임포트 테이블에서 Name필드 다음에 있는 FirstThunk 필드를 읽어 보면 0x2000이다. 이를 파일 오프셋으로 변환하면 0xe00 되고, 파일에서 찾아보면 <화면 6>과 같은 부분이 나타난다. 이 곳에

있는 내용은 IMAGE_THUNK_DATA 구조체다. 이름으로 기록된 경우에는 RVA를 나타내기 때문이 0x22bc 부분을 파일에서 찾아보면 <화면 7>과 같다. 즉, lstrlenA를 가리키는 곳이다.



화면 6 임포트 테이블의 FirstThunk 필드가 가리키는 부분



화면 7 FirstThunk가 가리키는 부분

Dll 로더를 제작할 때 살펴보았듯이 <화면 6>에 나타난 부분은 파일로 존재할 때에는 함수 이름에 대한 RVA를 저장하고 있지만 로딩되고 나면 실제 lstrlenA의 주소로 채워진다. 앞서 우리가 call의 주소로 얻었던 0x00402000을 다시 떠올려 보자. 이 주소를 파일 위치로 변환해 보면 0xe00(0x00402000 – 0x400000 – 0x2000 + 0xe00)이 되는 것을 알 수 있다. 즉, call 명령어의 주소는 IAT를 가리키고 있는 것이다. 따라서 이 곳의 주소 값만 바꾸면 해당 함수를 호출하는 모든 명령어가 이동되는 위치를 변경할 수 있다.

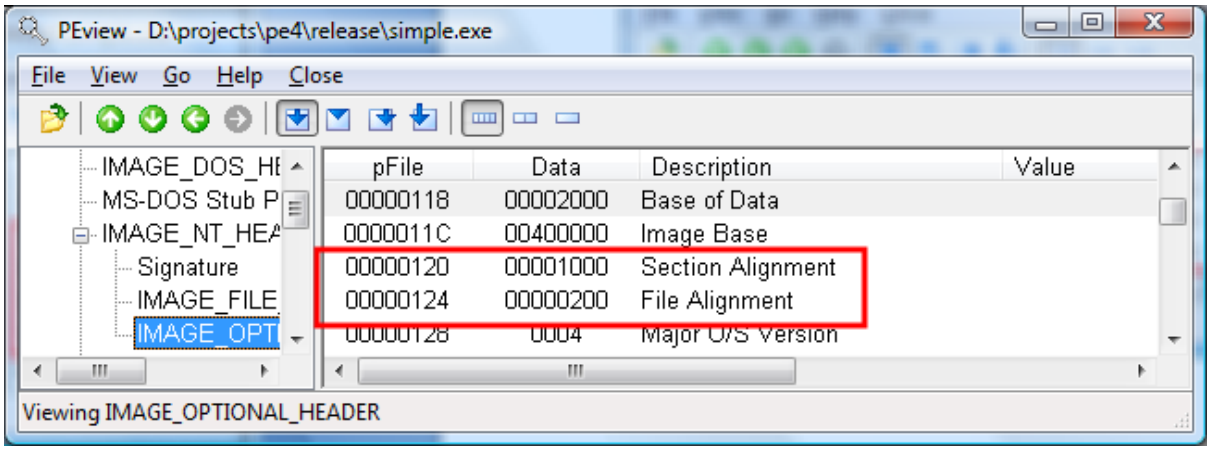
코드 추가

앞서 우리가 워밍업 부분에서 실습했던 것들은 모두 코드를 수정하는 작업에 관한 것이었다. 이후 NOP 명령어를 통해서 임의의 코드를 삭제할 수 있다는 것도 발견했다. 이런 작업을 흐름 제어라 부른다. 프로그램의 실행 흐름을 변형시키는 게 주된 목적이기 때문이다. 흐름 제어를 통해서 패스워드 검사를 무력화 시키거나, 시리얼 코드를 묻는 화면을 없애거나, 보안 프로그램을 분리해 내는 작업들을 할 수 있다. 하지만 정작 기존의 프로그램에 새로운 기능을 추가할 수는 없다.

새로운 기능을 추가하기 위해서 가장 중요한 것은 코드를 기록할 공간을 확보하는 것이다. 공간만 확보된다면 어쨌든 어셈블리 코드를 추가할 수 있기 때문이다. 여기서 우리는 두 가지 방법을 살펴볼 것이다. 하나는 PE 포맷에 숨겨진 자투리 공간을 활용하는 것이고, 다른 하나는 PE 포맷을 확장해서 전혀 새로운 공간을 추가하는 방법이다. 새로운 코드는 간단한 것을 실습할 것이기 때문에 직접 명령어 코드를 만들어서 기록하는 방법을 사용한다.

정렬을 위해 패딩된 공간

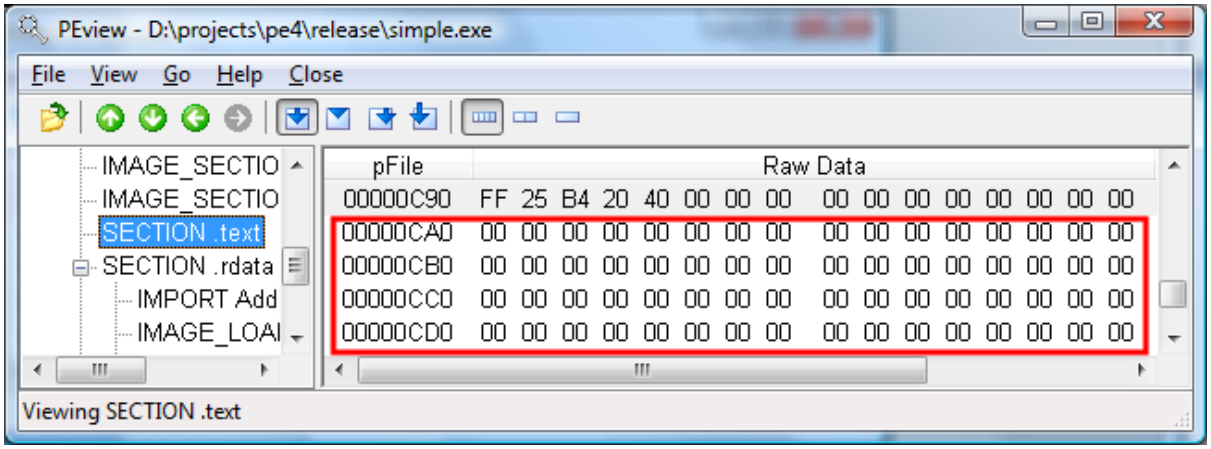
PE 포맷에는 두 가지 종류의 정렬 필드가 있다. 하나는 FileAlignment 필드로 파일로 존재할 때 디스크 상에서 섹션의 경계를 맞추기 위해서 사용된다. 다른 하나는 SectionAlignment 필드로 PE 파일이 메모리로 올라가 맵핑될 때 섹션 경계를 맞추기 위해서 사용된다.



화면 8 PE 포맷의 정렬 필드

<화면 8>에 simple.exe의 정렬 필드 값들이 나와 있다. FileAlignment는 0x200으로 512 바이트 단위로 정렬된다는 것을 나타낸다. 각 섹션의 SizeOfRawData는 반드시 이 필드의 배수가 되어야 한다. SectionAlignment 필드는 0x1000으로 4096 바이트 단위로 정렬된다는 것을 알 수 있다. 각 섹션의 VirtualAddress 필드는 SectionAlignment의 배수가 되어야 한다.

<화면 8>과 같은 단위의 정렬 필드 값이 사용될 때 코드 섹션의 실제 크기가 513바이트라면 코드 섹션이 실제 파일에서 차지하는 크기는 얼마가 될까? 당연히 1024 바이트가 된다. 나머지 511 바이트는 정렬을 위한 패딩공간으로 <화면 9>와 같기 0으로 채워진다. 대부분의 실행 파일의 경우 정확하게 정렬 필드 값의 배수배가 되는 섹션 크기를 가지는 일은 없다. 따라서 거의 모두가 정렬을 맞추기 위한 추가적인 공간을 가지고 있다. 이 공간에 들어갈 수 있을 정도로 작은 코드라면 정렬을 위해서 남겨진 공간에 추가적인 코드를 기록하고 그 위치를 사용할 수 있다.



화면 9 정렬을 맞추기 위해서 패딩된 공간

그러면 이제 실제로 이 공간을 사용해서 코드를 고쳐 보도록 하자. 앞서 만든 simple.exe의 경우 암호가 틀릴 경우에는 아무런 메시지도 출력하지 않고 종료한다. 참으로 불친절한 프로그램이 아닐 수 없다. 이를 고쳐서 잘못된 패스워드를 입력하면 'Wrong password'라는 말을 출력하도록 프로그램을 변경해보자.

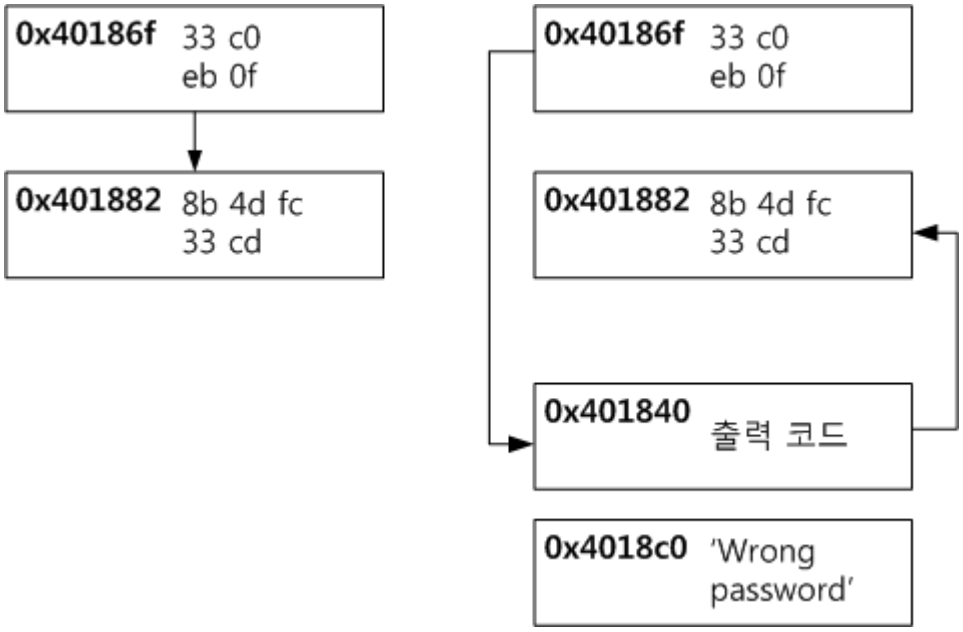
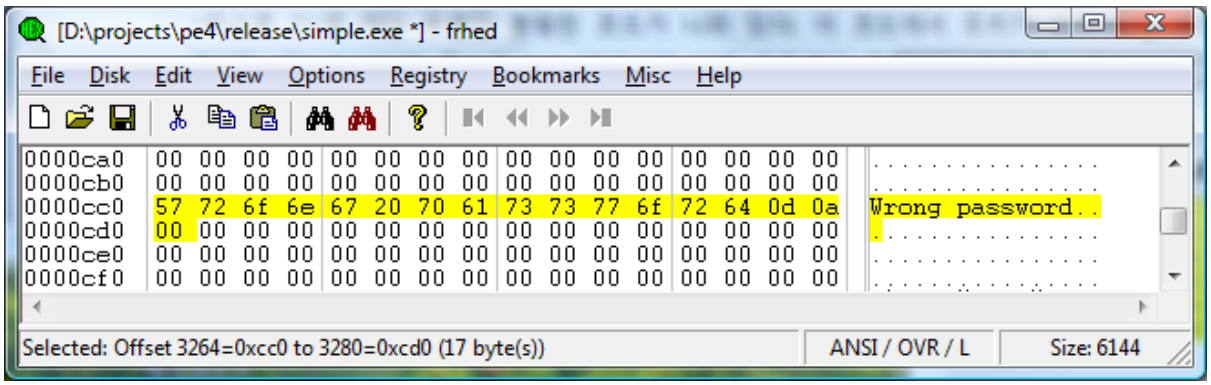


그림 1 코드 추가 흐름도

변경을 하기에 앞서서 <그림 1>에 나와 있는 흐름도를 살펴보는 것이 도움이 된다. <화면 2>의 코드와 같이 보면 이해하기가 훨씬 수월하다. 각 박스의 왼쪽에 진하게 표시된 부분은 주소다. 왼쪽은 기존의 코드 흐름을, 오른쪽은 우리가 수정한 상태의 코드 흐름을 나타낸다. 0x401869는 비밀번호가 틀렸을 경우에 실행되는 코드이고, 0x401882는 프로그램이 끝나는 지점에 실행되는 코드다. 우리는 뒤쪽에 코드와 데이터를 추가하고 비밀번호가 틀린 경우에 추가된 쪽으로 점프를 하도록 만들 것이다. 추가된 코드가 모두 실행되고 나면 원래 순서대로 프로그램 종료하는 부분으로 다시 점프를 해준다. 작업을 하기에 앞서 마지막으로 각 주소에 대해서 한번 더 살펴보자. simple.exe는 메모리 상의 0x400000 번지에 로드되고, text 섹션의 파일 오프셋은 0x400이고, 메모리 상에는 0x1000에 맵핑된다. 따라서 0x4018c0의 파일 오프셋은 0xcc0(0x4018c0 – 0x400000 – 0x1000 + 0x400)이고, 0x401840의 파일 오프셋은 0xca0이다. 실제 코드를 추가해 보도록 하자.

우선 제일먼저 할 일은 'Wrong password'를 기록하는 일이다. 헥사 에디터로 열어서 0xcc0 오프셋에 'Wrong password'를 기록하자. 이는 text 섹션의 패딩을 위한 공간이다. 추가를 하면 <화면 10>과 같이 된다. 0x0d, 0x0a는 "₩r₩n"을 의미한다.



화면 10 0xCE0에 Wrong password를 추가한 부분

이제 'Wrong password'를 출력하는 코드를 작성할 차례다. 'Wrong password'를 출력하는 부분은 simple.exe에 있는 Print 함수를 사용하면 된다. <화면 2>에서 'Good Job'을 출력하는 코드를 보도록 하자. push로 'Good Job'을 스택에 넣고, sub_4017E0를 호출한다. 이것이 전부다. 그곳에 있는 바이트 코드를 그대로 사용하도록 한다.

리스트 2 'Good Job'을 출력하는 코드 부분

```
68 f8 20 40 00   push offset aGoodJob
e8 63 ff ff ff   call sub_4017e0
83 c4 04          add esp, 4
```

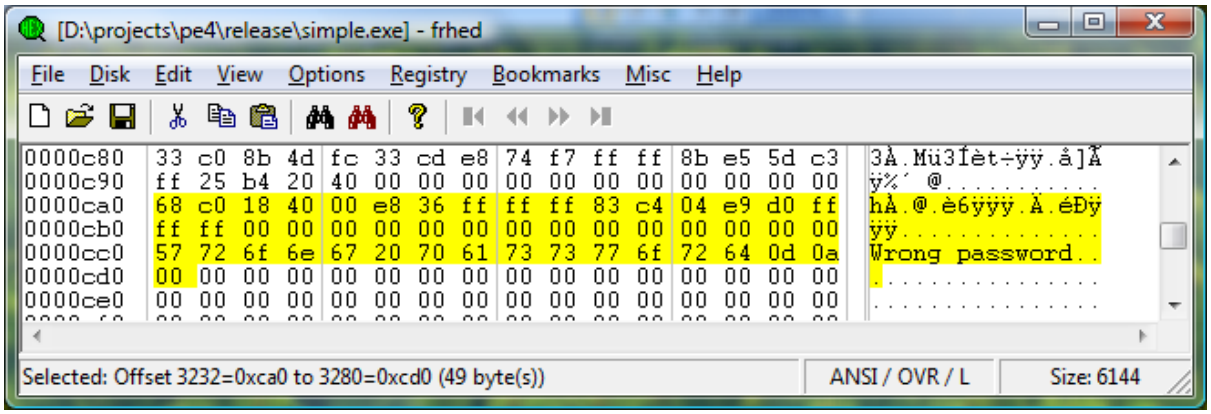
<리스트 2>에 해당 부분만 발췌한 코드가 나와 있다. 이 코드에서 우리가 고쳐야 할 부분은 두 곳이다. 'Good Job'의 주소와 sub_4017e0의 상대 오프셋이 그것이다. 각각을 수정하는 방법을 살펴보자.

0x68, 0xf8, 0x20, 0x40, 0x00에서 0x68은 push에 대한 명령어 코드다. 그 뒤로 나오는 4바이트가 push될 값을 의미한다. 리틀 엔디안이기 때문에 역으로 읽어보면 0x004020f8이 된다. 우리는 이 주소를 'Wrong password'를 기록했던 0x4018c0으로 변경해 주면 된다.

다음으로 살펴볼 부분은 call sub_4017e0에 해당하는 코드인 0xe8, 0x63, 0xff, 0xff, 0xff이다. 0xe8은 call에 대한 명령어 코드이고, 뒤쪽에 있는 0x63, 0xff, 0xff, 0xff는 호출할 곳에 대한 상대 위치다. 현재 명령어 포인터(EIP)에 저장된 값에 0xffffff63이 더해진 곳을 호출한다는 의미다. 우리가 호출해야 하는 Print 함수의 절대 번지는 IDA가 분석해준 함수명에 있는 것처럼 0x4017e0다. 우리가 호출할 때의 EIP는 0x4018aa다. 이는 우리가 추가한 코드가 시작하는 0x4018a0에 push와 call 명령어의 길이인 10을 더해준 값이다. 이 위치에서 0x4017e0를 호출하기 위한 오프셋을 계산기로 계산해 보면 0xffffff36이 된다. 끝으로 우리가 0x4018a0에 만든 코드 마지막 부분에 0x401882로 점프하는 코드를 집어넣는다. 두 곳 사이의 오프셋은 0xffffffd0(0x401882 – 0x4018b2)다. 이렇게 생성한 코드가 <리스트 3>에 나와 있다. 앞서 설명한 데이터와 함께 코드를 추가하면 <화면 11>과 같은 형태가 된다.

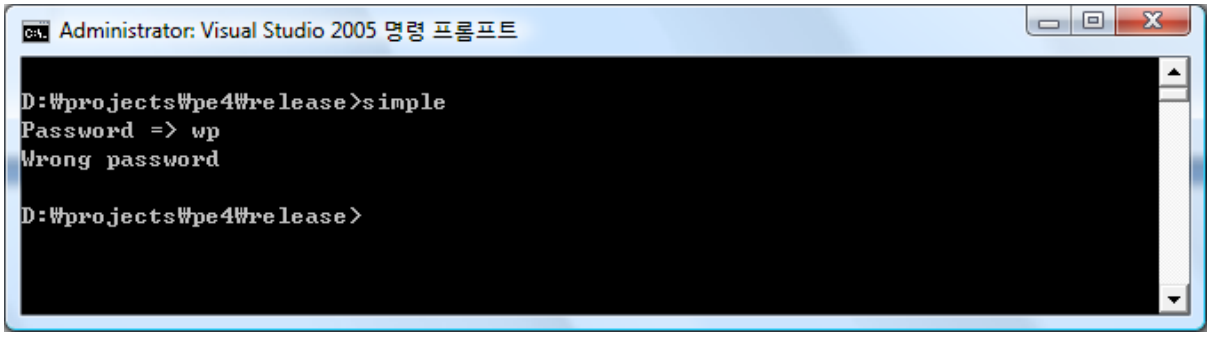
리스트 3 'Wrong password'를 출력하는 코드

```
68 c0 18 40 00    push 'Wrong password'
e8 36 ff ff ff    call sub_4017e0
83 c4 04           add esp, 4
e9 d0 ff ff ff    jmp loc_401882
```



화면 11 코드와 데이터를 모두 추가한 화면

이제 <그림 1>에 있는 한 가지 선만 연결해주면 된다. 바로 0x40186f에서 0x4018a0으로 점프하는 코드다. 기존의 0x40186f에서 0x401882로 점프하는 부분의 코드는 0xeb, 0x0f다. 이 명령어가 실행될 때의 EIP는 0x401873(0x40186f + 0x04)다. 따라서 우리가 점프해야 하는 곳인 0x4018a0까지의 오프셋은 0x2d(0x4018a0 – 0x401873)가 된다. 0x0f를 0x2d로 고쳐주면 된다. 이렇게 모든 부분을 고친 다음 프로그램을 실행해서 엉뚱한 패스워드를 집어넣은 것을 캡쳐한 것이 <화면 12>에 나와 있다.



화면 12 프로그램 실행 화면

섹션 추가하기

사실 패딩 영역을 사용해서 추가할 수 있는 코드에는 한계가 있다. 때로는 패딩 영역이 아예 존재하지 않을 수도 있다. 그런 경우에는 별도의 섹션을 추가해서 그곳에 코드를 기록하는 방법을 사용하면 된다. 이 경우에는 이론적으로는 거의 모든 코드를 추가할 수 있다는 장점이 있다. 반면에 섹션을 추가하는 복잡한 작업을 해야 한다는 불편함이 있다.

<리스트 4>에 파일에 섹션을 추가하는 AddSection 함수가 나와 있다. 일단 한번 코드를 쭉 살펴보도록 하자. 함수 중간에 나오는 WriteSection과 FillSection은 파일 끝에 새로운 섹션을 기록하는 역할을 한다.

리스트 4 파일에 섹션을 추가하는 함수

```
BOOL AddSection(LPCTSTR inPath      // 입력 파일 경로
```

```
                , LPCTSTR secPath   // 섹션 파일 경로
                , LPCTSTR secName   // 섹션 이름
                , DWORD secSize     // 섹션 크기
                , DWORD secAttr)    // 섹션 속성
{
    HANDLE in = INVALID_HANDLE_VALUE;
    HANDLE out = INVALID_HANDLE_VALUE;
    TCHAR outPath[MAX_PATH];
    PVOID buf = NULL;
    DWORD ntOffset, secOffset;
    BOOL ret = FALSE;

    DWORD hSize, rhSize; // 기존 헤더 크기, 새 헤더 크기
    int secNo; // 추가되는 섹션 번호
    DWORD fAlign, sAlign; // 파일 정렬, 섹션 정렬
    DWORD sizeOfImage; // 이미지 크기

    // 1. 파일을 열고, 헤더 내용을 검증한다.
    in = CreateFile(inPath, GENERIC_READ, 0, NULL
                    , OPEN_EXISTING, 0, NULL);

    if(in == INVALID_HANDLE_VALUE)
        return FALSE;

    IMAGE_DOS_HEADER dos;
    IMAGE_NT_HEADERS nt;
    IMAGE_SECTION_HEADER sec[MAXIMUM_SECTION_NUMBER];
    DWORD readed;

    if(!ReadFile(in, &dos, sizeof(dos), &readed, NULL))
        goto $cleanup;

    if(dos.e_magic != IMAGE_DOS_SIGNATURE)
        goto $cleanup;

    ntOffset = dos.e_lfanew;
    SetFilePointer(in, ntOffset, NULL, FILE_BEGIN);
    ReadFile(in, &nt, sizeof(nt), &readed, NULL);

    if(nt.Signature != IMAGE_NT_SIGNATURE)
        goto $cleanup;

    secNo = nt.FileHeader.NumberOfSections;
```

```
    if(secNo >= MAXIMUM_SECTION_NUMBER)
        goto $cleanup;

    secOffset = dos.e_lfanew
            + sizeof(nt.Signature)
            + sizeof(nt.FileHeader)
            + nt.FileHeader.SizeOfOptionalHeader;

    rhSize = secOffset
            + (nt.FileHeader.NumberOfSections + 1)
            * sizeof(IMAGE_SECTION_HEADER);

    SetFilePointer(in, secOffset, NULL, FILE_BEGIN);
    ReadFile(in
            , sec
            , secNo * sizeof(IMAGE_SECTION_HEADER)
            , &readed, NULL);

    fAlign = nt.OptionalHeader.FileAlignment - 1;
    sAlign = nt.OptionalHeader.SectionAlignment - 1;

    // 2. 새로운 헤더 크기를 계산한다.

    rhSize = MakeAlign(rhSize, fAlign);
    hSize = nt.OptionalHeader.SizeOfHeaders;
    if(hSize > sec[0].PointerToRawData)
        hSize = sec[0].PointerToRawData;

    if(rhSize > hSize)
    {
        if(rhSize > sec[0].VirtualAddress)
            goto $cleanup;
    }

    // 3. 출력 파일을 열고, 기존 섹션 내용을 기록한다.

    StringCbCopy(outPath, sizeof(outPath), inPath);
    StringCbCat(outPath, sizeof(outPath), _T(".nx"));

    out = CreateFile(outPath, GENERIC_WRITE, 0
                    , NULL, CREATE_ALWAYS
                    , FILE_ATTRIBUTE_NORMAL, NULL);

    if(out == INVALID_HANDLE_VALUE)
```

```
    goto $cleanup;

DWORD bufSize = GetFileSize(in, NULL);
buf = VirtualAlloc(NULL
                   , bufSize
                   , MEM_COMMIT | MEM_RESERVE
                   , PAGE_READWRITE);
if(!buf)
    goto $cleanup;

DWORD dataSize = bufSize - hSize;
SetFilePointer(in, hSize, NULL, FILE_BEGIN);
ReadFile(in, buf, dataSize, &readed, NULL);
SetFilePointer(out, rhSize, NULL, FILE_BEGIN);

DWORD written = 0;
WriteFile(out, buf, dataSize, &written, NULL);

// 4. 기존 파일에 새로운 섹션 내용을 추가한다.

if(secPath)
{
    secSize = WriteSection(out
                           , secPath
                           , secSize
                           , fAlign);
    if(secSize == 0)
        goto $cleanup;
}
else if(secSize)
{
    secSize = FillSection(out, secSize, fAlign);
    if(secSize == 0)
        goto $cleanup;
}
else
    goto $cleanup;

// 5. 헤더 필드를 계산해서 갱신한다.

if(rhSize > bufSize)
{
    VirtualFree(buf, 0, MEM_RELEASE);
```

```
        bufSize = rhSize;
        buf = VirtualAlloc(NULL
                        , bufSize
                        , MEM_RESERVE | MEM_COMMIT
                        , PAGE_READWRITE);
    }

    ZeroMemory(buf, rhSize);
    SetFilePointer(in, 0, NULL, FILE_BEGIN);
    ReadFile(in, buf, hSize, &readed, NULL);

    PIMAGE_NT_HEADERS pNt;
    PIMAGE_SECTION_HEADER pSec;

    pNt = (PIMAGE_NT_HEADERS) GetPtr(buf, dos.e_lfanew);
    pSec = (PIMAGE_SECTION_HEADER) GetPtr(buf, secOffset);

    if(pSec[0].PointerToRawData < rhSize)
    {
        DWORD diff = rhSize - pSec[0].PointerToRawData;
        for(int i=0; i<nt.FileHeader.NumberOfSections; ++i)
            pSec[i].PointerToRawData += diff;
    }

    PIMAGE_SECTION_HEADER src = pSec + secNo - 1;
    PIMAGE_SECTION_HEADER dst = pSec + secNo;

#ifdef _UNICODE
    char msecName[20];
    WideCharToMultiByte(CP_ACP, 0, secName, -1
                        , msecName, 20, NULL, NULL);
    StringCbCopyA((LPSTR)pSec[secNo].Name
                    , sizeof(pSec[secNo].Name), msecName);
#else
    StringCbCopy(pSec[secNo].Name
                , sizeof(pSec[secNo].Name)
                , secName);
#endif

    dst->Characteristics = secAttr;
    dst->Misc.VirtualSize = secSize;

    dst->PointerToRawData = src->PointerToRawData
                            + src->SizeOfRawData;
```

```
    dst->SizeOfRawData = MakeAlign(secSize, fAlign);
    dst->VirtualAddress = src->VirtualAddress
                        + src->Misc.VirtualSize, sAlign;

    dst->VirtualAddress = MakeAlign(dst->VirtualAddress
                                    , sAlign);


    ++pNt->FileHeader.NumberOfSections;
    pNt->OptionalHeader.SizeOfHeaders = rhSize;

    sizeOfImage = pNt->OptionalHeader.SizeOfImage
                + dst->Misc.VirtualSize;

    sizeOfImage = MakeAlign(sizeOfImage, sAlign);
    pNt->OptionalHeader.SizeOfImage = sizeOfImage;

    // 6. 새로운 파일의 시작 위치에 헤더를 기록한다.

    SetFilePointer(out, 0, NULL, FILE_BEGIN);
    WriteFile(out, buf, rhSize, &written, NULL);

    ret = TRUE;

$cleanup:
    if(buf)
        VirtualFree(buf, 0, MEM_RELEASE);

    if(in != INVALID_HANDLE_VALUE)
        CloseHandle(in);

    if(out != INVALID_HANDLE_VALUE)
        CloseHandle(out);

    return ret;
}
```

코드가 복잡해서 코드를 설명하는 것보다는 기본적인 아이디어를 설명하는 것이 좋겠다. PE 포맷은 섹션 단위로 관리되고 각 섹션은 독립적으로 존재하기 때문에 섹션을 삭제하고, 추가하는 것이 가능하다. 섹션을 추가한다고 생각했을 때 우리가 해야 할 작업은 무엇일까? 바로 새롭게 추가될 섹션 헤더와 섹션 데이터를 파일에 추가해 주는 것이다.

섹션 헤더를 추가할 때 주의해야 할 점은 두 가지다. 헤더가 꽉 차서 섹션 헤더를 추가할 만한

공간이 존재하지 않는 경우다. 이 때에는 전체 헤더 크기를 증가 시켜야 한다. 또한 전체 헤더 크기를 증가 시킬 때, 첫 번째 섹션이 메모리에 맵핑되는 위치보다 헤더의 크기가 커서는 안 된다는 점을 명심해야 한다. 만약 섹션 헤더를 늘렸다면 이후 나오는 섹션의 본문 부분이 모두 뒤로 증가된 만큼 이동돼야 하기 때문에 섹션 헤더의 PointerToRawData를 뒤로 이동시켜주어야 한다. 두 번째로 주의해야 할 점은 헤더 내에서 마지막 섹션 다음에 정보가 있는 경우다. 주로 바인드 정보가 이 위치에 기록된다. 이 경우에는 바인드 정보는 정보를 새로운 위치로 옮겨 주거나 해당 파일을 지원하지 않는 형태로 처리해야 한다.

섹션 내용을 추가할 때 주의해야 할 점은 섹션 내용의 크기는 FileAlignment의 배수가 되어야 한다는 점이다. 또한 새롭게 추가된 섹션은 마지막 섹션 다음에 오도록 VirtualAddress를 조정해 주어야 한다. 종종 디버그 정보가 섹션에 포함되지 않고 파일 끝에 독립적으로 존재하는 경우가 있다. 이것은 바인드 정보와 마찬가지로 옮겨주거나 지원하지 않는 형태로 처리해야 한다.

끝으로 섹션을 추가하게 되면 필연적으로 SizeOfImage가 변경된다. 이 크기는 추가된 섹션의 크기를 더해서 SectionAlignment에 정렬이 되도록 조정해 준다. 앞서 섹션 헤더를 추가하는 과정에서 헤더 크기를 늘린 경우라면 SizeOfHeaders 필드도 같이 수정해 주어야 한다. FileHeader에 있는 NumberOfSections 필드도 추가한 섹션의 개수만큼 증가시켜 주어야 한다.

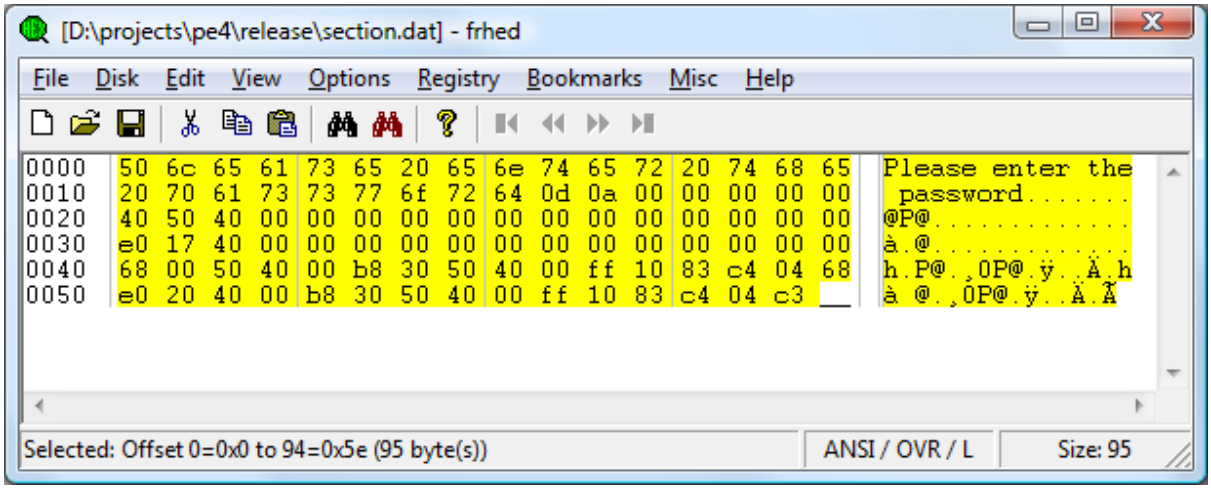
addsection 프로그램의 전체 코드는 이달의 디스크에 들어 있다. addsection 프로그램을 사용해서 simple.exe를 변형해 보도록 하자. simple.exe는 시작할 때 'Password =>'를 출력한다. 새로운 섹션을 추가해서 'Password =>'를 출력하기 전에 'Please enter the password'란 문구를 출력하도록 만들 것이다. 필자가 생각한 답을 보기 전에 직접 한번 풀어보도록 하자.

<리스트 5>에 필자가 생각해본 코드가 나와 있다. 앞서 패딩 공간에 코드를 추가할 때 느꼈겠지만 컴퓨터에게 상대주소는 굉장히 편리한 방식이지만 사람에게는 굉장히 힘든 방식이다. 그래서 이번 코드에는 상대 주소를 사용하지 않도록 만들었다. eax에 0x405030을 넣는 것을 볼 수 있다. 0x405030에는 미리 Print 함수의 주소인 0x4017e0를 기록해 두었다. 이렇게 하면 DLL 함수를 호출하는 것처럼 eax에 0x405030을 넣고 call을 하면 언제든지 Print를 호출할 수 있게 된다. 0x405000에는 'Please enter the password'를 기록해 두었다. 이렇게 전체를 구성하면 <화면 13>과 같이 구성된다.

리스트 5 'Please enter the password'를 출력하는 코드

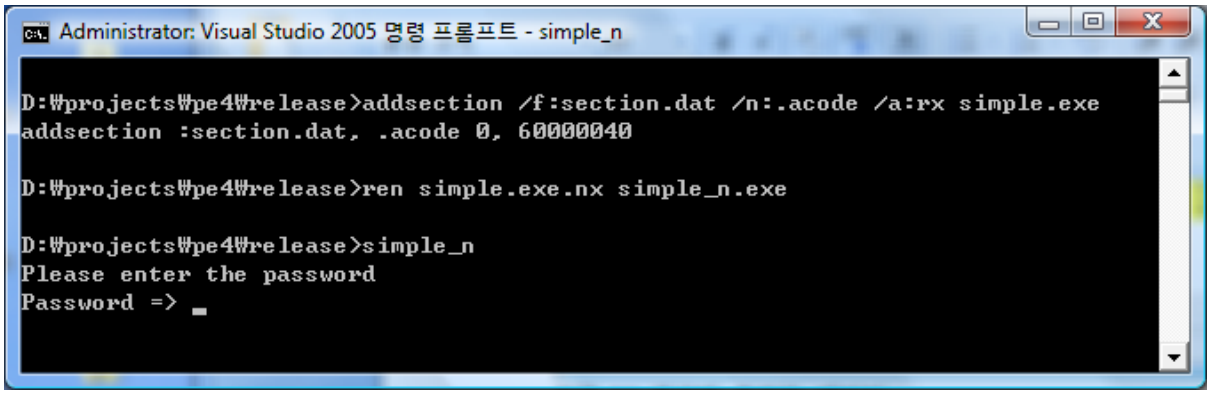
```
68 00 50 40 00     push 0x405000
b8 30 50 40 00     mov eax, 0x405030
ff 10                call dword ptr ds:[eax]
83 c4 04            add esp, 4
68 e0 20 40 00     push 0x4020e0
b8 30 50 40 00     mov eax, 0x405030
ff 10                call dword ptr ds:[eax]
83 c4 04            add esp, 4
```

```
c3                      retn
```



화면 13 섹션에 추가할 데이터

섹션을 simple.exe에 추가한 다음에 simple.exe에서 'Password =>'를 출력하는 부분의 코드를 0x405040을 호출하는 코드로 변경한다. 0x68, 0xe0, 0x20, 0x40, 0x00, 0xe8, 0xa2, 0xff, 0xff, 0xff로 되어 있는 부분을 0xb8, 0x20, 0x50, 0x40, 0x00, 0xff, 0x10, 0x90, 0x90, 0x90으로 고쳐주면 된다. 이 과정이 <화면 14>에 나와 있다.



화면 14 수정한 프로그램을 실행한 화면

도전 과제

마지막에 작성한 AddSection 함수는 바운드 정보나 디버그 정보에 대한 대비가 되어 있지 않다. 해당 정보가 존재하는 경우에는 적절히 옮겨서 충돌이 나지 않도록 만들어보자. 좀 더 관심이 있는 독자라면 조금 거리가 있지만 detour 라이브러리의 소스 코드를 분석해 보는 것도 도움이 될 것 같다.

내용이 이해하기는 힘들지만 관련 분야를 공부해보고 싶다는 생각이 든다면 참고 자료에 있는 것들을 활용하자. 거의 바이블로 불리는 자료들이기 때문에 일독한다면 내공 향상에 큰 도움이 될 것이다. 그리고 참고자료에 있는 인텔 매뉴얼의 경우 주문을 하면 나라에 관계없이 책으로 된 매

뉴얼을 보내준다. x86 시스템 프로그래밍이나 명령어 코드를 분석하는데 큰 도움이 되는 자료이므로 아직 마련하지 못했다면 꼭 주문해서 읽어보도록 하자. 물론 무료다.

참고자료

"Assembly Language For Intel-Based Computers 4/e"
KIP R. IRVINE저, Prentice Hall

"Reversing: Secrets of Reverse Engineering"
Eldad Eilam저, WILEY

"Hacker Disassembling Uncovered"
Kris Kaspersky저, A-List Publishing

Intel® 64 and IA-32 Architectures Software Developer's Manuals
http://www.intel.com/products/processor/manuals/index.htm

What Goes On Inside Windows 2000: Solving the Mysteries of the Loader
http://msdn.microsoft.com/msdnmag/issues/02/03/Loader/

"Windows 시스템 실행 파일의 구조와 원리"
이호동저, 한빛미디어

An In-Depth Look into the Win32 Portable Executable File Format
http://msdn.microsoft.com/msdnmag/issues/02/02/PE/

An In-Depth Look into the Win32 Portable Executable File Format, Part 2
http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
http://msdn2.microsoft.com/en-us/library/ms809762.aspx