

개발자를 위한 윈도우 후킹 테크닉

메시지 훅 이용한 Spy++ 훅내내기

우리는 지난 강좌에서 후킹을 하는 기초적인 방법과 사용되는 API 들을 비롯해서 후킹 함수가 호출되는 컨텍스트에 대해서 배웠다. 이번 강좌에서는 WH_GETMESSAGE 훅을 통해서 Spy++과 유사한 프로그램을 제작할 것이다. 이 과정에서 Windows 애플리케이션이 메시지를 처리하는 과정과 윈도우를 열거하는 방법에 대해서 알아보도록 하자.

목차

목차.....	1
필자 소개	1
연재 가이드.....	2
연재 순서	2
필자 메모	2
메시지는 어떻게 처리될까?	3
특수한 메시지	3
WM_COPYDATA.....	4
Spy++은 무엇에 쓰는 물건인고?.....	5
WH_GETMESSAGE 훅	9
윈도우 핸들을 열거하는 방법.....	10
메시지를 후킹해 보자.....	14
Spy.....	16
도전과제.....	19
참고자료.....	19

필자 소개

신영진 pop@jiniya.net

부산대학교 정보,컴퓨터공학부 4 학년에 재학 중이다. 모자란 학점을 다 채워서 졸업하는 것이 꿈이되 버린 소박한 괴짜 프로그래머. 병역특례 기간을 포함해서 최근까지 다수의 보안 프로그램 개발에 참여했으며, 최근에는 모짜르트에 심취해 있다.

연재 가이드

운영체제: 윈도우 2000/XP

개발도구: 마이크로소프트 비주얼 스튜디오 2003

기초지식: C/C++, Win32 프로그래밍

응용분야: 메시지 모니터링 프로그램

연재 순서

2006. 05 키보드 모니터링 프로그램 만들기

2006. 06 마우스 훅을 통한 화면 캡처 프로그램 제작

2006. 07 메시지훅 이용한 Spy++ 흉내내기

2006. 08 SendMessage 후킹하기

2006. 09 Spy++ 클론 imSpy 제작하기

2006. 10 저널 훅을 사용한 매크로 제작

2006. 11 WH_SHELL 훅을 사용해 다른 프로세스 윈도우 서브클래싱 하기

2006. 12 WH_DEBUG 훅을 이용한 훅 탐지 방법

2007. 01 OutputDebugString 의 동작 원리

필자 메모

이제껏 우리는 후킹 함수 내에서 실행 프로그램으로 정보를 교환하기 위해서 SendMessageTimeout 함수를 사용했었다. 하지만 전달되는 정보에 포인터가 포함된 것도 아니기 때문에 PostMessage 를 사용해도 될 것이란 의문은 누구나 품었을 것이다. 지난 5 월 호에 연재되었던 키보드 모니터링 프로그램의 경우 더더욱 그렇다. 아마도 PostMessage 로 변경해서 테스트 해보신 분들은 알겠지만 PostMessage 로 해당 훅 코드를 수행할 경우 이상한 동작을 한다. 테스트를 안 해봤다면 지금 한번 해보도록 하자. 훅 코드를 PostMessage 로 변경한 다음 훅을 시작 시키고 모니터링 프로그램에서 키보드를 막 눌러보자. 아마도 심하게 몇 번 누르다 보면 어느 순간 화면에 누르지도 않았는데 키보드 정보가 쉬지 않고 올라오는 것을 볼 수 있을 것이다.

그 원인을 얼마 전에 알 수 있었다. 원인은 다른 아닌 훅 코드에서 비교하는 code 변수 값과 관계가 있었다. 통상적으로 우리는 code 가 0 이상인 경우 수행하도록 했었다. 그것이 문제가 된 것이다. code 값이 HC_ACTION 인 경우에만 처리하도록 하면 그러한 문제가 없어진다.

메시지는 어떻게 처리될까?

API 프로그래밍을 한번도 해보지 않은 독자 분들이라면 아마도 애플리케이션에서 메시지를 어떻게 처리하는지 궁금할 것이다. 실제로 이 부분은 간단한 메시지 루프를 통해서 이루어진다. <리스트 1>은 가장 단순한 형태의 메시지 루프다. MFC 프로그램 같은 경우에도 프로그래머가 작성한 부분에 메시지 루프가 없다고 하더라도 MFC 내부적으로 <리스트 1>과 같은 형태의 메시지 루프를 가지고 있다.

리스트 1 간단한 메시지 루프

```
while(GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

메시지 루프가 하는 일은 매우 간단하다. 큐에서 메시지를 꺼내서, 적당히 가공한 다음 해당 메시지를 받을 윈도우의 메시지 핸들러를 호출한다. 이것이 메시지 처리의 전부다.

GetMessage 가 하는 역할은 메시지 큐에서 메시지를 하나 꺼내는 일이다. 메시지가 없는 경우는 메시지가 들어올 때까지 대기한다. 원활하게 메시지를 처리하기 위해서 GetMessage 는 메시지에 우선순위를 두어서 처리한다. 즉, 먼저 들어왔다고 항상 먼저 처리하는 것은 아니다.

TranslateMessage 가 하는 일은 WM_KEYDOWN, WM_SYSKEYDOWN, WM_KEYUP, WM_SYSKEYUP 메시지를 적절한 형태의 WM_CHAR, WM_SYSCHAR, WM_DEADCHAR, WM_SYSDEADCHAR 등의 메시지로 변환할 수 있는 경우에, 변환된 메시지를 메시지 큐에 추가한다. 기본적으로 Windows 시스템의 경우 키보드를 눌렀다 떼면 단순히 WM_KEYDOWN, WM_KEYUP 메시지만 발생한다.

DispatchMessage 가 하는 일은 가져온 메시지를 실제로 처리하는 일이다. 가져온 메시지에 해당하는 윈도우의 메시지 핸들러를 호출하는 역할을 한다고 생각하면 된다.

특수한 메시지

지난 강좌에서 Windows 에서 실행되는 모든 프로그램은 분리된 메모리 공간에서 실행된다고 배웠다. 하지만 몇몇 특수한 메시지는 이러한 사실이 마치 거짓말인 것처럼 만든다. 그 대표적인 예가 WM_GETTEXT 와 WM_SETTEXT 메시지다. WM_GETTEXT 의 경우 윈도우 텍스트를 구해서 저장할 버퍼의 배열을, WM_SETTEXT 의 경우 윈도우 텍스트를

설정할 문자열 포인터를 인자로 받아들인다. 하지만 이 두 메시지 모두 다른 프로세스에 존재하는 윈도우에 사용해도 제대로 동작한다. 지난 시간에 배웠던 이론 대로라면 전달한 포인터는 다른 프로세스로 넘어가게 되면 엉뚱한 번지가 되고 잘못된 메모리 참조 오류가 나와 정상일 것 같지만 그렇지 않은 것이다. 왜 그럴까?

그 비밀은 Windows 내부에 있다. Win16 시절에는 모든 프로세스가 동일한 메모리 공간을 공유한다고 말했었다. WM_GETTEXT 와 WM_SETTEXT 의 경우 그 시절부터 존재하던 아주 기초적인 메시지였다. 따라서 그 시절 프로그램들의 경우 모두 같은 공간에 있기 때문에 다른 프로세스에 대해서도 이러한 메시지를 보내는 일이 빈번했을 것이다. Win32 가 되면서 더 이상 이러한 일은 불법적인 것이 된다. 하지만 Microsoft 의 입장에서 볼 때 그렇게 바꾸게 되면 기존의 Win16 프로그램 중 많은 프로그램에서 오류가 발생할 것이 뻔해 보였다. 그러한 이유로 Microsoft 는 Win32 로 넘어오면서 기존에 사용하던 메시지에 대해서는 포인터 처리를 자동으로 해주자는 방침을 세운다. 이러한 이유로 Win16 부터 존재하던 대다수의 메시지는 지금도 여전히 다른 프로세스로 포인터를 전달해도 아무런 제약 없이 잘 동작하는 것이다. 그 내부에서는 공유 메모리를 할당하고 다른 프로세스로 그 메모리 포인터를 전달하는 등의 복잡한 일이 일어나고 있다.

WM_COPYDATA

그렇다면 앞에서 소개한 Win16 호환 메시지를 제외한 메시지들은 다른 프로세스로 포인터를 전송할 수 없을까? 그렇다. 하지만 Microsoft 에서는 다른 프로세스로 데이터를 전달하는 목적에 사용할 수 있는 범용 메시지를 하나 만들어 두었다. 그것이 바로 WM_COPYDATA 이다.

WM_COPYDATA 의 WPARAM 으로는 메시지를 보내는 윈도우의 핸들이 지정된다. 받을 윈도우가 아니라 보내는 윈도우라는 점에 주의하도록 하자. 만약 보내는 윈도우 핸들이 없는 경우 0 으로 지정하면 된다. LPARAM 으로는 COPYDATASTRUCT 의 포인터가 전달되어야 한다. COPYDATASTRUCT 는 아래와 같이 정의되어 있다(각 필드의 의미는 <표 1> 참고).

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

표 1 COPYDATASTRUCT 구조체 필드 별 의미

필드명	의미
dwData	다른 프로세스로 전달할 정수 데이터.
cbData	lpData 가 가리키는 내용의 데이터 크기.
lpData	다른 프로세스로 전달할 데이터의 포인터.

일반적으로 dwData 에는 현재 보내는 데이터에 대한 식별자가 들어간다. 앞서 말했듯이 WM_COPYDATA 는 범용적으로 설계된 메시지다. 따라서 이를 통해서 여러 데이터가 전송될 수 있기 때문에 각각을 구분할 무엇인가가 필요하다. 그것을 위한 용도로 보통 dwData 를 사용한다. lpData 에는 전달할 데이터의 포인터를, cbData 에는 해당 데이터의 크기를 넣어주면 된다. src 윈도우에서 dest 윈도우로 "hello, world"라는 문자열을 전송하는 경우라면 <리스트 2>과 같이 작성할 수 있다.

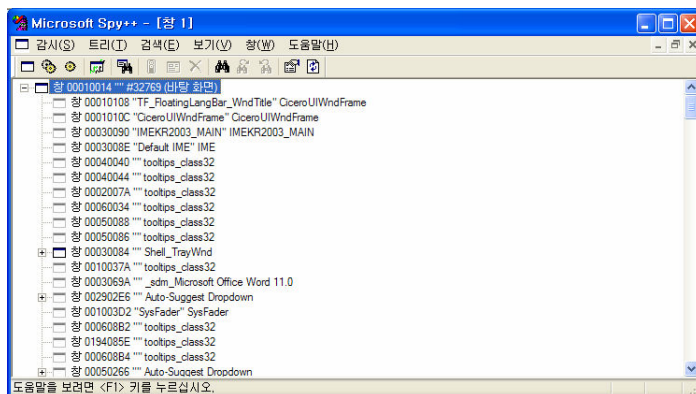
리스트 2 WM_COPYDATA 를 통해서 "hello, world"를 전송하는 예제

```
LPCTSTR data = "hello, world";
COPYDATASTRUCT cds;

cds.dwData = 1; // 주고 받는 쪽에서 약속된 임의의 번호
cds.lpData = data;
cds.cbData = strlen(data);
SendMessage(dest, WM_COPYDATA, (WPARAM) src, (LPARAM) &cds);
```

Spy++은 무엇에 쓰는 물건인가?

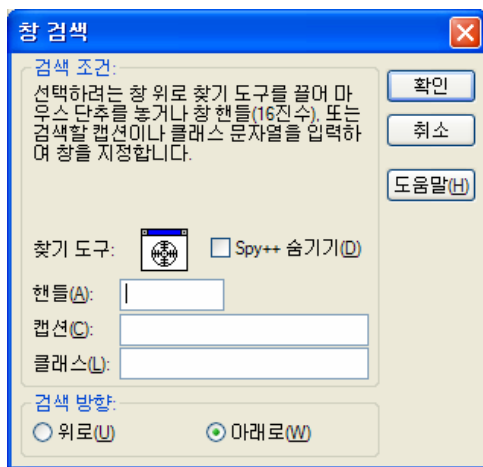
Spy++은 굉장히 유용한 유틸리티이다. 그럼에도 이것이 설명된 문서가 잘 없기에 처음 윈도우 프로그램 개발을 시작하는 분들은 잘 모르는 분들이 더러 있다. 몇 가지 자주 사용하는 기능만 간단히 살펴 보도록 하자.



화면 1 Spy++ 시작 화면

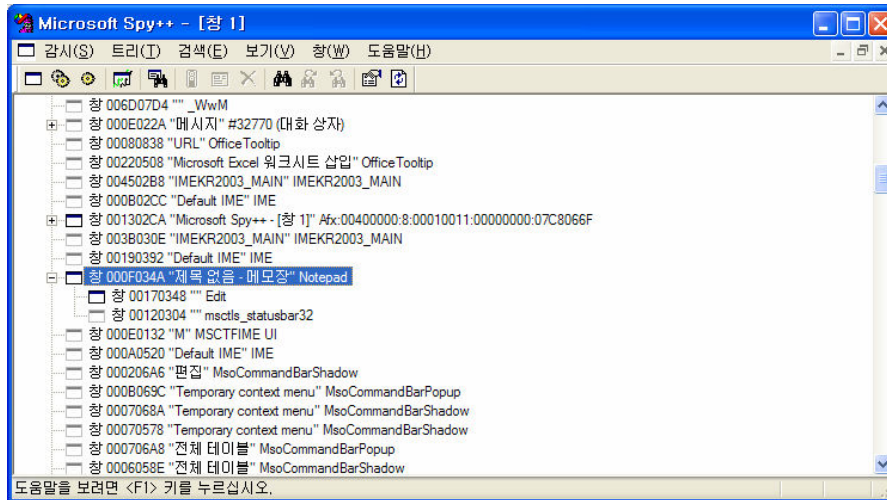
Spy++을 실행 시키면 <화면 1>과 같은 윈도우가 나타난다. 여기에 나타난 목록은 현재 생성된 모든 윈도우의 목록이다. 연하게 표시된 것은 화면에 표시되지 않는 숨김 윈도우이고, 진하게 표시된 것은 화면에 표시되고 있는 윈도우다. 트리를 통해서 자식 윈도우와 부모 윈도우의 관계를 손쉽게 파악할 수 있다.

화면에 나타난 윈도우가 굉장히 많기 때문에 특정 윈도우를 찾는 것이 쉽지 않다. 이러한 작업을 쉽게 하기 위해서 검색 도구가 제공된다. ALT + F3 을 눌러보자. 그러면 <화면 2>와 같은 윈도우가 나타날 것이다.



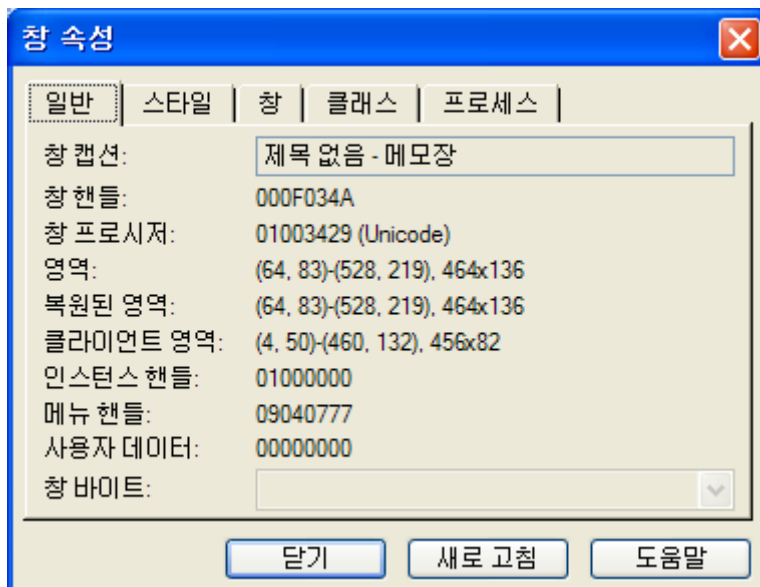
화면 2 Spy++ 찾기 화면

위 윈도우에서 찾기 도구 오른쪽의 아이콘을 드래그 하면 해당 아이콘이 포함된 윈도우를 찾아서 영역을 표시해 준다. 자신이 찾고 싶은 윈도우로 드래그 한 다음 놓으면 된다. 노트패드를 실행 시킨 후 해당 윈도우를 찾아보자. 드래그가 끝난 다음 확인을 누르면 된다.

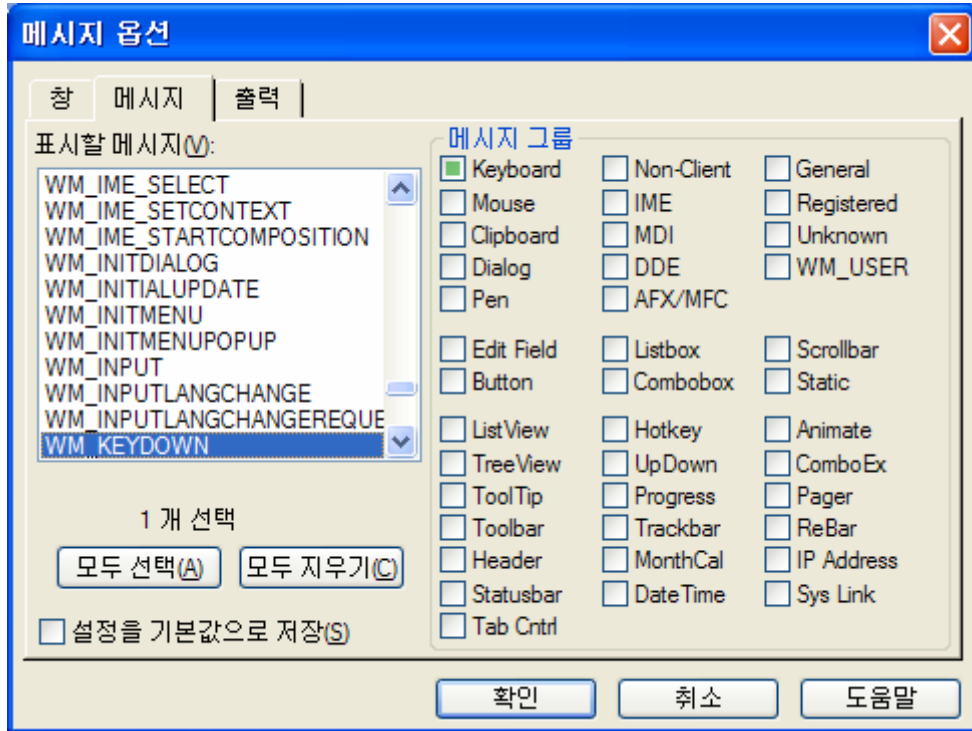


화면 3 Spy++ 에서 노트패드를 찾은 화면

<화면 3>은 Spy++에서 노트패드를 찾은 화면이다. 트리를 열어보면 어떤 자식 윈도우로 구성되어 있는지 알 수 있다. 위에서 선택된 윈도우에서 오른쪽 마우스 버튼을 누른 다음 속성을 선택하면 <화면 4>와 같은 속성 윈도우가 뜬다. 이 곳에서 해당 윈도우의 각종 정보를 알 수 있다.

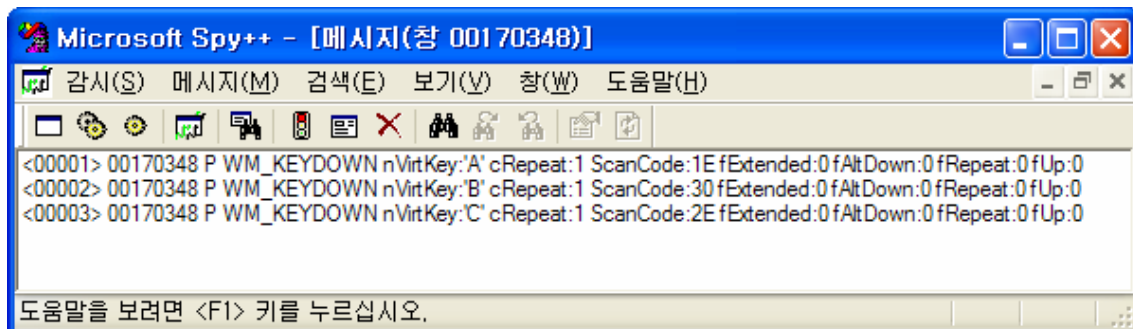


화면 4 Spy++에서 윈도우 속성을 확인하는 화면



화면 5 Spy++ 메시지 후킹 화면

Spy++의 가장 강력한 기능은 메시지 후킹 기능이다. Ctrl + M 을 누르면 <화면 5>와 같은 메시지 창이 뜬다. 여기서 메시지 후킹할 윈도우를 지정한 다음 메시지 탭에서 검사할 메시지를 선택한다. 우리는 WM_KEYDOWN 메시지를 선택했다. 그런 다음 확인을 눌러보자.



화면 6 Spy++을 통해서 노트패드로 전달되는 키보드 메시지를 살펴본 화면

<화면 6>은 위에서 설정한 윈도우에서 a,b,c 키를 누른 화면이다. WM_KEYDOWN 으로 A,B,C 가 발생한 것을 알 수 있다. 위에서 소개한 세 가지 기능이 Spy++에서 가장 많이 사용되는 기능이다. 각종 윈도우를 찾아보고 발생하는 메시지를 살펴 보도록 하자.

WH_GETMESSAGE 훅

WH_GETMESSAGE 훅은 특정 쓰레드로 전달된 메시지가 처리되는 것을 후킹한다. 이 훅의 후킹 프로시저는 쓰레드로 전달된 메시지가 GetMessage 나 PeekMessage 에 의해서 제거되거나 참조 될 때 호출 된다. 따라서 후킹된 쓰레드의 메시지 큐로 들어가서 처리되는 모든 메시지를 살펴볼 수 있다.

```
LRESULT CALLBACK GetMsgProc(int code, WPARAM wParam, LPARAM lParam);
```

code – [입력] code 값이 HC_ACTION 인 경우 훅 프로시저를 수행하고, 0 보다 작은 경우에는 훅 프로시저를 수행하지 않고 CallNextHookEx 를 호출한 다음 리턴 해야 한다.

wParam – [입력] 메시지가 메시지 큐에서 제거되었는지 아닌지를 나타낸다. 이 값의 의미는 <표 2>을 참고하자.

표 2 code 값 의미

값	의미
PM_NOREMOVE	메시지가 메시지 큐에서 제거되지 않고 참조된 경우다. PeekMessage 를 PM_NOREMOVE 플래그를 설정해서 호출한 경우다.
PM_REMOVE	메시지가 메시지 큐에서 제거된 경우다. GetMessage 나 PeekMessage 를 PM_REMOVE 플래그를 설정해서 호출한 경우다.

lParam – [입력] 발생한 메시지 정보를 담고 있는 MSG 구조체의 포인터를 저장하고 있다. MSG 구조체는 아래와 같은 형태를 하고 있다. MSG 구조체의 필드별 의미는 <표 3>를 참고하자.

```
typedef struct {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

표 3 MSG 구조체 필드 별 의미

필드명	의미
hwnd	메시지를 받을 윈도우 핸들.

message	발생한 메시지 ID.
wParam	메시지 wParam.
lParam	메시지 lParam.
time	메시지가 발생한 시간.
pt	메시지가 발생한 마우스 포인터.

리턴 값: code 가 0 보다 작은 경우에는 CallNextHookEx 의 리턴 값을 그대로 리턴 해야 한다. 그렇지 않은 경우에도 CallNextHookEx 의 리턴 값을 그대로 사용하는 것이 좋다. CallNextHookEx 를 통해서 다음 훅 체인을 호출하지 않은 경우엔 0 을 리턴 해야 한다.

윈도우 핸들을 열거하는 방법

Spy++과 같은 프로그램을 만들기 위해서 우리가 가장 먼저 해야 할 일을 윈도우 핸들을 열거하는 것이다. <화면 1>을 보면 현재 시스템에 동작하는 모든 윈도우가 열거되어 있는 것을 볼 수 있다. 이러한 기능을 구현하는데 핵심 역할을 하는 API 두 개를 살펴 보자.

BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);

EnumWindows 는 현재 시스템에 동작중인 top 윈도우를 열거하는 역할을 한다. top 윈도우는 <화면 2>에서 루트에 존재하는 데스크탑 윈도우(바탕 화면)의 직접적인 자식 윈도우들을 말한다. 첫 번째 파라미터로 열거를 수행할 콜백 함수를, 두 번째 인자로 해당 콜백 함수에 전달될 파라미터를 넣어주면 된다. 성공한 경우에 0 이 아닌 값을 리턴 하고, 실패한 경우에 0 을 리턴 한다. 콜백 함수 lpEnumFunc 에서 0 을 리턴 한 경우에도 0 을 리턴 한다.

BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam);

EnumWindows 의 첫 번째 인자로 넘어가는 콜백 함수는 위와 같은 원형을 가지고 있다. 첫 번째 파라미터로 현재 열거된 윈도우의 핸들이, 두 번째 인자로 EnumWindows 로 프로그래머가 전달한 파라미터가 넘어 온다. TRUE 를 리턴 하면 열거 작업이 계속 진행 되고, FALSE 를 리턴 하면 열거 작업이 중단 되고 EnumWindows 함수가 0 을 리턴 한다. <리스트 3>에 EnumWindows 를 사용해서 top 윈도우를 열거하는 코드가 나와있다.

리스트 3 top 윈도우를 열거하는 코드

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
```

```

BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam)
{
    TCHAR windowName[MAX_PATH] = {0,};
    TCHAR className[MAX_PATH] = {0,};

    if(!GetWindowText(hwnd, windowName, sizeof(windowName)) &&
        GetLastError() != ERROR_SUCCESS)
        return FALSE;

    if(!GetClassName(hwnd, className, sizeof(className)) &&
        GetLastError() != ERROR_SUCCESS)
        return FALSE;

    printf( "Window name = \"%s\"\n"
           "   Class name = \"%s\"\n",
           windowName,
           className );

    return TRUE;
}

int main(int argc, char* argv[])
{
    SetLastError(0);
    EnumWindows(EnumWindowsProc, 0);

    return 0;
}

```

```

BOOL EnumChildWindows(HWND hWndParent, WNDENUMPROC lpEnumFunc, LPARAM lParam);

```

EnumChildWindows 는 특정 윈도우의 자식 윈도우를 열거하는 역할을 하는 함수다. 첫 번째 파라미터로 열거할 부모 윈도우의 핸들을, 두 번째 파라미터로 콜백 함수 포인터를, 세 번째 인자로 콜백 함수로 전달될 파라미터를 넣어주면 된다. 성공한 경우 TRUE, 실패한 경우 FALSE 를 리턴 한다. 이 함수의 첫 번째 인자를 NULL 로 주고 호출하면 위에서 소개한 EnumWindows 와 동일한 역할을 수행한다. <리스트 4>는 <리스트 3>의 콜백 부분을 수정해서 자식 윈도우까지 모두 출력하도록 만든 것이다. 수행 결과가 <화면 7>에 나와있다.

리스트 4 EnumChildWindows 를 사용해서 자식 윈도우를 열거하는 콜백 함수

```

BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam)
{
    TCHAR windowName[MAX_PATH] = {0,};
    TCHAR className[MAX_PATH] = {0,};

    if(!GetWindowText(hwnd, windowName, sizeof(windowName)) &&

```

```

GetLastError() != ERROR_SUCCESS)
return FALSE;

if(!GetClassName(hwnd, className, sizeof(className)) &&
    GetLastError() != ERROR_SUCCESS)
return FALSE;

printf( "%sWindow name = \"%s\\\"
        "   Class name = \"%s\\\"\\n",
        lParam,
        "",
        windowName,
        className );

EnumChildWindows(hwnd, EnumWindowsProc, lParam+4);

return TRUE;
}

```

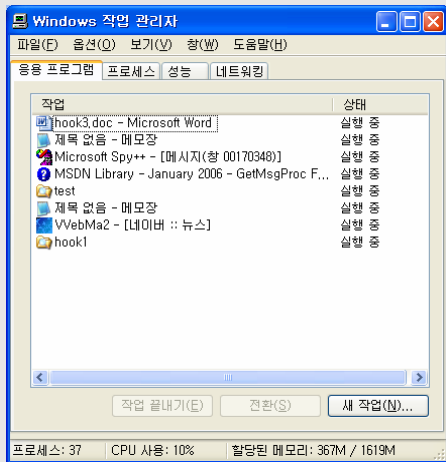
```

Window name = ""   Class name = "MsgIMEWindowClass"
Window name = ""   Class name = "MsnMsgrUIManager"
Window name = "Acrobat IEHelper"   Class name = "Acrobat IEHelper Object"
Window name = ""   Class name = "tooltips_class32"
Window name = ""   Class name = "WorkerW"
Window name = ""   Class name = "ComboBox"
Window name = ""   Class name = "TPUtilWindow"
Window name = ""   Class name = "tooltips_class32"
Window name = ""   Class name = "CSCHiddenWindow"
Window name = "전원 측정기"   Class name = "SystemTray_Main"
Window name = "전원 상태"   Class name = "Button"
Window name = "작업 표시줄에 항상 아이콘 표시(&A)"   Class name = "Button"
Window name = "각 배터리의 정보 표시(&B)"   Class name = "Button"
Window name = ""   Class name = "Static"
Window name = ""   Class name = "#32770"
Window name = ""   Class name = "Static"
Window name = "현재 전원:"   Class name = "Static"
Window name = "남은 총 배터리 양:"   Class name = "Static"
Window name = ""   Class name = "msctls_progress32"
Window name = "알 수 없음"   Class name = "Static"
Window name = "AC 전원"   Class name = "Static"
Window name = "<충전>"   Class name = "Static"
Window name = "100%"   Class name = "Static"
Window name = ""   Class name = "Button"
Window name = "없음"   Class name = "Static"

```

화면 7 자식 윈도우 열거 화면

박스 1 응용 프로그램을 열거하는 방법



화면 8 작업 관리자에 나타난 응용 프로그램 목록

EnumWindows 를 사용하면 작업 표시줄에 나타나는 응용 프로그램을 열거할 수 있다. 작업 관리자는 화면에 표시되는 top 윈도우 중에서 다음 속성을 만족하는 것을 열거한다.

WS_EX_APPWINDOW 속성을 가지고 있다.

WS_EX_TOOLWINDOW 속성을 가지고 있지 않으면서, 소유주(owner) 윈도우가 없다.

위의 두 가지 조건을 체크하는 코드를 만들어 보면 다음과 같다.

```
DWORD exStyle = GetWindowLong(hwnd, GWL_EXSTYLE);

BOOL isVisible = IsWindowVisible(hwnd);
BOOL isToolWindow = (exStyle & WS_EX_TOOLWINDOW);
BOOL isAppWindow = (exStyle & WS_EX_APPWINDOW);
BOOL isOwned = GetWindow(hwnd, GW_OWNER) ? TRUE : FALSE;

if(isVisible && (isAppWindow || (!isToolWindow && !isOwned)))
{
    // 응용 프로그램
}
```

위에서 if 문을 통과한 top 윈도우를 열거해 보면 작업 관리자의 응용 프로그램에 나타난 윈도우 목록과 동일함을 알 수 있다.

메시지를 후킹해 보자.

메시지를 후킹하는 기본적인 함수들은 지난 시간에 소개한 것과 모두 동일하기 때문에 별도로 설명하지 않겠다. 새롭게 추가된 부분만 살펴 보도록 하자. <리스트 5>에 InstallHookEx 함수의 코드가 나와있다. 이 함수는 기존에 사용하던 InstallHook 과 동일한 역할을 한다. 하지만 다른 점은 기존의 InstallHook 의 경우 훅을 전역으로 설치 했었다. 반면에 InstallHookEx 의 경우 인자로 받아들인 쓰레드 ID 에 해당하는 쓰레드만 후킹한다.

리스트 5 InstallHookEx 코드

```
MSGHK_API BOOL WINAPI
InstallHookEx(DWORD tid)
{
    BOOL ret = FALSE;

    if(!g_hHook)
    {
        g_hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMessageProc, g_hInst, tid);
        if(g_hHook)
            ret = TRUE;
    }

    return ret;
}
```

GetMessage 와 PeekMessage 모두 모든 윈도우에서 기본적으로 가지고 있는 메시지 루프의 일부분 이기 때문에 호출 빈도가 매우 높기 때문에 전역으로 후킹할 경우 시스템 성능이 상당히 저하될 수 있다. 따라서 메시지를 살펴보고 싶은 윈도우를 생성한 쓰레드만 후킹하는 것이 바람직하다. <리스트 6>는 InstallHookEx 를 사용해서 특정 윈도우에 해당하는 쓰레드만 후킹 하는 방법을 보여주고 있다.

리스트 6 InstallHookEx 를 사용해서 후킹하는 부분

```
void CspyDlg::OnBnClickedHook()
{
    // 선택한 항목의 핸들을 얻어온다.
    HTREEITEM item = m_treeWindows.GetSelectedItem();
    if(!item)
    {
        AfxMessageBox("훅을 설치할 윈도우를 선택해 주세요");
        return;
    }

    // 해당 핸들에 대한 윈도우 핸들을 구한다.
    HWND hwnd = (HWND) m_treeWindows.GetItemData(item);
    DWORD tid = GetWindowThreadProcessId(hwnd, NULL);
```

```
// 윈도우를 생성한 스레드를 후킹한다.
if(!InstallHookEx(tid))
{
    AfxMessageBox("훅 설치에 실패하였습니다.");
    return;
}

// 후킹할 메시지 맵 생성
m_hookMsgs.clear();
for(int i=0; i<g_msgsCnt; ++i)
{
    if(m_lstMsgs.GetSel(i) > 0)
        m_hookMsgs.insert(make_pair(g_msgs[i].id, g_msgs[i].desc));
}

// 리스트 초기화
m_lstHookMsg.DeleteAllItems();
}
```

WH_GETMESSAGE 훅 프로시저는 <리스트 7>에 나와있다. WPARAM 값이 PM_REMOVE 인 경우만 체크해서 메시지가 실제로 메시지 큐에서 꺼내지는 시점에만 메시지를 보내도록 되어 있다. 메시지 정보를 다른 프로세스로 보내기 위해서 WM_COPYDATA 를 사용하고 있다.

리스트 7 WH_GETMESSAGE 훅 프로시저

```
LRESULT CALLBACK
GetMessageProc(int code, WPARAM w, LPARAM l)
{
    if(code == HC_ACTION && w == PM_REMOVE)
    {
        PMSG msg = (PMSG) l;

        if(IsWindow(g_targetWnd))
        {
            COPYDATASTRUCT cds;

            cds.cbData = sizeof(MSG);
            cds.lpData = msg;
            cds.dwData = g_callbackMsg;

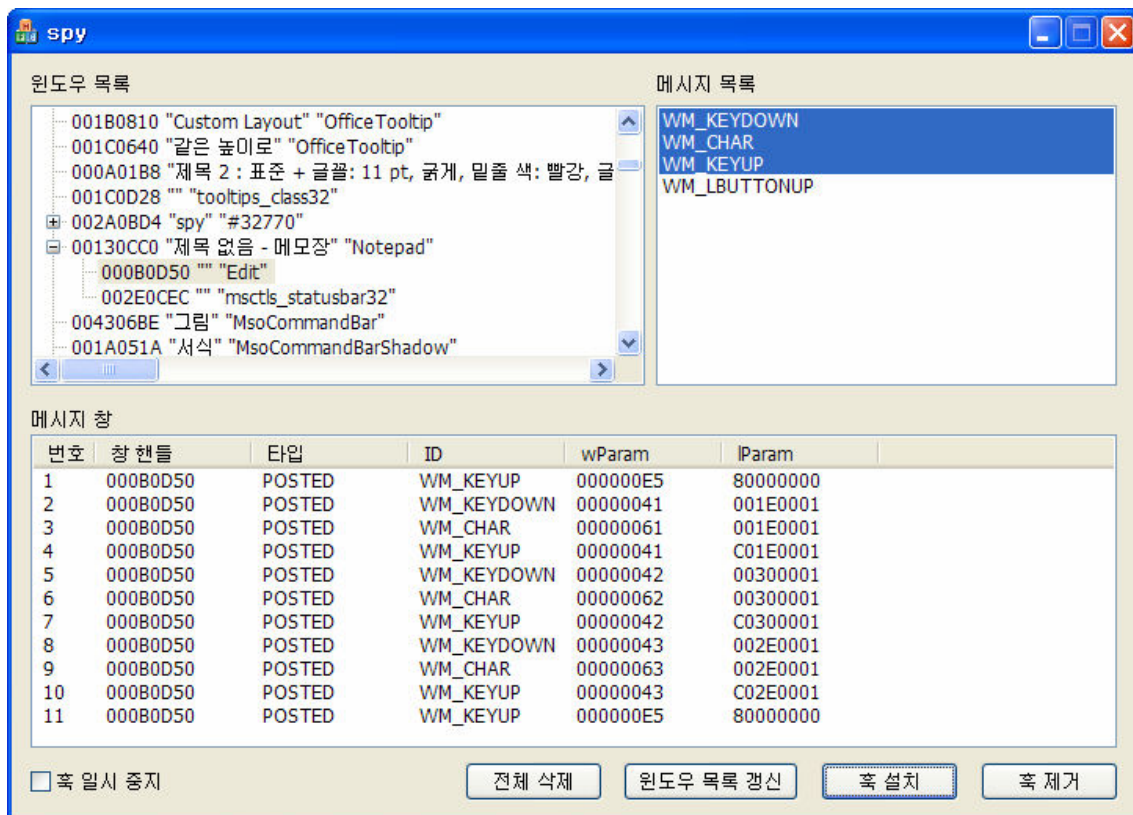
            SendMessageTimeout( g_targetWnd,
                                WM_COPYDATA,
                                0,
                                (LPARAM) &cds,
                                SMTO_BLOCK|SMTO_ABORTIFHUNG,
                                50,
                                NULL );
        }
    }
}
```

```
return CallNextHookEx(NULL, code, w, l);
}
```

Spy

이번 강좌에서 우리가 제작해 볼 프로그램은 Spy++의 변종 Spy 프로그램이다. Spy의 실행 화면이 <화면 9>에 나와있다. 화면은 전체적으로 세 부분으로 나뉘어져 있다. 왼쪽 위에 나오는 부분이 윈도우 목록을 열거하는 부분 이다. 그 옆으로 있는 리스트는 후킹해서 검사할 메시지 목록을 나타내고 있다. 지금은 네 개의 메시지만 등록되어 있다. 아래 쪽에 있는 목록은 후킹한 윈도우에서 발생한 메시지를 나타낸다.

훅 일시 중지 버튼이 선택되면 훅이 설치된 상태라도 훅 메시지가 올라오지 않는다. 체크가 해제 되면 다시 메시지가 올라온다. 전체 삭제 버튼을 누르면 지금까지 추가되었던 후킹 메시지가 모두 제거된다. 윈도우 목록 갱신 버튼은 윈도우 목록을 최신의 상태로 업데이트 시키는 역할을 한다. 훅 설치 버튼은 윈도우 목록에서 선택한 윈도우에 메시지 목록에 해당하는 메시지를 후킹하는 훅 프로시저를 설치하는 역할을 한다. 훅 제거 버튼은 설치된 훅을 제거하는 역할을 한다.



화면 9 WM_GETMESSAGE 훅을 사용한 Spy 프로그램 실행 화면

윈도우 목록을 트리로 표현하는 것과 관련된 코드가 <리스트 8>에 나타나 있다. 앞서 소개한 EnumWindows 와 EnumChildWindows 함수를 사용하고 있다. 트리에 표시하기 위해서 트리 컨트롤의 핸들과 추가할 아이템 노드를 콜백 함수의 파라미터로 전달하고 있다. 트리에 뭔가를 재귀적으로 추가하는 경우에 많이 사용되는 코드 이므로 한번 찬찬히 살펴보도록 하자.

리스트 8 윈도우 목록을 구해서 트리로 만드는 코드

```
// FillTreeProc 콜백 파라미터 구조체
typedef struct _FTCBPARAM
{
    CTreeCtrl *pTree; // 트리 핸들
    HTREEITEM parent; // 부모 아이템 핸들
} FTCBPARAM, *PFTCBPARAM;

BOOL CALLBACK
FillTreeProc(HWND hwnd, LPARAM p)
{
    FTCBPARAM np;
    PFTCBPARAM param = (PFTCBPARAM) p;
    TCHAR windowName[1024];
    TCHAR className[1024];
    TCHAR title[1024];

    // 윈도우 이름과 클래스 이름을 구한 다음 추가할 문자열을 만든다.
    GetWindowText(hwnd, windowName, sizeof(windowName));
    GetClassName(hwnd, className, sizeof(className));
    StringCbPrintf(title, sizeof(title), "%08X \"%s\" \"%s\"", hwnd, windowName,
        className);

    // 아이템을 추가한 다음 자식을 열거한다.
    np.pTree = param->pTree;
    np.parent = param->pTree->InsertItem(title, 0, 0, param->parent, TVI_LAST);
    param->pTree->SetItemData(np.parent, (DWORD_PTR) hwnd);
    EnumChildWindows(hwnd, FillTreeProc, (LPARAM) &np);
    return TRUE;
}

void CspyDlg::RefreshWindowList()
{
    FTCBPARAM param;

    param.pTree = &m_treeWindows;
    param.parent = TVI_ROOT;

    m_treeWindows.LockWindowUpdate();
    m_treeWindows.DeleteAllItems();
    EnumWindows(FillTreeProc, (LPARAM) &param);
}
```

```
m_treeWindows.UnlockWindowUpdate();
}
```

실제로 후킹된 메시지를 리스트에 표시하는 메시지 핸들러는 <리스트 9>에 코드가 나와있다. 앞서 살펴 보았듯이 이번 강좌에 사용된 메시지 후킹의 경우 정보 전달을 위해서 WM_COPYDATA 를 사용했었다. 따라서 후킹 메시지도 WM_COPYDATA 에서 처리되어야 한다. 훅의 일시 중지 상태와 콜백 메시지 여부를 확인한 다음 처리하고 있다. 우리가 모니터링 하고 싶은 메시지인지를 빠르게 확인하기 위해서 맵을 사용했다.

리스트 9 후킹 메시지 핸들러

```
BOOL CspyDlg::OnCopyData(CWnd* pWnd, COPYDATASTRUCT* pCopyDataStruct)
{
    if(!m_hookPaused && pCopyDataStruct->dwData == WM_MSGH00KNOTIFY)
    {
        CString buf;
        int id;
        PMSG msg = (PMSG) pCopyDataStruct->lpData;

        if(msg->hwnd == m_hookWnd &&
m_hookMsgs.find(msg->message) != m_hookMsgs.end())
        {
            id = m_lstHookMsg.GetItemCount();

            buf.Format("%d", id+1);
            m_lstHookMsg.InsertItem(LVIF_TEXT, id, buf, 0, 0, 0, 0);

            buf.Format("%08X", msg->hwnd);
            m_lstHookMsg.SetItemText(id, 1, buf);

            m_lstHookMsg.SetItemText(id, 2, "POSTED");

            m_lstHookMsg.SetItemText(id, 3, m_hookMsgs[msg->message]);

            buf.Format("%08X", msg->wParam);
            m_lstHookMsg.SetItemText(id, 4, buf);

            buf.Format("%08X", msg->lParam);
            m_lstHookMsg.SetItemText(id, 5, buf);
        }
    }

    return CDialog::OnCopyData(pWnd, pCopyDataStruct);
}
```

메시지 맵과 관련된 코드가 <리스트 10>에 나와있다. 기본적으로 화면에 보이는 메시지 목록은 배열을 통해서 미리 저장해 둔 것을 출력한다. 그 중에 사용자가 선택한 메시지만 맵으로 들어가게 된다. 맵에 메시지를 추가하는 부분은 <리스트 6> 코드를 참고하자.

리스트 10 메시지 맵과 관련된 코드

```
// 메시지 목록 배열 구조체 부분

typedef struct _MSGDESC
{
    UINT    id;
    LPCTSTR desc;
} MSGDESC, *PMSGDESC;

MSGDESC g_msgs[] =
{
    { WM_KEYDOWN,    "WM_KEYDOWN"    },
    { WM_CHAR,       "WM_CHAR"       },
    { WM_KEYUP,      "WM_KEYUP"      },
    { WM_LBUTTONDOWN, "WM_LBUTTONDOWN" },
};

const int
g_msgsCnt = sizeof(g_msgs) / sizeof(MSGDESC);

// 사용자가 선택한 목록만 저장할 맵

typedef std::map<UINT, LPCTSTR>          MsgMap;
typedef std::map<UINT, LPCTSTR>::iterator MsgMit;

MsgMap    m_hookMsgs;    // 후킹할 메시지 목록
```

도전과제

이번 달에 만든 예제 프로그램도 부족한 부분이 너무도 많다. 좀 더 그럴 듯 하게 만들어 보도록 하자. 우리가 추가한 네 개의 메시지보다 많은 수의 메시지를 손쉽게 추가할 수 있는 방법을 생각해 보자. 그리고 개별 메시지 마다 파라미터를 보기 좋게 출력하는 방법을 연구해 보자. Spy++의 경우 WPARAM과 LPARAM의 형태로 나타내는 것이 아니라, 각각이 의미하는 바를 정리해서 보여준다.

다음 달은 WH_CALLWNDPROC과 WH_CALLWNDPROCRET를 사용해서 SendMessage를 후킹해서 보여주는 기능을 Spy 프로그램에 추가하는 것을 배울 것이다. 여러 개의 윈도우를 후킹했을 때 그것들을 관리하는 방법과 WM_COPYDATA 외의 다른 자료 교환 방법에 대해서 한번 생각해 보도록 하자.

참고자료

참고자료 1. Jeffrey Richter. <<Programming Applications for Microsoft Windows (4/E)>> Microsoft Press

- 참고자료 2. 김상형, <<Windows API 정복>> 가남사
- 참고자료 3. 김성우, <<해킹/파괴의 광학>> 와이미디어
- 참고자료 4. Spy++ 매뉴얼 - <http://www.winapi.co.kr/toollec/Spy/Spy.htm>