**개발자를 위한 윈도우 후킹 테크닉**

# Spy++ 클론 imSpy 제작하기

지난 시간까지 우리는 WH_GETMESSAGE, WH_CALLWNDPROC, WH_CALLWNDPROCRET 을 사용하는 이론적인 방법과 동기화 방법에 대해서 살펴보았다. 이번 시간에는 세 가지 훅을 사용해서 Spy++과 같이 메시지 처리 과정을 보여주는 프로그램을 작성할 것이다.

## 목차



## 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

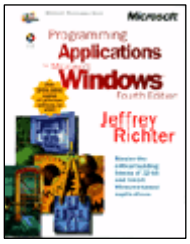
응용분야: 메시지 모니터링 프로그램

## 연재 순서

2006. 05 키보드 모니터링 프로그램 만들기
2006. 06 마우스 훅을 통한 화면 캡쳐 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

훌륭한 책은 마음의 양식이 되기도 하고 한 사람의 삶을 바꾸어 놓기도 한다. 물론 기술 서적중에 이러한 책은 잘 없지만, 저자의 깊은 이해와 설명에 감탄이 나오는 책은 종종 만날 수 있다.



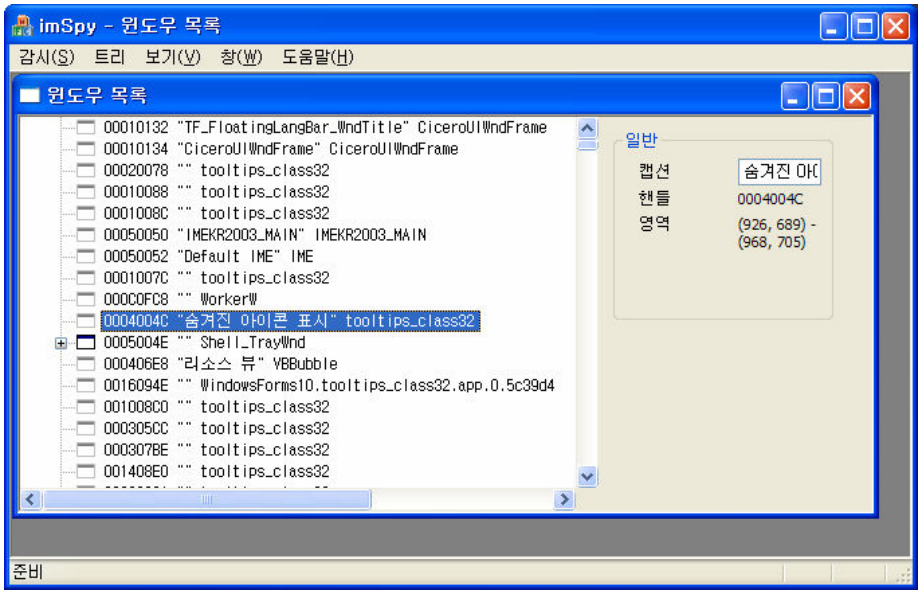

Windows 프로그래밍 관련 책 중에 위와 같은 느낌을 받은 책을 한 권만 꼽으라면 주저 없이 Jeffrey Richter 의 Programming Applications for Microsoft Windows(4/e)를 선택할 것이다. 이 책은 Windows 시스템 프로그래밍의 거의 대부분의 영역을 커버하고 있으며, Windows 시스템의 동작 원리에 대해서도 자세하게 설명하고 있다. 아직까지 책을 읽어보지 않았다면 꼭 읽어 보길 권하고 싶다.

필자가 이 책을 처음 접한 건 2003 년 이었다. 하지만 지금까지도 이 책을 볼 때면 눈물을 흘린다. 이 책을 읽고 난 후에는 Windows 시스템에 대해서 좀 더 깊이 있는 이해를 할 수 있게 될 것이다.

## Intro

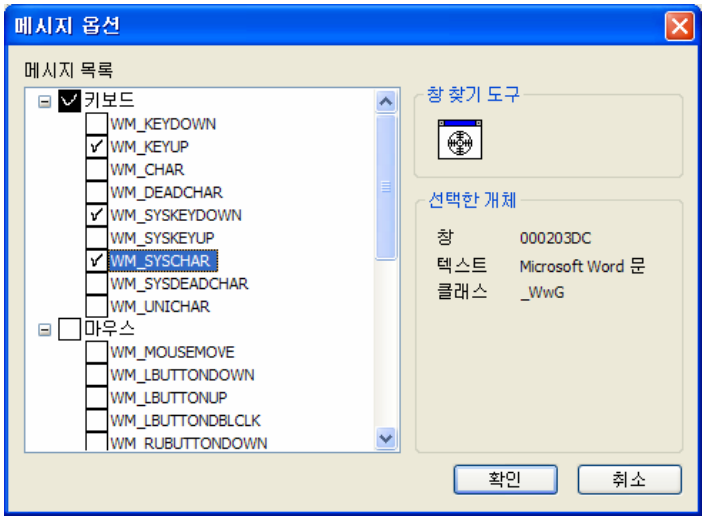
이번 시간에 우리는 지금까지 공부했던 세 가지 훅(WH_GETMESSAGE, WH_CALLWNDPROC, WH_CALLWNDPROCRET)를 사용해서 윈도우로 전송되는 메시지를 모니터링 하는 프로그램을 작성해 볼 것이다. 이 프로그램이 하는 일은 Spy++과 유사하다. Spy++의 기능

중 일부를 구현한 것이라 생각하면 편하다. 우리가 제작할 imSpy 를 간단히 살펴보도록 하자.



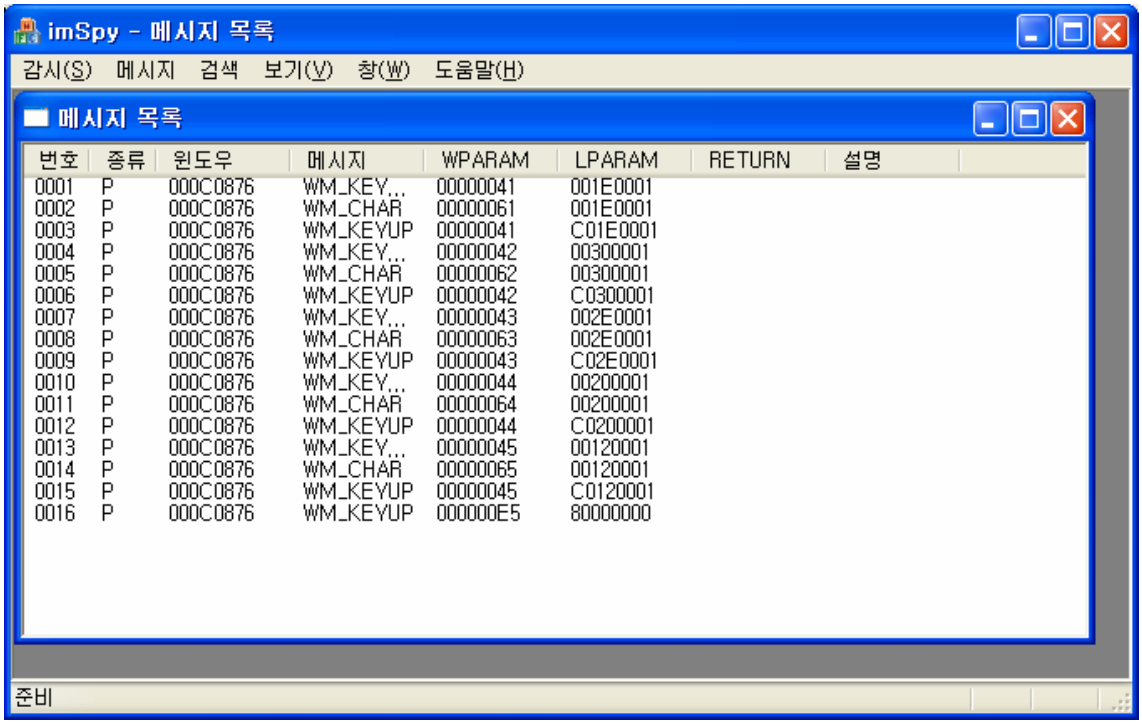

**화면 1 imSpy 에서 윈도우 목록을 표시한 화면**

<화면 1>은 imSpy 에서 윈도우 목록을 표시한 화면이다. 기본적으로 Spy++과 동일하게 트리뷰로 윈도우의 계층 구조를 표시해 주며, 오른쪽에는 선택된 윈도우에 대한 일부 속성을 구해서 표시해 준다.




**화면 2 imSpy 에서 메시지 옵션을 설정하는 화면**

감시 메뉴에서 로그 메시지를 선택하면 <화면 2>와 같은 대화 상자가 표시된다. 왼쪽에는 모니터링할 메시지 목록이 트리 형태로 나타난다. 오른쪽에는 창 찾기 도구와 선택한 창에 대한 속성이 나타난다. 창 찾기 도구는 Spy++과 마찬가지로 드래그를 통해서 윈도우를 찾는 기능을 한다.




**화면 3 imSpy 에서 메시지 후킹을 통해서 메시지를 확인하는 화면**

<화면 3>은 imSpy 를 사용해서 키보드 메시지를 모니터링 하고 있는 화면이다. 종류는 Spy++과 마찬가지로 P, S, R 이 출력된다. RETURN 부분에는 SendMessage 로 전달된 메시지가 리턴될 때의 결과 값이 출력된다.

## 공용 자료들

imSpy 전역에서 공통으로 사용되는 자료 형이 <리스트 1>에 나와있다. 주석에 개략적인 설명이 나와있다. IMSPYMSGDATA 구조체는 파일 매핑 오브젝트에 저장되는 내용이다. type 은 메시지가 발생된 상황을 나타낸다. 우리는 세 개의 훅을 설치해서 메시지 정보를 받기 때문에 총 세 가지 종류가 나올 수 있다. MSGT_POST, MSGT_SEND, MSGT_SENDRET 가 그것이다. result 는 type 이 MSGT_SENDRET 인 경우에만 의미를 가진다.

**리스트 1 공용 자료들**

```
const UINT MSGT_POST       = 1; // WH_GETMESSAGE
```

```
const UINT MSGT_SEND       = 2; // WH_CALLWNDPROC
const UINT MSGT_SENDRET    = 4; // WH_CALLWNDRETPROC

typedef struct _IMSPYMSGDATA
{
   UINT   type;            // 메시지 종류

   HWND   hwnd;            // 메시지 발생 윈도우
   UINT   message;         // 메시지 번호
   WPARAM wParam;          // WPARAM
   LPARAM lParam;          // LPARAM
   LRESULT result;         // SendMessage 리턴 값
   BYTE   param[2048];     // 메시지 관련 정보

} IMSPYMSGDATA, *PIMSPYMSGDATA;

#ifndef _T
#define _T TEXT
#endif

LPCTSTR const IMSPY_MUTEXNAME = _T("imspy_mutex");
LPCTSTR const IMSPY_BUFFER_EVENTNAME = _T("imspy_bufferReady");
LPCTSTR const IMSPY_DATA_EVENTNAME = _T("imspy_dataReady");
LPCTSTR const IMSPY_FILEMAPNAME = _T("imspy_filemap");
LPCTSTR const IMSPYHK_DLLNAME = _T("imspyhk.dll");
```

param 은 메시지 별로 추가적인 정보를 담는 역할을 한다. 메시지로 전달되는 WPARAM 이나 LPARAM 에 포인터가 포함된 경우 해당 포인터에 대한 정보는 컨텍스트가 변경되면 알 수 없다. 이러한 정보를 훅 프로시저 내에서 복사하는데 사용한다. 이러한 메시지의 대표적인 예로 WM_WINDOWPOSCHANGED 가 있다. 이 메시지의 경우 LPARAM 으로 WINDOWPOS 구조체의 포인터가 넘어 온다. 이 경우에 WINDOWPOS 구조체를 훅 프로시저에서 복사해 두지 않는다면 나중에 해당 내용을 참조할 수가 없다.

다음으로 나오는 문자열들은 전역 커널 오브젝트의 이름을 나타낸다. 각 오브젝트가 하는 일은 <표 1>에 나타나 있다. IMSPYHK_DLLNAME 은 imSpy 에서 사용할 후킹 DLL 모듈의 이름을 나타낸다.

**표 1 커널 오브젝트 역할**

| 오브젝트 이름 | 역할 |
|---|---|
| IMSPY_MUTEXNAME | imSpy 의 중복 실행을 방지하는 역할을 한다. 또한 훅 프로시저에서 무한 대기를 방지하는 용도로 사용된다. |
| IMSPY_BUFFER_EVENTNAME | 파일 맵 오브젝트에 훅 프로시저가 접근해도 됨을 알리는 이벤트다. |

| | |
|---|---|
| IMSPY_DATA_EVENTNAME | 파일 맵 오브젝트에 훅 프로시저가 데이터 기록을 완료했음을 알리는 이벤트다. |
| IMSPY_FILEMAPNAME | 프로세스간 통신을 위해 사용되는 파일 맵 오브젝트다. |

## 훅 프로시저

세 가지 훅에 대한 훅 프로시저 코드가 <리스트 2>에 나타나 있다. GetMessageProc 은 WH_GETMESSAGE 훅에 대한, CallWndProc 은 WH_CALLWNDPROC 에 대한, CallWndRetProc 은 WH_CALLWNDPROCRET 에 대한 훅 프로시저다. 세 개의 훅 프로시저 모두 두 개의 헬퍼 함수를 사용해서 작업을 수행한다. 메시지 정보를 수집하는 FillMsgData 함수와, 수집된 메시지를 응용 프로그램에게 전달하는 NotifyMsgData 가 그것이다.

**리스트 2 훅 프로시저 코드**

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

      FillMsgData(&data,
MSGT_POST,
msg->hwnd,
msg->message,
msg->wParam,
msg->lParam,
0 );
      NotifyMsgData(&data);
   }

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

LRESULT CALLBACK
CallWndProc(int code, WPARAM w, LPARAM l)
{
   LRESULT ret = CallNextHookEx(NULL, code, w, l);
   if(code == HC_ACTION)
   {
      PCWPSTRUCT cwp = (PCWPSTRUCT) l;
      IMSPYMSGDATA data;

      FillMsgData(&data,
MSGT_SEND,
cwp->hwnd,
cwp->message,
```

```
cwp->wParam,
cwp->lParam,
0 );
       NotifyMsgData(&data);
   }

   return ret;
}

LRESULT CALLBACK
CallWndRetProc(int code, WPARAM w, LPARAM l)
{
   LRESULT ret = CallNextHookEx(NULL, code, w, l);
   if(code == HC_ACTION)
   {
       PCWPRETSTRUCT cwpr = (PCWPRETSTRUCT) l;
       IMSPYMSGDATA data;

       FillMsgData(   &data,
                   MSGT_SENDRET,
                   cwpr->hwnd,
                   cwpr->message,
                   cwpr->wParam,
                   cwpr->lParam,
                   cwpr->lResult );

       NotifyMsgData(&data);
   }

   return ret;
}
```

FillMsgData 함수가 하는 일은 인자로 넘어온 정보를 이용해서 PIMSPYMSGDATA 의 각 필드를 설정하는 것이다(<리스트 3> 참고). 필드 설정이 끝나고 나면 param 데이터를 복사하는 부분이 나온다. 현재 함수에는 두 가지 메시지에 대한 param 정보 설정 코드만 나타나 있다. 실제 Spy++과 같이 전체 메시지에 대한 구조체 정보를 표시하기 위해서는 이 부분에 포인터가 전달되는 모든 메시지에 대한 처리 루틴을 넣어야 한다.

**리스트 3 FillMsgData 코드**

```
void FillMsgData(   PIMSPYMSGDATA data,
                UINT type,
                HWND hwnd,
                UINT msg,
                WPARAM w,
                LPARAM l,
                LRESULT result )
{
   ASSERT(data != NULL);
```

```
    data->type = type;
    data->hwnd = hwnd;
    data->message = msg;
    data->wParam = w;
    data->lParam = l;
    data->result = result;

    switch(msg)
    {
    case WM_WINDOWPOSCHANGED:
        CopyMemory(data->param, (PWINDOWPOS) l, sizeof(WINDOWPOS));
        break;

    case WM_NEXTMENU:
        CopyMemory(data->param, (PMDINEXTMENU) l, sizeof(MDINEXTMENU));
        break;
    }
}
```

<리스트 4>는 FillMsgData 함수를 통해서 수집된 정보를 실제로 응용 프로그램으로 전달되는 부분이다. NotifyMsgData 는 일단 버퍼에 기록을 허용하는 이벤트를 대기한다. 해당 이벤트가 발생하면 FillMsgData 에서 지역 변수에 수집된 메시지를 파일 맵 오브젝트로 복사를 한다. 그리고 데이터 기록이 완료되었다는 이벤트를 전달한다. 여기서 굳이 지역 변수를 사용해서 메시지를 수집한 주된 이유는 훅 프로시저가 지나치게 오랜 시간 동안 잠기지 않도록 하기 위해서다. (물론 지금은 FillMsgData 가 굉장히 간단하기 때문에 CopyMemory 하는 것이 더 오래 걸릴 수도 있다.)

SafeWaitForSingleObject 는 Wait 함수에 의해서 훅 프로시저가 무한 대기에 빠지는 현상을 방지하기 위해서 사용된 것이다. 메인 프로그램이 강제 종료되면 대기 중인 이벤트에 대한 처리를 마무리 할 수 없다. 만약 운이 나쁘게 아주 좋지 않은 상황에 메인 프로그램이 강제 종료 된다면 훅 루틴은 메인 프로그램이 다시 기동하기 전까지 잠기게 된다. 이러한 상황을 방지하기 위해서 메인 프로그램이 동작 중인지를 체크해서 없는 경우에는 대기하지 않고 빠져나가도록 처리한 것이다. 이와 같은 방법을 사용함으로써 우리는 최악의 상황에도 훅 프로시저는 500ms 이하의 대기만 수행한다는 보장을 할 수 있다.

**리스트 4 메시지 정보를 응용 프로그램으로 전달하는 함수들**

```
DWORD WINAPI
CheckMutex(LPCTSTR mutexName)
{
    HANDLE h = OpenMutex(SYNCHRONIZE, FALSE, mutexName);
    if(h)
    {
        CloseHandle(h);
        return TRUE;
```

```
    }

    return FALSE;
}

DWORD WINAPI
SafeWaitForSingleObject(HANDLE h, LPCTSTR mutexName)
{
    DWORD r = WAIT_FAILED;

    while(CheckMutex(mutexName))
    {
        r = WaitForSingleObject(h, 500);
        if(r != WAIT_TIMEOUT)
            break;
    }

    return r;
}

void NotifyMsgData(PIMSPYMSGDATA data)
{
    if(SafeWaitForSingleObject(g_bufferReadyEvent, IMSPY_MUTEXNAME) == WAIT_OBJECT_0)
    {
        CopyMemory(g_msgData, data, sizeof(*data));
        SetEvent(g_dataReadyEvent);
    }
}
```

**박스 1 TerminateProcess**

프로세스의 강제 종료는 후킹 프로그램에게는 재앙과도 같은 일이다. 작업 관리자에서 프로세스를 강제 종료 시키면 TerminateProcess 라는 API 가 호출된다. 이 API 가 호출되면 운영체제는 해당 프로세스를 강제로 메모리에서 내리고 자원을 반환한다. 하지만 이 과정에서 해당 프로그램에게는 어떠한 통지도 가지 않는다. 이 경우에 혼자 동작하는 프로그램의 경우에는 큰 문제가 없다. 할당된 메모리와 커널 오브젝트의 경우 운영체제에서 알아서 해제해 주기 때문이다. 하지만 여러 프로세스에 걸쳐서 동기화 해서 움직이는 후킹 프로그램의 경우 큰 문제가 발생할 수 있다. 왜냐하면 동기화에 사용된 오브젝트를 해당 프로그램에서 적법한 절차를 거치지 않고 파기 시켰기 때문에 다른 프로세스에서 해당 동기화 오브젝트에 접근할 수 없는 일이 발생하기 때문이다. 만약 해당 프로그램이 전체 프로세스를 후킹 하고 있었다면 최악의 경우에는 컴퓨터를 껐다 켜야 하는 상황이 발생할 수도 있다. 따라서 후킹 프로그램을 작성할 때에는 항상 TerminateProcess 가 발생하더라도 안정적으로 동작하도록 하는 데 많은 신경을 써야 한다.

## 메시지 관리

Windows 에는 수천 종류의 메시지가 있다. 이렇게 많은 메시지를 분류해서 관리하는 작업은 상당히 힘든 일이다. 실제로 imSpy 에는 모든 종류의 메시지가 들어 있진 않다. 100 여 가지의 메시지에 대한 정보만 추가되어 있다. 메시지 정보를 저장하는 구조체는 <리스트 5>에 나타나 있다. 메시지 정보를 저장하는 구조체와 해당 메시지가 포함될 카테고리를 저장할 구조체가 있다. 두 구조체를 사용해서 프로그램에서 처리할 메시지 목록을 <리스트 6>과 같이 구조체 배열을 사용해서 저장한다.

**리스트 5 메시지 정보 구조체**

```
// 메시지 정보 구조체
typedef struct _MSGDATA
{
   UINT    category;   // 카테고리
   UINT    id;        // 메시지 번호
   LPCTSTR name;      // 메시지 이름
} MSGDATA, *PMSGDATA;

// 카테고리 정보 구조체
typedef struct _MSGCATDATA
{
   UINT    category;   // 카테고리
   LPCTSTR name;      // 카테고리 명
} MSGCATDATA, *PMSGCATDATA;
```

**리스트 6 메시지 정의 부분**

```
const UINT MSGCT_KEYBOARD   = 1; // 키보드 메시지
const UINT MSGCT_MOUSE     = 2; // 마우스 메시지
const UINT MSGCT_CLIPBOARD  = 3; // 클립보드 메시지
const UINT MSGCT_DLG       = 4; // 다이알로그 메시지

// ... 중략

#define DEFMSG(category, id, name)  { category, id, name }
#define DEFMSG_KEY(id, name) DEFMSG(MSGCT_KEYBOARD, id, name)
#define DEFMSG_MOUSE(id, name) DEFMSG(MSGCT_MOUSE, id, name)
#define DEFMSG_CLIP(id, name) DEFMSG(MSGCT_CLIPBOARD, id, name)

// ... 중략

const MSGCATDATA CMsgData::m_msgCatData[] =
{
   { MSGCT_KEYBOARD, _T("키보드") },
   { MSGCT_MOUSE, _T("마우스") },
   { MSGCT_CLIPBOARD, _T("클립보드") },
```

```
// ... 중략

const MSGDATA CMsgData::m_msgData[] =
{
    DEFMSG_KEY(WM_KEYDOWN, _T("WM_KEYDOWN")),
    DEFMSG_KEY(WM_KEYUP, _T("WM_KEYUP")),
    DEFMSG_KEY(WM_CHAR, _T("WM_CHAR")),
    DEFMSG_KEY(WM_DEADCHAR, _T("WM_DEADCHAR")),
    DEFMSG_KEY(WM_SYSKEYDOWN, _T("WM_SYSKEYDOWN")),
    DEFMSG_KEY(WM_SYSKEYUP, _T("WM_SYSKEYUP")),
    DEFMSG_KEY(WM_SYSCHAR, _T("WM_SYSCHAR")),
    DEFMSG_KEY(WM_SYSDEADCHAR, _T("WM_SYSDEADCHAR")),
    DEFMSG_KEY(WM_UNICHAR, _T("WM_UNICHAR")),

    DEFMSG_MOUSE(WM_MOUSEMOVE, _T("WM_MOUSEMOVE")),
    DEFMSG_MOUSE(WM_LBUTTONDOWN, _T("WM_LBUTTONDOWN")),
    DEFMSG_MOUSE(WM_LBUTTONUP, _T("WM_LBUTTONUP")),
    DEFMSG_MOUSE(WM_LBUTTONDBLCLK, _T("WM_LBUTTONDBLCLK")),

// ... 중략
```

메시지를 실제 사용자가 알아보기 쉽게 문자열로 변환해 주는 작업은 <리스트 7>에 나타난 자료 구조를 통해서 수행된다. FDECODEMSG 는 메시지를 문자열로 변환해 주는 함수 포인터다. 각각의 메시지는 하나 이상의 변환 함수를 가진다. MSGDECODER 를 보면 변환 함수가 벡터로 구성된 것을 볼 수 있다. MsgIndexMap 은 메시지 번호와 거기에 따른 정적 배열의 인덱스 넘버와 변환 함수를 저장한다. 즉, 메시지 번호가 날라오면 단번에 인덱스 번호와, 변환 함수 목록을 알 수 있는 것이다. MsgIndexMap 은 프로그램이 로딩될 때 생성된다.

다음으로 중요한 자료 구조는 MsgSet 이다. 각각의 메시지 모니터링 윈도우는 서로 다른 메시지를 처리하게 된다. 각각의 윈도우는 자신이 처리해야하는 메시지를 MsgSet 에 저장해서 각자 담아둔다. 메시지가 발생하면 모든 윈도우로 해당 정보를 전송하고, 해당 윈도우는 자신의 MsgSet 에 메시지 번호가 존재하는 경우에만 해당 내용을 처리해서 추가한다.

**리스트 7 메시지 디코딩 정보를 저장할 구조체**

```
// 메시지 디코딩 함수
typedef BOOL (CALLBACK *FDECODEMSG)(LPTSTR buf, UINT size, IMSPYMSGDATA &data);

// 메시지 디코딩 함수 목록을 저장할 벡터
typedef std::vector<FDECODEMSG>                 DecodeFuncVec;
typedef std::vector<FDECODEMSG>::iterator         DecodeFuncVIt;
typedef std::vector<FDECODEMSG>::reverse_iterator   DecodeFuncVRIt;

// 메시지 디코더 정보
```

```
typedef struct _MSGDECODER
{
   UINT          no;
   DecodeFuncVec   fn;
} MSGDECODER, *PMSGDECODER;

typedef std::set<UINT>           MsgSet;
typedef std::set<UINT>::iterator   MsgSIt;

typedef std::map<UINT, MSGDECODER>           MsgIndexMap;
typedef std::map<UINT, MSGDECODER>::iterator   MsgIndexMIt;
```

위에서 소개한 모든 데이터를 저장하고 있는 클래스가 CMsgData 이다(<리스트 8> 참고). 메시지 목록을 정적 배열로 가지고 있다. 해당 메시지 목록에 대한 MsgIndexMap 은 생성자에서 초기화 한다. 가장 기본적인 메시지 디코딩 함수인 Default 를 포함하고 있다. Get 을 통해서 메시지 번호에 대한 메시지 정보 구조체에 접근할 수 있고, Decode 를 통해서 수신된 메시지에 대한 정보를 생성할 수 있다.

**리스트 8 CMsgData 클래스**

```
class CMsgData
{
private:
   static const MSGDATA    m_msgData[];
   static const MSGCATDATA m_msgCatData[];
   MsgIndexMap m_index;

protected:
   CMsgData();
   static BOOL CALLBACK Default(LPTSTR buf, UINT size, IMSPYMSGDATA &data);

public:
   const PMSGDATA Get(UINT msg);
   BOOL Enum(CMsgDataCallback &cb);
   BOOL Enum(CMsgCatDataCallback &cb);

   BOOL Decode(LPTSTR buf, UINT size, IMSPYMSGDATA &data);
   BOOL AddDecoder(UINT msg, FDECODEMSG fn);
   BOOL DeleteDecoder(UINT msg, FDECODEMSG fn);

   friend CMsgData &MsgData();
};
```

## IPC 쓰레드

<리스트 9>에 훅 DLL 과 통신하는 IPC 쓰레드의 코드가 나와있다. 쓰레드는 굉장히 간단하다. 시작과 동시에 bufferReady 이벤트를 설정해서 다른 DLL 들이 버퍼에 접근할 수 있도록 만든다. 그리고 dataReady 이벤트가 설정되면 공유 메모리의 내용을 지역 버퍼로

복사한 후 큐에 추가하고 메인 윈도우에 메시지가 발생했음을 알려준다. m_exitEvent 는 쓰레드 내부적으로 종료 체크를 하기 위해서 사용된다.

**리스트 9 훅 프로시와 통신하는 쓰레드 코드**

```
void CCommThread::Go()
{
   // ... 중략
   IMSPYMSGDATA msg;

   HANDLE events[2] = { m_exitEvent, dataReady };
   HANDLE lock[2] = { m_exitEvent, m_lockMutex };

   DWORD s = 0;
   for(;;)
   {
      SetEvent(bufferReady);

      s = WaitForMultipleObjects(2, events, FALSE, INFINITE);
      if(s == WAIT_OBJECT_0)
         break;

      if(s != WAIT_OBJECT_0 + 1)
         continue;

      CopyMemory(&msg, data, sizeof(IMSPYMSGDATA));

      s = WaitForMultipleObjects(2, lock, FALSE, INFINITE);
      if(s == WAIT_OBJECT_0)
         break;

      if(s == WAIT_OBJECT_0 + 1 || s == WAIT_ABANDONED)
      {
         CMutexLocker locker(m_lockMutex);
         m_msgs.push(msg);
         PostMessage(m_notifyWnd, WM_MSGFIRE, 0, 0);
      }
   }
}
```

위에서 전달한 WM_MSGFIRE 의 메시지 핸들러가 <리스트 10>에 나와있다. 메시지 핸들러에서는 쓰레드 큐에서 메시지 데이터를 하나 꺼내서 MDI 자식들을 순회하면서 메시지를 추가해 주는 일을 한다. m_lockMutex 는 쓰레드의 큐에 접근을 동기화 시키기 위한 뮤텍스다.

**리스트 10 메시지 처리 함수**

```
LRESULT CMainFrame::OnMsgFire(WPARAM w, LPARAM l)
{
```

```
    CWnd *wnd;
    CMsgListFrm *p;
    IMSPYMSGDATA data;

    for(;;)
    {
        // 버퍼에서 메시지 하나를 빼낸다.
        {
            CMutexLocker locker(m_commThread->m_lockMutex);
            if(m_commThread->m_msgs.empty())
                break;

            data = m_commThread->m_msgs.front();
            m_commThread->m_msgs.pop();

        }

        // 메시지 목록 윈도우에 메시지를 추가한다.
        for(wnd = m_mdiClient.GetWindow(GW_CHILD);
            wnd != NULL;
            wnd = wnd->GetWindow(GW_HWNDNEXT))
        {
            if(wnd->IsKindOf(RUNTIME_CLASS(CMsgListFrm)))
            {
                p = (CMsgListFrm *) wnd;
                p->AddMessge(data);
            }
        }
    }

    return 0;
}
```

실제로 메시지를 리스트에 추가하는 부분은 <리스트 11>에 나타나있다. m_hookEnabled 는 현재 훅이 활성화 되어 있는지를 나타난다. 로그 중지 메뉴를 선택하면 m_hookEnabled 가 FALSE 로 설정된다. m_watchWnd 는 후킹 대상 윈도우를 나타낸다. m_watchMsgs 는 모니터링 중인 메시지 목록을 나타낸다.

**리스트 11 리스트에 메시지를 추가하는 부분**

```
void CMsgListFrm::AddMessge(IMSPYMSGDATA &data)
{
    if( m_hookEnabled &&
        data.hwnd == m_watchWnd &&
        m_watchMsgs.find(data.message) != m_watchMsgs.end())
    {
        LPCTSTR type[] = { _T(""), _T("P"), _T("S"), _T("R") };
        CString buf;
        int cnt = m_lvcMsgs.GetItemCount();
```

```
        buf.Format(_T("%04d"), cnt+1);
        m_lvcMsgs.InsertItem(cnt, buf);

        m_lvcMsgs.SetItemText(cnt, 1, type[data.type]);

        buf.Format(_T("%08X"), data.hwnd);
        m_lvcMsgs.SetItemText(cnt, 2, buf);

        // ... 중략
    }
}
```

## 윈도우 찾기

많은 사람들에게 Spy++ 하면 가장 먼저 떠오르는 기능은 아마도 마우스로 커서를 드래그 하면서 윈도우를 찾는 기능일 것이다. 이 기능의 경우 눈에 보이는 윈도우를 바로 찾아준다는 점에서 굉장히 유용한 기능이다. 이 기능은 imSpy 에서는 메시지 옵션을 설정하는 대화 상자에 구현되어 있다. <화면 2>에서 창 찾기 도구에 나타난 아이콘을 드래그 하면 Spy++과 같이 창이 찾아 지는 것을 확인할 수 있다.

마우스 포인터 위의 윈도우를 찾는 함수로 WindowFromPoint, ChildWindowFromPoint, RealChildWindowFromPoint 등의 함수가 있다. 하지만 ChildWindowFromPoint 의 경우 그룹 상자 위에 놓여진 Static 컨트롤 등을 정확하게 찾아내지 못한다는 단점이 있고, RealChildWindowFromPoint 는 그러한 문제가 해결되었으나, 95 에서는 지원하지 않고 마우스 위치에 있는 가장 작은 자식 윈도우를 정확하게 찾아내진 못한다는 단점이 있다. <리스트 12>에는 이러한 문제를 해결한 코드가 나와있다.

**리스트 12 마우스 포인터 위의 윈도우를 찾는 코드**

```
HWND FindSmallestChildWindowFromPoint(HWND hwnd, CPoint &pt)
{
    HWND child = GetWindow(hwnd, GW_CHILD);
    HWND result = NULL;

    if(child)
    {
        HWND h;
        CRect rc;
        CRect tmp;
        BOOL isSmall;

        // 자식을 조사한다.
        for(child = GetWindow(hwnd, GW_CHILD);
            child != NULL;
            child = GetWindow(child, GW_HWNDNEXT))
        {
```

```
            // 자기보다 더 깊은 곳의 자식을 먼저 찾는다.
            h = FindSmallestChildWindowFromPoint(child, pt);

            // 찾았으면 리턴 한다.
            if(h)
                return h;

            GetWindowRect(child, &tmp);

            // 현재 윈도우 영역에 마우스 포인터가 포함되고
            // 처음 이거나 이전에 찾은 윈도우 보다 작은 경우
            isSmall = tmp.Width() <= rc.Width() && tmp.Height() <= rc.Height();
            if(tmp.PtInRect(pt) && (!result || isSmall))
            {
                result = child;
                rc = tmp;
            }
        }
    }

    return result;
}

HWND FindSmallestWindowFromPoint(CPoint &pt)
{
    HWND hwnd = WindowFromPoint(pt);
    if(hwnd)
    {
        HWND child = FindSmallestChildWindowFromPoint(hwnd, pt);
        if(child)
            return child;
    }

    return hwnd;
}
```

기본 적인 아이디어는 WindowFromPoint 를 사용해서 큰 윈도우를 찾은 다음, 그 아래 자식들을 기준으로 깊이 우선 탐색(depth-first search)를 하는 것이다. 이렇게 할 경우 가장 깊은 자식에게 우선 순위가 간다. 같은 깊이의 자식들 중에는 마우스 포인터를 포함하는 가장 작은 영역을 가진 윈도우를 찾도록 되어 있다.

그럼 이제 마우스 트래킹을 하면서 윈도우를 찾는 방법을 알아 보자. 마우스 트래킹을 하기 위해서는 네 개의 메시지 핸들러를 작성해 주어야 한다. WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP, WM_CAPTURECHANGED 가 그것이다. WM_LBUTTONDOWN 이 발생하면 SetCapture 를 통해서 마우스를 캡쳐하고 트래킹을 시작하면 된다(<리스트 13> 참고). WM_MOUSEMOVE 에서는 트래킹이 시작되었다면 마우스 아래에 있는 윈도우를 찾아서 표시하는 일을 한다(<리스트 15> 참고).

WM_LBUTTONUP 과 WM_CAPTURECHANGED 에서는 ReleaseCapture 를 통해서 마우스 캡쳐를 해제하고 트래킹을 중지하면 된다(<리스트 14> 참고).

**리스트 13 마우스 트래킹 시작 함수**

```
void CMsgOptionDlg::StartFindWindow()
{
    m_finding = TRUE; // 트래킹 중인지를 나타내는 멤버 변수
    m_watchWnd = 0; // 최종적으로 발견한 윈도우 핸들

    ChangeFinderIcon(); // 파인더 아이콘 변경
    m_prevCursor = SetCursor(m_finderCursor); // 커서 변경
    //ShowWindow(SW_HIDE);
    SetCapture();
}
```

**리스트 14 마우스 트래킹 중지 함수**

```
void CMsgOptionDlg::StopFindWindow()
{
    m_finding = FALSE;

    ChangeFinderIcon();
    SetCursor(m_prevCursor);

    //ShowWindow(SW_SHOW);
    ReleaseCapture();
    DrawBorder();
}
```

**리스트 15 마우스 포인터에 있는 윈도우를 찾아서 표시하는 함수**

```
void CMsgOptionDlg::Find()
{
    CPoint pt;

    GetCursorPos(&pt);
    HWND hwnd = FindSmallestWindowFromPoint(pt);
    if(hwnd && hwnd != m_watchWnd)
    {
        DrawBorder(); // 이전에 그려진 경계를 지운다
        m_watchWnd = hwnd;
        UpdateWindowInfo();
        DrawBorder(); // 새롭게 찾은 윈도우의 경계를 그린다.
    }
}
```

## 도전 과제

필자가 처음에 imSpy 를 제작할 때에 윈도우 목록 필터링 기능과 메시지 디코딩 기능을 넣으려 했었으나 시간 관계로 기능을 추가하지 못했다. 이번 달 도전 과제는 이 기능을 추가하는 것으로 해보자.

윈도우 목록 필터링 기능은 많은 윈도우 목록 중에 자신이 원하는 특정 윈도우만 보여주는 기능을 하는 것이다. 예를 들면 화면에 보이는 창만 출력한다거나 아니면 특정 프로세스의 창만 출력하는 기능 등을 생각해 볼 수 있다. 이 경우 한 가지 조심해야 할 것은 자신은 화면에 나타나지 않는 창이라고 무조건 제거해서는 안 된다는 점이다. 자식이 화면에 표시되는 창이라면 그 부모가 화면에 나타나지 않아도 표시해 주어야지 정확한 계층구조를 확인할 수 있다는 점이다.

메시지 디코딩 기능은 메시지 디코딩 함수를 추가해서 설명 부분에 디코딩한 결과를 출력하는 것이다. Spy++은 메시지가 출력될 때에 WPARAM, LPARAM 이런 식으로 표시하지 않고 각 메시지에 맞게 항목을 보기 좋게 출력해 주는 것을 볼 수 있다. 이러한 기능을 하도록 구현해 보자. 설명 부분에 추가하면 될 것 이다.

## 진짜 Spy++을 제작하고 싶은 분들을 위한 팁

이번 달의 imSpy 샘플은 진짜 Spy++의 클론을 구현하기 위한 좋은 파일럿 프로그램이 될 수 있다. 하지만 이번 달의 우리가 제작한 imSpy 는 지역 훅을 사용했기 때문에 기능 구현에 한계를 가지고 있다. Spy++의 전체 기능을 작성해야 한다면 전역 훅을 사용하는 것이 편리하다.

지난 시간 까지 우리는 전역 훅의 경우 시스템 성능이 느려질 수 있고, 위험하기 때문에 사용하지 않는 것이 좋다고 했었다. 이 말은 실제로 맞는 말이다. 하지만 항상 트레이드 오프는 있는 법이다. 느려지고 위험하지만 그것보다 얻는 이득이 크다면 설치해서 사용해 볼 수 있다는 점이다. 전역 훅을 사용할 경우엔 어떤 기능을 구현하기 쉬울까?

첫째는 윈도우 속성 구하는 기능이다. 윈도우 속성중 일부 항목은 해당 윈도우를 생성한 프로세스의 컨텍스트에서만 구할 수 있다. 따라서 모든 윈도우의 속성을 구하기 위해서는 모든 윈도우의 프로세스 컨텍스트에서 실행이 되어야 하고 이는 곧 전역적으로 훅이 설치되어야 함을 의미한다. 실제로 Spy++은 전역 훅을 설치하고 해당 윈도우로 WM_NULL 메시지가 포스팅 되면 해당 윈도우의 속성을 구해서 공유 섹션 부분에 기록한다.

두 번째 기능은 여러 윈도우를 동시에 모니터링 하는 기능이다. Spy++의 메시지 옵션 부분을 보면 부모 윈도우, 소유자 윈도우, 같은 프로세스 윈도우등을 같이 모니터링 할 수 있는 기능이 있다. 이러한 기능의 경우 우리와 같이 지역 훅을 사용할 때에는 매우 구현이 까다롭게 된다. 왜냐하면 이러한 윈도우들이 같은 쓰레드에 존재하라는 법이 없기 때문이다.

물론 이 두 가지를 모두 전역 훅을 설치하지 않고 할 수 있다. 하지만 그렇게 할 경우엔 배보다 배꼽이 더 커지는 격이 될 수 있음을 기억하자.

## 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++과 같이 윈도우를 찾아 내는 방법
http://www.codeproject.com/dialog/windowfinder.asp