개발자를 위한 윈도우 후킹 테크닉

WH_DEBUG 훅을 이용한 훅 탐지 방법

지금까지 우리는 다양한 종류의 훅을 사용해서 다른 프로그램으로 전달되는 메시지를 가로채는 방법을 법을 배웠다. 이번 시간에는 WH_DEBUG 훅을 사용해서 훅 프로시저를 탐지하는 방법에 대해서 다룬다.

목차

필자 소개

신영진 pop@jiniya.net, http://www.jiniya.net

"너는 네 세상 어디에 있느냐? 너에게 주어진 몇몇 해가 지나고 몇몇 날이 지났는데, 그래 너는 네 세상 어디쯤에 와 있느냐?" 2006 년도 이제 몇 달 남지 않았다. 후회가 되지 않는 한 해를 보내기란 버그 없는 프로그램을 만드는 것만큼 힘든 일인 것 같다.

연재 가이드

운영체제: 윈도우 98/2000/XP
개발도구: 마이크로소프트 비주얼 스튜디오 2003
기초지식: C/C++, Win32 프로그래밍
응용분야: 훅 탐지 프로그램, 보안 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

언젠가 뉴스그룹에 누군가가 프로그램을 해킹하는 것을 막는 방법이 있냐고 질문한 적이 있었다. 여러분이 어떻게 생각하는가?

폰 노이만식 아키텍처(프로그램과 데이터가 같은 메모리에 저장되는 방식) 를 사용하는 유사 튜링 머신(읽기, 쓰기, 실행 형태로 동작) 형태를 취하고 있는 현재의 컴퓨터 시스템에서 완벽한 보안이란 있을 수 없다고 필자는 굳게 믿고 있다. 왜냐하면 모든 것이 결국은 메모리에 있어야 하고, CPU 에 의해서 해석될 수 있는 코드 집합이기 때문이다. 압축을 하고 암호화를 해도 그것을 실행하려면 압축을 풀고 암호를 해독해야 한다. 데이터도 마찬가지다.

그때 그 토론도 그런 식으로 결론이 났었다. 하지만 완벽할 수 없다고 해서 보안에 대한 노력이 무의미한 것은 아니다. 해커도 사람이기 때문이다. 사람의 인내심과 이해력의 한계는 분명히 있다. 가장 최선의 보안은 해커를 가장 귀찮게 만드는 것이다.

Intro

지금까지 우리는 다양한 훅을 통해서 다른 프로그램을 훔쳐보는 방법을 배웠다. 늘 그렇듯이 훔쳐보는 것은 재미난 일이지만 자신이 공들여 만든 프로그램이 다른 프로그램에게 감시 당하는 것은 썩 기분 좋은 일은 아니다. 키로거에 의해서 패스워드를 도난 당하거나 악성 프로그램에 의해서 데이터가 파손되는 일들이 자신의 프로그램에서 벌어지지 않으리란 법은 없는 것이다.

이번 시간에는 위와 같은 질문에 대한 해답을 찾아 본다. WH_DEBUG 훅을 사용해서 다른 훅들을 감시하고 그것들의 실행을 중지시키는 방법을 배운다. 이 과정에서 훅 디텍터를 제작해 보고 toolhelp 라이브러리를 사용해서 프로세스, 스레드, 모듈 정보를 구하는 방법에 대해서 다룬다.

WH_DEBUG 훅

WH_DEBUG 훅은 시스템에 설치된 다른 훅을 디버깅 하기 위해서 만들어진 훅이다. 하지만 이 훅을 디버깅을 위해서 사용하는 경우는 정말 드물다. 대부분의 프로그램은 다른 후킹 프로그램을 탐지하기 위해서 WH_DEBUG 훅을 사용한다. WH_DEBUG 훅을 사용하면 WH_KEYBOARD_LL, WH_MOUSE_LL, WH_DEBUG 훅을 제외한 모든 훅을 검출할 수 있다.

```
LRESULT CALLBACK DebugProc(int nCode, WPARAM wParam, LPARAM lParam);
```

code – [입력] HC_ACTION 인 경우엔 훅 프로시저를 수행하면 되고, 0 보다 작은 경우에는 훅 프로시저를 수행하지 않고 CallNextHookEx 를 호출한 후 바로 끝내야 한다.

wParam – [입력] 탐지한 훅 프로시저의 종류를 나타낸다. WH_KEYBOARD_LL, WH_MOUSE_LL, WH_DEBUG 훅은 감지 되지 않는다.

lParam – [입력] DEBUGHOOKINFO 구조체 포인터를 가리킨다. DEBUGHOOKINFO 구조체의 원형은 다음과 같다. 필드별 의미는 <표 1> 에 나와있다.

```
typedef struct {
    DWORD idThread;
    DWORD idThreadInstaller;
    LPARAM lParam;
    WPARAM wParam;
    int code;
} DEBUGHOOKINFO, *PDEBUGHOOKINFO;
```

표 1 DEBUGHOOKINFO 구조체 필드별 의미

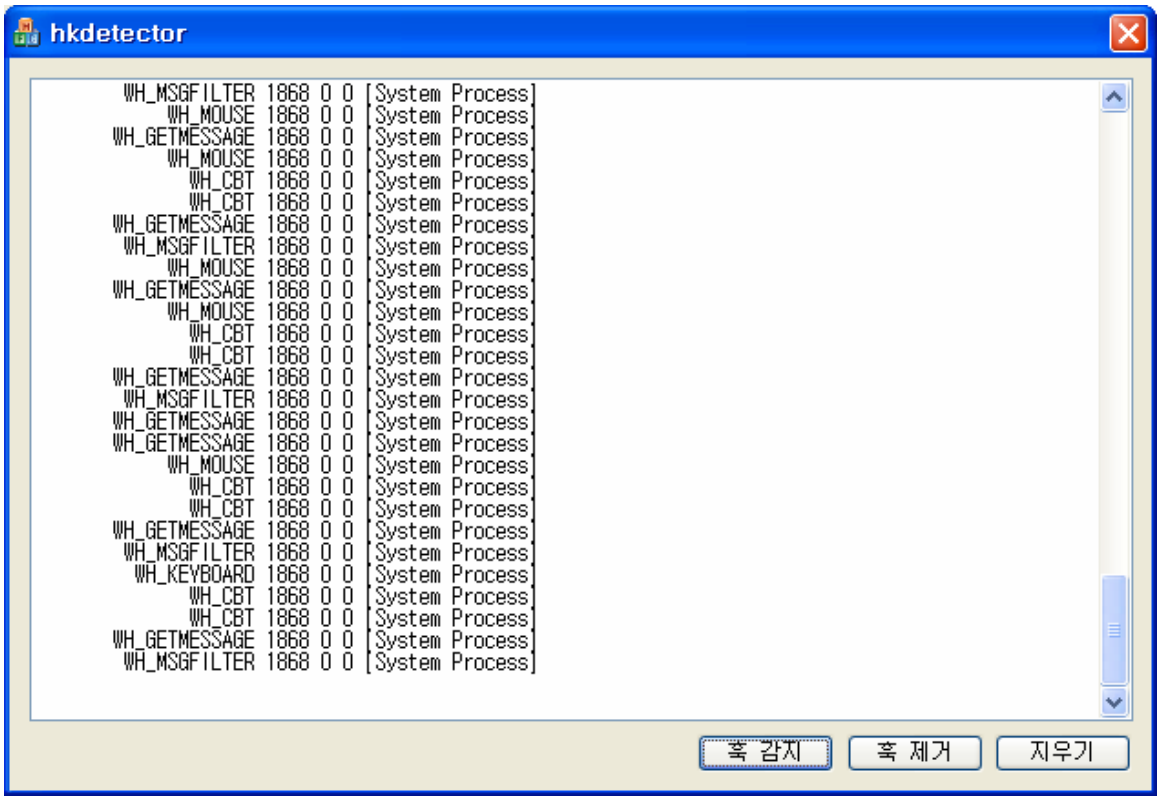
필드명	의미
idThread	훅 프로시저가 수행되고 있는 스레드 ID
idThreadInstaller	훅을 설치한 스레드 ID (문서의 내용과 달리 NT 계열의 운영체제에서는 이 값이 항상 0 으로 넘어온다.)
lParam	탐지된 훅 프로시저로 전달될 lParam
wParam	탐지된 훅 프로시저로 전달될 wParam

code	탐지된 훅 프로시저로 전달될 code

리턴 값: WH_DEBUG 훅에 검출된 훅 함수를 수행하지 않으려면 0 이 아닌 값을 리턴 해야 한다. 그렇지 않은 경우라면 CallNextHookEx 의 리턴 값을 그대로 사용하는 것이 좋다.

훅 디텍터

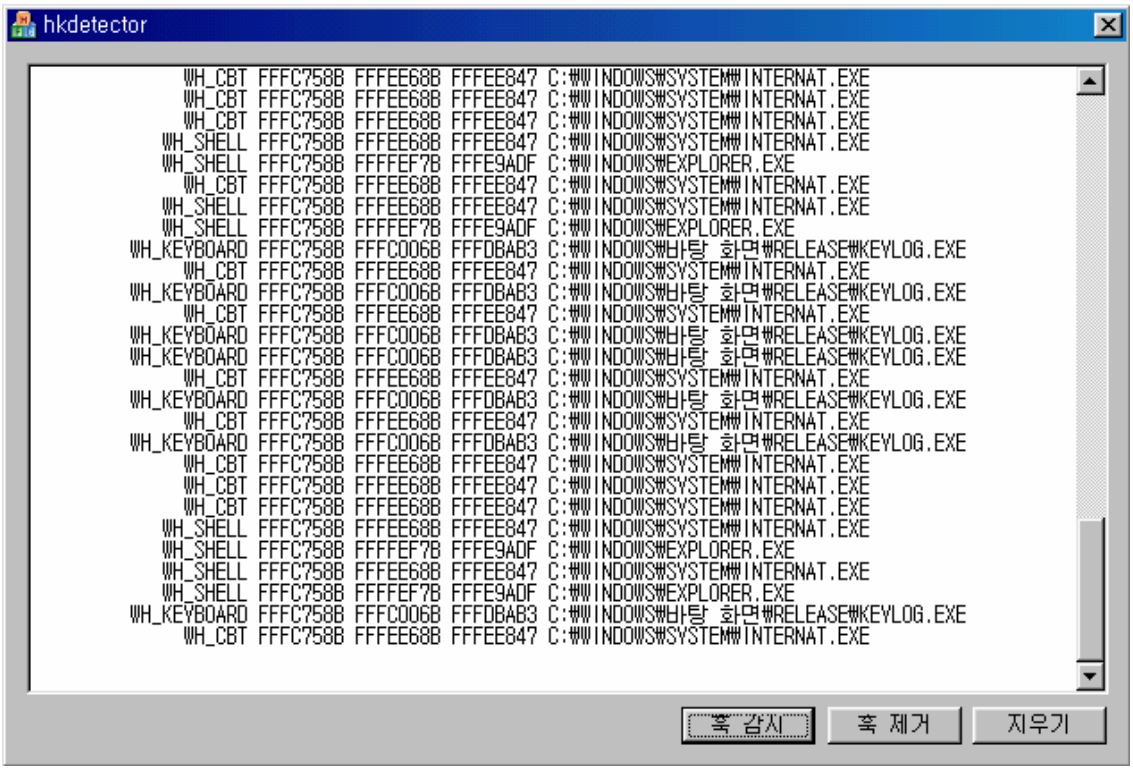
이제 WH_DEBUG 훅을 사용해서 훅 디텍터를 제작해 보자. 이 프로그램은 다른 프로그램의 훅 코드가 수행됨을 사용자에게 알려주는 역할을 한다. 프로그램의 동작 모습은 <화면 1>에 나와있다. 각 줄은 훅 종류, 훅이 수행되는 스레드 ID, 훅을 설치한 스레드 ID, 훅을 설치한 프로세스, 훅을 설치한 프로세스 명으로 구성된다.



화면 1 Windows XP 에서 훅 탐지 프로그램을 실행한 경우

결과 화면을 보면 알 수 있지만 한 가지 이상한 현상이 존재한다. 다름 아닌 훅을 설치한 스레드의 ID 가 모두 0 이라는 점이다. NT 이상의 운영체제의 경우에는 WH_DEBUG 훅으로 넘어오는 DEBUGHOOKINFO 구조체의 idThreadInstaller 항목이 문서와 달리 전부 0 으로 넘어왔다. 따라서 훅 프로시저가 호출된다는 건 알 수 있어도 누가 설치한 훅 프로시저인지 검출하는 것은 불가능하다. 반면에 9x 계열의 운영체제에서는 DEBUGHOOKINFO 구조체의

모든 값이 정상적으로 넘어오기 때문에 훅을 설치한 프로그램을 추적하는 것이 가능하다. <화면 2>는 Windows 98 SE 컴퓨터에서 프로그램을 동작시킨 모습을 보여주고 있다.



```
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFFEF7B FFFE9ADF C:₩WINDOWS₩EXPLORER.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFFEF7B FFFE9ADF C:₩WINDOWS₩EXPLORER.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFFEF7B FFFE9ADF C:₩WINDOWS₩EXPLORER.EXE
   WH_SHELL FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
   WH_SHELL FFFC758B FFFFEF7B FFFE9ADF C:₩WINDOWS₩EXPLORER.EXE
WH_KEYBOARD FFFC758B FFFC006B FFFDBAB3 C:₩WINDOWS₩바탕 화면₩RELEASE₩KEYLOG.EXE
     WH_CBT FFFC758B FFFEE68B FFFEE847 C:₩WINDOWS₩SYSTEM₩INTERNAT.EXE
```

화면 2 Windows 98 SE 에서 훅 탐지 프로그램을 실행한 경우

훅 프로시저는 정말 간단하다. <리스트 1>에 코드가 나와있다. 훅 디텍터의 경우 자신을 후킹하는 외부 프로그램을 검출하는 것이 목적이기 때문에 WH_DEBUG 훅의 범위를 자신의 스레드로 제한해서 설치하였다. 따라서 훅 프로시저가 실행 프로그램과 동일한 컨텍스트에서 호출되기 때문에 포인터를 메시지로 전달해도 된다.

리스트 1 WH_DEBUG 훅 프로시저

```
LRESULT CALLBACK DebugProc(int code, WPARAM w, LPARAM l)
{
    if(code == HC_ACTION)
        SendMessage(AfxGetMainWnd()->GetSafeHwnd(), WM_HOOKDETECTED, w, l);

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

훅 프로시저에서 SendMessage 로 날린 메시지에 대한 핸들러는 <리스트 2>에 나와있다. 훅 종류를 조사해서 정보를 출력하는 단순한 역할을 한다. 자신이 설치한 훅을 출력하는

것을 방지하기 위해서 실행 하고 있는 스레드 ID 와 설치한 스레드 ID 가 다른 경우에만 출력하도록 했다. 또 한 가지 주의해야 할 점은 ID 를 출력할 때에 NT 계열의 경우 단순히 정수로 표시하지만, 9x 계열의 운영체제는 16 진수로 출력해 주어야 한다는 것이다.

리스트 2 WM_HOOKDETECTED 메시지 핸들러

```
LRESULT ChkdetectorDlg::OnHookDetected(WPARAM w, LPARAM l)
{
   // 훅 종류에 대한 이름
   HOOKNAME hookNames[] = {
      {WH_MSGFILTER, "WH_MSGFILTER"},
      {WH_JOURNALPLAYBACK, "WH_JOURNALPLAYBACK"},
      {WH_JOURNALRECORD, "WH_JOURNALRECORD"},
      {WH_KEYBOARD, "WH_KEYBOARD"},
      {WH_MOUSE, "WH_MOUSE"},
      {WH_KEYBOARD_LL, "WH_KEYBOARD_LL"},
      {WH_MOUSE_LL, "WH_MOUSE_LL"},
      {WH_DEBUG, "WH_DEBUG"},
      {WH_SHELL, "WH_SHELL"},
      {WH_FOREGROUNDIDLE, "WH_FOREGROUNDIDLE"},
      {WH_GETMESSAGE, "WH_GETMESSAGE"},
      {WH_CALLWNDPROC, "WH_CALLWNDPROC"},
      {WH_CALLWNDPROCRET, "WH_CALLWNDPROCRET"},
      {WH_HARDWARE, "WH_HARDWARE"},
      {WH_CBT, "WH_CBT"},
      {WH_SYSMSGFILTER, "WH_SYSMSGFILTER"}
   };

   CString result;
   TCHAR processName[MAX_PATH] = {0,};
   DWORD pid = 0;

   // 훅 종류 검색
   for(int i=0; i<sizeof(hookNames)/sizeof(HOOKNAME); ++i)
   {
      if(w == hookNames[i].id)
      {
         DEBUGHOOKINFO *info = (DEBUGHOOKINFO *) l;
         // 자신이 설치한 훅인지 판별
         if(info->idThreadInstaller != info->idThread)
         {
            // 훅을 설치한 프로세스 이름 구하기
            pid = GetProcessIdFromThreadId(info->idThreadInstaller);
            GetProcessNameFromPID(processName, sizeof(processName), pid);

            // 운영체제가 9x 계열인 경우
            if(GetVersion() & 0x80000000)
            {
               result.Format(  "%20s %X %X %X %s\r\n",
                           hookNames[i].name,
                           info->idThread,
```

```
                            info->idThreadInstaller,
                            pid,
                            processName );
            }
            else
            {
                result.Format(  "%20s %d %d %d %s\r\n",
                            hookNames[i].name,
                            info->idThread,
                            info->idThreadInstaller,
                            pid,
                            processName );
            }
        }
    }
}

    // 에디터의 마지막에 탐지된 훅 프로시저 기록
int len = m_edtResult.GetWindowTextLength();
m_edtResult.SetSel(len, len);
    m_edtResult.ReplaceSel(result);
    return 0;
}
```

설치한 프로세스 ID 구하기

DEBUGHOOKINFO 구조체로부터 우리가 알 수 있는 것은 훅이 설치한 스레드와 실행되는 스레드 ID 가 전부다. 사실 이러한 정보는 사용자에게 어떠한 도움을 주지도 못하는 정보다. 최종 사용자에게는 프로세스와 관련된 정보를 제공하는 것이 일반적이다. 그래야 사용자가 해당 프로세스를 종료하거나 파일을 제거하는 등의 작업을 할 수 있기 때문이다.

<표 2>에는 자주 사용하는 프로세스와 스레드 관련 함수들이 열거되어 있다. 살펴보면 알 수 있겠지만 각각의 ID 와 핸들은 손쉽게 변환할 수 있지만 프로세스 ID 로부터 스레드 ID 를 구하거나, 스레드 ID 에 대한 부모 프로세스 ID 를 구하는 작업을 하는 것은 힘들다. 후자의 작업에 사용할 수 있는 API 로 GetProcessIdOfThreadId 라는 함수가 있지만, 이 함수의 경우 Windows Server 2003 이상이 되어야만 사용할 수 있기 때문에 아직 사용하기에는 이른 함수라고 할 수 있다.

표 2 프로세스/스레드 함수들

함수명	기능
OpenThread	스레드 ID 에 대한 핸들을 반환한다.
GetThreadId	스레드 핸들에 대한 ID 를 반환한다.
TerminateThread	스레드를 강제로 종료 시킨다.

GetCurrentThread	현재 스레드 핸들을 반환한다.
GetCurrentThreadId	현재 스레드 ID 를 반환한다.
OpenProcess	프로세스 ID 에 대한 핸들을 반환한다.
GetProcessId	프로세스 핸들에 대한 ID 를 반환한다.
TerminateProcess	프로세스를 강제로 종료 시킨다.
GetCurrentProcess	현재 프로세스 핸들을 반환한다.
GetCurrentProcessId	현재 프로세스 ID 를 반환한다.
GetProcessIdOfThread	스레드 핸들에 대한 부모 프로세스 ID 를 반환한다.

API 가 없다면 직접 만드는 수 밖에는 없다. Toolhelp 라이브러리를 사용하면 시스템에서 실행중인 스레드/프로세스/모듈을 손쉽게 열거할 수 있다. Toolhelp 라이브러리에서 스레드 정보를 알려주는 구조체인 THREADENTRY32 의 th32OwnerProcessID 를 참고하면 부모 프로세스의 ID 를 구할 수 있다. <리스트 3>에 toolhelp 를 사용해서 스레드 ID 에 해당하는 프로세스 ID 를 구하는 Emulate_GetProcessIdOfThreadId 함수가 나와있다.

Windows 2003 이상의 시스템에서는 굳이 우리가 직접 작성한 Emulate_GetProcessIdOfThreadId 를 수행하는 것보다 실제 GetProcessIdOfThread 를 사용하는 것이 좋다. 시스템 API 의 경우 이미 검증된 것이고 성능이 우리가 작성한 것보다 좋을 수 있기 때문이다. 아마도 Windows 2003 이상에서 구현된 GetProcessIdOfThread 함수는 단순한 포인터 참조로 구현되었을 가능성이 높다. 따라서 시스템에 API 가 없는 경우에만 Emulate_GetProcessIdOfThreadId 를 사용하는 것이 바람직하다.

<리스트 3>을 살펴보면 그런 경우를 처리하는 일반적인 방법을 알 수 있다. GetProcessIdOfThread 는 스레드 핸들을 인자로 받는다. 스레드 ID 를 바로 프로세스 ID 로 변환할 수 없기 때문에 소스가 다소 복잡하다(<표 3> 참고).

표 3 사용된 심벌 이름및 역할

이름	역할
Real_GetProcessIdOfThread	2003 이상에 존재하는 GetProcessIdOfThread 함수 포인터를 저장할 변수
GetProcessIdOfThreadId	프로그램에서 사용할 함수를 저장할 함수 포인터. Emulate_GetProcessIdOfThreadId 와 Real_GetProcessIdOfThread 중 하나를 가짐.
Emulate_GetProcessIdOfThreadId	toolhelp 라이브러리를 사용해서 스레드 ID 를 프로세스 ID 로 변환하는 함수

Real_GetProcessIdOfThreadId	Real_GetProcessIdOfThread 함수를 사용해서 스레드 ID 를 프로세스 ID 로 변환하는 함수
Probe_GetProcssIdOfThreadId	Emulate_GetProcessIdOfThreadId 와 Real_GetProcessIdOfThreadId 중에 어떤 함수를 사용할지 결정하는 함수

코드의 동작 원리는 간단하다. GetProcessIdOfThread 를 실행될 때 찾아보고 있는 경우엔 그 함수를 사용하는 버전의 함수를 사용하고, 없는 경우에는 에뮬레이팅 하는 버전의 함수를 사용하는 것이다. 또한 Probe_GetProcessIdOfThreadId 는 로드될 때 한번만 호출되기 때문에 호출 빈도에 따른 부하도 없다.

리스트 3 특정 스레드 ID 의 부모 프로세스 ID 를 구하는 함수

```
#include <tlhelp32.h>

#ifdef __cplusplus
extern "C" {
#endif

#ifdef WANT_GETPROCESSIDOFTHREAD_WRAPPER

// GetProcessIdOfThreadId 함수 원형
typedef DWORD (WINAPI *FGetProcessIdOfThreadId)(DWORD);

// GetProcessIdOfThread 함수 원형
typedef DWORD (WINAPI *FGetProcessIdOfThread)(HANDLE);

#ifdef COMPILE_API_STUB

// 2003 이상에 존재하는 GetProcessIdOfThread 함수 포인터를 저장할 변수
FGetProcessIdOfThread Real_GetProcessIdOfThread = NULL;

// toolhelp 를 사용해 구현한 함수
// 스레드 ID 를 프로세스 ID 로 변환함
DWORD CALLBACK
Emulate_GetProcessIdOfThreadId(DWORD tid)
{
    DWORD ret = 0;

    // 스레드를 열거할 toolhelp 스냅샷 생성
    HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, 0);
    if(snap == INVALID_HANDLE_VALUE)
        return ret;

    THREADENTRY32 te;
    ZeroMemory(&te, sizeof(te));
    te.dwSize = sizeof(te);
```

```
    // 첫 번째 스레드 정보 구함
    if(!Thread32First(snap, &te))
        goto $cleanup;

    // 스냅샷의 끝까지 반복하면서 스레드 ID 검색
    do
    {
        if(te.th32ThreadID == tid)
        {
            // 찾고자 하는 스레드 ID 를 발견한 경우 PID 저장
            ret = te.th32OwnerProcessID;
            break;
        }

    } while(Thread32Next(snap, &te));

$cleanup:
    // 생성한 스냅샷을 해제하고 리턴
    CloseHandle(snap);
    return ret;
}

// GetProcessIdOfThread API 를 사용해 구현한 함수
// 스레드 ID 를 프로세스 ID 로 변환함
DWORD CALLBACK
Real_GetProcessIdOfThreadId(DWORD tid)
{
    HANDLE thread = OpenThread(THREAD_QUERY_INFORMATION, FALSE, tid);
    if(thread == NULL)
        return 0;

    DWORD pid = Real_GetProcessIdOfThread(thread);
    CloseHandle(thread);

    return pid;
}

// 어떤 버전의 함수를 사용할지 결정하는 함수
// 사용할 수 있는 적절한 버전의 함수 포인터를 리턴 한다.
FGetProcessIdOfThreadId
Probe_GetProcessIdOfThread()
{
    HINSTANCE inst;

    inst = GetModuleHandle(TEXT("KERNEL32"));
    if(!inst)
        return Emulate_GetProcessIdOfThreadId;

    // GetProcessIdOfThread 함수 포인터 저장
    Real_GetProcessIdOfThread =
        (FGetProcessIdOfThread) GetProcAddress(inst, "GetProcessIdOfThread");

    if(!Real_GetProcessIdOfThread)
```

```
        return Emulate_GetProcessIdOfThreadId;

    return Real_GetProcessIdOfThreadId;
}

// 사용할 함수 포인터를 저장할 변수
FGetProcessIdOfThreadId
GetProcessIdOfThreadId = Probe_GetProcessIdOfThread();

#endif
#endif

#ifdef __cplusplus
}
#endif
```

프로세스 이름 구하기

프로세스 ID 를 구했다고 모든 작업이 마무리된 것은 아니다. 9x 계열의 운영체제에서는 프로세스 ID 에 대한 이미지 이름을 보여주는 기본 유틸리티가 없기 때문에 사용자 편의를 위해서 프로세스 이름을 보여주는 것이 바람직하다. toolhelp 라이브러리를 사용하면 프로세스 ID 에 대한 이름 또한 쉽게 구할 수 있다. <리스트 4>에 그 코드가 나와있다.

GetProcessNameFromPID 함수는 인자로 넘어온 pid 에 해당하는 프로세스 이름을 buf 에 저장한다. size 에는 buf 의 크기를 넘겨주면 된다. pid 를 찾은 경우엔 TRUE 를 찾지 못한 경우엔 FALSE 를 반환 한다.

리스트 4 특정 프로세스 ID 에 대한 이미지 이름 구하기

```
BOOL GetProcessNameFromPID(LPTSTR buf, UINT size, DWORD pid)
{
    DWORD ret = FALSE;

    // 프로세스를 열거하기 위한 toolhelp 스냅샷 생성
    HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if(snap == INVALID_HANDLE_VALUE)
        return ret;

    PROCESSENTRY32 pe;
    ZeroMemory(&pe, sizeof(pe));
    pe.dwSize = sizeof(pe);

    // 첫 번째 프로세스 정보 구하기
    if(!Process32First(snap, &pe))
        goto $cleanup;

    // 스냅샷의 마지막까지 반복
    do
```

```
    {
        // 프로세스 ID 를 발견한 경우 프로세스 이름 복사
        if(pe.th32ProcessID == pid)
        {
            StringCbCopy(buf, size, pe.szExeFile);
            ret = TRUE;
            break;
        }

    } while(Process32Next(snap, &pe));

// 스냅샷 핸들 해제 및 결과 리턴
$cleanup:
    CloseHandle(snap);
    return ret;
}
```

프로세스에 로드된 모듈 열거하기

WH_DEBUG 훅의 가장 큰 단점은 NT 계열에서 idThreadInstaller 값이 제대로 구해지지 않는다는 점이다. 사실 NT 계열에서는 무용지물이다. 그렇다면 NT 계열에서 다른 프로그램이 설치한 훅의 존재 유무를 판단할 수 없을까? 물론 할 수 있다.

다른 프로세스를 후킹하기 위해서는 DLL 에 훅 프로시저를 두어야 했다. 이것은 결국 DLL 이 대상 프로세스 공간으로 로드 된다는 말이다. 따라서 우리는 굳이 훅 프로시저를 찾을 필요 없이 불필요한 모듈이 있는지를 찾으면 된다. toolhelp 라이브러리의 모듈 열거 기능을 사용하면 손쉽게 구현할 수 있다.

<리스트 5>에 toolhelp 라이브러리를 사용해서 자신의 프로세스에 로드된 모듈의 이름과 베이스 주소를 출력하는 프로그램이 나와있다. <화면 3>은 이 프로그램을 실행시킨 화면이다. EXE 를 제외하고 두 개의 모듈이 로드 되었음을 알 수 있다.

열거한 모듈 이름 중에서 자신이 사용하지 않은 모듈 이름이 발견된다면 후킹 당하고 있는 것이다. 물론 이 방법의 경우 WH_DEBUG 훅처럼 정확한 후킹 정보를 알 순 없다. 하지만 WH_DEBUG 훅의 경우는 윈도우 메시지 훅만 탐지할 수 있지만, 이 방법은 DLL 인젝트에 기반을 둔 대부분의 공격에 방어할 수 있다는 장점이 있다.

리스트 5 자신의 프로세스에 로드된 모듈을 열거하는 코드

```
#include <windows.h>
#include <tlhelp32.h>

int _tmain(int argc, _TCHAR* argv[])
{
```

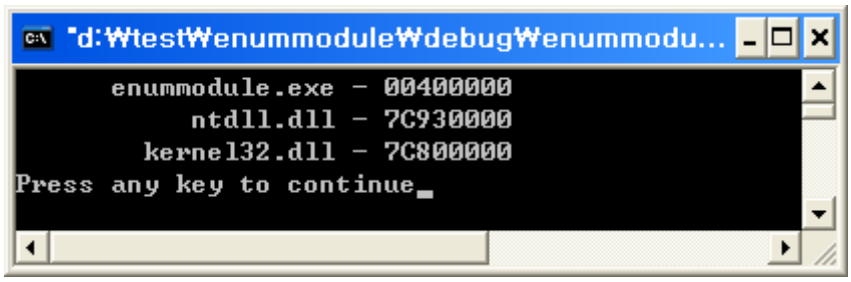
```
    DWORD ret = FALSE;
    // 모듈을 열거할 스냅샷 생성, 두 번째 인자로 프로세스 ID 를 넣어준다.
    HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, GetCurrentProcessId());
    if(snap == INVALID_HANDLE_VALUE)
        return 0;

    MODULEENTRY32 me;
    ZeroMemory(&me, sizeof(me));
    me.dwSize = sizeof(me);

    // 첫 번째 모듈 정보를 구함
    if(!Module32First(snap, &me))
        goto $cleanup;

    // 스냅샷의 끝까지 반복하면서 모듈 이름과 베이스 주소를 출력한다.
    do
    {
        printf("%20s - %08X\n", me.szModule, me.modBaseAddr);
    } while(Module32Next(snap, &me));

$cleanup:
    CloseHandle(snap);
    return 0;
}
```



화면 3 로드된 모듈 목록을 출력하는 프로그램

지난 시간에 배웠듯이 메시지 훅 DLL 이 침투하는 시점을 알 수 없다. 그렇다면 언제 이 검사를 수행해야 할까? 가장 무식한 한 가지 방법 밖에는 없다. 루프를 도는 것이다. 별도의 스레드로 루프를 돌건 아니면 타이머로 돌건 주기적으로 검사해 주는 방법뿐이다.

NT 시스템에서는 유저 레벨에서 프로세스의 생성이나 모듈이 로드 되는 시점을 판단할 수 있는 함수를 제공하지 않는다. 커널 레벨에서는 PsSetImageLoadNotifyRoutine 이나 PsSetCreateProcessNotifyRoutine 을 사용하면 시스템이 프로세스나 모듈이 로드 되는 시점에 등록한 콜백 함수를 호출해 준다.

박스 1 인터럽트와 폴링

I/O 를 처리하는 방식에는 크게 두 가지가 있다. 폴링과 인터럽트가 그것이다. 앞서 예에서 우리는 모듈이 로드 되는 시점을 알 수 없기 때문에 주기적으로 그것을 검사해야 한다고 했다. 이것이 폴링 이다. 이벤트 발생 시점을 알 수 없기 때문에 지속적으로 이벤트가 발생했는지를 체크해야 하는 것이다. 반면에 인터럽트는 이벤트가 발생하면 이벤트가 발생한 곳에서 통보를 해주는 형태다. 앞의 예가 인터럽트 방식이 되려면 시스템에서 모듈이 로드 되는 시점을 우리에게 콜백 함수나 메시지 등을 통해서 알려 주어야 한다. 커널에서 제공하는 함수들이 인터럽트 방식으로 동작한다고 할 수 있다.

도전 과제

훅 디텍터를 좀 더 강화시켜 보자. 단순히 훅 프로시저가 호출된다는 것을 알려주는 것에서 좀 더 나아가서 사용자에게 경고 창을 띄워서 훅 프로시저를 수행할지 건너뛸지 물어보도록 만들어 보자. 또한 그것을 포장해서 다른 사람이 쉽게 쓸 수 있도록 훅 탐지 라이브러리로 제작해 보는 것도 재미있을 것 같다.

이걸로 이제껏 진행했던 훅 강좌는 실질적으로 끝이다. 다음 시간에는 우리의 손과 발을 꽁꽁 묶어 두었던 디버그뷰 유틸리티를 대체할 수 있는 프로그램을 작성하는 방법을 배울 것이다.

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. 프로세스 생성 탐지 방법
http://www.codeproject.com/threads/ProcMon.asp