개발자를 위한 윈도우 후킹 테크닉

OutputDebugString 의 동작 원리

우리는 지금까지 후킹 DLL 을 OutputDebugString 과 DebugView 를 이용해서 동작 내용을 확인했다. 하지만 DebugView 가 활성화 된 상태에서 후킹 DLL 에서 OutputDebugString 을 수행하면 시스템이 잠시 동안 멈추는 현상이 발생한다. 이번 시간에는 이러한 DebugView 의 불편함을 해소하기 위해서 커스텀 디버깅 뷰를 제작하는 방법에 대해서 배운다.

목차

필자 소개

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

시스템 프로그래밍에 관심이 많으며 다수의 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽을 맡고 있으며, Microsoft Visual C++ MVP 로 활동하고 있다.
최근에는 python 과 lua 같은 스크립트 언어를 배우려고 노력하고 있다.

연재 가이드

운영체제: 윈도우 2000/XP

개발도구: 마이크로소프트 비주얼 스튜디오 2003
기초지식: C/C++, Win32 프로그래밍
응용분야: 커스텀 디버그 메시지 뷰어

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

모든 일에 재미란 요소는 굉장히 중요하다. 몇 시간 동안 한 가지 일에 집중할 수 있는 가장 원초적인 이유는 아마도 그 일이 재미있기 때문일 것이다. "학문을 아는 자는 이를 좋아하는 사람만 못하고 학문을 좋아하는 자는 이를 즐기는 자만 못하다."라는 공자님께서 남기신 명언도 재미의 중요성을 강조한 것이라 할 수 있다.
개발도 예외가 아니다. 리누스 토발즈의 <리눅스, 그냥 재미로>, 존 카맥을 그린 <둠>과 같은 책들을 읽어본다면 필자의 생각에 동의할 것이다. 2007 년의 목표로 발전된 개발자를 그리고 있는 독자라면 이러한 프로그래밍의 재미를 찾는데 주력하는 것이 좋을 것 같다. 프로그래밍을 하면서 컴퓨터랑 채팅하고 있다는 생각을 할 정도면 합격이 아닐까?

Intro

이제껏 후킹 DLL 을 디버깅 하면서 한번도 DebugView 의 불편함을 겪지 않았다면 반성해야 할 것 같다. 디버깅 중에 DebugView 때문에 본의 아니게 커피 타임을 가져본 독자라면 아마도 DebugView 의 대안 프로그램을 찾기도 했을 것이다. 이번 시간에는 그런 독자들을 위해서 디버그 메시지 출력에 사용되는 OutputDebugString 의 동작 원리를 살펴보고 DebugView 처럼 메시지를 모니터링 하는 간단한 프로그램을 제작해 본다.

OutputDebugString

OutputDebugString 함수는 디버그 메시지를 출력하는 기능을 한다. 활성화된 디버거가 있는 경우엔 그 곳에 디버그 메시지를 출력하고, 없는 경우라면 아무런 일도 하지 않는다. OutputDebugString 의 원형은 아래와 같다. lpDebugString 으로 전달된 내용을 디버거에 출력해 준다.

```
void OutputDebugString(LPCTSTR lpDebugString);
```

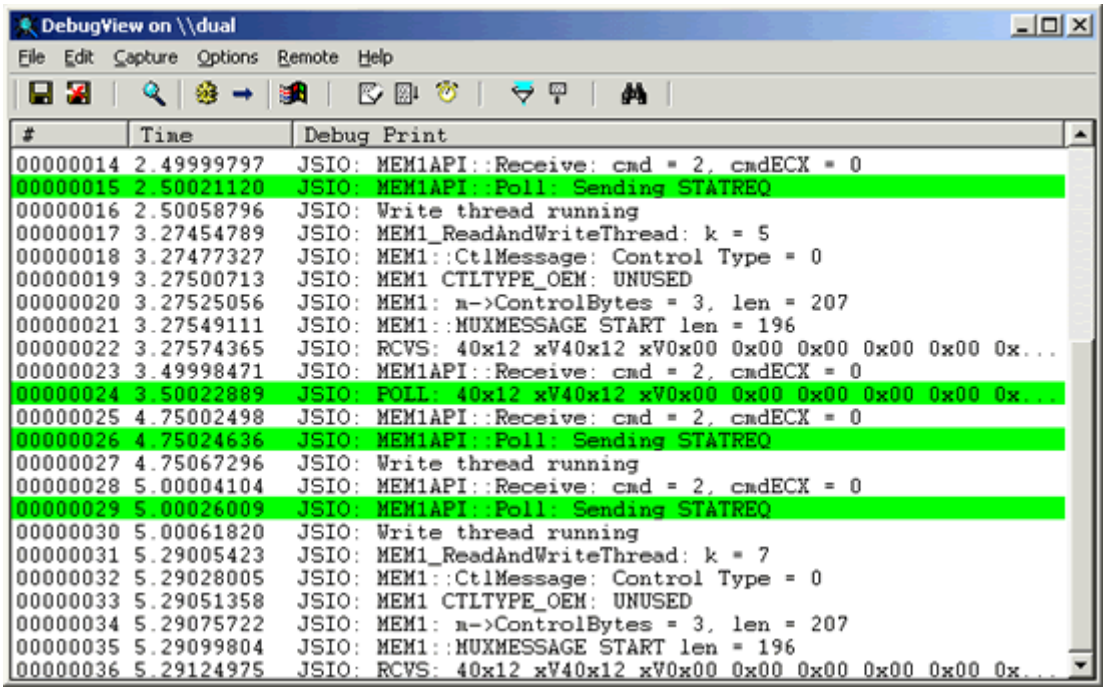
OutputDebugString 을 사용하는 경우는 대부분 특정 함수의 실행 흐름을 추적하거나, 아니면 특정 순간의 변수 값을 확인하고 싶을 때이다. OutputDebugString 은 인자를 하나만 받기 때문에 매번 문자열을 조합해줘야 하는 불편함이 있다. 또한 별도의 매크로로 처리하지 않으면 릴리즈 버전에서도 디버그 메시지를 마구 출력해 버린다. 이런 불편함을 해결하기 위해서 우리는 XTRACE 라는 매크로를 제작해서 사용했었다. <리스트 1>에는 XTRACE 매크로의 소스가 나와있다. XTRACE 의 경우 printf 와 같이 가변 인자를 지원하고, 릴리즈 버전에서는 동작하지 않도록 제작되어 있다. 릴리즈 버전에서도 디버그 메시지를 확인하기 위해서는 FORCE_XTRACE 를 선언해 주면 된다.

리스트 1 xtrace.h

```
#if defined(_DEBUG) || defined(FORCE_XTRACE)

#include <strsafe.h>

#define XTRACE_BUF_SIZE 512
#define XTRACE        _DbgPrintf

inline void __cdecl _DbgPrintf(LPCTSTR str, ...)
{
   TCHAR     buff[XTRACE_BUF_SIZE];
   va_list    ap;

   va_start(ap, str);
   StringCbVPrintf(buff, sizeof buff, str, ap);
   va_end(ap);

   OutputDebugString(buff);
}

#else

#define XTRACE 1 ? (void) 0 : _DbgPrintf
inline void _DbgPrintf(const char *str, ...) {}

#endif
```

DebugView

앞서 OutputDebugString 의 경우 활성화된 디버거가 있는 경우에 메시지를 출력해 준다고 했다. 그러나 디버그 메시지를 사용하는 대부분의 경우는 디버깅 작업이 용의하지 않은 경우가 많다. 이럴 때 디버거를 동작시키지 않고도 디버그 메시지를 출력해 주는 유틸리티가 DebugView 다. 활성화된 디버그가 있으면 DebugView 에 결과가 나타나지 않는다. <화면 1> 에는 DebugView 의 동작 화면이 나와 있다.



화면 1 DebugView 동작 화면

우리는 지금까지 주로 후킹 DLL 의 호출 시점, 컨텍스트, 넘어온 안자 값을 확인하기 위해서 디버그 메시지를 사용했었다. 그런데 DebugView 는 이상하게 자신이 활성화된 상태에서 디버그 메시지를 출력하면 시스템이 멈추는 현상이 발생한다. 이 문제를 재연하는 것은 매우 간단하다. 간단한 마우스 후킹 DLL 을 작성한 다음 WM_LBUTTONDOWN 에서 디버그 메시지를 출력하도록 만든다. 그리고 DebugView 를 활성화 시킨 다음, 그 위에서 마우스 왼쪽 버튼을 눌러보자. 아마도 시스템이 몇 초간 정지한 듯이 동작할 것이다.

이러한 문제 때문에 DebugView 로 후킹 DLL 을 디버깅 하는 개발자들 사이에는 몇 가지 불문율이 있다. 절대로 DebugView 를 디버깅 도중에 건드리지 않거나, 후킹 DLL

컨텍스트의 프로세스를 구하는 부분을 넣어 dbgview.exe 면 건너 뛰도록 제작하는 것이다. 그러나 이 두 가지 모두 근본적인 해결책은 될 수 없다.

OutputDebugString 의 동작 원리

NT 계열의 운영체제에서 OutputDebugString 은 몇 가지 커널 오브젝트를 사용해서 동작하기 때문에 디버그 메시지를 캡쳐하는 것은 무척 간단하다. <표 1>에는 OutputDebugString 이 사용하는 커널 오브젝트들의 이름과 역할이 나와있다.

표 1 OutputDebugString 에 사용되는 커널 오브젝트

이름	타입	역할
DBWinMutex	뮤텍스	OutputDebugString 중복 실행 보호
DBWIN_BUFFER	공유 메모리	OutputDebugString 으로 출력할 문자열
DBWIN_BUFFER_READY	이벤트	DBWIN_BUFFER 에 데이터를 기록할 수 있음
DBWIN_DATA_READY	이벤트	DBWIN_BUFFER 에 데이터가 기록됐음

<그림 1>에는 OutputDebugString 과 DebugView 의 동작 순서도가 나와있다. 왼쪽 편은 OutputDebugString 의 오른쪽 편은 DebugView 의 동작 순서도다. OutputDebugString 부분을 살펴보자. DBWIN_MUTEX 는 OutputDebugString 이 동시에 호출되는 것을 방지하는 역할을 한다. DBWIN_BUFFER 파일 맵, DBWIN_BUFFER_READY, DBWIN_DATA_READY 이벤트는 디버거 쪽에서 생성해 두어야 한다. OutputDebugString 이 이러한 객체를 여는데 실패하면 아무 일도 하지 않고 리턴 한다. 이후 DBWIN_BUFFER_READY 를 대기해서 버퍼에 기록을 해도 되는지 체크한다. 대기가 성공적으로 끝나면 버퍼에 메시지를 기록하고, DBWIN_DATA_READY 를 설정해서 디버거에게 디버그 메시지가 기록됐음을 알려준다.

DebugView 측은 OutputDebugString 이 제대로 동작하도록 만들어 주면 된다. 시작할 때에 각종 커널 오브젝트를 생성해 준다. 그런 다음 DBWIN_BUFFER_READY 를 설정해서 디버거가 버퍼에 기록할 수 있도록 해준다. 이 후 DBWIN_DATA_READY 이벤트를 대기한다. 누군가 OutputDebugString 을 호출해서 해당 이벤트가 설정되면 DBWIN_BUFFER 의 내용을 화면에 출력해 주고, DBWIN_BUFFER_READY 이벤트를 설정해서 다음 OutputDebugString 이 버퍼를 쓸 수 있도록 해준다.

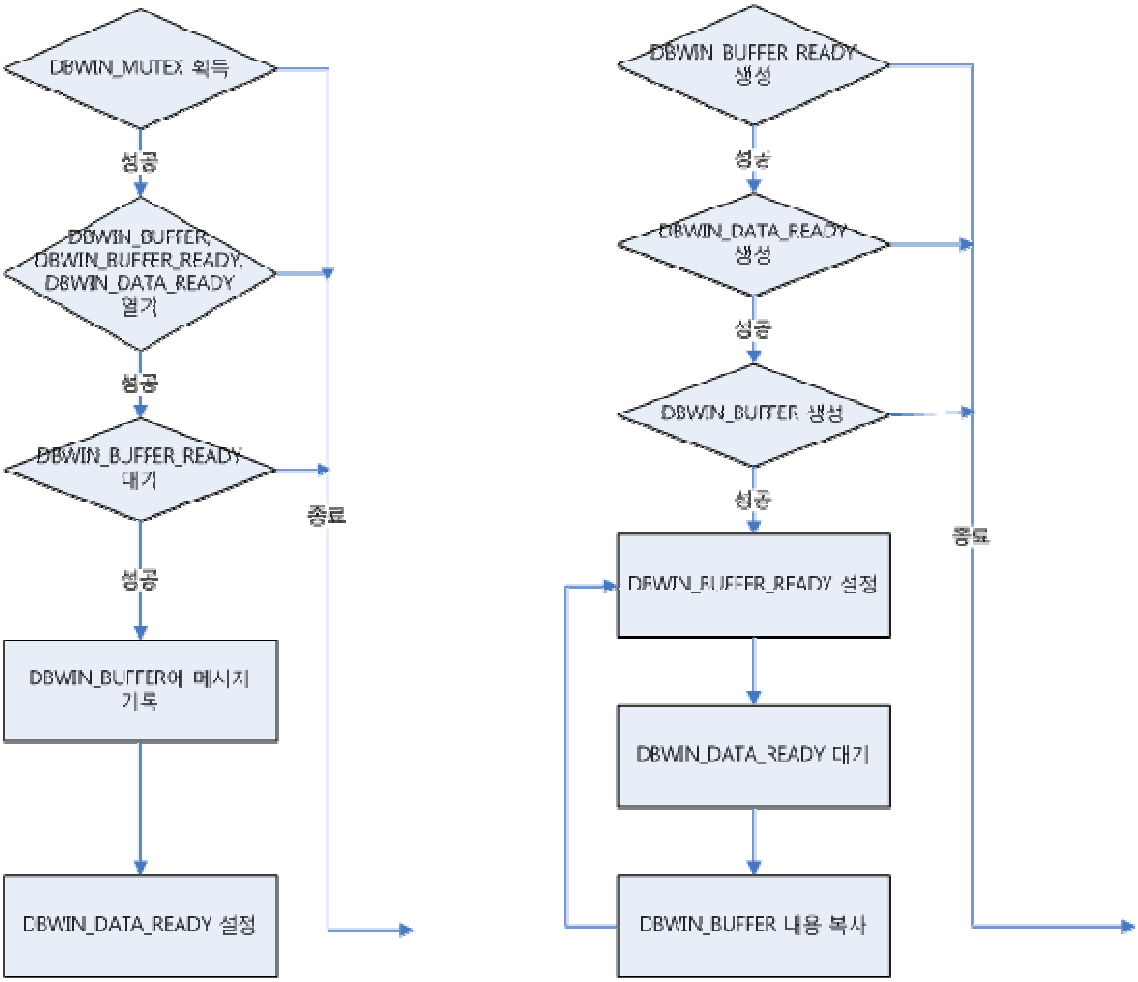


그림 1 OutputDebugString 및 DebugView 순서도

우리가 Spy++의 클론 프로그램인 imSpy 를 제작할 때 살펴보았던 것과 같이 이러한 전역 커널 오브젝트를 사용해서 데이터를 주고 받는 프로그램의 경우에 데이터를 수신하는 프로그램이 여러 개가 되면 엉뚱하게 동작할 수 있다. 동기화가 깨어지기 때문이다. 따라서 모니터링 프로그램은 반드시 커널 오브젝트가 이미 존재하는 경우에는 사용자에게 이미 다른 디버거가 존재함을 알려주고 모니터링을 중지해야 한다.

<리스트 2>에는 파일맵 오브젝트인, DBWIN_BUFFER 에 기록되는 내용이 무엇인지를 담고 있다. 파일맵 오브젝트엔 pid 와 출력할 문자열이 저장되어 있다. msg 는 1 글자를 저장할 수 있는 배열이 아니라, 가변 길이를 지원하기 위해서 저렇게 사용한 것이다. 이와 관련된 내용은 C-Faq 2.6 에 자세히 소개되어 있다(http://c-faq.com/struct/structhack.html 참고).

리스트 2 DBWIN_BUFFER 에 기록된 내용

```
// 파일맵 오브젝트에 기록된 내용
typedef struct _DEBUGMMF
{
    DWORD pid;
    char msg[1];
} DEBUGMMF, *PDEBUGMMF;
```

박스 1 9x 에서 OutputDebugString 메시지 모니터링 하기

지금까지 설명한 내용은 모두 NT 계열의 운영체제 국한된 내용이다. 9x 계열의 OutputDebugstring 은 커널 객체를 사용해서 정보를 전달하지 않기 때문에 이러한 방법으로 메시지를 모니터링 할 수 없다. 하지만 간단한 방법으로 동일하게 디버그 메시지를 감시할 수 있는 방법이 있다. OutputDebugString 을 직접 구현하는 방법이다. 참고 자료에 있는 dbwin32 소스를 보면 그런 방법이 사용되어 있다. 물론 이 경우 기존의 프로그램을 수정해야 한다는 불편함이 있긴 하다.

OutputDebugString 감시 스레드

이제 모니터링 프로그램의 가장 핵심적인 부분은 디버그 메시지를 감시하는 스레드를 살펴보자. <리스트 3>에 감시 스레드의 코드가 나와있다. <그림 1>에 나와있는 순서도와 동일한 절차로 구현되었다.

리스트 3 OutputDebugString 감시 스레드

```
UINT CWatchThread::Go()
{
    // 모든 객체가 접근할 수 있도록 NULL DACL 을 만든다.
    CSecurityAttributes sa;
    CSecurityDesc sd;

    sd.SetDacl(TRUE);
    sa.Set(sd);

    // DBWIN_BUFFER_READY 이벤트를 생성한다.
    am::mate<HANDLE> bufReadyEvent(
        CreateEvent(&sa, FALSE, FALSE, "DBWIN_BUFFER_READY")
        , &CloseHandle );
    if(bufReadyEvent == NULL)
    {
        bufReadyEvent.dismiss_mate();
        return 0;
    }

    if(GetLastError() == ERROR_ALREADY_EXISTS)
        return 0;

    // DBWIN_DATA_READY 이벤트를 생성한다.
    am::mate<HANDLE> dataReadyEvent(
```

```
      CreateEvent(&sa, FALSE, FALSE, "DBWIN_DATA_READY")
      , &CloseHandle );
   if(dataReadyEvent == NULL)
   {
      dataReadyEvent.dismiss_mate();
      return 0;
   }

   // DBWIN_BUFFER 파일 맵 오브젝트를 생성한다.
   am::mate<HANDLE> mmf(
      CreateFileMapping(INVALID_HANDLE_VALUE, &sa, PAGE_READWRITE, 0, 4096,
"DBWIN_BUFFER")
      , &CloseHandle );
   if(mmf == NULL)
   {
      mmf.dismiss_mate();
      return 0;
   }

   // DBWIN_BUFFER 를 사용할 수 있도록 맵핑한다.
   am::mate<PVOID> shareMem(
      MapViewOfFile(mmf, FILE_MAP_READ, 0, 0, 512)
      , &UnmapViewOfFile );
   if(shareMem == NULL)
   {
      shareMem.dismiss_mate();
      return 0;
   }

   PDEBUGMMF dmsg = (PDEBUGMMF)(PVOID) shareMem;

   HANDLE objs[2] = { m_exitEvent, dataReadyEvent };
   DWORD obj;
   DEBUGMSG log;

   for(;;)
   {
      // 버퍼에 기록할 수 있음을 알린다.
      SetEvent(bufReadyEvent);

      // 버퍼에 데이터가 기록될 때까지 대기한다.
      obj = WaitForMultipleObjects(2, objs, FALSE, INFINITE);

      // 종료 이벤트인 경우엔 루프를 탈출한다.
      if(obj == WAIT_OBJECT_0)
         break;

      // 로그 기록이 중지된 경우엔 루프를 새로 시작한다.
      if(IsPaused())
         continue;

      {
         // 로그 벡터에 접근하기 위해서 뮤텍스를 획득한다.
```

```
            am::mate<DWORD> locker(
               WaitForSingleObject(GetMutex(), INFINITE)
               , am::lambda::bind(&ReleaseMutex, GetMutex()) );
            if(locker != WAIT_OBJECT_0 || locker != WAIT_ABANDONED)
               continue;

            // 로그 정보를 설정한다.
            log.msg = dmsg->msg;
            log.pid = dmsg->pid;
            log.t = time(NULL);

            // 로그 벡터에 추가한다.
            m_logs.push_back(log);
         }

         // 메인 윈도우에 로그가 추가되었음을 알린다.
         AfxGetMainWnd()->PostMessage(WM_DEBUGMSGNOTIFY, 0, 0);
      }

      return 0;
   }
```

스레드의 가장 앞 쪽에는 ATL 의 CSecurityAttributes, CSecurityDesc 를 사용해서 NULL DACL 을 생성하는 부분이 있다. 이 코드를 디버그 버전으로 수행하면 NULL DACL 생성 오류가 발생한다. 이 오류는 코드가 잘못돼서 나는 것이 아니라 NULL DACL 은 보안상 위험하기 때문에 개발자에게 알려주기 위해서 오류를 내는 것이다. 디버깅 하는데 이 부분이 번거롭다고 생각한다면 <리스트 4>에 나타난 것과 같이 API 를 사용해서 생성하면 된다.

이렇게 NULL DACL 을 보안 속성으로 지정하는 이유는 모니터링 프로그램의 권한보다 낮은 권한으로 동작하는 프로그램에서도 이 객체들을 원활하게 열 수 있도록 하기 위해서다. 만약 디버거가 관리자 권한으로 객체를 만들게 되면, 사용자 권한으로 실행되는 프로그램에서는 해당 객체가 존재하더라도 접근 거부를 받는다. 결국은 디버그 메시지를 출력하지 못하는 것이다.

리스트 4 API 를 사용해서 NULL DACL 을 생성하는 부분

```
security_attributes sa;
security_descriptor sd;

sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.bInheritHandle = TRUE;
sa.lpSecurityDescriptor = &sd;

if(!InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION))
   return;
```

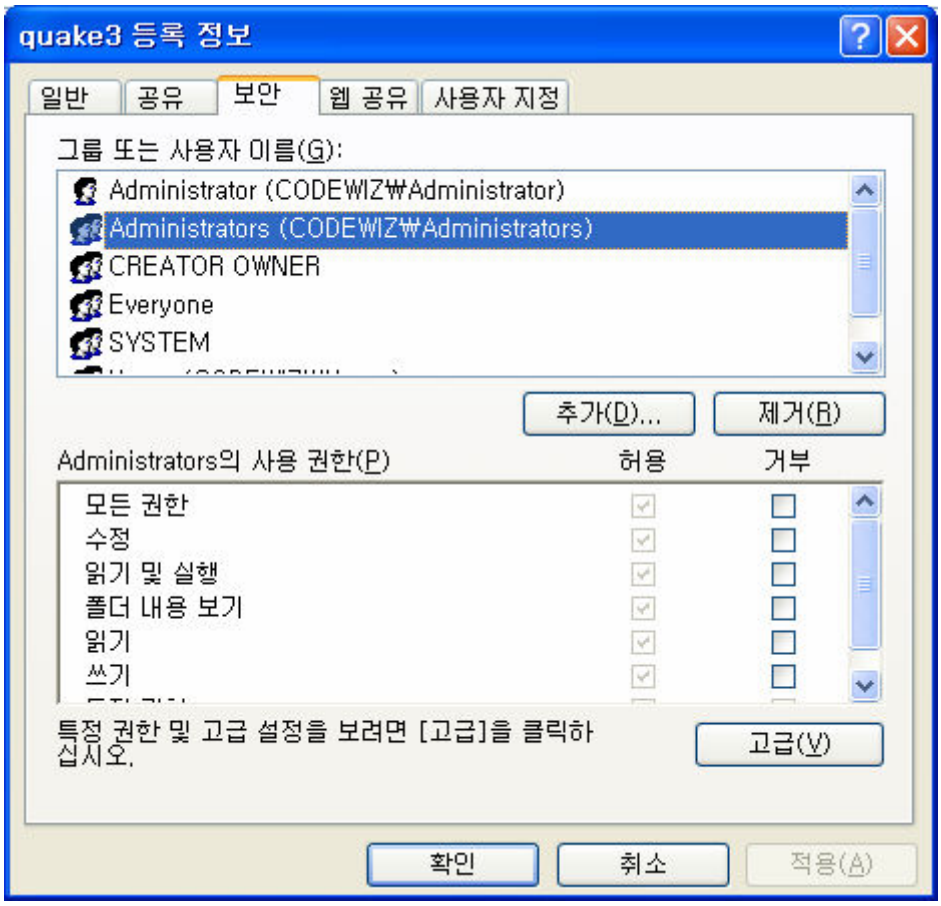
```
if(!SetSecurityDescriptorDacl(&sd, TRUE, (PACL)NULL, FALSE))
   return;
```

박스 2 mate 클래스

예외에 안전한 코드를 작성하기 위해서 가장 중요한 점은 할당된 자원을 정확하게 반환하는 것이다. a, b, c, d 란 자원을 순차적으로 할당하는 함수를 생각해보자. a 가 할당된 다음 예외가 발생했다면 a 만 해제해야 한다. c 까지 할당된 상황에서 예외가 발생했다면 a, b, c 를 해제한 다음 함수가 종료해야 할 것이다. 사용하는 자원의 개수가 많아질수록 코드의 관리가 힘들어진다. 이런 경우에 C++의 생성자와 소멸자를 사용하면 손쉽게 자원을 관리할 수 있다. 생성자에서 자원을 할당하고, 소멸자에서 해제하는 것이다.

mate 는 생성자와 소멸자를 사용해서 자원의 관리를 자동으로 해주는 템플릿 클래스다. 데브피아의 최재욱님께서 작성하신 것으로 굉장히 사용하기 편리하고 실용적이다. mate 클래스에 대해서 더 자세한 정보는 devpia 의 강좌 게시판(http://www.devpia.com/Forum/BoardView.aspx?no=7528&forumname=vc_lec)에 나와있다.

박스 3 ACL



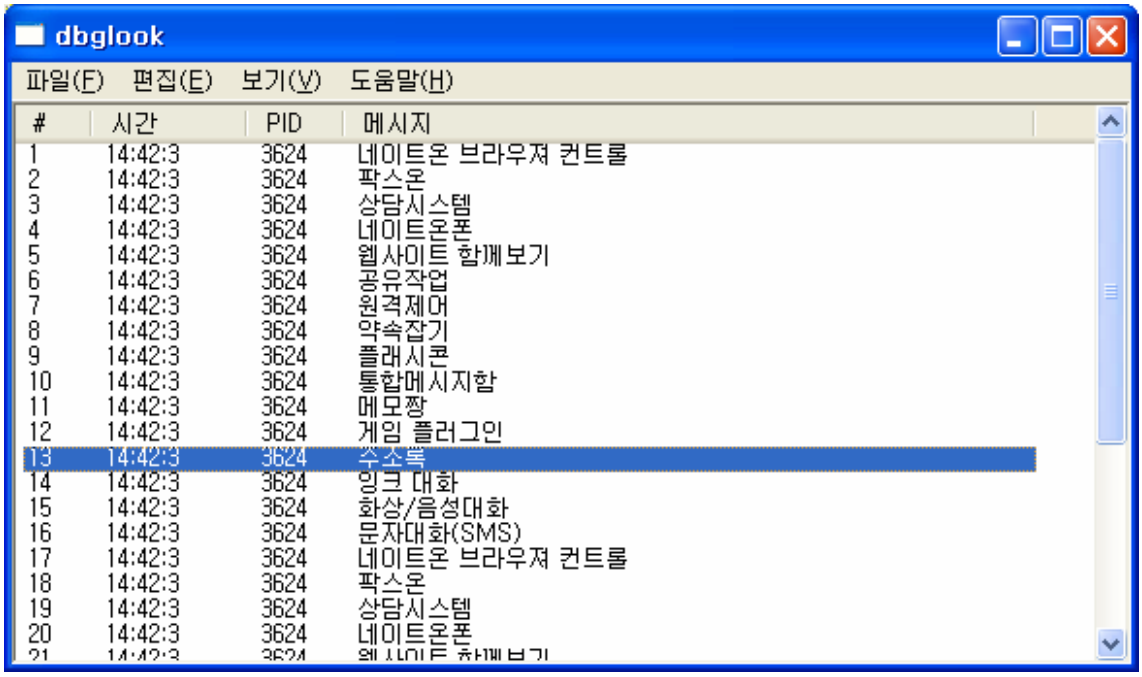
화면 2 특정 폴더에 지정된 ACL 을 확인하는 화면

ACL 은 Windows NT 계열의 운영체제에서 사용하는 리소스의 접근 제어 방법이다. 가장 쉽게 볼 수 있는 곳은 NTFS 의 파일이나 폴더의 권한을 설정하는 화면이다(<화면 2> 참고). 그런데 이 ACL 을 생성하고 관리하는 API 는 사용하는 방법이 굉장히 어렵고 복잡하다. 아래 문서는 ACL 에 관해서 자세하게 설명하고 있는 문서다. 아직 ACL 에 대해서 잘 모르고 있다면 꼭 읽어 보도록 하자.

http://www.codeproject.com/win32/accessctrl1.asp
http://www.codeproject.com/win32/accessctrl2.asp
http://www.codeproject.com/csharp/accessctrl3.asp
http://www.codeproject.com/win32/accessctrl4.asp

DbgLook

이번 시간에 샘플로 제공할 프로그램인 DbgLook 은 앞서 소개한 원리에 기초해서 기본적인 내용만 구현한 것이다. <화면 3>에 실행 화면이 나와 있다. 프로그램을 실행하면 바로 모니터링을 시작한다. 각종 인터페이스 관련 코드는 하나도 없다. 따라서 모니터링 중지도, 리스트 삭제도 없다.



화면 3 DbgLook 동작 화면

<리스트 5>에는 내부적으로 사용한 구조체와 정의 부분이 나와있다. 디버그 메시지에 저장하는 내용은 pid 와 메시지 문자열, 메시지가 발생한 시간이 전부다.

리스트 5 디버그 메시지를 저장하는 구조체

```
#include <vector>

typedef struct _DEBUGMSG
{
   DWORD   pid;
   CString msg;
   time_t  t;
} DEBUGMSG, *PDEBUGMSG;

typedef std::vector<DEBUGMSG> DMVec;
typedef std::vector<DEBUGMSG>::iterator DMVIt;

const DWORD WM_DEBUGMSGNOTIFY = WM_USER + 1394;
```

<리스트 6>에는 앞서 살펴 보았던 감시 스레드에서 메시지를 전달한 경우에 수행되는 메시지 핸들러가 나와있다. 로그 벡터에 접근하기 위한 뮤텍스를 획득하고 로그 벡터의 내용을 리스트 뷰에 추가한다. 작업이 완료되면 로그 벡터를 삭제한다.

리스트 6 스레드에서 보낸 메시지에 대한 핸들러

```
LRESULT CMainFrame::OnDebugMsgNotify(WPARAM w, LPARAM l)
{
   // 로그 벡터에 접근하기 위한 뮤텍스를 획득한다.
   HANDLE mutex = m_watchThread.GetMutex();
   am::mate<DWORD> locker(
      WaitForSingleObject(mutex, INFINITE)
      , am::lambda::bind(&ReleaseMutex, mutex) );
   if(locker != WAIT_OBJECT_0 || locker != WAIT_ABANDONED)
   {
      locker.dismiss_mate();
      return 0;
   }

   // 로그 벡터의 내용을 윈도우에 추가 시킨다.
   DMVIt it = m_watchThread.m_logs.begin();
   DMVIt end = m_watchThread.m_logs.end();
   for( ; it != end; ++it)
   {
      m_wndView.AddMsg(*it);
   }

   // 로그 벡터의 내용을 지운다.
   m_watchThread.m_logs.clear();
   return 0;
}
```

메시지 핸들러에서 AddMsg 함수를 호출한 경우에 수행되는 부분이 <리스트 7>에 나타나 있다. 리스트 컨트롤에 메시지를 출력하는 단순한 부분이다. 이 경우에 몇 가지 주의해야 할 점이 있다. 에디터 컨트롤을 사용한다면 단순히 메시지를 그대로 출력하면 되지만 우리와 같이 리스트 컨트롤을 사용하는 경우에는 메시지를 ₩n 을 기준으로 분리해서 출력해 주는 것이 보기가 좋다. 또한 탭도 적당한 공백으로 확장해서 출력해 주는 것이 좋다.

리스트 7 뷰에 디버그 정보를 출력하는 부분

```
void CChildView::AddMsg(DEBUGMSG &msg)
{
   int pos = 0;
   CString res;
   CString buf;

   // \n 기준으로 분리한다.
```

```
    while((res = msg.msg.Tokenize("\r\n", pos)) != "")
    {
        // 로그 번호를 출력한다.
        int cnt = m_lvcMsgs.GetItemCount();
        buf.Format("%d", cnt+1);
        m_lvcMsgs.InsertItem(LVIF_TEXT, cnt, buf, 0, 0, 0, 0);

        // 탭을 공백으로 확장 시킨다.
        res.Replace("\t", "    ");

        // 정보를 출력한다.
        CTime t(msg.t);
        buf.Format("%d:%d:%d", t.GetHour(), t.GetMinute(), t.GetSecond());
        m_lvcMsgs.SetItemText(cnt, 1, buf);

        buf.Format("%d", msg.pid);
        m_lvcMsgs.SetItemText(cnt, 2, buf);
        m_lvcMsgs.SetItemText(cnt, 3, res);
    }
}
```

₩n 으로 토큰을 분리할 때 한 가지 생각해야 할 점이 있다. "₩n₩nabc"와 같은 문자열을 출력한 경우다. 이 경우 텍스트 에디터라면 빈 줄이 두 개 삽입되고 abc 가 나온다. 그러나 <리스트 7>의 코드와 같이 토큰을 분리하면 빈 줄 없이 그냥 abc 가 삽입된다. strtok 나 Tokenize 와 같은 함수가 구분자가 연속으로 있는 경우 그 부분을 하나의 토큰으로 분리해 내지 않기 때문이다. 이 경우 빈 줄을 정확하게 표시하기 위해서는 토큰 분리 함수를 직접 구현해야 한다.

박스 4 릴리즈 버전과 디버그 메시지

필자는 릴리즈 버전에서는 모든 디버그 메시지가 제거되는 것이 바람직하다고 생각한다. 왜냐하면 디버그 메시지 출력은 성능을 저하시킬 뿐 아니라 다른 개발자가 디버깅 할 때에는 마치 공해와도 같은 존재이기 때문이다. 자신의 디버그 메시지 하나를 확인하려고 기다리고 있는데 엉뚱한 프로그램에서 마구 디버그 메시지를 출력한다고 생각해보자. 아마도 자신의 메시지는 찾기도 힘들 것이다.

디버그 메시지를 남겨두는 입장은 릴리즈 버전에서 문제가 생긴 컴퓨터의 정보를 손쉽게 확인할 수 있다는 것이다. 일반적인 유저는 디버그 메시지 뷰어를 켜두지 않을 것이고 그런 의미에서 성능 저하도 없다고 생각한다. 하지만 일반적인 유저엔 개발자도 포함된다는 사실을 간과해서는 안 된다.

만약 자신이 후자의 생각이 더 바람직하다고 생각한다면 DLL 을 사용해서 문제가 없는 컴퓨터엔 피해가 가지 않도록 하는 것이 좋겠다. 방법은 간단하다. DLL 에 디버그 출력

함수를 구현하는 것이다. 함수 이름을 MyOutputDebugString 이라고 가정해 보자. DLL 을 두 종류를 만든다. 일반적인 릴리즈 버전의 MyOutputDebugString 은 아무 일도 하지 않는다. 다른 버전은 MyOutputDebugString 함수가 단순히 OutputDebugString 을 호출하도록 만든 것이다. 특정 컴퓨터에서 문제가 생긴다면, 후자의 DLL 을 다운로드 시켜서 디버그 메시지를 수집하면 된다.

도전 과제

이번 달의 도전과제는 우리가 제작한 dbglook 을 각자 자신의 입맛에 맞게 고쳐 보는 것이다. 사용해 보면서 불편함을 느낀 점들을 한 가지씩 개선해 보도록 한다.

이것으로 지난 9 개월간 연재했던 후킹 강좌는 모두 마무리 되었다. 여기가 끝이라고 생각해서는 안 된다. 끝은 항상 새로운 시작의 출발점일 뿐이다. 메시지 후킹은 여러 기법들 중에서 가장 기초적인 방법이다. 좀 더 발전된 후킹 방법이나 주제에 대해서 공부하고 싶다면 참고 자료에 있는 책들을 읽어보도록 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. Gary McGraw, Greg Hoglund <<소프트웨어 보안: 코드 깨부수기>> 정보문화사

참고자료 5. Kris Kaspersky, Natalia Tarkova, Julie Laing, <<Hacker Disassembling Uncovered>> A-List Publishing

참고자료 6. Greg Hoglund, Jamie Butler, <<Rootkits: Subverting the Windows Kernel>> Addison-Wesley Professional

참고자료 7. OutputDebugString 의 동작 원리 http://www.unixwiz.net/techtips/outputdebugstring.html

참고자료 8. Dbwin32 소스 코드 http://grantschenck.tripod.com/dbwinv2.htm