Windows Hook

Jerald Lee
Contact Me : lucid7@paran.com

본 문서는 저자가 Windows Hook을 공부하면서 알게 된 것들을 정리할 목적으로 작성되었습니다. 본인이 Windows System에 대해 아는 것의 거의 없기 때문에 기존에 존재하는 문서들을 짜집기 한 형태의 문서로밖에 만들 수가 없었습니다. 문서를 만들면서 참고한 책, 관련 문서 등이 너무 많아 일일이 다 기술하지 못한 점에 대해 원문 저자들에게 매우 죄송스럽게 생각합니다.
본 문서의 대상은 운영체제와 Win32 API를 어느 정도 알고 있다는 가정하에 쓰여졌습니다. 본 문서에 기술된 일부 기법에 대한 자세한 설명은 책을 참조하시길 바랍니다.

제시된 코드들은 Windows XP Service Pack2, Visual Studio .net 2003(Eng)에서 테스트 되었습니다. 문서의 내용 중 틀린 곳이나 수정해야 할 부분이 있으면 연락해 주시기 바랍니다.

목 차

1. CreateRemoteThread() API

이번에 살펴볼 방법은 CreateRemoteThread API를 이용하는 방법이다. 이 방법은 1994년 Microsoft Systems Journal 에서 Jeffrey Ritcher에 의해 공개되었으며 그의 기사인 “Load Your 32-bit DLL into Another Process’s”에서 설명되었다.
해당 주제에 대해 발표한 많은 문서에서 Reference로 참고되었다고 기록되었음에도 불구하고 불행히도 Jeffrey Ritcher의 문서는 구할 수 없었다. 대신 Programming Applications Microsoft Windows 4th Edition의 “Injecting a DLL Using Remote Threads”를 참고하였다.

1.1. 요약

CreateRemoteThread API를 이용한 DLL Injection은 여러 많은 기법들 중에서도 대부분의 사람들이 선호하는 기법중의 하나이다.
결론부터 말하자면 이 기법은 다음의 순서로 요약될 수 있다.

1. VirtualAllocEx API를 사용해 타겟 프로세스 주소 공간에 메모리를 할당한다.
2. WriteProcessMemory API를 이용해 주입시키고자 하는 DLL의 Pathname을 할당한 메모리로 복사한다.
3. GetProcAddress API를 사용해 LoadLibraryA 또는 LoadLibraryW API의 주소를 가져온다.
4. CreateRemoteThread API를 이용해 타겟 프로세스의 1에서 할당한 공간에 LoadLibrary API를 호출하는 스레드를 만든다.
5. VirtualFreeEX API를 이용해 1에서 할당한 메모리를 해제한다.
6. GetProcAddress API를 이용해 FreeLibrary API의 주소를 가져온다.
7. CreateRemoteThread API를 이용해 주입한 DLL의 HINSTANCE를 사용, FreeLibrary를 호출하는 스레드를 타켓 프로세스에 만든다.

앞 절에서 다루었던 Debug Technique 보다 훨씬 쉽고 깔끔하다. 많은 사람들이 선호하는 이유가 바로 여기에 있는지도 모르겠다.(실제로 나중에 다루어 볼 IAT Patching이나 EAT Patching은 조금 문제가 있다. 아니 있다고 한다 -_-;)

개념은 더욱 단순하고도 명확하다. 타겟 프로세스에 Dll을 주입하기 위해 타겟 프로세스에서 LoadLibrary API를 이용해 Dll을 로드한다는 것이다. 그리고 LoadLibrary API를 이용하기 위해 CreateRemoteThread API를 사용한다는 것이다.

1.2. 코드 분석

여기까지 오면 다음과 같은 궁금한 점이 생길 것이다. CreateRemoteThread API는 이름이 그러하드시 Thread를 생성시키는 API인데 어떻게 이걸 이용해서 LoadLibrary를 쓸 것인가?

먼저 CreateRemoteThread를 살펴보자. 이 API의 함수 원형은 다음과 같다.

```
HANDLE WINAPI CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

hProcess : [in], 스레드가 생성될 타겟 프로세스. 스레드 생성에 성공하기 위해서는 이 핸들에 대해 PROCESS_CREATE_THREAD, PROCESS_QUERY_INFORMATION, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, PROCESS_VM_READ 권한이 있어야 한다. lpThreadAttributes : [in], 보안 속성을 지정. NULL이면 Default 값을 가져오게 된다.
dwStackSize : [in], 스택의 사이즈를 나타낸다. 0을 설정하면 Default size를 할당한다.
lpStartAddress : [in], 생성할 스레드 함수의 주소. 이 함수는 반드시 타겟 프로세스 내에 존재하는 함수이어야 한다. 더 자세한 정보는 ThreadProc를 참고.
lpParameter : [in], 함수에 전달되는 변수
dwCreationFlags : [in], suspend와 같은 스레드 속성을 지정한다. 0을 설정하면 스레드는 즉시 시작하게 된다.
lpThreadId : [out], 스레드 ID값을 받을 변수, NULL 지정 시 스레드 ID값이 반환되지 않는다.

우리가 관심을 가져야 할 부분은 바로 LPTHREAD_START_ROUTINE lpStartAddress 이 부분이다. 위쪽에도 설명이 되어 있듯이 이 변수는 스레드로 생성할 함수를 가리키는 주소이다.

스레드의 함수 원형은 아래와 같다.

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
```

그리고 LoadLibrary API 시리즈의 함수 원형은 아래와 같다.

```
WINBASEAPI HMODULE WINAPI LoadLibraryA(IN LPCSTR lpLibFileName);
```

```
WINBASEAPI HMODULE WINAPI LoadLibraryW(IN LPCWSTR lpLibFileName);

#ifdef UNICODE
#define LoadLibrary   LoadLibraryW
#else
#define LoadLibrary   LoadLibraryA
#endif
```

스레드 함수와 LoadLibrary API는 비슷한 형태를 가지고 있다. 동일한 호출규약인 WINAPI를, 하나의 매개변수를 받아서 동일한 크기의 값을 리턴한다.
(DWORD, HINSTANCE 모두 unsigned long형의 32비트 정수 값이다.)
즉, ThreadFunc 대신 LoadLibraryA나 LoadLibraryW의 주소를 전달하는 것이 이 기법의 핵심 내용이며 그 주소 값은 GetProcAddress API로 구할 수 있다.

```
hThread = ::CreateRemoteThread(hUserAPI, NULL, 0, pfnLoadLibrary, "Path to dll", 0, NULL);
```

위의 코드가 이 문서의 목적이다.

위의 코드에서 pfnLoadLibrary는 GetProcAddress로 구해 온 LoadLibrary의 주소이다. 직접 LoadLibrary를 호출하지 않고 GetProcAddress를 쓴 이유는 굳이 이야기 하지 않아도 다들 알 것이라고 믿는다. -_-
하지만 노파심에서 다시 줄줄이 이야기 해 보자.
Windows의 경우 특정 DLL에서 export하는 함수를 다른 어플리케이션에서 불러와서 사용하기 위해서는 해당 DLL의 Entry Point가 어플리케이션의 메모리에 올라와 있어야 하는데 이 Entry Point는 IAT(Import Address Table)에 기록되고 어플리케이션은 이 테이블을 참조하여 Import하게 된다. 해당 DLL의 Entry Point가 IAT에 기록되는 과정은 컴파일 시에 이루어지게 되고 실행될 때마다 그 위치는 달라지게 된다.
따라서 CreateRemoteThread API의 네 번째 변수로 직접 LoadLibrary를 호출하게 되면 현재 프로세스 내의 Import Table에 기록되어 있는 Entry Point를 참고하게 되고 잘못된 주소를 참조하게 됨으로써 대부분 Access Violation을 일으키게 된다.
LoadLibrary API가 포함되어 있는 Kernel32.dll은 거의 대부분의 Windows Application에서 필요로 하기 때문에 우리는 GetProcAddress API를 이용해 LoadLibrary의 주소를 구해올 수가 있는 것이다.

LoadLibrary의 주소인 pfnLoadLibrary는 아래와 같이 구할 수 있다.

```
[Ansi]
PTHREAD_START_ROUTINE pfnLoadLibrary =
(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
```

```
                                                                    “LoadLibraryA”
);
[Unicode]
PTHREAD_START_ROUTINE pfnLoadLibrary =
(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT(“Kernel32”)),
                                                                    “LoadLibraryW
”);

[WINBASE.H]
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(
    LPVOID lpThreadParameter
    );
```

PTHREAD_START_ROUTINE는 함수 포인터로서 생소한 사람들도 있을 것이다.(적어도 나는 그랬다 -_-;). 잠깐 함수 포인터를 정리하고 넘어가보자.

함수 포인터는 말 그대로 함수를 가리키는 포인터를 말한다. 일반적으로 포인터는 메모리를 가리키는 것이 일반적이지만 함수 역시 메모리 번지를 가리키는 변수이므로 함수를 가리키는 포인터를 당연히 쓸 수 있다.

함수 포인터를 선언하는 방법은 아래와 같다.

```
type (* 변수명)(인수)
```

```
type : 함수 포인터가 가리키는 함수의 리턴 값
변수명 : 함수 포인터의 이름
인수 : 함수 포인터가 가질 값
```

함수를 함수 포인터로 변환하는 쉬운 방법은 함수의 원형을 써놓고 함수명을 변수명으로 바꾸고 앞에 *를 붙인 후 * 변수명을 괄호로 싸면 된다.

아래는 winapi 사이트 자료에서 가져온 일반함수를 함수 포인터로 변환하는 방법이다.

```
원형 함수 : int func(int a)
① int pf(int a) : 함수명을 변수명으로 바꿈
② int *pf(int a) : 변수명 앞에 *를 붙임
③ int(*pf)(int) : 변수를 괄호로 쌈. 형식 인수의 이름은 생략 가능
```

실제 사용 예는 아래와 같다.

```
#include <stdio.h>

int func(int a)
{
	return a*2;
}

int main()
{
	int i;
	int (*pf)(int a);
	pf = func;
	i=(*pf)(2);
	printf("%d₩n", i);
}
```

타입이 다른 함수 포인터끼리 강제 대입하는 예제는 아래와 같다.

```
int (*pf1)(char *);
void (*pf2)(double);
pf1=(int (*)(char *))pf2;
```

일반적으로 명확한 표기 및 사용을 위해서 typedef로 함수 포인터 타입을 따로 정의한 후 사용하는 방식이 널리 쓰이고 있는데 아래와 같다.

```
typedef int (*PFTYPE)(int);
PFTYPE pf;
PFTYPE arpf[5];
PFTYPE *ppf;
```

자 이 정도면 함수 포인터에 대한 개념이 어느 정도 이해가 되었을 것이다. 안되었다면 다른 자료를 참고할 것을 권장한다.
이제 위로 다시 돌아가서 코드를 분석해보자.

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(
	LPVOID lpThreadParameter
	);
```

위의 코드는 DWORD형의 type을 반환하고 LPVOID형의 변수를 인자로 받는 PTHREAD_START_ROUTINE 라는 함수 포인터이다. WINAPI는 호출 규약이므로 신경 쓰지

않아도 된다. (호출 규약이란 함수가 호출될 때 함수의 매개변수와 어플리케이션에서의 호출 지점 정보가 스택에 어떻게 push 될지와 호출한 함수와 호출된 함수 중 어느 쪽에서 스택을 비울지에 대한 규약을 말한다. 더 자세한 내용은 윈도우 프로그래밍 관련 서적을 참고하기 바란다)
그러므로 LoadLibraryA의 주소를 알아오는 아래의 코드는

```
PTHREAD_START_ROUTINE pfnLoadLibrary =
(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
"LoadLibraryA");
```

GetProcAddress API의 결과로 리턴된 값을 PTHREAD_START_ROUTINE 함수 포인터형으로 변환하여 DWORD type의 값을 리턴하는 pfnLoadLibrary 함수 포인터의 인자 값으로 전달한다는 의미이다.

왜 이렇게 복잡한 방법을 쓰는 것일까? MSDN은 그 이유를 다음과 같이 설명하고 있다.

> DLL 함수는 포인터를 통해 호출하며 컴파일 타임 형식 검사가 수행되지 않으므로, 이 함수에 대한 매개 변수가 정확해야만 스택에 할당되는 메모리 한도가 초과되거나 액세스 위반이 발생하지 않게 됩니다. 형식 안전성을 보장하는 한 가지 방법은 내보내기 함수의 함수 프로토타입을 확인하고 해당 함수 포인터에 맞는 형식 정의(typedef)를 만드는 것입니다

라고 하니 그냥 권고하는 대로 쓰는 것이 좋을 듯하다 -_-;
LoadLibrary를 GetProcAddress를 이용해 사용하는 것과 동일한 문제로 타겟 프로세스에 주입할 DLL Name 역시 타겟 프로세스의 메모리로 복사한 후 참조해야만 한다.

자 이제 마지막으로 VirtualAllocEx API와 VirtualFreeEx API를 살펴보자.

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flAllocationType,
  DWORD flProtect
);
```

hProcess : [in], 타겟 프로세스의 핸들
lpAddress : [in], 할당 받고 싶은 페이지 영역의 시작 주소를 가리킴. NULL To Auto.
dwSize : [in], 할당 받고 싶은 메모리 영역의 크기. Byte 단위, NULL To lpAddress부터 다음 페이지 경계까지 자동으로 할당
flAllocationType : [in], 메모리 할당의 타입을 결정한다. 여러 타입이 있는데 자세한 것은 MSDN을 참고하면 된다. MEM_COMMIT는 Commit을, MEM_RESERVE는 예약을 나타낸다는

것 정도만 알아두자.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualallocex.asp

flProtect : [in], MEM_COMMIT 시 할당된 메모리에 대한 보호모드를 나타낸다. 읽고 쓰기 가능한 옵션인 PAGE_EXECUTE_READWRITE 정도만 알아두면 되고 자세한 것은 MSDN을 참고한다.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_protection_constants.asp

```
BOOL VirtualFreeEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD dwFreeType
);
```

hProcess : [in], 동일
lpAddress : [in], 동일, dwFreeType이 MEM_RELEASE일 경우 VirtualAllocEx API에 의해 예약된 메모리 영역의 시작 주소를 가리켜야 한다.
dwSize : [in], dwFreeType이 MEM_RELEASE일 경우 반드시 0 이어야 한다.
dwFreeType : [in], MEM_RELEASE와 MEM_DECOMMIT의 두 가지 타입이 있다.

이제 필요한 기본 지식은 다 알아보았다. 이제 실제 코드를 살펴보도록 하자.

그 전에 참고로 아래와 같은 경우가 발생할 수 있다.

```
방법 :
1. 코드:
OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&hToken);
LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &Val);
tp.PrivilegeCount = 1;
tp.Privileges[0].Luid = Val;
tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
CloseHandle(hToken);

로부터 제 프로그램에게 DebugPrivilege 부여
```

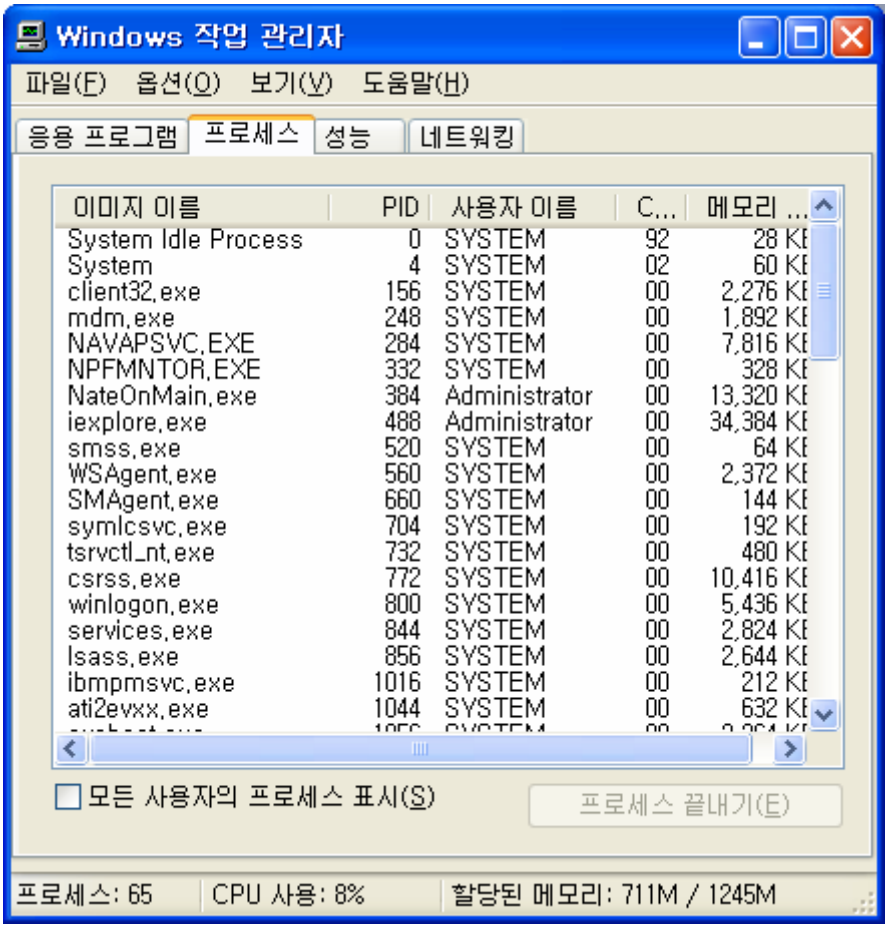
2. 코드:
hTarget = FindWindow("목표프로세스Class", NULL);
GetWindowThreadProcessId(hTarget, &processId);
hProcess = OpenProcess(PROCESS_ALL_ACCESS, TRUE, dwProcessId);

로부터 목표 프로세스의 핸들값 얻기 시도

문제점 :
물론 컴파일과 빌딩 시 에러 없이 실행파일이 만들어진 상태에서
spy++을 이용해 목표프로세스의 클래스와 processId가 정확함을 확인한 후
위 1,2번 시퀀스를 돌린 결과 원하는 프로세스의 핸들값이 NULL이 나옵니다. GetLastError()를 통한 에러 원인은 5번, 즉 ACCESS_DENIED 이구요.

질문 :
관리자 계정으로 로긴했고 제 프로세스에 DebugPrivilege를 줬음에도 OpenProcess()로 그 핸들 값을 넘겨받을 수 없는 일반 응용프로그램의 성립이 어떻게 가능한지 여쭙고 싶습니다. 솔직히 제 짧은 지식으로는 이 접근하지 못할 프로세스의 성격조차도 규정하기 힘듭니다. ^^;
출처) http://www.jiniya.net/bbs/viewtopic.php?p=1232

제프리 리쳐의 문서에 따르면 OpenProcess API에 PROCESS_ALL_ACCESS 플래그를 매개변수로 넘겨주었을 경우 일부 작은 ID의 Process경우 NULL을 반환하기도 한다고 한다. 여기서 작은 ID의 Process란 위에서 기술한 시스템 프로세스를 의미하는 것이라고 생각할 수 있다. 왜냐? 시스템 프로세스의 경우 일반적으로 실행 순서가 빠르므로 프로세스의 ID 역시 작을 수 밖에 없다. ^^



어쨌든 이때에는 PROCESS_ALL_ACCESS 플래그 대신 다른 플래그를 변수로 넘겨주어야 하는데 그 결과는 알 수 없다. -_-;

소스 중 가장 중요한 부분인 InjectLib 함수와 EjectLib 함수를 살펴보자.

-InjectLib-

```
BOOL WINAPI CCreateRemoteProcessDlg::InjectLib(DWORD dwProcessId, LPCTSTR szLibFile)
{
  BOOL fOk = FALSE;
  HANDLE hProcess = NULL, hThread = NULL;
  szLibFileRemote = NULL;

  __try
  {
    // 타겟 프로세스의 핸들을 얻어 옴
    hProcess = OpenProcess(
```

```
      PROCESS_QUERY_INFORMATION |  // 프로세스의 정보 요청
      PROCESS_CREATE_THREAD     |  // CreateRemoteThread를 사용하기 위해
      PROCESS_VM_OPERATION      |  // VirtualAllocEx/VirtualFreeEx를 사용하기 위해
      PROCESS_VM_WRITE,            // WriteProcessMemory를 사용하기 위해
      FALSE, dwProcessId);


    if(hProcess == NULL)
    {
      AfxMessageBox("OpenProcess Failed : Line 195");
      __leave;
    }

    // DLL 파일 전체경로의 바이트 수 계산
    int cch = 1 + strlen(szLibFile);
    int cb  = cch * sizeof(TCHAR);

    // 타겟 프로세스에 DLL 파일경로의 크기만큼의 공간 할당
    szLibFileRemote = (TCHAR *)VirtualAllocEx(hProcess, NULL, cb, MEM_COMMIT,
PAGE_READWRITE);

    if(szLibFileRemote == NULL)
    {
      AfxMessageBox("VirtualAllocEx Failed : Line 213");
      __leave;
    }

    // 확보한 타겟 프로세스의 공간에 DLL 경로 복사
    if (!WriteProcessMemory(hProcess, szLibFileRemote, (LPVOID)szLibFile, cb, NULL))
    {
      AfxMessageBox("WriteProcessMemory Failed : Line 221");
      __leave;
    }

    // LoadLibraryA의 주소를 얻어 옴
    PTHREAD_START_ROUTINE pfnThreadRtn =
(PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
```

```
"LoadLibraryA");

    if(pfnThreadRtn == NULL)
    {
      AfxMessageBox("GetProcessAddress Failed : Line 228");
      __leave;
    }

    // 타겟 프로세스에 DLL을 로드하는 스레드 생성
    hThread = CreateRemoteThread(hProcess, NULL, 0, pfnThreadRtn, szLibFileRemote, 0,
NULL);

    if(hThread == NULL)
    {
      AfxMessageBox("CreateRemoteThread Failed : Line 236");
      __leave;
    }

    // CreateRemoteThread에 의해 DLL이 성공적으로 로드된 후 DLL 상의 코드가 실행됨
    // DLL에서 실행된 프로그램이 끝날때까지 기다림
    WaitForSingleObject(hThread, INFINITE);

    fOk = TRUE; // Success InjectLib
    }

    __finally {
      if(szLibFileRemote != NULL)
        VirtualFreeEx(hProcess, szLibFileRemote, 0, MEM_RELEASE);

      if (hThread  != NULL)
        CloseHandle(hThread);

      if (hProcess != NULL)
        CloseHandle(hProcess);
    }
    return(fOk);
} // end of InjecLib()
```

InjectLib는 인젝션 버튼을 눌렀을 때 시작되는 함수이다.
먼저 타겟 프로세스를 OpenProcess API를 이용하여 연 다음 VirtualAllocEx와 WriteProcessMemory API를 이용하여 타겟 프로세스에 dll 이름을 복사한다. 그리고 GetProcAddress로 LoadLibraryA의 주소를 얻어온 다음 CreateRemoteThread API로 dll을 인젝션한다. __try{}의 마지막에 도달한 이후 __finally{}로 진입하면서 VirtualFreeEX와 CloseHandle API를 사용하여 뒷처리를 한다.

여기서 사용된 __try {} ~ __finally{}와 __leave는 눈에 익으면서도 조금 생소할텐데 __try, __finally, __leave는 C++의 예외처리 지시자인 try, catch, throw 의 VC 버전이라고 생각하면 된다. 이를 일러 SEH(Structured Error Handling)라고 하는데 C++의 예외처리 지시자와 서로 다른 장단점을 가지고 있다. 결론적으로 말하자면 만약 비 MFC를 사용한다면 SEH 를 사용할 것을 권장한다. 즉 가급적이면 C++의 예외처리를 사용하라는 소리다.
자세한 것은 MSDN에서 살펴볼 수 있으니 참고하도록 한다.
(http://msdn.microsoft.com/library/kor/default.asp?url=/library/KOR/vccore/html/_core_exception_handling_topics_.28.general.29.asp)

다음으로 OpenProcess API를 살펴보자. 해당 API는 워낙 잘 알려진 API지만 다시 공부해보는 차원에서 정리해보았다.

함수의 원형은 아래와 같다.

```
HANDLE WINAPI OpenProcess(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  DWORD dwProcessId
);
```

dwDesiredAccess : [in], 타겟 프로세스로의 접근 권한을 지정한다. 이 접근 권한은 프로세스를 위해 security descriptor와 조합되며 1개 이상일 수 있다. 만약 호출자가 SeDebugPrivilege 권한을 요청했다면 해당 호출은 security descriptor의 내용에 상관없이 접근 권한을 주게 된다.
조합 가능한 값들은 아래 표와 같다.
bInheritHandle : [in], 이 값이 TRUE라면 해당 프로세스에 의해 만들어진 프로세스는 핸들을 상속받게 되며 반대의 경우 핸들을 상속받지 못하게 된다.
dwProcessId : [in], 타겟 프로세스의 아이디

조합 가능한 권한 값들의 종류

Value	설명
PROCESS_ALL_ACCESS (0x1F0FFF)	모든 권한
PROCESS_CREATE_PROCESS (0x0080)	프로세스를 생성
PROCESS_CREATE_THREAD (0x0002)	스레드를 생성
PROCESS_DUP_HANDLE (0x0040)	DuplicateHandle API를 이용하여 핸들을 복제
PROCESS_QUERY_INFORMATION (0x0400)	token, exit code, priority class와 같은 프로세스의 특정 정보를 회수
PROCESS_QUERY_LIMITED_INFORMATION (0x1000)	프로세스의 제한된 형태의 특정 정보를 회수
PROCESS_SET_QUOTA (0x0100)	SetProcessWorkingSetSizef 를 사용하여 메모리 한계를 설정
PROCESS_SET_INFORMATION (0x0200)	priority class와 같은 프로세스의 특정 정보를 설정
PROCESS_SUSPEND_RESUME (0x0800)	프로세스를 suspend 또는 resume
PROCESS_TERMINATE (0x0001)	TerminateProcess 를 사용하여 프로세스를 종료
PROCESS_VM_OPERATION (0x0008)	프로세스의 주소 공간 안에서의 조작을 허용 (VirtualProtectEx, WriteProcessMemory 사용 시 쓰임)
PROCESS_VM_READ (0x0010)	ReadProcessMemory를 사용하여 프로세스 내의 메모리를 읽음
PROCESS_VM_WRITE (0x0020)	WriteProcessMemory를 사용하여 프로세스 내의 메모리에 씀
SYNCHRONIZE (0x00100000L)	wait 기능을 이용하여 종료시킬 프로세스를 wait

코드에서 사용된 값들이 쓰인 이유는 주석을 달아놨으니 참고하도록 한다.

이제 EjectLib를 살펴보자.

```
BOOL WINAPI CCreateRemoteProcessDlg::EjectLib(DWORD dwProcessId, LPCTSTR szLibFile)
{
  BOOL fOk = FALSE;
  HANDLE hthSnapshot = NULL;
```

```
HANDLE hProcess = NULL, hThread = NULL;

__try {
    // 타겟 프로세스의 스냅샷을 가져옴(모듈)
    hthSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwProcessId);

    if (hthSnapshot == NULL)
    {
      AfxMessageBox("CreateToolhelp32Snapshot Failed : Line 292");
      __leave;
    }

    // 모듈을 열거
    MODULEENTRY32 me;
    me.dwSize = sizeof(me);
    BOOL fFound = FALSE;
    BOOL fMoreMods = Module32First(hthSnapshot, &me);

    for (; fMoreMods; fMoreMods = Module32Next(hthSnapshot, &me))
    {
      fFound = ( lstrcmpi(me.szModule, szLibFile) == 0) ||
                (lstrcmpi(me.szExePath, szLibFile) == 0 );

        if (fFound)
          break;
      }

      if (!fFound)
          __leave;

      // 타겟 프로세스의 핸들을 얻어 옴
      hProcess = OpenProcess(
              PROCESS_QUERY_INFORMATION |   // Required by Alpha
              PROCESS_CREATE_THREAD     |
              PROCESS_VM_OPERATION,  // For CreateRemoteThread
              FALSE, dwProcessId);
      if (hProcess == NULL)
```

```
            __leave;

        // Kernel32.dll로부터 FreeLibraryA의 주소를 얻어 옴
        PTHREAD_START_ROUTINE pfnThreadRtn =
        (PTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(TEXT("Kernel32")),
        "FreeLibraryA");

        if (pfnThreadRtn == NULL)
            __leave;

        // 타겟 프로세스에 DLL을 로드하는 스레드 생성
       hThread = CreateRemoteThread(hProcess, NULL, 0, pfnThreadRtn, me.modBaseAddr, 0,
                                    NULL);

        if (hThread == NULL)
            __leave;

        // CreateRemoteThread에 의해 DLL이 성공적으로 로드된 후 DLL 상의 코드가 실행됨
        // DLL에서 실행된 프로그램이 끝날 때 까지 기다림
        WaitForSingleObject(hThread, INFINITE);

        fOk = TRUE; // success EjectLib
    }
    __finally { // 뒷정리 시작

        if (hthSnapshot != NULL)
          CloseHandle(hthSnapshot);

        if (hThread     != NULL)
          CloseHandle(hThread);

        if (hProcess    != NULL)
          CloseHandle(hProcess);
    }
    return(fOk);
} // end of EjectLib
```

__try{}에 들어가자마자 CreateToolhelp32Snapshot 라는 생소한 API가 눈에 띈다.
해당 API는 특정 프로세스의 스냅샷(힙, 모듈, 스레드)을 가져오는 역할을 하며 원형은 아래와 같다.

```
HANDLE WINAPI CreateToolhelp32Snapshot(
  DWORD dwFlags,
  DWORD th32ProcessID
);
```

dwFlags : [in], 스냅샷에 포함된 시스템의 일부분이며 값의 종류는 아래 표와 같다.
th32ProcessID : [in], 스냅샷에 포함된 프로세스의 식별자. 이 값은 dwFlags 값이 TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPALL가 지정되었을 때 사용되며 나머지의 경우는 무시되며 모든 프로세스들이 스냅샷에 포함된다.

Value	설명
TH32CS_INHERIT	스냅샷의 핸들이 상속됨
TH32CS_SNAPALL	시스템의 모든 프로세스와 스레드를 포함하며 th32ProcessID에서 지정된 프로세스의 힙과 모듈을 스냅샷에 포함
TH32CS_SNAPHEAPLIST	th32ProcessID에서 지정된 프로세스의 모든 힙을 스냅샷에 포함.
TH32CS_SNAPMODULE	th32ProcessID에서 지정된 프로세스의 모든 모듈을 스냅샷에 포함
TH32CS_SNAPPROCESS	시스템의 모든 프로세스를 스냅샷에 포함
TH32CS_SNAPTHREAD	시스템의 모든 스레드를 스냅샷에 포함 지정된 프로세스 안에 포함된 스레드를 식별하기 위해 스레드를 열거할 때 THREADENTRY32 구조체의 th32OwnerProcessID 멤버의 프로세스 식별자와 비교

CreateToolhelp32Snapshot 을 이용하여 타겟 프로세스의 모듈 정보(ModuleEntry32 구조체)를 스냅샷에 포함시킨 다음 모든 모듈을 나열한다. 모듈을 나열하기 위해 쓰이는 API가 Module32First와 Module32Next API 이며 원형은 아래와 같다.

```
BOOL WINAPI Module32First(
  HANDLE hSnapshot,
  LPMODULEENTRY32 lpme
);
```

hSnapshot : [in], CreateToolhelp32Snapshot API에 의해 리턴된 스냅샷의 핸들
lpme : [in, out], MODULEENTRY32 구조체를 가리키는 포인터

```
BOOL WINAPI Module32Next(
  HANDLE hSnapshot,
  LPMODULEENTRY32 lpme
);
```

```
hSnapshot : [in], 동일
lpme : [out], 동일
```

ModuleEntry32 구조체에 대한 자세한 정보는 MSDN을 참고한다.
(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/moduleentry32_str.asp)

일반적으로 프로세스, 모듈, 스레드를 나열하는 방법은 거의 동일하게 쓰인다.
아래에서 매우 잘 설명된(한글 -_-b) WIN32 API를 사용한 프로세스 열거 예제를 확인할 수 있다.

- 프로세스 열거 예제 -
http://support.microsoft.com/default.aspx?scid=kb%3Bko%3B175030

- 모듈 열거 예제 -
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/traversing_the_module_list.asp

모듈을 열거하면서 lstrcmpi API를 이용해 모듈의 이름과 exe 파일의 절대경로를 비교하여 타겟 프로세스의 모듈을 찾는다. lstrcmpi는 대소문자를 비교하지 않을 뿐 lstrcmp와 완전히 동일하며 모듈을 찾는 이유는 CreateRemoteThread API를 호출할 때 다섯번 째 인수로 모듈이 시작되는 베이스 주소의 사이즈를 넘기기 위해서인데 왜 이 값을 넘기는 지 솔직히 모르겠다. 아마도 모듈의 베이스 주소를 나타내는 공간의 사이즈와 스레드가 필요로 하는 공간의 사이즈가 동일하기 때문일 것이라고 추측은 되는데 관련 자료를 찾지 못하여 장담할 수 없다.
이후 InjectLib와 마찬가지로 CreateRemoteThread API가 끝날 때까지 WaitForSingleObject API로 기다린 후 __finally{} 로 들어가 뒷정리를 하게 된다.

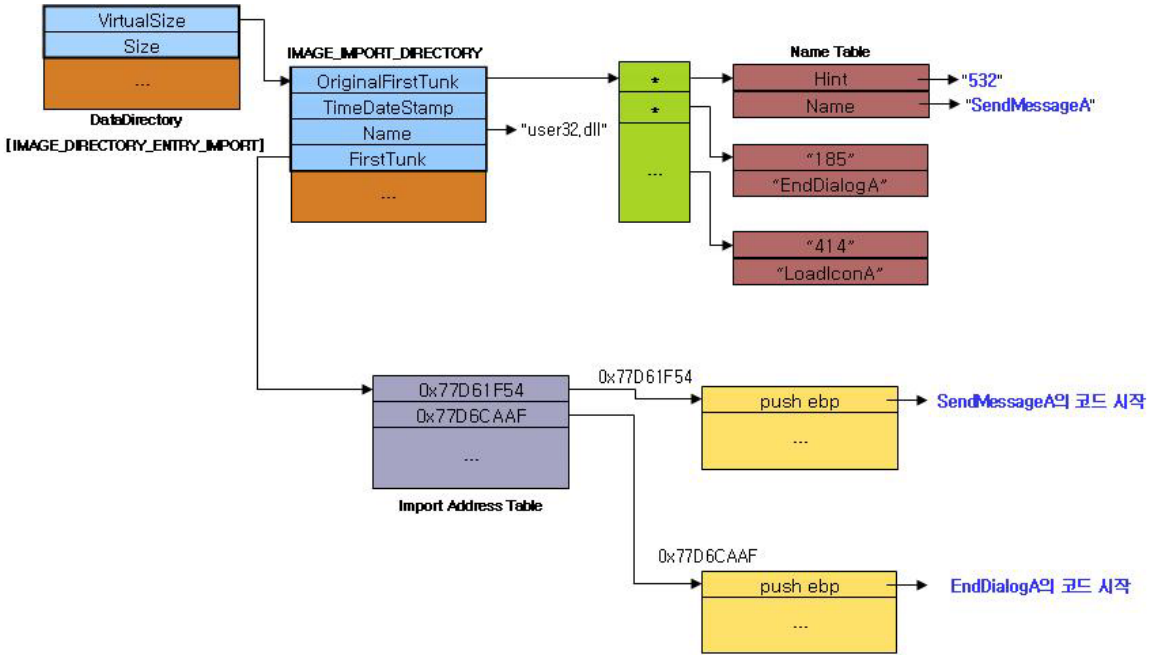
소스 프로그램은 아래에서 받을 수 있다.
http://backrush.com/~lucid7/WindowsHook.zip

※ 해당 프로그램은 실행 중인 프로세스에 접근하는 것이기 때문에 프로그램을 종료하면 프로세스도 같이 종료된다. CreateProcess를 사용하여 프로세스를 시작시킴과 동시에 Dll Injection을 시도하면 문제를 해결할 수 있다.

2. 추가

이동우 님의 “CreateRemoteThread API를 이용한 DLL Injection” 문서에 나와있는 릴리즈 모드로 컴파일 해야 하는 이유를 허락 없이 고대로 베껴왔다. -_-;; 이동우님의 선처를 바랄 뿐이다 ㅠ_-

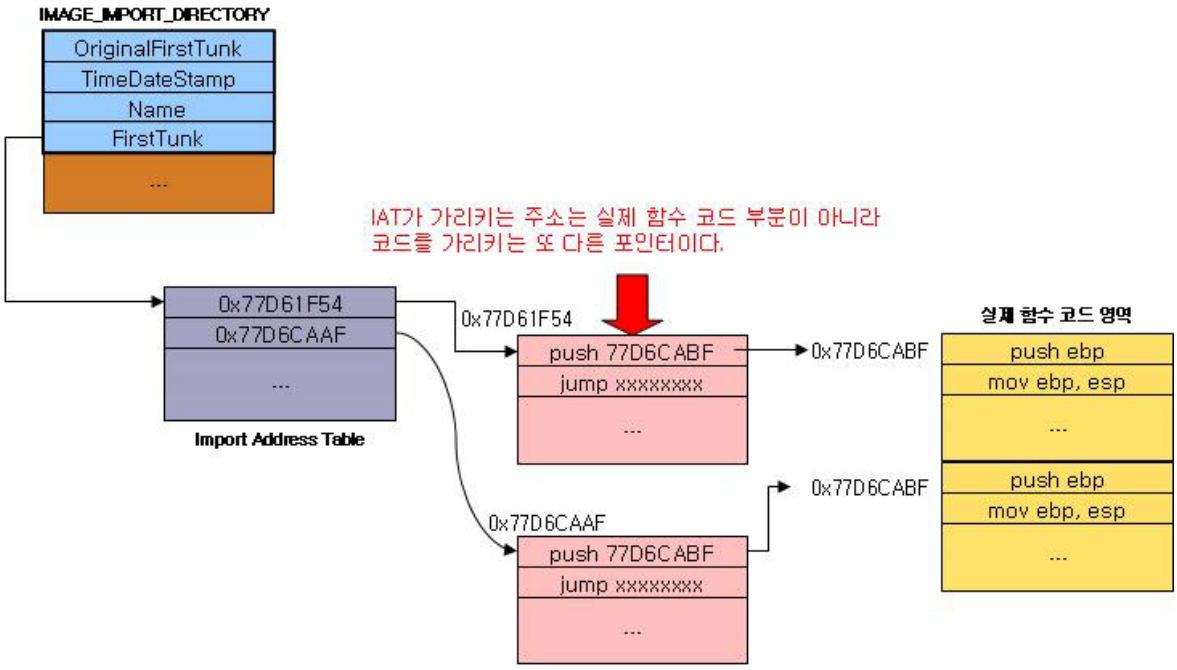
Debug version에서 LoadLibrary를 Call할 때 죽는 이유
디버그 모드로 빌드하면 .idata의 stub들이 DLL내의 함수로 연결되는 매커니즘이 약간 다르기 때문이다. 아래는 릴리즈 모드로 빌드했을 때의 DLL 함수 Call 구조를 도식화 한 그림이다.



Release version일 때 DLL 함수 Call 도식도

위 그림에서 보는 바와 같이 IAT(Import Address Table)가 가리키는 함수 포인트에서 바로 DLL 함수의 시작 코드가 자리잡고 있다.

다음은 디버그 모드로 빌드했을 때의 DLL 함수 Call 구조를 도식화 한 그림이다.



위 그림의 빨간색 화살표가 가리키듯이 IAT가 포인트 하는 주소에 바로 DLL 함수가 시작하지 않고 또 다른 포인터 주소를 가리키게 된다. 따라서 IAT가 가리키는 주소를 바로 가져와서 거기에 stub을 덮어 써 버리면 exception 에러가 발생하게 된다.

3. 참고문서

WebSite

1. api hooking revealed
http://www.codeproject.com/system/hooksys.asp

2. 동우의 홈페이지
http://dasomnetwork.com/~leedw/mywiki/moin.cgi/

3. Win32 API 연구 사이트
http://www.winapi.co.kr/

Documents

1. Programming Applications Microsoft Windows 4th Edition, Jeffrey Richer

4. 후기

이직 및 기타 사생활 관련(-_-;)한 본인의 개인적인 문제로 인해 해당 문서는 절반 정도 쓰여진 채 약 3개월 간 올스톱의 상태에 있었다. 이제야 겨우 문서를 완성하게 되어 기쁘기 그지 없다. 비록 별 볼일 없는 난이도의 문서지만 그래도 나름대로 완성했다는 데에 큰 비중을 둔다. 이런 문서라도 도움이 되는 사람이 반드시(-_-;) 있으리라 믿어 의심치 않는다.(__)

이번 장에서는 코드의 설명에 대부분의 중점을 두었다. 앞의 두 문서에 비해 코드의 난이도가 비교적 어렵고(적어도 나에게는 그랬다 -_-;) 난해했기 때문이다.
비교적 쉽게 설명하려고 했지만 생소한 API를 좀 많이 다루어서 그런지 그냥 MSDN을 갖다 붙여놓은 것 같은 느낌이 없지 않아 있지만 어쩌랴. 본인의 실력이 이것 밖에 안 되는 것을. 죄송한 마음을 감출 길이 없다;;;

앞으로 쓰여지게 될 문서들의 진도는 다음과 같이 예상하고 있다.
IAT Patching –> EAT Patching
여기까지 이르게 되면 User Mode에서의 Hooking을 다 다루게 된다.(devpia에 보면 dynamic api hooking이라는 문서도 있는데 귀찮아서 그냥 생략하기로 했다. 다른 방법, 다른 코드들이 분명히 존재할 것이라고 생각하고 있으나 역시 귀찮다 -__-)

이후에는 Kernel Mode에서의 Hooking을 다루게 될 것이다. 때문에 필히 Device Driver를 다루어야 한다. 본인은 지금 Device Driver에 흥미를 느끼고 책을 보고 있는 중이다.(아직까지 책 보면서 토할 것 같은 느낌을 받는 경지이다. -_-;)
가급적이면 연말부터는 Device Driver 관련 문서들을 생산(?)할 예정이지만 언제나, 또 누구에게나 그렇듯이 예정은 예정이다 -__-