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 2003에서 테스트 되었습니다. 문서의 내용 중 틀린 곳이나 수정해야 할 부분이 있으면 연락해 주시기 바랍니다.

목 차

1. 사용되는 기법들

이번에는 Debuging Technique을 이용해 Hook을 하는 방법을 다루어본다. 이 기술은 타겟 프로세스를 디버그 모드로 접근한 뒤 Dll Injection으로도 Api Hooking으로도 사용할 수 있으나 난이도에 비해 결과가 썩 훌륭하지는 않다.
무엇보다도 디버깅을 중지하면 타겟 프로세스까지 중지되므로 실제 공격용으로는 사용하기가 벅차고(멀티 스레딩을 통한 방법이 있다고는 하는데..글쎄 본인의 실력으로는 구현 불가능이다. -___-) 그냥 원리를 이해한다는 측면에서 정리해 보았다.

1.1. 디버그 모드로 해당 프로세스의 주소 공간으로 침투하기

Windows는 두 가지 종류의 디버거를 제공하는데 사용자 모드 디버거와 커널모드 디버거가 바로 그것이다. 사용자 모드 디버거의 경우 응용프로그램을 디버깅하기 위해, 커널 모드 디버거의 경우는 운영체제의 커널을 디버깅 하기 위해 사용되며 일반적으로 디바이스 드라이버 개발자들이 대부분 사용한다. 사용자 모드 디버거의 특징은 Win32 Debugging API를 사용한다는 점이며 우리는 이 API를 이용해서 타겟 프로세스의 주소 공간으로 침투할 것이다.

사용자 모드 디버거는 타겟 프로세스를 디버깅하기 위해 CreateProcess API를 이용하는데, 함수의 전달인자 중 dwCreationFlags 변수에 DEBUG_ONLY_THIS_PROCESS 값을 사용하여 디버깅을 시작한다. 그리고 발생하는 다양한 이벤트들을 받기 위해 WaitForDebugEvent API를 호출하고 특정한 이벤트를 처리한 후에는 ContinueDebugEvent 함수를 호출한다.
이 방법의 문제점은 디버거가 실행되는 동안 타겟 프로세스도 멈춰있다는 것인데 이 문제는 멀티 스레드를 사용해서 해결할 수 있다.

WaitForDebugEvent API가 호출되었을 때 전달되는 이벤트 정보를 저장하는 DEBUG_EVENT 구조체는 다음과 같다.

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;

  union {
    EXCEPTION_DEBUG_INFO Exception;
    CREATE_THREAD_DEBUG_INFO CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO ExitThread;
```

```
    EXIT_PROCESS_DEBUG_INFO ExitProcess;
    LOAD_DLL_DEBUG_INFO LoadDll;
    UNLOAD_DLL_DEBUG_INFO UnloadDll;
    OUTPUT_DEBUG_STRING_INFO DebugString;
    RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT;
```

디버그가 위 구조체의 유니온 안에 나열된 이벤트를 처리하는 동안 타겟 프로세스는 멈춰있게 되며 ContinueDebugEvent 함수가 호출되면 타겟 프로세스는 다시 실행이 된다.

일단 여기까지의 내용을 의사코드를 통해 살펴보자.

```
void _tmain(int argc, TCHAR *argv[])
{
  CreateProcess( NULL,                        // No module name (use command line).
                 strProgramPath,              // Command line.
                 NULL,                        // Process handle not inheritable.
                 NULL,                        // Thread handle not inheritable.
                 FALSE,                       // Set handle inheritance to FALSE.
                 CREATE_NEW_CONSOLE |
                 DEBUG_ONLY_THIS_PROCESS , // No creation flags.
                 NULL,                        // Use parent's environment block.
                 NULL,                        // Use parent's starting directory.
                 &si,                         // Pointer to STARTUPINFO structure.
                 &pi ) ;                      // Pointer to PROCESS_INFORMATION structure.

  while(WaitForDebugEvent(lpDebugEvent, 1000) == 1)
  {
    if(EXIT_PROCESS) break;
    else {
      Do..something...;
    }
    ContinueDebugEvent(dwProcessId, dwThreadId, dwContinueStatus);
  }
}
```

먼저 strProgramPath 인자로 넘어온 프로그램을 CreateProcess 함수를 이용하여 디버그 모드로 디버깅을 시작한 다음 WaitForDebugEvent 함수로 특정 이벤트를 기다린다. 특정 이벤트에 대한 처리를 해준 뒤 ContinueDebugEvent 함수로 진행을 계속한다.

발생하는 디버깅 이벤트는 아래의 표와 같다.

디버깅 이벤트	설명
CREATE_PROCESS_DEBUG_EVENT	새로운 프로세스가 디버깅되는 프로세스 내에서 생성될 때나 디버거가 이미 활성화 되어 있는 프로세스를 디버깅할 때마다 생성된다.
CREATE_THREAD_DEBUG_EVENT	디버깅되고 있는 프로세스에서 새로운 스레드가 생성되거나 디버거가 이미 활성화되어 있는 프로세스에 대한 디버깅을 시작할 때마다 발생하며 새로운 스레드가 사용자 모드에서 실행되기 전에 생성된다.
EXCEPTION_DEBUG_EVENT	디버깅되고 있는 프로세스에서 예외가 발생할 때마다 생성된다. 액세스가 불가능한 메모리에 대한 액세스 시도, 브레이크 포인트 명령 실행, 0으로 나눔 명령 실행 또는 MSDN의 “Structured Exception Handling”에서 설명하고 있는 다른 예외들을 포함한다.
EXIT_PROCESS_DEBUG_EVENT	프로세스에 있는 마지막 스레드가 디버거를 나갈 때나 ExitProcess 함수를 호출할 때마다 생성된다. 이 이벤트는 커널이 프로세스의 DLL을 언로드하고 프로세스의 종료 코드를 업데이트한 후에 곧바로 발생한다.
EXIT_THREAD_DEBUG_EVENT	디버깅되고 있는 프로세스의 일부인 스레드가 종료할 때마다 생성된다. 커널은 스레드의 종료코드를 업데이트하자마자 이 이벤트를 생성한다.
LOAD_DLL_DEBUG_EVENT	디버깅되는 프로세스가 DLL을 로드할 때마다 생성된다. 이 이벤트는 시스템 로더가 DLL을 연결하거나 디버깅되는 프로세스가 LoadLibrary 함수를 사용할 때 발생하며 DLL이 해당 주소 공간에 로드될 때마다 호출된다.
OUTPUT_DEBUG_STRING_EVENT	디버깅되는 프로세스가 OutputDebugString 함수를 호출할 때 생성된다.
UNLOAD_DLL_DEBUG_EVENT	디버깅되는 프로세스가 FreeLibrary 함수를 사용하여 DLL을 언로드 할 때마다 생성되며 DLL이 프로세스의 주소공간으로부터 마지막으로 언로드 될 때만 발생한다.

WaitForDebugEvent API에서 반환되는 디버그 이벤트를 처리하고 있는 동안에 타겟 프로세스

의 주소 공간으로부터 읽거나 쓰기 작업을 할 수 있다. 디버깅 당하는 타겟 프로세스의 메모리로부터 데이터를 읽기 위해서 ReadProcessMemory API를 호출한다.

ReadProcessMemory API의 원형은 아래와 같다.

```
BOOL ReadProcessMemory(
  HANDLE hProcess,
  LPCVOID lpBaseAddress,
  LPVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesRead
);
```

hProcess : [in], 메모리를 읽어들일 타겟 프로세스의 핸들, PROCESS_VM_READ 권한이 있어야 한다.
lpBaseAddress : [in], 타겟 프로세스의 Base Address
lpBuffer : [out], 읽어들인 메모리를 저장할 버퍼의 포인터
nSize : [in], 메모리로부터 읽어들일 바이트의 갯수
lpNumberOfBytesRead : [out], 버퍼에 저장된 바이트의 개수

메모리에 직접 쓰는 API는 WriteProcessMemory이며 API의 원형은 아래와 같다.

```
BOOL WriteProcessMemory(
  HANDLE hProcess,
  LPVOID lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T nSize,
  SIZE_T* lpNumberOfBytesWritten
);
```

hProcess : [in], 수정할 메모리 프로세스의 핸들, PROCESS_VM_WRITE 권한과
 PROCESS_VM_OPERATION 권한이 있어야 한다.
lpBaseAddress : [in], 타겟 프로세스의 Base Address
lpBuffer : [in], 타겟 프로세스의 메모리에 쓸 데이터를 가지고 있는 버퍼의 포인터
nSize : [in], 메모리에 쓰여질 바이트의 개수
lpNumberOfBytesWritten : [out], 메모리에 쓰여진 바이트의 개수

일반적으로 WriteProcessMemony API를 이용하여 메모리에 직접 브레이크 포인트를 삽입한 뒤 발생하는 EXCEPTION_DEBUG_EVENT를 잡아채서 이후의 작업을 하게 된다. 브레이크 포인트를 설정하기 위해서는 브레이크 포인트를 설정하고자 하는 메모리 주소를 갖고 해당 위치의 opcode를 저장한 후 해당 주소에 브레이크 포인트 명령을 쓰면 되며 Intel Pentium 계열에서 브레이크 포인트를 뜻하는 명령은 INT 3 또는 opcode 값 0xCC이다. 브레이크 포인트를 설정해서 원하는 작업을 수행한 후에는 원래의 opcode를 다시 원상태로 복원시켜 주어야 한다.

브레이크 포인트 설정 후 특정 DLL을 삽입하거나 또는 특정 DLL로부터 발생하는 특정 API를 Hook 할 수 있다. 전자는 DLL Injection이 될 것이고 후자는 API Hook이 될 것이다.

1.2. Debug API를 이용한 DLL Injection

먼저 Dll Injection을 알아보자.

① 대상 프로그램을 디버그 모드로 실행한다.
② EXCEPTION_DEBUG_EVENT의 EXCEPTION_BREAKPOINT가 발생하면(디버그 모드로 시작하자마자 발생하는 첫 번째 예외) 타겟 프로세스의 컨텍스트와 코드섹션의 첫 페이지를 저장한다.
③ 저장한 영역에 LoadLibrary로 특정 dll을 로드하는 코드와 0xcc를 삽입한다.
④ EIP에 실행시킬 코드의 주소를 삽입한다.
⑤ ContinueDebugEvent를 실행한다.
⑥ 삽입한 LoadLibrary 실행 후 브레이크 포인트가 발생하면 저장했던 컨텍스트와 코드섹션의 첫 페이지, 그리고 EIP를 원상태로 복귀한다.
⑦ ContinueDebugEvent를 실행한다.

주의할 것은 디버그 모드로 들어가자마자 첫 번째 EXCEPTION_BREAKPOINT가 발생한다는 것이며 이 때 타겟 프로세스는 실행되기 전이다. 코드를 삽입하는 시기가 바로 이 때이며 두 번째 예외 발생 시에는 원래 코드를 복원시켜 주어야 한다.

②번에서 첫 번째 코드 페이지를 가져오는 코드는 아래와 같다.

```
(전역 변수)
CProcessInfo = DebugEvent.u.CreateProcessInfo;
pProcessBase = CProcessInfo.lpBaseOfImage;

LPVOID CKeyboardHookDlg::GetFirstCodePage(HANDLE hProcess, PVOID pProcessBase)
{
    DWORD baseOfCode;
    DWORD peHdrOffset;
    DWORD dwRead;
    BOOL bRet;

    // IMAGE_NT_HEADER의 시작 오프셋 값을 읽어옴
    bRet = ReadProcessMemory(hProcess, (PBYTE)pProcessBase + offsetof(IMAGE_DOS_HEADER,
e_lfanew),&peHdrOffset, sizeof(peHdrOffset), &dwRead);

    if (!bRet || sizeof(peHdrOffset) != dwRead) {
```

```
        OutputDebugString("ERROR");
    }

    // Read in the IMAGE_NT_HEADERS.OptionalHeader.BaseOfCode field
    bRet = ReadProcessMemory(hProcess, (PBYTE)pProcessBase + peHdrOffset + 4 +
IMAGE_SIZEOF_FILE_HEADER + offsetof(IMAGE_OPTIONAL_HEADER, BaseOfCode), &baseOfCode,
sizeof(baseOfCode), &dwRead);

    if(!bRet || sizeof(baseOfCode) != dwRead) {
        OutputDebugString("ERROR");
    }
    return (LPVOID)((DWORD)pProcessBase + baseOfCode);   // 첫번째 섹션코드의 RVA
}
```

코드를 자세히 살펴보자.

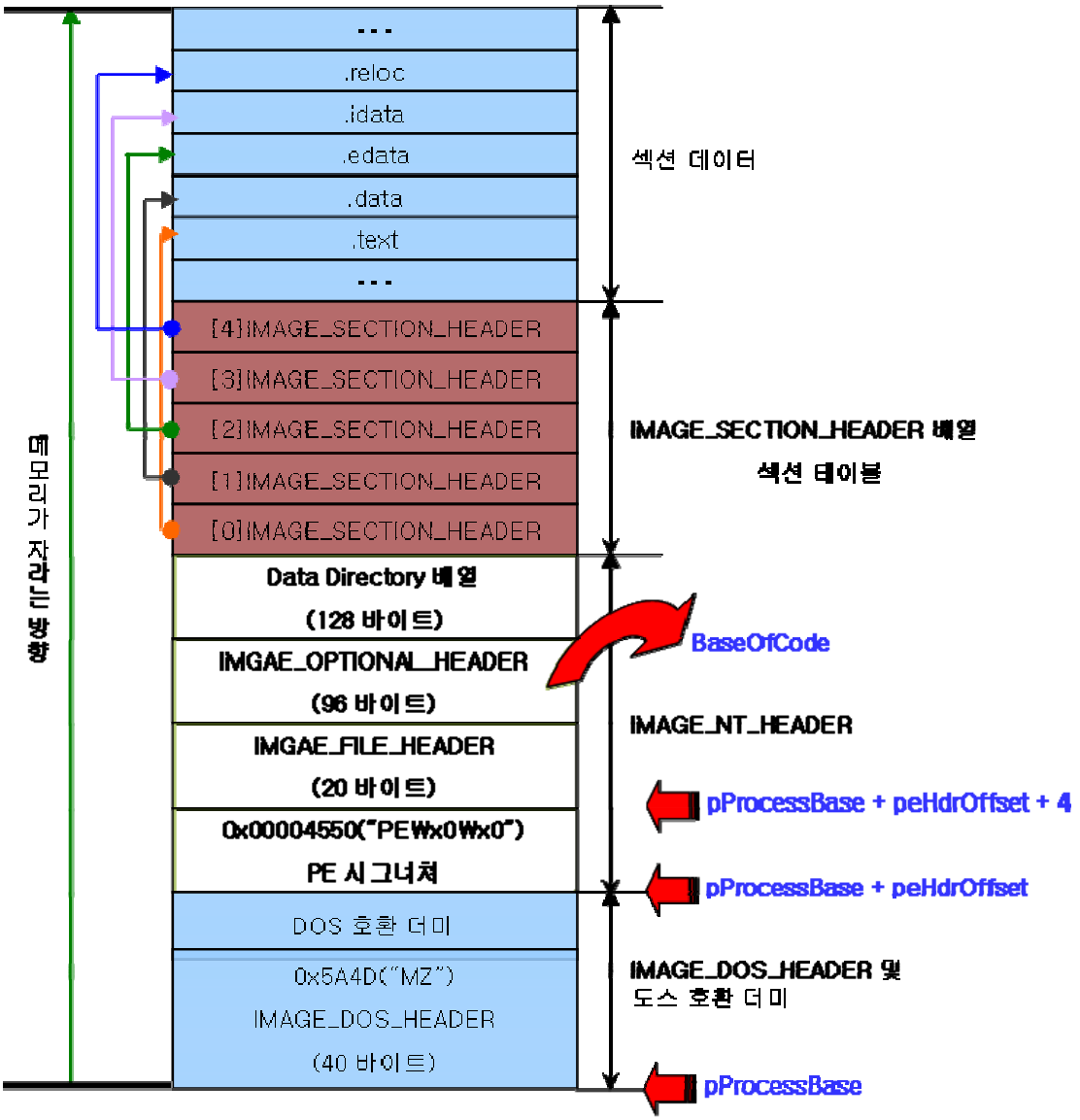
```
bRet = ReadProcessMemory(hProcess, (PBYTE)pProcessBase + offsetof(IMAGE_DOS_HEADER,
e_lfanew),&peHdrOffset, sizeof(peHdrOffset), &dwRead);
```

파일의 선두로부터 IMAGE_DOS_HEADER 구조체의 e_lfanew 값을 읽어 들여 이 값을 peHdrOffset 변수에 저장시킨다. e_lfanew에는 IMAGE_NT_HEADER 구조체의 시작 오프셋 값이 저장되어 있다.

```
bRet = ReadProcessMemory(hProcess, (PBYTE)pProcessBase + peHdrOffset + 4 +
IMAGE_SIZEOF_FILE_HEADER + offsetof(IMAGE_OPTIONAL_HEADER, BaseOfCode), &baseOfCode,
sizeof(baseOfCode), &dwRead);
```

파일의 선두로부터 IMAGE_NT_HEADER 구조체로 접근한 다음(peHdrOffset) 4바이트를 더하고(PE₩x0₩x0), IMAGE_SIZEOF_FILE_HEADER의 값을 더한 뒤(IMAGE_FILE_HEADER 구조체 다음에 IMAGE_OPTINAL_HEADER 구조체가 위치하고 있다.) IMAGE_OPTINAL_HEADER 구조체의 BaseOfCode 값을 읽어 들여 이 값을 baseOfCode 변수에 저장한다. BaseOfCode에는 코드 영역의 시작 주소가 저장되어 있다.(물론 첫번째 코드 섹션이 시작되는 RVA를 의미한다.)

무슨 말인지 이해가 가지 않는 사람을 위해 다시 한번 PE 파일 포맷을 살펴보자




```
return (LPVOID)((DWORD)pProcessBase + baseOfCode);
```

첫번째 코드 섹션이 시작되는 주소를 반환한다.

다음으로 ③번에서 dll을 로드하고 브레이크 포인트를 삽입한 뒤 실행하는 코드는 아래와 같다.

```
BOOL CKeyboardHookDlg::InjectionDll(HANDLE hProcess, HANDLE hThread, VOID * HModuleBase,
CString szPathNamedll)
{
  FARPROC LoadLibProc;
  DWORD dwRead=0;
  BOOL bRet;
  *pOriginalCodePage=0;
```

```
pFirstCodePage=NULL;
*pOriginalCodePage=NULL;

// LoadLibraryA의 주소 얻어옴
LoadLibProc = GetProcAddress(GetModuleHandle("KERNEL32.dll"), "LoadLibraryA");

if (!LoadLibProc) {
   OutputDebugString("ERROR");
}

// 타겟 프로세스의 첫번째 코드 페이지의 주소를 알아옴
pFirstCodePage = GetFirstCodePage(hProcess, HModuleBase);

if(!pFirstCodePage) {
 OutputDebugString("ERROR");
}

OriginalContext.ContextFlags = CONTEXT_CONTROL;

if(!GetThreadContext(hThread, &OriginalContext)) {
  OutputDebugString("ERROR");
}

// 실행 프로세스의 첫번째 페이지 백업
bRet = ReadProcessMemory(hProcess, pFirstCodePage, pOriginalCodePage,
                         sizeof(pOriginalCodePage), &dwRead);
if(!bRet || sizeof(pOriginalCodePage) != dwRead) {
  OutputDebugString("ERROR");
}

// 구조체 초기화
pMyLoadLibraryA pNewCode = (pMyLoadLibraryA)pFakeCodePage;

// sup esp, 1000h
pNewCode->instr_SUB = 0xEC81;
pNewCode->operand_SUB_value = PAGE_SIZE; // 페이지크기(4096)
```

```
  // push <매개변수>
  pNewCode->instr_PUSH = 0x68;
  pNewCode->operand_PUSH_value = (DWORD)pFirstCodePage +
                                                    offsetof(MyLoadLibraryA, DllName);

  // call <함수주소> ; LoadLibraryA() 호출
  pNewCode->instr_CALL = 0xE8;
  pNewCode->operand_CALL_offset = (DWORD)LoadLibProc - (DWORD)pFirstCodePage -
                                            offsetof(MyLoadLibraryA, instr_CALL) - 5;

  // 마지막에 브레이크 포인트 삽입
  pNewCode->instr_INT_3 = 0xCC;

  strcpy(pNewCode->DllName, szPathNamedll.GetBuffer(szPathNamedll.GetLength()));

  // 우리의 루틴을 실행프로세스에 Write !!
  bRet = WriteProcessMemory(hProcess, pFirstCodePage, &pFakeCodePage,
                                                      sizeof(pFakeCodePage), &dwRead);
  if(!bRet || sizeof(pFakeCodePage) != dwRead) {
    OutputDebugString("ERROR");
  }

  // 실행 포인터(EIP)를 첫번째 페이지로 설정
  FakeContext = OriginalContext;
  FakeContext.Eip = (DWORD)pFirstCodePage;

  // 실행 스레드 컨텍스트 설정
  if(!SetThreadContext(hThread, &FakeContext)) {
    OutputDebugString("ERROR");
  }
  return TRUE;
}
```

코드를 자세히 살펴보자

```
LoadLibProc = GetProcAddress(GetModuleHandle("KERNEL32.dll"), "LoadLibraryA");
pFirstCodePage = GetFirstCodePage(hProcess, HModuleBase);
```

먼저 KERNEL32.dll 로부터 LoadLibraryA의 주소를 얻어온다. 이후에 LoadLibProc를 이용해 임의의 특정 dll을 로드한다. 주소를 얻어온 뒤 첫번째 코드 섹션의 주소를 가져온다.

```
OriginalContext.ContextFlags = CONTEXT_CONTROL;
bRet = ReadProcessMemory(hProcess, pFirstCodePage, pOriginalCodePage,
                                                    sizeof(pOriginalCodePage), &dwRead);
```

프로세스의 컨텍스트를 컨트롤하기 위해 ContextFlags에 CONTEXT_CONTROL 값을 할당한다. ContextFlags는 여러 값들이 들어갈 수 있는데 아래와 같다.

```
CONTEXT_CONTROL                 // SS:SP, CS:IP, FLAGS, BP
CONTEXT_INTEGER                 // AX, BX, CX, DX, SI, DI
CONTEXT_SEGMENTS                // DS, ES, FS, GS
CONTEXT_FLOATING_POINT          // 387 state
CONTEXT_DEBUG_REGISTERS         // DB 0-3,6,7
CONTEXT_FULL=(CONTEXT_CONTROL | CONTEXT_INTEGER > CONTEXT_SEGMENTS)
```

프로세스는 시스템의 현재 상태의 총합으로 생각할 수 있는데 프로세스는 실행될 때마다 프로세스의 레지스터와 스택 등을 사용하며 이것을 프로세스 컨텍스트라고 한다.
일반적으로 GetThreadContext API를 이용하여 특정 스레드의 컨텍스트 값을 가져오기 위해서는 먼저 적절한 값들로 초기화시켜야 하는데 사용된 CONTEXT_CONTROL을 설정할 경우 주석과 같이 SS:SP, CS:IP, FLAGS, BP를 저장할 수 있게 된다.
그리고 첫 번째 코드 섹션을 읽어와서 pOriginalCodePage에 저장한다.

```
pMyLoadLibraryA pNewCode = (pMyLoadLibraryA)pFakeCodePage;

// sup esp, 1000h
pNewCode->instr_SUB = 0xEC81;
pNewCode->operand_SUB_value = PAGE_SIZE; // 페이지크기(4096)

// push <매개변수>
pNewCode->instr_PUSH = 0x68; // push
pNewCode->operand_PUSH_value = (DWORD)pFirstCodePage +
                                                    offsetof(MyLoadLibraryA, DllName);

// call <함수주소> ; LoadLibraryA() 호출
pNewCode->instr_CALL = 0xE8; // call
pNewCode->operand_CALL_offset = (DWORD)LoadLibProc - (DWORD)pFirstCodePage -
```

```
                                        offsetof(MyLoadLibraryA, instr_CALL) - 5;

  // 마지막에 브레이크 포인트 삽입
  pNewCode->instr_INT_3 = 0xCC;

  strcpy(pNewCode->DllName, szPathNamedll.GetBuffer(szPathNamedll.GetLength()));
```

이제 첫번째 코드 페이지에 삽입할 구조체를 설정하며 해당 구조체는 LoadLibraryA를 호출하게 된다.

삽입하게 될 구조체의 모양은 다음과 같다.

```
typedef struct _MyLoadLibraryA {
    WORD        instr_SUB;
    DWORD       operand_SUB_value;
    BYTE        instr_PUSH;
    DWORD       operand_PUSH_value;
    BYTE        instr_CALL;
    DWORD       operand_CALL_offset;
    BYTE        instr_INT_3;
    char        DllName[1];
} MyLoadLibraryA, *pMyLoadLibraryA;
```

LoadLibrary("load.dll");을 호출하는 코드를 기계어로 작성하기 위해 만든 구조체이다.
LoadLibrary를 호출하는 코드를 어셈블리어로 표현하면 아래와 같다.

```
push ebp
mov ebp,esp
sub esp,40h
push [dll의 주소]
call [LoadLibraryA의 주소]
```

LoadLibraryA의 주소는 각 윈도우의 버전, 서비스 팩의 버전에 따라 달라지는데 Dependency Walker를 이용해서 Kernel32.dll의 Base Address에서 LoadLibraryA API의 Offset을 더한 값을 사용하는 것이 가장 쉽지만 각 윈도우 버전, 서비스 팩의 버전만큼 구현해야 한다는 것이 다소 번거롭다. 성상훈 님의 강좌에서는 다음과 같이 해당 주소를 구해서 사용하였다.

```
pNewCode->operand_CALL_offset = (DWORD)LoadLibProc - (DWORD)pFirstCodePage -
                                        offsetof(MyLoadLibraryA, instr_CALL) - 5;
```

call이나 jmp 명령등의 실행제어를 변경하는 명령어들은(OP Code) 32비트 환경에서 보통 5바이트의 크기를 가지는데 OP Code 1바이트 + 이동할 주소 4바이트가 결합된 크기이다. 이 때 이동할 주소는 RVA이므로 현재 OP Code를 수행하고 돌아올 리턴 주소를 실제 이동할 주소에서 뺀 값으로 기계어 코드를 생성하게 된다.

위의 코드에서

```
LoadLibraryA의 주소 : LoadLibProc
현재 실행 중인 코드의 주소 : pFirstCodePage - offsetof(MyLoadLibraryA, instr_CALL)
현재 명령어(OP Code)의 크기 : 5
```

가 되는 것이다.

```
pNewCode->instr_INT_3 = 0xCC;
strcpy(pNewCode->DllName, szPathNamedll.GetBuffer(szPathNamedll.GetLength()));
```

이제 마지막에 Break Point를 삽입하고 삽입할 Dll 이름을 복사한다.
여기까지 오면 이제 삽입할 구조체가 다 완성된 것이다.
자료를 다 채워 넣은 구조체를 WriteProcessMemory API를 이용해서 실행되고 있는 타겟 프로세스에 쓴다.

```
bRet = WriteProcessMemory(hProcess, pFirstCodePage, &pFakeCodePage,
                                                    sizeof(pFakeCodePage), &dwRead);
```

타겟 프로세스(hProcess)의 첫 번째 코드 페이지에(pFirstCodePage) 임의의 Dll을 로드하는 코드가 들어있는 구조체를(pFakeCodePage) 삽입한다.

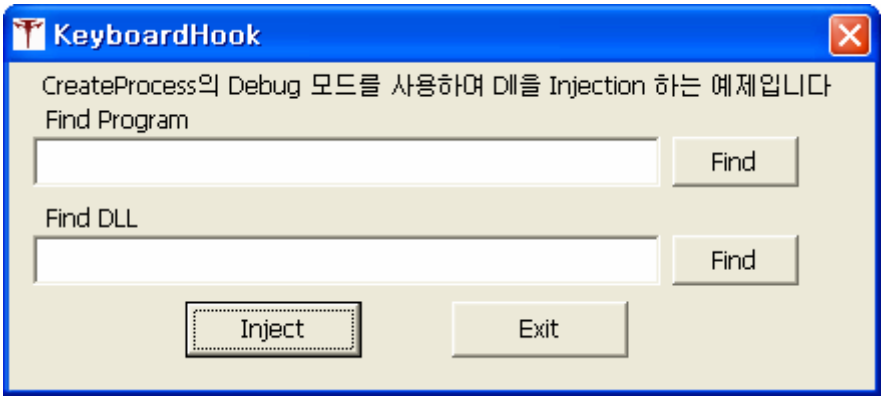
자 이제 ④번, 즉 EIP를 변경시키기만 하면 된다.

```
FakeContext = OriginalContext;
FakeContext.Eip = (DWORD)pFirstCodePage;

// 실행 스레드 컨텍스트 설정
if(!SetThreadContext(hThread, &FakeContext)) {
    OutputDebugString("ERROR");
}
```

변조할 컨텍스트에 현재 프로세스의 컨텍스트를 저장한 후 EIP를 첫 번째 코드 페이지를 가리키도록 수정한다. 이후 SetThreadContext API를 이용해 현재 프로세스의 스레드에 변조한 컨텍스트를 삽입한다.

다음은 첨부한 코드의 실행 화면이다.

실행 중이 아닌 프로세스를 선택하여 임의의 Dll을 주입할 수 있다.



소스 파일은 아래의 주소에서 받을 수 있다.(실행 파일 포함: KeyboardHook)

소스 파일 : http://consult.skinfosec.co.kr/~lucid7/Document/DebugApi.zip

1.3. Debug API를 이용한 API Hooking

이번에는 API Hook을 알아보자.
전자에서는 실행 중이 아닌 프로그램을 CreateProcess API에 DEBUG_ONLY_THIS_PROCESS 플래그를 주어서 디버그 모드로 진입했었다. 이번에는 실행 중인 프로그램을 Attach하여 디버그 모드에서 API를 Hooking하는 방법을 알아본다.

현재 실행 중인 Process를 Debug 모드로 Attach하기 위해서 DebugActiveProcess API를 사용한다.

```
BOOL DebugActiveProcess(
  DWORD dwProcessId
);
```

DebugActiveProcess API에 넘겨주어야 할 정보는 단지 Process ID뿐이다. 타겟 프로세스의 ID만 넘겨주면 나머지는 알아서 하게 된다.

DebugActiveProcess를 사용하여 타겟 프로세스를 Debug Mode로 Attach한 이후 진행되는 과정은 CreateProcess와 같다.
본 문서에서는 Dual님의 "Win32 유저계층에서의 API" 문서에서 소개된 소스를 바탕으로 진행을 한다. 다만 소스에서 약간 틀린 곳이 있어 수정하였음을 미리 밝힌다.(Dual님이 누락한 것 같아 보이는데 정확히는 알 수가 없다 -_-;)

방법은 거의 똑같다. 앞 절에서는 첫 번째 코드 페이지에 Break Point를 삽입했지만 이번에는 Hooking하기 원하는 타겟 API의 첫 번째 바이트에 Break Point를 삽입하는 것 뿐이다.
다만 이번에는 실행 중인 프로세스에 접근하는 것이기 때문에 VirtualQueryEx API와 VirtualProtectEx API를 사용하여 보호모드 속성을 조정한 후 WriteProcessMemory를 써야 한다. 왜냐하면 CreateProcess API를 사용하여 프로세스를 실행시켰을 경우에는 자연스럽게 해당 프로그램에 대한 제어권이 CreateProcess API를 사용한 프로그램으로 넘어오지만 다른 사용자가 실행시킨, 실행 중인 프로세스의 메모리는 다른 프로세스에게는 읽기 전용으로 되어 있을 것이기 때문이다.

여기서 앞 절에서 빼먹어버린 "Copy-On-Write"라는, 매우 중요한 개념을 설명하기로 한다.
Windows는 가능한 많은 메모리 페이지를 다른 프로세스와 공유하려 한다. 만약 해당 페이지를 사용하는 프로세스 중 어느 하나가 디버거에서 실행 중이어서 이들 페이지 중 하나가 브레이크 포인트를 갖고 있다면, 브레이크 포인트는 모든 프로세스가 공유하는 해당 페이지에 있어서는 안된다. 만약 그렇다면 디버거 외부에서 실행 중인 프로세스가 해당 코드를 실행하자마자, 해당 프로세스는 충돌하게 될 것이다. 이러한 상황을 피하기 위해서 운영체제는 특정한 프로세

스를 위해서 변경된 페이지가 있는지 살펴보고 브레이크 포인트가 쓰인 페이지를 특정한 프로세스만 접근할 수 있도록 복사본을 만든다. 따라서 프로세스가 페이지에 쓰자마자, 운영체제는 해당 페이지를 복사한다.(Debugging Applications for Microsoft .net and Microsoft Windows 에서 발췌함)
따라서 VirtualQueryEx API로 보호모드 속성을 얻은 후 VirtualProtectEx API로 보호모드 속성을 조정하는데(PAGE_EXECUTE_READWRITE) 이 때 Windows는 Copy-On-Write를 준비한다.

VritualQueryEx API의 함수원형은 아래와 같다.

```
SIZE_T VirtualQueryEx(
  HANDLE hProcess,
  LPCVOID lpAddress,
  PMEMORY_BASIC_INFORMATION lpBuffer,
  SIZE_T dwLength
);
```

```
hProcess : [in], 메모리 정보가 필요한 타겟 프로세스의 핸들
lpAddress : [in], 메모리 지역 정보
lpBuffer : [out], 반환된 메모리의 정보가 담긴 MEMORY_BASIC_INFORMATION 구조체를
                  가리키는 포인터
dwLength : [in], lpBuffer의 사이즈, Byte
```

이 API를 이용하면 해당 프로세스의 메모리 영역에 대한 정보를 얻을 수 있다.

VirtualProtectEx API의 함수 원형은 아래와 같다.

```
BOOL VirtualProtectEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flNewProtect,
  PDWORD lpflOldProtect
);
```

```
hProcess : [in], 보호모드 속성을 조정할 메모리의 프로세스 핸들
lpAddress : [in], 보호모드 속성을 조정할 메모리의 지역 정보
dwSize : [in], 접근하는 보호모드 속성 메모리의 크기
flNewProtect : [in], 변경할 보호모드의 속성
```

lpflOldProtect : [out], 원래 보호모드의 속성을 가리키는 포인터

기본적인 설명은 대충 했으니 이제 코드를 살펴보도록 한다. 기본적인 것은 앞 절과 대부분 동일하므로 핵심적인 부분만 보도록 한다.

```
// 프로세스를 처음 Attach 했을 때 발생하는 메시지 (case 3)
case CREATE_PROCESS_DEBUG_EVENT:
{
  // 실행된 프로세스의 정보를 얻어옴
  CProcessInfo = DebugEvent.u.CreateProcessInfo;

  // Send API의 주소를 가진 라이브러리의 핸들을 가져옴
  if( !(Wsock_Handle = GetModuleHandle("WS2_32.DLL")) )
  {
    if( !(Wsock_Handle = LoadLibrary("WS2_32.DLL")) ) // 없을 경우 Load한다.
      break; // 로드에 실패하면 Break
  }

  // Send API의 주소를 구한다.
  if( !(Send_Adr = GetProcAddress(Wsock_Handle,"send")) )
    break; // 주소값을 가져오지 못하면 Break

  // send API의 메모리 정보를 가져온다.
  VirtualQueryEx(CProcessInfo.hProcess, Send_Adr, &mbi, sizeof(mbi));

  // send API의 메모리 프로텍트를 가져와 수정한다.
  NewProtect = mbi.Protect;
  NewProtect &= ~(PAGE_READONLY | PAGE_EXECUTE_READ); // 제외시키고
  NewProtect |= (PAGE_READWRITE); // ReadWrite 추가

  // 보호 모드 조정
  VirtualProtectEx(CProcessInfo.hProcess, Send_Adr, sizeof(char), NewProtect,
                   &OldProtect);

  // 첫 바이트를 읽어온다.
  ReadProcessMemory(CProcessInfo.hProcess, Send_Adr, &FirstByte, sizeof(FirstByte),
                    &cbByte);
```

```
  // 읽어온 첫 바이트에 Break Point 설정. EXCEPTION_EVENT(0xCC) 기록
  WriteProcessMemory(CProcessInfo.hProcess, Send_Adr, &BreakByte, sizeof(BreakByte),
                              &cbByte);

  // 보호모드 다시 원래대로 조정
  VirtualProtectEx(CProcessInfo.hProcess, Send_Adr, sizeof(char), OldProtect,
                          &NewProtect);
}
```

CREATE_PROCESS_DEBUG_EVENT 발생 시 작동하는 코드이다. 코드에 주석을 상세히 달아 놓았으니 별 어려움을 없으리라고 본다.
SEND API의 주소를 가지고 있는 WS2_32.DLL을 로드한 다음 GetProcAddress API를 이용해 SEND API의 주소를 구한다.
VirtualQueryEx API로 SEND API의 메모리 정보를 가져온 다음 메모리 보호모드 설정을 가져와 Read-Write 권한을 추가한다.
수정한 보호모드를 VirtualProtectEx API를 이용해서 적용시킨 다음 SEND API의 첫 바이트를 읽어와 Break Point를 삽입하고 다시 보호모드를 원래대로 복원시킨다.

```
case EXCEPTION_BREAKPOINT: // 브레이크 포인트 이벤트이면
{
  if(FirstHit == FALSE) // 첫번째 브포 변수 체크
    FirstHit = TRUE;
  else
  {
    // 첫 바이트를 원래 대로 돌림
    WriteProcessMemory(CProcessInfo.hProcess, Send_Adr, &FirstByte,sizeof(FirstByte),
                                &cbByte);

    // Context Mode
    Org_Context.ContextFlags = CONTEXT_FULL;
    GetThreadContext(CProcessInfo.hThread, &Org_Context);      // Context를 구해온다.

    ESP = Org_Context.Esp;      // API가 끝나고 리턴(돌아갈) 주소
    ESP4 = ESP + 4;      //S
    ESP8 = ESP + 8;      //buf Adr
    ESPC = ESP + 0xC;      //len
```

```
ESP10 = ESP + 0x10;      //flag

// len읽어옴
ReadProcessMemory(CProcessInfo.hProcess, (void *)ESPC, &Len, sizeof(DWORD),
                        &cbByte);
buffer = malloc(Len); // 길이 만큼 메모리 할당

// Buffer 주소 읽어옴
ReadProcessMemory(CProcessInfo.hProcess, (void *)ESP8, &BufAdr, sizeof(DWORD),
                        &cbByte);

/*
// Buffer 읽어옴
ReadProcessMemory(CProcessInfo.hProcess, (void *)BufAdr, buffer, Len, &cbByte);
*/

memset(buffer, 0x44, Len); // 내용 조작

WriteProcessMemory(CProcessInfo.hProcess, (void *)BufAdr, buffer, Len, &cbByte);

New_Context = Org_Context; // 복사본을 만든다.
New_Context.Eip =
          (unsigned long)DebugEvent.u.Exception.ExceptionRecord.ExceptionAddress;

SetThreadContext(CProcessInfo.hThread,&New_Context);      // Context를 적용한다.
ContinueDebugEvent(DebugEvent.dwProcessId, DebugEvent.dwThreadId,
                        dwContinueStatus);

// 대상 프로그램에 결과 반영
/*
// 첫 바이트를 읽어온다.
ReadProcessMemory(CProcessInfo.hProcess, Send_Adr, &FirstByte, sizeof(FirstByte),
                        &cbByte);
*/

// 0xCC를 다시 기록
WriteProcessMemory(CProcessInfo.hProcess, Send_Adr, &BreakByte, sizeof(BreakByte),
```

```
                                &cbByte);
    free(buffer);      // 동적 할당한 메모리를 놓아준다.
    AfxMessageBox("send함수가 발생하였습니다.");      //시각적 효과

    dwContinueStatus = DBG_CONTINUE;
  }
  dwContinueStatus = DBG_CONTINUE;
}
```

EXCEPTION_BREAKPOINT, 즉 Break Point 발생 시 작동하는 함수이다.
Break Point 이벤트가 발생하면 먼저 변경했던 SEND API의 첫 번째 바이트를 원래대로 돌려놓는다. 이제 SEND API의 Parameter를 조작하기 위해 GetThreadContext API를 이용해 CONTEXT를 정보를 가져온다.

Stack Overflow를 이해한 사람이라면 무슨 말인지 알겠지만 혹시나 기억이 가물가물한 사람들을 위해 간단히 살펴본다.

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
        int i=10;
        char c='A';

        printf("%d %c", i, c);
        return 0;
}
```

위의 소스코드는 간단하게 콘솔에 i와 c를 출력하는 프로그램이다.

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
00411A30   push          ebp
00411A31   mov           ebp,esp
00411A33   sub           esp,0D8h
00411A39   push          ebx
00411A3A   push          esi
```

```
00411A3B   push          edi
00411A3C   lea           edi,[ebp-0D8h]
00411A42   mov           ecx,36h
00411A47   mov           eax,0CCCCCCCCh
00411A4C   rep stos      dword ptr [edi]
          int i=10;
00411A4E   mov           dword ptr [i],0Ah
          char c='A';
00411A55   mov           byte ptr [c],41h

          printf("%d %c", i, c);
00411A59   movsx         eax,byte ptr [c]
00411A5D   push          eax
00411A5E   mov           ecx,dword ptr [i]
00411A61   push          ecx
00411A62   push          offset string "%d %c" (4240C8h)
00411A67   call          @ILT+1170(_printf) (411497h)
00411A6C   add           esp,0Ch
          return 0;
00411A6F   xor           eax,eax
}
```

위는 원래의 소스 코드를 Disassembly 한 화면이다.

파란색 글씨를 보면 Parameter인 i 값과 c 값을 먼저 Stack에 push 한 다음 printf를 CALL 하고 있다. 즉, SEND API가 CALL 되기 직전에 Break Point에 의해 멈추어진 상태라면 SEND API에 들어가는 Parameter 값은 이미 Stack에 push된 상태임을 짐작할 수 있다.

(기억이 잘 나지는 않는데 이게 아마 파스칼 방식이던가? 뭐 그랬던 것 같다. 그리고 Intel은 파스칼 방식을 사용한다.)

자 다시 앞으로 돌아가서 얻어 온 CONTEXT 정보 중에서 ESP 값을 기준으로 Return Address를 수정한다.(왜 ESP 값을 기준으로 하는지 모른다면 Statck Overflow 강좌를 한번 읽어 볼 것을 권한다.

```
int send(SOCKET s, const char* buf, int len, int flags);
```

주석에도 설명을 해놓았지만 위의 SEND API의 원형에서 볼 수 있듯이(MFC에서는 CSocket::Send를 사용하며 Parameter가 틀리지만 내부적으로는 send() 함수를 이용하기 때문에 이상 없이 작동한다.) ESP4는 Parameter s를, ESP8은 Parameter buf의 주소를, ESPC는

Parameter len을, ESP10은 Parameter flags를 가리킨다.
위의 변수 중 우리가 주목해야 할 것은 바로 ESP8과 ESPC이다. ESP8에는 Send API를 사용해 전송되는 문자열이, ESPC는 전송된 문자열의 길이를 가리키게 된다.

```
// len읽어옴
ReadProcessMemory(CProcessInfo.hProcess, (void *)ESPC, &Len, sizeof(DWORD),
                          &cbByte);
buffer = malloc(Len); // 길이 만큼 메모리 할당

// Buffer 주소 읽어옴
ReadProcessMemory(CProcessInfo.hProcess, (void *)ESP8, &BufAdr, sizeof(DWORD),
                        &cbByte);

// Buffer 읽어옴
ReadProcessMemory(CProcessInfo.hProcess, (void *)BufAdr, buffer, Len, &cbByte);
```

위의 코드가 실제 Send API를 통해 전송되는 문자열을 읽어오는 핵심적인 부분이다.
보내진 문자열의 길이를 읽어와서 그 길이만큼 메모리를 할당한다. 문자열이 저장된 곳의 주소를 읽어 온 다음 그 주소에 위치한 문자열을 읽어와 buffer 변수에 저장한다.
(buffer에 저장된 값을 문자열로 변환하는 방법을 몇 일 고민했지만 본인의 내공이 딸리는 관계로 그냥 넘어갔다 -_-;)

이후의 소스는 buffer 변수에 임의의 문자를 덮어써서 전송되는 문자열을 조작하는 것을 나타내고 있다.

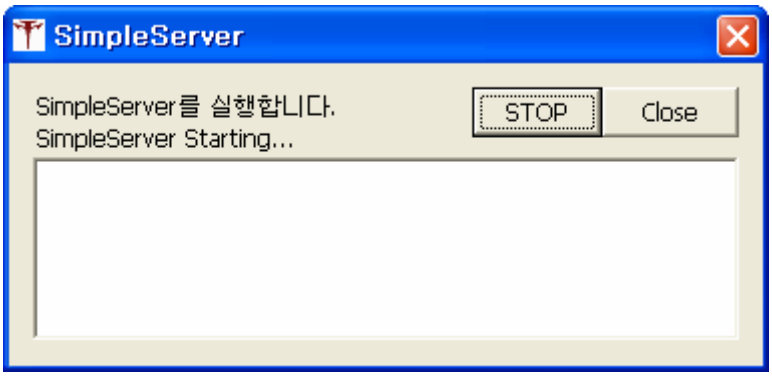
```
memset(buffer, 0x44, Len); // 내용 조작

WriteProcessMemory(CProcessInfo.hProcess, (void *)BufAdr, buffer, Len, &cbByte);

New_Context = Org_Context; // 복사본을 만든다.
New_Context.Eip =
(unsigned long)DebugEvent.u.Exception.ExceptionRecord.ExceptionAddress;

SetThreadContext(CProcessInfo.hThread,&New_Context);      // Context를 적용한다.
ContinueDebugEvent(DebugEvent.dwProcessId, DebugEvent.dwThreadId,
                                dwContinueStatus);
```

```
// 0xCC를 다시 기록
WriteProcessMemory(CProcessInfo.hProcess, Send_Adr, &BreakByte, sizeof(BreakByte),
                                &cbByte);
free(buffer);      // 동적 할당한 메모리를 놓아준다.
```

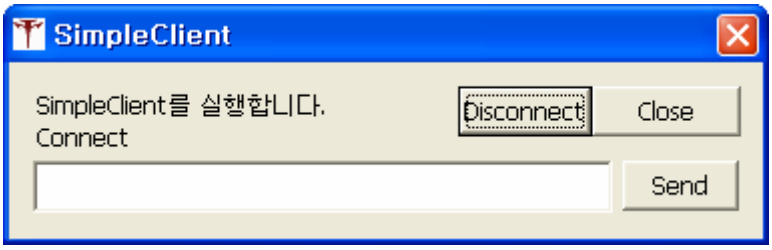
Buffer 변수에 0x44로 값으로 채운 후 WriteProcessMemory API를 이용해 메모리에 덮어쓴다. 그리고 변경한 CONTEXT 값을 다시 설정하는데 설정하기 전 EIP 값을 변경시킨다. DebugEvent.u.Exception.ExceptionRecord.ExceptionAddress 값은 Exception이 발생한 곳의 주소를 가리킨다. 즉 EIP 값을 Debug Event가 발생한 곳으로 돌려줌으로써 마치 Debug Event가 발생하지 않았던 것처럼 위장할 수 있다.
지금까지 설명했던 동작들을 계속 반복하기 위해 Break Point를 다시 설정한 후 buffer의 메모리를 해제한다.

다음은 해당 프로그램을 이용해 실제로 전송되는 내용을 조작하는 방법과 화면이다.
간단하게 제작한 C/S 프로그램과 API Hooking 프로그램을 사용해 전송되는 문자열을 DDD로 조작할 수 있다.
(해당 프로그램은 버그가 꽤 많이 존재한다. 역시나 내공이 부족해 버그를 다 잡지 못했으니 알아서 고쳐쓰기를 바랄뿐이다. -__-)

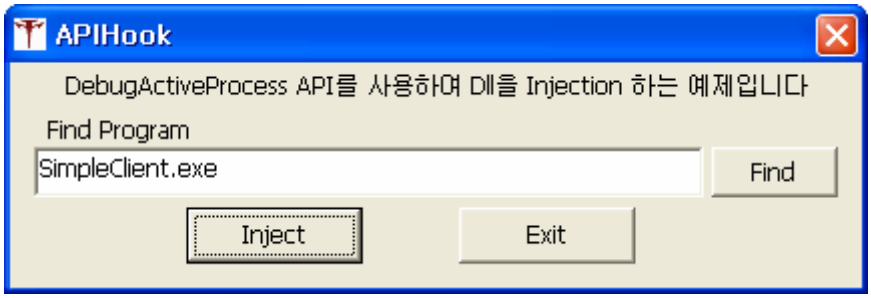
1. SimpleServer.exe를 실행한 후 START 버튼을 누른다.



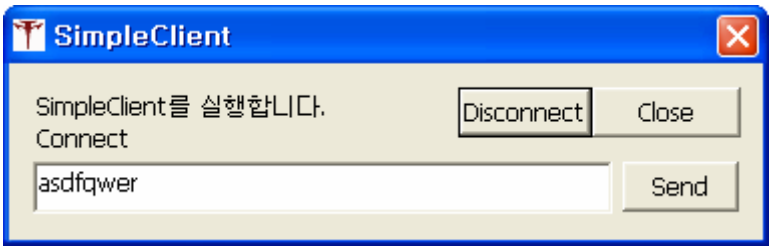
2. SimpleClient.exe를 실행한 후 Connect 버튼을 누른다.



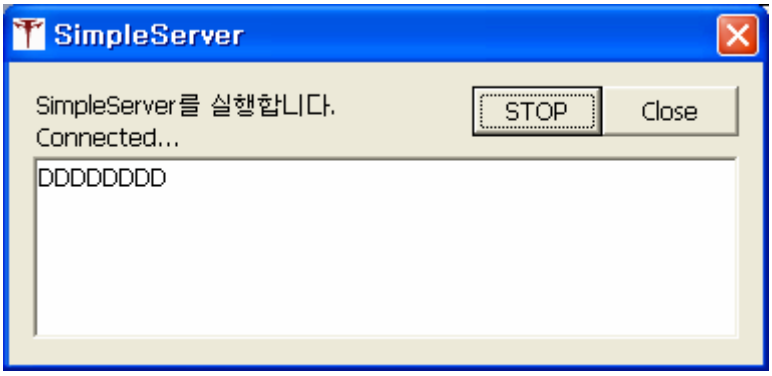
3. API Hook Program을 실행하여 Find 버튼을 눌러 SimpleClient를 선택한 뒤 Injection 버튼을 누른다.



4. SimpleClient의 에디트 창에 임의의 문자를 쓴 다음 Send 버튼을 누른다.



5. SimpleServer 화면에 DDDDDDDD이 전송되었음을 알 수 있다.



APIHook의 소스 파일 및 SimpleC/S의 실행 파일은 아래의 주소에서 받을 수 있다.

소스 파일(APIHook) : http://consult.skinfosec.co.kr/~lucid7/Document/DebugApi.zip

소스 파일(SimpleC/S) : http://consult.skinfosec.co.kr/~lucid7/Document/SimpleCS.zip

1.4. 참고문서

WebSite

1. Microsoft PE File Format

http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx

http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx

2. API Hooking Revealed

(http://www.codeproject.com/system/HookSys.asp)

3. 동우의 홈페이지

http://dasomnetwork.com/~leedw/mywiki/moin.cgi/

4. #44u6115f:p

http://dualpage.muz.ro/

5. 지니야 닷넷

http://www.jiniya.net/tt/

6. WIN32 - Inside Debug API

http://www.woodmann.com/fravia/iceman1.htm

Book

1. Debugging Applications for .net and windows, Microsoft Press

2. Visual C++ .net programming bible 2nd Edition, 김용성 저

Special Thanks to

Code Test and Modify : 이재익(fishacker, duckgu9@nate.com)

박경호(gafim2@nate.com)