윈도우 환경에서의 메모리 인젝션 기술과 인젝션 된 DLL 분석 기술

황현욱* · 채종호* · 윤영태*

요 약

최근 개인 PC 해킹과 경제적 이익을 목적으로 하는 게임 해킹이 급증하면서 윈도우즈 시스템을 대상으로 하는 특정 목적의 악성코드들이 늘어나고 있다. 악성코드가 은닉 채널 사용이나 개인 방화벽과 같은 보안 제품 우회, 시스템 내 특정 정보를 획득하기 위한 기술로 대상 프로세스의 메모리 내에 코드나 DLL을 삽입하는 기술이 보편화되었다.

본 논문에서는 대상 프로세스의 메모리 영역에 코드를 삽입하여 실행시키는 기술에 대해 분석한다. 또한 피해 시스템에서 실행중인 프로세스 내에 인젝션 된 DLL을 추출하기 위해 파일의 PE 포맷을 분석하여 IMPORT 테이블을 분석하고, 실행중인 프로세스에서 로딩중인 DLL을 추출하여 명시적으로 로딩된 DLL을 추출하고 분석하는 기법에 대해 설명하였다. 인젝션 기술 분석과 이를 추출하는 기술을 통해 피해시스템 분석시 감염된 프로세스를 찾고 분석하는 시발점이 되는 도구로 사용하고자 한다.

Memory Injection Technique and Injected DLL Analysis Technique in Windows Environment

Hyun Uk Hwang* · Jong Ho Chae* · Young Tae Yun*

ABSTRACT

Recently the Personal Computer hacking and game hacking for the purpose of gaining an economic profit is increased in Windows system. Malicious code often uses methods which inject dll or code into memory in target process for using covert channel for communicating among them, bypassing secure products like personal firewalls and obtaining sensitive information in system.

This paper analyzes the technique for injecting and executing code into memory area in target process. In addition, this analyzes the PE format and IMPORT table for extracting injected dll in running process in affected system and describes a method for extracting and analyzing explicitly loaded dll files related with running process. This technique is useful for finding and analyzing infected processes in affected system.

Key words : Memory Injection, DLL injection, Malicious Code

* 국가보안기술연구소

1. 서 론

최근 해킹 유형은 유닉스와 리눅스 시스템의 서버군 해킹에서 개인 PC를 대상으로 하는 윈도우즈 시스템 해킹으로 확장되고 있다. 또한 해킹의 목적도 과거 자신의 실력을 표현하는 형태에서 경제적 이익을 취하려는 형태로 변화하고 있다. 이러한 공격 유형은 트로이 목마나 백도어가 감염되게끔 웹사이트를 통해 유포되는 형태나 메신저 게시물의 형태로 감염시킴으로써 개인 사용자들에게 많은 위험을 주고 있다. 윈도우즈 시스템의 권한을 획득한 후에는 피해 시스템에 지속적으로 악성코드를 상주시켜 중요한 정보를 획득하고 외부와의 통신을 통해 정보를 유출시키는 행위를 한다. 이렇게 중요 정보를 획득하고 은닉 채널을 형성하고 방화벽과 같은 보안 프로그램을 우회하는 행위를 위해 사용되는 기술로서 프로세스 메모리 내에 DLL이나 코드를 인젝션 시키는 기술이 이용되고 있다[1, 2, 12].

악성코드는 DLL 또는 코드 인젝션을 이용하여 정상적인 시스템 프로세스의 주소영역에 임의의 DLL이나 코드를 인젝션하여 해당 프로세스의 메모리 영역을 자유롭게 접근하여 함수의 후킹이나 전달되는 데이터의 조작까지 가능하게 된다. 최근 이루어지는 게임 해킹도 이 기술의 범주에 속한다고 볼 수 있다. 또한 메모리 조작을 통한 악성코드들은 정상적인 프로세스와의 구별이 힘들어 일반 사용자 입장에서는 악의적인 목적의 프로세스인지 그 여부를 판별하기가 힘들다.

본 논문에서는 DLL과 코드 인젝션 기술에 대해 알아보고, 피해 시스템 분석 시 감염된 프로세스의 여부를 판단하기 위해 파일의 PE 포맷 분석[3, 10]과 실행중인 프로세스를 분석함으로써 임의로 삽입된 DLL을 추출한 기법에 대해 기술한다.

본 논문의 구성은 다음과 같다. 2장에서는 다른 프로세스의 주소 공간에 인젝션 한 후 대상 프로세스 컨텍스트 내에서 이 코드를 실행하는 방법을 살펴보고, 3장에서는 인젝션 된 DLL의 추출하기 위한 방법에 대해 기술한다. 마지막으로 4장에서 본 논문의 결론을 맺는다.

2. 메모리 인젝션 기술

다른 프로세스의 메모리에 코드를 삽입함으로써 악의적인 사용자는 임의의 코드를 실행하거나 DLL을 로딩하여 다양한 목적으로 이용할 수 있다[1, 4, 11]. 여기에서는 대표적으로 사용하는 SetWindowsHookEx를 이용한 DLL 삽입기술, CreateRemoteThread를 통해 원격 대상 프로세스에 DLL을 로딩하는 기술, 디버깅 API를 이용하여 인젝션 하는 기술, 마지막으로 WriteProcessMemory를 통해 코드를 기술하고 CreateRemoteThread를 통해 실행시키는 인젝션 기술에 대해 기술한다.

2.1 윈도우즈 훅(SetWindowsHookEx)을 이용한 기법

메시지 기반으로 동작하는 윈도우즈에서는 운영체제와 응용 프로그램, 또는 응용 프로그램과 응용 프로그램 사이에서 많은 메시지를 주고받게 된다. 이러한 메시지 사이에 위치하여 감시와 조작이 가능한 프로시저가 훅(Hook)이다. 훅 프로시저는 메시지를 감시하거나 변경하거나 전달되지 않게 하는 것도 가능하다.

훅 프로시저가 어떤 메시지를 받을 것인가는 훅 타입과 훅의 범위에 따라 달라진다. 훅 타입은 WH_로 시작하는 매크로 상수로 정의된다. 예를 들면, WH_KEYBOARD는 키보드 메시지를 감시하며, WH_CALLWNDPROC는 SendMessage 함수로 메시지를 보내기 전에 처리하고, WH_CALLWNDPROCRET는 윈도우 프로시저가 메시지를 처리한 후에 호출하는 훅 프로시저이다[5].

또한 훅은 범위에 따라 모든 쓰레드에서 발생하

는 메시지를 가로챌 수 있는 시스템 전역 훅 프로시저와 특정 쓰레드에서 발생하는 메시지만 가로챌 수 있는 쓰레드 한정적 훅 프로시저로 나누어진다[6].

훅 프로시저는 필요에 따라 언제든지 설치될 수 있으며 훅 타입에 따라 여러 개의 훅 프로시저가 존재할 수도 있어서 운영체제는 훅 프로시저들을 훅 체인으로 관리한다.

이러한 훅 프로시저를 설치할 때는 SetWindows HookEx를 사용한다<표 1>.

〈표 1〉 SetWindowsHookEx 함수 원형

```
HHOOK SetWindowsHookEx(
  int idHook,            // hook type
  HOOKPROC lpfn,     // hook procedure
  HINSTANCE hMod,   // handle to application
instance
  DWORD dwThreadId / thread identifier
);
```

첫 번째 인수의 idHook은 설치하고자 하는 훅 타입을 지정하며, lpfn은 훅 프로시저의 번지이고, hMod는 훅 프로시저를 가진 인스턴스 핸들이며, dwThreadId는 훅 프로시저가 감시할 쓰레드의 ID이다. ID 값이 0이면 시스템의 모든 쓰레드에서 발생하는 메시지가 훅 프로시저로 전달된다.

이러한 윈도우즈 훅을 이용하여 악성코드에서는 훅 프로시저를 DLL에 두고, 훅 프로시저를 포함하는 DLL을 후킹하려는 대상 프로세스의 쓰레드 주소 공간으로 사상(mapping)하게 된다. 다시 말해 DLL은 윈도우즈 훅을 경유하여 원격 프로세스에 DLL을 사상하게 된다. 하지만 윈도우즈는 SetWindowsHookEx에 대한 성공적인 호출 후에 DLL을 후킹된 쓰레드의 주소 공간으로 자동으로 사상하지만 반드시 바로 사상하는 것은 아니다. 사상이 되기 위해서는 메시지 이벤트가 발생되어야만 사상이 되게 된다. <표 2>에서와 같이 WH_CALLWNDPROC과 같은 훅이 설치되었다면 메시지가 대상 쓰레드에 실제 전송될 때까지 DLL이 대상 프로세스에 사상되지 않게 된다. 만약 대상 프로세스에 메시지가 전달되기 전 UnhookWindowsHookEx가 호출된다면 DLL은 결코 대상 프로세스에 사상되지 않는다. 마찬가지로 UnhookWindowsHookEx 호출 후의 DLL의 unmapping에 대한 것도 적절한 메시지 이벤트가 발생할 때까지 DLL은 사실 상 unmapping 되지 않게 된다. <표 2>는 SetWindowsHookEx를 호출하여 DLL을 인젝션 하는 의사(pseudo) 코드이다.

〈표 2〉 SetWindowsHookEx API를 사용한 의사 코드

```
int InjectDll( HWND hWnd )
{
        // hWnd : 시작버튼 윈도우 핸들
          g_hWnd = hWnd;

          // Hook "explorer.exe"
          g_hHook = SetWindowsHookEx
(WH_CALLWNDPROC,
          (HOOKPROC)HookProc,
          hDll,
GetWindowThreadProcessId(hWnd, NULL));
          SendMessage(hWnd,WM_HOOKEX,0,1
);
          return g_bSubclassed ;
}
```

2.2 CreateRemoteThread를 이용한 기법 (CreateRemoteThread + LoadLibrary)

CreateRemoteThread를 이용한 기법은 최근 악성코드에서 가장 많이 사용하는 기법중 하나이다[7]. 이는 대상 프로세스에서 DLL을 동적으로 실행이 가능하다. CreateRemoteThread() API를 이용한 DLL Injection은 다음과 같은 흐름을 갖게 된다.

① 원격 프로세스에 대한 HANDLE을 얻는다 (Open Process).

② 원격 프로세스 내에서 DLL 이름을 위한 메모리를 할당한다(VirtualAllocEx).
③ 할당된 메모리에 전체 경로를 포함하는 DLL 이름을 기록한다(WriteProcessMemory).
④ CreateRemoteThread와 LoadLibrary를 이용하여 원격 프로세스에 DLL을 사상한다.
⑤ 원격 쓰레드가 종료할 때까지 기다린다(WaitForSingleObject). 이것은 LoadLibrary에 대한 호출이 반환할 때까지 기다리는 것이다. 이 쓰레드는 DllMain(DLL_PROCESS_ATTACH라는 reason으로 호출됨)이 반환하는 순간 종료하게 된다.
⑥ 원격 쓰레드의 탈출 코드를 얻는다(GetExitCode Thread). 이 값은 LoadLibrary에 의해 반환되는 값으로서 사상된 DLL의 base address (HMODULE)이다.
⑦ 단계 ②에서 할당된 메모리를 해제한다(VirtualFreeEx).
⑧ CreateRemoteThread와 FreeLibrary를 이용하여 원격 프로세스에서 DLL을 언로드한다. FreeLibrary에 단계 6에서 반환된 HMODULE을 전달한다(Create RemoteThread의 lpParameter를 경유한다). 인젝션된 DLL이 새로운 쓰레드를 생성했다면 이 DLL을 언로드하기 전에 종료되었는지를 확인해야 한다.
⑨ 쓰레드가 종료할 때까지 기다린다(WaitForSingle Object). 또한 작업이 끝났다면 모든 핸들들을 닫아야 한다. 단계 ④와 ⑧에서 생성된 두 개의 쓰레드에 대한 핸들과 단계 ①에서 반환된 원격 프로세스에 대한 핸들을 닫는다.

2.3 디버깅 API를 이용한 기법

윈도우즈에서는 강력한 디버깅 API를 제공하고 있다[8]. 이는 악의적인 의도보다 프로그램을 디버깅하는 용도로 만들어놓은 것이지만 메모리를 다룰 수 있는 방법 중의 하나가 된다. 대표적인 API는 ReadProcessMemory와 WriteProcessMemory이다. CreateProcess의 인자인 dwCreationFlag에서 DEBUG_PROCESS, DEBUG_ONLY_THIS_PROCESS를 선택해 생성시키면, CreateProcess를 부른 프로세스가 디버거가 된다. 이는 WaitForDebugEvent를 통해 디버그 이벤트를 받을 수 있다.

윈도우즈 디버깅 API를 이용하여 DLL을 삽입하는 시나리오는 다음과 같다.

① 대상 프로세스를 디버깅용 API를 써서 실행 즉시 suspend 시킨다.
② 대상 프로세스의 컨텍스트(process context: 프로그램 카운터와 레지스터등 프로세서의 상태를 말한다)와 코드를 삽입할 영역(코드 섹션의 첫 페이지)을 읽어 와서 다른 메모리에 백업한다(ReadMemoryProcess() 이용).
③ 해당 영역에 임의의 dll을 로드하는 코드를 덮어쓴다(WriteProcessMemory() 이용).
④ 인스트럭션 레지스터인 EIP에 실행시킬 코드의 주소를 저장시켜서 프로세스를 run시킨다. 물론 이때 dll을 로드하는 코드의 마지막 부분에는 브레이크 포인트를 set하는 코드(int 3h)가 있어야 한다[9].
⑤ dll을 로딩하는 코드가 실행된 후 타깃 프로세스는 INT3 명령에 의해 브레이크가 걸리게 된다. 백업해 놓았던 프로세스 컨텍스트와 코드를 원래대로 되돌려놓고 EIP를 복구시킨다(Write ProcessMemory() 이용).
⑥ 프로세스를 다시 run시킨다(디버깅용 API 이용).

2.4 코드 인젝션 기법(WriteProceessMemory + CreateRemoteThread)

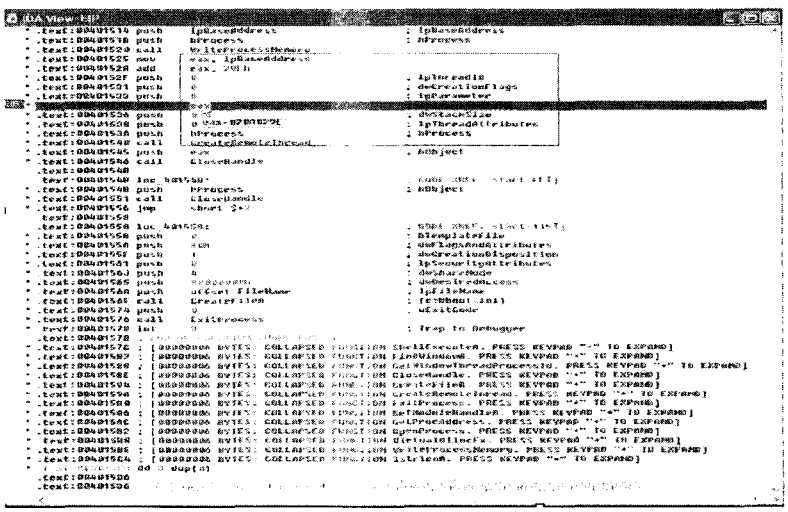
다른 프로세스의 주소 공간에 특정 코드를 복사한 후, 이 프로세스의 컨텍스트 내에서 실행하는 또

다른 방법은 원격 쓰레드와 WriteProcessMemory API를 이용하는 방법이다. 독립적인 DLL을 작성하는 대신 코드를 원격 프로세스에 직접 복사한 후(WriteProcessMemory를 이용) CreateRemoteThread를 이용하여 실행할 수 있다.

코드 인젝션 시나리오는 다음과 같다.

① 원격 프로세스에 대한 HANDLE을 얻는다(Open Proces).
② 주입될 데이터를 위해 원격 프로세스의 주소 공간에 메모리를 할당한다(VirtualAllocEx).
③ 할당된 메모리에 초기화된 INJDATA 구조체의 복사본을 기록한다(WriteProcessMemory).
④ 주입될 코드를 위해 원격 프로세스의 주소 공간에 메모리를 할당한다.
⑤ 할당된 메모리에 ThreadFunc의 복사본을 기록한다.
⑥ CreateRemoteThread를 이용하여 ThreadFunc의 원격 복사본을 시작한다.
⑦ 원격 쓰레드가 종료할 때까지 기다린다(Wait For SingleObject).
⑧ 원격 프로세스에서 결과를 얻는다(ReadProcess Memory 또는 GetExitCodeThread).
⑨ 단계 ②와 ④에서 할당되었던 메모리를 해제한다(VirtualFreeEx).
⑩ 단계 ⑥과 ①에서 얻었던 핸들들을 닫는다(Close Handle).

코드 인젝션을 이용하기 위해서는 유의할 점이 존재한다. 첫째, 코드 인젝션은 DLL 인젝션과 달리 컴파일과 링킹의 시점에 제공될 수 있는 API들의 주소정보를 얻을 수 없다. 따라서 코드 인젝션에서는 사용하는 라이브러리를 LoadLibrary()와 GetProcAddress를 이용해 주소를 직접 구해 사용해야 한다. 둘째, 정적 문자열 사용에 대한 문제이다. 컴파일 후에는 문자열을 데이터 영역에 저장하고 주소를 넘겨주어 처리하지만 코드 인젝션 후에는 오프셋이 틀려지게 된다. 따라서 문자열을 넘길 때는 인자로 넘겨주어 해결해야 한다. 셋째, 디버그 모드에서는 _chkesp를 통해 스택을 점검하는 루틴이 추가되므로, 코드 인젝션이 일어난 후 오류가 발생하게 된다. 따라서 릴리즈 모드로 이를 해결해야만 한다. 윈도우즈 NT 계열에서는 이러한 문제를 CreateRemoteThread의 인자로 넘김



(그림 1) code injection을 수행하는 인젝터 역할을 하는 악성코드의 디버깅 모습

으로써 해결이 가능하며, 릴리즈 모드로 컴파일 함으로써 문제점을 해결할 수 있다. (그림 1)은 code injection을 수행하는 인젝터 역할을 하는 악성코드를 디버깅한 모습이다. CreateRemoteThread와 그 인자가 수행되었음을 알 수 있다.

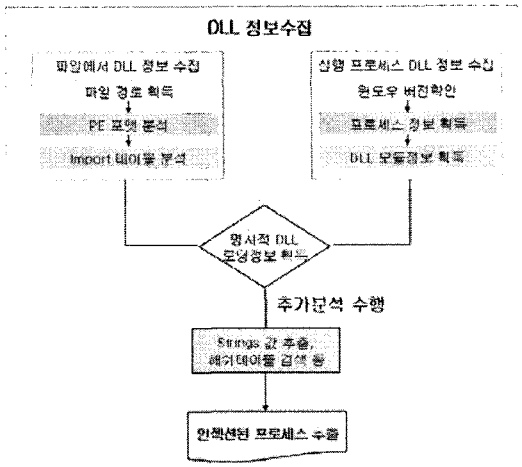
3. DLL 인젝션 탐지 기법

3.1 인젝션된 DLL 분석

2장에서 살펴 본 것과 같이 악성코드가 사용할 수 있는 인젝션 기법은 다양하다. 이러한 인젝션을 탐지하기 위해 다양한 관점과 방법이 존재한다. 인젝션 되는 순간을 모니터링 하다가 경고를 내는 형태나, 인젝션과 관련된 API들을 감시하는 기법 등이 존재한다. 본 논문에서는 피해시스템 분석 관점에서 피해 시스템에서 해킹이 일어난 후 프로세스를 검사하여 인젝션된 DLL을 찾는 방법에 대해 기술하고자 한다.

악성코드들의 인젝션은 이미 컴파일된 프로그램들을 통해 인젝션이 발생하므로 LoadLibrary API를 이용한 명시적 로딩을 이용한다. 명시적 로딩은 DLL의 로딩 여부가 링크 시에 결정되는 것이 아니라 사용자가 원할 때 해당 DLL을 로드하여 원하는 함수를 사용할 수 있게 하는 방법이다. 따라서 분석자 관점에서는 해당 프로세스들이 실행 전에 임포트하고 있는 DLL 정보와 실행 후 명시적으로 로딩되는 정보를 얻어 비교하면 분석의 범위를 줄일 수 있게 된다. 이렇게 프로그램이 실행된 후의 프로세스에 로딩된 DLL을 추출하여 해당 DLL에 대해 문자열 값 추출, 패턴 비교 등을 통하면 악의적인 DLL 정보를 찾는데 유용한 정보를 제공할 수 있다. (그림 2)는 인젝션된 DLL을 찾기 위한 분석 과정이다.

바이너리 파일의 PE 파일 포맷을 추적하면 import



(그림 2) 프로세스에 인젝션된 DLL 정보 추출 흐름도

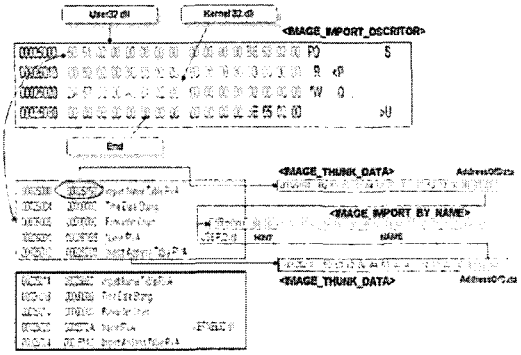
table을 통해 심볼을 임포트하는 DLL 정보를 얻을 수 있다. PE의 섹션 테이블에 ".idata"라는 이름으로 지정된다. 임포트 섹션은 IMAGE_IMPORT_DESCRIPTOR라는 구조체의 배열로 시작한다. 링크된 DLL 개수+1 만큼의 IMAGE_IMPORT_DESCRIPTOR 구조체가 연속적으로 존재하며 마지막은 NULL로 채워져

〈표 3〉 IMAGE_IMPORT_DESCRIPTOR 구조체

```
typedef struct
_IMAGE_IMPORT_DESCRIPTOR {
    union {
    DWORD    Characteristics;// 쓰이지 않음

    // IMAGE_THUNK_DATA 구조체 배열
    DWORD    OriginalFirstThunk;
    };
    DWORD    TimeDateStamp;
    DWORD    ForwarderChain;

   //임포트된 DLL의 이름을 담고 있는
   //NULL로 끝나는 아스키 문자열에 대한
   //RVA 값 가짐, DLL 이름의 RVA, DLL
   //이름의 포인터
    DWORD    Name;

    DWORD    FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

있다. 표의 "DWORD Name" 변수는 임포트된 DLL의 이름을 담고 있는 NULL로 끝나는 아스키 문자열에 대한 RVA 값을 가진다.

실제 파일을 덤프해보면 각 DLL에 따른 IMAGE_IMPORT_DESCRIPTOR 구조체가 각각 존재함을 알 수 있다<표 3>. NAME의 RVA 값을 파일 오프셋으로 변환하면 실제 임포트하는 DLL 정보를 찾을 수 있다.



(그림 3) IMAGE_IMPORT_DESCRIPTOR 구조체를 실행파일에서 추적한 모습

이와 같이 추출된 DLL들에서 모두 임포트하는 정보를 추출하면 해당 프로세스의 PE 이미지에서 임포트하는 DLL의 정보를 모두 추출할 수 있다. (그림 3)은 실제 실행 파일에서 IMAGE_IMPORT_DESCRIPTOR 구조체를 추적한 모습을 나타낸다.

3.2 프로세스에서의 로드된 모듈 분석

윈도우즈 NT 이상 버전의 윈도우 운영체제에서는 현재 시스템의 프로세스 목록에 관한 정보를 제공하기 위해 PSAPI 라이브러리를 제공한다. 이 API는 'PSAPI.dll'에 있으며 이것은 Platform SDK와 함께 배포된다. PSAPI 라이브러리에 있는 API를 이용하기 위해서는 'PSAPI.lib'을 링크해야 하고, 'PSAPI.h'를 소스 파일에서 포함해야 한다. PSAPI 라이브러리는 17여 종의 API를 제공하고 있다. 현재 수행중인 프로세스 목록에 대한 정보를 얻기 위해 EnumProcesses(), EnumProcessModules(), OpenProcess(), GetModuleBaseName()과 같은 API를 사용한다.

EnumProcesses()는 <표 4>와 같이 정의되어 있다. EnumProcesses()는 현재 수행중인 프로세스들의 PID를 얻어오기 위해 사용되는 API이다. 해당 API에는 DWORD 형의 배열에 대한 포인터 값(pProcessIds, 프로세스들의 PID를 저장하기 위한 메모리 공간)과 배열의 전체 크기(cb, 저장 할 수 있는 최대 프로세스 개수), 그리고 API 수행 결과 배열에 저장된 데이터의 크기를 저장하기 위한 DWORD 형 변수의 포인터(pBytesReturned)가 파라미터로 요구된다. EnumProcesses()를 호출한 뒤, 'pBytesReturned / sizeof(DWORD)'를 계산하면 프로세스의 개수를 알 수 있다.

〈표 4〉 EnumProcesses API 원형

```
BOOL EnumProcesses(
  DWORD* pProcessIds,
  DWORD cb,
  DWORD* pBytesReturned
);
```

또한 다른 방법으로 ToolHelper API를 이용하는 방법이 있다. ToolHelp32 라이브러리는 'Kernel32.dll'에 위치해 있으며 윈도우 표준 API이다. ToolHelp32 라이브러리는 위에서 설명한 PSAPI 라이브러리와 비슷하게 시스템 내의 프로세스, 혹은 쓰레드들을 나열할 수 있고 각 프로세스나 쓰레드의 메모리와 모듈 정보를 얻을 수 있는 API들로 구성되어 있다. 프로세스 목록을 열거하기 위해서는 CreateToolhelp32Snapshot(), Process32First(), Process32Next()와 같은 3개의 API를 사용한다. ToolHelp32 라이브러리에 있는 API를 이용하기 위해, 가장 먼저 해야 할 일은 "스냅샷"을 생성하는 일인데, 이것은 CreateToolHelp32Snapshot()을 이용한다. CreateTool

Help32Snapshot() 함수는 현재 시스템의 특정 프로세스의 현재 상황에 대한 스냅샷과 힙, 모듈, 쓰레드들에 대한 스냅샷을 얻을 수 있다. <표 5>는 이 API의 정의를 나타낸다.

〈표 5〉 CreateToolhelp32Snapshot API 원형

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

CreateToolHelp32Snapshot()은 2개의 인자를 요구한다. 'dwFlags'는 어떤 스냅샷을 생성할 것인지에 대한 정보를 표시한다. TH32CS_SNAPMODULE을 이용하면 th32ProcessID에 명시된 프로세스가 사용하는 모든 모듈에 대한 스냅샷 정보를 구할 수 있다. 'th32ProcessID'는 생성하는 스냅샷에 포함되어야 하는 프로세스 ID이다. 이 값이 0 인 경우 현재 프로세스를 의미하고, 몇 가지 정의된 값을 가질 수 있다. <표 6>은 DLL 모듈정보를 구하는 간단한 의사 코드이다.

〈표 6〉 CreateToolhelp32Snapshot를 이용하여 모듈 정보를 얻는 의사 코드

```
void GET_Module(int pid)
{
        HANDLE hSnap;
        MODULEENTRY32 me;
        char str[256];

// 모듈정보에 대한 스냅샷
hSnap=CreateToolhelp32Snapshot(TH32CS_SNA
PMODULE,pid);
        if (hSnap == (HANDLE)-1)
                return;
        me.dwSize=sizeof(MODULEENTRY32);
        ...
if (Module32First(hSnap,&me)) { // 모듈
                do {
        wsprintf(str,"%s",me.szExePath);
} while (Module32Next(hSnap,&me));
        }
        ...
        CloseHandle(hSnap);
}
```

참고로, ToolHelper API는 윈도우즈 NT 4.0에서는 사용할 수 없다.

3.3 추출된 DLL 정보 분석

파일의 PE 포맷을 분석하여 추출된 DLL의 정보와 프로세스에서 로딩된 DLL 정보를 비교하는 방법은 악성코드들이 사용하는 메모리 인젝션 기술 분석을 바탕으로 임의의 DLL을 사용하기 위해서는 대상 프로세스의 메모리 영역 내에서 명시적으로 DLL을 로딩하는 특성을 이용한 것 뿐이다. 따라서 악성코드가 악의적으로 사용한 DLL인지를 명확히 판단하기에서는 추가적인 작업이 필요하게 된다. 가장 기본적으로 DLL의 추가적인 분석을 위한 문자열 값을 추출하거나 DLL 디버깅 과정이 들어가야 한다. 또한 악의적인 DLL의 해쉬 테이블을 구성하여 검색하는 방법의 보조적인 분석 기술을 통해 실제 악의적으로 삽입된 DLL을 밝혀낸다.

4. 결 론

본 논문에서는 최근 악성코드들이 사용하는 메모리 변조 기법의 세부 기술에 대해 논하였다. 대표적인 후킹 함수인 SetWindowsHookEx를 이용하여 DLL을 로딩하고, CreateProcessThread를 이용해 DLL을 로딩하거나, 디버깅 API를 이용한 기법, WriteProcessMemory 함수와 CreateProcess Thread 함수를 통해 코드를 삽입하는 기술이 그 대표적인 기술들이다. 각 기법들에 대한 구현은 윈도우즈 버전과 그 목적에 따라 달라진다.

이렇게 삽입된 DLL을 추출하기 위해 본 논문에서는 파일의 PE 포맷을 분석하여 IMPORT 테이블의 DLL을 추출하고, 실행중인 프로세스에서 로

딩된 DLL을 추출하여 명시적으로 로딩된 DLL을 찾아내고 있다. 인젝션 기술에서 살펴본 것처럼 대상 프로세스에서 DLL을 삽입하기 위해서는 명시적으로 로딩해야 하므로 이러한 DLL을 구별해 주는 것은 악의적인 DLL을 구별하는데 도움이 된다. 이렇게 추출된 DLL로부터 문자열 값을 추출하고 사전에 만들어놓은 시그니처의 해쉬 테이블과 비교 검색하면 악의적인 DLL을 찾는 분석 도구로 사용될 수 있으며, 피해시스템 분석 관점에서 감염된 프로세스에 인젝션 된 DLL을 찾는다는 것은 분석의 시발점이 될 수 있다.

참 고 문 헌

[1] Load Your 32-bit DLL into Another Process's Address Space Using INJLIB by Jeffrey Richter. MSJ, May 1994.

[2] MSDN, DLLs, Processes, and Threads, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/dynamic_link_libraries.asp

[3] Peering Inside the PE : A Tour of the Win32 Portable Executable File Format by Matt Pietrek, March 1994.

[4] Codeguru, DLL Injection and hooks, http://www.codeguru.com/forum/archive/index.php/t-226892.html.

[5] Tutorial 24 : Windows Hooks by Iczelion.

[6] HOWTO : Subclass a Window in Windows 95 ; Microsoft Knowledge Base Article-125680.

[7] Robert Kuster, "Three Ways to Inject Your Code into Another Process", http://www.codeproject.com/threads/winspy.asp

[8] MSDN, Debugging and Error Handling, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_events.asp

[9] Intel Architecture Software Developer's Manual, Volume 2 : Instruction Set Reference.

[10] 이호동, "WIndows 시스템 실행파일의 구조와 원리", 한빛미디어.

[11] 이동우, DLL injection, http://www.dasomnetwork.com/~leedw

[12] 김성우, "해킹/파괴의 광학", 정보게이트.

황 현 욱

2000년 조선대학교 정보통신공학과(공학사)
2002년 조선대학교대학원 전자공학과(공학석사)
2004년 전남대학교대학원 정보보호학과(이학박사)
2004년 ~ 현재 국가보안기술연구소 연구원

채 종 호

1998년 성균관대학교 정보공학과(공학사)
2001년 성균관대학교대학원 컴퓨터공학과(공학석사)
2004년 BCQRE 선임연구원
2004년 ~ 현재 국가보안기술연구소 연구원

윤 영 태

1995년 충남대학교 컴퓨터과학과(공학사)
1997년 현대전자 정보시스템 사업본부
1995년 충남대학교 컴퓨터과학과(이학석사)
2006년 충남대학교 컴퓨터과학과(이학박사)
1999년 ~ 현재 국가보안기술연구소 선임연구원