DLL Injection

REVERSING

CreateRemoteThread()를 중점으로 DLL Injection 에 대하여 설명

[Rnfwoa] 김경민

Rnfwoa

목차

그림 / 표 목차

1 개요

최근에 컴퓨터와 사람과의 관계는 결코 분리할 수 없는 관계가 되었다. 이러한 관계를 악용하고자 예로부터 악성코드들이 존재하였으며 이들은 대부분의 사람들에게 피해를 입힌다. 하지만 컴퓨터 시스템이 진화함에 따라 이러한 악성코드 역시 진화해오고 있는데 악성코드 제작자들은 악성코드를 은닉하기 위해 일반 윈도우 환경에 악성코드를 숨기는 기법들을 개발했다.

가장 일반적인 위장 실행 기법이 바로 DLL Injection 기법이다. 이는 주로 LoadLibrary 와 CreateRemoteThread API 를 이용하여 다른 프로세스에 해당 DLL 을 주입하는 방식이다. 이에 대하여 많은 문서들이 있지만 사용되는 함수 인자에 대한 설명이 구체적이지 않거나 정리가 너무 난잡하게 되어있어 직접 만들면서 공부도 다시하고자 만들었다.

인젝션에는 다양한 방법들이 있지만 이 문서에서는 CreateRemoteThread 를 이용한 원격 스레드 생성을 통한 DLL Injection 에 초점을 맞추고 있다. 다른 방법들의 경우는 다른 문서들을 참고하는 것이 더욱 좋을 것 같다.

또한 DLL 인젝션에 사용될 DLL 의 악성코드는 주로 DllMain()의 위치에 있으며 이는 프로세스에 해당 Dll 이 로드되면 자동적으로 실행이 되는 부분이다. 그렇기에 Dll 에서 다른 Export 함수를 정의하는 것이 아니라 인젝션되는 악성코드들은 DllMain()의 부분에 기록되어 있다. 이제 이러한 DLL Injection 에 대하여 우리는 살펴볼 것이다.

2 DLL (Dynamic Link Library)

DLL 은 흔히 '동적 연결 라이브러리'라고 한다. 예전 16 비트 DOS 시절에는 DLL 의 개념이 없고 그냥 'Library'만 존재하였다. 그로 인하여 함수를 사용할 때 컴파일러는 해당 라이브러리에서 해당 함수의 Binary 코드를 그대로 가져와서 프로그램에 포함시켜야 했다. 하지만 Windows OS 에서는 멀티 태스킹을 지원하기에 이러한 방식이 비효율적이 되었고, 이렇기에 효율적인 방법이 요구 되었다. 그래서 Windows OS 설계자들은 아래와 같은 DLL 개념을 생각했다.

리눅스와 유닉스의 경우 아직도 정적 링크가 일반적으로 사용된다.

- 프로그램에 라이브러리를 포함시키지 않고 별도의 파일로 구성
- 구성된 별도의 라이브러리들을 필요할 때마다 불러 사용
- 로딩된 DLL 의 코드, 리소스는 메모리 맵핑 기술로 여러 Process 에서 공유해서 사용

이러한 방식을 사용할 경우 프로그램이 해당 함수 코드를 직접 포함하지 않기에 메모리에 적재되는 용량을 줄일 수가 있다. 또한 해당 라이브러리가 변경될 때마다 프로그램을 통째로 교체하는 것이 아니라 해당 라이브러리만 교체를 하면 되기에 편의성을 제공한다.

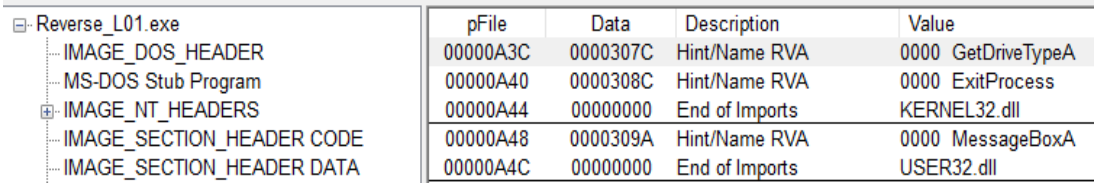


그림 1.Reverse_L01.exe 에서 사용될 DLL 목록

위의 그림의 경우 프로그램이 메모리에 로딩될 때, PE Loader 에 의하여 해당 DLL 들이 메모리에 올라가게 되며 프로세스가 해당 라이브러리의 함수를 호출할 수 있게 된다.

3 AppInit_DLLs

악성코드 제작자는 AppInit_DLLs 라고 불리는 특별한 Registry 경로를 통하여 자신들의 DLL 을 Injection 시키기도 하며 지속적으로 유지되게 할 수 있다. 이 레지스트리는 User32.dll 을 로드하는 모든 프로세스에서 해당 DLL 을 로딩하게 한다.

HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs

위의 경로에 Injection 시키고자 하는 DLL 의 경로를 넣어주고 재부팅하면 Windows 는 재부팅하면서 실행되는 User32.dll 을 로드하는 프로세스들에게 해당 DLL 을 인젝션 시킨다.

그림 2. AppInit_DLLs

Windows XP 이후로 LoadAppInit_DLLs 라는 부분이 생겼는데 Windows XP 까지는 APPInit_DLLs 의 부분에 경로만 입력해주면 되었지만, XP 이후로는 LoadAppInit_DLLs 부분에 '1' 의 값을 넣어주어야 해당 DLL 이 인젝션 된다.

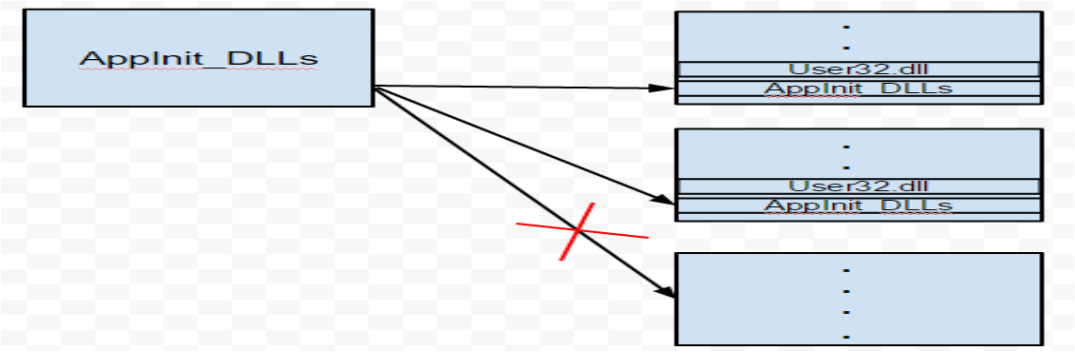


그림 3.User32.dll 을 로드하는 프로세스에게만 인젝션

이 방법을 사용할 경우 많은 프로세스에 인젝션되기 때문에 자칫 큰 CPU 소비를 가지고 올 수도 있으므로 감염자가 인지할 확률이 높다. 그렇기에 이를 통한 악성코드의 경우 특정 프로세스에게서만 동작하도록 설계되는 것이 보통이다. 이러한 선별적 작업으로 인하여 부하를 줄일 수가 있으며 인지될 확률을 줄일 수가 있다.

아래는 myhack2.dll 이 많은 프로세스에 로딩된 것을 확인할 수가 있다. 이 예제 DLL 의 경우 아무 작업도 하지 않아 아무런 부작용이 없지만, 아무 기능이나 포함된 DLL 을 로딩 시킬 경우 BSOD 가 나타날 수도 있으며 프로세스마다 오류창이 나타날 수도 있다.

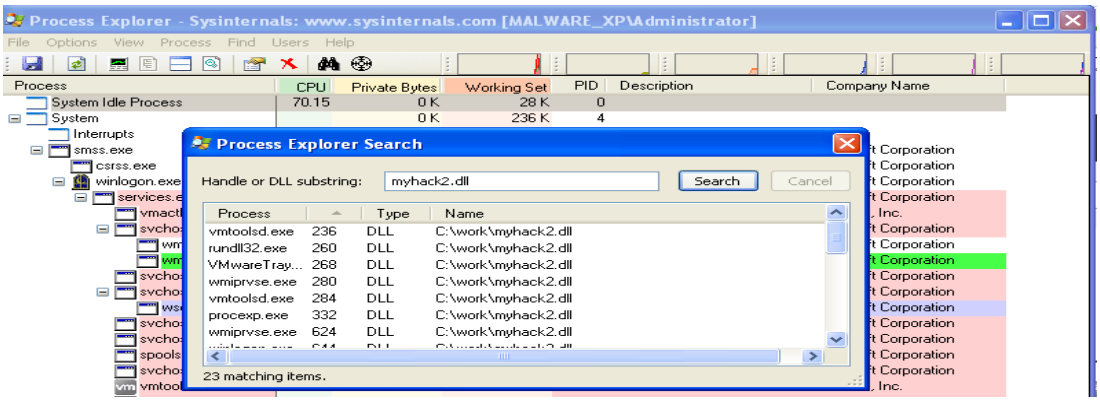


그림 4. AppInit_DLLs_myhack2.dll

.

이러한 방법을 쓰는 악성코드의 경우 Registry 를 비교하면 바로 탐지할 수가 있다. 해당 영역의 Registry 는 기본적으로 비어 있으며, 만약 어떠한 악성코드로 인하여 추가 되었다고 하더라 해당 DLL 의 경로가 나타나기에 제거 또한 쉽다고 할 수 있다.

4 원격 스레드 생성

이제 DLL Injection 을 구현하는 방법 중 가장 유명한 방법인 CreateRemoteThread() API 를 이용하는 방식에 대하여 알아볼 것이다. 우선 CreateRemoteThread 에 대해 알아보자. 아래는 CreateRemoteThread 에 대한 MSDN 의 문서이다. 함수명과 같이 다른 프로세스에서 실행될 스레드를 생성하는 것이다.


CreateRemoteThread function

Creates a thread that runs in the virtual address space of another process.

Use the CreateRemoteThreadEx function to create a thread that runs in the virtual address space of another process and optionally specify extended attributes.

그림 5. MSDN_CreateRemoteThread

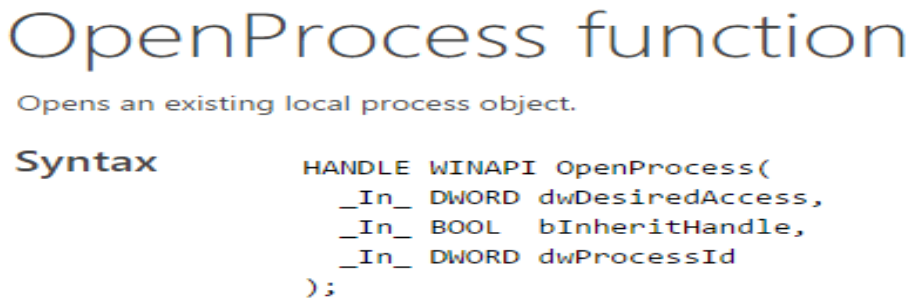
DLL Injection 은 LoadLibrary 를 호출하는 원격 프로세스에 코드를 삽입해 해당 프로세스에 DLL 을 강제적으로 로드하게 만드는 것이다. 감염된 프로세스가 해당 DLL 을 로드하면 운영체제는 자동으로 DllMain() 함수를 호출하게 된다. 이러한 인젝션을 하기 위해서는 다음과 같은 몇 단계의 차례가 필요하다.

1. DLL 을 인젝션 시킬 해당 프로세스의 핸들을 구해야 함.
2. 인젝션 시킬 DLL 의 이름을 넣어주어야 하므로 이름을 넣어줄 공간을 할당.
3. 할당된 공간에 해당 DLL 의 Name 을 기록해야 함.
4. Kernel32.dll 의 LoadLibrary 의 주소를 구해야 함..
5. 위에서 구한 것들을 가지고 CreateRemoteThread 를 통하여 원격 스레드를 생성

그림 6. DLL Injection 과정

4.1 핸들 확보

대상 프로세스 내부에 DLL 을 인젝션하기 위해서는 우선 목표 프로세스의 핸들을 확보해야 한다. 대상 핸들이 있어야만 그 프로세스를 다룰 수 있기에 우리는 해당 핸들의 확보를 위하여 해당 프로세스의 PID 를 이용할 것이며 이를 통해 OpenProcess API 를 호출할 것이다.


OpenProcess function

Opens an existing local process object.

Syntax

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

그림 7. MSDN_OpenProcess API

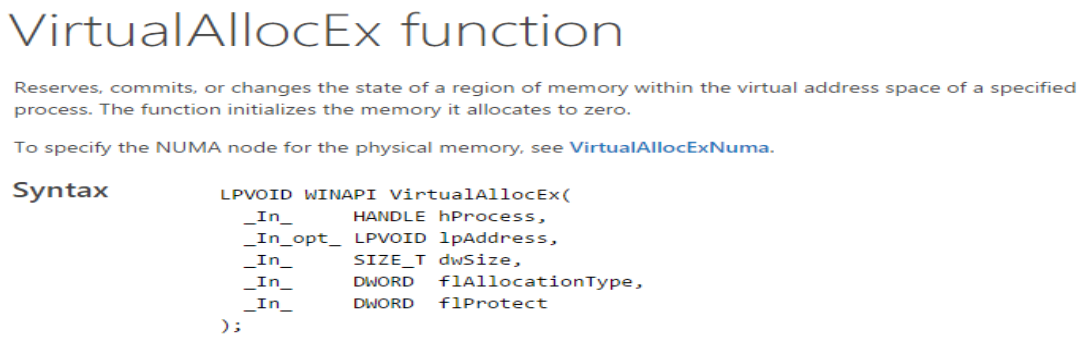
dwDesireAccess 를 통하여 권한을 설정하고 bInheritHandle 을 통하여 상속여부를 결정하며 마지막의 dwProcessId 에 대상 프로세스의 PID 를 넣어주므로 대상 프로세스의 핸들을 구할 수 있다.

핸들에 대한 권한으로 PROCESS_ALL_ACCESS(0x1F0FFF) 를 넣어주므로 핸들에 대한 모든 접근 권한을 갖게 된다. 상속 여부에 True 를 입력할 경우 이 프로세스에 의해 생성된 프로세스는 핸들을 상속하게 될 것이며 False 를 입력할 경우 상속하지 않는다. 여기서 우리는 False 를 입력할 것이다. 그리고 마지막 인자로는 해당 PID 를 주어야 한다.

```
Hproc = OpenProcess(0x1F0FFF, False, PID)
```

4.2 공간 할당

해당 DLL 을 로드시키게 하기 위해서는 LoadLibrary 함수와 그 DLL 의 이름이 필요하다. 이러한 인젝션 시킬 DLL 의 이름을 프로세스에 넣기 위하여 우리는 공간을 할당해야 한다. 이를 위하여 우리는 VirtualAllocEx API 를 사용할 것이다.



VirtualAllocEx function

Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero.

To specify the NUMA node for the physical memory, see VirtualAllocExNuma.

Syntax

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

그림 8. MSDN_VirtualAllocEx API

VritualAllocEx 함수는 원격 프로세스에 대한 핸들이 전달된다면 원격 프로세스에서 공간을 할당한다. hProcess 에는 우리가 위에서 구했던 대상 프로세스의 핸들을 넘겨주어야하며 lpAddress 에는 우리가 할당을 원하는 주소를 입력받는다. dwSize 에는 얼마큼의 크기를 할당 받을지를 정하는 것이며 flAllocationType 은 할당 유형을 결정 짓고 마지막 flProtect 는 보호(권한)에 대하여 설정을 한다.

hProcess 에는 위에서 구한 핸들('hproc'라 하자.)을 넘겨주며 lpAddress 에 NULL 값이 입력될 경우 함수가 알아서 할당할 위치를 정해줄 것이다. dwSize 에는 우리가 입력할 DLL 의 길이만큼 공간을 할당 받을 것이기에 len(dll_file)만큼이며 할당 유형으로 **MEM_COMMIT 과 MEM_RESERVE** 을 넣어주기 위하여 우리는 0x3000 으로 설정할 것이다. 마지막 보호(권한)는 **PAGE_READWRITE** 을 위하여 0x04 의 값을 넣어준다.

```
Virtual_addr = VirtualAllocEx(hproc, None, len(dll_file), 0x3000, 0x04)
```

4.3 DLL Name 기록

위에서 할당한 대상 프로세스의 공간에 우리가 인젝션하고자 하는 DLL 의 이름을 기록할 것이다. 이를 위하여 우리는 WriteProcessMemory API 를 사용할 것이다.

WriteProcessMemory function

Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Syntax

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

그림 9. MSDN_WriteProcessMemory API

hProcess 에는 대상 프로세스의 핸들을 넣어주고 lpBaseAddress 에는 어느 주소에 기록을 할 것인지를 넣어주어야 한다. lpBuffer 에는 쓰고자 하는 내용, nSize 는 쓰고자 하는 내용의 크기, 마지막 lpNumberOfBytesWritten 는 기록된 프로세스 변수의 포인터를 설정해주는 것이다.

우리는 hProcess 에는 위에서 구했던 hProc 를 줄것이며 BaseAddress 에는 바로 위에서 VirtualAllocEx 를 통하여 할당된 공간(virtual_addr 이라 하자.)을 인자로 넣어주어야한다. 기록할 내용인 Buffer 에는 해당 DLL 파일의 이름을 넣어주며 그 DLL 길이 만큼 nSize 를 설정하며 마지막 인자에는 NULL 을 넘겨줄 것이다.

```
WriteProcessMemory(hproc, virtual_addr, Dll_file, len(Dll_file), None)
```

4.4 LoadLibrary 함수 주소 구하기

** LoadLibrary API*

Loads the specified module into the address space of the calling process

-MSDN

강제로 로딩시키고자 하는 DLL 의 이름이 대상 프로세스에 기록되었다. 이제 LoadLibrary 를 통하여 해당 DLL 을 대상 프로세스가 로딩하게끔 만들어야한다. 하지만 그러기 위해선 해당 함수인 Kernel32.dll 에 있는 LoadLibrary 의 주소를 먼저 구해야 한다.

Kernel32.dll 모듈 핸들을 먼저 구한 후 해당 모듈에서 함수의 주소를 구할 것이다. 이를 위하여 우리는 GetModuleHandle 과 GetProcAddress API 를 사용할 것이다.

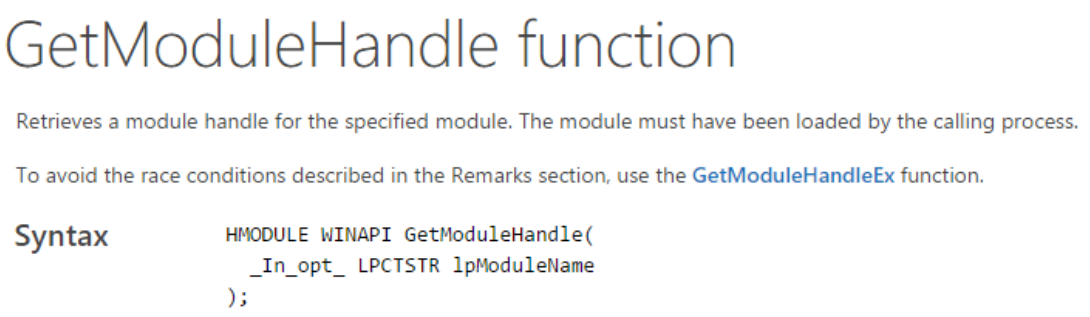
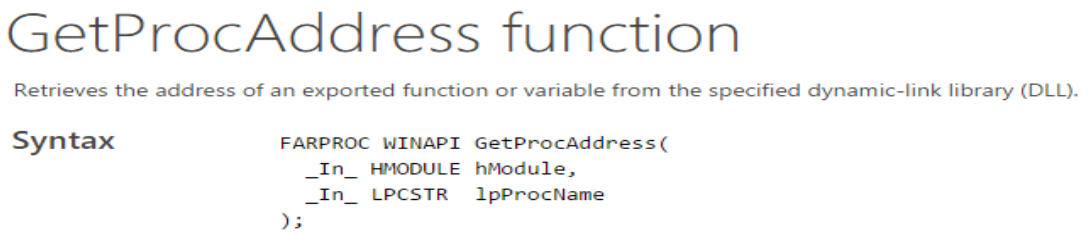


그림 10. MSDN_GetModuleHandle API

GetModuleHandle 함수는 해당 함수에 대한 모듈 핸들을 갖게 해준다. 여기서 우리는 LoadLibrary 함수를 위하여 Kernel32 모듈이 필요하므로 이를 인자로 넣어주면 된다.

```
Hmod = GetModuleHandleA('kernel32')
```



GetProcAddress function

Retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).

Syntax

```
FARPROC WINAPI GetProcAddress(
  _In_ HMODULE hModule,
  _In_ LPCSTR  lpProcName
);
```

그림 11. MSDN_GetProcAddress API

GetProcAddress 는 해당 모듈에서 함수의 주소를 찾게 해준다. 첫 번째 인자인 hModule 에는 모듈의 핸들을 넣어주어야하며 여기서 우리는 위의 GetModuleHandle 에서 구한 핸들(hmod 라고 하자.)을 넣을 것이며, lpProcName 은 DLL 을 로딩시키기 위한 LoadLibraryA 를 넣어줄 것이다.

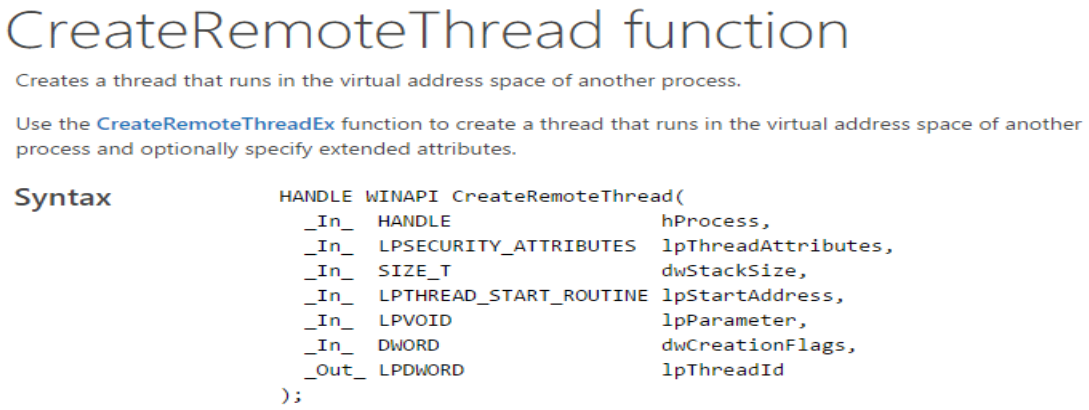
Load_addr = GetProcAddress(hmod, 'LoadLibraryA')

여기서 하나 의문점이 생길 수도 있다. LoadLibrary 함수는 DLL 을 로딩하는 것인데 우리는 지금 인젝터에서 해당 함수의 주소를 구하고 있다. 보통 DLL 이 로딩될 때 ImageBase 가 100000 으로 A .dll 이 로드되고 B.dll 이 로드 될때 해당 위치에 이미 로드된 DLL 이 있을 경우 Relocation 을 통하여 다른 주소를 찾아서 로드된다. 그러므로 인젝터 프로세스에서 해당 함수 주소를 구해도 대상 프로세스의 DLL 의 위치에 따라 실행이 안될 것이다.

하지만 Microsoft 는 OS 핵심 DLL 파일들의 ImasgeBase 를 미리 정리해 놓았으며 이는 절대 겹치지 않도록 되어있다. 다시 말해 DLL Relocation 이 발생하지 않는데 이러한 특성으로 인해 인젝터에서 해당 함수의 주소를 구해도 지장이 없는 것이다.

4.5 원격 스레드 생성

DLL Injeciotn 을 위한 마지막 순서로 CreateRemoteThread API 를 이용해 원격의 스레드를 생성해야 한다. 이 스레드로 대상 프로세스에서 LoadLibrary 를 이용해 해당 DLL 을 로드하게 만드는 것이다.



CreateRemoteThread function

Creates a thread that runs in the virtual address space of another process.

Use the CreateRemoteThreadEx function to create a thread that runs in the virtual address space of another process and optionally specify extended attributes.

Syntax

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

그림 12. MSDN_CreateRemoteThread API

-hProcess 에는 대상 프로세스의 핸들을 넣어야 한다. OpenProcess 를 통해 구한 hproc 를 넣어준다.
-lpThreadAttribute 는 SECURITY_ATTRIBUTES 로써 NULL 일 경우 스레드의 보안 설정이 Default 가 되며 핸들이 상속되지 않는다. 따라서 우리는 NULL 을 넣을 것이다.
-dwStackSize 에는 스택의 초기 크기를 설정하는 것이며 0 으로 설정할 것이다.
-lpStartAddress 는 스레드에 의해 실행될 함수나 명령어의 주소를 넣어야한다. 여기선 LoadLibrary 의 주소를 가리키는 load_addr 을 넣는다.
-lpParameter 는 스레드의 내용이 전달될 위치를 넣어야 한다. Virtual_aadr 을 넣을 것이다.
-dwCreationFlags 는 스레드의 속성을 설정한다. 별도로 설정하지 않을 것이기에 0
-lpThreadId 는 스레드의 ID 를 설정하는 인자이지만 우리는 NULL 로 한다.

```
CreateRemoteThread(hproc,None,0,load_addr,virtual_addr,0,None)
```

아래는 직접 인젝터를 만들어서 실습을 해본 사진이다. 아래의 사진을 보면 카카오톡이라는 프로세스에 msgbox.dll 이라는 메시지를 출력하는 박스를 띄우는 단순한 dll 을 인젝션할 것이다. CMD > python Dll_injector.py <PID> Dll_patch 과 같이 입력한다.

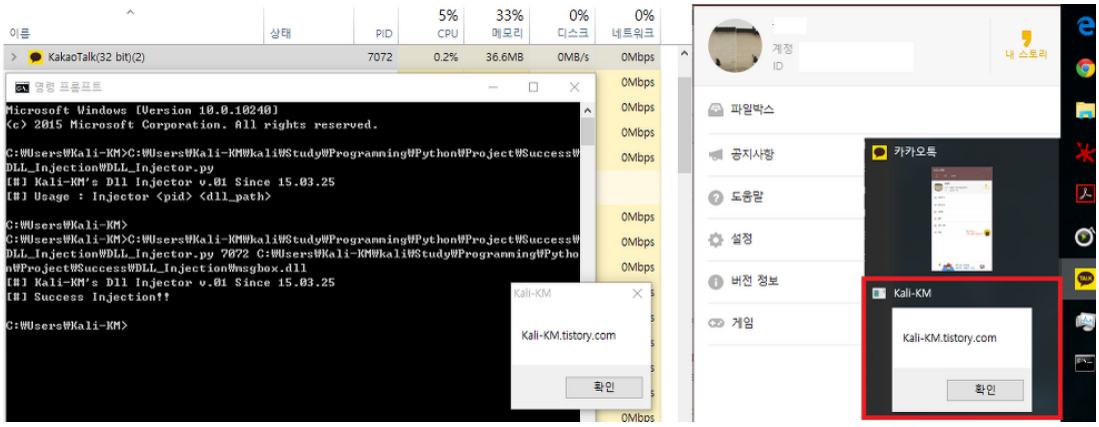


그림 13. DLL Injection - KaKaoTalk 1

인젝션에 성공할 경우 메시지 박스가 나오면서 이게 오른쪽에서와 같이 카카오톡과 같은 소속임을 알 수가 있다. 그리고 아래에서와 같이 ProcExp 를 통해 확인하면 카카오톡이 해당 DLL 을 로드한 것을 확인할 수가 있다.

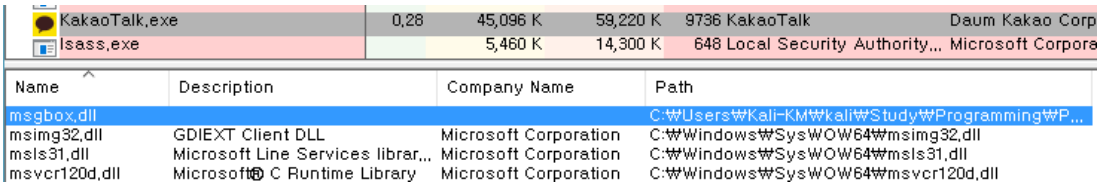


그림 14. DLL Injcetion - KaKaoTalk 2

이렇게 인젝션된 DLL 은 해당 프로세스와 같은 권한을 갖는다. 이러한 방법을 통해 악성코드들은 DLL 인젝션을 시도하여 다른 프로세스에서 악성 코드를 실행하도록 한다. 그리고 그 자신의 모습은 삭제되므로 흔적을 최소화하고자 할 것이다.

```
# Dll_Injector.py
# Kali-KM                    Blog: kali-km.tistory.com
from ctypes import *
import sys,ctypes


def Injection(pid,dll_file):
        hproc = kernel32.OpenProcess(0x1F0FFF,False,pid)
        virtual_addr = kernel32.VirtualAllocEx(hproc,None,len(dll_file),0x3000,0x04)
        kernel32.WriteProcessMemory(hproc,virtual_addr,dll_file,len(dll_file),None)
        hmod=kernel32.GetModuleHandleA('kernel32')
        load_addr=kernel32.GetProcAddress(hmod,'LoadLibraryA')

        if not kernel32.CreateRemoteThread(hproc,None,0,load_addr,virtual_addr,0,None):
                print "[#] Failed Injection.."
        else:
                print "[#] Success Injection!!"

print"[#] Kali-KM's Dll Injector v.01 Since 15.03.25"

if not sys.argv[1:]:
        print"[#] Usage : Injector <pid> <dll_path>"
        sys.exit(0)

kernel32 = windll.kernel32
pid=int(sys.argv[1])
dll_file=str(sys.argv[2])
Injection(pid,dll_file)
```

그림 15. DLL Injector Code - Python

5 결론

이렇게 DLL 인젝션을 살펴보았다. 사실 Code Injection 또한 매우 유사한 방식으로 진행이 되는데 단지 LoadLibrary 함수가 필요하지 않기에 더욱 코드의 이식이 쉽지만 프로세스의 안정을 위하여 최적화된 코드 또한 필요하다. 악성코드를 분석할 때 위와 같은 API 들이 나올 경우 DLL Injection 을 의심할 수가 있으며 어떠한 DLL 이 인젝션 되는지 확인 후 해당 DLL 을 분석할 수 있어야 한다.

이제 CreateRemoteThread 를 이용하여 DLL 을 인젝션하는 방법에 대하여 이해를 했을 것이다. 위에 Python 을 통한 DLL Injector 의 코드를 같이 넣긴 하지만 본인이 직접 MSDN 등을 찾아보며 해보는 것이 더 깊숙히 이해가 가능할 것이다.

참고

http://kali-km.tistory.com/

리버싱 핵심 원리 (악성 코드 분석가의 **리버싱** 이야기)

이승원 저 인사이트 2012.09.30.

실전 악성코드와 멀웨어 분석 (Practical Malware Analysis)

마이클 시코스키, 앤드류 호닉 저 여성구, 구형준 외 1 명 역 에이콘출판 2013.10.29.

https://snipt.net/raw/9ea72eda2b96eff2d0d60956bd216aaa/?nice

https://msdn.microsoft.com/

http://hisjournal.net/doc/How_Does_the_DLL_Injection.pdf

http://www.reversecore.com/ http://john09.tistory.com/53

http://gmng.tistory.com/40 http://hackersstudy.tistory.com/75

http://kuaaan.tistory.com/112