DLL INJECTION

using NtCreateThreadEx, RtlCreateUserThread

[TECHNOLOGY REPORT]

Dec, 2017

By WOODONGGYU

목 차

1. 개요

1-1. 배경

해당 문서는 "[WOODONGGYU, 20171213] DLL Injection using CreateRemoteThread" 문서와 내용적인 면에서는 크게 다를 바 없다. 다만 Windows Vista 이 후부터는 보안 정책이 바뀌면서 기존의 CreateRemoteThread 함수를 이용해 DLL Injection 하는 것이 불가능해졌다. 변경된 부분은 아래와 같다.

> [1]. OpenProcess() 호출 시 호출하는 프로세스와 같거나 더 낮은 Integrity Level 의 접근 권한만 획득 가능
>
> [2]. Session Isolation 정책에 따라 CreateRemoteThread() 는 같은 세션에 해당하는 프로세스들에만 DLL Injection 이 가능하다.

이에 해당 문서에서는 위와 같은 접근 통제 정책과 Session Isolation 우회를 통한 DLL Injection 방법에 대해 다루도록 한다.

1-2. 동작 과정

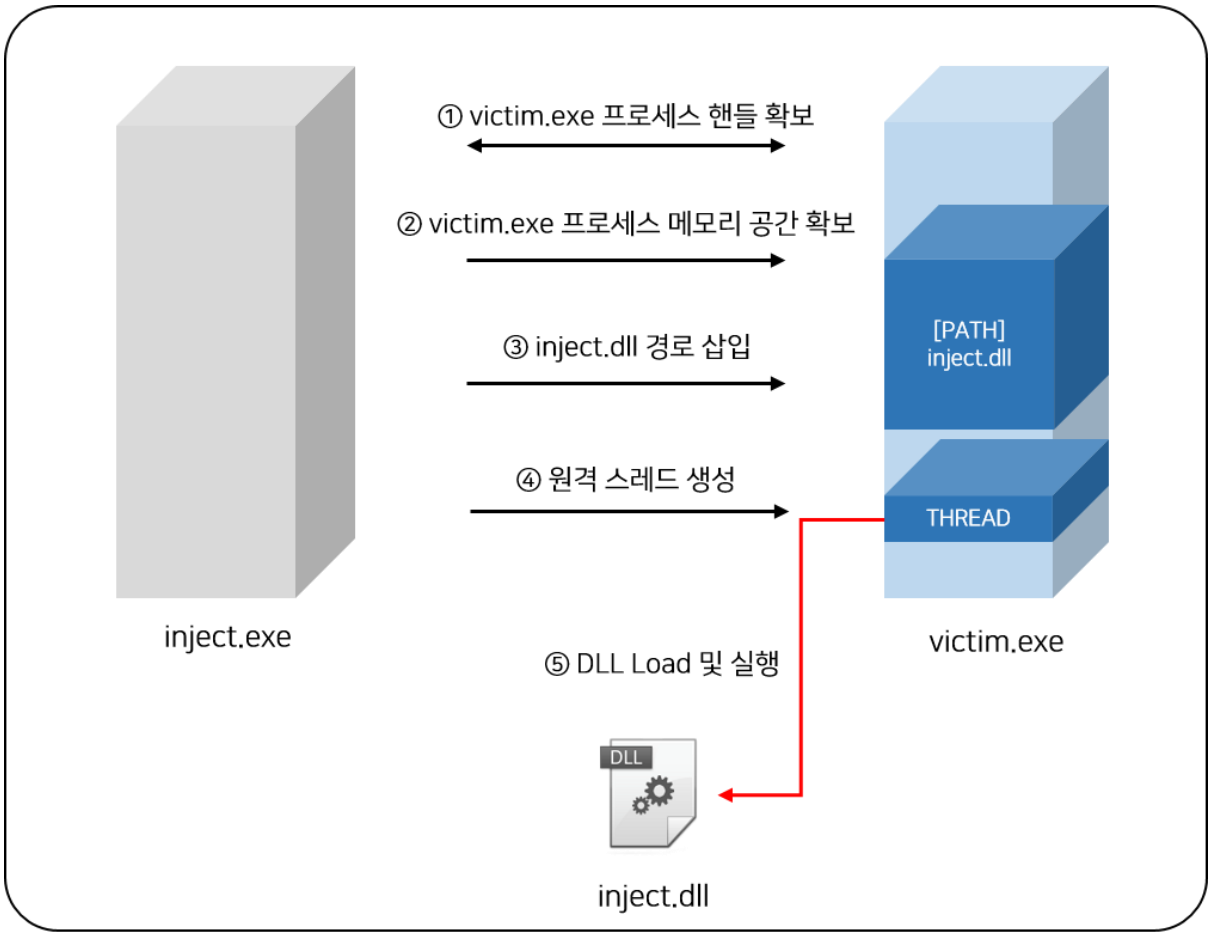


Figure 1. 동작 과정

2. 분석 및 구현

2-1. 우회

간단했던 윈도우 XP 와 달리 Vista 이 후부터는 많은 변화가 생겼다. 이에 대한 자세한 내용은 생략하고 우회할 수 있는 방법에 대해 설명하도록 하겠다.

기본적으로 우리는 DLL Injection 을 수행하기 위해 가장 필요한 것은 대상 프로세스의 핸들이다. OpenProcess 함수를 사용해 필요한 접근 권한과 함께 핸들을 얻어올 수 있었다. 하지만 Vista 이 후부터는 OpenProcess 함수 호출 시, 호출하는 프로세스와 같거나 더 낮은 Integrity Level 을 가진 프로세스에 대해서만 접근 권한을 얻어올 수 있다.

만약 현재 프로세스가 관리자 권한을 가지고 있다고 해도 시스템 서비스들은 High Integrity Level 보다 높은 System Integrity Level 을 가지고 있다. 이를 우회하는 방법은 프로세스에 "SeDebugPrivilege" 권한을 주는 것이다. 이 디버그 권한을 갖는다면 이러한 접근 통제 정책을 우회할 수 있다. 하지만 SeDebugPrivilege 권한은 관리자 계정에게만 주어지기 때문에 관리자 권한(High Integrity)이어야 SeDebugPrivilege 권한을 활성화시킬 수 있다.

접근 통제 정책을 우회하여 원하는 권한을 가진 핸들을 받았다고 가정하더라도 Session Isolation 정책으로 인해 기존의 CreateRemoteThread 함수를 통한 DLL Injection 이 불가능하다. 이 정책으로 인해 CreateRemoteThread 함수는 같은 세션에 해당하는 프로세스들에만 DLL Injection 이 가능하다. 즉, 보통 시스템 서비스들은 세션 0 을 갖고 각 사용자들은 다른 숫자들을 부여 받는다. 이에 주로 DLL Injection 대상이 되어왔던 시스템 서비스들은 더 이상 DLL Injection 이 통하지 않는다. 하지만 NtCreateThreadEx(), RtlCreateUserThread() 등의 네이티브 API 함수를 사용하면 Session Isolation 정책에 대해 쉽게 우회할 수 있다.

2-2. 동작 분석

DLL Injection 을 위한 동작 순서(위 -> 아래)에 따른 사용되는 함수와 인자들에 대해 설명한다.

비활성화된 SeDebugPrivilege 권한을 활성화하여 System 권한을 가진 프로세스에 접근이 가능하다. 즉, 관리자 계정의 경우 High Integrity Level 임에도 불구하고 더 높은 권한인 System Integrity Level 에 접근할 수 있다.

```
int set_privileges(void) {

    TOKEN_PRIVILEGES tPriv = { 0 };
    HANDLE hToken = NULL;

    LUID luid = { 0 };

    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken)) {
        if (LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &luid)) {
            tPriv.PrivilegeCount = 1;
            tPriv.Privileges[0].Luid = luid;
            tPriv.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

            if (AdjustTokenPrivileges(hToken, FALSE, &tPriv, sizeof(TOKEN_PRIVILEGES), (PTOKEN_PRIVILEGES)NULL, (PDWORD)NULL) == 0) {
                printf("[!] AdjustTokenPrivileges Error. [%d]\n", GetLastError());
                return -1;
            }
        }
    }

    return 1;
}
```

Figure 2. SeDebugPrivilege 활성화

· *OpenProcess – 프로세스의 HANDLE 값을 얻어온다.*

Return Value : 지정된 프로세스의 HANDLE 값.

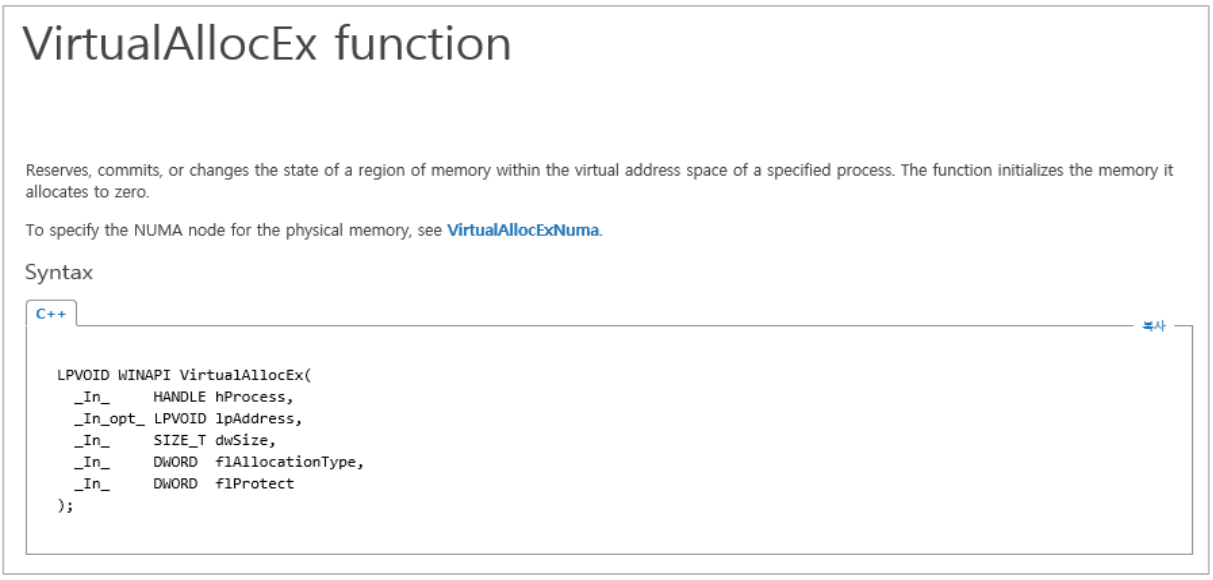
- dwDesiredAccess : PROCESS_ALL_ACCESS(프로세스 객체에 대한 모든 가능한 권한 확보)
- bInheritHandle : FALSE(프로세스가 핸들 상속하지 않음)
- dwProcessId : PID(HANDLE 값을 얻어올 프로세스 식별자)

Figure 3. OpenProcess Function

· *VirtualAllocEx – 지정된 프로세스의 메모리 할당한다.*

Return Value : 메모리에 할당 된 공간의 주소.

- hProcess : 지정된 프로세스 HANDLE 값
- lpAddress :, NULL(비어있는 공간 할당)
- dwSize : NULL(하나의 페이지 크기 지정)
- flAllocationType : MEM_COMMIT(지정 예약된 메모리에 대한 할당)
- flProtect : PAGE_EXECUTE_READWRITE(읽기, 쓰기 및 실행 가능)



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

C++

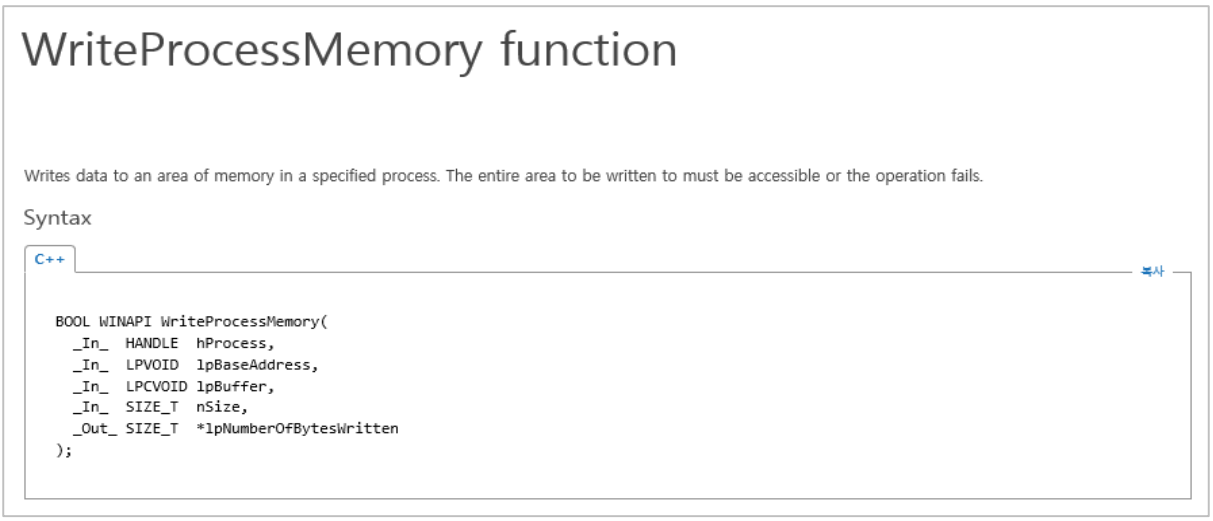
```
LPVOID WINAPI VirtualAllocEx(
  _In_     HANDLE hProcess,
  _In_opt_ LPVOID lpAddress,
  _In_     SIZE_T dwSize,
  _In_     DWORD  flAllocationType,
  _In_     DWORD  flProtect
);
```

Figure 4. VirtualAllocEx Function

· *WriteProcessMemory – 지정된 프로세스의 할당된 메모리 공간에 데이터를 쓴다.*

Return Value : 정상적으로 쓰여졌다면 0 이 아닌 값.

- hProcess : 지정된 프로세스의 HANDLE ㄷ 값
- lpBaseAddress : VirtualAllocEx Return 값(지정된 프로세스의 데이터가 쓰여지는 메모리 공간의 주소)
- lpBuffer : 지정된 공간에 쓰일 데이터의 주소
- nSize : 쓰여질 데이터의 바이트 수
- *lpNumberOfBytesWritten : NULL(삽입한 데이터의 크기를 저장하지 않음)



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

C++

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

Figure 5. WriteProcessMemory Function

· *NtCreateThreadEx, RtlCreateUserThread– 지정된 프로세스에서 실행되는 Thread 를 생성한다.*

```
typedef DWORD(WINAPI *PFNTCREATETHREADEX) (
        PHANDLE                                 ThreadHandle,
        ACCESS_MASK                             DesiredAccess,
        LPVOID                                  ObjectAttributes,
        HANDLE                                  ProcessHandle,
        LPTHREAD_START_ROUTINE  lpStartAddress,
        LPVOID                                  lpParameter,
        BOOL                                    CreateSuspended,
        DWORD                                   dwStackSize,
        DWORD                                   dw1,
        DWORD                                   dw2,
        LPVOID                                  Unknown
);
```

Figure 6. NtCreateThreadEx Function

```
typedef DWORD(WINAPI *PFRtlCreateUserThread) (
        HANDLE                                  ProcessHandle,
        PSECURITY_DESCRIPTOR            SecurityDescriptor,
        BOOLEAN                                 CreateSuspended,
        ULONG                                   StackZeroBits,
        PULONG                                  StackReserve,
        PULONG                                  StackCommit,
        PTHREAD_START_ROUTINE           StartAddress,
        PVOID                                   Parameter,
        PHANDLE                                 ThreadHandle,
        PCLIENT_ID                              ClientId
);
```

Figure 7. RtlCreateUserThread Function

2-2. 구현 및 결과

소스코드는 아래의 링크를 통해 다운로드 및 테스트할 수 있다.

https://github.com/woodonggyu/injection-techniques/tree/master/NtCreateThreadEx%2C%20RtlCreateUserThread

테스트 결과는 아래와 같이 정상적으로 동작한 것을 확인할 수 있다.

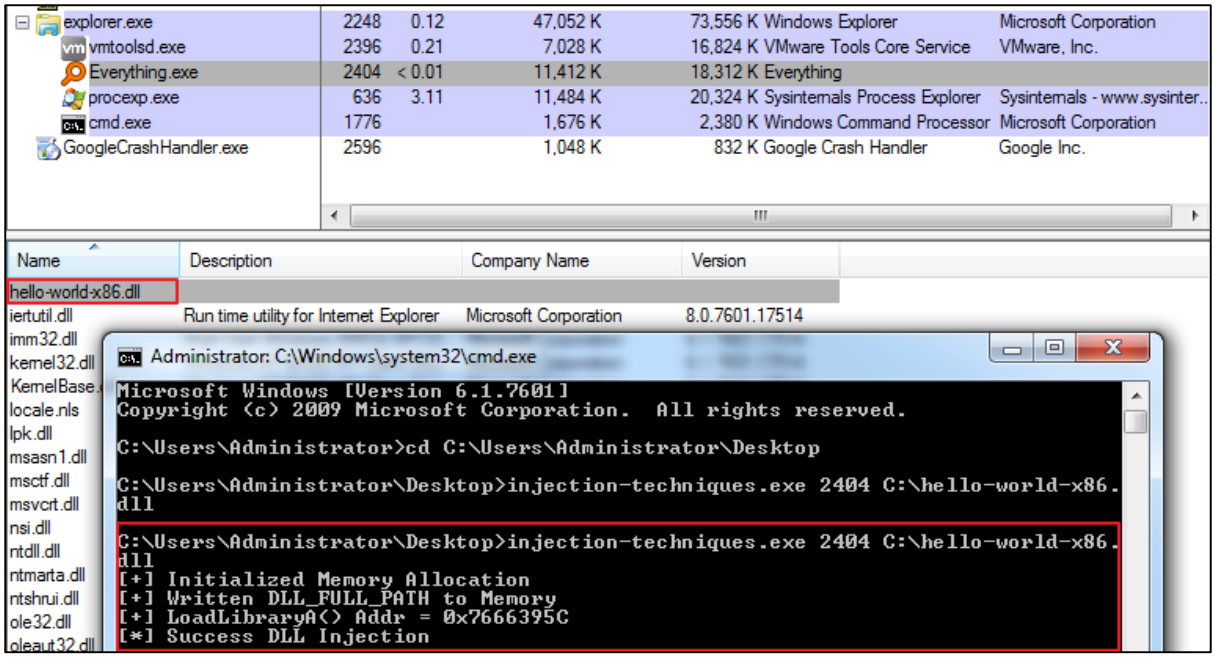


Figure 7. 실행 결과

2-3. 탐지 방안

LdrDllNotification 메커니즘을 이용하면 간단히 탐지할 수 있다. 이 메커니즘은 프로그래머가 등록한 CALLBACK 을 DLL 이 Load/Unload 될 때 호출해준다. 동작은 DLL 이 가상 주소 공간에 매핑이 되어 있지만 동적 링크가 이뤄지기 전에 Load 되는 Thread Context 안에서 호출된다. DLL Injection 탐지 흐름은 다음과 같다.

1. LdrRegisterDllNotification() API 를 통해 DLL Load/Unload 에 대한 CALLBACK 등록한다.
2. CALLBACK 에서 DLL Load 이벤트에 대한 현재 Thread 시작 주소 얻는다.
3. 현재 Thread 시작 주소가 LoadLibrary() 류의 함수인지 판별한다.
4. 만약, LoadLibrary() 류의 함수라면 현재 Load 중인 DLL 의 Entry Point 를 변경한다.
5. 변경할 Entry Point 를 프로세스 종료, 의미 없는 코드 삽입 등 여러가지 방법을 통해 차단 가능하다.

3. 기타

3-1. 레퍼런스

- *[Book] 리버싱 핵심 원리*
- *Injection 기법_20140417_공개 버전.pdf*
- *https://securityxploded.com/ntcreatethreadex.php*
- *https://docs.microsoft.com/en-us/windows/desktop/secauthz/access-tokens*