Windows API Hooking

Reversing

윈도우 운영체제에서 사용할 수 있는 API 후킹에 대해 알아보자

4/13/2016

By Kali KM

목차

그림

1 Intro

리버싱을 하는 데 있어 흔히 "Art of Reversing is an API Hooking"이라는 말과 같이 API 후킹은 리버싱의 꽃이라 일컬어진다. 어떤 윈도우 응용프로그램을 개발하기 위해서 우리는 다양한 종류의 언어[1]나 도구[2]를 사용할 수가 있다. 이런 언어나 도구를 사용하여 개발하는 방법은 다르더라도 결국, 개발된 프로그램의 내부로 들어가면 윈도우 운영체제가 제공하는 API 를 호출한다.

이러한 API 는 사용자 영역뿐만 아니라 커널 영역에서도 Native API 의 형태로 존재하기 때문에 API 후킹을 이해하는 것은 윈도우의 많은 부분을 조작할 수 있음을 의미한다. 따라서 이번 문서에서는 API 후킹에 대한 이해를 도모하며, 기본적인 후킹의 방법에 대해 이해하므로 다른 후킹 방법 또한 낯설지 않도록 하는 것이 목적이다.

단, 후킹을 진행하는 자세한 코드들은 포함하지 않고, 어느 부분을 어떠한 방식으로 후킹 하는지 중점적으로 살펴볼 것이다. 코드들의 경우 다른 좋은 소스가 많으므로 그러한 요소들을 찾아보면 좋을 것이다. 이제부터 API 후킹에 대해 알아보자.

[1] C 나 C++, C#과 같은 프로그래밍 언어

[2] Visual Studio 와 GCC 와 같은 컴파일 도구 등

2 Prior Knowledge

이번 장에서는 구체적인 API 후킹에 대하여 알아보기 전에 우리가 알고 있는 API 가 무엇인지 다시 한 번 정리해보고 API 후킹의 기본적인 개념에 대하여 알아볼 것이다.

2.1 What is an API?

API 란 Application Programming Interface 의 약자로 단어 자체만으로는 무슨 뜻인지 이해하기 힘들다. 쉽게 말해 운영체제가 응용프로그램을 위해 제공하는 함수의 집합으로 응용프로그램과 디바이스를 연결해주는 역할을 한다. 좀 더 구체적으로 응용프로그램이 메모리, 파일, 네트워크, 비디오, 사운드 등 시스템 자원을 사용하고 싶더라도 이들에 대해 직접 접근할 수가 없다. 이러한 시스템 자원은 운영체제가 직접 관리하며 보안이나 효율 등 여러 면에서 사용자 응용프로그램이 접근할 수 없도록 막아놓았기 때문이다.

따라서 사용자 응용프로그램이 시스템 커널에게 이러한 시스템 자원의 사용을 요청해야 하며, 이 방법이 바로 MS 제공한 Win32 API 를 이용하는 것이다. 즉 API 함수 없이는 프로세스, 스레드, 메모리, 파일, 네트워크, 레지스트리 등 시스템 자원에 접근할 수 있는 프로그램을 만들 수가 없다. 아래 그림은 32 비트 Windows OS 의 프로세스 메모리를 간략히 나타낸 그림이다.

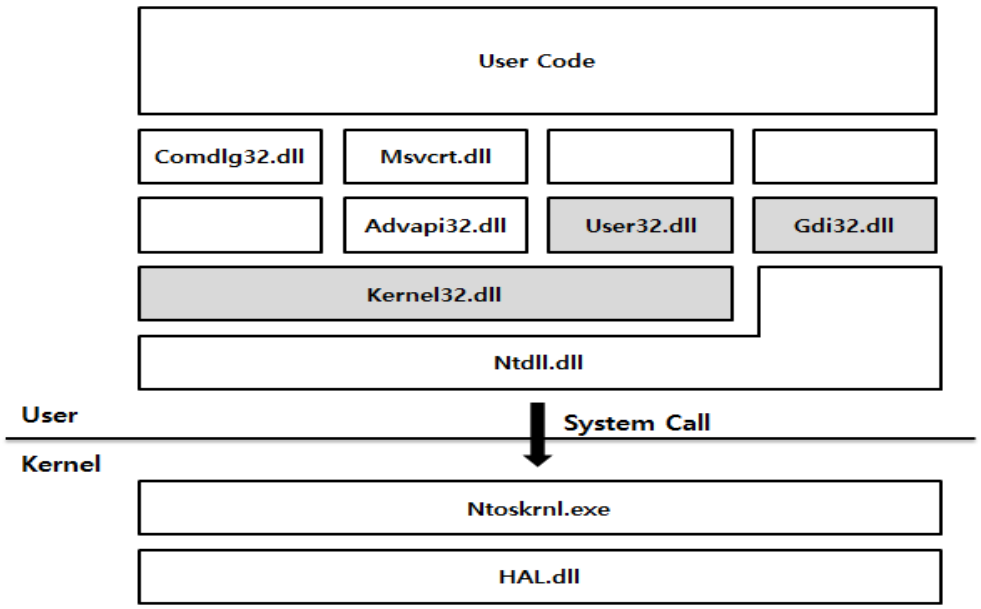

그림 1. User Mode & Kernel Mode

실제 응용프로그램 코드를 실행하기 위해서는 많은 DLL 이 로딩된다. 모든 프로세스는 기본적으로 kernel32.dll 이 로딩되며, kernel32.dll 은 ntdll.dll 을 로딩한다.

바로 이 ntdll.dll 의 역할이 사용자 모드에서 커널 모드로 요청하는 작업[3]을 수행한다. 이러한 방식을 통해 시스템 자원에 접근할 수 있게 되는 것이다.

2.2 What is an API Hooking?

API 후킹에 관해 이야기하기 전에 후킹에 대하여 먼저 알아보자. 후킹이란 이미 작성되어 있는 코드의 특정 지점을 가로채서 동작 방식에 변화를 주는 기술이라 할 수 있다. Hook 이라는 단어 자체가 낚싯바늘 같은 갈고리 모양을 가지는데 이를 통해 무엇인가를 낚아채는 것과 같이 컴퓨터에서도 무엇인가를 낚아채는 형태로 사용될 수 있다.

그렇다면 API 를 후킹 한다는 말은 무슨 뜻일까? 바로 Win32 API 가 호출되는 중간에서 가로채어 제어권을 얻어낸다는 것이다. 그렇다면 어느 시점에 가로채는지 궁금할 수가 있다. API 후킹에는 많은 기법이 존재하는데 이러한 분류가 주로 어떤 방식을 사용하는지, 그리고 바로 어느 지점에서 가로채는지에 따라 분류되므로 이에 대해서는 각 방식에서 자세히 알아볼 것이다. 기본적인 후킹의 개념은 아래의 그림과 같다.

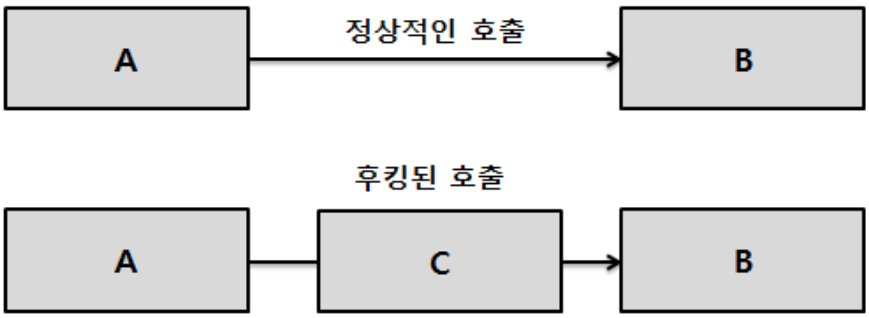

그림 2. 정상호출과 후킹된 호출

정상적인 경우 A 단계가 진행된 다음에 B 가 진행되어야 하지만, 후킹 된 호출의 경우 A 단계 다음에 B 가 진행되는 것이 아니라 C 가 진행된 뒤 B 가 진행된다. 이를 통해 A 의 요청을 조작하거나 특정한 조건에만 B 가 실행되도록 조작할 수가 있다. 위와 같은 방식으로 API 를 후킹 하면 그 함수의 기능을 사용하지 못하게 할 수도 있고 어떻게 사용하는지 감시만 할 수도 있다. 심지어 전혀 다른 내용으로 바뀌게끔 할 수도 있다.

[3] 시스템 호출(System Call)로 커널 영역 후킹에서 더 자세히 다룰 것이다.

3 User Mode Hooking

Windows 에서는 크게 사용자 모드(User Mode)의 후킹과 커널 모드(Kernel Mode)의 후킹으로 나누어진다. 이번 장에서는 사용자 영역에서 어떻게 API 호출이 이루어지는지, 그리고 어떻게 이들을 후킹할 수 있는지에 대하여 알아보자.

3.1 IAT Hooking

3.1.1 IAT(Import Address Table)

IAT 후킹은 IAT(Import Address Table)를 후킹 하는 것으로, IAT 는 쉽게 말해 해당 프로그램이 어떤 라이브러리에서 어떤 함수를 사용하고 있는지를 기술한 테이블이다. 예를 들어, 어떤 프로그램이 Kernel32.dll 의 GetDriveType API 를 호출한다면 해당 API 를 사용하기 위한 주소를 IAT 에서 참조할 수 있다.

pFile	Data	Description	Value
00000A50	0000307C	Hint/Name RVA	0000 GetDriveTypeA
00000A54	0000308C	Hint/Name RVA	0000 ExitProcess
00000A58	00000000	End of Imports	KERNEL32.dll
00000A5C	0000309A	Hint/Name RVA	0000 MessageBoxA
00000A60	00000000	End of Imports	USER32.dll

그림 3. PE View 로 본 IAT

IAT 후킹은 바로 이 IAT 에서 후킹하고자 하는 함수 주소를 내가 원하는 함수(이후 후킹 함수)로 교체하고, 후킹 함수에서 파라미터나 리턴 값을 조작하고 원래 함수를 호출하는 방법이 바로 IAT 후킹이다. 가장 쉬우면서도 안정적인 방법이라 일반적으로 자주 사용되는 후킹 방식이다. 그렇다면 일반적으로 API 가 호출되는 상황을 보자.

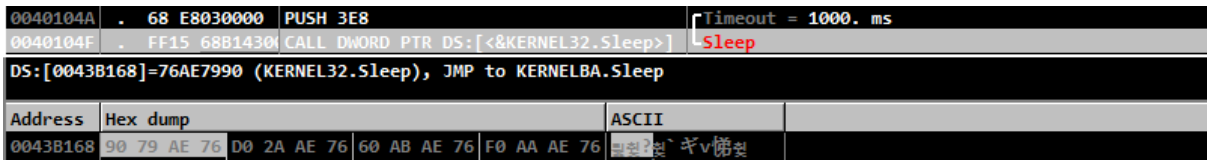

그림 4. Sleep API

Sleep()를 호출할 때 직접 호출하지 않고 0x43B168 주소에 있는 값을 가져와서 호출한다. 0x43B168 주소는 해당 프로그램의 IAT 메모리 영역으로 0x76AE7990 이라는 값이 존재하고 있다. 이제 이 값이 바로 해당 프로세스 메모리에 로딩된 Kernel32.dll 의 Sleep 함수의 주소이다.

그렇다면 왜 0x40104F 에서는 CALL 0x76AE7990 이라 하지 않고 다른 곳을 참조하는 하여 접근하는 것일까? 이는 운영체제 버전이나, 어떤 언어, 서비스 팩이냐에 따라 DLL 의 버전이 다르며, 해당 함수의 위치가 달라지기 때문이다. 모든 환경에서 Sleep() 함수 호출을 보장하기 위해 컴파일러는 Sleep()의 실제 주소가 저장될 위치(0x43B168)를 준비하고 CALL DWORD PTR DS:[43B168]을 적어두기만 한다. 그 후 파일이 실행되는 순간 PE Loader 가 0x43B168 위치에 Sleep() 함수의 주소를 입력해준다.

3.2.2 Hook

그렇다면 이러한 IAT 의 개념에 대해 알아보았지만, 대체 이 부분 중 어느 곳을 후킹해야 한다는 것인가 의문을 가질 수 있다. 위의 Sleep 함수를 예로, 사용자 모드에서 Sleep API 가 호출되는 과정은 아래의 그림과 같이 나타낼 수 있다.

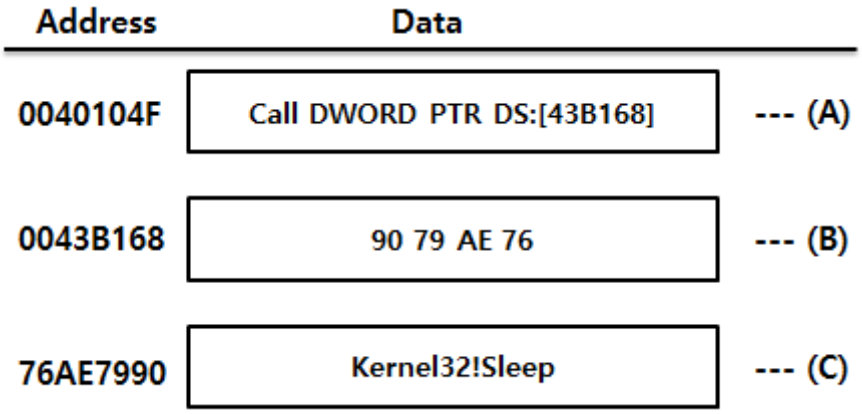

그림 5. 메모리에서 Sleep API

(A)는 3.2.1 에서 이야기한 바와 같이 해당 함수를 사용하기 위한 주소 공간을 만들어 놓은 것이다. 이렇게 만들어 놓은 주소에 프로그램이 실행되며 PE 로더가 Sleep 함수의 주소를 채워 넣는다. 여기서 만들어진 주소 공간 43B168 은 RVA 값이며, 이에 해당하는 파일 Offset[4]은 39168 이다. 39168 은 바로 IAT 의 Sleep 함수가 위치한 Offset 이다.

pFile	Data	Description	Value
00039164	0003B2A0	Hint/Name RVA	022F IsDebuggerPresent
00039168	0003B2B4	Hint/Name RVA	0349 Sleep

그림 6. Sleep API in IAT

만약 (A)에 존재하고 있는 값인 43B168 을 다른 값으로 바꾸면 어떻게 될까? 직접 Sleep 함수를 호출하는 부분에 IsDebuggerPresent API 의 IAT 주소를 넣어보자. 주소는

[4] RVA 는 메모리에서의 상대주소이며 Offset 은 파일에서의 위치이다.

위 그림에서와 같이 Offset 39164 이므로 이는 RVA 43B164 가 된다. 이제 아래와 같이 코드를 패치해보자.

```
00001050  15 68 B1 43 00 3B F4 E8 B4 71 00 00 8B F4 FF 15   Before
00001060  64 B1 43 00 3B F4 E8 A5 71 00 00 85 C0 74 0F 68
00001050  15 64 B1 43 00 3B F4 E8 B4 71 00 00 8B F4 FF 15   After
00001060  64 B1 43 00 3B F4 E8 A5 71 00 00 85 C0 74 0F 68
```

그림 7. 코드 패치

원래 Sleep()을 호출하던 부분에 IsDebuggerPresent()의 IAT 주소로 변경을 해주었다. 그렇다면 이제 위에서 언급한 바와 같이 PE 로더가 0x40104F 에서 호출하는 함수의 주소를 0x43B164 에서 참조하기 때문에 아래의 그림처럼 IsDebuggerPresent()가 위치하게 된다.

```
0040104A   .  68 E8030000     PUSH 3E8                          Timeout = 1000. ms
0040104F   .  FF15 68B14300   CALL DWORD PTR DS:[<&KERNEL32.Sleep>]   Sleep
00401055   .  3BF4            CMP ESI,ESP                       Before
00401057   .  E8 B4710000     CALL 04.00408210
0040105C   .  8BF4            MOV ESI,ESP
0040105E   .  FF15 64B14300   CALL DWORD PTR DS:[<&KERNEL32.IsDebugge   IsDebuggerPresent
0040104A   .  68 E8030000     PUSH 3E8
0040104F   .  FF15 64B14300   CALL DWORD PTR DS:[<&KERNEL32.IsDebugge   IsDebuggerPresent
00401055   .  3BF4            CMP ESI,ESP                       After
00401057   .  E8 B4710000     CALL 04.00408210
0040105C   .  8BF4            MOV ESI,ESP
0040105E   .  FF15 64B14300   CALL DWORD PTR DS:[<&KERNEL32.IsDebugge   IsDebuggerPresent
```

그림 8. 호출할 함수 주소 변경

이를 정리하자면, 프로그램이 사용하고자 하는 API 를 호출할 때 해당 명령어에는 "CALL DWORD PTR DS:[*IAT 에 존재하는 해당 함수의 RVA*]"로 나타내며, 프로그램이 실행되면서 실제 주소가 해당 DS:[RVA]에 올라오게 된다. 따라서 위와 같이 (A)를 후킹하기 위해선 DS:[HookFunc]로 바꿔주어야 한다..

이번에는 (B)의 경우를 생각해보자. (A)는 프로그램이 실행되기 이전 파일의 형태에서 조작이 가능하였지만 (B)의 경우 파일이 실행되면서 43B168 에 실제 Sleep API 의 주소를 가지고 온다. 그러므로 (B) 부분은 파일이 아닌 프로세스일 경우에만 조작이 가능함을 알 수가 있다. 따라서 쉽게 DLL Injection 과 같은 기법을 통해 프로세스의 메모리를 조작할 수 있지만, 이번 문서는 코드와 관련된 부분은 최대한 제외하고 원리를 이해함을 목적으로 할 것이기 때문에 필자는 디버거를 통해서 접근할 것이다.

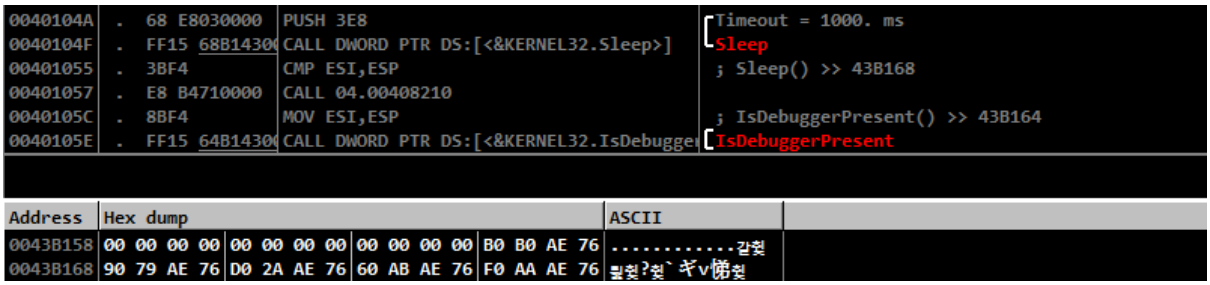

그림 9. 함수 호출 - Debugger

위 그림을 보면 Sleep 함수를 호출할 때 43B168 을 참조하며, IsDebuggerPresent 함수를 호출할 때는 43B164 를 참고하는 것을 확인할 수 있다. 아래 덤프 영역을 보면 각각 해당하는 함수의 주소가 PE 로더에 의해 지정된 것을 확인할 수 있다. Sleep 함수가 가리키는 곳에는 76AE7990 이 위치하며 다른 곳에는 76AEB0B0 가 위치하고 있다. 해당 지점에는 각 함수의 실제 실행과 관련된 부분이 존재하고 있다. 따라서 이 위치를 변경하면 후킹을 진행할 수 있다. 만약 43B168(Sleep)에 76AEB0B0 를 넣어주면 어떻게 될까?

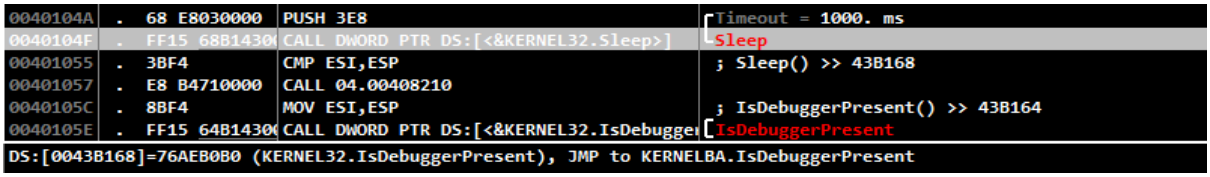

그림 10. Sleep 이 호출하는 주소 변경

위 그림처럼 분명 디버거는 Sleep()이라고 나타내지만 하단의 DS 에서 가리키는 곳은 KERNEL32.IsDebuggerPresent 이다. 이를 통해 디버거 또한 IAT 를 기준으로 주석에 나타내주는 것임을 알 수가 있다. 그렇다면 좀 더 심화 학습을 진행해보자. 흔히 코드 케이브[5](Code Cave)라 할 수 있는 방법을 여기에 적용해볼 수가 있다.

코드 케이브에 대해 간단히 설명하자면 해당 프로세스의 빈 공간에 사용자가 정의한 코드를 넣어준 뒤, 이 부분으로 프로그램이 전개되도록 하는 방식이다. 예제 프로그램에서 빈 공간은 Sleep 이 호출되는 윗부분에 존재하고 있다. 따라서 Call 명령어를 통해 DS:[43B168]을 참조하게 되는데, 이때 43B168 은 코드 케이브의 위치인 401023 이 존재하고 있다. 401023 에서 Sleep API 의 인자로 사용되기 위해 스택에 저장되어 있는 값을 증가시키므로 흐름을 방해한다. ADD 명령 다음에는 원래의 기능을 수행하기 위해 조작되기 이전 값인 76AE7990 로 점프를 해주면 된다.

[5] Inline Patch 기법과 유사한 개념으로 이름만 다르게 불리는 것 같다.

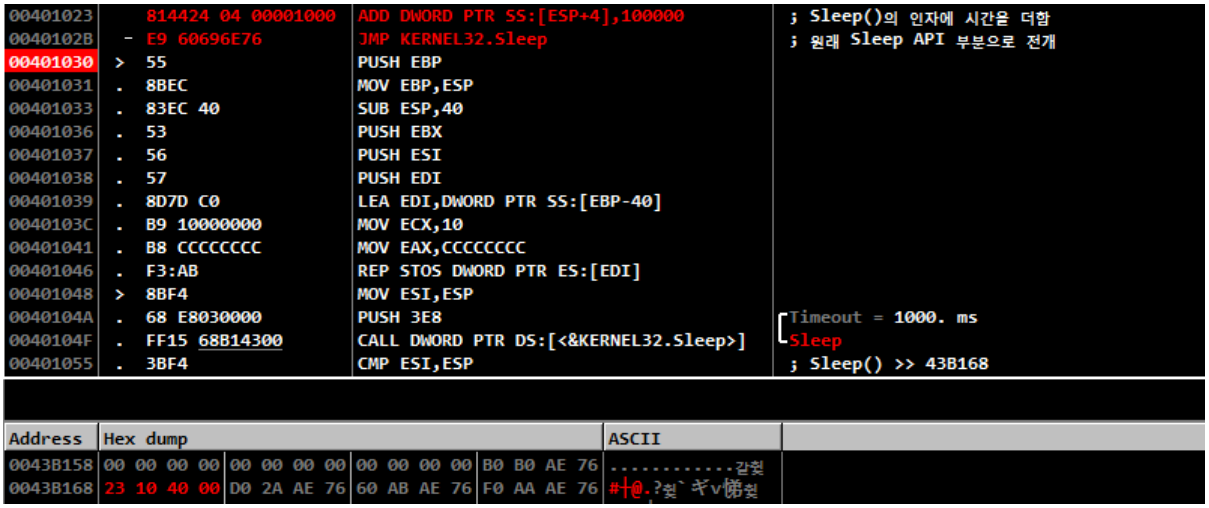

그림 11. Code Cave 를 사용한 후킹

이렇게 조작을 했는데 과연 제대로 프로그램이 동작할까? 당연히 제대로 동작한다. 이전 (A)에서는 IAT 에 존재하는 다른 API 로 변경하는 것만 진행했지만, 이번 과정에서는 인자로 전달되는 값을 직접 수정할 수가 있다. 이는 사용자로부터 입력 받은 값을 조작하여 특정한 함수의 흐름을 조작할 수 있는 것과 같이 다른 면에도 유용하게 활용할 수 있다. (B)에 적용한 방식은 (C)의 시작 부분을 JMP Hook_Address 로 이동하여 유사하게 적용할 수 있다.

(C)의 경우 실제 함수의 명령어들이 위치하고 있다. 따라서 이 부분을 후킹 하기 위해서는 명령어를 조작하여야 하는데 어떻게 조작해야 할까? (C)의 내용은 밑에 그림과 같다. 자세히 보면 상단의 세 명령어의 OPCODE 는 총 5 바이트이다. 바로 이 부분을 조작하므로 우리는 원하는 함수를 후킹할 수 있다. 아래의 그림을 보자.

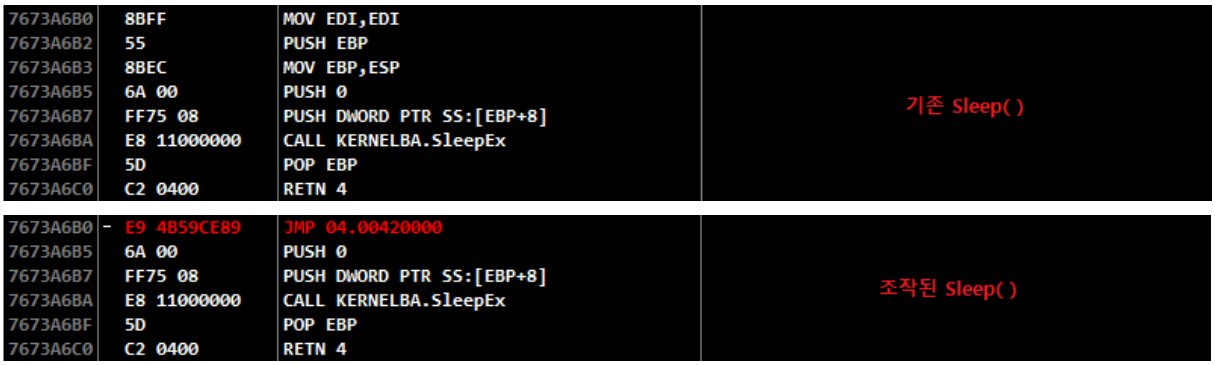

그림 12. (C) 단계 원래 명령어와 조작된 명령어

Sleep() 함수의 원래 주소 7673A6B0 의 명령어를 JMP 명령어로 변경한 것을 확인할 수 있다. 이렇게 변경된 Sleep()함수는 호출될 때마다 해당 부분으로 이동하게 되며 공격자가 의도한 대로 흐름이 변경될 것이다. 필자가 의도한 조작은 아래 그림과 같이

Sleep() 함수의 시간 값을 증가시킨 다음, 원래대로 코드를 복원하고 복원된 지점으로 이동하는 것이다.

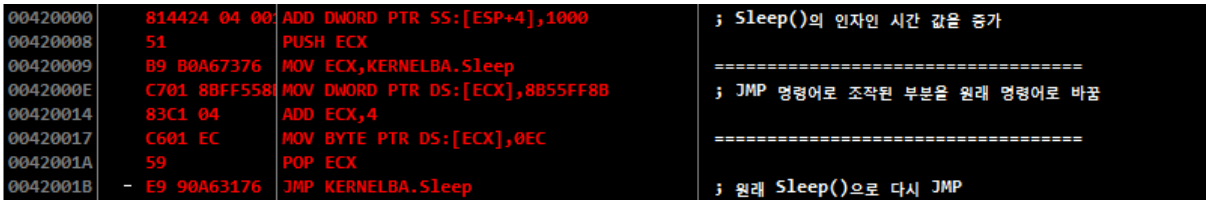

그림 13. 조작 코드

이렇게 코드가 아닌 어셈블리로 조작한 것은 글을 읽는 사람들의 이해를 돕고자 진행한 것이다. 실제로 이렇게 하는 사람은 거의 없으며, 특히 42000E 에서 원래대로 코드를 복원하는 부분은 VirtualProtect 로 권한을 변경하여 진행하는 등 어셈블리로 진행할 경우 복잡하게 이루어지기 때문에 넣지 않았다. 하지만 실제 C/C++와 같은 언어로 후킹 함수를 제작할 때는 더욱 편리할 것이다.

마지막으로 예제를 통해 진행한 각 방식의 차이에 대하여 알아보자. (A)를 후킹 할 때 해당 주소의 코드를 직접 변경하였는데, 이로 인해 해당 주소가 아닌 지점에서 Sleep()을 호출할 경우 정상적인 Sleep()이 호출된다. 반면에 (B)의 경우 IAT 가 가리키는 DS 에 존재하는 값을 변경하였고, 이로 인해 Sleep() 함수를 호출하는 모든 곳이 조작된 DS 를 가리킨다. 따라서 (B)의 방식의 경우 해당 프로세스에서 Sleep() 함수는 후킹 된 코드 케이브를 지나가게 된다. 마찬가지로 (C) 또한 실제 Sleep() 함수의 부분을 JMP 명령어로 조작하였기 때문에 해당 프로세스의 모든 Sleep() 함수가 후킹된 것이다.

특히 (C)의 방식은 Ntdll.dll 의 함수도 조작이 가능하다. 아래의 그림을 보면 실제 Ntdll.ZwDelayExecution 을 호출하는 것을 확인할 수 있으며, 해당 부분의 실제 코드의 5 바이트를 JMP 명령어로 수정하여 원하는 후킹을 진행할 수가 있다.

그림 14. Ntdll.dll 의 API

단, 이러한 IAT 후킹의 경우, 후킹 하고자 하는 함수가 IAT 에 존재해야만[6] 후킹할 수 있기 때문에 동적으로 API 를 로드(예로, GetProcAddress API 등을 통해)하는 경우 IAT 를 참조하지 않아 IAT 후킹을 사용할 없다. 또한 IAT 는 프로세스마다 각각 존재하기 때문에 a.exe 의 IAT 를 후킹하였다고 b.exe 까지 후킹 된 것이 아님을 유의해야 한다.

[6] (C)방식의 Ntdll 후킹은 IAT 에 나타나지 않아 Detours 같은 방법을 사용해야 한다.

3.2 Message Hooking

윈도우 운영체제는 사용자에게 GUI 를 제공해주고, 사용자는 제공받은 GUI 를 이용하여 원하는 동작을 할 수 있다. 동작을 수행하는데 있어 마우스를 움직이거나 클릭, 또는 키보드 버튼을 누르게 되는데 이러한 동작은 윈도우 운영체제가 Event Driven 방식으로 처리한다. 다시 말해 이러한 동작을 이벤트로 발생시켜 운영체제가 그 이벤트에 맞는 메시지를 해당 응용프로그램에게 전달하여 처리하는 방식이다.

아래 그림을 보면 메시지 후킹이 어떤 지점에서 이루어지는지 나타낸 것이다. 사용자가 어떠한 행위를 했을 때 이벤트가 발생되고, 이벤트 발생으로 인해 OS 에서 응용프로그램으로 보낼 메시지들이 OS Message Queue 에 존재하고 있다. 예를 들어 키보드 입력 이벤트가 발생하면 WM_KEYDOWN 메시지가 OS Message Queue 에 추가된다. 운영체제는 해당 이벤트가 어느 응용프로그램에서 발생했는지 파악한 다음, 큐에서 메시지를 꺼내어 해당 응용프로그램의 메시지 큐에 전달한다. 해당 응용프로그램은 자신의 Application Message Queue 에 WM_KEYDOWN 메시지가 추가된 것을 확인하고 해당 이벤트 핸들러를 호출한다. 이러한 방식으로 윈도우는 메시지를 전달한다.

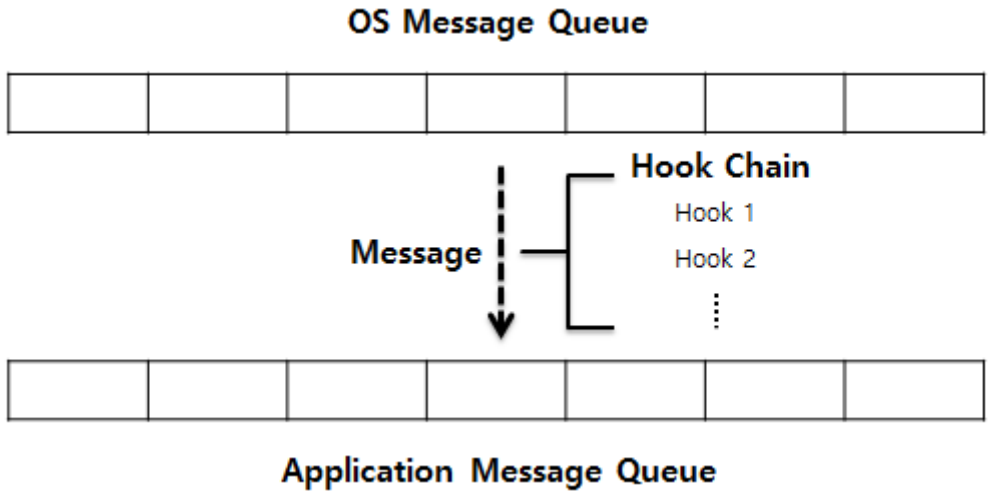

그림 15. 메시지 전달 방식

재미있는 사실은 윈도우 운영체제에서 이러한 메시지 훅기능을 기본적으로 제공한다는 것이다. 바로 SetWindowsHookEx API 가 그 주인공으로 훅 체인에 응용프로그램이 정의한 후크 프로시저를 설치하며 이를 통해 사용자는 특정 유형의 이벤트를 모니터링 할 수 있다.

```
HHOOK WINAPI SetWindowsHookEx(
    _In_ int          idHook      // 훅 종류
    _In_ HOOKPROC     lpfn,       // 지정한 이벤트 발생시 처리하는 프로시저 주소
    _In_ HINSTANCE    hMod,       // lpfn 이 있는 DLL 의 핸들
    _In_ DWORD        dwThreadId
);
```

그림 16. SetWindowsHookEx API

만약 해당 API 를 구현하는 HookKey.dll 이 존재하며 이를 실행하기 위한 HookMain.exe 를 제작하였다고 가정하자. HookMain.exe 를 실행하면 HookKey.dll 이 해당 프로세스에 로드되며 SetWindowsHookEx()가 호출된다. 이렇게 메시지 후킹이 걸린 상태에서, 다른 프로세스가 해당 이벤트를 발생시킨다면 HookKey.dll 은 그 프로세스에서도 로딩이 된다.

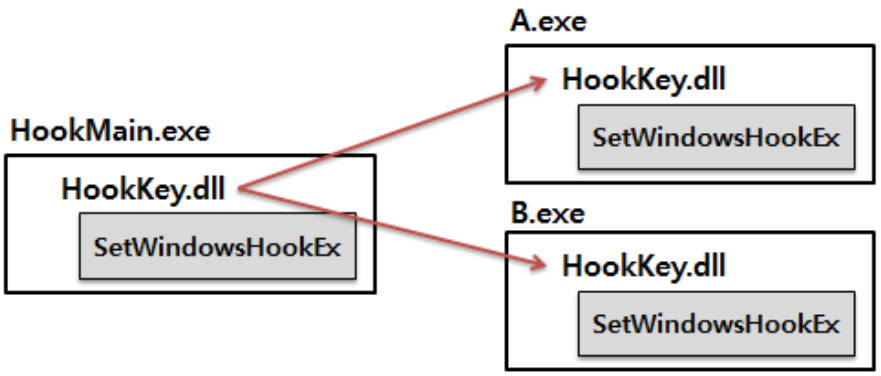

그림 17. DLL Injection

위 그림과 같이 나타낼 수 있으며, 다시 말해 후킹 된 이벤트가 발생하는 모든 프로세스에 DLL 인젝션이 일어나는 것이다. 이러한 방식을 통해 메시지를 후킹할 수 있으며, 이는 DLL 인젝션의 한 방법으로 사용할 수 있다.

4 Kernel Mode Hooking

3 장에서는 사용자 모드 후킹에 대하여 알아보았다면 이번 장에서는 커널 모드 후킹에 대하여 알아볼 것이다. 커널 모드에서 이루어지는 후킹의 경우 단순히 JMP 명령어를 설치하는 것이 아니라, 특정한 구조체에 포함된 값을 수정하는 등 작업을 수행해야 하기 때문에 아무래도 사용자 모드의 후킹보다 복잡하다. 이제 이러한 커널 모드 후킹에 대하여 알아보자.

4.1 System Call

운영체제는 사용자 모드(Ring 3)와 커널 모드(Ring 0)라는 두 가지 형태의 권한이 존재하고 있다. 이렇게 분리되는 이유는 다양하지만, 아무래도 보안과 관련된 점 또한 매우 중요하다. 만약 분리되어 있지 않을 경우 어떠한 프로세스든지 운영체제의 핵심 기능을 조작할 수 있게 되므로 이를 방지하기 위해선 분리되는 것이 좋다. 그렇기에 커널 모드에서는 사용자 모드를 조작할 수 있지만, 반대로 사용자 모드에서 커널 모드는 조작할 수가 없다.

하지만 커널 영역에 접근할 수 없다는 것은 해당 프로세스가 디스크의 내용을 읽을 수가 없게 되고 그 외에도 하드웨어나 프로세스에 직접 접근할 수 없게 된다. 그렇다면 어떻게 우리가 제작한 사용자 모드의 프로그램이 프로세스나 디스크와 관련된 작업을 수행할 수 있을까? 이는 바로 시스템 호출(System Call)을 사용하여 사용자 모드의 프로세스가 커널 영역에 접근을 요청할 수 있기 때문이다.

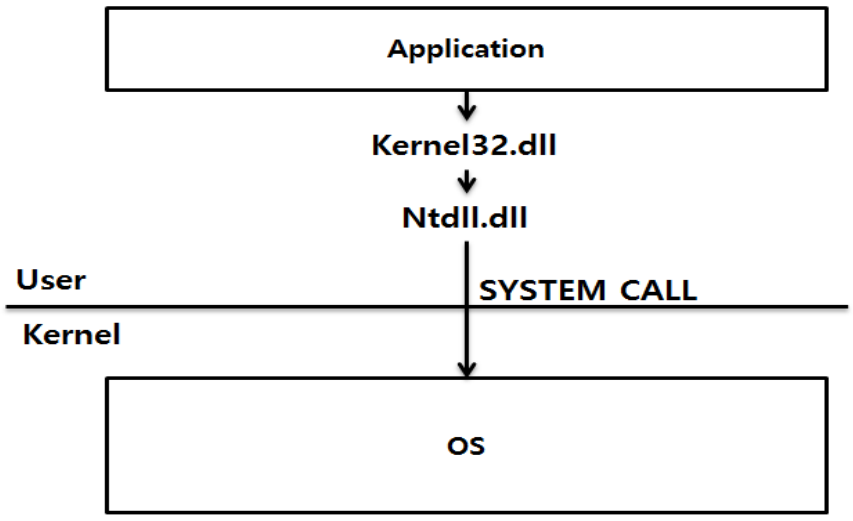

그림 18. System Call 과정

위 그림과 같이 응용프로그램이 Kernel32.dll 의 API 를 호출하면 해당 API 는 Ntdll.dll 의 함수를 호출한다. 그리고 호출된 Ntdll.dll 은 자신이 커널에 요청해야 할 서비스 번호를 가지고 시스템 호출을 진행하며 이 과정이 바로 응용프로그램이 커널에게 시스템 자원 접근을 요청하는 과정이다. 이때 지정한 서비스를 요청하기 위해 EAX 에 원하는 서비스 번호를 저장하고 EDX 에는 이 서비스에 사용될 인자를 가리키는 포인터를 넘겨준다. 이러한 과정을 통해 커널에서는 어떠한 서비스가 필요한지, 어떠한 인자를 넘겨주었는지 알 수가 있다.

System Call 은 "INT 0x2E"와 "SYSENTER" 두 가지 명령어로 나누어진다. 이렇게 나누어지는 기준은 바로 Windows XP 이전과 이후로, 이전에는 INT 0x2E 를 사용하였으며, XP 부터는 SYSENTER 를 사용한다. INT 0x2E 의 경우 상대적으로 무거운 인터럽트를 진행하므로 클럭 수를 많이 소모하였기 때문에, 이를 보완하기 위해 SYSENTER 가 나온 것이다. 이러한 차이 외에 앞으로 진행할 후킹 과정에서도 차이점을 가지므로 이에 대하여 알아보자.

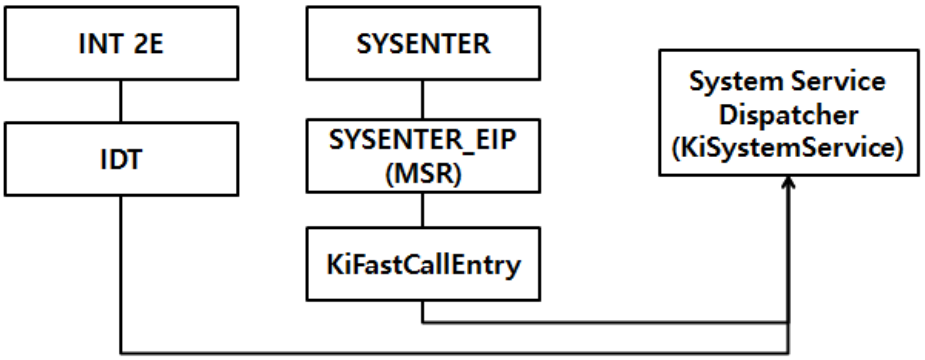

그림 19. INT 0x2E 와 SYSENTER

우선 INT 0x2E 의 경우 IDT(Interrupt Descriptor Table)을 참조하여 바로 System Service Dispatch(KiSystemService)로 간다. 하지만 SYSENTER 는 SYSENTER_EIP(MSR)를 참조하여 KiFastCallEntry 로 진행한 다음 KiSystemService 로 간다. 마지막으로 INT 0x2E 의 경우 IRET 라는 명령어로 커널 모드에서 다시 사용자 모드로 복귀하고, SYSENTER 의 경우 SYSEXIT 라는 명령어를 통해 사용자 모드로 복귀한다. 이것이 별로 중요하게 느껴지지 않을 수 있지만, 후킹을 진행할 때 두 명령어에 따라 후킹 지점이 달라진다. 이러한 각 후킹 방법에 대해서는 바로 뒤에서 알아보자.

4.2 INT 0x2E Hooking

INT 0x2E 는 인터럽트 0x2E 로 IDT 에 정의된 인터럽트 서비스 루틴(ISR)을 수행한다. 여기서 IDT 는 256 개의 Entry 로 이루어진 배열로 엔트리 하나당 하나의 인터럽트에

대응하며 각 인터럽트는 IDT 로부터 처리할 함수의 주소(ISR)을 전달받는다. 각 엔트리에는 지정된 값이 담겨 있으며 WinDBG[7]로 확인했을 경우 아래와 같은 모습을 볼 수가 있으며 INT 0x2E 의 경우 IDT 에서 바로 KiSystemService 를 가리키고 있는 것을 확인할 수 있다.

```
kd> !idt
Dumping IDT: 8003f400
...(skip)
2e:     8053f481 nt!KiSystemService
37:     806d3728 hal!PicSpuriousService37
3d:     806d4b70 hal!HalpApcInterrupt
41:     806d49cc hal!HalpDispatchInterrupt
50:     806d3800 hal!HalpApicRebootService
...(skip)
```

그림 20. IDT 구조

이 과정을 요약하면, XP 이전 버전에는 응용프로그램이 API 를 호출하면 Ntdll.dll 의 Zw*, Nt* 함수를 호출하게 된다. 이러한 함수는 결국 INT 0x2E 를 통해 운영체제에게 커널 모드 작업을 요청한다. 이때 INT 0x2E 가 IDT 에서 KiSystemService 의 주소를 참조하여 진행하는 것이다. 따라서 우리가 후킹 해야 할 부분은 바로 IDT[8]이다. IDT 가 가리키는 2E 의 주소로 가보면 실제로 KiSystemSerive 가 존재하고 있는 것을 아래와 같이 확인할 수 있다.

```
nt!KiSystemService:
8053f481 6a00            push    0
8053f483 55              push    ebp
8053f484 53              push    ebx
8053f485 56              push    esi
8053f486 57              push    edi
8053f487 0fa0            push    fs
8053f489 bb30000000      mov     ebx,30h
```

그림 21. INT 0x2E 의 ISR(KiSystemService)

[7] 커널 디버깅이므로 유저 모드 디버깅으로는 접근할 수 없어 WinDBG 를 사용한다.

[8] INT 0x2E Hooking 은 결국 IDT 를 후킹 하는 것으로 IDT Hooking 이라고도 한다.

IDT 를 후킹 할 것이므로 우선 IDT 의 주소를 알아야 하는데, IDTR 레지스터에 IDT 의 Base Address 와 IDT 의 크기가 저장되어 있다. WinDBG 를 통해 알 수 있는 방법은 IDTR 레지스터의 값(주소)를 출력하거나 "!idt"를 통해 해당 주소[9]를 알아낼 수가 있다. IDT 의 Entry 구조는 아래 그림과 같이 8 바이트씩으로 이루어져 있으며 Entry 가 가리키는 ISR 의 주소가 하위 2 바이트, 상위 주소 2 바이트로 나누어져 있다.

```
kd> r idtr
idtr=8003f400
kd> !idt
Dumping IDT: 8003f400

kd> dt _KIDTENTRY
ntdll!_KIDTENTRY
    +0x000 Offset           : Uint2B      // 하위 오프셋
    +0x002 Selector         : Uint2B
    +0x004 Access           : Uint2B
    +0x006 ExtendedOffset   : Uint2B      // 상위 오프셋
```

그림 22. IDT 주소와 각 엔트리 구조

하나의 엔트리가 8 바이트로 이루어져 있다는 것을 확인했다. 그렇다면 우리가 찾고자 하는 엔트리는 0x2E 번째 엔트리이므로 IDT Base Address 에 0x170 을 더한 위치에 존재하고 있다. 아래 결과와 같이 8003f570 부터 해당 엔트리가 존재하고 있다. 하위 2 바이트와 상위 2 바이트를 조합하여 INT 0x2E 의 ISR 은 8053f481 이라는 것을 알 수 있다.

```
kd> db 8003f400 8003fC00
8003f400  9c 01 08 00 00 8e 54 80-14 03 08 00 00 8e 54 80  ......T.......T.
8003f410  3e 11 58 00 00 85 00 00-e4 06 08 00 00 ee 54 80  >.X...........T.
...(skip)
8003f560  80 fc 08 00 00 ee 53 80 - c0 05 08 00 00 ee 54 80  ......S.......T.
8003f570  81 f4 08 00 00 ee 53 80-80 27 08 00 00 8e 54 80  ......S..'....T.
...(skip)
```

그림 23. IDT 0x2E 번째 Entry

[9] Assembly 의 경우 "sidt" 명령어를 사용하여 IDT 의 주소를 알 수 있다.

이제 우리가 어떤 주소(8003f570)를 후킹 해야 하는지 알았으니, 본격적인 후킹을 진행해보자. 이번 후킹 역시 프로그래밍을 통해 진행하는 것이 아니라 원리를 이해하기 위해 커널 디버거 WinDBG 를 통해 진행할 것이다. 해당 0x2E 의 ISR 을 FFFFFFFF 로 조작하는 과정으로 "0000"으로 채운 곳은 오프셋이 아닌 부분으로 구분을 위해 "0"으로 채운 것이다.

```
kd> !idt 2e                                    // 기존 2E 의 ISR 확인
2e:      8053f481 nt!KiSystemService
kd> ed 8003f570                                // Windbg 를 통해 직접 수정
8003f570 0008f481 0000ffff
8003f574 0000ffff ffff0000
kd> !idt 2e                                    // 조작된 2E 의 ISR 확인
2e:      ffffffff
kd> db 8003f570 8003f580                       // 조작된 IDT 2E Entry 확인
8003f570   ff ff 00 00 00 00 ff ff-80 27 08 00 00 8e 54 80   ........'....T.
```

그림 24. IDT Entry 0x2E 후킹

이처럼 IDT 가 후킹 된 상황에서 INT 0x2E 를 통한 시스템 호출(System Call)이 발생하면 사용자 모드에서 커널 모드로 넘어갈 때 후킹된 주소로 가게 된다. 실제 후킹도 이와 같은 과정으로 진행되며, 대신 IDT 의 주소를 얻을 때 "sidt" 명령어를 사용하여 주소를 얻는다. Sidt 명령어는 IDT 의 주소를 저장하고 있는 IDTR 레지스터의 값을 참조하여 값을 얻어 온다. 또한 디버거를 통해 수정하는 것이 아니라 프로그래밍을 통해 수정하고자 할 때, 인터럽트를 비활성화("CLI" 명령어)해야 한다. 그리고 후킹이 완료되면 인터럽트를 다시 활성화("STI" 명령어) 시켜 정상적으로 구동되게끔 해야 한다.

4.3 SYSENTER Hooking

윈도우 XP 이상의 버전에서는 INT 0x2E 가 아닌 SYSENTER 를 사용한다. 시스템 호출이 요청되면 NTDLL 은 EAX 레지스터에 해당 시스템 호출의 번호를 저장하고 EDX 레지스터에는 인자로 사용될 주소를 넣어준다. 그리고 SYSENTER 명령을 실행하여 커널 영역으로 들어오게 되는데, 이때 바로 커널로 들어가는 것이 아니라 실행될 커널의 주소(KiFastCallEntry)를 SYSENTER_EIP(MSR 0x176 레지스터)에서 참조하여 KiFastCallEntry 로 넘어가게 되는 것이다.

MSR 은 Model-Specific Register 로 디버깅이나 프로그램 실행 추적, 컴퓨터 성능 모니터링, 특정 CPU 기능 전환에 사용되는 각종 제어 레지스터 x86 명령어의 집합이다.

이러한 집합 중 MSR 0x176 에는 IA_SYSENTER_EIP(KiFastCallEntry)가 존재하고 있으며, MSR 에 접근하고자 할 때는 "rdmsr", "wrmsr" 명령어를 통해 접근할 수가 있다. 우리가 찾아야 할 것은 MSR 0x176 이며 다음과 같은 결과를 얻을 수가 있다.

```
kd> rdmsr 176
msr[176] = 00000000`8053f540

kd> u 8053f540
nt!KiFastCallEntry:
8053f540 b923000000          mov       ecx,23h
8053f545 6a30                push      30h
...(skip)
```

그림 25. Read MSR 0x176

MSR 0x176 이 제대로 KiFastCallEntry 를 가리키고 있는 것을 확인할 수 있다. 바로 이 부분의 값을 바꾸어 SYSENTER 를 후킹[10] 할 수 있다. MSR 레지스터를 읽을 때는 rdmsr 명령어로 읽었다면, MSR 레지스터의 값을 변경할 때는 "wrmsr" 명령어를 사용하면 된다. 아래의 그림을 보자.

```
kd> wrmsr 176 11111111                    // MSR 0x176 의 값을 변경
kd> rdmsr 176                             // MSR 0x176 값 변경 확인
msr[176] = 00000000`11111111
```

그림 26. Write MSR 0x176

실제로 위와 같이 옳지 않은 주소로 변경하면 당연히 블루 스크린을 맞이할 수 있을 것이다. 그렇다면 abex'sCrackMe01.exe 를 가지고 직접 코드의 흐름을 조작하여 보자. 우선 아래의 코드는 정상적인 흐름을 나타낸다. Ntdll.dll 의 함수를 추적하여 들어가보면 KiFastSystemCall 이라는 부분을 볼 수 있는데 이는 SYSENTER 명령어를 통해 KiFastCallEntry 로 가기 위한 부분이다. KiFastSystemCall 에는 SYSENTER 명령어가 위치하고 있는 것을 확인할 수 있다.

```
ntdll!KiFastSystemCall:
001b:7c93e4f0 8bd4            mov       edx,esp
001b:7c93e4f2 0f34            sysenter
```

[10] SYSENTER Hooking 은 결국 MSR 을 후킹 하므로 MSR Hooking 이라고도 한다.

```
kd> rdmsr 176                          // MSR 0x176 의 주소를 확인
msr[176] = 00000000`8053f540

kd> bp 8053f540                        // MSR 0x176 의 주소에 BP 설정

kd> p                                  // Kernel 의 KiFastCallEntry 에 올바르게 진입
Breakpoint 3 hit
nt!KiFastCallEntry:
8053f540 b923000000        mov        ecx,23h
```

그림 27. 정상적인 SYSENTER 진입

정상적인 SYSENTER 의 흐름을 wrmsr 을 통해 조작해보자. MSR 0x176 를 11111111 로 수정하므로 비정상적인 흐름으로 동작하도록 수정하였다. 그리고 이전과 같이 SYSENTER 명령어를 실행하기 전에 원래 MSR 0x176(KiFastCallEntry)의 주소에 BP 를 설정한 다음 진행을 해보자. 정상적인 흐름이라면 KiFastCallEntry 에서 멈추어야 하지만, 조작된 MSR 0x176 이 가리키는 주소 11111111 로 흐름이 바뀌었다.

```
ntdll!KiFastSystemCall:
001b:7c93e4f0 8bd4              mov        edx,esp
001b:7c93e4f2 0f34              sysenter

kd> bp 8053f540                        // 기존 KiFastCallEntry 에 BP 설정
kd> wrmsr 176 11111111                 // MSR 0x176 주소 조작

kd> p                                  // 조작한 주소로 이동
Access violation - code c0000005 (!!! second chance !!!)
11111111 ??                    ???
```

그림 28. 후킹 된 SYSENTER 진입

이러한 방법을 통해 SYSENTER Hooking(또는 MSR Hooking)을 진행할 수 있으며, 실제 11111111 에 후킹 함수가 존재할 경우 시스템 호출이 발생할 때마다 후킹 함수를 지나가게 된다. 공격자의 입장에서 이러한 System Call Hooking 을 진행하며 후킹 함수에 과도한 조건을 걸어놓는다면 시스템 성능이 크게 하락하여 사용자가 쉽게 알아차릴 수 있을 것이다.

4.4 SSDT Hooking

SSDT(System Service Dispatch Table)는 시스템 호출을 요청한 뒤, 전달되는 서비스 번호에 맞는 함수를 찾을 때 참조한다. 위 과정에서 시스템 호출을 요청하는 두 가지 명령어(INT 0x2E 와 SYSENTER)에 대하여 알아보았는데, 결국 두 명령어 모두 KiSystemService(System Service Dispatcher)를 호출한다고 언급하였다.

KiSystemService 가 호출될 때 EAX 에는 사용자 영역에서 요청한 서비스 번호가 저장되어 있으며, EDX 에는 이러한 서비스에 사용될 인자가 저장되어있다. 이러한 시스템 호출 번호(EAX)에 맞게 KeServiceDescriptorTalbe 에서 Native API 의 주소를 가지고 온다. 그 후 시스템 호출을 종료하고 다시 사용자 모드로 복귀하게 되는데, 이러한 과정은 아래의 그림과 같다.

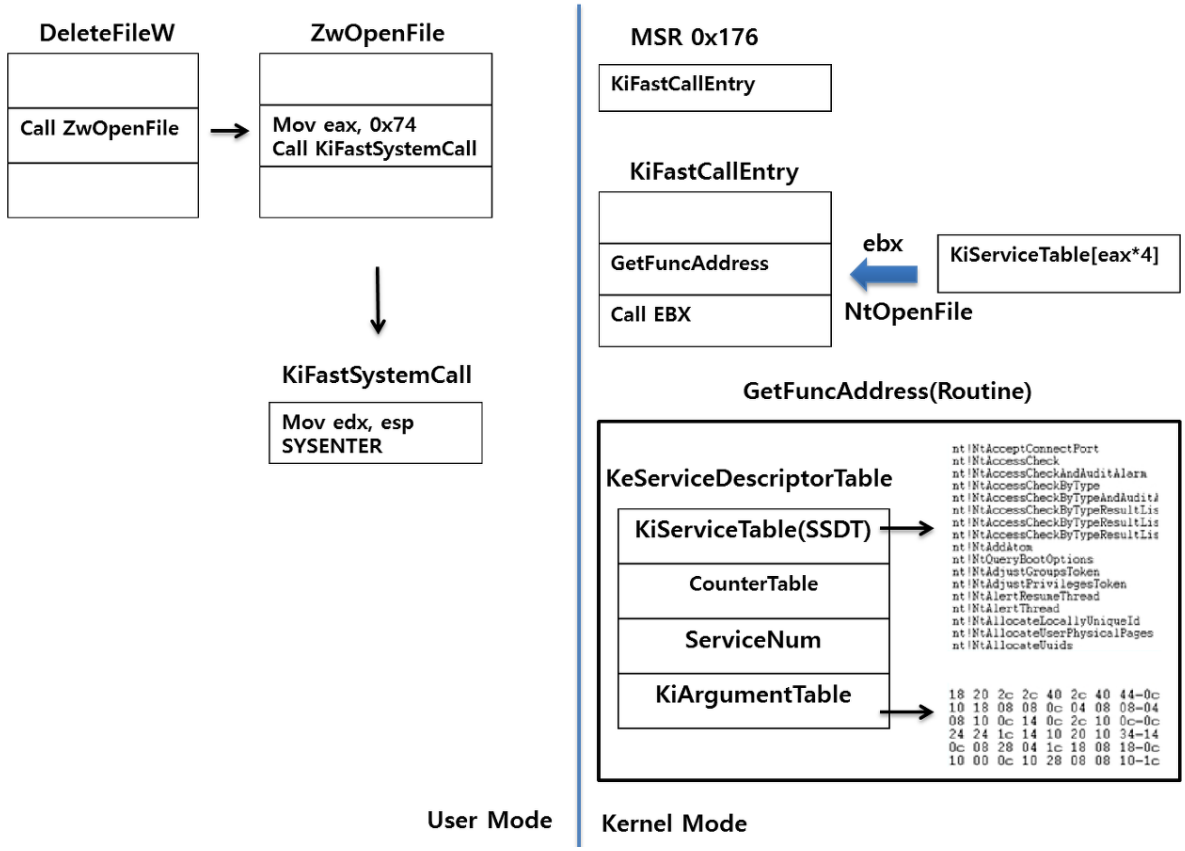

그림 29. 전체적인 시스템 호출 과정

결국 SSDT(KiServiceTable)에서 서비스 호출 번호에 맞는 주소를 얻은 다음 이를 호출하는 형태로 진행되는 것이다. 그렇다면 SSDT Hooking 은 어느 부분을 후킹해야 하는 것일까? 위 그림의 과정에서 설명하자면 바로 GetFuncAddress 과정에서 후킹한다고 할 수 있다. SYSENTER 를 통해 KiFastCallEntry 로 진입한 후 서비스 번호에

맞는 서비스 루틴을 SSDT 에서 얻어온다. 따라서 SSDT 에 존재하고 있는 각 서비스 루틴의 주소를 조작하므로 후킹을 진행할 수 있다. 이를 표현한 그림은 아래와 같다.

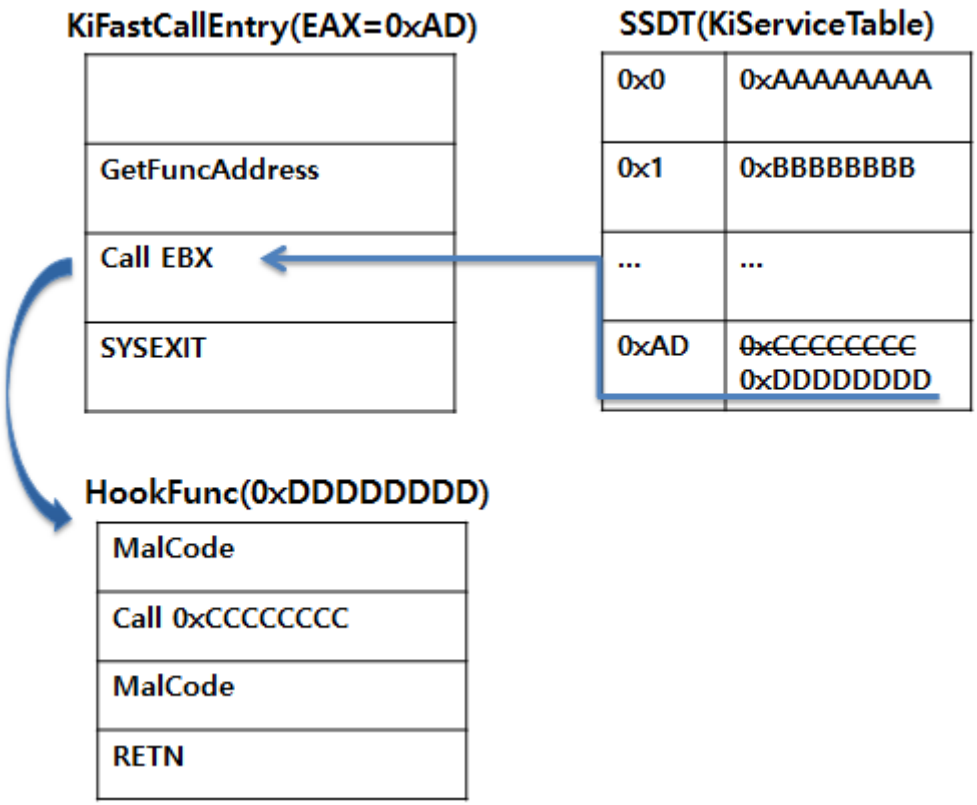

그림 30. SSDT Hooking 과정

해당 시스템 호출의 서비스 루틴을 가지고 오는 과정에서 SSDT 를 참조하는데, SSDT 의 해당 번호가 나타내는 주소를 후킹하므로 우리가 원하는 흐름으로 조작할 수 있다. 위 그림을 예로 시스템 호출이 요청되었을 때 서비스 번호가 저장되어 있는 EAX 의 값이 0xAD 라면 SSDT 에서 0xAD 가 가리키는 서비스 루틴의 주소 0xCCCCCCCC 가 반환되어 이를 호출한다. 하지만 만약 공격자가 SSDT 를 후킹하여 0xDDDDDDDD 로 서비스 루틴의 주소를 변경하였다면, 시스템 호출 0xAD 가 발생할 때마다 0xDDDDDDDD 를 지나가게 된다.

KeServiceDescriptorTable 은 네 가지 항목을 가지고 있는 구조체로 아래 그림과 같이 나타나는 것을 확인할 수 있으며 중요한 첫 번째 항목과 네 번째 항목에 대하여 알아보자. 첫 번째 항목은 KiServiceTable(SSDT)의 주소를 담고 있는 항목으로 이 값을 통해 SSDT 에 접근하여 Native API 의 주소를 얻을 수가 있으며. 네 번째 항목은 ParamTableBase 는 KiArgumentTable 의 주소 값을 담고 있는데, 이들 각각은 SSDT 의 Native API 와 일 대 일로 대응한다.

```
kd> dd KeServiceDescriptorTable
80554fa0   80503b8c 00000000 0000011c 80504000 //ServiceDescriptor[0]
80554fb0   00000000 00000000 00000000 00000000   //ServiceDescriptor[1]
80554fc0   00000000 00000000 00000000 00000000   //ServiceDescriptor[2]
80554fd0   00000000 00000000 00000000 00000000   //ServiceDescriptor[...]
...(skip)
```

그림 31. KeServiceDescriptorTable 구조[11]

SSDT 의 주소는 첫 번째 항목의 값인 0x80503b8c 로 해당 주소를 확인해보면 여러 주소들이 존재하고 있는 것을 아래와 같이 확인할 수 있다. 각 값들은 Native API 의 실제 주소이며 해당 주소를 확인해보면 Native API 의 이름을 같이 볼 수 있다.

```
kd> d 80503b8c       // SSDT Base
80503b8c   8059b948 805e8db6 805ec5fc 805e8de8
80503b9c   805ec636 805e8e1e 805ec67a 805ec6be
80503bac   8060ddfe 8060eb50 805e41b4 805e3e0c
80503bbc   805ccde6 805ccd96 8060e424 805ad5ae
80503bcc   8060da3c 8059fdbe 805a7a00 805ce8c4
...(skip)

kd> u 8059b948
nt!NtAcceptConnectPort:
8059b948 689c000000        push      9Ch
8059b94d 6838b14d80        push      offset nt!_real+0x128 (804db138)
8059b952 e8b9e5f9ff        call      nt!_SEH_prolog (80539f10)
```

그림 32. SSDT 를 통한 Native API 접근

시스템 호출을 통해 커널 모드로 진입할 때 EAX 에는 요청한 서비스 번호를 저장하고 있고 EDX 에는 인자로 사용될 포인터를 포함하고 있다고 하였다. 그러므로 어떠한 Native API 를 요청하는지 알기 위해선 SSDT 의 주소에 [EAX*4]를 더해주면 그 주소를 알 수 있다. 실제 SSDT Hooking 도 이와 같은 방식으로 진행한다. 그렇다면 이제 실제 SSDT 의 주소를 변경해보자.

---

[11] Windows 는 기본적으로 네 개의 SSDT 를 갖지만, Native 함수를 위한 것과 win32k.sys 의 GUI 함수를 위한 SSDT 만을 사용한다.

SSDT 를 통해 접근할 수 있는 Native API 함수 5 개의 주소를 변경해보자. WinDBG 를 사용하기 때문에 특정 주소 값을 수정하기 위한 "ed" 명령어를 사용하였으며, 기존의 주소를 아무 의미 없는 값들로 변경하였다. 그 후 SSDT 를 확인해보면 위 그림 32 에서 확인할 수 있던 주소들이 내가 수정한 값으로 변경되어 있는 것을 확인할 수 있다.

```
kd> ed 80503b8c
80503b8c 8059b948 ffffffff
80503b90 805e8db6 00000000
80503b94 805ec5fc ffffffff
80503b98 805e8de8 00000000
80503b9c 805ec636 ffffffff
80503ba0 805e8e1e 11111111

kd> d 80503b8c
80503b8c   ffffffff 00000000 ffffffff 00000000
80503b9c   ffffffff 11111111 805ec67a 805ec6be
80503bac   8060ddfe 8060eb50 805e41b4 805e3e0c
80503bbc   805ccde6 805ccd96 8060e424 805ad5ae
```

그림 33. SSDT Hooking

이렇게 후킹을 하면 해당 Native API 가 요청될 때마다 후킹된 주소로 넘어가게 된다. 위 실습은 아주 극단적인 예를 보여주기 위한 과정으로 바로 블루 스크린이 나타난다. 실제 후킹 공격을 진행하기 위해선 메모리 쓰기 보호(Write Protect)를 해제하는 작업을 추가해야 하며, 매크로와 같은 방식을 통해 공격을 진행한다.

5 Conclusion

사용자 영역 후킹과 커널 영역 후킹에 대해 디버거를 통해 접근해보며 어떻게 후킹이 이루어지는지 알아보았다. 실제 공격을 하는데 있어 디버거를 사용하는 것보다는 프로그래밍을 통해 쉽게 공격이 이루어질 수 있도록 한다. 그렇기에 다른 사람들의 글 대부분이 이러한 프로그래밍에 초점을 맞추고 어떻게 코드를 설계하는지, 코드가 의미하는 것이 어떤 내용인지 잘 설명하고 있으므로 이후에 코드와 관련된 내용을 학습하면 더 좋을 것이다.

필자도 코드를 통해 어떻게 동작하겠구나 생각해볼 수 있었지만 직접 디버거를 통해 접근해볼 때마다 "여기를 수정하면 어떻게 될까?"라는 등 좀 더 깊이 있는 생각을 해볼 수가 있었다. 이후에는 여기서 다루지 못한 후킹들에 대하여 추가로 학습해볼 것이며 윈도우 운영체제와 관련된 내용을 더 공부해보아야겠다는 생각을 할 수가 있었다.

6 Reference

[+] 리버싱 핵심 원리(악성 코드 분석가의 리버싱 이야기) |이승원|인사이트|2012.09.30

[+] http://blog.naver.com/ikariksj/140056467421

[+] http://www.codeproject.com/Articles/2082/API-hooking-revealed

[+] http://www.reversecore.com/23

[+] http://yokang90.tistory.com/58

[+] http://xcoolcat7.tistory.com/542

[+] http://egloos.zum.com/maxtrain/v/2775961

[+] http://kernel32.tistory.com/15

[+] https://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx

[+] https://blogs.msdn.microsoft.com/kocoreinternals/2009/03/16/idt-isr/

[+] https://en.wikipedia.org/wiki/Model-specific_register

[+] http://amur.tistory.com/entry/커널모드에서-유저모드-분석하기