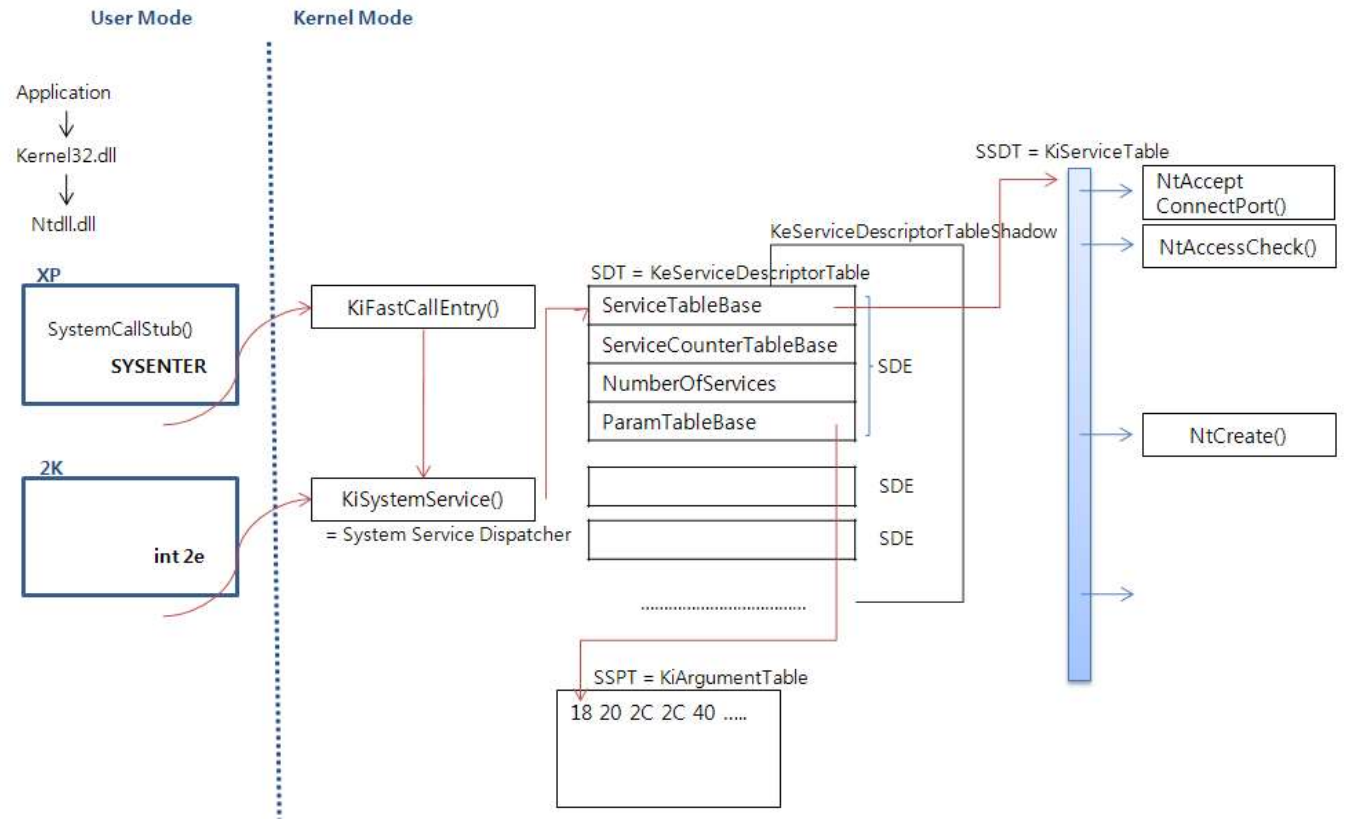
System Call & SSDT Hooking

Reversing/Theory 2016.04.10 19:03

System Call

Windows 운영체제는 사용자 모드와 커널 모드라는 두 가지 형태의 권한이 존재하고 있다. 굳이 하나가 아닌 두 가지로 분류되는 것은 모든 프로세스가 하나의 권한으로만 동작할 경우, 각 프로세스는 하드웨어나 프로세스에 직접 접근할 수 있게 된다. 이는 어떠한 프로세스라도 운영체제의 핵심 기능을 조작할 수 있게 되는 것이므로 보안에 있어 매우 취약하게 된다. 이러한 요소를 방지하기 위해 두 개의 영역으로 분류되었고, 당연히 사용자 모드에 존재하고 있는 프로세스는 커널 영역에 접근할 수가 없다.

하지만 커널 영역에 접근할 수 없다는 것은 해당 프로세스가 디스크의 내용을 읽을 수가 없게 되고, 그 외에도 많은 작업들에 제한이 생긴다. 따라서 이러한 불편함을 보완하기 위해 "사용자 모드의 프로세스가 커널 영역에 접근할 수 있는 방법을 만들자."라는 의도에 부합되는 것이 바로 시스템 호출(System Call)이다. 즉, Windows는 커널에 대한 직접적인 접근을 제한하는 대신 사용자 영역에서 API를 호출하면 NTDLL.DLL을 거쳐 커널 모드로 진입하게 된다.



시스템 콜은 크게 2가지 방법으로 진행될 수 있는데, 바로 "INT 0x2E"와 "SYSENTER"이다. INT 0x2E의 경우 소프트웨어 인터럽트로 Windows XP 이전에는 이를 통해 시스템 콜을 진행하였다면, XP부터는 SYSENTER를 통해 시스템 콜을 진행하였다. 이 둘의 차이점이 몇 가지 존재하고 있지만 가장 큰 차이점은 바로, SYSENTER는 수행 시간에 따른 부하가 적다는 것이다. 또한 각 명령어가 호출된 후 과정에 차이가 있는데 이는 아래의 그림에서와 같이 SYSENTER 명령어가 진행될 경우 KiFastCallEntry()를 호출한 후, 이를 통해 다시 KiSystemService()를 호출한다. 이에 반해 INT 0x2E의 경우 KiFastCallEntry()를 거치지 않고 바로 호출된다는 것이다.

INT 0x2E나 SYSENTER 명령어의 경우 우리가 직접 확인할 수도 있다. 흔히 자주 접할 수 있는 OllyDBG와 같은 유저 모드 디버거를 통해 특정한 함수를 호출하는 부분을 계속 따라가다 보면 ntdll.dll의 함수를 볼 수가 있다. 그리고 이러한 함수를 계속 트레이싱하다 보면 INT 0x2E나 SYSENTER 명령어를 확인할 수 있다. 단, 유저 모드 디버거이기 때문에 해당 명령어가 수행되는 자세한 과정은 확인할 수가 없으므로 이를 확인하기 위해선 WinDBG와 같은 커널 디버거를 이용해야 한다. 진입하는 과정은 아래의 그림과 같다.

```
742199D0   B8 10110500   MOV EAX,51110          UNICODE "ecurity-efs-l1-1-0"
742199D5   BA 30EA2174   MOV EDX,USER32.7421EA30
742199DA   FFD2          CALL EDX
742199DC   C2 0800       RETN 8
```

CALL EDX를 통해 지정된 주소를 호출하게 되는데, 해당 부분은 아래의 그림과 같다. 여기서 FS:[0x30]은 TEB의 0x30번째에 있는 값을 나타낸다. 이는 PEB를 나타내는 것으로 이러한 방법을 통해 PEB에 접근할 수가 있다. 그리고 PEB의 0x464번째 바이트 값을 통해 해당 값이 2인지 비교한 다음 만약 2라면 INT 0x2E를 통해 커널 모드에 진입하게 된다.

```
7421EA30   64:8B15 300000(  MOV EDX,DWORD PTR FS:[30]
7421EA37   8B92 64040000    MOV EDX,DWORD PTR DS:[EDX+464]
7421EA3D   F7C2 02000000    TEST EDX,2
7421EA43   74 03            JE SHORT USER32.7421EA48
7421EA45   CD 2E            INT 2E
7421EA47   C3               RETN
7421EA48   EA 4FEA2174 33(  JMP FAR 0033:7421EA4F          Far jump
```

만약 해당 값이 2가 아니라면 JMP 명령어를 통해서 KiFastSystemCall 부분으로 넘어오게 되는데, 해당 함수의 부분은 SYSENTER 명령어를 통해 KiFastCallEntry로 이동하게 된다.

```
774C8F00   894424 04        MOV DWORD PTR SS:[ESP+4],EAX    KiFastSystemCall
774C8F04   895C24 08        MOV DWORD PTR SS:[ESP+8],EBX
774C8F08   E9 86CEFEFF      JMP ntdll.774B5D93
774C8F0D   8D49 00          LEA ECX,DWORD PTR DS:[ECX]
774C8F10   8BD4             MOV EDX,ESP
774C8F12   0F34             SYSENTER                        Into KiFastCallEntry
774C8F14   8DA424 0000000(  LEA ESP,DWORD PTR SS:[ESP]
774C8F1B   EB 03            JMP SHORT ntdll.KiFastSystemCallRet
774C8F1D   CC               INT3
774C8F1E   CC               INT3
774C8F1F   CC               INT3
774C8F20   C3               RETN
```

이렇게 커널 모드에 접근하는 과정을 확인할 수가 있었다. 그렇다면 어떻게 다시 사용자 영역으로 복귀할까? 이 역시 두 명령어가 차이를 보인다. SYSENTER 명령어의 경우 커널 영역에서 "SYSEXIT" 명령어를 통해 사용자 영역으로 돌아오게 되며, INT 0x2E의 경우 "IRET" 명령어를 통해 원래의 코드로 복귀가 가능하다.

IDT Hooking

IDT는 256개의 Enyrt로 이루어진 배열이며 엔트리 하나당 하나의 인터럽트에 대응되며 이러한 각 인터럽트는 IDT로부터 처리할 함수의 주소(ISR)를 전달받는다. 다시 말해, IDT에서는 각 인터럽트를 처리할 함수의 주소를 갖고 있다. 인터럽트 테이블의 메모리 주소를 얻으려면 IDTR 레지스터 값을 읽어야 하는데 이는 "sidt" 명령을 통해 알 수가 있으며, 반대로 'lidt' 명령을 통해 IDTR 레지스터의 값을 변경할 수 있다. sidt 명령에 의해 반환되는 idt 데이터 구조는 아래와 같다.

```
typedef struct
{
    unsigned short IDTLimit;
    unsigned short LowIDTbase;
    unsigned short HiIDTbase;
}
```

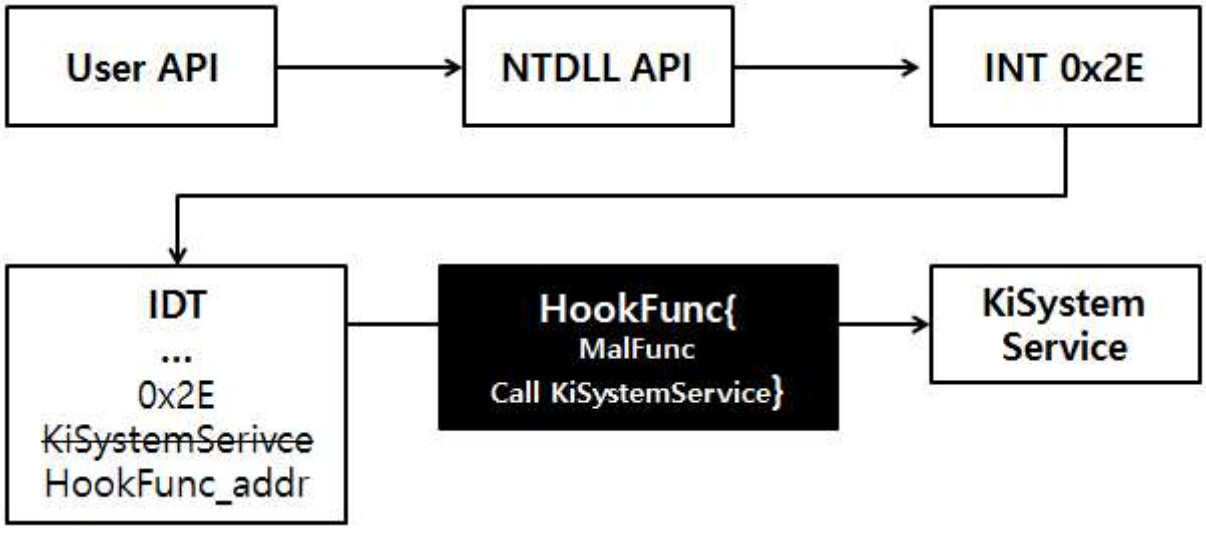
IDT의 주소는 위 구조체에서 확인할 수 있듯이, 하위 주소와 상위 주소가 나누어져 있다. 이를 통해 IDT의 주소를 확인할 수가 있는데, 그렇다면 IDT에는 어떠한 내용이 포함되어 있는지 아래의 그림을 보자. 크게 중요한 내용은 바로 위 시스템 콜에서 볼 수 있었던 0x2E와 키보드 인터럽트

인 0x93이 존재하고 있다. 0x93의 경우 키보드 메시지 후킹과 관련된 내용이므로 자세히 다루지 않고 0x2E를 위주로 살펴보자.

```
00:    8054019c nt!KiTrap00
01:    80540314 nt!KiTrap01
02:    Task Selector = 0x0058
03:    805406e4 nt!KiTrap03
...
2a:    8053f9de nt!KiGetTickCount
2b:    8053fae0 nt!KiCallbackReturn
2c:    8053fc80 nt!KiSetLowWaitHighThread
2d:    805405c0 nt!KiDebugService
2e:    8053f481 nt!KiSystemService
2f:    80542780 nt!KiTrap0F
30:    8053eb40 nt!KiUnexpectedInterrupt0
...
91:    8053ef0a nt!KiUnexpectedInterrupt97
92:    8053ef14 nt!KiUnexpectedInterrupt98
93:    8632d74c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8632d710)
94:    8053ef28 nt!KiUnexpectedInterrupt100
...(생략)
```

INT 0x2E

응용 프로그램이 운영체제가 제공하는 API를 호출하면 NTDLL.DLL은 EAX 레지스터에 해당 시스템 함수의 번호를 로드하고 EDX 레지스터에는 그 시스템 함수로 전달되는 인자가 저장된 사용자 영역의 스택 주소를 로드한다. 그리고 INT 0x2E 명령을 수행하여 인터럽트를 발생시킨다. 이 인터럽트에 의해 유저 모드에서 커널 모드로 전환된다. 따라서 INT 0x2E 명령을 수행하여 인터럽트가 발생한 다음, IDT로부터 ISR의 주소를 얻어 실행하고자 할 때 바로 ISR의 주소를 바꾸어 IDT를 후킹 할 수가 있는 것이다. 이를 표현하면 아래와 같다.



IDT를 후킹 하는 데 있어 동작하고 있는 중 그 값이 변경되면 오류를 일으킬 수 있기 때문에 'cli' 명령을 통해 인터럽트 처리를 정지시킨 후, 해당 값이나 주소를 변경해야 한다. 모든 수정이 완료

된 이후에는 다시 인터럽트 처리를 활성화시켜야 하기 때문에 'sti' 명령을 사용하여야만 한다. 이를 위한 코드는 아래와 같다.

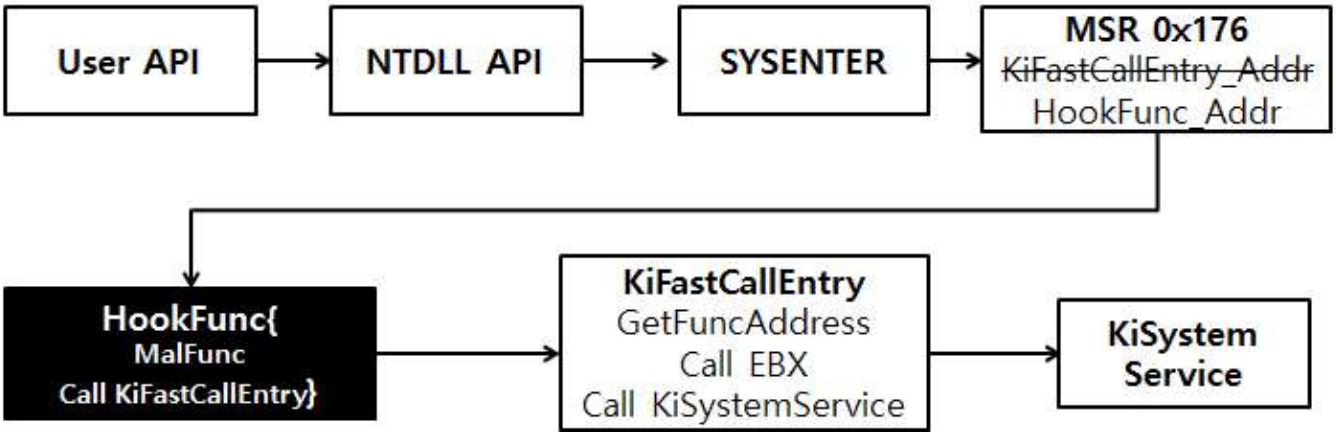
```
_asm {
    sidt idt_info;
}

idt_entries = (*IDTENTRY) MAKELONG (idt_info.LowIDTbase, idt_info.HiIDTbase);
int2e_entry = &(idt_entries[0x2E])    //IDT에서 0x2E번째 엔트리 주소

_asm {
    cli;                                                    // 인터럽트 Disable
    lea eax, HookKiSystemService;           // EAX에 후킹 함수의 주소를 저장
    mov ebx, int2e_entry;                       // IDT에서 0x2E번째 엔트리의 주소
전달
    mov [ebx], ax;                                    // 후킹 함수 하위 16비트 주소
를 IDT엔트리에 기록
    shr eax, 16;
    mov [ebx+6], ax;                                // 후킹 함수 상위 16비트 주소
또한 기록
    sti;                                                    // 인터럽트 Enable
}
```

MSR Hooking

그렇다면 SYSENTER의 경우는 어떻게 될까? 윈도우 XP 이상에서는 INT 0x2E가 아닌 SYSENTER를 사용하는데, 시스템 콜이 요청되면 NTDLL은 EAX에 해당 시스템 콜의 번호를 로드하고 EDX 레지스터에서는 포인터 레지스터인 ESP를 로드한다. 그다음 NTDLL은 SYSENTER 명령을 실행한다. SYSENTER 명령이 실행되면 커널 영역에 들어가게 된다. 이때 그냥 커널로 들어가는 것이 아니라 실행되어질 커널의 주소인 MSR 0x176 레지스터(KiFastCallEntry) 내부에 있는 값으로 이동하게 된다. 그 후 KiFastCallEntry는 EAX에 저장되어 있는 시스템 콜 번호를 가지고 SSDT에서 Nt 함수 주소로 가져오고 해당 함수를 호출한다. SSDT에 대해선 뒤에서 더 자세히 알아볼 것이다.



위 그림과 같이 나타낼 수 있으며, 바로 MSR 0x176의 값을 변경하므로 이를 수정하는 것이다. 이를 변경하기 위해서는 rdmsr 명령어와 wrmsr 명령어를 사용하여 msr의 값을 읽고 조작하여야 한다. 이를 위한 코드는 아래와 같다.

```
_asm {
    mov ecx, 0x176          // MSR 0x176 지정
    rdmsr                   // 어떠한 주소가 있지 읽음
    mov  OrigFunc, eax    // 기존에 있던 값을 저장
    mov eax, HookFunc
    wrmsr                   // MSR 0x176에 HookFunc의 주소를 기록함
}
```

이렇게 후킹 작업을 완료한 다음, SYSENTER 명령으로 인해 MSR 0x176에 존재하고 있는 KiFastCallEntry의 주소가 아닌 훅 함수의 주소가 호출되므로 인해 KiFastCallEntry가 아닌 HookFunc()이 먼저 호출된 다음 KiFastCallEntry가 호출된다. 실제로 이러한 후킹 작업이 성공적으로 이루어졌다면 훅 함수에서는 EAX 레지스터에 있는 값을 통해 어떠한 시스템 콜을 요청했는지 확인할 수가 있다. 이는 다시 말해 특정한 시스템 콜을 지정하여 해당 시스템 콜은 거부되도록 할 수 있다.

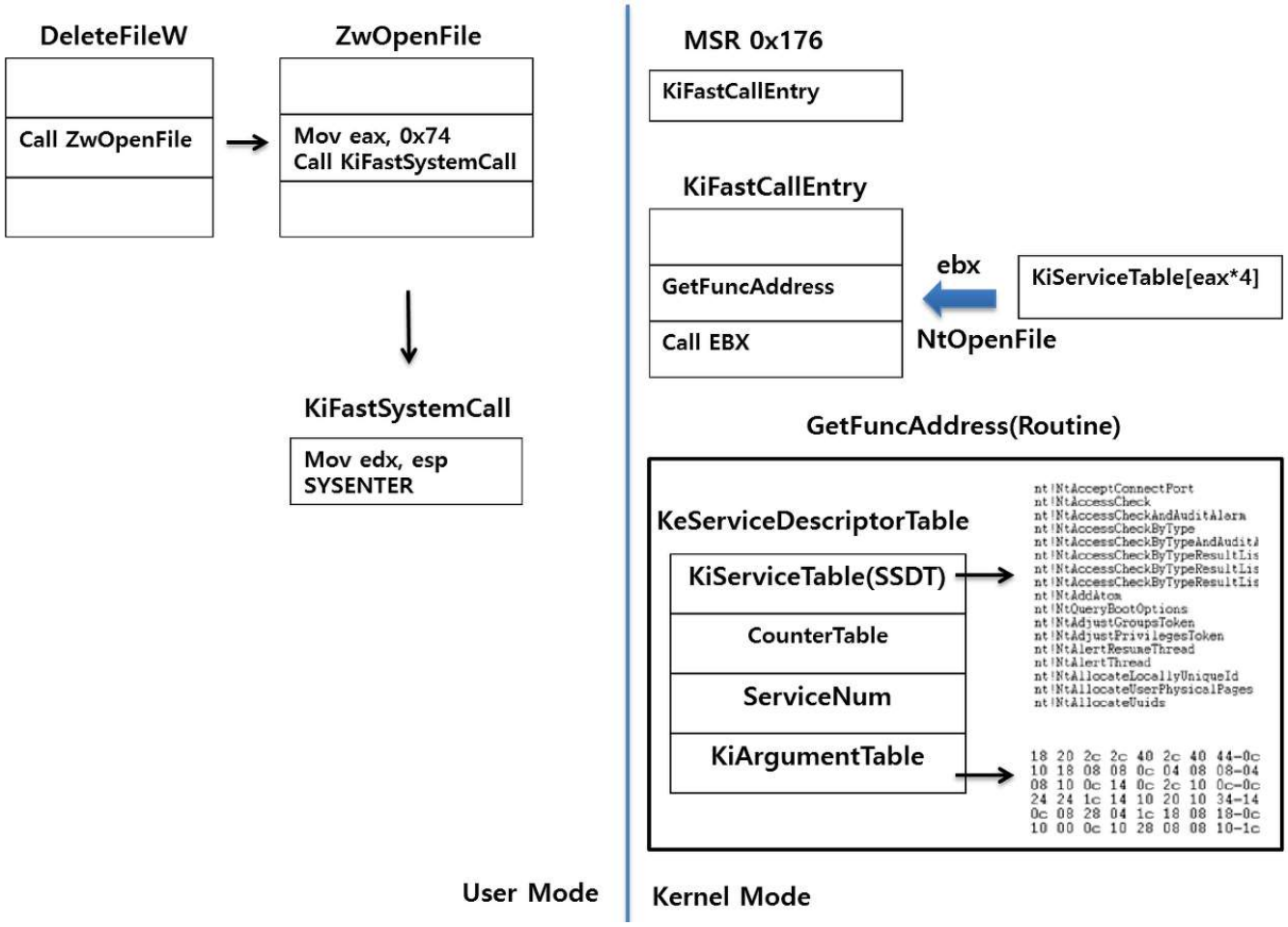
단, 시스템 콜은 빈번하게 이루어지는 작업이기 때문에 과도한 조건문을 걸어놓는 것은 시스템의 성능을 크게 하락시키므로 사용자가 속도의 변화를 체감할 수도 있다. 따라서 빈번하게 호출되는 만큼 최적화된 내용만을 후킹 함수에 넣어야 한다.

SSDT Hooking

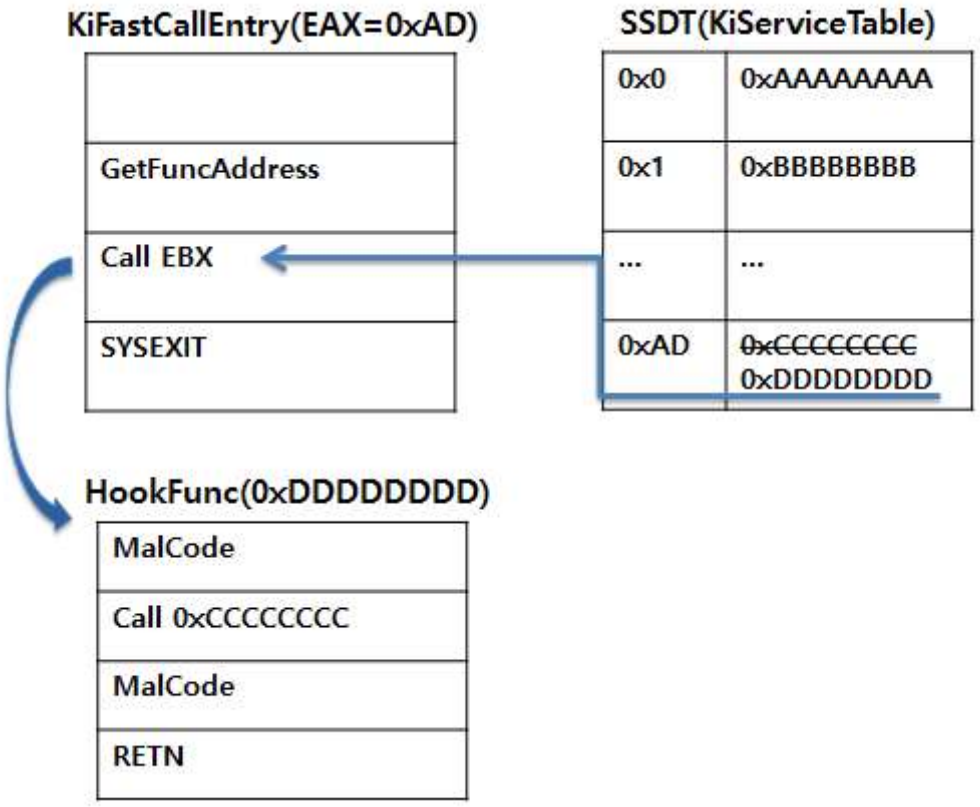
시스템 서비스 디스패치 테이블은 시스템 콜을 처리하기 위한 함수를 찾을 때 사용된다. 위에서 프로그램이 시스템 콜을 호출하는 두 가지 방법(INT 0x2E, SYSENTER)에 대하여 알아보았다.

이 두 명령어를 통해 결국 KiSystemService 함수(시스템 서비스 디스패처)를 호출하게 되는데, 이 함수는 EAX 레지스터에서 시스템 콜 번호를 읽어 SSDT에서 해당 시스템 콜의 루틴을 찾는다.

또한 KiSystemService는 시스템 콜의 인자를 유저 모드 스택에서 커널 모드 스택으로 복사하는데, 이때 EDX 레지스터가 시스템 콜에 사용할 인자를 가리키고 있다(몇몇 루트킷은 이런 시스템 콜 처리 과정에서 시스템 콜의 인자를 변경하거나 시스템 콜의 수행 루틴을 변경시키기도 한다). 시스템 콜 번호에 맞게 KeServiceDescriptorTable을 참조하여 Native API를 호출한다. 그 후 시스템 콜을 종료하고 각각 IRET나 SYSEXIT 명령어를 사용하여 유저 모드로 복귀한다. 아래의 그림은 이를 종합적으로 표현한 내용이다.



결국 SSDT에서 서비스 호출 번호에 맞는 주소를 얻은 다음 이를 호출하는 형태로 진행되는 것이다. 그렇다면 SSDT Hooking은 어느 부분에서 후킹 해야 하는 것일까? 바로 GetFuncAddress 과정에서 인덱스 번호에 맞는 함수의 주소를 SSDT에서 가지고 올 때이다. 예를 들어, 0xAD 서비스 함수(EAX=0xAD) 시스템 콜이 발생하면 SSDT에서 0xAD번째에 위치한 함수의 주소를 가지고 오게 된다. 그리고 해당 함수를 호출하므로 진행되는 방식인데, 만약 SSDT를 변조하므로 0xAD번째에 위치한 함수의 주소를 HookFunc의 주소로 대체하면 해당 시스템 콜이 발생할 때마다 HookFunc가 호출된다.



위 예를 좀 더 구체적으로 제시하자면 0xAD 시스템 콜의 루틴이 SSDT에서 0xCCCCCCCC로 존재하고 있는 상황에 이 주소를 Hook함수의 주소인 0xDDDDDDDD로 대체하게 되면 0xAD 시스템 콜이 발생할 때마다 0xDDDDDDDD의 함수가 호출되게 된다. 대개 이러한 후킹 함수는 정상적인 루틴 0xCCCCCCCC를 조작하는데, 어떠한 인자가 오느냐에 따라 이를 진행하지 않거나 원래 함수의 결과가 특정 값일 경우 이를 변조하여 반환하는 등의 조작을 가할 수가 있다.

하지만 SSDT는 Read Only로 설정되어 있기 때문에 SSDT Hooking을 진행하기 위해서는 쓰기 권한이 필요하다. 이를 우회하기 위한 방법 중 하나인 CR0 Register에 대하여 알아보자. CR0 레지스터는 WP(Write Protect) Bit를 포함하고 있는데 이 값이 0 이면 메모리 보호 기능이 해제되고 1 이면 메모리 보호가 활성화된다. 따라서 메모리 보호 기능을 비활성화하므로 쓰기 권한을 얻을 수가 있다. 해당 코드는 다음과 같다.

```
__asm {    // 메모리 보호 기능 비활성화
    push eax
    mov eax, CR0
    and eax, 0xFFFEFFFF
    mov CR0, eax
    pop eax}

__asm {    // 메모리 보호 기능 활성화
    push eax
    mov eax, CR0
    or eax, NOT 0xFFFEFFFF
```

```
    mov CR0, eax
    pop eax}
```

이러한 후킹은 목적은 대개 흔적을 남기지 않는 것이 중요하므로, SSDT를 후킹 한 다음에는 다시 CR0를 조작하여 메모리 보호 기능을 활성화해주어야 한다. 이는 비활성화와는 반대로 or eax, NOT 0xFFFEFFFF 연산을 해주어야 한다. 이제 SSDT에 값을 기록할 수 있으므로 어떻게 변조할 것인가에 대하여 알아보자. SSDT Hooking에는 유용하게 사용되는 몇 가지 매크로가 존재한다.

"SYSTEMSERVICE" 매크로는 ntoskrnl.exe에서 제공하는 Zw* 함수의 주소를 입력받아 그에 상응하는 Nt* 함수의 주소를 SSDT에서 구할 때 사용한다. "SYSCALL_INDEX" 매크로는 Zw* 함수의 주소를 입력받아서 SSDT에서 해당 함수의 인덱스 번호를 구하는 데 사용할 수 있다. 이렇게 "SYSTEMSERVICE"와 "SYSCALL_INDEX" 매크로가 해당 함수의 시작 부분(_func)에 +1을 하는 이유는 opcode에 따라 [mov eax, 인덱스 값] 이므로 해당 인덱스 값은 함수의 시작점에서 mov eax의 opcode 값 다음에 오므로 +1을 더해 해당 값을 알 수 있기 때문이다. 이 두 매크로를 통해 각각 Nt* 함수의 주소와 인덱스 번호를 구할 수가 있다.

```
#define SYSTEMSERVICE(_func)  KeServiceDescriptorTable.ServiceTableBase[ *
(PULONG)
                                                                ((PUCHAR)_fu
#define SYSCALL_INDEX (_func) *(PULONG) ((PUCHAR) _func+1)
```

HOOK_SYSCALL과 UNHOOK_SYSCALL 매크로는 후킹을 수행할 Zw* 함수의 주소와 SSDT에서의 인덱스, 그리고 새로운 HookFunc 함수의 주소를 이용해 SSDT 안에서의 주소를 변경해준다. 밑의 코드 중에서 "InterlockedExchange"가 있는데 이는 두 개의 인자 값이 일치하지 않을 경우 첫 번째 인자가 지정하는 메모리의 값을 두 번째 인자로 바꾸는 함수이다.

```
#define HOOK_SYSCALL (_func, _Hook, _Orig)₩
            _Orig=(PVOID) InterlockedExchange((PLONG)₩
            &MappedSystemCallTable[SYSCALL_INDEX(_func)],(LONG) _Hook)
#define UNHOOK_SYSCALL(_func, _Hook, _Orig)₩
            InterlockedExchange((PLONG)₩
            &MappedSystemCallTable[SYSCALL_INDEX(_func)], (LONG) _Orig)
```

이러한 과정을 통해 SSDT에 존재하고 있는 서비스 루틴을 훅 함수로 대체할 수가 있고, 다시 원래대로 되돌릴 수가 있다.

Reference

http://luckyyowu.tistory.com/133

운영체제 04 : 시스템 콜 (시스템 호출, System Call)

참고 도서는 'Operating System Concepts 8th' 입니다. (포스팅 하단부 참고) 개인공부 후 자료를 남기기 위한 목적으로 포스팅합니다. 내용 상에 오류가 있을

luckyyowu.tistory.com

http://hackerspace.tistory.com/entry/시스템-호출-처리순서

윈도우 시스템 호출 처리 과정

개요 Windows Application 에서 WinAPI 함수로 시스템 호출을 하였을때 내부 동작 간략 정리 [Application 에서 OpenFile API 함수를 호출

hackerspace.tistory.com

http://bbolmin.tistory.com/158

SSDT 후킹

[ref - SSDT_Hooking.pdf - Written by 백구] Native API의 호출과정은 위의 그림과 같다. 1. INT 0x2E나 SYSENTER 같은 시스템 콜

bbolmin.tistory.com

http://luckey.tistory.com/86

SYSENTER 와 INT 0x2E

사용자 특권 즉 어플리케이션 레벨에서 커널특권으로 이동할 경우 사용되는 명령어는 SYSENTER과 INT 0x2E 가 있다. SYSENTER과 INT 0x2E에 대해서 일단 알아보

luckey.tistory.com

http://securityfactory.tistory.com/158

[개념 이해] Windows System Call

오늘은 Windows에서 System Call을 하는 과정에 대해 살펴보고자 합니다. 우리가 사용하는 Windows OS는 User 영역과 Kernel 영역으로 나뉘어져 있습니다

securityfactory.tistory.com

https://en.wikipedia.org/wiki/Interrupt_descriptor_table

Interrupt descriptor table - Wikipedia, the free encyclopedia

The Interrupt Descriptor Table (IDT) is a data structure used by the x86 architecture to implement a

en.wikipedia.org

http://blog.naver.com/wwwkasa/80167132015

IDT 후킹 - 키보드 키로거

IDT 후킹 - 키보드 키로거 인터럽트 처리 함수의 주소를 가지고 있는 IDT를 후킹하여 인터럽트 발생

blog.naver.com

http://ezbeat.tistory.com/279

SYSENTER Hooking

이번에 해볼 내용은 SYSENTER를 후킹해보도록 하겠습니다. SYSENTER를 후킹하는건 간단하데 이것을 사용해 무언을 구현해 보려니 그 부분이 좀 시간이 걸렸습니다 먼저

ezbeat.tistory.com

SSDT(System Service Descriptor Table) Hooking.pdf - Written by 백구

공감

'Reversing > Theory' 카테고리의 다른 글

Hooking, IDT, Kernel, Kernel_Hooking, reversing, SSDT, SystemCall, windows

Trackback +0

Comment +0

확인

‹ Prev 1 ··· 13 14 15 16 17 18 19 20 21 ··· 50 Next ›

최근에 올라온 글

최근에 달린 댓글

잘보고 갑니다. 원리 설명을..

부족한 지식으로 답변 남기자..

여기는 외계인만 모여있나? ㅎ..

안녕하세요. 오래된 글이지만..

진짜 정리 잘하셨네요 감사합..

안녕하세요 포스팅해주시는 글..

궁금한게있는데요 윈도우환경..

Nethost.exe 실제 악성프로그..

전...복호화하기 귀찮아서 분..

저는 운이 좋아서 30분만에 풀..

공지사항

글 보관함

2018/10 (1)

2018/04 (2)

2018/02 (1)

2017/11 (1)

최근에 받은 트랙백

« 2018/12 »

일	월	화	수	목	금	토
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

링크

rooov.com

total : 146,574
today : 0
yesterday : 146