

## 목차

목차.....	1
저작권.....	1
소개.....	1
연재 가이드.....	1
필자소개.....	2
Introduction.....	2
SDT 후킹.....	2
SDT 복구.....	6
SDT 재배치.....	7
SDT를 찾는 또 다른 방법.....	7
전용 SDT.....	9
인라인 후킹 vs 트랩펄린.....	10
후킹을 넘어서.....	12
극과 극은 통한다.....	13
참고자료.....	13

## 저작권

Copyright © 2009, 신영진

이 문서는 Creative Commons 라이선스를 따릅니다.

<http://creativecommons.org/licenses/by-nc-nd/2.0/kr>

## 소개

유저모드에서 시작된 루트킷과 보안제품의 싸움은 커널모드에서도 계속된다. 3부에서는 커널모드 API 후킹의 대표적인 기법인 SDT 후킹의 원리에 대해서 살펴보고, 이를 둘러싼 해커와 보안 개발자들 사이의 전쟁에 관한 이야기를 들어본다. 다양한 기술 속에서 그들의 아이디어를 훑쳐보자.

## 연재 가이드

운영체제: Windows XP

개발도구: Visual Studio 2005

기초지식: 어셈블리, windbg 사용법

응용분야: 보안 프로그램

## 필자소개

신영진 pop@jiniya.net, <http://www.jiniya.net>

웹비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

## Introduction

사람들은 싸움 구경 하는 것을 좋아한다. 고대 그리스 로마 시대부터 지금까지 격투기가 끊이지 않고 인기를 얻고 있는 것을 보면 그러한 사람의 본능을 어느 정도는 짐작할 수 있다. 이러한 본능은 물리 세계를 떠나서 정신적인 영역에까지 이어지고 있다. 3부에서는 SDT 후킹을 둘러싼 해커와 보안 개발자들 사이의 전쟁에 관한 이야기를 다룰 것이다.

지면 관계상 세부적인 구현보다는 전체적인 흐름과 각 기술의 아이디어에 집중해서 설명할 것이다. 준비해야 할 것은 연습장, 필기구, 가상 PC에 설치된 Windows XP, 그리고 windbg이 전부다. 약간의 어셈블리 지식과 커널모드 프로그래밍 지식이 있다면 좀 더 큰 도움이 될 것이다. 그러한 번 시작해보자.

## SDT 후킹

SDT(Service Descriptor Table) 후킹은 시스템 서비스 테이블의 값을 조작하는 API 후킹의 한 방법이다. 유저모드 API 후킹의 경우 프로세스간 컨텍스트가 다르기 때문에 시스템 전역 후킹을 하기 위해서는 일일이 모든 프로세스에 관련 코드를 주입(injection) 시켜야 한다는 불편한 점이 있었다. 이러한 문제를 해결한 한 가지 방법이 SDT 후킹이다. SDT 후킹은 커널모드에서 한 번만 후킹을 하면 모든 프로세스에 적용되기 때문에 별도로 컨텍스트 관리를 할 필요가 없기 때문이다.

SDT 후킹의 방법을 이해하기 위해서는 우선 Windows의 API 호출이 어떻게 커널로 전달되는지를 알아야 한다. 간단하게 windbg를 통해서 TerminateProcess 호출이 커널로 전달되는 과정을 살펴 보도록 하자. 응용 프로그램에서 호출한 TerminateProcess는 ntdll.dll의 NtTerminateProcess로 이어진다. <리스트 1>에 windbg를 통해서 디스어셈블한 NtTerminateProcess 함수가 나와있다. eax에 0x101를 넣고, 0x7ffe03000에 저장된 번지를 호출하는 것을 알 수 있다.

### 리스트 1 NtTerminateProcess

```
kd> uf ntdll!NtTerminateProcess
ntdll!NtTerminateProcess:
7c93e88e b801010000      mov     eax,101h
7c93e893 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
7c93e898 ff12           call   dword ptr [edx]
7c93e89a c20800        ret     8
```

<리스트 2>에 나와있는 것처럼 0x7ffe0300은 edx에 현재 스택 포인터를 저장하고 sysenter 명령

어를 호출하는 코드임을 알 수 있다. sysenter 명령어는 유저모드에서 커널모드로 진입하기 위한 명령어로 MSR에서 실행할 커널 함수 주소를 가져와서 그 번지로 이동시키는 역할을 한다. rdmsr 명령어를 통해서 sysenter 명령어가 수행되었을 때 실행되는 함수를 찾아보면 KiFastCallEntry라는 것을 알 수 있다. KiFastCallEntry는 eax에 저장되어 있는 서비스 번호에 해당하는 함수를 호출해주는 역할을 한다.

## 리스트 2 0x7ffe0300 덤프 내용

```
kd> dd 7ffe0300
7ffe0300 7c93eb8b 7c93eb94 00000000 00000000

kd> uf 7c93eb8b
ntdll!KiFastSystemCall:
7c93eb8b 8bd4          mov     edx,esp
7c93eb8d 0f34          sysenter
7c93eb8f 90            nop
7c93eb90 90            nop
7c93eb91 90            nop
7c93eb92 90            nop
7c93eb93 90            nop
7c93eb94 c3           ret

kd> rdmsr 0x176
msr[176] = 00000000`804e06f0

kd> u 804e06f0
nt!KiFastCallEntry:
804e06f0 b923000000    mov     ecx,23h
804e06f5 6a30          push   30h
804e06f7 0fa1          pop    fs
804e06f9 8ed9          mov    ds,cx
804e06fb 8ec1          mov    es,cx
804e06fd 8b0d40f0dfff  mov    ecx,dword ptr ds:[0FFDFF040h]
804e0703 8b6104          mov    esp,dword ptr [ecx+4]
804e0706 6a23          push   23h
```

여기까지 과정을 요약하면 이렇다. TerminateProcess 호출은 ntdll.dll의 NtTerminateProcess로 이어지고, NtTerminateProcess는 eax에 TerminateProcess API의 서비스 함수 번호인 0x101을 넣고 커널모드에서 KiFastCallEntry를 호출한다. KiFastCallEntry는 SDT를 참조해서 eax에 저장된 서비스 번호에 해당하는 함수를 호출한다.

이제 실제로 SDT에 대해서 살펴보도록 하자. SDT는 총 네 개의 서비스 테이블로 구성된다. 각 테이블은 <리스트 3>에 나와있는 SDE 구조체로 이루어져 있다. SDE 테이블의 각 필드별 설명은 <표 1>에 나와 있다. SDT의 첫 번째 테이블은 Windows의 네이티브(native) API 함수를 저장하고 있는 ntoskrnl 테이블이다. 두 번째 테이블은 GUI 함수들을 저장하고 있는 win32k 테이블이다. 세 번째, 네 번째 테이블은 추후 사용하기 위해서 예약된 테이블이다.

### 리스트 3 SDT 구조체

```

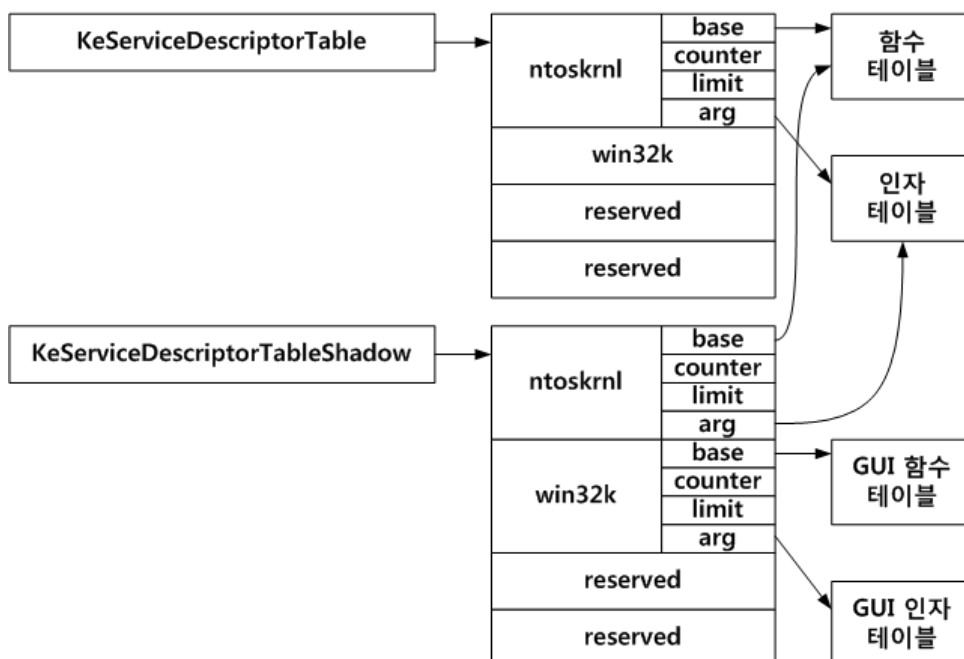
typedef struct ServiceDescriptorEntry {
    PDWORD base;
    PDWORD counter;
    DWORD limit;
    PBYTE arg;
} SDE;

typedef struct ServiceDescriptorTable {
    SDE ntoskrnl;
    SDE win32k;
    SDE reserved[2];
} SDT;
    
```

표 1 SDE 필드별 의미

필드명	역할
base	함수 테이블의 시작 주소를 가리킨다.
counter	함수 호출 회수 테이블의 시작 주소를 가리킨다. 디버그 버전 커널에서만 사용되며, 각 함수가 몇 번 호출되었는지를 기록하는 용도로 쓰인다.
limit	함수 테이블에 포함된 함수의 개수를 저장하고 있다.
arg	각 함수로 넘어오는 인자 크기를 저장하고 있는 테이블의 시작 주소를 가리킨다. 인자 크기는 바이트 단위로 저장된다.

Windows 커널에는 기본적으로 두 종류의 SDT가 포함되어 있다. KeServiceDescriptorTable은 일반적인 스레드에 사용되는 테이블로 ntoskrnl 테이블만 사용하고, 나머지 테이블은 모두 사용하지 않는다. KeServiceDescriptorTableShadow는 GUI 스레드에 사용되는 테이블로 ntoskrnl과 win32k 테이블을 사용한다. <그림 1>은 이러한 구조를 도식화한 것이다.



## 그림 1 SDT 구조

windbg를 통해서 실제 커널의 SDT를 살펴보도록 하자. <리스트 4>에 커널의 SDT 구조체를 덤프한 내용이 나와있다. 살펴보면 ntoskrnl 테이블의 base 주소는 0x84e46a8, counter는 0, limit은 0x11c, arg 필드 값은 0x80514eb8인 것을 알 수 있다. 함수 테이블을 덤프한 것을 보면 첫 번째 함수의 시작 주소는 0x80581302인 것을 알 수 있고, 인자 크기는 0x18 바이트란 것을 알 수 있다. 한 가지 주의해서 보아야 할 사실은 GUI 함수 테이블과 GUI 인자 테이블은 커널 내부가 아닌 win32k.sys 내부에 존재한다는 것이다. <리스트 4>에서도 이 두 테이블의 주소만 확연히 틀린 것을 확인할 수 있다.

## 리스트 4 커널 내부 SDT 구조체의 셀체

```
kd> dd KeServiceDescriptorTable
8055b680 804e46a8 00000000 0000011c 80514eb8
8055b690 00000000 00000000 00000000 00000000
8055b6a0 00000000 00000000 00000000 00000000
8055b6b0 00000000 00000000 00000000 00000000

kd> dd KeServiceDescriptorTableShadow
8055b640 804e46a8 00000000 0000011c 80514eb8
8055b650 bf999280 00000000 0000029b bf999f90
8055b660 00000000 00000000 00000000 00000000
8055b670 00000000 00000000 00000000 00000000

kd> dd 804e46a8
804e46a8 80581302 8057ab8c 8058c7ae 805917e4
804e46b8 805915fe 806387a0 8063a931 8063a97a
804e46c8 8057660b 806491cf 80637f5f 80590b85
804e46d8 806300a4 8057ce31 8058dc26 806271bd

kd> db /c 8 80514eb8
80514eb8 18 20 2c 2c 40 2c 40 44 . , , @ , @D
80514ec0 0c 08 18 18 08 04 04 0c .....
80514ec8 10 18 08 08 0c 04 08 08 .....
80514ed0 04 04 0c 08 0c 04 04 20 .....
80514ed8 08 10 0c 14 0c 2c 10 0c .....,...
```

이제 SDT에서 TerminateProcess API의 서비스 함수를 찾아 보도록 하자. <리스트 1>을 보면 eax에 0x101을 집어넣고 있으므로 TerminateProcess API의 서비스 함수 번호는 0x101이란 것을 알 수 있다. 함수 테이블은 4바이트 기준이고, 인자 테이블은 한 바이트 기준이란 점을 생각해서 해당 내용을 덤프 해보면 <리스트 5>와 같다. TerminateProcess의 서비스 함수 주소는 0x80586740이고, 인자는 0x08 바이트 크기인 것을 알 수 있다. 해당 주소를 디스어셈블하면 커널 내부의 NtTerminateProcess 함수가 출력된다.

## 리스트 5 TerminateProcess에 해당하는 서비스 함수

```
kd> dd 804e46a8 + 0x101 * 4
804e4aac 80586740 8057dbba 8057e5fb 80546fe0
```

```
804e4abc 806491e3 8061a730 8064e317 8064e514
```

```
kd> db /c 8 80514eb8 + 0x101
80514fb9 08 08 00 10 10 04 04 08 .....
80514fc1 14 10 08 08 10 14 0c 04 .....
```

```
kd> u 80586740
nt!NtTerminateProcess:
80586740 8bff          mov     edi,edi
80586742 55             push   ebp
80586743 8bec          mov     ebp,esp
```

이제 끝으로 SDT 후킹을 하는 방법을 생각해보자. SDT 후킹이라 함수 테이블의 번지를 바꾸는 작업을 하는 것을 말한다. <리스트 5>에서 0x101번에 해당하는 서비스 함수 주소는 0x80586740이다. 이 값을 우리가 만든 임의의 함수 주소로 변경한다면, 응용 프로그램에서 TerminateProcess를 호출할 때 원본 서비스 함수가 아닌 우리가 새롭게 작성한 함수가 호출된다.

## SDT 복구

루트킷은 API 호출 결과를 조작하기 위해서, 보안 프로그램은 시스템 상황을 모니터링하기 위해서 SDT 후킹을 광범위하게 사용한다. 이런 경쟁 조건에서는 먼저 실행되어서 SDT를 후킹한 쪽에 우선권이 주어진다. 예를 들어 루트킷이 NtQuerySystemInformation 함수를 후킹한 상태를 생각해 보자. 이후 실행되는 보안 프로그램에서 호출하는 NtQuerySystemInformation의 결과값은 루트킷에 의해서 조작된 값이기 때문에 신뢰할 수 없게 된다. 따라서 보안 프로그램은 자신이 동작하는 환경이 안전한지를 점검하는 것이 최우선 과제라 할 수 있다.

이러한 문제를 해결하기 위해서 나온 것이 SDT 복구 기술이다(<Win2K/XP SDT Restore 0.2> 참고). 이 기술의 원리는 간단하다. 현재 시스템의 SDT와 파일로 존재하는 커널의 SDT를 비교해서 같은지 다른지를 비교하는 것이다. SDT 후킹을 하더라도 변경되는 것은 현재 메모리의 내용일 뿐 파일의 내용은 원본 그대로이기 때문이다. 이 방법의 전체적인 실행 순서는 다음과 같다.

1. 현재 로드된 커널의 이미지 이름과 로드 주소를 알아낸다.
2. 로드된 커널의 KeServiceDescriptorTable의 base 주소를 알아낸다.
3. 동일한 이름의 커널을 LoadLibraryEx를 통해서 메모리에 로드시킨다.
4. 로드된 이미지에서 base와 동일한 오프셋에 있는 테이블 내용을 비교한다.

<리스트 6>은 이를 간단한 의사 코드로 만들어본 것이다. 핵심적인 부분은 함수 테이블 주소의 오프셋을 사용해서 LoadLibraryEx로 로드한 커널에서 해당 부분을 찾아내는 것이다. 현재 로드된 커널의 이미지 이름과 베이스 주소는 NtQuerySystemInformation 함수를 사용해서 구할 수 있다.

### 리스트 6 SDT 복구 의사 코드

```
kernelBase = 로드된 커널 주소
kernelName = 로드된 커널 이름
```

```

kernelSDTBase = KeServiceDescriptorTable[0].base
kernelSDTLimit = KeServiceDescriptorTable[0].limit

imageBase = LoadLibraryEx(kernelName, DONT_RESOLVE_DLL_REFERENCES);
imageSDTBase = imageBase + (kernelSDTBase - kernelBase);

for(DWORD i=0; i<kernelSDTLimit; ++i)
    kernelSDTBase[i] = imageSDTBase[i] - imageBase + kernelBase;

```

## SDT 재배치

참 아이러니 한 사실은 보안 제품에서 루트킷을 잡기 위해서 개발된 SDT 복구 기술이 역으로 루트킷에서 사용한다는 것이다. 보안 제품 또한 시스템 모니터링을 하기 위해서 SDT 후킹을 사용하는데 루트킷이 보안 제품을 무력화하기 위해서 SDT 복구를 사용하는 것이다.

이러한 루트킷에 대항하기 위해서 만들어진 기술이 SDT를 재배치하는 방법이다. 이 방법은 앞서 살펴본 SDT 복구가 로드된 커널의 KeServiceDescriptorTable[0].base에서 찾는다라는 사실을 이용한다. base 필드의 값이 커널 이미지 외부에 있다면 로드한 커널 파일에서는 그 부분을 찾을 수 없다. 메모리를 할당한 다음 해당 영역으로 함수 테이블을 복사하는 것이 핵심이다. 그리고 커널의 base 필드 값을 할당된 메모리로 연결해주면 된다. 이렇게 변경된 경우의 테이블 구조가 <그림 2>에 나와 있다.

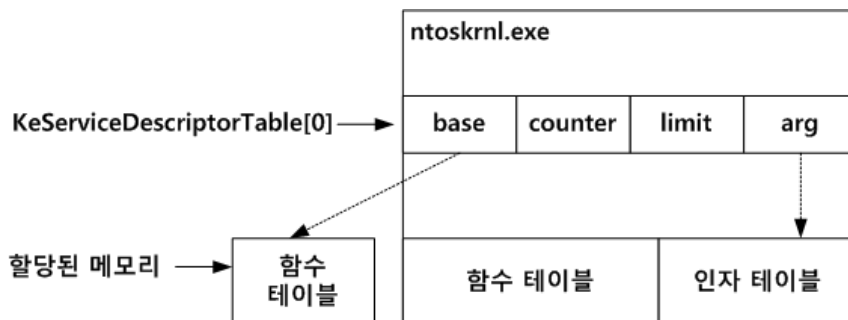


그림 2 SDT 재배치 후의 구조

## SDT를 찾는 또 다른 방법

SDT 재배치 방법이 나오기 얼마지 않아 루트킷 진영에서는 오리지널 커널 이미지 파일로부터 SDT의 주소를 알아내는 새로운 방법이 나왔다(<A more stable way to locate real KiServiceTable> 참고). 이 방법은 커널의 초기화 코드에서 base 주소를 찾아내는 방법을 사용한다. KeServiceDescriptorTable은 커널의 KiInitSystem이란 함수에서 <리스트 7>과 같은 형태로 초기화 된다. base 주소에는 KiServiceTable이란 고정적인 값이 할당되기 때문에 이 부분을 통해서 base 주소의 기본 값을 찾아낼 수 있다.

### 리스트 7 KeServiceDescriptorTable 초기화 과정

```

KeServiceDescriptorTable[0].base = &KiServiceTable[0];
KeServiceDescriptorTable[0].count = NULL;

```

```

KeServiceDescriptorTable[0].limit = KiServiceLimit;
KeServiceDescriptorTable[0].arg = &KiArgumentTable[0];
for (Index = 1; Index < NUMBER_SERVICE_TABLES; Index += 1) {
    KeServiceDescriptorTable[Index].Limit = 0;
}

```

이 방법의 전체적인 실행 순서는 다음과 같다. 글만 보고는 의미를 이해하기가 쉽지 않기 때문에 <그림 3>에 나와 있는 명령어 구조와 재배치 오프셋의 관계를 보면서 의미를 파악하도록 하자. <리스트 8>에는 재배치 정보에서 SDT 주소를 찾아내는 과정에 대한 의사코드가 나와있다.

1. 현재 로드된 커널의 이미지 이름을 알아낸다.
2. 동일한 이름의 커널을 LoadLibraryEx를 통해서 메모리에 로드한다.
3. GetProcAddress를 통해서 KeServiceDescriptorTable 주소를 구한다.
4. 모든 재배치 정보를 조사해서 재배치하는 곳의 참조 주소가 KeServiceDescriptorTable인 것을 찾는다.
5. 찾아낸 부분의 명령어 코드 형태가 mov [mem32], imm32 형태 인지를 조사한다.
6. 5단계 까지 일치했다면 재배치 주소 + 4에 있는 값이 KeServiceDescriptorTable[0].base의 초기 값이 된다.



그림 3 재배치 오프셋과 명령어의 관계

### 리스트 8 커널 코드에서 SDT를 찾는 의사 코드

```

while(모든 재배치 정보에 대해서 반복)
{
    type = 재배치 타입
    offset = 재배치 오프셋

    // 오프셋이 가리키는 주소가 SDT인지 체크
    if(*offset == KeServiceDescriptor[0].base)
    {
        opcode = (PWORD)((PBYTE) offset - 2);

        // 명령어가 mov mem32, imm32인지 체크
        if(*opcode == 0x05c7)
        {
            // base 주소는 offset + 4에 기록되어 있는 값
            base = (PDWORD)((PBYTE) offset + 4);
            return *base;
        }
    }
}

```

## 전용 SDT

SDT를 둘러싼 이러한 싸움이 한창일 때 SDT 후킹에 대한 원론적인 의문을 품은 사람들이 있었다. 그 의문은 다름아닌 SDT 후킹이 과연 시스템 전역 후킹인가에 대한 것이었다. 그들은 좀 더 세밀하게 커널을 분석했고, SDT 후킹이 시스템 전역이 아니라는 결론에 도달하기에 이르렀다(<ksthb v1.0> 참고).

<리스트 9>에 나와있는 것처럼 커널 스레드 구조체에는 ServiceTable이란 필드가 있다. 이 필드의 역할은 해당 스레드의 서비스 테이블을 가리키는 역할을 한다. 앞서 살펴본 것과 같이 일반적인 경우에 모든 스레드는 <그림 4>에 나타난 것처럼 KeServiceDescriptorTable이나 KeServiceDescriptorTableShadow 중에 하나의 서비스 테이블을 가리키기 때문에 SDT 후킹이 시스템 전역인 것처럼 보였던 것이다.

### 리스트 9 KTHREAD 구조체 필드

```
kd> dt _kthread
ntdll!_KTHREAD
+0x000 Header          : _DISPATCHER_HEADER
+0x010 MutantListHead  : _LIST_ENTRY
+0x018 InitialStack    : Ptr32 Void
...
... 종략 ...

+0x0e0 ServiceTable    : Ptr32 Void
+0x0e4 Queue           : Ptr32 _KQUEUE
+0x0e8 ApcQueueLock    : Uint4B
+0x0f0 Timer           : _KTIMER
...
... 종략 ...
```

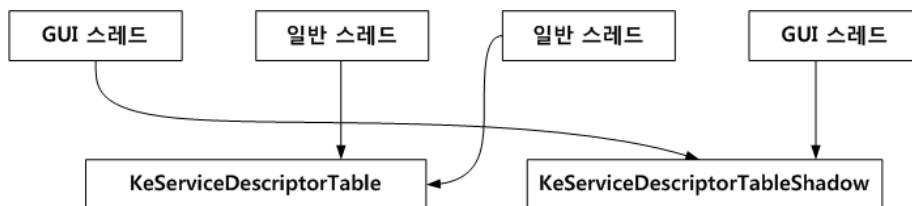


그림 4 일반적인 경우의 스레드 서비스 테이블

전용 SDT란 <그림 5>에 나타난 것처럼 특정 스레드의 ServiceTable을 수정해서 전혀 새로운 서비스 테이블과 연결시키는 방법이다. 앞서 살펴보았던 방법을 통해서 커널 파일에서 원본 SDT를 추출한 다음 그 정보를 기반으로 새로운 서비스 테이블을 만들면 SDT 후킹이 이루어진 상태라 하더라도 해당 스레드는 SDT 후킹으로부터 자유로울 수 있다.

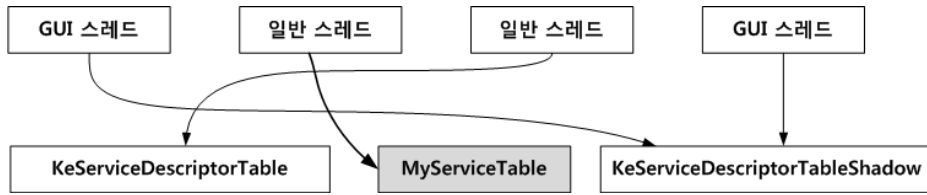


그림 5 스레드의 서비스 테이블을 변경한 경우

## 인라인 후킹 vs 트램펄린

SDT를 새롭게 만들어서 스레드에 연결시키는 방법이 나오면서 더 이상 SDT 후킹은 의미가 없어졌다. 보안 개발자들은 좀 더 하위레벨의 코드를 직접 후킹하는 방법을 선택했다. 인라인(inline) 후킹으로 불리는 이 방법은 커널 함수 도입부를 후킹된 함수로 점프하는 코드로 패칭하는 것을 말한다. <리스트 10>과 같은 NtTerminateProcess 함수 본문을 <리스트 11>과 같이 바꾸는 것이다.

### 리스트 10 NtTerminateProcess 도입부

```
kd> uf nt!NtTerminateProcess
nt!NtTerminateProcess:
80586740 8bff          mov     edi,edi
80586742 55           push   ebp
80586743 8bec          mov     ebp,esp
80586745 83ec10       sub     esp,10h
80586748 53           push   ebx
```

### 리스트 11 후킹된 NtTerminateProcess

```
jmp MyNtTerminateProcess
sub esp, 10h
push ebx
```

후킹에 관심이 있는 개발자라면 대부분 알고 있듯이 이러한 인라인 후킹은 트램펄린 함수를 사용해서 손쉽게 무력화 시킬 수 있다. 트램펄린 함수란 원본 함수와 도입부를 동일하게 구현하고 점프 코드 다음으로 바로 이동시키는 것을 말한다. <리스트 12>에는 이러한 트램펄린 함수의 한 예가 나와있다.

### 리스트 12 NtTerminateProcess 인라인 후킹을 무력화 시키는 트램펄린 함수

```
__declspec(naked) NTSTATUS TL_NtTerminateProcess(HANDLE process, NTSTATUS exitcode)
{
    __asm mov edi, edi
    __asm push ebp
    __asm mov ebp, esp
    __asm jmp NtTerminateProcess + 5
}
```

보안 개발자들의 다음 선택은 단순히 트램펄린 함수에 무력화 되지 않도록 여러 함수를 동시에 인라인 후킹하는 방법이었다. 다중 인라인 후킹으로 불리는 이 방법은 후킹을 하고자 하는 함수

에서 사용하는 다른 함수도 같이 인라인 후킹을 하는 것이다. NtTerminateProcess 함수는 내부적으로 PspTerminateThreadByPointer, ObfDereferenceObject, ObClearProcessHandleTable와 같은 함수를 사용한다. 이들 함수를 동시에 후킹하면 앞서 살펴본 TL\_NtTerminateProcess를 호출하더라도 TerminateProcess 함수 본문에서 호출하는 다른 함수에서 걸리기 때문에 단순 트램펄린 함수로는 우회하기가 쉽지 않다. <그림 6>은 해커가 이러한 상황을 공격하기 위해서 각각에 대한 트램펄린 함수를 만들어둔 화면을 보여주고 있다. 그림에 나와 있듯이 이렇게 각각의 트램펄린을 만든다고 하더라도 CALL로 연결된 2번 선 때문에 인라인 후킹을 무력화 할 수 없다.

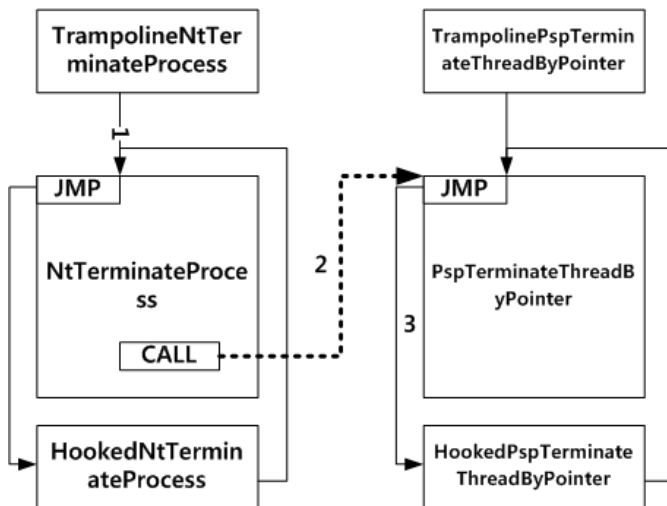


그림 6 다중 인라인 후킹

물론 그렇다고 이러한 다중 인라인 후킹을 무력화할 수 있는 방법이 전혀 없는 것은 아니다. 트램펄린 대신 함수를 통째로 복사해서 사용한다면 다중 인라인 후킹도 무력화할 수 있다. <그림 7>은 이런 방식으로 함수를 통째로 복사해서 다중 인라인 후킹을 무력화하는 방법이 나와있다. NtTerminateProcess를 원본 운영체제 코드를 복사해서 그대로 만들되 CALL 하는 부분만 PspTerminateThreadByPointer가 아닌 TrampolinePspTerminateThreadByPointer를 호출하도록 만드는 것이다. 인라인 후킹된 함수가 많다면 굉장히 번거로운 작업이 되겠지만 결국은 시간과 노력만 투자한다면 어떤 종류의 인라인 후킹이든 우회할 수는 있다.

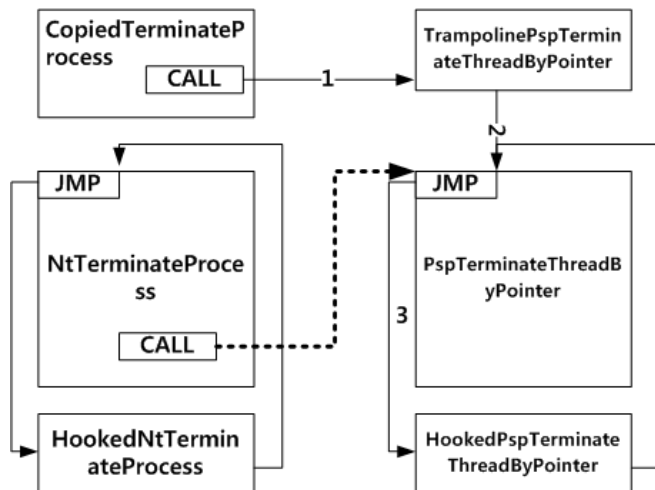


그림 7 전체 함수를 복사해서 다중 인라인 후킹을 우회

물론 복잡하게 트램펄린을 만들지 않고 JMP 부분만 원본 코드로 덮어 쓴다면 쉽게 후킹을 우회할 수 있다. 하지만 이렇게 값을 덮어 쓰는 방법은 SDT 후킹과 마찬가지로 경쟁 조건을 유발시킨다. 서로 JMP와 원본 코드를 끊임없이 덮어쓴다면 그 결과는 예측할 수 없다. 또한 일부 보안 프로그램에서는 자신의 후킹 코드가 손상되는 경우에는 시스템을 중단시키기도 하기 때문에 값을 덮어쓰는 방법은 신중하게 결정해야 한다.

## 후킹을 넘어서

점프 코드 지뢰밭을 일일이 제거하면서 나가는 것도 재미는 있지만 속도는 더디게 마련이다. 지뢰밭을 굳이 제거하지 않고 그 위를 날아간다면 그 또한 좋은 방법일 것이다. 앞서 우리가 원본 SDT를 커널 파일에서 찾은 것처럼 커널 파일에 있는 원본 코드를 사용한다면 지뢰밭을 뚫지 않고 날아가는 것도 가능하다.

<그림 8>에는 이러한 아이디어가 나와있다. 커널 파일 자체도 PE 포맷이라는 점과 재배치 섹션이 존재한다는 점을 생각한다면 커널을 로딩하는 것이 불가능한 일은 아니라는 것을 알 수 있다. 커널 모드 코드에서 메모리를 할당하고 그곳에 커널을 올린 다음 재배치를 수행하면 <그림 8>의 왼쪽과 같은 구조가 된다. 커널 코드가 두 벌이라는 것과 함께 각각의 커널은 자신의 자료 구조를 가질 것이다. 하지만 이렇게 되서는 곤란하다. 왜냐하면 우리는 자료 구조는 전부 기존 커널의 것을 이용해야 하기 때문이다. 약간의 트릭을 써서 커널을 할당된 메모리에 올린 다음 재배치를 기존 커널의 주소에 로딩된 것처럼 수행한다면 오른쪽 그림과 같은 구조가 된다. 커널의 코드는 두 벌이 되지만 참조하는 자료 구조는 하나가 된다. 이 상황에서 새로 로딩한 커널의 NtTerminateProcess를 호출한다면 트램펄린 함수를 하나도 만들지 않고 모든 인라인 후킹을 우회할 수 있다.

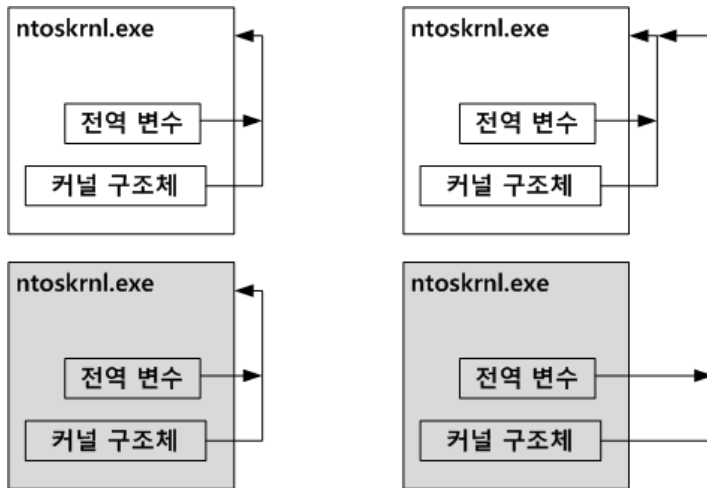


그림 8 커널 로더

## 극과 극은 통한다

다소 제한된 지면에 많은 내용을 소개하려다 보니 설명이 조금은 부실한 느낌이 든다. 전체를 읽은 여러분의 느낌이 어떤지 궁금하다. 이 글을 쓰면서 필자가 느낀 한 가지 생각은 극과 극은 통한다는 것이었다. 결국 보안 제품의 기술이나 루트킷 기술이나 비슷하다는 말이다. 앞서 살펴본 것처럼 루트킷에서 사용된 기술이 보안 제품에 차용되기도 하고, 보안 제품에서 개발된 기술이 루트킷에서 이용되기도 하는 것이다. 여러분에게 주어진 칼이 좋은 일에 쓰이길 기도해 본다.

## 참고자료

Win2K/XP SDT Restore 0.2 (Proof-Of-Concept)

<http://www.security.org.sg/code/sdtrestore.html>

A more stable way to locate real KiServiceTable

<http://www.rootkit.com/newsread.php?newsid=176>

ksthb v1.0

[http://zap.pe.kr/index.php?page=pages/researches/winternals\\_kr.php](http://zap.pe.kr/index.php?page=pages/researches/winternals_kr.php)