[문서번역]

Stealth Hooking
: Another Way to subvert the Windows Kernel

by mxatone and ivanlef0u

==========

==Phrack Inc.==
Volume 0x0c, Issue 0x41, Phile #0x04 of 0x0f

번역자 : **이동수**
alonglog@gmail.com

Content

1. Introduction on anti-rootkit technologies and bypass

현재 루트킷과 안티-루트킷은 IT 보안 업계에서 더욱 더 중요해지고 있다. 좋든 싫든 루트킷은 숨겨지고, 작고, 하드웨어에 가까우며 독창적이고 나쁜 백도어의 성배로 생각되어진다. 백도어는 컴퓨터를 원격이나 로컬로 통제하여 공격자에게 최선의 선택을 하도록 만들어 준다.

안티-루트킷은 악성 프로그램을 발견하고 삭제하기 위해서 동작한다.

루트킷 기술과 복잡성은 빠르게 발달하고 있고, 오늘날 루트킷이나 안티-루트킷을 개발하는 것은 매우 어려운 일이다.

이 문서는 윈도우 기반에서 동작하는 루트킷을 다룬다. 더 구체적으로 말하면 윈도우 커널에서 동작하는 가로채기 기술의 새로운 종류에 대해서 다룬다. 독자들이 윈도우 루트킷 기술에 대해 알고 있다고 가정하겠다.

1.1 Rootkits and anti-rootkits technics

루트킷은 운영체제의 행동을 가로챈다. 이 일을 성공하기 위해서 간단히 운영체제의 바이너리 값을 수정할 수 있지만, 이 일은 매우 조심스럽지 못하다. 대부분의 루트킷은 중요한 함수에 훅을 사용하고 결과를 바꾼다. 기본적인 후킹은 함수의 시작부분이나 함수 포인터를 바꿈으로써 실행 흐름을 재설정한다. 하지만 이것이 루틴을 가로챌 수 있는 유일한 방법은 아니다. 가장 대표적인 예가 syscall 함수 시작 주소를 가지고 있는 SSDT(System Service Descriptor)[1]이다. 당신이 이 테이블의 포인터를 수정할 수 있다면 당신은 한 가지 함수의 행동을 컨트롤 할 수 있을 것이다. 이것은 루트킷의 과정을 보여주는 한 가지 일뿐이고, 분명히 공격자에 의해 통제될 수 있는 많은 위험한 영역이 존재한다.

안티-루트킷은 이 영역을 검사하지만 이 행동은 매우 어렵다. 대부분이 안티-루트킷 프로그램은 디스크 상의 바이너리와 프로그램의 메모리 이미지를 비교하거나 어떤 것이 변화하였는지 확인할 수 있는 함수 포인터 테이블을 검증한다.

루트킷을 만드는 사람들과 안티-루트킷을 만드는 사람들 간의 전쟁양상은 취약한 운영체제의 특징을 후킹하기 위한 최적의 장소와 방법을 찾는 것이다. 아래 설명된 영역은 루트킷에 의해 종종 사용되는 윈도우즈 영역이다.

- SSDT(Kernel syscalls table) 와 shadow SSDT(win32k syscall table)은 가장 간단한 해결책이다.

- MSR (Model Specific Registers)은 루트킷에 의해 수정될 수 있다. 윈도우즈에서 MSR_SYSENTER_EIP는 ring0 모드로 들어가기 위한 어셈블리 명령어 'sysenter'에 의

1) SSDT는 native API의 주소를 가지고 있는 테이블이다.

해 사용된다. MSR을 가로채는 것은 공격자가 시스템을 통제할 수 있도록 한다.

- MajorFunction은 다른 디바이스와 I/O를 처리하기 위한 드라이버에 의해 사용되는 함수이다. 이 함수를 후킹하는 것은 루트킷에게 유용하다.

- IDT(Interrupt Descriptor Table)는 예외와 인터럽트를 처리하기 위해 시스템이 사용하는 테이블이다.

다른 기술도 나왔다. 커널 오브젝트에 접근함으로써 루트킷은 프로세스, 스레드, 로드된 모듈, 다른 자원의 정보를 쉽게 바꿀 수 있다. 그 기술은 DKOM (Direct Kernel Object Manipulation)이라고 불린다. 예를 들어, 윈도우즈 커널은 실행중인 프로세스 정보를 유지하기 위해 (프로세스 구조체인) PsActiveProcessList라고 불리는 더블 링크드 리스트를 사용한다. 리스트 중 링크가 끊어지면 당신의 프로세스는 작업관리자같은 프로세스 리스트에 나타나지 않을 것이다. 하지만, 그 프로세스는 여전히 실행중이다.

커널 오브젝트 수정을 막기 위해서 안티-루트킷은 다른 영역을 조사한다. 프로세스 경우, PID(Process Identifier)와 TID(Thread Identifier)의 테이블을 가지는 PspCidTable을 읽는다.
PspCidTable과 PsActiveProcessList를 비교하면 숨겨진 프로세스를 찾을 수 있다. 이런 공격에 대비하여 안티-루트킷은 변경된 오브젝트를 찾기 위한 방법과 비교해야 될 영역을 알고 있다.

윈도우즈 Stealth에 관한 최초의 문서 중 하나는 Holy Father님이 쓰신 “Invisibility on NT boxes"[1]이다. 이 문서에서는 유명한 VXing mag 29A[3]의 Holy Father님과 Ratter님이 작성하신 Ring0 드라이버 루트킷인 Hacker Defender[2]의 일반적인 실행 코드가 소개 되었다. 이 드라이버는 토큰을 조작하여 프로세스 권한을 올릴 수 있다. 다른 루트킷은 파일과 레지스트리를 숨기거나 dll 인젝션을 프로세스에 감염시키기 위한 유저 기반 훅들을 사용한다. 완전한 ring0 루트킷의 좋은 예는 Greg Hoglund님의 NT Rootkit[4]이 있다. 이 드라이버는 몰래 동작하기 위해서 SSDT 훅을 사용한다. 이 드라이버는 IRP(I/O Request Packets)을 필터링하기 위해 NTFS 파일시스템과 Keyboard 디바이스 위에 기재한다. 또한 네트워크와 통신하기 위한 NDIS 프로토콜 디바이스를 제공한다. 이 루트킷이 NT 4.0과 Win2k에서 작동된다면 초보자를 위한 좋은 예가 될 것이다.

후에 Fuzen_op님이 작성하신 FU[5]와 유명한 기술 잡지 Uniformed[6]에서 발표한 개선된 FUto같은 진보된 ring0 루트킷이 나타났다. 드라이버를 검증하는 비스타의 개선점은 대부분 하드웨어 특징에 기반한 새로운 루트킷을 만들게 하였다. Eeye님이 작성하신 BootRoot[7]와 Pixie[8]는 어떤 보호모드가 작동하기 전에 로드된다. 마침내 Joanna Rutkowska님은 그녀의 Blue Pill 프로젝트에서 운영체제와 하드웨어 사이에 계층을 생성하는 가상기술을 사용하였다.

예전의 루트킷은 대부분 lame mail spamming이나 botnet을 위해 사용되었다. 루트킷은 종종 오래된 기술을 사용하였지만 Rustock[10] 시리즈나 StormWorm[11] 그리고 MBR 루트킷[12]은 흥미롭다. 이 루트킷들은 데이터 흐름 바꾸기(ADS), 코드 판독을 힘들게하기, 안티-디버그, 안티-VM[2], 다형의 코드 같은 수많은 트릭을 사용한다. 이런 트릭의 목적은 커널을 부수는 것뿐이 아니라 분석을 느리게 하는 것이다. 그리고 실행을 중지시키는 것을 어렵게 만들었다.

비록 루트킷에서 사용된 기술들이 점점 세련되고 있지만, 언더그라운드 커뮤니티는 현재 기술을 개선하기위한 여전히 활발한 POC를 하고 있다. Unreal[13]과 AK992[14]는 좋은 예이다. 처음에 루트킷을 숨기기위해 ADS(데이터 흐름 변경하기)와 NTFS MajorFunctions 후킹을 사용했고, 두 번째는 데이터가 디스크 드라이버에 보내질 때 IRP 도달점을 검사한다. rootkit.com에서 루트킷 기술의 충분한 예제를 볼 수 있다.

마지막으로, 안티-루트킷에 대해 언급하지 않을 수 없다. 가장 유명한 안티-루트킷은 UG North 팀에 있는 MP_ART님과 EP_XOFF님의 Rk Unhooker이다. 다른 안티-루트킷으로는 CardMagic님dml KarkSpy[15]와 pjf님과 Gmer님의 IceSword[16]가 있다.

1.2 - About kernel level protections

우리는 보호에 대한 이야기를 할 때 시스템에서 보호가 일어나는 장소에 주목해야 한다. 보호가 하이 레벨에서 동작한다면 그 레벨에 한해서 유효하다. PaX와 Exec Shield 같은 보호는 커널로부터 유저모드 기반의 프로그램들을 보호하기 때문에 효율적이다.

PatchGuard같은 보호 프로그램과 HIPS 또한 시스템을 보호하지만 공격자가 자신의 레벨에서 작동하는 보호를 공격할 수 있는 방법을 찾는 한 이런 보호는 쓸모가 없다. 보호는 공격자에 의해 변조되지만 않았다면 신뢰할만하다. 뛰어난 공격자가 보호 내부에 코드를 삽입할 수 있는 기술을 찾았다면 당신의 컴퓨터는 죽었다고 생각해라.

그것이 PatchGuard[18]가 효율적이지 못한 이유이다. 하지만 우리는 무능하거나 파괴당하는 보호 프로그램은 매우 떠들썩하다는 것을 알고 있다. 가장 좋은 방법은 휘발성 때문에 검사할 수 없는 특별한 오브젝트와 이벤트에서 작동하는 탐지기 아래에서 작동시키는 것이다.

2006년 6월에 Greg Hoglund님이 KOH(Kernel Object Hooking)[19]을 발표하였다. 새로운 우회코드 실행 방법은 정적인 코드 섹션을 바꿀 수 없지만 동적으로 할당된 DPC(Deferred Procedure Calls)같은 구조체들과 데이터들을 바꿀 수는 있다. 보호 프로그램의 불안정성 때문에 그 영역을 검증하고 찾는 것은 어렵다.

다른 좋은 오브젝트는 IRP(I/O Request Paket)[3]이다. 이 오브젝트는 윈도우 커널 I/O

2) 안티-VM은 가상머신에서 실행되는 것을 막는 기술이다.
3) I/O 장치에서 발생하는 정보를 담고 있는 구조체이다.

매니저가 디바이스와 통신을 할 때 사용되는 오브젝트이다. 하드웨어에서 각 I/O 동작은 IRP를 발생시키고 Syscall들은 그의 디바이스를 통해 드라이버에게 IRP를 보낸다. 일반적으로 드라이버는 몇몇 디바이스를 가지고 있다. 그들 중에는 IOCTL을 사용하여 유저 모드의 프로그램을 성명하는데 사용하는 디바이스도 있다. 다른 디바이스는 필터링을 통해 IRP를 관리하거나 요청된 일을 수행한다.

IRP는 MajorFunctions 테이블을 사용하는 드라이버에게 보내진다. 이 테이블은 드라이버가 제공하는 다른 상관성들을 포함하고 있다. 당신은 MajorFunction에 의해 리턴되는 값을 IRP에 설치함으로써 검사할 수 있다. IRP 휘발성 오브젝트이다. 이 오브젝트를 검사하거나 통제하기는 매우 어렵다.

사실 당신이 모든 것을 검사하기를 원한다면 OS의 구조를 완전히 재설계를 해야 한다. 언제나 모든 장소를 보호할 수 없다는 것을 명심하라. 이것에 대해서는 다음 파트에서 설명할 것이다.

1.3 - Concept key: use kernel code against itself

이 생각의 기본은 커널코드를 이용하는 것이다. 이 방법은 입력이 코드의 행동을 정의하기 때문에 가능하다. 취약한 소프트웨어에서 정교한 입력을 제시한다는 것을 코드가 실행 될 수 있다는 것을 의미한다. 물론 위험한 입력은 당신의 목표에 의해서 결정된다. 커널 공간은 당신이 환경을 바꿀 수 있기 때문에 여분의 시나리오를 가지고 있다. 루트킷이 인자로 들어오는 기본 입력을 바꿀 수는 없다. 하지만 코드가 작동하는 환경을 바꿀 수는 있다. 언링킹[4]을 사용하는 HEAP 이용 기술이 좋은 예제이다. 메모리 블록 구조체를 바꾸기 위해서 당신은 4바이트를 덮어써야 하다. 어떤 기술들은 다음에 할당된 블록 주소를 바꿀 수 있다. 이런 결과는 프로그램이 그 정보를 신뢰하기 때문에 발생한다. 커널에서 당신은 환경위에서 전체적인 통제를 한다. 또한 커널을 완전하게 검사하는 것은 성능을 저하시키고 전체적으로 불가능하다.

코드 환경을 바꾸는 것은 phide2 루트킷[21]기술에서 성공적으로 사용되었다. 이 루트킷은 윈도우즈 스케줄러를 후킹하지 않고 쓰레드를 숨길 수 있는 것이 인상적이다. 코드에 의해 작동하는 루틴을 신뢰하도록 만들기 위해서는 풍부한 리버스 지식이 필요하다. 알려지지 않은 운영체제 행동을 통해 이 개념을 확대해야 한다. 일반적인 보호 프로그램은 일반적인 가정에 기반을 둔다. 코드 훅을 검사하기위해 오직 드라이버 이미지만 검사하는 것처럼 말이다. 오늘날 운영체제 설계는 루트킷에 대항하는 보호 프로그램과 진보된 소프트웨어 루트킷 기술들을 요구한다.

2. Introducing stealth hooking on IDT

IDT에 기반을 두는 예제를 가지고 stealth hooking에 관한 개념을 소개하겠다. 우선 IDT가 무엇이며 목적이 무엇인지 볼 것이다. 그 후 우리는 하드웨어 인터럽트와 윈도우

4) 다음 정보를 가리키는 부분을 삭제하여 링크를 끊어버리는 행위이다.

즈가 그들을 어떻게 다루는지에 대해 논의할 것이다.

IDT(Interrupt Descriptor Table)은 커널에 위치한 구체적인 선형 테이블이다. IDT는 ring3 권한 레벨에서 읽을 수 있지만 IDT에 쓰고자 한다면 ring0의 권한을 가지고 있어야 한다. IDT는 256개의 KIDTENTRY 구조체로 구성되어있고 IDT의 정의를 보기 위해 윈도우즈의 디버깅 툴에 있는 Kernel Debugger(KD)[22]를 이용할 수 있다.

```
kd> dt nt!_KIDTENTRY
   +0x000 Offset           : Uint2B
   +0x002 Selector         : Uint2B
   +0x004 Access           : Uint2B
   +0x006 ExtendedOffset   : Uint2B
```

이 문서에서 우리는 IDT의 구조에 대해 설명하고 싶지 않다. 그래서 당신에게 IDT가 무엇이고 어떻게 작동하는지 알고 싶다면 Phrack 59호에 실린 Kad님의 글[23]을 보라고 권고한다.

IDT의 처음 32개의 엔트리는 CPU가 예외를 처리하기 위해 예약되어져 있다. 다른 엔트리는 하드웨어 인터럽트와 특별한 시스템 이벤트를 처리하기 위해 사용된다.

윈도우즈 IDT의 처음 64개의 엔트리를 덤프하였다.

```
kd> !idt -a

Dumping IDT:

00:      804e1bff nt!KiTrap00
01:      804e1d7c nt!KiTrap01
02:      Task Selector = 0x0058
03:      804e215b nt!KiTrap03
04:      804e22e0 nt!KiTrap04
05:      804e2441 nt!KiTrap05
06:      804e25bf nt!KiTrap06
07:      804e2c33 nt!KiTrap07
08:      Task Selector = 0x0050
09:      804e3060 nt!KiTrap09
0a:      804e3185 nt!KiTrap0A
0b:      804e32ca nt!KiTrap0B
0c:      804e3530 nt!KiTrap0C
0d:      804e3827 nt!KiTrap0D
```

```
0e:     804e3f25 nt!KiTrap0E
0f:     804e425a nt!KiTrap0F
10:     804e437f nt!KiTrap10
11:     804e44bd nt!KiTrap11
12:     Task Selector = 0x00A0
13:     804e462b nt!KiTrap13
14:     804e425a nt!KiTrap0F
15:     804e425a nt!KiTrap0F
16:     804e425a nt!KiTrap0F
17:     804e425a nt!KiTrap0F
18:     804e425a nt!KiTrap0F
19:     804e425a nt!KiTrap0F
1a:     804e425a nt!KiTrap0F
1b:     804e425a nt!KiTrap0F
1c:     804e425a nt!KiTrap0F
1d:     804e425a nt!KiTrap0F
1e:     804e425a nt!KiTrap0F
1f:     806effd0 hal!HalpApicSpuriousService
20:     00000000
21:     00000000
22:     00000000
23:     00000000
24:     00000000
25:     00000000
26:     00000000
27:     00000000
28:     00000000
29:     00000000
2a:     804e1417 nt!KiGetTickCount
2b:     804e1522 nt!KiCallbackReturn
2c:     804e16c7 nt!KiSetLowWaitHighThread
2d:     804e2032 nt!KiDebugService
2e:     804e0ea6 nt!KiSystemService
2f:     804e425a nt!KiTrap0F
30:     804e0560 nt!KiStartUnexpectedRange
31:     804e056a nt!KiUnexpectedInterrupt1
32:     804e0574 nt!KiUnexpectedInterrupt2
33:     804e057e nt!KiUnexpectedInterrupt3
34:     804e0588 nt!KiUnexpectedInterrupt4
35:     804e0592 nt!KiUnexpectedInterrupt5
```

```
36:      804e059c nt!KiUnexpectedInterrupt6
37:      806ef728 hal!PicSpuriousService37
38:      804e05b0 nt!KiUnexpectedInterrupt8
39:      804e05ba nt!KiUnexpectedInterrupt9
3a:      804e05c4 nt!KiUnexpectedInterrupt10
3b:      804e05ce nt!KiUnexpectedInterrupt11
3c:      804e05d8 nt!KiUnexpectedInterrupt12
3d:      806f0b70 hal!HalpApcInterrupt
3e:      804e05ec nt!KiUnexpectedInterrupt14
3f:      804e05f6 nt!KiUnexpectedInterrupt15
40:      804e0600 nt!KiUnexpectedInterrupt16
41:      806f09cc hal!HalpDispatchInterrupt
42:      804e0614 nt!KiUnexpectedInterrupt18
43:      804e061e nt!KiUnexpectedInterrupt19
44:      804e0628 nt!KiUnexpectedInterrupt20
45:      804e0632 nt!KiUnexpectedInterrupt21
46:      804e063c nt!KiUnexpectedInterrupt22
47:      804e0646 nt!KiUnexpectedInterrupt23
48:      804e0650 nt!KiUnexpectedInterrupt24
49:      804e065a nt!KiUnexpectedInterrupt25
4a:      804e0664 nt!KiUnexpectedInterrupt26
4b:      804e066e nt!KiUnexpectedInterrupt27
4c:      804e0678 nt!KiUnexpectedInterrupt28
4d:      804e0682 nt!KiUnexpectedInterrupt29
4e:      804e068c nt!KiUnexpectedInterrupt30
4f:      804e0696 nt!KiUnexpectedInterrupt31
50:      806ef800 hal!HalpApicRebootService
51:      804e06aa nt!KiUnexpectedInterrupt33
52:      804e06b4 nt!KiUnexpectedInterrupt34
53:      804e06be nt!KiUnexpectedInterrupt35
54:      804e06c8 nt!KiUnexpectedInterrupt36
55:      804e06d2 nt!KiUnexpectedInterrupt37
56:      804e06dc nt!KiUnexpectedInterrupt38
57:      804e06e6 nt!KiUnexpectedInterrupt39
58:      804e06f0 nt!KiUnexpectedInterrupt40
59:      804e06fa nt!KiUnexpectedInterrupt41
5a:      804e0704 nt!KiUnexpectedInterrupt42
5b:      804e070e nt!KiUnexpectedInterrupt43
5c:      804e0718 nt!KiUnexpectedInterrupt44
5d:      804e0722 nt!KiUnexpectedInterrupt45
```

```
5e:      804e072c nt!KiUnexpectedInterrupt46
5f:      804e0736 nt!KiUnexpectedInterrupt47
60:      804e0740 nt!KiUnexpectedInterrupt48
61:      804e074a nt!KiUnexpectedInterrupt49
62:      81f77dd4 atapi!IdePortInterrupt (KINTERRUPT 81f77d98)
63:      804e075e nt!KiUnexpectedInterrupt51
64:      804e0768 nt!KiUnexpectedInterrupt52
65:      804e0772 nt!KiUnexpectedInterrupt53
66:      804e077c nt!KiUnexpectedInterrupt54
67:      804e0786 nt!KiUnexpectedInterrupt55
68:      804e0790 nt!KiUnexpectedInterrupt56
69:      804e079a nt!KiUnexpectedInterrupt57
6a:      804e07a4 nt!KiUnexpectedInterrupt58
6b:      804e07ae nt!KiUnexpectedInterrupt59
6c:      804e07b8 nt!KiUnexpectedInterrupt60
6d:      804e07c2 nt!KiUnexpectedInterrupt61
6e:      804e07cc nt!KiUnexpectedInterrupt62
6f:      804e07d6 nt!KiUnexpectedInterrupt63
70:      804e07e0 nt!KiUnexpectedInterrupt64
71:      804e07ea nt!KiUnexpectedInterrupt65
72:      804e07f4 nt!KiUnexpectedInterrupt66
73:      81d63044 NDIS!ndisMIsr (KINTERRUPT 81d63008)
74:      804e0808 nt!KiUnexpectedInterrupt68
75:      804e0812 nt!KiUnexpectedInterrupt69
76:      804e081c nt!KiUnexpectedInterrupt70
77:      804e0826 nt!KiUnexpectedInterrupt71
78:      804e0830 nt!KiUnexpectedInterrupt72
79:      804e083a nt!KiUnexpectedInterrupt73
7a:      804e0844 nt!KiUnexpectedInterrupt74
7b:      804e084e nt!KiUnexpectedInterrupt75
7c:      804e0858 nt!KiUnexpectedInterrupt76
7d:      804e0862 nt!KiUnexpectedInterrupt77
7e:      804e086c nt!KiUnexpectedInterrupt78
7f:      804e0876 nt!KiUnexpectedInterrupt79
80:      804e0880 nt!KiUnexpectedInterrupt80
81:      804e088a nt!KiUnexpectedInterrupt81
82:      81f7c9fc atapi!IdePortInterrupt (KINTERRUPT 81f7c9c0)
83:      81e889d4 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
(KINTERRUPT 81e88998)
                  USBPORT!USBPORT_InterruptService (KINTERRUPT 81d8f008)
```

```
84:     804e08a8 nt!KiUnexpectedInterrupt84
85:     804e08b2 nt!KiUnexpectedInterrupt85
86:     804e08bc nt!KiUnexpectedInterrupt86
87:     804e08c6 nt!KiUnexpectedInterrupt87
88:     804e08d0 nt!KiUnexpectedInterrupt88
89:     804e08da nt!KiUnexpectedInterrupt89
8a:     804e08e4 nt!KiUnexpectedInterrupt90
8b:     804e08ee nt!KiUnexpectedInterrupt91
8c:     804e08f8 nt!KiUnexpectedInterrupt92
8d:     804e0902 nt!KiUnexpectedInterrupt93
8e:     804e090c nt!KiUnexpectedInterrupt94
8f:     804e0916 nt!KiUnexpectedInterrupt95
90:     804e0920 nt!KiUnexpectedInterrupt96
91:     804e092a nt!KiUnexpectedInterrupt97
92:     804e0934 nt!KiUnexpectedInterrupt98
93:     81e2b2f4        i8042prt!I8042KeyboardInterruptService        (KINTERRUPT
81e2b2b8)
94:     804e0948 nt!KiUnexpectedInterrupt100
95:     804e0952 nt!KiUnexpectedInterrupt101
96:     804e095c nt!KiUnexpectedInterrupt102
97:     804e0966 nt!KiUnexpectedInterrupt103
98:     804e0970 nt!KiUnexpectedInterrupt104
99:     804e097a nt!KiUnexpectedInterrupt105
9a:     804e0984 nt!KiUnexpectedInterrupt106
9b:     804e098e nt!KiUnexpectedInterrupt107
9c:     804e0998 nt!KiUnexpectedInterrupt108
9d:     804e09a2 nt!KiUnexpectedInterrupt109
9e:     804e09ac nt!KiUnexpectedInterrupt110
9f:     804e09b6 nt!KiUnexpectedInterrupt111
a0:     804e09c0 nt!KiUnexpectedInterrupt112
a1:     804e09ca nt!KiUnexpectedInterrupt113
a2:     804e09d4 nt!KiUnexpectedInterrupt114
a3:     81e77944 i8042prt!I8042MouseInterruptService (KINTERRUPT 81e77908)
a4:     804e09e8 nt!KiUnexpectedInterrupt116
a5:     804e09f2 nt!KiUnexpectedInterrupt117
a6:     804e09fc nt!KiUnexpectedInterrupt118
a7:     804e0a06 nt!KiUnexpectedInterrupt119
a8:     804e0a10 nt!KiUnexpectedInterrupt120
a9:     804e0a1a nt!KiUnexpectedInterrupt121
aa:     804e0a24 nt!KiUnexpectedInterrupt122
```

```
ab:      804e0a2e nt!KiUnexpectedInterrupt123
ac:      804e0a38 nt!KiUnexpectedInterrupt124
ad:      804e0a42 nt!KiUnexpectedInterrupt125
ae:      804e0a4c nt!KiUnexpectedInterrupt126
af:      804e0a56 nt!KiUnexpectedInterrupt127
b0:      804e0a60 nt!KiUnexpectedInterrupt128
b1:      81fe07e4 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 81fe07a8)
b2:      81d7e044 serial!SerialCIsrSw (KINTERRUPT 81d7e008)
b3:      804e0a7e nt!KiUnexpectedInterrupt131
b4:      804e0a88 nt!KiUnexpectedInterrupt132
b5:      804e0a92 nt!KiUnexpectedInterrupt133
b6:      804e0a9c nt!KiUnexpectedInterrupt134
b7:      804e0aa6 nt!KiUnexpectedInterrupt135
b8:      804e0ab0 nt!KiUnexpectedInterrupt136
b9:      804e0aba nt!KiUnexpectedInterrupt137
ba:      804e0ac4 nt!KiUnexpectedInterrupt138
bb:      804e0ace nt!KiUnexpectedInterrupt139
bc:      804e0ad8 nt!KiUnexpectedInterrupt140
bd:      804e0ae2 nt!KiUnexpectedInterrupt141
be:      804e0aec nt!KiUnexpectedInterrupt142
bf:      804e0af6 nt!KiUnexpectedInterrupt143
c0:      804e0b00 nt!KiUnexpectedInterrupt144
c1:      806ef984 hal!HalpBroadcastCallService
c2:      804e0b14 nt!KiUnexpectedInterrupt146
c3:      804e0b1e nt!KiUnexpectedInterrupt147
c4:      804e0b28 nt!KiUnexpectedInterrupt148
c5:      804e0b32 nt!KiUnexpectedInterrupt149
c6:      804e0b3c nt!KiUnexpectedInterrupt150
c7:      804e0b46 nt!KiUnexpectedInterrupt151
c8:      804e0b50 nt!KiUnexpectedInterrupt152
c9:      804e0b5a nt!KiUnexpectedInterrupt153
ca:      804e0b64 nt!KiUnexpectedInterrupt154
cb:      804e0b6e nt!KiUnexpectedInterrupt155
cc:      804e0b78 nt!KiUnexpectedInterrupt156
cd:      804e0b82 nt!KiUnexpectedInterrupt157
ce:      804e0b8c nt!KiUnexpectedInterrupt158
cf:      804e0b96 nt!KiUnexpectedInterrupt159
d0:      804e0ba0 nt!KiUnexpectedInterrupt160
d1:      806eed34 hal!HalpClockInterrupt
d2:      804e0bb4 nt!KiUnexpectedInterrupt162
```

```
d3:      804e0bbe nt!KiUnexpectedInterrupt163
d4:      804e0bc8 nt!KiUnexpectedInterrupt164
d5:      804e0bd2 nt!KiUnexpectedInterrupt165
d6:      804e0bdc nt!KiUnexpectedInterrupt166
d7:      804e0be6 nt!KiUnexpectedInterrupt167
d8:      804e0bf0 nt!KiUnexpectedInterrupt168
d9:      804e0bfa nt!KiUnexpectedInterrupt169
da:      804e0c04 nt!KiUnexpectedInterrupt170
db:      804e0c0e nt!KiUnexpectedInterrupt171
dc:      804e0c18 nt!KiUnexpectedInterrupt172
dd:      804e0c22 nt!KiUnexpectedInterrupt173
de:      804e0c2c nt!KiUnexpectedInterrupt174
df:      804e0c36 nt!KiUnexpectedInterrupt175
e0:      804e0c40 nt!KiUnexpectedInterrupt176
e1:      806eff0c hal!HalpIpiHandler
e2:      804e0c54 nt!KiUnexpectedInterrupt178
e3:      806efc70 hal!HalpLocalApicErrorService
e4:      804e0c68 nt!KiUnexpectedInterrupt180
e5:      804e0c72 nt!KiUnexpectedInterrupt181
e6:      804e0c7c nt!KiUnexpectedInterrupt182
e7:      804e0c86 nt!KiUnexpectedInterrupt183
e8:      804e0c90 nt!KiUnexpectedInterrupt184
e9:      804e0c9a nt!KiUnexpectedInterrupt185
ea:      804e0ca4 nt!KiUnexpectedInterrupt186
eb:      804e0cae nt!KiUnexpectedInterrupt187
ec:      804e0cb8 nt!KiUnexpectedInterrupt188
ed:      804e0cc2 nt!KiUnexpectedInterrupt189
ee:      804e0cc9 nt!KiUnexpectedInterrupt190
ef:      804e0cd0 nt!KiUnexpectedInterrupt191
f0:      804e0cd7 nt!KiUnexpectedInterrupt192
f1:      804e0cde nt!KiUnexpectedInterrupt193
f2:      804e0ce5 nt!KiUnexpectedInterrupt194
f3:      804e0cec nt!KiUnexpectedInterrupt195
f4:      804e0cf3 nt!KiUnexpectedInterrupt196
f5:      804e0cfa nt!KiUnexpectedInterrupt197
f6:      804e0d01 nt!KiUnexpectedInterrupt198
f7:      804e0d08 nt!KiUnexpectedInterrupt199
f8:      804e0d0f nt!KiUnexpectedInterrupt200
f9:      804e0d16 nt!KiUnexpectedInterrupt201
fa:      804e0d1d nt!KiUnexpectedInterrupt202
```

```
fb:      804e0d24 nt!KiUnexpectedInterrupt203
fc:      804e0d2b nt!KiUnexpectedInterrupt204
fd:      806f0464 hal!HalpProfileInterrupt
```

이 덤프는 윈도우즈 IDT의 형태를 보여준다. 당신은 IDT 엔트리 번호 뒤에 따라오는 핸들러의 주소와 이름을 볼 수 있다. 처음 32개의 엔트리는 예외를 처리하는 KiTrap*로 시작하는 함수들로 채워져 있다. KiSystemService와 KiCallbackReturn과 같은 특별한 시스템 인터럽트와 I8042KeyboardInterruptService 또는 I8042MouseInterruptService 같은 핸들러들이 사용되도록 나머지는 시스템에게 주어져 있다.

2.1 - How Windows manage hardware interrupts

인터럽트에 대해 이야기하기 전에 IRQL(Interrupt ReQuest Level)의 개념을 소개해야 할 것같다. 커널은 높은 숫자일수록 높은 우선권을 갖는 X86의 0부터 32까지 전체적으로 IRQL을 보여주고 있다. 커널이 소프트웨어 인터럽트를 위해 IRQLs를 표준으로 지정하고 있지만 HAL(Hardware Abstraction Layer)이 IRQLs에 하드웨어 인터럽트 번호를 지정한다.

```
   +----------------+
31 |    Highests    | \
to |    IRQLs       | | Clock, system failure.
27 |                | /
   +----------------+
26 |                | \
to |  DEVICE_IRQL   | | Hardware interrupts.
3  |                | /
   +----------------+
2  | DISPATCH_LEVEL | Scheduler, DPC.
   +----------------+
1  |    APC_LEVEL   | Used when dispatching APC.
   +----------------+
0  |  PASSIVE_LEVEL | Threads run at this IRQL.
   +----------------+
```

각 프로세서는 자신의 IRQL을 가지고 있다. 다른 것은 PASSIVE_LEVLE[5]에서 실행되는 반면에 당신은 IRQL=DISPATCH_LEVEL[6]에서 실행되는 프로그램을 가질 수 있다. 현재 레벨보다 높은 IRQL의 인터럽트는 프로세서를 인터럽트하는 반면, 현재레벨과 동등하거나 낮은 IRQLs의 인터럽트는 실행중인 스레드가 IRQL을 해제할 때까지 대기한다.

몇몇 시스템 요소는 IRQL>=DISPATCH_LEVEL에서 코드가 실행할 때 접근하지 못한다. 페이지된 메모리(디스크로 바뀔 수 있는 메모리)에 접근하는 것은 불가능하고 많은 커널 함수를 사용할 수 없다.

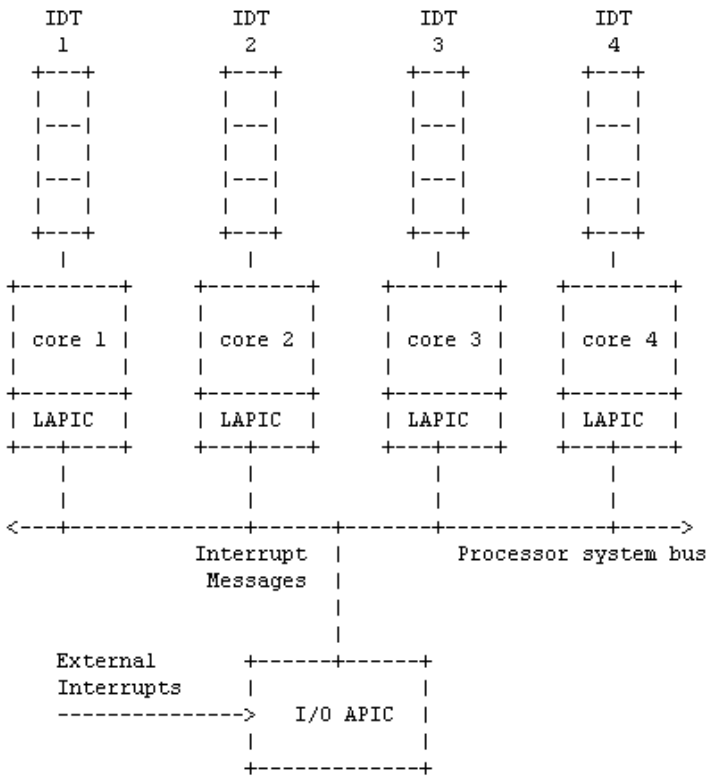
---

5) 가장 낮은 IRQL 권한
6) 모든 PASSIVE_LEVEL 프로그램에 언터럽트를 걸 수 있는 권한

하드웨어 인터럽트는 비동기적이며 외부의 주변장치에 의해 작동된다. 예를 들어 당신이 키를 눌렀을 때, 당신의 키보드 디바이스는 Southbridge[24]에 의해 Northbridge[25]를 통해서 인터럽트 컨트롤러로 IRQ를 보낸다. Southbridge는 I/O 컨트롤러 허브같은 칩이다. 이 칩은 모든 I/O 외부 인터럽트를 받고 Northbridge로 보낸다. Northbrige는 메모리와 고속 그래픽 버스, CPU에 직접 연결되어 있다. 이 칩은 또한 메모리 컨트롤러 허브로도 알려져 있다.

대부분이 x86 시스템은 i82489라고 불리는 Advanced Programmable Interrupt Controller(APIC) 칩셋을 사용한다. APIC는 CPU당 하나인 I/O APIC와 각 프로그램에 있는 LAPIC(Local APIC)로 구성되어 있다. I/O APIC는 가장 적합한 프로그램에서 인터럽트를 처리하기 위해 길 찾기 알고리즘을 사용한다. 지역성의 법칙에 따르면 이전의 I/O APIC는 디바이스 인터럽트를 처리하기 위해 프로그램에게 전달한다.[26]

이후에 LAPIC는 IRQ를 인터럽트 벡터인 8비트 값으로 번역한다. 이 인터럽트 벡터는 핸들러와 연관된 IDT의 엔트리 인덱스 번호이다. 프로그램이 인터럽트를 처리할 준비가 되었을 때, 명령어 흐름은 IDT 엔트리에 지정된 곳으로 재설정된다.



2.3.1 Hardward interrupts dispatching on Windows

윈도우즈에서 인터럽트 핸들러는 코드 템플릿 때문에 바로 실행되지 않는다. 이 템플릿은 KiInterruptTemplate 함수에서 실행되고 2가지 행위를 한다. 처음으로 현재 코어

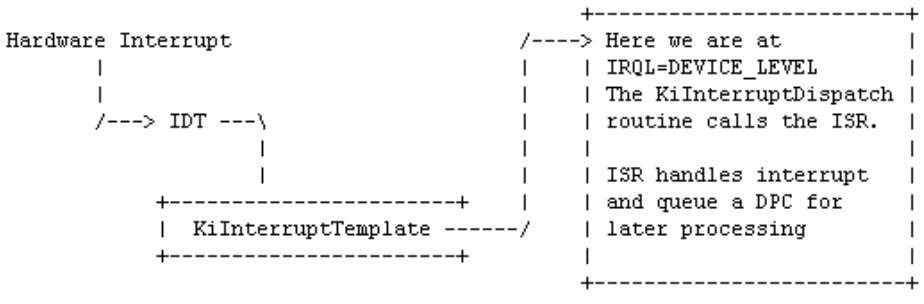
의 상태 정보를 스택에 저장하고 코드 흐름을 올바른 인터럽트 디스패처로 처리한다.

인터럽트가 발생하였을 때, 프로그램 상태가 저장된 후에 코드 흐름은 IDT에 정의된 인터럽트 핸들러로 넘어간다. 사실 IDT의 인터럽트 핸들러는 KinIterrupTemplate 경로를 가리킨다. KinIterruptTemplate는 아래 동작을 수행하는 KiInterruptDispatch를 호출할 것이다.

- 서비스 루틴의 스핀락을 얻는다.
- DEVICE_IRQL에서 IRQL이 발생하였을 때, 주워진 인터럽트 벡터의 IRQL은 27d에서 인터럽트 벡터를 빼서 계산된다.
- ISP(Interrupt Service Routine)같은 인터럽트 핸들러를 호출한다.
- IRQL을 낮춘다.
_ 서비스 루틴의 스핀락을 해제한다.

예를 들어 키보드 디바이스 ISR은 I8042KeyboardInterruptService이다. ISR은 리눅스 커널속의 top-halves같은 인터럽트를 처리하기 위한 루틴이다. WDK(Windows Driver Kit)에 따르면, ISR은 인터럽트를 처리하기위해서 디바이스에게 무엇이든지 적당한 행동을 해야 한다. 그 후, 오직 스테이지를 저장하기위해 필요한 무엇과 DPC에 넣어야 한다. 인터럽션관리는 ISR의 실행 동안보다는 낮은 IRQL에서 일어난다는 것을 의미한다. I/O 처리는 DPC에서 일어난다.

DPC(Deferred Procedure Call)은 리눅스에서 bottom-halves와 동등하다. DPC는 ISR의 IRQL보다 낮은 IRQL DISPATCH_LEVEL에서 작동한다. 사실 ISR은 너무많은 시간동안 작업하는 선점을 피하기 위해서 낮은 IRQL에 있는 모든 인터럽트를 프로세스에 있는 DPC 에 넣을 것이다. keyboard를 위한 DPC는 I8042KeyboardIsrDpc이다. 아래는 인터럽트 과정을 요약해서 보여준다.



KiInterruptDispatch는 KiInterrupTemplate으로부터 중요한 인자 하나를 받는다. 그 인자는 EDI 레지스터에 저장되어 있는 인터럽트 오브젝트의 포인터이다. 인터럽트 오브젝트는 KINTERRUPT 구조체에 정의 되어 있다.

```
kd> dt nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry : _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32       unsigned char
   +0x010 ServiceContext   : Ptr32 Void
   +0x014 SpinLock         : Uint4B
   +0x018 TickCount        : Uint4B
   +0x01c ActualLock       : Ptr32 Uint4B
   +0x020 DispatchAddress  : Ptr32       void
   +0x024 Vector           : Uint4B
   +0x028 Irql             : UChar
   +0x029 SynchronizeIrql  : UChar
   +0x02a FloatingSave     : UChar
   +0x02b Connected        : UChar
   +0x02c Number           : Char
   +0x02d ShareVector      : UChar
   +0x030 Mode             : _KINTERRUPT_MODE
   +0x034 ServiceCount     : Uint4B
   +0x038 DispatchCount    : Uint4B
   +0x03c DispatchCode     : [106] Uint4B
```

이 구조체 안에서 스핀락과 서비스루틴 정보를 찾을 수 있다. SynchronizeIrql은 ISR이 실행되었을 때 IRQL을 포함하고 있다.

하드웨어 인터럽트를 처리하기 위한 IDT의 각 엔트리인 KiInterruptTemplate은 KINTERRUPT 구조체의 DispatchCode 테이블 안에 포함되어 있다.

키보드 디바이스의 구조체를 살펴보자.

```
kd> dt nt!_KINTERRUPT 81dbcd98
   +0x000 Type             : 22
   +0x002 Size             : 484
   +0x004 InterruptListEntry : _LIST_ENTRY [ 0x81dbcd9c - 0x81dbcd9c ]
   +0x00c ServiceRoutine   : 0xf8630495       unsigned char
i8042prt!I8042KeyboardInterruptService+0
   +0x010 ServiceContext   : 0x81e63a58
   +0x014 SpinLock         : 0
   +0x018 TickCount        : 0xffffffff
   +0x01c ActualLock       : 0x81e63b18  -> 0
```

```
    +0x020 DispatchAddress   : 0x804dcd62       void   nt!KiInterruptDispatch+0
    +0x024 Vector            : 0x193
    +0x028 Irql              : 0x8 ''
    +0x029 SynchronizeIrql   : 0x9 ''
    +0x02a FloatingSave      : 0 ''
    +0x02b Connected         : 0x1 ''
    +0x02c Number            : 0 ''
    +0x02d ShareVector       : 0 ''
    +0x030 Mode              : 1 ( Latched )
    +0x034 ServiceCount      : 0
    +0x038 DispatchCount     : 0xffffffff
    +0x03c DispatchCode      : [106] 0x56535554
```

KiInterruptTemplate의 시작 부분을 살펴보자.

```
nt!KiInterruptTemplate:
804dcdfc 54              push    esp
804dcdfd 55              push    ebp
804dcdfe 53              push    ebx
804dcdff 56              push    esi
804dce00 57              push    edi
804dce01 83ec54          sub     esp,54h
804dce04 8bec            mov     ebp,esp
804dce06 89442444        mov     dword ptr [esp+44h],eax
kd> u
nt!KiInterruptTemplate+0xe:
804dce0a 894c2440        mov     dword ptr [esp+40h],ecx
804dce0e 8954243c        mov     dword ptr [esp+3Ch],edx
804dce12 f744247000000200 test   dword ptr [esp+70h],20000h
804dce1a 0f852a010000    jne     nt!V86_kit_a (804dcf4a)
804dce20 66837c246c08    cmp     word ptr [esp+6Ch],8
804dce26 7423            je      nt!KiInterruptTemplate+0x4f (804dce4b)
804dce28 8c642450        mov     word ptr [esp+50h],fs
804dce2c 8c5c2438        mov     word ptr [esp+38h],ds
kd>
nt!KiInterruptTemplate+0x34:
804dce30 8c442434        mov     word ptr [esp+34h],es
804dce34 8c6c2430        mov     word ptr [esp+30h],gs
804dce38 bb30000000      mov     ebx,30h
```

```
804dce3d b823000000        mov       eax,23h
804dce42 668ee3            mov       fs,bx
804dce45 668ed8            mov       ds,ax
804dce48 668ec0            mov       es,ax
804dce4b 648b1d00000000    mov       ebx,dword ptr fs:[0]
kd>
nt!KiInterruptTemplate+0x56:
804dce52 64c70500000000ffffffff mov dword ptr fs:[0],0FFFFFFFFh
804dce5d 895c244c          mov       dword ptr [esp+4Ch],ebx
804dce61 81fc00000100      cmp       esp,10000h
804dce67 0f82b5000000      jb        nt!Abios_kit_a (804dcf22)
804dce6d c744246400000000 mov        dword ptr [esp+64h],0
804dce75 fc                cld
804dce76 8b5d60            mov       ebx,dword ptr [ebp+60h]
804dce79 8b7d68            mov       edi,dword ptr [ebp+68h]
kd>
nt!KiInterruptTemplate+0x80:
804dce7c 89550c            mov       dword ptr [ebp+0Ch],edx
804dce7f c74508000ddbba    mov       dword ptr [ebp+8],0BADB0D00h
804dce86 895d00            mov       dword ptr [ebp],ebx
804dce89 897d04            mov       dword ptr [ebp+4],edi
804dce8c f60550f0dfffff    test      byte ptr ds:[0FFDFF050h],0FFh
804dce93 750d              jne       nt!Dr_kit_a (804dcea2)
nt!KiInterruptTemplate2ndDispatch:
804dce95 bf00000000        mov       edi,0
nt!KiInterruptTemplateObject:
804dce9a e9c3fcffff        jmp       nt!KeSynchronizeExecution+0x10 (804dcb62)
```

각 KINTERRUPT는 특별한 코드를 갖고 있다는 것을 기억하라. KiInterruptDispatch가 EDI 레지스터(인터럽트의 KINTERRUPT 포인터)로부터 인자를 하나 받는다고 전에 말했었다. KiInterruptTemplate에서 우리는 아래의 적은 코드를 볼 수 있다.

```
nt!KiInterruptTemplate2ndDispatch:
804dce95 bf00000000        mov       edi,0
nt!KiInterruptTemplateObject:
804dce9a e9c3fcffff        jmp       nt!KeSynchronizeExecution+0x10 (804dcb62)
```

여기에서는 mov 'edi, 0'을 하고 나서 점프를 하는 것만을 볼 수 있지만 키보드의 KINTERRUPT 속에 포함된 KiInterruptTemplate 코드를 살펴보면

```
ffb72525 bf5024b7ff      mov     edi,0FFB72450h ; Keyboard KINTERRUPT
ffb7252a e9a9839680      jmp     nt!KiInterruptDispatch (804da8d8)
```

코드가 바뀌었다. 커널은 동적으로 KiInterruptTemplate code안의 2가지 명령어를 바꾼다. EDI에서 KINTERRUPT 오브젝트를 찾고 KiInterruptDispatch 분기로 점프한다.

왜 이렇게 실행할까? 그 처리 핸들러를 쉽게 바꿀 수 있기 때문이다. KiInterruptDispatch를 가지고 있지만 종종 KiFloatingDispatch 또는 KiChainDispatch를 볼 수 있다. KiChaineDispatch는 다중 인터럽트 오브젝트 사이에 벡터를 공유하기위해 존재하고, KiFloatingDispatch는 KiInterruptDispatch와 같지만 떠다니는 코어 상태 정보를 저장한다.

윈도우즈는 IDT의 인터럽트에 접근할 수 있는 API를 제공한다. IoConnectInterrupt와 IoConnectInterruptEx이다. WDK에 따른 이 함수의 구조를 살펴보자.

```
NTSTATUS
  IoConnectInterrupt(
    OUT PKINTERRUPT  *InterruptObject,
    IN PKSERVICE_ROUTINE  ServiceRoutine,
    IN PVOID  ServiceContext,
    IN PKSPIN_LOCK  SpinLock  OPTIONAL,
    IN ULONG  Vector,
    IN KIRQL  Irql,
    IN KIRQL  SynchronizeIrql,
    IN KINTERRUPT_MODE     InterruptMode,
    IN BOOLEAN  ShareVector,
    IN KAFFINITY  ProcessorEnableMask,
    IN BOOLEAN  FloatingSave
    );
```

IoConnectInterrupt가 KINTERRUPT 구조체인 InterruptObject 인자에 리턴값을 저장하는 것을 볼 수 있듯이 IDT에서 검색하는 것과 동일하다. 이전에 당신은 KiInterruptTemplate에서 2가지 레이블을 보았다. KiInterruptTemplateObject와 KiInterruptTemplate2ndDispatch이다. 두 레이블은 KiInterruptTemplateRoutine에서 두 가지 명령어를 찾기 위해서 커널 함수에 의해 사용된다. KeInitializeInterrupt는 “jmp Ki*Dispatch"를 업데이트 하기위해 KiInterruptTemplateObject 레이블을 사용한다. KiConnectVectorAndInterruptObject 함수는 ”mov edi, <&Kinterrupt>"를 수정하기 위해 KiInterruptTemplate2ndDispatch를 사용한다.

2.3.2 ninja 같은 하드웨어 IDT 후킹

생각해보자. 우리는 아무도 모르게 IDT를 훅하고 싶다, 엔트리를 직접적으로 바꾸는 것을 좋은 방법이 아니다. 안티-루트킷은 동적으로 할당된 KiInterruptTemplate 루틴을 체크하지 않는다. 그래서 우리는 원하는 곳으로 이 루틴을 수정할 수 있다. 아래는 가능한 3가지 방법이다.

- dispatch 루틴에서 "jmp Ki*Dispatch"를 우리가 원하는 Dispatch 루틴으로 어렵지 않게 바꿀 수 있다.

- “mov edi, <&Kinterrupt>" 명령에 의해 사용되는 EDI의 kinterrupt의 주소를 바꾸는 것이다. 새로운 KINTERRUPT는 이전과 동일하지만 서비스루틴은 우리에 의해 바뀌어 질 것이다.

- 우리의 KiInterrupt를 만드는 방법이 있지만 어렵다.

이 문서에서는 가장 쉬운 방법을 선택하였다. 우리는 “mov edi, <&Kinterrupt>"를 ”mov edi, <&OurKinterrupt>"로 바꾼다. 이 명령어는 jmp뒤에 따라온다. 그래서 어셈블리 엔진으로 "jmp nt!KiInterruptDispatch"전에 명령어를 검색할 수 있고 고칠 수 있다. 서비스 루틴이 실행될 때 인터럽트는 아직 처리되지 않는 다는 것을 명심해야한다. 그리고, DEVICE_IRQL IRQL에서 실행한다. 이는 많은 커널 함수에 접근할 수 없기 때문에 공평한 상황이 아니다. 모든 ISR은 DPC에 들어가고, ISR가 실행된 후, 현재 코어의 DPC 큐의 마지막 엔트리는 인터럽트 DPC를 포함하고 있다.

인터럽트에 의해 발생한 데이터에 접근하고 싶다면 ISR같이 처리해야한다. 하드웨어 디바이스에 너무 의존적이기때문에 우리의 ISR로 실제 ISR을 대체하기는 매우 어렵다. 실제 I/O가 DPC에 의해 실행되어 KiInterruptTemplate이 우리의 서비스 루틴을 호출할 때, 우리는 실제 서비스 루틴을 호출하고 마지막 DPC 엔트리를 수정한다.

DPC는 KDPC 구조체에 나타난다.

```
kd> dt nt!_KDPC
   +0x000 Type             : Int2B
   +0x002 Number           : UChar
   +0x003 Importance       : UChar
   +0x004 DpcListEntry     : _LIST_ENTRY
   +0x00c DeferredRoutine  : Ptr32       void
   +0x010 DeferredContext  : Ptr32 Void
   +0x014 SystemArgument1  : Ptr32 Void
   +0x018 SystemArgument2  : Ptr32 Void
   +0x01c Lock             : Ptr32 Uint4B
```

DPC 리스트는 현재 프로세서의 KPRCB(Kernel Processor Control Region Block) 구조체에서 찾을 수 있다. KPRCB는 현재 프로세서의 FS:[0x1C]에 위치한 KPCR(Kernel Processor Control Block)에 의해 처리된다. KPRCB는 KPCR 구조체의 시작점부터 0x120 바이트이다.

```
kd> dt nt!_KPRCB
   ...
   +0x860 DpcListHead        : _LIST_ENTRY
   +0x868 DpcStack           : Ptr32 Void
   +0x86c DpcCount           : Uint4B
   +0x870 DpcQueueDepth      : Uint4B
   +0x874 DpcRoutineActive : Uint4B
   +0x878 DpcInterruptRequested : Uint4B
   +0x87c DpcLastCount       : Uint4B
   +0x880 DpcRequestRate     : Uint4B
   +0x884 MaximumDpcQueueDepth : Uint4B
   +0x888 MinimumDpcRate     : Uint4B
   +0x88c QuantumEnd         : Uint4B
   +0x890 PrcbPad5           : [16] UChar
   +0x8a0 DpcLock            : Uint4B
   +0x8a4 PrcbPad6           : [28] UChar
   +0x8c0 CallDpc            : _KDPC
   +0x8e0 ChainedInterruptList : Ptr32 Void
   +0x8e4 LookasideIrpFloat : Int4B
   +0x8e8 SpareFields0       : [6] Uint4B
   +0x900 VendorString       : [13] UChar
   +0x90d InitialApicId      : UChar
   +0x90e LogicalProcessorsPerPhysicalProcessor : UChar
   +0x910 MHz                : Uint4B
   +0x914 FeatureBits        : Uint4B
   +0x918 UpdateSignature    : _LARGE_INTEGER
   +0x920 NpxSaveArea        : _FX_SAVE_AREA
   +0xb30 PowerState         : _PROCESSOR_POWER_STATE
```

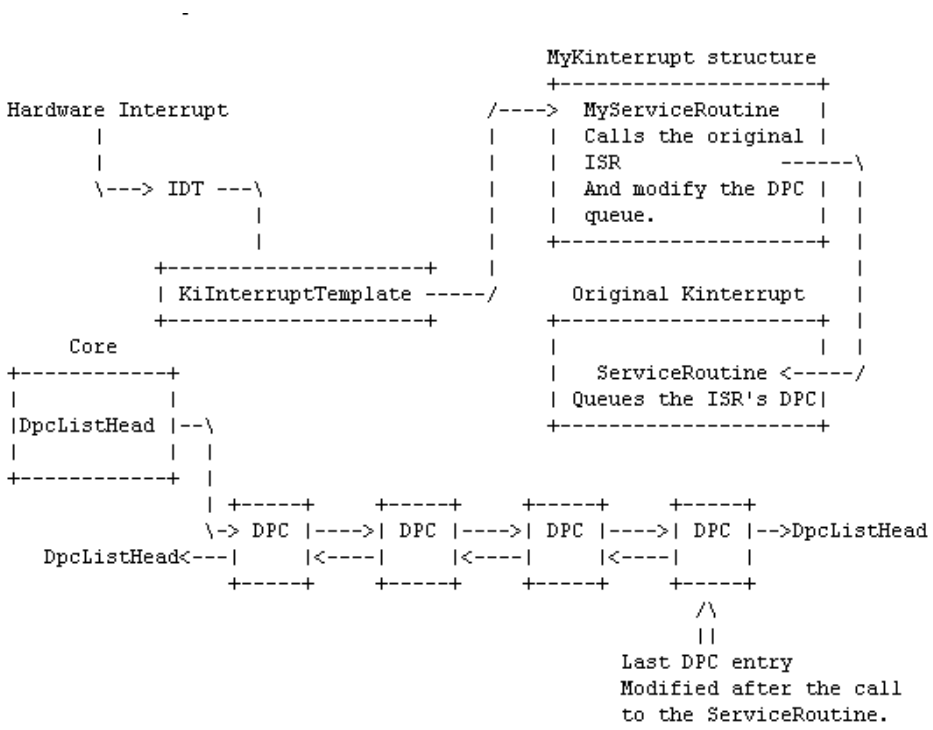
이제 우리는 인터럽트의 DPC를 검색할 수 있는 방법을 알았으니, 우리는 쉽게 바꿀 수 있고 데이터를 처리한다.

키보드를 위한 DPC는 키보드의 ISR이 호출한 IxQueuCurrentKeyboardInput안에 있는 KeInsertQueueDpc에 의해 큐에 들어간다.

```
kd> dt nt!_KDPC 80e3461c
```

```
+0x000 Type                : 19 ; 19=DpcObject
+0x002 Number              : 0 ''
+0x003 Importance          : 0x1 ''
+0x004 DpcListEntry        : _LIST_ENTRY [ 0xffdff980 - 0x80559684 ]
+0x00c DeferredRoutine     : 0xfa815650      void  i8042prt!I8042KeyboardIsrDpc
+0x010 DeferredContext     : 0x80e343b8
+0x014 SystemArgument1     : (null)
+0x018 SystemArgument2     : (null)
+0x01c Lock                : 0xffdff9c0   -> 0
```

아래는 공격의 특징이다.



2.3.3 Applicaton 1 : Kernel keylogger

POC를 설계할 시간이다. 이 예제에서 우리는 키보드 키스트로크의 스니핑 방법을 보게 될 것이다. 이전에 보았듯이 우리는 인터럽트에 의해 발생한 DPC를 통제할 수 있다. DPC의 키보드 인터럽트를 설정할 수 있는 I8042KeyboardIsrDpc를 탈취할 것이다. 우리의 DPC 핸들러로 실제 루틴의 행동을 재생할 수 있을 것이지만 유감스럽게 이런 종류의 코드는 작성하기가 힘들다 그래서 우리는 코드를 몇 조각으로 쪼갰고 리버싱 기술(lazy hacker style로 알려진)을 사용하였다.

우리의 DPC 핸들러는 KeyboardClassServiceCallback 루틴을 호출해야 하는데 이 루

틴은 Kbdclass 드라이버에 의해 제공된다. 이 콜백은 디바이스의 입력 데이터 버퍼를 클래스 데이터 큐로 전달한다. 키보드 드라이버 함수는 DPC 루틴에서 이 클래스 서비스 콜백을 호출해야 한다.

KeyboardClassServiceCallback의 프로토타입을 보여준다.

```
VOID
  KeyboardClassServiceCallback (
    IN PDEVICE_OBJECT   DeviceObject,
    IN PKEYBOARD_INPUT_DATA   InputDataStart,
    IN PKEYBOARD_INPUT_DATA   InputDataEnd,
    IN OUT PULONG   InputDataConsumed
    );
```

인자값에 대해 알아보자.

DeviceObject : 클래스 디바이스 오브젝트의 포인터이다.

InputDataStart : 포트 디바이스의 입력 데이터 버퍼에 있는 첫번째 키보드 입력 데이터 패킷의 포인터이다.

InputDataEnd : 포트 디바이스의 입력 데이터 버퍼에 있는 마지막 키보드 입력 데이터 패킷의 포인터이다.

InputDataConsumed : 루틴에 의해 전달된 키보드 입력 데이터 패킷의 개수를 나타내는 포인터이다.

KEYBOARD_INPUT_DATA는 아래와 같이 정의 되어 있다.

```
typedef struct _KEYBOARD_INPUT_DATA {
  USHORT   UnitId;
  USHORT   MakeCode;
  USHORT   Flags;
  USHORT   Reserved;
  ULONG   ExtraInformation;
} KEYBOARD_INPUT_DATA, *PKEYBOARD_INPUT_DATA;
```

DPC 핸들러는 단지 MakeCode 멤버를 조사하여 KEYBOARD_INPUT_DATA 구조체에 넣으면 된다. MakeCode(또는 scancode)는 키가 누르거나 떼었을 때 시스템으로 보내지는 키보드에 의해 보내지는 데이터를 보여준다. 그 키는 자신의 스캔 코드를 가지고 있으며, 시스템은 보통 그 스캔코드를 code page에 따라서 번역하여 준다. 예를 들어 scancode 19d는 고전적인 US 키보드에서 'e'로 번역된다.

CAPSLOCK가 활성화되어있는지 알기 위해서는 작동하는 키보드 디바이스에 IOCTL을 보내야 하지만 우리는 오직 PASSIVE_LEVEL의 IRQL을 보낼 수 있다. 그래서 우리는 IoBuildDeviceIoControlRequest를 사용해 IOCTL을 보낼 수 있는 시스템 스레드를 사용할 것이다. 사실 스캔코드는 스핀락에 의해 잠겨 있는 리스트에 들어가 있고 스레드는 세마포어와 동시에 작동한다. 그 스레드는 오는 키스트로크를 읽어서 키코드로 스캔코드를 변환할 것이다. 커널 키로거인 Klog[29]가 한 거처럼 작동할 것이다.

2.3.4 Application 2 : NDIS packet sniffer

같은 방법으로 인터럽트는 네트워크 카드가 패킷을 받았을 때도 발생한다. 인터럽트가에 발생하면 NDIS ISR 핸들러(ndisMIsr) 루틴은 miniport ISR 인터럽트 핸들러를 실행한다. 그 ndisMIsr 루틴은 miniport ISR과 DPC를 감싸는 역할로써 사용된다. 아래와 같은 엔트리를 IDT에서 확인할 수 있다.

```
73:      81d63044 NDIS!ndisMIsr (KINTERRUPT 81d63008)
```

인터럽트가 발생하였을 때 ndisMIsr 루틴이 직접적으로 당신의 ISR 핸들러를 호출하지 않는다는 것을 의미한다. Miniport ISP은 ndisMIsr에 의해 호출되고, miniport DPC 또한 이 루틴에서 큐에 들어가게 된다. 큐에 들어간 DPC는 우리의 DPC miniport handler를 감싼 ndisMDpc이다. 결국 NDIS는 NDIS 5.1인 윈도우즈 XP에서 ndisMIsr과 ndisMDpc 루틴으로 모든 인터럽트 프로세스를 감싼다. 이 실행이 NDIS 6.0인 윈도우즈 비스타에서도 여전히 가능한지는 알지 못한다.

우리는 우리의 핸들러로 ndisMDpc 핸들러를 가로챌 수 있다. NDIS로 MiniprotDPC 루틴을 훅하는 게 아니라 직접적으로 ndisMDpc 루틴을 훅하는 같은 방법으로 처리한다. 왜일까? 우리가 ndisMDpc가 MiniportDpc 루틴을 보호하는 것을 알고 있기 때문이다. 그리고 사실 MiniportDpc는 너무 miniport 디바이스의 하드웨어에 의존한다. 각 miniport 디바이스는 NDIS_MINIPORT_BLOCK 구조체에서 보여준다. 이 구조체는 아래에서 보이는 NDIS_MINIPORT_INTERRUPT 구조체에서 참조하는 것을 확일 할 수 있다.

```
kd> dt ndis!_NDIS_MINIPORT_INTERRUPT
   +0x000 InterruptObject   : Ptr32 _KINTERRUPT
   +0x004 DpcCountLock      : Uint4B
   +0x008 Reserved          : Ptr32 Void
   +0x00c MiniportIsr       : Ptr32 Void
   +0x010 MiniportDpc       : Ptr32 Void
```

```
   +0x014 InterruptDpc       : _KDPC
   +0x034 Miniport           : Ptr32 _NDIS_MINIPORT_BLOCK
   +0x038 DpcCount           : UChar
   +0x039 Filler1            : UChar
   +0x03c DpcsCompletedEvent : _KEVENT
   +0x04c SharedInterrupt    : UChar
   +0x04d IsrRequested       : UChar
```

ndisMDpc 루틴을 보았다면 첫 번째 인자만을 사용하고, 이 인자는 NDIS_MINIPORT_INTERRUPT 구조체를 참조한다는 것에 주목하라. ndisMDpc 함수는 이 구조체의 MiniportDpc 필드를 호출할 것이다. 우리는 시스템으로 오는 패킷을 컨트롤하기 위해서 이 포인터를 가로채기만 하면 된다.

NDIS 문서는 miniport DPC 루틴은 관련된 프로토콜 드라이버에게 받은 패킷의 배열은 NdisMIndicateReceicePacket 함수를 호출함으로써 이용할 수 있다는 것을 알려야 한다고 설명하고 있다.

```
VOID
  NdisMIndicateReceivePacket(
    IN NDIS_HANDLE   MiniportAdapterHandle,
    IN PPNDIS_PACKET   ReceivePackets,
    IN UINT   NumberOfPackets
    );
```

ndis.h 헤더파일에서 얻은 내용이다.

```
#define NdisMIndicateReceivePacket(_H, _P, _N)                ₩
{                                                             ₩
    (*((PNDIS_MINIPORT_BLOCK)(_H))->PacketIndicateHandler)( ₩
                          _H,                                 ₩
                          _P,                                 ₩
                          _N);                                ₩
}
```

miniport로 오늘 패킷을 걸러내기 위해서 MiniportDpc 루틴 안에서 PacketIndicateHandler를 가로채야 한다. 가끔은 NDIS_MINIPORT_BLOCK안에 존재하는 ethFilterDprIndicateReceivePacket 루틴을 가로채야 한다. 포인터를 가로 채었다면 우리는 모든 것을 처리하는 실제 MiniportDpc 루틴을 호출한다. 그 후에 우리는 숨기기 위해서 NDIS_MINIPORT_BLOCK안에 있는 PacketIndicateHandler 핸들러를 복구한다. 우리가 해야만 하는 내용이다.

- ndisMIsr 루틴에 의해 큐에 들어간 DPC의 루틴을 가로 채야한다.

- ndisMDpc를 가로챈 후에 miniport의 NDIS_MINIPORT_BLOCK 구조체 안에 있는 PacketIndicateHandler를 수정 한다.

- MiniportDpc 루틴은 MdisMIndicateReceivePacket 매크로를 호출한다. 필터함수는 우리의 일을 하기위해 호출된다.

필터를 통하여 우리는 오는 패킷을 수정하거나 모니터할 수 있다. 예를 들면 우리의 PacketIndicateHandler 혹은 설치된 루트킷이 함수를 실행시킬 때 태크를 통해 오는 패킷을 검색할 수 있다.

2.4 Conclusion about stealth hooking on IDT

이 파트에서 우리는 모든 인터럽트 전용의 전역 템플릿 함수를 사용함으로서 윈도우가 하드웨어 인터럽트를 어떻게 관리하는 지 보았다. 각 인터럽트를 착실히 전진시키는 템플릿 루틴의 요소는 직접적으로 탐지할 수 없는 위조된 템플릿 루틴을 만들 수 있는 이 공격의 중점이다. 공격을 숨기기 위한 두 가지 포인트를 남겼다.

- 우리는 오직 동적으로 할당되고 차근차근 실행되는 코드를 수정한다.

- 우리는 실행될 때 코어를 항상 선점하는 높은 권한을 일시적으로 갖고 동적으로 할당되는 구조체를 가로챈다.

그래서 공격 범위를 제한한다 할지라도 하드웨어를 컨트롤하는 것은 중요한 구성성분에 도달할 수 있는 최고의 방법이다. 마지막으로 우리는 이런 특징을 통하여 시스템을 속였다. 그리고 그 목적은 루트킷을 숨기는 것이다.

3. Owning NonPaged pool[7] using stealth hooking

루트킷의 정교함은 커널을 어떻게 부수는 가에 의존한다. 많은 복잡한 기술들이 커널과 하드웨어에 대한 이해도가 진보되면서 발표되었다. 오늘날 커널을 부수기 위한 방법이 너무 많다. 강한 보호는 부수기를 어렵게 만들었다. 우리는 컨트롤을 얻는 다른 방식을 보여줄 것이다. 다음 기술들은 커널 메모리 할당자에게 접근하는 것을 허락한다.

우리의 목표는 어떤 훅의 사용도 없이 모든 Nonpaged[8] 위치에서의 실행을 획득하는 것이다. 어떠한 후킹의 검증 그리고 코드 기반 비교 또는 해싱조차도 우회해야한다. 할당자에 의해 사용되는 데이터를 고치는 것에 의해 실행될 것이다. 우리는 단지 그것에 거스르는 코드를 사용하는 것의 개념을 허락하였다. 우리는 이 개념은 다른 요소나 다른

7) 페이징되지 않는 메모리 공간을 관리하는 프로세서
8) 페이징되지 않는 메모리 공간을 뜻한다.

방법에서도 성공적으로 사용된다고 믿는다.

우리는 이 기술이 완벽하다고 당신에게 확신을 드리지는 않는다. 현재 보호와 탐지 시스템을 피해야 한다. 가장 중요한 것은 커널 코드 행동에 기반을 둔 공격을 예방하고 막기위해 간단한 수정보다 더 많은 것이 필요하다는 것이다.

3.1 Kernel allocation layout review

모든 운영체제처럼 윈도우즈 커널을 메모리 할당과 헤제를 위해 몇 가지 함수를 가지고 있다. 가상 메모리는 페이지라고 불리는 메모리 블록으로 구성된다. 인텔 x86 구조에서 페이지 사이즈는 4096 바이트이고 모든 할당 요청은 작다. 그러므로 ExAllocatePoolWithTag와 ExFreePoolWithTag같은 커널 함수는 다음 할당을 위해 사용하지 않은 메모리 블록을 유지한다. 내부의 함수는 페이지가 필요한 시기에 직접적으로 하드웨어와 상호 작용한다. 모든 프로시저는 드라이버가 커널 실행을 신뢰하기 때문에 복잡하고 섬세하다.

3.1.1 - Difference between Paged and NonPaged pool

커널 시스템 메모리는 두 가지의 다른 pool로 나누어져 있다. 사용된 메모리를 구분하기 위해 나누어졌다. 시스템은 페이지가 상주되어야 하는지 임시적으로 사용되어야 하는지 알아야 한다. 페이지 폴트 핸들러는 IRQL이 DPC또는 DISPATCH 레벨보다 낮을 때만 페이지가능한 메모리를 복구한다. 페이지된 풀은 시스템의 안과 밖으로 페이지가 될 수 있다. 페이지 아웃된 메모리 블록은 파일 시스템에 저장될 것이고 페이지된 메모리의 사용되지 않은 부분은 메모리에 상주하지 않을 것이다. Nonpaged pool은 모든 IRQL 레벨에서 나타나고 중요한 일을 위해 이용된다.

page.sys 파일에는 페이지 아웃된 메모리를 포함한다. 비스타 커널[32] 속에 서명이 없는 코드를 삽입하여 공격된다. 몇 가지 해결책은 커널 메모리 페이징을 억제하는 방법이 논의 되었다. Joanna Rutkowska님은 보다 보안적인 방법으로 이 해결책을 막았지만 작은 실제 메모리를 손실해야 했다. 마이크로소프트는 단지 낮은 디스크의 접근을 거부한다. Paged와 NonPaged layout은 윈도우즈 커널[33]의 중요한 특징이다.

PagePool 핸들링은 전체적으로 다르기 때문에 NonPaged pool 레이아웃에 중점을 두고 있다. NonPaged pool은 전형적인 힙 실행 다음이라서 어느 정도 고려된다. 시스템 pool에 관한 많은 정보는 Microsoft Windows Internals[34]에서 찾아볼 수 있다.

3.1.2 - NonPaged pool tables

할당 알고리즘은 사용되는 크기만큼 빠르게 할당되어야 한다. 그래서 세 개의 다른 테이블이 존재하고 가가 사이즈 범위가 한정되어 있다. 우리는 대부분 메모리 관리 알고리즘에서 이 구조를 알고 있다. 하드웨어로부터 메모리 블록을 검색하는 것은 시간이 걸린

다. 윈도우즈는 빠른 응답과 메모리 공간 낭비를 피하기 위한 밸런스를 가지고 있다. 메모리 블록이 다음 할당공간을 위해 정보를 저장한다면 응답 시간은 빨라질 것이다. 이런한 경우 너무 많은 메모리를 사용한다면 메모리를 사용할 수 없게 될 수도 있다.

각 테이블은 메모리 블록을 저장하기 위해 다른 방법을 수행한다. 우리는 각 테이블과 그 테이블을 어디서 찾을 수 있는지 보게 될 것이다.

NonPaged lookaside는 256byte와 동등하거나 적은 크기를 덮는 프로세서마다 있는 테이블이다. 각 프로세서는 오직 IRQL, GDT, IDT 같은 싱글 프로세서에 관한 데이터를 저장하는 PCR(processor control register)을 가지고 있다. PCRB(processor control region)이라고 불리는 확장은 lookasides 테이블을 포함하고 있다. 다음 Windbg 덤프는 NonPaged lookaside 테이블과 구조체를 보여준다.

```
kd> !pcr
KPCR for Processor 0 at ffdff000:
    Major 1 Minor 1
  NtTib.ExceptionList: 805486b0
      NtTib.StackBase: 80548ef0
     NtTib.StackLimit: 80546100
   NtTib.SubSystemTib: 00000000
        NtTib.Version: 00000000
    NtTib.UserPointer: 00000000
        NtTib.SelfTib: 00000000

               SelfPcr: ffdff000
                  Prcb: ffdff120
                  Irql: 00000000
                   IRR: 00000000
                   IDR: ffffffff
        InterruptMode: 00000000
                   IDT: 8003f400
                   GDT: 8003f000
                   TSS: 80042000

        CurrentThread: 80551920
           NextThread: 00000000
           IdleThread: 80551920

            DpcQueue:  0x80551f80 0x804ff29c
```

```
kd> dt nt!_KPRCB ffdff120
   ...
   +0x5a0 PPNPagedLookasideList : [32]
      +0x000 P                  : 0x819c6000 _GENERAL_LOOKASIDE
      +0x004 L                  : 0x8054dd00 _GENERAL_LOOKASIDE
   ...
```

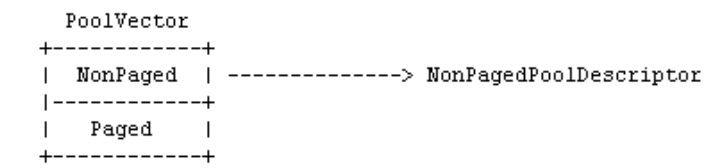
```
kd> dt nt!_GENERAL_LOOKASIDE
   +0x000 ListHead           : _SLIST_HEADER
   +0x008 Depth              : Uint2B
   +0x00a MaximumDepth       : Uint2B
   +0x00c TotalAllocates     : Uint4B
   +0x010 AllocateMisses     : Uint4B
   +0x010 AllocateHits       : Uint4B
   +0x014 TotalFrees         : Uint4B
   +0x018 FreeMisses         : Uint4B
   +0x018 FreeHits           : Uint4B
   +0x01c Type               : _POOL_TYPE
   +0x020 Tag                : Uint4B
   +0x024 Size               : Uint4B
   +0x028 Allocate           : Ptr32       void*
   +0x02c Free               : Ptr32       void
   +0x030 ListEntry          : _LIST_ENTRY
   +0x038 LastTotalAllocates : Uint4B
   +0x03c LastAllocateMisses : Uint4B
   +0x03c LastAllocateHits : Uint4B
   +0x040 Future             : [2] Uint4B
```

looaside 테이블은 더블 링크드 리스트보다 빠른 블록 검색을 하게 해준다. 이 최적화를 위해 잠그는 시간을 실제로 중요하고 싱글 링크드 리스트는 소프트웨어 락킹보다 빠른 미케니즘이다.

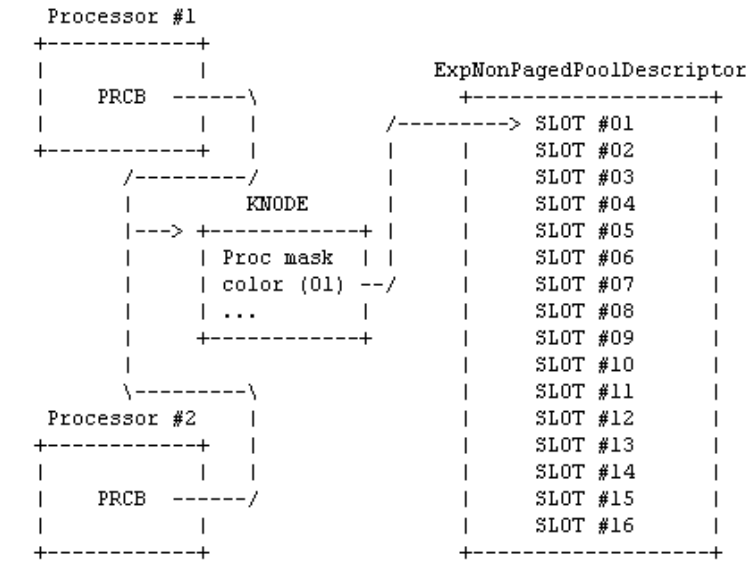
ExInterlockedPopEntrySList 함수는 lock 명령어를 사하여 잠그는 하드웨어를 사용하는 싱글 링크드 리스트로부터 엔트리를 가져오는데 사용한다.

PPNPageLockasideList는 전에 이야기했던 lookaside 테이블이다. P와 L이라는 2개의 리스트를 포함하고 있다.GENERAL_LOOKASIDE 구조체의 Depth 필드는 싱글 리스트 리스트헤드에 있는 많은 엔트리의 개수를 정의하고 있다. 그 시스템은 정규적으로 다른 카운터를 사용하여 업데이트를 한다. 업데이트 알고리즘은 프로세스 번호에 기반을 두고 있고 P와 L은 서로 다른 번호를 사용한다. P 리스트의 Depth는 매우 작은 블록에서의 실행을 최적화하기 위해서 L 리스트보다 자주 업데이트를 한다.

2번째 테이블은 얼마나 많은 테이블이 사용되고 시스템이 그들을 관리하는 방법에 의존한다. 할당 시스템은 크기가 4080바이트보다 적거나 크다면, 또는 lookaside 실패했다면 걷는다. 타겟 테이블이 바낄 수는 있지만 항상 같은 POOL_DESCRIPTOR 구조체를 가진다. 싱글 프로세서에서 PoolVector라고 불리는 변수는 NonPagedPoolDescriptor 포인터를 검색하는 데 사용된다. 멀티 프로세서에서 ExpNonPagePoolDescriptor 테이블인 pool descriptors를 포함하는 16슬롯을 가지고 있다. 각 프로세서 PRCB는 KNODE 구조체를 가르킨다. 노드는 한 프로세서 이상과 연결될 수도 있고, ExpNonPagedPoolDescriptor의 인덱스로 사용된 color 필드를 포함하고 있다. 다음 그림은 이 알고리즘을 설명하고 있다.



[Figure 1 - Single processor pool descriptor]



[Figure 2 - Multiple processor pool descriptor]

전역 변수는 ExpNumberOfNonPagePools는 멀티 프로세서가 사용되는 경우에 저장된다. 프로세서 번호를 반영해야하지만 운영체제 버전에 따라 바뀔 수 있다.

다음 덤프는 POOL_DESCRIPTOR 구조를 보여준다.

```
kd> dt nt!_POOL_DESCRIPTOR
   +0x000 PoolType         : _POOL_TYPE
   +0x004 PoolIndex        : Uint4B
```

```
   +0x008 RunningAllocs      : Uint4B
   +0x00c RunningDeAllocs    : Uint4B
   +0x010 TotalPages         : Uint4B
   +0x014 TotalBigPages      : Uint4B
   +0x018 Threshold          : Uint4B
   +0x01c LockAddress        : Ptr32 Void
   +0x020 PendingFrees       : Ptr32 Void
   +0x024 PendingFreeDepth   : Int4B
   +0x028 ListHeads          : [512] _LIST_ENTRY
```

HAL 라이브러리 부분의 큐에 들어간 스핀락 동기화는 pool 디스크립터에서 병행성을 제한하는데 사용한다. 오직 한 스레드와 한 프로세서만이 pool 디스크립터로부터의 엔트리를 접근하거나 링크를 해제할 수 있다. HAL 라이브러리는 다른 구조들을 바꾼다. 그리고 싱글 프로세서에서 발생하는 간단한 IRQL인 HAL 라이브러리는 멀티 프로세서에서는 더욱 복잡한 큐에 들어간 시스템이 된다. 기본 pool 디스크립터에서 일반적인 NonPaged 큐에 들어간 spinlock는 잠긴다(LockQueueNonPagePoolLock). 그 밖의 일반적인 큐에 들어간 spinlock는 생성된다.

3번째오 마지막 테이블은 4080byte 보다 큰 프로세서들에 의해 공유된다. MmNonPagedPoolFreeListHead는 다른 테이블이 메모리를 필요로 할 때 사용된다. 시스템이 차지하는 모든 상위 페이지를 가지고 있는 마지막 하나를 제외하고 각각 하나의 페이지 번호를 나타내는 4개의 LIST_ENTRY로 구성된다. 이 테이블에 대한 접근은 LockQueueNonPagePoolLock이라고 불리는 nonpaged queued spinlock에 의해 보호된다. 작은 블록의 프로시저가 해제되는 동안 ExFreePoolWithTag는 이전 블록과 다음 블록을 현재 블록과 합병한다. 하나의 페이지와 동등하거나 더 큰 블록을 만들 수 있다. 이 경우 새로운 블록은 MmNonPagedPoolFreeListHead 테이블에 추가도니다.

3.1.3 - Allocation and free algorithms

커널 할당은 OS 버전이 큰 영향을 주지는 않지만 알고리즘은 유저모드의 힙 할당만큼 어렵다. 이 파트에서 우리는 프로시저를 할당하고 해제하는 동안 테이블들 사이의 기본적인 행동을 설명할 것이다. 많은 세부 사항이 통기화 미케니즘 같은 방법에 나와있다. 알고리즘은 기술 설명을 이해하도록 도와주지만 또한 커널 할당의 기본 요소를 이해해야 한다. 커널 익스플로잇이 이 문서에는 없지만 pool 오버플로우는 이 알고리즘의 몇 군데를 이해하는데 필요한 흥미로운 주제이다.

NonPaged Pool 할당 알고리즘(ExAllocatePoolWithTag)

```
   IF [ Size > 4080 bytes ]
```

```
    [
      - MiAllocatePoolPages 함수를 호출한다.
        - MmNonPagePoolFreeListHead LIST_ENTRY 테이블을 참고한다.
        - 필요하다면 하드웨어 메모리를 검색한다.
        - (헤더없이) 할당된 메모리 페이지르 리턴한다.
    ]

  IF [ Size <= 256 bytes ]
    [
      - PPNPagedLookasideList 테이블로부터 엔트리를 팝한다.
      - 발견되면 메모리 블록을 리턴한다.
    ]

  IF [ ExpNumberOfNonPagedPools > 1 ]
    - ExpNumberOfNonPagedPools에 있는 PoolDescriptor와 사용된 인덱스는
      PRCB KNODE color로부터 온다.
  ELSE
    - PoolDescriptor은 NonPagedPoolDescriptor로 심볼이 지정된 첫 번째 엔트리
      PoolVerctor이다.

  FOREACH [ >= Size entry of PoolDescriptor.ListHeads ]
    [
      IF [ Entry is not empty ]
        [
          - 필요하다면 엔트리를 언링크하고 쪼갠다.
          - 메모리 블록을 리턴한다.
        ]
    ]
- MiAllocatePoolPages 함수를 호출한다.
    - MmNonPagedPoolFreeListHead LIST_ENTRY 테이블을 참고한다.
    - 크기에 맞게 정확하게 쪼갠다.
    - 새로운 메모리 블록을 리턴한다.
```

NonPaged pool 해제 알고리즘(ExFreePoolWithTag)

```
  IF [ 메모리블록이 페이지 정렬되어 있다. ]
    [
      - MiFreePoolPages 함수를 호출한다.
        - 블록 형태를 결정한다. (Paged or NonPaged)
        - MmNonPagePoolFreeListHead에 얼마나 많은 블록이 있는지에 의존하여
```

```
        하드웨어에게 양도한다.
    ]
  ELSE
    [
      - 가능하다면 이전 블록과 다음 블록을 현재 블록과 합병한다.

      IF [ NewMemoryBlock size <= 256 bytes ]
        [
          - PPNPageLookasideList 엔트리 depth 필드를 살펴보고 유지해야하는지
            확인한다.
          - 메모리 블록 lookaside 리스트에 넣을 수 있다면 리턴한다.
        ]

      IF [ NewMemoryBlock size <= 4080 bytes ]
        [
          - PooDescriptor가 사용되야 하는지 결정하기위해 POOL_HEADER
            PoolIndex 변수를 사용한다.
          - 적절한 LIST_ENTRY 배열 엔트리에 삽입한다.
          - 모든 것이 잘 되었다면 리턴한다.
        ]

      - MmNonPagePoolFreeListHead에 얼마나 많은 블록이 있는지에 의존하여
        하드웨어에게 양도한다.
    ]
```

Paged pool 알고리즘 페이지 정렬된 블록에게 특히 매우 다르다. 작은 크기 관리는 NonPaged에서 먼 것이 아니라 NonPaged와 Paged pool이 전체적으로 분할된 것을 어세블리 코드에서 명확하게 볼 수 있다.

3.2 Getting code execution abusing allocation code

우리의 목적은 오직 NonPaged Pool에 항상 할당 시도를 하는 실행코드를 얻는 것이다. 이 결과는 타켓이 된 코드에 의해 사용된 데이터를 바꾸는 것만 해야 한다. 우리의 목적은 커널 코드가 대표적인 데이터 환경을 바꿈으로서 우리의 재미를 충족시킬 수 있다는 것을 증명하는 것이다. 우리의 목표는 NonPaged allocation를 통제하도록 개발된 새로운 루트킷에 기초를 두고 있다.

우리는 한 개의 페이지와 똑같거나 더 많은 할당을 위한 실행 코드를 만들어보자. 우리가 이전 파트에서 보았듯이 세 번째와 마지막 테이블에 관계된다.

3.2.1 - Data corruption of MmNonPagedPoolFreeListHead

MmNonPagedPoolFreeListHead는 페이지 정렬된 메모리 블록을 메모리 할당을 빨리 하기 위해 보존한다. 링크는 LIST_ENTRY 구조체를 사용하여 메모리 블록을 연결한다. 이 구조체는 일반적으로 사용되고 예블들면 윈도우즈 힙에서 사용된다.

```
kd> dt nt!_LIST_ENTRY
   +0x000 Flink            : Ptr32 _LIST_ENTRY
   +0x004 Blink            : Ptr32 _LIST_ENTRY
```

MmNonPagedPoolFreeListHead 접근은 일반적으로 NonPaged queued spinlock인 LockQueueNonPagePoolLock에 의해 보호된다. 오직 한 스레드와 한 프로세서만이 이 구조체를 잠그고 고칠 수 있다는 것을 보장한다.

그래서 우리는 완벽해 보이는 할당과 링크를 끊는 프로시저를 통제할 수 있는 방법을 필요하다. 우리는 엔트리를 속이거나 가능한 아주 큰 크기를 이용해 링크가 끊어지면 현재 실행되는 코드를 고칠 수 있는 링크드 리스트를 무력화 시킬 수 있다. 커널 레벨에서 당신은 아무 보호도 없는 데이터인 코드를 고칠 수 있다. Unlinking은 힙 깨부실 때 사용되었지만 코드를 수정하는 것은 유저모드에서는 불가능하다. spinlock이 우리에게 독점적 권리를 보장하기 때문에 레이스 컨디션 같은 위험은 없다. Page guard protection revers 문서에서 코드는 오직 5분 동안만 체크된다는 것을 보여준다. 수정이 되었는지 알든 모르든, 실제 코드는 바뀐다.

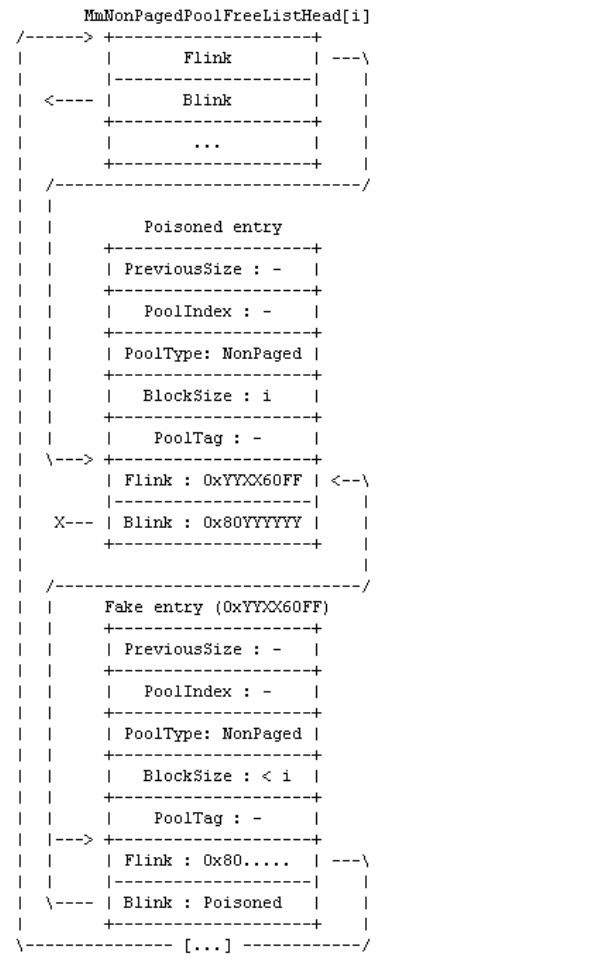
이 방법은 많은 이점 뿐 아니라 단점도 가지고 있다. 모든 단점을 열거해보자.

- unlinking을 기본 실행은 리스트를 참조 못하게 만든다.
- 페이지가 안된 메소드와 항상 리스트의 처음 블록을 통해 지나가야한다. 그렇지 않으면 몇몇 호출은 실패할 수 있다.
- 코드 경로를 부수고 바로 또는 조금 후에 가로채기를 당한 것을 모르고 모든 것이 잘 작동하도록 리턴해 줘야 한다.
- 프로세서 선행패치는 자신의 코드 수정 위험이 있다.

언링킹은 opcode와 새 루틴을 만들기 위해 4바이트를 덮어써야 한다. 우리의 경우, 우리에게 영향을 받은 현재 문맥과 레지스터는 언링킹 엔트리를 가리켜야 한다. 커널 버전과 service pack들의 변화 때문에 싱글 레지스터 선택하지 않고 가리켜야 한다. 우리는 문맥에 대해 논의하면서 일반적인 상황에 대한 이야기도 할 것이다. 우리는 헥사코드로 FF60XX인 jmp [reg+XX]를 사용하도록 선택했다.

이 기술의 효율성은 MmNonPagePoolFreeListHead를 참고 가능하도록 유지하는데 있다. LIST_ENTRY같은 더블 링크드 리스트는 Flink가 정상이라면 참고가능하다. 그러므로 우리는 0xXXXX60FF인 Flink 주소를 선택할 수 있다. 공백은 코드 주소를 가르킨다. 리틀 엔디안 방식을 사용하는 Intel x86 구조는 주소를 찾기가 매우 쉽다. 우리는

opcode 오프셋을 점검하고 너무 어려운 가능성을 버려야 한다. 다음 그림은 무력화된 엔트리를 설명하고 있다.



Unlinking instruction : mov [0x80YYYYYY], 0xYYXX60FF
New Opcode after unlinking : jmp [reg+XX] (FF 60 XX)

[Figure 3 - Poisoned double linked list]

이 그림은 MmNonPagedPoolFreeListHead 엔트리 계층을 보여준다. 이는 예상되는 언링킹과 그 후 코드 실행을 보장하는 그림이다. 우리는 이 계층을 유지하거나 위치를 잃어버릴 것이다. NonPaged block은 두 가지 다른 가상 메모리 주소를 가진다. 두 번째 메모리 영역의 시작은 MmNonPagePoolExpansionStrat에 저장된다. 비워주는 함수는 때때로 확장 NonPaged pool로부터 메모리를 해제하기 위해 호출된다. 비워지는 것을 피하기 위해서 우리는 잠긴 paged pool 블록을 사용해야 한다.메모리 블록을 잠그기 위해서는 MmProbeAndLockPages 함수를 호출하면 된다. 이 잠금은 상주하는 메모리 영역으로 기록된다. 더욱 안전한 방법은 MmMapLockePagesSpecifyCache 함수를 사용하여 NonPaged 블록을 재매핑하는 것이다. 이 매핑은 확장 NonPaged pool 메모리 영역 전이기 때문에 안전하다. 잠긴 Paged pool 블록을 사용하는 것은 전체적으로 다르게 주소를 생성한다. NonPaged 블록의 주소를 찾아보면 확연한 차이점이 보일 것이다. 가상 메모리는 매우 크지만 0xYYXX60FF같은 주소를 찾는데 너무 많은 시간이 걸리지는 않

는다. 우리의 기술이 실행될 때까지 페이지의 락을 풀 수 없다.

우리는 코드 경로를 부수기 위해 두 가지 다른 상태로 구분한다. 첫 번째 상태는 우리 블록이 선택되어졌을 때이다. 두 번째 상태는 우리의 블록이 언링크 되었을 때이다. 만약 다음의 속인 엔트리를 선택하여 첫 번째 단계로 리턴할 수 있다면 우리는 쉽게 코드를 참고하는 것을 계속할 수 있다. 우리는 일반적인 접근을 사용하여 성공했다. DISPATCH_LEVEL과 동등한 IRQL에서 우리는 MmNonPagePoolFreeListHead 엔트리에 몇몇 잘못된 포인터를 넣어 오류를 일으킨다. 페이지 폴트 핸들러를 훅하여 우리는 매 번 올바른 문맥을 복구하고, 그들의 상태 사이에 문맥 변화를 저장하는 첫 번째와 두 번째 스테이지를 볼 수 있다.

MiAllocatePoolPages의 어셈블리 코드:

```
lea      eax, [esi+8] ; Stage #1 esi is selected block and esi+8 its size
  cmp      [eax], ebx   ; Check with needed size
  mov      ecx, esi
  jnb      loc_47014B
  [...]

loc_47014B:
  sub      [esi+8], ebx
  mov      eax, [esi+8]
  shl      eax, 0Ch
  add      eax, esi
  cmp      _MmProtectFreedNonPagedPool, 0 ; Protected mode, don't care
  mov      [ebp+arg_4], eax
  jnz      short loc_47016E
  mov      eax, [esi]       ; ₩ Stage #2
  mov      ecx, [esi+4]     ; | Unlinking
  mov      [ecx], eax       ; | procedure
  mov      [eax+4], ecx     ; /
  jmp      short loc_470174
```

이제 인터럽트 폴트 핸들러(int 0xE)를 훅한 기술을 테스트하는 동안 어떤 일을 하는지 살펴보자 :

```
  lea      eax, [esi+8]
                        ; Stage #1 - Check with needed size
  cmp      [eax], ebx   ; ----> PAGE FAULT esi = 0xAAAAAAAA | eax = esi
+ 8
```

```
                        ;       - We keep EIP and all registers
                        ;       - Scan all registers for 0xAAAAAAAA +/- 8
                        ;         and correct the current context. Continue.
  mov       ecx, esi
  jnb       loc_47014B
  [...]

loc_47014B:
  sub       [esi+8], ebx
  mov       eax, [esi+8]
  shl       eax, 0Ch
  add       eax, esi
  cmp       _MmProtectFreedNonPagedPool, 0 ; Protected mode, don't care
  mov       [ebp+arg_4], eax
  jnz       short loc_47016E
  mov       eax, [esi]        ; ₩ Stage #2 - Unlinking procedure
  mov       ecx, [esi+4]      ; |
  mov       [ecx], eax        ; | ------> PAGE FAULT ecx = 0xBBBBBBBB
                              ; |                   eax = 0xCCCCCCCC
                              ; |       - Keep EIP and sub this context from
                              ; |         Stage #1 saved context
                              ; |       - Change fault registers and
                              ; |         structure pointers. Continue.
  mov       [eax+4], ecx      ; /
  jmp       short loc_470174
```

잘못된 주소 0xAAAAAAAA, 0xBBBBBBBB 그리고 0xCCCCCCCC는 페이지 폴트를 유도하기 위해서 잘못된 주소를 가르키고 있어야 한다. 이 테스트는 오직 한 번 우리가 모든 프로세서에 대해 독점적인 권리를 가지고 있을 때 만들어 진다. int 0xE(페이지 폴트) 핸들러는 후에 복구된다.

이 일반적인 기술은 단지 선택된 블록 크기가 체크되기 전에 사용가능한 문맥을 복구할 수 있도록 허락한다. 일단 우리가 코드 실행을 하면 우리는 문맥 변화를 적용하고 현재 블록 레지스터를 바꾼 후 첫 번째 스테이지 주소로 리턴한다. 두 스테이지가 매우 가깝기 때문에 잘 작동한다. 일단 선택된 블록의 크기가 체크되면 언링킹은 즉시 실행된다.

주어진 예는 MmNonPagePoolFreeListHead 테이블의 싱글 LIST_ENTRY에 기반을 두었지만 당신은 모든 엔트리를 무력화해야 한다. 만약 주어진 엔트리가 비였다면 그 알고리즘은 다음 엔트리를 찾게 된다. 그것은 우리가 할당할 때마다 한 번 이상 호출되어야 한다는 것을 뜻한다. 우리는 싱글 할당에서 다양한 호출을 관리하기 위한 메커니즘을

만들었다. 만약 첫 번째 엔트리가 비었다면 두 번째 엔트리가 사용된다. 그 후 우리는 두 번 또는 더 많은 호출될 수가 있다. 현재 테이블을 체크 해봄으로써 우리는 같은 할당에서 미래의 코드 실행을 예측할 수 있고 할당 요청시마다 한 번 이상의 실행 페이로드를 피할 수 있다.

선챙처리는 실행하기 전에 메모리에서 한 명령어보다 많은 것을 검색할 수 있는 프로세서이다. 몇몇 프로세서는 가능한 많은 명령어를 실행하기 위해 복잡한 분기 예측 알고리즘을 사용한다. 몇 가지 테스트 후에 우리는 캐쉬에 입련된 메모리 어드레스에서 수정이 일어 날 때 캐쉬에 입련된 코드가 무용지물이 되는 것을 보았다. 우리의 드라이버는 코드 수정 후에도 현재의 명령어 후에 정상적으로 작동하는 곳을 지원한다. 우리는 또한 far jump와 같은 cache를 통한 선행처리를 비울 수 있는 구체적인 명령어를 찾을 수 있지만 옵션으로만 사용할 수 있다.

이 기술은 한 페이지랑 동등하거나 많은 NonPaged 할당을 위해 코드 실행을 우리에게 준다. 커널 코드로 만들어지고 우리의 루틴에 의해 지워진 stealth 혹으로 이루었다. 이 할당들은 많이 사용되지 않아서 완벽하지는 않다. 다음 파트에서 이 기술이 어떻게 모든 NonPaged pool 할당을 통제하는 있는 지에 대해 설명하겠다.

3.2.2 - Expend it for every size

다른 리스트들은 동기화 메커니즘이 독점적이지 않아서 같은 방법으로 가로챌 수가 없다. 한 번에 여러 스레드가 실행된다면 몇몇 어셈블리 코드를 바꾸는 것은 까다롭게 되었다. 우리의 방법은 이전 기술들이 어떤 할당에서도 동작한다고 보장한다. 일단 우리가 통제권한을 가지면 우리는 옳은 리턴값으로 ExAllocatePoolWithTag 문맥을 복구하는 방법을 찾을 수 있다. 우리는 메모리 할당자의 정보 기록없이 해야만 한다. 우리의 할당자를 만드는 것은 가능하다 하지만 윈도우즈는 방대하다. 그리고 우리가 그 일을 완벽하게 할 수 있을 것이다.

할당 동안에 lookaside 리스트는 처음으로 체크된다. 엔트리를 팝하는 데 엔트리 값이 널이 아니라면 사용한다. 이 엔트리는 GENERAL_LOOKASIDE ListHeader 필드에서 가져온다. 이 필드 구조체는 SLIST_HEADER이다.

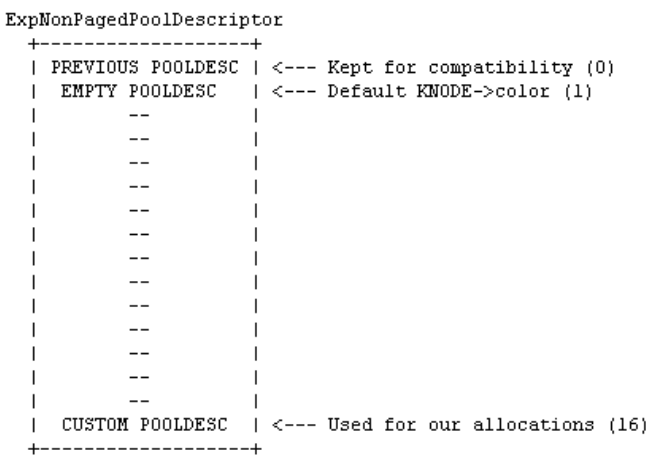
```
kd> dt nt!_SLIST_HEADER .
   +0x000 Alignment          : Uint8B
   +0x000 Next               :
      +0x000 Next               : Ptr32 _SINGLE_LIST_ENTRY
   +0x004 Depth              : Uint2B
   +0x006 Sequence           : Uint2B
```

ExInterlockedPopEntrySList 함수는 SLIST_HEADER 구조체에서 엔트리를 팝한다. Next 필드는 다음 SLIST 노드(싱글 링크드 리스트)를 가리키고 있다. Depth 필드는 얼마나 많은 엔트리가 리스트에 있는지 나타낸다. ExFreePoolWithTag는

GENERAL_LOOKASIDE 최적의 depth를 현재 SLIST_HEADER depth와 비교한다. ExAllocatePoolWithTag는 이 필드를 체크하지 않는다. 단지 몇몇 엔트리가 Next 필드로 팝이 된다면 살펴볼 뿐이다. NonPaged lookable 테이블에 있는 할당과 해제 프로시저를 저해하는 방법은 Netx 필드를 널로, Depth 필드는 0xFFFF로 설정하는 것이다. 이 상태는 보존될 것이고 이 테이블은 더 이상 사용되지 않을 것이다.

ExpNonPagedPoolDescriptor 테이블이 사용된 방법을 파헤치는 우리의 기술 확대는 전체적으로 신뢰한다. 이전 파트에서 우리는 프로세스에 포함되는 전역 변수 ExpNumberOfNonPagedPools에 대해 설명하였다. NonPaged pools의 수를 확장하는 것이 가능하고 그 후 현재 노드 color 필드를 이용하여 실행시킨다. 할당동안 KNODE color 필드는 pool 디스크립터가 사용된 것을 정의한다. 그 후 해제 프로시져가 실행되는 동안 POOL_HEADER의 PoolIndex 필드는 pool descriptor color를 유지한다.

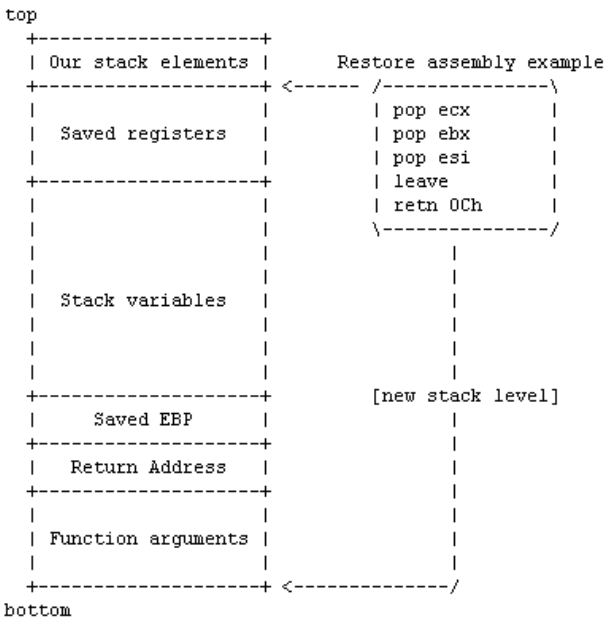
우리는 우리의 이점을 주는 좋은 특징을 사용할 수 있다. 모든 프로세서에서 기본적으로 KNODE color는 빈 pool 디스크립터를 가리킨다. 그것은 우리의 기본 기술을 사용하여 코드 실행을 만들 도록 이끈다. MiAllocatePoolPages 함수가 리턴한 주소가 고전적인 페이지 원형할당을 사용한 주소가 아니라면 작은 할당이 일어난다. 우리는 사용자 color를 복사하기 위해 PRCB KNODE 포인터를 바꾸고 ExAllocatePoolWithTag를 호출한다. 할당과 블록 관리에 관련된 모든 것은 운영체제에 따라 다르지만 필요할 때 실행된다. 완벽하게 일을 했다면 리턴된 블록 PoolIndex는 우리의 pool 디스크립터를 가리킬 것이고 프로시저를 해제한다. 싱글 프로세서에서 어떻게 되는지 살펴보자.



[Figure 4 - Corrupted ExpNonPagedPoolDescriptor]
[on single processor]

이 설정은 단지 예일 뿐이다. 당신은 당신이 원하는 만큼 배열을 관리할 수 있다. 우리의 오래된 pool 디스크립터로부터 이전을 블록을 새로운 블록과 바꿀 수 있다. 여러개의 pool 디스크립터를 사용하는 것도 가능하다. 멀티 프로세서 구조에서 특별히 이상한 행동을 할 수 있으므로 시스템 pool 디스크립터 재순환 과정을 인지하고 있어야 한다.

일단 우리가 새롭게 할당된 블록을 가지고 있다면 우리는 ExAllocatePoolWithTag 리턴 주소에 리턴할 수 있다. MiAllocatePoolPages 새로운 페이지를 검색하기 위해 호출되고 현재 pool 디스크립터에 검색결과를 채운다. 우리가 정상적으로 리턴할 수 없는 건 명백하고 페이지 할당이 일어나도록 시킨다. Intel x86 구조 스택은 지역 변수, 인자값, 저장된 레지스터를 저장하기 위해 사용된다. 윈도우즈 컴파일러는 지역 변수와 수정되기 전의 레지스터들을 저장하면서 시작한다. 다음 사진은 코드 실행 후의 스택 구성을 보여준다.



[Figure 5 - Stack context after code execution]
[~ small blocks case ~]

복구된 어셈블리 부분은 완벽하게 문맥을 복구한 현재 함수의 정확한 어셈블리를 보여준다. 리턴 전에 하는 pop 명령어의 첫 번째 시리즈와는 같지 않다. 몇몇 레지스터를 넣지 않는 것은 매우 위험하다. 스택 변수가 저장되었을 때 함수의 시작부분을 살펴보면 입력된 레지스터의 개 수를 추론할 수 있다. 윈도우즈 컴파일러에서는 정말 간다하고 쉽게 입력된 레지스터 개수를 계산할 수 있다. pop 레지스터의 개수가 필요하면 간단한 디어셈블리 분석을 통해 얻을 수 있다. 개수를 알아내는 것은 MiAllocatePoolPages와 ExAllocatePoolWithTag을 위해 해야 한다. 우리는 스택에 저장된 리턴 주소를 바꾸고 예상한 MiAllocatePoolPages 주소로 간다. 마지막 단계는 리턴 값을 위해 eax 레지스터를 설정하는 것이다. 두 함수 모두 eax 값을 보전하여 리턴한다. 우리의 분석기는 다이나믹하고 각 레지스터를 팝하고 등록한다. 그 방법은 버전이 다를지라도 우리가 적절한 문맥을 복구하도록 해준다.

윈도우즈 컴파일러는 예측하기가 쉽고 너무 이상한 어셈블리 코드를 생성하지 않다. 이 기술은 이론상으로 sdtcall 기법을 사용하는 모든 어셈블리 코드에서 가능하다. 접근하는 방식이 다른 컴파일러와 달랐다.

3.3 Exploit our position

이 문서는 오직 데이터를 수정하여서 윈도우즈 커널을 깨부수는 방법을 보여준다. 함수 포인터도 정적인 후킹도 아니고 다른 고전적인 방법도 아니다. 어떤 다른 설명에도 없다. 하지만 몇 가지 구체적인 예제없이는 완성할 수 없다. 나는 그 제한이 상상이라고 몸소 믿는다.

3.3.1 Generic stack redirection

할당은 당신이 신뢰해야하는 알려진 문맥과 함수같이 너무 많은 공간에서 일어난다. 일단 모든 것이 설정되고 독점적으로 실행되기 전이라면 몇몇 스택 리다이렉션 데이터베이스가 만들어 진다.

첫 번째 방법은 스택 백트레이싱에 구체적인 함수가 보인다면 핸들러를 호출하는 것이다. 스택 백트레이싱은 오직 리턴 주소를 보여줄 뿐 함수를 호출하지는 않는다. 디버거는 깊은 분석과 심볼을 참조하여 그 함수들을 해결하였다. 이 특징을 실행하기 위해서는 너무 많은 시간이 걸린다. 그냥 ExAllocatePoolWithTag 스택 프레임에 있는 구체적인 리턴 주소를 목표로 하는 것이 낳다. 이 방법은 점검 속도를 확실히 개선한다. 이 경우 우리는 우리가 목표로 삼은 구체적인 함수인 스택 리다이렉션 API를 나타내야 한다. 그 후 전상적인 호출을 실행하거나 우리의 함수로 오는 프로시저를 실행하면 된다. 이 시간 동안 모든 할당은 중요한 백트레이스 스택을 보여줄 것이다.

우리는 IRP가 목표이다. 우리는 함수가 IRP 디스패치 테이블을 참고하여 처리한다는 것을 알고 있다. 우리는 또한 NonPaged 블록을 할당한다는 것을 리버싱을 통해 알고 있다. I/O 요청이 생기면 우리의 API는 몇 가지 NonPaged call을 등록하고 나서 인지한다.

넓은 의미에서 예비 문맥 정보를 참고하여 적절한 핸들러를 호출할 것이다. 때때로 문맥을 얻는 것만으로는 충분하지 않다. 두 번째 방법은 같은 원리에 머무르는 것이지만 우리의 핸들러가 함수가 끝난 후에 호출된다는 것을 보장하기 위해 스택을 고치는 것이다. 효율성은 당신의 목표가 무엇이고 어떻게 수정했는지에 의존한다.

3.3.2 Userland process code injection

이 기술은 또한 안전한 어플리케이션을 깨부수기 위해 유저랜드에 코드를 삽입하는데도 사용된다. NonPaged 할당은 커널 모드에서 많이 발생하고 모든 프로세스에서 일어난다. win32k.sys같은 몇몇 커널 드라이버는 많이 유저랜드를 호출한다. 이 호출은 KeUserModeCallback 함수에 의해 이루어진다. 유저랜드를 호출하기 위해 임시적으로

바뀌는 유저랜드 스택을 수정하는 것이다. 이용가능한 함수는 테이블에 의해 제한된다.

커널에서 하는 유저랜드 인젝션은 상주하면 안 되고 오직 브라우저같이 알려진 실뢰할 수 있는 어플리케이션에 관해서만 고려해야 한다. 인젝션은 신뢰되는 프로그램의 숨겨진 객체를 실행할 수 있는 explorer.exe에서 할 수 있다. KeUserModeCallback 알고리즘은 쉽게 따라 만들거나 카피하고 재배치 할 수 있다. Redirection 테이블은 호출을 재설정하여 깨부술 수 있다. 우리는 또한 유저랜드 호출을 깨부수는 것에 관해서도 생각해볼 수 있다. 이용할 수 있는 함수 점검을 추가하기 위해 어떤 센스를 만들지는 않는다.

4. Detection

이 문서에서 IDT를 깨부수거나 진보된 기술을 사용한 할당 알고리즘이 장래성이 있는지 당신에게 신뢰를 주기위해 노력하지 않는다. 대부분의 탐지 툴은 오직 이 컴퓨터에 루트킷이 있는지 없는지만 나타낸다. 그것은 모듈이 신뢰할만한 건지 확인해야 하는 번거로움이 있다. 그것은 루트킷으로서 안티바이러스나 방화벽을 탐지한다. 보호 계층은 모든 루트킷처럼 행동하기 때문에 루트킷으로써 자신 스스로를 탐지한다. 그리고 루트킷을 막거나 지우기 위해서 보호 계층을 요구하지 않는다. 루트킷 문서는 쉽게 보호를 우회하는 방법을 많이 소개한다. 하지만 우리는 루트킷이 그 순간동안 그것들을 필요로 하지 않기 때문에 전체적으로 쉽게 그들의 기술을 볼 수 없다.

행동 변화 소프트웨어는 입증할 수 있는 운영체제의 부분을 탐지할 수 있다. 기본적으로 알려진 메모리 구조체를 점검하는 루틴을 가지고 있다. 필요하다면 LIST_ENTRY 구조체의 무결성을 점검할 수 있다. 우리는 우리가 원하는 만큼 루트킷 보호를 비난할 수 있지만 운영체제에 근접해서 루트킷을 탐지하는 것은 대부분 불가능하다. 커널 구성요서에 대한 많은 정보를 주는 것은 확실히 더욱 세련된 공격을 이끌 것이다. 반면에, 표면적인 공격은 줄일 수 있을 것이다. 운영체제 지향의 방어는 정말 좋다. 앞으로 나올 보호 개선은 운영체제에서 되어야 할 것이다.

hypervisors같은 가상화를 이용한 하드웨어 개선 방법이 있으니 루트킷을 탐지하거나 막기 위해 하드웨어로 확장해야 한다. 그 것은 커널 계층의 검색없이 운영체제의 행동을 실제로 통제할 수 있게 제공한다. PAX같이 윈도우즈 환경에서 실행이 불가능한 몇몇 보호 기술은 하드웨어 특징에 의지한다. 우리의 기술은 프로세서에서 발생하는 이벤트를 모니터링하거나 기록하는 것을 통해 발견될 수 있다. 오늘날 그렇게 하는 것은 가능해졌지만 수행 결과는 중요하다.

우리의 공격은 signatures같은 보호프로그램을 사용하면 막을 수 있다. 공격은 일반적인 보호를 만드는데 걸리는 시간이 얼마인지에 따라 결정된다. 이런 경우에 Patchguard는 중요한 도구이다.

5. Conclusion

이 문서는 세련된 소프트웨어 가로채기는 여전히 대부분의 보호 기술을 피할 수 있고 어떤 실행 또는 불안정한 행동을 피할 수 있다. 비록 기술들이 거의 믿겨지지 않지만 개념의 기술적인 증명으로 오직 고려되어 져야 한다. 새로운 보호 기술은 충분히 효율적이지 않거나 기존의 것이다. 새로운 보호기술들은 수백만 대의 컴퓨터를 목표로 하는 루트킷의 위협을 대표하지는 않는다. 리버싱은 소프트웨어 루트킷 기술을 개선하기 위한 중요한 도구이다. 현재 있는 루트킷을 탐지하는 것만으로는 충분하지 않다. 루트킷을 제거하거나 감염을 막을 수 없는 보호 프로그램은 무용지물이다. 드라이버 서명은 현재 감연 엔트리를 멈추기위해 고안된 좋은 아이디어이다. 하지만 감염 예방은 로컬 커널 이용을 포함하고 있다. 일반적인 공격 탐지는 안티-루트킷 보호와 운영체제 설계에서 중요한 개선이 필요로 하다.

6. References

[1] Holy Father, Invisibility on NT boxes, How to become unseen on Windows NT (Version: 1.2)
http://vx.netlux.org/lib/vhf00.html

[2] Holy Father, Hacker Defender
https://www.rootkit.com/vault/hf/hxdef100r.zip

[3] 29A
http://vx.netlux.org/29a

[4] Greg Hoglund, NT Rootkit
https://www.rootkit.com/vault/hoglund/rk_044.zip

[5] fuzen_op, FU
http://www.rootkit.com/project.php?id=12

[6] Peter Silberman, C.H.A.O.S, FUto
http://uninformed.org/?v=3&a=7

[7] Eeye, Bootroot
http://research.eeye.com/html/tools/RT20060801-7.html

[8] Eeye, Pixie
http://research.eeye.com/html/papers/download/
eEyeDigitalSecurity_Pixie%20Presentation.pdf

[9] Joanna Rutkowska and Alexander Tereshkin, Blue Pill project
http://bluepillproject.org/

[10] Frank Boldewin, A Journey to the Center of the Rustock.B Rootkit
http://www.reconstructer.org/papers/
A%20Journey%20to%20the%20Center%20of%20the%20Rustock.B%20Rootkit.zip

[11] Frank Boldewin, Peacomm.C - Cracking the nutshell
http://www.reconstructer.org/papers/
Peacomm.C%20-%20Cracking%20the%20nutshell.zip

[12] Stealth MBR rootkit
http://www2.gmer.net/mbr/

[13] EP_X0FF and MP_ART, Unreal.A, bypassing modern Antirootkits
http://www.rootkit.com/newsread.php?newsid=647

[14] AK922 : Bypassing Disk Low Level Scanning to Hide File
http://rootkit.com/newsread.php?newsid=783

[15] CardMagic and wowocock, DarkSpy
http://www.fyyre.net/~cardmagic/index_en.html

[16] pjf, IceSword
http://pjf.blogone.net

[17] Gmer
http://www.gmer.net/index.php

[18] Pageguard papers (Uniformed) :

- Bypassing PatchGuard on Windows x64 by skape & Skywing
  http://www.uninformed.org/?v=all&a=14&t=sumry

- Subverting PatchGuard Version 2 by Skywing
  http://www.uninformed.org/?v=all&a=28&t=sumry

- PatchGuard Reloaded: A Brief Analysis of PatchGuard Version 3 by Skywing
  http://www.uninformed.org/?v=all&a=38&t=sumry

[19] Greg Hoglund, Kernel Object Hooking Rootkits (KOH Rootkits)

http://www.rootkit.com/newsread.php?newsid=501

[20] Windows Heap Overflows - David Litchfield
http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

[21] Bypassing Klister 0.4 With No Hooks or Running a Controlled Thread Scheduler by 90210 - 29A
http://vx.netlux.org/29a/magazines/29a-8.rar

[22] Microsoft, Debugging Tools for Windows
http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[23] Kad, Phrack 59, Handling Interrupt Descriptor Table for fun and profit
http://phrack.org/issues.html?issue=59&id=4#article

[24] Wikipedia, Southbridge
http://en.wikipedia.org/wiki/Southbridge_(computing)

[25] Wikipedia, Northbridge
http://en.wikipedia.org/wiki/Northbridge_%28computing%29

[26] The NT Insider, Stop Interrupting Me -- Of PICs and APICs
http://www.osronline.com/article.cfm?article=211 (login required)

[27] Russinovich, Solomon, Microsoft Windows Internals, Fourth Edition
Chapter 3. System Mechanisms -> Trap Dispatching

[28] MSDN, KeyboardClassServiceCallback
http://msdn2.microsoft.com/en-us/library/ms793303.aspx

[29] Clandestiny, Klog
http://www.rootkit.com/vault/Clandestiny/Klog%201.0.zip

[30] Alexander Tereshkin, Rootkits: Attacking Personal Firewalls
www.blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf

[31] MSDN, NdisMIndicateReceivePacket
http://msdn2.microsoft.com/en-us/library/aa448038.aspx

[32] Subverting VistaTM Kernel For Fun And Profit by Joanna Rutkowska

http://invisiblethings.org/papers/
joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt

[33] Vista RC2 vs. pagefile attack by Joanna Rutkowska
http://theinvisiblethings.blogspot.com/2006/10/
vista-rc2-vs-pagefile-attack-and-some.html

[34] Russinovich, Solomon, Microsoft Windows Internals, Fourth Edition
Chapter 7. Memory Management -> System Memory Pools

[35] KeUserModCallback ref - "Ring0 under WinNT/2k/XP" by Ratter - 29A
http://www.illmob.org/files/text/29a7/Articles/29A-7.003

[36] Joanna Rutkowska - Towards Verifiable Operating Systems
http://theinvisiblethings.blogspot.com/2007/01/
towards-verifiable-operating-systems.htm