The "Ultimate" AntiDebugging Reference

5/4/2011

Peter Ferrie

v 0.2

번역 by Montouesto

2016/09/14

Contents

리버스 엔지니어링을 할 때 가장 흔하게 사용되는 툴로는 디버거가 그리고 IDA 같은 디스어셈블러 툴은 그 다음으로 여겨진다. 결과적으로 안티 디버깅 트릭들은 아마 리버스 엔지니어링을 방해하는 코드의 가장 흔한 특징일 것이고 안티 디스어셈블리 구조는 그 다음일 것이다. 이러한 트릭들은 간단하게 디버거의 존재를 탐지하는것 부터 시작해서 디버거를 비활성화하기, 디버거의 통제로부터 벗어나기 그리고 심지어는 디버거의 취약점을 공격하는 것까지도 존재한다. 디버거의 존재는 간접적으로 추론될 수 있고 특정한 디버거의 경우에는 탐지될 수도 있다. 디버거를 못쓰게 하거나 벗어나게 하는 것은 일반적인 방법들 그리고 특정한 방법들을 통해 달성될 수 있다. 그러나 취약점을 공격하는 것은 특정한 디버거에서만 가능하다. 물론 디버거는 굳이 취약점 공격이 시도될 수 있도록 존재할 필요는 없다.

일반적으로 디버거가 로드되면 디버기와 상호작용할 수 있게 하기 위해 디버기의 환경은 운영 체제에 의해 변화된다(이것의 예외로 Obsidian 디버거가 있다). 이러한 변화들 중 어떠한 것은 다른 것들보다 명확하며 디버기의 동작에 다른 방식으로 영향을 미친다. 이 환경은 또한 디버거가 프로세스를 생성하는 데 사용되었는지 또는 디버거가 이미 실행 중인 프로세스를 어태치 하였는지에 따라 다른 방식들로 바뀔 수 있다.

다음은 디버거의 존재를 탐지하는데 사용되는 알려진 기법들 중에 선택된 것이며 몇몇은 그것에 대한 방어이다.

주의: 이 문서는 32비트 그리고 64비트의 많은 양의 코드를 포함한다. 간단함을 위해서 64비트 버전들은 모든 스택, 힙 포인터 그리고 핸들들을 32비트로 가정하였다.

1. NtGlobalFlag

프로세스 환경 블록(Process Environment Block)의 NtGlobalFlag 필드는 시스템이 만드는 가장 간단한 변화들 중 하나이면서 또한 가장 오해받는 것 중 하나이다. NtGlobalFlag 필드는 PEB에 존재하는데 윈도우 32비트 버전의 경우 오프셋 0x68에, 64비트 버전의 경우 0xBC에 존재한다. 필드의 기본 값은 0이다. 이 값은 디버거가 프로세스를 어태치할 때는 바뀌지 않지만 프로세스 제어 하에서는 어느 정도 바뀔 수 있다. 이 외에도 특정한 값으로 설정할 때 사

용될 수 있는 두 레지스트리 키들이 있다. 이러한 공헌자들 없이 디버거에 의해 생성되는 프로세스는 기본적으로 이 필드에서 고정된 값을 가질 것이지만 이 특정한 값은 특정한 환경 변수를 사용함에 따라 변경될 수 있다. 이 필드는 플래그들의 집합으로 구성된다. 디버거에 의해 생성된 프로세스는 다음과 같은 플래그 집합을 가질것이다.

FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)

그래서 디버거의 존재를 탐지하는 방법이 바로 이러한 플래그들의 조합을 검사하는 것이다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서, 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
mov eax, fs:[30h]        ; Process Environment Block
mov al, [eax+68h]        ; NtGlobalFlag
and al, 70h
cmp al, 70h
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
push 60h
pop rsi
gs:lodsq                 ; Process Environment Block
mov al, [rsi*2+rax14h]   ; NtGlobalFlag
and al, 70h
cmp al, 70h
je being_debugged
```

64비트 버전의 32비트 프로세스에서는 각각의 프로세스 환경 블록이 존재한다는 점을 주의해야 한다. 64비트 부분의 이 필드들은 32비트 부분과 같은 방식으로 영향을 받는다.

그러므로 64비트 윈도우 환경을 검사하기 위해 이 32비트 코드를 사용하는 다른 검사가 존재한다.

```
mov eax, fs:[30h]        ; Process Environment Block
                         ; 64bit Process Environment Block
                         ; follows 32bit Process Environment Block
mov al, [eax+10bch]      ; NtGlobalFlag
```

```
and al, 70h
cmp al, 70h
je being_debugged
```

흔한 실수들 중 하나는 먼저 다른 비트들을 마스킹하지 않고 직접 비교를 하는 것이다. 이 경우에 만약 다른 비트들이 설정된 경우 디버거의 존재를 놓치게 된다.

디버거가 이 기법을 막는 방법은 프로세스를 재개하기 전에 값을 다시 0으로 바꾸는 것이다. 그러나 위에서 주의하였 듯이 초기값은 4가지 방법들 중 하나로 인해 바뀔 수 있다. 첫 번째 방법은 시스템의 모든 프로세스들에 영향을 미치는 레지스트리 값과 관련된다. 이 레지스트리 값은 "HKLM₩System₩CurrentControlSet₩Control₩Session Manager" 레지스트리 키의 "GlobalFlag" 문자열 값이다. 여기의 이 값은 NtGlobalFlag 필드에 위치하므로 추후에 윈도우에 의해 바뀔 수 있다. 이 레지스트리 값의 변화는 효과를 발휘하기 위해서 리부팅을 요구한다. 이 요구는 디버거의 존재를 탐지하는(또한 이 레지스트리 값을 인식하는) 다른 방식으로 이끌어 진다. 만약 디버거가 자신의 존재를 숨기기 위해 레지스트리 값을 복사해서 NtGlobalFlag 필드에 복사한다면 그리고 레지스트리 값이 변화됐지만 시스템이 리부팅되지 않았다면, 디버거는 아마 실제 값 대신 새로운 값을 사용하도록 속아 넘어갈 것이다. 만약 프로세스가 실제 값이 레지스트리 값에서 보이는 것과 다르다는 것을 알았다면 디버거는 발각될 것이다. 실제 값을 결정하는 한 방식으로 프로세스가 다른 프로세스를 실행하게 하고 자신의 NtGlobalFlag 값을 질의하는 것이 있다. 레지스트리 값을 알지 못하는 디버거는 또한 이 방식으로 들쳐진다.

두 번째 방법도 레지스트리 값과 관련있지만 단지 named 프로세스에만 영향을 미친다. 이 레지스트리 값은 또한 "GlobalFlag" 문자열 값이지만 "HKLM₩Software₩Microsoft₩Windows NT₩CurrentVersion₩Image File ExecutionOptions₩<filename>" 레지스트리 키의 것이다. "<filename>"은 반드시 실행 파일(DLL이 아니라)의 이름(파일이 실행될 때 플래그들은 이 이름에 적용된다)으로 대체되어야한다. 위에서 봤듯이 비록 윈도우에 의해 추후에 바뀌게 되지만 이 값은 NtGlobalFlag 필드에 위치한다. 이 방식을 사용해서 설정된 값은(만약 프로세스들이 존재한다면) 모든 프로세스들에 적용된 값과 병합된다.

값을 바꾸는 세번째 방법은 Load Configuration Table의 두 필드를 필요로 한다. 한 필드(GlobalFlagsClear)는 클리어할 플래그들을 열거하고, 다른 필드(GlobalFlagsSet)는 설정할 필드를 열거한다. 이러한 설정들은 GlobalFlag 레지스트리 값이 적용된 이후에 적용되서 GlobalFlag 레지스트리 값에서 명시된 값을 오버라이드할 수 있다. 그러나 이것들은 특정한 플래그들이 설정된채로 남아있을 때 윈도우가 설정한 값을 오버라이드할 수 없다(비록 이것들은 디버거가 프로세스를 생성할 때 설정된 플래그들을 제거할 수 있지만). 예를들면 FLG_USER_STACK_TRACE_DB (0x1000) 플래그를 설정하는 것은 윈도우가 FLG_HEAP_VALIDATE_PARAMETERS (0x40) 플래그를 설정하도록 유발한다. 만약 FLG_USER_STACK_TRACE_DB 플래그가 GlobalFlag 레지스트리 값들 중 하나로 설정되

면 FLG_HEAP_VALIDATE_PARAMETERS 플래그가 Load Configuration Table에서 클리어링하기 위해 마크됨에도 불구하고 이후 프로세스 로드 기간에 윈도우에 의해서 계속 설정될 것이다.

네 번째 방식은 디버거가 프로세스를 생성할 때 윈도우가 만드는 특유의 변경이다. "_NO_DEBUG_HEAP" 환경 변수를 설정하면 디버거 때문에 NtGlobalFlag의 세 힙 플래그들은 설정되지 않을 것이다. 물론 이것들은 여전히 Load Configuration Table에서 GlobalFlag 레지스트리 값들 또는 GlobalFlagSet 필드에 의해 설정될 수 있다.

2. Heap flags

힙은 NtGlobalFlag와 함께 초기화되는 두 플래그를 포함한다. 이 필드들의 값은 디버거의 존재에 영향을 받지만 윈도우의 버전에 따라 다르다. 또한 이 필드들의 위치도 윈도우의 버전에 따라 다르다. 이 두 플래그의 이름은 "Flags"와 "ForceFlags"이다. Flags 필드는 32비트 윈도우 NT, 윈도우 2000, 윈도우 XP의 경우 힙의 오프셋 0x0C에 위치한다. 64비트 윈도우 XP의 경우 플래그 필드는 힙의 오프셋 0x14에 위치하며, 64비트 윈도우 비스타 이후 버전에서는 힙의 0x70에 위치한다. ForceFlags 필드는 32비트 윈도우 NT, 윈도우 2000, 윈도우 XP의 경우 오프셋 0x18에, 64비트 윈도우 비스타 버전 이후에는 힙의 오프셋 0x74에 위치한다.

Flags 필드의 값은 일반적으로 모든 버전의 윈도우에서 HEAP_GROWABLE (2)로 설정된다. ForceFlags 필드의 값은 일반적으로 모든 버전의 윈도우에서 0이다. 그러나 32비트 프로세스의 경우(64비트 프로세스는 이런 의존성이 없다) 이 값들 모두 호스트 프로세스의 서브시스템 버전에 의존한다. 필드 값들은 서브시스템 버전이 3.103.50 이상인 경우에만 언급된다. 만약 서브시스템 버전이 3.103.50이라면 두 필드 모두 HEAP_CREATE_ALIGN_16 (0x10000) 플래그가 설정된다. 만약 서브시스템 버전이 3.10보다 낮다면 파일은 실행되지 않는다. 이것은 특히 흥미로운데 왜냐하면 일반적인 기법은 디버거의 존재를 숨기기 위해 그것들 각각의 필드에 2와 0의 값을 넣기 때문이다. 그러나 만약 서브시스템 버전이 확인되지 않으면 이 행동은 디버거의 행동을 숨기려는 시도를 드러낼 것이다.

윈도우 NT, 윈도우 2000, 32비트 윈도우 XP에서 디버거가 존재할 때 Flags 필드는 일반적으로 이 플래그들의 조합으로 설정된다.

HEAP_GROWABLE (2)
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_SKIP_VALIDATION_CHECKS (0x10000000)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

64비트 윈도우 XP, 윈도우 비스타 이후에서는 이 Flags 필드는 일반적으로 다음의 플래그들의 조합으로 설정된다.

HEAP_GROWABLE (2)
HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

디버거가 존재할 때, ForceFlags 필드는 일반적으로 이 플래그들의 조합으로 설정된다.

HEAP_TAIL_CHECKING_ENABLED (0x20)
HEAP_FREE_CHECKING_ENABLED (0x40)
HEAP_VALIDATE_PARAMETERS_ENABLED (0x40000000)

NtGlobalFlag 필드에서 FLG_HEAP_ENABLE_TAIL_CHECK가 설정되면, 힙 필드에서 HEAP_TAIL_CHECKING_ENABLED 플래그가 설정된다. FLG_HEAP_ENABLE_FREE_CHECK 플래그가 NtGlobalFlag 필드에서 설정되면 힙 필드에서 HEAP_FREE_CHECKING_ENABLED 플래그가 설정된다. 만약 FLG_HEAP_VALIDATE_PARAMETERS 플래그가 NtGlobalFlag 필드에서 설정되면 HEAP_VALIDATE_PARAMETERS_ENABLED 플래그가(그리고 윈도우 NT와 윈도우 2000에서는 HEAP_CREATE_ALIGN_16 (0x10000)가) 힙 필드에서 설정된다.

윈도우 XP 이후에서 이 행위는 환경 변수 "_NO_DEBUG_HEAP"를 생성함으로써 대신 기본 값을 사용하도록 유발하여 예방될 수 있다.

힙 플래그들은 또한 "HKLM₩Software₩Microsoft₩Windows NT₩CurrentVersion ₩Image File Execution Options₩<filename>" 레지스트리 키의 "PageHeapFlags" 문자열 값을 통해서 각 프로세스에 기반하여 제어될 수 있다.

힙의 위치는 여러 방식으로 검색될 수 있다. 한 방식은 kernel32 GetProcessHeap() 함수를 사용하는 것이다. 이것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하는 것과 같다.

```
mov eax, fs:[30h]          ; Process Environment Block
mov eax, [eax+18h]         ; get process heap base
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
push 60h
```

```
pop rsi
gs:lodsq                ; Process Environment Block
mov eax, [rax+30h]      ; get process heap base
```

64비트 버전의 윈도우에서 프로세스 환경 블록은 힙이 32비트 부분과 64비트 부분으로 나눠져서 각각 존재한다. 64비트 부분에서 필드들은 32비트 부분과 같은 방식으로 영향을 받는다.

그러므로 64비트 윈도우 환경을 검사하기 위해 이 32비트 코드를 사용하는 다른 검사가 존재한다.

```
mov eax, fs:[30h]       ; Process Environment Block
                        ; 64bit Process Environment Block
                        ; follows 32bit Process Environment Block
mov eax, [eax+1030h]    ; get process heap base
```

다른 방식은 kernel32 GetProcessHeaps() 함수를 사용하는 것이다. 이 함수는 간단하게 ntdll RtlGetProcessHeaps()로 포워드 된다. 이 함수는 프로세스 힙들의 배열을 반환한다. 리스트의 첫번째 힙은 kernel32 GetProcessHeap() 함수에 의해 반환된 것과 같다. 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 질의는 또한 이 32비트 코드를 사용하여 수행될 수 있다.

```
push 30h
pop esi  fs:lodsd       ; Process Environment Block
                        ; get process heaps list base
mov esi, [esi+eax+5ch]
lodsd
```

또는 64비트 윈도우 환경을 조사하기 위해 이 32비트 코드를 사용할 수 있다.

```
mov eax, fs:[30h]       ; Process Environment Block
                        ; 64bit Process Environment Block
                        ; follows 32bit Process Environment Block
mov esi, [eax+10f0h]    ; get process heaps list base
lodsd
```

그러므로 디버거의 존재를 탐지하는 방법은 Flags 필드에서 플래그들의 특별한 조합을 검사하는 것이다. 이 검사는 서브시스템 버전이 3.103.50 사이인 경우에 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h]          ; Process Environment Block
mov eax, [eax+18h]         ; get process heap base
mov eax, [eax+ebx+0ch]           ; Flags
                           ; neither HEAP_CREATE_ALIGN_16
                           ; nor HEAP_SKIP_VALIDATION_CHECKS
and eax, 0effeffffh        ; HEAP_GROWABLE
                           ;+ HEAP_TAIL_CHECKING_ENABLED
                           ;+ HEAP_FREE_CHECKING_ENABLED
                           ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000062h
je being_debugged
```

서브시스템 버전이 3.51 이상인 경우에는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용할 수 있다.

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h]          ; Process Environment Block
mov eax, [eax+18h]         ; get process heap base
mov eax, [eax+ebx+0ch]           ; Flags
                           ; not HEAP_SKIP_VALIDATION_CHECKS
bswap eax
and al, 0efh               ; HEAP_GROWABLE
                           ;+ HEAP_TAIL_CHECKING_ENABLED
                           ;+ HEAP_FREE_CHECKING_ENABLED
                           ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
                           ; reversed by bswap
cmp eax, 62000040h
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
push 60h
pop rsi
gs:lodsq                ; Process Environment Block
mov ebx, [rax+30h]      ; get process heap base
call GetVersion
cmp al, 6
sbb rax, rax
and al, 0a4h            ; HEAP_GROWABLE
                        ;+ HEAP_TAIL_CHECKING_ENABLED
                        ;+ HEAP_FREE_CHECKING_ENABLED
                        ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp d [rbx+rax+70h], 40000062h ; Flags
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 32비트 코드를 사용할 수 있다.

```
push 30h
pop eax
mov ebx, fs:[eax]       ; Process Environment Block
                        ; 64bit Process Environment Block
                        ; follows 32bit Process Environment Block
mov ah, 10h
mov ebx, [ebx+eax]      ; get process heap base
call GetVersion
cmp al, 6
sbb eax, eax
and al, 0a4h            ; Flags
                        ; HEAP_GROWABLE
                        ;+ HEAP_TAIL_CHECKING_ENABLED
                        ;+ HEAP_FREE_CHECKING_ENABLED
                        ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp [ebx+eax+70h], 40000062h
je being_debugged
```

kernel32 GetVersion() 함수 호출은 더 나아가 간단하게 2Gb 사용자 공간 설정에서는 오프셋 0x7ffe026c에 위치한 KUSER_SHARED_DATA 구조체의 NtMajorVersion 필드에서 값을 직접적으로 검색함으로써 난독화될 수 있다. 이 값은 모든 32비트 또는 64비트 윈도우 버전에

서 사용 가능하다.

디버거의 존재를 탐지하는 다른 방법은 ForceFlags 필드에서 특별한 플래그들의 조합을 검사하는 것이다. 이 검사는 서브시스템 버전이 3.103.50 사이인 경우 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h]          ; Process Environment Block
mov eax, [eax+18h]         ; get process heap base
mov eax, [eax+ebx+10h] ; ForceFlags
                           ; not HEAP_CREATE_ALIGN_16
btr eax, 10h               ; HEAP_TAIL_CHECKING_ENABLED
                           ;+ HEAP_FREE_CHECKING_ENABLED
                           ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000060h
je being_debugged
```

서브시스템 버전이 3.51 이상인 경우에는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
call GetVersion
cmp al, 6
cmc
sbb ebx, ebx
and ebx, 34h
mov eax, fs:[30h]          ; Process Environment Block
mov eax, [eax+18h]         ; get process heap base
                           ; ForceFlags
                           ; HEAP_TAIL_CHECKING_ENABLED
                           ;+ HEAP_FREE_CHECKING_ENABLED
                           ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp [eax+ebx+10h], 40000060h
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
push 60h
pop rsi
gs:lodsq                ; Process Environment Block
mov ebx, [rax+30h]      ; get process heap base
call GetVersion
cmp al, 6
sbb rax, rax
and al, 0a4h            ; ForceFlags
                        ; HEAP_TAIL_CHECKING_ENABLED
                        ;+ HEAP_FREE_CHECKING_ENABLED
                        ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp d [rbx+rax+74h], 40000060h
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위한 이 32비트 코드를 사용할 수 있다.

```
call GetVersion
cmp al, 6
push 30h
pop eax
mov ebx, fs:[eax]       ; Process Environment Block
                        ; 64bit Process Environment Block
                        ; follows 32bit Process Environment Block
mov ah, 10h
mov ebx, [ebx+eax]      ; get process heap base
sbb eax, eax
and al, 0a4h            ; ForceFlags
                        ; HEAP_TAIL_CHECKING_ENABLED
                        ;+ HEAP_FREE_CHECKING_ENABLED
                        ;+ HEAP_VALIDATE_PARAMETERS_ENABLED
cmp [ebx+eax+74h], 40000060h
je being_debugged
```

3. Heap

힙이 초기화될 때 힙 플래그들이 검사되고 설정된 플래그들에 따라서 환경에 추가적인 변화

가 생길 수 있다. 만약 HEAP_TAIL_CHECKING_ENABLED 플래그가 설정되면 연속적인 0xABABABAB가 32비트 윈도우 환경에서 할당된 블록의 정확히 끝에서 두 번 덧붙여진다. 만약 HEAP_FREE_CHECKING_ENABLED 플래그가 설정되면 다음 블록까지 slack 공간에 채우기 위해 추가적인 바이트들이 요구될 경우에 0xFEEEFEEE(또는 이것의 부분)이 덧붙여질 것이다. 그러므로 디버거의 존재를 탐지하는 방법은 이러한 값들을 검사하는 것이다. 만약 힙 포인터가 알려지면 이 검사는 힙 데이터를 직접적으로 검사함으로써 처리될 수 있다. 그러나 윈도우 비스타 이후에서는 32비트 그리고 64비트 플랫폼 모두에서 블록 사이즈를 인코드하기 위한 XOR 키의 도입을 통해 힙 보호를 사용한다. 이 키의 사용은 옵션이지만 기본적으로 사용된다. 또한 overhead 필드의 위치는 윈도우 NT/2000/XP/비스타 그리고 이후에서도 다르다. 그러므로 윈도우의 버전은 반드시 고려되어야 한다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor ebx, ebx
 call GetVersion
 cmp al, 6
 sbb ebp, ebp
 jb l1                      ; Process Environment Block
 mov eax, fs:[ebx+30h]
 mov eax, [eax+18h]     ; get process heap base
 mov ecx, [eax+24h]     ; check for protected heap
 jecxz l1
 mov ecx, [ecx]
 test [eax+4ch], ecx
 cmovne ebx, [eax+50h]          ; conditionally get heap key
l1: mov eax, <heap ptr>
 movzx edx, w [eax-8]   ; size
 xor dx, bx
 movzx ecx, b [eax+ebp-1] ; overhead
 sub eax, ecx
 lea edi, [edx*8+eax]
 mov al, 0abh
 mov cl, 8
 repe scasb
 je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 xor ebx, ebx
```

```
 call GetVersion
 cmp al, 6
 sbb rbp, rbp
 jb l1                    ; Process Environment Block
 mov rax, gs:[rbx+60h]
 mov eax, [rax+30h]       ; get process heap base
 mov ecx, [rax+40h]       ; check for protected heap
 jrcxz l1
 mov ecx, [rcx+8]
 test [rax+7ch], ecx
 cmovne ebx, [rax+88h] ; conditionally get heap key
l1: mov eax, <heap ptr>
 movzx edx, w [rax-8]    ; size
 xor dx, bx
 add edx, edx
 movzx ecx, b [rax+rbp-1] ; overhead
 sub eax, ecx
 lea edi, [rdx*8+rax]
 mov al, 0abh
 mov cl, 10h
 repe scasb
 je being_debugged
```

64비트 힙이 32비트 힙 함수로 분석될 수 없기 때문에 64비트 윈도우 환경을 검사하기 위한 동등한 32비트 코드는 없다.

만약 아무 포인터도 알려지지 않았다면 kernel32 HeapWalk() 또는 ntdll RtlWalkHeap() 함수를(또는 심지어 kernel32 GetCommandLine() 함수) 사용함으로써 검색될 수 있다. 반환된 블록 크기 값은 자동적으로 디코드되며 윈도우의 버전은 이 경우 더 이상 문제가 되지 않는다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
 mov ebx, offset l2       ; get a pointer to a heap block
l1: push ebx
 mov eax, fs:[30h]        ; Process Environment Block
 push d [eax+18h]         ; save process heap base
 call HeapWalk
 cmp w [ebx+0ah], 4       ; find allocated block
 jne l1
```

```
 mov edi, [ebx]          ; data pointer
 add edi, [ebx+4]        ; data size
 mov al, 0abh
 push 8
 pop ecx
 repe scasb
 je being_debugged
 ...
l2: db 1ch dup (0)       ; sizeof(PROCESS_HEAP_ENTRY)
```

또는 64비트 윈도우 환경을 조사하기 위한 이 64비트 코드를 사용할 수 있다.

```
 mov rbx, offset l2      ; get a pointer to a heap block
l1: push rbx
 pop rdx
 push 60h
 pop rsi
 gs:lodsq                ; Process Environment Block
                         ; get a pointer to process heap base
 mov ecx, [rax+30h]
 call HeapWalk
 cmpw [rbx+0eh], 4       ; find allocated block
 jne l1
 mov edi, [rbx]          ; data pointer
 add edi, [rbx+8]        ; data size
 mov al, 0abh
 push 10h
 pop rcx
 repe scasb
 je being_debugged
 ...
l2: db 28h dup (0)       ; sizeof(PROCESS_HEAP_ENTRY)
```

64비트 힙이 32비트 힙 함수로 분석될 수 없기 때문에 64비트 윈도우 환경을 검사하기 위한 동등한 32비트 코드는 존재하지 않는다.

4. Thread Local Storage

십 년도 전부터 알려져 있지만 아직까지도 사용할 새로운 방식이 등장하고 있기 때문에 Thread Local Storage는 존재하는 안티 디버깅 기법 중 가장 흥미롭다. TLS는 스레드가 시작하기 전에 스레드 specific 데이터를 초기화하기 위해 존재한다. 모든 프로세스가 적어도 하나의 스레드를 포함하지만 이 행위는 main 스레드가 실행되기 전에 데이터를 초기화하는 능력을 포함한다. 초기화는 메모리 내용을 동적으로 초기화하기 위해서 동적으로 할당된 메모리에서 복사된 정적 버퍼를 명시함으로써 수행된다. 이것은 가장 자주 남용되는 콜백 배열이다.

TLS 콜백 배열은 런타임시에 바뀌고(이후 엔트리들이 수정될 수 있다) 확장(새로운 엔트리들이 덧붙여질 수 있다)될 수 있다. 새로 추가된 또는 수정된 콜백들은 새로운 주소들을 사용하며 호출될 것이다. 배치될 수 있는 콜백들의 수는 제한되어 있지 않다. 확장은 32비트 또는 64비트 윈도우 버전에서 이 코드를 사용하여 만들어질 수 있다(32비트와 64비트에서 같다).

```
l1: mov d [offset cbEnd], offset l2
 ret
l2: ...
```

l2의 콜백은 l1의 콜백이 반환하면 호출될 것이다.

TLS 콜백 주소들은 예를들면 새로 로드된 DLL들 같이 이미지의 외부를 가리킬 수 있다. 이것은 DLL을 로드하고 반환된 주소를 TLS 콜백 배열로 배치함으로써 간적적으로 수행될 수 있다. 또한 DLL의 로딩 주소가 알려졌다면 직접적으로 수행될 수도 있다. 만약 DLL이 Data Execution Prevention을 공격할 방법으로 구조화되었다면 이것이 활성화된 경우에 ImageBase 값은 콜백 주소로서 사용될 수 있으며 또는 유효한 익스포트 주소가 검색되고 사용될 수도 있다. 이것을 호출하는 것은 32비트 또는 64비트 윈도우 버전에서 이 32비트 코드를 사용하여 만들어질 수 있다.

```
l1: push offset l2
 call LoadLibraryA
 mov [offset cbEnd], eax
 ret
l2: db "myfile", 0
```

또는 64비트 버전의 윈도우에서 이 64비트 코드를 사용할 수 있다.

```
 mov rcx, offset l2
 call LoadLibraryA
 mov [offset cbEnd], rax
```

```
 ret
l2: db "myfile", 0
```

이 경우 l1의 콜백이 리턴했을 때 "tls2.dll"라는 이름의 파일의 "MZ" 헤더는 실행될 것이다. 대안으로 파일은 32비트 또는 64비트 윈도우 버전에서 이 32비트 코드를 사용함으로써 자신을 참조할 수 있다.

```
l1: push 0
 call GetModuleHandleA
 mov [offset cbEnd], eax
 ret
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
l1: xor ecx, ecx
 call GetModuleHandleA
 mov [offset cbEnd], rax
 ret
```

이 경우 현재 프로세스의 "MZ" 헤더는 l1의 콜백이 반환했을 때 샐행될 것이다.

임포트 주소 테이블이 콜백 배열을 가리키게 변경되면 TLS 콜백 주소들은 다른 DLL들에서 임포트된 주소들의 RVA들을 포함한다. 임포트들은 콜백들이 호출되기 전에 resolved되서 임포트된 함수들은 콜백 배열 엔트리에 도달하면 정상적으로 호출될 것이다.

TLS 콜백들은 함수들에서 직접적으로 넘겨진 세 스택 파라미터를 받는다. 첫 번째 파라미터는 호스트 프로세스의 ImageBase이다. 예를들면 이것은 kernel32LoadLibrary() 함수 또는 kernel32 WinExec() 함수에 의해 사용될 수 있다. ImageBase 파라미터는 kernel32 LoadLibrary() 또는 kernel32 WinExec() 함수에 의해서 로드 또는 실행할 파일 이름에 대한 포인터로 해석될 것이다. TLS 콜백은 "MZ[some string]"("[some string]"은 호스트 파일 헤더 내용들을 매치한다)로 불리는 파일을 생성함으로써 명시적인 참조 없이 파일에 접근할 것이다. 물론 문자열의 "MZ"부분은 또한 런타임 시에 manually 대체될 수 있지만 많은 함수들은 이 시그니처에 의존해서 이 변화의 결과는 예측할 수 없다.

프로세스가 kernel32 DisableThreadLibraryCalls()
나 ntdll LdrDisableThreadCalloutsForDll() 함수를 호출하지 않는 한 TLS 콜백들은 스레드가 생성되거나 소멸될 때마다 호출된다. 윈도우 XP와 그 이전에서는 디버거가 프로세스를 어태치할 때 생성된 스레드를 포함한다(윈도우 비스타와 그 이후에서는 TLS 핸들러에 디버그 스레드 이벤트를 전달하지 않는다). 디버거 스레드는 이것의 엔트리포인트가 이미지 내부를 가

리키지 않는다는 점에서 특별하다. 대신 이것은 윈도우 2000과 그 이전에서 kernel32.dll 내부를 가리키며 윈도우 XP에서는 0을 가리킨다. 그러므로 간단한 디버거 탐지 방법은 생성된 각 스레드의 시작 주소를 질의하기 위해서 TLS 콜백을 사용하는 것이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
push eax
mov eax, esp
push 0
push 4
push eax          ; ThreadQuerySetWin32StartAddress
push 9
push -2           ; GetCurrentThread()
call NtQueryInformationThread
pop eax
test eax, eax
je being_debugged      ; Windows XP
cmp eax, offset l1
jnb being_debugged     ; Windows NT/2000
...
l1: <code end>
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
xor ebp, ebp
enter 20h, 0
push 4
pop r9
push rbp
pop r8            ; ThreadQuerySetWin32StartAddress
push 9
pop rdx
push -2           ; GetCurrentThread()
pop rcx
call NtQueryInformationThread
leave
test rbp, rbp
je being_debugged
...
```

l1: <code end>

TLS 콜백들은 디버거가 제어를 얻기 전에 실행될 수 있으므로 콜백은 일반적으로 엔트리포인트 호스트에 위치한 브레이크 포인트를 삭제하는 것 같은 다른 변화를 만들 수 있다. 패치는 32비트 또는 64비트 윈도우 버전에서 이 코드를 사용해 만들어질 수 있다(32비트와 64비트에서 같다).

```
;<val> is byte at l1
mov b [offset l1], <val>
ret
l1: <host entrypoint>
```

이 기법에 대한 방어는 매우 쉬우며 필요성이 증가하고 있다. 이것은 호스트 엔트리포인트 대신 첫 번째 TLS 콜백의 첫 번째 바이트에 브레이크 포인트를 삽입하는 문제이다. 이것은 프로세스에서 실행될 수 있는 모든 코드 이전에 디버거가 제어를 얻을 수 있게 한다(물론 로드된 DLL들은 제외한다). 그러나 위에서 언급했듯이 주소의 원래 값이 임포트된 함수의 RVA일 수 있기 때문에 콜백 주소와 관련된 것은 주의를 필요로 한다. 그래서 그 주소는 파일에서 읽어질 수 없다. 이것은 반드시 이미지 메모리에서 읽어져야 한다.

TLS 콜백의 실행은 또한 플랫폼에 구체적이다. 만약 실행 가능한 임포트들이 단지 ntdll.dll 또는 kernel32.dll에서 온 것이라면 윈도우 XP와 이후 버전에서 콜백들은 "어태치 된" 동안 호출되지 않을 것이다. 프로세스가 시작할 때 ntdll LdrInitializeThunk() 함수는 InLoadOrderModuleList 리스트를 처리한다. InLoadOrderModuleList 리스트는 프로세스에 대한 DLL들의 리스트를 포함한다. 참조된 구조체에서 Flags 값은 언태치될 때 호출되는 TLS 콜백들을 위한 적어도 하나의 DLL에서 반드시 LDRP_ENTRY_PROCESSED 비트를 클리어 해야 한다.

그 비트는 항상 ntdll.dll을 위해 설정되어 있어서 단지 ntdll.dll에서의 파일 임포팅은 어태치 시에 TLS 콜백들을 실행되게 하지 않을 것이다. 윈도우 2000과 그 이전에서는 파일이 kernel32.dll에서 명시적(즉, kernel32.dll에서 직접적으로 임포팅하는)으로나 암묵적(즉 kernel32.dll에서 임포트하는 DLL에서 임포트하는 또는 체인이 얼마나 긴지와 관계없이 kernel32.dll에서 임포트하는 DLL에서)으로 임포트되지 않았으면 크래시 버그를 갖는다.

이 버그는 윈도우 XP에서 호스트 임포트 테이블을 처리하기 전에 ntdll.dll이 명시적으로 kernel32.dll을 로드하게 강제함으로써 고쳐졌다. kernel32.dll이 로드되면 InLoadOrderModuleList에 추가된다. 문제는 이 수정이 부작용을 낳았다는 것이다.

부작용은 ntdll.dll이 LdrGetProcedureAddressEx() 함수를 통해 익스포트된 함수 주소를 kernel32.dll에서 검색할 때 발생한다. 부작용은 익스포트된 함수의 검색의 결과로 인해 유

발되지만 다음의 함수들 중 하나의 주소를 검색하는 ntdll에 의한 이 특정한 경우에 유발된다 : BaseProcessInitPostImport() (윈도우 XP 그리고 윈도우 서버 2003에서), BaseQueryModuleData() (BaseProcessInitPostImport() 함수가 존재하지 않는다면 윈도우 XP 그리고 윈도우 서버 2003 에서), BaseThreadInitThunk() (윈도우 비스타 그리고 이후 버전들) 또는 BaseQueryModuleData() (BaseThreadInitThunk()이 존재하지 않는다면 윈도우 비스타 그리고 이후 버전들에서).

부작용은 ntdll LdrGetProcedureAddressEx() 함수가 InLoadOrderModuleList 리스트에서의 kernel32.dll엔트리를 위해 LDRP_ENTRY_PROCESSED 플래그를 설정하는 것이다. 결과적으로 오직 kernel32.dll에서의 파일 임포팅은 더 이상 어태치된 TLS 콜백들을 실행되게 하지 않는다. 이것은 윈도우의 버그로 고려될만 하다.

문제에 대한 제 2의 해결책으로 DLL이 0이 아닌 엔트리포인트를 갖을 때 다른 DLL에서 몇몇을 임포트하는 것이 있다. 그러면 TLS 콜백들은 어태치된 채 실행될 수 있다. 제 2의 해결책은 Flags 필드 값 때문에 DLL의 LDRP_ENTRY_PROCESSED 비트를 클리어할 것이다.

윈도우 비스타와 이후에서 동적으로 로드된 DLL들 또한 TLS를 지원한다. 이것은 "정적으로 선언된 TLS 데이터 객체들" 다시 말해서 "오직 정적으로 로드된 이미지 파일에서만 사용될 수 있다"라고 언급하는 Portable Executable Format Documentation에 직접적으로 모순된다. 이 사실은 당신이 이 DLL(또는 정적으로 이것과 링크된 것)이 LoadLibrary API 함수로 동적으로 로드되지 않을 것이라는 걸 알지 않는 이상 DLL에서 정적 TLS 데이터를 사용하는 것을 신뢰할 수 없게 만든다. 더 나아가 TLS 콜백들은 임포트 테이블에 무엇이 존재하는 간에 호출될 것이다. 그러므로 DLL은 ntdll.dll에서 또는 kernel32.dll에서 또는 심지어 전혀 DLL이 존재하지 않은 경우에도 임포트할 수 있으며 콜백들은 호출될 것이다.

5. AntiStepOver

대부분의 디버거들은 "call"과 "rep" sequence 같은 특정한 명령어들을 step over하는 것을 지원한다. 이 경우에 보통 소프트웨어 브레이크포인트가 명령어 흐름 가운데 위치하며 프로세스는 실행을 재개할 수 있다. 소프트웨어 브레이크포인트에 도달되면 디버거는 일반적으로 다시 제어를 받는다. 그러나 "rep" sequence의 경우 디버거는 반드시 rep 접두어 이후의 명령어를 확인해야 한다. 몇몇 디버거들은 어떤 rep 접두어라도 문자열 명령어에 앞선다고 가정한다. 이것은 rep 접두어 이후의 명령어가 전체적으로 다른 명령어일 때 취약점을 발생시킨다. 구체적으로 말해서 문제는 만약 그 명령어가 흐름에 위치한 소프트웨어 브레이크포인트를 제거할 때 생긴다. 이 경우 명령어가 step over하면 그리고 소프트웨어 브레이크포인트가 명령어에 의해 제거되면 실행은 프로세스의 완전한 제어 하에 재개되며 디버거로 반환되지 않는다. 에제 코드는 다음과 같다.

```
 rep
l1: mov b [offset l1], 90h
l2: nop
```

만약 stepover가 l1에서 시도되면 실행은 l2에서 자유롭게 재개될 것이다.

더 일반적인 방법은 브레이크포인트를 제거하기 위해 문자열 명령어들을 사용한다. 이 패치는 32비트 또는 64비트 윈도우 버전에서 다음의 32비트 코드를 사용해 수행될 수 있다.

```
 mov al, 90h
 xor ecx, ecx
 inc ecx
 mov edi, offset l1
 rep stosb
l1: nop
```

또는 64비트 버전의 윈도우에서 이 64비트 코드를 사용할 수 있다.

```
 mov al, 90h
 xor ecx, ecx
 inc ecx
 mov rdi, offset l1
 rep stosb
l1: nop
```

"rep stos" 대신 "rep movs"를 사용하는 것 같은 이 기법의 변형도 존재한다. direction flag는 메모리 쓰기의 방향을 역으로하기 위해 사용될 수 있어서 overwrite는 간과될 수 있다. 패치는 32비트 또는 64비트 윈도우 버전에서 이 32비트 코드를 사용해서 만들어질 수 있다.

```
 mov al, 90h
 push 2
 pop ecx
 mov edi, offset l1
 std
 rep stosb
 nop
l1: nop
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
 mov al, 90h
 push 2
 pop rcx
 mov rdi, offset l1
 std
 rep stosb
 nop
l1: nop
```

이 문제의 해결책은 문자열 명령어들의 stepover 동안에 하드웨어 브레이크포인트를 사용하는 것이다. 이것은 위치한 브레이크포인트가 실행될 브레이크포인트인 경우 디버거가 알 수 없다고 여겨질 때 특히 중요하다. 만약 프로세스가 브레이크포인트를 제거하면 이것은 또한 이후에 브레이크포인트를 복구하고 평소와 같이 브레이크포인트를 실행할 수 있다. 디버거는 예상되는 브레이크포인트 예외를 볼 것이고 평소처럼 행동할 것이다. 이 기법은 32비트 또는 64비트 윈도우 버전에서 이 32비트 코드를 사용해 만들어질 수 있다.

```
 mov al, 90h
l1: xor ecx, ecx
 inc ecx
 mov edi, offset l3
l2: rep stosb
l3: nop
 cmp al, 0cch
l4: mov al, 0cch
 jne l1
l5: ...
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
 mov al, 90h
l1: xor ecx, ecx
 inc ecx
 mov rdi, offset l3
l2: rep stosb
l3: nop
 cmp al, 0cch
l4: mov al, 0cch
```

```
 jne l1
l5: ...
```

이 예시에서 l2 라인에서 명령어를 step over하는 것은 코드가 l4까지 닿게할 것이고 l1로 반환할 것이다. 이것은 브레이크포인트를 두 번째 pass의 l2로 대체되게 할 것이고 l3에 의해 실행된다. 그 때 명백한 차이점은 AL 레지스터가 예상되는 0x90 대신 값 0xCC를 hold할 것이라는 것이다. 이것은 2 pass 대신 1 pass로 보이는 곳으로 l5가 도달하게 할 것이다. 물론 전체적으로 다른 codepaths를 포함하는 가능한 미묘한 변형들이 많이 존재한다.

이 기법의 변형은 간단하게 디버거의 존재를 탐지하기 위해 사용될 수 있다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 아래의 32비트 코드를 사용해 수행될 수 있다.

```
 xor ecx, ecx
 inc ecx
 mov esi, offset l1
 lea edi, [esi + 1]
 rep movsb
l1: mov al, 90h
l2: cmp al, 0cch
 je being_debugged
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
 xor ecx, ecx
 inc ecx
 mov rsi, offset l1
 lea rdi, [esi + 1]
 rep movsb
l1: mov al, 90h
l2: cmp al, 0cch
 je being_debugged
```

이 코드는 l1에 위치한 브레이크포인트를 탐지할 것이다. 이것은 l1의 값을 l1+1의 "90h"로 복사함으로써 수행된다. 값은 그 후 l2와 비교된다.

6. Hardware

A. Hardware breakpoints

예외가 발생할 때 윈도우는 예외 핸들러로 보낼 context structure를 생성한다. 구조체는 범용 레지스터들, selector들, 제어 레지스터들 그리고 디버그 레지스터들의 값을 포함할 것이다. 만약 디버거가 존재하고 예외를 사용된 하드웨어 브레이크포인트들과 함께 디버기에게 보내면 디버그 레지스터들은 디버거의 존재를 드러낼 값들을 포함할 것이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 이 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor eax, eax
 push offset l1
 push d fs:[eax]
 mov fs:[eax], esp
 int 3                  ; force an exception to occur
 ...
l1:                     ; execution resumes here when exception occurs
 mov eax, [esp+0ch]     ; get ContextRecord
 mov ecx, [eax+4]       ; Dr0
 or ecx, [eax+8]        ; Dr1
 or ecx, [eax+0ch]      ; Dr2
 or ecx, [eax+10h]      ; Dr3
 jne being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 mov rdx, offset l1
 xor ecx, ecx
 inc ecx
 call AddVectoredExceptionHandler
 int 3                  ; force an exception to occur
 ...
l1:                     ; execution resumes here when exception occurs
 mov rax, [rcx+8]       ; get ContextRecord
 mov rcx, [rax+48h]     ; Dr0
 or rcx, [rax+50h]      ; Dr1
 or rcx, [rax+58h]      ; Dr2
 or rcx, [rax+60h]      ; Dr3
 jne being_debugged
```

32비트 버전의 윈도우에서 디버그 레지스터들을 위한 값들은 또한 실행을 재개하기 이전

에 바뀌어질 수 있으며 소프트웨어 브레이크포인트가 적절한 곳에 위치하지 않는 한 이것은 제어되지 않은 실행을 야기한다.

B. Instruction Counting

Instruction Counting은 예외 핸들러를 등록하고 그 후 하드웨어 브레이크포인트들을 특정한 주소들에 설정함으로써 수행될 수 있다. 상응하는 주소가 hit되면 EXCEPTION_SINGLE_STEP (0x80000004) 예외가 유발될 것이다. 이 예외는 예외 핸들러로 전달되며 예외 핸들러는 명령어 포인터가 새로운 명령어를 가리키게 바로잡도록 선택할 수 있다. 그리고 마음대로 추가적인 하드웨어 브레이크포인트들을 특정한 주소에 설정하고 실행을 재개할 수 있다. 브레이크포인트들을 설정하는 것은 context structure에 대한 접근을 필요로 한다. context structure의 복사본은 필요한 경우 하드웨어 브레이크포인트들을 위해 초기 값들을 설정하게 해주는 kernel32 GetThreadContext() 함수를 호출할 때 요구되어질 수 있다. 그 후 예외가 발생했을 때 예외 핸들러는 자동적으로 context structure의 사본을 받을 것이다. 디버거는 디버거가 존재하지 않을 경우와 비교해서 다른 명령어들의 count를 야기하며 singlestepping을 방해할 것이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor eax, eax
 push offset l5
 push d fs:[eax]
 mov fs:[eax], esp
 int 3                    ; force exception to occur
l1: nop
l2: nop
l3: nop
l4: nop
 cmp al, 4
 jne being_debugged
 ...
l5: push edi
 mov eax, [esp+8]         ; ExceptionRecord
 mov edi, [esp+10h]       ; ContextRecord
 push 55h                 ; localenable DR0, DR1, DR2, DR3
 pop ecx
 incd [ecx*2+edi+0eh]     ; Eip
 mov eax, [eax]           ; ExceptionCode
 sub eax, 80000003h       ; EXCEPTION_BREAKPOINT
```

```
jne l6
mov eax, offset l1
scasd
stosd                   ; Dr0
inc eax                 ; l2
stosd                   ; Dr1
inc eax                 ; l2
stosd                   ; Dr2
inc eax                 ; l4
stosd                   ; Dr3
                        ; localenable breakpoints
                        ; for compatibility with old CPUs
mov ch, 1
xchg ecx, eax
scasd  stosd            ; Dr7
xor eax, eax
pop edi
ret
l6: dec eax             ; EXCEPTION_SINGLE_STEP
jne being_debugged
inc b [ecx*2+edi+6]     ; Eax
pop edi
ret
```

이 기법이 Structured Exception Handler를 사용하기 때문에 64비트 버전의 윈도우에서는 사용될 수 없다. 대신 Vectored Exception Handler를 사용하는 코드는 쉽게 재작성 될 수 있다. 디버그 레지스터들이 64비트 버전의 윈도우 Vectored Exception Handler의 내부에서 assign될 수 없기 때문에 스레드를 생성하고 이것의 context를 바꾸는 것을 요구된다.

32비트 또는 64비트 버전의 윈도우 XP와 그 이후에서 이것을 검사하는 것은 이 32비트 코드를 사용해 만들어질 수 있다.

```
xor ebx, ebx
push eax
push esp
push 4                  ; CREATE_SUSPENDED
push ebx
push offset l1
push ebx
```

```
 push ebx
 call CreateThread
 mov esi, offset l7
 push esi
 push eax
 xchg ebp, eax
 call GetThreadContext
 mov eax, offset l2
 lea edi, [esi+4]
 stosd                  ; Dr0
 inc eax
 stosd                  ; Dr1
 inc eax
 stosd                  ; Dr2
 inc eax
 stosd                  ; Dr3
 scasd
 push 55h               ; localenable DR0, DR1, DR2, DR3
 pop eax
 stosd                  ; Dr7
 push esi
 push ebp
 call SetThreadContext
 push offset l6
 push 1
 call AddVectoredExceptionHandler
 push ebp
 call ResumeThread
 jmp $
l1: xor eax, eax
l2: nop
l3: nop
l4: nop
l5: nop
 cmp al, 4
 jne being_debugged
 ...
l6: mov eax, [esp+4]
 mov ecx, [eax]         ; ExceptionRecord
```

```
                        ; ExceptionCode
cmp [ecx], 80000004h    ; EXCEPTION_SINGLE_STEP
jne being_debugged
mov eax, [eax+4]        ; ContextRecord
cdq
mov dh, 1
inc b [eax+edx50h]      ; Eax
inc d [eax+edx48h]      ; Eip
or eax, -1              ; EXCEPTION_CONTINUE_EXECUTION
ret
l7: dd 10002h           ; CONTEXT_i486+CONTEXT_INTEGER
db 0b0h dup (?)
```

또는 64비트 버전의 윈도우에서 이 64비트 코드를 사용할 수 있다.

```
push rax
push rsp
push 4                  ; CREATE_SUSPENDED
sub esp, 20h
xor r9d, r9d
mov r8, offset l1
xor edx, edx
xor ecx, ecx
call CreateThread
mov ebp, eax
mov rsi, offset l7-30h
push rsi
pop rdx
xchg ecx, eax
call GetThreadContext
mov rax, offset l2
lea rdi, [rsi+48h]
stosq                   ; Dr0
inc rax
stosq                   ; Dr1
inc rax
stosq                   ; Dr2
inc rax
stosq                   ; Dr3
```

```
 scasq
 push 55h                ; localenable DR0, DR1, DR2, DR3
 pop rax
 stosd                   ; Dr7
 push rsi
 pop rdx
 mov ecx, ebp
 call SetThreadContext
 mov rdx, offset l6
 xor ecx, ecx
 inc ecx
 call AddVectoredExceptionHandler
 mov ecx, ebp
 call ResumeThread
 jmp $
l1: xor eax, eax
l2: nop
l3: nop
l4: nop
l5: nop
 cmp al, 4
 jne being_debugged
 ...
l6: mov rax, [rcx]       ; ExceptionRecord
                         ; ExceptionCode
 cmp d [rax], 80000004h ; EXCEPTION_SINGLE_STEP
 jne being_debugged
 mov rax, [rcx+8]        ; ContextRecord
 inc b [rax+78h]                 ; Eax
 inc q [rax+0f8h]        ; Eip
 or eax, -1              ; EXCEPTION_CONTINUE_EXECUTION
 ret
l7: dd 10002h            ; CONTEXT_i486+CONTEXT_INTEGER
 db 0c4h dup (?)
```

C. Interrupt 3

소프트웨어 예외가 발생할 때마다 예외 주소와 EIP 레지스터 값은 예외를 일으킨 명령어

의 다음 명령어를 가리킬 것이다. 브레이크포인트 예외는 특별한 경우로 다뤄진다. EXCEPTION_BREAKPOINT (0x80000003) 예외가 발생하면, 윈도우는 이것이 한 바이트 "CC" 옵코드 ("INT 3" 명령어)에 의해 유발되었다고 여긴다. 윈도우는 가정된 "CC" 옵코드를 가리키기 위해 예외 주소를 감소시키고 예외를 예외 핸들러에게 보낸다. EIP 레지스터 값은 영향을 받지 않는다. 그러므로 만약 "CD 03" 옵코드("INT 03"의 긴 형태)가 사용되면 예외 주소는 예외 핸들러가 제어를 받을 때 "03"을 가리킬 것이다.

D. Interrupt 0x2d

인터럽트 0x2D는 특별한 경우이다. 이것이 실행되면 윈도우는 현재 EIP 레지스터 값을 예외 주소로 사용하고 이후 EIP 레지스터 값을 증가시킨다. 그러나 윈도우는 또한 어떻게 예외 주소를 조정할지를 결정하기 위해 EAX 레지스터에서 값을 검사한다. 모든 윈도우 버전에서 만약 EAX 레지스터가 1, 3 또는 4 값을 가지면 또는 윈도우 비스타 이후에서 값 5를 가지면 윈도우는 예외 주소를 증가시킬 것이다. 마지막으로 이것은 디버거가 존재하는 경우 EXCEPTION_BREAKPOINT (0x80000003) 예외를 발생시킨다. 인터럽트 0x2D의 행위는 디버거들에게 문제를 일으킬 수 있다. 문제는 다른 디버거들이 재개할 주소로 예외 주소를 사용하는 반면 몇몇 디버거들이 재개할 주소로 EIP 레지스터 값을 사용한다는 것이다. 이것은 스킵되는 단일 바이트 명령어를 초래하거나 첫 번째 바이트가 놓쳐지기 때문에 완전히 다른 명령어의 실행을 초래한다. 이러한 행동들은 디버거의 존재를 추론하기 위해 사용될 수 있다. 이 검사는 32비트 또는 64비트 윈도우 환경에서 다음의 코드(32비트와 64비트에서 동일하다)를 사용하여 만들어질 수 있다.

```
xor eax, eax            ; set Z flag
int 2dh
inc eax                 ; debugger might skip
je being_debugged
```

E. Interrupt 0x41

커널 모드 디버거의 존재에 따라 인터럽트 0x41은 다른 행동을 보여준다. 인터럽트 0x41 디스크립터는 보통 인터럽트가 ring 3에서 성공적으로 실행될 수 없다는 것을 의미하는 0의 DPL을 갖는다. 이 인터럽트를 직접적으로 실행하려는 시도는 CPU에 의한 general protection fault(interrupt 0x0D)를 초래하며, 결국 EXCEPTION_ACCESS_VIOLATION (0xC0000005) 예외를 초래하게 된다. 그러나 몇몇 디버거들은 인터럽트 0x41을 후킹하고 이것의 DPL을 3으로 조정해서 인터럽트가 사용자 모드에서 성공적으로 호출되게 한다. 이 사실은 커널 모드 디버거의 존재를 추론하기 위해 사용될 수 있다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경

을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 mov rdx, offset l1
 xor ecx, ecx
 inc ecx
 call AddVectoredExceptionHandler
 push 4fh
 pop rax
 int 41h
 jmp being_debugged
l1:                          ; execution resumes here if no debugger present
  ...
```

F. MOV SS

Intel CPUs의 초기부터 사용되어 온 singlestepping을 탐지하는 간단한 트릭이 존재한다. 이것은 DOS 시절부터 상당히 일반적으로 사용되어왔지만 아직도 윈도우의 모든 버전에서 통하고 있다. 트릭은 특정한 명령어들이 다음 명령어를 실행하는 동안 모든 인터럽트들
을 disabled되게 한다는 사실을 기반으로 한다. 특히 SS 레지스터를 로딩하는 것은 다음 명령어가 스택 오염의 위험 없이 [E]SP 레지스터를 로드하게 허용하기 위해 인터럽트들을 clear한다. 그러나 다음 명령어가 [E]SP 레지스터에 어떤 것을 로드하는 요구사항은 없다. 어떤 명령어도 SS 레지스터의 로드 이후에 따라올 수 있다. 만약 디버거가 코드에서 singlestep을 사용하는데 사용되고 있다면 T 플래그는 EFLAGS 이미지에 설정될 것이다. 각 디버거 이벤트가 전달된 이후 T 플래그가 EFLAGS 이미지에서 clear되기 때문에 이것은 일반적으로 보이지 않는다. 그러나 디버거 이벤트가 전달되기 전에 만약 플래그들이 스택에 저장되면 T 플래그는 보이게 될 것이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push ss
pop ss
pushfd
test b [esp+1], 1
jne being_debugged
```

VirtualPC에서 CPUID 명령어가 같은 방식으로 동작하는 윈도우 2000을 돌릴 때 흥미로운 상황이 존재한다. 왜 이것이 발생하는지는 알려지지 않았다.

SS Selctor이 64비트 환경에서는 지원되지 않기 때문에 64비트 코드 예제는 존재하지 않는

다.

7. APIs

디버거는 표준 운영체제 함수들을 사용함으로써 탐지되고 disabled 되고 회피(디버기에 한 제어를 잃는)될 수 있다. 함수들은 일반적인 기능에 기초한 여러 그룹들을 호출한다.

A. Heap functions

BasepFreeActivationContextActivationBlock
BasepFreeAppCompatData
ConvertFiberToThread
DeleteFiber
FindVolumeClose
FindVolumeMountPointClose
HeapFree SortCloseHandle

이러한 함수들 모두의 공통점은 이것들이 ntdll RtlFreeHeap() 함수를 호출한다는 것이다.

kernel32 BasepFreeActivationContextActivationBlock()과 kernel32 SortCloseHandle() 함수는 윈도우 7과 그 이후부터 존재한다. kernel32 BasepFreeAppCompatData()는 특별한 경우로서 ntdll RtlFreeHeap() 함수를 호출한다(구체적으로 말해서 hFindVolumeMountPoint 파라미터가 성공적으로 닫힐 수 있는 핸들에의 유효한 포인터일 경우).

그러나 중요한 것은 ntdll RtlFreeHeap() 함수가 디버거와 같이 사용되도록 설계되었다는 특징을 갖는다(ntdll DbgPrint() 함수로의 호출)는 점이다. 문제는 ntdll DbgPrint() 함수가 구현된 방식이며 이것은 함수가 호출될 때 애플리케이션이 디버거의 존재를 탐지하도록 해준다.

ntdll DbgPrint() 함수가 호출되면 이것은 DBG_PRINTEXCEPTION_C (0x40010006) 예외를 일으키지만 예외는 특별한 경우에 다뤄지며 등록된 SEH는 이것을 보지 못할 것이다. 이유는 윈도우가 디버거가 처리하지 않을 시에 예외를 처리하는 자신의 SEH를 내부적으로 등록하기 때문이다. 그러나 윈도우 XP와 그 이후에서는 어떤 등록된 VEH도 윈도우가 등록한 SEH 이전에 실행될 수 있다. 이것은 윈도우에서의 버그로 여겨진다. 예외를 처리하는 디버거의 존재는 이제 예외의 부재를 통해 추론될 수 있다. 더 나아가 디버거가 존재하지만 그 예외를 처리할 수 없거나 디버거가 전혀 존재하지 않은 경우 다른 예외가 VEH로 전달된다. 만약 디버거가 존재하지만 예외를 처리할 수 없다면 윈도우는 DBG_PRINTEXCEPTION_C (0x40010006) 예

외를 전달할 것이다. 만약 디버거가 존재하지 않는다면 윈도우는 EXCEPTION_ACCESS_VIOLATION (0xC0000005) 예외를 전달할 것이다. 디버거의 존재는 이제 예외의 부재 또는 예외의 값에 의해 추론될 수 있다.

다른 함수들 중에서 힙과 리소스 함수들에 적용되는 디버그 브레이크를 일으키게 하는 추가적인 경우가 존재한다. 그것들이 공통으로 갖는것은 PEB의 BeingDebugged 플래그를 검사하는 것이다. 디버거의 존재 여부는 인터럽트 3 예외를 일으키게 해서 속일 수 있으며 예외는 디버기에서 보여야 한다. 그러므로 만약 예외를 놓치면 디버거의 존재는 드러난다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor eax, eax
 push offset l1
 push d fs:[eax]
 mov fs:[eax], esp        ; Process Environment Block
 mov eax, fs:[eax+30h]
 inc b [eax+2]            ; set BeingDebugged
 push offset l2
 call HeapDestroy
 jmp being_debugged
l1:                       ; execution resumes here due to exception
 ...
l2: db 0ch dup (0)
 dd 40000000h             ; HEAP_VALIDATE_PARAMETERS_ENABLED
 db 30h dup (0)
 dd 40000000h             ; HEAP_VALIDATE_PARAMETERS_ENABLED
 db 24h dup (0)
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 mov rdx, offset l1
 xor ecx, ecx
 inc ecx
 call AddVectoredExceptionHandler
 push 60h
 pop rsi
 gs:lodsq                 ; Process Environment Block
 inc b [rax+2]            ; set BeingDebugged
 mov rcx, offset l2
```

```
 call HeapDestroy
 jmp being_debugged
l1:                     ; execution resumes here due to exception
 ...
l2: db 14h dup (0)
 dd 40000000h           ; HEAP_VALIDATE_PARAMETERS_ENABLED
 db 58h dup (0)
 dd 40000000h           ; HEAP_VALIDATE_PARAMETERS_ENABLED
 db 30h dup (0)
```

윈도우의 버전에 따라 가능한 플래그 필드의 위치에 둘다 위치하기 때문에 플래그 값은 각 경우에서 두 번 보인다. 이것은 버전을 검사할 필요를 줄여준다.

윈도우 비스타와 그 이후의 버전들에 대해 언급하자면 행위가 약간 바뀌었다. 전에는 디버그 브레이크가 ntdll DbgBreakPoint()에 한 호출에 의해 유발되었다. 지금은 코드 흐름에 직접적으로 저장된 인터럽트 3 명령어에 의해 유발된다. 결과는 각 경우에 같다.

탐지 또한 약간 확장될 수 있다. TEB의 LastErrorValue는 직접적으로 또는 kernel32 SetLastError() 함수를 호출함으로써 모두 함수를 호출하기 전에 0으로 설정될 수 있다. 만약 예외가 일어나지 않는다면 함수의 반환에서 (또한 kernel32 GetLastError() 함수에 의해 반환된) 필드의 값은 ERROR_INVALID_HANDLE (6)으로 설정될 것이다.

B. Handles

OpenProcess
CloseHandle
CreateFile
LoadLibrary
ReadFile

i. OpenProcess

kernel32 OpenProcess() 함수(또는 ntdll NtOpenProcess() 함수)는 "csrss.exe" 프로세스에서 사용될 때 가끔씩 디버거의 존재를 탐지하기 위해 사용된다. 이것은 정확하지 않다. 이것이 함수 호출이 몇몇 디버거들의 존재를 탐지하는데 성공할 수 있다는 점에서는 사실이지만 이 점은 디버거의 행동의 부작용(정확히 말하면 디버그 권한을 얻는)에 의한 것이지 디버거 자체(이것은 특정한 디버거를 사용할 때는 성공하지 못하기 때문에 명백하다)에 의한 것이 아니다. 이것이 드러내는 모든 것은 프로세스의 사용자 계정이 관리자 그룹의 멤버이고 디

버그 권한을 갖는다는 것이다. 이유는 이 함수 호출의 성공이나 실패는 단지 프로세스 권한 수준에 제한되기 때문이다. 만약 프로세스의 사용자 계정이 관리자 그룹의 멤버이고 디버그 권한을 갖는다면 함수 호출은 성공할 것이지만 아니라면 실패한다. 이것은 일반적인 사용자가 디버그 권한을 갖게 하는데 충분하지 않으며 관리자는 이것 없이도 함수를 성공적으로 호출할 수 있다. csrss.exe 프로세스의 프로세스 ID는 윈도우 XP와 그 이후에서 ntdll CsrGetProcessId()에 의해 획득될 수 있다(이전 버전의 윈도우들에서도 방법이 존재하며 "Explorer.exe" 프로세스를 찾는 context 이후에 보여진다). 32비트 또는 64비트 윈도우 버전에서 호출은 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
call CsrGetProcessId
push eax
push 0
push 1f0fffh                ; PROCESS_ALL_ACCESS
call OpenProcess
test eax, eax
jne admin_with_debug_priv
```

또는 64비트 버전의 윈도우에서 이 64비트 코드를 사용할 수 있다.

```
call CsrGetProcessId
push rax
pop r8
cdq
mov ecx, 1f0fffh            ; PROCESS_ALL_ACCESS
call OpenProcess
test eax, eax
jne admin_with_debug_priv
```

32비트 또는 64비트 윈도우 버전에서 디버그 권한은 다음의 32비트 코드를 사용해서 얻어질 수 있다.

```
xor ebx, ebx
push 2                      ; SE_PRIVILEGE_ENABLED
push ebx
push ebx
push esp
push offset l1
push ebx
call LookupPrivilegeValueA
```

```
 push eax
 push esp
 push 20h                ; TOKEN_ADJUST_PRIVILEGES
 push -1                 ; GetCurrentProcess()
 call OpenProcessToken
 pop ecx
 push eax
 mov eax, esp
 push ebx
 push ebx
 push ebx
 push eax
 push ebx
 push ecx
 call AdjustTokenPrivileges
 ...
l1: db "SeDebugPrivilege", 0
```

또는 64비트 버전의 윈도우에서 이 64비트 코드를 사용할 수 있다.

```
xor ebx, ebx
 push 2                  ; SE_PRIVILEGE_ENABLED
 push rbx
 push rbx
 mov r8d, esp
 mov rdx, offset l1
 xor ecx, ecx
 call LookupPrivilegeValueA
 push rax
 mov r8d, esp
 push 20h                ; TOKEN_ADJUST_PRIVILEGES
 pop rdx
 or rcx, -1              ; GetCurrentProcess()
 call OpenProcessToken
 pop rcx
 push rax
 mov r8d, esp
 push rbx
 push rbx
```

```
 sub esp, 20h
 xor r9d, r9d
 cdq
 call AdjustTokenPrivileges
 ...
l1: db "SeDebugPrivilege", 0
```

ii. CloseHandle

디버거를 탐지하는 잘 알려진 기법으로 kernel32 CloseHandle() 함수와 관련된 것이 있다. 만약 유효하지 않은 핸들이 kernel32 CloseHandle()로 (또는 직접적으로 ntdll NtClose()로 또는 윈도우 2000 이후에서 kernel32 FindVolumeMountPointClose() 함수로(이것은 간단하게 kernel32 CloseHandle() 함수를 호출한다)) 전달되고 동시에 디버거가 존재한다면 EXCEPTION_INVALID_HANDLE (0xC0000008) 예외가 발생할 것이다. 이 예외는 예외 핸들러에 의해 가로채질 수 있으며 디버거가 실행 중이라는 것을 말한다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
xor eax, eax
push offset being_debugged
push d fs:[eax]
mov fs:[eax], esp        ; any illegal value will do
                         ; must be dwordaligned
                         ; on Windows Vista and later
push esp
call CloseHandle
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
mov rdx, offset being_debugged
xor ecx, ecx
inc ecx
call AddVectoredExceptionHandler  ; any illegal value will do
                         ; must be dwordaligned
                         ; on Windows Vista and later
mov ecx, esp
call CloseHandle
```

그러나 대신 보호된 핸들을 사용하는 것과 관련된 두 번째 경우가 존재한다. 만약 보호된 핸

들이 kernel32 CloseHandle() 함수로(또는 직접적으로 NtClose() 함수로) 전달되고 디버거가 존재하면 EXCEPTION_HANDLE_NOT_CLOSABLE (0xC0000235) 예외가 발생할 것이다. 이 예외는 예외 핸들러에 의해서 가로채질 수 있으며 디버거가 돌아가는 중이라는 것을 의미한다. 32비트 또는 64비트 윈도우 환경에서 검사는 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
 xor eax, eax
 push offset being_debugged
 push d fs:[eax]
 mov fs:[eax], esp
 push eax
 push eax
 push 3                  ; OPEN_EXISTING
 push eax
 push eax
 push eax
 push offset l1
 call CreateFileA
 push 2                  ; HANDLE_FLAG_PROTECT_FROM_CLOSE
 push -1
 push eax
 xchg ebx, eax
 call SetHandleInformation
 push ebx
 call CloseHandle
 ...
l1: db "myfile", 0
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 mov rdx, offset being_debugged
 xor ecx, ecx
 inc ecx
 call AddVectoredExceptionHandler
 cdq
 push rdx
 push rdx
 push 3                  ; OPEN_EXISTING
 sub esp, 20h
```

```
 xor r9d, r9d
 xor r8d, r8d
 mov rcx, offset l1
 call CreateFileA
 mov ebx, eax
 push 2                    ; HANDLE_FLAG_PROTECT_FROM_CLOSE
 pop r8
 or rdx, -1
 xchg ecx, eax
 call SetHandleInformation
 mov ecx, ebx
 call CloseHandle
 ...
l1: db "myfile", 0
```

이러한 것들을 막는 방식은 (FirstHandler Vectored Exception Handler가 예외를 숨기고 조용히 실행을 재개하기 위한 목적으로 디버거에 의해서 등록될 수 있는) 윈도우 XP와 그 이후에서는 쉽다. 그러나 다른 핸들러가 첫번째 핸들러로 등록되는 것을 막기 위해 kernel32 AddVectoredExceptionHandler() 함수를 (또는 ntdll RtlAddVectoredExceptionHandler() 함수를) 투명하게 후킹하는 것에 대한 문제가 존재한다. 첫 번째 핸들러로 등록하기 위한 시도를 가로채서 마지막 핸들러로 만드는 것은 충분치 않다. 이유는 핸들러들이 다른 순서로 호출되기 때문에 이것이 두 핸들러들을 등록함으로써 드러나지고 예외를 일으키기 때문이다. 또한 함수를 후킹하고 이러한 변화된 요청을 탐지하며 간단하게 이것을 되돌리는 것에 대한 잠재적인 문제가 존재한다. 이것을 고칠 다른 방식은 이것이 list head를 hold하는 베이스 포인터를 찾기 위해 함수를 디스어셈블하는 것이지만 이것은 플랫폼에 종속적이다. 물론 함수가 handler structure에 한 포인터를 반환하지만 list는 핸들러를 등록하고 구조를 파싱 함으로써 순회될 수 있다.

이 상황은 아직도 예외를 숨기기 위해 SEH를 등록하기 위해 윈도우 NT와 윈도우 2000에서 ntdll NtClose() 함수를 투명하게 후킹하는 문제보다는 낫다.

디버거가 존재하지 않더라도 예외적인 행동을 생산하기 위해 설정될 수 있는 플래그가 존재한다. 유효하지 않거나 보호된 핸들이 함수에 전달되면 "HKLM₩System₩CurrentControlSet₩Control₩Session Manager₩GlobalFlag" 레지스티리 값에 있는 FLG_ENABLE_CLOSE_EXCEPTIONS (0x400000) 플래그를 설정하고 리부팅함으로써 kernel32 CloseHandle() 함수와 ntdll NtClose() 함수는 항상 예외를 발생시킬 것이다. 윈도우 NT 기반의 64비트와 32비트 버전에서 이 효과는 시스템 전체적으로 지원된다.

핸들을 받는 다른 함수들에서 비슷한 행동을 유발하는 또 다른 플래그가 존재한다. 유효하

지 않은 핸들이 ntoskrnl ObReferenceObjectByHandle() 함수를 호출하는 함수로 전달되면 (kernel32 SetEvent() 함수 같은) "HKLM₩System₩CurrentControlSet₩Control₩Session Manager₩GlobalFlag" 레지스트리 값에서 FLG_APPLICATION_VERIFIER (0x100) 플래그를 설정하고 리부팅함으로써 ntoskrnl ObReferenceObjectByHandle() 함수는 항상 예외를 발생시킬 것이다. 또한 효과는 시스템 전체적이다.

이러한 플래그들 중 하나는 현재 부정확하게 문서화되어 있으며 다른 것들은 불완전하게 문서화되어 있다는 점을 주의해야 한다. "Enable close exception" 플래그 (http://msdn.microsoft.com/en-us/library/ff542887.aspx) 는 ntoskrnl NtClose() 함수가 아닌 다른 함수로 전달된 유효하지 않은 핸들들에서 예외를 발생시킨다고 문서화되어 있지만 이것은 사실이 아니다. "Enable bad handles detection" 플래그 (http://msdn.microsoft.com/en-us/library/ff542881.aspx)는 ntoskrnl NtClose() 함수가 아닌 다른 함수로 전달되는 유효하지 않은 핸들들에 대해 예외를 발생시킬 것이지만 이 행위는 문서화되어 있지 않다.

iii. CreateFile

디버거의 존재를 탐지하기 위한 약간 신뢰할 수 없는 방식은 현재 프로세스의 파일을 배타적으로 여는 것이다. 몇몇 디버거들이 존재한다면 이 행동은 항상 실패할 것이다. 이유는 프로세스가 디버깅을 위해 시작되면 파일에 대한 핸들이 열리기 때문이다. 이것은 디버거가 파일에서 디버그 정보를 읽을 수 있게 한다(이것이 존재한다고 가정하고). 핸들 값은 CREATE_PROCESS_DEBUG_EVENT 이벤트가 발생할 때 채워지는 구조체에 저장된다. 만약 핸들이 디버거에 의해 닫히지 않았자면 파일은 배타적인 권한으로 열 수 없다. 디버거가 파일을 열지 않았기 때문에 이것을 닫는 것을 잊는 일은 흔한 편이다. 물론 다른 애플리케이션 (hex editor 같은)이 파일을 검사중이라면 여는 것은 또한 같은 이유로 실패할 것이다. 이것은 이 기법이 신뢰할 수 없는 이유이지만 주의함에 따라 이러한 false positives는 받아들일 만 할 수 있다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push 104h               ; MAX_PATH
mov ebx, offset l1
push ebx
push 0                  ; self filename
call GetModuleFileNameA
cdq
push edx
push edx
push 3                  ; OPEN_EXISTING
push edx
```

```
push edx
inc edx
ror edx, 1
push edx                ; GENERIC_READ
push ebx
call CreateFileA
inc eax
je being_debugged
...
l1: db 104h dup (?)     ; MAX_PATH
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다. 그러나 이 기법은 64비트 프로세스들에서는 통하지 않는다.

```
mov r8d, 104h           ; MAX_PATH
mov rbx, offset l1
push rbx
pop rdx
xor ecx, ecx            ; self filename
call GetModuleFileNameA
cdq
push rdx
push rdx
push 3                  ; OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
inc edx
ror edx, 1              ; GENERIC_READ
push rbx
pop rcx
call CreateFileA
inc eax
je being_debugged
...
l1: db 104h dup (?)     ; MAX_PATH
```

kernel32 CreateFile() 함수도 또한 디버거에 속할 수 있는 커널 모드 드라이버들의 존재를 탐지하는데 사용될 수 있다. 커널 모드 드라이버들을 사용하는 툴들은 또한 이 드라이버들

과 통신하는 방법을 필요로 한다. 가장 흔한 방식은 named device들의 사용을 통한 것이다. 그러므로 이러한 디바이스를 열려고 시도함에 따라 어떠한 성공도 디버거의 존재를 가리킨다고 할 수 있다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor eax, eax
 mov edi, offset l2
l1: push eax
 push eax
 push 3                    ; OPEN_EXISTING
 push eax
 push eax
 push eax
 push edi
 call CreateFileA
 inc eax
 jne being_debugged
 or ecx, -1
 repne scasb
 cmp [edi], al
 jne l1
 ...
l2: <array of ASCIIZ strings, zero to end>
```

전형적인 목록은 아래의 이름들을 포함한다.

```
db "₩₩.₩EXTREM", 0          ; Phant0m
db "₩₩.₩FILEM", 0           ; FileMon
db "₩₩.₩FILEVXG", 0         ; FileMon
db "₩₩.₩ICEEXT", 0          ; SoftICE Extender
db "₩₩.₩NDBGMSG.VXD", 0     ; SoftICE
db "₩₩.₩NTICE", 0           ; SoftICE
db "₩₩.₩REGSYS", 0          ; RegMon
db "₩₩.₩REGVXG", 0          ; RegMon
db "₩₩.₩RING0", 0           ; Olly Advanced
db "₩₩.₩SICE", 0            ; SoftICE
db "₩₩.₩SIWVID", 0          ; SoftICE
db "₩₩.₩TRW", 0             ; TRW
db "₩₩.₩SPCOMMAND", 0       ; Syser
```

```
db "₩₩.₩SYSER", 0                ; Syser
db "₩₩.₩SYSERBOOT", 0            ; Syser
db "₩₩.₩SYSERDBGMSG", 0          ; Syser
db "₩₩.₩SYSERLANGUAGE", 0        ; Syser
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 mov rdi, offset l2
l1: xor edx, edx
 push rdx
 push rdx
 push 3                          ; OPEN_EXISTING
 sub esp, 20h
 xor r9d, r9d
 xor r8d, r8d
 push rdi
 pop rcx
 call CreateFileA
 inc eax
 jne being_debugged
 or ecx, -1
 repne scasb
 cmp [rdi], al
 jne l1
 ...
l2: <array of ASCIIZ strings, zero to end>
```

그러나 사용자 모드 서비스들을 제공하는 커널 모드 디버거들의 부족으로 인해 현재 64비트 list는 존재하지 않는다.

오직 버전 4.0 이전의 SoftICE에서 "₩₩.₩NTICE" 드라이버 이름이 유효하다는 것에 주의하자. SoftICE v4.x는 윈도우 NT 기반 플랫폼에서 이러한 이름으로 디바이스를 생성하지 않는다. 대신 디바이스 이름은 "₩₩.₩NTICExxxx"이며 "xxxx"는 네 16진수 문자들이다. 이 문자들의 source는 "Serial" 레지스트리 값에서의 데이터에서 온 문자들로서 9th, 7th, 5th 그리고 3rd이다. 이 값은 레지스트리에서 여러 곳에 존재한다. SoftICE 드라이버는 "HKLM₩System₩CurrentContrlSet₩Services₩NTice₩Serial" 레지스트리 값을 사용한다. nmtrans DevIO_ConnectToSoftICE() 함수는 "HKLM₩Software₩NuMega₩SoftIce₩Serial" 레지스트리 값을 사용한다. SoftICE가 사용하는 알고리즘은 문자열을 reverse하고 세 번째 문자를 시작으로 네 문자들에 해서 모든 두번째 문자를 받는다. 물론 이것을 archieve하

는 더 간단한 방식이 존재한다. 이름은 32비트 버전의 윈도우에서 32비트 윈도우 환경을 검사하기 위해 이 32비트 코드를 사용해서 만들어질 수 있다. SoftICE는 64비트 버전의 윈도우에서 돌아가지 않는다.

```
 xor ebx, ebx
 push eax
 push esp
 push 1                  ; KEY_QUERY_VALUE
 push ebx
 push offset l2
 push 80000002h          ; HKLM
 call RegOpenKeyExA
 pop ecx
 push 0dh                ; sizeof(l3)
 push esp
 mov esi, offset l3
 push esi
 push eax                ; REG_NONE
 push eax
 push offset l4
 push ecx
 call RegQueryValueExA
 push 4
 pop ecx
 mov edi, offset l6
l1: mov al, [ecx*2+esi+1]
 stosb
 loop l1
 push ebx
 push ebx
 push 3                  ; OPEN_EXISTING
 push ebx
 push ebx
 push ebx
 push offset l5
 call CreateFileA
 inc eax
 jne being_debugged
 ...
```

```
l2: db "Software₩NuMega₩SoftIce", 0
l3: db 0dh dup (?)
l4: db "Serial", 0
l5: db "₩₩.₩ntice"
l6: db "xxxx", 0
```

iv. LoadLibrarary

kernel32 LoadLibrary() 함수는 디버거를 탐지하는데 놀라울 정도로 간단하고 효과적이다. 디버거가 존재하는 경우 파일이 kernel32 LoadLibrary() 함수를 사용해서 (또는 kernel32 LoadLibraryEx() 함수 또는 ntdll LdrLoadDll() 함수 같은 이것의 어떠한 변형이라도) 로드되면 파일에 대한 핸들이 열린다. 이것은 존재한다는 가정 하에 디버거가 파일로부터 디버그 정보를 읽을 수 있게 한다. 핸들 값은 LOAD_DLL_DEBUG_EVENT 이벤트가 발생할 때 채워지는 구조체에 저장된다. 만약 핸들이 디버거에 의해 닫히지 않으면(DLL을 언로딩하는 것은 이것을 닫지 않는다) 파일은 배타적으로 액세스하기 위해 열릴 수 없다. 디버거가 파일을 열지 않았기 때문에 닫는것을 잊기 쉽다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 mov ebx, offset l1
 push ebx
 call LoadLibraryA
 cdq
 push edx
 push edx
 push 3                  ; OPEN_EXISTING
 push edx
 push edx
 inc edx
 ror edx, 1              ; GENERIC_READ
 push edx
 push ebx
 call CreateFileA
 inc eax
 je being_debugged
 ...
l1: db "myfile.", 0
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
mov rbx, offset l1
push rbx
pop rcx
call LoadLibraryA
xor edx, edx
push rdx
push rdx
push 3                  ; OPEN_EXISTING
sub esp, 20h
xor r9d, r9d
xor r8d, r8d
inc edx
ror edx, 1              ; GENERIC_READ
push rbx
pop rcx
call CreateFileA
inc eax
je being_debugged
...
l1: db "myfile.", 0
```

대체 방식으로 내부적으로 kernel32 CreateFile() 함수를 호출하는(또는 이것의 변형 : ntdll NtCreateFile() 함수 또는 ntdll NtOpenFile() 함수) 다른 함수를 호출하는 방법이 있다. 예시로 kernel32 EndUpdateResource() 함수 같은 resourceupdating 함수들이 있다. kernel32 EndUpdateResource() 함수가 통하는 이유는 이것이 새로운 resource table을 쓰기 위해 결국 kernel32 CreateFile() 함수를 호출하기 때문이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어 질 수 있다.

```
mov ebx, offset l1
push ebx
call LoadLibraryA
push 0
push ebx
call BeginUpdateResourceA
push 0
push eax
call EndUpdateResourceA
test eax, eax
```

```
 je being_debugged
 ...
l1: db "myfile.", 0
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다(그러나 이 기법은 64비트 프로세스들에게는 통하지 않는다).

```
 mov rbx, offset l1
 push rbx
 pop rcx
 call LoadLibraryA
 xor edx, edx
 push rbx
 pop rcx
 call BeginUpdateResourceA
 cdq
 xchg ecx, eax
 call EndUpdateResourceA
 test eax, eax
 je being_debugged
 ...
l1: db "myfile.", 0
```

V. ReadFile

kernel32 ReadFile() 함수는 호출 이후에 그 위치의 파일 내용을 읽음으로써 코드 흐름의 자가 변조를 수행할 때 사용될 수 있다. 이것은 또한 디버거가 코드 흐름에 놓을(특히 호출 직후에) 소프트웨어 브레이크포인트를 제거하는데 사용될 수 있다. 결론적으로 코드는 자유롭게 실행되게 된다. 호출은 32비트 또는 64비트 윈도우 버전에서 이 32비트 코드를 사용하여 만들어질 수 있다.

```
 push 104h                ; MAX_PATH
 mov ebx, offset l2
 push ebx
 push 0                   ; self filename
 call GetModuleFileNameA
 cdq
 push edx
 push edx
```

```
 push 3                 ; OPEN_EXISTING
 push edx               ; FILE_SHARE_READ
                        ; because a debugger might prevent
                        ; exclusive access to the running file
 inc edx
 push edx
 ror edx, 1
 push edx               ; GENERIC_READ
 push ebx
 call CreateFileA
 push 0
 push esp
 push 1                 ; more bytes might be more useful
 push offset l1
 push eax
 call ReadFile
l1: int 3               ; replaced by "M" from the MZ header
 ...
l2: db 104h dup (?)     ; MAX_PATH
```

또는 64비트 버전의 윈도우에서 이 64비트 코드를 사용할 수 있다.

```
 mov r8d, 104h          ; MAX_PATH
 mov rbx, offset l2
 push rbx
 pop rdx
 xor ecx, ecx           ; self filename
 call GetModuleFileNameA
 cdq
 push rdx
 push rdx
 push 3                 ; OPEN_EXISTING
 sub esp, 20h
 xor r9d, r9d           ; FILE_SHARE_READ
                        ; because a debugger might prevent
                        ; exclusive access to the running file
 inc edx
 push rdx
 pop r8
```

```
 ror edx, 1                ; GENERIC_READ
 push rbx
 pop rcx
 call CreateFileA
 push 0
 mov r9d, esp
 sub esp, 20h
 push 1                    ; more bytes might be more useful
 pop r8
 mov rdx, offset l1
 xchg ecx, eax
 call ReadFile
l1: int 3                  ; replaced by "M" from the MZ header
  ...
l2: db 104h dup (?)        ; MAX_PATH
```

이 기법을 막는 한 방법은 함수 호출을 step over할 때 소프트웨어 브레이크포인트 대신 하드웨어 브레이크포인트를 사용하는 것이다.

C. Execution Timing

RDPMC
RDTSC
GetLocalTime
GetSystemTime
GetTickCount
KiGetTickCount
QueryPerformanceCounter
timeGetTime

디버거가 존재하고 코드를 singlestep하고 있을 때 native 실행과 비교해서 각 명령어들 사이의 실행 사이에 상당한 딜레이가 존재하게 된다. 이 딜레이는 여러 가능한 time source들 중 하나를 사용해서 측정될 수 있다. 이러한 source들은 RDPMC 명령어(그러나 이 명령어는 PCE 플래그가 CR4 레지스터에서 설정되어있어야 함을 요구하지만 이것은 기본 설정이 아니다), RDTSC 명령어(그러나 이 명령어는 TSD 플래그가 CR4 레지스터에서 clear되어있어야 함을 요구하지만 이것은 기본 설정이다), kernel32 GetLocalTime() 함수, kernel32 GetSystemTime() 함수, kernel32 QueryPerformanceCounter() 함

수, kernel32 GetTickCount() 함수, ntoskrnl KiGetTickCount() 함수(32비트 버전의 윈도우에서 인터럽트 0x2A 인터페이스를 통해 노출되는) 그리고 winmm timeGetTime() 함수를 포함한다. 그러나 winmm timeGetTime() 함수의 resolution은 변수이며 이것이 내부적으로 kernel32 GetTickCount() 함수로 분기하느냐에 의존하여 작은 intervals를 측정하는 것을 매우 신뢰할 수 없게 만든다. RDMSR 명령어는 또한 time source로서 사용될 수 있지만 사용자 모드에서는 사용될 수 없다. RDPMC 명령어로 32비트 또는 64비트 윈도우 환경을 조사하기 위한 검사는 다음의 코드를 사용해서(32비트와 64비트에서 같다) 만들어질 수 있다.

```
xor ecx, ecx                ; read 32bit counter 0
rdpmc
xchg ebx, eax
rdpmc
sub eax, ebx
cmp eax, 500h
jnbe being_debugged
```

32비트 또는 64비트 윈도우 환경을 조사하기 위한 RDTSC 명령어를 이용한 검사는 다음의 코드를 사용해서(32비트와 64비트에서 같다) 만들어질 수 있다.

```
rdtsc
xchg esi, eax
mov edi, edx
rdtsc
sub eax, esi
sbb edx, edi
jne being_debugged
cmp eax, 500h
jnbe being_debugged
```

32비트 또는 64비트 윈도우 환경에서 32비트 윈도우 환경을 조사하기 위한 kernel32 GetLocalTime() 함수를 이용한 검사는 이 코드를 사용해서 만들어질 수 있다.

```
mov ebx, offset l1
push ebx
call GetLocalTime
mov ebp, offset l2
push ebp
call GetLocalTime
mov esi, offset l3
```

```
push esi
push ebx
call SystemTimeToFileTime
mov edi, offset l4
push edi
push ebp
call SystemTimeToFileTime
mov eax, [edi]
sub eax, [esi]
mov edx, [edi+4]
sbb edx, [esi+4]
jne being_debugged
cmp eax, 10h
jnbe being_debugged
...
l1: db 10h dup (?)        ; sizeof(SYSTEMTIME)
l2: db 10h dup (?)        ; sizeof(SYSTEMTIME)
l3: db 8 dup (?)          ; sizeof(FILETIME)
l4: db 8 dup (?)          ; sizeof(FILETIME)
```

또는 64비트 윈도우 환경을 조사하기 위해 다음의 64비트 코드를 사용할 수 있다.

```
mov rbx, offset l1
push rbx
pop rcx
call GetLocalTime
mov rbp, offset l2
push rbp
pop rcx
call GetLocalTime
mov rsi, offset l3
push rsi
pop rdx
push rbx
pop rcx
call SystemTimeToFileTime
mov rdi, offset l4
push rdi
pop rdx
```

```
 push rbp
 pop rcx
 call SystemTimeToFileTime
 mov rax, [rdi]
 sub rax, [rsi]
 cmp rax, 10h
 jnbe being_debugged
 ...
l1: db 10h dup (?)        ; sizeof(SYSTEMTIME)
l2: db 10h dup (?)        ; sizeof(SYSTEMTIME)
l3: db 8 dup (?)          ; sizeof(FILETIME)
l4: db 8 dup (?)          ; sizeof(FILETIME)
```

kernel32 GetSystemTime() 함수를 이용한 검사는 함수 이름을 제외하고 kernel32 GetLocalTime() 함수에서와 같은 코드를 사용해서 만들어질 수 있다.

32비트 또는 64비트 윈도우 환경을 조사하기 위한 Rkernel32 GetTickCount() 함수를 이용한 검사는 이 코드를 사용해서(32비트와 64비트에서 같다) 만들어질 수 있다.

```
call GetTickCount
xchg ebx, eax
call GetTickCount
sub eax, ebx
cmp eax, 10h
jnbe being_debugged
```

32비트 또는 64비트 윈도우 환경을 조사하기 위한 ntoskrnl KiGetTickCount() 함수를 이용한 검사는 다음의 32비트 코드를 사용해서 만들어질 수 있다. 이 인터럽트는 64비트 버전의 윈도우에서는 지원되지 않는다.

```
int 2ah
xchg ebx, eax
int 2ah
sub eax, ebx
cmp eax, 10h
jnbe being_debugged
```

32비트 또는 64비트 윈도우에서 32비트 윈도우 환경을 조사하기 위한 kernel32 QueryPerformanceCounter() 함수를 이용한 검사는 다음의 32비트 코드를 사용해서 만들어

질 수 있다.

```
 mov esi, offset l1
 push esi
 call QueryPerformanceCounter
 mov edi, offset l2
 push edi
 call QueryPerformanceCounter
 mov eax, [edi]
 sub eax, [esi]
 mov edx, [edi+4]
 sbb edx, [esi+4]
 jne being_debugged
 cmp eax, 10h
 jnbe being_debugged
 ...
l1: db 8 dup (?)          ; sizeof(LARGE_INTEGER)
l2: db 8 dup (?)          ; sizeof(LARGE_INTEGER)
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 mov rsi, offset l1
 push rsi
 pop rcx
 call QueryPerformanceCounter
 mov rdi, offset l2
 push rdi
 pop rcx
 call QueryPerformanceCounter
 mov rax, [rdi]
 sub rax, [rsi]
 cmp rax, 10h
 jnbe being_debugged
 ...
l1: db 8 dup (?)          ; sizeof(LARGE_INTEGER)
l2: db 8 dup (?)          ; sizeof(LARGE_INTEGER)
```

32비트 또는 64비트 윈도우에서 32비트 윈도우 환경을 조사하기 위
한 winmm timeGetTime() 함수를 이용한 검사는 이 코드를 사용해서(32비트와 64비트에서 같

다) 만들어질 수 있다.

```
call timeGetTime
xchg ebx, eax
call timeGetTime
sub eax, ebx
cmp eax, 10h
jnbe being_debugged
```

D. Processlevel

CheckRemoteDebuggerPresent
CreateToolhelp32Snapshot
DbgSetDebugFilterState
IsDebuggerPresent
NtQueryInformationProcess
RtlQueryProcessHeapInformation
RtlQueryProcessDebugInformation
SwitchToThread
Toolhelp32ReadProcessMemory
UnhandledExceptionFilter

i. CheckRemoteDebuggerPresent

kernel32 CheckRemoteDebuggerPresent() 함수는 윈도우 NT부터 존재해왔던 값을 질의하기 위해 윈도우 XP SP1에서 도입되었다. 여기서 "Remote"는 같은 machine에서 독립된 프로세스를 가리킨다. 디버거가 존재하면(즉 현재 프로세스에 어태치되었으면) 함수는 pbDebuggerPresent argument가 가리키는 값을 0xffffffff로 설정한다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push eax
push esp
push -1                    ; GetCurrentProcess()
call CheckRemoteDebuggerPresent
pop eax
test eax, eax
jne being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
enter 20h, 0
mov edx, ebp
or rcx, -1                    ; GetCurrentProcess()
call CheckRemoteDebuggerPresent
leave
test ebp, ebp
jne being_debugged
```

ii. Parent Process

사용자들은 일반적으로 shell process(Explorer.exe)에 보여지는아이콘을 클릭해서 애플리케이션을 실행한다. 결과적으로 실행되는 프로세스의 부모 프로세스는 Explorer.exe가 될 것이다. 물론 애플리케이션이 커맨드 라인에서 실행되면 부모 프로세스는 command window process가 될 것이다. 디버깅하면서 애플리케이션을 실행하는 것은 실행되는 프로세스의 부모 프로세스를 디버거 프로세스로 만들어 준다.

커맨드 라인에서 애플리케이션을 실행하는 것은 몇몇 애플리케이션들에서 문제를 일으키는데 이것들은 부모 프로세스가 Explorer.exe라고 예상하기 때문이다. 몇몇 애플리케이션들은 부모 프로세스 이름을 검사하고 이것이 "Explorer.exe"이길 예상한다. 어떤 애플리케이션은 부모 프로세스 ID를 Explorer.exe와 비교한다. 두 경우 모두 값이 다른 경우에 애플리케이션은 자신이 디버깅 당하고 있다고 생각하는 결과를 초래할 것이다.

이 관점에서 우리는 약간 주제를 돌리고 필연적으로 이후 등장할 토픽을 소개한다. Explorer.exe의 프로세스 ID를 얻는 가장 쉬운 방법은 user32 GetShellWindow() 와 user32 GetWindowThreadProcessId() 함수를 호출하는 것이다. 이것은 ProcessBasicInformation 클래스로 ntdll NtQueryInformationProcess() 함수를 호출함으로써 얻어지는 현재 프로세스의 부모 프로세스의 ID와 이름을 가르쳐준다. 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 호출들은 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
call GetShellWindow
push eax
push esp
push eax
call GetWindowThreadProcessId
push 0
```

```
 push 18h                ; sizeof(PROCESS_BASIC_INFORMATION)
 mov ebp, offset l1
 push ebp
 push 0                  ; ProcessBasicInformation
 push -1                         ; GetCurrentProcess()
 call NtQueryInformationProcess
 pop eax                 ;I nheritedFromUniqueProcessId
 cmp [ebp+14h], eax
 jne being_debugged
 ...                     ; sizeof(PROCESS_BASIC_INFORMATION)
l1: db 18h dup (?)
```

또는 64비트 윈도우 환경을 조사하기 위한 이 64비트 코드를 사용할 수 있다.

```
 call GetShellWindow
 enter 20h, 0
 mov edx, ebp
 xchg ecx, eax
 call GetWindowThreadProcessId
 leave
 push 0
 sub esp, 20h
 push 30h                ; sizeof(PROCESS_BASIC_INFORMATION)
 pop r9
 mov rbx, offset l1
 push rbx
 pop r8
 cdq                     ; ProcessBasicInformation
 or rcx, -1              ; GetCurrentProcess()
 call NtQueryInformationProcess  ;InheritedFromUniqueProcessId
 cmp [rbx+20h], ebp
 jne being_debugged
 ...                     ; sizeof(PROCESS_BASIC_INFORMATION)
l1: db 30h dup (?)
```

그러나 이 코드는 심각한 문제를 가지고 있는데 만약 "HKCU₩Software₩Microsoft ₩Windows₩CurrentVersion₩Explorer₩Advanced₩SeparateProcess" 레지스트리 값(윈도우 2000에서 도입된)이 0이 아니라면 단일 세션 내에 Explorer.exe의 여러 instance들이 존재할 수 있다는 것이다. 이것은 열린 모든 윈도우를 위한 Explorer.exe의 독립된 복사본들을 실

행시키는 효과를 갖는다. 결과적으로 셸 윈도우는 현재 프로세스의 부모 프로세스가 아닐 수 있으며 Explorer.exe가 부모 프로세스 이름이 아닐 수 있다.

iii.CreateToolhelp32Snapshot

Explorer.exe와 현재 프로세스의 부모 프로세스의 프로세스 ID, 이 부모 프로세스의 이름 모두 kernel32 CreateToolhelp32Snapshot() 함수와 kernel32 Process32Next() 함수 enumeration에 의해 얻어질 수 있다. 이 호출은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하기 위해 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor esi, esi
 xor edi, edi
 push esi
 push 2                    ; TH32CS_SNAPPROCESS
 call CreateToolhelp32Snapshot
 mov ebx, offset l9
 xchg ebp, eax
l1: push ebx
 push ebp
 call Process32First
l2: mov eax, fs:[eax+1fh] ; UniqueProcess
 cmp [ebx+8], eax         ; th32ProcessID
 cmove edi, [ebx+18h]  ; th32ParentProcessID
 test edi, edi
 je l3
 cmp esi, edi
 je l7
l3: lea ecx, [ebx+24h]    ; szExeFile
 push esi
 mov esi, ecx
l4: lodsb
 cmp al, "₩"
 cmove ecx, esi
 or b [esi1], " "
 test al, al
 jne l4
 sub esi, ecx
 xchg ecx, esi
 push edi
```

```
 mov edi, offset l8
 repe cmpsb
 pop edi
 pop esi
 jne l6
 test esi, esi
 je l5
 mov esi, offset l10
 cmp cl, [esi]
 adc [esi], ecx
l5: mov esi, [ebx+8]      ; th32ProcessID
l6: push ebx
 push ebp
 call Process32Next
 test eax, eax
 jne l2
 dec b [offset l10+1]
 jne l1
 jmp being_debugged
l7: ...                   ; trailing zero is converted to space
l8: db "explorer.exe "
l9: dd 128h               ; sizeof(PROCESSENTRY32)
db 124h dup (?)
l10:db 0ffh, 1, ?, ?
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 xor esi, esi
 xor edi, edi
 xor edx, edx
 push 2                   ; TH32CS_SNAPPROCESS
 pop rcx
 call CreateToolhelp32Snapshot
 mov rbx, offset l9
 xchg ebp, eax
l1: push rbx
 pop rdx
 mov ecx, ebp
 call Process32First
```

```
l2: mov eax, gs:[rax+3fh] ; UniqueProcess
 cmp [rbx+8], eax         ; th32ProcessID
 cmove edi, [rbx+20h]   ; th32ParentProcessID
 test esi, esi
 je l3
 cmp esi, edi
 je l7
l3: lea ecx, [rbx+2ch]   ; szExeFile
 push rsi
 mov esi, ecx
l4: lodsb
 cmp al, "₩"
 cmove ecx, esi
 or b [rsi1], " "
 test al, al
 jne l4
 sub esi, ecx
 xchg ecx, esi
 push rdi
 mov rdi, offset l8
 repe cmpsb
 pop rdi
 pop rsi
 jne l6
 test esi, esi
 je l5
 mov rsi, offset l10
 cmp cl, [rsi]
 adc [rsi], ecx
l5: mov esi, [rbx+8]       ; th32ProcessID
l6: push rbx
 pop rdx
 mov ecx, ebp
 call Process32Next
 test eax, eax
 jne l2
 dec b [offset l10+1]
 jne l1
 jmp being_debugged
```

```
l7: ...                     ; trailing zero is converted to space
l8: db "explorer.exe "
l9: dd 130h                 ; sizeof(PROCESSENTRY32)
db 12ch dup (?)
l10:db 0ffh, 1, ?, ?
```

이 정보가 커널에서 왔기 때문에 사용자 모드 코드에게 이 호출이 디버거의 존재를 드러내는 것을 막을 쉬운 방법은 없다. 이것을 막으려고 시도하는 일반적인 기법은 kernel32 Process32Next() 함수가 FALSE를 반환하게 강제해서 루프를 일찍 끝내게 하는 것이다. 그러나 만약 Explorer.exe나 현재 프로세스가 보이지 않는다면 이것은 의심스러운 상태여야 이다.

Explorer.exe의 복사본이 단지 하나만 존재한다면 코드는 single pass에서 실행될 것이다. 만약 여러 복사본들이 존재한다면 코드는 second pass를 수행할 것이다. first pass에서 부모 프로세스 ID가 얻어지고 second pass에서 부모 프로세스 ID는 Explorer.exe의 각 instance의 프로세스 ID와 비교될 것이다.

이 코드에는 만약 여러 사용자들이 동시에 로그인하면 그들의 프로세스들 또한 보일 것이라는 사소한 문제가 있다. 적어도 그들 중 하나는 Explorer.exe일 것이고 이것은이 세션에서 어떠한 프로세스의 부모 프로세스도 아닐 것이다. 이것은 현재 세션에서 Explorer.exe의 단지 한 instance만 존재하기 때문에, 심지어 그들 중 단지 하나만 충분하다면 코드가 두 pass를 실행하게 만든다.

이 문제를 피할 방식으로 프로세스의 사용자 이름과 도메인 이름을 결정하고 발견된 프로세스를 받아들이기 전에 match하는 것이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 xor esi, esi
 xor edi, edi
 push esi
 push 2                     ; TH32CS_SNAPPROCESS
 call CreateToolhelp32Snapshot
 mov ebx, offset l16
 xchg ebp, eax
l1: push ebx
 push ebp
 call Process32First
l2: mov eax, fs:[eax+1fh] ; UniqueProcess
 cmp [ebx+8], eax          ; th32ProcessID
```

```
 cmove edi, [ebx+18h]  ; th32ParentProcessID
 test edi, edi
 je l3
 cmp esi, edi
 je l9
l3: lea ecx, [ebx+24h]    ; szExeFile
 push esi
 mov esi, ecx
l4: lodsb
 cmp al, "₩"
 cmove ecx, esi
 or b [esi-1], " "
 test al, al
 jne l4
 sub esi, ecx
 xchg ecx, esi
 push edi
 mov edi, offset l15
 repe cmpsb
 pop edi
 pop esi
 jne l8
 mov eax, [ebx+8]        ; th32ProcessID
 push ebx
 push ebp
 push esi
 push edi
 call l10
 dec ecx                           ; invert Z flag
 jne l6
 push ebx
 push edi
 dec ecx
 call l11
 pop esi
 pop edx
 mov cl, 2                ; compare user names
                          ; then domain names
l5: lodsb
```

```
 scasb
 jne l6
 test al, al
 jne l5
 mov esi, ebx
 mov edi, edx
 loop l5
l6: pop edi
 pop esi
 pop ebp
 pop ebx
 jne l8
 test esi, esi
 je l7
 mov esi, offset l17
 cmp cl, [esi]
 adc [esi], ecx
l7: mov esi, [ebx+8]       ; th32ProcessID
l8: push ebx
 push ebp
 call Process32Next
 test eax, eax
 jne l2
 dec b [offset l17+1]
 jne l1
 jmp being_debugged
l9: ...  l10:push eax
 push 0
 push 400h                 ; PROCESS_QUERY_INFORMATION
 call OpenProcess
 xchg ecx, eax
 jecxz l14
l11:push eax
 push esp
 push 8                    ; TOKEN_QUERY
 push ecx
 call OpenProcessToken
 pop ebx
 xor ebp, ebp
```

```
l12:push ebp
 push 0                    ; GMEM_FIXED
 call GlobalAlloc
 push eax
 push esp
 push ebp
 push eax
 push 1                    ; TokenUser
 push ebx
 xchg esi, eax
 call GetTokenInformation
 pop ebp
 xchg ecx, eax
 jecxz l12
 xor ebp, ebp
l13:push ebp
 push 0                    ; GMEM_FIXED
 call GlobalAlloc
 xchg ebx, eax
 push ebp
 push 0                    ; GMEM_FIXED
 call GlobalAlloc
 xchg edi, eax
 push eax
 mov eax, esp
 push ebp
 mov ecx, esp
 push ebp
 mov edx, esp
 push eax
 push ecx
 push ebx
 push edx
 push edi
 push d [esi]
 push 0
 call LookupAccountSidA
 pop ecx
 pop ebp
```

```
 pop edx
 xchg ecx, eax
 jecxz l13
l14:ret                  ; trailing zero is converted to space
l15:db "explorer.exe "
l16:dd 128h              ; sizeof(PROCESSENTRY32)
 db 124h dup (?)
l17:db 0ffh, 1, ?, ?
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
 xor esi, esi
 xor edi, edi
 xor edx, edx
 push 2                  ; TH32CS_SNAPPROCESS
 pop rcx
 call CreateToolhelp32Snapshot
 mov rbx, offset l16
 xchg ebp, eax
l1: push rbx
 pop rdx
 mov ecx, ebp
 call Process32First
l2: mov eax, gs:[rax+3fh] ; UniqueProcess
 cmp [rbx+8], eax        ; th32ProcessID
 cmove edi, [rbx+20h]    ; th32ParentProcessID
 test esi, esi
 je l3
 cmp esi, edi
 je l9
l3: lea ecx, [rbx+2ch]   ; szExeFile
 push rsi
 mov esi, ecx
l4: lodsb
 cmp al, "₩"
 cmove ecx, esi
 or byte [rsi-1], " "
 test al, al
 jne l4
```

```
 sub esi, ecx
 xchg ecx, esi
 push rdi
 mov rdi, offset l15
 repe cmpsb
 pop rdi
 pop rsi
 jne l8
 mov r8d, [rbx+8]        ; th32ProcessID
 push rbx
 push rbp
 push rsi
 push rdi
 call l10
 dec ecx                 ; invert Z flag
 jne l6
 push rbx
 push rdi
 dec rcx
 call l11
 pop rsi
 pop rdx
 mov cl, 2               ; compare user names
                         ; then domain names
l5: lodsb
 scasb
 jne l6
 test al, al
 jne l5
 mov esi, ebx
 mov edi, edx
 loop l5
l6: pop rdi
 pop rsi
 pop rbp
 pop rbx
 jne l8
 test esi, esi
 je l7
```

```
 mov rsi, offset l17
 cmp cl, [rsi]
 adc [rsi], ecx
l7: mov esi, [rbx+8]       ; th32ProcessID
l8: push rbx
 pop rdx
 mov ecx, ebp
 call Process32Next
 test eax, eax
 jne l2
 dec b [offset l17+1]
 jne l1
 jmp being_debugged
l9: ...
l10:cdq
 xor ecx, ecx
 mov ch, 4                 ; PROCESS_QUERY_INFORMATION
 enter 20h, 0
 call OpenProcess
 leave
 xchg ecx, eax
 jrcxz l14
l11:push rax
 mov r8d, esp
 push 8                    ; TOKEN_QUERY
 pop rdx
 call OpenProcessToken
 pop rbx
 xor ebp, ebp
l12:mov edx, ebp
 xor ecx, ecx              ; GMEM_FIXED
 enter 20h, 0
 call GlobalAlloc
 leave
 push rbp
 pop r9
 push rax
 pop r8
 push rax                  ; simulate enter
```

```
 mov ebp, esp
 push rbp
 sub esp, 20h
 push 1                  ; TokenUser
 pop rdx
 mov ecx, ebx
 xchg esi, eax
 call GetTokenInformation
 leave
 xchg ecx, eax
 jrcxz l12
 xor ebp, ebp
l13:mov ebx, ebp
 mov edx, ebp
 xor ecx, ecx            ; GMEM_FIXED
 enter 20h, 0
 call GlobalAlloc
 xchg ebx, eax
 xchg edx, eax
 xor ecx, ecx            ; GMEM_FIXED
 call GlobalAlloc
 leave
 xchg edi, eax
 push rbp
 mov ecx, esp
 push rbp
 mov r9d, esp
 push rax
 push rsp
 push rcx
 push rbx
 sub esp, 20h
 push rdi
 pop r8
 mov edx, [rsi]
 xor ecx, ecx
 call LookupAccountSidA
 add esp, 40h
 pop rcx
```

```
 pop rbp
 xchg ecx, eax
 jrcxz l13
l14:ret                         ; trailing zero is converted to space
l15:db "explorer.exe "
l16:dd 130h  ;sizeof(PROCESSENTRY32)
 db 12ch dup (?)
l17:db 0ffh, 1, ?, ?
```

이 코드는 단지 입증 목적을 위한 것이라는 점에 주의해야 한다. 이것은 제품을 위해 만들어지지 않은 코드들로서 핸들과 메모리 누수 같은 많은 좋지 않은 프로그래밍 예제들을 포함한다.

특정한 툴들의 이름을 찾는 이 기법의 변형은 디버거나 다른 비슷한 것들이 존재할 수도 있다는 것을 암시한다. 이 경우 로그인된 다른 사용자들의 세션들에서 툴들이 존재한다고 생각할 수 있다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용해서 만들어질 수 있다.

```
 push 0
 push 2                         ; TH32CS_SNAPPROCESS
 call CreateToolhelp32Snapshot
 mov ebx, offset l6
 xchg ebp, eax
l1: push ebx
 push ebp
 call Process32First
l2: lea ecx, [ebx+24h]    ; szExeFile
 mov esi, ecx
l3: lodsb
 cmp al, "₩"
 cmove ecx, esi
 or b [esi1], " "
 test al, al
 jne l3
 sub esi, ecx
 xchg ecx, esi
 mov edi, offset l5
l4: push ecx
 push esi
```

```
 repe cmpsb
 pop esi
 pop ecx
 je being_debugged
 push ecx
 or ecx, -1
 repne scasb
 pop ecx
 cmp [edi], al
 jne l4
 push ebx
 push ebp
 call Process32Next
 test eax, eax
 jne l2
 ...
l5: <array of ASCII strings  space then zero to end each one  zero to end the list  >
l6: dd 128h                 ; sizeof(PROCESSENTRY32)
db 124h dup (?)
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
xor edx, edx
push 2                      ;TH32CS_SNAPPROCESS
pop rcx
call CreateToolhelp32Snapshot
mov rbx, offset l6
xchg ebp, eax
l1: push rbx
pop rdx
mov ecx, ebp
call Process32First
l2: lea ecx, [rbx+2ch]     ; szExeFile
mov esi, ecx
l3: lodsb
cmp al, "₩"
cmove ecx, esi
or b [rsi1], " "
test al, al
```

```
jne l3
sub esi, ecx
xchg ecx, esi
mov rdi, offset l5
l4: push rcx
push rsi
repe cmpsb
pop rsi
pop rcx
je being_debugged
push rcx
or ecx, -1
repne scasb
pop rcx
cmp [rdi], al
jne l4
push rbx
pop rdx
mov ecx, ebp
call Process32Next
test eax, eax
jne l2
...
l5: <array of ASCII strings  space then zero to end each one  zero to end the list  >
l6: dd 130h  ;sizeof(PROCESSENTRY32)
db 12ch dup (?)
```

프로세스의 이름과 space가 보이는 곳에서 정확히 끝나는 substring을 이름으로 갖는 프로세스를 구분하는 시도가 실제로 없기 때문에 이름에 공백을 가진 프로세스를 실제로 탐지하는 것에는 문제가 있을 수 있다는 점에 주의해야 한다. 그러나 다시 한 번 말하지만 이러한 상황은 디버거의 존재로 인정될 수 있다.

iv. DbgBreakPoint

ntdll DbgBreakPoint() 함수는 디버거가 이미 실행중인 프로세스를 어태치할 때 호출된다. 이 함수는 디버거가 가로챌 수 있는 예외를 유발하므로 디버거가 제어를 얻을 수 있게 한다. 그러나 이것은 함수가 intact로 남아 있어야 함을 요구한다. 만약 함수가 바뀌면 이것은 디버거의 존재를 드러내거나 어태치를 불허하는데 사용될 수 있다. 이 어태칭은 간단하게 브레이크포인트를 지움으로써 예방될 수 있다. 윈도우의 32비트 또는 64비트 버전에서 이 32비

트 코드를 사용해 패치를 만들 수 있다.

```
push offset l1
 call GetModuleHandleA
 push offset l2
 push eax
 call GetProcAddress
 push eax
 push esp
 push 40h                ; PAGE_EXECUTE_READWRITE
 push 1
 push eax
 xchg ebx, eax
 call VirtualProtect
 mov b [ebx], 0c3h
 ...
l1: db "ntdll", 0
l2: db "DbgBreakPoint", 0
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
 mov rcx, offset l1
 call GetModuleHandleA
 mov rdx, offset l2
 xchg rcx, rax
 call GetProcAddress
 push rax
 pop rbx
 enter 20h, 0
 push rbp
 pop r9
 push 40h                ; PAGE_EXECUTE_READWRITE
 pop r8
 xor edx, edx
 inc edx
 xchg rcx, rax
 call VirtualProtect
 mov b [rbx], 0c3h
 ...
```

```
l1: db "ntdll", 0
l2: db "DbgBreakPoint", 0
```

v. DbgPrint

적으로 ntdll DbgPrint() 함수는 예외를 발생시키지만 등록된 SEH는 알아차리지 못할 것이다. 이유는 윈도우가 자신의 SEH를 내부적으로 등록하기 때문이다. 디버거가 하지 않으면 이 핸들러가 예외를 받을 것이다. "일반적으로" 이 경우는 윈도우 2000 이후부터는 예외가 suppressed 된다는 사실을 의미한다.

윈도우 NT에서는 ntdll _vsnprintf() 함수가 후킹되어 있는 상태에서 ntdll DbgPrint() 함수를 호출한다면 문제가 발생한다. 이 경우 결과는 스택 오버플로우가 발생할 때까지 recursive 호출일 것이다. 이 버그는 TEB의 오프셋 0xf74에 "busy" 플래그를 추가함으로써 윈도우 2000부터 고쳐졌다. 이 함수는 플래그가 설정되면 즉시 종료할 것이다. 그러나 버그는 윈도우 2000에서 추가된 ntdll DbgPrintReturnControlC() 함수에서 재도입되었으며 또한 윈도우 XP에서도 존재한다. 이 버그는 윈도우 비스타에서 TEB의 오프셋 0xfca에 1비트의 "busy" 플래그를 추가함으로써 고쳐졌다(그러나 또한 TEB의 오프셋 0xf74에 있는 busy 플래그도 제거되었다).

윈도우 XP는 ntdll DbgPrintEx() 함수와 ntdll vDbgPrintEx() 함수를 도입하였는데 이것은 행동을 더 확장시켰다. 두 함수들은 파라미터로 severity level을 받아들인다. 만약 값이 0xffffffff라면 ntdll NtQueryDebugFilterState() 함수가 component ID와 severity level과 함께 호출된다. 만약 상응하는 엔트리가 커널 디버거 테이블에서 0이라면 함수는 즉시 반환한다.

윈도우 XP에는 더 심각한 버그가 존재하는데 이것은 ntdll vDbgPrintExWithPrefix() 함수가 prefix를 스택 버퍼에 복사하는 동안 이것의 길이를 검사하지 않는다는 것이다. 만약 문자열이 충분히 길다면 반환 주소(그리고 optionally SEH 레코드)는 대체될 수 있고 잠재적으로 임의적인 코드의 실행을 야기할 수 있다. 그러나 중요한 mitigrating factor는 이것이 내부 함수이며 직접적으로 호출될 수 없다는 것이다. ntdll vDbgPrintExWithPrefix() 함수를 호출하는 공용 함수들은 모두 빈 prefix를 사용한다. 윈도우 XP 이전에 고정 길이 복사가 수행되었었다. 대부분의 경우에 이것이 불필요할 정도로 크고 잠재적인 크래시 유발을 가지기 때문에 윈도우 XP에서 문자열 복사로 체되었다. 그러나 문자열의 길이는 복사를 수행하기 전에 검증되지 않았고 이에 따라 버퍼 오버플로우를 야기하였다. 이러한 종류의 버그는 윈도우 XP에서 적어도 한개의 다른 DLL에 존재하는 것으로 알려진다.

윈도우 비스타는 이 행위를 다시 변경하였다. 만약 디버거가 존재하고 severity level이 0x65라면 ntdll NtQueryDebugFilterState() 함수는 호출되지 않는다. 만약 severity level이 0xffffffff가 아니고 디버거가 존재하지 않거나 severity level이 0x65가 아니라면 ntdll NtQueryDebugFilterState() 함수는 component ID와 severity level과 함께 호출될 것

이다. 전처럼 상응하는 엔트리가 커널 디버거 테이블에서 0이면 함수는 즉시 반환한다. busy 플래그는 단지 이 지점에서 검사되며 플래그가 설정되어 있지 않다면 함수는 즉시 리턴한다.

만약 사용자 모드 디버거가 존재한다면(이것은 PEB의 BeingDebugged 플래그를 읽음으로써 결정된다), 또는 커널 모드 디버거가 윈도우 비스타와 이후에서 존재하지 않는다면(이것은 KUSER_SHARED_DATA 구조체의 오프셋 0x7ffe02d4의 KdDebuggerEnabled 멤버를 읽음으로써 결정된다) 윈도우는 DBG_PRINTEXCEPTION_C (0x40010006) 예외를 발생시킬 것이다. 만약 사용자 모드 디버거가 존재하지 않고, 커널 모드 디버거가 윈도우 비스타와 이후에서 존재한다면 윈도우는 인터럽트 0x2d(이것은 예외를 야기하지 않는다)를 실행하고 리턴할 것이다.

윈도우 XP와 이후에서는 예외가 발생한다면 어느 등록된 VEH라도 윈도우가 등록한 SEH보다 먼저 실행될 것이다. 이것은 윈도우에서 버그로 여겨진다. 최소한 이것은 특이한 실수이지만 궁극적으로 보면 이러한 세부 사항들 모두 만약 예외가 VEH에 전달되지 않는다면 디버거의 존재가 추론될 수 없다는 것을 의미한다.

vi. DbgSetDebugFilterState

비록 이것의 이름이 보여주긴 하지만, ntdll DbgSetDebugFilterState() 함수(또는 ntdll NtSetDebugFilterState() 함수, 이것들은 윈도우 XP에서 도입되었다)는 디버거가 존재하는 경우에 단순히 커널 모드 디버거에 의해 검사되는 테이블의 플래그를 설정하기 때문에 디버거의 존재를 탐지하는데 사용될 수 없다. 함수 호출이 몇몇 디버거들이 존재할 때 성공하는 것은 사실이지만 이것은 디버거의 행동의 부작용에 의한 것이며(구체적으로 디버그 권한을 요구하는) 디버거 자체에 의한 것이 아니다(이것은 함수 호출이 특정한 디버그들에서 사용될 때 성공하지 못하기 때문에 명백하다). 이것이 드러내는 모든 것들은 프로세스를 위한 사용자 계정이 관리자 그룹의 멤버이고 디버그 권한을 갖는다는 것이다. 이유는 함수의 성공이나 실패가 단지 프로세스 권한 수준에 제한적이기 때문이다. 만약 프로세스의 사용자 계정이 관리자 그룹의 멤버이고 디버그 권한을 갖는다면 함수 호출은 성공할 것이다; 만약 아니라면 실패한다. 이것은 표준 사용자가 디버그 권한을 요구하게 하는데 충분하지 않으며 관리자가 이것 없이 함수를 성공적으로 호출할 수 있게 하지도 못한다. 윈도우의 32비트 또는 64비트 버전들에서 filter 호출은 이 32비트 코드를 사용해서 만들어질 수 있다.

```
push 1
push 0
push 0
call NtSetDebugFilterState
xchg ecx, eax
jecxz admin_with_debug_priv
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
push 1
pop r8
xor edx, edx
xor ecx, ecx
call NtSetDebugFilterState
xchg ecx, eax
jecxz admin_with_debug_priv
```

vii. IsDebuggerPresent

kernel32 IsDebuggerPresent() 함수는 윈도우 95에서 도입되었다. 이것은 디버거가 존재하지 않는다면 0이 아닌 값을 반환한다. 32비트 또는 64비트 윈도우 환경을 검사하기 위한 검사는 이 코드(32비트와 64비트에서 같다)를 사용해서 만들어질 수 있다.

```
call IsDebuggerPresent
test al, al
jne being_debugged
```

내부적으로 이 함수는 간단하게 BeingDebugged 플래그의 값을 반환한다. 윈도우의 32비트 또는 64비트 버전들에서 32비트 윈도우 환경을 검사하는 것은 이 32비트 코드를 사용해서 만들어질 수 있다.

```
mov eax, fs:[30h]        ; Process Environment Block
cmp b [eax+2], 0         ; check BeingDebugged
jne being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
push 60h
pop rsi
gs:lodsq                 ; Process Environment Block
cmp b [rax+2], 0         ; check BeingDebugged
jne being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 32비트 코드를 사용할 수 있다.

```
mov eax, fs:[30h]        ; Process Environment Block
```

```
                        ; 64bit Process Environment Block
                        ; follows 32bit Process Environment Block
cmp b [eax+1002h], 0    ; check BeingDebugged
jne being_debugged
```

이러한 방식들을 방해하기 위해서는 단지 BeingDebugged 플래그를 0으로 만들면 된다.

viii. NtQueryInformationProcess

a. ProcessDebugPort

ntdll NtQueryInformationProcess() 함수는 질의할 정보의 클래스인 파라미터를 받아들인다. 대부분의 클래스들은 문서화되어 있지 않다. 하지만 문서화된 클래스들 중 하나는 ProcessDebugPort (7) 이다. 포트의 존재(값이 아니라)를 질의하는 것은 가능하다. 프로세스가 디버깅 중이지 않다면 반환 값은 0xffffffff이다. 내부적으로 함수는 EPROCESS 구조체에 있는 DebugPort 필드가 0이 아닌지를 질의한다. 이것은 내부적으로 kernel32 CheckRemoteDebuggerPresent() 함수가 동작하는 방법이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push eax
mov eax, esp
push 0
push 4                  ; ProcessInformationLength
push eax
push 7                  ; ProcessDebugPort
push -1                 ; GetCurrentProcess()
call NtQueryInformationProcess
pop eax
inc eax
je being_debugged
```

또는 64비트 윈도우 환경을 조사하는 이 64비트 코드를 사용할 수 있다.

```
xor ebp, ebp
enter 20h, 0
push 8                  ; ProcessInformationLength
pop r9
push rbp
pop r8
```

```
push 7                  ; ProcessDebugPort
pop rdx
or rcx, -1              ; GetCurrentProcess()
call NtQueryInformationProcess
leave
test ebp, ebp
jne being_debugged
```

이 정보는 커널에서 오기 때문에, 사용자 모드 코드가 이 호출이 디버거의 존재를 드러내는 것을 막기 위한 쉬운 방법은 존재하지 않는다.

b. ProcessDebugObjectHandle

윈도우 XP는 "debug object"를 도입하였다. 디버깅 세션이 시작하면 디버그 오브젝트는 생성되며 핸들은 이것과 어울린다. 문서화되지 않은 ProcessDebugObjectHandle (0x1e) 클래스를 사용해서 이 핸들의 값을 질의하는 것은 가능하다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
push 0
mov eax, esp
push 0  push 4          ; ProcessInformationLength
push eax
push 1eh                ; ProcessDebugObjectHandle
push -1                 ; GetCurrentProcess()
call NtQueryInformationProcess
pop eax
test eax, eax
jne being_debugged
```

또는 64비트 윈도우 환경을 조사하는 이 64비트 코드를 사용할 수 있다.

```
xor ebp, ebp
enter 20h, 0
push 8                  ; ProcessInformationLength
pop r9
push rbp
pop r8
push 1eh                ; ProcessDebugObjectHandle
pop rdx
or rcx, -1              ; GetCurrentProcess()
```

```
call NtQueryInformationProcess
leave
test ebp, ebp
jne being_debugged
```

이 정보는 커널에서 오기 때문에 사용자 모드 코드가 이 호출이 디버거의 존재를 드러내는 것을 막기 위한 쉬운 방법은 존재하지 않는다. 64비트 시스템에서 디버거가 존재할 때 디버그 오브젝트 핸들을 질의하는 것은 핸들의 참조 count를 증가시킬 수 있으며 그래서 디버기가 종료되지 않게 예방한다는 점을 주의하라.

c. ProcessDebugFlags

문서화되지 않은 ProcessDebugFlags (0x1f) 클래스는 EPROCESS 구조체의 NoDebugInherit 비트의 inverse 값을 반환한다. 즉 반환 값은 디버거가 존재하는 경우 0이 된다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 이 32비트 코드를 사용하여 만들어질 수 있다.

```
push eax
mov eax, esp
push 0
push 4                  ; ProcessInformationLength
push eax
push 1fh                ; ProcessDebugFlags
push -1                 ; GetCurrentProcess()
call NtQueryInformationProcess
pop eax
test eax, eax
je being_debugged
```

또는 64비트 윈도우 환경을 조사하는 이 64비트 코드를 사용할 수 있다.

```
xor ebp, ebp
enter 20h, 0
push 4                  ; ProcessInformationLength
pop r9
push rbp
pop r8
push 1fh                ; ProcessDebugFlags
pop rdx
or rcx, -1              ; GetCurrentProcess()
```

```
call NtQueryInformationProcess
leave
test ebp, ebp
je being_debugged
```

이 정보는 커널에서 오기 때문에 사용자 모드 코드가 이 호출이 디버거의 존재를 드러내는 것을 막기 위한 쉬운 방법은 존재하지 않는다.

ix. OutputDebugString

kernel32 OutputDebugString() 함수는 윈도우의 버전과 디버거가 존재하는지에 따라 또한 다른 행동을 입증할 수 있다. kernel32 GetLastError() 함수에 의해 반환된 값으로 표현되는 행동에서의 가장 명백한 차이는 만약 디버거가 존재한다면 바뀌지 않을 것이다(이것은 만약 항상 0으로 설정된다면 그런 경우가 존재하지 않는다). 그러나 이것은 단지 윈도우 NT/2000/XP에서만 적용된다. 윈도우 비스타와 그 이후에서 에러 코드는 모든 경우에 바뀌지 않는다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
xor ebp, ebp
mov fs:[ebp+34h], ebp ; LastErrorValue
push ebp
push esp
call OutputDebugStringA
cmp fs:[ebp+34h], ebp ; LastErrorValue
je being_debugged
```

또는 64비트 윈도우 환경을 조사하는 이 64비트 코드를 사용할 수 있다.

```
xor ebp, ebp
mov gs:[rbp+68h], ebp ; LastErrorValue
enter 20h, 0
mov ecx, ebp
call OutputDebugStringA
cmp gs:[rbp+68h], ebp ; LastErrorValue
je being_debugged
```

이것이 통했던 이유는 윈도우가 "DBWIN_BUFFER"라고 불리는 객체에 한 매핑을 여려고 시도했기 때문이다. 이것이 실패했을 때 에러 코드가 설정된다. 위에서 언급했듯이 디버거가 존재하면 예외는 디버거에 의해 받아들여 지고 에러 코드는 호출되었

던 kernel32 OutputDebugString() 함수 전에 가졌던 값에 저장된다. 만약 디버거가 존재하지 않는다면 예외는 윈도우에 의해 처리되고 에러 코드는 유지될 것이다. 그러나 윈도우 비스타와 이후에서 에러 코드는 호출되었던 kernel32 OutputDebugString() 함수 전에 가졌던 값에 저장된다. 이것은 명시적으로 cleared 되지 않으며 이 탐지 기법을 완전히 신뢰할 수 없게 한다.

이 함수는 이것의 사용으로 인해 생기는 OllyDbg v1.10에서의 버그 때문에 아마 가장 잘 알려져 있을 것이다. OllyDbgsms userdefined 데이터를 직접적으로 msvcrt _vsprintf() 함수로 보낸다. 이러한 데이터는 string-formatting token들을 포함한다. 특정한 위치의 특정한 토큰은 전달된 파라미터들 중 하나를 가지고 함수가 메모리에 접근하게 만든다. 공격의 수많은 변형들이 존재하며 이것들 모두 근본적으로는 동작하게 하는 랜덤하게 선택된 토큰 결합들이다. 그러나 모든 요구되는 것들은 세 토큰들이다. 첫번째 두 토큰들은 완전히 임의적이다. 세번째 토큰은 반드시 "%s"여야 한다. 이것은 _vsprintf() 함수가 __vprinter() 함수를 호출하고 0을 네번째 파라미터로서 전달하기 때문이다. "%s"이 거기서 쓰인다면 네번째 파라미터는 세번째 토큰에 의해 접근된다. 결과는 널 포인터 접근과 크래시이다. 버그는 임의의 코드를 실행할 수 있게 익스플로잇될 수 없다.

x. RtlQueryProcessHeapInformation

ntdll RtlQueryProcessHeapInformation() 함수는 현재 프로세스의 프로세스 메모리로부터 힙 플래그들을 읽기 위해 사용될 수 있다. 서브시스템 버전이 3.103.50 사이라면 윈도우의 32비트 또는 64비트 버전에서 32비트 윈도우 환경을 조사하는 검사는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push 0
push 0
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
call RtlQueryProcessHeapInformation
mov eax, [ebx+38h]      ; HeapInformation
mov eax, [eax+8]        ; Flags
                        ; neither CREATE_ALIGN_16
                        ; nor HEAP_SKIP_VALIDATION_CHECKS
and eax, 0effeffffh     ; GROWABLE
                        ; + TAIL_CHECKING_ENABLED
                        ; + FREE_CHECKING_ENABLED
                        ; + VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000062h
```

```
je being_debugged
```

또는 서브시스템 버전이 3.51 이상인 경우 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push 0
push 0
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
call RtlQueryProcessHeapInformation
mov eax, [ebx+38h]      ; HeapInformation
mov eax, [eax+8]        ; Flags
bswap eax               ; not HEAP_SKIP_VALIDATION_CHECKS
and al, 0efh            ; GROWABLE
                        ;+ TAIL_CHECKING_ENABLED
                        ;+ FREE_CHECKING_ENABLED
                        ;+ VALIDATE_PARAMETERS_ENABLED
cmp eax, 62000040h
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위한 이 64비트 코드를 사용할 수 있다.

```
xor edx, edx
xor ecx, ecx
call RtlCreateQueryDebugBuffer
mov ebx, eax
xchg ecx, eax
call RtlQueryProcessHeapInformation
mov eax, [rbx+70h]      ; HeapInformation
                        ; Flags
                        ; GROWABLE
                        ;+ TAIL_CHECKING_ENABLED
                        ;+ FREE_CHECKING_ENABLED
                        ;+ VALIDATE_PARAMETERS_ENABLED
cmp d [rax+10h], 40000062h
je being_debugged
```

xi. NtQueryVirtualMemory

ntdll NtQueryVirtualMemory() 함수는 파일로 부터의 페이지에 위치한 브레이크 포인트를 탐지하는데 사용될 수 있다. 파일에서의 페이지가 써질 때 윈도우는 페이지의 복사본을 만들고 복사본에 변화를 준다. 내부 페이지 속성들 중에서 한 비트는 페이지가 변경되었기 때문에 이 페이지가 더 이상 공유 가능하지 않다는 것을 알리기 위해 클리어된다. 32비트 또는 64비트 윈도우에서 32비트 윈도우 환경을 조사하기 위한 검사는 다음의 코드를 사용해서 만들어질 수 있다(32비트와 64비트에서 같다).

```
 xor ebx, ebx
 xor ebp, ebp
 jmp l2
l1: push 8000h          ; MEM_RELEASE
 push ebp
 push esi
 call VirtualFree
l2: xor eax, eax
 mov ah, 10h            ; MEM_COMMIT
 add ebx, eax           ; 4kb increments
 push 4                 ; PAGE_READWRITE
 push eax
 push ebx
 push ebp
 call VirtualAlloc      ; function does not return required length
 push ebp               ; must calculate by brute-force
 push ebx
 push eax
 push 1                 ; MemoryWorkingSetList
 push ebp
 push -1                ; GetCurrentProcess()
 xchg esi, eax
 call NtQueryVirtualMemory
                        ; presumably STATUS_INFO_LENGTH_MISMATCH
 test eax, eax
 jl l1
 lodsd
l3: lodsd
 mov dl, ah
 and ax, 0f000h
```

```
cmp eax, offset l3 and 0fffff000h
jne l3
test dl, 1                ; WSLE_PAGE_SHAREABLE
je being_debugged
```

xii. RtlQueryProcessDebugInformation

ntdll RtlQueryProcessDebugInformation() 함수는 요청된 프로세스의 프로세스 메모리로부터 힙 플래그들을 포함한 특정한 필드들을 읽는데 사용될 수 있다. 함수는 힙 플래그들을 위해 내부적으로 ntdll RtlQueryProcessHeapInformation() 함수를 호출함으로써 수행한다. 서브시스템 버전이 3.103.50 사이라면 윈도우의 32비트 또는 64비트 버전에서 32비트 윈도우 환경을 조사하는 검사는 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
xor ebx, ebx
push ebx
push ebx
call RtlCreateQueryDebugBuffer
push eax
xchg ebx, eax
push 14h                  ; PDI_HEAPS + PDI_HEAP_BLOCKS
push d fs:[eax+20h]       ; UniqueProcess call RtlQueryProcessDebugInformation
mov eax, [ebx+38h]        ; HeapInformation
mov eax, [eax+8]          ; Flags
                          ; neither CREATE_ALIGN_16
                          ; nor HEAP_SKIP_VALIDATION_CHECKS
and eax, 0effeffffh       ; GROWABLE
                          ;+ TAIL_CHECKING_ENABLED
                          ;+ FREE_CHECKING_ENABLED
                          ;+ VALIDATE_PARAMETERS_ENABLED
cmp eax, 40000062h
je being_debugged
```

또는 서브시스템 버전이 3.51 이상이라면 윈도우의 32비트 또는 64비트 버전에서 32비트 윈도우 환경을 조사하는 검사는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
xor ebx, ebx
push ebx
push ebx
call RtlCreateQueryDebugBuffer
```

```
push eax
push 14h                ; PDI_HEAPS + PDI_HEAP_BLOCKS
xchg ebx, eax
push d fs:[eax+20h]     ; UniqueProcess
call RtlQueryProcessDebugInformation
mov eax, [ebx+38h]      ; HeapInformation
mov eax, [eax+8]        ; Flags
bswap eax               ; not HEAP_SKIP_VALIDATION_CHECKS
and al, 0efh            ; GROWABLE
                        ;+ TAIL_CHECKING_ENABLED
                        ;+ FREE_CHECKING_ENABLED
                        ;+ VALIDATE_PARAMETERS_ENABLED
                        ; reversed by bswap
cmp eax, 62000040h
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
xor edx, edx
xor ecx, ecx
call RtlCreateQueryDebugBuffer
push rax
pop r8
push 14h        ; PDI_HEAPS + PDI_HEAP_BLOCKS
pop rdx
mov ecx, gs:[rdx+2ch] ; UniqueProcess
xchg ebx, eax
call RtlQueryProcessDebugInformation
mov eax, [rbx+70h]  ; HeapInformation
                        ; Flags
                        ; GROWABLE
                        ;+ TAIL_CHECKING_ENABLED
                        ;+ FREE_CHECKING_ENABLED
                        ;+ VALIDATE_PARAMETERS_ENABLED
cmp d [rax+10h], 40000062h
je being_debugged
```

xiii. SwitchToThread

kernel32 SwitchToThread() 함수는 (또는 ntdll NtYieldExecution()) 현재 스레드가 자신의 남은 시간 슬라이스를 포기하는 것을 제공하며 다음으로 스케줄된 스레드를 실행시킨다. 만약 실행할 스레드로 스케줄된 것이 없다면(또는 시스템이 특정한 방식으로 바쁘고 switch가 발생하는 것을 허락하지 않을 때), ntdll NtYieldExecution() 함수는 STATUS_NO_YIELD_PERFORMED (0x40000024) 상태를 반환하는데 이것은 kernel32 SwitchToThread() 함수가 0을 반환하게 한다. 애플리케이션이 디버깅될 때 코드를 지나는 single-stepping의 행동은 디버그 이벤트들을 유발시키며 가끔은 no yield가 허용된다. 그러나 이것이 또한 높은 우선 순위에서 실행되는 스레드의 존재도 탐지하기 때문에 이것은 절망적이게도 디버거를 탐지하는데 신뢰할 수 없는 방법이다. 이 검사는 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 push 20h
 pop ebp
l1: push 0fh
 call Sleep
 call SwitchToThread
 cmp al, 1
 adc ebx, ebx
 dec ebp
 jne l1
 inc ebx                    ; detect 32 nonyields
 je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위해 다음의 64비트 코드를 사용할 수 있다.

```
 push 20h
 pop rbp
l1: push 0fh
 pop rcx
 call Sleep
 call SwitchToThread
 cmp al, 1
 adc ebx, ebx
 dec ebp
 jne l1
 inc ebx                    ; detect 32 nonyields
 je being_debugged
```

xiv. Toolhelp32ReadProcessMemory

윈도우 2000에서 소개된 kernel32 Toolhelp32ReadProcessMemory() 함수는 프로세스가 다른 프로세스의 메로리를 열고 읽을 수 있게 한다. 이것은 kernel32 OpenProcess() 함수와 kernel32 ReadProcessMemory() 함수(또는 kernel32 CloseHandle() 함수)를 결합한다. 이 함수가 현재 어떻게 현재 프로세스로부터 데이터를 복사하는지에 해서는 부정확하게 문서화되었다는 점에 주의해야 한다. 이 함수는 현재 프로세스의 메모리를 읽기 위해 프로세스 ID를 0으로 받아들인다고 문서화(http://msdn.microsoft.com/en-us/library/ms686826(VS.85).aspx)되어 있지만 이것은 사실이 아니다. 프로세스 ID는 반드시 항상 유효해야 한다. 현재 프로세스로부터 읽기 위해서, 프로세스 ID는 반드시 kernel32 GetCurrentProcessId() 함수로부터 반환된 값이어야 한다. 이 함수는 함수 호출 이후에 브레이크포인트를 검사함으로써 stepover condition을 탐지하는 또다른 방식으로 사용될 수 있다. 호출은 32비트 또는 64비트 윈도우 버전에서 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
 push eax
 mov eax, esp
 xor ebx, ebx
 push ebx
 inc ebx
 push ebx
 push eax
 push offset l1
 push d fs:[ebx+1fh]     ; UniqueProcess
 call ToolhelpReadProcessMemory
l1: pop eax
 cmp al, 0cch
 je being_debugged
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
 xor ebp, ebp
 enter 20h, 0
 push 1
 pop r9
 push rbp
 pop r8
 mov rdx, offset l1
 mov ecx, gs:[rbp+40h]            ; UniqueProcess
```

```
 call Toolhelp32ReadProcessMemory
l1: leave
 cmp bpl, 0cch
 je being_debugged
```

xv. UnhandledExceptionFilter

예외가 발생하고 등록된 예외 핸들러가 없거나(Structured나 Vectored 모두) 아무 등록된 핸들러들도 예외를 처리할 수 없다면 가장 마지막에 kernel32 UnhandledExceptionFilter()이 호출될 것이다. 만약 디버거가 존재하지 않는다면(이것은 ntdll NtQueryInformationProcess() 함수를 ProcessDebugPort 클래스와 함께 호출함으로써 결정된
다) kernel32 SetUnhandledExceptionFilter() 함수에 의해 등록된 핸들러가 호출될 것이다. 만약 디버거가 존재한다면 호출은 거기에 이르지 못할 것이다. 대신 예외는 디버거에 전달된다. 함수는 ntdll NtQueryInformationProcess 함수를 ProcessDebugPort 클래스와 함께 호출함으로써 디버거의 존재를 결정한다. 놓친 예외는 디버거의 존재를 추론하는데 사용될 수 있다. 윈도우의 32비트 또는 64비트 버전에서 호출은 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push offset l1
call SetUnhandledExceptionFilter          ; force an exception to occur
int 3
jmp being_debugged
l1:                                       ; execution resumes here if exception occurs
...
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다(그러나 64비트 프로세스들에서는 이 기법이 같은 방식으로 통하지 않는다).

```
 mov rcx, offset l1
 call SetUnhandledExceptionFilter  ; force an exception to occur
 int 3
 jmp being_debugged
l1:                                ; execution resumes here if exception occurs
...
```

xvi. VirtualProtect

kernel32 VirtualProtect() 함수는 (또는 kernel32 VirtualProtectEx() 또는 ntdll NtProtectVirtualMemory() 함수) "guard" 페이지들을 할당하기 위해 사용될 수 있

다. Guard 페이지들은 처음 그들이 접근될 때 예외를 유발시키는 페이지들이다. 이것들은 일반적으로 회복 불능이 되기 전에 잠재적인 위험을 가로채기 위해 스택의 바닥에 위치한다. Guard 페이지들은 또한 디버거를 탐지하는데 사용될 수 있다. 두 예비 단계는 예외 핸들러를 등록하고 guard 페이지를 할당하는 것이다. 이 단계들의 순서는 중요치 않다. optional이지만 일반적으로 페이지는 몇몇 content가 위치하게 하기 위해 쓰기 가능하고 읽기 가능하게 초기에 할당된다. 다음 단계는 guard 페이지에서 어떤 것을 실행하려고 시도하는 것이다. 이것은 예외 핸들러에 의해 받아지는 EXCEPTION_GUARD_PAGE (0x80000001) 예외를 초래한다. 그러나 디버거가 존재한다면 디버거는 예외를 가로채고 실행이 계속되게 한다. 이 행위는 OllyDbg에서 발생하는 것으로 알려져 있다. 호출은 32비트 또는 64비트 윈도우 버전에서 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
xor ebx, ebx
push 40h                ; PAGE_EXECUTE_READWRITE
push 1000h              ; MEM_COMMIT
push 1
push ebx
call VirtualAlloc
mov b [eax], 0c3h
push eax
push esp
push 140h               ; PAGE_EXECUTE_READWRITE+PAGE_GUARD
push 1
push eax
xchg ebp, eax
call VirtualProtect
push offset l1
push d fs:[ebx]
mov fs:[ebx], esp
push offset being_debugged  ; execution resumes at being_debugged
                            ; if ret instruction is executed
jmp ebp
l1:                         ; execution resumes here if exception occurs
```

또는 윈도우의 64비트 버전들에서 다음의 64비트를 사용할 수 있다(OllyDbg가 윈도우의 64비트 버전에서 돌아가지 않기 때문에 비록 이것은 OllyDbg 외의 디버거들에 적용되지만).

```
push 40h                ; PAGE_EXECUTE_READWRITE
pop r9
mov r8d, 1000h          ; MEM_COMMIT
```

```
push 1
pop rdx
xor ecx, ecx
call VirtualAlloc
mov b [rax], 0c3h
push rax
pop rbx
enter 20h, 0
push rbp
pop r9                  ; PAGE_EXECUTE_READWRITE+PAGE_GUARD
mov r8d, 140h
push 1
pop rdx
xchg rcx, rax
call VirtualProtect
mov rdx, offset l1
xchg ecx, eax
call AddVectoredExceptionHandler
push offset being_debugged  ;execution resumes at being_debugged
                            ;if ret instruction is executed
jmp rbx
l1:  ;execution resumes here if exception occurs
 ...
```

E. System-level

FindWindow
NtQueryObject
NtQuerySystemInformation

i. FindWindow

user32 FindWindow() 함수는 이름이나 클래스에 따라 윈도우를 검색하는데 사용될 수 있다. 디버거가 GUI를 가질 시에 디버거의 존재를 탐지하는데 쉬운 방법이다. 예를들면 OllyDbg는 찾을 클래스 이름으로 "OLLYDBG"을 넘겨줌으로써 발견될 수 있다. WinDbg는 찾을 클래스 이름으로 "WinDbgFrameClass"을 넘겨줌으로써 발견될 수 있다. 윈도우의 32비트 또는 64비트 버전들에서 이러한 툴들의 존재는 다음의 32비트 코드를 사용해서 검사될 수 있다.

```
  mov edi, offset l2
l1: push 0
  push edi
  call FindWindowA
  test eax, eax
  jne being_debugged
  or ecx, -1
  repne scasb
  cmp [edi], al
  jne l1
  ...
l2: <array of ASCIIZ strings, zero to end>
```

또는 위도우의 64비트 버전들에서 다음의 64비트 코드를 사용할 수 있다.

```
  mov rdi, offset l2
l1: xor edx, edx
  push rdi
  pop rcx
  call FindWindowA
  test eax, eax
  jne being_debugged
  or ecx, -1
  repne scasb
  cmp [rdi], al
  jne l1
  ...
l2: <array of ASCIIZ strings, zero to end>
```

일반적인 리스트는 다음의 이름들을 포함한다.

```
db "OLLYDBG", 0
db "WinDbgFrameClass", 0       ; WinDbg
db "ID", 0                     ; Immunity Debugger
db "Zeta Debugger", 0
db "Rock Debugger", 0
db "ObsidianGUI", 0
db 0
```

ii. NtQueryObject

윈도우 XP는 "debug object"를 도입하였다. 디버깅 세션이 시작될 때 debug object가 생성되면 핸들은 이것과 연관된다(associated). ntdll NtQueryObject() 함수를 사용함으로써 존재하는 객체들의 리스트를 질의하고 존재하는 debug object와 연관된 핸들들의 수를 질의하는 것이 가능하다(이 함수는 어떤 윈도우 NT 기반 플랫폼에서도 호출될 수 있지만 윈도우 XP와 이후에서는 리스트에서 debug object를 가질 것이다). 32비트 또는 64비트 윈도우 버전에서 이 검사는 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어 질 수 있다.

```
  xor ebx, ebx
  xor ebp, ebp
  jmp l2
l1: push 8000h          ; MEM_RELEASE
  push ebp
  push esi
  call VirtualFree
l2: xor eax, eax
  mov ah, 10h           ; MEM_COMMIT
  add ebx, eax          ; 4kb increments
  push 4                ; PAGE_READWRITE
  push eax
  push ebx
  push ebp
  call VirtualAlloc     ; function does not return required length
                        ; for this class in Windows Vista and later
  push ebp              ; must calculate by bruteforce
  push ebx
  push eax
  push 3                ; ObjectAllTypesInformation
  push ebp
  xchg esi, eax
  call NtQueryObject    ; presumably STATUS_INFO_LENGTH_MISMATCH
  test eax, eax
  jl l1  lodsd          ; handle count
  xchg ecx, eax
l3: lodsd               ; string lengths
  movzx edx, ax         ; length
  lodsd                 ; pointer to TypeName
```

```
  xchg esi, eax             ; sizeof(L"DebugObject")
                            ; avoids superstrings
                            ; like "DebugObjective"
  cmp edx, 16h
  jne l4
  xchg ecx, edx
  mov edi, offset l5
  repe cmpsb
  xchg ecx, edx
  jne l4                    ; checking TotalNumberOfHandles
                            ; works only on Windows XP
                            ; cmp [eax], edx          ; TotalNumberOfHandles
                            ; check TotalNumberOfObjects instead
  cmp [eax+4], edx          ; TotalNumberOfObjects
  jne being_debugged
l4: lea esi, [esi+edx+4]    ; skip null and align
  and esi, -4               ; round down to dword
  loop l3
  ...
l5: dw "D","e","b","u","g"
  dw "O","b","j","e","c","t"
```

또는 64비트 윈도우 환경을 조사하는 이 64비트 코드를 사용할 수 있다.

```
  xor ebx, ebx
  xor ebp, ebp
  jmp l2
l1: mov r8d, 8000h          ; MEM_RELEASE
  xor edx, edx
  mov ecx, esi
  call VirtualFree
l2: xor eax, eax
  mov ah, 10h               ; MEM_COMMIT
  add ebx, eax              ; 4kb increments
  push 4                    ; PAGE_READWRITE
  pop r9
  push rax
  pop r8
  mov edx, ebx
```

```
  xor ecx, ecx
  call VirtualAlloc       ; function does not return required length
                          ; for this class in Windows Vista and later
  enter 20h, 0            ; must calculate by bruteforce
  push rbx
  pop r9
  push rax
  pop r8
  push 3                  ; ObjectAllTypesInformation
  pop rdx
  xor ecx, ecx
  xchg esi, eax
  call NtQueryObject
  leave                   ; presumably STATUS_INFO_LENGTH_MISMATCH
  test eax, eax
  jl l1
  lodsq                   ; handle count
  xchg ecx, eax
l3: lodsq                 ; string lengths
  movzx edx, ax           ; length
  lodsq                   ; pointer to TypeName
  xchg esi, eax           ; sizeof(L"DebugObject")
                          ; avoids superstrings
                          ; like "DebugObjective"
  cmp edx, 16h
  jne l4
  xchg ecx, edx
  mov rdi, offset l5
  repe cmpsb
  xchg ecx, edx
  jne l4                  ; checking TotalNumberOfHandles
                          ; works only on Windows XP
                          ; cmp [rax], edx   ; TotalNumberOfHandles
                          ; check TotalNumberOfObjects instead
  cmp [rax+4], edx        ; TotalNumberOfObjects
  jne being_debugged
l4: lea esi, [rsi+rdx+8]  ; skip null and align
  and esi, -8             ; round down to dword
  loop l3
```

```
  ...
l5: dw "D","e","b","u","g"
  dw "O","b","j","e","c","t"
```

이 정보는 커널에서 오므로 이 호출이 디버거의 존재를 드러내는 것을 막기 위해 사용자 모드 코드를 사용하는 쉬운 방법은 존재하지 않는다.

iii. NtQuerySystemInformation

a. SystemKernelDebuggerInformation

ntdll NtQuerySystemInformation() 함수는 질의할 정보의 클래스를 파라미터로 받는다. 대부분의 클래스들은 문서화되어 있지 않다. 이것은 SystemKernelDebuggerInformation (0x23) 클래스를 포함하는데 윈도우 NT부터 존재했다. SystemKernelDebuggerInformation 클래스는 두 플래그들의 값을 반환한다 : al에는 KdDebuggerEnabled, ah에는 KdDebuggerNotPresent. 그래서 ah에의 반환 값은 디버거가 존재하는 경우 0이다. 호출은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
push eax
mov eax, esp
push 0
push 2                  ; SystemInformationLength
push eax
push 23h                ; SystemKernelDebuggerInformation
call NtQuerySystemInformation
pop eax
test ah, ah
je being_debugged
```

또는 64비트 윈도우 환경을 조사하는 이 64비트 코드를 사용할 수 있다.

```
push rax
mov edx, esp
xor r9, r9
push 2                  ; SystemInformationLength
pop r8
push 23h                ; SystemKernelDebuggerInformation
pop rcx
call NtQuerySystemInformation
```

```
pop rax
test ah, ah
je being_debugged
```

이 정보는 커널에서 오므로 이 호출이 디버거의 존재를 드러내는 것을 막기 위해 사용자 모드 코드를 사용하는 쉬운 방법은 존재하지 않는다. 이 함수는 입력 크기에 관계없이 단지 두 바이트만 쓴다. 이 작은 크기는 흔치 않으며 이러한 툴들이 일반적으로 목적지에 4 바이트를 쓰기 때문에 몇몇 숨겨진 툴들의 존재를 드러낼 수 있다.

이 함수는 간단하게 오프셋 0x7ffe02d에서 KUSER_SHARED_DATA 구조체의 KdDebuggerEnabled 필드에서 직접적으로 값을 검색함으로써 더 나아가 난독화될 수 있다. 이 값은 윈도우의 모든 32비트 그리고 64비트 버전들에서 사용 가능하다. 흥미롭게도 이 값은 독립된 위치에서 온 함수 호출에 의해 반환되는데 그래서 디버거를 숨기려는 어떤 툴도 두 위치들에서 값을 patch할 필요가 있다.

b. SystemProcessInformation

Explorer.exe와 현재 프로세스의 부모 프로세스 모두의 프로세스 ID와 부모 프로세스의 이름은 SystemProcessInformation (5) 클래스와 함께 ntdll NtQuerySystemInformation() 함수에서 얻을 수 있다. 함수에 대한 단일 호출은 실행중인 프로세스들의 전체 리스트를 반환하며 이것은 반드시 그 후에 직접적으로 파싱되어야 한다. 이것은 kernel32 CreateToolhelp32Snapshot() 함수가 내부적으로 호출하는 함수이다. kernel32 CreateToolhelp32Snapshot() 함수 알고리즘과 함께 사용자 이름과 도메인 이름은 반드시 잘못된 매칭을 피하기 위해 그리고 잠재적으로 루틴의 pass들의 수를 감소시키기 위해 검사되어야 한다. 윈도우의 32비트 또는 64비트 버전에서 32비트 윈도우 환경을 검사하는 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
  xor ebp, ebp
  xor esi, esi
  xor edi, edi
  jmp l2
l1: push 8000h          ; MEM_RELEASE
  push ebp
  push ebx
  call VirtualFree
l2: xor eax, eax
  mov ah, 10h           ; MEM_COMMIT
  add esi, eax          ; 4kb increments
  push 4                ; PAGE_READWRITE
  push eax
```

```
  push esi
  push ebp
  call VirtualAlloc         ; function does not return
                            ; required length for this class
  push ebp                  ; must calculate by bruteforce
  push esi
  push eax
  push 5                    ; SystemProcessInformation
  xchg ebx, eax
  call NtQuerySystemInformation  ; presumably STATUS_INFO_LENGTH_MISMATCH
  test eax, eax
  jl l1
  push ebx
  push ebx
l3: push ebx
  mov eax, fs:[20h]         ; UniqueProcess
  cmp [ebx+44h], eax        ; UniqueProcessId
                            ; InheritedFromUniqueProcessId
  cmove edi, [ebx+48h]
  test edi, edi
  je l4
  cmp ebp, edi
  je l11
l4: mov ecx, [ebx+3ch]  ; ImageName
  jecxz l9
  xor eax, eax
  mov esi, ecx
l5: lodsw
  cmp eax, "₩"
  cmove ecx, esi
  push ecx
  push eax
  call CharLowerW
  mov [esi-2], ax
  pop ecx
  test eax, eax
  jne l5
  sub esi, ecx
  xchg ecx, esi
```

```
  push edi
  mov edi, offset l17
  repe cmpsb
  pop edi
  jne l9
  mov eax, [ebx+44h]    ; UniqueProcessId
  push ebx
  push ebp
  push edi
  call l12
  dec ecx               ; invert Z flag
  jne l7
  push ebx
  push edi
  dec ecx
  call l13
  pop esi
  pop edx
  mov cl, 2             ; compare user names
                        ; then domain names
l6: lodsb
  scasb
  jne l7
  test al, al
  jne l6
  mov esi, ebx
  mov edi, edx
  loop l6
l7: pop edi
  pop ebp
  pop ebx
  jne l9
  test ebp, ebp
  je l8
  mov esi, offset l18
  cmp cl, [esi]
  adc [esi], ecx
l8: mov ebp, [ebx+44h] ; UniqueProcessId
l9: pop ebx
```

```
  mov ecx, [ebx]          ; NextEntryOffset
  add ebx, ecx
  inc ecx
  loop l10
  pop ebx
  dec b [offset l18+1]
l10:jne l3                ; and possibly one pointer left on stack
                          ; add esp, b [offset l18]*4
  jmp being_debugged      ; and at least one pointer left on stack
                          ; add esp, (b [offset l18+1]b [offset l18]+1)*4
l11:...
l12:push eax
  push 0
  push 400h               ; PROCESS_QUERY_INFORMATION
  call OpenProcess
  xchg ecx, eax
  jecxz l16
l13:push eax
  push esp
  push 8                  ; TOKEN_QUERY
  push ecx
  call OpenProcessToken
  pop ebx
  xor ebp, ebp
l14:push ebp
  push 0                  ; GMEM_FIXED
  call GlobalAlloc
  push eax
  push esp
  push ebp
  push eax
  push 1                  ; TokenUser
  push ebx
  xchg esi, eax
  call GetTokenInformation
  pop ebp
  xchg ecx, eax
  jecxz l14
  xor ebp, ebp
```

```
l15:push ebp
  push 0                    ; GMEM_FIXED
  call GlobalAlloc
  xchg ebx, eax
  push ebp
  push 0                    ; GMEM_FIXED
  call GlobalAlloc
  xchg edi, eax
  push eax
  mov eax, esp
  push ebp
  mov ecx, esp
  push ebp
  mov edx, esp
  push eax
  push ecx
  push ebx
  push edx
  push edi
  push d [esi]
  push 0
  call LookupAccountSidA
  pop ebp
  pop ecx
  xchg ebp, ecx
  pop edx
  xchg ecx, eax
  jecxz l15
l16:ret
l17:dw "e", "x", "p", "l", "o", "r", "e", "r"
  dw ".", "e", "x", "e", 0
l18:db 0ffh, 1, ?, ?
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
  xor ebp, ebp
  xor esi, esi
  xor edi, edi
  jmp l2
```

```
l1: mov r8d, 8000h      ; MEM_RELEASE
  xor edx, edx
  mov ecx, ebx
  call VirtualFree
l2: xor eax, eax
  mov ah, 10h           ; MEM_COMMIT
  add esi, eax          ; 4kb increments
  push 4                ; PAGE_READWRITE
  pop r9
  push rax
  pop r8
  mov edx, esi
  xor ecx, ecx
  call VirtualAlloc     ; function does not return
                        ; required length for this class
  xor r9d, r9d          ; must calculate by brute-force
  push rsi
  pop r8
  mov edx, eax
  push 5                ; SystemProcessInformation
  pop rcx
  xchg ebx, eax
  call NtQuerySystemInformation         ; presumably STATUS_INFO_LENGTH_MISMATCH
  test eax, eax
  jl l1
  push rbx
  push rbx
l3: push rbx
  call GetCurrentProcessId
  cmp [rbx+50h], eax    ; UniqueProcessId
                        ; InheritedFromUniqueProcessId
  cmove edi, [rbx+58h]
  test edi, edi
  je l4
  cmp ebp, edi
  je l11
l4: mov ecx, [rbx+40h] ; ImageName
  jrcxz l9
  xor eax, eax
```

```
  mov esi, ecx
l5: lodsw
  cmp eax, "₩"
  cmove ecx, esi
  push rcx
  xchg ecx, eax
  enter 20h, 0
  call CharLowerW
  leave
  mov [rsi-2], ax
  pop rcx
  test eax, eax
  jne l5
  sub esi, ecx
  xchg ecx, esi
  push rdi
  mov rdi, offset l17
  repe cmpsb
  pop rdi
  jne l9
  mov r8d, [rbx+50h]    ; UniqueProcessId
  push rbx
  push rbp
  push rdi
  call l12
  dec ecx               ; invert Z flag
  jne l7
  push rbx
  push rdi
  dec rcx
  call l13
  pop rsi
  pop rdx
  inc ecx               ; compare user names
                        ; then domain names
l6: lodsb
  scasb
  jne l7
  test al, al
```

```
  jne l6
  mov esi, ebx
  mov edi, edx
  loop l6
l7: pop rdi
  pop rbp
  pop rbx
  jne l9
  test ebp, ebp
  je l8
  mov rsi, offset l18
  cmp cl, [rsi]
  adc [rsi], ecx
l8: mov ebp, [rbx+50h] ; UniqueProcessId
l9: pop rbx
  mov ecx, [rbx]         ; NextEntryOffset
  add ebx, ecx
  inc ecx
  loop l10
  pop rbx
  dec byte [offset l18+1]
l10:jne l3               ; and possibly one pointer left on stack
                         ; add esp, -b [offset l18]*4
  jmp being_debugged
                         ; and at least one pointer left on stack
                         ; add esp, (b [offset l18+1]-b [offset l18]+1)*4
l11:...
l12:cdq
  xor ecx, ecx
  mov ch, 4              ; PROCESS_QUERY_INFORMATION
  enter 20h, 0
  call OpenProcess
  leave
  xchg ecx, eax
  jrcxz l16
l13:push rax
  mov r8d, esp
  push 8                 ; TOKEN_QUERY
  pop rdx
```

```
  call OpenProcessToken
  pop rbx
  xor ebp, ebp
l14:mov edx, ebp
  xor ecx, ecx              ; GMEM_FIXED
  enter 20h, 0
  call GlobalAlloc
  leave
  push rbp
  pop r9
  push rax
  pop r8
  push rax                  ; simulate enter
  mov ebp, esp
  push rbp
  sub esp, 20h
  push 1                    ; TokenUser
  pop rdx
  mov ecx, ebx
  xchg esi, eax
  call GetTokenInformation
  leave
  xchg ecx, eax
  jrcxz l14
  xor ebp, ebp
l15:mov ebx, ebp
  mov edx, ebp
  xor ecx, ecx              ; GMEM_FIXED
  enter 20h, 0
  call GlobalAlloc
  xchg ebx, eax
  xchg edx, eax
  xor ecx, ecx              ; GMEM_FIXED
  call GlobalAlloc
  leave
  xchg edi, eax
  push rbp
  mov ecx, esp
  push rbp
```

```
  mov r9d, esp
  push rax
  push rsp
  push rcx
  push rbx
  sub esp, 20h
  push rdi
  pop r8
  mov edx, [rsi]
  xor ecx, ecx
  call LookupAccountSidA
  add esp, 40h
  pop rcx
  pop rbp
  xchg ecx, eax
  jrcxz l15
l16:ret
l17:dw "e","x","p","l","o","r","e","r"
  dw ".","e","x","e",0
l18:db 0ffh, 1, ?, ?
```

iv.Selectors

Selector 값들은 안정적으로 보일 수 있지만 실제로는 특정한 상황에서 불안하며 또한 윈도우의 버전에 의존한다. 예를들면 셀렉터 값은 스레드 내에서 설정될 수 있지만 오랫 동안 값을 갖지는 않는다. 특정한 이벤트들은 셀렉터 값을 이것의 기본 값으로 변하게 만든다. 이러한 이벤트 중 하나로 예외가 있다. 디버거의 문맥에서 single-step 예외는 계속 예외이며 이것은 몇몇 예상치 못한 행동을 유발할 수 있다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하기 위한 다음의 32비트 코드를 사용하여 만들어 질 수 있다.

```
xor eax, eax
push fs
pop ds
l1: xchg [eax], cl
xchg [eax], cl
```

DS Selector가 이 환경에서 지원되지 않으므로 64비트 코드 예시는 존재하지 않는다. l1에 도달한다 하더라도 DS 셀렉터가 자신의 기본 값으로 저장될 것이기 때문에 윈도우의 64비

트 버전들에서 이 코드를 single-stepping하는 것은 l1에서 access violation 예외를 유발할 것이다. 윈도우의 32비트 버전들에서 DS 셀렉터는 디버깅이 아닌 예외가 발생 시에 자신의 값이 저장되게 하지 않을 것이다. 버전에 구체적인 행위에서의 차이는 SS 셀렉터가 사용되는 경우에 더 확장된다. 윈도우의 64비트 버전들에서 SS 셀렉터는 DS 경우 처럼 자신의 기본 값으로 저장될 것이다. 그러나 윈도우의 32비트 버전들에서 SS 셀렉터 값은 예외가 일어나더라도 저장되지 않을 것이다. 그러므로 우리가 다음과 같이 코드를 바꾸면:

```
 xor eax, eax
 push offset l2
 push d fs:[eax]
 mov fs:[eax], esp
 push fs
 pop ss
 xchg [eax], cl
 xchg [eax], cl
l1: int 3                ; force exception to occur
l2:                      ; looks like it would be reached
                         ; if an exception occurs
 ...
```

"int 3" 명령어가 11에 도달하고 브레이크포인트 예외가 발생할 때 12의 예외 핸들러는 예상과 같이 호출되지 않는다. 대신 프로세스는 간단하게 종료된다.

이 기법의 변형은 간단하게 assignment가 성공했는지를 검사함으로써 single-step 이벤트를 탐지한다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push 3
pop gs
mov ax, gs
cmp al, 3
jne being_debugged
```

FS와 GS 셀렉터들은 특별한 경우들이다. 특정한 값들에서 이들은 심지어 윈도우의 32비트 버전들에서도 single-step 이벤트에 영향을 받는다. 그러나 FS 셀렉터(그리고 기술적으로 GS 셀렉터)의 경우 만약 이것이 0부터 3까지의 값으로 설정되어 있다면 이것은 윈도우의 32비트 버전들에서 자신의 기본 값으로 저장된다. 대신 이것은 0으로 설정될 것이다(GS 셀렉터는 같은 방식으로 영향받지만 기본 값은 0이다). 윈도우의 64비트 버전들에서 이것들은 자신의 기본 값으로 저장된다.

이 코드는 또한 thread-switch 이벤트에 의해 발생하는 race condition에 취약하다. thread-switch 이벤트가 발생하면 이것은 예외처럼 행동하고 셀렉터 값들을 변경하게 만드는데 FS 셀렉터의 경우 0으로 설정된다는 것을 의미한다.

이 기법의 변형은 의도적으로 thread-switch 이벤트가 발생하게 기다림으로써 문제를 해결한다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  push 3
  pop gs
l1: mov ax, gs
  cmp al, 3
  je l1
```

그러나 이 코드는 원본 assignment가 성공적이었다면 검사를 하지 않기 때문에 첫번째 장소에서 탐지하려고 시도하는 문제에 취약하다. 물론 두 코드 snippet들은 thread-switch 이벤트가 발생할 때까지 기다리고 다음 것이 발생할 때까지 존재해야 하는 time의 윈도우 내에서 assignment를 수행함으로써 요구된 효과를 생산하기 위해 결합될 수 있다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  push 3
  pop gs
l1: mov ax, gs
  cmp al, 3
  je l1
  push 3
  pop gs
  mov ax, gs
  cmp al, 3
  jne being_debugged
```

F. User-interface

BlockInput
NtSetInformationThread

SuspendThread
SwitchDesktop

i. BlockInput

user32 BlockInput() 함수는 모든 마우스와 키보드 이벤트들을 막거나 막지 않을 수 있다 (ctrl-alt-delete를 제외하고). 효과는 프로세스가 존재하거나 함수가 opposite 파라미터와 함께 다시 호출되기 전까지 지속된다. 이것은 디버거들을 비활성화하기 위한 매우 효과적인 방법이다. 윈도우의 32비트 또는 64비트 버전들에서 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
push 1
call BlockInput
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
xor ecx, ecx
inc ecx
call BlockInput
```

이 호출은 호출하는 스레드가 DESKTOP_JOURNALPLAYBACK (0x0020) 권한을 가질 것을 요구한다(기본으로 설정된). 윈도우 비스타와 이후에서는, 권한 상승이 활성화된 상태에서 "HKLM₩Software₩Microsoft₩Windows₩CurrentVersion₩Policies₩System ₩EnableUIPI" 레지스트리 값이 존재하지 않거나(현재 기본 값과 설정이 이 경우에 사용된다) 0이 아닌 값을 갖는다면(그리고 이것은 관리자 권한이 바뀔 것을 요구한다), 이것은 또한 프로세스가 높은 integrity level에서 실행되어야 함을 요구한다(즉 프로세스는 표준 또는 더 낮은 사용자 계정에서 실행중이라면 elevation을 요구한다).

함수는 입력이 연달아 두 번 막히거나 막히지 않는 것을 허용하지 않는다. 그러므로 만약 같은 요청이 함수에 두 번 만들어 졌다면 반환 값은 달라야 한다. 그들 대부분이 입력에 관계없이 간단하게 성공을 반환하기 때문에 이 사실은 호출을 가로채는 많은 툴들의 존재를 탐지하는 데 사용될 수 있다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
push 1
call BlockInput
xchg ebx, eax
push 1
call BlockInput
```

```
xor ebx, eax
je being_debugged
```

또는 64비트 윈도우 환경을 조사하기 위한 이 64비트 코드를 사용할 수 있다.

```
xor ecx, ecx
inc ecx
call BlockInput
xchg ebx, eax
xor ecx, ecx
inc ecx
call BlockInput
xor ebx, eax
je being_debugged
```

ii. FLD

OllyDbg는 부동소수점 연산시 에러들을 비활성화하지 않기 때문에 부동소수점 명령어들의 OllyDbg's analyser에 문제가 존재한다. doubleextended precision을 정수로 변환할 때 이것은 두 값들이 부동소수점 에러들을 유발하게 하며 OllyDbg가 충돌하게 한다. __fuistq() 함수에서 문제가 되는 코드는 다음과 같다.

```
mov eax, [esp+04]
mov edx, [esp+08]
...
fld t [edx]
fistp q [eax]
wait
retn
```

두 값은 +/-9.2233720368547758075e18이다. 윈도우의 32비트 또는 64비트 버전들에서 문제는 이러한 32비트 코드들을 사용함으로써 입증된다.

```
fld t [offset l1]
...
l1: dq -1
dw 403dh
```

그리고

```
fld t [offset l1]
...
l1: dq -1
dw 0c03dh
```

연산을 모두 스킵하거나 대체 명령어(이것은 단지 현대적인 CPU에서만 존재하며 명령어가 지원되지 않으면 OllyDbg가 다른 크래시를 expose한다)를 사용하는것 같이 인간이 이 문제를 해결하는 여러 방식들이 존재하며 이것들 모두는 정도에 따라 다르다. 정확한 수정은 간단하게 부동소수점 예외 마스크를 이러한 에러들을 무시하게 바꾸는 것이다. 이것은 값 0x1333(현재 값 0x1332에서)을 FPU의 control word에 로드함으로써 달성할 수 있다.

iii. NtSetInformationThread

윈도우 2000은 함수 확장을 도입하였는데 이것은 언뜻 보기에는 단지 안티 디버깅 목적으로 존재하는 것처럼 보인다. 이것은 ThreadInformationClass 클래스의 ThreadHideFromDebugger (0x11) 멤버이다. 이것은 ntdll NtSetInformationThread() 함수를 호출함에 따라 각 스레드 기반으로 설정된다. 외부 프로세스에 의해 사용되도록 의도되었지만 어떠한 스레드도 자신에게 이것을 사용할 수 있다. 윈도우의 32비트 또는 64비트 버전들에서 호출은 다음의 32비트 코드를 사용함으로써 만들어질 수 있다.

```
push 0
push 0
push 11h                ; ThreadHideFromDebugger
push 2                  ; GetCurrentThread()
call NtSetInformationThread
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
xor r9d, r9d
xor r8d, r8d
push 11h                ; ThreadHideFromDebugger
pop rdx
push 2                  ; GetCurrentThread()
pop rcx
call NtSetInformationThread
```

함수가 호출되면 스레드는 실행을 계속할 것이지만 디버거는 더 이상 스레드와 관련된 어떠한 이벤트도 받지 않을 것이다. 만약 main 스레드가 감춰진 것이었다면 놓친 이벤트들 중

에 종료된 프로세스가 존재한다. 이 함수가 존재하는 이유는 외부 프로세스가 디버기에 대한 정보를 질의하기 위해 ntdll RtlQueryProcessDebugInformation() 함수를 사용할 때 예상치 못한 방해을 피하기 위한 것이다. ntdll RtlQueryProcessDebugInformation() 함수는 프로세스에 한 정보를 모으기 위해 스레드를 디버기에 삽입한다. 만약 삽입된 스레드가 디버거로부터 감춰져 있다면 스레드가 시작할 때 디버거는 제어를 얻을 것이고 디버기는 실행을 중지할 것이다.

iv. SuspendThread

kernel32 SuspendThread() 함수(또는 ntdll NtSuspendThread() 함수)는 사용자 모드 디버거들을 비활성화하기 위한 또다른 매우 효과적인 방식이다. 이것은 주어진 프로세스의 스레드들을 enumerating하거나 named window를 찾고 이것의 주인 스레드를 열고 그 스레드를 supend함으로써 달성된다. 윈도우의 32비트 또는 64비트 버전들에서 32비트 윈도우 환경을 검사하는 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다(부모 프로세스를 찾고 EDI 레지스터에서 프로세스 ID를 위치시킨 이전 코드를 기반으로).

```
  push edi
  push 4                    ; TH32CS_SNAPTHREAD
  call CreateToolhelp32Snapshot
  push 1ch                  ; sizeof(THREADENTRY32)
  push esp
  push eax
  xchg ebx, eax
  call Thread32First
l1: push esp
  push ebx
  call Thread32Next
  cmp [esp+0ch], edi        ; th32OwnerProcessID
  jne l1
  push d [esp+8]            ; th32ThreadID
  push 0
  push 2                    ; THREAD_SUSPEND_RESUME
  call OpenThread
  push eax
  call SuspendThread
```

또는 윈도우의 64비트 버전들에서 다음의 64비트 코드를 사용할 수 있다(부모 프로세스를 찾고 EDI 레지스터에서 프로세스 ID를 위치시킨 이전 코드를 기반으로).

```
  mov edx, edi
  push 4                ; TH32CS_SNAPTHREAD
  pop rcx
  call CreateToolhelp32Snapshot
  mov ebx, eax
  push 1ch              ; sizeof(THREADENTRY32)
  pop rbp
  enter 20h, 0
  mov edx, ebp
  xchg ecx, eax
  call Thread32First
l1: mov edx, ebp
  mov ecx, ebx
  call Thread32Next
  cmp [rbp+0ch], edi    ; th32OwnerProcessID
  jne l1
  mov r8, [rbp+8]       ; th32ThreadID
  cdq
  push 2                ; THREAD_SUSPEND_RESUME
  pop rcx
  call OpenThread
  xchg ecx, eax
  call SuspendThread
```

v. SwitchDesktop

윈도우 NT 기반 플랫폼들은 세션 당 다중 데스크탑들을 지원한다. 이전의 활성화 데스크탑의 윈도우를 숨기는 효과를 갖는 방식으로 여러 다른 활성화 데스크탑을 선택하는 것이 가능한데 다시 이전으로 돌리는 명백한 방법은 존재하지 않는다(ctrl-alt-delete가 이것을 수행하지 않는다). 더 나아가 그들의 소스가 더 이상 공유되지 않기 때문에 디버기의 데스크탑에서의 마우스와 키보드 이벤트들은 더 이상 디버거로 전달되지 않을 것이다. 이것은 명백하게 디버깅을 불가능하게 만든다. 윈도우의 32비트 또는 64비트 버전들에서 호출은 다음의 32비트 코드를 사용해서 만들 수 있다.

```
  xor eax, eax
  push eax              ; DESKTOP_CREATEWINDOW
                        ; + DESKTOP_WRITEOBJECTS
                        ; + DESKTOP_SWITCHDESKTOP
  push 182h
```

```
  push eax
  push eax
  push eax
  push offset l1
  call CreateDesktopA
  push eax
  call SwitchDesktop
  ...
l1: db "mydesktop", 0
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
  xor edx, edx
  push rdx                ; DESKTOP_CREATEWINDOW
                          ;+ DESKTOP_WRITEOBJECTS
                          ;+ DESKTOP_SWITCHDESKTOP
  push 182h
  sub esp, 20h
  xor r9d, r9d
  xor r8d, r8d
  mov rcx, offset l1
  call CreateDesktopA
  xchg ecx, eax
  call SwitchDesktop
  ...
l1: db "mydesktop", 0
```

G. Uncontrolled execution

CreateProcess
CreateThread
DebugActiveProcess
Enum...
NtSetLdtEntries
QueueUserAPC
RaiseException
RtlProcessFlsData

WriteProcessMemory
Intentional exception

i. CreateProcess

프로세스가 디버거의 제어로부터 벗어나는 가장 쉬운 방법은 자신의 다른 복사본을 실행하는 것이다. 일반적으로 프로세스는 무한하게 반복되는 것을 막기 위해 뮤텍스 같은 동기화 객체를 사용할 것이다. 첫번째 프로세스는 뮤텍스를 생성하고 프로세스의 복사본을 실행할 것이다. 두번째 프로세스는 첫번째 프로세스가 그렇더라도 디버거의 제어 하에 있지 않을 것이다. 두번째 프로세스는 또한 뮤텍스가 존재하기 때문에 자신이 복사본임을 알 것이다. 윈도우의 32비트 또는 64비트 버전에서 32비트 윈도우 환경을 검사하는 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
xor ebx, ebx
push offset l2
push ebx
push ebx
call CreateMutexA
call GetLastError
cmp eax, 0b7h                ; ERROR_ALREADY_EXISTS
je l1
mov ebp, offset l3
push ebp
call GetStartupInfoA
call GetCommandLineA
sub esp, 10h                 ; sizeof(PROCESS_INFORMATION)
push esp
push ebp
push ebx
push ebx
push ebx
push ebx
push ebx
push ebx
push ebx
push eax
push ebx
call CreateProcessA
pop eax
push -1                      ; INFINITE
```

```
  push eax
  call WaitForSingleObject
  call ExitProcess
l1: ...
l2: db "mymutex", 0
l3: db 44h dup (?)                  ; sizeof(STARTUPINFO)
```

또는 윈도우의 64비트 버전들에서 이 64비트 코드를 사용할 수 있다.

```
  mov r8, offset l2
  xor edx, edx
  xor ecx, ecx
  call CreateMutexA
  call GetLastError
  cmp eax, 0b7h          ; ERROR_ALREADY_EXISTS
  je l1
  mov rbp, offset l3
  push rbp
  pop rcx
  call GetStartupInfoA
  call GetCommandLineA
  mov rsi, offset l4
  push rsi
  push rbp
  xor ecx, ecx
  push rcx
  push rcx
  push rcx
  push rcx
  sub esp, 20h
  xor r9d, r9d
  xor r8d, r8d
  xchg edx, eax
  call CreateProcessA
  or rdx, -1                ; INFINITE
  mov ecx, [rsi]
  call WaitForSingleObject
  call ExitProcess
l1: ...
```

```
l2: db "mymutex", 0
l3: db 68h dup (?)        ; sizeof(STARTUPINFO)
l4: db 18h dup (?)        ; sizeof(PROCESS_INFORMATION)
```

kernel32 WaitForSingleObject() 함수 대신 kernel32 Sleep() 함수를 사용하는 것을 흔히 볼 수 있지만 이것은 race condition을 유발한다. 실행 시에 CPU-intensive한 활동이 존재하면 문제가 발생한다. 두번째 프로세스의 sufficiently complicated 보호(또는 내부적 딜레이) 때문이다; 그러나 또한 네트워크를 브라우징하거나 아카이브에서 파일들을 추출하는 것 같은 실행 중에 사용자가 수행할 행동들은 진행 중이다. 결과적으로 두번째 프로세스가 딜레이가 만료되기 전에 뮤텍스 검사에 도달하지 못할 것이다(이것이 첫번째 프로세스라는 생각에 이르게 하면서). 만약 일어난다면 프로세스는 자신의 다른 복사본을 실행할 것이다. 이 행동은 프로세스들 중 하나가 결국 뮤텍스 검사를 성공적으로 완료할 때까지 반복적으로 일어난다.

또한 첫 번째 프로세스는 반드시 두번째 프로세스가 종료되거나(프로세스 핸들을 기다림으로써) 최소한 자신의 성공적인 시작을 signal할(예를들면 뮤텍스를 사용하는 것 대신 이벤트 핸들을 기다림으로써) 때까지 기다려야 한다는 점에 주의하라. 반면 첫 번째 프로세스는 두번째 프로세스가 상태 검사를 수행하기 전에 종료할 것이고 두 번쨔 프로세스는 이것이 첫 번째 프로세스라고 생각할 것이고 사이클는 반복된다.

자가 실행 방식의 확장은 자가 디버깅(self-debugging)이다. 자가 디버깅은 이름이 말해주듯이 자신의 복사본을 실행하는 것이며 디버거로서 어태치한다. 이것은, 불가능한, 자신을 단일 프로세스 디버깅하는 것을 의미하지 않는다. 한번에 단지 한 디버거가 프로세스에 어태치될 수 있기 때문에 두번째 프로세스는 일반적인 의미에서 "undebuggable"하게 된다. 비록 자신이 확실히 할 수 있음에도 불구하고 첫번째 프로세스는 어떤 디버거 관련된 행동도 할 필요가 없다. 간단한 경우에서 첫번째 프로세스는 프로세스 종료 이벤트를 제외하고는 어떤 디버거 관련된 이벤트(DLL 로딩 같은)들도 무시할 수 있다. 더 발전된 경우에서는 두번째 프로세스는 하드 코딩된 브레이크포인트 같은 일반적인 안티디버거 기법들을 포함할 것이지만 오직 단일 프로세스만을 사용함으로써 디버깅 환경을 시뮬레이팅하는 것을 매우 어렵게 만듦으로써 현재 첫 번째 프로세스에 대해 특별한 의미를 가질 것이다. 윈도우의 32비트 또는 64비트 버전들에서 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
xor ebx, ebx
push offset l4
push ebx
push ebx
call CreateMutexA
call GetLastError
cmp eax, 0b7h          ; ERROR_ALREADY_EXISTS
je l3
```

```
  mov ebp, offset l5
  push ebp
  call GetStartupInfoA
  call GetCommandLineA
  mov esi, offset l6
  push esi
  push ebp
  push ebx
  push ebx
  push 1                ; DEBUG_PROCESS
  push ebx
  push ebx
  push ebx
  push eax
  push ebx
  call CreateProcessA
  mov ebx, offset l7
  jmp l2
l1: push 10002h         ; DBG_CONTINUE
  push d [esi+0ch]      ; dwThreadId
  push d [esi+8]        ; dwProcessId
  call ContinueDebugEvent
l2: push -1             ; INFINITE
  push ebx
  call WaitForDebugEvent
  cmp b [ebx], 5        ; EXIT_PROCESS_DEBUG_EVENT
  jne l1
  call ExitProcess
l3:                     ; execution resumes here in second process
 ...
l4: db "mymutex", 0
l5: db 44h dup (?)      ; sizeof(STARTUPINFO)
l6: db 10h dup (?)      ; sizeof(PROCESS_INFORMATION)
l7: db 60h dup (?)      ; sizeof(DEBUG_EVENT)
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
  mov r8, offset l4
  xor edx, edx
```

```
  xor ecx, ecx
  call CreateMutexA
  call GetLastError
  cmp eax, 0b7h          ; ERROR_ALREADY_EXISTS
  je l3
  mov rbp, offset l5
  push rbp
  pop rcx
  call GetStartupInfoA
  call GetCommandLineA
  mov rsi, offset l6
  push rsi
  push rbp
  xor ecx, ecx
  push rcx
  push rcx
  push 1                 ; DEBUG_PROCESS
  push rcx
  sub esp, 20h
  xor r9d, r9d
  xor r8d, r8d
  xchg edx, eax
  call CreateProcessA
  mov rbx, offset l7
  jmp l2
l1: mov r8d, 10002h      ; DBG_CONTINUE
  mov edx, [rsi+14h]     ; dwThreadId
  mov ecx, [rsi+10h]     ; dwProcessId
  call ContinueDebugEvent
l2: or rdx, -1           ; INFINITE
  push rbx
  pop rcx
  call WaitForDebugEvent
  cmp b [rbx], 5         ; EXIT_PROCESS_DEBUG_EVENT
  jne l1
  call ExitProcess
l3:                      ; execution resumes here in second process
  ...
l4: db "mymutex", 0
```

```
l5: db 68h dup (?)        ; sizeof(STARTUPINFO)
l6: db 18h dup (?)        ; sizeof(PROCESS_INFORMATION)
l7: db 0ach dup (?)       ; sizeof(DEBUG_EVENT)
```

ii. CreateThread

스레드들은 브레이크포인트가 적절한 위치에 위치하지 않는 경우에 디버기가 (실행이 자유롭게 재개되는 곳에서) 메모리 위치에 대한 제어를 옮기는 쉬운 방법이다. 이것들은 또한 다른 스레드의 실행과 상호 작용하는데 사용될 수 있다(main 스레드를 포함해서). 한 예로 주기적으로 소프트웨어 브레이크포인트나 디버거가 코드 흐름에 유발하는 다른 메모리 변경을 검사하는 것이 있다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  xor eax, eax
  push eax
  push esp
  push eax
  push eax
  push offset l2
  push eax
  push eax
  call CreateThread
l1:                       ; here could be obfuscated code
                          ; with lots of dummy function calls
                          ; that would invite step-over
                          ; and thus breakpoint insertion
 ...
l2: xor eax, eax
  cdq
  mov ecx, offset l2 - offset l1
  mov esi, offset l1
l3: lodsb
  add edx, eax            ; simple sum to detect breakpoints
  loop l3
  cmp edx, <checksum>
  jne being_debugged
  mov ch, 1               ; small delay then restart
  push ecx
  call Sleep
```

```
  jmp l2
```

또는 64비트 윈도우 환경을 조사하기 위해 이 64비트 코드를 사용할 수 있다.

```
  xor edx, edx
  push rdx
  push rsp
  push rdx
  sub esp, 20h
  xor r9d, r9d
  mov r8, offset l2
  xor ecx, ecx
  call CreateThread
l1:                     ; here could be obfuscated code
                        ; with lots of dummy function calls
                        ; that would invite step-over
                        ; and thus breakpoint insertion
...
l2: xor eax, eax
  cdq
  mov ecx, offset l2 - offset l1
  mov rsi, offset l1
l3: lodsb
  add edx, eax          ; simple sum to detect breakpoints
  loop l3
  cmp edx, <checksum>
  jne being_debugged
  mov ch, 1             ; small delay then restart
  call Sleep
  jmp l2
```

iii. DebugActiveProcess

kernel32 DebugActiveProcess() 함수는 (또는 ntdll DbgUiDebugActiveProcess() 함수 또는 ntdll NtDebugActiveProcess() 함수) 이미 실행 중인 프로세스에 디버거로서 어태치할 때 사용될 수 있다. 단지 한 디버거가 한번에 한 프로세스에 어태치될 수 있기 때문에 파일에의 어태치에 대한 실패는 다른 디버거의 존재를 암시한다(비록 security descriptor restriction 같은 실패에 대한 또 다른 이유가 있을 수 있지만). 윈도우의 32비트 또는 64비트 버전들에서 32비트 윈도우 환경을 조사하는 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
  mov ebp, offset l4
  push ebp
  call GetStartupInfoA
  xor ebx, ebx
  mov esi, offset l5
  push esi
  push ebp
  push ebx
  push ebx
  push ebx
  push ebx
  push ebx
  push ebx
  push offset l3
  push ebx
  call CreateProcessA
  push d [esi+8]          ; dwProcessId
  call DebugActiveProcess
  test eax, eax
  je being_debugged
  mov ebx, offset l6
  jmp l2
l1: push 10002h           ; DBG_CONTINUE
  push d [esi+0ch]        ; dwThreadId
  push d [esi+8]          ; dwProcessId
  call ContinueDebugEvent
l2: push -1               ; INFINITE
  push ebx
  call WaitForDebugEvent
  cmp b [ebx], 5          ; EXIT_PROCESS_DEBUG_EVENT
  jne l1
  call ExitProcess
l3: db "myfile", 0
l4: db 44h dup (?)        ; sizeof(STARTUPINFO)
l5: db 10h dup (?)        ; sizeof(PROCESS_INFORMATION)
l6: db 60h dup (?)        ; sizeof(DEBUG_EVENT)
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
  mov rbp, offset l4
  push rbp
  pop rcx
  call GetStartupInfoA
  mov rsi, offset l5
  push rsi
  push rbp
  xor ecx, ecx
  push rcx
  push rcx
  push rcx
  push rcx
  sub esp, 20h
  xor r9d, r9d
  xor r8d, r8d
  mov rdx, offset l3
  call CreateProcessA
  mov ecx, [rsi+10h]            ; dwProcessId
  call DebugActiveProcess
  test eax, eax
  je being_debugged
  mov rbx, offset l6
  jmp l2
l1: mov r8d, 10002h             ; DBG_CONTINUE
  mov edx, [rsi+14h]            ; dwThreadId
  mov ecx, [rsi+10h]            ; dwProcessId
  call ContinueDebugEvent
l2: or rdx, -1                  ; INFINITE
  push rbx
  pop rcx
  call WaitForDebugEvent
  cmp b [rbx], 5                ; EXIT_PROCESS_DEBUG_EVENT
  jne l1
  call ExitProcess
l3: db "myfile", 0
l4: db 68h dup (?)              ; sizeof(STARTUPINFO)
l5: db 18h dup (?)              ; sizeof(PROCESS_INFORMATION)
l6: db 0ach dup (?)             ; sizeof(DEBUG_EVENT)
```

iv. Enum...

많은 enumeration 함수들이 있으며 이 중 몇몇은 kernel32.dll이 아닌 DLL에 있다. 이것들은 모두 (브레이크포인트가 적절한 위치에 위치하지 않는 이상) 실행이 자유롭게 재개되는 곳에서 디버기가 메모리 위치에 대한 제어를 옮기기 위해 사용될 수 있다. 윈도우의 32비트 또는 64비트 버전들에서 32비트 윈도우 환경을 조사하는 호출은 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  push 0
  push 0
  push offset l1
  call EnumDateFormatsA
  jmp being_debugged
l1:                         ; execution resumes here during enumeration
  ...
```

또는 64비트 윈도우 환경을 조사하기 위한 이 64비트 코드를 사용할 수 있다.

```
  xor r8d, r8d
  xor edx, edx
  mov rcx, offset l1
  call EnumDateFormatsA
  jmp being_debugged
l1:                         ; execution resumes here during enumeration
  ...
```

v. GenerateConsoleCtrlEvent

콘솔 윈도우가 focus를 가지는 동안 사용자가 Ctrl-C 또는 Ctrl-Break 키 조합을 누르면 윈도우는 이벤트가 다뤄져야 하는지 무시되어야 하는지를 검사한다. 만약 이벤트가 다뤄져야 한다면 윈도우는 등록된 console control handler를 호출하거나 kernel32 CtrlRoutine() 함수를 호출한다. kernel32 CtrlRoutine() 함수는 디버거의 존재를 검사하며(이것은 PEB의 BeingDebugged 플래그를 읽음으로써 결정한다), 존재한다면 DBG_CONTROL_C (0x40010005) 예외 또는 DBG_CONTROL_BREAK (0x40010008) 예외를 만들어 낸다. 이 예외는 예외 핸들러나 이벤트 핸들러에 의해 가로채질 수 있지만, 예외는 대신 디버거에 의해 다루어질 것이다. 결과적으로 놓친 예외는 디버거의 존재를 추론하는데 사용될 수 있다. 애플리케이션은 SEH를 등록하거나 kernel32 SetConsoleCtrlHandler() 함수를 호출함으로써 이벤트 핸들러를 등록할 수 있다. 예외는 kernel32 GenerateConsoleCtrlEvent() 함수를 호출함으로

써 강제적으로 발생된다. 윈도우의 32비트 또는 64비트 버전들에서 32비트 윈도우 환경을 조사하는 호출은 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  xor eax, eax
  push offset l1
  push d fs:[eax]
  mov fs:[eax], esp        ; Process Environment Block
  mov ecx, fs:[eax+30h]
  inc b [ecx+2]
  push eax
  push eax                 ; CTRL_C_EVENT
  call GenerateConsoleCtrlEvent
  jmp $
l1: ;execution resumes here if exception occurs
  ...
```

또는 64비트 윈도우 버전에서 이 64비트 코드를 사용할 수 있다.

```
  mov rdx, offset l1
  xor ecx, ecx
  inc ecx
  call AddVectoredExceptionHandler
  push 60h
  pop rsi
  gs:lodsq                 ; Process Environment Block
  inc b [eax+2]
  push 0
  push 0                   ; CTRL_C_EVENT
  call GenerateConsoleCtrlEvent
  jmp $
l1:                        ; execution resumes here if exception occurs
  ...
```

vi. NtSetInformationProcess

Local Descriptor Table (LDT) 레지스터의 값이 물리적 윈도우 환경(가상의 반대)에서의 사용자 모드에서 0이기 때문에 일반적으로 디버거들에 의해서 적절하게(또는 전혀) 지원되지 않는다. 결과적으로 이것은 간단한 안티 디버깅 기법으로서 사용될 수 있다. 구체적으로 말해서 몇몇 코드를 매핑하는 새로운 LDT 엔트리가 생성될 수 있다. 이것

은 ntdll NtSetInformationProcess() 함수를 호출하고 ProcessInformationClass 클래스의 ProcessLdtInformation (0x0a) 멤버를 전달함으로써 각 프로세스 기반으로 수행될 수있다. 그러면 제어 명령어(call, jump, ret, etc.)의 새로운 LDT 엔트리로의 전달은 디버거가 새로운 메모리 주소를 혼동하게 만들거나 심지어 거부하게 만들 것이다. 32비트 윈도우 버전에서 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다(이 호출은 윈도우의 64비트 버전들에서 지원되지 않는다).

```
        ; base must be <= PE->ImageBase
        ; but no need for 64kb align
 base equ 12345678h
        ; sel must have bit 2 set
        ; CPU will set bits 0 and 1
        ; even if we don't do it
 sel equ 777h

        ; 4k granular, 32-bit, present
        ; DPL3, exec-only code
        ; limit must not touch kernel mem
        ; calculate carefully to use functions
 push (base and 0ff000000h) + 0c1f800h + ((base shr 10h) and 0ffh)
 push (base shl 10h) + 0ffffh
 push 8
 push sel and -8 ;bits 0-2 must be clear here
 mov eax, esp
 push 10h
 push eax
 push 0ah ;ProcessLdtInformation
 push -1 ;GetCurrentProcess()
 call NtSetInformationProcess
        ; jmp far sel:l1
 db 0eah
 dd offset l1 – base
 dw sel
l1:     ; execution continues here but using LDT selector
 ...
```

vii. NtSetLdtEntries

ntdll NtSetLdtEntries() 함수 또한 LDT 값들을 직접적으로 설정하게 하지만 단지 현재 프로

세스에서만 통한다. 이것은 위에 나온 ProcessLdtInformation 기법과는 약간 다른 파라미터 포맷과 함께하는 완전히 독립된 함수이다. 윈도우의 32비트 버전에서 호출은 다음의 32비트 코드를 사용하여 만들어질 수 있다(이 호출은 윈도우의 64비트 버전들에서 지원되지 않는다).

```
        ; base must be <= PE->ImageBase
        ; but no need for 64kb align
  base equ 12345678h
        ; sel must have bit 2 set
        ; CPU will set bits 0 and 1
        ; even if we don't do it
  sel equ 777h

  xor eax, eax
  push eax
  push eax
  push eax
        ; 4k granular, 32-bit, present
        ; DPL3, exec-only code
        ; limit must not touch kernel mem
        ; calculate carefully to use functions
  push (base and 0ff000000h) + 0c1f800h + ((base shr 10h) and 0ffh)
  push (base shl 10h) + 0ffffh
  push sel
  call NtSetLdtEntries
        ; jmp far sel:l1
  db 0eah
  dd offset l1 – base
  dw sel
l1:     ; execution continues here but using LDT selector
 ...
```

viii. QueueUserAPC

kernel32 QueueUserAPC() 함수(또는 ntdll NtQueueApcThread() 함수)는 Asynchronous Procedure Calls (APCs)를 등록하는데 사용될 수 있다. APC들은 kernel32 Sleep() 함수를 호출하는것 같이 연관된 스레드가 "alertable" 상태에 들어올 때 호출된다. 이것들은 또한 만약 스레드가 실행을 시작하기 전에 등록되었다면 스레드 엔트리포인트 이전에 호출된다. 이것들은 브레이크포인트가 적절한 위치에 위치하지 않

는 한 (디버기가 실행이 자유롭게 재개될 수 있는) 메모리 위치에 대한 제어를 전달하는 흥미로운 방식이다. 이것들은 만약 대상 스레드가 alertable 함수들 중 하나를 호출하는것이 알려졌다면 kernel32 CreateRemoteThread() 함수를 호출하는 것 대신 사용될 수 있다. 윈도우의 32비트 또는 64비트에서 호출은 다음의 32비트 코드를 사용해서 만들어질 수 있다.

```
  xor eax, eax
  push eax
  push esp
  push eax
  push eax
  push eax                ; thread entrypoint is irrelevant
  push eax
  push eax
  call CreateThread
  push eax
  push eax
  push offset l1
  call QueueUserAPC
  jmp $
l1:                       ; execution resumes here when thread starts
 ...
```

또는 윈도우의 64비트 버전에서 이 64비트 코드를 사용할 수 있다.

```
  xor edx, edx
  push rdx
  push rsp
  push rdx
  sub esp, 20h
  xor r9d, r9d
  xor r8d, r8d            ; thread entrypoint is irrelevant
  xor ecx, ecx
  call CreateThread
  xchg edx, eax
  mov rcx, offset l1
  call QueueUserAPC
  jmp $
l1:                       ; execution resumes here when thread starts
 ...
```

ix. RaiseException

kernel32 RaiseException() 함수(또는 ntdll RtlRaiseException() 함수와 ntdll NtRaiseException() 함수)는 특정한 예외들이 발생하게 강제하는데 사용될 수 있는데 이것은 디버거가 일반적으로 처리하는 것이다. 다른 디버거들은 다른 예외들의 집합들을 처리한다. 특정한 디버거의 존재에서 발생했을 때 결과적으로 적절한 집합에서의 어느 예외들이라도 디버기 대신 디버거에 전달될 수 있다. 놓친 예외는 특정한 디버거의 존재를 추론하는데 사용될 수 있다. 몇몇 디버거들은 특정한 또는 모든 예외들이 모든 상황에서 디버기로 전달되도록 한다. 그러나 이것은 디버거가 디버기에 대한 제어를 잃게되는 것을 초래한다. 거의 일반적으로 consume되는 예외는 DBG_RIPEVENT exception (0x40010007)이다. 이것을 검사하는 것은 32비트 또는 64비트 윈도우 버전에서 32비트 윈도우 환경을 조사하는 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  xor eax, eax
  push offset l1
  push d fs:[eax]
  mov fs:[eax], esp
  push eax
  push eax
  push eax
  push 40010007h        ; DBG_RIPEVENT
  call RaiseException
  jmp being_debugged
l1:                     ; execution resumes here due to execption
 ...
```

또는 64비트 윈도우 환경을 조사하기 위한 이 64비트 코드를 사용할 수 있다.

```
  mov rdx, offset l1
  xor ecx, ecx
  inc ecx
  call AddVectoredExceptionHandler
  xor r9d, r9d
  xor r8d, r8d
  cdq
  mov ecx, 40010007h   ; DBG_RIPEVENT
  call RaiseException
  jmp being_debugged
```

```
l1:                     ; execution resumes here due to exception
  ...
```

한 디버거는 소프트웨어 브레이크포인트를 파라미터들(프로세스 메모리의 어디에서든 임의의 바이트로 체하는 것과 같은 예외와 함께 제공되는)에 기반한 특정한 행동들을 수행하게 지시하는 것으로 유명하다.

어떤 디버거는 어떤 관련된 로깅 정보를 출력하려고 시도할 때 off-by-one 버그 때문에 함수가 직접적으로 호출될 때 충돌하는 것으로 유명하다.

x. RtlProcessFlsData

ntdll RtlProcessFlsData() 함수는 윈도우 비스타에서 도입된 문서화되지 않은 함수이다. 이것은 kernel32 FlsSetValue()와 kernel32 DeleteFiber() 함수에 의해 호출된다. 적절한 파라미터와 메모리 값들과 함께 호출되면 함수는 메모리의 userspecified 포인터에서 코드를 실행한다. 만약 디버거가 이 사실을 인식하지 못한다면 실행 제어를 잃게될 것이다. 윈도우의 32비트 또는 64비트 버전에서 32비트 윈도우 환경을 조사하는 호출은 다음의 32비트 코드를 사용하여 만들어질 수 있다.

```
  push 30h
  pop eax
  mov ecx, fs:[eax]
  mov ah, 2
  inc d [ecx+eax-4]       ; must be at least 1
  mov esi, offset l2-4
  mov [ecx+eax-24h], esi
  lodsd
  push esi
  call RtlProcessFlsData
  jmp being_debugged
l1:                       ; execution resumes here during processing
  ...
l2: dd 0, offset l1, 0, 1
```

또는 윈도우의 64비트 버전에서 이 64비트 코드를 사용할 수 있다.

```
  push 60h
  pop rax
  mov rcx, gs:[rax]
```

```
  mov ah, 3
  inc d [rcx+rax-10h]     ; must be at least 1
  mov rsi, offset l2-8
  mov [rcx+rax-40h], rsi
  lodsq
  push rsi
  pop rcx
  call RtlProcessFlsData
  jmp being_debugged
l1:                       ; execution resumes here during processing
  ...
l2: dq 0, offset l1, 0, 1
```

xi. WriteProcessMemory

kernel32 WriteProcessMemory() 함수(또는 ntdll NtWriteVirtualMemory() 함수)는 데이터의 source가 디스크의 파일이 아니라 프로세스 메모리 공간이라는 것을 제외하고는 위에서 설명한 kernel32 ReadFile() 함수와 같은 방식으로 사용될 수 있다. 윈도우의 32비트 또는 64비트 버전들에서 호출은 이 32비트 코드를 사용해서 만들어질 수 있다.

```
  push 0          ; replace byte at l1 with byte at l2
                  ; step-over will also result
                  ; in uncontrolled execution
  push 1
  push offset l2
  push offset l1
  push -1         ; GetCurrentProcess()
  call WriteProcessMemory
l1: int 3
l2: nop
```

또는 윈도우의 64비트 버전에서 이 64비트 코드를 사용할 수 있다.

```
  push 0
  sub esp, 20h            ; replace byte at l1 with byte at l2
                          ; step-over will also result
                          ; in uncontrolled execution
  push 1
  pop r9
```

```
  mov r8, offset l2
  mov rdx, offset l1
  or rcx, -1                ; GetCurrentProcess()
  call WriteProcessMemory
l1: int 3
l2: nop
```

이 기법을 쓰지 못하게 하는 한 방법은 함수 호출들을 step over할 때 소프트웨어 브레이크포인트 대신 하드웨어 브레이크포인트를 사용하는 것이다.

xii. Intentional exceptions

디버거에 의해 다뤄지지 않는 예외는 (브레이크포인트가 적절한 위치에 위치하지 않았을 때) 디버기가 실행이 자유롭게 재개될 수 있는 곳에 메모리 위치에 한 제어를 옮기는 쉬운 방법이다.

```
xor eax, eax
push offset handler     ; step 1
push d fs:[eax]         ; step 2
mov fs:[eax], esp       ; step 3  [code to force exception]
```

이것은 SEH 방식이며 단지 32비트 윈도우 환경들에서만 존재한다. 64비트 윈도우 환경에서 VEH 핸들링이 예외 핸들러를 동적으로 등록하는 대신 사용된다. 예외 핸들러를 등록하는 어느 지점에서 예외가 생성된다. 금지된 또는 권한을 가진 명령어들을 사용하거나 금지된 메모리 접근 같은 예외를 발생시키는 많은 방식들이 존재한다.

명령어들 중 몇몇을 난독화하는 방법들이 존재한다. 첫 번째는 FS 셀렉터에 한 참조를 제거하는 것이다. fs:[18h]의 값은 FS region에 한 포인터이지만 이 방식으로 GS가 아닌 다른 셀렉터를 사용해서 접근할 수 있다.

```
mov eax, fs:[18h]
push d [eax]            ; step 2
mov [eax], esp          ; step 3
```

그 후 push를 간적접으로 만드는 방법이 있다.

```
mov ebp, [eax]
enter 0, 0              ; step 2
mov [eax], ebp          ; step 3
```

kernel32 GetThreadSelectorEntry() 함수를 사용하는 것 같이 FS 셀렉터의 base 주소를 얻는 다른 방법이 존재하지만 일반적인 단계들의 순서는 같게 된다. 그러나 심지어 메모리 쓰기를 간접적으로 만드는 공개된 방식(http://vx.netlux.org/lib/vrg03.html)이 존재한다. 이것은 다음과 같다.

```
  push 8
  pop eax
  call l2
  push -1                 ; push fs:[0]      ; step 2
  mov ecx, fs
  mov ss, ecx
  xchg esp, eax
  call l1                 ; change stack limit
l1: push eax              ;mov fs:[0], esp           ; step 3
  mov esp, ebp            ; point esp somewhere legal
l2: pop esp
  call esp                ; push offset handler    ; step 1
                          ; handler code here
```

이 코드는 시작 전에 쓰기 가능한 메모리가 없다고 가정한다. 만약 있다면 코드는 두 바이트로 줄어들 수 있다. 이것은 또한 메모리에서 스택이 코드 보다 낮다고 가정한다. 만약 이 경우가 아니라면 예외 발생 시 윈도우는 프로세스를 종료할 것이다. 예외 발생 시 셀렉터들의 행동에 대한 가정 때문에 흥미롭게도 이 버전은 윈도우의 64비트 버전들에서 32비트 환경에서만 통한다. 구체적으로 위치 l2가 두번 도달되고 second time에 SS 셀렉터 값과 FS 셀렉터 값은 같게 된다. 이 가정은 l2의 "pop esp" 명령어가 그 때 예외를 일으키며(ESP 레지스터 값이 FS 세그먼트의 한계를 뛰어넘기 때문에) SS 셀렉터 값이 그것의 정확한 값으로 복구된다는 것이다. 이것은 윈도우의 32비트 버전들에서는 발생하지 않는다. 윈도우의 32비트 버전들과의 호환을 달성하기 위해 그리고 동시에 모든 가정들을 정확하게 하기 위해 코드는 다음과 같을 필요가 있다.

```
                  ; writable page before this point
  push 0ch
  pop eax
  call l2
  push -1         ; push fs:[0]              ; step 2
  push fs
  pop ss
  xchg esp, eax
```

```
  push esp       ; change stack base
  call l1        ; change stack limit
l1: push eax     ; mov fs:[0], esp          ; step 3
  mov ecx, ds
  mov ss, ecx    ; restore ss
  xchg esp, eax ; point esp somewhere legal
  push esp       ; point to exception instruction
l2: pop esp      ; get address so call will fault
  call esp       ; push offset handler    ; step 1
                 ; handler code here
```

a. Nanomites

Nanomites는 일반적으로 분기 명령어들을 "int 3" 명령어로 대체하고 그 후에 "int 3"이 Nanomite인지 아니면 디버그 브레이크인지 결정하기 위해 언패킹 코드에 있는 테이블들을 사용함으로써 동작한다. 만약 "int 3"이 Nanomite라면 테이블들은 또한 분기가 받아들여져야 하는지를, 분기가 받아들여지는 경우 목적지의 주소가 그리고 분기가 받아들여지지 않은 경우 명령어가 얼마나 큰지를 결정하는데 사용될 수 있다.

nanomites에 의해 보호된 프로세스는 일반적으로 self-debugging을 요구한다. 이것은 nanomite에 도달되었을 때 디버거가 디버기에 의해 생성된 예외들을 처리하게 한다.

그러나 안티 디버깅 기법을 제외하면 self-debugging을 위한 요구사항은 존재하지 않는다. 단일 프로세스는 자신의 예외 핸들러를 등록하고 자신의 예외들을 처리할 수 있다. 이 행동의 가장 극단적인 버전을 보여주는 적어도 하나의 바이러스가 존재한다. 이 바이러스는 랜덤하게 자신의 명령어들 중 모두를 주문하며 다음 명령어의 위치에 대한 정보를 전달하는 nanomete들을 사용하면서 이것들 모두를 링크한다.

H. Conclusion

지금까지 살펴 보았듯이 매우 많은 안티 디버깅 기법들이 존재한다. 몇몇은 알려져 있는 것들이고 그렇지 않은 새로운 것들도 존재하지만 이것들 모두가 제대로 된 방어 기법이 존재하는 것은 아니다. 우리가 필요한 것은 이것들 모두를 아는 궁극적인 디버거이다.