기 술 문 서

Packing/Anti-debugging의 개념 및 Reversing의 개념 이해

작성자 : 김원기 (akanad@gmail.com)
이호성 (gdlhs2000@gmail.com)
임재창 (rjaechang@gmail.com)
임원섭 (Luus2001@gmail.com)
한승범 (seungbemi@naver.com)

03 Nov. 2009

◉ 목 차 ◉

1. Packing/Anti-Debugging 의 개념

Packing은 프로그램을 압축/암호화하고 그 내용을 풀 수 있는 해독기를 프로그램에 붙여 실행시 그것을 이용하여 복원한 내용을 메모리에 올리고 원래 프로그램의 EP를 실행하는 방식으로 진행하게 된다. Packing은 2가지 이점이 있다. 첫 번째 프로그램의 용량이 일반적으로 줄어들고, 두 번째 디버깅 툴에 의한 디버깅이 어려워지게 되었다.

Anti-Debugging이란 프로그램을 디버깅 하지 못하게 하기 위해서 하는 일련의 작업을 말한다. Binary 파일을 보호하는 하나의 방법인 쉐어웨어, 인증과 관련된 프로그램, 기타 중요한 프로그램들의 내부 알고리즘과 중요 데이터들을 분석하지 못하도록 파일에 Packing을 하거나 Anti-Debugging을 적용함으로써 디버깅을 방지하고 분석을 하지 못하도록 하는 데 큰 목적이 있다. Anti-Debugging을 적용한 프로그램이 실행 중에 있을 때 디버깅을 당한다면 디버깅을 하지 못하도록 해당 디버거 프로그램을 종료시키거나 에러를 발생 시키는 방법 등 다양한 방법을 사용하여 분석을 방해한다.

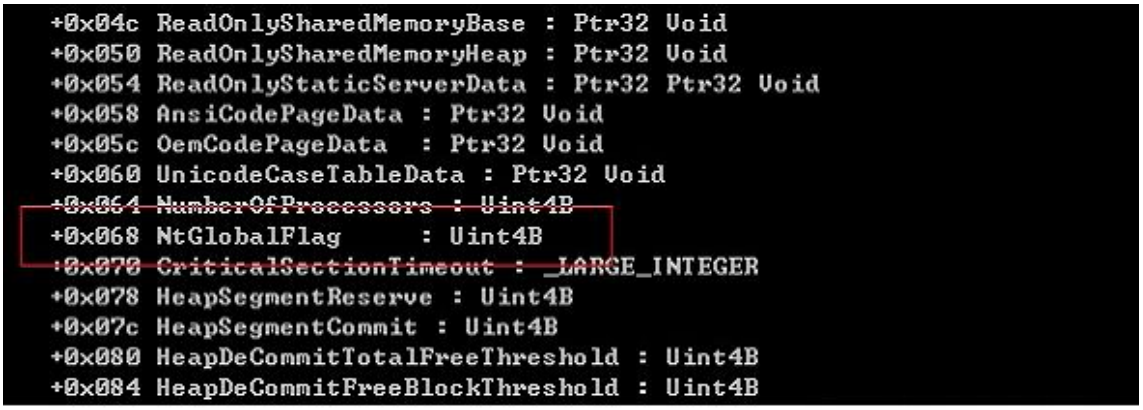
*** Anti-Debugging 기법**

Anti-Debugging 기법	설명
IsDebuggerPresent	PEB 구조체의 디버깅 상태값을 확인한다. 디버깅을 당할 경우 1, 그렇지 않을 경우 0을 리턴하게 된다.
IsDebugged	PEB 구조체의 BeingDebugged 맴버의 값을 확인한다. 디버깅을 당할 경우 1, 그렇지 않을 경우 0을 리턴한다.
NtGlobalFlags	PEB 구조체에 0x68에 위치해 있는 NtGlobalFlag 값을 확인한다. 0 이면 정상, 디버깅을 당했을 경우 0이 아닌 값이 리턴된다.
CheckRemoteDebuggerPresent	Win Xp 이상부터 사용할 수 있으며 ZwQueryInformationProcess()를 사용하여 디버깅 정보를 얻게 된다. 0이면 정상, 디버깅을 당했을 경우 다른 값을 리턴한다.
FindWindow	FindWindow() 함수를 사용하여 트정 윈도우 이름이나 클래스 이름을 찾아 특정 프로그램이 실행중인지를 확인한다.
Heap	flags PEB 구조체의 0x18에 위치해 있는 Heap flags를 검사한다. 정상일 경우 2의 값을 리턴하며 디버깅 당할 경우 그외의 값을 리턴한다.
Debugger	Interrupts 인터럽트 수행 중에 같은 인터럽트 코드를 만났을 때 예외처리 하지 않는 경우 디버깅 중으로 판단한다.

2. Packing/Anti-Debugging 소스분석

PEB.NtGlobalFlag

PEB은 BeingDebugged 플래그 외에도, PEB는 NtGlobalFlag라는 필드를 갖고 있는데 NtGlobalFlag는 PEB으로부터 0x68 위치에 존재한다. LiveKD를 이용하여 PEB의 구조체 값을 확인 해보았다.



이 플러그인의 값은 디버깅 중이 아니라면 0x0 값이 담기지만 디버깅 중이라면 0x70 값이 담겨진다. 그와 같은 것을 이용하여 디버깅 탐지를 한다.

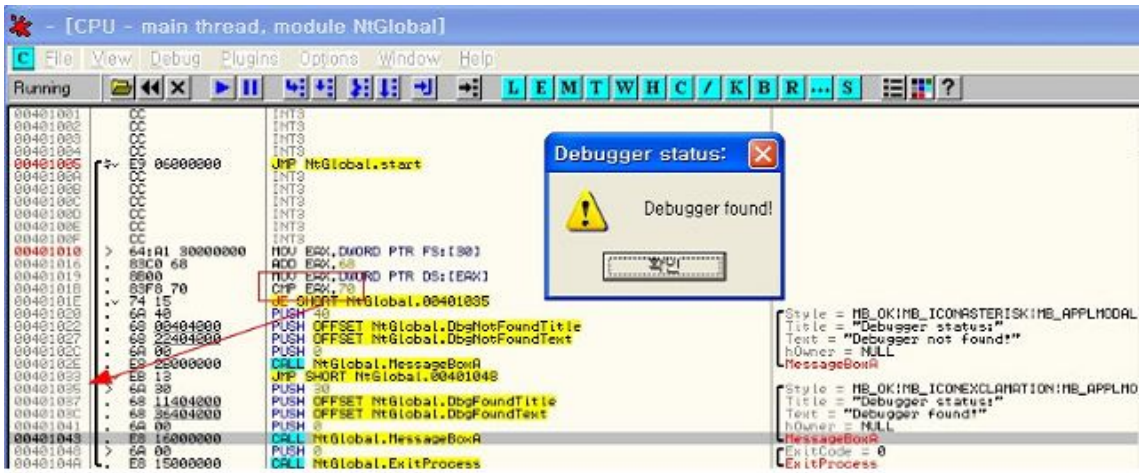
다음은 NtGlobalFlag Code 부분이다.

```
 .386
.model flat, stdcall
option casemap :none   ; case sensitive

include c:₩masm32₩include₩windows.inc
include c:₩masm32₩include₩user32.inc
include c:₩masm32₩include₩kernel32.inc
includelib c:₩masm32₩lib₩user32.lib
includelib c:₩masm32₩lib₩kernel32.lib

.data
DbgNotFoundTitle db "Debugger status:",0h
DbgFoundTitle db "Debugger status:",0h
DbgNotFoundText db "Debugger not found!",0h
DbgFoundText db "Debugger found!",0h
.code
start:
```

```
ASSUME FS:NOTHING
MOV EAX,DWORD PTR FS:[30h]
ADD EAX,68h
MOV EAX,DWORD PTR DS:[EAX]
CMP EAX,70h
JE @DebuggerDetected

PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox

JMP @exit
@DebuggerDetected:

PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox

@exit:

PUSH 0
CALL ExitProcess

end start
```

여기서 ASSUME FS:NOTHING 이란 구문은 세그먼트 레지스트의 주소 값을 재 할당 하는 것이 아니라 이 디렉티브를 만나면 실행 시에 어셈블러가 주소를 계산하는 방법을 변경한다. 즉 FS 세그먼트에 NOTHING이라는 속성을 붙이는 의미이다. FS:[0] 은 Exception handler를 Default를 가지고 있다. MASM 컴파일러는 기본적으로 이 레지스터를 사용할 때 ERORR를 뱉기 때문에 위와 같이 ASSUM FS:NOTHING 을 써주면 그와 같은 ERORR Check를 Remove 해준다고 설명이 되어 있다.

다음으로FS:[30] 는 PEB의 위치에서 ADD EAX,68h 에는 NtGlobalFlag 가 존재한다. 해당 코드를 올리로 열어 보았다.

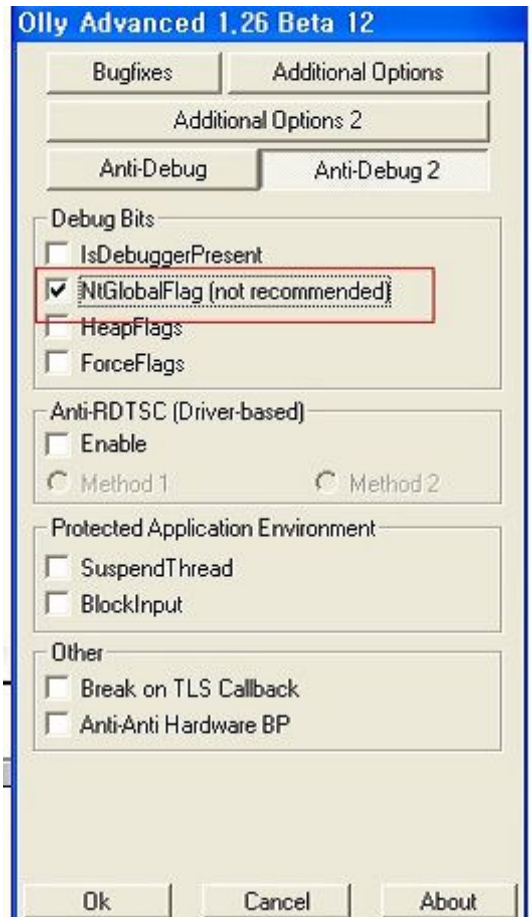


Olly 로 확인한 NtGlobalFlag 루틴

NtGlobalFlag 우회

해당 탐지를 우회는 간단하다. 직접 코드 패치를 하여 바꾸거나 혹은 플래그 값을 수정하거나 이다. 하지만 이것도 역시 올리에서 플러그인 형태로 지원을 하고 있다. 해당 플러그 인을 실행하고 NtGlobalFlag를 체크 한 뒤 확인 해주면 자동적으로 리턴 시 플래그 값을 수정하여 디버거 탐지를 우회 하게 된다. 그 외 코드 패치 등 다양한 방법이 존재한다. 툴로써 좀 더 편하게 하는 것이 좋다.

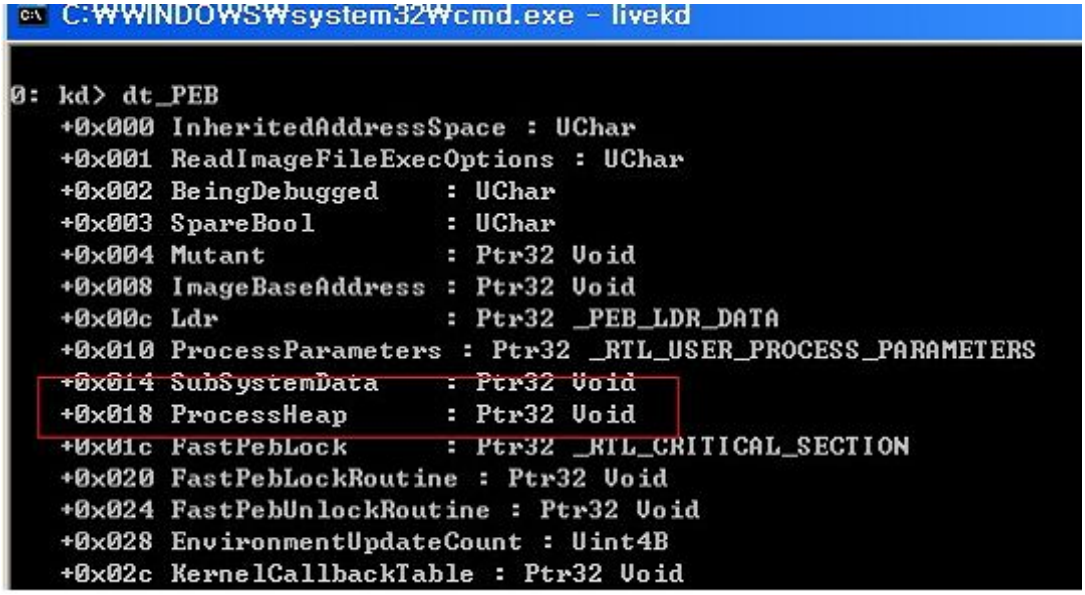
다음은 Olly Advanced NtGlobalFlag 이다.



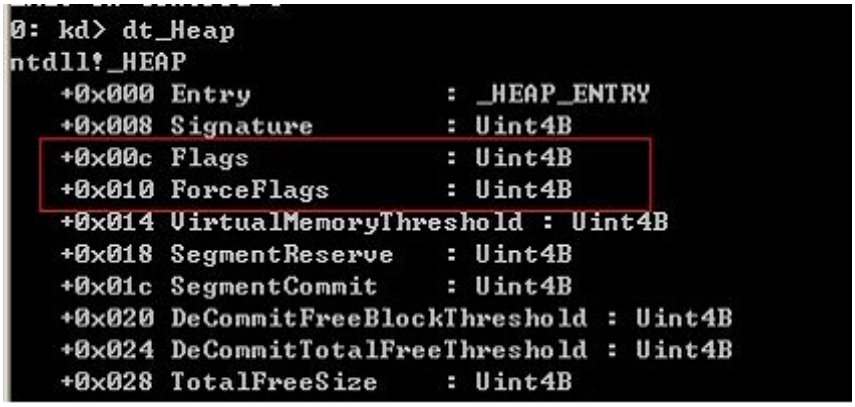
Heap.HeapFlag , Heap.ForceFlags

ProcessHeap 은 PEB 에서 0x18 만큼 떨어져 있고 이 플래그는 프로세서가 디버깅 될 때 heap 이 만들어 지고 이때 설정 되는 플래그들은 HeapFlag와 ForceFlas 이다. 처음 프로그램이 힙 영역을 만들면 ForceFlags 에는 0x0 값이 HeapFlag 에는 0x2 값이 설정된다. 하지만 디버깅 중이라면 NtGlobalFlag 에 따라 두개의 플래그 값이 변경된다.
만약 디버깅이 탐지 되어 변경 되었다면 FoceFlags에는 0x40000060값이 할당 되며 HeapFlag 에는 0x50000062 값이 할당 된다. 이때 변경된 값을 체크함으로써 디버깅의 유무를 판별한다.

다음은 PEB 의 ProcessHeap 구조체 offset 이다.



여기서 ProcessHeap의 구조체 를 살펴보면 언급한 두개의 값들이 존재한다.



다음은 Process_Heap Code 부분이다.

```
 .386
.model flat, stdcall
option casemap :none   ; case sensitive

include c:₩masm32₩include₩windows.inc
include c:₩masm32₩include₩user32.inc
include c:₩masm32₩include₩kernel32.inc

includelib c:₩masm32₩lib₩user32.lib
includelib c:₩masm32₩lib₩kernel32.lib

.data
```

```
DbgNotFoundTitle db "Debugger status:",0h
DbgFoundTitle db "Debugger status:",0h
DbgNotFoundText db "Debugger not found!",0h
DbgFoundText db "Debugger found!",0h
.code

start:

ASSUME FS:NOTHING
MOV EAX,DWORD PTR FS:[18h] ;TEB
MOV EAX,DWORD PTR [EAX+30h] ;PEB
MOV EAX,DWORD PTR[EAX+18h] ;Process_Heap
CMP DWORD PTR DS:[EAX+10h],0 ;Force_Flags
JNE @DebuggerDetected

MOV EAX,DWORD PTR FS:[18h] ;TEB
MOV EAX,DWORD PTR [EAX+30h] ;PEB
MOV EAX,DWORD PTR[EAX+18h] ;Process_Heap
CMP DWORD PTR DS:[EAX+0ch],2 ;Force_Flags
JNE @DebuggerDetected

PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox

JMP @exit
@DebuggerDetected:

PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox

@exit:

PUSH 0
CALL ExitProcess

end start
```

Heap.HeapFlag , Heap.ForceFlags 우회

역시 앞선 방법과 동일하게 코드 패치 및 플래그 변조로 우회를 할 수 있다. 또는 올리 플러그인 OllyAdvanced 를 이용하면 쉽게 플래그 값을 자동으로 수정 해준다.

다음은 Olly Advanced HeapFlag,ForceFlags 우회 부분이다.

NtQueryInformationProcess

CheckRemoteDebuggerPresent API는 디버거가 프로세서를 attach 하는 것을 감지한다. 이 API는 내부적으로 NtQueryInformationProcess를 호출한다. 이 함수는 또한 내부적으로 커널 구조체인 EPROCESS의 DebugPort 플래그를 검사한다. 가령 유저 모드 디버거가 프로세서를 attach 한 상태이면 DebugPort 플래그 값은 0이 아닌값으로 설정이 된다.
앞서 PEB에 대해 간략하게 알아 보았다. PEB은 유저모드 레벨 프로세스에 대한 추가 정보 이며 여기서 설명하는 EPROCESS는 커널에서 프로세스를 관리하기 위해 사용하는 구조체 정도가 되겠다. 또한 NtQueryInformationProcess 을 내부적으로 불러낼 때는 인자 값 중 ProcessInformation 값은 7이 된다.
또한 DebugPort의 오프셋은 0x120 위치에 존재하며 유저모드 에서 디버깅 중이라면 NtQueryInformationProcess 함수의 인자 값 중 hProcess는 0xffffffff 값이 되며 그렇지 않을 경우에는 0이라는 값이 담기게 된다.

다음은 NtQueryInformationProcess Code 부분이다.

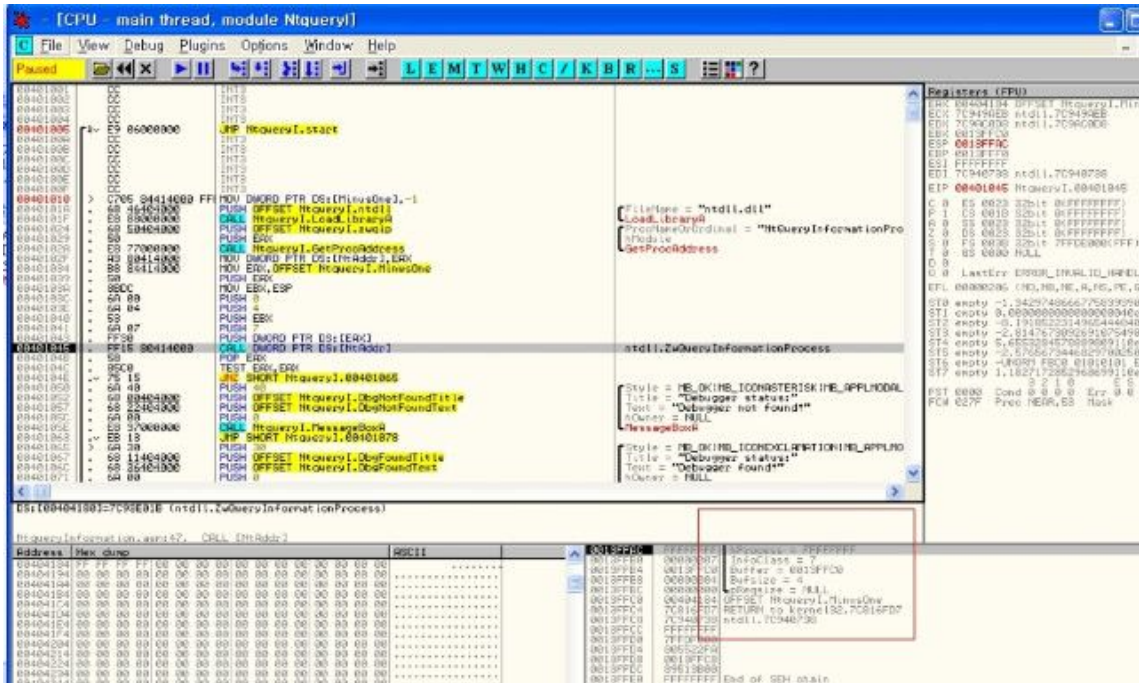
```
.386
        .model flat, stdcall
        option casemap :none    ; case sensitive

        include c:₩masm32₩include₩windows.inc
        include c:₩masm32₩include₩user32.inc
        include c:₩masm32₩include₩kernel32.inc

        includelib c:₩masm32₩lib₩user32.lib
        includelib c:₩masm32₩lib₩kernel32.lib

      .data
          DbgNotFoundTitle db "Debugger status:",0h
```

```
        DbgFoundTitle db "Debugger status:",0h
        DbgNotFoundText db "Debugger not found!",0h
        DbgFoundText db "Debugger found!",0h
        ntdll db "ntdll.dll",0h
        zwqip db "NtQueryInformationProcess",0h
    .data?
        NtAddr dd ?
        MinusOne dd ?
    .code

start:

MOV [MinusOne],0FFFFFFFFh

PUSH offset ntdll ;ntdll.dll
CALL LoadLibrary

PUSH offset zwqip ;NtQueryInformationProcess
PUSH EAX
CALL GetProcAddress

MOV [NtAddr],EAX

MOV EAX,offset MinusOne
PUSH EAX
MOV EBX,ESP

PUSH 0
PUSH 4
PUSH EBX
PUSH 7
PUSH DWORD PTR[EAX]
CALL [NtAddr]

POP EAX

TEST EAX,EAX
JNE @DebuggerDetected

PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox
```

```
JMP @exit
  @DebuggerDetected:

PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox

  @exit:

PUSH 0
CALL ExitProcess

end start
```

위의 코드에서 ? 는 변수를 선언할 때 값을 할당하지 않은 상태일 것이라 생각된다. 어쨌든 위의 코드에서 커널 API를 수행하기 위해서 LoadLibrary와 GetProcAddress를 이용하여 해당 함수 주소를 얻은 뒤, 앞서 설명한 인자값을 토대로 호출 하고 디버깅 중이라면 리턴되는 값을 체크하여 디버거 존재 유무를 판별 하고 있습니다.

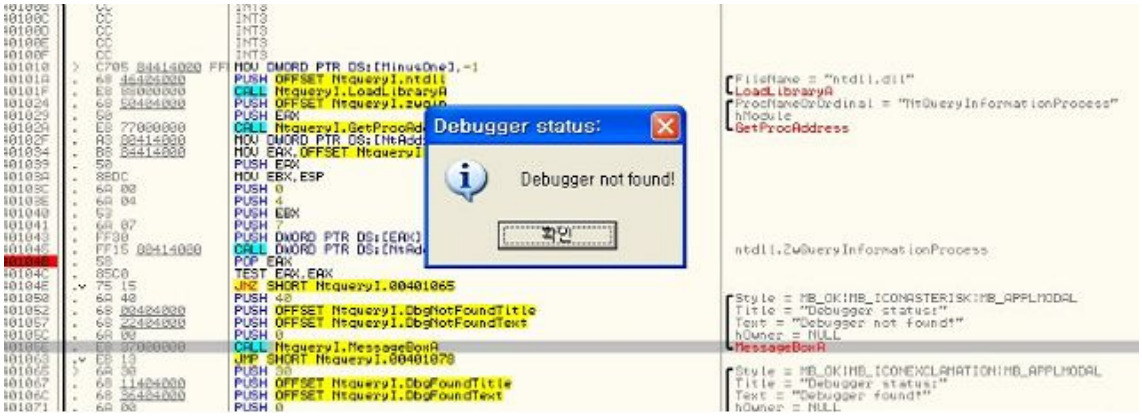
다음 그림은 위의 코드를 Olly에서 확인한 장면(NtQueryInformationProcess in Olly)이다.



그림에서 알 수 있듯이 인자 값이 저런 식으로 들어가 호출하게 되어야지 디버깅 체크를 할 수 있다.

NtQueryInformationProces 우회

이 방법을 우회하는 방법 역시 여러 가지가 존재 할 수 있다. 다만 올리플러그 인중 Olly Advanced 는 NtQueryInformationProcess 의 인자 값중 하나인 hProcess를 0으로 만듦으로써 해당 루틴을 우회하게 될 것이라 생각했으나 잘 되지 않는다. 일단은 그냥 주 루틴이 나오면 리턴되는 부분에 EAX값을 0을 설정하면 우회가 된다.



Debugger Interrupts

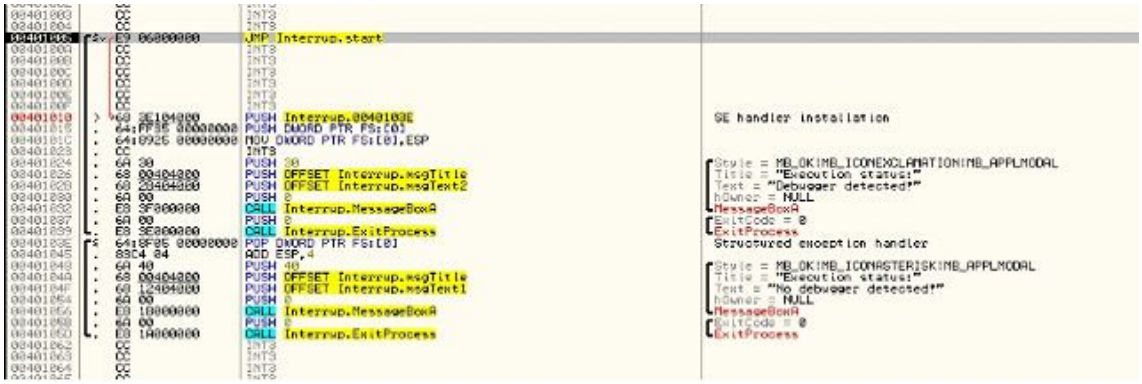
이 방법은 디버거가 하나의 소프트 브레이크 포인트를 수행하려고 할 때 INT3(0xcc)를 삽입 함으로 써 수행이 된다. 인터럽터가 수행이 된다는 말은 즉 예외처리가 일어난다는 의미이다. 하지만 디버거가 디버깅 수행중 일 때 같은 OP 코드인 INT3을 만난 다면 예외 처리 없이 수행을 하게 된다.
이 Debugger Interrupts는 바로 이러한 것을 착안하여 안티 디버깅을 수행한다. 즉 Interrupt 수행중에 같은 인터럽터 코드를 만났을 때 exception을 하지 않는 경우를 디버깅 중이라고 판별하게 된다. 좀 더 유연한 이해를 위해 아래 코드를 참고 한다.

다음은 Debugger Interrupts 부분이다.

```
.386
        .model flat, stdcall
        option casemap :none    ; case sensitive

        include c:₩masm32₩include₩windows.inc
        include c:₩masm32₩include₩user32.inc
        include c:₩masm32₩include₩kernel32.inc

        includelib c:₩masm32₩lib₩user32.lib
        includelib c:₩masm32₩lib₩kernel32.lib

    .data
msgTitle db "Execution status:",0h
```

```
msgText1 db "No debugger detected!",0h
msgText2 db "Debugger detected!",0h
    .code

start:

ASSUME FS:NOTHING
PUSH offset @Check
PUSH FS:[0]
MOV FS:[0],ESP

; Exception
INT 3h

PUSH 30h
PUSH offset msgTitle
PUSH offset msgText2
PUSH 0
CALL MessageBox

PUSH 0
CALL ExitProcess

; SEH handleing
@Check:
POP FS:[0]
ADD ESP,4

PUSH 40h
PUSH offset msgTitle
PUSH offset msgText1
PUSH 0
CALL MessageBox

PUSH 0
CALL ExitProcess

end start
```

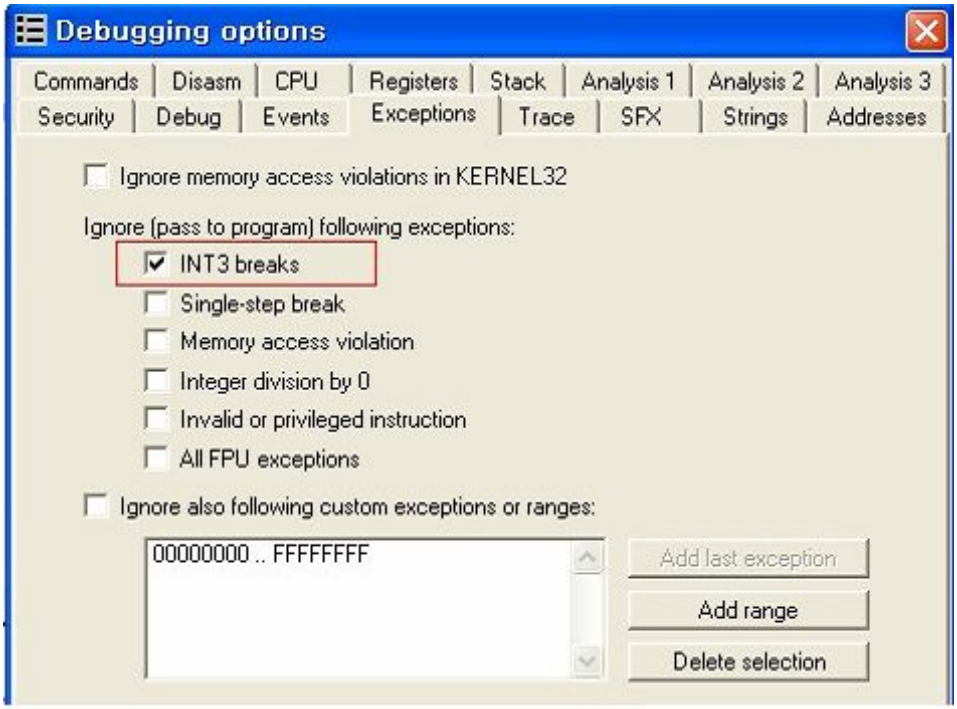
위에 코드에서 @Check는 exception이 일어 날 때 등록되는 SEH 이다. INT 3이 수행될 때 익셉션이 일어나지 않는다면 Debuger dectect 메시지가 출력 될 것이다. 다음은 올리(Debugger Interrupt in Olly)에서 살펴본 모습이다.



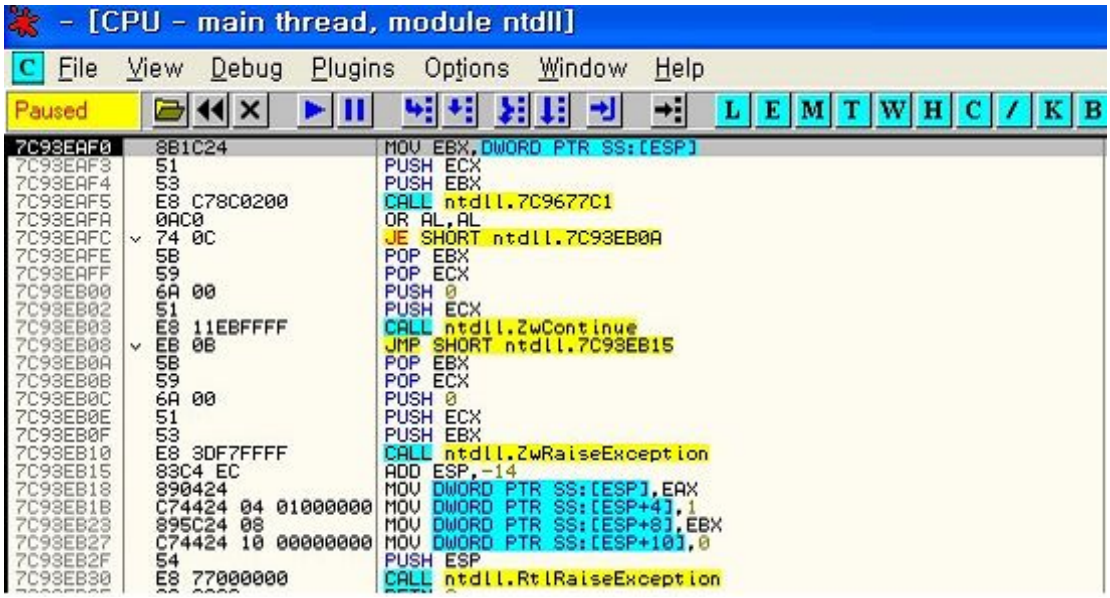
저기에서 INT3을 유심히 살펴보면 저기가 분기문 이라고 이해하는 편이 더 수월 할 수도 있다. INT3이 exception을 일으킨다면 0x0040103e로 향하게 될 것이며 그렇지 않을 경우 0x00401024 코드로 향하게 될 것이다.

5.2 Debug Interrupts 우회

우회 방법은 간단하다. 올리에서 디버깅 옵션에 보면 INT 3 을 만났을 때 무시하지 않고 SEH를 following 하라는 옵션을 선택하여 준다.



다음과 같이 우회를 하게 된다면 Exception Handle로 빠지는 루틴(Exception Handle 을 호출하는 루틴)을 볼 수 있다.


- [CPU - main thread, module ntdll]
File View Debug Plugins Options Window Help
Paused
L E M T W H C / K B
7C93EAF0 8B1C24 MOV EBX,DWORD PTR SS:[ESP]
7C93EAF3 51 PUSH ECX
7C93EAF4 53 PUSH EBX
7C93EAF5 E8 C78C0200 CALL ntdll.7C9677C1
7C93EAFA 0AC0 OR AL,AL
7C93EAFC 74 0C JE SHORT ntdll.7C93EB0A
7C93EAFE 5B POP EBX
7C93EAFF 59 POP ECX
7C93EB00 6A 00 PUSH 0
7C93EB02 51 PUSH ECX
7C93EB03 E8 11EBFFFF CALL ntdll.ZwContinue
7C93EB08 EB 0B JMP SHORT ntdll.7C93EB15
7C93EB0A 5B POP EBX
7C93EB0B 59 POP ECX
7C93EB0C 6A 00 PUSH 0
7C93EB0E 51 PUSH ECX
7C93EB0F 53 PUSH EBX
7C93EB10 E8 3DF7FFFF CALL ntdll.ZwRaiseException
7C93EB15 83C4 EC ADD ESP,-14
7C93EB18 890424 MOV DWORD PTR SS:[ESP],EAX
7C93EB1B C74424 04 01000000 MOV DWORD PTR SS:[ESP+4],1
7C93EB23 895C24 08 MOV DWORD PTR SS:[ESP+8],EBX
7C93EB27 C74424 10 00000000 MOV DWORD PTR SS:[ESP+10],0
7C93EB2F 54 PUSH ESP
7C93EB30 E8 77000000 CALL ntdll.RtlRaiseException