



# [The Art of Unpacking]

Mark Vincent Yason

Malcode Analyst, X-Force Research & Development

IBM internet Security Systems, email:[myason@us.ibm.com](mailto:myason@us.ibm.com)

**(이 문서는 beistlab 멤버 ashine 이 번역한 문서입니다. Email: ash1n2@gmail.com)**

**요 약:** 언패킹은 예술입니다. 이것은 지적인 도전이고 가장 흥분되는 리버싱의 게임입니다. 어떤 경우에는 리버서 들은 운영체제의 본질에 대해서도 알아야 되고 매우 까다로운 안티리버싱 기술과 패커나 프로텍터들에 대해서도 알아야 됩니다. 인내와 영리함은 성공적인 언팩을 하게 해줍니다. 언패킹은 패커를 만든 사람들과 다른 일을 하려는 사람들 모두 도전합니다. 이들 모두 보호를 우회 할려는 사람들입니다.

이문서의 첫 번째 목적은 안티리버싱 기술들과 패커와 프로텍터들에 대해 보여줍니다. 그리고 쉽게 구할수 있는 툴로 프로텍터를 우회하거나 무력하게 만드는 것도 할 수 있습니다. 이정보는 특히 연구원들이 패킹된 악성코드들을 분석 할 수 있게 해줍니다. 그리고 다음장에 나올 안티리버싱 기술은 성공적인 분석을 방해합니다. 그리고 두 번째 목적은 이 정보가 알려지면서 연구원들은 보호코드를 넣어서 리버서들이 소프트웨어를 분석하는 속도를 늦추는 것을 할 수 있게 됩니다. 그러나 물론 리버서들을 멈추게 할 수는 없습니다.

키워드 : reverse engineering, packers, protectors, anti-debugging, anti-reversing

# 차례

1. 서론 .....	3
2. TECHNIQUES : DEBUGGER DETECTION.....	4
2.1 PEB.BEINGDEBUGGED FLAG: ISDEBUGGERPRESENT() .....	4
2.2 PEB.NTGLOBALFLAG, HEAP.HEAPFLAGS, HEAP.FORCEFLAGS.....	5
2.3 DEBUGPORT: CHECKREMOTEDEBUGGERPRESENT() / NTQUERYINFORMATIONPROCESS() .....	7
2.4 DEBUGGER INTERRUPTS .....	10
2.5 TIMING CHECKS .....	12
2.6 SEDEBUGPRIVILEGE .....	14
2.7 PARENT PROCESS .....	15
2.8 DEBUGOBJECT: NTQUERYOBJECT() .....	16
2.9 DEBUGGER WINDOW.....	17
2.10 DEBUGGER PROCESS.....	18
2.11 DEVICE DRIVERS .....	19
2.12 OLLYDBG:GUARD PAGES .....	20
3. THCHMIQUES : BREAKPOINT AND PATCHING DETECTION.....	22
3.1 SOFRWARE BREAKPOINT DETECTION.....	22
3.2 HARDWARE BREAKPOINT DETECTION.....	23
3.3 PATCHING DETECTION VIA CODE CHECKSUM CALCULATION .....	26
4. TECHNIQUES:ANTI-ANALYSIS .....	27
4.1 ENCRYPTION AND COMPRESSION .....	27
4.2 GARBAGE CODE AND CODE PERMUTATION .....	29
4.3 ANTI-DISASSEMBLY .....	34
5. THCHNIQUES : DEBUGGER ATTACKS .....	37
5.1 MISDIRECTION AND STOPPING EXECUTION VIA EXCEPTIONS .....	38
5.2 BLOCKING INPUT.....	40
5.3 THREADHIDEFROMDEBUGGER.....	41
5.4 DISABLING BREAKPOINTS.....	42
5.5 UNHANDLED EXCEPTION FILTER .....	44
5.6 OLLYDBG:OUTPUTDEBUGSTRION() FORMAT STRING BUG .....	45
6.TECHNIQUES:ADVANCED AND OTHER TECHNIQUES .....	46
6.1 PROCESS INJECTION.....	46
6.2 DEBUGGER BLOCKER.....	48
6.3 TLS CALLBACKS.....	49
6.4 STOLEN BYTES .....	51
6.5 API REDIRECTION.....	53
6.6 MULTI-THREADED PACKERS.....	54
6.7 VIRTUAL MACHINES .....	54
7. TOOLS.....	55
7.1 OLLYDBG .....	55
7.2 OLLYSCRIPT .....	56
7.3 OLLY ADVANCED .....	56
7.4 OLLYDUMP.....	57
7.5 IMPREC.....	57
8. REFERENCES.....	57

## 1. 서론

리버스엔지니어링에서 패커는 가장 흥미로운 퍼즐 풀기중 하나입니다. 이 퍼즐을 푸는 과정에 리버서들은 OS 의 본질과 리버싱의 트릭, 툴 그리고 기술에 대해 더 많은 지식을 얻습니다.

패커(이문서 에서는 프로텍터와 실행압축 둘다를 의미)는 파일을 분석하는 것을 보호해줍니다.

패커는 합법적으로 상용프로그램을 크랙 하거나 합부로 변경하는 행위를 예방합니다.

불행하게도 악성코드 또한 패커를 같은 목적으로 사용합니다.

그러나 악의적인 목적으로 사용합니다.

패킹된 악성코드의 숫자가 늘어나는 관계로 연구원과 악성코드 분석자들은 분석을 위해서 언팩 기술을 발전시켰습니다. 그러나 시간이 흐르면서 리버서들의 성공적인 언팩을 방해하기 위해서 새로운 안티리버싱 기술은 끊임없이 패커에 추가 되어졌습니다. 그리고 매년 새로운 안티 리버싱 기술이 개발되면 리버서들은 그 기술을 우회 하는 기술과 툴을 개발합니다.

이 문서의 중점은 패커의 진보된 안티 리버싱 기술과 툴을 어떻게 우회하거나 무력하게 만드는지에 대해 보여줍니다. 반대로 어떤 패커들은 쉽게 우회하거나 무력하게 만들 수 있는데 그래서 그 안티 리버싱 기술은 처리할 필요가 없다는 것을 보여줍니다. 그러나 보호된 코드는 반드시 트레이싱 하고 분석하여야 합니다. 그 예로:

- 보호된 코드의 부분이 우회될 목적으로 덤프 되어 임포트 테이블 리빌딩 툴로 훌륭하게 언팩이 됩니다.
- 보호된 코드는 백신 지원 언패킹의 목적을 위해서 철저히 분석됩니다.

부가적으로, 알려진 안티리버싱 기술이 악성코드에 직접 응용 되 있을 때 악성코드를 분석하고 추적하기 위해 리버싱에 능숙한 것도 가치가 있습니다.

이 문서는 모든 안티리버싱 기술을 나열하지 않았고 알려진 패커 들의 일반적으로 익숙하고 흥미로운 기술에 대해 나와있습니다. 안티리버싱과 리버싱에 대해 더 공부하고 싶은 독자는 마지막장에 있는 레퍼런스를 참고 하기 바랍니다.

필자는 독자들이 이 문서 에서 유용한 팁과 트릭의 기술들에 대해 알게 되었으면 합니다.

해피 언패킹 !

## 2. TECHNIQUES : DEBUGGER DETECTION

이번장의 기술들은 패커가 시스템에서 디버거가 사용중인지 또는 프로세스를 디버그 하고 있는지를 검사하기 위해 사용됩니다. 이 디버거들을 찾아 내는 기술은 매우 단순한(그리고 분명한) 체크입니다. 이것은 native API 들과 커널 객체에서 제공해줍니다.

### 2.1 PEB.BeingDebugged Flag: IsDebuggerPresent()

가장 기본적인 디버거를 찾아 내는 기술은 **Process Environment Block(PEB)**에서 BeingDebugged flag 가 포함되어있는지 검사하는 것입니다. kernel32!IsDebuggerPresent() 는 유저모드의 디버거가 프로세스를 디버깅하고 있는지 flag 값으로 확인합니다.

아래 코드는 IsDebuggerPresent() 함수가 수행중인 것을 볼 수 있습니다. **Thread Environment Block(TEB)** 에 접근하여 PEB 의 주소를 얻은 다음 PEB 를 검사하고 PEB + 0x02 위치의 BeingDebugged flag 를 확인합니다.

```
mov eax, large fs:18h
mov eax, [eax+30h]
movzx eax, byte ptr [eax+2]
retn
```

직접 IsDebuggerPresent()를 불러오는 대신에 일부 패커 들은 PEB 의 BeingDebugged flag 를 수동으로 체크합니다. 그러면 리버서들이 API 를 패치 하거나 브레이크포인트를 거는 것을 방지할 수 있게 됩니다.

### 예제

아래의 예제코드에서 IsDebuggerPresent() 함수와 PEB.BeingDebugged flag 가 디버거를 감지하는 부분을 볼 수 있습니다.

```
; call kernel32!IsDebuggerPresent()
call [IsDebuggerPresent]
test eax,eax
jnz .debugger_found

; check PEB.BeingDebugged directly
mov eax,dword [fs:0x30] ;EAX = TEB.ProcessEnvironmentBlock
```

```

mobzx eax,byte [eax+0x02] ;AL = PEB.BeingDebugged

test eax,eax

jnz .debugger found

```

이 체크들은 매우 정확 하여서 패커 들은 뒤에서 나오는 쓰레기코드와 안티 디어셈블리 기술들과 함께 사용합니다.

## 해결방법

이 기술은 PEB.BeingDebugged flag 를 0x00 값으로 패치 하여 원활하게 우회할 수 있습니다. 올리디버그 에서 PEB 를 보려면 Ctrl+G(Goto Expression)을 누른 뒤 fs:[30] 이라고 입력하면 볼 수 있습니다.

부가적으로는 올리스크립트 명령어인 "dbh" 는 이것을 패치 하여 줍니다:

```
Dbh
```

마지막으로 Olly Advanced 플러그인은 옵션에서 BeingDebugged field 를 0 으로 세팅 해주면 됩니다.

## 2.2 PEB.NtGlobalFlag, Heap.HeapFlags, Heap.ForceFlags

**PEB.NtGlobalFlag.** PEB 의 또 다른 필드인 NtGlobalFlag(offset 0x68)는 디버거가 로드 되어 있는지 감지 하는데 사용합니다. 프로세서가 디버깅중이 아니라면 NtGlobalFlag 필드 값은 0x0 입니다. 그러나 프로세서가 디버그 중이라면 필드 값은 0x70 이 됩니다.

다음의 플래그들을 설정합니다:

- FLG\_HEAP\_ENABLE\_TAIL\_CHECK (0x10)
- FLG\_HEAP\_ENABLE\_FREE\_CHECK (0x20)
- FLG\_HEAP\_VALIDATE\_PARAMETERS (0x40)

이 플래그 는 ntdll!LdrpInitializeExecutionOptions() 에서 설정됩니다.

기본값으로 저장된 PEB.NtGlobalFlag 는 gflags.exe 툴이나 다음의 레지스트리키로 무시 할 수 있습니다:

```
HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
```

**Heap Flags.** NtGlobalFlag 의 설정으로 인해서 몇 개의 플래그 값들이 활성화 됩니다. 그리고 이것은 ntdll!RtlCreateHeap() 함수 내에서 확인할 수 있습니다. 프로세서의 첫 번째 heap 이 생성될 때 플래그들 과 ForceFlags 필드를 0x02 (HEAP\_GROWABLE) 그리고 0x0 으로 설정 해줍니다. 그러나 프로세서가 디버깅중 일때는 이 플래그들은 0x50000062

(NtGlobalFlag 에 따라) 와 0x4000060 (플래그들과 0x6001007D 에 따라) 으로 설정됩니다. 기본적으로 프로세서가 디버그 될 때 heap 이 만들어지고 아래와 같이 플래그가 설정됩니다.

- HEAP\_TAIL\_CHECKING\_ENABLED (0x20)
- HEAP\_FREE\_CHECKING\_ENABLED (0x40)

### 예제

아래의 예제 코드는 PEB.NtGlobalFlag 가 0 이 아니고 프로세서(PEB.ProcessHeap)의 첫번째 heap 이 생성될 때 플래그들이 설정 되는 것을 보여줍니다.:

```
;ebx = PEB
mov edx,[fs:0x30]

;Check if PEB.NtGlobalFlag != 0
cmp dword [ebx+0x68],0
jne .debugger_found

;eax = PEB.ProcessHeap
mov eax,[ebx+0x18]

;Check PEB.ProcessHeap.Flags
cmp dword [eax+0x0c],2
jne .debugger_found

;Check PEB.ProcessHeap.ForceFlags
cmp dword [eax+0x10],0
jne .debugger_found
```

### 해결방법

이 것은 PEB.NtGlobalFlag 와 PEB.HeapProcess 플래그 들이 프로세스가 디버깅 중이 아닌것 처럼 패치 합니다.

예를 들어 올리스크립트로 패치 할 땐 다음과 같이합니다:

```

var peb

var patch_addr

var process_heap

//PEB 에는 하드코딩된 TEB 주소를 가져옵니다. (first thread: 0x7ffed000)

mov peb,[7ffde000+30]

//PEB.NtGlobalFlag 를 패치합니다

lea patch_addr,[peb+68]

mov [patch_addr],0

//PEB.ProcessHeap.Flags/ForceFlags 를 패치합니다.

mov process_heap,[peb+18]

lea patch_addr,[process_heap+0c]

mov [patche_addr],2

lea patch_addr,[process_heap+10]

mov [patch_addr],0

```

또한, Olly Advanced 플러그인 옵션에서 PEB.NtGlobalFlags 와 PEB.ProcessHeap 플래그들을 셋팅 해줘도 됩니다.

### 2.3 DebugPort: CheckRemoteDebuggerPresent() / NtQueryInformationProcess()

Kernel32!CheckRemoteDebuggerPresent() 는 디버거가 프로세스를 어태치 하는 것을 알 수 있습니다. 이 API 에서 ntdll!NtQueryInformationProcess() 을 내부적으로 불러낼 때 ProcessInformationClass 의 파라미터는 ProcessDebugPort(7)이 됩니다. 게다가 NtQueryInformationProcess() 함수는 커널구조체인 EPROCESS 의 DebugPort 의 플래그를 체크합니다. 유저모드의 디버거가 프로세스를 디버깅 중일 때 는 DebugPort 필드에 0 이 아닌값 이 나타납니다. 이 경우에 ProcessInformation 의 값이 0xFFFFFFFF 입니다. 그렇지 않은 경우에는 PorcessInformation 은 0 이 됩니다.

Kernel32!CheckRemoteDebuggerPresent() 함수는 2 개의 파라미터를 받아들입니다. 첫 번째 파라미터는 프로세스 핸들입니다.

그리고 두 번째 파라미터는 부울변수의 포인터입니다. 이것은 프로세스가 디버그중 일 땐 참 값을 가집니다.

```

BOOL CheckRemoteDebuggerPresent(
HANDLE hProcess,
PBOOL pbDebuggerPresent
)

```

Ntdll!NtQueryInformationProcess() 함수는 다른 한편으로는 5 개의 파라미터를 가집니다.

디버거를 디텍팅 하기 위해 PorcessInformationClass 는 ProcessDebugPort(7)를 설정합니다:

```

NTSTATUS NTAPI NtQueryInformationProcess(
HANDLE ProcessHandle,
PROCESSINFOCLASS ProcessInformationClass,
PVOID ProcessInformation,
ULONG ProcessInformationLength,
PULONG ReturnLength
)

```

## 예제

CheckRemoteDebuggerPresent() 함수와 NtQueryInformation() 함수 가 프로세스를 디버깅 중일 때 보여지는 전형적인 콜입니다.

```

; using kernel32!CheckRemoteDebuggerPresent()
lea eax,[.bDebuggerPresent]
push eax ;pbDebuggerPresent
push 0xffffffff ;hProcess
call [CheckRemoteDebuggerPresent]
cmp dword [.bDebuggerPresent],0
jne .debugger_found

```



```

; using ntdll!NtQueryInformationProcess(ProcessDebugPort)

lea eax,[.dwReturnLen]

push eax ;Returnlength

push 4 ;ProcessInformationLength

lea eax,[.dwDebugPort]

push eax ;ProcessInformation

push ProcessDebugPort ;ProcessInformationClass (7)

push 0xffffffff ;ProcessHandle

call [NtQueryInformationProcess]

cmp dword [.dwDebugPort],0

jne .debugger_found

```

### 해결방법

한가지 방법은 NtQueryInformationProcess() 함수가 리턴 되는 곳 에서 브레이크포인트를 설정하여 브레이크포인트가 작동할 때 ProcessInformation 은 DWORD 값을 0 으로 패치 합니다. 아래는 자동으로 작동하는 올리스크립트의 예제입니다.

```

var bp_NtQueryInformationProcess

// set a breakpoint handler

eob bp_handler_NtQueryInformationProcess

// set a breakpoint where NtQueryInformationProcess returns

qpa "NtQueryInformationProcess", "ntdll.dll"

find $RESULT, #c21400# //retn 14

mov bp_NtQueryInformationProcess,$RESULT

bphws bp_NtQueryInformationProcess,"x"

run

bp_handler_NyQueryInformationProcess:

```

```

//ProcessInformationClass == ProcessDebugPort?

cmp [esp+8], 7

jne bp_handler_ntQueryInformationProcess_continue

//patch ProcessInformation to 0

mov patch_addr, [esp+c]

mov [patch_addr], 0

//clear breakpoint

bphwc bp_NtQueryInformationProcess

bp_handler_NtQueryInformationProcess_continue:

run

```

Olly Advanced 플러그인은 NtQueryInformationProcess()를 패치 하는 옵션을 갖고 있습니다.

이 패치는 NtQueryInformationProcess() 의 리턴 값을 조작하여 코드에 삽입합니다.

## 2.4 Debugger Interrupts

이 기술은 디버거 에서 INT3 와 INT1 명령어를 지나갈 때 디버거가 이런 중단점을 그냥 지나 갈수 있기 때문에 기본적으로 예외처리를 하지 않습니다. Debugger Interrupts 는 바로 이런 사실을 이용하고 있습니다. 패커가 예외 핸들러 안에서 플래그들을 설정합니다 그리고 플래그가 INT 명령 이후에 설정 되어있지 않다면 이것은 프로세스가 디버깅중 이라는 것을 의미합니다. 추가적으로 kernel32!DebugBreak() 내부에서 INT3 을 담고 있습니다. 그리고 몇몇 패커는 이 API 를 대신 사용합니다.

### 예제

이 예제는 EAX 를 예외 핸들러 안에서 0xFFFFFFFF(CONTEXT 레코드에 의하여) 값으로 설정합니다. 예외 핸들러 안에서 불러오는 것을 나타냅니다:

```

;set exception handler

push .exception_handler

```

```
push dword [fs:0]

mov [fs:0], esp

;reset flag (eax) invoke int3

xor eax,eax

int3

;restore exception handler

pop dword [fs:0]

add esp,4

;check if the flag had been set

test eax,eax

je .debugger_found

...

.exception_handler:

;EAX = ContextRecord

mov eax, [esp+0xc]

;set flag (ContextRecord,EAX)

mov dword [eax+0xb0],0xffffffff

;set ContextRecord.EIP

inc dword [eax+0xb8]

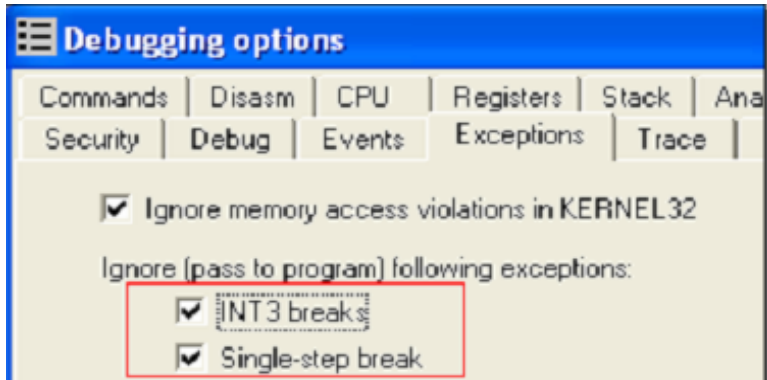
xor eax,eax

retn
```

## 해결방법

올리디버거에서 단계별로 실행하거나 또는 그냥 실행하던 도중에 디버거가 인터럽트에 걸리게 되면 멈추게 됩니다.

브레이크포인트에 걸렸을 때 예외 핸들을 확인합니다. 예외 핸들러 주소(via View -> SEH Chain)를 식별 한 뒤 Shift+F9 를 눌러서 디버거 인터럽트/예외 를 예외 핸들러를 보냅니다. 예외 핸들러로 보내지면 계속 트레이싱 할 수 있게 됩니다.



또 다른 방법은 디버거 인터럽트를 자동으로 예외 핸들러로 보낼 수 있습니다. 올리디버그 에서 Option -> Debugging Options -> Exceptions -> "Ignore Following exceptions" 그리고 "INT 3 Breaks" 그리고 "Single-step break" 를 체크해줍니다.

## 2.5 Timing Checks

프로세스가 디버깅 중일 때 디버거의 이벤트 핸들링코드, 지나온 경로, 명령등이 CPU 를 사용합니다. 리버서들이 단계별로 명령어를 지나갈 때 패커 들은 몇 개의 명령어 사이에서 사용 되어지는 시간을 이용합니다. 일반적인 실행보다 사용시간이 더 길다면 디버거에서 프로세스를 실행중 이라는 것을 의미합니다.

### 예제

아래의 간단한 예제는 시간을 체크하는 예제입니다. 이것은 RDTSC(Read Time-Stamp Counter)명령을 내리기 전과 몇 개의 명령을 실행 한 뒤에 증가량을 계산하는 것을 이용합니다. 증가량값 0x200 은 두 개의 RDTSC 명령이 실행될 때 실행량 에 결정됩니다.

```
Rdtsc
mov ecx,eax
mov ebx,edx
; ... more instructions
nop
push eax
pop eax
```

```

nop

; ... more instructions

;compute delta between RDTSC instructions

rdtsc

;Check high order bits

cmp edx,ebx

ja .debugger_found

;Check low order bits

sub eax,ecx

cmp eax,0x200

ja .debugger_found

```

시간체크는 kernel32!GetTickCount() 함수를 사용하거나 수동으로 TickCountLow 의 값과

TickCountMultiplier 필드의 SharedUserData 데이터 구조체의 주소는 항상 0x7FFE0000 인 것을 체크합니다.

쓰레기코드와 다른 기술들을 사용하여 코드를 숨기고 RDTSC 를 사용하면 더욱 찾기 힘들어집니다.

## 해결방법

한가지 방법은 시간체크를 하는 부분의 코드를 트레이싱하여 우회 합니다. 리버서는 증가량코드를 비교 하기 전에 브레이크포인트를 걸고 브레이크포인트가 걸릴 때까지 트레이싱 합니다. 이외에도 GetTickCount() 함수를 불러올 때 브레이크포인트를 설정하여 리턴값을 수정할 수 있습니다.

Olly Advanced 를 이용한 또 다른 방법 - 커널모드 드라이버를 인스톨하고 프로그램을 시작합니다:

1. control register CR4 안의 Time Stamp Disable Bit(TSD)를 설정합니다. RDTSC 명령이 ring 0 가 아닐 때 실행하여 비트가 설정되면 General Protection (GP) 예외가 유발될 것입니다.

2. Interrupt Descriptor Table (IDT) 가 GP 예외를 후킹 하거나 RTDSC 필터링을 해서 실행하게 설정 합니다. GP 를 후킹 했다면 단지 리턴값 에 1 을 더해주면 됩니다.

시스템에 불안정성을 초래할지도 모르는 드라이버는 주의해야 합니다. 그러므로 virtual machine 이나 부서져도 되는 시스템에서 항상 테스트를 해봐야 됩니다.

## 2.6 SeDebugPrivilege

기본적으로 프로세스는 SeDebugPrivilege 권한이 없습니다. 그러나 프로세스가 올리디버그나 WinDbg 에 의해 불러와 진다면 SeDebugPrivilege 권한을 사용할 수 있습니다. 디버거가 프로세스를 불러올 때 SeDebugPrivilege 권한을 사용할 수 있게 되고 SeDebugPrivilege 권한은 상속 됩니다.

일부 패커 들은 CSRSS.EXE 프로세스를 이용하여 디버깅 중인지를 간접적으로 SeDebugPrivilege 를 사용하여 확인합니다. CSRSS.EXE 프로세스를 열수 있다면 프로세스는 SeDebugPrivilege 권한이 있음을 의미합니다. 그러므로 프로세스가 디버깅중인 것을 알 수 있습니다. 이 체크방법은 보안 디스크립터 인 CSRSS.EXE 프로세스는 오직 시스템권한만이 액세스 할 수 있습니다. 그러나 만약 SeDebugPrivilege 권한 프로세스를 갖고 있다면 보안 디스크립터를 무시하고 다른 프로세스에 접근할 수 있습니다. 기본적인 Administrators 그룹의 멤버 들은 특별권한에 주의해야 합니다.

### 예제

아래는 예제 체크입니다:

```
;query for the PID of CSRSS.EXE
call [CsrGetProcessId]

;try to open the CSRSS.EXE process

push eax

push FALSE

push PROCESS_QUERY_INFORMATION

call [OpenProcess]

;if OpenProcess() was successful,
```

```

; process is probably being debugged

test eax,eax

jnz .debugger_found

```

이 체크는 CSRSS.EXE 의 PID 를 ntdll!CsGetProcessId() API 함수를 사용해서 가져옵니다. 그러나 패커는 수동으로 프로세스를 열거하여 CSRSS.EXE 의 PID 를 얻습니다. OpenProcess() 가 성공한다면 이것은 SeDebugPrivilege 권한이 쓰여지고 있음을 의미하고 프로세스가 디버깅 중이라는 것을 의미합니다.

## 해결방법

한가지 방법은 ntdll!NtOpenProcess() 가 리턴 될 때 브레이크포인트를 걸은 뒤 실행을 하여 브레이크포인트가 걸리면 EAX 값을 0xC0000022 (STATUS\_ACCESS\_DENIED) 로 설정 해줍니다.

## 2.7 Parent Process

전형적으로 프로세스의 부모 프로세스가 explorer.exe(더블클릭 하여 실행했을 때)일 때 부모프로세스가 explorer.exe 가 아니고 다른 어플리케이션에서 생성된 것 일 때 이것은 디버깅중일 가능성이 있습니다.

아래는 이런 검사를 하는 한가지 방법입니다..

1. TEB(TEB.ClientId) 또는 GetCurrentProcessId()를 사용하여 current process 의 PID 를 가져옵니다.
2. Process32First/Next() 함수를 사용해 모든 프로세스 리스트를 얻어 explorer.exe PID (PROCESSENTRY32.szExeFile 을 통하여)와 그리고 현재 프로세스는 PROCESSENTRY32.th32ParentProcessID 를 사용하여 부모 프로세스의 PID 를 얻어 기록해둡니다.
3. 부모 프로세스의 PID 가 explorer.exe 의 PID 가 아니라면 타겟 은 디버깅중일 가능성이 높습니다.

그러나 이 디버거 체크는 cmd 또는 다른셸을 통하여 패커가 실행 중 일땐 오류를 일으킬 수 있음을 주의하여야 합니다.

## 해결방법

Ollly Advanced 가 제공하는 방법은 Process32Next()를 계속 fail 을 리턴 하게 설정하는 것입니다.

이러면 패커의 프로세스 열거 코드가 실패하여 PID 체크를 점프하게 됩니다.

이것은 kernel32!Process32NextW()의 엔트리 부분을 EAX 값을 0 으로 한 다음 바로 리턴 하게 패치해주면 됩니다.

## 2.8 DebugObject: NtQueryObject()

프로세스가 디버깅중인지 확인하는 방법 대신 시스템에서 디버거가 작동중일 때 미치는 영향을 체크하여 디버거가 실행중인지 확인합니다.

리버싱 포럼에서 한가지 재미있는 방법이 논의 됐습니다.. DebugObject 타입의 커널 객체의 번호로 체크하는 방법에 대해서입니다. 이 방법은 어플리케이션이 디버깅중인지 계속해서 체크합니다. 디버깅중에는 커널 안에서 DebugObject 타입의 한 객체가 생성됩니다.

DebugObject 의 번호는 ntdll!NtQueryObject() 함수를 사용하여 모든 객체의 타입의 정보를 얻을 수 있습니다.

NtQueryObject 는 5 개의 파라미터를 받아들입니다. 그리고 모든객체의 타입을 질의하기위해 ObjectHandle 파라미터는 NULL 로 설정되고 ObjectInformationClass 는 ObjectAllTypeInfo(3)입니다:

```
NTSTATUS NTAPI NtQueryObject(
    HANDLE ObjectHandle,
    OBJECT_INFORMATION_CLASS objectInformationClass,
    PVOID objectInformation,
    ULONG Length,
    PULONG ResultLength
)
```

API 가 리턴 될 때 OBJECT\_ALL\_INFORMATION 구조체 안에서 NumberOfObjectsTypes 필드는 ObjectTypeInformation 배열의 총 오브젝트 타입의 개수를 저장합니다.

```
typedef struct _OBJECT_ALL_INFORMATION {
    ULONG NumberOfObjectsTypes;
    OBJECT_TYPE_INFORMATION objectTypeInformation[1];
}
```

이 탐지 루틴은 ObjectTypeInformation 배열의 구조체를



```
typedef struct _OBJECT_TYPE_INFORMATION {
[00] UNICODE_STRING TypeName;

[08] ULONG TotalNumberOfHandles;

[0c] ULONG TotalNumberOfObjects;

... more fields ...
}
```

TypeName 필드는 유니코드 스트링 "DebugObject"를 비교할 때 TotalNumberOfObject 또는 TotalNumberHandles 필드의 값이 0 이 아닌지 체크합니다.

## 해결방법

NtQueryInformationProcess() 해결법과 유사합니다. NtQueryObject() 함수를 리턴 하는 곳 에서 브레이크포인트를 걸어 줄수 있습니다. OBJECT\_ALL\_INFORMATION 구조체가 리턴 될 때 패치 할 수 있습니다. NumberOfObjectsTypes 필드가 0 으로 설정하여 특별하게 ObjectTypeInformation 배열을 통해 패커가 수정 하는 것 을 예방할 수 있습니다.

NtQueryInformationProcess() 와 비슷한 해결 방법은 올리스크립트를 만들어 이 것을 할 수도 있습니다.

비슷한 것으로는 Olly advanced 플러그인으로 NtQueryObject() API 함수에 코드를 집어넣을 수 있다.

ObjectAllTypeInfoInformation 타입의 쿼리가 버퍼에서 리턴 될 때 항상 0 이 되도록 할 수 있습니다.

## 2.9 Debugger Window

시스템에서 디버거가 실행중인지 기록하여 확인하는 디버거 윈도우 가 있습니다. 디버거 윈도우는 특별한 이름을 사용합니다.(OLLYDBG for OllyDbg,WinDbgFrameClass for WinDbg) 이 디버거 윈도우는 user32!FindWindow() 나 user32!FindWindwEx() 함수를 사용하여 쉽게 확인할 수 있습니다.

## 예제

아래의 예제코드는 FindWindow() 함수를 사용하여 올리디버그 나 WinDbg 가 시스템에서 윈도우를 만들어냈는지 확인합니다.

```
push NULL

push .szWindowClassOllyDbg
```

```

call [FindWindowA]

test eax,eax

jnz .debugger_found

push NULL

push .szWindowClassWinDbg

call [FindWindowA]

test eax,eax

jnz .debugger_found

.szWindowClassOllyDbg db "OLLYDBG",0

.szWindowClassWinDbg db "WinDbgFrameClass",0

```

## 해결방법

한가지 방법은 FindWindow()/FindWindowEx() 함수가 들어갈 때 브레이크포인트를 걸어줍니다. 실행하여 브레이크포인트가 걸릴 때 lpClassName 의 스트링 파라미터를 변경해줍니다. 또 한가지 방법은 리턴값을 NULL 로 셋팅 해줍니다.

## 2.10 Debugger Process

시스템에서 실행중인 디버거를 확인하는 또 다른 방법은 시스템의 모든 프로세스중에서 디버거( 예를들어 OLLYDBG.EXE, windbg.exe, 기타등등)프로세스의 이름을 체크하는 것 입니다. Process32First/Next()함수를 사용하여 디버거가 실행 중일 때 프로세스 이름을 확인하여 직접 체크하는 방법입니다.

몇몇 패커 들은 프로세스의 메모리를 kernel32!ReadProcessMemory() 함수를 사용하여 디버거와 관계 있는 문자열( 예를 들어 "OLLYDBG")을 찾을 때 리버서들은 실행중인 디버거의 이름을 바꿉니다. 디버거가 발견되었을 때 패커는 화면에 에러메시지를 띄우고 조용히 끝내거나 디버거를 종료시킵니다.

## 해결방법

비슷한 방법으로 부모 프로세스를 체크합니다. 이 방법은 `kernel32!Process32NextW()` 함수에서 항상 프로세스를 열거하는 것을 실패하게 하면 됩니다.

## 2.11 Device Drivers

디바이스 드라이버에 접근하는 것을 시도하는 시스템에서 활성화된 커널모드의 디버거를 찾아 내는 고전적인 방법입니다. 이 기술은 아주 단순합니다. `kernel32!CrateFile()` 함수가 불러와질 때 커널모드 디버거인 SoftICE 는 잘 알려진 디바이스 이름들을 사용합니다.

### 예제

단순한 체크코드입니다:

```
push NULL
push 0
push OPEN_EXITSTING
push NULL
push FILE_SHARE_READ
push GENERIC_READ
push .szDeviceNameNtice
call [CreateFileA]
cmp eax,INVALID_HANDLE_VALUE
jne .debugger_found

.szDeviceNameNtice db "WW.WNTICE",0
```

일부 버전의 SoftICE 는 디바이스 이름에 숫자를 추가하여 체크를 실패하게 만듭니다. 리버싱 포럼에서 부르트포싱을 하여 디바이스의 이름을 찾아 내는 방법이 얘기 되었습니다. 또한 새로운 패커들은 디바이스 드라이버를 찾아 내는 기술을 이용하여 Regmon 이나 Filemon 같은 시스템모니터링 툴도 찾아 냅니다.

## 해결방법

간단한 해결방법은 kernel32!CreateFileFileW() 함수 안에서 브레이크포인트를 걸은 뒤, 프로그램이 실행되어 브레이크포인트가 걸리면 파일이름의 파라미터를 바꿔주거나 INVALID\_HANDLE\_VALUE (0xFFFFFFFF) 으로 리턴값을 조작합니다.

## 2.12 OllyDbg:Guard Pages

이 검사는 올리디버그만을 체크하는 기술입니다. 올리디버그는 메모리에 on-access/write 를 하여 브레이크포인트를 걸 수 있는데 이 특성을 이용합니다.

하드웨어, 소프트웨어 브레이크 포인트 외에 올리디버그는 메모리에 on-access/write 브레이크포인트를 걸 수 있습니다.

이러한 종류의 브레이크는 Guard page 를 통해 수행됩니다. 간단히 말하면 Guard page 는 응용프로그램의 어느 한 메모리 부분에 접근할 때 이런 경로를 얻을 수 있게끔 제공해 줍니다.

Guard page 는 PAGE\_GUARD 의 page protection modifier 을 통해 설정됩니다. 접근한 메모리주소가 Guard page 의 주소이면 STATUS\_GUARD\_PAGE\_VIOLATION (0x80000001) 예외를 일으킵니다. 그러나 올리디버그로 디버깅하여 Guard page 에 접근하게 되면 예외는 발생하지 않게 됩니다. 이러한 접근은 메모리 브레이크로 처리하게 됩니다.

## 예제

아래의 예제코드 에서는 메모리를 할당하고 할당된 메모리에 코드를 넣어서 PAGE\_GUARD 의 속성을 enable 시켜줍니다.

할당된 메모리에 page guard 가 실행되게 되면 eax 가 0 이 되고 STATUS\_GUARD\_PAGE\_VIOLATION 을 일으킵니다.

올리디버그로 이 코드를 디버깅하면 표시된 부분의 예외 핸들러를 불러오지 않게 됩니다.

```

; set up exception handler

push .exception_handler

push dword [fs:0]

mov [fs:0], esp

; allocate memory

push PAGE_READWRITE

push MEM_COMMIT

```

```
push 0x1000

push NULL

call [VirtualAlloc]

test eax,eax

jz .failed

mov [.pAllocatedMem],eax

; store a RETN on the allocated memory

mov byte [eax],0xc3

; then set the PAGE_GUARD attribute of the allocated memory

lea eax,[.dwOldProtect]

push eax

push PAGE_EXECUTE_READ | PAGE_GUARD

push 0x1000

push dword [.pAllocatedMem]

call [VirtualProtect]

; set marker (eax) as 0

xor eax,eax

; trigger a STATUS_GUARD_PAGE_VIOLATION exception

call [.pAllocatedMem]

; check if marker had not been changed (exception handler not called)

test eax,eax

je .debugger_found

:::

.exception_handler
```

```

;EAX = CONTEXT record

mov eax, [esp+0xc]

;set marker (CONTEXT.EAX) to 0xffffffff

; to signal that the exception handler was called

mov dword [eax+0xb0],0xffffffff

xor eax,eax

retn

```

## 해결방법

가드 페이지들이 예외를 발생시키므로 리버서들은 의도적으로 예외 핸들러를 불러올 수 있습니다. 이 예제중 에서 리버서는 RETN 명령을 INT 3 명령으로 바꿀 수 있습니다. INT3 으로 바꾼뒤 Shift +F9 를 눌러 강제로 디버거에게 예외 핸들러를 불러오도록 합니다. 예외 핸들러가 불러와질 때 EAX 는 정확한 값으로 설정되게 됩니다. 그리고 RETN 명령을 실행하게 됩니다.

만약 예외 핸들러가 STATUS\_GUARD\_PAGE\_VIOLATION 에 이상이 있는지 체크 하려고 한다면 리버서는 예외 핸들러에 브레이크포인트를 걸은뒤에 ExceptionRecord 의 전달되는 파라미터를 수정 합니다. 구체적으로 말하면 ExceptionRecord 입니다. 수동으로 Exception Code 를 STATUS\_GUARD\_PAGE\_VIOLATION 으로 설정하면 됩니다.

## 3. THCHMIQUES : BREAKPOINT AND PATCHING DETECTION

이번 장에서는 패커가 자주 사용하는 소프트웨어 브레이크포인트,하드웨어 브레이크포인트와 패칭 디텍션 대해 얘기하겠습니다.

### 3.1 Software Breakpoint Detection

소프트웨어 브레이크포인트는 브레이크포인트를 걸을 곳에 0xCC (INT3/Breakpoint Interrupt)로 설정 하는 것 입니다. 패커는 보호된 코드와 API 함수 안에서 0xCC 로 된 바이트를 찾아내어 소프트웨어 브레이크포인트를 식별합니다.

#### 예제

아래와 같은 간단한 방법으로 체크합니다.

```

Cld
mov edi,Protected_Code_Start
mov ecx,Protected_Code_End - Protected_Code_Start
mov al,0xcc
repne scasb
jz .breakporin_found

```

어떤 패커는 비교되는 바이트를 약간의 연산을 하여 정확하게 체크하기가 어렵습니다.

```
if(byte XOR 0x55 == 0x99) then breakpoint found
```

```
where: 0x99 == 0xCC XOR 0x55
```

## 해결방법

소프트웨어 브레이크포인트를 확인 할 수 있다면 리버서들은 그대신 하드웨어 브레이크포인트를 사용할 수 있습니다. API 코드 안에서 브레이크포인트가 필요 할 때 패커가 API 코드 안에서 브레이크포인트를 찾는 것을 시도한다면 리버서들은 UNICODE 버전의 API 함수를 ANSI 버전(예: LoadLibraryExW 대신 LoadLibraryA)의 함수 또는 일치하는 Native API (ntdll!LdrLoadDll) 함수로 대신 불러 올 수도 있습니다.

## 3.2 Hardware Breakpoint Detection

또 다른 타입의 브레이크포인트는 하드웨어 브레이크포인트입니다. 하드웨어 브레이크포인트는 디버거 레지스터들로 설정됩니다. 이 레지스터들의 이름은 Dr0 부터 Dr7 까지 있습니다. Dr0 - Dr3 에는 4 개의 브레이크포인트 주소를 포함하고 있습니다. Dr6 은 브레이크포인트가 어디서 유발 되었는지를 담고 있습니다.. Dr7 은 4 개의 하드웨어 브레이크포인트를 읽기/쓰기/중지를 enabling/disabling 하는 것을 담고 있습니다.

Ring3 모드 에서는 디버거 레지스터에 접근할 수 없어서 하드웨어 브레이크 포인트를 찾을 수 없기 때문에 미리 코드를 준비해야 됩니다.. 그래서 패커는 CONTEXT 구조체를 가지고 있는 디버거 레지스터의 값을 이용합니다. CONTEXT 구조체는 예외 핸들러에서 발생한 ContextRecord 파라미터에 접근 할수 있습니다.

## 예제

여기의 예제는 디버그 레지스터들과 질의를 합니다.

```
; set up exception handler

push .exception_handler

push dword [fs:0]

mov [fs:0], esp

; eax will be 0xffffffff if hardware breakpoints are identified

xor eax,eax

; throw an exception

mov dword[eax],0

; restore exception handler

pop dword [fs:0]

add esp,4

; test if EAX was updated (breakpoint identified)

test eax,eax

jnz .breakpoint_found

...

.exception_handler

;EAX = CONTEXT record

mov eax,[esp+0xc]
```



```
;check if debug Registers Context. Dr0-Dr3 is not zero
```

```
cmp dword [eax+0x04],0
```

```
jne .hardware_bp_found
```

```
cmp dword [eax+0x08],0
```

```
jne .hardware_bp_found
```

```
cmp dword [eax+0x0c],0
```

```
jne .hardware_bp_found
```

```
cmp dword [eax+0x10],0
```

```
jne .hardware_bp_found
```

```
jmp .exception_ret
```

```
.hardware_bp_found
```

```
; set Context.EAX to signal breakpoint found
```

```
mov dword [eax+0xb0],0xffffffff
```

```
.exception_ret
```

```
; set Context.EIP upon return
```

```
add dword [eax+0xb8],6
```

```
xor eax,eax
```

```
retn
```

어떤 패커는 디버그 레지스터들의 일부를 복호화 키로 사용합니다. 이런 레지스터들은 특정 값으로 초기화 하거나 0 으로 값을 줍니다. 그래서 디버그 레지스터가 수정되면 복호화가 실패하게 됩니다. 이 복호화 코드가 패킹된 프로그램의 언패킹하는 코드의 일부분 일때 코드가 수정되게 되면 복호화가 실패하게 되면 유효하지 않은 명령이 발생되어 예상치 못한 종료가 일어납니다.

## 해결방법

패커가 소프트웨어 브레이크포인트를 체크하지 않는다면 리버서들은 소프트웨어 브레이크포인트를 사용할 수 있습니다. 올디버거를 사용하여 메모리에 접근/쓰기 할 때 브레이크포인트를 걸 수 있습니다. 리버서들이 API 브레이크포인트의 설정이 필요할 때 Native API 또는 UNICODE 버전의 API 내부에 소프트웨어 브레이크포인트를 설정해도 됩니다.

### 3.3 Patching Detection via Code Checksum Calculation

패칭 디텍션(Patching Detection)은 코드를 수정 하였는지 식별할 수 있습니다. 패커의 코드를 수정 한다는 것은 안티 디버깅루틴이 비활성화 된 것을 의미합니다. 그리고 소프트웨어 브레이크포인트를 설정하였는지도 식별할 수 있습니다. 패칭 디텍션은 코드를 체크섬 합니다. 체크섬 계산은 간단한 계산 과 복잡한 체크섬/해쉬 알고리즘을 포함합니다.

#### 예제

아래는 비교적 간단한 체크섬 계산 예제입니다.

```

mov esi,Protected_Code_Start

mov ecx,Protected_Code_End - Protected_Code_Start

xor eax,eax

.checksum_loop

movzx ebx,byte [esi]

add eax,ebx

rol eax,1

inc esi

loop .checksum_loop

cmp eax,dword [.dwCorrectChecksum]

jne .patch_found

```

#### 해결방법

코드 체크섬 루틴에서 소프트웨어 브레이크포인트를 식별해냈다면 하드웨어브레이크포인트로 대체할 수 있습니다.

체크섬에서 코드가 패치 되는 것을 식별해 냈다면 리버서는 패치 된 주소를 통하여 메모리 접근 브레이크포인트를 설정하여 체크섬 코드루틴이 있는 곳을 찾을 수 있습니다. 그리고 코드 체크섬 루틴을 발견했다면 체크섬값 과 추측 값을 수정하거나 비교가 실패한 후 적절한 플래그로 변경해줍니다.

## 4. TECHNIQUES:ANTI-ANALYSIS

anti-analysis 기술은 리버서들이 패킹된 프로그램이나 보호된 코드를 이해하고 분석하는 속도를 늦춰줍니다.

암호화/압축, 쓰레기코드, 코드변형, 안티 디어셈블리 기술들을 얘기합니다. 이런 기술의 목표는 코드를 뒤섞어 인내심을 시험하거나 리버서의 시간을 낭비하기 위함 입니다. 이런 문제를 해결하려면 리버서는 인내심이 있어야 하고 똑똑해야 합니다.

### 4.1 Encryption and Compression

암호화 하거나 압축을 하는 것은 가장 기본적인 유형의 anti-analysis 입니다. 리버서가 보호된 실행프로그램을 불러와 디어셈블러로 편안하게 분석을 시작하려고 할 때 초기에 방어 합니다. **암호화**. 패커는 보통 보호된 프로그램과 보호 코드 둘다 암호화 합니다. 암호화 알고리즘은 패커 안에서 많은 변화를 합니다. 아주 간단한 XOR 반복 부터 매우 복잡한 계산도 있습니다. 다형성 패커는 분석 툴이 패커의 정확한 정보를 식별하는 것을 막기 위하여 패킹을 할 때마다 변화를 주는 방식의 암호화 알고리즘을 사용합니다.

복호화 루틴은 하나의 수를 계산하고 데이터 연산을 저장하는 것을 반복하기 때문에 쉽게 알아 볼 수 있습니다.

아래의 예제는 암호화된 DWORD 값을 여러번 XOR 연산 하는 간단한 복호화 루틴입니다.

```
0040A07c LODS DWORD PTR DS:[ESI]
0040A07d XOR EAX,EBX
0040A07f SUB EAX,12338CC3
0040A084 ROL EAX,10
0040A087 XOR EAX,799F82D0
0040A08C STOS DWORD PTR ES:[EDI]
0040A08D INC EBX
0040A08E LOOPD SHORT 0040A07c ;decryption loop
```

이곳은 또 다른 다형성 패커의 복호화 루틴 예제입니다.

**00476056 MOV BH,BYTE PTR DS:[EAX]**

00476058 INC ESI

00476059 ADD BH,0BD

0047605C XOR BH,CL

0047605E INC ESI

0047605F DEC EDX

**00476060 MOV BYTE PTR DS:[EAX],BH**

00476062 CLC

00476063 SHL EDI,CL

::: More garbage code

00476079 INC EDX

0047607A DEC EDX

**0047607B DEC EAX**

0047607C JMP SHORT 0047607E

0047607E DEC ECX

0047607F JNZ 00476056 ;decryption loop

그리고 아래는 같은 다형성 패커의 또 다른 복호화 루틴을 생성하고 있습니다.

**0040C045 MOV CH,BYTE PTR DS:[EDI]**

0040C047 ADD EDX,EBX

0040C049 XOR CH,AL

0040C04B XOR Ch,0D9

0040C04E CLC

**0040C04F MOV BYTE PTR DS:[EDI],CH**

0040C051 XCHG AH,AH

0040C053 BTR EDX,EDX

0040C056 MOVSX EBX,CL

```

::: More garbage code

0040C067 SAR EDX,CL

0040C06C NOP

0040C06D DEC EDI

0040C06E DEC EAX

0040C06F JMP SHORT 0040C071

0040C071 JNZ 0040C045 ;decryption loop

```

위의 두 예제에서 굵은 글씨는 복호화 명령에서 중요한 부분입니다. 그 외의 부분은 리버서를 혼란스럽게 만드는 쓰레기 코드입니다. 어떻게 레지스터가 변하고 바뀌는지 그리고 두 예제에서 복호화 방법이 어떻게 바뀌었는지도 잘 살펴 봐야 합니다.

**압축.** 압축의 주목적은 실행코드와 데이터들의 크기를 줄이는 것입니다. 그러나 이 결과로 원본 실행파일의 읽을 수 있는 문자열들도 압축된 데이터가 됩니다. 이것은 우리 리버서들을 난처하게 만듭니다. 패커 들이 사용하는 압축 엔진의 몇가지 예를 보면 UPX 는 NRV(Not Really Vanished) 와 LZMA(Lempei-Ziv-Markov chain-Algorithm)을 사용하고 FSG 는 aPlib, Upack 도 LZMA, yoda's protector 는 LZO 를 사용합니다. 몇몇 압축 엔진은 무료이고 비상업적으로 사용할 수 있지만 상업적으로 사용하려면 허가/등록 이 필요합니다.

## 해결방법

복호화와 압축을 푸는 것은 아주 쉽게 우회 할 수 있습니다. 리버서는 단지 복호화와 압축을 푸는 것이 언제 끝나는 지만 찾아낸다면 끝날 때 브레이크포인트를 걸어주면 됩니다. 어떤 패커는 복호화가 끝난 뒤에 브레이크포인트를 감지하는 코드를 넣기도 합니다.

## 4.2 Garbage Code And Code Permutation

**쓰레기 코드.** 언패킹 루틴에 쓰레기 코드를 넣는 것은 리버서를 혼란스럽게 하는 효과적인 방법입니다. 복호화 루틴 이나 안티 리버싱 루틴, 디버거 체크 기능을 하는 코드를 숨기는 것이 진짜 목적입니다. 쓰레기 코드가 추가되면 이 문서에서 나오는 디버거/브레이크포인트/패칭 검사 기술들을 숨겨서 아무것도 하지 않는 것처럼 보이거나 복잡한 명령으로 보이게 합니다. 게다가 효과적인 쓰레기 코드는 올바른 일을 하는 코드처럼 보입니다.

**예제**

아래의 예제는 복호화 루틴에 쓰레기 코드를 넣은 것입니다.

```
0044A21A JMP SHORT sample.0044A21F
0044A21C XOR DWORD PTR SS:[EBP],6E4858D
0044A223 INT 23
0044A225 MOV ESI,DWORD PTR SS:[ESP]
0044A228 MOV EBX,2C322FF0
0044A22D LEA EAX,DWORD PTR SS:[EBP+6EE5B321]
0044A233 LEA ECX DWORD PTR DS:[ESI+543D583E]
0044A239 ADD EBP,742C0F15
0044A23F ADD DWORD PTR DS:[ESI],3CB3AA25
0044A245 XOR EDI,7DAC77E3
0044A24B CMP EAX,ECX
0044A24D MOV EAX,5ACAC514
0044A252 JMP SHORT sample.0044A257
0044A254 XOR DWORD PTR SS:[EBP],AAE4725
0044A25B PUSH ES
0044A25C ADD EBP,5BAC5C22
0044A262 ADD ECX,3D71198C
0044A268 SUB ESI,-4
0044A26B ADC ECX,3795A210
0044A271 DEC EDI
0044A272 MOV EAX,2F57113F
0044A277 PUSH ECX
0044A278 POP ECX
0044A279 LEA EAX,DWORD PTR SS:[EBP+3402713D]
0044A27F DEC EDI
```

```

0044A280 XOR DWORD PTR DS:[ESI],33B568E3
0044A286 LEA EBX,DWORD PTR DS:[EDI+57DEFEE2]
0044A28C DEC EDI
0044A28D SUB EBX,7ECDAE21
0044A293 MOV EDI,185C5C6C
0044A298 MOV EAX,4713E635
0044A29D MOV EAX,4
0044A2A2 ADD ESI,EAX
0044A2A4 MOV ECX,1010272F
0044A2A9 MOV ECX,7A49B614
0044A2AE CMP EAX,ECX
0044A2B0 NOT DWORD PTR DS:[ESI]

```

예제에서 복호화에 관련된 명령어 :

```

0044A225 MOV ESI,DWORD PTR SS:[ESP]
0044A23F ADD DWORD PTR DS:[ESI],3CB3AA25
0044A268 SUB ESI,-4
0044A280 XOR DWORD PTR DS:[ESI],33B568E3
0044A29D MOV EAX,4
0044A2A2 ADD ESI,EAX
0044A2B0 NOT DWORD PTR DS:[ESI]

```

**코드 변형.** 코드 변형은 좀더 진보된 패커들이 사용하는 기술입니다. 코드 변형을 통하여 간단한 명령어를 좀더 복잡한 명령어들로 변환시킵니다. 패커가 명령어를 이해하고 이와 비슷한 새로운 명령어들로 생성을 합니다.

간단한 코드 변형의 예입니다:

```

mov eax,ebx
test eax,eax

```

아래의 명령어들로 변환합니다:

```
push ebx  
  
pop eax  
  
or eax,eax
```

쓰레기 코드와 같이 사용하는 코드변형은 리버서가 보호된 코드를 이해하는 속도를 늦춰주는 효과적인 기술입니다.

## 예제

예를 들어 설명하겠습니다. 아래의 예제코드는 디버거 탐지 루틴에 코드변형과 쓰레기 코드를 넣었습니다.

```
004018A3 MOV EBX,A104B3FA  
  
004018A8 MOV ECX,A104B412  
  
004018AD PUSH 004018C1  
  
004018B2 RETN  
  
004018B3 SHR EDX,5  
  
004018B6 ADD ESI,EDX  
  
004018B8 JMP SHORT 004018BA  
  
004018BA XOR EDX,EDX  
  
004018BC MOV EAX,DWORD PTR DS:[ESI]  
  
004018BE STC  
  
004018BF JB SHORT 004018DE  
  
004018C1 SUB ECX,EBX  
  
004018C3 MOV EDX,9A01AB1F  
  
004018C8 MOV ESI,DWORD PTR FS:[ECX]  
  
004018CB LEA ECX,DWORD PTR DS:[EDX+FFFF7FF7]  
  
004018D1 MOV EDX,600  
  
004018D6 TEST ECX,2B73  
  
004018DC JMP SHORT 004018B3  
  
004018DE MOV ESI,EAX  
  
004018E0 MOV EAX,A35ABDE4
```



```

004018E5 MOV ECX,FAD1203A
004018EA MOV EBX,51AD5EF2
004018EF DIV EBX
004018F1 ADD BX,445A
004018F6 ADD ESI,EAX
004018F8 MOVZX EAX,BYTE PTR DS:[ESI]
004018FB OR EDI,EDI
004018FD JNZ SHORT 00401906

```

이 예제는 간단한 디버거 탐지 루틴인 것을 보여줍니다:

```

00401081 MOV EAX,DWORD PTR FS:[18]
00401087 MOV EAX,DWORD PTR DS:[EAX+30]
0040108A MOVZX EAX,BYTE PTR DS:[EAX+2]
0040108E TEST EAX,EAX
00401090 JNZ SHORT 00401099

```

## 해결방법

쓰레기 코드들과 코드 변형은 인내심을 시험하고 리버서의 시간을 낭비합니다. 그러므로 이런 기술이 숨긴 중요한 명령어를 아는 것이 중요합니다(예:단지 복호화만 하는지 패커가 초기화인지 등등).

이러한 숨어있는 명령어를 트레이싱 할 수 있는 방법 중 하나는 패커가 가장 자주 사용하는 API 함수(예: VirtualAlloc, VirtualProtect, LoadLibrary, GetProcAddress, 등등)들에 브레이크포인트를 걸어 이런 API 들을 추적 목표로 잡고 트레이싱 하면 됩니다. 만약 그곳에서 에러(디버거 나 브레이크포인트 탐지)가 발생한다면 그 코드를 추적 목표로 잡고 자세하게 코드를 분석합니다. 추가적으로 리버서는 접근/쓰기 브레이크포인트들을 설정하여 보호된 코드의 필요한 부분을 수정/접근 해야지 대량의 코드를 분석 하는 건 좋지 않습니다.

마지막으로, VMWare 에서 올리디버그를 실행하여 주기적으로 디버깅 세션을 저장하여야 합니다. 이러면 리버서는 어느 한 특정 추적 부분의 상태로 돌아올 수 있습니다. 에러가 나도 특정세션부분으로 돌아와 계속 분석할 수 있습니다.

### 4.3 Anti-Disassembly

리버서를 혼란스럽게 하는 또 다른 방법은 디스어셈블리를 혼란스럽게 출력하는 것입니다.

안티 디스어셈블리는 정적인 분석을 거친 바이너리코드의 과정을 이해하여 복잡하게 만드는 방법입니다. 그리고 쓰레기 코드와 코드 변형을 같이 사용한다면 더욱 효과적일 것입니다.

안티 디스어셈블리기술의 한가지 예로는 쓰레기 바이트를 넣어두고 분기문에서 넘어와 쓰레기 바이트를 실행하게 됩니다. 하지만 분기문은 항상 FALSE 이기 때문에 영원히 쓰레기 바이트들은 실행되지 않습니다. 그러나 디스어셈블러는 쓰레기 바이트 주소를 디스어셈블링을 하여 최종적으로 잘못된 디스어셈블리 출력을 하게 되는 트릭입니다.

#### 예제

이 예제는 간단한 PEB.BeingDebugged 플래그 체크 에 안티 디스어셈블리 코드를 추가한 것 입니다. 굵은 글씨로 된 부분은 주요 명령어이고 그 외는 안티 디스어셈블리 코드입니다. 쓰레기 바이트인 0xFF 를 추가하여 분기문에서 거짓 조건으로 점프한 뒤 쓰레기 바이트들을 디스어셈블러가 트레이싱 합니다:

```
;Anti-disassembly sequence #1
```

```
push .jmp_real_01
```

```
stc
```

```
jnc .jmp_fake_01
```

```
retn
```

```
.jmp_fake_01:
```

```
db 0xff
```

```
.jmp_real_01:
```

```
mov eax,dword [fs:0x18]
```

```
;Anti-disassembly sequence #2
```

```
push .jmp_real_02
```

```
clc
```

```
jc .jmp_fake_02
```

```
retn
```

```

 jmp_fake_02:
 db 0xff
 jmp_real_02
 ;-----
 mov eax,dword [eax+0x30]
 movzx eax,byte [eax+0x02]
 test eax,eax
 jnz .debugger_found

```

아래는 WinDbg 의 디스어셈블리 출력입니다:

```

0040194a 6854194000 push 0x401954
0040194f f9 stc
00401950 7301 jnb image00400000+0x1953 (00401953)
00401952 c3 ret
00401953 ff64a118 jmp dword ptr [ecx+0x18]
00401957 0000 add [eax],al
00401959 006864 add [eax+0x64],ch
0040195c 194000 sbb [eax],eax
0040195f f8 clc
00401960 7201 jb image00400000+0x1963 (00401963)
00401962 c3 ret
00401963 ff8b40300fb6 dec dword ptr [ebx+0xb60f3040]
00401969 40 inc eax
0040196a 0285c0750731 add al,[ebp+0x310775c0]

```

그리고 올리디버그 의 디스어셈블리 출력입니다:

```

0040194A 68 54194000 PUSH 00401954
0040194F F9 STC

```

```

00401950 73 01 JNB SHORT 00401953

00401952 C3 RETN

00401953 FF64A1 18 JMP DWORD PTR DS:[ECX+18]

00401957 0000 ADD BYTE PTR DS:[EAX],AL

00401959 0068 64 ADD BYTE PTR DS:[EAX+64],CH

0040195C 1940 00 SBB DWORD PTR DS:[EAX],EAX

0040195F F8 CLC

00401960 72 01 JB SHORT 00401963

00401962 C3 RETN

00401963 FF8B 40300FB6 DEC DWORD PTR DS:[EBX+B60F3040]

00401969 40 INC EAX

0040196A 0285 C0750731 ADD AL,BYTE PTR SS:[EBP+310775C0]

```

마지막으로 IDAPro 디스어셈블리의 출력입니다:

```

0040194A push (offset loc_401953+1)

0040194F stc

00401950 jnb short loc_401953

00401952 retn

00401953 ; -----

00401953

00401953 loc_401953: ; CODE XREF: sub_401946+A

00401953 ; DATA XREF: sub_401946+4

00401953 jmp dword ptr [ecx+18h]

00401953 sub_401946 endp

00401953

00401953 ; -----

00401957 db 0

00401958 db 0

```

```
00401959 db 0
0040195A db 68h ; h
0040195B db offset unk_401964
0040195F db 0F8h ; °
00401960 db 72h ; r
00401961 db 1
00401962 db 0C3h ; +
00401963 db 0FFh
00401964 unk_401964 db 8Bh ; ï
00401965 db 40h ; @
00401966 db 30h ; 0
00401967 db 0Fh
00401968 db 0B6h ; |
00401969 db 40h ; @
0040196A db 2
0040196B db 85h ; à
0040196C db 0C0h ; +
0040196D db 75h ; u
```

이 세가지 디스어셈블러/디버거가 어떻게 안티 디스어셈블리 트릭에 빠져들었는지 주목해야 합니다. 이것들은 리버서가 코드를 분석하는데 매우 귀찮고 혼란스럽습니다. 다른 방법으로도 리버서를 혼란스럽게 할 수 있습니다. 이것은 단지 그 중 하나일 뿐입니다. 추가적으로 안티 디스어셈블리 코드는 매크로로 코딩할 수 있어서 어셈블리 소스는 매우 깨끗합니다.

독자들은 Eldad Eliam의 훌륭한 리버싱 책을 참고하길 바랍니다. 이 책에는 안티 디스어셈블리 기술과 다른 리버싱 주제에 대해 자세하게 나와 있습니다.

## 5. THCHNIQUES : DEBUGGER ATTACKS

이번 장에서는 패커가 디버거를 공격하는 기술을 열거하였습니다. 프로세스를 디버깅중 이라면 갑자기 정지 되고 브레이크포인트도 해제 되게 됩니다. 그리고 이것에 anti-analysis 기술을 같이 사용하여 숨기면 더 효과적일 것입니다.

## 5.1 Misdirection and Stopping Execution via Exceptions

단순한 형태의 코드는 리버서 들이 쉽게 코드의 목표를 이해하고 장악하게 됩니다. 그래서 어떤 패커들은 몇몇 기술을 사용하여 단순한 형태가 되지 않게 하여 코드를 분석하는데 시간을 낭비하게 합니다.

일반적으로 언패킹을 하는 과정에서 예외를 발생시켰을 때 리버서는 EIP 가 어디를 가리키는지 이해하여야 하고 예외 핸들러 실행이 끝난 후에 EIP 가 또 어디를 가르 키는지를 알아야 합니다.

추가적으로, 예외는 패커가 반복적으로 언패킹 코드의 실행을 멈추게 하는 방법 중 하나입니다. 왜냐하면 프로세스를 디버깅할 때 예외가 발생하면 디버거는 언패킹 코드를 실행하는 것을 일시적으로 멈추게 됩니다.

패커는 일반적으로 Structured Exception Handling(SEH)를 사용하여 예외 핸들링을 쓰는데 새로운 패커들은 벡터식 예외(Vectored Exceptions) 를 사용하기 시작했습니다.

### 예제

아래의 예제코드는 오버플로우 예외(INTO 를 통해)를 발생 시킬 때 몇 개의 루프를 돌아 오버플로우 플래그는 ROL 명령어에 의해 설정된 후 예외를 발생시킵니다. 그러나 오버플로우 예외는 트랩(trap) 예외 이므로 EIP 는 JMP 명령을 가르킵니다. 리버서가 올리디버그를 사용하여 프로세스에 예외(Shift + F7/F8/F9 를 통하여)를 처리하지 않고 진행하게 되면 리버서는 영원히 루프를 돌 것입니다.

```
; set up exception handler
push .exception_handler
push dword [fs:0]
mov [fs:0],esp

; throw an exception
mov ecx,1
```

```
.loop:
rol ecx,1
into
jmp .loop

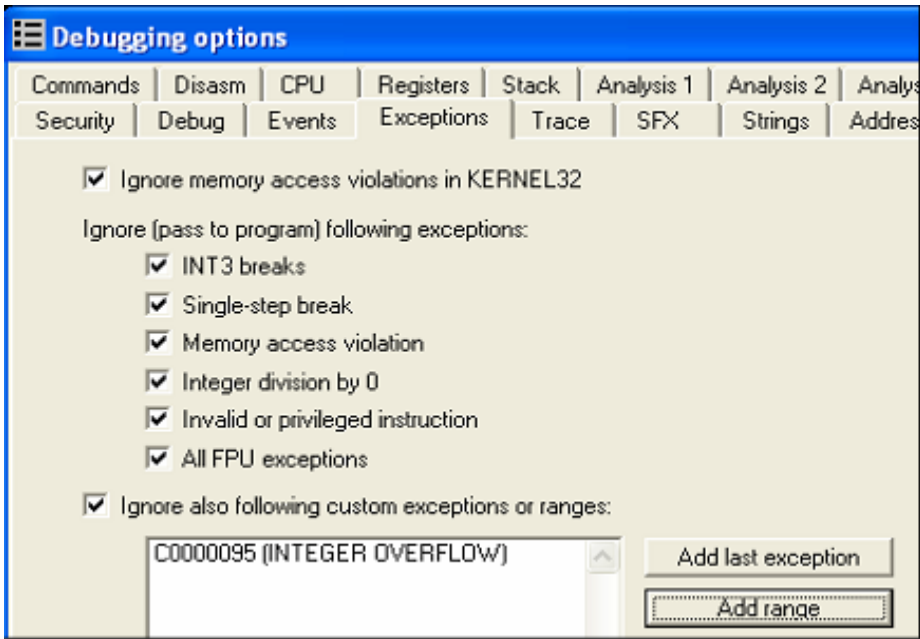
; restore exception handler
pop dword [fs:0]
add esp,4
...

.exception_handler
;EAX = CONTEXT record
mov eax,[esp+0xc]
;set Context.EIP upon return
add dword [eax+0xb8],2
xor eax,eax
retn
```

패커는 일반적으로 access violations(0xC0000005), breakpoint(0x80000003),single step(0x80000004) 예외를 일으킬수 있습니다.

### 해결방법

패커가 예외를 사용하여 단지 결과로 다른 부분의 코드실행을 하기 위해서 일 때 올리디버그는 예외 핸들러 처리를 자동으로 해줄수 있습니다. 옵션에서 -> Debugging Options -> Exceptions. 예외 핸들링을 설정할 수 있는 아래와 같은 화면을 볼 수 있습니다.



리버서는 체크박스를 선택하여서 예외 처리를 해줄 수 있습니다.

패커가 중요한 명령을 예외 핸들러에서 처리할 때 리버서는 예외 핸들러에 브레이크포인트를 걸어 주소가 올디버거에서 보이게 할 수 있습니다. 방법은 Shift+F7/F8/F9 눌러서 예외 핸들러를 전해준 뒤 View -> SEH Chain 에서 확인할 수 있습니다.

## 5.2 Blocking Input

패커는 언패킹 루틴이 실행될 때 user32!BlockInput() API 함수를 사용하여 키보드와 마우스의 입력을 막아서 리버서가 디버거를 조정하는 것을 방해할 수 있습니다. 쓰레기 코드와 안티 디스어셈블리 기술을 통하여 숨어서 리버서에게 들리지 않는다면 매우 유용합니다. 프로그램이 실행되었을 때 시스템이 반응이 없으면 리버서는 떠나게 되어 언패킹에 실패하게 됩니다.

일반적인 예제는 리버서가 GetProcAddress() 에 브레이크포인트를 걸어 언패킹을 할 때 몇 개의 쓰레기 코드들을 지나가고 패커는 BlockInput() 함수를 불러옵니다. 그리고 GetProcAddress() 에 브레이크포인트가 걸릴 때 리버서는 갑자기 디버깅 툴을 조정할 수 없게 되어 난처해 하다가 종료 할 것입니다.

### 예제

BlockInput() 은 boolean 형의 fBlockIt 파라미터를 받습니다. TRUE 이면 키보드와 마우스는 정지되고 FALSE 이면 정지가 해제됩니다.

```
; Block input
```



```

push TRUE

call [BlockInput]

; ...Unpacking code...

; Unblock input

push FALSE

call [BlockInput]

```

### 해결방법

다행히도 간단한 방법으로 BlockInput()를 패치 하여 리턴 시킬 수 있습니다. 이 올리스크립트는 user32!BlockInput()의 안에서 패치를 해줍니다.

```

gpa "BlockInput", "user32.dll"

mov [$RESULT]. #c20400# //retn 4

```

Oly Advanced 플러그인 에서도 BlockInput() 함수를 패치 하는 옵션이 있습니다.

추가적으로 CTRL+ALT+DELETE 를 눌러 수동으로 정지를 해제 할수도 있습니다.

## 5.3 ThreadHideFromDebugger

이 기술은 thread's priority 를 설정하는 ntdll!NtSetInformationThread() 함수를 사용합니다. 이 API 함수는 디버깅을 방지 할 수 있습니다.

NtSetInformationThread() 의 파라미터는 아래와 같습니다. 이 기술은 수행하려면 ThreadHideFromDebugger (0x11)을 ThreadInformationClass 의 파라미터를 사용 하여 보내야 합니다. ThreadHandle 은 일반적으로 (0xffffffff) 으로 설정하여 사용합니다:

```

NTSTATUS NTAPI NtSetInformationThread(

HANDLE ThreadHandle,

THREAD_INFORMATION_CLASS ThreadInformationClass,

```

```
PVOID ThreadInformation,
ULONG ThreadInformationLength
);
```

커널구조체의 ETHREAD 의 HideThreadFromDebugger 필드는 ThreadHideFromDebugger 에 의해 설정 될 것입니다.

설정이 된 후 내부의 커널함수 \_DbgkpSendApiMessage() 는 이벤트를 디버거에게 보내지 못합니다.

## 예제

NtSetInformationThread() 함수를 사용하는 전형적인 예제:

```
push 0 ;InformationLength
push NULL ;ThreadInformation
push ThreadHideFromDebugger ;0x11
push 0xffffffff ;GetCurrentThread()
call [NtSetInformationThread]
```

## 해결방법

ntdll!NtSetInformaionThread() 함수에 브레이크포인트를 걸 수 있습니다. 그리고 브레이크포인트가 걸리면 리버서는 EIP 를 조작하여 API 가 커널에 도달하는 것을 방지 할 수 있습니다. ollyscript 를 통하여 자동으로 할 수도 있습니다. 그리고 Olly Advanced 플러그인 에서도 선택하여 패치 할 수 있습니다. 이 API 는 ThreadInformationClass 의 파라미터가 HideThreadFromDebugger 로 설정되면 커널 코드를 불러오지 않고 단순히 리턴 해줍니다.

## 5.4 Disabling Breakpoints

디버깅들을 공격하는 또 다른 방법은 브레이크포인트를 사용하지 못하게 하는 것 입니다. 패커는 CONTEXT 구조체를 통하여 디버그 레지스터를 수정해 하드웨어 브레이크포인트를 금지시킵니다.

## 예제

이 예제에서 예외 핸들러의 CONTEXT 레코드를 전달하는 것을 통해 디버그 레지스터를 비웁니다:

```
; set up exception handler
```

```
push .exception_handler
push dword [fs:0]
mov [fs:0],esp

; throw an exception
xor eax,eax
mov dword [eax],0

; restore exception handler
pop dword [fs:0]
add esp,4
:::

.exception_handler
;EAX = CONTEXT record
mov eax,[esp+0xc]

;Clear Debug Registers: Context .Dr0-Dr3,Dr6,Dr7
mov dword [eax+0x04].0
mov dword [eax+0x08].0
mov dword [eax+0x0c].0
mov dword [eax+0x10].0
mov dword [eax+0x14].0
mov dword [eax+0x18].0

;set Context.EIP upon return
add dword [eax+0xb8],6
```

```
xor eax,eax
```

```
retn
```

소프트웨어 브레이크포인트에 대하여 패커는 직접 INT3(0xCC)를 검색할 수 있고 임의의 코드로 바꿀 수 있습니다. 그러면 브레이크포인트는 해제됩니다.

## 해결방법

정확히 하드웨어 브레이크포인트가 검출된 경우에는 소프트웨어 브레이크포인트로 대체할 수 있습니다. 만약 두 가지 모두 검출된다면 올디버거의 on-memory access/write 브레이크포인트를 사용할 수 있습니다.

## 5.5 Unhandled Exception Filter

MSDN에서는 하나의 예외로 unhandled exception filter(kernel32!UnhandledExceptionFilter)에 도달하고 프로그램이 디버깅이 되지 않았을 때 unhandled exception filter는 top level exception filter을 불러와서 Kernel32!SetUnhandledExceptionFilter() 함수 안의 파라미터를 지정한다고 나와있습니다.. 패커는 이점을 이용하여 exception filter를 설정한 뒤 예외를 발생시켜 만약 디버깅중 이라면 이 예외를 디버거가 받게 됩니다. 디버깅중이 아니라면 exception filter로 전송되어 계속 실행하게 됩니다.

## 예제

아래의 예제는 SetUnhandledExceptionFilter()을 이용하여 top level exception filter를 설정하고 access violation을 발생시킵니다. 프로세스가 디버깅중이라면 디버거는 두 개의 예외 메시지를 받게 되고 디버깅중이 아니라면 exception filter는 CONTEXT.EIP로 설정되어 계속 실행됩니다.

```
;set the exception filter
push .exception_filter
call [SetUnhandledExceptionFilter]
mov [.original_filter],eax
;throw an exception
xor eax,eax
mov dword [eax],0
```

```

;restore exception filter

push dword [.original_filter]

call [SetUnhandledExceptionFilter]

...

.exception_filter:
;EAX = ExceptionInfo.ContextRecord

mov eax,[esp+4]

mov eax,[eax+4]

;set return EIP upon return

add dword [eax+0xb8],6

;return EXCEPTION_CONTINUE_EXECUTION

mov eax,0xffffffff

retn

```

어떤 패커는 `SetUnhandledExceptionFilter()` 함수를 사용하는 대신 `kernel32!BasepCurrentTopLevelFilter` 을 직접 사용하여 exception filter 를 설정합니다. 리버서가 이 API 브레이크 포인트를 거는 것을 방지합니다.

## 해결방법

흥미롭게도 `kernel32!UnhandledExceptionFilter()` 함수의 코드 안에서 `ntdll!NtQueryInformationProcess (ProcessDebugPort)`를 사용하여 디버깅 중인지 판단합니다. 여기로부터 등록된 exception filter 를 불러올 것 인지 아닌지 결정합니다. 그러므로 DebugPort 디버거 검사 기술과 같은 방법으로 해결하면 됩니다.

## 5.6 OllyDbg:OutputDebugStrin() Format String Bug

이 디버거 공격 은 올리디버그 에서만 가능합니다. 올리디버그 는 아시다시피 포맷 스트링 버그의 취약점을 가지고 있습니다.

이 버그는 `kernel32!OutputDebugString()` 함수에 비정상적인 문자열 파라미터를 주게되면 일어납니다. 이 버그는

올리디버그(1.10)에서 아직도 존재하고 있으며 패치되지 않았습니다.

## 예제

이 간단한 예제는 올리디버그가 access violation 또는 예기치 못한 종료를 일으킬 것입니다.

```

Push .szFormatString
call [OutputDebugStringA]
...
.szFormatString db "%s%s",0
  
```

## 해결방법

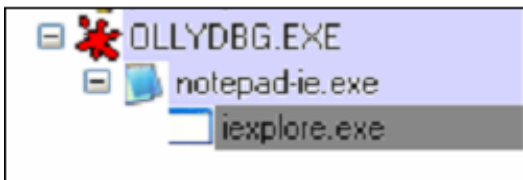
kerne32!OutputDebugStringA() 의 엔트리를 패치 해줘서 직접적으로 리턴 하여 해결할 수 있습니다.

## 6.TECHNIQUES:ADVANCED AND OTHER TECHNIQUES

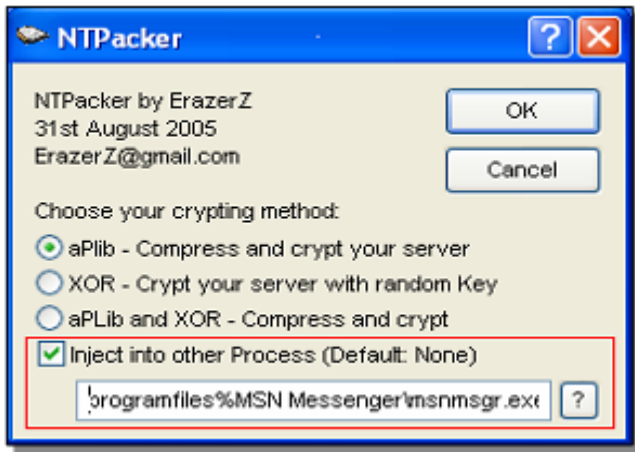
이번 장에서는 앞에서의 장에 속하지 않는 기술들과 더 고급기술들에 대해서 열거하였습니다.

### 6.1 Process Injection

프로세스 인젝션은 이미 몇몇 패커들의 특징이 되었습니다. 이 특징과 더불어 패킹을 푸는 코드는 특정한 호스트프로세스(예: 자기 자신, explorer.exe, iexplorer.exe, 기타등등)를 선택한 뒤 호스트 프로세스 안에 언패킹된 프로그램을 넣습니다.

	1	OllyDbg, 32-bit a...
	1	
	1	

아래는 프로세스 인젝션을 지원하는 패커의 스크린샷입니다.



악성코드는 패커의 이러한 특징을 이용하여 몇몇 방화벽을 우회하는데 사용합니다. 이런 방화벽은 프로세스 안에서 네트워크 연결의 응용프로그램 리스트를 허용할 수 있습니다.

패커가 사용하는 프로세스 인젝션을 하는 방법들은 다음과 같습니다.

1. kernel32!CreateProcess() 에 CREATE\_SUSPENDED 의 프로세스 생성 플래그를 넘겨주어서 호스트 프로세스는 자식 프로세스를 만드는 것을 지연시킵니다. 이때 하나의 초기화된 쓰레드가 만들어지고 멈추게 됩니다. DLL 들은 여전히 로더 루틴(ntdll!LrdInitializeThunk)이 불러오지 않아서 로드 되어있지 않습니다. 이 상황에서 쓰레드는 PEB 주소의 정보와 호스트 프로세스의 엔트리 포인트를 레지스터 값에 설정합니다.
2. kernel32!GetThreadContext() 함수를 사용하여 자식프로세스의 초기화 쓰레드를 가져옵니다.
3. CONTEXT.EBX 를 통하여 PEB 주소와 자식 프로세스를 가져옵니다.
4. PEB.ImageBase(PEB+0x8)를 읽어서 자식프로세스의 이미지 베이스를 가져옵니다.
5. 자식프로세스 안의 원래의 호스트 이미지가 ntdll!NtUnmapViewOfSection() 함수를 사용하여 unmap 할 때 BaseAddress 파라미터는 가져온 이미지 베이스를 가리킵니다.
6. 패킹을 푸는 코드가 kernel32!VirtualAllocEx() 함수를 사용하여 자식프로세스 안에 메모리를 할당할 때 dwSize 파라미터는 언패킹된 프로그램의 이미지 사이즈와 같습니다.
7. kernel32!WriteProcessMemory() 함수를 사용하여 언패킹된 프로그램의 PE 헤더와 섹션에 자식 프로세스가 써 넣습니다.

8. PEB.ImageBase 의 자식프로세스가 업데이트될 때 언패킹된 프로그램의 이미지 베이스랑 일치합니다.

9. kernel32!SetThreadContext()을 통하여 자식 프로세스의 초기화 쓰레드를 업데이트하여 이 중의 CONTEXT.EAX 를 패킹을  
 끝후의 엔트리 포인트로 설정합니다.

10. kernel32!ResumeThread()를 통하여 자식 프로세스의 실행을 재개합니다.

자식 프로세스가 생성될 때 엔트리 포인트를 디버깅하기 위해 리버서는 WriteProcessMemory() 함수에 브레이크포인트를 걸 수 있습니다. 그리고 엔트리 포인트를 담고 있는 섹션에서 자식프로세스가 쓰기를 하려고 할 때 엔트리 포인트 의 코드는 (“자기자신으로 점프”) (0xEB 0xFE)명령어로 패치 됩니다. 메인 쓰레드의 자식프로세스가 재개될 때 자식프로세스는 엔트리 포인트 A 에서 끝없는 반복 구간으로 들어갑니다. 명령어를 수정하여 복구하게 되면 계속해서 디버깅을 할 수 있습니다.

## 6.2 Debugger Blocker

Armadillo 패커는 Debugger Blocker 를 불러오는데 이것에 대해 소개하겠습니다. 이것은 리버서가 보호된 프로세스를 어태치 하는 것을 막아줍니다. 이 보호는 윈도우즈가 제공하는 디버깅 함수를 사용하여 수행합니다.

정확히 말하면 패킹을 푸는 코드는 하나의 디버거(부모 프로세스) 역할을 하고 이것을 통해 언팩된 프로그램을 담고 있는 것을 자식프로세스를 생성하여 디버그/컨트롤을 합니다.

explorer.exe	1680	11	Windows Explorer	Microsoft Corpor
VMwareUser.exe	1768	2	VMwareUser	VMware, Inc.
proccxp.exe	1336	2	6.06 Sysinternals Pro...	Sysinternals
OLLYDBG.EXE	896	1	OllyDbg, 32-bit a...	
videodrv.exe	1752	3		
videodrv.exe	1800	2		

보호받고 있는 프로세스는 이미 디버깅중이기 때문에 kernel32!DebugActiveProcess()를 통해 디버거로 어태치 하려고 하면 실패할것입니다. 이미 native API 인 ntdll!NtDebugActiveProcess() 가 STATUS\_PORT\_ALREADY\_SET 을 리턴 하기 때문입니다. NtDebugActiveProcess() 가 실패한 원인은 커널 구조체인 EPROCESS 의 DebugPort 필드가 이미 설정되었기 때문입니다.



보호받고 있는 프로세스를 디버깅 하기 위해서 몇몇의 리버싱 포럼에서 발표한 해결방법은 부모 프로세스의 컨텍스트안에서 Kernel32!DebugActiveProcessStop()을 이용하는 것입니다.

어태칭 할 디버거의 부모프로세스에서 kernel32!WaitForDebugEvent() 내부에 브레이크포인트를 걸은 뒤 실행하여 브레이크포인트가 걸릴 때 DebugActiveProcessStop(ChildProcessPID)를 불러내는 코드를 넣어서 실행하면 디버거는 보호받고 있는 프로세스를 어태칭 할 수 있습니다.

### 6.3 TLS Callbacks

또 다른 기술은 패커가 실제 엔트리 포인트를 실행 하기전에 코드를 실행 하는 것 입니다. 이것은 Thread Local Storage(TLS) 콜백 함수를 사용하여 이루어 집니다. 패커가 디버거 체크 나 복호화 루틴을 콜백 함수 안에 넣는 다면 리버서는 이 루틴들을 트레이싱 할 수 없을 것 입니다.

TLS 콜백은 pedump 와 같은 PE 파일 분석툴로 식별 할 수 있습니다. 실행파일 안에 TLS 디렉토리가 존재한다면 Data 디렉토리 안에서 볼 수 있습니다.

```
Data Directory
EXPORT rva: 00000000 size: 00000000
IMPORT rva: 00061000 size: 000000E0
...
TLS rva: 000610E0 size: 00000018
...
IAT rva: 00000000 size: 00000000
DELAY_IMPORT rva: 00000000 size: 00000000
COM_DESCRIPTOR rva: 00000000 size: 00000000
unused rva: 00000000 size: 00000000
```

TLS 디렉토리의 실제 내용들을 보여줍니다. AddressOfCallBacks 필드 포인트들은 null 로 종료되는 것 과 콜백 함수들의 배열을 가리킵니다:

```
TLS directory:
StartAddressOfRawData: 00000000
EndAddressOfRawData: 00000000
```

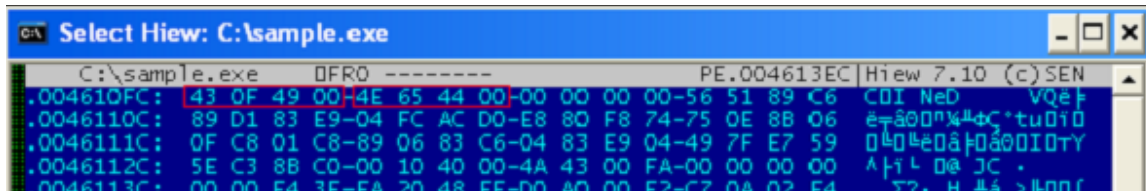
AddressOfIndex: 004610F8

**AddressOfCallBacks: 004610FC**

SizeOfZeroFill: 00000000

Characteristics: 00000000

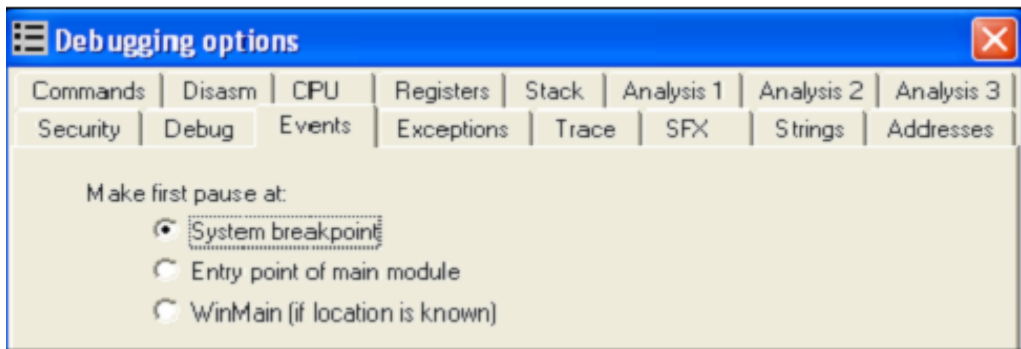
이 예제에서 RVA 0x4610fc 는 콜백 함수 포인터를 가리킵니다.(0x490f43 , 0x44654e):



기본적으로, 올디버그로 예제를 불러온다면 엔트리 포인트에서 멈출 것 입니다. 실제 엔트리 포인트에 도착 하기전에 TLS 콜백 함수들은 이미 실행이 됩니다. 올디버그에서 TLS 콜백을 불러 오기전에 중단된 실제 loader 에서 정지하도록 설정하여야 합니다.

Option -> Debugging Option -> Events -> Make First pause at -> System breakpoint.

이러한 설정을 통하여 ntdll.dll 안에서 중단된 실제 loader 코드에서 멈춥니다.



이렇게 설정 한 후에 올디버그는 TLS 콜백의 ntdll!\_LdrpRunInitializeRoutines() 전의 ntdll!\_LdrpInitializeProcess() 로 실행된 자리에서 중단됩니다. 이때 콜백 함수 에 브레이크포인트를 걸어 트레이싱 할 수 있습니다.

PE 파일 포맷의 더 많은 정보 와 pedump 의 실행파일/소스 는 아래의 링크에서 구하실 수 있습니다:

An In-Depth Look into the Win32 Portable Executable File Format by Matt Pietrek

<http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx>

An In-Depth Look into the Win32 Portable Executable File Format,Part 2 by Matt Pietrek

<http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/>

최신버전의 PE 파일 포맷은 아래 연결을 통해 얻을 수 있습니다:

microsoft Portable Executable and Common Object File Format Specification

<http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.msp>

## 6.4 Stolen Bytes

Stolen Bytes는 패커가 이동시킨 보호된 프로그램의 코드의 일부분(보통은 엔트리 포인트의 몇 개의 명령어)입니다. 이 부분의 명령어는 이동된 곳이나 할당 받은 메모리 공간에서 실행됩니다. 보호된 프로세스의 메모리가 덤프 되었을 때 Stolen Bytes를 복구하지 못한다면 덤프된 실행파일은 작동하지 않습니다.

여기 이 예제는 실행될 수 있는 파일의 오리지널 엔트리 포인트(OEP)입니다:

```
004011CB MOV EAX,DWORD PTR FS:[0]
004011D1 PUSH EBP
004011D2 MOV EBP,ESP
004011D4 PUSH -1
004011D6 PUSH 0047401C
004011DB PUSH 0040109A
004011E0 PUSH EAX
004011E1 MOV EWORD PTR FS:[0],ESX
004011E8 SUB ESP,10
004011EB PUSH EBX
004011EC PUSH ESI
004011ED PUSH EDI
```

그리고 아래는 Enigma Protector 패커로 패키징하여 앞부분의 두 개의 명령어를 훔쳐낸 동일한 코드입니다:

```
004011CB POP EBX
004011CC CMP EBX,EBX
```

```
004011CE DEC ESP
004011CF POP ES
004011D0 JECXZ SHORT 00401169
004011D2 MOV EBP,ESP
004011D4 PUSH -1
004011D6 PUSH 0047401C
004011DB PUSH 0040109A
004011E0 PUSH EAX
004011E1 MOV DWORD PTR FS:[0],ESP
004011E8 SUB ESP,10
004011EB PUSH EBX
004011EC PUSH ESI
004011ED PUSH EDI
```

이 것은 ASProtect 패커로 몇 개의 명령어를 훔친 동일한 예제입니다. jump 명령어를 추가하여 훔쳐진 명령어의 실행 루틴을 가르킵니다. 훔쳐진 명령어와 쓰레기 코드가 함께 있으면 훔쳐진 명령어를 복구하기가 매우 어렵습니다.

```
004011CB JMP 00B70361
004011D0 JNO SHORT 00401198
004011D3 INC EBX
004011D4 ADC AL,0B3
004011D6 JL SHORT 00401196
004011D8 INT1
004011D9 LAHF
004011DA PUSHFD
004011DB MOV EBX,1D0F0294
004011E0 PUSH ES
004011E1 MOV EBX,A732F973
004011E6 ADC BYTE PTR DS:[EDX-E],CH
```

```
004011E9 MOV ECX,EBP
004011EB DAS
004011EC DAA
004011ED AND DWORD PTR DS:[EBX+58BA76D7],ECX
```

## 6.5 API Redirection

API 리다이렉션은 리버서들이 보호된 프로그램의 임포트 테이블을 쉽게 리빌딩 하는 것을 방지 하기 위한 기술입니다.

오리지널 임포트 테이블은 파괴되고 할당된 메모리 안에 불러온 리다이렉트되는 API 들의 루틴들이 존재합니다. 이 루틴들은 API 를 불러올 때 사용됩니다.

이 예제에서는 kernel32!CopyFileA() 함수를 불러옵니다:

```
00404F05 LEA EDI,DWORD PTR SS:[EBP-20C]
00404F0B PUSH EDI
00404F0C PUSH DWORD PTR SS:[EBP-210]
00404F12 CALL <JMP.&KERNEL32.CopyFileA>
```

call 대신 JMP 를 사용하며 점프하는곳의 주소는 임포트 테이블을 참조합니다.

```
004056B8 JMP DWORD PTR DS:[<&KERNEL32.CopyFileA>]
```

그러나 ASProtect 패커가 리다이렉트한 kernel32!CopyFileA() 함수를 불러올 때 할당된 메모리에서 CALL 루틴으로 수정 되어 kernel32!CopyFileA() 의 훅쳐진 명령은 결국 가장중요한 실행이 됩니다.

```
004056B8 CALL 00D90000
```

아래 설명은 훅쳐진 명령어가 어떻게 사용되는지 설명합니다. 처음 7 개의 명령어들은 kernel!CopyFileA() 코드에서 복사된 것입니다. 그리고 코드안의 0x7C83005E 을 콜 하는 부분 또 한 복사되었습니다. RETN 명령어를 통하여 kernel32.dll 안의 kernel32!CopyFileA() 루틴 중간인 0x7C830063 으로 갑니다:

Stolen instructions from kernel32!CopyFileA

```

00D80003 MOV EDI,EDI
00D80005 PUSH EBP
00D80006 MOV EBP,ESP
00D80008 PUSH ECX
00D80009 PUSH ECX
00D8000A PUSH ESI
00D8000B PUSH DWORD PTR SS:[EBP+8]
00D8000E JMP SHORT 00D80013
00D80011 INT 20
00D80013 PUSH 7C830063 ;return EIP
00D80018 MOV EDI,EDI
00D8001A PUSH EBP
00D8001B MOV EBP,ESP
00D8001D PUSH ECX
00D8001E PUSH ECX
00D8001F PUSH ESI
00D80020 MOV EAX,DWORD PTR FS:[18]
00D80026 PUSH DWORD PTR SS:[EBP+8]
00D80029 LEA ESI,DWORD PTR DS:[EAX+BF8]
00D8002F LEA EAX,DWORD PTR SS:[EBP-8]
00D80032 PUSH EAX
00D80033 PUSH 7C80E2BF
00D80038 RETN
    
```

Actual kernel32!CopyFileA code

```

7C830053 MOV EDI,EDI
7C830055 PUSH EBP
7C830056 MOV EBP,ESP
7C830058 PUSH ECX
7C830059 PUSH ECX
7C83005A PUSH ESI
7C83005B PUSH DWORD PTR SS:[EBP+8]
7C83005E CALL kernel32.7C80E2A4
7C830063 MOV ESI,EAX
7C830065 TEST ESI,ESI
7C830067 JE SHORT kernel32.7C8300A6
    
```

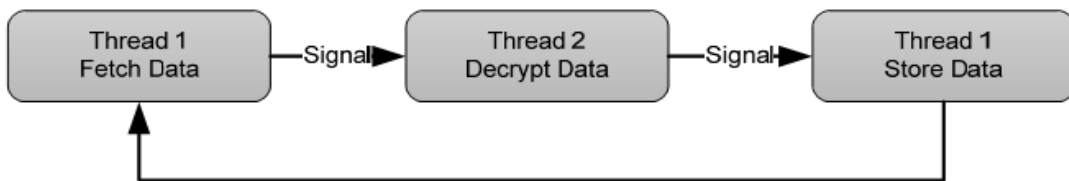
어떤 패커는 더 먼 곳의 전체 DLL 이미지를 할당된 메모리에 불러옵니다. 그리고 리다이렉트된 API 함수를 불러올 때 안에서 DLL 이미지를 복사합니다. 이 기술은 실제 API 함수에 브레이크포인트를 거는 것을 어렵게 만드는 효과가 있습니다.

### 6.6 Multi-Threaded Packers

멀티스래드 패커에서 또다른 스래드는 보호된 프로그램을 복호화 하는데 필요한 몇몇 명령어를 생성하는데 사용합니다. 멀티스래드 패커의 복잡도는 증가하여 코드가 복잡해져서 코드를 이해하기도 매우 어려워졌습니다.

멀티스래드 패커의 한가지 예로는 PECrypt 가 있습니다. 이것의 두번째 스래드는 복호화 하는데 사용하고 메인 스래드는 이 데이터를 사용합니다. 이런 스래드는 실시간으로 진행됩니다.

PECrypt 에서 스래드를 사용한 명령어 처리:

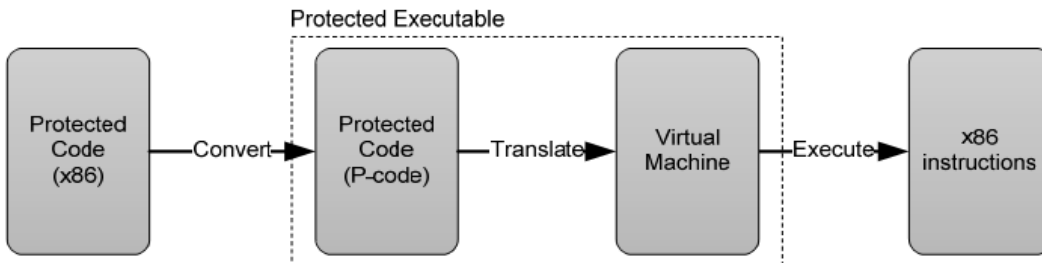


### 6.7 Virtual Machines

가상 머신을 사용한 이 개념은 아주 간단합니다. 리버서는 최종적으로 어떻게 우회/안티디버깅, 안티리버싱의 기술을 해결할 것인가를 생각합니다. 보호된 프로그램은 메모리안에서 복호화 및 실행될 때 정적분석을 하면 취약합니다.

가상 머신의 출현으로 보호된 부분의 코드가 p-code 로 변환되어 p-code 가 기계어로 변환되어서 실행됩니다. 원래의 기계 명령어는 치환되고 코드 해석의 복잡도도 상승합니다.

아래는 이 개념의 간단한 그림입니다:



Oreans technologies 의 CodeVirtualizer 와 StarForce 와 같은 최신 패커도 가상 머신의 개념을 응용하여 프로그램을 보호합니다.

가상 머신에 대응하려면 분석이 필요합니다. p-code 가 가상 머신에 의해 변환된 구조라면 충분한 정보를 얻은 후 p-code 를 분석하여 이것을 기계어 또는 이해할 수 있는 명령어로 변환하는 디스어셈블러를 개발합니다.

p-code 디스어셈블러를 개발하는 예제와 가상머신이 실행된 자세한 정보는 아래의 링크를 통하여 얻을수 있습니다:

Defeating HyperUnpackMe2 With an IDA Processor Module, Rolf Rolles III

[http://www.openrce.org/articles/full\\_view/28](http://www.openrce.org/articles/full_view/28)

## 7. TOOLS

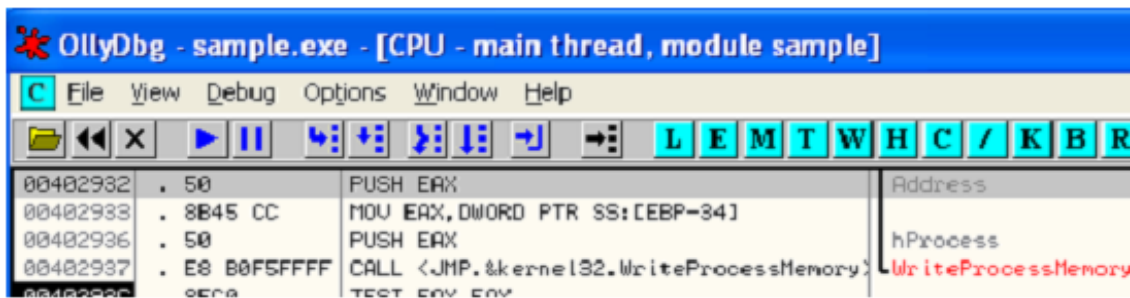
이번 장에서는 리버서나 악성코드 분석원들이 패커를 분석하거나 언패킹을 할 때 공개적으로 쓰는 툴에 대해 열거하였습니다.

필독: 이것은 모두 제 3 자의 툴입니다; 필자는 이 툴 사용으로 인한 시스템의 불안정이나 영향을 주는 문제에 대해서는 책임을 지지 않습니다. 테스트 또는 악성코드를 분석하는 환경에만 이 툴을 사용할 것을 건의 합니다.

### 7.1 OllyDbg

<http://www.ollydbg.de/>

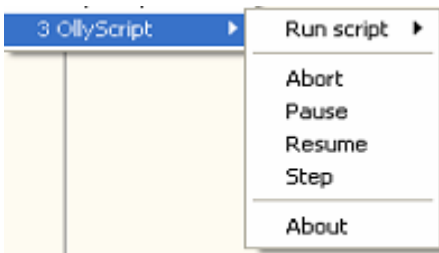
리버서와 악성코드 분석자들이 사용하는 강력한 ring 3 모드 디버거입니다. 이 툴의 플러그인을 리버서들이 만들어 내는 것이 가능하며 이것을 이용하면 리버싱과 언패킹을 푸는것이 더욱 쉽습니다.



## 7.2 Ollyscript

<http://www.openrce.org/downloads/details/106/OllyScript>

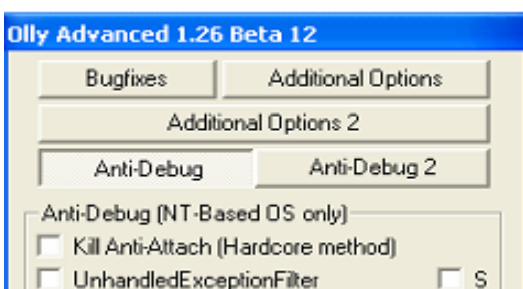
올리디버그 스크립트는 어셈블리 언어와 같은 스크립트를 통하여 설정/브레이크포인트 핸들링, 코드/데이터 패치 등을 할 수 있습니다. 반복작업이나 자동으로 언패킹을 하는데 가장 유용합니다.



## 7.3 Olly Advanced

[http://www.openrce.org/downloads/details/241/Olly\\_Advanced](http://www.openrce.org/downloads/details/241/Olly_Advanced)

리버서를 대상으로 코드가 만약 패커라는 갑옷이 있다면 리버서에게는 이 플러그인이 디버깅툴의 갑옷입니다. 이것은 여러 가지 옵션으로 안티디버깅 기술을 우회하고 올리디버그를 패커로 부터 숨겨줍니다. 그리고 가장 유용합니다.

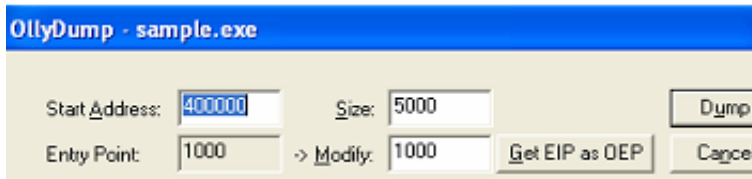




## 7.4 OllyDump

<http://www.openrce.org/downloads/details/108/OllyDump>

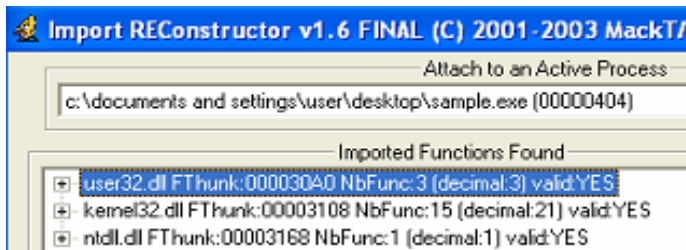
성공적으로 언팩을 한 후에 이 플러그인으로 프로세스를 덤프 한 뒤 임포트 테이블을 리빌딩 합니다.



## 7.5 ImpRec

<http://www.woodmann.com/crackz/Unpackers/Imprec16.zip>

마지막으로 이것은 프로세스를 덤프 하거나 임포트 테이블을 복구하는 툴입니다. 이것은 가장 훌륭한 임포트 테이블 복구 툴입니다.



## 8. REFERENCES

서적 : Reverse Engineering, Software Protection

□ Reversing: Secrets of Reverse Engineering. E.Eilam.Wiley, 2005

□ Crackproof Your Software , P.Cerven.No Starch Press, 2002

서적 : Windows and Processor Internal

□ Microsoft Windows Internal, 4<sup>th</sup> Edition . M. Russinovich, D. Solomon, Microsoft Press,

□ IA-32 Intel Architecture Software Developer's Manual. Volume 1-3, Intel Corporation, 2006

링크: Windows Internals

□ ReactOS Project

<http://www.reactos.org/en/index.html>

Source Search: <http://www.reactos.org/generated/doxygen/>

□ Wine Project

<http://www.winehq.org/>

Source Search: <http://source.winehq.org/source/>

□ The Undocumented Functions

<http://undocumented.ntinternals.net>

□ MSDN

<http://msdn2.microsoft.com/en-us/default.aspx>

링크: Reverse Engineering, Software Protection, Unpacking

□ OpenRCE

<http://www.openrce.org>

□ OpenRCE Anti Reverse Engineering Techniques Database

[http://www.openrce.org/reference\\_library/anti\\_reversing](http://www.openrce.org/reference_library/anti_reversing)

□ RCE Forums

<http://www.woodmann.com/forum/index.php>

□ EXETOOLS Forums

<http://forum.exetools.com>