Technique of Anti-Debugging

Diveax, May 10, 2014

http://diveax.tistory.com

목차

기술 설명은 아래와 같은 순서로 진행하겠습니다.

1. API 를 이용한 안티 디버깅
2. PEB 접근을 이용한 안티 디버깅
3. 예외를 이용한 안티 디버깅
4. 시간 차를 이용한 안티 디버깅
5. 하드웨어 & 레지스터를 이용한 안티 디버깅
6. 기타 기술들...

- 1. API 를 이용한 안티 디버깅

A. IsDebuggerPresent()

처음으로 볼 API 를 이용한 안디 디버깅 기법은 IsDebuggerPresent 라는 API 함수이다. 이 함수는 PEB 구조체에 있는 IsDebugged 값을 참조해서 현재 이 프로세스가 디버깅 중인지를 판단합니다. 만약 리턴값이 0 이라면 정상적으로 실행 중인 상태고 0 이 아니면 디버깅 중이라는 뜻이다.

```
#include <windows.h>
void main(){
    if (IsDebuggerPresent()){
        MessageBox(0, L"IsDebuggerPresent", L"Detected!", MB_OK);
        ExitProcess(1);
    }
}
```

B. CheckRemoteDebuggerPresent()

이번에 볼 함수는 CheckRemoteDebuggerPresent 라는 이름 참 긴 함수입니다. 이 함수도 IsDebuggerPresent 함수와 똑같이 PEB 구조체에 있는 값을 참조합니다. 대신 참조하는 값이 다르겠죠. BeingDebugged 라는 값을 참조합니다. 함수 사용법은 1번째 인자에 타켓 프로세스의 핸들을 넣어주고, 2번째 인자에는 리턴값이 저장 될 곳을 넣어주면 됩니다. 추가로 이 함수는 Windows XP sp1 이상에서 사용 가능합니다.

```
#include <Windows.h>
main(){
    BOOL IsDebugged = FALSE;
    CheckRemoteDebuggerPresent(GetCurrentProcess(), &IsDebugged);

    if (IsDebugged){
        MessageBox(0, L"CheckRemoteDebuggerPresnet", L"Detected!", MB_OK);
    }
}
```

C. FindWindow()

이번에 볼 함수는 FindWindow 함순데, 처음엔 전혀 이 함수가 안티 디버깅하고 관련이 있는지? 하고 의문이 들 겁니다. 이 함수를 열려있는 윈도우 창의 타이틀을 체크하는데 사용함으로 예를 들어 Ollydbg 같은 문자열이 있는지 확인 해서 디버거가 열려 있구나 라고 체크가 가능합니다.

```
#include <Windows.h>
void main(){
    HWND ck = FindWindow(L"String", 0);
    if (ck != NULL) {
        MessageBox(0, L"FindWindow", L"Detected!", MB_OK);
    }
}
```

D. OutputDebugString()

함수 기능은 디버거의 메세지를 출력하는 역할을 하네요. 자세한 처리 단계를 보면, RaiseException() 의 매개변수인 dwexceptioncode 의 값으로 DBG_PRINTEXCEPTION_C 를 전달하고 만약 이걸 디버거에서 처리를 하면 최종적으로 메세지를 출력하게 되고, 디버깅 중이 아니면 아무런 작업을 하지 않습니다. 이때 TEB 구조체에 있는 LastError 값이 다른걸 이용해서 디버깅 여부 판단이 가능합니다. 추가로 이 함수는 Windows 2000 이상에서 사용 가능합니다.

```
#include <Windows.h>
void main(){
    DWORD Val = 666;
    SetLastError(Val);
    OutputDebugString(L"anything");
    if (GetLastError() == Val){
        MessageBox(NULL, L"OutputDebugString", L"Detected!", MB_OK);
    }
}
```

E. NtQueryInformationProcess()

이제 Nt 계열 함수 네요!. 이 함순 ntdll.dll 파일에 위치 해 있고, 타켓 프로세스로 부터 정보를 받아옵니다. 당연 디버깅 관련 정보도 받아오겠죠. 함수 원형은 다음과 같습니다.

Syntax

```
NTSTATUS WINAPI NtQueryInformationProcess(
  _In_          HANDLE ProcessHandle,
```

```
  _In_      PROCESSINFOCLASS ProcessInformationClass,
  _Out_      PVOID ProcessInformation,
  _In_      ULONG ProcessInformationLength,
  _Out_opt_  PULONG ReturnLength
);
```

ProcessInformationClass 값이 ProcessDebugPort(0x7), ProcessDebugFlags(0x1f) , ProcessObjectHandle(0x1e) 인지를 이용해서 디버그 탐지가 가능합니다.

1. ProcessDebugPort

```
#include <windows.h>
typedef NTSTATUS(WINAPI *pfnNtQueryInformationProcess)(HANDLE, UINT, PVOID, ULONG, PULONG);
pfnNtQueryInformationProcess NtQueryInfoProcess = NULL;
void main(){
    unsigned long Ret, IsRemotePresent = 0;
    HMODULE hNtDll = LoadLibrary(TEXT("ntdll.dll"));

    NtQueryInfoProcess = (pfnNtQueryInformationProcess)
        GetProcAddress(hNtDll, "NtQueryInformationProcess");
    Ret = NtQueryInfoProcess(GetCurrentProcess(), 0x7, &IsRemotePresent, 4, NULL);

    if (Ret == 0x0 && IsRemotePresent != 0){
        MessageBox(0, L"NtQueryInformationProcess", L"Detected!", MB_OK);
        ExitProcess(0);
    }
}
```

만약에 디버거가 켜져 있다면 리턴값으로 디버거가 작동하는 포트값을 반환 할 것이다. 다시 말하면 만약 0 값을 갖지 않으면 디버거가 작동하고 있다는 뜻이다.

2. ProcessDebugFlags

```
#include <Windows.h>
typedef NTSTATUS(WINAPI *pNtQueryInformationProcess)(HANDLE, UINT, PVOID, ULONG, PULONG);
void main(){
    unsigned long DebugFlag;

    pNtQueryInformationProcess NtQIP = (pNtQueryInformationProcess)
        GetProcAddress(GetModuleHandle(TEXT("ntdll.dll")),
        "NtQueryInformationProcess");

    NtQIP(GetCurrentProcess(),  0x1f, &DebugFlag, 4, NULL);

    if (DebugFlag == 0x0){
        MessageBox(0, L"NtQueryInformationProcess - DebugFlags", L"Detected!", MB_OK);
        ExitProcess(0);
    }
}
```

만약 타켓 스레드에 디버가 존재하면 DebugFlags 값이 0 이란 값을 가질 겁니다..

3. ProcessObjectHandle

이 값도 위 방법과 비슷하게 사용합니다. Windows XP 부터, 만약 프로세스가 디버깅 당하고 있ㄴ

으면 Debug Object 에 새로운 핸들을 만듭니다. 즉, 이를 이용해서 탐지가 가능합니다.

```
#include <Windows.h>
typedef NTSTATUS(WINAPI *pNtQueryInformationProcess)(HANDLE, UINT, PVOID, ULONG, PULONG);
void main(){
    HANDLE DebugObject = NULL;

    pNtQueryInformationProcess NtQIP = (pNtQueryInformationProcess)
        GetProcAddress(GetModuleHandle(TEXT("ntdll.dll")),
        "NtQueryInformationProcess");

    NtQIP(GetCurrentProcess(),  0x1e, &DebugObject, 4, NULL);

    if (DebugObject){
        MessageBox(0, L"NtQueryInformationProcess - DebugObject", L"Detected!", MB_OK);
        ExitProcess(0);
    }
}
```

F. NtQueryObject

이 함수는 오브젝트 정보 리스트를 받는 역할을 하고, 리스트를 받아서 디버그 오브젝트가 있는지 확인하는 방법으로 디버깅을 탐지합니다.

```
#include <Windows.h>
#include <TlHelp32.h>
#include <tchar.h>
typedef struct _LSA_UNICODE_STRING { ... } LSA_UNICODE_STRING, *PLSA_UNICODE_STRING, UNICODE_STRING, *PUNICODE_STRING;
typedef struct _OBJECT_TYPE_INFORMATION { ... } OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;
typedef struct _OBJECT_ALL_INFORMATION { ... } OBJECT_ALL_INFORMATION, *POBJECT_ALL_INFORMATION;
void main(){
    typedef NTSTATUS(NTAPI *pNtQueryObject)(HANDLE, UINT, PVOID, ULONG, PULONG);
    POBJECT_ALL_INFORMATION pObjectAllInfo = NULL;
    void *pMemory = NULL;
    NTSTATUS Status;
    unsigned long Size = 0;
    pNtQueryObject NtQO = (pNtQueryObject) GetProcAddress(GetModuleHandle(TEXT("ntdll.dll")), "NtQueryObject");

    Status = NtQO(NULL, 3, &Size, 4, &Size);
    pMemory = VirtualAlloc(NULL, Size, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    Status = NtQO((HANDLE) -1, 3, pMemory, Size, NULL);
    pObjectAllInfo = (POBJECT_ALL_INFORMATION) pMemory;
    unsigned char *pObjInfoLocation = (unsigned char*) pObjectAllInfo->ObjectTypeInformation;
    ULONG NumObjects = pObjectAllInfo->NumberOfObjects;

    for (UINT i = 0; i < NumObjects; i++){
        POBJECT_TYPE_INFORMATION pObjectTypeInfo = (POBJECT_TYPE_INFORMATION) pObjInfoLocation;

        if (wcscmp(L"DebugObject", pObjectTypeInfo->TypeName.Buffer) == 0){
            if (pObjectTypeInfo->TotalNumberOfObjects > 0){
                VirtualFree(pMemory, 0, MEM_RELEASE);
                MessageBox(0, L"NtQueryObject", L"Detected!", MB_OK);
                ExitProcess(0);
            }
            else{
                VirtualFree(pMemory, 0, MEM_RELEASE);
            }
        }

        pObjInfoLocation = (unsigned char*) pObjectTypeInfo->TypeName.Buffer;
        pObjInfoLocation += pObjectTypeInfo->TypeName.Length;
        ULONG tmp = ((ULONG) pObjInfoLocation) & -4;
        pObjInfoLocation = ((unsigned char*) tmp) + sizeof(unsigned long);
    }
    VirtualFree(pMemory, 0, MEM_RELEASE);
}
```

Syntax

```
NTSTATUS NtQueryObject(
  _In_opt_    HANDLE Handle,
  _In_        OBJECT_INFORMATION_CLASS ObjectInformationClass,
  _Out_opt_   PVOID ObjectInformation,
  _In_        ULONG ObjectInformationLength,
  _Out_opt_   PULONG ReturnLength
);
```

간단히 코드 설명을 하면, NtQO 함수로 실제적인 오브젝트 리스트를 받아오고, 리스트에서 DebugObject 라는 목록이 보이면 디버깅 중이라 판단을 한다.

G. NtSetDebugFilterState

이 함수는 디버그 필터 상태를 변경하는 함수입니다. 디버깅 시에는 필터값 변경이 불가능하다는 걸 이용해서 리턴값 체크를 통해 디버깅 여부를 판단합니다. 만약 정상 실행중이면 0 을 리턴합니다. 추가로 비정상이면 0xC00000022 예외가 뜹니다.
또 이 방법으로는 프로세스가 Attach 된 경우에만 탐지를 합니다. 예를들어 디버거에 그래그를 해서 올리는 경우에는 탐지를 못하는 경우가 있습니다.

Syntax

```
NTSTATUS WINAPI NtSetDebugFilterState(ULONG ComponentId, unsigned int Level, char State)
```

```
#include <windows.h>
#include <tchar.h>
typedef NTSTATUS(WINAPI* PNTSETDEBUGFILTERSTATE)(ULONG, ULONG, BOOLEAN);
main(){
    FARPROC pFunc;
    NTSTATUS state;

    pFunc = GetProcAddress(GetModuleHandle(_T("ntdll.dll")), "NtSetDebugFilterState");
    state = ((PNTSETDEBUGFILTERSTATE)pFunc)(0, 0, TRUE);
    if (state != 0){
        MessageBox(0, L"NtSetDebugFilterState", L"Detected!", MB_OK);
    }
}
```

H. NtQuerySystemInformation

NtQuerySystemInformation 함수를 사용해서 Debug 모드로 부팅이 되었는지를 체크하는 함수입니다. 아래는 함수 원형입니다.

Syntax

```
NTSTATUS WINAPI NtQuerySystemInformation(
  _In_        SYSTEM_INFORMATION_CLASS SystemInformationClass,
  _Inout_     PVOID SystemInformation,
  _In_        ULONG SystemInformationLength,
  _Out_opt_   PULONG ReturnLength
);
```

```c
#include <Windows.h>
#include <tchar.h>
typedef NTSTATUS(WINAPI *NTQUERYSYSTEMINFORMATION)
(ULONG SystemInformationClass, PVOID SystemInformation,
ULONG SystemInformationLength,
PULONG ReturnLength);
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION { ... } SYSTEM_KERNEL_DEBUGGER_INFORMATION,
*PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
void main(){
    SYSTEM_KERNEL_DEBUGGER_INFORMATION DebuggerInfo = { 0, };
    NTQUERYSYSTEMINFORMATION NtQuerySystemInformation;
    ULONG SystemKernelDebuggerInformation = 0x23;
    ULONG ulReturnedLength = 0;

    NtQuerySystemInformation = (NTQUERYSYSTEMINFORMATION)
        GetProcAddress(GetModuleHandle(L"ntdll"),
        "NtQuerySystemInformation");

    NtQuerySystemInformation(SystemKernelDebuggerInformation,
        (PVOID) &DebuggerInfo,
        sizeof(DebuggerInfo),      // 2 bytes
        &ulReturnedLength);

    if (DebuggerInfo.DebuggerEnabled){
        MessageBox(0, L"NtQuerySystemInformationProcess", L"Detected!", MB_OK);
        ExitProcess(0);
    }
}
```

I. NtSetInformationThread

정상적으로 실행했을 때는 Thread 에 대한 이벤트를 계속 받지만 디버깅 시에는 Thread 에 대한 이벤트를 받을 수 없음을 이용한 안티 디버깅 기법입니다. 2 번째 인자 값을 0x11(ThreadHideFromDebugger) 로 설정해 줌으로 디버거를 Detach 할 수 있습니다.

Syntax

```
NtSetInformationThread(
  IN HANDLE          ThreadHandle,
  IN THREAD_INFORMATION_CLASS ThreadInformationClass,
  IN PVOID           ThreadInformation,
  IN ULONG           ThreadInformationLength
 );
```

```c
#include <Windows.h>
typedef enum _THREAD_INFOMATION_CLASS { ... } THREAD_INFORMATION_CLASS, *PTHREAD_INFORMATION_CLASS;
typedef NTSTATUS(*PNTSETINFORMATIONTHREAD)(
    IN HANDLE ThreadHandle,
    IN THREAD_INFORMATION_CLASS ThreadInformationClass,
    IN PVOID ThreadInformation,
    IN ULONG ThreadInformationLength
    );
void main(){
    PNTSETINFORMATIONTHREAD NtSetInformationThread;

    NtSetInformationThread = (PNTSETINFORMATIONTHREAD)GetProcAddress(GetModuleHandle(L"ntdll.dll"),
        "NtSetInformationThread");
    NtSetInformationThread(GetCurrentThread(), 0x11, NULL, (ULONG)NULL);
}
```

J. Self-Debugging

간단히 원리를 설명하면, 자신의 프로세스를 디버깅 하게 되면, 원칙적으로는 다른 디버거에서 이 프로세스를 Attach 하지 못합니다. 이를 이용해서 디버깅을 막을 수 있습니다.

```
#include <Windows.h>
void main(){
    HANDLE hProcess = NULL;
    PROCESS_INFORMATION Pi;
    DEBUG_EVENT De;
    STARTUPINFO S;

    ZeroMemory(&Pi, sizeof(PROCESS_INFORMATION));
    ZeroMemory(&S, sizeof(STARTUPINFO));
    ZeroMemory(&De, sizeof(DEBUG_EVENT));
    GetStartupInfo(&S);
    CreateProcess(NULL, GetCommandLine(), NULL, NULL, FALSE, DEBUG_PROCESS, NULL, NULL, &S, &Pi);

    ContinueDebugEvent(Pi.dwProcessId, Pi.dwThreadId, DBG_CONTINUE);
    WaitForDebugEvent(&De, INFINITE);
}
```

K. Csrss with OpenProcess

프로세스를 디버깅 하려면 먼저 디버거에 Attach 를 해야 합니다. 그리고 몇 개의 디버거들은 타켓 프로세스의 권한을 원래 권한으로 돌아가는데 성공하지만 일부는 실패합니다. CSRSS 를 사용하면 특정 권한이 높아졌는지 원래상태인지를 판별이 가능 합니다. 즉, 이를 이용해서 탐지가 가능합니다. (Process_All_Access 플래그 사용)

하지만 일반적으로 현재 프로세스에서 csrss 프로세스를 PAA 권한으로 OpenProcess() 함수로 열 수가 없습니다. 그러나, 디버거에 프로세스가 Attach 되면 권한이 상승되기 때문에 이는 문제가 되지않죠. 이 때 OpenProcess() 함수로 csrss 프로세스를 열 수 있습니다.

```
#include <Windows.h>
#include <tchar.h>
void main(){
    FARPROC pFunc;
    HANDLE hProcess;

    pFunc = GetProcAddress(GetModuleHandle(_T("ntdll.dll")),
        "CsrGetProcessId");

    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pFunc());

    if (hProcess != NULL){
        MessageBox(0, L"csrss", L"Detected!", MB_OK);
    }
}
```

L. DeleteFiber

Fiber 를 삭제하는 명령. 만약 함수 호출 후 에러코드가 0x57(매개변수 에러) 면 정상 실행인 상태고. 0x0(정상) 면 디버깅 상태입니다. 그럼 먼저 코드부터 보겠습니다.

```
#include <Windows.h>
main(){
    char fiber[1024] = { 0 };

    DeleteFiber(fiber);

    if (GetLastError() == 0x00000057){ ExitProcess(0); }
    MessageBox(0, L"DeleteFiber", L"Detected!", MB_OK);
}
```

Syntax

VOID WINAPI DeleteFiber(
__in LPVOID lpFiber
);

에러코드가 0x57 일 경우에 정상인 상태인지에 대해 문이 생깁니다. 뭐 이 코드에서는 당연히 0x57 에러가 뜰 수 밖에 없는게 단순히 선언만 한 후에 DeleteFiber 함수를 사용했기 때문이죠.

그럼 디버깅 시에는 정상적으로 작업을 완료할 수 있는 이유는?
DeleteFiber 함수는 Heap 과 직접 연관이 있습니다. (DeleteFiber 함수를 직접 분석 해 보세요)
그리고 힙 메모리는 힙의 특징을 가지는 플래그들을 설정하는데, 이 플래그들은 디버그시와 정상 실행시에 따라 값이 다릅니다. 그리고 실행 시에 이 플래그 값들이 설정됩니다.
(Windows NT 계열 운영체제 에서는 PspUserThreadStartup 를 통해 설정됩니다)

#define **FLG_HEAP_ENABLE_TAIL_CHECK** 0×00000010
#define **FLG_HEAP_ENABLE_FREE_CHECK** 0×00000020
#define **FLG_HEAP_VALIDATE_PARAMETERS** 0×00000040

다음이 그 플래그 값들인데, 정상 실행 할 때와 디버깅 할 시에 플래그 값이 다르기 때문에 디버그 시에 DeleteFiber 함수는 정상적으로 호출이 되는 겁니다.

- 2. PEB 를 이용한 안티 디버깅

이번에는 API 를 사용하지 않고 직접 PEB 구조체에 접근을 해서 값을 가져오는 방법을 적어보겠습니다.
그 전에 기술을 보기전에 PEB 에 대한 설명을 간략하게 해 보겠습니다.

- **PEB (Process Environment Block)**

이 구조체에는 프로세스에 대한 정보가 저장됩니다. 그럼 한번 이 구조체의 멤버들에는 어떤 값들이 있는지 살펴 봅시다. (필요한 것만 추려서 정리했습니다)
(구조체에 대한 설명은 자세하게 하지는 않겠습니다)

Struct

……
+0x002 BeingDebugged : UChar
……
+0x00c Ldr : Ptr32 _PEB_LDR_DATA
+0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014 SubSystemData : Ptr32 Void
+0x018 ProcessHeap : Ptr32 Void
+0x01c FastPebLock : Ptr32 _RTL_CRITICAL_SECTION
……
+0x064 NumberOfProcessors : Uint4B
+0x068 NtGlobalFlag : Uint4B
+0x070 CriticalSectionTimeout : _LARGE_INTEGER

– ProcessHeap
……
+0x036 Reserved
+0x038 UCRSegmentList
+0x040 Flags
+0x044 ForceFlags

PEB 구조체는는 TEB 구조체에서 포인트 되고, 0x30 에 위치합니다. 그리고 TEB 구조체는 fs 레지스터로 접근이 가능합니다.
그리고 위에 진한색칠 해 놓은 값들을 유심히 봐 주세요. 저 값들은 디버깅 시와 정상 실행시에 값이 달라지기 때문에 다버깅 여부를 판단이 가능합니다.

A. IsDebuggerPresent – Direct PEB

IsDebuggerPresent API 함수와 작동원리는 당연히 똑같고, 단지 차이는 API 함수 사용이냐 직접 PEB 구조체에 접근해서 가져왔느냐 입니다.

```
#include <Windows.h>
int IsDebuggerPresent_PEB(){
    __asm{
        mov eax, fs:[30h]
            mov eax, [eax + 2]
            and eax, 0x11
            test eax, eax
    }
}
main(){
    if (IsDebuggerPresent_PEB() == 1){
            MessageBox(0, L"IsDebuggerPresent_PEB", L"Detected!", MB_OK);
    }
}
```

코드설명을 하자면, fs:[30h] 로 TEB 0x30, PEB 로 가고, PEB 구조체 2 번째 값인 BeingDebugged 를 참조합니다.

B. NtGlobalFlag

해당 프로세스를 디버깅 할 때 설정되는 Flag 중 하나가 NtGlobalFlag 인데, PEB 0x68 에 위치하고, 정상 실행 경우엔 0 을 리턴합니다. 만약 디버깅 중이면 0x70 을 리턴하는데, 0x70 인 이유는 FLG_HEAP_ENABLE_TAIL_CHECK (0x10), FLG_HEAP_ENABLE_FREE_CHECK (0x20), FLG_HEAP_VALIDATE_PARAMETERS (0x40) 이 플래그 들이 디버깅 시에 세팅되기 때문에 3 개를 더한 0x70 이 되는 거다.

```
#include <Windows.h>
int NTGlobalFlags(){
    __asm{
        mov eax, fs:[30h]
            mov eax, [eax + 68h]
            and eax, 0x70
            test eax, eax
    }
}
main(){
    if (NTGlobalFlags() != 0){
        MessageBox(0, L"NTGlobalFlags", L"Detected!", MB_OK);
    }
}
```

C. ProcessHeap

PEB 0x18 에 위치한 ProcessHeap 에 Flag (0xC) 하고 ForceFlag(0x10) 값이 각각 0x2 와 0x0 을 갖기 않으면 디버깅 중이다.

```
#include <Windows.h>
int Flags(){
    __asm{
        mov eax, fs : [30h]
            mov eax, [eax + 0x18]
            mov eax, [eax + 0xC]
            cmp eax, 0x2
    }
}
int ForceFlags(){
    __asm{
        mov eax, fs : [30h]
            mov eax, [eax + 0x18]
            mov eax, [eax + 0x10]
            cmp eax, 0x10
    }
}
```

만약 비교한 값이랑 같으면 정상이고 같지 않으면 디버깅 중인 상태이다.

D. Ldr

PEB 구조체 0xC 에 위치하는 값인데, 디버깅중 힙을 할당하게 될 텐데, 사용하지 않는 힙영역을 0xFEEEFEEE 값으로 채워넣는다.

```
#include <Windows.h>
#include <tchar.h>
main(){
    FARPROC pProc = NULL;
    LPBYTE pTEB = NULL;
    LPBYTE pPEB = NULL;

    pProc = GetProcAddress(GetModuleHandle(L"ntdll.dll"), "NtCurrentTeb");
    pTEB = (LPBYTE)(*pProc)();
    pPEB = (LPBYTE)*(LPDWORD)(pTEB + 0x30);

    DWORD pLdrSig[4] = { 0xEEFEEEFE, 0xEEFEEEFE, 0xEEFEEEFE, 0xEEFEEEFE };
    LPBYTE pLdr = (LPBYTE)*(LPDWORD)(pPEB + 0xC);

    __try{
        for(;;){
            if (!memcmp(pLdr, pLdrSig, sizeof(pLdrSig))){
                MessageBox(0, L"Ldr", L"Detected!", MB_OK);
                break;
            }
            pLdr++;
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER){
        MessageBox(0, L"Ldr", L"Not Detected!", MB_OK);
    }
}
```

- **3. 예외 를 이용한 안티 디버깅**

A. Int 3, Int 2D

Int 3 은 우리가 주로 디버깅 할 때 사용하는 소프트웨어 브레이크 포인트다. 하는일은 인터럽트 3번을 호출하는거다. 디버거 에선 중단점을 트리거한다.

Int 2D 는 쉽게는 Int 3하고 비슷한 명령이다. 커널 모드에서 작동하는 BreakPoint 예외를 발생시키는 명령어이고, (유저에서도 예외를 발생함) 만약 디버거가 탐지되면 예외없이 넘어가고, 탐지되면 예외 처리가 된다.

(사용법은 아래에)

```
#include <Windows.h>
main(){
    __try{
        __asm{ int 3 }
    }

    __except (EXCEPTION_EXECUTE_HANDLER){
        MessageBox(0, L"Int 3", L"Not Detected!", MB_OK);
        ExitProcess(0);
    }
    MessageBox(0, L"Int 3", L"Detected!", MB_OK);
}
```

```
#include <Windows.h>
BOOL Int2D(){
    __try{
        __asm{
            int 0x2D
                xor eax, eax
                add eax, 2
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER){
        return FALSE;
    }
    return TRUE;
}
main(){
    if (Int2D()){
        MessageBox(0, L"Int 2D", L"Detected!", MB_OK);
    }
}
```

B. Trap Flag

Trap Flag(TF) 는 EFLAGS 레지스터 9 번째에 있는 값 입니다. 만약 TF 값이 1 로 설정 되어 있으면 CPU 는 Single Step 모드로 바뀌게 됩니다. 이 때 CPU 는 하나의 명령어를 실행 후 예외(EXCEPTION_SINGLE_STEP)를 발생시킵니다. 그리고 다시 TF 값은 0 으로 초기화 됩니다.

```
#include <Windows.h>
void main(){
    __try {
        __asm {
            pushfd
            or byte ptr ss:[esp+1], 0x1
            popfd
            nop
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        MessageBox(NULL, L"Trap Flag", L"Not Detected!", MB_OK);
        ExitProcess(1);
    }
    MessageBox(NULL, L"Trap Flag", L"Detected!", MB_OK);
}
```

C. Unhandled Exception Filter

만약 프로세스가 실행중에 예외가 발생하게 되면(윈도우에서), 미리 정의된 예외 핸들이 실행됩니다. 그리고 다른 예외 핸들에서 예외가 처리가 안되면, 마지막으로 Unhandled Exception Filter 로 넘겨집니다.

Syntax

```
LONG WINAPI UnhandledExceptionFilter(
  _In_  struct _EXCEPTION_POINTERS *ExceptionInfo
);
```

```
#include <Windows.h>
LONG WINAPI UnhandledExcepFilter(PEXCEPTION_POINTERS pExcepPointers){
    SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)pExcepPointers->ContextRecord->Eax);
    pExcepPointers->ContextRecord->Eip += 2;

    return EXCEPTION_CONTINUE_EXECUTION;
}
main(){
    SetUnhandledExceptionFilter(UnhandledExcepFilter);
    __asm{xor eax, eax}
    __asm{div eax}

    MessageBox(0, L"UnhandledExceptionFilter", L"Not Detected!", MB_OK);
}
```

이 코드에서 보면, 여기서 발생한 예외를 디버거에서 처리하지 못하기 때문에 결국 크래쉬가 날 겁니다.

D. PAGE GUARD

디버거가 보안 옵션이 PAGE_GUARD 인 메모릴 건드리게 되면 예외를 발생시킵니다. 그럼 보안 옵션이 PAGE_GUARD 인 곳을 만든 후 그곳을 실행하게 되면 만약 디버거가 실행 되어 있다면 예외가 발생될 겁니다.

```
#include <Windows.h>
void main(){
    DWORD OldProt;
    FARPROC Proc;
    LPVOID Func = VirtualAlloc(NULL, 0x10000, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

    RtlFillMemory(Func, 0x10, 0xC3);
    VirtualProtect(Func, 0x10, PAGE_EXECUTE_READWRITE | PAGE_GUARD, &OldProt);
    Proc = (FARPROC)Func;

    __try{
        Proc();
    }
    __except (EXCEPTION_EXECUTE_HANDLER){
        ExitProcess(1);
    }
    MessageBox(0, L"PAGE_GUARD", L"Detected!", MB_OK);
}
```

E. PreFix Handling

디버깅 중에, 어셈 명령인 prefix 명령은 실행되지 않고, 몇몇 다버거에선 예를들어 rep 같은 명령은 무시되는 경우가 있다. 그럼 아래 코드 같은 경우에는 디버깅 중에는 인터럽트가 실행되지 않고 예외가 실행되지 않을 것이다.

```
#include <Windows.h>
void main(){
    __try{
        __asm __emit 0xF3
        __asm __emit 0x64
        __asm __emit 0xF1
    }
    __except (EXCEPTION_EXECUTE_HANDLER){
        MessageBox(0, L"Prefix", L"Not Detected!", MB_OK);
        ExitProcess(1);
    }
    MessageBox(0, L"Prefix", L"Detected!", MB_OK);
}
```

0xF3 0x64 -> prefix rep

0xF1 -> int 1

F. Close Handle

존재하지 않는 핸들 값을 넣어서 예외를 발생시는 방법이다. 내부적으로 존재하지 않는 핸들 값으로 CloseHandle 이 호출되고, 그 다음 ZwClose 함수가 호출되 예외를 발생한다.
그럼 정상적으로 실행해도 예외가 발생하지 않나? 라는 의문이 들 텐데..
CloseHandle 함순는 특이하게 만약 프로세스가 디버거에 Attach 되었을 경우에만 예외를 발생시킨다.

```
#include <Windows.h>
void main(){
    __try {
        CloseHandle((HANDLE) 0x12345678);
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        MessageBox(NULL, L"CloseHandle", L"Detected!", MB_OK);
    }
}
```

G. Control + C

콘솔 프로그램일 경우, 디버깅 중이면 Ctrl + C 는 EXCEPTION_CTL_C 를 발생시키고, 정상 실행일 경우에는 signal-handler 가 호출됩니다.
AddVectoredExceptionHandler(1, (PVECTORED_EXCEPTION_HANDLER)ExHandler);
SetConsoleCtrlHandler((PHANDLER_ROUTINE)SignalHandler, TRUE);
success = GenerateConsoleCtrlEvent(CTRL_C_EVENT, 0);
이런 식으로 사용하면 된다.

H. CMPXCHG8B and LOCK

어셈블리 lock prefix 는 프로세서가 공유 메모리 영역에 접근하는 단일 프로세서인지 확인하는데 사용되어지는 핀을 지정하는데 사용합니다. 또 멀티 프로세서 시스템들에 사용되죠.
CMPXCHG8B 하고 LOCK prefix 명령은 같이 동작하지는 않는데, 만약 이 두개가 실행되면 성공적으로 존재하지 않는 인스트럭션 에러가 발생합니다. 만약 이 코드가 다버거에 의해 실행되면 디버가는 존재하지 않는 인스트럭션 예외를 받고 프로세스를 종료시킬 겁니다.
만약 정상 실행중이면 예외를 트랩 가능하고 계속 정상적으로 코드 실행이 가능하죠.

```
#include <Windows.h>
void error(){
    MessageBox(0, L"No Debugger Detected", L"Not Detected!", MB_OK);
}
void main(){
    SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)error);
    __asm {
        __emit 0xf0;
        __emit 0xf0;
        __emit 0xc7;
        __emit 0xc8;
    }
}
```

- **4. 시간차 를 이용한 안티 디버깅**

먼저 시간을 재는 방법을 설명하면 크게 2가지로 나눌 수 있다

-CPU 카운터를 재는 방법

RDTSC

QueryPerformanceCounter() / NtQueryPerformanceCounter()

GetTickCount

-시간을 재는 방법

timeGetTime()

_ftime()

주로 RDTSC(ReaD Time Stamp Counter) 를 이용해서 측정을 하는 이유는 다른 카운터를 재는 함수보다 더 정밀하기 때문이죠. 이유는 RDTSC 는 CPU 내부에 있는 카운터기 때문이죠

구현은 카운터를 재는 방법이나 시간을 재는 방법이나 비슷합니다.

A. RDTSC

X86 CPU에는 TSC 라는 CPU 카운터를 재는 64비트 레지스터가 있습니다. RDTSC 는 TSC 값을 EDX:EAX 형식으로 읽어옵니다. (edx 값이 상위 32비트, eax값이 하위 32비트)

```
#include <Windows.h>
void Func(){
    for (int i = 0; i < 1000; i++);
}
void main(){
    DWORD time = 0;
    __asm {
        pushad
            rdtsc; First Time Check
            push edx
            push eax

            call Func; Doing Something...

            rdtsc; Second Time Check
            pop esi
            pop edi

            cmp edx, edi; Compare Time
            ja Detected

            sub eax, esi
            mov time, eax
            cmp eax, 0xffff; Time Check
            jb NotDetected

        detected:
        rdtsc
            push eax
            retn; Go Somewhere..

        NotDetected:
        popad
    }
    MessageBox(0, L"RDTSC", L"Not Detectd!", MB_OK);
}
```

어려운 부분은 딱히 없고 주석을 달아놓았으니 설명은 스킵!

B. GetTickCount

앞에 rdtsc 하고 똑같은 역할을 하는 함수이다. 이 함수는 시간을 밀리세컨드 단위로 반환한다.

```
T1 = GetTickCount(); T2 = GetTickCount();
if ((T2-T1) > 0x10) { // Detected!
}
```

- **5. 하드웨어 & 가상장치를 이용한 안티 디버깅**

A. Hardware Break Point

IA32 에서 제공하는 디버그 레지스터들을 이용해서 만들어진 브레이크 포인터를 말합니다. IA32 에는 DR0~DR7 까지 총 8 개의 디버그 레지스터가 있습니다.
그 중 DR0~DR3 는 BP 를 설정할 주소를 지정하는데 사용됩니다. 여기서 한번에 최대 4 개의 하드웨어 브레이크 포인트를 설치할 수 밖에 없는 이유가 이겁니다.
+ DR4,DR5 는 사용하지 않습니다.

```
#include <Windows.h>
void main(){
    CONTEXT ctx;

    ZeroMemory(&ctx, sizeof(CONTEXT));
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;

    if (GetThreadContext(GetCurrentThread(), &ctx) == 0){
        MessageBox(0, L"HardwareBreakPoints", L"Detected!", MB_OK);
    }

    if (ctx.Dr0 != 0 || ctx.Dr2 != 0 || ctx.Dr3 != 0 || ctx.Dr6 !=0 || ctx.Dr7 != 0){
        MessageBox(0, L"HardwareBreakPoints", L"Detected!", MB_OK);
    }
}
```

B. VM Detection

1. LDT Register

SIDT(Store Interrupt Descriptor Table) 는 IDTR(Interrupt Descriptor Table Register) 의 정보를 저장 하는데, 내부 여러 인터럽트 관련 정보를 저장하는 테이블 입니다.
IDTR 은 단 한개 밖에 존재하지 않기 때문에 호스트하고 게스트 운영체제가 VMM 위에 동시에 돌아가려면 서로 두개의 운영체제가 IDTR 이 겹치지 않게 다른 메모리 위에서 돌아가게 해야 합니다. 이렇게 겹치지 않게 해 주는걸 Memory Relocation 이라 합니다.
그런데 재배치 된 메모리는 IDTR 에 올라온 주소와는 좀 다릅니다.
윈도우일 경우엔 IDT 가 0x80FFFFFF , Vmware 위에서 재배치된 IDT 주소는 0xFF 로 시작합니다.
(다른 가상화 프로그램에서도 또 각각 다름)
탐지하는 방법은 아래 코드와 같습니다.

```
int IDTCheck() {
unsigned char m[6];
__asm { sidt m }
if(m[5] > 0xd0){
return 1; } return 0;
}
```

첫째 바이트를 0xd0 와 비교해서 크면 가상장치 위에 있다는걸 판단합니다. 문제는 이 방법으로는 멀티코에서는 되지 않습니다.

왜냐하면 싱글코어에선 IDT 위치가 변경되지 않기 때문에 재배치된 IDT 인지 판별이 가능했지만 멀티코어에서는 서로 다른 코어에서 프로세스가 동작할 때 IDT 값이 바뀔 수 있습니다. 그래서 LDT(Local Descriptor Table) 를 사용해 탐지를 합니다.

```
int LDTCheck(){
unsigned char m[6];
__asm { sldt m }
if(m[0] != 0x0 && m[1] != 0x0){
return 1; }
return 0; }
```

2. STR Register

STR(Store Task Register)로, ldt 하고 비슷한거다. 첫번째 바이트가 x00 이고 2번째 바이트가 0x40 이면 VMware 로 탐지를 한다. 코드는 아래와 같습니다.

```
unsigned char m[4] = {0};
__asm { str mem }
if ((m[0] == 0x00) && (m[1] == 0x40)){
MessageBox(NULL, L"VMware", L"Detected!", MB_OK);
}
```

3. Register

직접 Vmware 레지스터 값을 찾아서 현재 서비스가 실행되어 있는지를 체크한다.

```
#include <Windows.h>
Void main(){
Char lszValue[100];
HKEY hkey;
Int i=0;
RegOpenKeyEx(HKEY_LOCAL_MACHINE,"SYSTEM₩CurrentControlSet₩Servives₩Disk₩Enum", 0 ,
KEY_READ, & hkey);
RegQueryValue(hkey, "0", lszValue, sizeof(lszValue));
RegCloseKey(hkey);
```

- **6. 기타 안티 디버깅 기법**

A. Memory Break Point

```
#include <Windows.h>
void main(){
    UCHAR *pMem = NULL;
    SYSTEM_INFO sysinfo = { 0 };
    DWORD OldProt = 0;
    void *Alloc = NULL;

    GetSystemInfo(&sysinfo);
    Alloc= VirtualAlloc(NULL, sysinfo.dwPageSize, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
    pMem = (UCHAR*)Alloc;
    *pMem = 0xc3;
    VirtualProtect(Alloc, sysinfo.dwPageSize, PAGE_EXECUTE_READWRITE | PAGE_GUARD, &OldProt);

    __try{
        __asm{
            mov eax, Alloc
                push MemBpBeingDebugged
                jmp eax
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER){
        VirtualFree(Alloc, (SIZE_T)NULL, MEM_RELEASE);
        return;
    }

    __asm{MemBpBeingDebugged:}
    VirtualFree(Alloc, (SIZE_T)NULL, MEM_RELEASE);
    MessageBox(0, L"Memory BreakPoints", L"Detected!", MB_OK);
}
```

간략하게 코드를 설명하면

pMem = (UCHAR*)Alloc;

*pMem = 0xc3;

메모리에 ret(opcode 0xc3) 을 넣고

mov eax, Alloc 부분은 Alloc 주소를 eax 에 넣고

eax 로 점프한다. // 만약 디버거에 Attach 되어있으면 예외 발생함

B. Erase PE Header

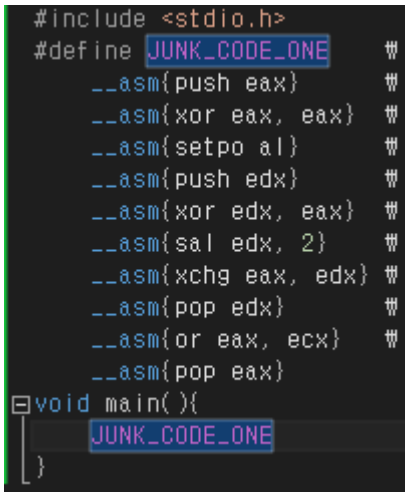
메모리에 있는 PE Header 부분을 지워서 덤프 과정을 방해함. 안티 디버깅 보다는 안티 덤핑 기술입니다.

```
#include <Windows.h>
void main(){
    char *BaseAddr = (char *)GetModuleHandle(NULL);
    DWORD Protect = 0;

    VirtualProtect(BaseAddr, 4096, PAGE_READWRITE, &Protect);
    ZeroMemory(BaseAddr, 4096);
}
```

C. Junk Code

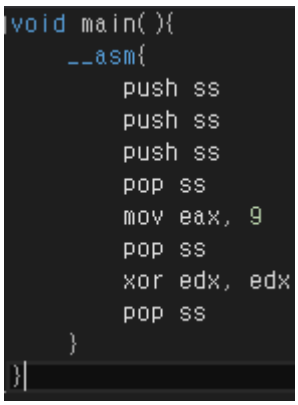
이 기법도 안티 디버깅 기법은 아니지만, 의미없는 코드를 많이 넣음으로 리버서가 프로그램을 분석하는데 어려움을 주는 기술? 입니다.



D. Stack Segment

Stack Segment 레지스터, push / pop ss 하면 다음 코드는 실행은 되지만 Step Into 가 불가능 합니다. 분석을 약간 불편하게 할 수 있는 기법이다.

* 원리를 다시 설명하면, 원래는 디버거가 한 Step 씩 실행하는 원리가 TF 플래그를 설정 해 놓으면 명령 1 개 실행하고 예외가 발생하고 cpu 가 다시 디버거로 이동 해서 다음 instruction 을 실행하지만, pop ss 같은 경우에는 다음 instruction 이 실행 될 때 까지는 아무런 인터럽트를 발생 하지 않게 하기 때문에 pop ss 명령 바로 뒤 명령에는 Step In 할 수 없습니다.



E. Ollydbg OutputDebugString Format String Bug

올리디버그에 몇개의 버그들이 존재하는데 그 중 하나가 포맷스트링 버그다. OutputDebugString 함수로 %s%s%s... %s%s 이런식으로 많은 포맷스트링을 보내게 되면 크래쉬가 난다.

__try {

OutputDebugString(TEXT("%s%s%s%s%s%s%s%s%s%s%s")

```
, TEXT("%s%s%s%s%s%s%s%s%s%s%s"),
TEXT("%s%s%s%s%s%s%s%s%s%s%s"),
TEXT("%s%s%s%s%s%s%s%s%s%s%s") );
}
__except (EXCEPTION_EXECUTE_HANDLER){printf("Crash!₩a₩n");}
```

F. Ollydbg File Format String Bug

올리디버그 그 다음으로 볼 버그는. 파일명 버그인데. 파일명을 예를들어 %s%s 같이 해 놓으면 크래쉬가 난다. 그럼 크래쉬가 나니까 리버서들은 파일 이름을 다른 거로 바꿀 것이다.
그럼 이를 방지하기 위해 아래와 같이 프로그램 이름을 바꾸지 못하게 하는 코드를 만들면 된다.

```
GetModuleFileName(0, (LPWCH)&pathname, 512);
filename = wcsrchr(pathname, L'₩₩');
if (wcsncmp(filename, L"₩₩%s%s.exe", 9) == 0) {
MessageBox(NULL, L"이름 안바뀜", L"정상임", MB_OK);
} else
MessageBox(NULL, L"이름 수정됨 ㅇㅇ", L"정상아님", MB_OK);
```

G. CRC Checking

CRC Checking은 예를들어 한 함수가 있으면 그 함수가 리버서에 의해서 패칭되는걸 막기나 브레이크 포인트가 찍히는걸 막기 위해서 그 함수의 CRC 값을 미리 구해 저장해 놓은 뒤에 CRC 값 확인 루틴을 만들어 그 함수가 수정되었나를 체크하는 기법이다.

```
void crc(){
crc = CRCCCITT((unsigned char*)&crc,(DWORD)&check-(DWORD)&crc, 0xffff, 0); }
return;
}
void check() {
if (crc != original_crc){
MessageBox(NULL, L"CRC Check", L" Detected!", MB_OK);
}
return;
}
```

대충 이런 형태로 사용한다.

H. TLS CallBack

TLS(Thread Local Storage) 란 스레드별로 전역이나 정적 데이터를 지역데이터 처음 사용하기 위해 스레드별로 할당되는 독립적인 공간이다.

Callback 함수란 특정상황에 시스템에서 자동으로 호출해 주는 함수를 말하는데, 그 중에서 이번에 볼 TLS Callback 함수는 프로세스의 스레드가 생성되거나 종료될 때마다 자동으로 호출되는 함수를 뜻합니다. 당연히 main 함수도 포함하고요.

(자세한 내용은 기술 문서니 생략합니다)

```
#include <windows.h>
#pragma comment(linker, "/INCLUDE:__tls_used")

void NTAPI TlsCallBackFunction(PVOID instance, DWORD reason, PVOID reserved){
    if (IsDebuggerPresent()){
        MessageBox(0, L"TLS Callback", L"Detected!", MB_OK);
        ExitProcess(0);
    }
}

#pragma data_seg(".CRT$XLA")
PIMAGE_TLS_CALLBACK tls [] = { TlsCallBackFunction, 0 };
#pragma data_seg()

main(){
    MessageBox(0, L"Main Function", L"Main", MB_OK);
}
```