Anti-Revering Techniques

kozistr@gmail.com

http://zer0day.tistory.com

김 형 찬

2016/9/29

Contents

1. Introduction

이번 문서에서는 안티 리버싱 기법에 대한 전반적인 내용을 간략하게 다뤄볼 것이다. 이에 대한 세부적인 내용은 인터넷 등에 쉽게 검색으로 찾을 수 있기 때문에 자세한 설명은 생략하겠다.

1.1. Reversing

우리가 흔히 말하는 Reversing 은 Reverse Engineering 을 간략하게 줄여 말하는 거다. Reversing 이란 그 뜻 그대로 역으로 분석을 하는 거다. 우리가 주로 타겟으로 하는 리버싱 대상은 .dll, .exe, .sys 파일 등이 되겠고 이러한 실행 파일들을 대상으로 우리는 원시 코드들을 분석해서 프로그램이 어떤 역할을 하는지 알아내는 작업을 한다. 물론 분석을 위해선 해당 파일마다 다른 언어, 다른 아키텍쳐에서 작성 되었으니 각 assembly 언어도 잘 알고 있어야 한다.

1.2. Anti Reversing

이번에 중점적으로 다룰 Anti-Reversing 이란 주제는 간략하게 말해서 프로그램을 분석할 시에 분석하는 사람이 분석을 어렵게 만드는 작업을 말한다. 예를 들어 코드를 난독화 해두거나 안티 디버깅 기능을 넣어 디버깅이 어렵게 만들 던가. 그래서 주로 이런 기법들은 상용 게임이나 영상 및 노래(DRM), 악성코드 등에 사용되어 분석이 되는 것을 최대한 방지하는 목적으로 사용된다. 하지만 현대에는 이러한 기법들이나 기술들이 대부분 알려져 있고 분석하는 툴이 좋아지고 분석가의 실력이 나날이 높아져 아무리 견고하게 만들어도 크랙이 되기 나름이다. 그래서 현대에 안티 리버싱 기법의 목적은 “뚫리지 않은 장치” 가 아닌 “최대한 분석을 지연시키는” 장치로 사용된다. 이번 문서에서 다룰 기법들은 잘 알려져 있는 안티 리버싱 기법부터 잘 알려지지 않은 기법이나 악성코드에서 사용되고 있는 안티 리버싱 기법 등까지 간략하게 다뤄 볼 예정이다. 기법 설명에 들어가기 앞서서 크게 2 분야로 나눠보자면 안티 디버깅과 안티 리버싱인데, 안티 디버깅에선 디버거를 대상으로 탐지하는 방법부터 요즘 주로 분석 환경인 VM 이나 Sandbox 위에서의 탐지 기법까지 알아볼 것이다. 안티 어셈블리 파트 에서는 코드 난독화의 여러 방법부터 안티 덤핑까지 알아볼 것이다.

2. Anti-Debugging

아래처럼 안티 디버깅 기법을 크게 7 가지로 분류해 봤다. 아래 기법들은 윈도우 바이너리를 대상으로 한 기법들이다. 물론 특정 윈도우 버전(ex) Vista, NT 계열이 아닌 Windows)이나 아키텍쳐(x86, x64)에 차이가 날 수 있음을 명시한다.

2.1. Based on API Calls

A) CheckRemoteDebuggerPresent

이번 함수는 IsDebuggerPresent 함수와 동일한 역할을 하는데 다른 점은 현재 시점을 기준으로 값을 불러오는 점이다. 즉, 호출 될 때마다 값을 얻어 오기 때문에 IsDebuggerPresent 와 달리 BeingDebugged 값이 수동으로 바꿔 탐지를 못하는 것에 대한 문제는 없어진다.

구현을 해 보자면 아래와 같다.

```
void main() {
	BOOL Debugged;
	CheckRemoteDebuggerPresent(GetCurrentProcess(), &Debugged);

	if (Debugged)
		printf("Debugged\n");
}
```

B) FindWindow

다음 API 는 현재 실행되어 있는 윈도우 창을 타이틀 명으로 검색을 하는 함수이다. 즉. 이 함수로 현재 실행되어 있는 디버거의 타이틀 명 (ex) OLLYDBG, MyDEBUG, IDA, etc…) 으로 검색을 해 만약 있으면 디버거 사용 중이라 탐지를 하는 방식이다.

구현을 해 보자면 아래와 같다.

```
void main() {
	HWND ck = FindWindow(L"MyDEBUG", 0);

	if (ck != NULL)
		printf("Debugged");
}
```

C) OpenProcess

해당 기법은 OpenProcess 함수로 csrss.exe 를 PROCESS_ALL_ACCESS 권한으로 실행해 만약 실패하면 디버깅 중이 아닌 것이고 성공하면 디버깅 중인 것이다. 왜냐하면 일반적인 실행 때는 해당 권한으로 실행을 할 수 없지만 디버깅 시에는 이미 해당 권한을 사용해 프로세스를 attach 하거나 open 하기 때문에 성공적으로 PROCESS_ALL_ACCESS 권한으로 열 수 있다.

구현을 해 보자면 아래와 같다.

```
void main() {
	FARPROC proc = GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"CsrGetProcessId"
	);
	DWORD PID = proc(); // Get csrss.exe PID
	HANDLE h = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);

	if (h != NULL) // If it fails, return 0
		printf("Debugged");
}
```

D) Self-Debugging

이 기법은 간단하게 말하면 자기 자신은 디버깅 하는 기법으로, 자식 프로세스를 만들고 그 프로세스가 부모 프로세스를 디버깅 하는 방식으로 이뤄진다. 한번에 한 프로세스를 디버깅 할 수 있는 사실을 이용해 외부 디버거가 디버깅을 하지 못하게 할 수 있는 좋은 기법 중 하나다. 하지만 이 기법의 단점은 EPROCESS 구조체의 DebugPort 값을 0 으로 설정을 하면 우회가 가능하다.

구현을 해 보자면 아래와 같다.

```
void Self_Debugging() {
	DEBUG_EVENT de;
	STARTUPINFO si;
	PROCESS_INFORMATION pi;

	ZeroMemory(&de, sizeof(DEBUG_EVENT));
	ZeroMemory(&si, sizeof(STARTUPINFO));
	ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));

	GetStartupInfo(&si);
	CreateProcess(NULL, GetCommandLine(), NULL, NULL, FALSE, DEBUG_PROCESS,
		NULL, NULL, &si, &pi); // Create a Copy of itself
```

```
	ContinueDebugEvent(pi.dwProcessId, pi.dwThreadId, DBG_CONTINUE); // Continue
	WaitForDebugEvent(&de, INFINITE);
}

void main() {
	Self_Debugging();
}
```

E) OutputDebugString

해당 API 는 만약 debugging 중이면 성공적으로 문자열을 출력하고 만약 아니면 에러코드를 발생시킨다. (출력을 할 때 내부적으로 DbgPrint 사용). 즉 사용 했을 시 에러가 발생하지 않으면 디버깅 중이라는 뜻이다. 참고로 OllyDbg 패치가 적용되지 않은 구 버전에선 %s 같이 여러 개의 포맷스트링을 OutputDebugString 으로 건네 주게 되면 crash 가 나는 현상이 발생합니다.

이번엔 예제 코드가 짧아서 간단하게 SEH 도 추가해서 꾸며 보았다. 물론 GetLastError 함수를 사용해 값이 0 일 때를 비교해 디버깅 중이라는 걸 판별 할 수도 있다.

```
void main() {
	__try {
		OutputDebugStringW(L"asdf");
		__asm { mov ebx, [eax] } // EAX would be 0 or 1
					// Windows XP, 10 -> 1, 7 -> 0
		printf("Debugged");
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		;
	}
}
```

[+] 하나 OutputDebugString 함수 원형에 대해 재미있는 점을 발견할 수 있었는데. 이전 버전의 outputDebugString 함수 원형은 아래와 같았다.

```
void WINAPI OutputDebugStringA(LPCSTR lpOutputString) {
	ULONG_PTR args[2];
	args[0] = (ULONG_PTR)strlen(lpOutputString) + 1;
	args[1] = (ULONG_PTR)lpOutputString;

	__try {
		RaiseException(0x40010006, 0, 2, args);
		printf("Debugged");
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
```

```
            ;
        }
}
```

예외 처리 코드가 DBG_PRINTEXCEPTION_C(0x4001006) 였는데 이게 Windows 10 으로 넘어오면서 예외 처리 코드가 새로운 게 생겼다.

```
void WINAPI _OutputDebugString(LPCWSTR lpOutputString) {
        char outputDebugStringBuffer[1000] = { 0, };

        ULONG_PTR args[4];
        WideCharToMultiByte(CP_ACP, 0, lpOutputString, -1, outputDebugStringBuffer,
sizeof(outputDebugStringBuffer), 0, 0);

        // Unicode
        args[0] = (ULONG_PTR)wcslen(lpOutputString) + 1;
        args[1] = (ULONG_PTR)lpOutputString;

        //Ansi for compatibility
        args[2] = (ULONG_PTR)wcslen(lpOutputString) + 1;
        args[3] = (ULONG_PTR)outputDebugStringBuffer;

        __try {
                RaiseException(0x4001000A, 0, 4, args);
                printf("Debugged");
        }

        __except (EXCEPTION_EXECUTE_HANDLER) {
                ;
        }
}
```

DBG_PRINTEXCEPTION_WIDE_C(0x4001000a)가 새로 생겼다. 또한 Unicode 이외에 ANSI 버전에 대한 값도 요청을 한다.

F) BlockInput

아마 지금까지 본 함수 중에서 사용법이 제일 간단한 기법인 거 같다. 이 함수는 단순히 키보드나 마우스에서 발생된 이벤트 들을 막아주는 역할을 한다. 인자 값으로 1 을 넣게 되면 block 모드가 되고 0 을 넣으면 unblock 모드가 된다. 하지만 Ctrl + Alt + Delete 를 누르면 우회가 간단히 된다.

사용법은 아래와 같다.

```
BlockInput(1);  // block
BlockInput(0); // unblock
```

G) DeleteFiber

해당 API 는 Fiber 와 관련된 모든 정보를 지우는 함수이다. 그 모든 정보에는 물론 스택 내, 레지스터의 일부분, 피버 데이터를 포함한다. 특징으로 디버깅 중에 함수 사용 시 에러 반환 값이 0x57 이 아닌 값이 된다. 이걸 이용해서 탐지가 가능하다.

내부적으로 해당 함수는 다음과 같은 코드를 가지고 있다.

```
MOV EAX,DWORD PTR FS:[18] // TEB Structure

MOV ECX,DWORD PTR DS:[EAX+10] // Access TIB Structure -> FiberData Area

MOV ESI,DWORD PTR SS:[EBP+8] // lpFiber Value

CMP ECX,ESI // compare lpFiber & FiberData
```

후에 NtFreeVirtualMemory 를 사용해 stack 을 비워주고 ExitThread 로 함수를 종료한다.

구현을 해 보면 다음과 같습니다.

```
void main() {
	char fiber[1024] = { 0, };
	DeleteFiber(fiber);

	if (GetLastError() != 0x57)
		printf("Debugged");
}
```

H) LDR

이번 안티 디버깅 기법으로는 PEB 구조체에 12 번째에 있는 Ldr 구조체를 이용해서 디버깅을 탐지 할 수 있다. 디버깅 당하는 프로세스는 Heap 메모리에 자신이 디버깅 당하는 프로세스다 라는 표시로 사용되지 않은 Heap 영역을 0xfeeefeee 와 0xabababab 값으로 채운다. 이 Heap 주소가 Ldr 에 저장이 되고, 해당 주소를 기준으로 위와 같은 데이터들을 검색해 디버깅을 탐지 할 수 있다.

- **PEB Structure**
- typedef struct _PEB {
- BYTE Reserved1[2];
- BYTE BeingDebugged;
- ...

```
-     Struct _PEB_LDR_DATA *        Lrd; // PEB+0xC
-   } PEB, *PPEB;
```

구현을 해 보면 다음과 같습니다.

```
void main() {
	FARPROC proc = GetProcAddress(GetModuleHandle(L"ntdll.dll"), "NtCurrentTeb");
	UINT teb = (UINT)proc(); // TEB Address
	UINT peb = *(UINT *)(teb + 0x30); // PEB Address
	UINT ldr = *(UINT *)(peb + 0xC); // PEB+0xC -> Ldr

	__try {
		for (int i = 0; ; ++i) {
			if (*(UINT *)(ldr + i) == 0xfeeefeee || *(UINT *)(ldr + i) ==
0xababababab)
				printf("Debugged");
		}
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		;
	}
}
```

2.2. Based on Windows Internals

A) NtQueryObject

먼저 NtQueyrObject 함수는 Debug Object 의 리스트를 뽑아오는 역할을 하는 함수이다. Debug Object 란 XP 때 처음 나타났으며 디버깅 시에 이 객체가 생성이 된다. 즉 해당 함수로 리스트를 뽑아와서 존재하는 객체의 개수를 카운팅 해서 디버깅을 탐지 할 수가 있다. 객체의 개수가 1 개일 때만 디버거가 발견되지 않을 때이다.

구현을 해 보자면 다음과 같다.

```
typedef struct _LSA_UNICODE_STRING {
	USHORT Length;
	USHORT MaximumLength;
	PWSTR Buffer;
} LSA_UNICODE_STRING, *PLSA_UNICODE_STRING, UNICODE_STRING, *PUNICODE_STRING;

typedef struct _OBJECT_TYPE_INFORMATION {
	UNICODE_STRING TypeName;
	ULONG TotalNumberOfHandles;
	ULONG TotalNumberOfObjects;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;

typedef struct _OBJECT_ALL_INFORMATION {
	ULONG NumberOfObjects;
```

```
	OBJECT_TYPE_INFORMATION ObjectTypeInformation[1];
} OBJECT_ALL_INFORMATION, *POBJECT_ALL_INFORMATION;

typedef NTSTATUS(NTAPI *pNtQueryObject)(HANDLE, UINT, PVOID, ULONG, PULONG);

void main() {
	ULONG size = 0;

	pNtQueryObject nqo = (pNtQueryObject)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtQueryObject");

	NTSTATUS stat = nqo(NULL,
		3, // ObjectAllTypesInformation
		&size, 4, &size);

	PVOID pMemory = VirtualAlloc(NULL, size, MEM_RESERVE | MEM_COMMIT,
PAGE_READWRITE);

	// Get Object List
	stat = nqo((HANDLE)-1, 3, pMemory, size, NULL);

	if (stat != 0) {
		VirtualFree(pMemory, 0, MEM_RELEASE);
		return;
	}

	POBJECT_ALL_INFORMATION pObjectAllInfo = (POBJECT_ALL_INFORMATION)pMemory;
	PUCHAR pObjInfoLocation = (PUCHAR)pObjectAllInfo->ObjectTypeInformation;
	ULONG NumObjects = pObjectAllInfo->NumberOfObjects;

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

		if (wcscmp(L"DebugObject", pObjectTypeInfo->TypeName.Buffer) == 0) {
			// Check num of Objects is 1
			if (pObjectTypeInfo->TotalNumberOfObjects != 1)
				printf("Debugged or Debug Tool Found");
		}

		pObjInfoLocation = (PUCHAR)pObjectTypeInfo->TypeName.Buffer;
		pObjInfoLocation += pObjectTypeInfo->TypeName.Length;

		ULONG tmp = ((ULONG)pObjInfoLocation) & -4;
		pObjInfoLocation = ((PUCHAR)tmp) + sizeof(ULONG);
	}

	VirtualFree(pMemory, 0, MEM_RELEASE);
}
```

B) NtQuerySystemInformation

해당 함수는 현재 운영체제의 정보들을 가져오는 함수로 주로 운영체제가 디버그 모드로 부팅 됐는지 안 됐는지를 판별하는 안티 디버깅 기법으로 주로 사용된다. 해당 클래스에 존재하는 35 번째에 존재하는 SystemKernelDebuggerInformation(0x23)을 사용하면 SYSTEM_KERNEL_DEBUGGER_INFORMATION 구조체의 DebuggerEnabled 값이 인지를 확인 한다. 반환 값으론 EAX 에 값이 반환되는데 al 에 KdDebuggerEnabled 값이 들어가고, ah 에 KdDebuggerNotPresent 값이 들어간다. 즉 ah 에 0 이 들어가 있으면 디버깅 중이라는 뜻이다.

구현한 코드는 아래와 같다.

```
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION {
	BOOLEAN KdDebuggerEnabled;
	BOOLEAN KdDebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION;

typedef NTSTATUS(WINAPI *ntqsi)(ULONG, PVOID, ULONG, PULONG);

void main() {
	SYSTEM_KERNEL_DEBUGGER_INFORMATION skdi;
	ntqsi qsi = (ntqsi)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtQuerySystemInformation");

	qsi(0x23, // SystemKernelDebuggerInformation
		&skdi, 2, NULL);

	if (skdi.KdDebuggerEnabled && !skdi.KdDebuggerNotPresent)
		printf("Debugged");
}
```

C) NtSetInformationThread

Windows 2000 쯤에 안티 디버깅을 목적으로 만들어진(?) class 가 있었는데 그것이 바로 NtQueryInformationThread 다. 이번에 다뤄 볼 것은 ThreadInformationClas 에 존재하는 17 번째 멤버인 ThreadHideFromDebugger 이다. 해당 멤버를 사용하게 되면 프로그램이 계속 실행이 되도 디버거는 해당 Thread 로부터 어느 이벤트도 받지 않게 일종의 은닉을 할 수 있게 된다.

구현 코드는 아래와 같다.

```
typedef NTSTATUS(NTAPI *ntsit)(HANDLE, UINT, PVOID, ULONG);

void main() {
	ntsit sit = (ntsit)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtSetInformationThread");

	NTSTATUS stat = sit(GetCurrentThread(),
		0x11, // ThreadHideFromDebugger
		0, 0);

	if (stat == 0) printf("Hide");
}
```

물론 GetCurrentThread 로 현재 Thread 를 넣는 대신 자신이 넣고 싶은 Thread 를 넣어도 된다.

D) NtSetDebugFilterState

이 함수는 디버그 필터를 수정 하는 함수인데, 디버깅 시에는 이 필터 값 변경이 불가능한 점을 이용해 디버거를 탐지할 수 있다. 만약 디버깅 중 수정하려고 하면 STATUS_ACCESS_DENIED(0xc0000022)예외가 발생한다. 단점으로는 이 기법으로는 프로세스가 디버거에 attach 된 경우에만 탐지가 가능하다는 점이다.

구현한 코드는 아래와 같습니다.

```
typedef NTSTATUS(WINAPI *ntsdfs)(ULONG, ULONG, BOOLEAN);

void main() {
	ntsdfs sdfs = (ntsdfs)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtSetDebugFilterState");

	NTSTATUS stat = sdfs(0, 0, TRUE);

	if (stat == 0)
		printf("Debugged");
}
```

E) NtQueryInformationProcess

이번에 소개 할 함수로는 크게 4 가지를 이용해 안티 디버깅을 구현 할 수 있다.

- ProcessBasicInformation(0x0)

해당 기법은 약간 Aggressive 한 기법인데, 기본적으로 윈도우 상에서 프로그램을 실행하게 되면 자식 프로세스들(실행 프로그램)들의 부모 프로세스는 explorer.exe 다. (물론 아닌 경우도 있음). 이를 이용해 디버깅 시에는 디버거 프로세스에 자식으로 attach 하니까 현재 자신의 부모 프로세스 PID 를 explorer.exe 하고 비교하는 기법이다.

해당 인자로 NtQueryInformationProcess 를 실행하게 되면 리턴 값으로 PROCESS_BASIC_INFORMATION 구조체가 리턴 되는데, 해당 구조체에서

```
typedef struct _PROCESS_BASIC_INFORMATION {
	NTSTATUS ExitStatus;
	PVOID PebBaseAddress;
	PVOID AffinityMask;
	PVOID BasePriority;
	ULONG_PTR UniqueProcessId;
	ULONG_PTR ParentProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;
```

주의 깊게 볼 건 마지막 멤버인 InheritedFromUniqueProcessId 다. 해당 멤버엔 부모 프로세스의 pid 가 담겨있다. 이 값과 explorer.exe pid 를 비교해 만약 틀리면 디버깅 중이라 판단 할 수 있다.

구현한 코드는 아래와 같다.

```
typedef struct _PROCESS_BASIC_INFORMATION {
	NTSTATUS ExitStatus;
	PVOID PebBaseAddress;
	PVOID AffinityMask;
	PVOID BasePriority;
	ULONG_PTR UniqueProcessId;
	ULONG_PTR ParentProcessId;
} PROCESS_BASIC_INFORMATION, *PPROCESS_BASIC_INFORMATION;

typedef NTSTATUS(WINAPI *qip)(HANDLE, UINT, PVOID, ULONG, PULONG);

DWORD ExplorerPID() {
	DWORD pid = 0;
	GetWindowThreadProcessId(GetShellWindow(), &pid);
	return pid;
}

DWORD ParentProcessPID() {
	PROCESS_BASIC_INFORMATION pbi;
```

```
	ZeroMemory(&pbi, 24);
	qip proc = (qip)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtQueryInformationProcess"
	);

	NTSTATUS stat = proc(GetCurrentProcess(),
		0, // ProocessBasicInformation
		&pbi, 24, 0);

	if (stat == 0)
		return pbi.ParentProcessId;
	return 0;
}

BOOL IsParentExplorerExe() {
	return ParentProcessPID() == ExplorerPID();
}

void main() {
	if (!IsParentExplorerExe())
		printf("Debugged");
}
```

- ProcessDebugPort(0x7)

이 기법은 이미 간접적으로 한번 접했던 방법이다. 위에선 자세히 설명을 안 했지만, CheckRemoteDebuggerPresent 함수 내부를 분석하면 내부적으로 NtQueryInformationProcess 를 사용해 Debug Port 를 확인하는 코드가 있다.
프로세스가 디버깅 중일 때 Debug Port 가 할당되게 되는데 만약 디버깅 중이라면 -1 을 주고 디버깅 중이 아니면 0 을 반환한다.

구현한 코드는 아래와 같다.

```
typedef NTSTATUS(WINAPI *ntqip)(HANDLE, UINT, PVOID, ULONG, PULONG);

void main() {
	DWORD flag;
	ntqip qip = (ntqip)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtQueryInformationProcess");

	NTSTATUS stat = qip(GetCurrentProcess(),
		0x7, // DebugPort
		&flag, 4, NULL);

	if (stat == 0 && flag == -1)
		printf("Debugged");
}
```

- ProcessDebugObjectHandle(0x1e)

Windows XP 부터, 어떤 프로그램이 디버깅이 되면 디버깅 세션이 생성되고 그 객체에 대한 핸들도 생성되었다. 물론 그 핸들은 디버깅이 되고 있다는 정보를 담고 있다. 즉, NtQueryInformationProcess 함수를 사용해 그 객체에 접근 해 핸들 정보를 받아와 디버깅을 탐지 해 낼 수 있다. 0x1e 번째 멤버를 보면 ProcessDebugObjectHandl 이란 것을 사용한다.

구현한 코드는 아래와 같다.

```
typedef NTSTATUS(WINAPI *ntqip)(HANDLE, UINT, PVOID, ULONG, PULONG);

void main() {
	HANDLE h;
	ntqip qip = (ntqip)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtQueryInformationProcess");

	NTSTATUS stat = qip(GetCurrentProcess(),
		0x1e, // ProcessDebugObjectHandle
		&h, 4, NULL);

	if (stat == 0 && h)
		printf("Debugged");
}
```

- ProcessDebugFlags(0x1f)

다음은 ProcessDebugFlags 를 사용하는 기법이다. 3 번째 인자에 EPROCESS 구조체의 NoDebugInherit 값의 역인 값을 반환한다. 즉 리턴 값이 0 이면 디버깅 중 인 것이다.

구현한 코드는 아래와 같다.

```
typedef NTSTATUS(WINAPI *ntqip)(HANDLE, UINT, PVOID, ULONG, PULONG);

void main() {
	DWORD nodebug;
	ntqip qip = (ntqip)GetProcAddress(
		GetModuleHandle(L"ntdll.dll"),
		"NtQueryInformationProcess");

	NTSTATUS stat = qip(GetCurrentProcess(),
		0x1f, // ProcessDebugFlags
		&nodebug, 4, NULL);

	if (stat == 0 && !nodebug)
		printf("Debugged");
}
```

2.3. Based on Exception

A) UnHandledExceptionFilter

예외가 발생했을 때 등록된 예외 처리 문 이나 SEH Chain 이 없다면 최후의 보루(?)로 UnHandledExceptionFilter 함수가 실행이 된다. 한마디로 Top Level 예외 처리 장치다. 만약 디버거가 attach 되면 다른 예외 핸들러와 다르게 계속 실행되지 않고 바로 프로세스를 종료한다. 그 디버거를 탐지할 때는 내부적으로 NtQueryInformationProcess 함수와 ProcessDebugPort 가 사용된다.

Divide by zero 를 예시로 구현을 해 보면

```
LONG WINAPI UnhandledExcepFilter(PEXCEPTION_POINTERS pExcepPointers) {
	SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)
		pExcepPointers->ContextRecord->Eax);

	pExcepPointers->ContextRecord->Eip += 2; // Skip Exception Codes
	return EXCEPTION_CONTINUE_EXECUTION;
}

int main() {
	SetUnhandledExceptionFilter(UnhandledExcepFilter);
	__asm {xor eax, eax}
	__asm {div eax} // Divide by zero

	printf("Not Debugged");
}
```

B) CloseHandle

해당 함수는 평소에도 많이 사용하듯이 사용된 핸들을 종료하는 함수이다. 하지만 생성되지 않은 핸들을 종료하게 함으로 예외를 발생시켜 디버거를 탐지해 낼 수 있다. 일반적으로 실행을 하면 예외 발생 없이 지나가지만 디버거가 존재할 시에는 EXCEPTION_INVALID_HANDLE(0xc0000008)이 발생하게 된다. 참고로 x64 운영체제 에서는 EXCEPTION_HANDLE_NOT_CLOSABLE(0xc0000235)가 발생한다.

구현 코드를 보면 다음과 같다.

```
void main() {
	__try {
		CloseHandle((HANDLE)0x12345678);
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("Debugged");
```

```
	}
}
```

C) INT 3

INT 3 인스트럭션은 Software Break Point 로 우리가 많이 사용하는 break point 중 하나다. 만약 디버깅 중이라면 이 인스트럭션이 실행될 때 break point 가 트리거 되면서 자연스럽게 BP 로 인식해 넘어가게 되지만 아닐 때는 STATUS_BREAKPOINT(0x80000003) 예외가 발생하며 예외 구문에 걸리게 된다.

해당 구현 코드는 아래와 같다.

```
void main() {
	__try {
		__asm {
			int 3
		}
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("Not Debugged");
	}
}
```

D) INT 2D

INT 2d 은 독특하게 실행이 될 때 예외 주소로 현재 EIP 주소를 쓰고 1 을 증가시킨다. 하지만 EAX 값에 1, 3, 4 가 있을 경우만이다(Vista 만 5). 또한 디버거가 존재할 때 실행되면 예외를 발생 시키고 아닐 땐 발생을 시키지 않는다.

구현한 코드는 아래와 같다.

```
void main() {
	__try {
		__asm {
			int 0x2d
		}
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("Not Debugged");
	}
}
```

E) INT 41

이 인스트럭션은 커널 디버거를 탐지할 때 주로 사용되는 인스트럭션이다. 주로 이 명령은 ring 3 모드 즉 유저 모드에서 실행이 안되는 명령이다. 만약 실행을 하게 되면 int d 가 수행 되 EXCEPTION_ACCESS_VIOLATION(0xc0000005)예외가 발생하게 된다. 그런데 int 41 은 주로 DPL 값을 0 으로 가지고 있는데 몇몇 디버거 들은 이 인스트럭션을 ring 3 모드에서 실행하기 위해 3 으로 가지고있다.

구현한 코드는 아래와 같다.

```
void main() {
	__try {
		__asm {
			int 0x41
		}
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("Not Debugged");
	}
}
```

F) Prefix Handling

Prefix handling 기법은 prefix 인스트럭션을 사용해 디버깅 할 때는 prefix rep 명령은 생략, step over 가 되서 그냥 넘어가 지는데 뒤에 0xf1 인스트럭션(ice breakpoint)도 생략 되게 되 일반적인 경우엔 SINGLE_STEP_EXCEPTION 예외가 발생하게 된다. 참고로 ICE 는 In-Circuit-Emulator 의 약자다.

구현한 코드는 아래와 같다.

```
void main() {
	__try {
		__asm __emit 0xf3
		__asm __emit 0x64 // prefix rep
		__asm __emit 0xf1 // ice break point
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("Not Debugged");
	}
}
```

G) CMPXCHG8B and LOCK Prefix

LOCK prefix는 공유 메모리에 접근 권한이 있는 프로세서에 특별한 pin신호를 주게 되어 프로세서를 LOCK하는 역할을 하는 인스트럭션이다. 또한 CMPXCHG8B prefix는 CMP하고 XCHG, 값을 바꾸는 역할을 하는 인스트럭션이다. 하지만 이 두 인스트럭션이 같이 사용되는 경우에는 제대로 작동하지 않아 EXCEPTION_ILLEGAL_INSTRUCTION (0xc000001d)예외가 발생한다. 디버거를 사용하고 있을 때 해당 예외가 발생해 프로세스가 종료되지만 일반적으론 예외 핸들러에 의해 처리되어 정상적으로 실행이 된다.
구현한 코드는 아래와 같다.

```
void error(void) {
	printf("Not Debugged");
}

void main() {
	SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)error);
	__asm {
	__emit 0xf0
	__emit 0xf0 // LOCK prefix
	__emit 0xc7
	__emit 0xc8 // CMPXCHG8B prefix
	}
}
```

H) Guard Pages

해당 기법은 PAGE_GUARD 모드로 보호가 된 메모리에 디버거가 접근을 할 때 예외를 발생시키는 것을 이용한 기법입니다. EXCEPTION_GUARD_PAGE(0x80000001)예외가 발생했을 때 계속 디버깅을 하게 되면 예외를 그냥 지나치게 된다.

구현한 코드는 아래와 같다.

```
void main() {
   DWORD oProt;
   PVOID func = VirtualAlloc(NULL, 0x10, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

   memset(func, 0x10, 0xc3); // filled with c3 : ret
   VirtualProtect(func, 0x10, PAGE_EXECUTE_READWRITE | PAGE_GUARD, &oProt);
   FARPROC proc = (FARPROC)func;

   __try { proc(); }

   __except (EXCEPTION_EXECUTE_HANDLER) {
      printf("Not Debugged");
   }
}
```

I) CLI & STI

해당 인스트럭션들은 ring 3(유저 모드) 가 아닌 ring 0(커널 모드)용 어셈블리들이다. 이 명령어들을 ring 3 에서 사용하면 STATUS_PRIVILEGED_INSTRUCTION(0xc0000096) 예외가 발생한다. 참고로 cli 는 인터럽트들을 disable 하고 sti 는 그 반대의 역할을 한다.

구현한 코드는 아래와 같다. 아래는 간단한 예제이고 직접 SEH 나 VEH 등 직접 예외 처리 루틴을 제작해 사용하면 된다.

```
void main() {
	__try {
		__asm {
			sti
			cli
		}
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("ring 3");
	}
}
```

J) SEH

이번에는 asm 으로 간단한 예외를 처리하는 SEH 를 제작해 볼 거다. 대부분의 멀웨어나 패커에서는 코드 난독화나 안티 디스어셈블리 등의 목적으로 custom seh handler 를 직접 만들어 코드 encrypt 나 decrypt, 예외 처리 등을 한다.

Closehandle 함수의 INVALID_HANDLE 예외를 처리하는 코드는 아래와 같다.

```
EXCEPTION_DISPOSITION handler(PEXCEPTION_RECORD pExceptionRecord, PVOID
pEstablisherFrame,
	PCONTEXT pContextRecord, PVOID pDispatcherContext) {

	if (EXCEPTION_INVALID_HANDLE == pExceptionRecord->ExceptionCode)
		printf("debugged");
	return ExceptionContinueExecution;
}

void main() {
	__asm {
		push handler // seh handler
		push fs : [0]
		mov fs : [0], esp
		nop
		push 0xdeadbeef
```

```
            call CloseHandle // EXCEPTION_INVALID_HANDLE
            nop
            mov eax, [esp] // restore handler
            mov fs : [0], eax
            add esp, 8
      }
}
```

2.4. Based on Break Points

A) Hardware Break Point

하드웨어 breakpoint 는 인텔 아키텍쳐에 존재하는 것 중 하나로 이것들을 다루기 위해 Dr0 ~ Dr7 의 특수한 레지스터를 사용한다. Dr0 부터 Dr3 레지스터는 32bit 레지스터로 breakpoint 주소를 담는데 사용되고, Dr4 ~ Dr5 레지스터는 다른 목적을 위해 예약된 레지스터, Dr6 ~ Dr7 은 breakpoint 의 행동을 제어하기 위해서 사용된다. 즉, 하드웨어 breakpoint 를 탐지하기 위해선 Dr0 부터 Dr3 레지스터를 확인해 값이 들어있으면 하드웨어 breakpoint 가 설정된 것이다.

구현 한 코드는 아래와 같다.

```
void main() {
      UINT bps = 0;
      CONTEXT c;

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

      if (GetThreadContext(GetCurrentThread(), &c) != 0) {
            if (c.Dr0 != 0)
                  ++bps;
            if (c.Dr1 != 0)
                  ++bps;
            if (c.Dr2 != 0)
                  ++bps;
            if (c.Dr3 != 0)
                  ++bps;
      }

      printf("%d bps", bps);
}
```

2.5. Based on Flags

A) BeingDebugged

해당 함수는 TEB(Thread Environment Block)과 PEB(Process Environment Block)를 사용하는데, PEB 구조체에 BeingDebugged flag 를 검사해 디버깅을 확인 후 디버깅 중이면 TRUE, 아니면 FALSE 를 반환한다.

- **PEB Structure**
- typedef struct _PEB {
- BYTE Reserved1[2];
- **BYTE BeingDebugged;**
- ...
- } PEB, *PPEB;

해당 함수를 disassemble 해 보면 다음과 같은 코드를 볼 수 있는데, 직접 PEB 구조체에 접근해서 BeingDebugged 값을 가져오는 것을 볼 수 있다.

```
mov eax, fs:[0x18] // TEB 접근, FS:[18] 엔 TEB 주소가 있음
mov eax, ds:[eax+0x30] // PEB 접근, TEB 0x30 에 PEB 가 있음
movzx eax, ds:[eax+0x2] // PEB 2번째 BeingDebugged 값을 eax 에 넣음
```

코드로 구현을 해 보자면 아래와 같다.

```
void main() {
	if (IsDebuggerPresent())
		printf("Debugged\n");
}
```

B) Trap Flag

TF(Trap Flag)는 EFLAGS 레지스터에 9 번째 bit 에 존재하며 해당 값을 1 로 세팅을 하면 CPU 는 Single Step mode 로 변경되고, 이 모드일 경우엔 명령어를 1 개 실행하고 EXCEPTION_SINGLE_STEP(0x80000004)를 발생시킨다. 그 후 TF 값을 다시 0 으로 세팅한다. 디버깅 중에는 해당 레지스터에 영향을 주지 않을 것이고 즉 예외는 발생하지 않을 것이다.

구현한 코드는 아래와 같습니다.

```
void main() {
	__try {
```

```
		__asm {
			pushf
			mov [esp], 0x100 // Set TF to 1
			popf
		}
	}

	__except (EXCEPTION_EXECUTE_HANDLER) {
		printf("Not Debugged");
	}
}
```

C) NtGlobal Flag

해당 Flag 는 PEB 구조체 0x68 번쨰에 존재하는 멤버로, 디버거가 존재하면 0 이 아닌 값이 들어가게 되고 존재하지 않으면 0x0 이 들어가게 된다.

```
- PEB Structure
- typedef struct _PEB {
- ...
-   ULONG                          NumberOfProcessors;
-   ULONG                          NtGlobalFlag;
-   UNION _LARGE_INTEGER           CriticalSectionTimeout;
- ...
- } PEB, *PPEB;
```

확인해 보면 0x70 이란 값이 들어가게 되는데 이는

FLG_HEAP_ENABLE_TAIL_CHECK 0x10

FLG_HEAP_ENABLE_FREE_CHECK 0x20

FLG_HEAP_ENABLE_VALIDATE_PARAMETERS 0x40

이 3 개 값이 합쳐 저서 0x70 이 된다.

구현한 코드는 아래와 같다.

```
void main() {
	__asm {
		mov eax, fs:[0x30]
		mov eax, [eax + 0x68]
		cmp eax, 0x0
		jne debug
			push 1
			call exit
		debug:
	}
```

```
        printf("Debugged");
}
```

D) Heap Flag

PEB 구조체에 0x18 에 존재하는 ProcessHeap 구조체에 Flags 와 ForceFlags 두 flag 를 이용해서 디버깅 탐지가 가능하다. Flags 는 0xc 에 위치하고 ForceFlags 는 0x10 에 위치한다.

```
- PEB Structure
- typedef struct _PEB {
- ...
-   ULONG                         MaximumNumberOfHeaps      ;
-   PPVOID                        *ProcessHeap;
-   PVOID                         GdiSharedHandleTable;
- ...
- } PEB, *PPEB;
```

Flags 는 디버깅 중이면 0x50000062 로 값이 설정되고 아닌 경우엔 0x2 로 설정된다. ForceFlags 같은 경우엔 디버깅 중이면 0x40000060 아닌 경우엔 0x0 으로 설정된다.

```
- ProcessHeap Structure
- typedef struct _ProcessHeap {
-   Entry                         _HEAP_ENTRY
-   UINT                          Signature;
-   UINT                          Flags;
-   UINT                          ForceFlags;
-   UINT                          VirtualMemoryThreshold;
- ...
- } ProcessHeap;
```

값들은 아래와 같은 정의된 값으로 설정이 된다.

HEAP_GROWABLE 0x2

HEAP_TAIL_CHECKING_ENABLED 0x20

HEAP_FREE_CHECKING_ENABLED 0x40

HEAP_SKIP_VALIDATION_CHECKS 0x10000000

HEAP_VALIDATE_PARAMETERS_ENABLED 0x40000000

구현한 코드는 아래와 같다.

- Flags Example

```
void main() {
    __asm {
        mov eax, fs:[0x30]
        mov eax, [eax + 0x18]
        mov eax, [eax + 0xc]
        cmp eax, 0x2
        jne debug
               push 1
               call exit
        debug:
    }

    printf("debugged");
}
```

- ForceFlags Example

```
    void main() {
        __asm {
            mov eax, fs:[0x30]
            mov eax, [eax + 0x18]
            mov eax, [eax + 0x10]
            cmp eax, 0x0
            jne debug
                push 1
                call exit
            debug :
        }

        printf("debugged");
}
```

2.6. Based on VM Detection

A) Red Pill

- SIDT

IDT(Interrupt Descriptor Table)란 인터럽트 처리에 관한 디스크립터들을 모아둔 구조체이다. 프로세서당 1 개의 IDT 레지스터(IDTR)가 존재하고 Guest OS 와 Host OS 가 구동 중이면 IDT 는 재배치 된다. IDTR 로 IDT 메모리 위치를 알 수 있는데 이를 통해서 VM 을 탐지해 낼 수 있다. SIDT(Store Interrupt Descriptor Table)명령으로 IDTR 을 가져 올 수 있다. VMware 나 VirtualPC 등의 VM 에서는 주로 0xd0000000 이상의 주소가 나오기 때문에 해당 값보다 크면 가상 머신으로 탐지를 할 수 있다. 하지만 이 기법은 멀티 프로세서같이 IDT 가 여러 개 존재할 가능성이 있어 오작동 가능성이 높다. 또한 요즘 VM 들의 EPT 기능 같이 메모리 가상화 기술을 사용하면 ID 우회가 가능하다.

참고로 VMware 는 0xffxxxxxx 의 주소고, VirtualPC 는 0xe8xxxxxx. 로컬에서 윈도우는 0x80xxxxxx, 리눅스는 0xc0xxxxxx.

구현한 코드는 아래와 같다.

```
void main() {
	unsigned char idt[6];
	__asm { sidt idt }

	if ((int)idt[5] >= 0xd0)
		printf("VM");
}
```

B) No Pill

No Pill 기법은 Red Pill 기법과 다르게 IDT 구조체를 운영체제가 아닌 프로세서에 할당하는게 차이이다. Host OS 에서는 LDT 위치는 0 이고 VM 에서는 0 이 아니다.

- SGDT

해당 기법은 SGDT(Store Global Descriptor Table)명령으로 GDTR(Global Descriptor Table Register)정보를 가져온다. 위에 LDT 를 이용한 탐지와 비슷하게 최상위 바이트가 0xff 인 것을 확인해 VM 을 판별한다.

구현한 코드는 아래와 같다.

```
void main() {
	UCHAR gdt[6];

	__asm { sgdt gdt }

	UINT gdta = *((PULONG)&gdt[2]);
	if ((gdta >> 24) == 0xff)
		printf("VM");
}
```

- SLDT

해당 기법은 SLDT(Store Local Descriptor Table)명령으로 LDTR(Local Descriptor Table Register)로 부터 Segment Selector 를 구해, 호스트 머신에선 주소가 0x0000 이니 만약 이 값이 아니면 VM 으로 탐지가 가능하다.

구현한 코드는 아래와 같다.

```
void main() {
	UCHAR ldt[2];

	__asm { sldt ldt }

	USHORT ldta = *((PUSHORT)&ldt);
	if (ldta != 0x0000)
		printf("VM");
}
```

C) I/O Port

I/O 포트는 Guest OS 하고 Host OS 가 통신을 할 수 있는 통신 채널이다. I/O 에 사용되는 명령어는 EFLAGS 레지스터를 이용해 실행되는데 이는 커널 레벨에서만 동작을 한다. 유저 레벨에서 실행하면 예외가 발생한다. I/O 를 위한 명령에서는 in 과 out 이 있는데, Guest OS 에서 in 명령 실행 시 예외가 발생하지 않는다.

구현한 코드는 아래와 같다.

- verison

```
void main() {
	UINT a = 0, b = 0;
	__try {
		__asm {
			pushad
			mov eax, 0x564d5868 // vmware magic valu, VMXh
```

```
                mov ecx, 0xa // vmware version
                mov dx, 0x5658 // vmware I/O port, VX
                in eax, dx
                mov a, ebx // check is it VM
                mov b, ecx // get version
                popad
            }
        }

        __except (EXCEPTION_EXECUTE_HANDLER) {
            return;
        }

        if (a == 'VMXh') {
            printf("VM, version : ");
            switch (b) {
            case 1: printf("Express"); break;
            case 2: printf("ESX"); break;
            case 3: printf("GSX"); break;
            case 4: printf("Workstation"); break;
            default: printf("Unknown"); break;
            }
        }
}
```

- memory size

```
void main(void) {
        __try {
            __asm {
                pushad
                mov eax, 0x564d5868 // vmware magic value : VMXh
                mov ecx, 0x14 // memory size
                mov dx, 0x5658 // vmware I/O port : VX
                in eax, dx
                cmp eax, 0
                jbe notvm
                    push 1
                    call exit
                notvm :
                    popad
            }

            printf("Not VM");
        }

        __except (EXCEPTION_EXECUTE_HANDLER) {
            return;
        }
}
```

D) STR

STR(Store Task Register)명령으로 Task Register 값을 받아오는데, 처음이 0x0040 으로 시작을 하게 되면 VM 인 것을 알 수 있다.

구현한 코드는 아래와 같다.

```
void main() {
	UCHAR mem[4] = { 0, };

	__asm { str mem }

	if (mem[0] == 0x00 && mem[1] == 0x40)
		printf("VM");
}
```

E) SMSW

해당 기법은 SMSW(Store Machine Status Word)명령으로 머신의 정보를 가져와 저장하는 명령이다. CR0 레지스터에서 값을 가져온다. 운영체제마다 값이 다르고 Host 와 Guest 에 따라 또 값이 다르다. 예를 들어 Windows 10 에서는 0x80050033 값을 가지고 VM Windows 10 에서는 0x80050031 이란 값을 가지고 VM Windows 7 에서는 0x80010031 을 가진다. 이런 값들을 기반으로 해서 탐지가 가능하다.

구현한 코드는 아래와 같다.

```
void main() {
	__asm {
		xor eax, eax
		smsw eax;
		cmp eax, 0x80050031 // Windows 10 on VM
		je vm
		cmp eax, 0x80010031 // Windows 7 on VM
		je vm
			push 1
			call exit
		vm:
	}

	printf("VM");
}
```

F) CPUID

CPUID 명령어는 CPU 식별을 위한 명령어이다. 파라메터(EAX 값)에 특정 값을 넣어 정보를 얻어 크게 2 가지 방법으로 VM 탐지가 가능한데, EAX 에 1 을 넣었을 때 하고 0x40000000 일 때이다. 먼저 EAX 에 1 을 넣었을 때는, Guest OS 가 켜져 있지 않은 평상시라면 EAX 값이 1 일 때 CPUID 실행 시 3 번째 ECX 값은 0 이여야한다. Guest OS 가 켜져 있으면 ECX 는 VMware 에서 지정해준 serial 값이 들어가 있어 1 일 것이다.

구현한 코드는 아래와 같다.

```
void main() {
	__asm {
		mov eax, 1
		xor ecx, ecx
		push ebx
		cpuid
		pop ebx
		sar ecx, 0x1f
		cmp cl, 1
		je detect
			push 1
			call exit
		detect:
	}

	printf("VM");
}
```

두번쨰로 EAX 값이 0x40000000 일 때, 12 바이트 길이의 문자열을 EBX, ECX, EDX 에 반환한다. 이 문자열에는 장치 정보가 담겨 있는데 로컬 머신일 경우엔 아무런 문자열이 보이지 않지만 VM 일 경우에는 해당 VM 에 관한 정보가 담겨있다. 예를 들어,

- VMware : “VMwareVMware”
- VirtualBox : “VBoxVBoxVBox”
- Parallels : “prl hyperv “
- KVM : “KVMKVMKVM\0\0\0”
- Xen : “XenVMM XenVMM”
- Microsoft Hyper-V, Windows Virtual PC : “Microsoft Hv”

하지만 이 기법 또한 CPU 가 Hyper Thread 를 지원하거나 Guest OS 가 이를 사용하면 오작동 가능성이 있다.

G) MAC Address

VM 장치들이 갖는 고유한 MAC Address 의 일부를 체크해 탐지하는 방법도 있다. 주로 맥 주소의 맨 앞의 3 개의 필트에는 제조사의 특정 값을 넣게 되는데 물론 VMware 나 Virtual PC 같은 VM 들도 특정 값들을 여러 개 가지고 있다.

- 00:05:69 (VMware)
- 00:0C:29 (VMware)
- 00:1C:14 (VMware)
- 00:50:56 (VMware)
- 08:00:27 (VirtualBox)
- 00:03:ff (Virtual PC)
- 00:0d:3a (Virtual PC)
- 00:50:f2 (Virtual PC)
- 7c:1e:52 (Virtual PC)
- 00:12:5a (Virtual PC)
- 00:15:5d (Virtual PC)
- 00:17:fa (Virtual PC)
- 28:18:78 (Virtual PC)
- 7c:ed:8d (Virtual PC)
- 00:1d:d8 (Virtual PC)
- 00:22:48 (Virtual PC)
- 00:25:ae (Virtual PC)
- 60:45:bd (Virtual PC)
- dc:b4:c4 (Virtual PC)
- 00:1c:42 (Parallels)

맥 주소를 구하는 코드는 아래와 같다.

```
void main() {
	ULONG len = 0;
	PIP_ADAPTER_INFO p;

	if (GetAdaptersInfo(p, &len) == ERROR_BUFFER_OVERFLOW) {
		p = (PIP_ADAPTER_INFO)new char[len];
		GetAdaptersInfo(p, &len);
```

```
            printf("MAC : %#x\n", p->Address);
            // VM Detect with MAC
            if (p->Address[0] == 0x08 && p->Address[1] == 0x00 &&
                p->Address[2] == 0x27)
                  printf("Virtual Box");
            delete p;
      }
}
```

H) ETC

위에 소개된 기법 이외에도 설명하지 못한 기법들이 여러가지 있다. 간략하게 정리해 보겠다.

- VM Process

VM 환경에서주로 볼 수 있는 프로세스들이 있다. 해당 프로세스 존재 유무로 VM 환경임을 알 수 있다.

- VMtoolsd.exe (VMware)
- VMwaretrat.exe (VMware)
- VMwareUser.exe (VMware)
- VMwareService.exe (VMware)
- VMwareTray.exe (VMware)
- Vmacthlp.exe (VMware)
- vboxservice.exe (VirtualBox)
- vboxtray.exe (VirtualBox)

- VM Files

다음은 VM 에 있는 주요 파일들을 기반으로 탐지하는 방법이다. 해당 파일 유무로 VM 을 판별할 수도 있다. 아래는 대표적 파일들을 적어보았다.

- C:\Windows\System32\Driver\Vmmouse.sys (VMware)
- C:\Windows\System32\Driver\vm3dgl.dll (VMware)
- C:\Windows\System32\Driver\vmdum.dll (VMware)
- C:\Windows\System32\Driver\vm3dver.dll (VMware)

- C:\Windows\System32\Driver\vmtray.dll (VMware)
- C:\Windows\System32\Driver\VMToolsHook.dll (VMware)
- C:\Windows\System32\Driver\vmmousever.dll (VMware)
- C:\Windows\System32\Driver\vmhgfs.dll (VMware)
- C:\Windows\System32\Driver\vmGuestLib.dll (VMware)
- C:\Windows\System32\Driver\VmGuestLibJava.dll (VMware)
- C:\Windows\System32\Driversvmhgfs.dll (VMware)
- C:\Windows\System32\Driver\VBoxMouse.sys (VirtualBox)
- C:\Windows\System32\Driver\VBoxGuest.sys (VirtualBox)
- C:\Windows\System32\Driver\VBoxSF.sys (VirtualBox)
- C:\Windows\System32\Driver\VBoxVideo.sys (VirtualBox)
- C:\Windows\System32\vboxdisp.dll (VirtualBox)
- C:\Windows\System32\vboxhook.dll (VirtualBox)
- C:\Windows\System32\mrxnp.dll (VirtualBox)
- C:\Windows\System32\vboxogl.dll (VirtualBox)
- C:\Windows\System32\vboxoglarrayspu.dll (VirtualBox)
- C:\Windows\System32\vboxoglerrorspu.dll (VirtualBox)
- C:\Windows\System32\vboxoglfeedbackspu.dll (VirtualBox)
- C:\Windows\System32\vboxoglpackspu.dll (VirtualBox)
- C:\Windows\System32\vboxoglpassthroughspu.dll (VirtualBox)
- C:\Windows\System32\vboxservice.exe (VirtualBox)
- C:\Windows\System32\vboxtray.exe (VirtualBox)
- C:\Windows\System32\VBoxControl.exe (VirtualBox)
- dbghelp.dll (Sandboxie)
- sbiedll.dll (Sandboxie)

- Running Services

이번에는 현재 실행중인 서비스들로 탐지하는 방법이다. 물론 이러한 프로세스들이 현재 실행 중이면 VM 으로 판별할 수 있다. 아래는 대표적 VM 관련 서비스들이다.

- VMTools (VMware)
- Vmxnet (VMware)

- Vmvss (VMware)
- Vmscsi (VMware)
- Vmhgfs (VMware)
- Vmmouse (VMware)
- VMMEMCTL (VMware)
- Vmrawdsk (VMware)
- Vmx_svga (VMware)
- Vmware Tools (VMware)
- Vmware Physical Disk Helper Service (VMware)

- Registry

VM 위에서는 레지스트리도 Host OS 와 차이가 있다. 또한 VM 관련 정보가 레지스트리에 일부 저장되어 있으니 값을 확인해 VM 판별이 또한 가능하다.

- SYSTEM\CurrentControlSet\Services\disk\Enum – 0 (VMware) [Hard Disk]

해당 필드에 VMware 가 있으면 VMware.

- SYSTEM\CurrentControlSet\Enum\IDE\DiskVMware_Virtual_IDE_Hard_Drive_____________00000001 (VMware) [Hard Disk Driver]

VMware 는 DiskVMware_Virtual_IDE_Hard_Drive_____________00000001 꼴의 경로를 가짐.

- SYSTEM\CurrentControlSet\Control\Class\{4D36E968~}\0000 - DriverDesc (VMware) [Video Driver]

VMware 는 vmx_svga 라는 비디오 드라이버를 사용. 해당 필드에 VMware 문자열이 있는지 확인. VMware SVGA 3D 이런 형식의 값이 들어있음.

- SYSTEM\CurrentControlSet\Control\VirtualDeviceDrivers (VMware)

해당 레지스트리는 Windows 7 에는 존재하고 Windows 10 에는 존재하지 않는다.

- SYSTEM\CurrentControlSet\Control\CriticalDeviceDatabase\root#vmwvmcihostdev – ClassGUID or Service (VMware)

해당 레지스트리는 Windows 7 에 존재하는 것으로 Windows 10 부터는 CriticalDeviceDatabase 자체가 없다. Service 에는 vmci 라는 값이 들어있다.

- SYSTEM\CurrentControlSet\Enum\SCSI\Disk&Ven_VMware_&Prod_Vmware_Virtual_S – [5&22be343f&0&000000] (VMware)

VMware 인 경우엔 SCSI 아래 저런 경로가 있지만 로컬머신에선 SCSI 자체도 존재하지 않는다.

- HARDWARE\DEVICEMAP\Scsi\Scsi Port 2\Scsi Bus 0\Target Id 0\Logical Unit Id 0 - Identifier (VMware)

VMware 는 VMware, Vmware Virtual S1.0 같은 이름을 가진다. 또 윈도우 7 같은 경우엔 Scsi Port 2 지만, 윈도우 10 인 경우엔 Scsi Port 0 이다.

- HKLM\SOFTWARE\Vmware Inc.\\\Vmware Tools (VMware)

- Invalid API Parameters

대부분의 API 함수들은 유효하지 않은 인자를 넘겨 받게 되면 에러 코드를 반환할 거다. 하지만 샌드박스 같은 안티 멀웨어 에뮬레이터에서는 에러 코드를 반환하지 않는 점이 있다.

- Modern CPU Instructions

예를 들어 인텔 인스트럭션 셋들 중 MMX, FPU, SSE 같은 명령어 셋들을 VM 에서 지원하지 않을 수도 있다. 결국 이러한 코드를 실행시킬 수 없거나 잘못된 인스트럭션 해석으로 오류가 나게 될 것이다.

- Undocumented Instructions

문서화 되지 않은 명령어들을 VM 에서 지원이 되지 않았을 수도 있다. 즉 만약 실행이 되면 아무 작동을 하지 않거나 예외가 발생하거나 둘 중에 하나일 것이다.

아래는 그러한 인스트럭션들이다.

```
db 0xf1; // icebp
db 0xd6; // setalc // salc
db 0xc0, 0xf0 // sal
db 0xf7, 0xc8 // test
db 0xf, 0x20 // mov eax, cr2
```

- Time Locks

대부분 멀웨어 자동화 검사 시스템(ex) malwares.com, virustotal.com)들은 time out 이나 실행할 수 있는 인스트럭션들에 제한이 있다. 이를 이용해서 예를 들어 time out 이 max 값이 30 초라면 프로그램을 30 초간 sleep 등으로 멈춰 있다 30 초가 지난 후에 작동을 하게 한다. 하지만 몇몇 에뮬레이터에서 이 과정을 자동으로 넘어가지게

하는 경우가 있다. 즉, 어떤 무의미한 작업을 오래하는 과정의 시간을 측정해 두었다 이게 지나치게 짧은 시간이 걸렸으면 이 과정을 지나쳤다고 판단할 수 있다.

구현한 코드는 아래와 같다.

```
void main() {
	__asm {
		rdtsc
		xchg ebx, eax
		mov ecx, 0x121212 // set loop cnt
		up:
		loop up
		rdtsc
		sub eax, ebx
		cmp eax, 0x1000 // must over 0x1000
		jbe emul
			push 1
			call exit
		emul:
	}

	printf("Emulator");
}
```

- ETC

위에서 Red Pill, No Pill 등 에서 소개한 명령어 이외에도 MM7, FST 등의 레지스터를 사용하는 방법도 있고 시간 기반으로 탐지하는 방법 등 여러가지 기법들이 있다.

2.7. Based on Timing

A) RDTSC

동적 분석을 할 때 주로 여러 곳에 breakpoint 를 세팅해 놓고 분석을 하게 된다. 만약 시간을 측정하는 명령인 rdtsc 와 rdtsc 사이에 있는 코드에 breakpoint 를 설치하고 값을 보고 넘어간다 했을 때 아무리 빨라도 아무런 방해 없이 실행될 때와 0.x 초의 차이는 있을 것이다. 이 때 시간 차이가 나는 걸 이용한 기법이 시간 기반 탐지 기법이다. RDTSC(ReaD Time Stamp Counter)명령어 말고도 API 함수로는 GetTickCount, QueryPerformanceTime 등의 함수를 사용할 수도 있고, 독특하게 CreateTransaction 함수를 사용해서도 구현을 해 볼 수 있다.

구현한 코드는 아래와 같다.

```
void count() {
	for (int i = 0; i < 10000; ++i);
}

void main() {
	UINT start, end;

	__asm {
		rdtsc
		mov start, eax
		call count // do something! loop 0 ~ 9999
		rdtsc
		mov end, eax
		mov eax, end
		sub eax, start
		cmp eax, 0x1000 // time
		jbe notdebug
		push 1
		call exit
		notdebug:
			mov start, eax
	}

	printf("diff %lu", start);
}
```

2.8. Based on Checksums

A) Hash Checking

이런 점을 이용해 Hash Checking 기법은 함수 코드들의 Hash 값을 미리 구해 둔 다음에 그 값과 실행 중간에 확인 한 Hash 값과 같은 지 확인 해 패칭이 됐는지 탐지 할 수 있다.

구현한 코드는 아래와 같다.

```
void func(void) {
	printf("Hello!");
}

void main() {
	DWORD checksum = 0x5792b8ac; // pre-calculated

	__asm {
		pushad
		mov ecx, offset main
		mov esi, offset func
		sub ecx, esi // func size - loop cnt
		xor eax, eax // checksum
		xor ebx, ebx
		calc : // checksum calc algorithm
			movzx ebx, ds:[esi]
			add eax, ebx
			rol eax, 1
			inc esi
		loop calc

		cmp eax, checksum // compare
		jne debug
		push 1
		call exit
		debug :
			popad
	}

	printf("Debugged");
}
```

2.9. Etc

A) Stack Segment

SS(Stack Segment)를 push 하고 pop 하게 되면 디버거가 인스트럭션을 잘못 해석해 바로 다음 코드가 실행은 되지만 step over 되고 그 다음 코드로 옮겨진다.

구현한 코드는 아래와 같다.

```
void stepover() {
	__asm { mov eax, 0xdeadbeef }
}

void main() {
	__asm {
		push ss
		pop ss
		call stepover // This would be stepped over
		xor eax, eax
	}
}
```

B) TLS Callback

TLS(Thread Local Storage)란 EP(Entry Point)가 시작 되기 전에 실행되는 루틴이다. 비유하면 리눅스에 ctors 같은 존재다. 직접 PE 구조를 확인해 보면 TLS 기능을 넣으면 tls section 이 따로 생성된다. 이렇게 main 함수가 시작되기 전에 해당 callback 이 실행되는 점을 이용해 해당 함수 안에 안티 디버깅 기법들을 주로 넣어두고 사용한다. data_seg 는 .CRT$XL? 형식으로 ?에 B~Z 까지 넣어 사용 가능하고 A 는 사용이 불가하다는 것.

구현한 코드는 다음과 같다. 실행결과 main 에서는 볼 수 없었던 연산인 ++stat; 실행으로 0 이 아닌 1 이 출력될 거다.

```
int stat = 0;

#pragma comment(linker, "/INCLUDE:__tls_used")
void NTAPI tls(PVOID instance, DWORD reason, PVOID reserved) { ++stat; }

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK TLS = tls;
// PIMAGE_TLS_CALLBACK TLS[] = { tls, 0 };
#pragma data_seg()

void main() {
	printf("State : %d\n", stat);
```

}

C) CC Scanning

주로 디버깅 시에 breakpoint 를 설정해서 동적 분석을 진행한다. 이 때 breakpoint 를 설정을 하면 0xcc(int 3) 란 코드가 추가가 된다. 이 값을 지정해준 함수(아래에선 func)를 1 바이트씩 돌아다니며 검사를 하는 기법이다.

구현한 코드는 아래와 같다.

```
void func() {
	__asm { mov eax, 0xdeadbeef }
}

void main() {
	__asm {
		popad
		mov ecx, offset main
		mov esi, offset func
		sub ecx, esi // size of func()
		xor ebx, ebx
		scan :
			movzx ebx, ds : [esi] // one by one
			cmp ebx, 0xcc // compare with 0xcc(bp)
			je bpx
			inc esi
			cmp ecx, 0
			je end
		loop scan
		end :
			jmp nobpx
		bpx :
			rdtsc
			call eax // go somewhere
		nobpx :
			call func // no breakpoints are detected
			popad
	}
}
```

3. Anti-Disassembly

안티 디스어셈블리 파트에서는 코드를 난독화 하는 방법과 디버거 같은 분석툴이 제대로 분석하지 못하게 하는 여러 기술에 대해 알아볼 것이다.

3.1. Code Obfuscating

- Code Virtualizing

이번 문서에서는 코드 가상화에 대해서 설명하고 구현 과정을 설명하기엔 많아 자세한 설명은 다루지 않을 것이다.

코드 가상화란 주로 우리는 인텔 인스트럭션 셋에 주어진 대로 정의된 opcode 에 명령어들을 사용한다. 하지만 이 기법은 직접 자신이 opcode 나 register 를 가상화 시켜 새로운 인스트럭션 체계를 만드는 것이다. 예를 들어 opcode 0x1 를 mov 로 정해놓고 직접 이에 해당하는 코드를 짜는 거다.

아마 해당 기술에 대해선 다음 문서에서 정리해 다루게 될 예정이다.

- VFTables

VFTable(Virtual Function Table)이란 클래스에 선언 시 가상 함수 타입으로 선언하게 되면 vftable 이란 곳에 가상 함수들의 주소가 저장되는 곳이다. 가상 함수를 선언해 vftable 주소가 있는 __vfptr 를 참조 하는 방식으로 코드를 짜게 되면 일종의 코드 난독화가 가능하다. 구체적인 vftable 에 관한 설명은 class 에 대한 설명과 가상 함수에 관한 설명을 찾아보길 바란다.

아래는 PoC 코드이다.

```
class _vft1 {
public:
	virtual int add(int a, int b) {
		return a + b;
	}

	virtual int sub(int a, int b) {
		return a - b;
	}
};


void main() {
```

```
	_vft1 v;
	PVOID *vf = (PVOID *)(&v);
	PVOID *vft = (PVOID *)*(&vf[0]); // point __vfptr

	auto add = reinterpret_cast<int(__thiscall *)(PVOID, int, int)>(vft[0]);
	auto sub = reinterpret_cast<int(__thiscall *)(PVOID, int, int)>(vft[1]);

	printf("%d\n", add(&v, 2, 2));
	printf("%d\n", sub(&v, 3, 1));
}
```

- Dummy Codes

더미 코드란 결론적으로 아무런 역할을 하지 않는 코드 뭉치로, 코드 사이사이에 삽입해 코드 난독화를 시키거나 분석툴이 인스트럭션들을 잘못 해석하게 하는 용도로 사용된다.

아래는 간단한 안티 디버깅 기법과 더미 코드를 섞어 만들어 본 예제 코드이다. 코드에서 강조한 부분이 실제로 하는 역할이고 나머지 코드는 전부 hexray 방해 코드, 더미 코드들이다. 실제로 실행 해 보면 정상적으로 작동하고 여러 분석 툴에서는 분석에 실패할 것이다.

```
void main() {
	__asm {
		pushad
		xor ebx, ebx
		shr ebx, 4
		add esi, ebx
		mov eax, fs:[0x18] // TEB
		mov ecx, 0xdeadbeef
		cmp ecx, 0xdead
		jne jumping
		add esp, 0x100
		call esp // never gonna happen
		jumping:
			sub ecx, ebx
			push offset gogo + 10
			add ecx, esi
			lea edx, [esi+ecx]
			mov eax, ds:[eax + 0x30] // TEB->PEB
			retn
			shr edx, 5
			mov esi, edx
		gogo:
			add esp, 0xfff
			call esp
			call ebx // never gonna happen
			movzx eax, ds:[eax + 2] // jump to here, TEB->PEB->IsDebugged
			lea ebx, [eax + ecx]
			xor ecx, ecx
```

```
                test eax, eax // check IsDebugged
                push esi
                lea esi, [esp + 0xc]
                pop edi
                push eax
                jnz debug
                push 1
                jmp goout
                popad
        debug:
                rdtsc
                call eax // go somewhere
        goout:
                popad
    }

    printf("Not Debugged");
}
```

3.2. Packing

A) Benchmark

코드를 보호하는 방법으로 Packing 이란 방법이 한가지 더 있다. Packing 이란 실행파일을 압축 및 암호화 해서 저장하는 방법을 말한다. 물론 압축하고 암호화를 해 저장할 때 다른 섹션을 만들고 그 곳에 복호화하는 알고리즘을 저장해 두어 실행 시 암호화된 데이터를 복호화 후 실행해 실행에는 문제가 없다.

- UPX

일단 프리웨어고 가벼우면서 다양한 포맷을 지원하고 효율이 좋으며 많이 쓰이는 패커 중 하나이다. 패킹 시 UPX0 하고 UPX1 섹션이 생성되고 UPX0 에는 Section 및 기타 정보 초기화 관련 코드가 존재하고 UPX1 에는 복호화 루틴이 존재한다.

- ASPack/ASProtect

상용 프로그램이고, x86, x64 윈도우 실행파일 타입 둘 다 지원을 한다. 코드 압축률 등에서는 별로 좋진 못하지만 암호화는 잘 되어있다. Stolen Bytes 기법이 최초로 구현된 패커다.

- Themida

가장 강력한 패커 중 하나로 안티 디버깅, 안티 덤핑, 안티 VM 등 여러가지 기능을 지원하며 직접 커스텀 할 수도 있다. Section 들의 이름은 랜덤으로 생성되고 여러 상용 프로그램에서도 사용 중이다. 현재 카카오톡 같은 여러 프로그램들도 Themida 패커를 사용하고 있다.

3.3. Anti-Dumping

A) Erase PE Header

해당 기법은 메모리 상에 존재하는 PE Header 를 지워 dumping 을 하는 과정을 방해할 수 있다.

구현한 코드는 아래와 같다.

```
void main() {
	DWORD oProt = 0;
	PCHAR base = (PCHAR)GetModuleHandle(NULL); // base address

	VirtualProtect(base, 4096, // x86 page size
		PAGE_READWRITE, &oProt);

	memset(base, 0, 4096);
}
```

B) PE Header Modification

메모리 상에 올라와 있는 PE 정보들을 지우거나 수정해 dumping 을 방해하는 방법도 있지만, 직접 바이너리의 PE 구조를 수정해서 올리디버그 같이 strict 한 PE 검사 루틴을 가지고 있는 여러 툴들을 무력화 시킬 수 있다.

다음은 PE Header(IMAGE_OPTIONAL_HEADER)구조다.

- Magic: 0x010B (HDR32_MAGIC)
- MajorLinkerVersion: 0x0e
- MinorLinkerVersion: 0x00 -> 14.00
- SizeOfCode: 0x00001400
- SizeOfInitializedData: 0x00001200
- SizeOfUninitializedData: 0x00000000
- AddressOfEntryPoint: 0x00001470
- BaseOfCode: 0x00001000
- BaseOfData: 0x00003000
- ImageBase: 0x00400000

```
-      SectionAlignment:             0x00001000
-      FileAlignment:                0x00000200
-      MajorOperatingSystemVersion:  0x0006
-      MinorOperatingSystemVersion:  0x0000 -> 6.00
-      MajorImageVersion:            0x0000
-      MinorImageVersion:            0x0000 -> 0.00
-      MajorSubsystemVersion:        0x0006
-      MinorSubsystemVersion:        0x0000 -> 6.00
-      Win32VersionValue:            0x00000000
-      SizeOfImage:                  0x00007000
-      SizeOfHeaders:                0x00000400
-      CheckSum:                     0x00000000
-      Subsystem:                    0x0003 (WINDOWS_CUI)
-      DllCharacteristics:           0x8000
-      SizeOfStackReserve:           0x00100000
-      SizeOfStackCommit:            0x00001000
-      SizeOfHeapReserve:            0x00100000
-      SizeOfHeapCommit:             0x00001000
-      LoaderFlags:                  0x00000000
-      NumberOfRvaAndSizes:          0x00000010
```

- ImageBase

기본적으로 Win32 프로그램에선 ImageBase 값은 0x400000 으로 세팅이 된다. 하지만 이 값을 조작해서 일부 분석툴에서 로딩이 안되게 하거나 분석을 방해할 수 있다.

- EntryPoint RVA

PE Header 에 AddressOfEntryPoint 인 EntryPoint RVA 값을 0 으로 세팅을 하게 되면 파일에 맨 처음인 'MZ' 부터 실행될 거다. 즉, MZ 부터 코드가 실행되니 opcode 로 dec ebx, pop edx 부터 시작되는 셈이다. 물론 EntryPoint 가 0 인 프로그램은 이후에 다시 OEP 로 복귀하는 코드를 가져야 크래쉬가 나지 않을 것이다. 이런 기법은 주로 패커들에서 많이 볼 수 있는 기법이다.

- SizeOfImage

일부 분석툴(구버전 LordPE, 등)에서 PE 헤더에 SizeOfImage 가 원래 크기가 아니면 크래쉬가 나기도 했었다. 현재 SizeOfImage 값보다 약간 큰 값으로 수정해 툴에서 분석되는걸 방해 할 수 있다. 이 값은 PEB->PEB_LDR_DATA->InOrderModuleList 에 SizeOfImage 값을 수정해 변경이 가능하다.

```
- PEB_LDR_DATA Structure
- typedef struct _PEB_LDR_DATA {
-   ULONG                          Length;
-   UCHAR                          Initialized[4];
-   ULONG                          SsHandle;
-   mLIST                          InLoadOrderList;
-   mLIST                          InMemOrderList;
-   mLIST                          InInitOrderList;
-   UCHAR                          EntryInProgress;
- } PEB_LDR_DATA;
```

PoC 코드는 아래와 같다.

```
void main() {
	__asm {
		mov eax, fs:[0x30]		// PEB
		mov eax, [eax + 0xc]		// PEB_LDR_DATA
		mov eax, [eax + 0xc]		// InOrderModuleList
		add [eax + 0x20], 0x3000 // SizeOfImage
	}
}
```

- LoderFlags & NumberOfRvaAndSizes

구버전 올리디버그 같은 몇몇 디버거에서는 엄격한 PE 구조 검사 때문에 해당 값들을 수정해 파일을 열게 되면 프로그램이 뻗기도 했다. NumberOfRvaAndSizes 에는 뒤에 사용되는 데이터 테이블의 갯수를 저장하는 곳인데, 최대 0x10 개의 테이블을 가질 수 있다. 그런데 이 값을 0x10 이상으로 수정을 해도 실행시에는 아무 문제가 없다. 이를 이용해 0x10 이상의 임의의 값으로 바꿀 수도 있다.

C) Sections Modification

- SizeOfRawData

Section Table 에서 SizeOfRawData 값을 정상 값인 VirtualAddress 와 VirtualSIze 의 차가 아닌 비정상적으로 큰 값을 넣어서 여러 분석 툴에서 크래쉬를 발생 시킬 수 있다. 예를 들어 구 버전 IDA 에서는 section 크기를 잘못 인식하게 돼 엄청나게 크게 메모리를 할당하는 경우도 있었다.

- No Section Table

PE Header 에 SectionAlignment 값이 4000(0xfa0)보다 작아지면, PE Header 는 writeable, excutable 한 영역이 되고, Section Table 이 필수가 아니고 옵션이 되게 된다. 즉, 모든 Section 들이 없는 거랑 마찬가지가 된다.

D) Stolen Bytes – Asprotect

OEP 주변에 몇 개의 바이트들을(인스트럭션) 분리된 메모리에 따로 복사한 다음에 그 것을 실행하는 기법을 Stolen Bytes 라고 한다. 하지만 단순하게 아무데서나 인스트럭션들을 복사해 실행할 수는 없고 그 코드들의 사이즈를 미리 알고있어야 잘못된 인스트럭션들이 되지 않고 예외없이 제대로 실행이 될 거다. PoC 코드는 아래 Reference 에 참고 링크를 확인하기 바란다.

E) Nanomites – Armadillo

해당 기법은 Armadillo 패커에서 처음으로 구현된 기능으로 모든 점프구문(jmp, je, jz, 등)을 위치나 정보를 다른 부분에 저장해 놓고 int 3(0xcc)로 교체하는 기법이다.
복호화 과정에서는 SEH 를 하나 만들어 놔 0xcc 를 만날 때 마다, 예외가 발생할 때 마다 하나씩 복호화를 해 나가는 과정을 거친다. PoC 코드는 아래 Reference 에 참고 링크를 확인하기 바란다.

3.4. Etc

A) Fake Signatures

Fake Signature 란 PEID 같은 바이너리 분석 프로그램에서 실제 정보와는 다른 정보로 인식되게 아니면 인식되지 못하게 하는 기술을 말한다. 예를 들어 UPX 로 패킹되지 않은 어떤 파일에서 UPX 에서 쓰이는 시그니처 코드를 사용해 UPX packed 프로그램이 아니지만 UPX packed 프로그램으로 탐지가 될 수 있다.

아래는 구현한 코드이다.

```
void main() {

}
```

```
extern "C" __declspec(naked) void fake(void) {
	__asm {
		// UPX signature
		__emit 0x60
		__emit 0xbe
		__emit 0x0
		__emit 0x80
		__emit 0x40
		__emit 0x0
		__emit 0x8d
		__emit 0xbe
		__emit 0x0
		__emit 0x90
		__emit 0xff
		__emit 0xff
		__emit 0x57
		__emit 0x83
		__emit 0xcd
		__emit 0xff
		__emit 0xeb
		__emit 0x10
		__emit 0x90
		__emit 0x90
		__emit 0x90
		__emit 0x90
		__emit 0x90
		__emit 0x90
		__emit 0x8a
		__emit 0x6
		__emit 0x46
		__emit 0x88
		__emit 0x7
		__emit 0x47
		__emit 0x1
		__emit 0xdb
		__emit 0x75
		__emit 0x7
		__emit 0x8b
		__emit 0x1e
		__emit 0x83
		__emit 0xee
		__emit 0xfc
		__emit 0x11
		__emit 0xdb
		__emit 0x72
		__emit 0xed
		__emit 0xb8
		__emit 0x1
		__emit 0x0
		__emit 0x0
		__emit 0x0
		__emit 0x1
		__emit 0xdb
		__emit 0x75
```

```
            __emit 0x7
            __emit 0x8b
            __emit 0x1e
            __emit 0x83
            __emit 0xee
            __emit 0xfc
            __emit 0x11
            __emit 0xdb
            __emit 0x11
            __emit 0xc0
            __emit 0x1
            __emit 0xdb
            __emit 0x73
            __emit 0xef
            __emit 0x75
            __emit 0x9
            __emit 0x8b
            __emit 0x1e
            __emit 0x83
            __emit 0xee
            __emit 0xfc
            __emit 0x11
            __emit 0xdb
            __emit 0x73
            __emit 0xe4
            __emit 0x31
            __emit 0xc9
            __emit 0x83
            __emit 0xe8
            __emit 0x3
            __emit 0x72
            __emit 0xd
            __emit 0xc1
            __emit 0xe0
            __emit 0x8
            push eax
            popad

            call main
      }
}
```

해당 코드를 컴파일 할 때, EP 를 fake 함수로 설정하기 위해선 /ENTRY:fake 라는 옵션을 넣어주면 된다.

해당 프로그램을 PEID 로 열어 scan 을 해 보면 아래와 같은 결과가 나온다.

UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo

4. Conclusion

지금까지 여러 안티 디버깅, 어셈블리, 덤핑 등 여러 기법들을 알아보았다.

위에서 소개한 기법들에 대한 약간 비관적인 이야기를 하자면 대부분이 이미 오래전에 발견 됐거나 또한 이를 자동으로 우회해 주는 디버거들의 플러그인(ex) idastealth, ollyadvance)들도 많이 생긴 지 오래다. 그래서 안티 디버깅 기법 만으로는 확실히 효과를 보기는 어렵다. 그래서 패커에서 추가해 주는 안티 디버깅 기법은 따로 치면 주로 악성코드에서 사용되는 안티 디버깅 기법으로는 API 후킹을 방지하기 위한 Anti-Hook 을 사용하는 경우와 breakpoint 를 scanning 하는 코드들 정도 등 인거 같다.

또한 코드 가상화와 Anti Dumping, Packing, Anti Sandbox, VM 등의 코드들이 사용되는 추세이다. 실제로 멀웨어들을 분석하다 보면 위와 같은 기능들이 구현된 것들을 흔히 볼 수 있다.

또한 이번 문서에서는 윈도우를 중심으로 많이 다뤘는데 물론 다른 플랫폼에서도 다양한 안티 리버싱 기법들이 존재하고 재밌는 기법들도 많다.

5. Reference

- PEB Structure :
- https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx
- NtQueryInformationProcess :
- https://msdn.microsoft.com/en-us/library/windows/desktop/ms684280(v=vs.85).aspx
- OutputDebugString :
- https://msdn.microsoft.com/en-us/library/windows/desktop/aa363362(v=vs.85).aspx
- OutputDebugString – new :
- https://ntquery.wordpress.com/2015/09/07/windows-10-new-anti-debug-outputdebugstringw/#more-32
- BlockInput :
- http://ezbeat.tistory.com/357
- DeleteFiber :
- https://msdn.microsoft.com/en-us/library/windows/desktop/ms682556(v=vs.85).aspx
- Anti-Reversing Techniques :
- http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide
- http://pferrie.host22.com/papers/antidebug.pdf
- http://www.slideshare.net/tylerxshields/antidebugging-a-developers-view
- Anti-VM :
- http://blog.cyberbitsolutions.com/anti-vm-and-anti-sandbox-explained/
- Stolen Bytes & Nanomites :
- http://resources.infosecinstitute.com/anti-memory-dumping-techniques/