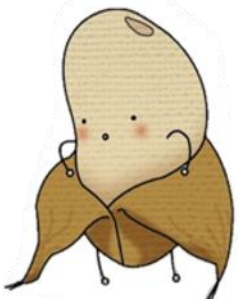
더미다 3.x 언패킹 및 tiger red 64 가상화 기법에 대한 분석보고서

TEAM 밥알까기



BoBalkkagi

1.서론

프로텍터는 실행 파일을 압축하거나 난독화하는 프로그램인 패커의 한 종류로, 실행 파일의 분석을 방해한다. 상용 프로텍터는 대표적으로 Themida와 VMProtect가 많이 사용되며 현재 우리나라에선 대표적으로 카카오톡, 금융보안 프로그램 등이 더미다를 사용하여 프로그램을 보호하는데 사용되고 있다. 그러나 분석가들의 분석을 방해하기 위해 악성코드, 게임 핵, 어뷰징 프로그램 역시 프로텍터를 악용하고 있다.

더미다에는 프로그램 분석을 방해하기 위해 패킹, Anti-Debugging, API Wrapping, 가상화 등의 기술을 사용하고 더미다로 보호된 프로그램을 분석하기 위해 분석방해 기술을 일일이 해제하면서 분석을 진행해야 되기 때문에 분석 시간이 늘어나 빠른 대처를 어렵게 한다. 따라서, 분석방해 기술의 작동원리를 파악하고 분석 방해 기술들을 패턴화하여 분석을 용이하게 돕는 도구의 개발이 필요하다.

본 프로젝트에서는 최신 버전의 더미다를 적용한 프로그램을 분석하고 더미다의 분석 방해 기법들의 특징과 기법을 확인할 수 있었다. 분석을 통해 얻은 특징을 이용해 더미다의 분석 방해 기술 Anti-Debugger Detection, Compress and Encrypt, API-Wrapping을 해제하는 도구를 개발하였으며 추후 더 많은 가상화 코드 분석을 통해 개발한 도구의 기능을 향상시키고 더미다 프로텍터를 악용하는 악성 프로그램의 분석을 용이하는데 기여하고자 한다.

본 문서에서 더미다가 적용된 프로그램을 분석하기 위해 사용한 도구는 다음 표와 같다.

	도구	설명
1	x64dbg	프로그램 동적분석, 디버깅
2	IDA	프로그램 정적분석, 디버깅
3	HyperHide	안티 디버깅 우회를 위한 x64dbg 플러그인
4	execution-trace-viewer	트레이스 로그 파일 뷰어
5	PPEE	PE 구조 분석

[표 1] 분석시 활용한 도구

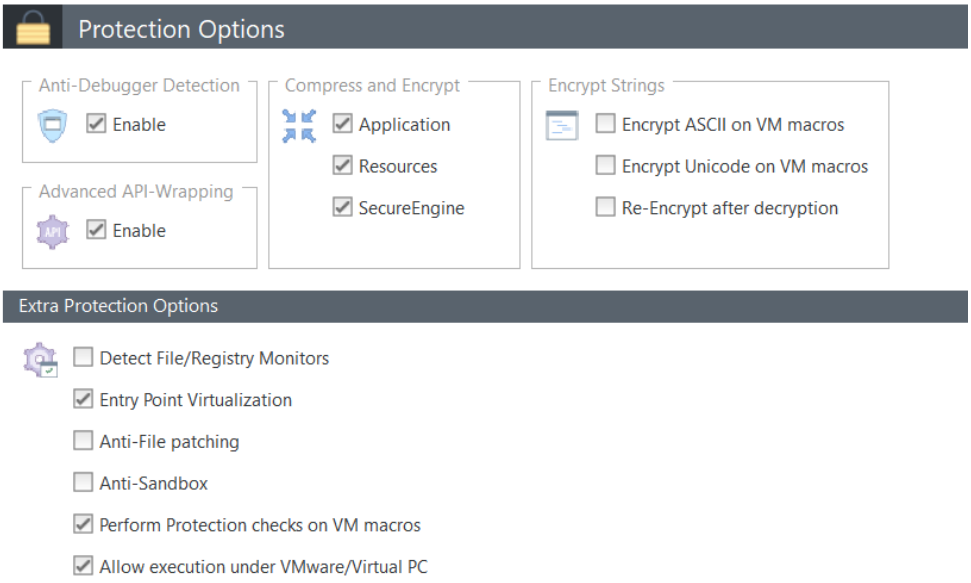
2. 더미다

Oreans Technologies의 소프트웨어 프로텍터로 현재 3.1.4.0 이 배포되고 있다. 더미다의 주요 기능으로는 Anti Debugger, API Wrapping, Code Virtualization 등이 존재한다.

[그림 1] Themida

더미다의 보호옵션들은 다음과 같은 기능들이 존재한다.

- Anti-Debugger Detection: 커널이나 소프트웨어 디버거가 디버깅하는 것을 탐지하는 기능
- Advanced API-Wrapping: 보호된 프로그램에서 사용하는 다양한 API를 식별하지 못하게 하는 기능
- Compress and Encrypt: 응용 프로그램, 리소스 및 보호 부트 섹션을 암호화하고 압축할지 여부를 선택하는 기능
- Encrypt Strings: 소스 코드에 가상 시스템 매크로를 삽입할 때(또는 외부 MAP 파일을 통해) 매크로 마커 내부에 나타나는 문자열에 대한 모든 참조를 암호화할 수 있는 기능(START-END 문자열을 암호화)
- Extra Protection Options
  - Detect File/Registry Monitor: Windows 파일 및 레지스트리 시스템 액세스를 모니터링하는 도구를 탐지하는 기능
  - Entry Point Virtualization: 프로그램에서 실행되는 첫 번째 명령에 VM 매크로를 넣는 것과 동등한 결과를 생성
  - Anti-File patching: 파일의 패치를 탐지
  - Anti-Sandbox: 샌드박스 어플리케이션을 탐지
  - Perform Protection checks on VM macros: 프로그램에 VM 매크로를 삽입하면 보호 매크로가 실행되기 전에 추가적인 보호 검사를 수행
  - Allow execution under VMware/Virtual PC: VMWare, Virtual PC, VirtualBox 등 가상환경에서 프로그램이 동작하는 것을 허가



[그림 2] Protection Options

더미다의 VM Macro의 종류는 각 동물명마다 (Black), (Red), (White) 로 나누어져 VM별로 복잡도, 속도, 크기가 표시되어 있고 [그림 3] Virtual Machine과 같다.

Used	Name	Instances	Complexity	Speed	Size
	DOLPHIN64 (Black)	1	5 %	72 %	3447
	DOLPHIN64 (Red)	1	38 %	74 %	2060
	DOLPHIN64 (White)	1	19 %	88 %	297
	EAGLE64 (Black)	1	96 %	1 %	1338
	EAGLE64 (Red)	1	93 %	1 %	1065
	EAGLE64 (White)	1	92 %	2 %	612
	FISH64 (Black)	1	35 %	4 %	3500
	FISH64 (Red)	1	20 %	85 %	320
	FISH64 (White)	1	10 %	90 %	150
	LION64 (Black)	1	60 %	62 %	3800
	LION64 (Red)	1	53 %	74 %	2275
	LION64 (White)	1	43 %	76 %	1560
	PUMA64 (Black)	1	89 %	3 %	3670
	PUMA64 (Red)	1	87 %	8 %	2160
	PUMA64 (White)	1	80 %	16 %	1755
	SHARK64 (Black)	1	93 %	1 %	3400
	SHARK64 (Red)	1	85 %	15 %	2360
	SHARK64 (White)	1	80 %	21 %	1160
	TIGER64 (Black)	1	23 %	91 %	3000
	TIGER64 (Red)	1	21 %	94 %	1900
	TIGER64 (White)	1	15 %	96 %	1000

[그림 3] Virtual Machine

본 보고서에서 더미다 프로텍터에서 Anti-Debugger Detection, Advanced API Wrapping, Entry Point Virtualization 옵션을 설정하고 VM을 TIGER64 Red로 설정하여 분석을 진행하였다.

3. Anti Debugging 기법

더미다는 Anti-Debugger Detection 옵션을 통해 디버깅 탐지 기능을 제공한다. 해당 기능들은 언패킹 루틴 진행중에 적용이 되며, 주로 디버깅 프로세스에서 자신이 디버깅 당하는지 여부를 파악하는 Static 기법이 사용된다.

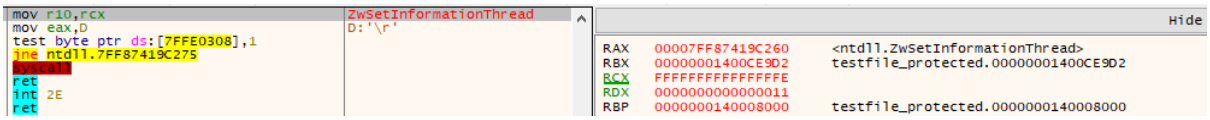
분석결과 더미다에 적용되는 Anti-Debugging 기법으론 NtSetInformationThread, NtQueryInformationProcess, NtUserGetForegroundWindow, RaiseException, VirtualProtect API들을 이용한 디버거 탐지가 있다.

- **NtSetInformationThread**

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationThread(
  [in] HANDLE          ThreadHandle,
  [in] THREADINFOCLASS ThreadInformationClass,
  [in] PVOID           ThreadInformation,
  [in] ULONG           ThreadInformationLength
);
```

[그림 4] NtSetInformationThread Syntax

해당 함수는 스레드의 우선순위를 설정하기 위해 제공되는 API이지만, 현재 Anti-Debugging의 목적으로 대부분 사용된다. 두번째 인자인 ThreadInformationClass에 ThreadHideFromDebugger(0x11) 값을 전달하게 되면 Handle에 해당하는 스레드를 디버거로부터 감추어 이 스레드와 관련된 이벤트를 수신하지 않게 된다.



[그림 5] NtSetInformationThread 동작

ThreadHandle 에는 현재 스레드 ID 값인 0xFFFFFFFFFFFFFFFE, ThreadInformationClass 에는 0x11 값을 전달하여 디버거를 탐지를 수행하고 있다. Class값이 0x11인 경우 0x0으로 변경하여 우회할 수 있다.

- **NtQueryInformationProcess**

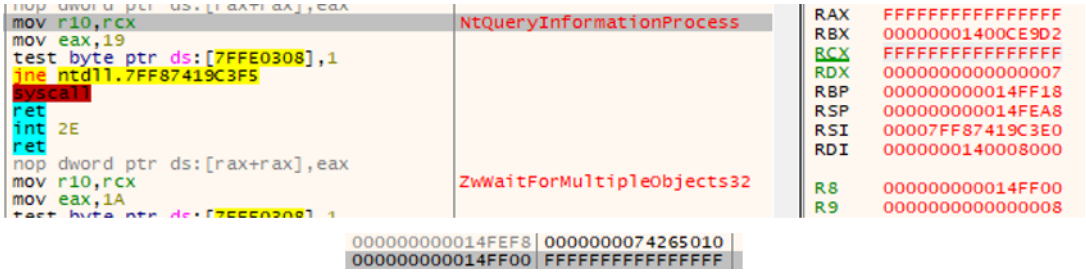
```
__kernel_entry NTSTATUS NtQueryInformationProcess(
  [in]            HANDLE           ProcessHandle,
  [in]            PROCESSINFOCLASS ProcessInformationClass,
  [out]           PVOID            ProcessInformation,
  [in]            ULONG            ProcessInformationLength,
  [out, optional] PULONG           ReturnLength
);
```

[그림 6] NtQueryInformationProcess Syntax

해당 함수는 프로세스의 다양한 정보를 가져올 때 사용하는 API다. 두번째 인자 ProcessInformationClass를 이용하여 총 3가지 방식으로 디버거 탐지가 가능하다.

- ProcessInformationClass = 0x7

ProcessInformationClass가 0x7로 설정되면 열린 디버깅 DebugPort정보를 가져온다는 점을 이용한 Anti-Debugging 방법이다. 디버깅 중이면 0xFFFFFFFFFFFFFFFE 값이 디버깅 중이 아니면 0x0 값이 ProcessInformation에 설정된다.

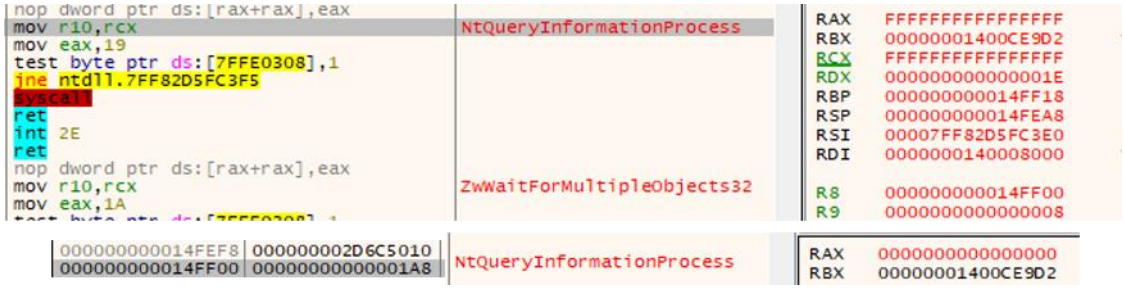


[그림 7] **NtQueryInformationProcess ProcessInformationClass = 0x7일때**

ProcessInformation값을 0x0으로 변경하여 Anti-Debugging을 우회할 수 있다.

- ProcessInformationClass = 0x1E

ProcessInformationClass가 0x1E(ProcessDebugObjectHandle)값으로 설정되면 ProcessInformation 파라미터에 Debug Object Handle Pointer가 담긴다. 디버깅 중이라면 해당 파라미터에 0이 아닌 값이 저장되고 Rax에 0x0을 반환한다.



[그림 8] NtQueryInformationProcess 0x1E 동작

ProcessInformation값을 0x0으로 바꾸고 반환 값(Rax)을 0xC0000353(STATUS_PROT_NOT_SET)으로 변경하면 Anti-Debugging을 우회할 수 있다.

- ProcessInformationClass = 0x1F

ProcessInformationClass가 0x1F(ProcessDebugFlags)값으로 설정되면 ProcessInformation 파라미터에 플래그 값을 전달한다. 디버깅 중이면 0x0 이 디버깅 중이 아니면 0x1이 설정된다.

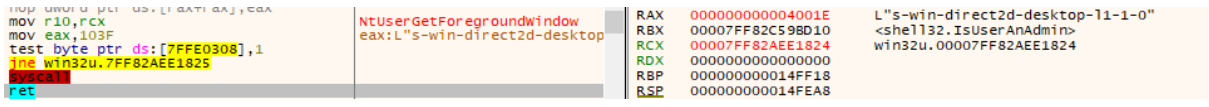
ProcessInformation 파라미터 값을 0x1으로 변경하면 Anti-Debugging을 우회할 수 있다.

● NtUserGetForegroundWindow

```
HWND GetForegroundWindow();
```

[그림 9] NtUserGetForegroundWindow Syntax

해당 함수는 전경창에 대한 핸들을 반환하는 API다. 디버깅 중이면 x64dbg와 같은 디버깅 툴의 전경 창 핸들을 반환하고 이를 통해 디버깅을 탐지한다.



[그림 10] NtUserGetForegroundWindow 동작

반환 값을 0x0으로 변경하면 Anti-Debugging을 우회할 수 있다.

- **VirtualProtect**

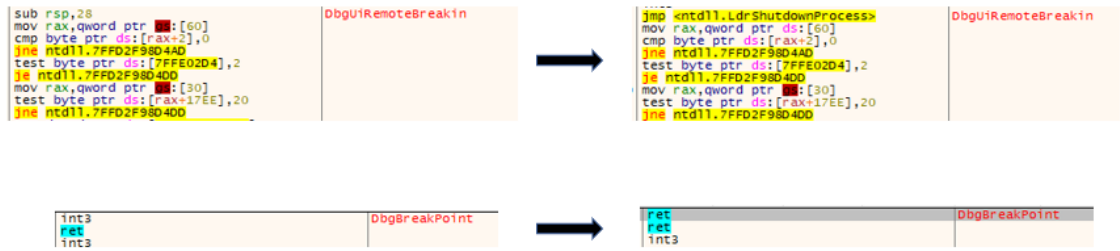
Address 메모리 공간의 권한을 변경하는 부분에서 DbgUiRemoteBreakin, DbgBreakPoint 함수에 Write 권한을 추가하여 특정 코드를 추가하고 있는데, 이는 Anti-Attach 기술로서 사용된다.

프로세스에 디버거를 붙이는 과정을 살펴보면 DebugActiveProcess – RtlCreateUserThread – DbgUiRemoteBreakin 흐름으로 진행된다. RtlCreateUserThread 함수는 Attach하고자 하는 프로세스에 새로운 스레드를 생성하고, 이 스레드가 실행하는 함수가 DbgUiRemoteBreakin 함수다.

[그림 11] DbgUiRemoteBreakin, DbgBreakPoint 함수

이 함수는 DbgBreakPoint함수를 통해 Attach한 프로세스에 Break Point를 걸고, 예외를 발생시켜 디버깅을 가능하게 만든다.

권한 변경 후 특정 코드가 추가된 이후의 함수를 살펴보면 [그림 12] 수정된 코드과 같다.s



[그림 12] 수정된 코드

DbgUiRemoteBreakin 함수의 경우 프로세스를 종료하는 LdrShutdownProcess함수로 점프하게 만들어 Attach할 프로세스에 새로운 스레드를 생성되어 이 함수가 실행되는 경우 프로세스를 종료하게 만들고, DbgBreakPont 함수의 경우 Software Break Point 발생을 제거하여 예외 발생으로 인한 디버깅을 불가능하게 만든다.

4. 언패킹 과정 분석

프로그램은 실행할 때 Entry Point(이하 EP)부터 코드를 실행한다. 하지만 프로그램에 프로텍터를 적용시키면 프로텍터가 원본 코드를 압축하여 실제 코드의 EP인 Original Entry Point (이하 OEP)를 확인할 수 없게 된다. 더미다의 경우 [그림 13] 더미다 적용 PE Section와 같이 .boot 섹션을 생성하고 [그림 14] 더미다 적용 Program EP처럼 EP가 .boot 섹션의 영역으로 바뀐다. 더미다가 적용된 보호 프로그램을 실행 시 .boot 섹션의 EP부터 코드가 실행되며 압축된 원본 코드를 메모리에 해제한다.

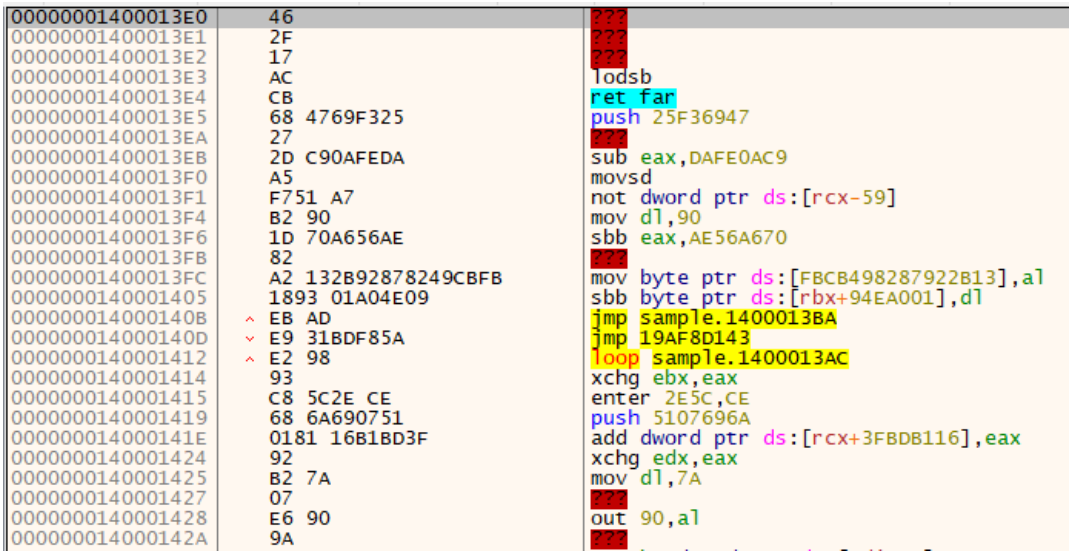
Name	VirtualAd...	VirtualSize	RawAddre...	RawSize	PtrToRelocs	PtrToLine...	NumOfRe...	NumbOfL...	Characteri...
	00001000	00000EAC	00000400	00000A00	00000000	00000000	0000	0000	60000020
	00002000	00001126	00000E00	00000400	00000000	00000000	0000	0000	40000040
	00004000	00000648	00001200	00000200	00000000	00000000	0000	0000	C0000040
	00005000	0000018C	00001400	00000200	00000000	00000000	0000	0000	40000040
	00006000	000001E0	00001600	00000200	00000000	00000000	0000	0000	40000040
	00007000	0000002C	00001800	00000200	00000000	00000000	0000	0000	42000040
.idata	00008000	00001000	00001A00	00000400	00000000	00000000	0000	0000	C0000040
.rsrc	00009000	00001000	00001E00	00000200	00000000	00000000	0000	0000	40000040
.themida	0000A000	00578000	00002000	00000000	00000000	00000000	0000	0000	E0000060
.boot	00582000	0032BC00	00002000	0032BC00	00000000	00000000	0000	0000	60000060

[그림 13] 더미다 적용 PE Section

Member	Value	Comment
Magic	020B	PE32+ (PE64)
MajorLinkerVersion	0E	
MinorLinkerVersion	1D	
SizeOfCode	00001000	
SizeOfInitializedData	00002000	
SizeOfUninitializedData	00000000	
AddressOfEntryPoint	00582058	.boot (Last section)
BaseOfCode	00001000	
BaseOfData	40000000	
ImageBase	0000000140000000	

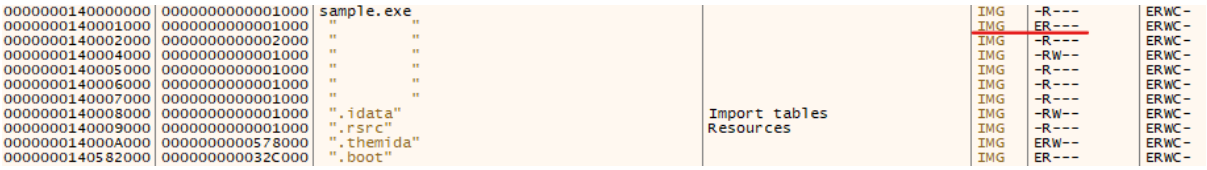
[그림 14] 더미다 적용 Program EP

보호된 프로그램의 동작을 분석하기 위해서는 실제 동작하는 코드의 시작 지점인, OEP를 찾아야 되는데 보호된 프로그램은 패킹되어 다음과 같이 해석하지 못한다.

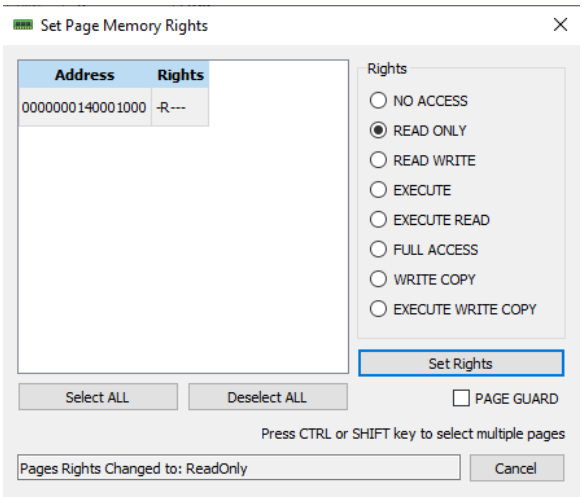


[그림 15] 실제 코드 영역

코드를 실행시키기 위해서 코드 영역에 실행권한이 필요하다는 것을 이용하여 쉽게 OEP를 찾을 수 있다. 실행 권한이 있는 영역을 찾아 실행 권한을 제거하면 패킹 해제가 끝난 후 코드를 실행시킬 때 그 영역에 실행 권한이 없기 때문에 OEP에서 에러가 발생한다.



[그림 16] 실행 권한 확인



[그림 17] 실행 권한 제거

```
EXCEPTION_DEBUG_INFO:
            dwFirstChance: 1
            ExceptionCode: C0000005 (EXCEPTION_ACCESS_VIOLATION)
           ExceptionFlags: 00000000
          ExceptionAddress: sample.0000000140013E0
          NumberParameters: 2
ExceptionInformation[00]: 0000000000000008 DEP Violation
ExceptionInformation[01]: sample.0000000140013E0 Inaccessible Address
First chance exception on 0000000140013E0 (C0000005, EXCEPTION_ACCESS_VIOLATION)!
```

[그림 18] OEP 0x1400014E0

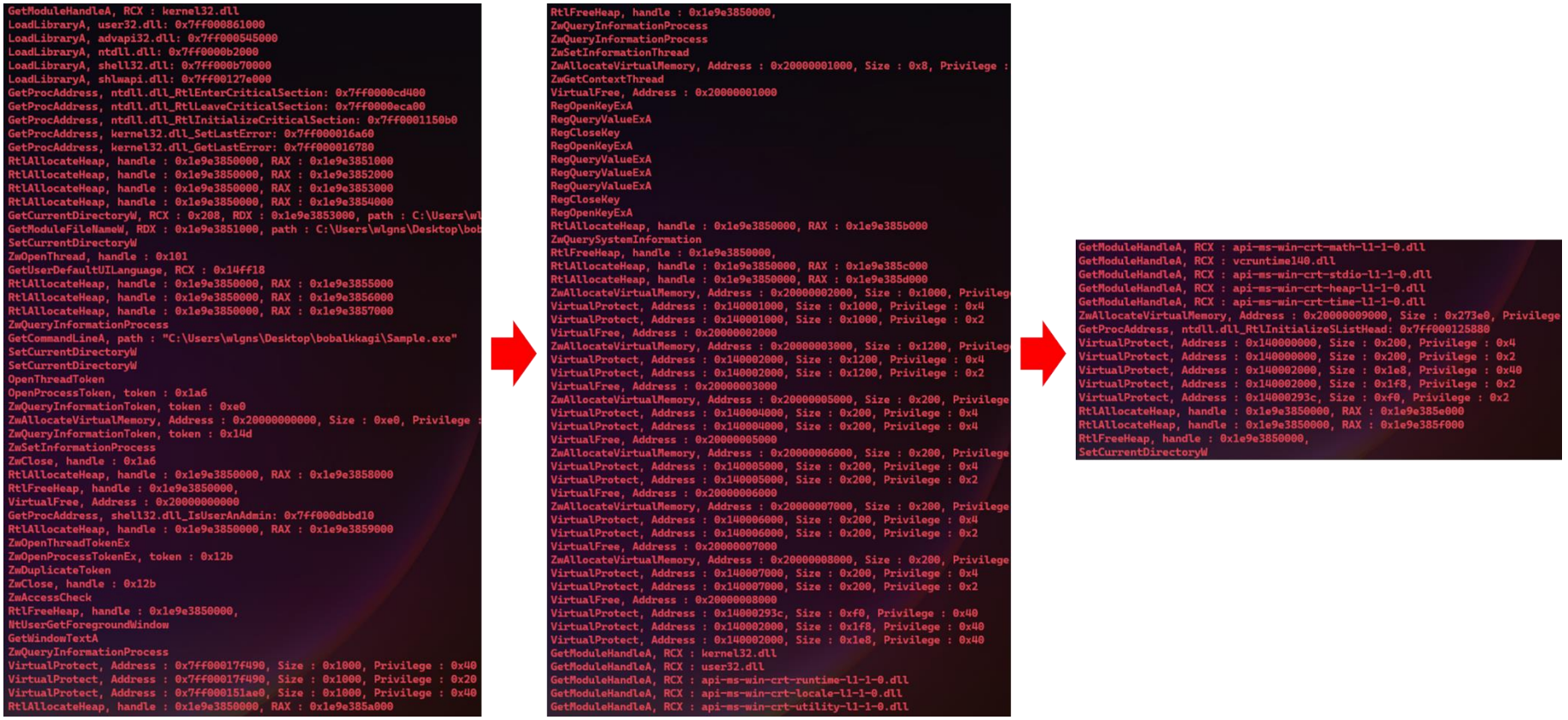
패킹이 해제되면 [그림 14] 더미다 적용 Program EP의 실제 코드 영역은 [그림 19] 패킹이 해제된 실제 코드 영역처럼 바뀌어 원본 프로그램의 동작을 수행한다.

```
0000000140013E0  E9 1CA25500      jmp sample.14055B601
0000000140013E5  50               push rax
0000000140013E6  A6               cmpsb
0000000140013E7  65:F9            stc
0000000140013E9  FE03             inc byte ptr ds:[rbx]
0000000140013EB  EE               out dx,al
0000000140013EC  C2 0516          ret 1605
0000000140013EF  A9 1A63CCCC      test eax,CCCC631A
0000000140013F4  40:53            push rbx
0000000140013F6  48:83EC 20       sub rsp,20
0000000140013FA  48:8BD9          mov rbx,rcx
0000000140013FD  33C9             xor ecx,ecx
0000000140013FF  FF15 4B0C0000    call qword ptr ds:[140002050]
0000000140001405  48:8BCB         mov rcx,rbx
0000000140001408  FF15 4A0C0000   call qword ptr ds:[140002058]
000000014000140E  FF15 EC0B0000   call qword ptr ds:[140002000]
0000000140001414  48:8BC8         mov rcx,rax
0000000140001417  BA 090400C0     mov edx,C0000409
000000014000141C  48:83C4 20      add rsp,20
0000000140001420  5B              pop rbx
0000000140001421  48:FF25 200C0000  jmp qword ptr ds:[140002048]
0000000140001428  48:894C24 08    mov qword ptr ss:[rsp+8],rcx
000000014000142D  48:83EC 38      sub rsp,38
0000000140001431  B9 17000000     mov ecx,17
0000000140001436  FF15 040C0000   call qword ptr ds:[140002040]
000000014000143C  85C0            test eax,eax
000000014000143E  74 07           je sample.140001447
0000000140001440  B9 02000000     mov ecx,2
```

[그림 19] 패킹이 해제된 실제 코드 영역

개발한 도구[1]를 이용하여 언패킹 시 호출과정을 출력한 결과 필요 라이브러리를 로드, 디버거 탐지 후 섹션 영역에 쓰기 권한을 부여하고 제거하는 과정을 거치며 원본 코드를 복원함을 확인할 수 있었고 전체 과정은 [그림 20] 언패킹 API Call Trace과 같다.

[1] https://github.com/BoB11-TEAM-bobalkkagi/bobalkkagi



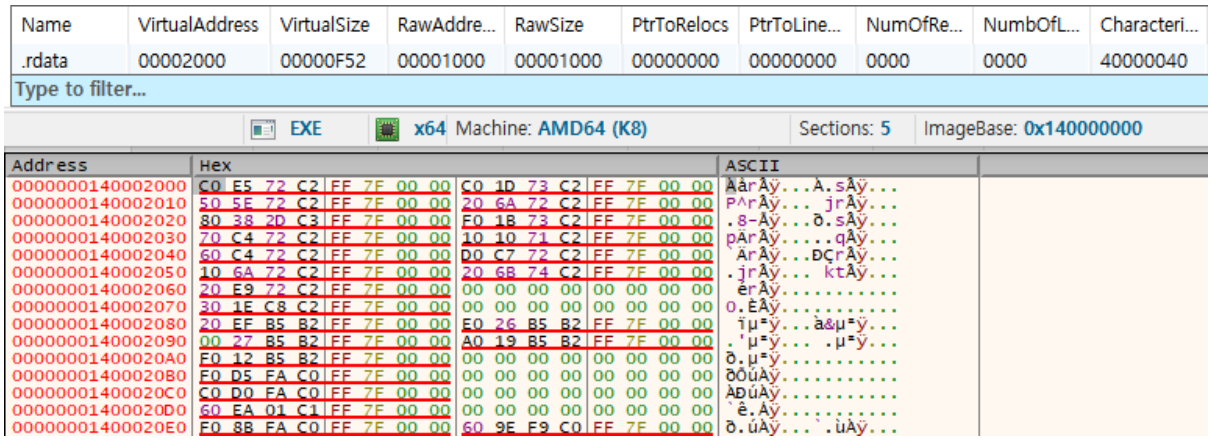
[그림 20] 언패킹 API Call Trace

언패킹 과정 동안 디버깅 탐지, 필요한 API를 가지고 있는 DLL들을 호출하고 섹션의 권한을 바꾸며 원본 코드 영역에 동작하는 코드가 복구됨을 알 수 있다.

4. API Wrapping 분석

4.1. PE 프로그램에서 API 호출 절차

PE 프로그램에서 .rdata섹션의 IAT로 할당된 메모리에 저장되는 API주소를 찾아 VirtualAddress 위치에 API주소가 저장된다. 다음 [그림 21] .rdata 섹션과 IAT 확인과 같이 .rdata 섹션의 정보를 확인하여 IAT를 확인할 수 있다.



Name	VirtualAddress	VirtualSize	RawAddre...	RawSize	PtrToRelocs	PtrToLine...	NumOfRe...	NumbOfL...	Characteri...
.rdata	00002000	00000F52	00001000	00001000	00000000	00000000	0000	0000	40000040

Type to filter...

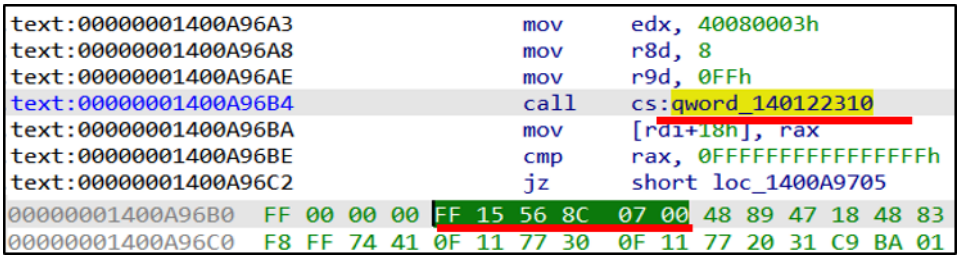
EXE | x64 Machine: AMD64 (K8) | Sections: 5 | ImageBase: 0x140000000

Address	Hex	ASCII
0000000140002000	C0 E5 72 C2 FF 7F 00 00 C0 1D 73 C2 FF 7F 00 00	ÀårÂÿ...À.sÂÿ...
0000000140002010	50 5E 72 C2 FF 7F 00 00 20 6A 72 C2 FF 7F 00 00	P^rÂÿ... jrÂÿ...
0000000140002020	80 38 2D C3 FF 7F 00 00 F0 1B 73 C2 FF 7F 00 00	.8-Ãÿ...ð.sÂÿ...
0000000140002030	70 C4 72 C2 FF 7F 00 00 10 10 71 C2 FF 7F 00 00	pÄrÂÿ.....qÂÿ...
0000000140002040	60 C4 72 C2 FF 7F 00 00 D0 C7 72 C2 FF 7F 00 00	`ÄrÂÿ...ÐÇrÂÿ...
0000000140002050	10 6A 72 C2 FF 7F 00 00 20 6B 74 C2 FF 7F 00 00	.jrÂÿ... ktÂÿ...
0000000140002060	20 E9 72 C2 FF 7F 00 00 00 00 00 00 00 00 00 00	érÂÿ..........
0000000140002070	30 1E C8 C2 FF 7F 00 00 00 00 00 00 00 00 00 00	0.ÈÂÿ..........
0000000140002080	20 EF B5 B2 FF 7F 00 00 E0 26 B5 B2 FF 7F 00 00	ïµ²ÿ...à&µ²ÿ...
0000000140002090	00 27 B5 B2 FF 7F 00 00 A0 19 B5 B2 FF 7F 00 00	.'µ²ÿ... .µ²ÿ...
00000001400020A0	F0 12 B5 B2 FF 7F 00 00 00 00 00 00 00 00 00 00	ð.µ²ÿ..........
00000001400020B0	F0 D5 FA C0 FF 7F 00 00 00 00 00 00 00 00 00 00	ðÕúÀÿ..........
00000001400020C0	C0 D0 FA C0 FF 7F 00 00 00 00 00 00 00 00 00 00	ÀÐúÀÿ..........
00000001400020D0	60 EA 01 C1 FF 7F 00 00 00 00 00 00 00 00 00 00	`ê.Áÿ..........
00000001400020E0	F0 8B FA C0 FF 7F 00 00 60 9E F9 C0 FF 7F 00 00	ð.úÀÿ...`.ùÀÿ...

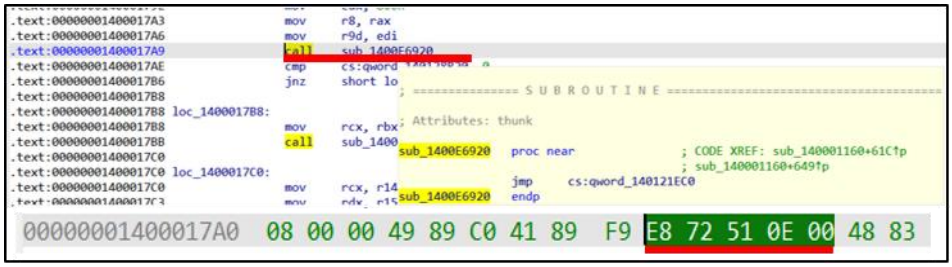
[그림 21] .rdata 섹션과 IAT 확인

4.2. API Wrapping 패턴

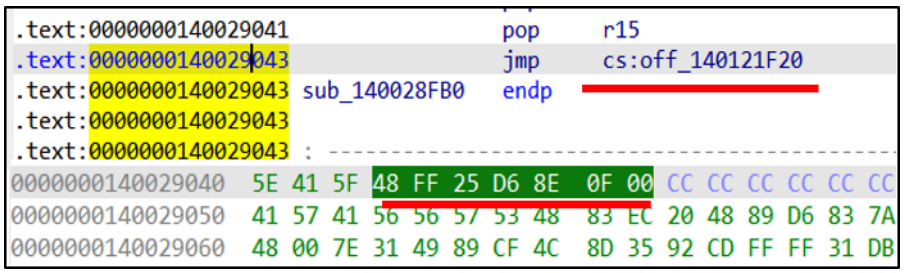
더미다에서 API Wrapping 옵션을 적용하게 되면 API를 호출할 때 바로 API 주소로 호출하는 것이 아니라 .themida 섹션의 Wrapping된 루틴을 호출한다. 이때 래핑을 하는 패턴에 따라 API 호출하는 opcode가 달라진다.



[그림 22] FF15 Call Wrapping 패턴



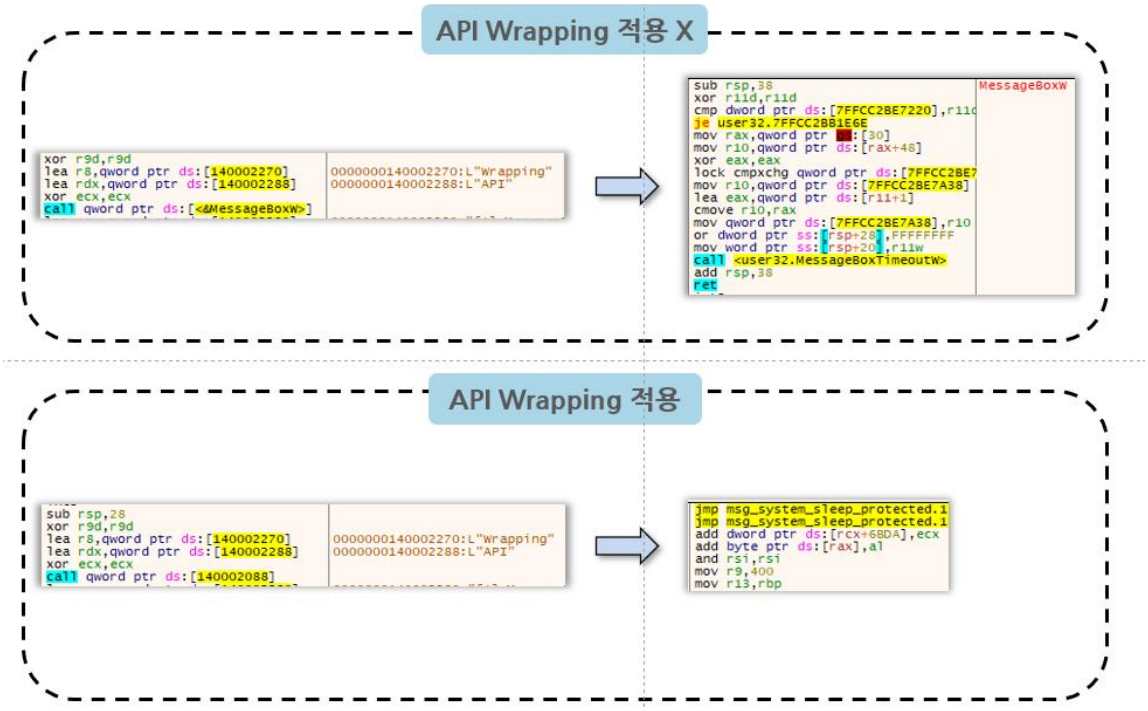
[그림 23] E8 Call Wrapping 패턴



[그림 24] jmp Wrapping 패턴

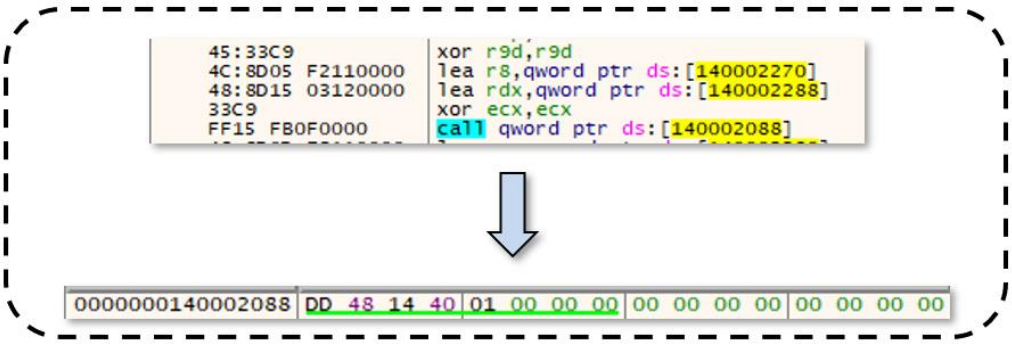
4.3. Wrapped API 호출 과정

API Wrapping 옵션을 적용하게 되면 API 함수 호출 시 일반적인 프로그램에서는 해당 함수 주소를 바로 호출하지만, 옵션 적용시 Wrapping 루틴을 호출 하는 것을 확인할 수 있다.



[그림 25] API Wrapping 옵션 적용 전 후 비교

아래와 같이 함수를 호출할 때 0x1401448DD라는 .themdia 섹션의 주소를 참조하는 것을 확인할 수 있다. 해당 주소는 Wrapping 루틴을 시작하는 주소이다.

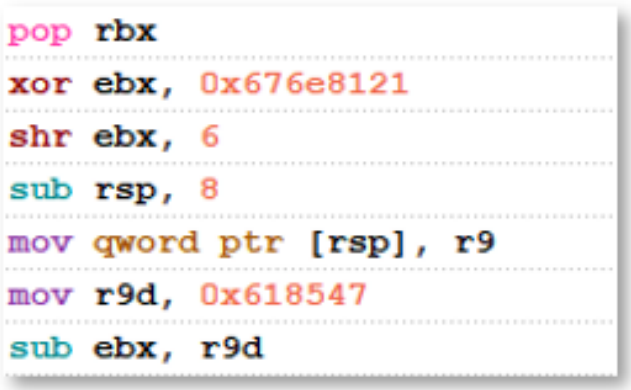


[그림 26] Wrapping 루틴 호출

Wrapping 루틴을 디버거를 이용해 추적하여 확인해보면 다음과 같이 jmp명령으로 실행 흐름이 이어지는 것을 확인할 수 있다.

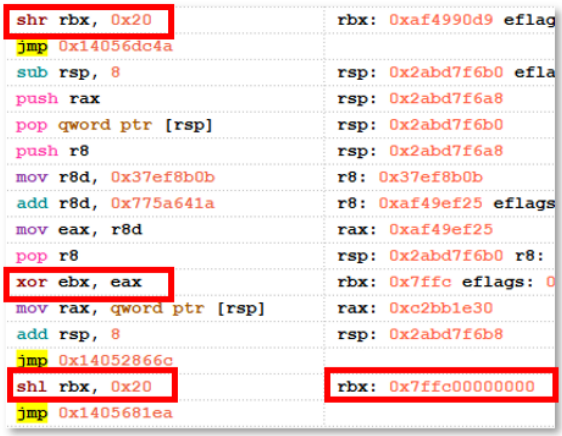
[그림 27] Wrapping 루틴 1

[그림 27] Wrapping 루틴 1을 보면 jmp명령어로 실행 흐름이 이어지며, 스택에 특정 값들을 push하는 것을 확인할 수 있다. 또한 다음과 같이 스택에 있는 값들을 사용하여 shr, add, sub, shl 등의 연산을 진행한다.



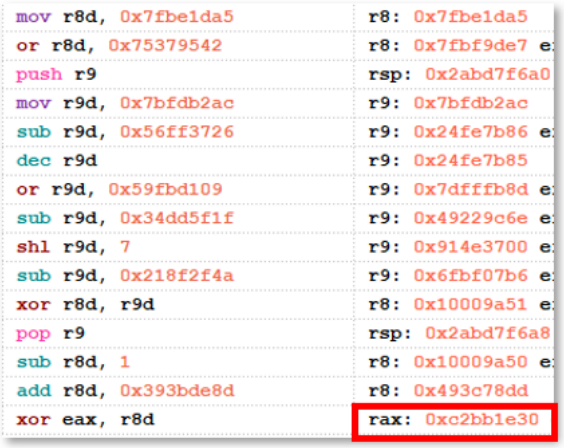
[그림 28] Wrapping 루틴 2

다음과 같이 shr연산, xor연산 후 shl연산을 통해 dll의 base 주소를 구하는 것을 확인할 수 있다. 해당 연산 후 rbx값은 0x7ffc00000000 이 된다.



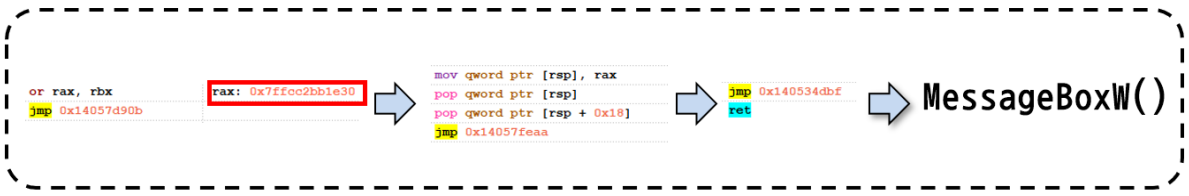
[그림 29] Wrapping 루틴 3

다음과 같이 or, sub, shift, add, xor등의 연산을 통해 rax값에 MessageBoxW함수의 오프셋인 가 들어가는 것을 확인할 수 있다.



[그림 30] Wrapping 루틴 4

최종적으로 "or rax, rbx" 명령을 통해 rax레지스터에 원본 API 주소가 들어가며 이후 해당 값을 스택의 복귀 주소에 저장한다. 이후 Wrapping 루틴이 끝나 ret 명령어 실행 시 Wrapping된 함수인 MessageBoxW 함수로 복귀하는 것을 확인할 수 있다.

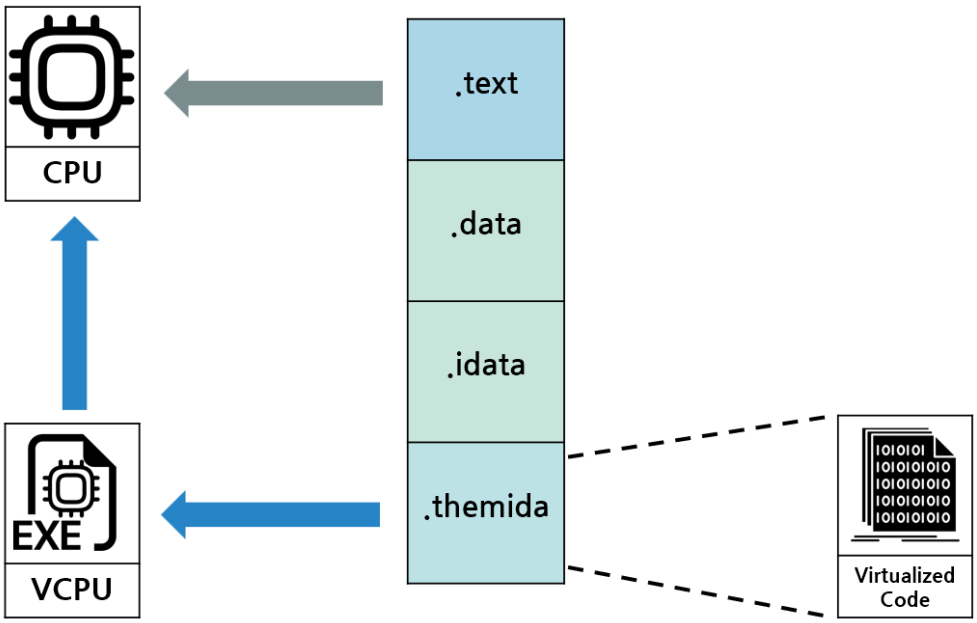


[그림 31] Wrapping 루틴 5

5.코드 가상화 분석

5.1. 코드 가상화 기법

코드 가상화는 Anti-Reversing 기법 중 하나로 프로그램 내에 가상 CPU를 구현하여 가상화 코드를 실행시키는 기법이다. 가상화 코드는 프로그램을 실행하는 CPU는 해석하지 못하며 프로그램 내에 구현된 가상 CPU에서 실행된다. 가상 CPU는 프로그램 내에 존재한다.

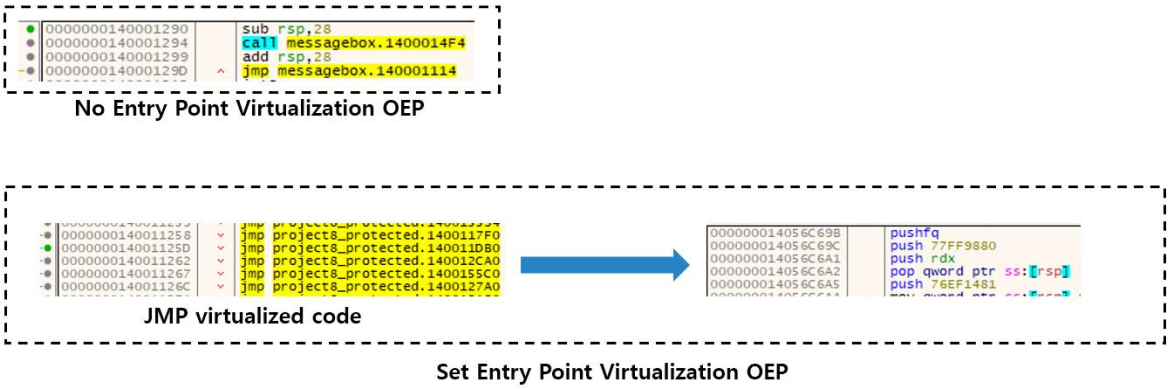


[그림 32] 가상화 코드 동작 방식

가상 CPU에서 가상화 코드를 어떻게 처리하는지 파악하기 힘들어 코드 가상화가 적용된 프로그램은 리버싱으로 분석이 어렵다. 그러나 많은 양의 반복 횟수를 처리하는 코드 등의 복잡한 알고리즘을 가상화 하면 가상화 코드를 해석하는 코드의 길이가 매우 길어져 프로그램의 크기가 매우 증가하게 된다. 따라서 코드 가상화를 적용할 때는 보호해야 할 핵심 알고리즘만 지정하여 적용하는 것이 일반적이다.

5.2. 더미다의 코드 가상화 기법

더미다에서 코드 가상화를 적용하는 옵션으로는 Entry Point Virtualization과 VM 매크로가 있다. Entry Point Virtualization은 PE 파일의 시작 지점인 EP를 가상화 하는 기법이다. 해당 기법을 적용하게 되면 [그림 33] Entry Point Virtualization 적용 유무 옵션 비교과 같이 EP의 명령어가 가상화 코드 영역으로 이동하는 JMP 명령임을 확인할 수 있다.



[그림 33] Entry Point Virtualization 적용 유무 옵션 비교

VM 매크로 옵션을 적용하게 되면 소스 코드에서 가상화 하여 보호할 코드의 영역을 지정하여 가상화 할 수 있으며 [그림 3] Virtual Machine과 같이 속도, 복잡도 등 다양한 종류의 가상화 옵션을 적용할 수 있다.

우리가 분석한 옵션은 기본 옵션으로 지정된 TIGER64 Red 옵션이다. 소스 코드에 가상화를 적용하기 위해 다음과 같이 VM_START와 VM_END 키워드로 가상화 할 코드의 영역을 지정한다.

```
void _messageBox(LPCWSTR msg, LPCWSTR title) {
    VM_START;
    MessageBox(NULL, msg, title, MB_OK);
    VM_END;
}
```

[그림 34] 더미다에서 부분 가상화 적용

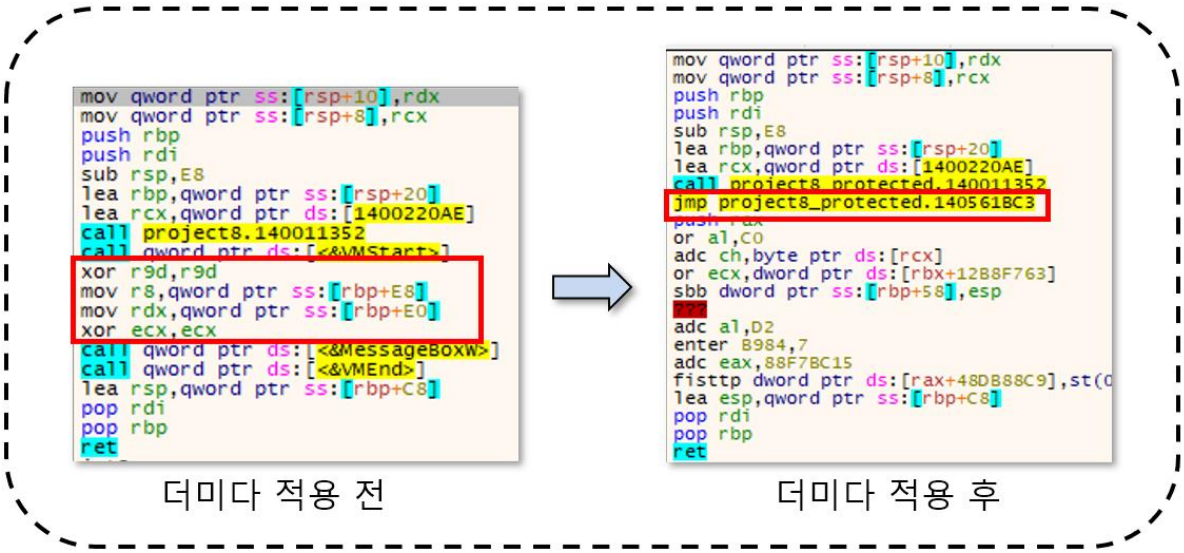
5.3. 가상화 코드 분석

다음과 같이 MessageBox 함수에 부분 가상화를 적용한 소스코드를 컴파일 하여 더미다를 적용하여 가상화 코드를 분석하였다.

```
1  #include <stdio.h>
2  #include <windows.h>
3  #include <tchar.h>
4  #include <ThemidaSDK.h>
5
6  void _messageBox(LPCWSTR* msg, LPCWSTR* title);
7
8  int main(int argc, char* argv[])
9  {
10     LPCWSTR msg = L"hello world!";
11     LPCWSTR title = L"title";
12     _messageBox(msg, title);
13     return 0;
14 }
15
16
17 void _messageBox(LPCWSTR* msg, LPCWSTR* title) {
18     VM_START;
19     MessageBox(NULL, msg, title, MB_OK);
20     VM_END;
21 }
```

[그림 35] 분석 대상 프로그램 소스코드

더미다를 적용하지 않은 프로그램에서 MessageBox 함수를 호출하는 _messagebox 함수와 더미다를 적용한 프로그램의 함수를 디버거로 비교하면 다음과 같다.



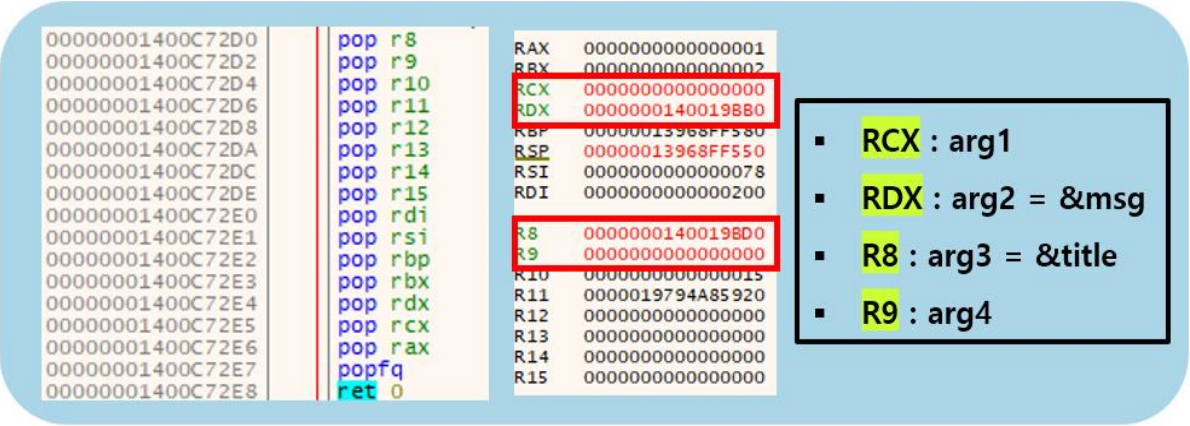
[그림 36] 더미다 적용 전 후 가상화 함수 비교

더미다를 적용하지 않은 프로그램은 위와 같이 함수 내에서 호출하는 MessageBox함수의 인자를 설정해주고, MessageBox함수를 호출하는 것을 확인할 수 있다. 반면 더미다를 적용한 프로그램은 MesageBox 함수 인자를 설정하는 과정과 MessageBox를 호출하는 코드를 확인할 수 없다.

더미다가 적용된 프로그램에서 jmp 명령어를 실행하게 되면 가상화 코드로 진입하게 된다.
예제로 사용한 MessageBox의 인자를 설정하는 코드가 6만줄가량의 어셈블리 코드로 변환되었고 해당 어셈블리 코드를 분석하여 가상화 코드의 행위와 가상화 코드가 갖는 패턴들을 분석하였다. 어셈블리 코드 분석을 보다 쉽게 하기 위해 가상화 코드를 디버거를 이용해 추적하고, 오픈 소스 execution-trace-viewer[2]를 활용하여 분석하였다.

[2] https://github.com/teemu-l/execution-trace-viewer 참고

가상화 코드가 시작되면 가상화 코드 내에서 동작하는 연산을 위해 레지스터가 사용되기 때문에 시작 전 레지스터 값들을 스택에 저장하는 구간이 있다. 이 후 가상화 코드가 종료될 때 pop 명령을 통해 가상화 코드 시작 전 레지스터 상태를 복구한다. 이때 가상화 되는 코드가 MessageBox 함수의 인자를 설정하는 코드이기 때문에 해당 인자가 설정된 상태로 복구된다.



[그림 37] VM EXIT 과정

가상화 코드에서는 코드 분석을 방해하기 위해 진행 흐름에 변화가 없는 더미 코드들이 존재한다. 더미 코드에는 자주 보이는 패턴들이 존재하며 분석하여 다음과 같은 몇몇 패턴들을 확인할 수 있었다.

- 같은 레지스터 add, sub 패턴

  같은 레지스터에 같은 값을 더하고 빼는 경우 명령어 수행 전, 후 레지스터와 스택 상태가 동일하기 때문에 더미코드로 판단할 수 있다.

- 같은 피연산자 연산 패턴

  and, or, mov와 같은 명령어의 경우 피연산자가 동일하면 명령어 수행 전, 후 레지스터와 스택 상태가 동일하기 때문에 더미 코드로 판단할 수 있다. 레지스터 값과 상수를 연산할 때 레지스터 값과 상수 값이 동일한 경우도 있다.

- Add, sub Zero 패턴

  add와 sub 명령어에서 0을 더하거나 빼는 명령어의 경우 더미코드로 판단할 수 있다.

- mov reg 실행 전 해당 레지스터 대상 연산 패턴

  mov reg 명령어로 레지스터 값을 초기화하게 되면 이 전 명령어 흐름에서 스택에 변화를 주는 명령어가 없을 시 해당 레지스터가 사용된 모든 연산을 더미 코드로 판단할 수 있다.

위에서 설명한 더미코드 패턴의 예시는 [표 2] 더미 코드 패턴 예시와 같다.

	패턴 구조	어셈블리 코드 예시
1	add-sub	add r9, 8 sub r9, 8

2	같은 피연산자	and rax, rax
		mov r8, r8
		mov r10, 0 ; r10 = 0x0

3	+ - Zero	add rax, 0
		sub rcx, 0
		sub rax, r12 ; r12 = 0x0

4	mov reg 이전 레지스터 대상 연산	add r10, 0x8 mov r10, 0x500

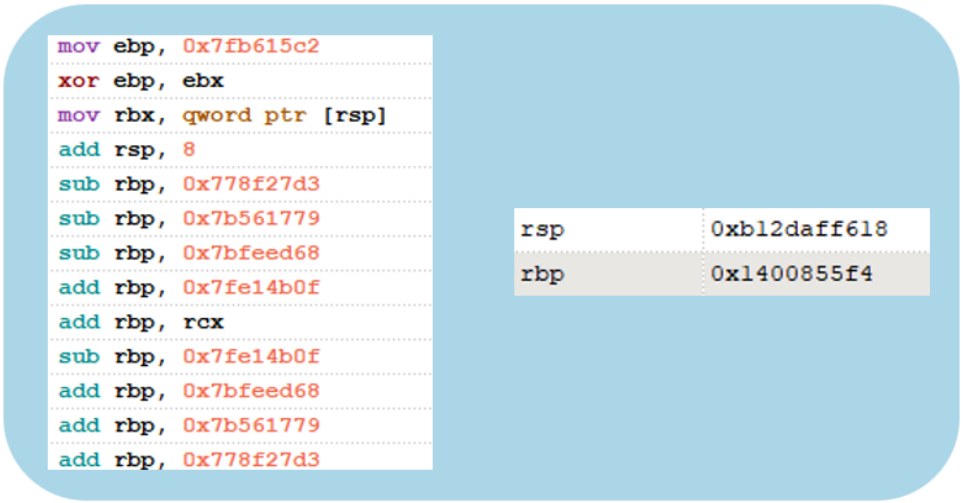
[표 2] 더미 코드 패턴 예시

이 외에도 어셈블리 코드 실행 전 후 스택과 레지스터 상태가 동일한 경우 더미 코드로 판단할 수 있다. 따라서 더미 코드의 패턴을 잘 찾아 제거하여 가상화 코드의 크기를 줄이면 더 쉽게 가상화 코드를 분석이 가능하다.

가상화 코드에서는 범위가 매우 큰 .themida 섹션 내에서 jmp 명령으로 실행 흐름을 이어가는데 이 때 대부분의 경우 "jmp reg" 명령 이 후 "jmp address" 명령을 통해 결과적으로 다음 실행할 주소로 점프하게 된다. "jmp reg" 명령을 실행하는 시점에서 해당 레지스터 값은 당연히 "jmp address" 명령어의 주소가 되기 때문에 레지스터 값을 파악할 수 있다면 가상화 코드의 실행 흐름을 파악할 수 있다.

[그림 38] 가상화 코드 실행 흐름

가상화 코드에서는 메모리의 특정 부분을 가상 CPU에서 사용할 가상화 스택으로 지정하고 레지스터들은 기존의 CPU에서 사용되었던 역할이 아니라 가상화 코드 진행을 위한 다른 역할을 하게 된다. 더미다로 가상화된 코드는 rbp를 가상화 스택을 가리키는 포인터로 사용하는데 [그림 39] 가상화 코드 내 rbp 값 수정 부분과 같이 가상화 코드 내에서 rbp값을 특정 .themida 섹션 내에 특정 주소를 가리키도록 수정하며 수정된 rbp 값은 가상화 코드가 끝날 때까지 해당 값을 유지한다. rbp값이 변환된 시점 이후 rbp는 일반적인 프로그램에서 스택 베이스 포인터의 역할을 하지 않으며 프로그램의 스택과는 전혀 다른 주소를 가리키는 것을 확인할 수 있다.



[그림 39] 가상화 코드 내 rbp 값 수정 부분

위와 같이 rbp값이 수정된 이후 rbp 값을 기준으로 메모리에 접근하여 해당 주소의 값을 가져오거나 연산하는 등의 행위를 많이 발견하여 rbp 레지스터를 VSP(Virtual Stack Pointer) [3]로 판단하였다. VSP를 기준으로 가상화 스택에 접근할 때는 다음과 같이 다른 레지스터에 rbp 값과 오프셋을 더하는 방식으로 가상화 스택에 접근한다. 이때 오프셋으로 사용되는 레지스터는 고정되어 있지 않다.

```
mov rsi, rbp
sub r8, 1
add rsi, 0xa9
mov r15, qword ptr [rsi]
```

[그림 40] 가상화 스택 접근 예시

[3] 가상화 코드에서 VSP는 일반적인 프로그램에서 ESP가 아닌 가상화 스택을 가리키는 포인터 역할

가상화 코드 내부에서 반복적으로 호출되는 함수를 찾을 수 있었는데 다음과 같다.

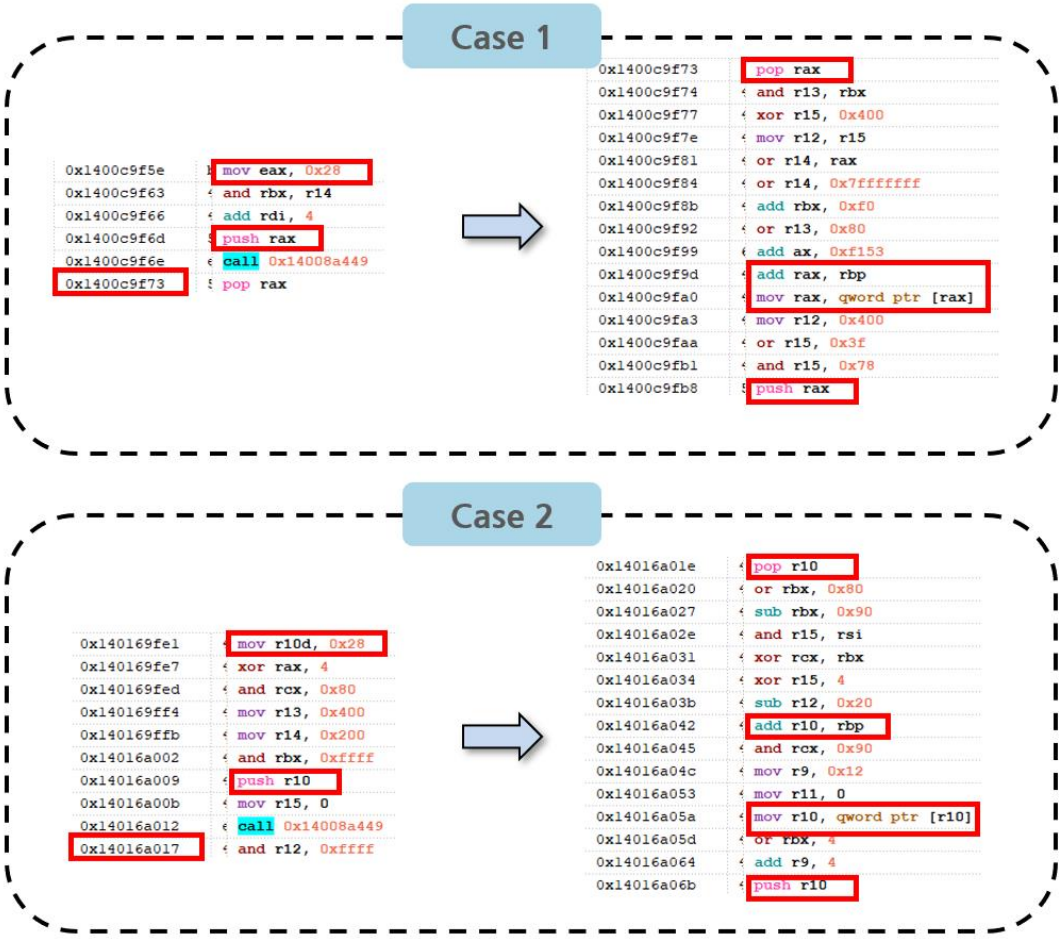
[그림 41] 가상화 코드 내 호출되는 함수

해당 함수가 호출되면 함수 호출 전 레지스터 상태를 스택에 저장하고 함수가 종료되기 전 함수 호출 전의 레지스터 상태로 복구하는 것을 확인할 수 있다. 함수 내부에서 함수의 행위를 보면 rbp를 기준으로 특정 메모리에 접근하여 연산하는데 위 함수를 IDA 헥스레이로 변환하면 [그림 42] 가상화 내 호출 함수 헥스레이 확인과 같이 함수가 실행되었을 때 rbp+8에 특정 값으로 비트 or 연산 후 비트 and 연산하는 행위를 확인할 수 있다.

```
1 void __fastcall vm_handler_selecter(__int64 a1, __int64 a2, __int64 a3, __int64 a4, __int64 a5)
2 {
3   __int64 _rbp; // rbp
4
5   *(_DWORD *)(_rbp + 8) |= 0x13F09AE0u;
6   *(_DWORD *)(_rbp + 8) &= 0x5F8FBEB4u;          // [rbp+8] or -> and
7 }
```

[그림 42] 가상화 내 호출 함수 헥스레이 확인

이 함수는 가상화 코드 내에서 여러 번 호출되는 함수이지만 호출 뒤 복귀 주소는 경우에 따라 달라진다. 해당 함수 호출 전 특정 레지스터를 스택에 push하게 되고, 해당 레지스터에는 특정 상수가 들어간다. 이 때 스택에 push 하는 레지스터의 종류에 따라 함수 호출 후 복귀 주소가 달라지고 복귀 후 해당 레지스터에 rbp값과 오프셋을 더해 가상화 스택에 접근하여 가상화 스택의 값을 가져와 스택에 push 한다.



[그림 43] 가상화 코드 내 함수 복귀 예시

Case1은 함수 호출 전 rax 레지스터를 push 하며 0x1400c9f73으로 복귀하고, Case2는 r10레지스터를 push 하며 0x14016a017로 복귀하는 것을 확인할 수 있다. 또한 push전 두 레지스터 값이 0x28로 동일한 것을 알 수 있는데 코드 가상화를 적용한 다른 프로그램과 비교하여 확인했을 때 push 전 레지스터에 저장되는 값이 0x28은 아니지만 같은 상수 값이 저장되며 저장한 레지스터의 종류에 따라 함수 호출 후 같은 주소로 복귀하는 것을 분석할 수 있었다.

함수 복귀 후에는 호출 전 push한 레지스터로 스택의 값을 가져와 해당 값을 활용하여 오프셋을 만들고, rbp 값과 더해 가상화 스택 안의 값을 가져온 후 원본 스택에 push한다. 해당 과정은 여러 번 반복되어 실행되며 MessageBox함수의 인자인 hWnd 0, lpText의 문자열의 주소, lpCaption의 문자열의 주소, uType값들이 원본 스택으로 들어가 가상화 코드가 종료될 때 스택에서 레지스터 값들을 복구하며 MessageBox 함수를 호출한다.

6. 결론

더미다는 분석을 방해하기 위해 NtSetInformationThread, NtQueryInformationProcess, NtUserGetForegroundWindow, RaiseException, VirtualProtect API들을 사용하는 다양한 Anti-Debugging기법들을 사용한다. 이 중 패킹을 해제하는 과정에 해당 기법을 넣어둠으로써 Anti-Debugging 우회가 쉽지 않게 한다는 점이 코드 분석 환경 설정을 더욱 힘들게 한다. Anti-Debugging 기법을 우회하여 원본 코드가 실행되는 시작지점에 도착해도 API Wrapping과 코드 가상화를 통해 어떤 API가 호출되고 어떤 인자를 사용하는지 확인하기가 어렵다. 하지만 Anti-Debugging 기법의 종류와 API Wrapping의 패턴을 분석하여 이를 해제하는 도구를 개발할 수 있었고 가상화 또한 일정한 패턴을 가지며 진행되고 있음을 알 수 있었다. 따라서 앞으로 해당 연구를 발전시켜 더 많은 가상화 코드를 패턴화하고 이를 이용해 현재 개발한 도구, bobalkkagi에 더 많은 기능을 구현할 예정이다.

7. Reference

[1] https://www.oreans.com/Themida.php

[2] https://learn.microsoft.com/ko-kr/windows/win32/api/

[3] 이재휘, 이병희, 조상현. 2019. 최신 버전의 Themida가 보이는 정규화가 어려운 API 난독화 분석방안 연구. 한국정보보호학회 VOL.29, NO.6. https://doi.org/10.13089/JKIISC.2019.29.6.1375