Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.27, NO.1, Feb. 2017 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2017.27.1.67

Themida의 API 난독화 분석과 복구방안 연구

이 재 휘,† 한 재 혁, 이 민 욱, 최 재 문, 백 현 우, 이 상 진‡
고려대학교 정보보호대학원

A Study on API Wrapping in Themida and Unpacking Technique

Jae-hwi Lee,† Jaehyeok Han, Min-wook Lee, Jae-mun Choi,
Hyunwoo Baek, Sang-jin Lee‡
Center for Information Security Technologies, Korea University

요 약

프로텍터란 프로그램의 핵심 아이디어를 보호하기 위해 실행파일을 압축하고 난독화를 적용하는 프로그램이다. 그리고 최근 악성코드가 자신의 악성행위를 분석하기 어렵게 만들기 위해 프로텍터를 적용하고 있다. 프로텍터가 적용된 실행파일을 정밀분석하기 위해서는 프로텍터를 해제하는 언패킹 작업이 필요하다. 기존의 연구에서는 OEP를 찾아 언패킹을 하는 것에 주력하였지만, 분석방해가 목적인 프로텍터의 경우 실행압축 해제가 완료된 이후의 실행에 분석방해 기능들이 남아있게 되어 여전히 분석에 어려움이 있다. 본 논문에서는 분석방해가 목적인 Themida 프로텍터가 사용하는 분석방해 기술을 분석하고 그에 대한 대응방안을 제안한다.

ABSTRACT

A protector is a software for protecting core technologies by using compression and encryption. Nowadays malwares use the protector to conceal the malicious code from the analysis. For detailed analysis of packed program, unpacking the protector is a necessary procedure. Lately, most studies focused on finding OEP to unpack the program. However, in this case, it would be difficult to analyze the program because of the limits to remove protecting functions by finding OEP. In this paper, we studied about the protecting functions in the Themida and propose an unpacking technique for it.

Keywords: Themida, API wrapping, Unpacking

I. 서 론

최근 악성코드가 자신의 악성행위를 분석하기 어렵게 만들기 위해 프로텍터를 적용하고 있다. 프로텍터는 패커의 한 종류로 프로그램의 핵심 아이디어를 보호하기 위해 실행파일을 압축하고 난독화를 적용한 후 프로그램이 메모리에 로드될 때 실행 가능한 형태로 복구해 주는 소프트웨어다. 역공학을 지연시키는 패커는 파일의 크기를 줄이는데 주력하는 컴프레서(compressor), 원본파일의 정보를 숨기기 위해 암호화나 난독화를 적용하는 크립터(crypter), 컴프레서와 크립터의 기능을 함께 적용하여 분석을 지연시키는 프로텍터(protector), 여러 파일을 하나의 실행파일로 만들어 작동하도록 만드는 번들러(bundler)로 나누어진다. 실행에 필요한 정보들을 압축하고 난독화하는 특성으로 인해 프로텍터가 적용된 실행파일을 자세히 분석하기 위해서는 프로텍터를 해제하는 과정이 먼저 수행되어야 한다. 그렇기 때문에 악성코드 분석가와 백신 프로그램은 프로텍터에 대응하기 위한 방안이 필요하다[1].

기존에 진행된 연구들은 엔트로피를 통해 패커의 압축해제 완료 시점을 찾는데 주력하였지만, 프로텍터의 경우 압축해제가 완료된 이후에도 분석을 방해

Received(10. 12. 2016), Accepted(12. 08. 2016)
† 주저자, slimv0x00@korea.ac.kr
‡ 교신저자, sangjin@korea.ac.kr(Corresponding author)

하기 위한 기능들이 남아 여전히 분석에 어려움이 있다. 본 논문에서는 알려진 상용 프로텍터 중 하나인 Themida 프로텍터[2]의 압축해제 완료시점 이후의 코드에서 분석방해 기능들을 연구하여 원본 실행파일을 복구하는 방법을 제안하고자 한다.

논문의 구성은 다음과 같다. 2장에서는 기존에 진행된 실행압축 해제와 Themida 언팩에 관한 연구들에 대해 알아본다. 3장에서는 프로텍터 중 하나인 Themida가 가지는 분석 방해 기능들의 작동방식에 대해 알아보고, 4장에서는 Themida로 보호된 실행파일의 분석방해 기능들을 해제하여 원본 실행파일을 복구하는 방법을 제시하고 그 결과를 확인한다. 마지막 5장에서는 본 연구의 결론 및 향후 연구에 대해 기술한다.

II. 관련연구

2.1 원본코드의 OEP 확인

패커 적용이 보편화되면서 이에 대응하기 위한 연구도 진행되고 있다. 패커가 적용된 실행파일의 원본 코드를 분석하기 위해서는 원본 코드의 시작 지점인 OEP(Original Entry Point)를 찾아야 한다. 기존의 연구는 엔트로피를 통해 OEP를 찾는 방법을 적용하고 있다. 패킹을 적용한 실행파일은 압축된 영역을 해제하는 코드가 존재하며, 이 압축을 해제하는 작업이 진행될 때에는 좁은 범위의 주소에 존재하는 명령어를 반복적으로 실행한다. 이후 압축해제가 완료되고 나면, OEP를 지나 정상코드로 진입하게 되어 실행하는 명령어의 주소 범위가 상대적으로 넓어진다. 이 특징을 이용해 명령어 주소의 엔트로피를 관찰하여 OEP를 찾을 수 있다[3].

다른 방법으로 엔트로피 값의 변화량을 이용하여 OEP를 찾을 수 있다. 일반적으로 패커를 적용하게 되면 파일 내부에 저장된 데이터의 엔트로피가 증가한다. 패킹을 적용한 파일은 실행 시에 압축된 데이터를 해제하면서 데이터의 엔트로피가 감소하며, 압축해제가 완료된 이후에 데이터의 엔트로피 변화가 거의 없다. 이 특징을 이용해 데이터의 엔트로피 변화를 관찰해 OEP를 찾을 수 있다[4][5][6].

패커를 해제하는 방법에 관한 연구가 진행되고 있지만, 다수의 분석방해 기술이 더해진 프로텍터의 경우 OEP를 찾아 언패킹에 성공하더라도 OEP 이후의 코드에 분석방해 기능들이 남아있는 경우가 종종 있다. 이러한 경우에는 앞선 연구들로는 프로텍터로 인한 분석방해 기능들을 완전하게 제거할 수 없다는 한계가 있다. 따라서 이러한 한계를 극복하기 위해서는 프로텍터가 사용하는 기술에 대한 연구가 필요하다.

프로텍터는 디버거가 사용하는 브레이크 포인트를 탐지하거나, 디버깅 중인 프로세스가 가지는 플래그 값 검사를 통해 디버거를 탐지한다. 디버거가 탐지되면 프로그램의 실행을 중단하거나 정상 흐름이 아닌 곳으로 이동해 분석을 방해한다. 또한 정상 실행흐름에 영향을 주지 않는 더미코드를 삽입해 코드의 양을 늘려 분석하는데 오랜 시간이 걸리게 만든다. 이때 매 실행마다 코드가 변하는 다형성 성질이 적용된 경우라면 분석은 더욱 어렵다[7].

이와 같은 프로텍터가 적용된 실행파일을 분석하기 위해서, 프로텍터가 사용하는 분석방해 기능들의 작동방식을 분석하고, OEP 이후에 남아있는 분석방해 기능들에 대응하기 위한 방안을 제시한다.

2.2 Themida 언팩 스크립트

현재 Themida로 보호된 실행파일을 복구하는 툴이나 방법들이 공개되어있다. 그러나 이들은 특정 어셈블리 명령어를 찾아 언패킹을 진행하기 때문에 다양한 대상에는 적용을 못하는 경우가 대부분이며, 언패킹 과정을 거친 이후에도 API 난독화와 난독화 해제 코드 등이 존재하여 실행파일의 크기가 원본보다 커지는 등의 제한사항이 있다.

Themida+WinLicense 2.x 스크립트[8]는 짧은 세 개의 스크립트를 통해 Themida가 적용된 실행파일을 언패킹한다. 먼저 분석방해 기능들이 실행되기 위해 필요한 메모리 할당과 라이브러리 접근을 추적하고, 분석방해 기능들이 사용하는 외부 라이브러리 함수들을 추적해 원본코드에서 사용하는 함수들의 정보를 복구한다.

WinLicense Ultra Unpacker 1.4 스크립트[9]는 주석 포함 14000줄 이상의 명령어로 이루어져 있으며, 다양한 언패킹 옵션을 통해 Themida가 적용된 실행파일을 언패킹 해 준다고 설명한다. 튜토리얼 형식으로 웹페이지에 게재된 내용[10]을 통해 언패킹 과정을 확인할 수 있다.

위의 스크립트들을 이용해 언패킹을 진행하면 Table 1.과 같이 복구한 파일의 크기가 증가하는 것을 알 수 있다. 이는 Themida가 생성한 분석방해 기능들을 완전하게 제거하지 못해 발생한다.

Table 1. Unpacked files increased in size

Script	Packed	Unpacked
Themida+WinLicense 2.x	1,585 KB	3,487 KB
Ultra Unpacker 1.4	7,706 KB	29,538 KB

III. Themida의 분석방해 기능

윈도우즈 운영체제 계열이 사용하는 실행파일 형식인 PE(Portable Executable)[11]에서는 실행파일에서 사용하는 외부 라이브러리 API들의 주소를 IAT(Import Address Table)[12]에 저장한다. IAT는 현재 실행 중인 메모리상의 API 주소를 가지고 있어야 하는데, 이는 PE 파일이 실행될 때 로더[13]가 실행에 필요한 데이터들을 메모리에 올리는 작업을 통해 이루어진다. Fig. 1.은 PE 파일이 실행될 때 로더를 통해 메모리에 올라가는 것을 보여준다. 따라서 실행 중인 프로세스를 분석한다면 IAT에서 관찰되는 API만으로도 프로세스의 행위를 추측할 수 있다.

Fig. 2.와 같이 프로텍터인 Themida는 실행파일에 분석방해 기능들을 추가한 후 실행압축을 이용하는 난독화를 적용한다. 적용된 난독화로 인해 정적 분석으로는 내부에 숨겨진 원본파일과 분석방해 기능에 대한 정보를 얻는 것이 어렵다. 그러므로 프로텍터로 보호된 실행파일은 주로 동적분석을 통해 분석을 진행한다.

프로텍터로 보호된 파일을 실행하게 되면 일반적

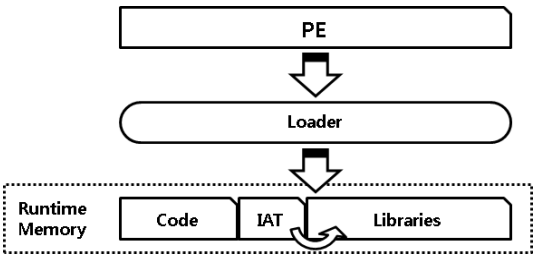


Fig. 1. Loader mapping executable on memory

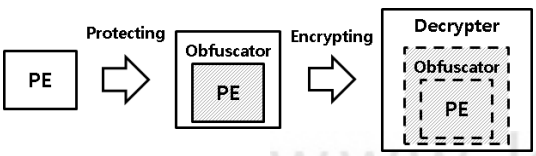


Fig. 2. Process of protecting PE

인 실행과는 다른 방식으로 진행된다. 난독화로 인해 Import table[14]이 손상되어 실행되기 전에는 사용하는 API에 대한 정보를 얻을 수 없으므로 로더는 IAT에 데이터를 저장하지 못한다. 따라서 원본 실행파일 상의 Import table 주소를 비교해보면 Fig. 3.처럼 원본 파일에서는 Import table이 온전하게 나타나고, 패킹을 적용한 파일에서는 아무런 데이터가 없는 것으로 나타난다.

Import table이 손상되었지만 언패킹 과정을 통해 숨겨져 있던 정보가 드러나면서 프로텍터가 직접 IAT에 데이터를 저장해 정상적으로 실행이 가능하도록 복구한다. 실행압축으로 인한 난독화가 모두 해제되고, IAT 복구가 완료되어 프로텍터의 작업이 끝나면 OEP가 실행되며 원본파일의 실행흐름으로 넘어간다.

Fig. 4.는 프로텍터로 보호된 실행파일이 메모리에 올라가는 것을 보여준다. 로더를 통해 실행압축을 해제하는 코드가 실행되고, 압축해제가 완료된 후 난독화 모듈이 실행된다. 보호된 원본 실행코드 영역은 난독화 모듈의 API 난독화가 완료된 이후 OEP에

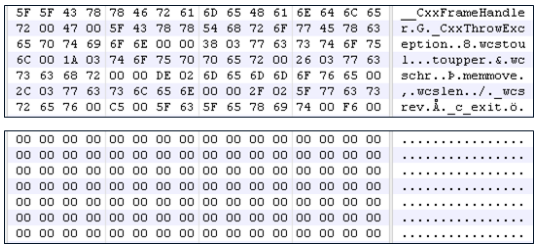


Fig. 3. Comparison at address of Import table section in original binary (top : original binary, bottom : unpacked binary)

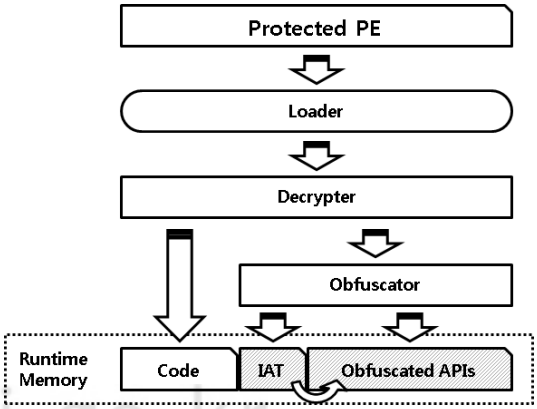


Fig. 4. Process of running protected executable

도달할 때 실행한다. OEP 이후 실행 중인 상태에서 정상 실행파일을 실행하던 Fig. 1.의 메모리와 Fig. 4.의 메모리를 비교해 보면, 라이브러리 내부의 함수를 가리키던 Fig. 1.의 IAT와는 달리 Fig. 4.의 IAT는 난독화가 적용된 API의 영역을 가리키고 있는 것을 볼 수 있다.

3.1 OEP 이후의 분석 방해 기능

보호된 실행파일이 실행되면 실행압축을 통해 난독화 된 영역들이 해제된다. 프로텍터는 분석을 방해하기 위해서 실행압축 해제 과정에 디버거를 탐지하는 안티디버깅, 디버깅에 활용하는 프로세스들을 탐지하는 프로세스 모니터링, 브레이크 포인트 또는 코드패치를 탐지하는 무결성 검사 등의 기능들을 동시에 적용한다. 하지만 안티디버깅과 프로세스 모니터링은 드라이버를 이용해 무력화가 가능하며, 무결성 검사는 하드웨어 브레이크 포인트와 외부 프로세스의 스크립트를 통해 무력화 할 수 있다. 또한 실행압축이 완료된 후 원본파일의 실행흐름으로 넘어가는 시점인 OEP를 찾아 덤프하게 되면, 앞선 기능들과 같이 실행압축 해제 과정 중에 실행되는 방해기능들은 제거할 수 있다. 따라서 OEP를 찾아 실행압축을 해제하는 앞선 연구들을 이용하면, 안티디버깅과 같이 압축해제 과정 중에 실행되는 방해기능들은 제거할 수 있다. 그러나 Themida의 경우 OEP 이후의 실행에 영향을 미치는 분석방해 기능들이 여전히 남아있다. OEP 이후의 실행에 영향을 미치는 기능들의 작동원리를 파악한다면 적용된 분석방해 기술의 대응방안을 연구할 수 있을 것이다.

3.1.1 Dummy code 삽입

더미코드는 실행흐름에 영향을 미치지 않는 코드를 다량으로 삽입하여 분석에 필요한 시간이 늘어나도록 만드는 분석방해 기술이다. Fig. 5.와 같이 원본 코드를 명령어 단위로 나누고 그 사이에 더미코드를 삽입하는 방식으로 작동한다. 적용한 기능 설정에 따라 매 실행마다 삽입되는 더미코드가 변하는 다형성 코드(polymorphic code) 생성 기술을 함께 적용해 코드영역 메모리 덤프 등과 같은 방식으로 분석을 보다 어렵게 만든다.

Table 2.와 Table 3.은 실행압축이 적용된 동일한 실행파일을 반복적으로 실행하며 실험한 결과이

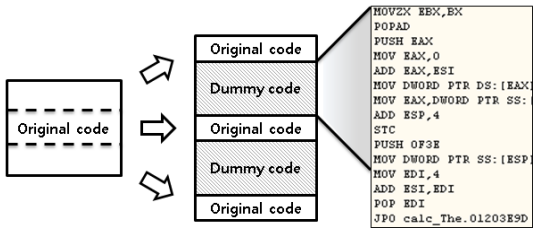


Fig. 5. Applying dummy code

다. Table 2.를 보면 서로 다른 실행에서 동일한 주소 0x01580100 상의 명령어가 모두 다른 것을 알 수 있다. Table 3.은 압축해제가 완료된 OEP 이후 사용하게 되는 코드영역을 덤프해 해시 값을 검사한 결과이다. OEP 이후의 코드에서도 더미코드의 영향으로 매 실행마다 해시 결과가 서로 다른 것을 알 수 있다.

Table 2. Instruction on address 0x01580100 for each execution

#	Instruction
1	OR EAX, DWORD PTR DS:[EAX]
2	POPAD
3	MOV CL, CH
4	RCL DWORD PTR DS:[EAX], CL
5	MOV AH, 0DD

Table 3. MD5 hash of code section for each execution

#	MD5
1	2b68eb841e94135f8bc1217d837c70d4
2	6a732824df927542b25c8d3b5904255c
3	fa644219f9ab66a377fa4dafa5dfcbe7
4	4ba7e6fb71f0dd763f89f2cd79e7f940
5	4fadc57f5e5938de50df3f548c59fe65

3.1.2 API 난독화 (API wrapping)

API 난독화는 실행 중에 호출하는 API를 파악하기 어렵게 만드는 분석방해 기술이다. 원본 API의 코드에 다형성이 적용된 더미코드를 삽입해 난독화를 진행한다. 이후 난독화가 적용된 API를 OEP가 지난 시점의 명령어 실행에서 호출하도록 만들어 분석가가 API의 정보를 알 수 없도록 분석을 방해한다.

API 난독화는 난독화가 적용된 API를 저장하기

위한 메모리 공간이 필요하다. 이때 실행파일 내부의 공간은 난독화가 적용된 API를 저장하기에는 부족하므로 동적 할당을 이용해 새로운 메모리를 할당받아 사용한다. 새로운 메모리 할당은 NTDLL의 NtAllocateVirtualMemory[15] 함수를 관찰하면 확인할 수 있다. 할당받는 메모리 영역은 난독화 진행을 위해 읽기, 쓰기 권한이 필요하며 난독화 이후 사용을 위해 실행 권한도 필요로 한다. 일반적으로 읽기, 쓰기, 실행 권한이 모두 적용되어 메모리를 할당받는 경우는 매우 드물다. 이 특징을 활용한다면 API 난독화를 위한 메모리 공간을 찾을 수 있다. Fig. 6.을 보면 API 난독화를 적용하기 위해 메모리가 0x1000 또는 0x2000 크기로 할당되며 읽기, 쓰기, 실행 권한을 모두 가지고 있는 것을 알 수 있다. 할당받은 메모리에는 난독화를 위하여 API의 명령어와 다형성 코드 기술이 적용된 더미코드가 교차하며 복사된다.

API 난독화가 완료되면 OEP 이후의 실행에서 난독화가 적용된 API를 호출하도록 만들어야 한다. 윈도우즈 운영체제 계열이 사용하는 실행파일 형식인 PE에서는 실행파일에서 사용하는 외부 라이브러리 API들의 주소를 IAT에 저장한다. 그러나 Themida로 보호된 실행파일에서는 원본 API의 주소 대신 난독화가 적용된 API의 주소를 IAT에 저장한다. Fig. 7.은 시스템 라이브러리인 KERNEL32.dll의 API들을 난독화 한 후, 원본

Address	Size	Access	Initial
01580000	00001000	RWE	RWE
01590000	00002000	RWE	RWE
015A0000	00001000	RWE	RWE
015B0000	00001000	RWE	RWE
015C0000	00001000	RWE	RWE

Fig. 6. Memories for API obfuscation

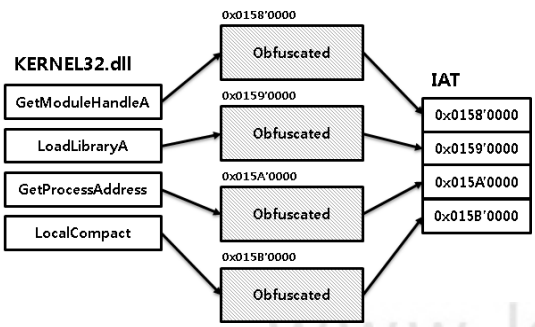


Fig. 7. Save obfuscated address in IAT

API의 주소가 아닌 난독화 된 영역의 주소 값을 IAT에 저장하는 형태이다.

3.1.3 CALL 명령어 수정

난독화가 끝나고 주소 값을 모두 IAT에 저장하고 나면, OEP 이후의 실행에서 난독화가 적용된 API를 호출하도록 코드영역을 수정한다. Fig. 8.과 같이 처음에는 함수를 호출하는 명령어가 있어야 할 주소에 아무런 작업도 하지 않는 명령어인 NOP(No Operation)[16]로 모두 초기화 되어 있다가, 코드가 수정되면서 난독화가 적용된 API를 호출하는 CALL 명령어로 덮어쓴다.

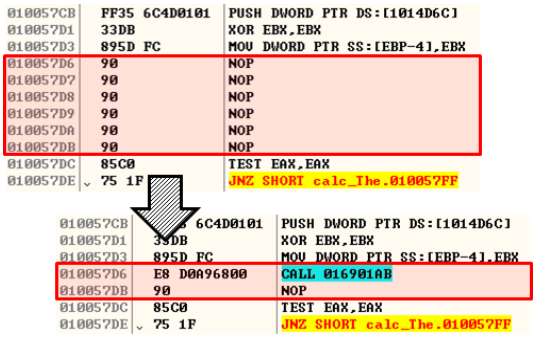


Fig. 8. NOP patched with CALL for obfuscated API

IV. Themida의 분석방해 기능 해제

Themida로 보호된 실행파일의 원본을 복구하기 위해서는 API 난독화에 대한 정보를 찾아야 한다. 찾아낸 정보를 기반으로 난독화 된 영역과 원본 API 간의 관계를 파악하고, 난독화가 적용된 API를 호출하는 CALL 명령어들이 원본 API를 호출하도록 수정해야 한다. 이후 OEP에서 정지한 상태의

Table 4. Process for unpacking Themida

#	Proc	Technique	Removal
1	API	API wrapping (Dummy code, Polymorphic)	Match API
2	Code	Patch CALL	Restore CALL
		Encrypted	Find OEP
3	Memory dump		
4	Rebuild PE		

메모리를 덤프 해 PE 파일 구조에 맞게 재조립하면 실행압축이 해제된 실행파일 복구가 완료된다. Table 4.는 단계별로 적용된 기능들을 해제하여 원본 실행파일을 복구하는 방법을 정리한 표이다.

4.1 API 난독화 복구

API 난독화는 원본 API에 다형성 코드 기술을 적용한 더미코드를 삽입한다. API를 난독화하기 위해서는 원본 API에 접근하는 상황이 발생하게 되므로, 난독화를 위해 할당받은 메모리에 명령어를 복사하는 과정을 분석하면 원본 API의 정보를 얻을 수 있다. Fig. 9.는 난독화 과정 중에 레지스터 값을 통해 원본 API의 주소가 나타나는 것을 보여준다. 원본 API의 주소와 난독화 결과가 저장되는 메모리의 주소를 연결해 복구하면 API 난독화를 무력화할 수 있다.

원본 API와 난독화가 적용된 주소를 연결하는 알고리즘은 Fig. 10.과 같다. 새로운 메모리가 할당된 경우 메모리의 접근권한이 읽기, 쓰기, 실행 권한을 가지고 있는지 검사한다. 권한이 확인되면 난독화를

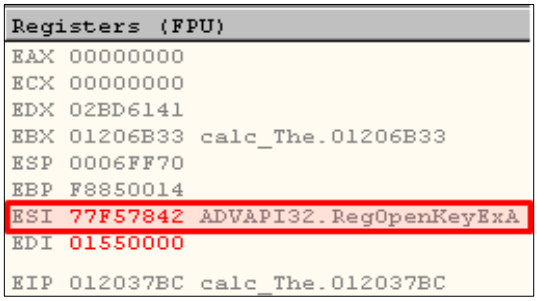


Fig. 9. Original API address appears at register

위해 할당받는 메모리의 크기가 0x1000 또는 0x2000인 특성을 이용해 난독화를 위한 할당인지 검사한다. 크기가 조건에 맞으면 해당 메모리는 API 난독화를 위해 할당받은 영역으로 판단하여 메모리 주소를 저장하는 리스트인 lMem에 저장한다. 메모리 복사 이벤트가 발생할 경우 복사되는 destination 주소가 lMem 리스트에 속해있는 영역인지 검사한다. lMem 리스트에 속한 주소로 확인된 경우 복사하는 source 주소를 검사해 해당 주소가 API의 주소인지 검사한다. API의 주소로 확인되면 API 난독화 작업으로 판단하여 source 주소

```
lastEvt := Latest event
newMem := Latest allocated memory
copySrc := Source address of copying
copyDst := Destination address of copying
nameApi := Name of API
lMem := List of memory spaces
dicWrap := Dictionary for save results
GetLastEvent() : Returns the last event
GetName() : Returns API name of given address

while lastEvt = GetLastEvent()
  if lastEvt = EVT_ALLOC_MEMORY then
    if newMem.protect = READ | WRITE | EXECUTE then
      if newMem.size = 0x1000 or newMem.size = 0x2000 then
        lMem.append(newMem)
  if lastEvt = EVT_MEM_COPY then
    if copyDst ∈ lMem then
      nameApi := GetName(copySrc)
      if nameApi != API_UNKNOWN then
        dicWrap(copyDst) = copySrc
```

Fig. 10. Algorithm for matching obfuscated area and original API

와 destination 주소를 1대 1로 연결하여 딕셔너리 형태로 dicWrap에 저장한다.

OEP 이후의 시점에서 알고리즘을 이용해 수집한 정보로 난독화가 적용된 영역의 주소를 원본 API의 주소로 치환하면 API 난독화를 무력화한다. Fig. 11.은 위 알고리즘을 적용하여 난독화가 적용된 영역의 주소와 원본 API를 연결한 결과이다. 기록된 결과를 보면 원본 API에 난독화를 적용해 저장하는 영역의 주소가 최근에 할당받은 메모리 영역의 주소에 속하는 것을 알 수 있다. 만약 프로텍터를 해제하지 않더라도 위 결과를 참고한다면 API 난독화가 적용된 영역을 분석하는 시간을 줄일 수 있을 것이다.

```
948F32 : VirtualAlloc(1000) = 15B0000
[*] 15B0000 <- ADVAPI32!RegQueryValueExW
948F32 : VirtualAlloc(1000) = 15C0000
[*] 15C0000 <- ADVAPI32!RegSetValueExW
948F32 : VirtualAlloc(1000) = 15D0000
[*] 15D0000 <- ADVAPI32!RegOpenKeyExA
948F32 : VirtualAlloc(1000) = 15E0000
[*] 15E0000 <- ADVAPI32!RegQueryValueExA
948F32 : VirtualAlloc(1000) = 15F0000
[*] 15F0000 <- ADVAPI32!RegCreateKeyExW
948F32 : VirtualAlloc(1000) = 1600000
[*] 1600000 <- ADVAPI32!RegCloseKey
948F32 : VirtualAlloc(1000) = 15A0000
[*] 160047E <- kernel32!FindResourceW
948F32 : VirtualAlloc(1000) = 1610000
[*] 1610000 <- kernel32!OutputDebugStringA
[*] 1610445 <- kernel32!SetHandleCount
[*] 161050D <- kernel32!LoadResource
[*] 16108F4 <- kernel32!lstrlenW
948F32 : VirtualAlloc(1000) = 1620000
[*] 1620000 <- kernel32!GetPrivateProfileIntW
```

Fig. 11. Result of using matching algorithm

4.2 CALL 명령어 복구

API 난독화가 완료되면 OEP 이후의 실행영역에서 난독화가 적용된 API를 호출하도록 CALL 명령어가 수정되기 때문에, 앞서 찾은 결과를 활용해 난독화 된 영역을 호출하도록 수정된 CALL 명령어를 원본 API를 호출하도록 복구해야 한다. 복구가 완료되면 분석가가 바로 원본 API를 호출하는 코드를 알 수 있으므로 분석을 쉽게 진행할 수 있다. Fig. 12.는 CALL 명령어가 난독화 된 API를 호출하지 않고 원본 API를 호출하도록 복구한 결과를 보여준다. 기존의 난독화 된 영역인 0x0169'01AB를 호출하는 CALL 명령어가 복구 이후 USER32의 API인 OpenClipboard를 호출하는 것을 명확하게 표시할 수 있다.

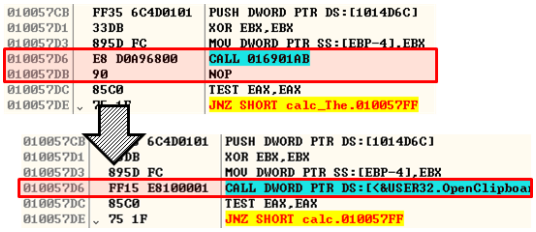


Fig. 12. Recovered CALL instruction

4.3 PE 재구성

API 난독화를 무력화하고, 코드 영역의 CALL 명령어를 복구한 후 각 섹션의 메모리를 덤프하여 PE 구조에 맞게 재조립하면 분석 가능한 수준의 실행파일을 얻을 수 있다. 이때 Themida로 인해 손상된 Import table을 복구하기 위해 새로운 IT 섹션을 마지막 영역에 추가한다.

4.4 원본코드 복구 수행결과

Themida로 실행파일을 보호한 후, 제안하는 기법을 활용해 OEP 이후에 남아있는 분석방해 기능들을 제거하였다. 실험환경은 Windows XP SP3 x86이고, 사용한 Themida의 버전은 2.4.5.0이다. 실험은 Windows 기본 실행파일인 계산기(calc.exe)와 메모장(notepad.exe), Visual Studio 2013으로 컴파일 한 실행파일(HelloThemida.exe)을 Themida로 보호한 후 복구를 진행하였다. 복구된 실행파일은 모두 정상적으로 작동하였다.

Table 5. MD5 hash of unpacked binaries

#	calc.exe's MD5
1	72629e5b82a08938eae3c8beabbac612
2	72629e5b82a08938eae3c8beabbac612
3	72629e5b82a08938eae3c8beabbac612
#	notepad.exe's MD5
1	037028b6d26f4026841064fc62f6a221
2	037028b6d26f4026841064fc62f6a221
3	037028b6d26f4026841064fc62f6a221
#	HelloThemida.exe's MD5
1	b7a4be14b5cef703ab3a29e5eb9944c7
2	b7a4be14b5cef703ab3a29e5eb9944c7
3	b7a4be14b5cef703ab3a29e5eb9944c7

OEP 이후에도 남아있던 분석방해 기능들을 모두 제거한 상태이므로 복구결과는 항상 동일할 것이다. Table 5.는 실행압축이 적용된 동일한 파일을 반복 실험을 통해 복구하여 실행파일의 해시 값을 검사한 결과이다. 서로 다른 복구결과를 검사하였지만 실행파일의 해시 값이 모두 동일한 것을 알 수 있다.

Table 6.은 보호된 실행파일과 복구한 실행파일의 PE 구조의 섹션[17]을 나타내는데 섹션의 수가 줄어든 것을 알 수 있다. 복구한 실행파일에서 CODE, RSRC 섹션은 보호된 실행파일의 섹션을 복구한 것으로 RVA와 섹션의 크기가 유사한 것을 볼 수 있으며, IT 섹션은 Import table 복구를 위해 새로 추가한 영역이다. 원본 실행파일에서의 Import table은 코드 섹션에 존재하며, 복구한 실행파일에서는 추가한 IT 섹션에 존재한다.

Table 6. Comparison of file's section

Section	Packed file (RVA (Size))	Unpacked file (RVA (Size))
calc.exe		
CODE	1000 (15000)	1000 (15000)
RSRC	16000 (8960)	16000 (9000)
IDATA	1F000 (1000)	X
IT	X	20000 (1000)
SFX00	20000 (1FB000)	X
SFX01	21B000 (134000)	X
SFX02	34F000 (1000)	X
notepad.exe		
CODE	1000 (A000)	1000 (A000)
RSRC	B000 (8304)	B000 (9000)
IDATA	14000 (1000)	X
IT	X	14000 (1000)
SFX00	15000 (1FC000)	X
SFX01	211000 (132000)	X
SFX02	343000 (1000)	X
HelloThemida.exe		
CODE	1000 (17000)	1000 (17000)
RSRC	18000 (1E0)	18000 (1000)
IDATA	19000 (1000)	X
IT	X	19000 (1000)
SFX00	1A000 (1F8000)	X
SFX01	212000 (12D000)	X
SFX02	33F000 (1000)	X

Table 7.은 복구가 완료된 실행파일과 Themida로 보호한 실행파일의 비교표로 파일의 크기가 확연히 줄어든 것을 알 수 있다. 또한 보호대상인 원본 실행파일의 크기와 비교했을 때 차이가 크지 않은 것을 알 수 있다. 원본 파일과 발생하는 차이는 PE가 메모리에 mapping될 때 페이지의 최소단위인 SYSTEM_INFO.dwPageSize[18]에 맞춰 할당되는 것과 Import table 복구를 위해 추가한 IT 섹션으로 인해 발생한다.

Themida를 적용하지 않은 원본파일과 복구한 실행파일의 각 섹션에 대한 해시를 비교하면 Table 8.과 같이 Import table이 속한 섹션의 해시를 제외한 나머지 영역들은 동일하게 나타난다. Import table은 실행파일 상의 위치에 제약이 없다. Import table은 계산기와 메모장의 경우 CODE 섹션에 존재하며, HelloThemida는 DATA 섹션에 존재한다. 해시 차이는 Import table에 의해 나타나므로 Import table을 제외하고 해당 섹션의 해시를 비교하면 동일한 것을 알 수 있다.

Table 7. Comparison of file's size

File type	File size
calc.exe	
Original file	112 KB
Packed file	1,282 KB
Unpacked file	132 KB
notepad.exe	
Original file	66 KB
Packed file	1,260 KB
Unpacked file	84 KB
HelloThemida.exe	
Original file	80 KB
Packed file	1,246 KB
Unpacked file	104 KB

Table 8. Comparison of MD5 hashes of sections between original binary and unpacked binary

Section	Original file's MD5
	Unpacked file's MD5
calc.exe	
CODE	42ee4a79790faee000aadbbfc49b445f
	8e002aefd18714b2f4a82aa25002f215

CODE (No IT)	ae008c60838dc0bd1335bcfc4d82dfdd
	ae008c60838dc0bd1335bcfc4d82dfdd
DATA	8bcf4df09c861a983840c849063c93c3
	8bcf4df09c861a983840c849063c93c3
RSRC	44b7157cc8637357bfd647678f5ec5b9
	44b7157cc8637357bfd647678f5ec5b9
notepad.exe	
CODE	412feb618167742571306861cec0c2e4
	54b1b1c38891a0f24f8e843e3f412175
CODE (No IT)	0f0d87bd91c44e1aa99be889828f7903
	0f0d87bd91c44e1aa99be889828f7903
DATA	05c63c7c0593b2a9f421263a5aa6fdee
	05c63c7c0593b2a9f421263a5aa6fdee
RSRC	5f171908505f3b10c339e872f343a7e2
	5f171908505f3b10c339e872f343a7e2
HelloThemida.exe	
CODE	07cb0dc3e7673e4da9657b15cf1aebe3
	07cb0dc3e7673e4da9657b15cf1aebe3
RDATA	1ddd1244f52f973d47c8cb48532d57a0
	7f06e5f121c1ae0b6f90b3410642fbed
RDATA (No IT)	17314ca2ae47369b6d45a93564ddbf81
	17314ca2ae47369b6d45a93564ddbf81
DATA	64e40e55d1e010669f77f369668437c4
	64e40e55d1e010669f77f369668437c4
RSRC	8ec87334ecce71a916dde4e38e309a1c
	8ec87334ecce71a916dde4e38e309a1c

V. 결 론

프로텍터 중 하나인 Themida가 사용하는 분석 방해 기술을 연구하고, 기존의 연구들이 해결하지 못하는 실행압축 완료 이후에 남아있는 분석방해 기능에 대한 대응방안을 기술하였다. 프로텍터가 적용된 실행파일을 분석하는 것은 행위기반 분석을 통해 일부 가능하지만, 정밀분석을 위해서는 프로텍터를 해제해야 한다. 악성코드가 자신의 악성행위 분석을 어렵게 만들기 위해 프로텍터를 이용하는 만큼, 프로텍터에 대한 대응방안 마련이 필요할 것으로 보인다. 기존의 연구들로 실행압축이 완료되는 지점(OEP)을 찾고, 본 연구를 발전시켜 프로텍터의 실행압축 이후에 남아있는 분석방해 기능들을 제거한다면 프로텍터가 적용된 실행파일들의 정밀분석을 위한 환경을 구축할 수 있다.

앞으로는 더 많은 종류의 프로텍터를 연구해 프로텍터가 보이는 보편적인 특징들을 찾아내고, 여러 종류의 프로텍터에 대응 가능한 알고리즘을 찾아내 자동화하는 연구를 진행할 예정이다.

References

[1] Yan, Wei, Zheng Zhang, and Nirwan Ansari, "Revealing packed malware", *IEEE seCurity & PrivaCy,* Vol. 6.5, pp. 65-69, Oct. 2008

[2] Oreans Technologies, Themida, Advanced Windows Software Protection System, Revision 2.4, May 2016 (also see Themida, http://www.oreans.com/themida.php)

[3] Won Lae Lee and Hyoung Joong Kim, "A Study on Generic Unpacking using Entropy of Opcode Address", *Journal of Digital Contents Society*, Vol. 15, No. 3, pp. 373-380, Jun. 2014

[4] Young-hoon Lee, et al. "A Study on Generic Unpacking using Entropy Variation Analysis", *Journal of The Korea Institute of Information Security & Cryptology*, Vol. 22, No. 2, pp. 179-188, Apr. 2012

[5] Guhyeon Jeong, et al. "Generic Unpacking using Entropy Analysis", *Journal of The Korea Institute of Information Technology*, Vol. 7, No. 1, pp. 232-238, Feb. 2009

[6] Ryoichi Isawa, Masaki Kamizono, and Daisuke Inoue, "Generic Unpacking method Based on Detecting Original Entry Point", *ICONIP 2013*, Part I, LNCS 8226, pp. 593-600, 2013

[7] Boo Joong Kang, and Eul Gyu Im, "A Study on Anti-Debugging in Yoda's Protector", *Journal of The Korea Institute of Communications and Information Sciences*, pp. 1229-1232, Jul. 2007

[8] LCF-AT, "Themida+WinLicense 2.x (Unpacking)", Jul. 2013 (also see https://tuts4you.com/download.php?view.3495)

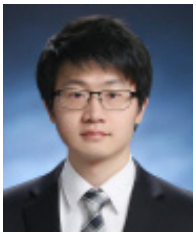
[9] LCF-AT, "Themida+WinLicense 2.x (Ultra Unpacker v1.4)", Jan. 2014 (also

see https://tuts4you.com/download.php?view.3526)
〔10〕 CriticalError, "How Unpack Themida 2.x.x (WXP)", Jun. 2015 (also see http://zenhax.com/viewtopic.php?t=1051)
〔11〕 Microsoft, Microsoft PE and COFF Specification, Revision 8.3, Feb. 2013 (also see Microsoft PE and COFF Specification, https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx)
〔12〕 Microsoft, Microsoft Portable Executable and Common Object File Format Specification, Revision 8.2, Sep. 2010 (also see Understanding the Import Address Table, http://sandsprite.com/CodeStuff/Understanding_imports.html)
〔13〕 Mark Russinovich, David A. Solomon, and Alex Ionescu, "Image Loader", Windows Internals Part 1, Sixth Edition, pp. 232-247, Microsoft press, 2012
〔14〕 Matt Pietrek, Microsoft, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format", Mar. 1994 (also see https://msdn.microsoft.com/en-us/library/ms809762.aspx)
〔15〕 Microsoft, Microsoft MSDN: NtAllocateVirtualMemory (also see NtAllocateVirtualMemory, https://msdn.microsoft.com/en-us/library/windows/hardware/ff556440(v=vs.85).aspx)
〔16〕 Intel, "NOP-No Operation", Intel® 64 and IA-32 architectures software developer's manual volume 2B: Instruction set reference, M-U, pp. 163, Jun. 2016 (also see Intel® 64 and IA-32 Architectures Software Developer Manuals, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html)
〔17〕 Microsoft, "Section Table (Section Headers)", Microsoft PE and COFF Specification, Revision 8.3, pp. 26-32, Feb. 2013 (also see Microsoft PE and COFF Specification, https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx)
〔18〕 Microsoft, Microsoft MSDN: SYSTEM_INFO structure (also see SYSTEM_INFO structure, https://msdn.microsoft.com/en-us/library/windows/desktop/ms724958(v=vs.85).aspx)

〈저 자 소 개〉

이 재 휘 (Jae-hwi Lee) 학생회원
2015년 2월: 경북대학교 컴퓨터학부 공학사
2015년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 디지털 포렌식, 역공학

한 재 혁 (Jaehyeok Han) 학생회원
2011년 2월: 서울시립대학교 수학과 졸업
2016년 2월: 고려대학교 정보보호대학원 공학석사
2016년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 디지털 포렌식, 파일시스템, 데이터 마이닝



이 민 욱 (Min-wook Lee) 학생회원
2014년 2월: 경기대학교 전자공학과 졸업
2016년 8월: 고려대학교 정보보호대학원 석사
〈관심분야〉 디지털 포렌식, 역공학

최 재 문 (Jae-mun Choi) 학생회원
2015년 2월: 호서대학교 정보보호학과 졸업
2015 3월~현재: 고려대학교 대학원 사이버국방학과 석사과정
〈관심분야〉 디지털 포렌식, 역공학

백 현 우 (Hyunwoo Baek) 학생회원
2015년 2월: 경희대학교 전자전파공학과 공학사
2015년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 디지털 포렌식, 정보보안정책

이 상 진 (Sang-jin Lee) 종신회원
1989년 10월~1999년 2월: ETRI 선임 연구원
1999년 3월~2001년 8월: 고려대학교 자연과학대학 조교수
2001년 9월~현재: 고려대학교 정보보호대학원 교수
2008년 3월~현재: 고려대학교 디지털포렌식연구센터 센터장
2015년 1월~현재: 고려대학교 정보보호대학원 부원장
〈관심분야〉 디지털 포렌식, 심층 암호, 해쉬 함수