碩 士 學 位 論 文

프로텍터 유형의 분석방해 기술 우회방안에 관한 연구

高 麗 大 學 校

情 報 保 護 大 學 院

情 報 保 護 學 科

李 在 輝

2017 年 6 月

석 사 학 위 논 문

프로텍터 유형의 분석방해 기술 우회방안에 관한 연구

고 려 대 학 교
정 보 보 호 대 학 원
정 보 보 호 학 과
이 재 휘

2017 년 6 월

李 相 珍 敎 授 指 導

碩 士 學 位 論 文

프로텍터 유형의 분석방해 기술 우회방안에 관한 연구

이 論文을 工學 碩士學位 論文으로 提出함.

2017 年 6 月

高 麗 大 學 校

情 報 保 護 大 學 院

情 報 保 護 學 科

李 在 輝 (인)

이 상 진 교 수 지 도

석 사 학 위 논 문

프로텍터 유형의 분석방해 기술 우회방안에 관한 연구

이 논문을 공학 석사학위 논문으로 제출함.

2017 년 6 월

고 려 대 학 교

정 보 보 호 대 학 원

정 보 보 호 학 과

이 재 휘 (인)

이재휘의 工學 碩士學位 論文 審査를
完了함.

2017 年 6 月

위원장 이 상 진 인

위 원 김 휘 강 인

위 원 윤 지 원 인

요 약

과거 메모리가 부족했던 시절 실행파일이 차지하는 용량을 줄이기 위해 실행파일의 기존 데이터를 압축해 보관하고, 압축을 풀어주는 코드를 실행파일에 추가해 실행 시에 압축된 데이터를 해제해 실행 가능한 형태로 복구하는 실행압축 프로그램인 패커가 개발되었다. 패커로 실행압축 된 실행파일은 압축이 해제되기 전에는 패커로 압축하기 이전의 원본 데이터를 식별할 수 없어 정적분석이 어려운 점을 이용해 실행압축을 역공학을 방해하는 용도로 사용하기 시작했다. 패커는 단순히 압축 또는 암호화를 통해 데이터를 식별하기 어렵게 만드는 형태에서 더 나아가 분석탐지 기술과 코드 난독화 그리고 코드 가상화를 이용하는 프로텍터 형태로 발전하였다.

초기에는 분석을 지연시킬 목적으로 프로텍터를 적용한 악성코드가 다수 발견되었으나, 최근에는 백신에서 실행파일 내부에 인증이 없고 프로텍터가 적용된 경우 위협요소로 탐지하여 프로텍터를 적용하지 않는 추세이다. 다만 범죄행위 또는 첩보활동에 사용되는 실행파일에 적용된 경우가 여전히 발생하므로 프로텍터에 대응하기 위한 연구가 반드시 필요하다.

프로텍터가 적용된 실행파일을 정밀분석하기 위해서는 우선적으로 프로텍터를 해제해 원본 실행파일을 복구해야 한다. 실행파일의 압축과 암호화의 경우 실행압축 해제가 완료된 시점에서는 분석에 더 이상 영향을 주지 않는다. 그러나 프로텍터가 가지는 분석방해 기술들의 경우 실행압축 해제 완료시점 이후에도 여전히 남아있어 정상적인 분석을 위해서는 해당 기술들을 우선적으로 우회해야 한다. 따라서 프로텍터의 분석방해 기술에 대한 연구와 각 기술에 대응하는 우회방안이 필요하다. 본 논문에서는 알려진 프로텍터가 가지는 분석방해 기술들을 분석하고 각 기술에 대응하는 우회방안을 제안하고자 한다.

목 차

그 림

표

제 1 장 서 론

2000년대 초반, 다양한 종류의 프로텍터가 나와 손쉽게 실행파일에 분석방해 기술을 적용할 수 있는 환경이 되어 프로텍터가 적용된 악성코드가 발견되기 시작했다. 당시에는 프로텍터가 적용된 악성코드에 어떻게 대응할 것인가에 대한 연구가 진행되었다. 하지만 최근에는 백신이 실행파일 내부에 유효한 인증이 없고 프로텍터가 적용된 경우 악성으로 판단하여 차단하므로, 더 이상 프로텍터를 악성코드에 적용하지 않는 추세이다. 다만 범죄행위 또는 첩보활동에 사용되는 실행파일의 경우 여전히 실행파일의 내부 알고리즘을 숨기고 분석을 방해하려는 목적으로 프로텍터를 적용하므로 프로텍터에 대응하기 위한 연구가 반드시 필요하다.

프로텍터가 적용된 실행파일을 정밀분석하기 위해서는 우선적으로 프로텍터를 해제해야 한다. 프로텍터가 사용하는 기술 중 실행파일의 압축과 암호화의 경우 실행압축 해제가 완료된 시점에서는 분석에 더 이상 영향을 주지 않아 간단히 해제가 가능하다. 그러나 프로텍터가 사용하는 분석방해 기술 중 원본 실행파일의 정보를 일부 손상시키는 유형의 경우, 실행압축 해제 과정에서 손상된 부분을 임의로 복구하므로 각 기술에 대응하는 추가적인 제거작업이 필요하다. 따라서 프로텍터가 사용하는 분석방해 기술에 대한 연구를 통해 각 기술에 대응하는 해제방안을 준비할 필요가 있다.

본 논문의 2장에서는 알려진 상용 프로텍터에 대해 알아보고, 프로텍터가 보편적으로 사용하는 분석방해 기술에 대해 서술한다. 3장에서는 제안하는 분석방해 기술 우회방안과 절차를 설명한다. 4장에서는 제안한 방법을 통해 프로텍터가 적용된 실행파일의 분석방해 기술을 우회하여 그 결과를 확인한다. 마지막 5장에서는 본 연구의 결론 및 향후 연구의 방향에 대해 기술한다.

제 2 장 프로텍터

2.1. 패커

과거 메모리가 부족했던 시절 실행파일이 차지하는 용량을 줄이기 위해 실행압축 프로그램인 패커(packer)가 개발되었다. 패커가 적용된 실행파일의 데이터는 패킹(packing)으로 인해 압축 또는 암호화되어 그 자체로는 식별이 어려운 형태로 변화한다. 이에 따라 패커는 역공학을 통한 소프트웨어 구조분석을 방해하는 용도로 사용되며 새로운 기술들을 추가로 적용해 나타나기 시작했다. 현재 패커는 단순히 실행파일의 용량 압축을 목적으로 하는 컴프레서(compressor), 원본 실행파일의 정보를 은닉하기 위해 코드를 암호화하는 크립터(crypter), 역공학을 통한 분석을 지연시키는 것을 목적으로 여러 분석방해 기술을 적용하는 프로텍터(protector), 여러 개의 실행파일과 환경을 통합하여 하나의 실행파일로 생성하는 번들러(bundler)로 나누어진다[1]. 이들 중 프로텍터는 개발자가 본인의 코드에 분석방해 기술을 추가하지 않아도 손쉽게 실행파일에 분석방해 기술을 적용할 수 있어 기업에서도 이용하고 있다.

실행파일을 패킹하면 변화한 내부구조에 따라 운영체제가 실행파일을 실행해주는 과정이 변하게 된다. 운영체제의 로더[2]가 실행파일의 정보를 읽어 메모리에 데이터를 올려주는데, 실행파일의 정보에는 사용하는 외부 라이브러리 API의 정보가 존재한다. 윈도우즈 운영체제가 사용하는 실행파일 형식인 PE(Portable Executable)[3]는 일반적인 실행파일의 경우 사용하는 외부 라이브러리 API의 정보를 임포트 테이블[4]에 저장한다. 실행파일을 실행하면 로더는 임포트 테이블을 읽어 실행에 필요한 라이브러리를 메모리에 올리고, 현재 프로세스 메모리에 적재된 API의 주소를 저장하는 IAT(Import Address Table)[5]에 주소를 저장한다. 이후 실행파일 내부에서 외부 라이브러리 API를 호출할 때 마다 API의 주소를 계산하지 않고, IAT를 참조하는 CALL 명령어를 이용해 API를 호출할 수 있다. 그림 1.은 패킹이 적용되지 않은 일반 실행파일이 실행되어 메모리에 올라가는 과정을 나타낸다.

패킹된 실행파일은 실행압축을 해제할 때 사용하는 API의 정보를 담은 임포트 테이블을 새로 만들어 추가하고, 원본 실행파일이 사용하는 외부 라이브러리 API의 정보를 담고 있던 임포트 테이블은 실행압축 해제가 완료되기 전에는 확인할 수 없는 형태로 변한다. 그림 2.는 실행파일에 패커의 한 종류인 프로텍터를 적용하기 전과 후의 임포트 테이블 모습을 보여준다. 위쪽의 프로텍터를 적용하기 전 실행파일은 사용하는 외부 라이브러리 API의 이름이 문자열 형태 존재하지만, 아래쪽의 프로텍터가 적용된 실행파일에서는 식별할 수 없는 데이터가 존재하는 것을 알 수 있다. 따라서 실행파일이 가지고 있는 데이터만을 가지고 정보를 판단하는 것이 불가능하다.

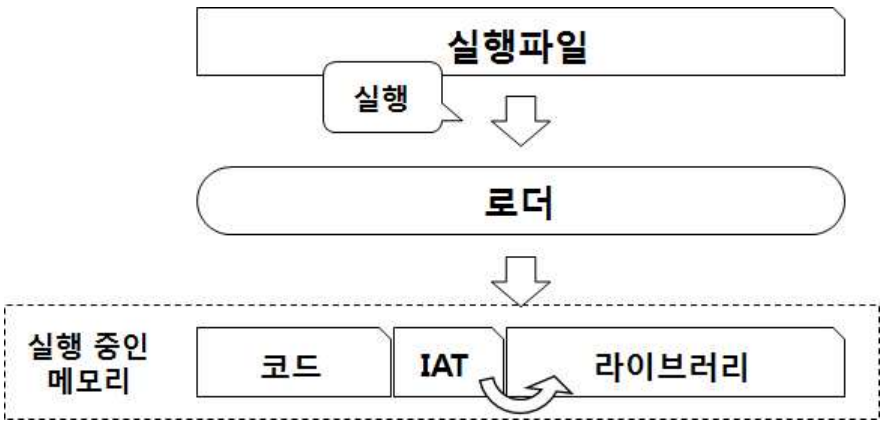

그림 1. 일반 실행파일 실행과정

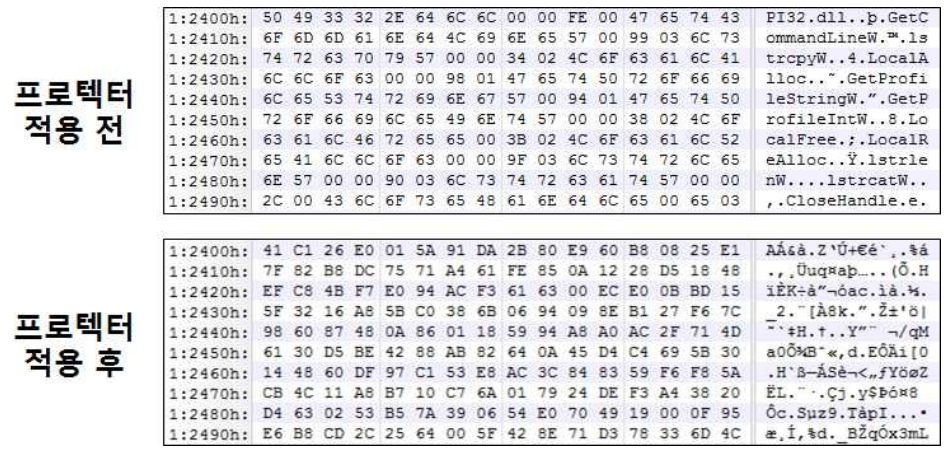

그림 2. 프로텍터 적용 전과 후의 임포트 테이블

패킹된 실행파일 자체로는 원본 실행파일이 사용하는 임포트 테이블을 식별할 수 없으므로 로더가 라이브러리를 준비해 줄 수 없다. 따라서 로더는 패커가 실행압축 해제에 사용하기 위해 생성한 임포트 테이블을 읽고, 실행파일을 메모리에 올려 실행하는 작업을 진행하게 된다. 이후 실행압축 해제 과정을 거치며 숨겨져 있던 사용하는 외부 라이브러리 API 정보가 드러나고, 드러난 정보를 이용해 실행압축 해제 과정에서 IAT에 주소를 채워주는 작업을 함께 진행한다. 그림 3.은 패킹된 실행파일이 실행되어 메모리에 올라가는 과정을 나타낸다. 그림 1.과 달리 로더가 실행파일을 메모리에 올린 다음 실행압축 과정을 거쳐 정상적인 코드가 나오게 된다.

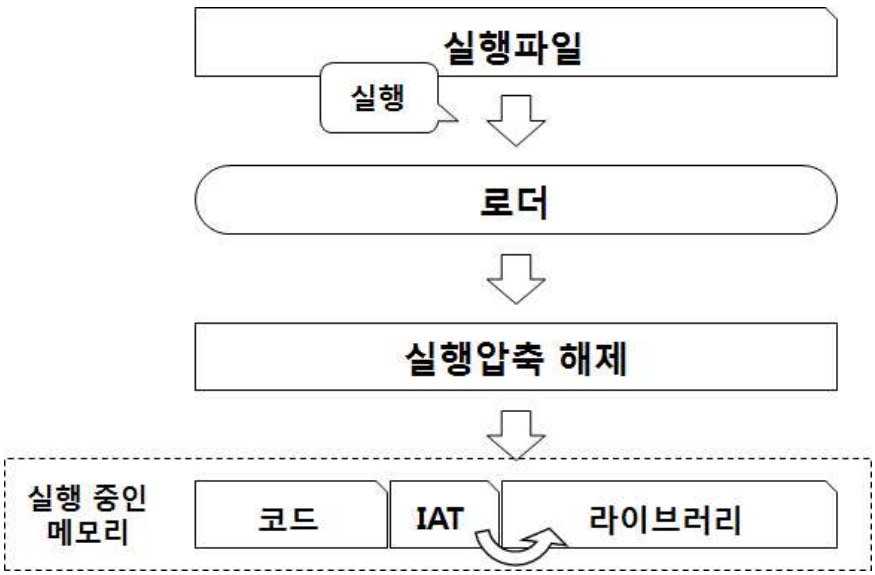

그림 3. 패킹된 실행파일 실행과정

2.2. 프로텍터

프로텍터는 기본적인 패커의 형식에서 역공학을 통한 실행파일의 흐름분석을 방해하기 위한 다양한 분석방해 기술들을 추가한 것이다. 알려진 프로텍터로는 ASProtect[6], Enigma[7], Obsidium[8], Themida[9], VMProtect[10] 등이 있다. 이들 모두 역공학을 통한 분석을 지연시키는 것이 목적이므로, 다양한 분석방해 기술들을 사용한

다. 사용하는 분석방해 기술의 유형은 코드를 식별하기 힘들도록 변경하는 코드 난독화, 역공학에 사용되는 분석 툴들을 탐지하는 안티 디버깅, 분석을 위한 코드패치나 소프트웨어 브레이크 포인트 삽입과 같이 실행파일의 데이터를 변경하는 행위를 탐지하는 무결성 검증, 사용하는 외부 라이브러리 API의 정보를 숨기고 난독화해 API 기반의 분석을 방해하는 API 난독화 그리고 실행파일 내부에 가상 CPU를 생성하고 가상화 대상 코드를 가상 CPU가 해석할 수 있는 바이트코드로 변경한 후 실행하는 코드 가상화가 있다. 표 1.은 분석방해 기술의 유형을 설명한 표이다.

표 1. 분석방해 기술 유형

분석방해 기술	설명
코드 난독화	코드 식별이 어렵도록 변경
안티 디버깅	분석 툴 탐지 및 대응
무결성 검증	데이터 변경 탐지 및 대응
API 난독화	외부 라이브러리 API 사용정보 은닉 및 난독화
코드 가상화	가상 CPU가 해석할 바이트코드 생성 및 원본코드 제거

가. 코드 난독화

코드 난독화는 분석시간을 늘리기 위해 코드를 식별하기 힘들도록 변경하는 기술이다. 프로텍터가 생성하는 코드는 기본적으로 코드 난독화가 적용된 형태를 가진다. 코드 난독화는 실행흐름에 영향을 미치지 않는 코드를 원본코드 사이사이에 삽입하는 더미코드와 작동흐름은 동일하지만 코드의 형태가 변화하는 폴리모픽 그리고 사용하는 시스템 라이브러리를 파일형태로 직접 메모리에 올린 다음 실행파일

의 구조를 해석해 라이브러리의 API를 호출하는 할로우 라이브러리[11]가 있다. 표 2.는 코드 난독화의 유형을 설명한 표이다.

표 2. 코드 난독화 유형

코드 난독화	설명	예제
더미코드	실행흐름에 영향을 미치지 않는 코드 삽입	**PUSH EAX** - EAX를 스택에 저장 **POP EAX** - 스택에서 EAX 복구 MOV ECX, EAX EAX를 스택에 저장한 다음 복구하여 마지막의 원본코드인 MOV 명령어에 영향을 미치지 않음

폴리모픽	매 실행 시 같은 작업을 하는 다른 코드 생성	**Form1.**
		MOV EAX, 402000h
		PUSH EAX
		
		Form2.
		MOV EAX, 1BAEA8h
		MOV ECX, 247158h
		LEA EAX, [EAX+ECX]
		PUSH EAX
		
		명령어의 구성은 차이가 있지만, 데이터 402000h를 스택에 저장

할로우 라이브러리	라이브러리를 파일형태로 직접 로드 후 사용	파일상의 위치(offset)와 메모리상의 상대주소(RVA)를 이용해 메모리상의 절대주소(VA) 계산 offset - section offset = RVA - section RVA VA = section VA + RVA

나. 안티 디버깅

안티 디버깅은 현재 실행환경이 디버깅용 환경인지, 역공학 분석을 위해 사용하는 툴이 현재 실행중인지를 탐지해 프로세스를 종료한다. 실행환경 검사는 주로 디버깅용 환경에서 설정되는 값을 검사해 판단하거나, 시간 값을 읽어 특정 명령어에서 그 이후에 위치한 명령어

에 도달하는데 걸린 시간을 검사해 일정 시간 이상 소요된 경우 디버깅 중인 상태로 판단한다. 분석에 사용되는 외부 툴 탐지는 현재 실행중인 프로세스의 목록에서 특정 툴의 이름의 존재여부를 확인한다. 또한 역공학 분석에 필수적으로 사용되는 브레이크 포인트의 존재여부를 검사해 디버깅 상태를 탐지하기도 한다. 표 3.은 안티 디버깅의 유형을 설명한 표이다.

표 3. 안티 디버깅 유형

안티 디버깅	설명
디버깅 환경 검사	프로세스가 디버깅용 환경인 경우 설정되는 값 검사
프로세스 목록 검사	현재 실행중인 프로세스의 목록에서 분석용 툴 존재여부 확인
브레이크 포인트 검사	브레이크 포인트 존재여부 확인

디버깅 환경 검사는 디버거가 대상 프로세스에 연결될 때 자동적으로 설정되는 값들을 검사한다. 프로세스 내부에는 TEB나 PEB[12]와 같이 현재 실행정보를 저장하는 다양한 시스템 구조체가 존재하며, 이 구조체 내부에는 프로세스가 디버깅 환경으로 설정된 정보를 표시하는 데이터가 존재한다. 표 4.는 환경 검사에 사용할 수 있는 데이터의 일반적인 값과 디버거가 연결된 프로세스가 가지는 값을 나타낸 표이다.

프로세스 목록 검사는 시스템으로부터 현재 실행중인 프로세스의 목록을 가져와 분석용 툴이 존재하는지를 확인한다. 프로세스의 목록을 가져오는 작업은 시스템 라이브러리인 KERNEL32가 제공하는 시스템 스냅샷 계열의 API[13]를 통해 할 수 있다. 그림 4.는 시스템 스냅샷 계열의 API를 이용해 프로세스의 목록에서 분석용 툴이 존재하는지 검사하는 알고리즘이다.

표 4. 디버깅 환경 검사용 데이터 정보

플래그	일반 상태값	디버깅 상태값
PEB.IsDebugged	0x0 (False)	0x1 (True)
PEB.NtGlobalFlag	0x0	0x70 (FLG_HEAP_ENABLE_TAIL_CHECK | FLG_HEAP_ENABLE_FREE_CHECK | FLG_HEAP_VALIDATE_PARAMETERS)
_HEAP.Flags	0x2 (HEAP_GROWABLE)	0x5000'0062 (HEAP_GROWABLE | HEAP_TAIL_CHECKING_ENABLED | HEAP_FREE_CHECKING_ENABLED | HEAP_SKIP_VALIDATION_CHECKS | HEAP_VALIDATE_PARAMETERS_ENABLED)
_HEAP.ForceFlags	0x0	0x4000'0060 (HEAP_TAIL_CHECKING_ENABLED | HEAP_FREE_CHECKING_ENABLED | HEAP_VALIDATE_PARAMETERS_ENABLED)
Trap Flag(TF)	0x0 (False)	0x1 (True)

브레이크 포인트는 소프트웨어 브레이크 포인트, 하드웨어 브레이크 포인트 그리고 메모리 브레이크 포인트가 있다. 소프트웨어 브레이크 포인트는 디버그 인터럽트 명령어인 “INT 3“을 나타내는 16진수 데이터 0xCC 값을 코드영역에 삽입하고, 해당 명령어가 실행되면 디버거에게 프로세스의 실행권한을 넘겨주는 방식으로 작동한다[14]. 따라서 소프트웨어 브레이크 포인트가 삽입되면 코드 영역의 데이터가 변하게 되므로 다음 소에서 설명할 무결성 검증과 같은 방식으로

코드영역을 검사하면 소프트웨어 브레이크 포인트 탐지가 가능하다.

하드웨어 브레이크 포인트는 CPU의 레지스터를 이용해 작동한다. IA32의 경우 DR0부터 DR7까지 총 8개의 디버그 레지스터를 가지고 있다. 디버거에서는 DR0부터 DR3까지 4개의 레지스터를 하드웨어 브레이크 포인트를 설정하는데 사용한다. 또한 DR7의 경우 디버그 컨트롤 레지스터로 현재 하드웨어 브레이크 포인트의 활성화 여부를 나타내는 정보를 가지고 있다[15]. 시스템 라이브러리인 KERNEL32의 API인 GetThreadContext[16]를 사용하면 DR 레지스터의 정보를 읽어와 하드웨어 브레이크 포인트 정보를 확인할 수 있다.

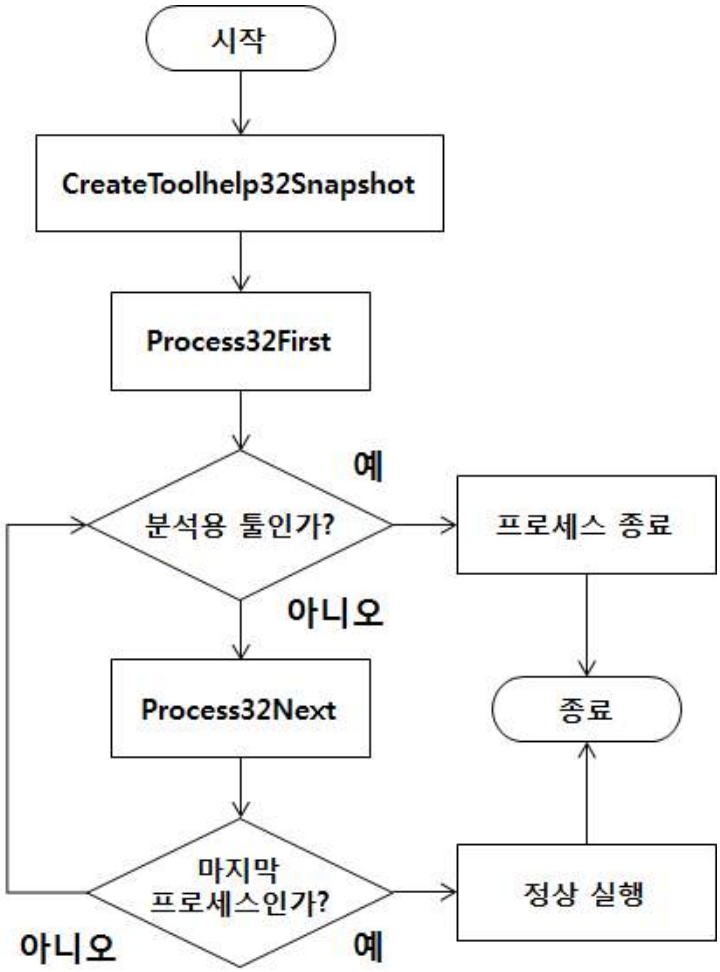

그림 4. 분석용 툴 프로세스 존재여부 검사

메모리 브레이크 포인트는 메모리에 설정된 접근권한을 변경하고, 해당 메모리 영역에 접근하는 경우 발생하는 STATUS_GUARD_PAGE_VIOLATION 오류를 이용한다[17]. 따라서 메모리 영역의 접근권한을 확인해 변경여부를 확인하거나, 접근오류를 넘겨받아 디버거에서 처리하도록 등록해놓은 SEH(Structured Exception Handling) 핸들러가 존재하는지 검사하면 메모리 브레이크 포인트를 탐지할 수 있다.

다. 무결성 검증

프로세스의 메모리를 일정한 구역으로 나눈 다음, 각 구역의 데이터를 CRC32 등의 방식으로 체크섬 계산한 결과를 지속적으로 비교해 무결성을 검증한다. 만약 코드패치 또는 소프트웨어 브레이크 포인트 삽입 등으로 인해 데이터가 변경되면 체크섬도 달라지므로 무결성이 훼손된 것으로 판단한다. 그림 5.는 검사대상 메모리 영역의 데이터를 이용해 체크섬 계산을 하고, 그 체크섬 결과 값을 검사하는 것을 나타낸다.

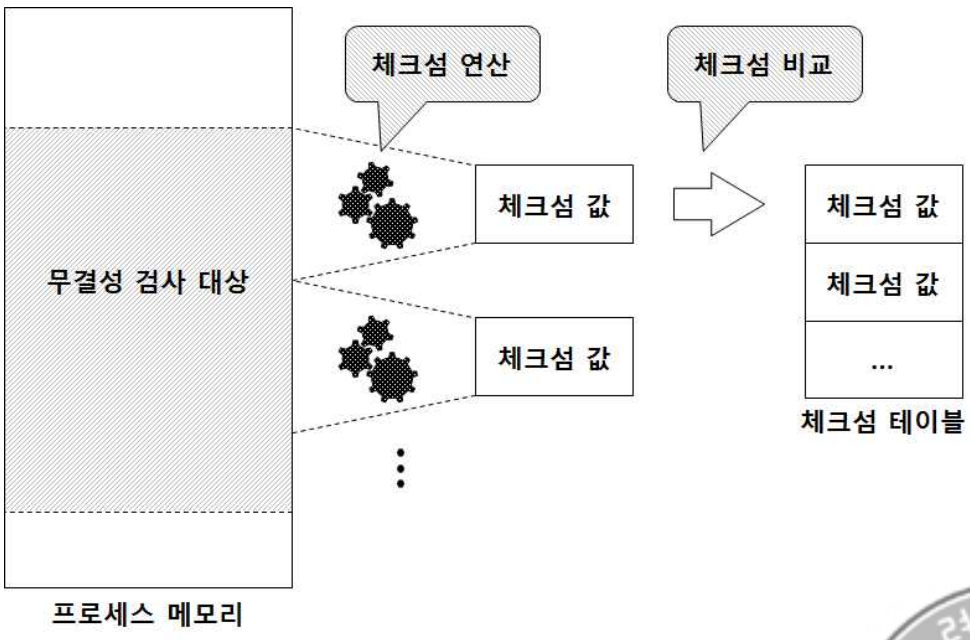

그림 5. 체크섬을 이용한 무결성 검사

라. API 난독화

API 난독화는 실행파일에서 사용하는 외부 라이브러리 API의 정보를 은닉하고 난독화 한다. 정상적인 윈도우즈 실행파일 구조에서는 실행 중인 프로세스의 IAT를 읽으면 사용하는 API를 정보를 알아낼 수 있다. 하지만 API 난독화를 적용하면 임포트 테이블의 정보를 손상시키고 은닉해 사용하는 외부 라이브러리 API의 정보를 식별하는 것이 어려워진다. 하지만 실행파일을 실행하면 실행압축 해제 과정을 거치면서 은닉된 정보가 복구되고, 드러난 정보를 이용해 코드 난독화가 적용된 API를 생성한 후 IAT에 원본 API의 주소 대신 저장한다. API 난독화가 적용된 실행파일이 메모리에 올라가는 과정은 그림 6.과 같다. 실행압축 해제 이후 난독화 기술을 통해 난독화 된 API를 생성하고, 아래쪽의 실행 중인 메모리에서 IAT가 난독화 된 API의 영역을 가리키도록 만든다.

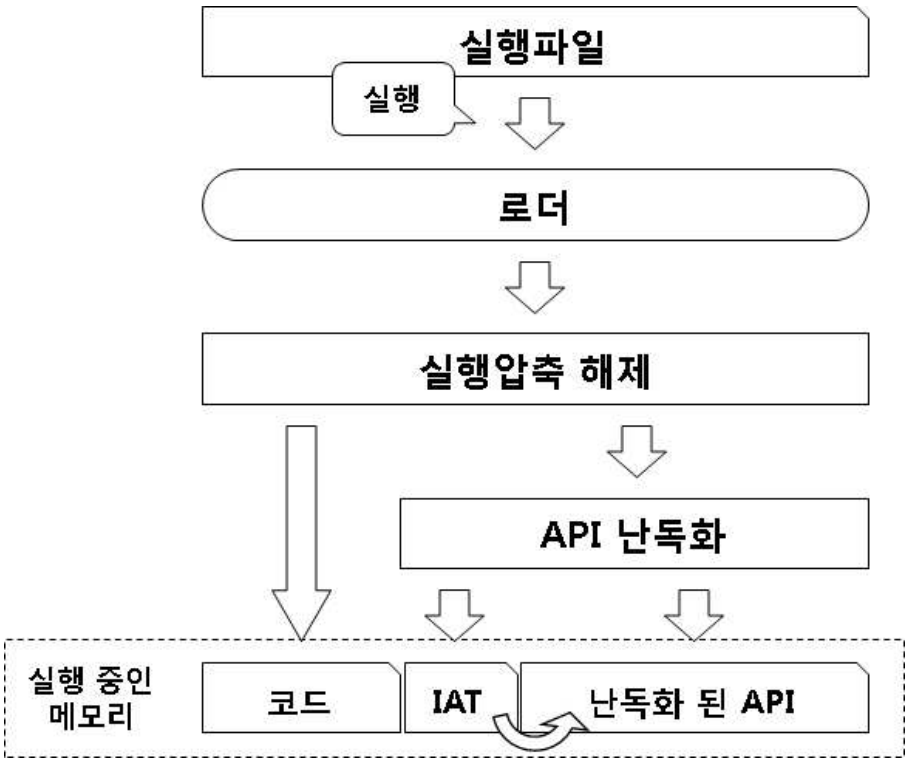

그림 6. API 난독화가 적용된 실행파일 실행과정

실행파일이 정상적으로 실행되려면 현재 시스템의 자원을 이용해야 한다. 마찬가지로 난독화가 적용된 API가 정상적으로 작동하려면 해당 시스템의 라이브러리 API를 이용해 난독화를 진행해야 한다. 따라서 프로텍터는 프로세스 실행 중에 현재 시스템에 존재하는 라이브러리의 API 코드를 가져와 난독화하거나 원본 API를 호출하는 난독화 된 코드를 생성한다. 그림 7.은 시스템 라이브러리인 KERNEL32의 API들을 난독화 한 후, 원본 API의 주소 대신 난독화 된 API의 주소를 IAT에 저장하는 유형을 나타낸다.

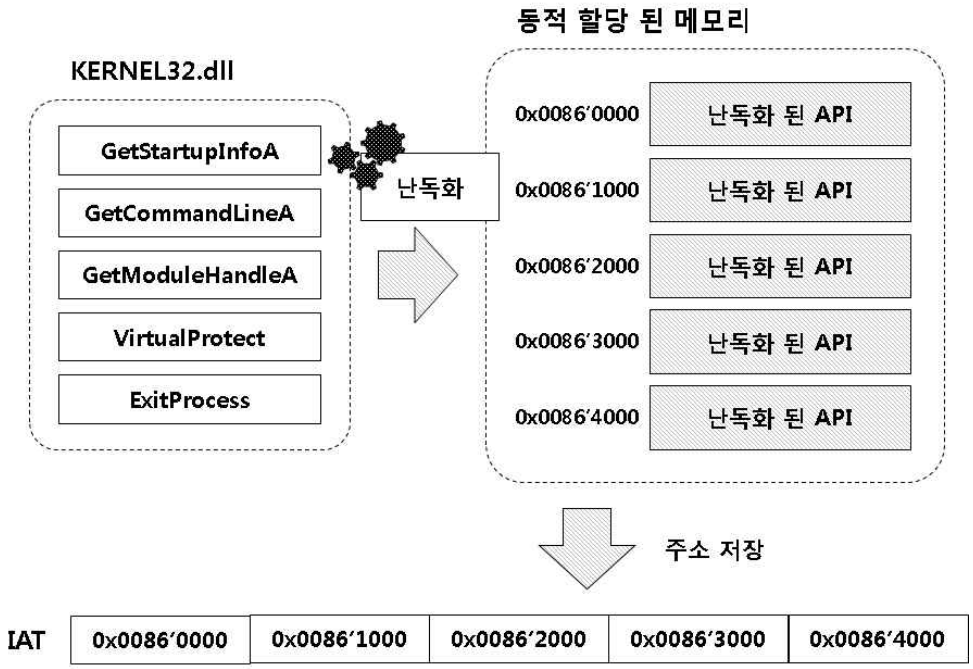

그림 7. 난독화 된 API의 주소를 IAT에 저장

마. 코드 가상화

코드 가상화는 가상화 적용 대상 실행파일에 가상 CPU를 생성해 추가하고, 원본코드를 가상 CPU가 해석할 수 있는 바이트코드로 변환한다. 변환이 완료되면 원본코드는 영구삭제 하여 가상화가 적용된 실행파일에서는 더 이상 찾을 수 없게 된다. 따라서 가상화가 적용된 실행파일에서는 가상 CPU가 영구삭제 된 코드를 의미하는 바이트코

드를 읽어 가상화된 영역을 실행한다. 코드 가상화가 적용된 실행파일을 복구하려면 가상 CPU 내부의 인터프리터의 구조를 파악해 바이트코드가 의미하는 명령어를 알아내야 하므로 가상화 영역 전체에 대한 분석이 필요하다. 또한 가상 CPU가 가지는 인터프리터의 구조는 실행파일에 가상화를 적용하는 시점에 결정되므로, 동일한 실행파일에 가상화를 적용하더라도 생성된 표본들은 서로 다른 인터프리터 구조를 가져 분석을 더욱 어렵게 만든다. 그림 8.은 코드 가상화가 적용된 영역의 구조를 나타낸다. 바이트 코드에서 실행할 바이트 코드를 가져와 인터프리터에서 해석한 후, 입력된 바이트 코드에 대응하는 핸들러를 호출해 연산을 수행한다.

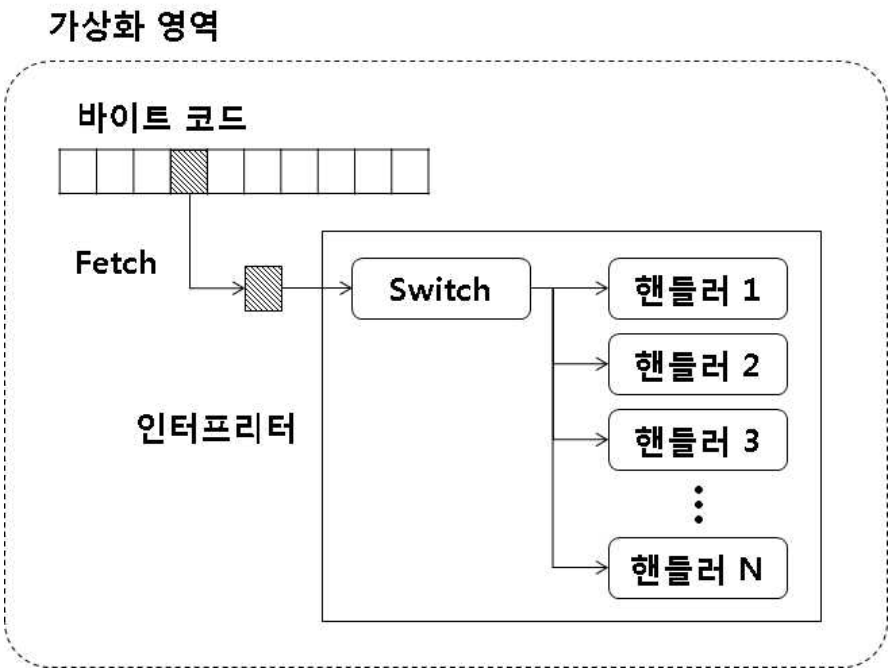

그림 8. 가상화 영역 실행 구조

제 3 장 프로텍터 분석방해 기술 우회방안

프로텍터가 적용된 실행파일을 정밀분석하기 위해서는 우선적으로 프로텍터가 가지는 분석방해 기술을 우회해 원본 실행파일을 복구해야 한다. 따라서 프로텍터가 적용된 실행파일을 복구하기 위해 앞서 살펴본 분석방해 기술에 대응할 수 있는 우회방안을 제안한다. 제안하는 각 분석방해 기술별 우회방안은 표 5.와 같다.

표 5. 분석방해 기술별 우회방안

분석방해 기술	우회방안
코드 난독화	OEP 탐지
안티 디버깅	디버거 검사 우회
무결성 검증	검증 무력화
API 난독화	원본 API 복구
코드 가상화	바이트코드 복구

그림 9.는 프로텍터가 적용된 실행파일을 복구하기 위한 전체적인 과정을 나타낸다. 코드 난독화의 경우 별도의 옵션 없이 기본적으로 적용되므로 코드 난독화 적용여부를 검사하는 과정은 존재하지 않는다. 코드 난독화 이외의 분석방해 기술들의 적용여부를 검사하고, 우회방안을 이용해 무력화 한 다음 코드 난독화에 대응하는 OEP 탐지를 진행한다. 이후 프로텍터가 생성해 추가한 영역인 SFX(Self-extracting archive) 영역을 제거한 다음 원본 실행파일의 메모리 영역만 덤프해 실행파일을 재조합하면 실행파일 복구가 완료된다.

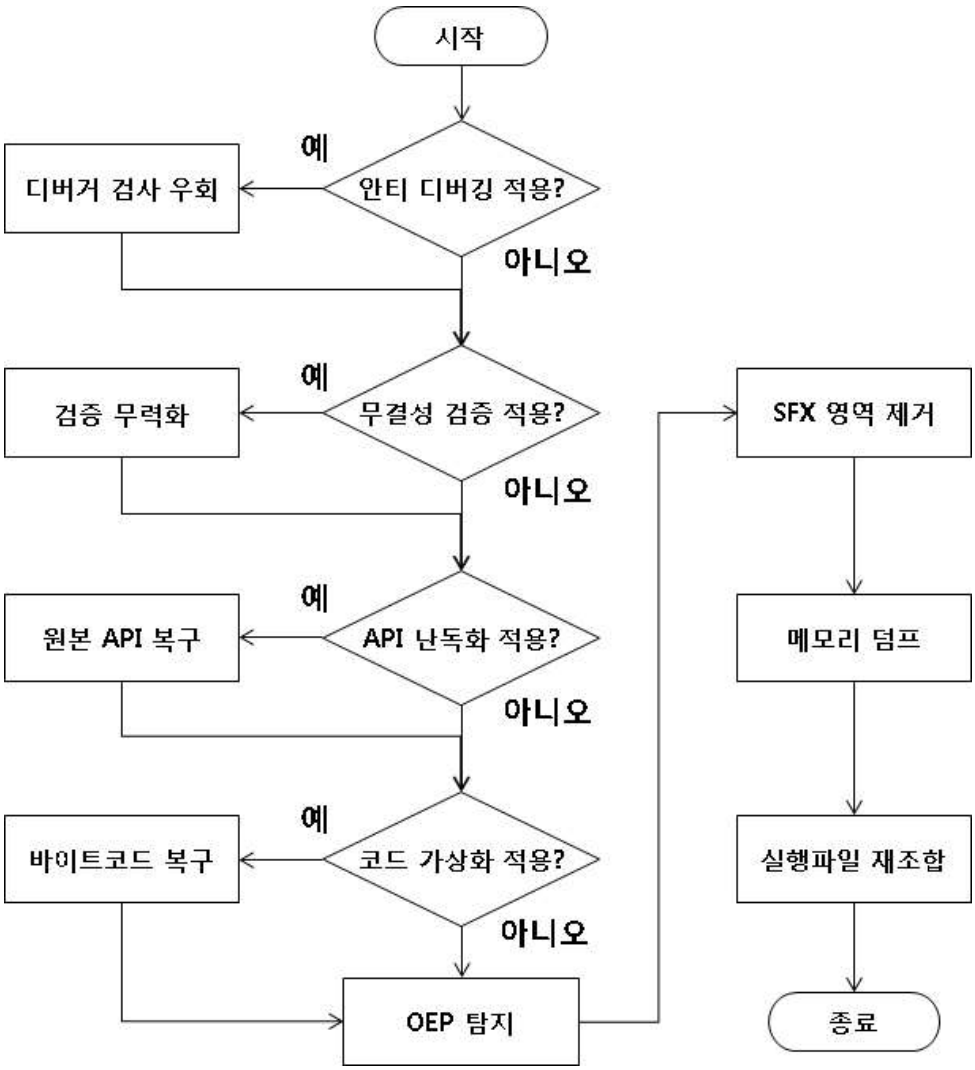

그림 9. 프로텍터가 적용된 실행파일 복구 과정

3.1. 코드 난독화 우회방안

실행파일에 프로텍터를 적용하기 전과 후의 실행파일 구조를 비교하면 그림 10.과 같다. 먼저 원본 실행파일의 섹션은 메모리상에서 차지하는 위치는 변하지 않고 암호화가 적용된다. 만약 주소 재배치(relocation)[18]를 지원하지 않는 프로텍터의 경우, 실행파일을 로드하는 주소가 고정되지 않은 경우에 포함하는 주소 재배치 섹션은 제거한다. 주소 재배치 섹션을 제외한 나머지 원본 실행파일이 포함하고 있던 섹션들은 모두 압축 또는 암호화되며, 암호화 된 원본 실행

파일의 섹션을 복구하기 위한 실행압축 해제 코드를 저장하고 있는 SFX 섹션을 추가한다.

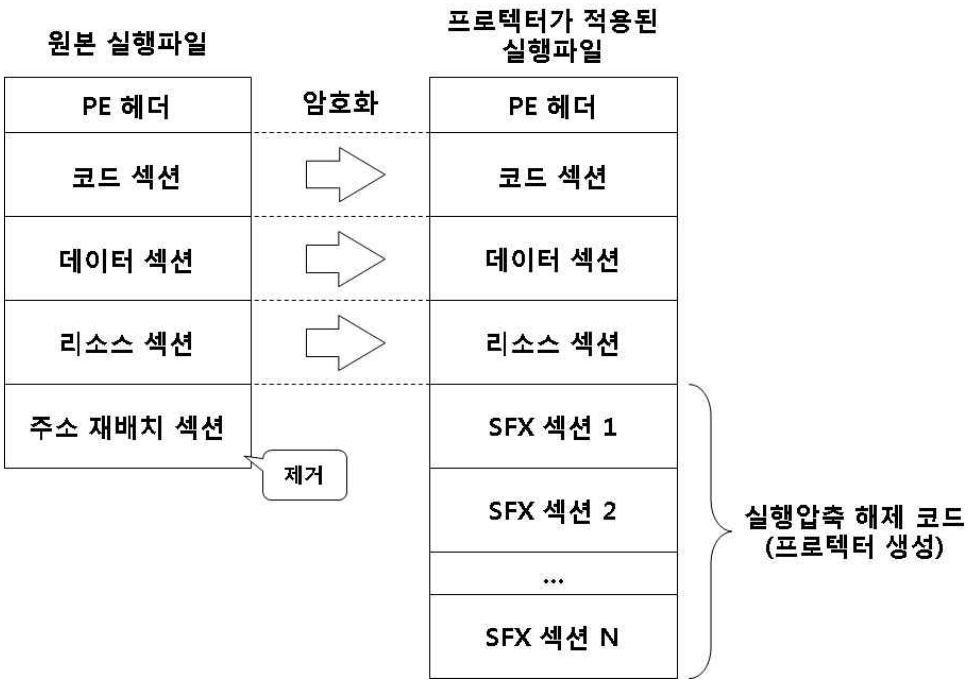

그림 10. 프로텍터 적용 전과 후의 실행파일 구조 비교

실행파일에 기본적으로 적용되는 코드 난독화의 경우 프로텍터가 생성한 실행압축 해제 코드 영역에 한해 적용되어 있다. 따라서 실행압축 해제 과정을 거쳐 복구되는 원본코드의 형태는 손상되지 않는다. 만약 실행압축 해제 과정이 끝난 후 복구된 원본코드를 최초로 실행하는 OEP(Original Entry Point)를 찾아 해당 위치에서 실행을 정지시킨 후 메모리를 덤프하면 코드 난독화가 해제된 원본코드를 획득할 수 있다. 표 6.은 프로텍터가 적용된 동일한 실행파일을 실행했을 때 프로텍터가 생성한 영역이 가지는 해시 값을 나타낸다. 코드 난독화의 영향으로 매 실행마다 서로 다른 해시 값을 가지는 것을 알 수 있다. 표 7.은 실행압축 해제가 완료되고, OEP에서 실행을 정지한 상태에서 원본코드 영역이 가지는 해시 값을 나타낸다. 코드 난독화의 영향을 받지 않아 해시 값은 동일한 것을 알 수 있다. 따라서 실행압축 해제가 끝난 후 OEP 지점에서 획득한 메모리를 이용하면 코드 난독화를 무력화 할 수 있다.

표 6. SFX 영역 해시 값 비교

#	SFX 영역 해시 값 (MD5)
1	c8d98571782072efd73012f1888e0fc9
2	ef9e9794ba08a76257e0d4783c2ed558
3	86da11d1dfa81d2c0c6dba3f32202ba3

표 7. 원본코드 영역 해시 값 비교

#	원본코드 영역 해시 값 (MD5)
1	3d1fd6bf7803781bdacec5279b62cc73
2	3d1fd6bf7803781bdacec5279b62cc73
3	3d1fd6bf7803781bdacec5279b62cc73

3.2. 안티 디버깅 우회방안

안티 디버깅은 디버거 탐지에 사용되는 기술들이 이용하는 정보들을 조작하면 우회가 가능하다. 프로세스 메모리에서 사용자 영역에 존재하는 시스템 구조체가 가지는 데이터를 확인하는 안티 디버깅의 경우 해당 데이터의 값을 디버거가 연결되지 않은 상태의 프로세스가 가지는 데이터 값으로 조작하면 탐지를 무력화하게 된다. 그리고 프로세스 메모리의 커널 영역에 존재하는 데이터인 경우, 사용자 영역에서는 권한의 문제로 직접적으로 대상 데이터에 접근이 불가능하다. 따라서 커널 영역에 존재하는 데이터를 검사하는 안티 디버깅 기술은 커널 영역의 데이터를 시스템에 요청하는 API를 호출한다. 이 경우에는 디버거 탐지여부가 API에 의존적이므로, 후킹을 통해 API를 변경하거나 직접 API에 접근해 코드를 패치하는 방식으로 무력화할 수 있다. 그림 11.은 프로세스의 메모리에 존재하는 구조체인 PEB에서 디버깅 상태인지를 나타내는 데이터인 BeingDebugged[19]

의 값을 보여준다. 일반 프로세스의 경우 0으로 설정되어 있지만, 디버깅 상태의 프로세스는 1으로 설정되어 있다. 만약 디버깅 상태의 프로세서에서 해당 구조체에 접근해 값을 0으로 설정하게 되면 디버깅 상태를 탐지할 수 없게 된다.

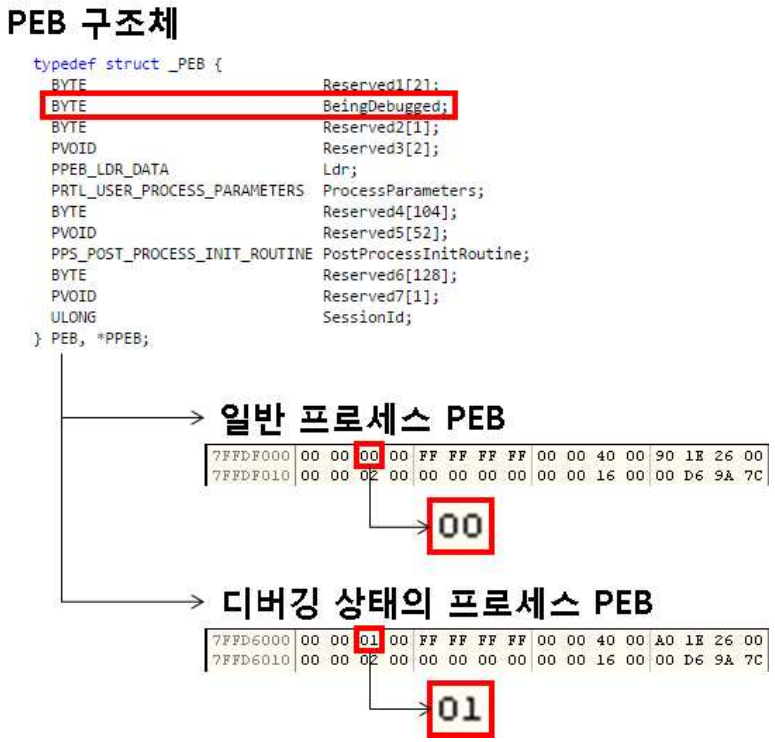

그림 11. 상태에 따른 BeingDebugged 값

3.3. 무결성 검증 우회방안

무결성 검증의 경우 검사대상 메모리 영역의 데이터를 직접 읽어와 체크섬 검사를 한다. 무결성 검증의 과정은 그림 12.와 같이 검사대상 메모리 영역의 데이터에 접근해 체크섬을 계산하는 작업을 우선적으로 진행하고, 계산된 체크섬을 미리 저장하고 있던 정상 체크섬과 비교해 다르면 무결성이 훼손된 것으로 판단한다. 따라서 저장하고 있는 정상 체크섬의 값을 코드패치한 상태의 체크섬 값으로 변경하거나, 체크섬 비교 이후의 분기문을 무조건 정상인 상태로 가도록 수정하면 무력화할 수 있다.

그림 13.은 무결성 검증 과정에서 검사대상 데이터에 접근해 체크섬을 계산하고, 검사하는 코드를 보여준다. 데이터 영역이 미리 저장해놓은 정상 체크섬 값과 데이터 접근을 통해 직접 계산한 체크섬 값을 비교한 결과에 따라 수행하는 명령이 달라진다. 이때 무조건 체크섬이 동일한 경우에 수행하는 명령의 흐름을 따라 실행하도록 분기문을 수정하면 무결성 검증이 무력화 된다.

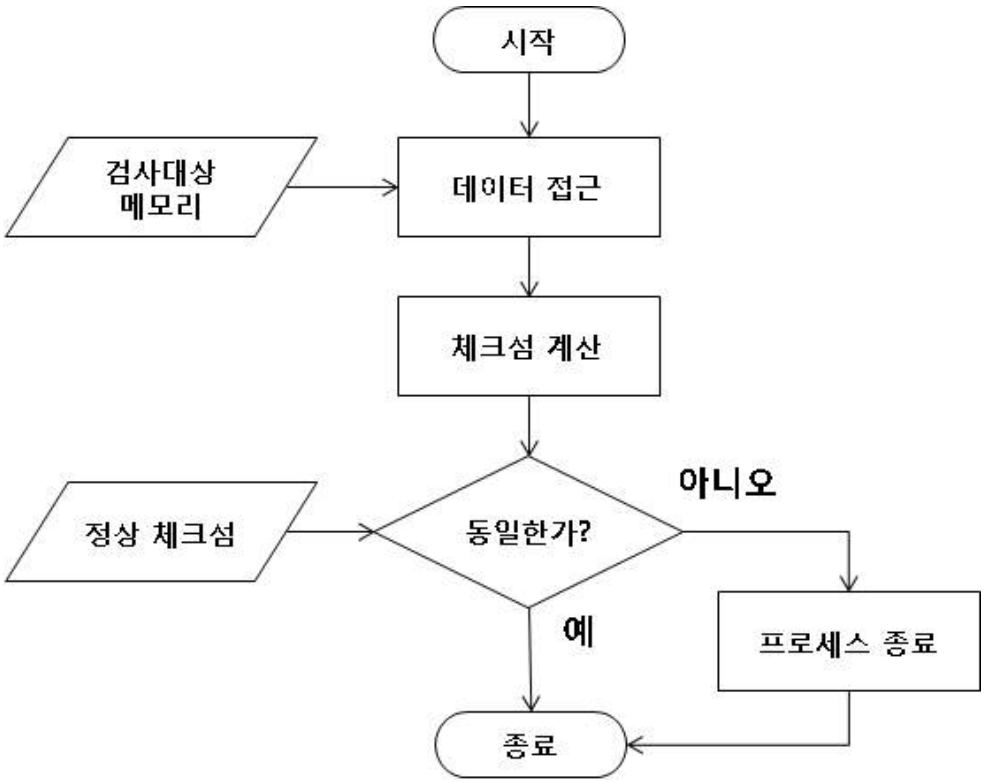

그림 12. 무결성 검증 과정

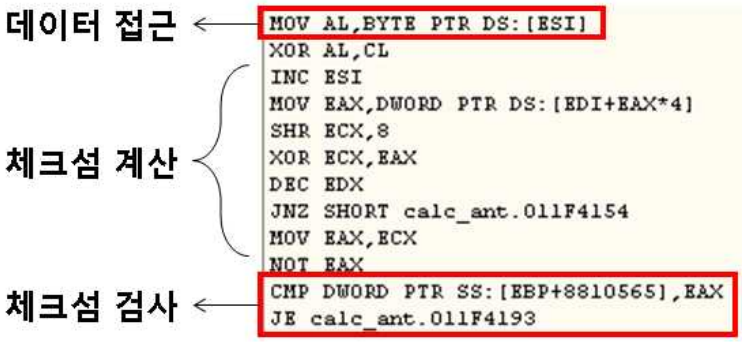

그림 13. 무결성 검증 코드

3.4. API 난독화 우회방안

API 난독화는 현재 실행중인 시스템에 존재하는 라이브러리의 API를 사용한다. API 난독화는 API를 호출하는 난독화 된 코드를 생성하는 유형과, 직접 API를 가져와 난독화한 다음 새로 할당받은 메모리 영역에 저장해 사용하는 유형이 있다. 이때 사용할 라이브러리 API의 주소를 저장하는 IAT에는 원본 API의 주소 대신 생성한 코드의 주소를 저장한다. 또한 분석을 더욱 어렵게 만들기 위해 코드 영역에서 IAT를 거치지 않고 직접 난독화 된 API를 호출하도록 명령어를 변경하기도 한다. 이때 API 난독화 된 함수를 호출하도록 변경하는 작업은 원본 실행파일의 코드영역이 대상이므로, API 난독화가 적용된 실행파일의 경우 코드영역의 해시 값이 원본 실행파일과 달라진다. 따라서 API 난독화의 완전한 무력화를 위해서는 코드영역의 복구 작업이 필요하다. 표 8.은 원본 실행파일과 프로텍터를 적용한 파일의 코드영역 해시 값을 나타낸다. 원본과 비교했을 때 모두 해시 값이 다른 것을 알 수 있다.

표 8. 프로텍터 적용 후 해시 값 비교

적용 프로텍터	해시 값 (MD5)
원본	d7588fb4f396eba902a38a4fbfcef18d
ASProtect	2385731708d6edb264357af679d8d66f
Enigma	4e72fd562cca0697268e6c5004b22ff8
Obsidium	98045cd927552969d35e7f8b86c48af9
Themida	4863c6afb8370fd7402e10beb5eace8b
VMProtect	8ceb1e1d718fcbdf929fb57bc0109cc0

변경된 코드영역의 명령어를 확인하면 표 9.와 같이 IAT에 저장된 외부 라이브러리 API의 정보를 참조하는 명령어가 변경된 것을 알 수 있다. 특정 메모리 주소에 저장된 데이터를 참조하는 명령어에서

직접적으로 데이터를 사용하는 형태로 변경하였다.

API 난독화를 무력화하려면 우선적으로 원본 API와 난독화 된 주소 간의 관계를 파악해야 한다. API 난독화가 적용된 영역을 사용하는 과정을 분석하면 난독화 된 주소와 원본 API의 관계 파악할 수 있다. 이 과정을 통해 획득한 정보를 이용하면 IAT에 저장된 주소를 원본 API의 주소로 복구하고, 코드영역에서 직접 난독화 된 API를 호출하는 명령어를 복구할 수 있다.

표 9. 프로텍터가 변경한 명령어 비교

프로텍터	변경 전	변경 후
ASProtect	CALL [Image+0x2008]	CALL 0x1AD'0000
Enigma	CALL [Image+0x208C]	CALL [0xE7'F068]
Obsidium	MOV ESI, [Image+0x2004]	MOV ESI, 0x7C94'3405
Themida	CALL [Image+0x2008]	CALL 0x266'0103
VMProtect	CALL [Image+0x208C]	CALL Image+0x5'D05C

그림 14.는 API 난독화 과정에서 원본 API와 난독화 된 API 간의 관계를 파악하는 방법을 나타낸다. 먼저 새롭게 할당된 메모리의 접근권한을 검새해 난독화 한 API를 저장해기 위해 생성한 메모리 영역을 탐지한다. 일반적인 코드영역인 경우 읽기, 실행 권한을 가지고, 데이터 영역의 경우 읽기, 쓰기 권한을 가진다. 하지만 난독화 된 코드영역의 경우 쓰기권한을 통해 난독화 된 코드를 저장하고, 난독화 된 코드의 실행을 위해 읽기, 쓰기 권한이 필요하다. 따라서 난독화를 위해 할당한 메모리 영역의 접근권한은 읽기, 쓰기, 실행 모두 가지는 특징을 이용해 난독화를 위해 생성한 메모리 영역을 구분할 수 있다. 탐지된 메모리의 주소는 lMem 리스트에 저장한다.

메모리 복사가 발생한 경우 복사 목적지의 주소(copyDst)가 lMem 리스트에 속하는지 검사하고, 만약 리스트에 등록된 주소라면 복사 대상의 주소(copySrc)가 API인지 검사한다. 복사 대상의 주소가 API로 확인되면 복사 대상 주소와 복사 목적지 주소를 1대 1로 연결해

저장하는 사전(dictionary)인 dictWrap에 저장한다. 이후 원본 코드영역에서 직접 난독화 된 API를 호출하는 명령어를 복구할 때 dictWrap에 저장된 정보를 활용할 수 있다.

분기가 발생한 경우 분기 출발지의 주소(branchSrc)와 분기 목적지의 주소(branchDst)를 검사해 원본 코드영역에서 외부의 다른 영역으로 이동하는지를 검사한다. 만약 외부로 이동한다면 현재 분기의 출발지 주소를 난독화 영역을 호출하는 출발지로 간주하고 obsBranchSrc에 저장한다. 만약 분기 목적지의 주소가 원본 코드영역 내부를 가리키고 있다면 API 난독화가 적용된 영역을 호출하는 것이 아니므로 obsBranchSrc를 NULL로 초기화한다. 현재 분기에서 obsBranchSrc가 NULL이 아니고, 분기 목적지의 주소가 API임을 확인하면, obsBranchSrc가 최종적으로 해당 API를 호출하는 코드를 담고 있는 난독화 영역으로 판단해 obsBranchSrc가 정상적으로 IAT를 참조해 API를 호출하는 형태로 명령어를 복구한다.

메모리 쓰기가 발생한 경우 우선 쓰기 목적지의 주소(writeDst)가 IAT에 속하는지 검사한다. IAT에 속한 경우 IAT를 복구하는 과정으로 판단하여 IAT 상의 주소와 저장하는 함수 주소 정보를 1대 1로 연결해 저장하는 사전인 dictIat에 저장한다. 코드 영역에서 IAT를 참조해 함수를 호출하는 경우에 정상 API를 호출하도록 활용할 수 있다. 만약 목적지의 주소가 IAT에 속하지 않고, 원본 코드영역에 속하는 경우 난독화 된 API를 직접 호출하도록 명령어를 수정하는 과정으로 판단한다. 해당 위치에서 호출하려는 주소를 dictWrap에서 검색해 원본 API를 찾고, 해당 API를 저장하고 있는 IAT 상의 주소를 dictIat에서 찾아 IAT를 참조해 API를 호출하도록 복구한다.

그림 15.는 제안하는 API 난독화 복구 방법을 이용해 난독화 정보를 획득한 결과이다. IAT 주소별로 저장된 난독화 된 API의 주소와 원본 API의 정보가 표시된 것을 볼 수 있다. 또한 해당 난독화 된 API를 직접 호출하도록 변경된 명령어의 주소 또한 참조하는 IAT 정보 하단에 표시되어 있다. 획득한 정보를 토대로 데이터를 고치면 프로텍터가 적용된 실행파일에서 API 난독화가 해제 된 실행파일을 복구할 수 있다.

lastEvt := 최근에 발생한 이벤트
newMem := 최근에 할당받은 메모리
copySrc := 복사 대상의 주소
copyDst := 복사 목적지의 주소
writeDst := 쓰기 목적지의 주소
branchSrc := 분기 출발지의 주소
branchDst := 분기 목적지의 주소
obsBranchSrc := 난독화 영역 호출 출발지의 주소
writeVal := 메모리에 쓸 데이터
nameApi := API의 이름
sIat := IAT 테이블의 범위
sCode := 코드영역의 범위
lMem := 메모리 영역을 저장하는 리스트

dictWrap := 메모리 주소와 API의 관계를 저장하는 사전
dictIat := IAT 상의 주소와 API의 관계를 저장하는 사전
GetLastEvent() : 가장 최근에 발생한 이벤트 반환
GetApiName() : 주어진 주소의 API 이름 반환
GetCallDest() : 주어진 주소의 CALL 명령어가 호출하는 주소 반환
GetBranchSrc() : 현재 분기 명령어의 출발지 주소 반환
GetBranchDst() : 현재 분기 명령어의 목적지 주소 반환
FixCall() : 주어진 주소의 함수 호출 명령어를 IAT를 참조하도록 변경

```
while lastEvt := GetLastEvent()
  if lastEvt = EVT_ALLOC_MEMORY then
    if newMem.protect = READ | WRITE | EXECUTE then
      lMem.append(newMem)
  else if lastEvt = EVT_MEM_COPY then
    if copyDst ∈ lMem then
      nameApi := GetApiName(copySrc)
      if nameApi != API_UNKNOWN then
        dictWrap[copyDst] := copySrc
  else if lastEvt = EVT_BRANCH then
    branchSrc := GetBranchSrc()
    branchDst := GetBranchDst()
    if branchDst < sCode.base and branchDst >= sCode.end then
      if branchSrc >= sCode.base and branchSrc < sCode.end then
        obsBranchSrc := branchSrc
    else obsBranchSrc := NULL
    if obsBranchSrc != NULL then
      if GetApiName(branchDst) != API_UNKNOWN then
        FixCall(obsBranchSrc, dictIat[branchDst])
  else if lastEvt = EVT_MEM_WRITE then
    if writeDst >= sIat.base and writeDst < sIat.end then
      dictIat[writeVal] := writeDst
    else if writeDst >= sCode.base and writeDst < sCode.end then
      if GetCallDest(writeDst) ∈ dictWrap then
        FixCall(writeDst, dictIat[dictWrap[GetCallDest(writeDst)]])
```

그림 14. API 난독화 과정의 정보 파악 및 복구 방법

```
[483F42] [IAT] 402000 <- 28B0000 ( kernel32!IsDebuggerPresent ( 7C813123 ) )
[4B407B] --- 40173F
[443AE2] --- 40173F
[483F42] [IAT] 402004 <- 28B0045 ( ntdll!RtlDecodePointer ( 7C943405 ) )
[483F42] [IAT] 402008 <- 28B00B7 ( kernel32!GetSystemTimeAsFileTime ( 7C8017E9 )
[4B407B] --- 4014A5
[443AE2] --- 4014A5
[483F42] [IAT] 40200C <- 28B0317 ( kernel32!GetCurrentThreadId ( 7C8097B8 ) )
[4B407B] --- 4014B4
[443AE2] --- 4014B4
[483F42] [IAT] 402010 <- 28B0361 ( kernel32!GetCurrentProcessId ( 7C8099B0 ) )
[4B407B] --- 4014BD
[443AE2] --- 4014BD
[483F42] [IAT] 402014 <- 28B03CC ( kernel32!QueryPerformanceCounter ( 7C80A4B7 )
[4B407B] --- 4014CA
[443AE2] --- 4014CA
[483F42] [IAT] 402018 <- 28B0722 ( ntdll!RtlEncodePointer ( 7C9433DF ) )
[4B407B] --- 4010B3
[443AE2] --- 4010B3
[4B407B] --- 40152C
[443AE2] --- 40152C
```

그림 15. 제안하는 방법을 이용해 획득한 난독화 정보

3.5. 코드 가상화 우회방안

프로텍터가 가지는 기본적인 분석방해 기술들은 어느 정도 우회할 수 있는 방안이 마련되었다. 그러나 코드 가상화의 경우 원본 코드가 훼손되고 가상 CPU를 통해 실행 구조를 파악하기가 어려워 이를 해결하기 위한 다양한 연구가 진행되고 있다[20][21][22][23]. 코드 가상화는 가상화 적용 대상 실행파일에 가상 CPU를 생성해 추가하고, 사용자가 지정한 사용자정의 함수 코드를 가상 CPU가 해석할 수 있는 바이트코드로 변환한다. 코드 가상화를 통해 변환한 바이트코드는 가상화 영역 내부에 저장하고, 코드 가상화가 적용된 실행파일을 실행하면 가상 CPU가 바이트코드를 읽으며 연산을 진행한다. 따라서 일반적인 실행파일의 실행흐름과는 달리 가상 CPU 내부에서 바이트코드를 해석하고 알맞은 핸들러를 호출하는 형태가 반복적으로 나타난다. 그림 16.은 Themida로 실행파일에 코드 가상화를 적용한 후 가상화 영역 내부의 실행흐름을 그래프로 나타낸 결과이다. 왼쪽의 가상화 대상 사용자정의 함수의 분기가 없는 단순한 구조가 오른쪽의 코드 가상화가 적용된 영역의 실행흐름과 같이 복잡한 형태로 변한 것을 알 수 있다. 코드의 흐름이 복잡하게 얽혀있는 중앙 부분이 가상 CPU의 영역임을 유추할 수 있다.

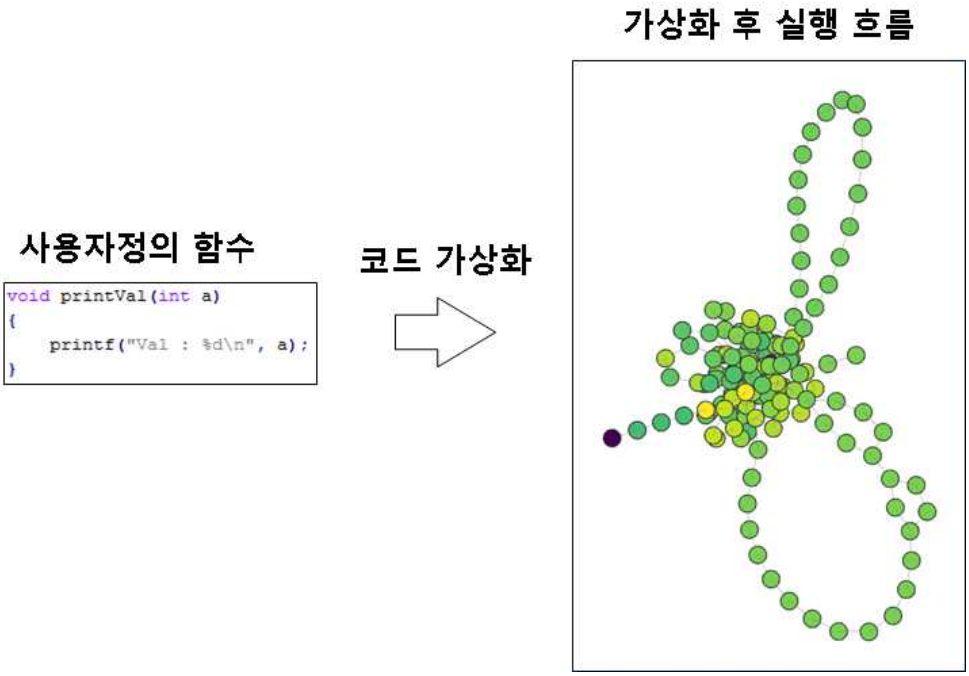

그림 16. 코드 가상화로 인한 실행흐름 변화

코드 가상화가 적용된 실행파일은 가상화를 적용한 코드를 제외하면 코드영역은 원본과 동일한 데이터를 가지고 있다. 그림 17.은 프로텍터의 코드 가상화 기술을 실행파일에 적용하기 전과 후의 코드 영역 데이터 차이를 보여준다. 가상화 적용 대상 코드 이외의 코드는 모두 원본과 동일하며, 가상화 적용 대상 코드가 있던 영역에는 가상화 영역으로 이동하는 JMP 명령어가 존재한다. 그리고 가상화 대상 코드 영역에서 JMP 명령어로 덮어씌워진 부분을 제외한 나머지 명령어는 훼손되어 사용할 수 없게 된다. 만약 바이트코드를 해석해 가상화가 적용되기 전의 코드를 유추할 수 있다면, 가상화 영역에서의 작업을 실행 가능한 형태의 기계어 코드로 복구하고, 코드 가상화 영역을 제거해 원본 실행파일을 복구할 수 있을 것이다.

코드 가상화를 우회해 실행파일을 복구하기 위한 단계는 표 10.과 같다. 우선 분기를 통해 이동하는 주소를 이용해 복구 대상이 되는 가상화 영역을 실행파일 내부에서 식별하고, 가상화 영역에서 사용하는 정보들의 위치를 찾아내 가상화 내부 구조를 파악한다. 파악한 가상화 내부 구조에서 바이트코드 실행과 관련된 정보들을 분석해 각

바이트코드가 가지는 의미를 찾아내고, 바이트코드를 실행 가능한 형태의 기계어로 복구한다. 코드 가상화가 적용된 실행파일에서 가상화 영역을 제거한 후, 복구한 기계어를 실행파일에 추가하면 코드 가상화 기술을 제거한 실행파일을 획득할 수 있다.

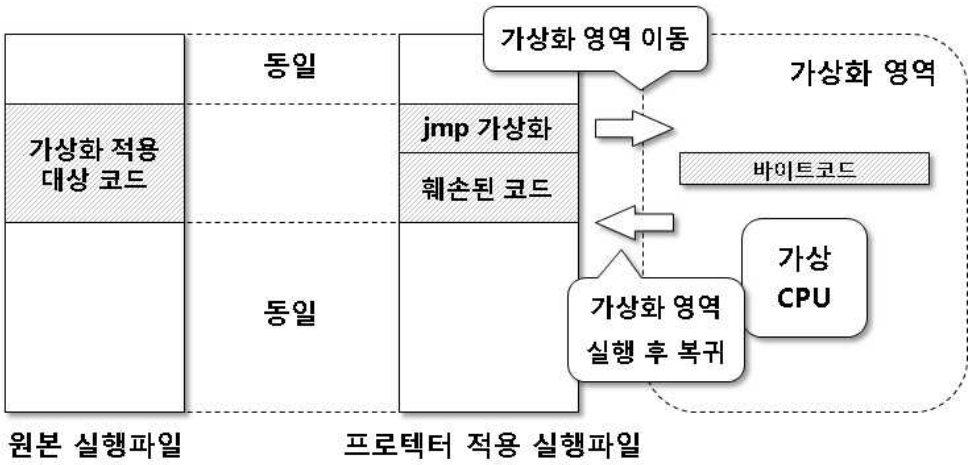

그림 17. 코드 가상화 적용 전과 후의 코드영역 비교

표 10. 코드 가상화 우회방안

#	우회 단계	우회 방법
1	가상화 영역 식별	가상화 영역으로 이동하는 분기점 식별
2	가상화 내부 구조 파악	바이트코드, 핸들러, 가상 레지스터 등의 위치 파악
3	바이트코드 의미 파악	각 바이트코드에 대응하는 핸들러의 작업 분석
4	바이트코드 복구	핸들러 분석 내용에 기반해 바이트코드를 기계어로 복구
5	실행파일 복구	복구한 기계어를 포함한 실행파일 복구

가. 가상화 영역 식별

코드 가상화 적용대상 코드는 바이트코드로 변환되어 가상화 영역에서 실행하도록 변화한다. 따라서 바이트코드를 실행하기 위해서는 반드시 가상화 영역으로 진입하게 된다. 그림 18.은 코드 가상화를 통해 변한 사용자함수 코드의 변화를 보여준다. 가상화 적용 전에 보이던 코드는 가상화 적용 후 가상화 영역으로 실행흐름을 넘기는 JMP 명령어만 남기고 나머지는 모두 훼손되어 쓰레기 값으로 채워진 것을 알 수 있다. 이때 JMP 명령어가 존재하는 영역은 원본 실행파일이 가지고 있던 코드영역 내부지만, JMP 명령어를 통해 이동하는 영역은 원본 실행파일이 가지고 있던 영역이 아니라 프로텍터가 생성한 가상화 영역이다. 따라서 JMP나 CALL과 같은 명령어를 통해 원본 실행파일의 코드영역에 속하는 주소에서 프로텍터가 생성한 영역의 주소로 이동한다면 가상화 영역으로 진입하는 시점이라 판단할 수 있다.

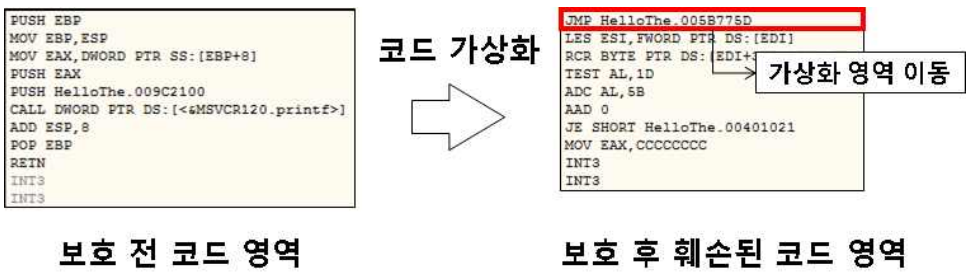

그림 18. 코드 가상화로 인해 훼손된 원본코드 영역

나. 가상화 내부 구조 파악

가상 CPU는 생성한 바이트코드를 처리할 핸들러를 테이블 형식으로 저장하고 있으며, 생성한 바이트코드는 가상화 영역 내부에서 확인할 수 있다. 이때 프로텍터가 생성한 각각의 핸들러가 수행하는 연산의 의미를 파악할 수 있다면, 파악한 결과를 토대로 원본코드를 복구할 수 있을 것이다. 핸들러가 처리하는 바이트코드의 크기를 확인

해 보면 바이트코드의 크기가 일정하지 않은 것을 알 수 있다. 그림 19.는 Themida에서 가상화 영역에 저장하고 있는 바이트코드와 핸들러 테이블을 보여준다. Themida의 가상화 영역에서 EBP를 기준으로 특정 거리만큼 떨어진 주소를 따라가면 각각의 정보가 나온다. 이때 EBP를 기준으로 특정 정보를 접근하는 것은 동일하지만 바이트코드와 핸들러 테이블이 EBP와 떨어진 거리, 각각의 핸들러에 대응하는 바이트코드의 데이터는 실행파일에 가상화를 적용할 때 마다 무작위로 생성된다.

그림 19. 가상화 영역의 바이트코드와 핸들러 테이블

가상화 영역에서 EBP를 기준으로 떨어진 거리는 무작위로 생성되지만, EBP를 참조해 위치를 찾아가는 것은 고정되어 있다. 따라서 바이트코드를 읽고 필요한 핸들러로 이동하는 분기가 발생한다면, 이동할 주소를 찾을 때 반드시 EBP를 참조할 것을 예상할 수 있다. 만약 분기 명령어를 기준으로 이전에 실행된 명령어의 순서를 분석한다면, 이동할 주소를 찾는 작업에 EBP가 참조되었는지를 알 수 있을 것이다. EBP를 참조한 경우 가상화 영역에서 핸들러를 찾는 작업으로 간주하고, 참조한 영역의 정보를 분석하면 핸들러 테이블의 위치와 바이트코드의 위치를 알 수 있을 것이다. 그림 20.은 분기 명령어를 기준으로 이전에 실행된 명령어의 순서를 통해 사용한 레지스터와 메모리의 연산 정보를 기록한 결과를 보여준다. JMP 명령어를 통한 분기에서 ESI 레지스터를 사용하고 있다. 이때 기록해 둔 연산 정보를 보면 EBP를 참조해 ESI의 값을 정하는 것을 알 수 있다. 연산 정보를 보면 EBP+Ah에 위치한 데이터를 읽어 몇 가지 연

산을 거친 다음 최종적으로 EBP+8Eh에 위치한 데이터에서 ESI의 값을 정한다. 따라서 본 가상화 코드 샘플에서는 EBP+8Eh에 위치한 데이터는 핸들러 테이블을 가리키고 있으며, EBP+Ah에 위치한 데이터는 실행할 바이트코드를 가리키고 있는 가상 프로그램 카운터(PC)인 것을 알 수 있다.

실제로 직접 분석해보면 위의 데이터들은 핸들러 테이블과 바이트코드를 가리키고 있으며, 분기 주소인 004534CAh는 핸들러 테이블에 존재하는 주소이다. 또한 접근한 메모리 중 0040Fxxxh 주소대의 데이터는 가상화 영역에서 사용하는 변수 또는 가상 레지스터를 나타낸다.

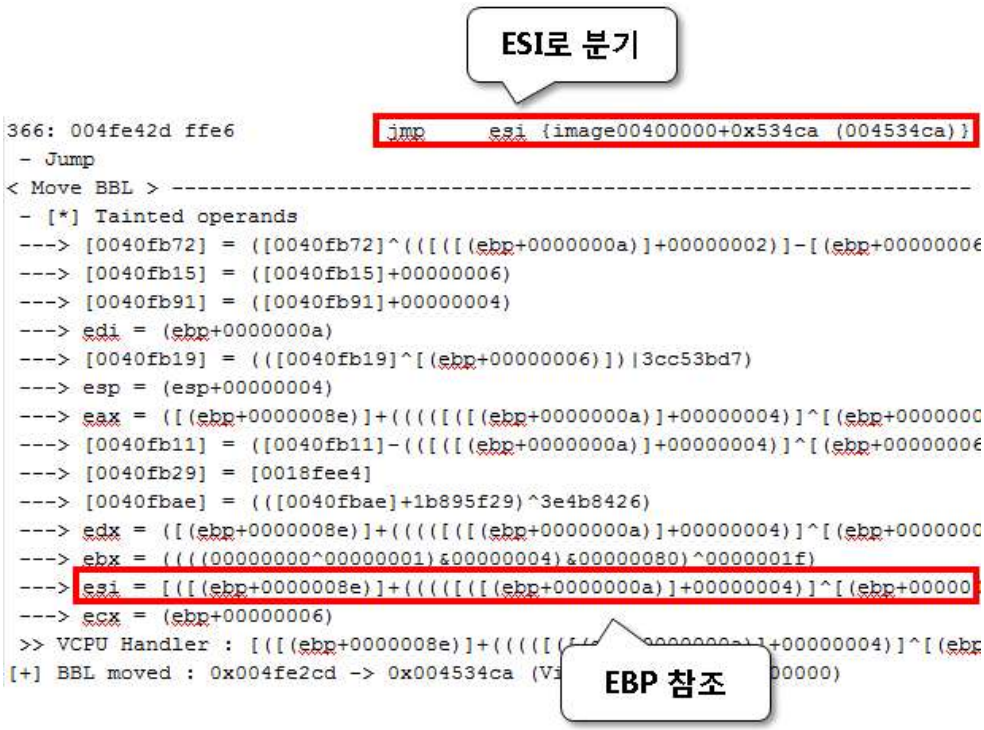

그림 20. 분기 명령어를 기준으로 기록한 연산 정보

다. 바이트코드 의미 파악

찾아낸 핸들러 테이블에 위치한 핸들러는 각각 하나의 바이트코드가 나타내는 가상화 명령어와 대응될 것이다. 따라서 앞서 찾아낸 분

기별 실행된 명령어의 정보를 분석하면 각 분기별로 하나의 가상화 된 명령어를 찾을 수 있을 것이다. 그림 21.은 가상화 코드에서 PUSH 명령어의 연산 정보를 보여준다. PUSH를 통해 가상화 영역의 변수인 0040FB72h에 저장된 데이터를 스택에 저장하고, ESP의 정보를 저장하고 있는 가상화 영역의 변수인 0040FB91h에 저장된 정보를 변경한다. 따라서 0046AE98h의 핸들러는 PUSH 명령어인 것을 알 수 있다.

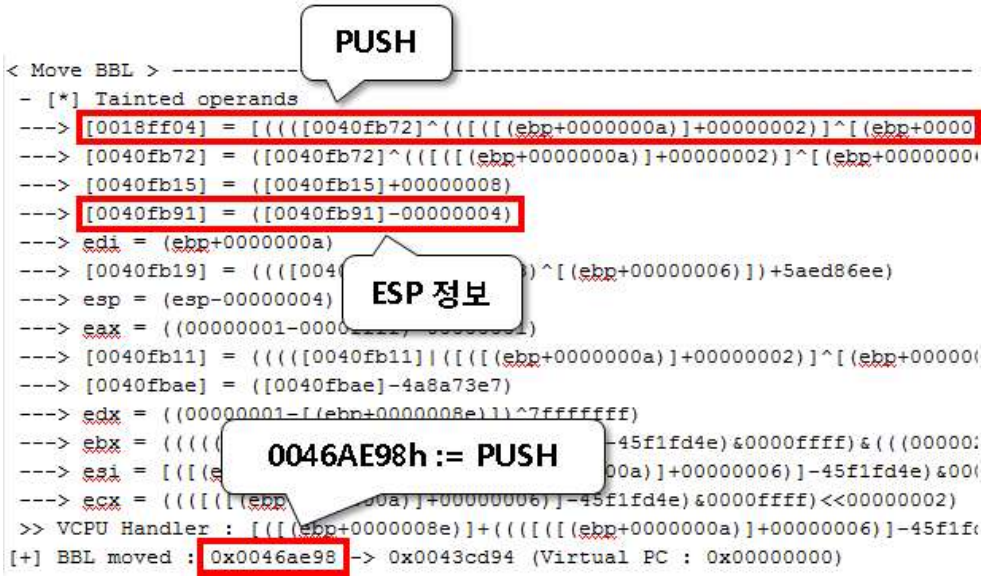

그림 21. 가상화 코드의 PUSH 명령어

라. 바이트코드 복구 및 실행파일 복구

핸들러의 연산 정보를 분석해 각 바이트코드가 나타내는 명령어를 찾으면, 가상화 영역에서 실행되는 전체 명령어를 알 수 있다. 가상화 된 명령어의 전체 연산을 간소화 하고, 간소화한 결과를 실행 가능한 형태의 기계어로 복구해 원본코드가 존재했던 영역에 저장한다. 가상화 영역에서 실행되는 명령어를 복구하였으므로, 더 이상 가상 CPU를 사용할 필요가 없다. 따라서 프로텍터가 생성한 가상화 영역을 제거하면, 코드 가상화가 적용된 실행파일에서 가상화를 제거한 실행파일을 복구할 수 있다. 그림 22.는 바이트코드에서 복구한 기계

어를 실행파일에 추가하는 것을 보여준다. 프로텍터가 적용된 실행파일의 코드영역을 훼손하지 않고 중간에 복구한 기계어 코드를 추가하는 것은 어려우므로 코드영역의 뒷부분에 복구한 기계어를 추가하는 것을 보여준다.

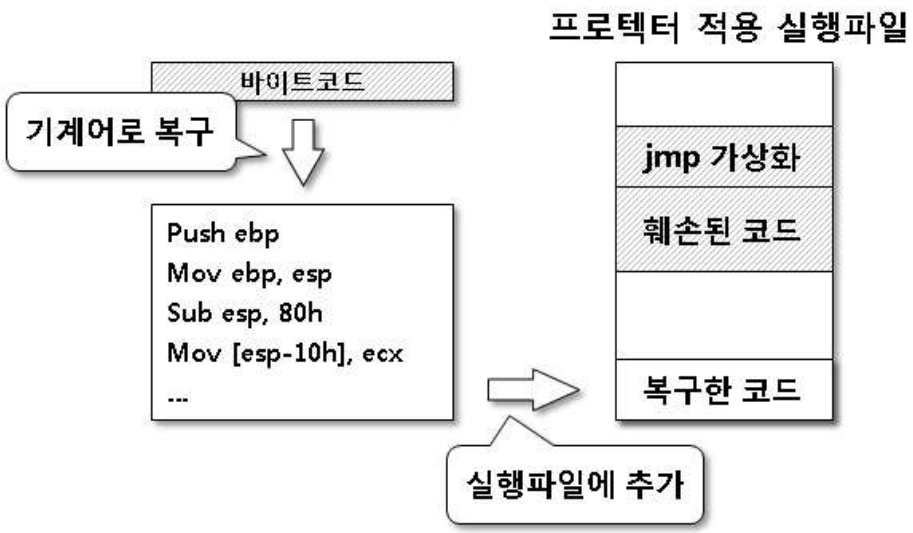

그림 22. 복구한 기계어를 이용해 실행파일 복구

제 4 장 프로텍터 분석방해 기술 우회결과

4.1. 실험 환경

실행파일에 프로텍터를 적용한 후, 제안한 기법을 활용해 실행파일을 복구하였다. 안티디버깅을 무력화하기 위한 디버거 검사 우회는 커널 레벨에서의 작업이 필요하지만 Windows Vista 이후의 버전은 드라이버를 설치하려면 전자서명을 받아야 한다[24]. 하지만 실행파일을 복구하는 과정에 OS의 버전이 영향을 미치지는 않으므로 드라이버 서명이 필요 없는 Windows XP SP3 x86을 실험환경으로 사용하였다. 프로텍터 적용 대상 실행파일은 Visual Studio 2013으로 컴파일 한 실행파일이며, 실행파일에 적용한 프로텍터의 버전은 표 11.과 같다.

표 11. 실험한 프로텍터 버전

프로텍터	버전
ASProtect	2.76
Enigma	4.40
Obsidium	1.5.9
Themida	2.4.6.0
VMProtect	3.0.9

4.2. 프로텍터 분석방해 기술 우회결과

분석방해 기술을 우회하는 실험은 표 12.와 같이 진행하였다. 프로텍터가 생성한 영역과 원본 실행파일의 영역을 분리할 수 있는 코드

난독화, 안티 디버깅, 무결성 검증 그리고 API 난독화를 실행파일에 적용하고, 복구를 진행하였다. 그리고 가상 CPU가 생성되고 원본 코드영역에서 실행영역 분리가 불가능한 코드 가상화 기술만을 적용하고, 복구하는 실험은 Themida에 한해 진행하였다.

표 12. 분석방해 기술 우회 실험 유형

#	실험 유형	우회 기술	대상 프로텍터	실험 방법
1	코드영역 복구	코드 난독화 안티 디버깅 무결성 검증 API 난독화	ASProtect Enigma Obsidium Themida VMProtect	코드영역 복구 후 해시 값 비교
2	실행파일 복구		Themida	실행파일 복구 후 파일의 정보 비교
3	가상화 구조 분석	코드 가상화	Themida	생성된 가상화 영역 내부 구조 파악

가. 코드영역 복구

프로텍터가 가지는 분석방해 기술 중 코드 난독화, 안티 디버깅, 무결성 검증 그리고 API 난독화를 설정한 다음 실행파일에 프로텍터를 적용하였다. 그리고 프로텍터가 적용된 실행파일에 제안한 대응기법을 적용해 코드영역의 데이터를 복구하였다. 프로텍터가 실행파일에 적용한 분석방해 기술을 모두 우회해 무력화했으므로, 복구한 데이터 원본 데이터와 동일한 데이터를 저장하고 있다. 표 13.은 프로텍터가 적용된 실행파일의 코드영역 해시 값과 복구한 데이터 중 코드영역의 해시 값을 나타낸다. 복구한 코드영역 데이터 모두 원본 실행파일이 가지는 코드영역의 해시 값과 동일한 것을 알 수 있다. 실행파일이 실행할 때 수행하는 명령어들이 속해있는 코드영역의 데

이터가 동일하므로, 복구한 코드영역은 원본 실행파일과 동일한 작동을 한다고 말 할 수 있다.

표 13. 복구한 코드영역 해시 값 비교

프로텍터	프로텍터가 적용된 코드영역 해시 값 (MD5) 복구한 코드영역 해시 값 (MD5)
원본	7875b054b988666c770b476c6f847007
ASProtect	f64e8305317fba3c9b0e59a0b433e4cc
	7875b054b988666c770b476c6f847007
Enigma	eae6634ebc9bd1ab65034f0ed8cee5ec
	7875b054b988666c770b476c6f847007
Obsidium	c762eeeb358e6393ab7059551fa35ab2
	7875b054b988666c770b476c6f847007
Themida	5bd6f3b2f37cae007f9186cd45109459
	7875b054b988666c770b476c6f847007
VMProtect	9ab1ff81b075ddccaad656df4383c1a3
	7875b054b988666c770b476c6f847007

나. 실행파일 복구

프로텍터가 데이터를 변경하는 특징을 파악하면 복구한 데이터를 이용해 실행파일을 생성하는 것 또한 가능하다. Themida를 이용해 실행파일에 분석방해 기술을 적용하고, Themida가 적용된 실행파일에서 분석방해 기술들을 제거해 데이터를 복구하였다. 그리고 복구한 데이터를 이용해 실행파일을 재조합해 Themida의 분석방해 기술을

제거한 실행파일을 복구하였다. 원본 실행파일과 Themida가 적용된 실행파일 그리고 복구한 실행파일의 크기는 표 14.를 통해 확인할 수 있다. Themida가 생성한 영역을 제거한 결과, Themida가 적용된 실행파일의 크기에 비해 복구한 실행파일의 크기가 크게 줄어든 것을 알 수 있다. 원본 실행파일과 복구한 실행파일의 크기가 차이가 나지만, 이는 실행파일이 메모리에 올라갈 때 OS에서 메모리에 할당하는 페이지의 최소 단위인 SYSTEM_INFO.dwPageSize[25]에 맞춰 할당하는 것과 Import table 복구를 위해 추가한 IT 섹션으로 인해 발생한다.

표 14. 실행파일 크기 비교

파일 종류	파일 크기
원본 실행파일	7 KB
프로텍터가 적용된 실행파일	1,233 KB
복구한 실행파일	24 KB

표 15.는 실행중인 프로세스 상에서 원본 실행파일과 Themida가 적용된 실행파일, 복구한 실행파일이 가지고 있는 섹션들의 정보를 나타낸다. Themida가 원본 실행파일에 적용되며 주소 재배치에 사용할 .reloc 섹션은 제거하고, Themida가 적용된 실행파일의 초기 설정을 위해 사용할 함수를 Import table로 저장하는 .idata 섹션과 실행압축 해제 및 분석방해 기술을 사용하는 SFX 섹션을 추가한 것을 알 수 있다. 복구한 실행파일의 구조는 Themida가 적용된 실행파일에서 Themida가 생성한 영역을 모두 제거하고, 복구한 실행파일이 사용할 Import table을 저장하는 IT 섹션을 추가한 것으로, Themida가 적용된 실행파일에서 복구 가능한 모든 원본 실행파일 정보를 포함하고 있다.

표 15. 프로세스 상에서 실행파일 섹션 정보 비교

섹션 명	원본 실행파일		프로텍터가 적용된 실행파일		복구한 실행파일	
	RVA	크기	RVA	크기	RVA	크기
.text	1000	1000	1000	1000	1000	1000
.rdata	2000	1000	2000	1000	2000	1000
.data	3000	1000	3000	1000	3000	1000
.rsrc	4000	1000	4000	1000	4000	1000
.reloc	5000	1000	X		X	
.idata	X		5000	1000	X	
SFX00	X		6000	203000	X	
SFX01	X		209000	133000	X	
SFX02	X		33C000	1000	X	
IT	X		X		5000	1000

원본 실행파일과 복구한 실행파일이 공통적으로 포함하고 있는 섹션의 해시 값을 비교한 결과는 표 16.과 같다. Import table을 포함하고 있는 .rdata 영역을 제외한 나머지 섹션들은 모두 동일한 해시 값을 가지고 있다. Import table은 IT 섹션을 이용해 다른 영역에 복구하였으므로, 원본 실행파일에서 Import table이 속해있던 섹션의 해시 값은 차이가 발생한다. 하지만 차이가 발생하는 섹션 내부에서 Import table을 제외한 나머지 영역의 해시 값을 비교하면 동일한 것을 알 수 있다. 따라서 Themida가 적용된 실행파일에서 원본 실행파일과 동일한 작동을 하는 실행파일을 성공적으로 복구하였다고 볼 수 있다.

표 16. 실행파일 섹션 해시 값 비교

섹션 명	원본 실행파일의 MD5
	복구한 실행파일의 MD5
.text	7875b054b988666c770b476c6f847007
	7875b054b988666c770b476c6f847007
.rdata	6412e7655fc74aeed9a1ca5587755967
	26946a8d0b4018515c99a8d48d7a4b92
.rdata (No IT)	f758e59e670cd8ab60f6d9973fd100c6
	f758e59e670cd8ab60f6d9973fd100c6
.data	418e3a02c1c10ef7bccf8d496c1e325a
	418e3a02c1c10ef7bccf8d496c1e325a
.rsrc	5a0f9dc39d51e89d07df5c63d90a4db4
	5a0f9dc39d51e89d07df5c63d90a4db4

다. 가상화 구조 분석

프로텍터가 가지는 분석방해 기술인 코드 가상화를 우회하는 실험을 진행하였다. 표 17.은 Themida의 코드 가상화 기술이 적용된 실행파일들을 앞선 실험의 대응기법을 그대로 적용해 코드영역을 복구한 결과를 보여준다. 전체 코드영역의 해시 값은 차이가 나지만, 코드 가상화 기술을 적용한 코드영역을 제외하면 모두 동일한 것을 알 수 있다. 따라서 코드 가상화가 적용된 영역을 제외하면 모두 정상적으로 복구가 가능한 것을 알 수 있다. 그러므로 생성된 바이트코드를 분석해 원본 코드를 찾아내고, 코드 가상화가 적용된 영역에 저장한다면 코드 가상화 우회가 가능하다.

표 17. 코드 가상화에서 복구한 코드영역 해시 값 비교

#	전체 코드 MD5
	가상화 제외 MD5
원본 코드	7875b054b988666c770b476c6f847007
	84c798b9a8e7848b78df4f1e99938990
1	8ae38de0526fe658781d15f57c68fb23
	84c798b9a8e7848b78df4f1e99938990
2	4a441243a56e50a3de29488edbf25687
	84c798b9a8e7848b78df4f1e99938990
3	4bda8c9105dfb13a5fd030e3efbcc8cd
	84c798b9a8e7848b78df4f1e99938990

아래 표 18.은 Themida의 코드 가상화가 적용된 실행파일들에서 찾아낸 가상화 영역에서 사용하는 정보들의 EBP 기준 위치를 나타낸다. 실행할 바이트코드의 위치를 가리키는 가상 프로그램 카운터인 VPC와 바이트코드에 대응하는 핸들러들을 저장하고 있는 핸들러 테이블 그리고 가상화 영역에서 사용하는 가상 레지스터인 VR0부터 VR9까지의 위치가 가지는 EBP 기준으로 떨어진 거리를 알 수 있다.

분기 명령어가 이동할 주소를 찾는데 EBP가 사용되었는가를 기준으로 찾아낸 바이트코드 핸들러의 수는 표 19.와 같다. 각 실행파일별로 생성된 바이트코드의 수가 일정하지 않는 것을 알 수 있다. 이때 찾아낸 핸들러에 대응하는 명령어를 모두 찾아내 간소화하면, 일반적인 코드 형태로 복구가 가능할 것이다. 복구한 명령어를 원본코드가 존재했던 영역에 저장하고, 가상화 영역을 제거하면 코드 가상화가 적용된 실행파일을 복구할 수 있다.

표 18. 가상화 영역이 사용하는 데이터 위치

#	1	2	3
VPC	+A h	+76 h	+45 h
핸들러 테이블	+8E h	+89 h	+15 h
VR0	+6 h	+1D h	+87 h
VR1	+A3 h	+85 h	+2C h
VR2	+2C h	+A7 h	+3D h
VR3	+57 h	+6E h	+30 h
VR4	+16 h	+7A h	+4 h
VR5	+26 h	+32 h	+56 h
VR6	+67 h	+49 h	+6D h
VR7	+61 h	+25 h	+26 h
VR8	+E h	+C h	+39 h
VR9	+92 h	+4B h	+91 h

표 19. 찾아낸 핸들러 수

#	핸들러 수
1	148
2	177
3	129

제 5 장 결 론

현재 실행파일의 핵심 알고리즘을 보호하기 위해 실행파일에 난독화를 적용하고 분석방해 기술을 적용하는 다양한 프로텍터가 존재한다. 프로텍터가 적용된 실행파일을 정밀분석하기 위해서는 우선적으로 프로텍터를 해제해 실행파일을 복구해야 한다. 따라서 경험이 적은 분석가는 정보를 파악하기 어려워 효과적으로 실행파일의 알고리즘을 보호할 수 있다. 그러나 순수한 목적과는 달리 악의적인 목적으로 개발된 실행파일에 프로텍터를 적용해 분석을 방해하는 사례가 있어 프로텍터가 적용된 실행파일에 대응하기 위한 방안을 마련할 필요가 있다. 악성코드에 프로텍터를 적용하는 비율은 백신에서 프로텍터가 적용되고, 유효한 인증이 없는 경우 악성으로 탐지하도록 대응책을 마련하여 감소하고 있다. 그러나 범죄행위나 첩보활동에 사용되는 실행파일에서는 여전히 분석을 방해하기 위해 프로텍터를 적용하고 있다. 따라서 단순히 프로텍터 적용여부를 확인하는 것이 아닌 프로텍터가 적용된 실행파일을 복구하는 방안을 마련할 필요가 있다. 또한 프로텍터가 사용하는 분석방해 기술도 계속해서 발전하므로, 프로텍터에 대응하기 위한 연구는 꾸준히 진행되어야 할 것이다.

본 논문에서는 알려진 프로텍터가 사용하는 분석방해 기술들을 연구하고, 각 기술들을 우회하기 위한 대응방안을 기술하였다. 연구를 통해 프로텍터가 생성하는 영역과 작동방식을 분석하면 실행파일을 정밀분석 할 수 있도록 복구가 가능한 것을 알 수 있었다. 따라서 각각의 프로텍터별로 연구를 진행한다면 연구가 진행된 프로텍터에 한해서는 대응이 가능할 것이다. 그러나 모든 프로텍터를 연구하는 것은 시간소요가 상당할 것으로 예상되므로, 앞으로는 프로텍터가 보이는 보편적인 특징을 찾아내고, 공통적으로 적용할 수 있는 프로텍터 대응 알고리즘을 연구할 것이다. 또한 프로텍터에 대한 연구가 충분히 진행되면, 더 나아가 알아낸 정보를 토대로 새로운 프로텍터가 발견되더라도 대응 가능한 형태의 솔루션을 연구할 예정이다.

참 고 문 헌

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

[2] Mark Russinovich, David A. Solomon, and Alex Ionescu, “Image Loader”, Windows Internals Part 1, Sixth Edition, pp. 232-247, Microsoft press, 2012

[3] Microsoft, Microsft PE and COFF Specification, Revision 8.3, “https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx”, Feb. 2013

[4] Matt Pietrek, Microsoft, Peering Inside the PE: A Tour of the Win32 Portable Executable File Format, “PE File Imports”, “https://msdn.microsoft.com/en-us/library/ms809762.aspx“, Mar. 1994

[5] Microsoft, Microsoft Portable Executable and Common Object File Format Specification, Revision 8.2, “http://sandsprite.com/CodeStuff/Understanding_imports.html”, Sep. 2010

[6] ASPack software, ASProtect32, What is ASProtect 32 (SKE)?, “http://www.aspack.com/asprotect32.html”, Revision 2.38, Mar. 2017

[7] The Enigma Protector Software Protection, Enigma, The Enigma Protector 32-bits version, “http://enigmaprotector.com/en/downloads.html”, Revision 5.60, Feb. 2017

[8] Obsidium Software, Obsidium, Obsidium (x86), “https://www.obsidium.de/show/download/en”, Revision 1.6.0, Apr. 2017

[9] Oreans Technologies, Themida, Advanced Windows Software Protection System, “http://www.oreans.com/themida.php”, Revision 2.4, Feb. 2017

[10] VMProtect Software, VMProtect, VMProtect Features, “http://vmpsoft.com/products/vmprotect/”, Revision 3.0.9, Jul. 2016

[11] John Leitch, Process Hollowing, “www.autosectools.com/process-hollowing.pdf”, Nov. 2013

[12] Mark Russinovich, David A. Solomon, and Alex Ionescu, “Process, Threads, and Jobs”, Windows Internals Part 1, Sixth Edition, pp. 359-485, Microsoft press, 2012

[13] Microsoft, Microsoft MSDN: Taking a Snapshot and Viewing Processes, “https://msdn.microsoft.com/ko-kr/library/windows/desktop/ms686701(v=vs.85).aspx”

[14] Debugger flow control: Hardware breakpoints vs software breakpoints, “http://www.nynaeve.net/?p=80”

[15] Intel, “Debug Registers”, Intel® 64 and IA-32 architectures software developer's manual combined volumes 3A, 3B, 3C, and 3D: System programming guide, Chapter 17.2, pp. 630-634, Mar. 2017

[16] Microsoft, Microsoft MSDN: GetThreadContext function,

"https://msdn.microsoft.com/ko-kr/library/windows/desktop/ms679362(v=vs.85).aspx"

[17] Defeating Memory Breakpoints, "http://waleedassar.blogspot.kr/2012/11/defeating-memory-breakpoints.html", Nov. 2012

[18] Microsoft, Microsoft MSDN: Peering Inside the PE: A Tour of the Win32 Portable Executable File Format, "PE File Base Relocations", "https://msdn.microsoft.com/en-us/library/ms809762.aspx"

[19] Mark Vincent Yason, The Art of Unpacking, Black Hat USA 07, Feb. 2007 (also see "http://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf")

[20] Rolf Rolles, "Unpacking virtualization obfuscators", Proceedings of the 3rd USENIX Conference on Offensive Technologies, pp. 1-1, Aug. 2009 (also see "https://www.usenix.org/legacy/events/woot09/tech/full_papers/rolles.pdf")

[21] Monirul Sharif, et al. "Automatic reverse engineering of malware emulators.", Proceedings of IEEE Symposium on Security and Privacy, 2009 (also see "http://ieeexplore.ieee.org/document/5207639/")

[22] Kevin Coogan Gen Lu, and Saumya Debray. "Deobfuscation of virtualization-obfuscated software: a semantics-based approach." Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011 (also see "https://www2.cs.arizona.edu/people/debray/Publications/ccs-unvirtualize.pdf")

[23] HexEffect, “Virtual Deobfuscator, Removing virtualization obfuscations from malware - a DARPA Cyber Fast Track funded effort”, Black Hat 2013, Jul. 2013 (also see “https://media.blackhat.com/us-13/US-13-Raber-Virtual-Deobfuscator-A-DARPA-Cyber-Fast-Track-Funded-Effort-Slides.pdf”)

[24] Microsoft, Microsoft MSDN: Driver Signing, “https://msdn.microsoft.com/en-us/windows/hardware/drivers/install/driver-signing”

[25] Microsoft, Microsoft MSDN: SYSTEM_INFO structure, “https://msdn.microsoft.com/en-us/library/windows/desktop/ms724958(v=vs.85).aspx”