임포트 테이블 기반의 언패킹 알고리즘 연구

민재원*, 김호연* 정성민*, 정태명**
*성균관대학교 전자전기컴퓨터공학과
**성균관대학교 정보통신공학부
{jwmin, hykim, smjung}@imtl.skku.ac.kr*, tmchung@ece.skku.ac.kr**

A Study on the Unpacking Algorithm based on Import Table

Jae-Won Min*, Ho-Yeon Kim*, Sung-Min Jung*, Tai-Myoung Chung**
*Dept of Electrical and Computer Engineering, Sungkyunkwan Univ.
**School of Information Communication Engineering, Sungkyunkwan Univ.

요 약

최근 악성코드들은 패킹을 하여 분석을 어렵게 한다. 패킹이란, 프로그램을 압축하거나 암호화를 해서 원래 의미를 알 수 없게 하는 과정을 뜻한다. 주로 압축을 해서 악성코드의 크기를 줄이거나, 분석 시간을 지연시키기 위해 사용된다. 따라서 악성코드의 행동을 분석하고 시그니처를 생성하기 위해서는 언패킹이 필요하다. 하지만 계속해서 새로운 패커가 개발되고 다중 패킹된 악성코드들이 등장을 하면서, 언패킹을 수동으로 하거나 전용 언패커를 만드는 것은 무의미해졌다. 따라서, 본 논문에서는 범용적인 임포트 테이블 기반의 언패킹 알고리즘을 제안한다.

1. 서론

최근 악성코드들은 자신을 암호화하거나 안티디버그 기능을 이용해서 백신의 탐지를 우회하고 분석 시간을 지연시킨다. 코드를 압축하거나 암호화해서 원래의 의미를 알 수 없게 하는 것을 패킹이라고 하며, 패킹을 하는 프로그램을 패커라고 부른다. 패킹은 한번 이상 적용될 수 있으며, 프로그램 전체에 적용 하거나 일부분만 적용 할 수도 있다. 보다 더 지능화된 패커들은 압축을 한 상태에서 안티 디버그, 안티 가상화 기술들을 적용해서 분석을 더욱 어렵게 한다. 패킹된 프로그램이 실행되면, 운영체제가 언패킹 모듈을 로드하고, 언패킹 모듈이 원래 프로그램 코드를 메모리에 복사를 한다. 이후, 언패킹된 프로그램의 임포트 테이블을 복구하고 Original Entry Point (OEP)로 EIP 레지스터를 설정해서 실행한다[1]. <표 1>은 윈도우 운영체제의 Portable Execuatble (PE) 파일 패커의 종류를 나타낸다.

<표 1> Portable Executable 패커의 종류

.netshrink	PELock
Armadillo Packer	PESpin
ASPack	Smart Packer Pro
ASPR (ASProtect)	Themida
Enigma Protector	UPX
EXE Bundle	VMProtect
EXE Stealth	CExe
eXPressor	RLPack Basic
MPRESS	XComp/XPack
Obsidium	NeoLite

분석가는 패킹된 상태에서는 프로그램이 어떤 기능을 하는지 알 수가 없기 때문에, 패킹되기 전의 코드를 얻는 것이 중요하다. 원래 코드를 얻기 위해서는 어떤 패킹 알고리즘이 사용되었는지 알아야 하는데, 해당 용도로 자주 사용되는 툴은 PEiD이다[2]. PEiD와 같은 프로그램들은 시그니처 데이터베이스를 기반으로 패킹 알고리즘을 탐지해서 원래 코드를 추출해낸다.

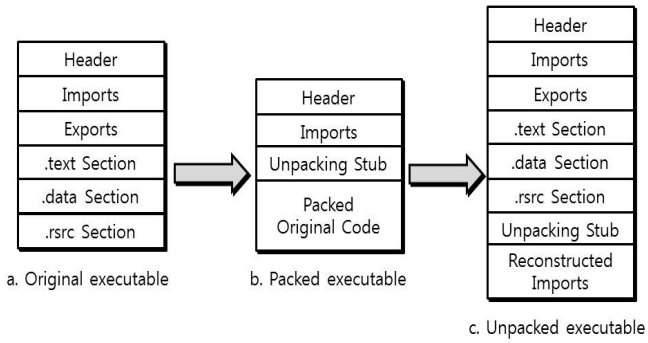
하지만 기존에 알려지지 않은 새로운 패킹 알고리즘과 다중 패킹된 악성코드들이 다수 등장을 하면서 시그니처 기반의 언패킹은 비효율적인 것이 되었다. 따라서, 시그니처 기반이 아닌 동적 분석 기반의 언패킹 알고리즘들이 제안되었다[2,3,4,5,6]. 동적 분석이란, 실제로 프로그램을 실행시키면서 프로그램 실행환경을 분석하는 방법을 뜻한다. 기존의 언패킹 알고리즘은 높은 확률로 원래 코드를 정상적으로 추출해내지만, 다음과 같은 단점이 존재한다. 첫째, 정적 분석과 동적 분석의 혼합 방식의 알고리즘인 경우, 패킹된 코드가 실행되기 전에 코드 모델을 생성해야하며, 동적 분석을 통해서 매번 코드 모델과 비교를 해야 하는 오버헤드가 발생한다. 둘째, 메모리를 모니터링하는 방식의 경우, 언패킹이 진행 중인 것은 확인할 수 있으나, 언패킹을 완료한 시점을 정확히 알 수가 없어서 코드를 추출할 시점을 판단하기 어렵다. 따라서 기존의 알고리즘들의 단점을 보완한 언패킹 알고리즘이 필요하다. 본 논문에서는 이런 단점들을 해결한 임포트 테이블 기반의 동적 언패킹 알고리즘을 제안한다.

논문의 구성은 다음과 같다. 2장에서는 패킹의 개념과 기존에 제안된 언패킹 알고리즘들을 소개하고, 3장에서 임포트 테이블 기반의 언패킹 알고리즘을 설명한다. 4장에서는 향후 수행해야 할 연구 내용을 제시하고 결론을 맺는다.

2. 관련 연구

2.1 패킹

패커가 실행이 되면, 원래 바이너리는 압축이 되고 새로운 파일 헤더와 임포트 테이블이 생성되어 메모리에 로드가 된다. 이때, 압축해제 루틴이 같이 메모리에 로드가 되어서 런타임 시 원래 코드를 복구한다[1]. 아래 그림은 프로그램이 패킹되고 런타임에 다시 언패킹될 때 프로그램의 변화과정을 설명하고 있다.

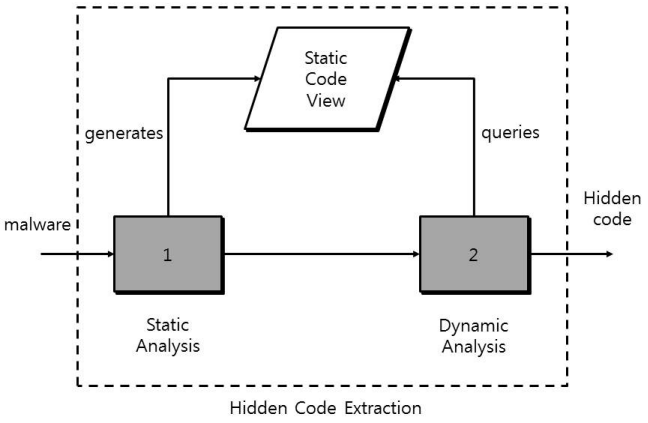


(그림 1) 패킹 프로세스[1]

a는 일반 PE 파일이 메모리에 로드되었을 때의 모습을 나타내고, b는 패킹된 파일이 디스크에 저장되어 있을 때의 모습을 나타낸다. a가 패킹이 되면 임포트 테이블은 프로그램 로더가 인식할 수 없게 된다. 따라서 패킹된 PE 파일에는 새로운 헤더와 임포트 테이블이 형성되며 언패커(Unpacking Stub)는 런타임에 패킹된 코드를 언패킹해서 c와 같이 메모리에 각 섹션을 복사한다. 그 다음, GetProcAddress, LoadLibrary 함수를 이용해서 원래 임포트 테이블을 재구축하고 명령어 포인터를 OEP 값으로 바꾼다. 최종적으로 언패킹된 파일의 모습은 c와 같다.

2.2 PolyUnpack

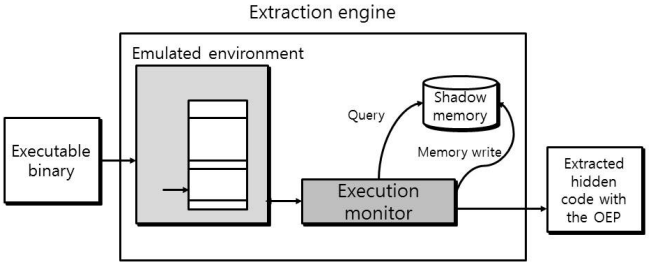
PolyUnpack는 정적 분석과 동적 분석을 혼합하여 악성 코드의 언패킹을 자동화하는 툴이다[3]. 입력으로 악성코드가 들어오면, 정적 분석을 통해 데이터와 코드를 분리시켜 코드 모델을 만들고 런타임에 싱글 스텝(single-step) 디버깅을 통해 현재 실행되는 코드가 코드 모델에 포함되어있는지 여부를 검사한다. 만약 런타임에 새로 생성된 코드가 실행 중이라면 악성코드의 실행을 중지하고 현재 명령어 주소에서 코드영역 끝까지 바이너리를 추출한다. 추출된 바이너리는 IDA Pro[4]와 같은 분석 툴로 추가적인 분석을 할 수가 있으며, 전 과정이 자동화되어 있기 때문에 다량의 악성코드를 분석할 때도 유용하게 사용할 수가 있다. (그림 2)는 PolyUnpack의 동작 과정을 설명한 그림이다.



(그림 2) PolyUnpack[3]

2.3 Renovo

Renovo는 동적 분석으로만 숨겨진 코드를 추출하는 프레임워크으로서, 언패킹하려는 바이너리에 대한 사전정보가 필요가 없다[5]. Renovo는 Bitblaze 프로젝트의 동적 분석기인 TEMU[6]를 이용해 구현되었다. TEMU는 시스템 에뮬레이터기 때문에, 하드웨어 정보만 제공을 한다. 따라서 분석하고자 하는 프로세스에 대한 운영체제 레벨의 정보를 얻기 위해서 커널 모듈을 만들어서 에뮬레이팅된 시스템에 설치를 한다. 이 모듈을 통해 Renovo는 프로세스에 대한 정보를 얻을 수 있다. Renovo는 바이너리를 실행하기 전, 섀도우 메모리 영역을 초기화 시키고, 런타임에 데이터를 쓴 메모리 영역을 바이트 단위로 더티 플래그를 설정한다. 실행중인 코드에 더티 플래그가 설정되었다면, 언패킹된 코드로 간주한다. 원래 코드가 다수의 압축과 암호화로 숨겨져 있을 경우에도, 각 패킹 계층이 제거될 때마다 섀도우 메모리를 클린 상태로 초기화하고 다시 분석을 시작하는 방식으로 정상적으로 언패킹된 코드를 추출할 수가 있다. (그림 3)은 Renovo가 패킹된 프로그램을 언패킹하는 과정을 설명한다.



(그림 3) Renovo[5]

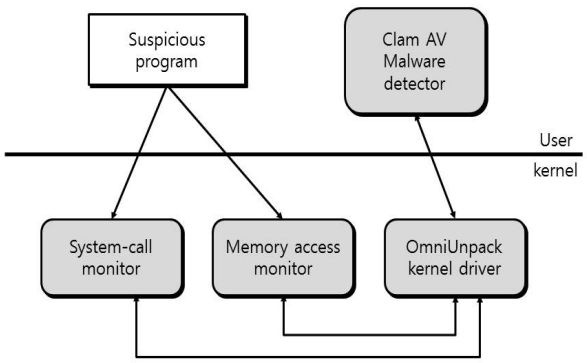
2.4 Saffron

Saffron은 두 가지 방법을 통해서 바이너리를 추출한다[7]. 첫 번째는 동적 인스트러먼테이션 (Dynamic Instrumentation)을 이용한 메모리 분석 방법이다. 동적 인스트러먼테이션은 가상환경에서 분석 코드를 삽입시켜 프로그램을 실행하는 기술이다. 두 번째는 페이지 폴트 핸

들러 수정을 통한 코드 실행 모니터링 방법이다. 페이지 폴트란, 가상 메모리 환경에서, 프로그램이 접근하려는 메모리가 실제 물리 메모리에 로드가 되지 않았을 때 발생하는 이벤트이다. Saffron이 실행되면, 프로세스의 가상 메모리 영역을 전부 스캔하여 페이지 경계 주소를 태그한다. 이 태그를 이용해서 메모리를 모니터링하고, 페이지 폴트가 발생하면 수정한 페이지 폴트 핸들러는 폴트가 발생한 메모리 주소를 로그에 남기는 방식으로 언패킹된 코드를 찾아낸다. 동적으로 메모리를 분석하여 언패킹된 코드를 찾아낸다는 관점에서 Renovo와 유사하다.

2.5 OmniUnpack

OmniUnpack는 메모리를 페이지 단위로 모니터링하기 때문에 명령어 단위로 모니터링하는 알고리즘보다 속도가 빠르다[8]. OmniUnpack는 프로그램의 실행을 모니터하다가 더티 메모리를 코드로 실행하려 하면 해당 코드가 언패킹되었다고 간주한다. 그리고 잠재적으로 악성행위를 할 수 있는 시스템 콜이 호출되면 언패킹이 완료되었다고 판단하며, 악성코드 탐지기를 호출하여 계속 프로그램을 실행시킬지 결정한다. OmniUnpack 어떤 개수의 패킹 알고리즘이 사용되었어도 올바르게 언패킹을 할 수 있으며, 디버깅 또는 가상화 등을 사용하지 않는 것이 특징이다. (그림 4)는 프로그램이 악성적인 시스템콜을 호출했을 때 탐지하는 과정을 설명하는 그림이다.



(그림 4) OmniUnpack[8]

2.6 Memory Behavior Based Unpacking

이 알고리즘은 접근되는 메모리 영역을 수식으로 점수를 부여해, 제일 높은 점수를 지닌 메모리 영역을 언패킹된 코드로 판단한다[9]. 메모리를 접근을 모니터링하는 방법은 시뮬레이터 기반의 방법과 인스트러멘테이션 기반의 밥법이 있다. 하지만 최근 악성코드들은 안티디버깅과 안티가상화 기술이 있어 메모리 모니터링을 봉쇄한다. 따라서 메모리 분석 방지 기술을 우회하기 위해 스텔스 디버거를 구현하였다. 스텔스 디버거는 프로세스로부터 투명하게 구현되었고, 커널 드라이버는 게스트 운영체제에 설치되어 운영체제 레벨의 정보를 제공하는 역할을 한다.

2.7 단점

2장에서 설명한 알고리즘들은 각각 단점들이 존재한다. 아래 표에 알고리즘 별 단점을 서술했다.

<표 3> 단점 비교

알고리즘	단점
PolyUnpack	코드 모델을 생성하고 명령어 단위로 분석하기 때문에 느리다.
Renovo Saffron	언패킹이 되지 않은 코드가 존재하는지 정확히 알 수 없다.
OmniUnpack	확인해야하는 시스템 콜의 숫자가 많다.
Memory Behavior -Based	수식을 계산하는 오버헤드가 존재한다.

3장에서는 이런 단점들을 해결한 임포트 테이블 기반의 알고리즘을 제안한다.

3. 임포트 테이블 기반 언패킹 알고리즘

제안하는 알고리즘은 패커가 다음과 같은 특징을 지닌다고 가정한다.

- 패커는 바이너리 전체를 패킹한다.
- 언패커는 바이너리 전체를 언패킹한다.
- 패킹된 프로그램은 GetProcAddress 함수와 LoadLibrary 함수를 임포트한다.
- 언패킹에 사용되는 가상 머신은 프로세스로부터 투명성을 지닌다.

알고리즘은 다음과 같이 3 가지 모듈로 구성되어있다.

3.1 섀도우 메모리

섀도우 메모리에는 프로세스 메모리의 각 바이트가 더티 바이트 인지를 표시하는 비트맵이 저장되어있다. 바이너리가 언패킹 되어 추출을 할 때, 비트맵을 보고 런타임에 새로 생성된 코드를 확인할 수 있다.

3.2 임포트 리스트

언패킹이 완료되는 시점을 추측하기 위해서 본 논문에서 제안하는 알고리즘은 임포트 테이블을 사용한다. 앞서 설명했듯이, 패커는 바이너리를 패킹할 시, 임포트 함수들을 숨기고 새로운 임포트 테이블을 생성한다. 언패킹 시에는 숨겼던 임포트 테이블을 재구축하여 원래 프로그램이 임포트하는 함수가 정상적으로 호출될 수 있게 한다. 임포트 테이블 재구축은 언패킹 과정 중 제일 마지막에 실행되는데, 그 이유는 임포트 테이블 재구축에 사용되는 LoadLibrary, GetProcAddress 함수의 인자가 문자열이기 때문이다. 이 문자열들은 분석툴에 의해 노출이 되지 않기 위해 패킹이 되어있어서 언패킹이 완료된 후에야 함수를 호출할 수 있다[10]. 따라서 임포트 테이블의 재구축이 완료되는 시점은 알면, 언패킹된 코드를 추출할 수 있다. 임

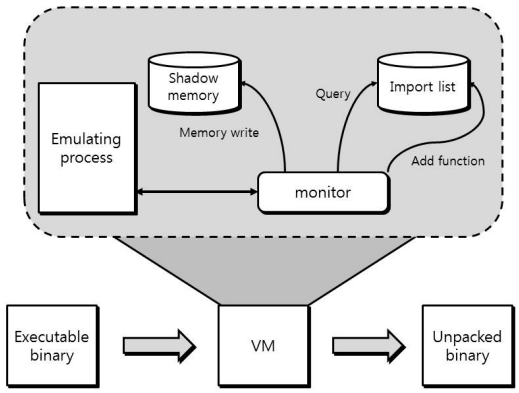
포트 테이블의 재구축 완료 시점을 알기 위해 임포트 리스트는 언패커가 임포트한 함수 리스트를 저장한다.

3.3 모니터

모니터 모듈은 크게 두 가지 행동을 수행한다. 첫째는 메모리 쓰기 명령어 모니터링이고 둘째는 함수 호출 모니터링이다. 런타임에 임의의 메모리에 데이터를 쓰는 경우, 해당하는 메모리 주소를 섀도우 메모리를 검색해서 찾은 후 더티 플래그를 설정한다. 함수 호출이 발생하는 경우, 해당 함수가 언패커가 임포트한 함수인지 여부를 임포트 리스트를 통해 확인한다.

3.4 전체 시스템

패킹된 바이너리가 입력으로 들어오면, 가상 머신에서 이를 실행을 한다. 이 때, 모니터 모듈을 가상 머신에서 실행되고 있는 바이너리를 모니터링하며 명령어 종류에 따라서 더티 플래그를 설정하거나 임포트 리스트에 함수 주소를 추가한다. 함수 호출이 발생했을 시에 임포트 리스트에 포함되지 않은 함수라면, 바이너리의 언패킹이 완료되었다고 판단하고 더티 플래그가 설정된 메모리 영역을 덤프한다. 만약에 패킹이 여러 번 되어있는 경우, 덤프된 바이너리가 여전히 패킹되어 있는지 판별한 후 다시 시스템에 입력시키면 올바르게 언패킹을 할 수가 있다. (그림 5)는 전체 시스템을 표현한 그림이다.



(그림 5) 전체 시스템

제안한 알고리즘을 사용할 경우, 기존에 제안된 언패킹 알고리즘을 사용할 때 보다 장점들이 있다. 첫째, 섀도우 메모리의 경우, 실제 메모리 1 바이트당 1 비트만 필요하기 때문에 섀도우 메모리의 사이즈가 현저하게 작다. 둘째, 정적 분석이 필요 없고 모든 명령어를 모니터링하지 않기 때문에 성능상의 이점도 있다. Pandoras Bochs는 GetProcAddress를 이용해서 임포트한 함수들의 호출 여부를 모니터링하는데, 임포트한 함수들이 많을수록 오버헤드가 발생한다[11]. 반면에 언패커의 임포트 함수들의 개수는 훨씬 적기 때문에 본 논문에서 제안하는 방식이 더 효과적이다.

4. 결론 및 향후 연구

최근의 악성코드들은 안티디버깅 기술을 사용하고 바이너리를 패킹하여 분석을 방해하고 있다. 특히, 사용하는 패킹 알고리즘은 계속 변화하기 때문에, 시그니쳐 기반의 언패킹이나 수동 언패킹은 무의미해졌다. 따라서 자동화된 범용 언패커의 필요성이 높아졌다. 기존에 제안된 범용 언패커들은 높은 확률로 임의의 패킹된 프로그램을 언패킹했으나, 여러 가지 단점들이 있었다.

따라서 본 논문에서는 임포트 테이블 기반의 언패킹 알고리즘을 제안했다. 메모리 쓰기, 그리고 함수 호출 명령어만 모니터링을 하면 단일 패킹 계층이나 다수의 패킹 계층이 있는 프로그램의 경우에도 효과적으로 메모리의 덤프 시점과 영역을 찾을 수 있다.

향후 연구에서는 본 논문에서 제안한 알고리즘의 구현 및 성능 평가가 이루어져야 할 것이다.

ACKNOWLEDGEMENT

본 논문은 중소기업청에서 지원하는 2011년도 산학연공동기술개발사업(No. 000443010111)의 연구수행으로 인한 결과물임을 밝힙니다.

참고문헌

[1] Michael Sikorski, Andrew Honig, “Practical Malware Analysis”, Wiley, Feb 2012.

[2] PEiD, http://www.peid.info, Mar 2012.

[3] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, Wenke Lee, “PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware”, ACSAC ‘06, Dec 2006.

[4] IDA Pro, http://www.hex-rays.com, Mar 2012.

[5] Min Gyung Kang, Pongsin Poosankam, Heng Yin "Renovo: A Hidden Code Extractor for Packed Executables", WORM ’07, Nov 2007.

[6] TEMU, http://bitblaze.cs.berkeley.edu, Mar 2012.

[7] Danny Quist, Valsmith “Covert Debugging Circumventing Software Armoring Techniques”, Offensive Computing, LLC, Blackhat USA, July 2007.

[8] Lorenzo Martignoni, Mihai Christodorescu, Somesh Jha, “OmniUnpack: Fast, Gerneric, and Safe Unpacking of Malware“, ACSAC ’07, Dec 2007.

[9] Yuhei Kawakoya, Makoto Iwamura, Mitsutaka Itoh, “Memory Behavior-Based Automatic Malware Unpacking in Stealth Debugging Environment”, 5th MALWARE 2010, Oct 2010.

[10] Chris Eagle, “The IDA Pro Book 2nd Edition”, NoStarch Press, July 2011

[11] Lutz Bohne, “Pandora’s bochs: Automatic unpacking of malware”, Diploma Thesis, Univ of Mannheim, Jan 2008