이학석사학위논문

악성코드 분석을 위한 PE 파일 언패킹 자동 시스템 설계 및 구현

김 선 균

강원대학교 대학원

컴퓨터과학과

2018년 2월

최 미 정 교 수 지 도
이학석사학위논문

악성코드 분석을 위한 PE 파일 언패킹 자동 시스템 설계 및 구현

Design and Implementation of PE File Unpacking Automatic System for Malware Analysis

강원대학교대학원

컴퓨터과학과

김 선 균

김선균의 석사 학위논문을
합격으로 판정함

2017년 12월

심사위원장　　문 양 세　　인

위　　원　　최 미 정　　인

위　　원　　임 현 승　　인

악성코드 분석을 위한 PE 파일 언패킹 자동 시스템 설계 및 구현

김 선 균

강원대학교 대학원 컴퓨터과학과

최근 컴퓨터 및 인터넷 기술이 하루가 다르게 발전해나가고 있으며 이러한 발전은 우리 생활에 엄청나게 긍정적인 영향을 미치고 있다. 이러한 발전은 우리 생활에 긍정적인 영향과 함께 부정적인 영향 또한 초래했다. 부정적인 영향들 중 악성코드에 의한 피해는 꾸준히 증가해오고 있으며 사용자들의 피해도 따라 증가했다. 하지만, 이에 따라 안티 바이러스 프로그램들도 발전하여 많은 악성코드들의 악의적인 행위를 억제하고 있지만 한 단계 더 발전한 악성코드들은 패킹을 통한 악성코드를 유포 및 빠른 대응을 방해하고 있다. 이처럼 발전한 악성코드로부터 발생하는 사이버 위협에 대응하기 위해 악성코드 파일에 대한 분석이 필요하며, 이를 위한 다양한 연구가 진행되고 있다. 사이버 위협에 대응하기 위해 활용하는 악성코드 분석 방법은 크게 초기 분석, 동적 분석(Dynamic-based analysis), 정적 분석(Statistics-based analysis)으로 나눌 수 있다. 악성코드의 피해를 최소화 하기 위해서는 실행이 되는 동적 분석 보다는 초기 분석과 정적 분석으로 악성코드를 탐지하고 분석하는 방법이 필요하다. 본 논문에서는 악성코드를 분석하기 위한 기초 단계롤 악성코드를 언패킹하는 시스템을 설계하고 구현하고자 한다. 제안하는 시스템은 크게 3단계로 구성되는데, 먼저 초기 분석을 통해 악성코드 및 PE 파일의 정보를 기반으로 패킹 여부를 탐지하고, 이와 함께 시그니쳐 기반으로 패커의 종류를 분석하고, 잘 알려진 패커로 패킹 되어있을 경우 해당 패커의 언패킹 툴을 사용하여 언패킹을 수행하는 시스템이다. 잘 알려진 실험 결과 PE 구조에서 올바른 정보를 추출하며 잘 알려진 패커로 패킹된 파일들의 패킹 여부를 탐지하였으며 잘 알려진 패커중 UPX로 패킹된 PE 파일의 경우 모두 언패킹을 성공적으로 수

행하여 패킹되지 않은 PE 파일을 복원하는 것으로 확인되었다. 악성코드 정적 분석을 진행하기 위해서는 패킹되지 않은 악성코드가 필요하다. 따라서, 본 시스템은 악성코드 분석에서 정적 분석을 위한 시스템이며 초기 분석과 시그니쳐 기반 패커 종류 탐지를 이용하여 정적 분석, 동적 분석 없이 언패킹을 수행할 수 있는 PE 파일에 한하여 언패킹을 수행할 수 있다는 점에서 큰 의미를 가진다. 추후, 악성코드 정적 분석을 위한 기반 시스템과 악성코드에 의한 피해 발생 시 빠른 대응과 피해 확산을 방지하는 방안으로 활용될 수 있다.

□ 핵심주제어

악성코드, 패킹, 언패킹, 패킹 여부 탐지, 패커 종류 탐지, PE 분석, 엔트로피

목 차

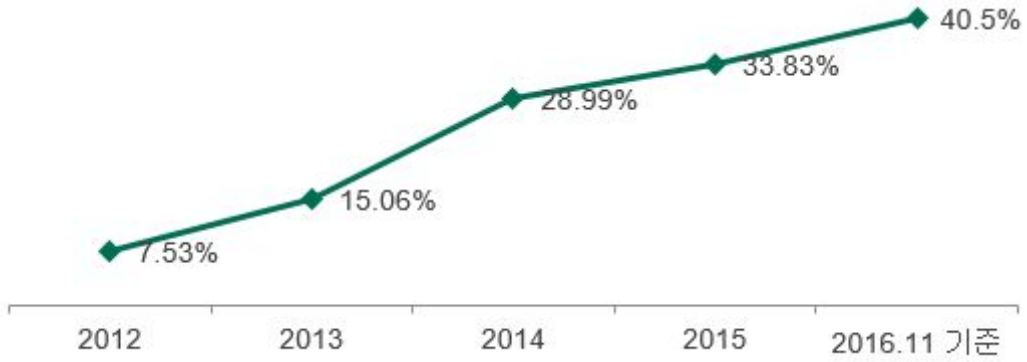
표 목 차

그 림 목 차

1. 서론

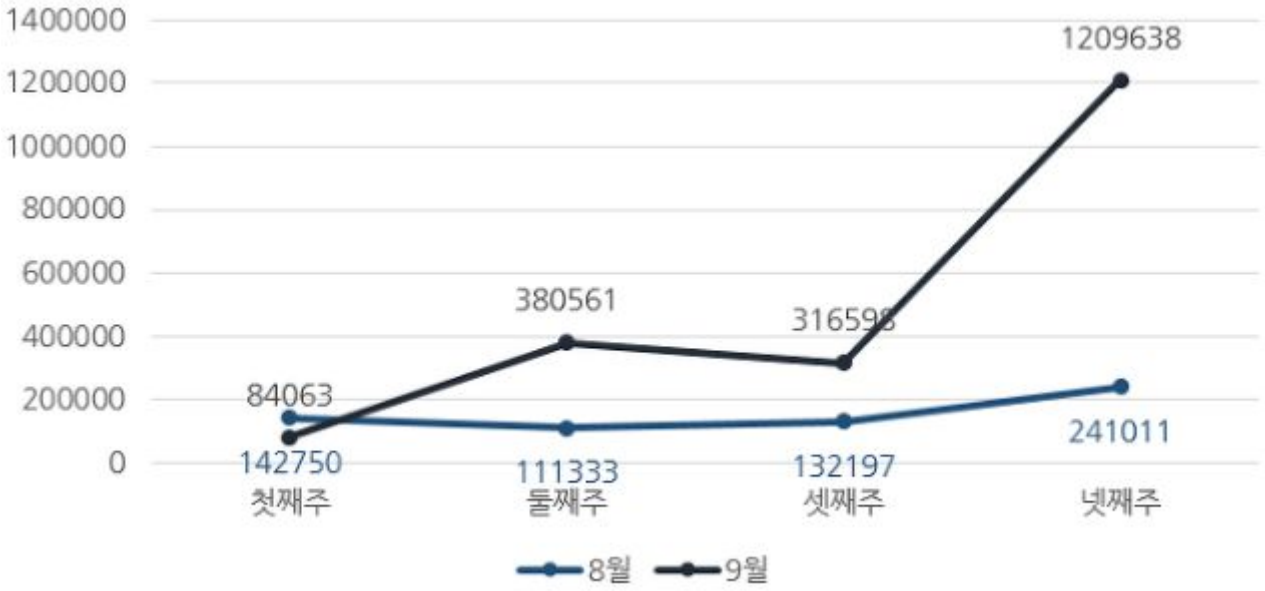
컴퓨터와 인터넷의 발전은 우리의 삶에 많은 변화를 가져왔다. 인터넷 속도가 빨라짐에 따라 대용량 파일들의 배포도 쉬워졌으며 그만큼 많은 파일들이 사용자에게 전달되고 있다. 또한, 응용프로그램 개발자는 사용자에게 다양한 서비스를 제공하기 위해 많은 파일들을 업로드하고 있으며, 사용자들은 서비스를 제공받기 위해 많은 응용 프로그램 파일들을 다운로드 하고 있다. 따라서, 현재 인터넷이 없다면 우리는 삶에서 많은 것을 하지 못하는 불편한 상황에 놓이게 될 것이라고 봐도 무방하다. 실제로 미래창조과학부의 2016 인터넷 이용실태 조사 자료[1]에 따르면 2016년 7월 기준 만3세 이상 인구의 인터넷 이용률은 88.3%이며, 이는 5년 전보다 10% 증가한 수준이다. 또한, 인터넷 이용자수는 약 43,636,000명이며, 남성의 91%와 여성 86%는 인터넷을 사용하며 연령대 별로 30대, 40대, 50대가 인터넷을 가장 많이 사용하였으며, 인터넷을 자주 접하지 않는 연령대를 제외하면 인터넷 이용률은 굉장히 높은 수치이다. 따라서, 우리나라 인구의 약 90%는 인터넷을 사용한다고 볼 수 있다. 뿐만 아리나, 인터넷 이용율은 계속해서 증가하는 추세이다.

이와 같이 인터넷의 이용률이 높아지면서 정보의 공유, 배포, 인터넷 쇼핑, 인터넷 방송 등과 같이 다양한 서비스가 발전한다는 측면에서 긍정적인 효과를 얻었다. 하지만, 그 이면에는 랜섬웨어, 악성코드, 스미싱 등 DDoS(Distributed Denial of Service) 및 사회기반 시설에 대한 공격과 같은 사이버 범죄의 위협이 증가하는 부정적인 효과도 발생하게 되었다. 미래창조과학부의 2017 악성코드 은닉사이트 탐지 동향 자료[2]에 따르면 현재 등장하고 있는 악성코드 피해 유형은 파밍 및 금융정보 탈취가 55%, 다운로더가 10%, 드롭퍼가 8%, 랜섬웨어가 5%, 백도어가 5%로 금융정보 탈취에 대한 악성코드 유형이 가장 주를 이루었다. 파밍은 사용자의 PC를 악성코드로 감염시켜 정확한 웹페이지 주소를 입력해도 가짜 웹페이지에 접속하게 되어 개인 정보를 탈취한다. 최근 이슈로 떠오르고 있는 랜섬웨어 또한 개인과 기업 모두에게 굉장한 위협이 되고 있으므로 악성코드에 대한 예방 및 대비가 필요하다[3].



[그림 1] 연도별 추가된 악성코드 개체 수(%)

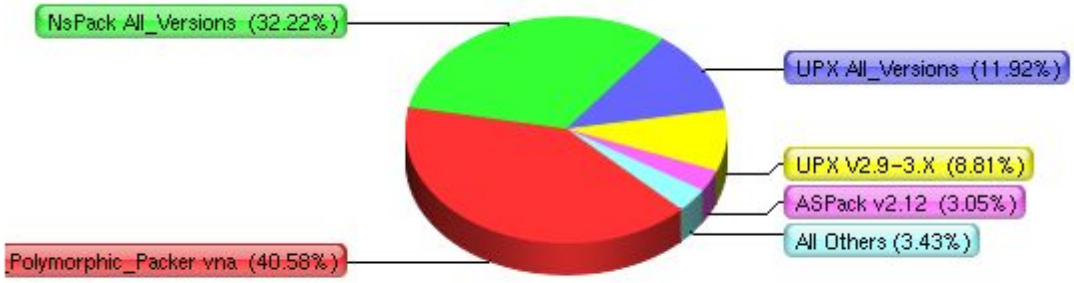
[그림 1]은 카스퍼스키랩의 클라우드 데이터베이스에 자동으로 추가된 악성코드의 각 년도별 비율이다[4]. 각 연도별로 악성코드의 개체 수가 계속해서 증가하는 것을 확인할 수 있다.



[그림 2] 2017년 9월 주간 악성코드 진단 현황

[그림 2]는 잉카인터넷의 주간 악성코드 진단 현황을 나타낸다.[5] [그림 2]에 따르면 주별로 악성코드 진단 횟수가 굉장히 늘어났다. 이는 악성코드에 의한 피해 발생 가능성이 증가했다고 볼 수 있다. AV-TEST[6]에 따르면 매일 새로운 악성코드를 약 25만 개를 등록한다고 한다. 악성코드 개체 수 및 진단 횟수가 증가한 이유로는 발전된 인터넷을 이용하여 빠르게 악성코드를 배포 하여 악성코드 제작자들이 경제적, 금전적

이득을 가질 수 있기 때문이다.[7]



[그림 3] 주간 패커 사용 통계

파일의 배포를 용이하게 하기위해서 패킹을하여 파일들을 배포한다. [그림 3]에서 볼 수 있듯이 잘 알려진 패커중 UPX, Nspack, Aspack의 사용이 많다. 따라서, 악성코드를 패킹할 때 위의 패커를 사용할 가능성이 높다고 볼 수 있다.[8]

최근 악성코드 위협에 대응하기 위해 다양한 연구들이 진행되고 있다. 진행되고 있는 악성코드 분석 연구는 정적 분석과 동적 분석 방법으로 구성된다. 정적 분석[9, 10, 11]은 악성코드를 실행하지 않은 상태에서 내부 코드와 구조를 확인하거나 헤더 정보 및 문자열 등의 여러 가지 정보를 이용하여 악성코드를 분석하는 것을 의미한다. 동적 분석은 일반적으로 악성코드를 직접 실행시켜 행동을 관찰 및 분석하여 코드의 흐름과 메모리 상태를 직접 모니터링 하는 것을 말한다. 최근 동적 분석에 대한 대처법으로 Anti-VM 기능이 발전함에 따라 악성코드가 동적 분석을 우회하거나 무력화 하고 있다. 따라서, 최근 악성코드 정적 분석에 대한 연구가 진행되고 있는 분야이다. 하지만, 악성코드 정적 분석을 위해서는 패킹된 악성코드를 언패킹된 악성코드를 분석해야 하므로 언패킹에 대한 연구가 진행이 연구되어야 한다. 각 분석 방법과 패킹 언패킹에 대해서는 2장에서 서술한다.

본 논문에서는 악성코드 정적 분석을 위한 언패킹을 수행하며, 패킹 여부 탐지, 패커 종류 탐지 및 잘 알려진 패커에 대해서 자동 언패킹을 수행하는 시스템을 설계 및 구현한다. 본 시스템의 최종 목표는 패킹된 악성코드의 정적 분석을 위한 자동 언패킹이다.

본 논문의 구성은 다음과 같다. 제 2장에서는 악성코드 분석과 PE 구조, 기존 PE 분

석 툴의 한계점, 패킹 및 언패킹에 대해서 소개한다. 제 3장에서는 본 논문에서 제시하는 시스템의 PE 추출, 패킹 여부 탐지, 패커 종류 탐지, 잘 알려진 패커 자동 언패킹에 대한 설계 과정에 대해 서술한다. 제 4장에서는 본 논문에서 제시한 시스템의 PE 추출, 패킹 여부 탐지, 패커 종류 탐지, 잘 알려진 패커 자동 언패킹에 대해서 검증한다. 마지막으로 제 5장에서는 결론과 함께 향후 연구를 제시한다.

2. 관련 연구

본 장에서는 악성코드 및 PE 파일 언패킹을 위한 기존 연구에 대해 소개한다. 제 2.1절에서는 악성코드를 분석 기법에 대해 서술한다. 제 2.2절에서는 본 시스템에서 추출되는 정보의 구조인 PE 구조에 대해서 설명한다. 제 2.3절에서는 PE 구조를 압축하며 코드 은닉화도 가능하게 하는 방법인 패킹(실행 압축)에 대해 설명 하고, 패킹을 수행 각 패커들을 소개한다.

2.1 악성코드 분석 기법

본 절에서는 악성코드를 분석 기법에 대해 서술한다. 악성코드 분석 크게 초기 분석, 동적 분석 그리고 정적 분석으로 구분된다. 본 시스템은 악성코드 정적 분석을 위한 초기 분석 기반으로 설계 및 구현 되었으며 이는 3장에서 서술한다.

2.1.1 초기 분석

초기 분석이란 말 그래로 악성코드를 맨 처음 분석하는 것이며 분석할 PE 파일의 헤더 정보를 분석하여 파일의 가지고 있는 기본 정보를 추출하는 것이다. 초기 분석의 역할은 PE 파일의 언패킹을 위한 사전 준비라고 생각할 수 있다. 추출하는 기본 정보는 파일 이름, 크기, 오프셋, 엔트리 포인트 주소, 엔트리 포인트 섹션 이름, First bytes 등이 있다. 언패킹을 위한 필수 추출 정보로는 PE 파일의 확장자, 비트, 만들어진 머신의 종류, 엔트로피 값, 속성 값, 진입점 섹션의 엔트로피 값, 진입점 섹션의 속

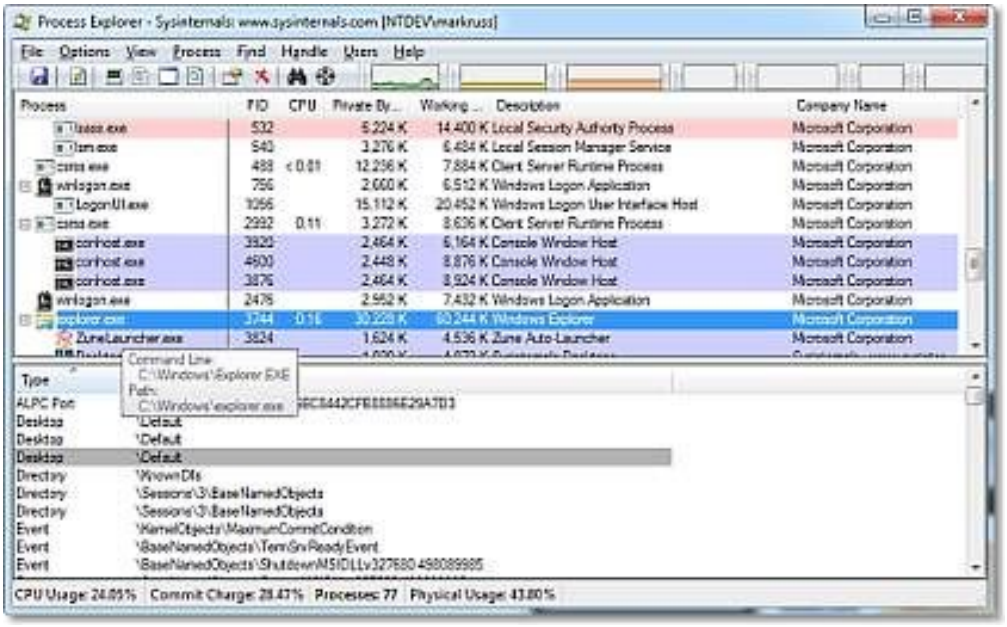
성 값 등을 추출한다. 패킹 여부 탐지를 위해 파일의 전체 엔트로피 값이 아닌 진입점 섹션의 엔트로피 값과 진입점 섹션의 속성 값을 확인하여 패킹 여부를 탐지한다. 패킹 여부 탐지를 위해 추출하는 엔트로피 값과 속성 값이 진입점 섹션의 값인 이유는 2.3 절에서 서술 한다. 악성코드 정적 분석을 위해서는 패킹이 되어있지 않은 악성코드이어야 정적 분석이 진행될 수 있다. 대부분의 악성코드는 배포의 용이함과 분석의 지연 및 방해를 위해 코드의 은닉화를 수행하는 패킹이 되어있다. 따라서, 초기 분석으로 악성코드의 패킹 여부 탐지와 패커 종류 탐지는 굉장히 중요하며 패커 종류 탐지에서 언패킹이 가능한 패커로 패킹이 수행된 악성코드는 자동으로 언패킹을 수행하여 바로 정적 분석을 가능하게 할 수 있고 이에 따라 악성코드 피해에 대해서 빠른 대응과 피해 확산 방지가 가능하다. 하지만, 초기 분석으로는 언패킹 방법이 공개되 어있거나 언패킹 툴이 존재하는 패커로 패킹된 경우만 언패킹이 가능하다는 한계점이 있다.

2.1.2 동적 분석

동적 분석이란 일반적으로 악성코드를 직접 실행시켜 행동을 관찰 및 분석하여 코드의 흐름과 메모리 상태를 직접 모니터링 하는 것을 말한다[12]. 따라서, 동적 분석은 주로 악성행위를 발견하고 악성코드의 실행을 감시 및 추적하는 목적과 악성코드의 실제 동작 방식을 분석한다. 정적 분석에 비교하여 동적 분석의 이점으로 원본 PE 파일에서의 변경에 영향을 받지 않는다[13]. PE 파일의 데이터, 제어흐름 변경, nop 삽입 등의 방법을 사용하여 안티 바이러스 프로그램을 우회 하며, PE 파일을 패킹할 경우 정적 분석을 방해하는 요소가 된다. 특히, 패킹된 PE 파일에 대해서 언패킹을 수행하지 않는다면 더 이상의 정적 분석은 불가능 하다. 또한, 동적 분석의 경우 API 호출정보를 기반으로 자원에 대한 접근 정보를 반복적으로 추적해줘야 하므로 정확한 정보 획득에 어려움이 많다.

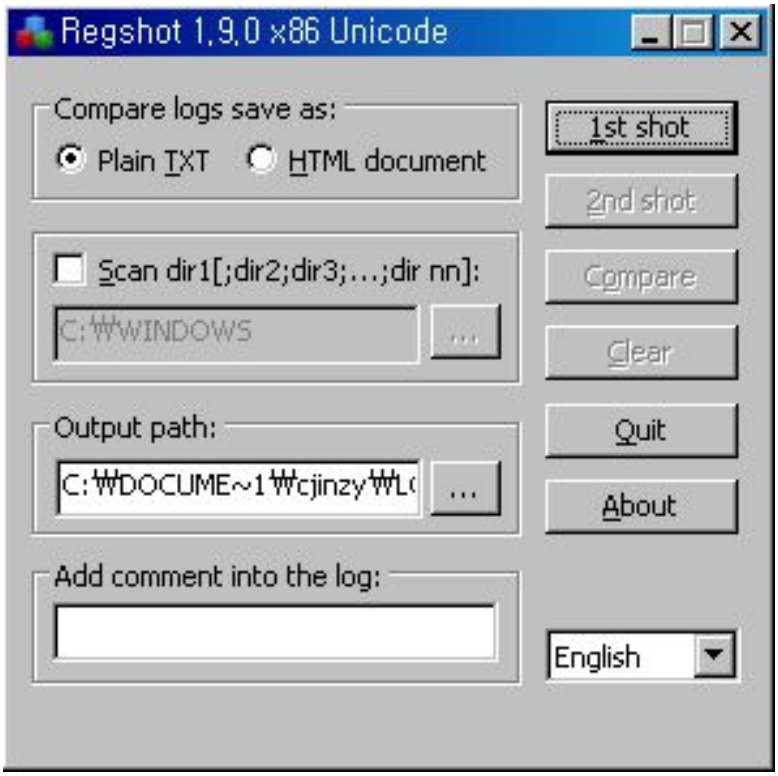
동적 분석 방법으로는 프로세스의 변화를 분석하기 위해 유저 모드 및 커널 모드에서 동작하고 있는 API를 보여주거나 파일 생성 및 삭제와 같은 특정 이벤트등과 같이

시스템에서 발생하는 변화를 관찰한다. 또한, 레지스트리를 분석하여 악성코드 감염의 전 후의 차이를 기반으로 변화를 탐지하여 분석하고 네트워크를 분석하여 공격자에게 수집된 로그를 전송한 흔적을 찾아내기 위하여 네트워크 변화를 분석하는 방법이 있다. 이러한 방법들은 대부분 각 방법에 알맞은 툴을 사용한다. 일반적으로 프로세스 변화 분석을 위해서는 “Process Explorer”를 사용한다. 프로세스 분석시에는 프로세스의 현재 상태 메모리 덤프를 수행할 수 있다. 다음 [그림 4]는 프로세스 분석 도구인 “Process Explorer” 이다.



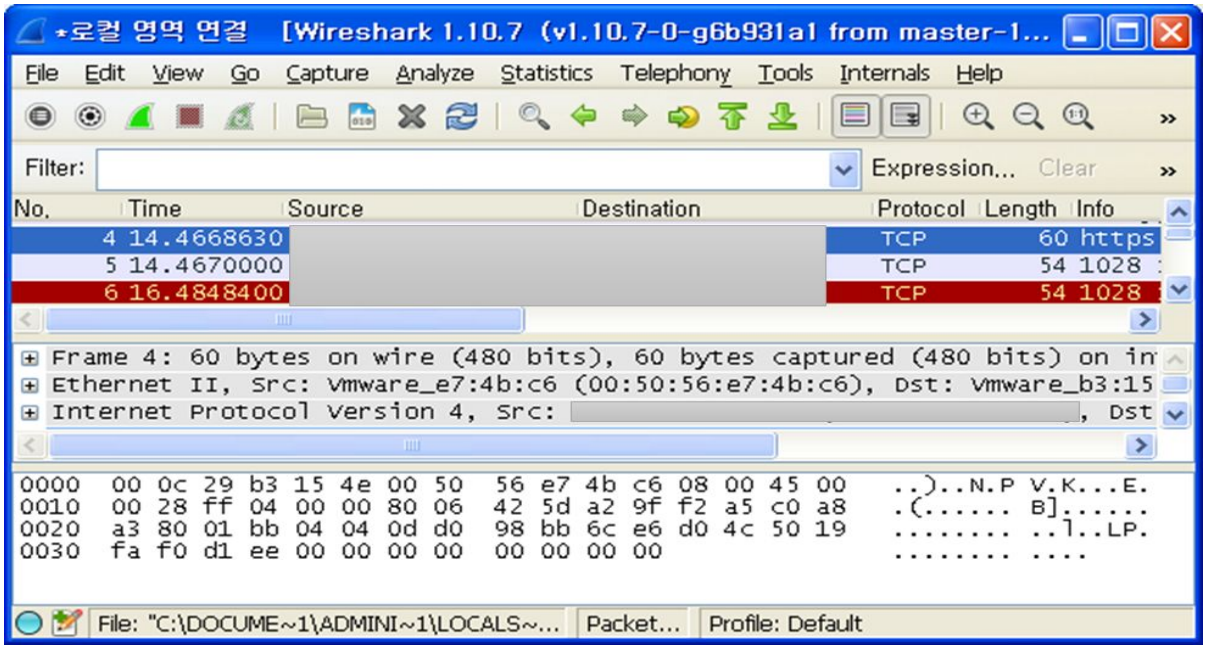
[그림 4] 프로세스 분석 도구 Process Explorer

레지스트리 분석을 위해서는 그리고 “Regshot”을 사용한다. 레지스트리 분석을 통해 Anti-VM 기법이 동작중인지 확인할 수 도 있다. 다음 [그림 5]는 레지스트리 분석 도구인 “Regshot” 이다.



[그림 5] 레지스트리 분석 도구 Regshot

네트워크 분석을 위해서는 “WireShark”를 주로 사용한다. 네트워크 분석시 패킷을 캡쳐하여 분석 후 공격에 사용된 주요포트나 흔적을 관찰하고 악성코드가 사용한 포트를 막아 악의적인 행위를 차단할 수 있다. 다음 [그림 6]는 네트워크 분석 도구인 “WireShark” 이다.



[그림 6] 네트워크 분석 도구 WireShark

동적 분석의 한계점으로 동적 분석은 악성코드를 실행시켜놓은 상태에서 분석을 진행기 때문에 악성코드의 악의적인 행위에 노출될 가능성이 높다. 따라서, 악성코드 동적 분석을 진행하기 위해서는 보통 가상머신(Virtual Machine)에서 분석을 진행해야 한다. 하지만, 현재 악성코드가 동작하고 있는 환경이 가상머신 이라는 것을 탐지하여 악의적인 행위를 숨기는 Anti-VM 기법을 동반하고 있는 악성코드라면 동적 분석을 진행하여도 악성코드는 정상 파일처럼 행동할 것이고 동적 분석은 무력화 된다. 이에 따라, 동적 분석이 정상적으로 진행되려면 Anti-VM 기법을 무력화 하여야 한다. 또한, 동적 분석시 Ollydbg와 같은 디버깅 툴을 사용하여 디버깅을 진행하여야 한다. 하지만, 이 또한 Anti-Debug 기법으로 인하여 무력화가 될 수 있다. 간단한 예로 현재 프로세스를 디버깅중에 있으면, 현재 프로세스가 디버깅중이라는 것을 알려주는 API(IsDebuggerPresent)가 존재하며 악성코드가 이 API의 리턴 값을 받아 무한 루프에 빠지게 하면 더 이상의 디버깅은 불가능 하다. 이러한 API 말고도 많은 Anti-Debug 기법들이 디버깅을 방해하는 요소로 작용하며 이러한 Anti-Debug 기법을 우회 하여야만 정상적인 분석이 가능하다. 또한, 현재 악성코드가 동작하고 있는 환경이 네트워크를 사용하지 않을 경우 악의적인 행동을 숨기는 경우도 존재하기 때문에

동적 분석 방법에는 위의 여러 가지 경우처럼 한계점이 존재한다.

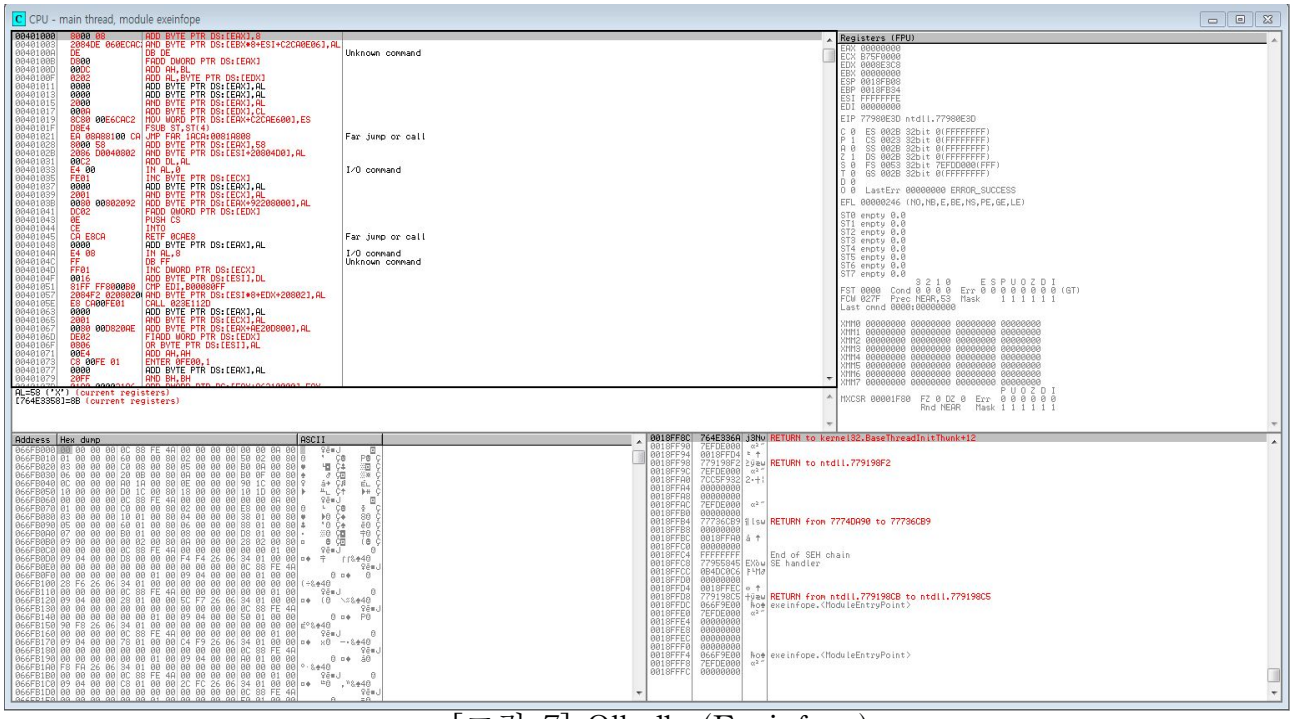
2.1.3 정적 분석

정적 분석이란 악성코드를 실행하지 않은 상태에서 내부 코드와 구조를 확인하거나 헤더 정보 및 문자열 등의 여러 가지 정보를 이용하여 악성코드를 분석하는 것을 의미한다. 정적 분석은 동적 분석과는 다르게 특정 실행 조건에 제한되지 않고 분석을 진행 할 수 있으며, 프로그램의 전체 실행에 대해 분석할 수 있다는 장점이 있다. 또한, 악성코드를 실행하지 않은 상태에서 분석을 진행하기 때문에 악성코드 감염의 위험에서 벗어날 수 있으며 악성코드를 실행하면서 발생하는 자원 과부하가 없다. 하지만, 정적 분석을 제대로 수행하기 위해서는 리버스 엔지니어링(Reverse Engineering)에 대한 전문적인 지식을 갖고 있어야한다.

정적 분석의 방법으로 Disassembler나 Decompiler를 이용하여 리버스 엔지니어링 기법으로 악성코드를 한 줄씩 실행하여 분석을 진행해 나가야 한다. 악성코드의 실행 없이 분석을 진행하기 때문에 실행 조건에 제한되지 않고 악성코드의 흐름과 동작에 대해 분석이 가능하다. 정적 분석은 앞서 이야기한 초기 분석과 비슷하게 수행된다. 악성코드 내에 API 정보를 추출 및 분석하여 악성코드가 어떤 악성행위를 동작하는지 추측하며 분석을 진행해야 한다.

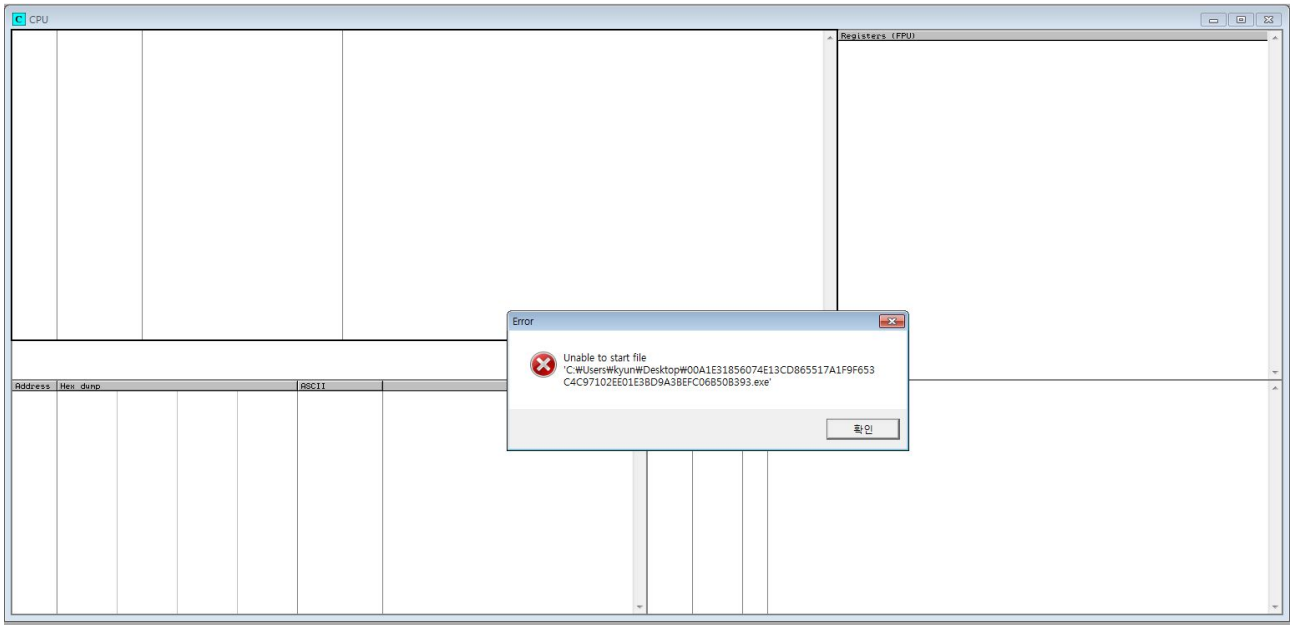
정적 분석은 악성코드가 패킹이 되어있을 경우 분석의 진행이 불가능하다는 한계점이 존재한다. 따라서, 정적 분석의 전제조건으로 패킹이 되어있으면 안되며 패킹이 되어있을 시에는 언패킹을 수행하여 원본 파일 상태에서 분석을 진행해야 한다. 또한, 악성코드가 암호화 등의 기법으로 제작되었을 경우 정적 분석을 진행하기 어렵다. 따라서, 정적 분석을 자동화로 만들기에는 어렵고, 시간과 노력이 굉장히 많이 소비되는 한계점이 존재한다. 정적 분석시 Ollydbg[14]를 사용하여 악성코드를 디스어셈블 후 악의적인 행위를 동작하게 하는 코드를 세밀하게 분석하여야 한다. 하지만, 앞서 언급한 Anti-Debug 기법이 동작하고 있다면 분석하기 위한 Ollydbg에 로드가 되지 않는다.

Ollydbg의 여러 가지 옵션으로 Anti-Debug를 우회하더라도 계속 새로운 Anti-Debug 방법이 출현한다면 정적 분석이 진행되기 어렵다. 다음 [그림 7]는 Ollydbg에 PE 분석 툴인 Exeinfope.exe를 로드한 것이다.



[그림 7] Ollydbg(Exeinfope)

다음 [그림 8]는 VirusShare에서 다운받은 PE 구조의 악성코드를 Ollydbg에 로드한 화면을 캡쳐한 것이다. Anti-Debug 때문에 Ollydbg에 로드 되지 않는다.



[그림 8] Ollydbg(악성코드)

현재 악성코드 동적 분석에 대한 연구가 꾸준히 진행되고 있었으나 다시 정적 분석에 대한 연구에 초점이 맞춰지고 있는 상황이다. 따라서, 본 시스템에서 악성코드 언패킹을 위한 패킹 여부/패커 종류 탐지를 위한 악성코드 분석은 큰 의미를 가진다. 또한, 악성코드 정적 분석의 방해요소나 한계점을 고려하여 올바른 환경에서 악성코드 분석을 진행해야 한다.

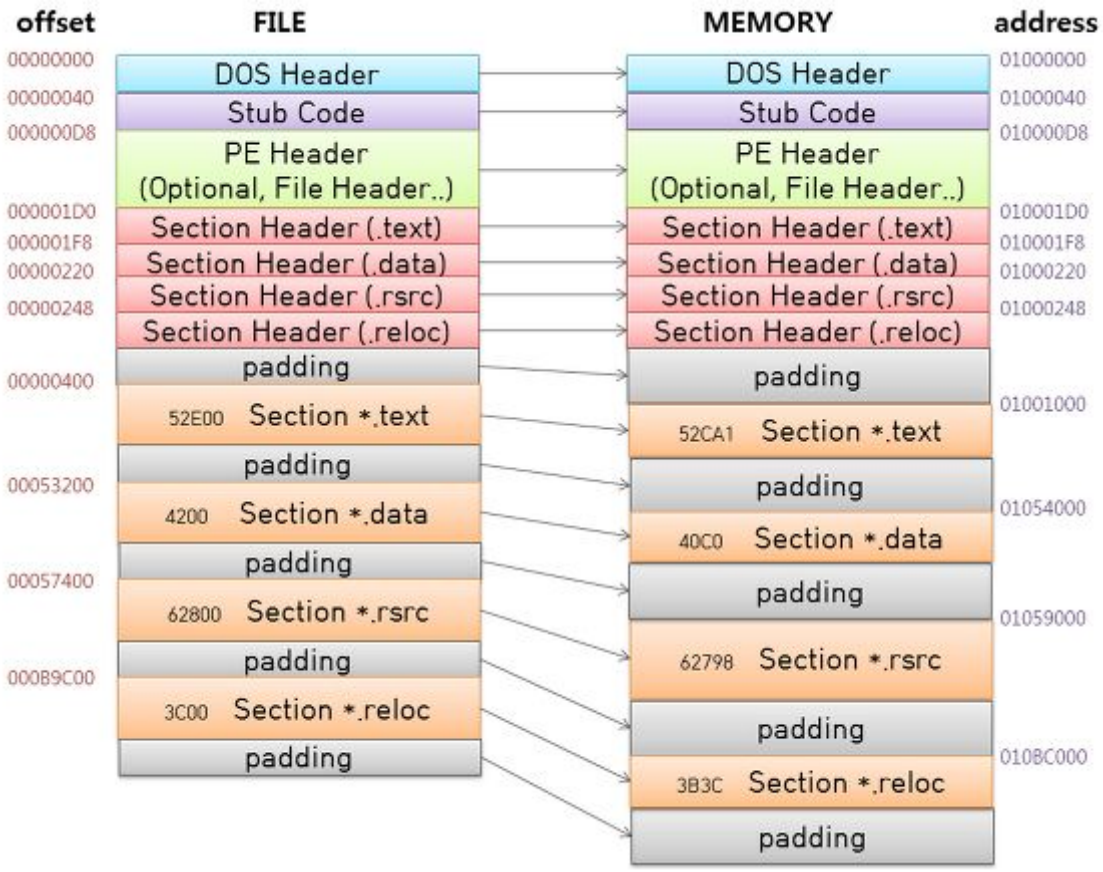
2.2 PE(Portable Executable) 포맷

본 절에서는 PE 포맷과 PE 포맷의 각 세부사항과 PE 구조가 가지고 있는 각 구조체에서 필요한 필드들에 대해서 설명한다. PE 포맷은 Windows 환경에서 사용되는 실행 파일, DLL 파일 등을 위한 파일 형식이며 악성코드 분석 및 언패킹을 위해서는 PE 포맷에 대한 이해가 필요하다.

2.2.1 PE 구조

PE 포맷은 윈도우 운영 체제에서 사용되는 실행 파일(exe), DLL 파일, object 파일, 폰트 파일, sys 파일 그리고 드라이버 파일 등을 위한 파일 형식이다. PE 포맷의 파일은 Windows 환경이라면 어디에서든 실행 가능한 파일 형식이다. Portable Executable 이라는 의미는 'Windows 운영체제라면 어디에서든 실행 가능'이라고 생각할 수 있다. PE 포맷으로 잘 알려진 확장자는 exe, ell, obj, sys 등이 있으며 PE 포맷은 윈도우 로더가 실행 가능한 코드를 관리하는데 필요한 정보를 캡슐화 한 데이터 구조체 이다 [15].

PE 구조는 DOS 헤더, PE 헤더, 각 Section 헤더 그리고 각 Section 테이블로 구성되어있다. 따라서, PE 포맷은 정해진 포맷을 가지고 있으며 이 정보들을 추출하여 분석하면 언제, 어떤 머신에서, 어떤 컴파일러로 제작되었나, 와 같은 정보들을 얻을 수 있다. PE 구조의 Section 테이블에는 데이터들이 들어있으며, 들어있는 데이터에 관한 메타데이터가 각 Section 헤더에 저장되어 있으므로 Section 헤더를 분석하여 정보를 추출해야 한다. 다음 [그림 9]는 PE 구조이다.



[그림 9] PE 구조

위의 그림에서 왼쪽은 PE 파일의 구조이며 오른쪽은 메모리에 로드 되었을때의 상태이다. 우리는 DOS 헤더와 PE 헤더 그리고 각 Section 헤더들을 분석하여 정보를 추출한다. PE 구조는 DOS 헤더를 시작으로 프로그램의 많은 정보들을 구조체 형태로 포함하고 있다. 일반적으로 PE 헤더는 DOS 헤더부터 Section 헤더 까지를 말한다.

2.2.2 PE 파일 헤더

PE 포맷에서 첫 번째로 나타나는 DOS 헤더에는 처음 문자열을 읽으면 'MZ'(4D5A)가 존재하는데 이는 도스 개발자의 이니셜을 적어놓은 것이며 DOS 시그니쳐를 의미한다. MS-DOS에서 사용하던 MZ 포맷으로 PE에서는 큰 의미는 없고 DOS 스텁에서 식

별 가능한 문자열은 “This program cannot be run in DOS mode.” 이다. 따라서, PE 포맷의 시작이 ‘MZ’라는 것을 알고 있어야 한다. 다음 [그림 10]는 DOS 헤더의 구조체인 IMAGE_DOS_HEADER 구조체 이다.

```
typedef struct _IMAGE_DOS_HEADER {
    WORD   e_magic;          // DOS signature : 4D5A ("MZ")
    WORD   e_cblp;
    WORD   e_cp;
    WORD   e_crlc;
    WORD   e_cparhdr;
    WORD   e_minalloc;
    WORD   e_maxalloc;
    WORD   e_ss;
    WORD   e_sp;
    WORD   e_csum;
    WORD   e_ip;
    WORD   e_cs;
    WORD   e_lfarlc;
    WORD   e_ovno;
    WORD   e_res[4];
    WORD   e_oemid;
    WORD   e_oeminfo;
    WORD   e_res2[10];
    LONG   e_lfanew;         // offset to NT header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

[그림 10] IMAGE_DOS_HEADER 구조체

e_magic와 e_lfanew를 제외한 나머지는 중요하지 않은 필드들이다. e_magic은 앞서 언급한 DOS 시그니쳐 이며 e_lfanew는 PE 헤더의 시작 이면서 실제 PE 파일의 시작이라고 할 수 있는 IMAGE_NT_HEADER 구조체를 가리킨다. e_lfanew 필드의 값만큼 파일 포인터를 이동시켜 헥사 값을 확인해보면 “50 45”로 시작함을 알 수 있다. 다음 [그림 11]는 IMAGE_NT_HEADER 구조체 이다.

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;                    // PE Signature : 50450000 ("PE"00)
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;

} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

[그림 11] IMAGE_NT_HEADER 구조체

Signature 필드는 PE 파일임을 의미하며 PE 파일인지 확인하려면 앞의 IMAGE_DOS_HEADER 구조체의 e_magic 필드와 IMAGE_NT_HEADER 구조체의 Signature 필드를 확인하여 각 값이 “MZ”와 “PE” 인지를 확인하면 된다.

다음 구조체는 IMAGE_FILE_HEADER 구조체이며, 이 구조체는 해당 파일의 머신 타입, 섹션의 수, 파일이 만들어진 시간 등의 값을 담고 있다. 다음 [그림 12]는 IMAGE_FILE_HEADER 구조체 이다.

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

[그림 12] IMAGE_FILE_HEADER 구조체

Machine 필드는 파일이 동작 가능한 머신 종류를 가리킨다. 머신 종류로는 32비트 기반 인텔 x86 아키텍처, 64비트 기반 인텔 x86 아키텍처 등 여러 가지 머신 타입들이 WinNT.h 파일에 정의 되어있다. NumberOfSection 필드는 파일에 존재하는 섹션의 개수 이며 TimeDateStamp는 파일이 만들어진 시간이다.

PE 파일의 마지막 구조체는 IMAGE_OPTIONAL_HEADER 구조체이다. 이 구조체는 중요하고 많은 정보를 담고 있으며, 악성코드 분석 및 패킹 여부 탐지에 아주 중요

한 요소로 사용된다. 중요 정보로는 진입점 섹션의 시작주소, 모든 코드 섹션의 크기를 합한 값, 첫 번째 코드 섹션이 시작되는 RVA(Relative Virtual Address), ImageBase 등과 같은 중요 정보를 담고 있는 필드가 IMAGE_OPTIONAL_HEADER 구조체가 갖고 있다. 본 시스템에서는 앞서 언급한 모든 헤더를 분석하여 필요한 정보만을 추출하여 DB에 저장한다. 또한 엔트로피 값을 기반으로 패킹 여부를 탐지하는 기법은 2.3절에서 자세히 설명한다. 다음 [그림 13]는 IMAGE_OPTIONAL_HEADER 구조체 이다.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

[그림 13] IMAGE_OPTIONAL_HEADER 구조체

[그림 13]에서 맨 위 Magic 필드는 IMAGE_OPTIONAL_HEADER를 나타내는 시그니처이면서 해당 PE 파일이 32비트인지 64비트를 구분해주는 값이기도 하다. IMAGE_FILE_HEADER 구조체의 Machine 필드의 값으로 비트를 구분하기보다 Magic 필드를 이용하는 것이 더 적절하다고 한다. SizeOfCode 필드는 모든 코드 섹션의 크기를 합한 값을 갖는다. AddressOfEntryPoint 필드는 이 파일이 메모리에 로드된 이후 실행을 시작할 주소 즉, 진입점 주소에 대한 RVA를 갖는다. 따라서, 해당 PE 파일이 메모리에 로드된 후 프로세스의가 동작하는 첫 번째 코드의 주소라고 볼 수 있다. 패킹된 악성코드를 언패킹할 때 중요한 것이 OEP(Original Entry Point)를 찾는 것이다. 이에 대해서는 2.3절에서 자세히 설명하겠다. BaseOfCode 필드는 첫 번째 코드 섹션이 시작되는 RVA값을 가지고 있다. 일반적으로 코드 섹션의 위치는 PE 헤더 다음 데이터 섹션 전에 위치한다. ImageBase 필드는 RVA 값에 대한 기준 주소가 되는 중요한 필드이다. ImageBase가 가지고 있는 값은 로더가 PE 파일을 로드할 때, 가상 주소 공간에 매핑 시키고자 하는 메모리 상의 시작 주소 이다. 매핑 시키고자 하는 주소 공간에 다른 파일이 로드되어 있다면 재배치 과정을 거쳐 다시 로드가 된다. SizeOfImage 필드는 로더가 메모리상에 PE 파일을 로드할 때 예약해야 할 충분한 크기를 갖고 있다. 이 값은 메모리에 로드될 수 있는 섹션의 전체 크기를 합하였으므로, 디스크 상의 실제 PE 파일의 크기보다 크다. SizeOfHeaders 필드는 PE 파일의 전체 헤더 크기를 갖는다. CheckSum 필드는 예전에는 중요하지 않았던 필드였으나 보안이 강조되면서 중요한 역할을 하는 필드이다. PE 구조의 특정 내용을 변경하였을 때 이 필드의 값을 체크하여 내용의 변경이 있었는지를 판별할 수 있으며 CRC 코드와 비슷한 역할을 한다.

PE 헤더에 뒤에오는 IMAGE_SECTION_HEADER 구조체는 각 Section 별로 존재한다. 다음 [그림 14]는 IMAGE_SECTION_HEADER 구조체 이다.

```
typedef struct _IMAGE_SECTION_HEADER {
 BYTE Name[IMAGE_SIZEOF_SHORT_NAME];
 union {
 DWORD PhysicalAddress;
 DWORD VirtualSize;
 } Misc;
 DWORD VirtualAddress;
 DWORD SizeOfRawData;
 DWORD PointerToRawData;
 DWORD PointerToRelocations;
 DWORD PointerToLinenumbers;
 WORD NumberOfRelocations;
 WORD NumberOfLinenumbers;
 DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

[그림 14] IMAGE_SECTION_HEADER 구조체

IMAGE_SECTION_HEADER 구조체에서 중요하게 사용되는 필드는 VirtualSize, VirtualAddress, SizeOfRawData, PointerToRawData, Characteristics 필드이다. VirtualSize 필드는 메모리에서 섹션이 차지하는 크기, VirtualAddress 필드는 메모리에서 섹션의 시작주소, SizeOfRawData는 파일에서 섹션이 차지하는 크기, PointerToRawData 필드는 파일에서 섹션의 시작 위치 이다. Characteristics 필드는 패킹 여부 탐지에서 사용되는 필드인데 이 값은 각 섹션의 속성을 나타내며 진입점 섹션의 속성 값중 “WRITE” 속성 값이 존재하면 패킹된 PE 파일이라고 판별 한다. 다음 [그림 15]는 Characteristics필드에 존재하는 플래그이다.

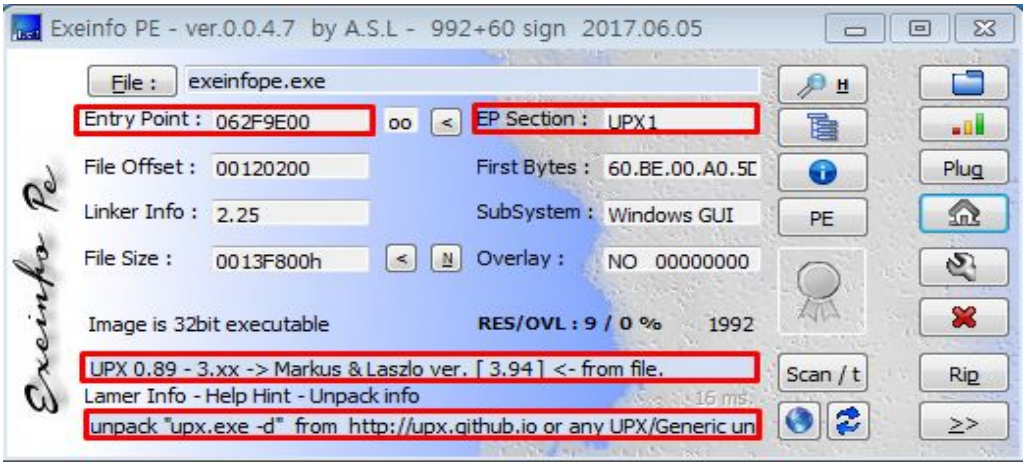
```
#define IMAGE_SCN_MEM_DISCARDABLE      0x02000000
#define IMAGE_SCN_MEM_NOT_CACHED       0x04000000
#define IMAGE_SCN_MEM_NOT_PAGED        0x08000000
#define IMAGE_SCN_MEM_SHARED           0x10000000
#define IMAGE_SCN_MEM_EXECUTE          0x20000000
#define IMAGE_SCN_MEM_READ             0x40000000
#define IMAGE_SCN_MEM_WRITE            0x80000000
```

[그림 15] Characteristics 필드 플래그

2.2.3 PE 분석 툴

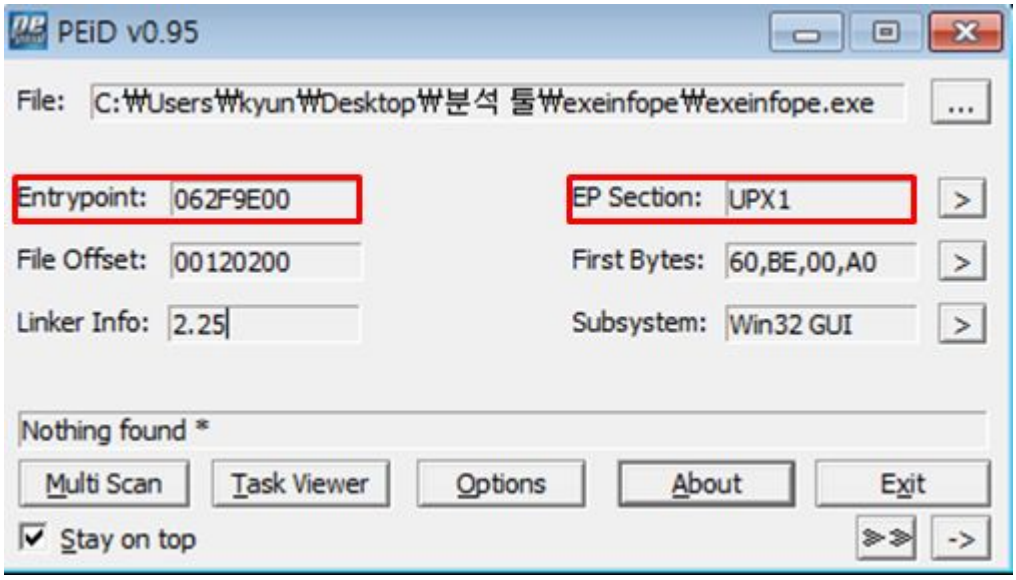
다양한 PE 분석 툴이 존재하며 일반적으로 패킹 여부를 탐지해주며 시그니쳐 기반으로 패커 여부를 탐지하기 때문에 시그니쳐가 존재하는 패커의 경우 패킹된 패커의 종류 및 언패킹을 도와주는 URL 까지 제공해준다. 하지만, 제공되는 URL에서 모두 언패킹이 가능한 것은 아니며, 언패킹을 직접 수행해주는 PE 분석 툴은 드물다. PE 분석 시에 자주 사용되는 툴은 Exeinfope, PEiD, PE view, PE Frame 등이 있다.

Exeinfope[17]는 본 시스템을 구현하면서 검증을 위해 가장 많이 사용한 PE 분석 툴이다. Exeinfope는 PE 파일의 기본 정보를 분석해주며 시그니쳐 기반으로 패커 종류를 탐지해준다. 시그니쳐 가 존재하는 패커의 경우 패커 이름과 언패킹 URL을 제공해준다. 다음 [그림 16]는 Exeinfope를 이용하여 Exeinfope.exe를 분석한 결과 화면이다.



[그림 16] Exeinfope(Exeinfope.exe) 분석 결과 화면

PEiD도 본 시스템을 구현할 때 검증을 위해 사용하였으며 현재는 개발을 멈춘 상태이다. 다음 [그림 17]는 PEiD를 이용하여 Exeinfope.exe를 분석한 결과 화면이다.



[그림 17] PEiD(Exeinfope.exe) 분석 결과 화면

Exeinfope와 PEiD 모두 같은 결과를 출력해주지만 PEiD에서는 패킹 결과가 출력되지 않았다. 이는 시그니쳐 기반으로 패커 탐지를 위해서 시그니쳐가 저장되어 있는 userdb.txt 파일이 업데이트 되어있지 않아서 나온 결과이다. 제 4장에서 검증에 대해서 서술할 때 위의 분석 툴의 결과와 본 시스템의 분석 결과를 비교하여 검증한다.

2.3 패킹(실행 압축)

본 절에서는 패킹과 언패킹에 대해서 설명하고 잘 알려진 패커 소개, 패킹 여부/패커 종류 탐지 방법에 대해서 서술한다.

2.3.1 패킹 및 언패킹

패킹이란 간단하게 실행 파일을 압축하는 것이다. 일반 압축과 다르게 단지 파일을 묶어주는 것이 아니라, 실행 파일의 형태를 유지하면서 파일 크기를 줄이는 압축 방식이다. 보통의 압축 파일 확장자인 zip, rar 등이 아니라, PE 파일을 압축하는 것이므로 확장자는 exe, dll, sys, obj 등이 해당된다.

패킹의 초기 목적은 일반 압축과 동일하게 파일의 크기를 줄여 저장 공간을 더 확보하기 위함이었다. 이는, 과거 저장 공간이 부족하던 시절의 목적이고 현재 목적은 패킹함으로써 코드의 은닉화 및 Anti-Reversing의 목적이 더 짙다고 볼 수 있다.

패킹은 크게 Compressor와 Protector 두 가지로 나누어 분류 할 수 있다. Compressor는 과거 저장 공간이 부족할 때 파일의 크기를 줄여 배포의 용이함을 주 목적으로 갖는다. 따라서, Compressor은 코드의 은닉화를 통한 분석 방해 보다는 단순히 일반 압축과 동일한 상황에 사용된다. Compressor를 대표하는 패커로는 UPX, Aspack, Nspack 등이 있다. Protector의 주 목적은 패킹하는 PE 파일을 보호하기 위함이며 Anti-VM, Anti-Debug, Garbage Code와 같은 Anti-Reversing 기법이 포함되어 있으므로 분석을 어렵게 만든다. 주로 상용 프로그램에서 파일을 패킹하거나 악성코드 제작자가 악성코드를 패킹할 때 사용된다. Protector을 대표하는 패커로는 Themida, Yoda's Protector, AsProtector, Upack 등이 있다. 다음 표 1은 일반 압축과 실행 압축을 비교한 것이다.

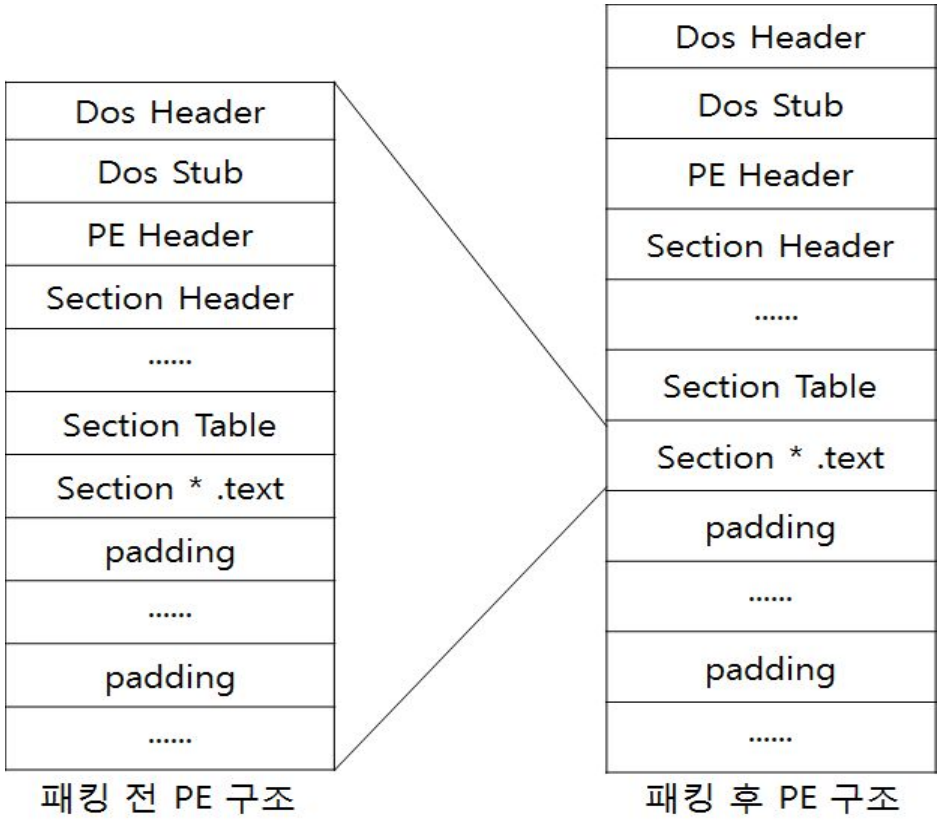
카테고리	일반 압축	패킹(실행 압축)
대상	모든 파일	PE 파일
결과물 확장자	zip, rar ...	exe, dll, sys, obj .....
압축 해제 방식	자체 실행 불가	자체 실행 가능
자체 실행 여부	불가능	가능
장점	모든 파일 압축 가능	압축 해제 프로그램 없이 바로 실행 가능
단점	전용 압축 해제 프로그램이 필요	실행시 언패킹 루틴이 호출됨

표 1. 일반 압축과 실행 압축 비교

패킹된 PE 파일의 동작 순서는 파일 어딘가에 삽입되어있는 언패킹 루틴을 수행함으로써 언패킹이 수행되어 원본 파일로 복원된 후 파일포인터가 원본 파일의 Entry Point를 가리키고 패킹 전 파일의 동작 방식과 같게 실행이 된다. 모든 패커가 같은 방식으로 동작하지 않지만 일반적으로 패킹 전의 PE 파일의 모든 데이터가 패킹 후 PE 구조의 어떤 Section에 들어가 있다고 볼 수 있다. 이에 따라, 패킹된 PE 파일의 동작 순서는 패킹된 파일의 Entry Point는 언패킹 루틴을 가리키고 있으며, 언패킹 루틴을 수행하여 패킹 전 파일로 복원한 후 원본의 Entry Point를 찾아 원래 동작 순서를 따라 정상적으로 언패킹이 되면서 PE 파일이 실행된다. Compressor와 Protector에 따라 Decompressing와 Decoding을 수행하여 패킹 전 PE 파일과 동일하게 각 섹션에 데이터를 저장하고 패킹 전 PE 파일의 Entry Point에 위치하게 된다. 따라서, 언패킹의 핵심 요소는 OEP(Original Entry Point)를 찾아내는 것이라고 볼 수 있다.

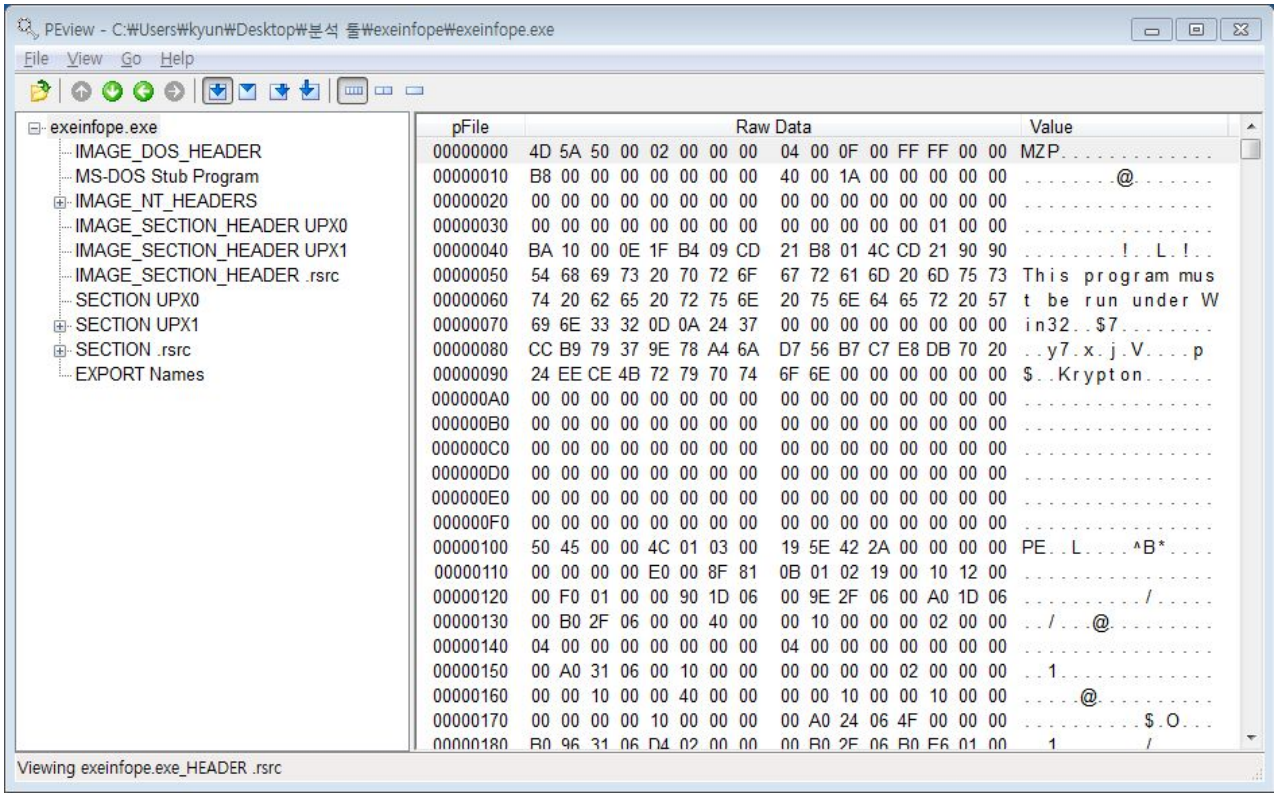
대부분의 패커들이 위의 패킹 동작 방식으로 패킹을 수행한다고 볼 수 있다. 언패킹 루틴은 각 패커마다 다르며 언패킹 루틴 즉, 언패킹 루틴을 분석하여 언패킹 알고리즘을 자동으로 추출해 낼 수 있다면 악의적인 목적을 가지며 알려지지 않은 패커 또는 직접 개발한 패커로 악성코드를 패킹을 수행해도 언패킹이 가능하기 때문에 정적 분석

이 가능하다. 다음 [그림 18]는 일반적인 패킹 동작 방식이다.



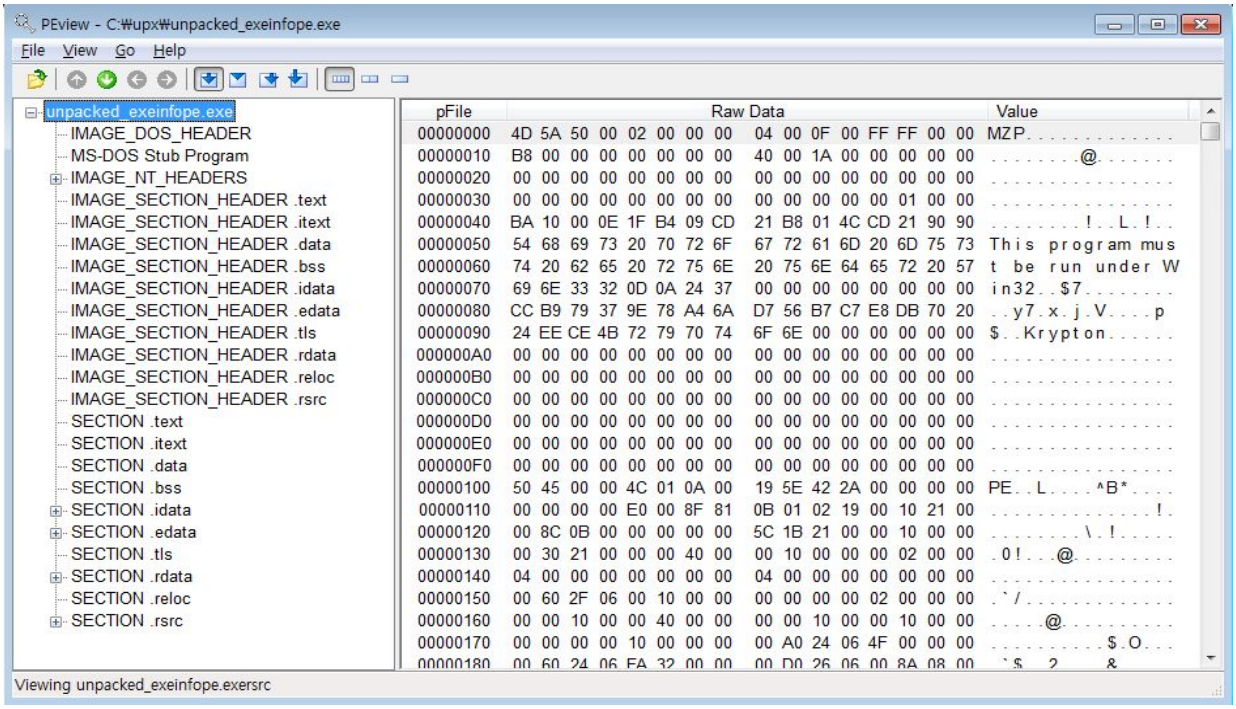
[그림 18] 일반적인 패킹 동작 방식

패킹 전후의 변화를 보면 일반적으로 대부분의 값이 변경된다. 그중 Entry Point의 값이 변해있는게 중요하다. 앞서 언급했듯이 OEP가 바뀌어 있다는 의미이다. 또한 진입점 섹션의 이름이 변경된다. UPX, Nspack, Aspack, Upack 그리고 Yoda's Protector 등은 진입점 섹션의 이름이 각각 'UPX1', 'nsP0', '.aspack', 'PSㅎ諾揖?', '.yP'로 변경된다. Exeinfope.exe는 UPX로 패킹되어 배포가 된다. 다음 [그림 19]는 UPX로 패킹된 Exeinfope.exe를 PE view를 이용해 PE 구조를 확인한 결과이며 [그림 20]는 UPX 툴로 언패킹을 수행한 파일을 PE view를 이용해 PE 구조를 확인한 결과이다.



[그림 19] Exeinfope(UPX)

[그림 20]의 PE 구조에서 앞서 언급했던 것과는 다르게 섹션 부분이 변경되었으며 SECTION UPX0 섹션에는 값들이 '0'으로 채워져 있다. SECTION UPX1 섹션에는 알 수 없는 문자열들로 채워져 있다. UPX로 패킹된 PE 파일을 실행할 때 UPX1 섹션의 데이터들이 UPX0 섹션에 있는 '0'값들에 패킹 전 데이터로 복원되 나가면서 언패킹을 수행한다.



[그림 20] Exeinfope(None)

[그림 20]의 PE 구조는 일반 PE 파일의 PE 구조와 동일하다. 따라서, 패킹을 했을 경우 PE 구조와 내부의 데이터들이 바뀌는 것을 알 수 있다. 하지만, Dos 헤더의 값들은 변하지 않았다. UPX로 패킹된 Exeinfope를 실행하면 언패킹 루틴을 수행한 후 언패킹된 Exeinfope와 같은 PE 구조로 복원된 후 정상 실행 된다.

일반적으로 패킹 파일은 Single_Layer File 즉, 한번의 패킹이 이루어진 파일과 한 개의 패커로 여러번 패킹한 Re_Packing File 그리고 여러 개의 패커로 여러번 패킹한 Multi_Layer File로 구분될 수 있다.

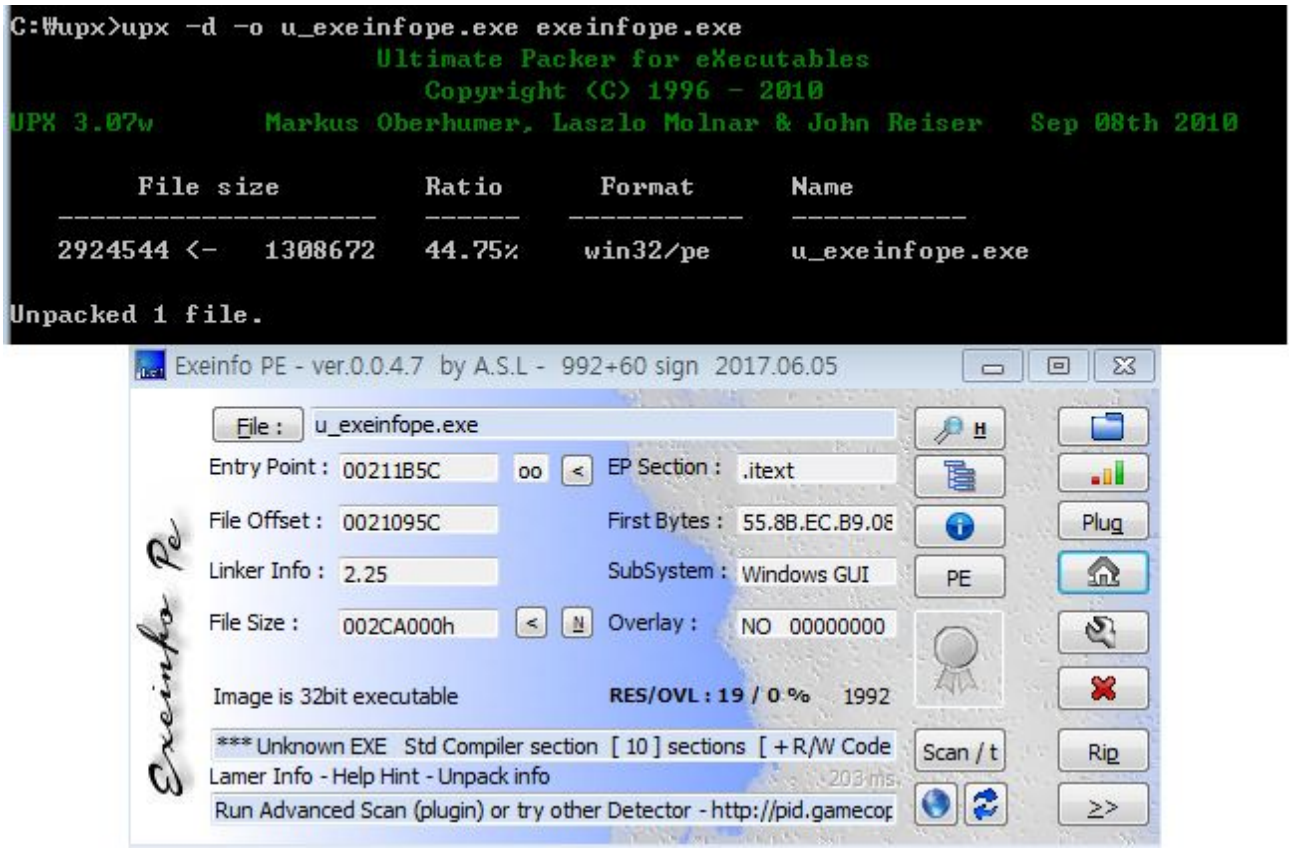
2.3.2 패커(Packer)

각 패커 마다 특징이 있으며, 현재 계속 서비스하는 패커와 서비스가 종료된 패커 그리고 언패킹 방법이 공개된 패커, 언패킹 툴이 존재하는 패커 등등 여러 패커가 존재한다. UPX, Aspack, Nspack, Yoda's Protector, Upack, Themida 등이 있으며 각각 프

리웨어, 사용화, 암호화, 악의적인 목적 등의 다양한 유형의 패커들이 존재한다. 본 논문에서는 UPX[18], Aspack[19], Nspack, Yoda's Protector, Upack등 시그니쳐 기반으로 탐지 할 수 있는 패커들을 잘 알려진 패커로 정의 한다. Custom 패커는 시그니쳐 기반으로 탐지가 불가능한 패커를 모두 Custom 패커라고 정의 한다.

잘 알려진 패커인 UPX는 패킹 툴에 언패킹 기능까지 탑재하고 있다. 따라서, 악성코드가 UPX로 패킹된 경우 언패킹이 가능하다.

Aspack의 경우 Aspack에서 제공하는 툴은 아니지만 언패킹 툴이 존재한다. 하지만, 계속되는 업데이트를 따라가지 못하여 현재 버전인 2.4 버전에서는 사용이 불가하며 MUP(Manual Unpacking) 방법을 사용하여 언패킹을 진행해야 한다. Nspack, Yoda's Protector, Upack도 마찬가지로 제공되는 툴이 존재하지 않으며 MUP 방법으로 언패킹을 진행해야 한다. 다음 [그림 21]는 UPX[툴을 이용하여 Exeinfope.ex의 언패킹 수행 결과이다.



[그림 21] UPX 언패킹 수행 결과

위에 언급한 5개의 패커들 말고도 많은 패커들이 존재하며, 각 패커들 고유의 언패킹 루틴 알고리즘으로 언패킹을 진행한다.

2.3.3 엔트로피(Entropy)

Lyda[20]는 엔트로피 분석을 기반으로 암호화, 압축 그리고 패킹된 악성코드를 탐지하는 기술을 설명했다. 엔트로피 계산은 파일에 존재하는 바이트의 의미는 중요하지 않고 실행파일에 포함된 바이트 즉, 문자열의 발생 빈도를 이용하여 엔트로피 값을 계산한다. Lyda는 전체파일에 대한 엔트로피 값 계산을 이용하여 패킹된 파일과 일반 파일을 구분할 수 있다고 말하고 있다. 하지만, 패킹이 되어 있지 않은 파일에서 엔트로피 값이 높게 계산되어 나오는 오탐이 발생할 수 있거나 패킹된 파일의 엔트로피 값이 낮게 나오는 미탐이 발생할 수 있다. 수식 (1)은 파일의 엔트로피를 계산하는 수식이다.

$$H(p) = -\sum_{x \in X} p(x) \log p(x) \tag{1}$$

일반적으로 알고 있는 엔트로피 개념은 열역학적 계의 상태 함수로 통계적인 무질서도를 나타내는 개념이다. Shannon[x]은 정보 엔트로피 개념을 사용하여 정보의 양을 수치화 하였고 이는 어떤 확률변수의 불확실성을 측정하는 것이다. 위의 수식에서 p(x)는 x가 발생할 확률이고 log p(x)는 이산 확률 변수 X의 자기 정보량을 의미한다 log의 밑으로 보통 2, 오일러 수 e, 그리고 10을 사용한다.

일반적으로 정보 이론에서의 엔트로피는 메시지 압축 분야에 대한 연구에서 자주 사용되며 압축 알고리즘의 압축률을 평가할 때 아주 좋은 지표가 된다. 엔트로피가 높은 데이터일수록 나타날 수 있는 모든 비트들이 골고루 존재함을 의미한다. 따라서, 패킹

된 파일의 엔트로피 계산 값이 높을수록 압축률이 높다는 의미가 된다.[Lyda] Lyda의 실험에 따르면 텍스트 파일, 실행 파일, 패킹한 실행 파일, 암호화된 실행 파일의 평균 엔트로피 값이 4.347, 5.099, 6.081 그리고 7.175의 값을 갖는다. 이는 엔트로피 값을 비교하여 패킹된 파일과 패킹되지 않은 파일을 구분할 수 있다.

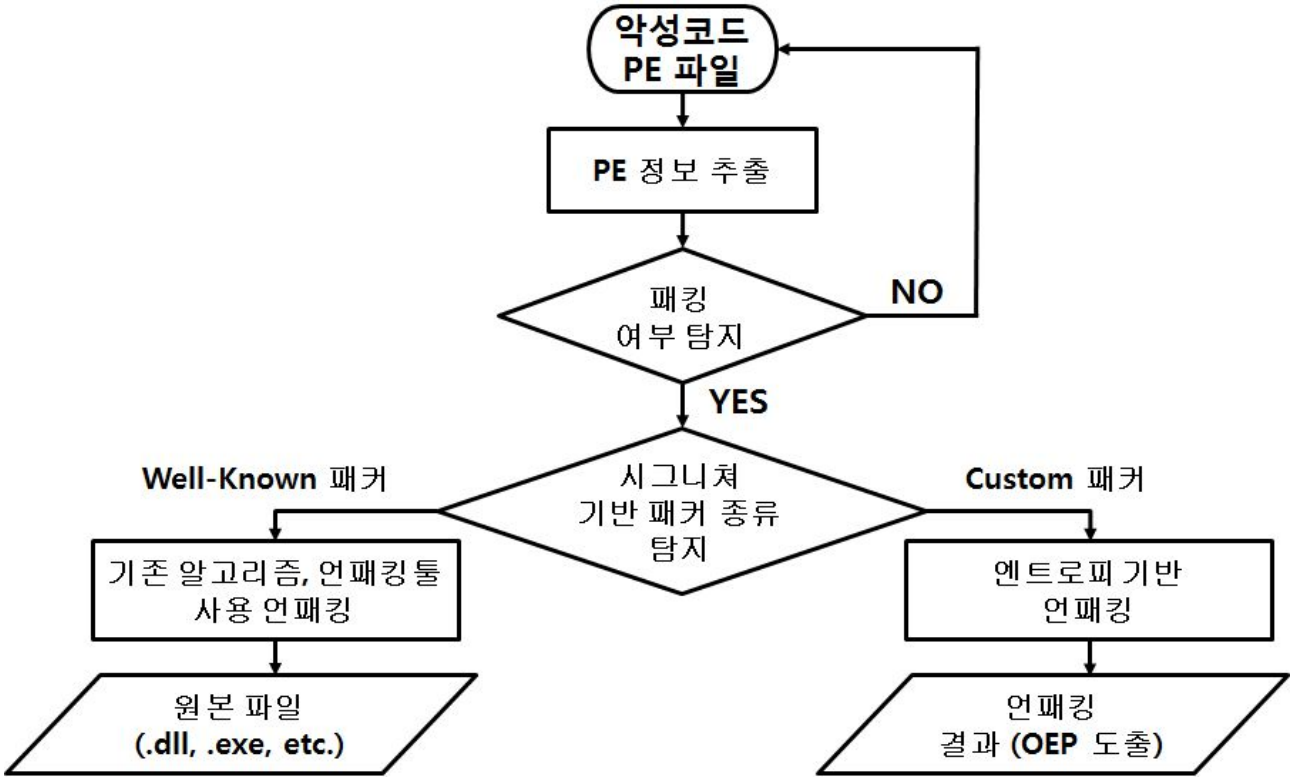
본 시스템에서도 패킹 여부 탐지를 위해 엔트로피 값을 사용한다. 본 시스템의 실험 결과 패킹된 파일의 대부분이 엔트로피 값이 7이상이였고 아주 드물게 6.8의 값을 갖는 파일도 존재했다. 엔트로피 값을 계산하기 위해서는 파일의 모든 스트링을 읽어와 계산하여야 하므로 대상 파일의 크기가 커질수록 엔트로피 값 계산에 소요되는 시간이 늘어났다.

3. 악성코드 분석을 위한 PE 파일 언패킹 자동 시스템 설계 및 구현

본 장에서는 패킹 여부/패커 종류 탐지 및 언패킹 자동 시스템의 설계 및 구현에 대해서 서술한다. 제 3.1절에서는 제안하는 시스템의 순서도에 대해서 설명한다. 제 3.2절에서는 PE 분석 시스템에 대해서 서술한다. 제 3.3절에서는 패킹 여부 탐지 알고리즘을 서술한다. 제 3.4절에서는 패커 종류 탐지 알고리즘을 서술한다.

3.1 시스템 순서도

본 절에서는 악성코드 분석을 위한 PE 파일 언패킹 자동 시스템의 순서도 대해 서술한다. PE 정보 추출, 패킹 여부 탐지, 패커 종류 탐지 그리고 언패킹 가능한 패커 자동 언패킹 수행의 단계로 구성된다. 다음 [그림 22]는 제안하는 시스템의 순서도를 나타낸다.



[그림 22] 악성코드 분석을 위한 PE 파일 언패킹 자동 시스템 순서도

본 시스템에 악성코드 및 PE 파일이 입력으로 주어지면 먼저 PE 정보를 추출한다. PE 정보를 추출할 때 PE 시그니쳐를 검사하여 PE 파일이 아닐 경우 본 시스템은 분석을 더 이상 진행하지 않는다. 다음으로 추출된 PE 정보를 기반으로 패킹 여부 탐지를 수행하여 패킹이 되어있지 않다면 다음 입력을 받는다. 패킹이 되어있다면 시그니처 기반 패커 종류 탐지를 수행한다. 잘 알려진 패커중 언패킹이 가능한 패커의 경우 자동으로 언패킹을 수행하여 패킹이 풀린 원본 파일을 출력한다. PE 정보 및 패킹 정보들은 DB에 저장된다.

3.2 PE 분석

본 절에서는 악성코드 및 PE 파일의 PE 정보를 추출하는 PE 분석 시스템에 대해서

서술한다. PE 분석은 Exeinfope와 같은 PE 분석툴에서는 볼 수 없는 파일의 세부 정보를 추출한다. 엔트로피 값, 각 섹션별 엔트로피 값, 각 섹션벌 속성 값 등 다양한 값을 추출한다. 본 시스템에서 추출하는 정보 다음 표 2와 같다.

File_Name	Dll_Chacking	File_Bit	File_Offset	File_ Imagebase
File_Entropy	File_Firstbytes	File_Size	File_Machine	Section_ Number
PE_Signature	EPS_Name	EPS_Address	EPS_Entropy	EPS_ Characteristics
DOS_Signature	Packing_Detect	Packer_Name	AVM_Detect	ADBG_Detect

표 2 시스템 추출 정보

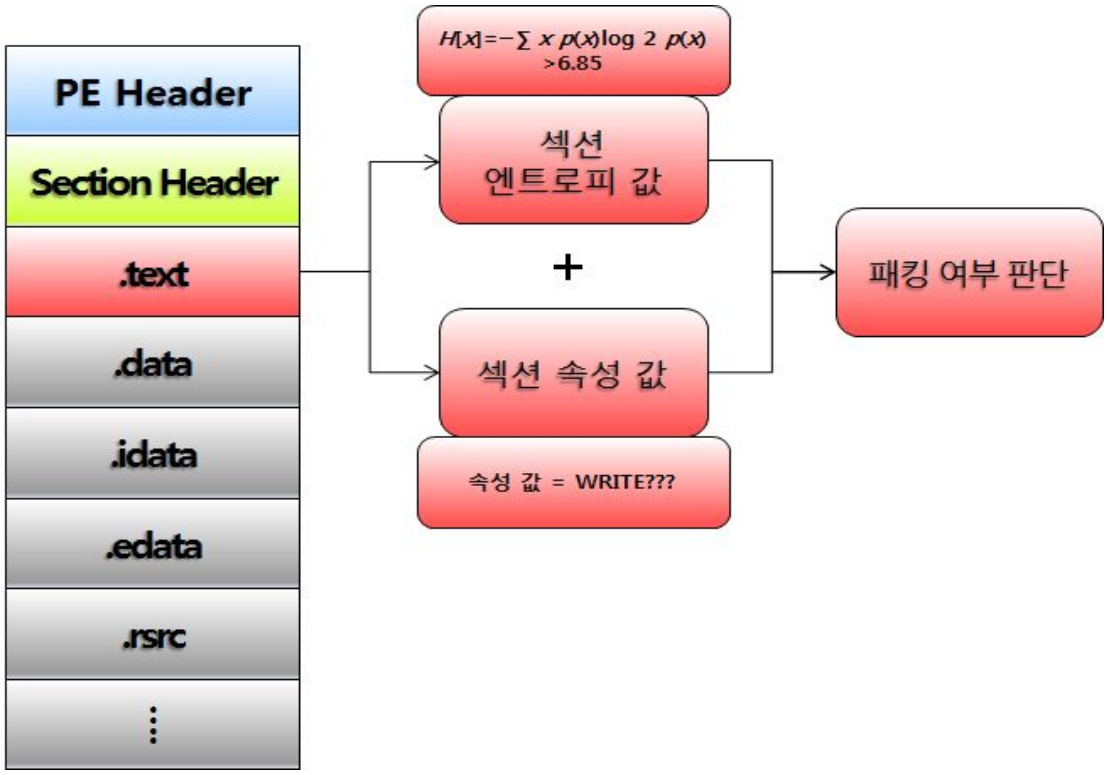
PE 파일의 기본적인 정보와 패킹 여부 탐지, 패커 종류 탐지와 간단한 Anti-VM 탐지, Anti-Debug 탐지 정보를 추출해낸다.

3.3 패킹 여부 탐지

본 절에서는 패킹 여부 탐지 방법에 대해서 서술한다. 패킹 여부를 탐지하기 위해서는 앞서 언급했던 정보 엔트로피를 기반으로 패킹 여부를 탐지한다. 패킹 여부 탐지 알고리즘은 추출된 PE 정보가운데 진입점 섹션의 엔트로피 값과 진입점 섹션의 속성 값을 기반으로 탐지한다. 파일 전체 엔트로피 값으로 패킹 여부를 탐지하게 되면 패킹된 파일과 일반 PE 파일과의 값이 겹쳐지는 모호한 부분이 생긴다.[21, 22, 23] 하지만, 진입점 섹션의 엔트로피 값으로 패킹 여부를 탐지하게 되면 패킹된 파일과 일반 PE 파일과의 값이 겹쳐지지 않아 모호한 부분이 없이 분명하게 판단할 수 있다. 따라서, 패킹 여부 탐지를 위해서는 진입점 섹션의 엔트로피 값을 비교해야 한다. 본 시스템에

서는 진입점 섹션 엔트로피 값의 기준을 6.85로 설정하였다. 이는 진입점 섹션의 엔트로피 값의 경계이다.

추출된 PE 정보가운데 진입점 섹션의 속성 값을 기준으로 'WRITE' 속성일 경우에는 패킹이 되어있다고 탐지한다. 진입점 섹션의 엔트로피 값이 기준 보다 현저히 낮은 경우에도 패킹이 되어있는 경우가 있다. 대표적으로 파일 자체의 엔트로피 값이 낮을 경우와 Aspack으로 패킹 하였을 때 Aspack 같은 경우 모든 파일은 아니지만 대부분의 경우가 5.9 정도로 수렴하였다. 따라서, 엔트로피 값이 아닌 다른 비교 조건이 필요 하였다. 패킹된 PE 파일은 파일이 실행되면서 언패킹 루틴 코드가 필요하다. 해제된 데이터를 쓰는 권한이 필요하며 이는 'READ', 'EXECUTE' 뿐만 아니라 'WRITE' 속성 값이 꼭 필요하다. 다음 [그림 23]는 패킹 여부 판단 프로세스 이다.



[그림 23] 패킹 여부 판단 프로세스

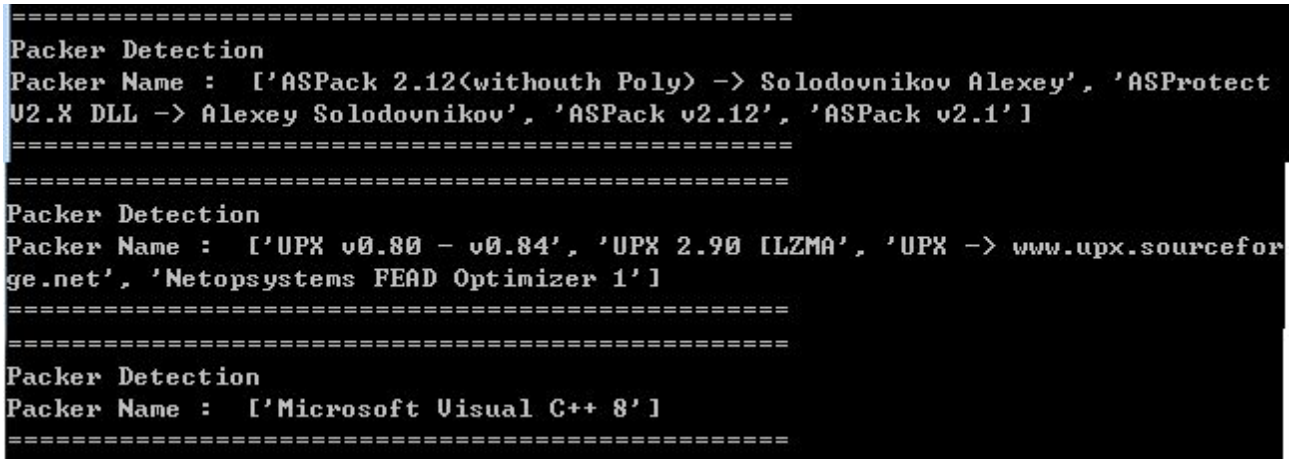
다음 [그림 24]는 패킹 여부 탐지를 수행하는 알고리즘이다.

```
Procedure Packing_Detect
Input File : 악성코드, PE 파일
Ouput PD : 패킹 여부 탐지
(1) Analysis PE_File;    //PE 파일 분석
(2) if(EPS_Characteristics  == "WRITE")
(3)     EPS_Char = TRUE
(4) end-if
(5) for e := 0 to length  in matrix[]   //파일의 크기만큼 스트링 저장
(6)     if(matrix[i]  != 0)
(7)         t = matrix[i]/length
(8)         EPS_Entropy = EPS_Entropy + t * log2 1/t
(9)      end-if
(10)end-for
(11)if(EPS_Entropy > 6.85 || EPS_Char == TRUE)
(12)PD = TRUE
```

[그림 24] 패킹 여부 탐지 알고리즘

3.4 패커 종류 탐지

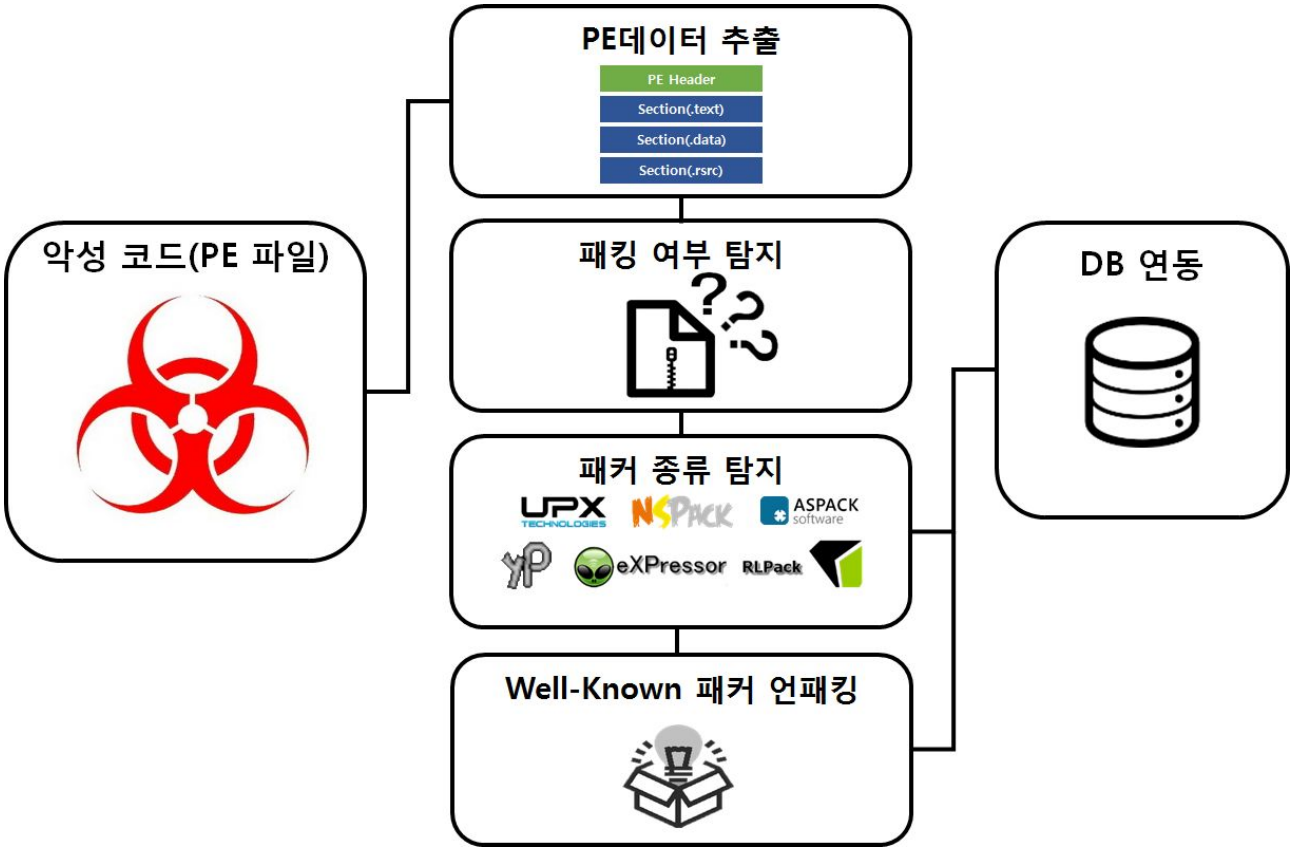
본 절에서는 시그니처 기반 패커 종류 탐지에 대해서 설명한다. 잘 알려진 패커들은 대부분 시그니처를 가지고 있고 이 시그니처를 파일의 문자열에서 검색하여 나타나면 어떤 패커로 패킹이 되었는지 찾아 낼 수있다. 시그니처는 userdb.txt 파일에 들어있으며 이는 대부분의 PE 분석 툴에서 사용한다. 본 시스템에서는 userdb.txt를 업데이트하여 약 6000개의 시그니처가 존재한다. 시그니처는 패커의 버전마다 존재하여 개수가 많다. 다음 [그림 25]는 본 시스템에서 시그니처 기반으로 패커 종류를 탐지한 결과 이다. Aspack, Upx, 패킹이 되지 않은 파일을 탐지하였다.



[그림 25] 시그니처 기반 패커 종류 탐지 결과

기존 PE 분석 툴에서는 대부분의 악성코드 분석 진행이 되지 않았다. Exeinfope 및 PEiD의 경우 대부분의 필드가 '??'로 채워져 있으며 Ollydbg의 경우 오류가 발생하면서 로드가 되지 않았다. 하지만, 본 시스템에서는 악성코드의 PE 정보 추출과 패킹 여부 탐지가 가능하다.

본 시스템에서 Anti-VM과 Anti-Debug 탐지를 수행하지만 아주 기본적인 부분만 수행한다. 여러 가지의 Anti-VM과 Anti-Debug 탐지 방법이 존재하지만 본 시스템에서는 시그니처 기반으로 탐지하는 방법만을 적용하였다. 다음 [그림 26]는 본 시스템의 전체 구성도 이다.



[그림 26] 시스템 전체 구성도

본 시스템은 모든 PE 데이터를 추출한다. 하지만, 파일의 기본 정보, 패커 종류 탐지, 패킹 종류 탐지에 필요한 정보와 Anti-VM 탐지, Anti-Debug 탐지, 패킹 유무, 패커 이름 정보만을 DB에 저장한다. 다음 [그림 27]는 DB에 저장된 PE Metadata 정보의 일부이다.

File_Name	Dll_Chack	File_Bit	File_Offset	File_Size	File_Entropy	File_Firstby	File_Imagebase	File_Machine	Section_Number	EPS_Name	EPS_Address	Packing_Detect	AVM_Detect	ADBG_Detect
AdapterTroul	0	32bit	0x12ce	27517.2	5.65732	e8 fe 02 0	0x1000000	IMAGE_FILE_M	4	.text	0x1ece	1	0	1
alg.exe	0	64bit	0xb1fc	59732.2	6.02139	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0xbdfc	1	0	1
appcmd.exe	0	32bit	0xb441	129834	6.67322	e8 11 04 0	0x1000000	IMAGE_FILE_M	4	.text	0xc041	1	0	1
AppLaunch.e	0	64bit	0xab70	52286.7	6.06856	48 83 ec 20	0x10000000	IMAGE_FILE_M	4	.text	0xb770	1	0	1
BdeHdCfg.ex	0	64bit	0x35e8	115038	7.04888	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0x41e8	1	0	1
BitLockerWiz	0	64bit	0x10fc	87594.7	6.983	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0x1cfc	1	0	1
CIDAEMON.E	0	64bit	0x147c	11015.7	5.37875	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0x207c	1	0	1
cmd.exe	0	32bit	0x54dc	173036	4.5903	e8 50 fe ff	0x4ad00000	IMAGE_FILE_M	4	.text	0x60dc	0	1	1
cmdl32.exe	0	32bit	0x8ad2	59541.3	6.55164	e8 1a 06 0	0x1000000	IMAGE_FILE_M	4	.text	0x96d2	1	0	1
cmmon32.ex	0	32bit	0x64df	31438.9	5.84801	e8 3d 06 0	0x1000000	IMAGE_FILE_M	4	.text	0x70df	1	0	1
cmstp.exe	0	32bit	0xd801	64185.3	6.04154	e8 2b 06 0	0x1000000	IMAGE_FILE_M	4	.text	0xe401	1	0	1
CNFNOT32.E	0	32bit	0x16cb5	116161	6.1383	e8 24 fc ff	0x2e000000	IMAGE_FILE_M	4	.text	0x178b5	1	0	1
ComSvcConf	0	32bit	0x24efe	115050	5.55396	ff 25 00 2	0x400000	IMAGE_FILE_M	3	.text	0x25efe	1	0	0
DataSvcUtil.e	0	32bit	0xeb0e	49024.2	5.18829	ff 25 00 2	0x400000	IMAGE_FILE_M	3	.text	0xfb0e	1	0	0
Defrag.exe	0	64bit	0xb6bc	166273	7.25701	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0xc2bc	1	0	1
DeviceEject.e	0	64bit	0x1c78	16715.9	5.22373	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0x2878	1	0	1
dfrgui.exe	0	32bit	0x135d9	478473	6.52369	e8 23 03 0	0x1000000	IMAGE_FILE_M	4	.text	0x141d9	1	0	1
diantz.exe	0	32bit	0xd040	78486	6.62889	e8 2c 03 0	0x1000000	IMAGE_FILE_M	4	.text	0xdc40	1	0	1
Dism.exe	0	32bit	0x196c0	147254	5.81022	e8 b1 0d 0	0x1000000	IMAGE_FILE_M	4	.text	0x1a2c0	1	0	1
ditrace.exe	0	64bit	0x2d064	228828	6.30587	48 83 ec 20	0x400000	IMAGE_FILE_M	5	.text	0x2dc64	1	0	1
dnscacheugc	0	32bit	0x3d8e	19924.8	5.55939	e8 a4 03 0	0x1000000	IMAGE_FILE_M	4	.text	0x498e	1	0	1
dplaysvr.exe	0	32bit	0x2a95	23279.4	6.38143	e8 07 03 0	0x1000000	IMAGE_FILE_M	4	.text	0x3695	1	0	1
dxdiag.exe	0	32bit	0x1de76	205991	6.22554	e8 96 37 0	0x1000000	IMAGE_FILE_M	4	.text	0x1ea76	1	0	1
Dxpserver.ex	0	64bit	0x2f8a8	195172	5.88719	48 83 ec 20	0x100000000L	IMAGE_FILE_M	5	.text	0x302a8	1	0	1

[그림 27] PE Metadata 정보(DB)

다음 [그림 28]는 DB에 저장된 Packingdata 정보의 일부이다.

File_Name	Packing_Detect	EPS_Entropy	EPS_Characteristics	Packer_Name
AddInProce	1	4.70522	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
AddInProce	1	4.70807	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
AddInUtil.e	1	4.6067	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
aitagent.ex	1	6.43215	0x60000020	Microsoft Visual C++ 8.0 (DLL),Microsoft Visual C++ 8.0
aitstatic.exe	1	6.36909	0x60000020	Microsoft Visual C++ 8.0 (DLL)
AddInProce	1	4.70522	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
AddInProce	1	4.70807	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
AddInUtil.e	1	4.6067	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
aitagent.ex	1	6.43215	0x60000020	Microsoft Visual C++ 8.0 (DLL),Microsoft Visual C++ 8.0
aitstatic.exe	1	6.36909	0x60000020	Microsoft Visual C++ 8.0 (DLL)
appidpolicy	1	5.75165	0x60000020	Microsoft Visual C++ 8.0 (DLL)
ARP.EXE	1	6.40092	0x60000020	Microsoft Visual C++ 8,VC8 -> Microsoft Corporation,Visual C++ 2005 Release -> Microsoft
ARPPRODU	1	5.4912	0x60000020	Microsoft Visual C++ 8,VC8 -> Microsoft Corporation,Visual C++ 2005 Release -> Microsoft
aspnetca.ex	1	6.54255	0x60000020	Microsoft Visual C++ 8,VC8 -> Microsoft Corporation,Visual C++ 2005 Release -> Microsoft
AddInProce	1	4.70522	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
AddInProce	1	4.70807	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
AddInUtil.e	1	4.6067	0x60000020	Morphine v1.2 (DLL),Microsoft Visual C# / Basic .NET,Microsoft Visual Studio .NET,.NET executable,Microsoft Visual C# v7.0 / Basic .NET
aitagent.ex	1	6.43215	0x60000020	Microsoft Visual C++ 8.0 (DLL),Microsoft Visual C++ 8.0
aitstatic.exe	1	6.36909	0x60000020	Microsoft Visual C++ 8.0 (DLL)
appidpolicy	1	5.75165	0x60000020	Microsoft Visual C++ 8.0 (DLL)
ARP.EXE	1	6.40092	0x60000020	Microsoft Visual C++ 8,VC8 -> Microsoft Corporation,Visual C++ 2005 Release -> Microsoft
ARPPRODU	1	5.4912	0x60000020	Microsoft Visual C++ 8,VC8 -> Microsoft Corporation,Visual C++ 2005 Release -> Microsoft
aspnetca.ex	1	6.54255	0x60000020	Microsoft Visual C++ 8,VC8 -> Microsoft Corporation,Visual C++ 2005 Release -> Microsoft

[그림 28] Packingdata 정보(DB)

4. 검증

본 장에서는 악성코드 분석을 위한 PE 파일 언패킹 자동 시스템의 검증에 대해 서술한다. 제 4.1절에서는 PE 정보 추출 모듈을 기존 PE 분석 툴과 비교하여 검증한다. 제 4.2절에서는 패킹 여부 탐지 모듈을 미리 패킹해놓은 데이터셋과 비교하여 검증한다. 제 4.3절에서는 패커 종류 탐지 모듈을 각 패커로 패킹해놓은 데이터셋과 비교하여 검증한다. 제 4.4절에서는 UPX로 패킹된 파일에 대해서 언패킹 수행을 검증한다. 제 4.5절에서는 악성코드 분석 결과를 설명한다. 본 시스템의 환경은 windows 7 64bit 이다.

4.1 PE 정보 추출 검증

본 절에서는 PE 정보 추출 검증에 대해서 서술한다. 기존 분석 툴에서 추출해주는 PE 정보와 본 시스템에서 추출해주는 PE 정보를 비교하여 본 시스템을 검증한다. 웹에서 다운 받을 수 있는 파일 10개를 선정하여 임의로 UPX, Aspack, Nspack, Upack, Yoda's Protector 패커 중 2개씩 패킹 후 Exeinfope를 사용하여 추출된 PE 정보와 본 시스템을 사용하여 추출된 PE 정보를 비교하였다. 다음 [그림 29]는 Exeinfope에서 추출된 PE 정보 이며, [그림 30]는 본 시스템에서 추출된 PE 정보 이다.

File_Name	File_Offset	File_Size	File_Firstbytes	File_Imagebase	File_Machine	EPS_Name	EPS_Address
Anaconda2-4.3.1-Windows-x86.exe	0x28	98657.6	be b0 11 40 00 ad 50 ff 76 34	0x400000	IMAGE_FILE_MACHINE_I386	'PS₩xff₩xd5₩	0x1018
DropboxInstaller.exe	0x9ae80	632192	60 e8 09 00 00 00 fe 3a 14 00	0x400000	IMAGE_FILE_MACHINE_I386	'.rsrc₩x00₩x0	0x143c80
ftp.exe	0x10f	19768.5	e9 41 19 01 00 b4 09 ba 0b 01	0x1000000	IMAGE_FILE_MACHINE_I386	nsp0	0x101b
npp.7.4.1.Installer.exe	0x6601	129903	60 e8 03 00 00 00 e9 eb 04 5d	0x400000	IMAGE_FILE_MACHINE_I386	.aspack	0x6a001
PIL-1.1.7.win32-py2.7.exe	0x17801	100399	60 e8 03 00 00 00 e9 eb 04 5d	0x400000	IMAGE_FILE_MACHINE_I386	.aspack	0x36001
ResTuner_setup.exe	0xe2	89857.4	e9 1b fd 03 00 b4 09 ba 0b 01	0x400000	IMAGE_FILE_MACHINE_I386	nsp0	0x101b
Universal-USB-Installer-1.9.7.9.exe	0x22b49	111050	e8 03 00 00 00 eb 01 c2 bb 55	0x400000	IMAGE_FILE_MACHINE_I386	.yP	0x63549
Wireshark-win32-2.2.6.exe	0x4dd49	229631	e8 03 00 00 00 eb 01 e8 bb 55	0x400000	IMAGE_FILE_MACHINE_I386	.yP	0x7a549
ImmunityDebugger_1_85_setup.exe	0x5980	22724600	60 be 15 f0 43 00 8d be eb 1f	0x400000	IMAGE_FILE_MACHINE_I386	UPX1	0x44780
NEW_GOMPLAYERSETUP.EXE	0x51b0	28581900	60 be 00 40 48 00 8d be 00 d0	0x400000	IMAGE_FILE_MACHINE_I386	UPX1	0x88db0

[그림 29] Exeinfope의 PE 정보 추출 결과

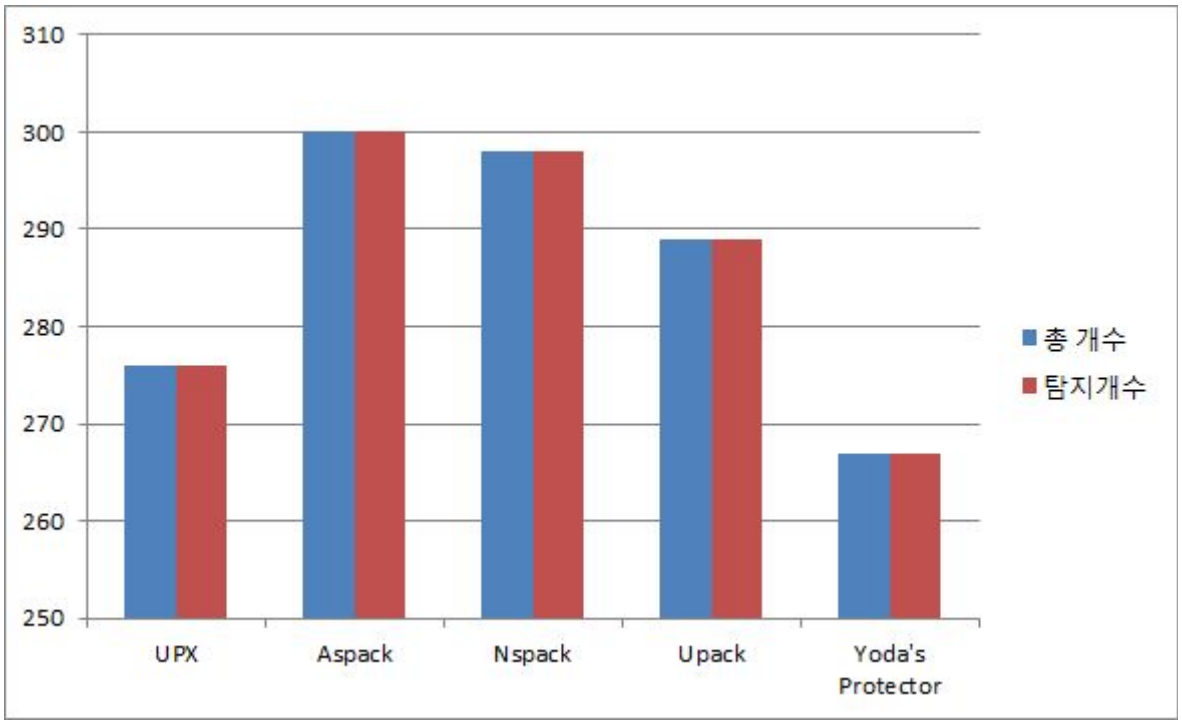
File_Name	File_Offset	File_Size	File_Firstbytes	File_Imagebase	File_Machine	EPS_Name	EPS_Address
Anaconda2-4.3.1-Windows-x86.exe	"00000018"	392052	BE B0 11 40 00 AD 50 FF 76 34	"00400000"	0x14c Intel I386	PS 諧揖?	"00001018"
DropboxInstaller.exe	"0009AE80"	635228	60 E8 09 00 00 00 FE 3A 14 00	"00400000"	0x14c Intel I386	.rsrc	"00143C80"
ftp.exe	"0000001B"	20467	E9 41 19 01 00 B4 09 BA 0B 01	"01000000"	0x14c Intel I386	nsp0	"0000101B"
npp.7.4.1.Installer.exe	"00006601"	165888	60 E8 03 00 00 00 E9 EB 04 5D	"00400000"	0x14c Intel I386	.aspack	"0006A001"
PIL-1.1.7.win32-py2.7.exe	"00017801"	101376	60 E8 03 00 00 00 E9 EB 04 5D	"00400000"	0x14c Intel I386	.aspack	"00036001"
ResTuner_setup.exe	"0000001B"	96418	E9 1B FD 03 00 B4 09 BA 0B 01	"00400000"	0x14c Intel I386	nsp0	"0000101B"
Universal-USB-Installer-1.9.7.9.exe	"00022B49"	155136	E8 03 00 00 00 EB 01 C2 BB 55	"00400000"	0x14c Intel I386	.yP	"00063549"
Wireshark-win32-2.2.6.exe	"0004DD49"	331776	E8 03 00 00 00 EB 01 E8 BB 55	"00400000"	0x14c Intel I386	.yP	"0007A549"
ImmunityDebugger_1_85_setup.exe	"00005980"	22725348	60 BE 15 F0 43 00 8D BE EB 1F	"00400000"	0x14c Intel I386	UPX1	"00044780"
NEW_GOMPLAYERSETUP.EXE	"000051B0"	28582032	60 BE 00 40 48 00 8D BE 00 D0	"00400000"	0x14c Intel I386	UPX1	"00088DB0"

[그림 30] 시스템의 PE 정보 추출 결과

검증 결과 File_Offset과 File_Size를 제외한 정보가 동일하게 추출되었다. File_Size의 경우 근소하게 차이를 보이며, File_Offset의 경우 같은 경우와 다른 경우가 존재한다. 이는 Nspack을 사용하여 패킹하였을 경우 발생하며, Upack으로 패킹하였을 경우 EPS_Name 필드에 “PS₩諾揖?” 라는 값이 저장되었으며, 이때 File_Offset이 다르게 추출되었다. Upack으로 298개의 파일을 패킹하여 확인한 결과 6개의 파일만이 File_Offset 필드가 다른 값으로 추출되었다. Nspack과 Upack의 몇 개의 File_Offset만을 제외 하면 PE 정보 추출의 검증이 잘되었다고 볼 수 있다.

4.2 패킹 여부 탐지 검증

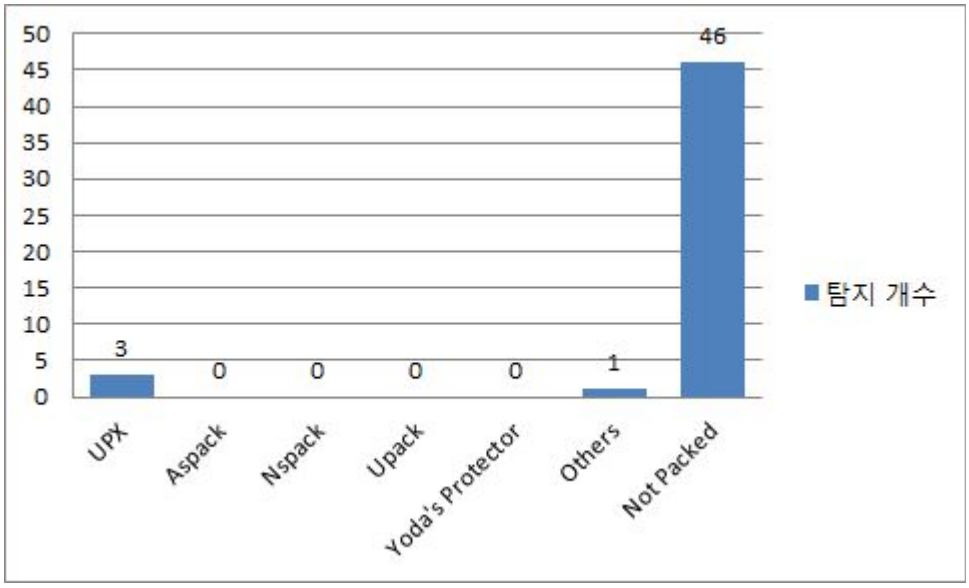
본 절에서는 패킹 여부 탐지 검증에 대해서 서술한다. 미리 패킹 해놓은 파일들의 패킹 여부에 대한 데이터와 본 시스템의 패킹 여부 탐지 결과를 비교한다. PE 파일은 Windows 폴더에 위치한 PE 파일을 대상으로 실험하였다. 똑같은 파일이라도 각 패커마다 패킹 수행 결과가 다르기 때문에 파일의 수가 각 패커 마다 조금 씩 다르다. Windows 폴더에서 700여개의 PE 파일을 가져왔으며, 그중 약 300개의 파일을 무작위로 선택하여 UPX, Aspack, Nspack, Upack, Yoda's Protector을 사용하여 각 파일들을 패킹하였다. 패킹한 파일의 개수는 UPX 276개, Aspack 300개, Nspack 298개, Upack 289개, Yoda's Protector 267개의 파일을 대상으로 패킹 여부 탐지를 검증 하였다. 따라서, UPX의 경우 267개의 파일이 패킹 탐지 결과가 도출되면 정상적으로 패킹 여부를 탐지하는 것이다.. 다음 [그림 31]는 각 패커별 패킹 여부 탐지율이다.



[그림 31] 각 패커별 패킹 여부 탐지율

[그림 31]의 결과 위의 5개 패커로 패킹하였을 경우 100%의 탐지율을 나타냈다. 따

라서, 본 시스템의 패킹 여부 탐지 기능은 정상적으로 동작한다고 볼 수 있다. 다음 [그림 32]는 Program Files(x86)에 있는 exe 파일 대상으로 50개의 파일을 선별하여 패킹 여부 탐지 결과이다. 파일에는 Melon.exe, Xshell.exe, ALFTP.exe, AcroRd32.exe, ALZip.exe, chrome.exe 등 평소 사용자들이 자주 사용하는 파일 이다.



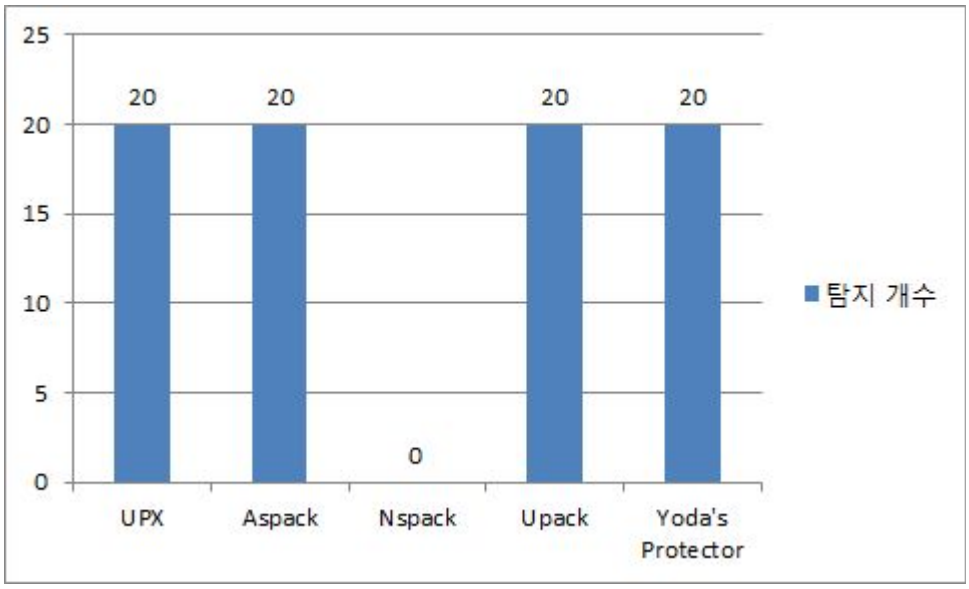
[그림 32] 상용 파일 패킹 여부 탐지 결과

위의 결과 4개의 파일에서 패킹이 탐지되었으며, 앞서 사용한 5개의 패커 이외에 Morphine라는 패커가 존재하였다. 하지만, 위 탐지 결과에서 패커 종류 필드에서 패킹 여부 탐지에서는 탐지 하지 못하였지만 패커 종류에서 탐지된 결과가 4개의 파일이 존재하였고 이 패커는 Armadillo, Nullsoft PiMP Stub 2가지 패커로 패킹 되어있었다.

4.3 패커 종류 탐지 검증

본 절에서는 패킹 종류 탐지 검증에 대해서 서술한다. 미리 패킹 해놓은 파일들의 패커 종류에 대한 데이터와 본 시스템의 패커 종류 탐지 결과를 비교한다. 제 4.2절에서 실험과 동일한 환경에서 각 패커별로 20개씩 선별하여 총 100개의 파일을 대상으로 검

증하였다. 검증 결과로 패커별로 20개씩 도출 된다면 패커 종류 탐지가 정상적으로 동작 하는 것이다. 다음 [그림 33]는 패커 종류 탐지 결과 이다.



[그림 33] 패커 종류 탐지 결과

패커 종류 탐지 결과 UPX, Aspack, Upack, Yoda's Protector의 경우 모두 패커 종류를 올바르게 탐지하였다. 하지만, Nspack의 경우 시그니처 기반으로 탐지가 되지 않았다. 이는, 현재 패킹한 Nspack의 버전이 userdb.txt에 등록 되어있지 않아서라고 판단할 수 있다. 본 시스템의 패커 종류 탐지가 정상적으로 동작한다고 볼 수 있다.

4.4 잘 알려진 패커 언패킹 검증

본 절에서는 UPX로 패킹이 탐지된 파일을 자동으로 언패킹 하여 본 시스템의 패킹 여부 탐지로 언패킹이 올바르게 수행되어 있는지 검증한다. 언패킹 검증에 사용된 파일은 상용 프로그램들이며 UPX로 패킹된 파일과 패킹이 되어있지 않은 파일을 UPX로 패킹을 수행하여 검증에 사용하였다. 다음 [그림 34]는 언패킹 전과 후의 패킹 데이터이다.

File_Name	Packing_Detect	EPS_Entropy	EPS_Characteristics		File_Name	Packing_Detect	EPS_Entropy	EPS_Characteristics
u_ALFTPServer10.exe	0	6.48095	0x60000020		ALFTPServer10.exe	1	7.83656	0xe0000040
u_exeinfope.exe	0	6.38688	0x60000020		exeinfope.exe	1	7.86259	0xe0000040
u_FileZilla_Server-0_9_60_2.exe	0	6.45023	0x60000020		FileZilla_Server-0_9_60_2.exe	1	7.82534	0xe0000040
u_ftp.exe	0	6.23921	0x60000020		ftp.exe	1	7.87845	0xe0000040
u_LinuxLive_USB_Creator_2.9.4.ex	0	6.4331	0x60000020		LinuxLive USB Creator 2.9.4.exe	1	7.82979	0xe0000040
u_npp.7.4.1.Installer.exe	0	6.47707	0x60000020		npp.7.4.1.Installer.exe	1	7.87493	0xe0000040
u_Nspack_Downloader.exe	0	6.60064	0x60000020		Nspack_Downloader.exe	1	7.88512	0xe0000040
u_PIL-1.1.7.win32-py2.7.exe	0	6.60182	0x60000020		PIL-1.1.7.win32-py2.7.exe	1	7.86618	0xe0000040
u_pywin32-221.win32-py2.7.exe	0	6.60182	0x60000020		pywin32-221.win32-py2.7.exe	1	7.91627	0xe0000040
u_ResTuner_setup.exe	0	5.7322	0x60000020		ResTuner_setup.exe	1	7.9295	0xe0000040
u_Sublime_Text_Build_3126_Setu	0	5.73207	0x60000020		Sublime Text Build 3126 Setup.exe	1	7.91627	0xe0000040
u_Universal-USB-Installer-1.9.7.9	0	6.4331	0x60000020		Universal-USB-Installer-1.9.7.9.exe	1	7.89398	0xe0000040
u_ImmunityDebugger_1_85_setu	0	6.06456	0x60300020		ImmunityDebugger_1_85_setup.exe	1	7.86993	0xe0000040
u_NEW_GOMPLAYERSETUP.EXE	0	6.45023	0x60000020		NEW_GOMPLAYERSETUP.EXE	1	7.84105	0xe0000040
u_python-3.6.2.exe	0	6.56223	0x60000020		python-3.6.2.exe	1	7.85865	0xe0000040
u_Wireshark-win32-2.2.6.exe	0	6.42419	0x60000020		Wireshark-win32-2.2.6.exe	1	7.99961	0xe0000040

[그림 34] 언패킹 전 후 패킹데이터 비교

위의 결과에서 왼쪽이 언패킹 수행 후의 패킹 데이터고 오른쪽이 언패킹 수행 전의 데이터이다. 언패킹 후 패킹 여부는 모두 탐지되지 않았으며, 각 여부 탐지 기준인 진입점 섹션의 엔트로피와 속성 값 또한 모두 패킹이 되어있지 않는 것으로 판명되었다. 따라서, UPX의 경우 자동 언패킹 기능이 정상적으로 동작한다고 볼 수 있다.

본 시스템은 악성코드 분석을 위해 PE 정보 추출, 패킹 여부 탐지, 패커 종류 탐지 그리고 자동 언패킹 시스템이다. 따라서, 악성코드 50개에 대해서 실험 하여 패킹 데이터를 추출하였다. 다음 [그림 35]는 악성코드 패킹 데이터의 일부분이다.

File_Name	Packing_Detect	EPS_Entropy	EPS_Characteristics	Packer_Name
00A1E3185	0	6.83776	0x60000020	Morphine v1.2 (DLL),I
00A450E04	1	7.95722	0x60000060	
00A6774B0	1	6.19326	0xe0000020	Microsoft Visual C++
00A78C579	1	4.01465	0xe0000020	Microsoft Visual C 2.
0A06AD05	0	6.67872	0x60000020	Microsoft Visual C++
0A0AF1867	1	7.97245	0xe0000040	UPX 3.02,UPX 2.93 (L
0A0C5D7F	1	7.85788	0xe0000060	
0A170B749	1	7.72646	0xc0040020	
0A192DA7	0	6.71317	0x60000020	Armadillo v1.71,Micro
0A1A2AC6	0	5.8422	0x60000020	Microsoft Visual C++
0A1B7A060	0	6.58914	0x60000020	Microsoft Visual C++
0A225E164	0	6.67191	0x60000020	Microsoft Visual C++
0A23BEC3C	0	6.5585	0x60000020	Microsoft Visual C++
0A29A2835	0	6.68471	0x60000020	Microsoft Visual C++
0A2AB5D1	1	6.23076	0xe0000020	Microsoft Visual C++
0A30BDB6	1	7.83676	0xc0040020	
0A3BE6C53	1	5.82142	0xe00000e0	MPRESS V2.00-V2.0X

[그림 35] 악성코드 패킹 데이터

악성코드는 50개중에 27개가 패킹 되어 있다. 이는, 일반 파일에 비해 패킹 되어있는 비율이 높다. 또한, 앞선 검증과 마찬가지로 “Armadillo”에 대해서는 검증이 되지 않았으며 검증에 사용한 5가지 패커가 아닌 “MPRESS”, “Ste@lth” 패커에 대해서 탐지 하였다.

5. 결론 및 향후 연구

본 논문에서는 악성코드 분석을 위한 PE 파일 언패킹을 자동으로 수행해주는 시스템을 설계 및 구현하였다. 일반적으로 사용되는 PE 분석 툴에서는 패킹 여부 탐지에 대한 문제점은 시그니처 기반으로만 패킹 여부를 탐지하기 때문에 시그니처가 없는 패커를 사용하여 패킹하였을 경우나 userdb.txt가 업데이트 되어있지 않을 경우 패킹 여부를 탐지하지 못하였다. 본 시스템에서는 첫 번째로, Custom 패커나, 시그니처가 아직 밝혀지지 않은 패커의 경우까지 패킹 여부를 탐지하기 위해 진입점 섹션의 엔트로피 값과 속성 값을 비교하는 이론을 적용하여 패킹 여부를 탐지하였다. 두 번째로, 시그니처 기반으로 패킹 여부를 탐지않고 패커 종류를 탐지하는 것이 더 나은 방법이라고 생각하여 패커 종류를 탐지하였고 시그니처가 담겨있는 userdb.txt 또한 최근 업데이트된 자료를 반영하여 기존 PE 분석 툴에서는 탐지하지 못하였던 패커 종류를 탐지 하였다. 세 번째로, 패킹 여부 탐지로 패킹이 되어있고 패커 종류를 탐지하여 언패킹이 가능한 패커의 경우 언패킹 툴과 연동하여 언패킹을 자동으로 수행해주는 시스템을 구현하였다. 검증 결과, PE 분석의 경우 파일 크기에서 근소한 차이를 보이며 Upack과 Nspack에서의 파일 오프셋이 분석 툴과 다르게 출력되었다. 파일 크기와 파일 오프셋을 제외한 모든 정보는 다른 PE 분석 툴과 동일한 정보를 추출하였으며, 진입점 섹션의 엔트로피 값과 진입점 섹션의 속성 값등 더욱더 세부 정보를 추출할 수 있다. 패킹 여부 탐지의 경우 UPX, Aspack, Nspack, Upack, Yoda's Protector의 패커들은 100% 탐지율을 도출 하였으며, 위의 패커들이 아닌 다른 패커로 패킹된 파일의 패킹 여부도 탐지해 내었다. 패커 종류 탐지는 시그니쳐에 존재하는 패커에 대해서는 잘 탐지하였지만 Nspack으로 패킹한 파일의 경우 userdb.txt에 Nspack의 시그니처가 존재하지만 버전 때문에 탐지되지 않는 것으로 생각된다. 잘 알려진 패커에 대해서 언패킹을 자동으로 수행해주는 기능은 현재 UPX의 경우에 대해서 100% 언패킹을 수행한 결과를 도출하였으며, 언패킹된 파일들에 대해서 본 시스템의 패킹 여부 탐지 기능으로 검증한 결과 모두 패킹 탐지가 되지 않았다. 또한, 본 시스템은 악성코드 분석을 위한 시스템이기 때문에 악성코드 50개에 대해서 실험을 하였고, 50개중 27개가 패킹이 되어있는

것으로 판단하였다. 향후 연구로는 현재 잘 알려진 패커로 사용한 5개의 패커 이외의 다른 패커들을 사용하여 검증하고, UPX가 아닌 Aspack, Nspack, Upack, Yoda's Protector 패커 등에 대해서도 언패킹 기능을 본 시스템과 연동 시킬 계획이다. 또한, 본 실험에서 사용하지 않은 패커들에 대해서도 언패킹 방법을 연구하여 본 시스템과 연동하며, Custom 패커를 언패킹 하기 위해 엔트로피 값 변화 기반 언패킹에 대해서 연구할 계획이다.

참고문헌

[1] 미래 창조 과학부, “2016 인터넷이용실태조사 최종보고서”, 2017

[2] 미래 창조 과학부, “악성코드 은닉사이트 탐지 동향 보고서” 17년 상반기

[3] 임채태, 오주형, and 정현철. "최신 악성코드 기술동향 및 분석 방안 연구." 정보과학회지 28.11 (2010): 117-126.

[4] Shadowserver, https://www.shadowserver.org/

[5] KASPERSKY LAB, http://www.kaspersky.co.kr/

[6] INCA Interset, http://erteam.nprotect.com/

[7] AV-TEST, https://www.av-test.org/en/

[8] 김준형, et al, “악성코드 탐지를 위한 바이너리 실행 파일 분석 기법.” 정보과학지 35.2 (2017)“ 48-54.

[9] 이경식, 최화재, and 박정찬. "악성코드 동적 분석 기법 우회 방법 및 대응 방안 연구." 한국정보과학회 학술발표논문집 (2017): 1069-1071.

[10] 백영태, 김기태, and 전상표. "이진 코드의 정적 실행 흐름 추적을 위한 프레임워크 설계 및 구현." 한국컴퓨터정보학회논문지 16.6 (2011): 51-59.

[11] 윤광택, and 이경호. "동적 악성코드 분석 시스템 효율성 향상을 위한 사전 필터링 요소 연구." 정보보호학회논문지 27.3 (2017): 563-577.

[12] 박재우, et al. "문자열과 API 를 이용한 악성코드 자동 분류 시스템." 보안공학연구논문지 (Journal of Security Engineering) 8.5 (2011).

[13] 최양서, et al. "악성프로그램 탐지를 위한 PE 헤더 특성 분석 기술." 융합보안논문지 8.2 (2008): 63-70.

[14] Ollydbg, http://www.ollydbg.de/

[15] PE Format, https://ko.wikipedia.org/

[16] 이호동(지은이), (2016), “윈도우 실행 파일 구조와 원리로 배우는 리버스 엔지니어링 1권”, 서울: 한빛미디어

[17] Exeinfope, http://exeinfo.atwebpages.com/

[18] UPX, https://upx.github.io/

[19] Aspack, http://www.aspack.com/

[20] Lyda, Robert, and James Hamrock. "Using entropy analysis to find encrypted and packed malware." IEEE Security & Privacy 5.2 (2007).

[21] 한승원, and 이상진. "악성코드 포렌식을 위한 패킹 파일탐지에 관한 연구." 정보처리학회 논문지 16.5 (2009): 555-562

[22] 이영훈, et al. "엔트로피 값 변화 분석을 이용한 실행 압축 해제 방법 연구." 정보보호학회논문지 22.2 (2012): 179-188.

[23] 정구현, et al. "엔트로피를 이용한 실행 압축 해제 기법 연구." 한국정보기술학회논문지 7.1 (2009): 232-238.

Design and Implementation of PE File Unpacking Automatic System for Malware Analysis

Sun-Kyun Kim

Department of Computer Science
Graduate School, Kangwon Nation University

Abstract

Recently, computer and internet technologies are evolving one day differently, and these developments have had a tremendous positive impact on our lives. This development has also had a positive impact on our lives and a negative impact. The damage caused by malicious code has been steadily increasing and the damage of users has also increased. However, antivirus programs have also been developed to prevent malicious behavior of many malicious codes, but malicious codes that have advanced further prevent malicious code from spreading through packings and prevent quick response. In order to cope with cyber threats arising from such malicious code, it is necessary to analyze the malicious code file. Various studies are under way to cope with cyber threats caused by malicious code. The malicious code analysis method used to cope with cyber threats consists of initial analysis, dynamic-based analysis, and statistics-based analysis. In this paper, we investigate whether packing is based on information of malicious code and PE file through initial analysis, and if it is packed with a Well-Known packer by detecting signatures based on signatures, we use the packer's unpacking tool We propose a system that performs unpacking. The proposed system consists of

three stages. First, it extracts essential information that is important for detecting basic information and packing of files through PE structure analysis of PE file. Next, the presence or absence of packing is detected by using the entry point section entropy value and the entry point section attribute value, which are important information for detecting whether or not the packing exists. This is typically a feature of packed files. Finally, if the detected type of packer is a Well-Known packer, the unpacking method of the corresponding packer is used to automatically unpack the pack, Restore original files that have not been restored. Experimental results show that the PE information is extracted from the PE structure and the packing of the packed files by the Well-Known packer is detected. In case of PE files packed with UPX among the Well-Knwon packers, unpacking is performed successfully, . Malicious code Unstuck malware is required for static analysis. Therefore, this system is a system for static analysis in malicious code analysis. It can perform unpacking only for PE files that can perform unstacking without static analysis and dynamic analysis using initial analysis and signature-based packer type detection It has a big meaning in point. In the future, it can be utilized as a system to prevent malicious code from analyzing the malicious code.

□ keywords

Malware, Packing, Unpacking, Packing Detection, Packer Class Detection, PE Analysis, Entropy