KIPS Tr. Comp. and Comm. Sys.
Vol.7, No.6 pp.155~164 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2018.7.6.155

Improved Original Entry Point Detection Method Based on PinDemonium

Kim Gyeong Min[†] · Park Yong Su[††]

ABSTRACT

Many malicious programs have been compressed or encrypted using various commercial packers to prevent reverse engineering, So malicious code analysts must decompress or decrypt them first. The OEP (Original Entry Point) is the address of the first instruction executed after returning the encrypted or compressed executable file back to the original binary state. Several unpackers, including PinDemonium, execute the packed file and keep tracks of the addresses until the OEP appears and find the OEP among the addresses. However, instead of finding exact one OEP, unpackers provide a relatively large set of OEP candidates and sometimes OEP is missing among candidates. In other words, existing unpackers have difficulty in finding the correct OEP. We have developed new tool which provides fewer OEP candidate sets by adding two methods based on the property of the OEP. In this paper, we propose two methods to provide fewer OEP candidate sets by using the property that the function call sequence and parameters are same between packed program and original program. First way is based on a function call. Programs written in the C/C++ language are compiled to translate languages into binary code. Compiler-specific system functions are added to the compiled program. After examining these functions, we have added a method that we suggest to PinDemonium to detect the unpacking work by matching the patterns of system functions that are called in packed programs and unpacked programs. Second way is based on parameters. The parameters include not only the user-entered inputs, but also the system inputs. We have added a method that we suggest to PinDemonium to find the OEP using the system parameters of a particular function in stack memory. OEP detection experiments were performed on sample programs packed by 16 commercial packers. We can reduce the OEP candidate by more than 40% on average compared to PinDemonium except 2 commercial packers which are can not be executed due to the anti-debugging technique.

Keywords : Computer Security, Malicious Code Analysis, Unpacking, OEP, Reverse Engineering, Dynamic Analysis, Code Obfuscation

PinDemonium 기반 Original Entry Point 탐지 방법 개선

김 경 민[†] · 박 용 수[††]

요 약

많은 악성프로그램은 역공학을 막기 위해 다양한 상용 패커를 사용해 압축 혹은 암호화를 했기 때문에 악성코드 분석가는 압축해제 혹은 복 호화를 먼저 수행해야 한다. OEP(Original Entry Point)는 암호화되거나 압축되어 있는 실행파일을 다시 원본 바이너리 상태로 되돌린 후 실행된 첫 번째 명령어의 주소이다. 여러 언패커는 OEP가 나타나기 전까지 패킹된 파일을 실행하며 주소를 기록한다. 그리고 기록된 주소들 중에 OEP를 찾는다. 그러나 일부 언패커에서 제공하는 OEP 후보들은 비교적 큰 OEP 후보 집합을 제공하거나 후보들 중에 OEP가 없는 경우가 있다. 이에 악성코드 분석가들은 더 적은 OEP 후보 집합을 제공하는 도구가 필요한 실정이다. 본 논문에서는 PinDemonium이라 불리는 언패커에 두 가지 OEP 탐지방법을 추가하여 더 적은 OEP 후보 집합을 제공하는 도구를 만들었다. 첫 번째 방법은 패킹된 프로그램이 완전히 원본 바이너리상태로 되돌아 간 후에는 원프로그램 함수 호출과 동일하다는 것을 활용한 OEP 탐지방법이다. C/C++ 언어로 작성된 프로그램은 바이너리 코드로 언어를 변환하는 컴파일 과정을 거친다. 컴파일 과정을 거친 프로그램에는 특정 시스템 함수들이 추가된다. 이 시스템 함수들은 컴파일러 별로 다르다. 컴파일러 별로 사용되는 시스템 함수를 조사한 후, 패킹된 프로그램에서 호출되는 시스템 함수와 패턴매칭하여 언패킹 작업이 끝났는지 탐지하는 방법이다. 두 번째 방법은 패킹된 프로그램이 완전히 원본 바이너리 상태로 돌아간 후 시스템함수에서 사용되는 매개변수가 원프로그램과 동일하다는 것을 활용한 OEP 탐지방법이다. 시스템함수에서 사용되는 매개변수의 값을 이용해 OEP를 찾는 방법이다. 본 연구는 16종의 상용 패커로 압축된 샘플 프로그램을 대상으로 OEP 탐지 실험을 했다. 본 연구에선 안티 디버깅 기법으로 프로그램을 실행하지 못하는 경우인 2종을 제외하고 PinDemonium 대비 평균 40% 이상 OEP후보를 줄일 수 있었다.

키워드 : 컴퓨터보안, 악성코드 분석, 언패킹, OEP, 역공학, 동적분석, 코드난독화

※ 이 논문은 2017년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No.2017R1D1A1B03029550).
† 준 회 원 : 한양대학교 컴퓨터소프트웨어학부 석사과정
†† 비 회 원 : 한양대학교 컴퓨터소프트웨어학부 교수
Manuscript Received : March 12, 2018
Accepted : May 3, 2018
* Corresponding Author : Kim Gyeong Min(casualab@hanyang.ac.kr)

1. 서 론

많은 악성코드는 분석을 어렵게 하기 위하여 프로그램 실행에 필요한 정보들을 압축 및 난독화 한다. 안티 바이러스 및 정보보호 관련 연구 기관인 AV-TEST사에 따르면 전체 악성코드 중 92% 이상이 압축 및 난독화가 적용되어 있다고 한다[1]. 패커는 실행 파일을 압축 및 암호화하고 복구하는 소프트웨어 프로그램이다. 패커가 압축 및 암호화 라이브러리를 사용하여 실행에 필요한 정보들을 압축, 난독화 하는 것을 패킹이라 한다[2]. 패킹된 프로그램을 실행시 언패킹 루틴이 실행되어 압축, 난독화 된 코드를 실행이 가능하도록 복구한 후 원프로그램을 실행한다. 그래서 분석가들은 패킹이 적용된 실행 파일을 자세히 분석하기 위해 통상 패킹 상태를 해제하는 과정을 수행한 후 분석을 진행한다[3]. 패킹 상태를 해제하기 위해서는 엔트리 포인트(entry point) 중 OEP (Original Entry Point)를 찾아야 한다.

엔트리 포인트는 프로그램의 수행 시작 시 운영체제에서 응용프로그램으로 제어가 전달된 지점이다[4]. 통상 엔트리 포인트는 프로그램의 시작지점으로 이해할 수 있다. 엔트리 포인트는 DLL 등 여러 프로그램에 존재한다.

패킹된 프로그램에서 패커가 이용하는 메모리 영역은 크게 두 가지주요 부분으로 구성되어 있다[5]. 첫 번째 부분에는 원본 실행 파일이 압축되거나 암호화된 형태로 저장되어 있다. 둘째 부분에는 압축 해제 모듈이 있다. 압축 해제모듈은 원래 실행 파일을 복원하는데 사용된다.

언패킹 과정은 패킹 과정을 반대로 수행한 것이다. 압축해제가 먼저 수행되고 실행 흐름이 압축 해제된 코드의 텍스트 메모리 영역(사용자가 C/C++ 등 언어로 작성한 코드가 기계어로 바뀌어 저장되어있는 메모리 영역) 첫 번째 명령으로 이동한다(주소 A). 이 첫 번째 명령어는 압축되기 이전 원바이너리코드의 엔트리 포인트에 있는 명령어와 동일한 명령어이다. 이 때, 우리는 주소 A를 OEP라 정의한다.

즉, 패킹이 적용된 실행 파일의 원본 코드를 분석하기 위해서는 패킹이 해제된 원본 코드의 시작지점인 OEP를 찾아야한다.

하지만 OEP를 찾는 일은 쉽지 않다. 왜냐하면 패킹된 프로그램은 디버거의 동작을 탐지하며 프로그램 실행을 중단하거나 정상적인 동작이 아닌 행동을 수행하여 분석을 방해하기 때문이다. 또한, 프로그램의 실행 흐름을 복잡하게 하여 분석까지 많은 시간을 소요하도록 만들기도 하며 프로그램 실행 시, 매번 코드가 변하는 다형성 성질이 적용되어 있다.

OEP를 찾는데 도움을 주는 도구로 언패커가 있다. 언패커의 목적 중 하나는 악성코드 페이로드를 찾는 것[3]에 있다. 많은 언패커들(Universal PE Unpacker[6], PolyUnpack[7] 등)은 OEP를 찾은 후 페이로드를 찾도록 설계되어 있다.

언패커 중 하나인 Uinversal PE Unpacker[6]의 경우에는 자동 언패커가 아니다. 그래서 악성코드 분석가는 사전에 OEP로 추정되는 메모리 주소의 범위를 악성코드 분석가가 알고 있어야 한다. 언패커 중 하나인 PolyUnpack은 바이너리 명령어를 하나하나 실행하며 사전에 분석된 단계와 동일한 부분이 나오는지 여부를 확인한다. 이 프로그램의 제작자는 패킹된 프로그램을 한 줄 실행시키고 해석(Disassemble)하기에 연산비용(computational complexity)이 매우 크다고 언급하고 있다[7]. 최근에 공개된 언패커로 PinDemonium[8]이 있다. PinDemonium은 인텔사의 Pin[9] 엔진으로 구현되어 있고 최대한 많은 OEP 후보를 찾아 Scylla[10]로 메모리 덤프를 수행하고 프로그램이 사용하는 라이브러리의 함수 테이블도 재구성해준다. 하지만 PinDemonium이 제공하는 OEP 후보의 수는 적지 않다. 또한 OEP를 확인하는 작업은 전적으로 악성코드 분석가의 역량에 달려있다. 언패커는 아니지만 언패킹에 이용되는 도구인 PEiD[11]의 경우는 데이터베이스를 사용해서 알려진 암호기법과 시스템함수를 저장해두고 패턴매칭으로 패커타입이나 압축형태를 알려준다. 하지만 PEiD는 패킹된 파일의 OEP를 찾지 못한다.

이에 본 논문에서는 x86 Windows 환경을 가정하고, 동적 바이너리 분석을 방해하는 기술들이 적용된 패킹된 실행 파일을 대상으로 OEP를 찾는 일반적인 방법을 제시한다.

본 논문에서는 기존에 개발된 언패커(PinDemonium[8]) 보다 더 적은 수의 OEP 후보를 제공하는 도구를 만들기 위해 Isawa et al.'s work[5]의 방법을 사용하고 있다. Isawa et al.'s work[5]의 방법은 WrittenAndExecute 주소들 중 OEP가 유력한 후보를 찾는 방법이다. WrittenAndExecute주소라 함은 패커가 원바이너리 코드를 복호화 하며 메모리 쓰기를 하고, 이 메모리 쓰기 주소를 실행한 주소다.

여기서 더 나아가 기존에 없는 새로운 두 가지 OEP 탐지방법을 언패커(PinDemonium[8])에 추가했다. 첫 번째 방법은 메인함수(int main(int argc, char *argv)) 호출 전에 불리는 시스템함수의 매개변수를 스택 메모리에서 찾아 OEP를 찾는 방법이다. 두 번째 방법은 컴파일러별 시스템 함수의 호출 순서를 활용해 언패킹 작업이 끝났는지 탐지하여 OEP를 찾는 방법이다. 이 방법은 언패킹 작업이 끝났다고 판단되면 WrittenAndExecute 주소를 활용해 OEP를 찾는 것이다. 즉 본 논문에서는 기존에 없는 두 가지 OEP 탐지방법을 PinDemonium에 추가하여 더 적은 수의 OEP 후보를 출력하는 OEP 탐지 방법을 만들었다.

새로운 두 가지 방법은 기존에 연구된 OEP 탐지 방법과는 다른 차별점이 있다. 기존의 연구들은 패킹된 코드의 패커 코드의 특징을 찾아 OEP를 찾는 방법이라면, 본 논문의 방법은 패킹된 프로그램의 원바이너리코드 특징을 찾아 OEP를 찾는 방법이다.

본 논문의 제안 방법을 테스트하기 위해 다양한 상용 패커로 패킹된 파일을 대상으로 OEP를 탐지하는 실험을 진행 하였다. UPX, PECompack, ASPack, VMProtect2.1, WWPack, Packman, petite, MEW, Enigma protector4.03 등의 패커에 있어서 PinDemonium 대비 평균 40% 이상 OEP 후보를 줄여줬다. 특히 ASProtect와 Themida에서는 OEP 후보를 90% 이상 줄이는 결과가 나왔다.

패킹된 악성코드는 악성코드 분석가로 하여금 악성코드를 분석하기 어렵도록 하는데 사용된다. 패킹된 악성코드를 해

제하기 위해 OEP를 찾기 위한 시간이 걸리는 동안 악성코드 전파 시간을 늘릴 수 있기 때문이다. 이와 반대로 만약 실행 압축을 빨리 해제하여 분석할 수 있다면, 그 피해규모 또한 현저히 줄일 수 있을 것이다. 본 연구는 악성코드 분석가가 패킹된 악성코드를 분석하여 OEP를 찾는 번거로운 시간을 줄이는데 도움이 된다. 이에 악성코드 분석가는 패킹된 악성코드에 더 민첩하게 대응할 수 있을 것이다.

2. 관련 연구

2.1 Isawa et al.'s work[5]

Isawa et al.'s work[5]에서는 OEP까지 언패킹 작업에 사용되는 코드를 파악하기 위한 동적 접근 방식을 제안하고 있다. 이 동적 접근 방식은 프로그램 실행 및 메모리 쓰기를 모니터링하고 실행중인 코드가 런타임에 생성되었는지 확인하는 방식이다. 이 방식은 크게 두 가지 부분 '패커의 복호화 루틴추적', 'OEP 후보 정렬'로 나뉜다.

'패커의 복호화 루틴추적'을 수행하기 위해서는 몇 가지 사전 작업들이 요구된다. 먼저, 특정 메모리 페이지(page)에 대한 용어를 정의해야 한다. 생성페이지(이하 generating page)는 나중에 실행될 페이지에 데이터를 쓰는 페이지를 칭한다. 그리고 공유페이지(이하 sharing page)는 generating page 혹은 sharing page에 데이터를 쓰는 페이지를 칭한다.

다음은 OEP 후보를 나열할 배열을 만들어야 한다. 먼저, 패킹된 프로그램을 메모리에 로드 후, 로드 된 프로그램을 페이지 단위(4KB)로 나눈다. 그 다음, 나눠진 페이지를 전부 R/X로 마킹한다. R/X는 read-only/executable, 즉 해당 페이지의 데이터를 읽고 실행하는 것이 가능하다는 것을 의미한다. 패킹된 프로그램이 수행되는 동안 R/X 페이지에 쓰기 명령어로 데이터를 쓰는 경우 writing page-fault 예외(exception)가 발생한다. 이 때, 예외가 발생한 페이지를 W/NX로 마킹한다. W/NX는 writable/not-executable, 즉 해당페이지는 데이터를 쓰는 것이 가능하나 실행은 불가능하다는 것을 의미한다. 이후에 W/NX 페이지에 있는 명령어가 실행될 경우 executing page-fault 예외가 발생한다. 이러한 마킹 과정을 패킹된 프로그램이 수행을 멈출 때까지 진행하며, 특정 예외가 발생할 때 마다 src, dst 주소 쌍을 원소로 가지는 배열 W와 하나의 주소를 원소로 가지는 배열 X를 만든다. 어떤 주소를 배열 원소로 가지는지는 Table 1에 묘사되어 있다.

'패커의 복호화 루틴 추적'은 다음과 같이 수행된다. W 배열에서 각 원소 쌍을 시간 순으로 나열한 뒤, dst 주소에 있는 명령어가 X배열 원소 주소의 명령어 수행 시간과 비교하였을 때 나중에 수행되었는지를 확인한다. 만약 나중에 수행되었을 경우 dst 주소와 쌍을 이루는 src 주소가 있는 페이지는 generating page이며, 해당 페이지와 src 주소를 언패킹 루틴의 일부라 판단하고 U로 마킹한다.

그 다음, 다시 W 배열에서 각 원소 쌍을 시간 순으로 나열한다. 만약 dst 주소가 U로 마킹되어 있을 경우 그 쌍을 이루는 src 주소가 있는 페이지는 sharing page이며, 해당 페이지와 src 주소를 U로 마킹한다. X 배열원소 주소들 중 U로 마킹된 페이지에 있는 주소가 있을 경우 해당 주소 역시 U로 마킹한다. 패킹된 프로그램의 수행 종료 후, X 배열의 모든 원소를 시간 순으로 나열한다. 나열한 주소들 중, U로 마킹된 마지막 주소에 가장 근접한 X 배열의 원소 주소를 최적의 OEP 후보로 선택한다.

다음은 'OEP 후보 정렬'이다. 'OEP 후보 정렬'은 '패커의복호화 루틴 추적'을 통해 선택된 최적의 OEP 후보를 기준으로 OEP 후보들을 정렬한다. 정렬식은 다음과 같다.

$$j = \begin{cases} 0 & (i = p) \\ 2|p-i| - (1 + sign(p-i))/2 & (otherwise) \end{cases} \quad (1)$$

후보 정렬을 위해 X 배열을 (x_0, x_1, …, x_i, …, x_p, …, x_{t-1})와 같이 시간 순으로 나열한다. 이 때, p 는 '패커의 복호화 루틴 추적'을 통해 선택된 최적의 OEP 후보이다. t-1 은 마지막으로 실행된 후보의 인덱스이며, t는 수행된 명령어의 개수이다. i는 0부터 t-1까지의 값을 갖는다. sign(・)은 $p < i$일 경우 입력값이 음수가 되므로 -1을 출력하고, 그렇지 않다면 1을 출력한다. 위계산식을 통해 X를 정렬할 경우 X는 (x_p, x_{p-1}, x_{p+1}, x_{p-2}, x_{p+2}, …)와 같이 정렬된다.

2.2 PinDemonium[8]

DBI(Dynamic Binary Instrumentation)는 런타임에 실행 코드를 삽입하여 바이너리 응용 프로그램의 동작을 분석하는 방법이다. PinDemonium[8]은 Intel의 DBI도구 Pin[9]을 엔진으로 사용하는 언패커다. PinDemonium은 WrittenAndExecute가 일어날 때 Scylla[10]라는 메모리 덤프도구를 이용해서 타겟 프로그램을 덤프한다. PinDemonium은 덤프된 프로그램에 휴리스틱 방법을 적용하여 언팩된 때와 OEP를 찾는다. PinDemonium에는 휴리스틱 방법들(엔트로피 분석, Long jump, Jump outer section, yara rule 등)이 적용되어 있다.

Table 1. The Elements of Array W and Array X

Classification	Description	
Array W	src	The address at which the writing page-fault exception has performed the write command to the R / X page that occurred.
	dst	The address at which the write command was performed on the R / X page where the writing page-fault exception occurred.
Array X	The address at which the command was performed on the W / NX page where executing page-fault exception occurred.	

엔트로피 분석 방법은 메모리 공간의 엔트로피를 측정하여 OEP를 찾는 방법이다. 프로그램을 메모리에 올리고 1 바이트 씩 읽어 메모리 안의 바이트 값 빈번도를 계산한다. 즉, 정해진 메모리 공간에 00h ~ FFh까지 바이트 값의 빈번도로 엔트로피를 계산한다. PinDemonium[8]은 오픈소스로 공개되어있고 구현된 코드를 보면 WrittenAndExecute 때 측정한 엔트로피 값 대비 패킹된 프로그램을 메모리 로드하며 최초로 측정한 엔트로피 값의 차가 임계값인 0.2보다 크면 최근에 수행한 WrittenAndExecute 주소가 OEP 후보라고 텍스트 파일로 알려준다.

Long jump와 Jump outer section의 경우에는 런타임 중 EIP 레지스터 값(이하 EIP)을 활용한 방법이다. PinDemonium[8]이 구현된 코드를 보면, Long jump의 경우 이전에 수행한 EIP 대비 현재 EIP가 임계값(0x200) 이상 이면 현재 주소가 OEP 후보라 탐지하는 방법이다. Jump outer section의 경우는 이전에 수행한 EIP의 메모리 섹션이 현재 EIP의 메모리 섹션 명이 바뀌었을 경우 EIP가 OEP 후보라고 탐지하는 방법이다.

yara rule은 yara[12]에서 악성코드 분류에 사용되는 규칙이다. yara는 악성코드를 분류하는 도구지만 암호화된 경우 작동하지 않는다. WrittenAndExecute 때 수행된 덤프파일에 yara rule을 패턴매칭해서 OEP 후보를 탐지하는 방법이다.

3. OEP가 가지는 특성과 제안기법

우리는 다양한 종류의 상용 패커들(UPX, PECompack 등)로 패킹한 테스트 프로그램을 이용하여 연구, 관찰했다. 이러한 노력을 통해 OEP가 다음과 같은 특성을 지님을 확인하였다.

3.1 OEP가 가지는 특성

본 논문에서 다루는 OEP의 특성은 4가지가 있다. 본 연구에서는 패커 16종의 특징을 관찰하여 '시스템함수 호출 순서(Sequence of system function call after OEP)', 'OEP 이후 패킹된 프로그램의 매개변수(Parameters of packed programs after OEP)'라는 특성을 찾았다. '프로그램 실행흐름이 바뀐 목적지 주소 중 OEP가 존재(OEP exists at program execution flow change)'와 'WrittenAndExecute 페이지 중 OEP가 존재(OEP in WrittenAndExecute pages)'는 기존에 연구된[5, 8] OEP의 특성이다.

1) Sequence of system function call after OEP

C/C++ 언어로 작성된 프로그램은 바이너리 코드로 언어를 변환하는 컴파일 과정을 거친다. 컴파일 과정을 거친 프로그램에는 특정 시스템함수들이 추가된다. 이 시스템 함수들은 컴파일러 별로 다르게 추가 된다. 이 시스템 함수들의 호출순서를 패킹된 프로그램과 원프로그램을 대상으로 관찰해본 결과, OEP 이후 패킹된 프로그램의 시스템함수 호출순서는 원프로그램과 동일하다는 특성을 지니고 있었다.

일반적인 원프로그램의 메인함수라 함은 int main(int argc, char *argv)이다. 원프로그램의 메인함수까지 호출되는 시스템 함수/명령어 호출 순서는 SE Handler를 설정하는 명령어를 수행 후에는 커널 버전, 프로세스나 쓰레드 ID, 시스템시간관련 함수 등이 불린다. 다음에는 포인터를 초기화 하거나 동적메모리를 할당하거나 초기화 하고 화면좌표나 어플리케이션 타입(GUI/CLI)관련 함수들이 불린다. 메인함수가 불리기 직전에는 문자열, 파일명, 매개변수 복사, 환경변수 설정 등의 다양한 함수가 불린다. 이 함수/명령어들이 실행흐름의 바뀜 없이 순차적으로 불렸다면 OEP를 지나왔다는 의미가 된다. OEP를 지났을 때 가장 최근의 WrittenAndExecute 주소가 OEP 후보다.

일례로 Fig. 1은 tdm gcc 5.1.0로 컴파일 된 프로그램이며 메인함수를 실행하기 전 _initterm, __set_app_type, __lconv_init 혹은 __gcov_init 함수를 순서대로 호출한다. _initterm은 함수 포인터 테이블을 초기화를 수행하는 함수다. __set_app_type은 어플리케이션의 유저 인터페이스 형식(CLI/GUI)을 설정한다. __lconv_init, __gcov_init은 어플리케이션의 인터페이스(CLI/GUI)에 따라 다르게 불리며 라이브러리 링킹(Runtime Library Linking)을 위한 함수다.

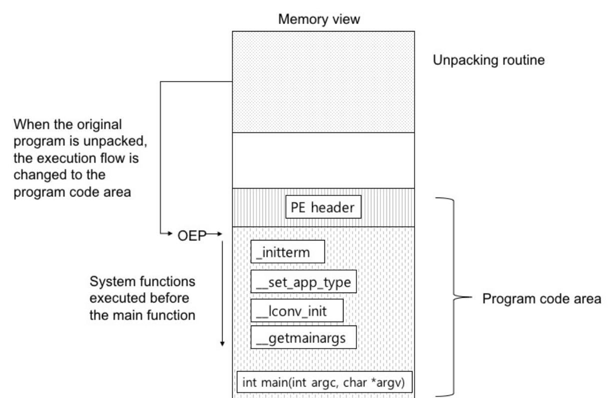

Fig. 1. System Functions after OEP of Program Compiled with TDM GCC 5.1.0 and packed with UPX

2) Parameters of packed programs after OEP

매개변수는 변수의 한 종류로 함수 등과 같은 서브루틴의 입력 값으로 제공되는 여러 데이터중 하나를 가리키기 위해 사용된다. 대표적으로 메인함수인 int main(int argc, char *argv)의 서브루틴 __getmainargs에서 사용되는 매개변수가 있다. 패킹된 프로그램은 __getmainargs를 사용해서 int main(int argc, char *argv)의 매개변수를 받았을 때 OEP 이후가 된다.

__getmainargs는 매개변수들의 일부를 메인함수 int main(int argc, char *argv)에서 받아온다. 시스템 함수인 int __getmainargs(int * _Argc,char *** _Argv, char *** _Env, Int _DoWildCard, _startupinfo * _StartInfo)는 5개의 매개변수를 지닌다[13]. int main(int argc, char*argv)는 사용자로부터 받은 입력값을 복사하기 위해 int argc, char *argv를 __getmainargs으로 전달한다. __getmainargs는 int argc, char

*argv를 매개변수로 사용한다. 그리고 매개변수 명은 int * _Argc, char ***_Argv가 된다. 이 두 매개변수의 기능은 다음과 같다. int * _Argc는 사용자로부터 입력받은 값의 수가 된다. 사용자가 전달한 입력값이 없어도 시스템 기본값이 있어 int * _Argc는 최소 1이 된다. char ***_Argv는 사용자로부터 입력받은 값의 배열형태다. char ***_Argv는 배열로 첫 번째에 실행할 프로그램 호출경로가 시스템 기본값으로 들어가 있다.

패킹된 프로그램이 완전히 언팩이 되고 OEP 이후 사용된 함수의 매개변수가 원프로그램과 동일하다는 예시로 PEcompact로 패킹된 프로그램을 들 수 있다. Fig. 2는 PEcompact로 패킹된 프로그램의 __getmainargs 실행 중 내부 스택 메모리에 들어있는 값이다. Fig. 3은 패킹을 하지 않은 프로그램의 __getmainargs 실행 중 스택 메모리에 들어있는 값이다. Fig. 2와 Fig. 3을 보면 실행할 프로그램호출경로가 시스템 기본값으로 들어있음을 볼 수 있다. OEP 이후 사용된 함수의 매개변수는 원프로그램과 패킹된 프로그램이 동일하다.

Fig. 2. Stack Memory Values Inside __getmainargs Function of a Program packed with PEcompact

Fig. 3. Stack Memory Values Inside __getmainargs Function of non-packed Program

3) OEP exists at program execution flow change

OEP는 프로그램 실행흐름이 바뀐 목적지 주소 중에 OEP가 있다. 프로그램 실행흐름이 바뀌었다는 것은 프로그램 실행 중 분기가 일어났다는 것이다. 분기가 일어나는 대표적인 경우로 jmp, call 명령어 수행이 있다. 패킹된 프로그램은 언패킹 영역에서 시작해서 언패킹 작업을 마치고 원바이너리코드로 프로그램 실행이 바뀐다. 이 때 분기가 발생한다. 그리고 이 분기의 목적지에 OEP가 존재한다.

‘프로그램 실행흐름이 바뀐 목적지 주소 중 OEP가 존재’한다는 특성은 PinDemonium의 Long jump와 Jump outer section 기법[8]의 기본이 되는 가정이다. Long jump와 Jump outer section은 프로그램 실행흐름 변경을 EIP로 찾는 기법이다[8]. Long jump는 현재 EIP와 바로 직전에 수행한 EIP의 차이를 활용하여 프로그램 실행흐름이 바뀐 지점을 찾는 기법이다. PinDemonium에서 구현된 코드를 보면 현재 EIP와 바로 직전에 수행한 EIP의 차이가 0x200 이상인 경우를 Long jump라 하며 Jump outer section은 현재 EIP와 직전에 수행한 EIP의 메모리 섹션이 달라진 경우다.

4) OEP in WrittenAndExecute pages

원바이너리 코드는 패커에 의해 복호화/압축해제 된다. 그 후 원바이너리 코드가 실행되기 때문에 OEP는 항상 메모리 쓰기 후 실행된 영역 안에 존재한다. 더 나아가 생각해보면 언패킹이 완벽하게 다 된 상태에서 마지막 메모리 쓰기 페이지에 실행된 명령어 주소가 OEP가 된다[3]. 패킹된 프로그램의 메모리 페이지에는 세 가지 유형[14]이 있고 각 유형에 대한 설명은 Table 2와 같다.

Table 2. Three Types of Memory Pages in a packed Program[14]

Classification	Description
Type 1	Pages executed after memory write (Modified and executed page).
Type 2	Pure status pages that have not been written or executed yet.
Type 3	Pages that have been written but never executed.

유형 1(Type 1)은 패커에 의해 복호화/압축해제 되고 원바이너리코드가 실행될 수 있으니 OEP가 존재한다. 유형 2(Type 2)는 패커에 의해 압축이 풀린 적 없고 실행도 된 적이 없으므로 OEP가 존재하지 않는다. 유형 3(Type 3)은 초기화가 된 데이터 페이지 혹은 복호화 된 페이지일 가능성이 있지만 원바이너리 코드가 실행되지 않았으니 OEP가 존재하지 않는다. 즉, 유형 1에만 OEP가 존재하고 유형 1은 메모리 쓰기 후 실행되었으므로 WrittenAndExecute 페이지라 할 수 있다.

하지만 유형 1 중 OEP가 아닌 경우가 있다. OEP는 프로그램 제작자가 작성한 코드가 들어가는 텍스트 메모리 영역 내 존재하므로 DLL 파일이 로드되는 메모리 영역에는 유형 1이라도 OEP가 없다.

3.2 OEP가 가지는 특성을 활용한 제안기법

본 절에서는 3.1에서 다룬 OEP가 가지는 특성 4가지를 이용해서 OEP를 찾는 방법을 제안한다. 전체적인 알고리즘은 Fig. 4와 같다. 사선으로 표시한 부분이 본 논문에서 새로이 추가한 제안기법이다. 가로선은 PinDemonium의 Long jump, Jump outer section[8]을 그대로 이용했다. 세로선은 Isawa et al.'s work[5] 중 언패킹 루틴을 마킹하는 방법을 이용하였다.

1) Sequence of system function call after OEP

본 제안기법은 3.1절의 1)의 특징을 활용하여 OEP를 찾는 방법이다. 본 제안기법을 수행 중 패킹된 프로그램이 비정상 종료되면 OEP후보를 출력하지 않는다. 이는 다양한 종류의 컴파일러(MSVC8.0, MSVC2013, tdm gcc 5.1.0 등)의 시스템 함수들 관찰한 결과를 토대로 만들었다(Table 3). 관찰에 따르면, 우선 패킹된 프로그램은 OEP 이후에 SE Handler를 설정하는 명령어를 실행한다. 두 번째로 패킹된 프로그램은 프

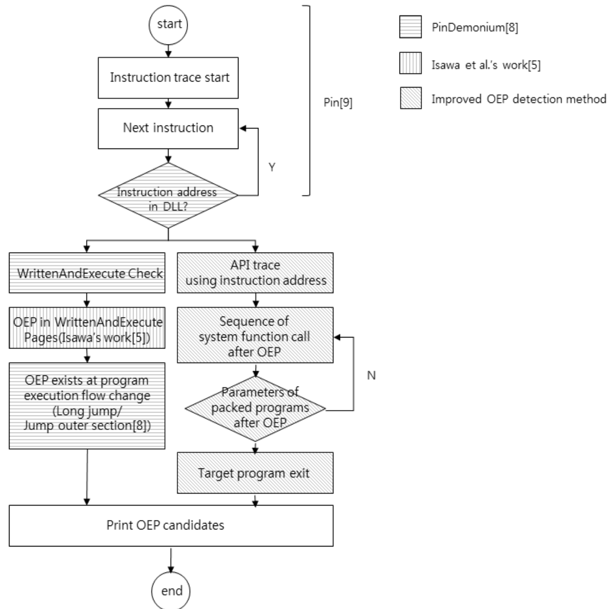

Fig. 4. Flow of the Suggestion

로세스 혹은 쓰레드의 ID나 Version 등을찾는 함수를 실행한다. 세 번째로 패킹된 프로그램은 포인터 변수 초기화 및 동적 메모리 공간(heap)을 할당하는 함수를 실행한다. 네 번째로 패킹된 프로그램은 인터페이스(CLI/GUI)형식을 설정하는 함수를 실행한다. 다섯 번째로 패킹된 프로그램은 초기값 설정을 위해 매개변수를 복사하거나 환경변수를 설정하는 함수를 실행한다. 위 과정을 Table 3의 구분과 동일하게 5단계로 나누면 아래와 같다.

- SE Handler installation 명령어인지 확인한다.
- ID, Version, Time 관련 함수인지 확인한다.
- MemoryAlloc, Initialize pointer 관련 함수인지 확인한다.
- AppType, ScreenInformation 관련 함수인지 확인한다.
- argument copy, enviromnet setting 관련 함수인지 확인한다.

위 구분의 함수/명령어가 순서대로 호출되면 되고 맨 마지막인 argument copy, enviromnet setting 관련 함수가 불리면 이 때 가장 최근에 발생한 WrittenAndExecute 주소가 OEP 후보가 된다.

2) Parameters of packed programs after OEP

함수 매개변수를 이용한 기법은 3.1절 2)의 OEP 이후 패킹된 프로그램의 시스템함수가 사용하는 매개변수는 원프로그램의 시스템함수가 사용하는 매개변수와 동일하다는 특성을 활용한 기법이다. 이 기법은 int main(int argc, char *argv)에 사용되는 매개변수인 프로그램명 혹은 프로그램실행경로를 스택 메모리에서 찾아 OEP 후보를 찾는 방법이다. 이 기법은 3단계로 구분되어 수행된다.

- 프로그램명과 실행경로가 매개변수로 사용되는 시스템함수(__getmainargs 등)가 불리는지 기다린다.
- 위 시스템 함수가 불리면 시스템 함수의 매개변수를 읽어 문자열로 저장한다.
- 문자열이 현재 실행중인 프로그램명과 동일한지 비교한다.

만일 현재 실행중인 프로그램명과 동일한 문자열이 존재한다면 이 때 가장 최근에 발생한 WrittenAndExecute 주소가 OEP 후보가 된다.

Table 3. Function Call Sequence for Each Compiler

Classification	SE Handler installation	ID, Version, Time	MemoryAlloc, Initialize pointer	AppType, ScreenInformation	argument copy, enviromnet setting
MSVC8.0	mov fs:[0]	GetVersion()	HeapCreate RtlAllocateHeap VirtualAlloc	GetStartupInfoA GetStdHandle GetFileType SetHandleCount	GetCommandLineA GetEnvironmentStringsW WideCharToMultiByte FreeEnvironmentStringsW GetACP GetCPInfo GetStringTypeW MultiByteToWideChar LCMapStringW GetModuleFileNameA IsBadReadPtr
MSVC2013	mov fs:[0]	GetSystemTimeAsFileTime GetCurrentThreadId GetCurrentProcessId QueryPerformanceCounter	_initterm_e _initterm	__set_app_type	RtlEncodePointer _controlfp_s _initterm_e _initterm __crtSetUnhandledException RtlDecodePointer __getmainargs
tdm gcc 5.1.0	mov fs:[0]	GetSystemTimeAsFileTime GetCurrentProcessId GetCurrentThreadId GetTickCount QueryPerformanceCounter	_initterm	__set_app_type _initterm __lconv_init __gconv_init	_initterm __getmainargs

3) OEP exists at program execution flow change

Long jump[8], Jump outer section[8] 을 이용한 기법은 3.1절 3)의 OEP가 가지는 '프로그램 실행흐름이 바뀐 목적지 주소 중 OEP가 존재' 한다는 특성을 활용한 기법이다. 이는 PinDemonium[8]에서 사용한 기법이고 본 논문에서도 동일하게 사용했다.

a) Long jump[8]: Long jump기법은 프로그램 실행간 EIP를 저장하고 다음 명령어를 실행하고 EIP를 저장한다. 전자에 저장한 EIP를 '이전 EIP'라 하고 후자에 저장한 EIP를 '현재 EIP'라 할 때, '현재 EIP'에서 '이전 EIP'를 빼면 '이전 EIP'대비 '현재 EIP'가 얼마나 이동했는지 알 수 있다. EIP의 이동이 임계값(0x200) 이상이 되면 Long jump가 된다. 본 제안기법에서는 코드영역 내 모든 Long jump의 목적지 주소를 OEP 후보로 출력한다.

b) Jump outer section[8]: Jump outer section 기법은 프로그램 실행간 EIP의 메모리 섹션명을 저장하고 다음 명령어를 실행하고 EIP의 메모리 섹션명을 저장한다. 전자에 저장한 EIP 의 메모리 섹션명을 '이전 EIP 섹션'라하고 후자에 저장한 EIP 의 메모리 섹션명을 '현재 EIP섹션'이라 할 때, '현재 EIP섹션'과 '이전 EIP 섹션'이 다르면 Jump outer section가 된다[8]. 이 때 Jump outer section의 목적지 주소를 OEP 후보로 출력한다. 본 제안기법에서는 코드영역 내 모든 Jump outer section의 목적지 주소를 OEP 후보로 출력한다.

4) OEP in WrittenAndExecute pages

본 제안기법은 3.1절의 4)의 특징을 활용하여 OEP를 찾는 기법이다. 이 기법은 OEP를 알려주기 보다는 WrittenAndExecute 주소들 모두를 OEP 후보라 했을 때 이들 중 OEP가 아닌 주소를 골라내는 기법이다. 이 기법은 Isawa et al.'s work의 U를 마킹하는 방법[5]과 동일하게 수행된다. 해당 단계에서 U 마킹이 된 주소는 OEP가 되지 못한다. 본 제안 기법은 4단계로 구분되어 수행된다.

- 메모리 쓰기 명령어를 추적한다.
- WrittenAndExecute 주소를 집합에 저장한다.
- WrittenAndExecute 주소 중 언패킹 작업에 사용된 주소에는 U마킹을 한다[5].
- U마킹이 된 WrittenAndExecute 주소는 OEP 후보로 출력하지 않도록 집합에 넣어 관리한다.

4. 실 험

4.1 실험환경

본 논문의 도구는 DBI인 Pin에 기반하여 PinDemonium[8]을 확장해서 구현했다.

Pin[9]은 DBI도구이며, IA-32, Intel(R) 64 아키텍처 용 Android, Linux, OS X, Windows 운영 체제 및 실행 파일을 지원한다. Pin은 실행 파일의 임의의 위치에 임의의 코드(C/C++)를 삽입 할 수 있다. 코드는 실행 파일이 실행되는 동안 동적으로 추가된다. 또한 이미 실행중인 프로세스에 Pin을 연결할 수 있다. Pin에는 캐시 시뮬레이터, 명령어 트레이스 생성기 등과 같은 기능들이 있다. 이 기능들을 활용하여 새로운 도구를 만들 수도 있고, 기능을 확장할 수도 있다. Pin에 대한 자세한 정보를 얻기 위해선 공식 문서 및 사용자 설명서[9]를 참조하면 된다.

실험은 16종(UPX, PECompack, ASPack, ASProtect, Obsidium, Themida, VMProtect, WWPack, Packman, petite, MEW, Enigma 등)의 패킹된 프로그램을 대상으로 했다. 사용된 프로그램은 단순한 텍스트를 콘솔창에 출력하며 tdm gcc 5.1.0으로 컴파일 했다. 실험환경의 OS는 Windows7 SP1을 사용했으며, Intel i7 CPU 3.4GHz, 14GB DDR3를 사용했다. 사용된 Pin 버전은 2.14이고, Pin의 플러그인은 VisualStudio 2010로 컴파일 했다. PinDemonium[8]은 마지막 수정 이력이 2016년 7월인 버전을 Github에서 다운받아 사용하였다.

4.2 제안기법 구현

본 제안기법은 3장 OEP가 가지는 특성과 제안기법 4가지를 토대로 구현했다. Table 4와 Table 5의 알고리즘은 Pin[9]의 API를 이용해 PinDemonium으로 구현했다. Table 6의 알고리즘은 PinDemonium[8]의 것을 그대로 이용했다. Table 7은 Isawa et al.'s work[5]을 참고하여 Pin[9]으로 직접 구현했다.

1) Sequence of system function call after OEP

본 단계에서는 시스템함수 호출순서로 OEP를 찾는다. 본 단계는 Table 3의 함수가 호출되었는지 확인하여 OEP 후보 주소를 반환한다. 본 단계의 알고리즘은 Table 4와 같다.

Table 4는 flag 변수들을 활용하여 Table 3의 함수들이 호출되었는지 확인하여 OEP 후보를 찾는다. 모든 flag 변수들이 TRUE가 되면 가장 최근의 WrittenAndExecute 주소를 OEP 후보로 반환한다.

Table 4. System Function Call Matching Pseudo Code

```
Algorithm: system function call matching
var flag 1, flag 2, flag 3, flag 4, flag 5 = FALSE;
while Pin api trace
  if(not long jump, jump outer section  detected)
    if(get SE Handler installation instruction)
    set flag 1 = TRUE;
    if(get ID, Version, Time function call)
    set flag 2 = TRUE;
    if(get MemoryAlloc, Initialize pointer function call)
    set flag 3 = TRUE;
    if(get AppType, ScreenInformation function call)
    set flag 4 = TRUE;
    if(get argument copy, enviromnet setting function call)
    set flag 5 = TRUE;
if(flag 1, flag 2, flag 3, flag 4, flag 5 == TRUE)
  Return recent WrittenAndExecute Address;
end while
```

2) Parameters of packed programs after OEP

본 단계에서는 시스템함수의 특정 매개변수로 OEP를 찾는다. 해당 단계에서는 int main(int argc, char *argv)의 매개변수를 활용해서 OEP 후보 주소를 반환한다. 본 단계의 알고리즘은 Table 5.와 같다.

Table 5. Parameters Matching Pseudo Code

```
Algorithm: parameters matching
var programName

while Pin instruction trace
  if(programName != Current Program name)
    if(get __getmainargs)
      get argument from stack memory
          at ESP+10 address;
      set programName;
    el if(get getCommnadLineA)
      get argument from stack memory
          at EAX value;
      set programName;
  if(programName == Current Program name)
    Return recent WrittenAndExecute Address;
end while
```

Table 5는 현재 실행된 프로그램명(Current Program name)과 동일한 문자열이 있는 매개변수를 찾아 OEP 후보를 찾는다. __getmainargs 실행 중 ESP+10에 있는 매개변수와 현재 실행된 프로그램명을 비교하여 동일하면 최근의 WrittenAndExecute 주소를 OEP 후보로 반환한다. 일부 프로그램은 __getmainargs을 사용하지 않는다. 이 경우는 getCommnadLineA 시스템 함수 실행 중 EAX 레지스터에 있는 주소값을 이용한다.

3) OEP exists at program execution flow change

본 단계에서는 Long jump와 Jump outer section을 찾는 부분으로 PinDemonium에 구현된 방법[8]과 동일하다. 본 단계는 분기를 활용하여 OEP 후보를 반환한다. 본 단계의 알고리즘은 Table 6과 같다.

Table 6. Execution Flow Change Detect Pseudo Code

```
Algorithm: execution flow change detect
var curEip
var preEip

while Pin instruction trace
  set preEip
  go next instruction
  set curEip

  if(|curEip-preEip| > 0x200) Return curEip;
  if(curEip's memory section name
      != preEip's memory section name) Return curEip;
end while
```

Table 6은 PinDemonium에 구현된 방법[8]과 동일하다. 자세한 정보는 PinDemonium[8]의 소스코드를 참조하면 된다. Table 6은 프로그램 실행 중 EIP를 활용해서 분기를 탐지하고 OEP 후보를 찾는다.

4) OEP in WrittenAndExecute pages

본 단계는 Isawa et al.'s work 의 U마킹하는 방법[5]과 동일하다. 하지만 Pin[9]으로 구현하며 Isawa et al.'s work 의 U마킹 하는 방법[5]과 약간 다른 점이 생겼다. Isawa et al.'s work는 OEP 탐지 도구를 구현함에 페이지 단위로 메모리 후킹해서 OEP를 찾고 있다. 하지만 Pin에는 타겟 프로그램이 수행하는 쓰기 명령어와 주소를 따라갈 수 있는 기능이 있어 Isawa et al.'s work[5]와는 다르게 페이지가 아닌 더 작은 단위인 주소를 사용해서 WrittenAndExecute 주소를 따라가는 도구를 구현했다. 본 단계는 Pin[9]으로 구현하며 달라진 점이 실험 결과에 영향을 주는지 알아보고자 주소를 페이지 단위로 바꿔 테스트되었다. 테스트는 본 논문의 실험 샘플들을 대상으로 했으며 결과는 본 논문의 실험결과인 Table 8과 동일하다.

본 단계는 WrittenAndExecute 주소 중 U마킹이 된 주소면 OEP 후보를 반환하지 않게 구현했다. 알고리즘은 Table 7과 같다.

Table 7. (Little) Modified Isawa et al.'s Work Pseudo Code

```
Algorithm: modified Isawa et al.'s work pseudo code
var writeAddressSet<src, dst>
var markedAddressSet<addr>

while Pin instruction trace
  if(IsWriteInstruction)
      writeAddressSet.push(src, dst)
  if(Eip in writeAddressSet's dst) marking U at src;
  if(any marked U writeAddressSet's src ==
      any in writeAddressSet's dst) marking U at dst;

  save marking U address at markedAddressSet;

  if(current WrittendAndExecute Address
      in markedAddressSet) Return ;
  else Return current WrittendAndExecute Address;
end while
```

Table 7은 패커코드에 의해 쓰여진 메모리 주소를 src(소스) 주소와 dst(목적지) 주소를 쌍으로 writeAddressSet에 저장하여 OEP 후보를 찾는다. WrittenAndExecute 주소는 writeAddressSet의 주소 중 쓰기 명령어의 목적지 주소(dst)와 EIP로 찾아진다. 프로그램 런타임 중 writeAddressSet의 dst 주소와 일치하는 EIP가 나오면 WrittenAndExecute 주소라 볼 수 있다. 이 때, writeAddressSet 안의 해당 쌍의 src 주소는 '생성주소'라 한다. 생성주소는 언패킹 루틴이 원바이너리코드를 생성하는 루틴이란 의미로 U라고 마킹한다. 그리고 writeAddressSet의 src, dst 주소 쌍을 시간 순으로 읽어 dst 주소에 U마킹이 있는지 찾는다. U마킹된 주소에 메모리

Table 8. Experiment Result

O = OEP exists in OEP candidates
X = OEP does not exists in OEP candidates

Classification	PinDemonium			Isawa et al.'s work			QuickUnpack2.2			Suggestion		
UPX 1.02	2	0.5	O	2	0.5	O	1	1.0	O	1	1.0	O
PECompack	5	0.2	O	2	0.5	O	1	1.0	O	1	1.0	O
ASPack 2.0	13	0.076	O	2	0.5	O	1	1.0	O	1	1.0	O
ASprotect1.2	92	0.010	O	16	0.062	O	1	-1.0	X	3	0.333	O
Obsidium1.3	68	0.014	O	31	0.032	O	1	-1.0	X	7	0.142	O
Themida2.2	592	0.001	O	26	0.038	O	1	-1.0	X	53	0.018	O
VMProtect 2.0	4	0.25	O	2	0.5	O	1	-1.0	X	1	1.0	O
WWPack	4	0.25	O	3	0.333	O	1	-1.0	X	1	1.0	O
Packman	2	0.5	O	2	0.5	O	1	1.0	O	1	1.0	O
petite	11	0.090	O	5	0.2	O	1	-1.0	X	2	0.5	O
MEW	3	0.333	O	3	0.333	O	1	1.0	O	1	1.0	O
Enigma4.03	10	0.1	O	9	0.111	O	1	-1.0	X	6	0.166	O
mpress	4	0.25	O	3	0.333	O	1	-1.0	X	2	0.5	O
nspack	4	0.25	O	2	0.5	O	1	-1.0	X	2	0.5	O
yoda1.3	2	-0.5	X	2	-0.5	X	1	-1.0	X	2	-0.5	X
safeingine 3.09	2	-0.5	X	2	-0.5	X	1	-1.0	X	2	-0.5	X

쓰기를 수행한 src 주소는 언패킹 루틴을 공유하고 있다는 의미이므로 이 역시 U라고 마킹한다.

본 단계는 언패킹 루틴에 이용되었다는 마킹 주소와 동일한 WrittenAndExecute는 OEP가 아니라고 판단하는 기법이다.

4.3 실험 결과

실험결과는 Table 8.과 같다. Table 8.의 'o'와 'x'는 출력된 OEP 후보들 중 실제 OEP가 있는지를 표시한 것이다. 그리고 정수는 OEP 후보의 수다. 실수는 OEP 후보 n개가 중 실제 OEP가 있을 확률이다. OEP 후보들 중 OEP가 존재한다면 1/n이다. OEP 후보들 중 OEP가 없는 경우 OEP가 있다고 오탐한 경우로 -1/n이다.

OEP 후보를 정한 방법은 다음과 같다. PinDemonium[8]의 경우는 모든 WrittenAndExecute 주소에 대해서 메모리 덤프를 수행하므로 PinDemonium이 메모리 덤프를 수행할 때 기준이 된 주소를 OEP 후보로 했다. Table 8.의 Isawa et al.'s work의 OEP 후보는 본 논문에서 Pin[9] 재현한 결과이며 OEP 후보들을 32개로 정렬하여 정렬된 OEP 후보 중 몇 번째 위치에 정렬되어 있는지를 표시한 것이다. QuickUnpack2.2[15]는 OEP로 추정되는 주소 1개를 출력하는 도구기에 OEP 후보 또한 1개만 출력한다. 본 논문에서 제안하는 방법으로 출력한 OEP 후보의 수는 'Suggestion'이다.

본 논문에서 제안하는 방법으로 OEP를 찾지 못하는 패커(yoda 1.3, safengine 3.09)는 Pin이 안티디버깅 기법에 탐지되어 프로그램을 끝까지 실행시키지 못하는 경우다. 순수 Pin으로는 안티디버깅 기법에 감지되어 Obsidium1.3으로 패킹된 프로그램을 실행시키지 못한다. 하지만 OEP가 찾아진 이유로 PinDemonium을 확장하여 구현했기 때문에 가능해 보인다. 이는 예외적인 경우다.

본 논문에서는 16종의 패킹된 파일 중 14종의 패킹된 파일에 대해 PinDemonium 대비 평균 40% 이상의 많은 OEP 후보를 줄였다. 특히 ASProtect와 Themida에서는 OEP 후보를 90% 이상 줄이는 결과가 나왔다. 2종의 경우는 패커에 적용된 안티 디버깅 기법에 Pin[9]이 탐지되어 OEP를 찾지 못한다.

본 논문의 연구는 OEP를 탐지하는 도구의 실행속도 향상을 고려하지 않았다. 대신 OEP 탐지도구가 제공하는 OEP 후보 집합의 크기를 줄여 OEP를 탐지하는 도구의 정확성을 향상시키고자 했다. Table 8.의 OEP를 찾을 확률의 합은 Suggestion이 8.159로 가장 높고 Isawa et al.'s work이 3.442로 다음이며, PinDemonium이 1.824, QuickUnpack2.2이 -6.0 순이 된다. QuickUnpack2.2가 음수의 값인 현재의 이유로는 16종의 패킹된 샘플에서 OEP를 찾는 경우보다 오탐인 경우가 더 많았기 때문이다. OEP를 찾을 확률의 합이 높다는 것은 적은 수의 OEP 후보들로 OEP를 찾아준다는 의미다.

패킹된 악성코드는 악성코드 분석가로 하여금 악성코드를 분석하기 어렵도록 하는데 사용되고 패킹된 악성코드를 해제하기 위해 OEP를 찾기 위한 시간이 걸리는 동안 악성코드 전파 시간을 늘릴 수 있다. 패킹된 악성코드는 변종이 많아 OEP를 검증하기 위해 악성코드 분석가가 직접 바이너리 코드를 읽는 시간이 필요하다.

좋은 OEP 탐지 도구는 적은 OEP 후보들을 출력하고 후보들 안에 OEP가 존재해야 한다. OEP 탐지 도구가 출력한 OEP 후보들의 수는 적지만 그 안에 OEP가 없다면, 악성코드 분석가들은 OEP 후보들을 검증하며 시간을 낭비한다. 반대로, OEP 후보들의 수가 너무 많으면 그 안에 OEP가 있더라도 악성코드 분석가들이 OEP 후보들을 검증하는데 많은 시간이 든다.

이에 악성코드 분석가들이 패킹된 악성코드에 민첩하게 대응하기 위해서 기존의 도구보다 정확성 높은 OEP 탐지 도구가 필요하다. 본 논문의 제안 방법은 악성코드 분석가들이 패킹된 악성코드의 OEP를 찾기 위한 수고와 번거로움을 줄일 수 있을 것이다.

5. 결 론

악성코드 분석가는 패킹된 악성코드를 분석하기 위해서 언패킹 작업을 해야 한다. 악성코드 분석가가 패킹된 악성코드를 언패킹하기 위해서는 패킹이 끝난 지점 OEP를 먼저 찾아야 한다.

패킹된 파일은 OEP를 쉽게 찾지 못하게 악성코드 분석 방해 기술이 적용되어 있다. 패킹된 프로그램은 실행 흐름을 복잡하게 하여 분석까지 많은 시간을 소요하도록 만들기도 하며 프로그램 실행 시, 매번 코드가 변하는 다형성 성질이 적용되어 있다.

따라서 본 논문에서는 새로운 방법 두 가지를 제안하고, 기존에 연구된 두 가지 방법을 섞어서 더 효과적인 OEP 찾는 도구를 개발하였다.

본 논문에서 제안한 첫 번째 방법은 패킹된 프로그램이 완전히 원본 바이너리 상태로 되돌아 간 후에는 원프로그램 함수 호출과 동일하다는 것을 활용한 OEP 탐지방법이다. 컴파일러 별로 사용되는 시스템 함수를 조사한 후, 패킹된 프로그램에서 호출되는 시스템 함수와 패턴매칭하여 언패킹 작업이 끝났는지 탐지하는 방법을 추가했다. 두 번째 방법은 메인함수인 int main(int argc, char *argv)의 서브루틴(__getmainargs 등)에서 사용되는 매개변수를 이용해 OEP를 찾는 방법이다.

기존에 연구된 방법을 사용한 것은 PinDemonium에 구현된 Long jump, Jump outer section[8]이다. 그리고 Isawa et al.'s work 의 U마킹 하는 방법[5]이다.

본 논문에서 제안한 알고리즘을 이용하면 악성코드 분석가는 안티디버깅 기법으로 인해 Pin으로 프로그램을 실행하지 못하는 경우인 2종을 제외하고 14개 패커에서 PinDemonium 대비 평균 40% 이상 OEP 후보를 줄일 수 있다. 특히 ASProtect와 Themida에서는 OEP 후보를 90% 이상 줄이는 결과가 나왔다.

본 논문에서 제안한 알고리즘은 패킹된 악성코드 분석을 위해 OEP를 찾고 언패킹 기술에 대한 추가적인 연구에 기여할 수 있을 것으로 예상된다.

References

[1] AV-TEST, publication [internet], https://www.av-test.org/fileadmin/pdf/publications/blackhat_2006_avtest_presentation_runtime_packers-the_hidden_problem.pdf

[2] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *ieee seCurity & PrivaCy*, Vol.6, No.5, 2008.

[3] J. Lenoir, "Implementing your own generic unpacker."

[4] Wikipedia, Entry point [Internet], https://en.wikipedia.org/wiki/Entry_point

[5] R. Isawa, D. Inoue, and K. Nakao, "An original entry point detection method withcandidate-sorting for more effective generic unpacking," *IEICE TRANSACTIONS on Information and Systems*, Vol.98, No.4, pp.883-893, 2015.

[6] Hex-ray, Universal PE Unpacker [Internet], https://www.hex-rays.com/products/ida/support/tutorials/unpack_pe/index.shtml

[7] P. Royal, M. Halpin, and D. Dagon, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-ExecutingMalware," In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, IEEE, pp.289-300, 2006.

[8] D'ALESSIO, S. T. E. F. A. N. O., and S. MARIANI, "PinDemonium: a DBI-based generic unpacker for Windows executables," Black hat, 2016.

[9] Intel, Pin User Manual [Internet], https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/

[10] Scylla, NtQuery Scylla - x64/x86 Imports Reconstruction, [internet], https://github.com/NtQuery/Scylla

[11] Jibz, Qwerton, XineohPSnaker, and PEiD BOB. "Peid." Availablein: http://www. peid. info/, 2011.

[12] YARA group, yara [internet], https://virustotal.github.io/yara/

[13] Microsoft, MSDN [internet], https://msdn.microsoft.com

[14] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin, "MutantX-S: Scalable Malware Clustering Based on Static Features," *USENIX Annual Technical Conference*, pp.187-198, 2013.

[15] AHTeam, Quick Unpack 2.2 [Internet], http://qunpack.ahteam.org/?author=1

김 경 민

https://orcid.org/0000-0003-3143-2706

e-mail : casualab@hanyang.ac.kr

2015년 중앙대학교 연극영화학부, 컴퓨터공학부(학사)

2016년～현 재 한양대학교 컴퓨터소프트웨어학부 석사과정

관심분야 : Computer Security, Malicious Code Analysis

박 용 수

https://orcid.org/0000-0002-7354-4434

e-mail : yongsu@hanyang.ac.kr

1996년 KAIST 전산학과(학사)

1998년 서울대학교 컴퓨터공학과(석사)

2003년 서울대학교 전기컴퓨터공학부(박사)

2003년～2004년 서울대학교 자동제어특화연구센터 연수연구원

2005년～현 재 한양대학교 컴퓨터소프트웨어학부 교수

관심분야 : Computer Security, Internet Security