ISSN 1598-0170 (Print)
ISSN 2287-1136 (Online)
http://www.jksii.or.kr

분석 환경에 따른 안티 디버깅 루틴 자동 탐지 기법

An automatic detection scheme of anti-debugging routines to the environment for analysis

박 진 우[1] 박 용 수[2*]
Jin-woo Park Yong-su Park

요 약

여러 가지 역공학 방지기술들 중 하나인 안티 디버깅 기술은 특정 프로그램을 대상으로 공격자나 분석가가 디버거를 사용하여 분석을 하지 못하도록 하기 위한 기술로써, 예전부터 악성코드 및 분석을 방지하고자 하는 여러 가지 프로그램들에 적용이 되었으며 현재까지도 많이 사용이 되고 있는 기술이다. 본 논문에서는 이러한 안티 디버깅 루틴에 대한 자동화 탐지 방법을 제안한다. 탐지는, 디버거 및 시뮬레이터를 통해 실행 명령어 및 API(Application Program Interface)에 대한 트레이스 정보들을 추출하고, 추출된 정보들을 비교하여 안티 디버깅 루틴으로 의심이 가는 지점을 찾는 방식으로 진행된다. 실험 결과, 알려진 25가지의 안티 디버깅 기법들 중 21가지에 대하여 정상적으로 탐지가 이루어졌다. 이와 같이, 본 기법은 특정 안티 디버깅 기술에 의존적이지 않으며, 추후 개발 및 발견되는 안티 디버깅 기술들에 대한 탐지의 경우에도 적용이 가능할 것으로 예상된다.

☞ 주제어 : 안티 디버깅, 악성코드, 명령어 트레이스, API 트레이스, 안티 리버싱, 역공학 방지 기술

ABSTRACT

Anti-debugging is one of the techniques implemented within the computer code to hinder attempts at reverse engineering so that attackers or analyzers will not be able to use debuggers to analyze the program. The technique has been applied to various programs and is still commonly used in order to prevent malware or malicious code attacks or to protect the programs from being analyzed. In this paper, we will suggest an automatic detection scheme for anti-debugging routines. With respect to the automatic detection, debuggers and a simulator were used by which trace information on the Application Program Interface(API) as well as executive instructions were extracted. Subsequently, the extracted instructions were examined and compared so as to detect points automatically where suspicious activity was captured as anti-debugging routines. Based on experiments to detect anti-debugging routines using such methods, 21 out of 25 anti-debugging techniques introduced in this paper appear to be able to detect anti-debugging routines properly. The technique in the paper is therefore not dependent upon a certain anti-debugging method. As such, the detection technique is expected to also be available for anti-debugging techniques that will be developed or discovered in the future.

☞ keyword : Anti-debugging, Malware, Instruction trace, API trace, Anti-reversing, Anti-reverse Engineering

1. 서 론

리버스 엔지니어링(reverse engineering)이라고 불리는 역공학 기술은 소프트웨어 공학의 한 분야로써, 이미 만들어진 시스템을 역으로 추적하여 처음의 문서나 설계기법 등의 자료를 얻어내는 기술을 말하며, 이 기술은 시스템을 이해하여 적절히 변경하는 소프트웨어 유지보수 과정의 일부로 사용되고 있다.

하지만 이러한 역공학 기술이 프로그램의 중요 기술 및 정보를 빼내는 경우와 같이 나쁜 의도로도 악용됨으로써 많은 피해가 발생되고 있다.

그 밖에 이러한 역공학 기술은 악성코드의 분석에도 사용이 되는데, 대다수의 악성코드들은 자신을 역공학을 통한 분석으로부터 보호하기 위해서 안티 디버깅, 실행 압축, 코드 가상화 등의 기술들을 사용한다.

이러한 분석 방해 기술들 중에서도 공격자나 분석가가 디버거를 통하여 분석을 하는 행위 자체를 막기 위한 목적을 가지는 기술인 안티 디버깅 기술의 경우, 6,222개의 악성코드 샘플들을 대상으로 조사한 결과 그 중 3,662(58.5%)

[1] Department of Electronics and Computer Engineering, Hanyang University, Seoul, 133-791, Rep. of Korea.
[2] Department of Computer Science and Engineering, Hanyang University, Seoul, 133-791, Rep. of Korea.
* Corresponding author (yongsu@hanyang.ac.kr)
[Received 20 August 2014, Reviewed 22 August 2014, Accepted 15 September 2014]

개의 샘플들에서 발견이 될 정도로 많이 사용이 되고 있는 기술이며[1], 여러 가지 다양한 방법들을 통한 디버거 탐지 기술들을 지니고 있다.

안티 디버깅 기술은 분석 대상 프로그램을 디버깅을 하지 못하도록 디버거를 강제 종료시키거나 에러를 발생시키는 방법 등 다양한 방법을 사용하여 분석을 하지 못하도록 방해를 한다. 그렇기 때문에 이러한 기술이 적용된 악성코드 및 기타 프로그램들의 경우 분석가들로 하여금 분석을 하기 어렵게 만든다. 이에 보다 빠른 분석을 할 수 있도록 하기 위하여 분석을 하고자 하는 프로그램의 안티 디버깅 기술 적용 유무 및 해당 루틴을 자동으로 탐지해내는 기술이 요구된다.

이에 본 논문에서는 안티 디버깅 기술이 적용된 악성코드를 대상으로 해당 안티 디버깅 기술의 적용 유무 및 루틴에 대한 자동화 탐지 기법에 대해 설명을 한다.

2. 관련 연구

2.1 분석 도구

본 논문은 80x86 컴퓨터 구조를 가정하고 있다. 80x86 구조에서 바이너리 코드를 분석하는데 사용되는 도구들에는 Pin[2], OllyDbg[3], IDA Pro[4], Immunity Debugger, WinDbg 등이 있으며, 이 중 본 논문에서 실험 및 분석에 사용한 툴들은 Pin, OllyDbg, IDA Pro 이다.

2.2 안티 디버깅 기술

안티 리버싱 기술 중 하나인 안티 디버깅은 말 그대로 공격자나 분석가가 디버거를 통하여 인증과 관련된 프로그램이나 악성코드 혹은 기타 중요한 프로그램들의 내부 알고리즘과 중요 데이터들에 대한 분석을 하지 못하도록 하는데 목적을 가지는 기술이다.

이러한 안티 디버깅 기술이 적용된 프로그램이 실행 중에 있을 경우에 디버깅을 당하게 된다면, 디버깅을 하지 못하도록 디버거를 강제 종료시키거나 에러를 발생시키는 방법 등 다양한 방법을 사용하여 분석을 하지 못하도록 방해를 한다.

안티 디버깅 기술에는 수많은 방법들이 존재하고 계속 발전하고 있으며, 이를 우회하는 방법 또한 계속적으로 발전되고 있다.

본 논문에서는 80x86환경에 초점을 맞추며, 안티 디버깅 기법들을 [5]의 방식에 따라 크게 6가지의 기술로 분류를 실시하고, 각각에 해당되는 세부 기술들을 일부 나열해보면 표 1과 같다.

2.3 안티 디버깅 탐지 관련 기술

안티 디버깅 기술에는 수많은 방법들이 존재하고 계속 발전하고 있으며, 이를 발견해내는 방법 또한 계속적으로 발전이 되고 있다. 하지만 아직까지 이러한 안티 디버깅 루틴을 자동으로 탐지하는 방법에 대하여 기술이 된 문서는 많지가 않은데, 그 중 한 가지 자동화 탐지 기법에 대하여 어떠한 방식으로 접근을 하고 있는지 살펴보면 다음과 같다.

먼저, Peidai Xie 등의 연구를 보면[6], 해당 논문에서는 패턴 매칭 방식을 이용한 안티 디버깅 자동화 탐지 기법에 대해서 다루고 있다. 탐지 방식을 살펴보면, 먼저 여러 가지 안티 디버깅 기술에 관한 명령어 부분을 하나의 데이터베이스와 같은 저장 매체를 통하여 수집을 한다. 그 다음 분석을 하고자 하는 프로그램을 대상으로 명령어 트레이스를 추출한다. 이렇게 추출된 명령어 트레이스는 먼저 정제 과정을 거치고, 그 다음 데이터베이스에 보관된 규칙들과의 코드 매칭을 통해 그 결과에 따라서 안티 디버깅 기술을 탐지해내는 방식이다.

다음으로, JaeKeun Lee 등의 연구를 보면[7], 룰 기반의 패턴 매칭 방식을 이용한 안티 디버깅 탐지 및 회피 기법에 대해서 다루고 있다. 탐지 방식을 살펴보면, 먼저 탐지 대상 바이너리 파일을 디스어셈블 시킨 뒤, 디스어셈블된 명령어들을 한 줄씩 검사해 나가면서 기존에 구성되어있는 안티 디버깅 탐지 시그니처들로 이루어진 룰들과 일치하는 부분이 있는지를 탐색한다. 그리고 만약 일치하는 명령어가 탐지될 경우 offset 정보와 함께 따로 로그를 남기게 된다. 마지막으로 바이너리 파일에서 로그로 남겨진 명령어의 위치가 식별되면, 해당 지점의 바이트 시퀀스 부분을 회피하기 위하여 기존에 구성되어있는 패칭과 관련된 룰에 따라서 새로운 바이트 시퀀스로 수정을 하게 된다.

이와 같은 패턴 매칭을 이용한 탐지 방식들의 경우에는 새로운 안티 디버깅 루틴이 발견될 때마다 해당 루틴의 코드 부분을 규칙들로 생성하여 데이터베이스화 하는 작업을 가져야한다는 단점이 있다.

그래서 이러한 패턴 매칭을 이용한 탐지 방식과는 다르게 본 논문에서는 Hardware instruction set에 해당되는 실행 명령어 및 System call API에 대한 트레이스 결과를 추출하고 이를 토대로 하여 명령어의 흐름 비교 작업을

통해서 서로 다른 분기가 발생되는 지점을 찾아낸다. 그 다음 해당 지점으로부터의 API 트레이스 결과에 대한 분기 발생 여부에 대한 확인 절차를 통하여 명확한 안티 디버깅 탐지를 수행한다.

다음으로는 본 논문에서 접근을 하고자 하는 방식 중 일부가 유사하게 사용되면서 이를 통해 악성코드의 이중적인 행위(Split Personality)를 탐지해내는 방식에 대하여 설명한 Davide Balzarotti 등의 연구를 보면[8], 해당 논문에서는 악성코드를 대상으로 하여 분석하고자 하는 악성코드의 이중적인 행위에 따라 서로 다르게 실행되는 시스템 콜 목록 및 인자 값들의 특성을 이용하여 단순하게 악성코드의 이중적인 행위만을 탐지해내는 것을 목적으로 한다. 분석 과정을 살펴보면, 먼저 서로 다른 분석 환경에서 동일한 악성코드에 대한 시스템 콜 트레이스들을 각각 추출한다. 그 다음 추출된 시스템 콜 목록 및 인자 값들에 대한 비교를 통해 서로 다른 시스템 콜이 사용되는 것을 파악하고 이를 통하여 해당 악성코드에 대한 이중적인 행위를 탐지해낸다.

이와 같은 방식은 단순 시스템 콜의 비교를 통하여 악성코드의 이중적인 행위여부를 판별해내는 탐지 방식으로써, 본 논문에서 접근하고자하는 명령어와 API 트레이스를 함께 추출하여 서로 다른 흐름의 발생 유무를 파악해내는 방식과 시스템 콜에 대한 트레이스의 비교 과정이 사용되어지는 점에서는 유사하다. 하지만 본 논문에서는 실행 명령어에 대한 트레이스를 비교하는 과정을 메인 탐지 기술로 사용을 하며, 이를 통해서 서로 다른 실행 흐름이 발생되기 시작하는 지점을 파악할 수 있다. 그리고 앞서 파악된 서로 다른 분기가 이루어지는 지점에서의 실행 흐름이 정상 실행일 경우의 흐름과 동일해지도록 강제 바이트 패치를 수행한 뒤, 추가적으로 서로 다른 실행 흐름이 발생되는 지점을 찾아낼 수 있도록 한다. 그러므로 본 논문에서 설명하고자 하는 탐지 방식은 위의 논문에서의 탐지 방식과는 다르다고 할 수 있다.

3. 안티 디버깅 탐지 방법

3.1 안티 디버깅 탐지 실험

먼저 안티 디버깅 기술이 적용되어 있는 악성코드를 대상으로 정상적인 실행일 경우와 디버거를 통한 실행일 경우에 사용되는 각각의 실험 도구로 Pin tool과 IDA Pro를 사용하여, 실행이 이루어지는 명령어 및 API에 대한 트레이스를 추출하는 과정을 거치게 된다. 이 때 명령어의 경우에는 전체 실행 명령어들을 기록할 경우 광대한 양의 로그 발생으로 인해 비교 분석이 어려울 수 있기 때문에, 실행 명령어들 중 분기가 발생되는 명령어들을 대상으로 로그를 남기며, 만일 분기의 타겟이 System call API일 경우에는 해당 API의 이름 및 실행되는 주소를 로그로 남기게 된다. 여기에서 분기를 발생시키는 명령어들을 정리해보면 다음과 같다.

· 분기 발생 명령어 : jnz, jmp, jz, ja, jae, jb, jbe, jc, je, jg, jge, jl, jna, jnb, jnc, jne, jng, jnl, jnp, js, js, jo, jp, retn, call

다음의 그림 1은 이와 같은 방식으로 Pin tool을 사용하여 정상적으로 실행이 이루어지는 경우에 해당되는 분기 명령어 트레이스와 IDA Pro를 사용하여 디버거 실행에 대한 탐지가 이루어질 경우에 해당되는 분기 명령어 트레이스를 추출한 결과의 일부인데, 여기에서 화살표를 기준으로 좌측에 표시된 주소는 분기가 이루어지기 이전의 명령어에 해당되는 주소 값이며, 우측에 표시된 주소는 분기가 이루어진 이후의 명령어에 해당되는 주소 값이다. 그리고 콜론이 포함되어진 경우는 API의 호출이 이루어진 경우이며, 콜론의 우측에 표시된 문자열은 해당 지점에서 호출된 API의 이름을 나타낸다.

정상적인 실행	디버거 실행
40f19c -> 40f0df	40f19c -> 40f0df
40f0fd -> 7c81e85c : DeleteFileA	40f0fd -> 7c81e85c : DeleteFileA
40f00e -> 40d137	40f00e -> 40d137
40d137 -> 40f2a5	40d137 -> 40f2a5
40f2a9 -> 40f2af	40f2a9 -> 40f2af
40f2af -> 40f2b5	40f2af -> 40f2b5
40f303 -> 40f035	40f303 -> 40f035
40f039 -> 40f408	40f039 -> 40f03f
40f408 -> 40d0bc	40f044 -> 40f40d
40d0fe -> 40f272	40f413 -> 40f242
40f277 -> 40d09c	40f246 -> 40f12e
40d0a4 -> 40f0cc	40f12e -> 40f418
...	...
40d045 -> 40f10c	40f19c -> 40f0df
40f121 -> 7c801d77 : LoadLibraryA	40f0fd -> 7c81e85c : DeleteFileA

(그림 1) 분기 명령어 트레이스의 일부
(Figure 1) Partial results for the trace of branch instructions

위와 같이 추출이 이루어진 분기 명령어들에 대한 트레이스 결과를 비교해보면, 0x40f039 지점으로부터 서로 다른 명령어들이 실행되는 것을 발견할 수 있다. 그리고 다음으로 해당 지점 이후로 실행되는 API를 비교해보면 서로 다른 API가 호출되는 것을 확인할 수 있다. 다음으로 안티 디버깅 루틴에 대한 정확한 확인을 위하여 서로 다른 명령어의 흐름이 이루어지기 시작하는 해당 주소지(0x40f039) 부분으로 이동하여 이전에 실행된 명령어들을 살펴보면, 아래의 그림 2와 같은 부분을 볼 수 있다.

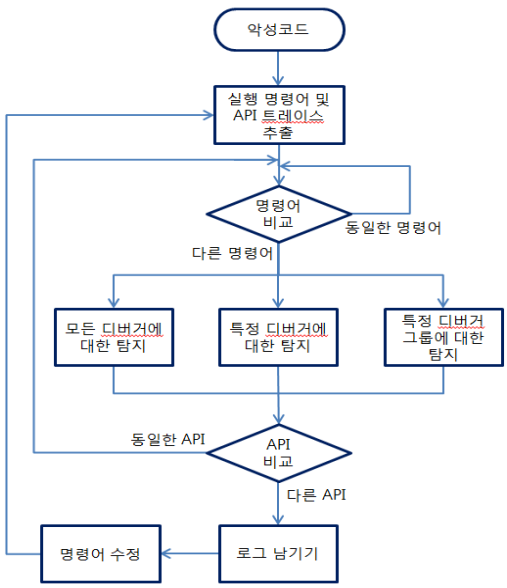
```
0x40f2ea :  mov eax, dword ptr fs:[0x30]
0x40f2f0 :  pushfd
0x40f2f1 :  nop
0x40f2f2 :  pushfd
0x40f2f3 :  mov eax, dword ptr [eax+0x18]
0x40f2f6 :  pushfd
0x40f2f7 :  nop
0x40f2f8 :  stc
0x40f2f9 :  mov eax, dword ptr [eax+0xc]
0x40f2fc :  stc
0x40f2fd :  mov byte ptr [esp], cl
0x40f300 :  cmp eax, 0x2
0x40f303 :  call 0x40f035
0x40f035 :  lea esp, ptr [esp+0x10]
0x40f039 :  jz 0x40f408
```

(그림 2) 분기 발생 이전의 안티 디버깅 루틴
(Figure 2) An Anti-debugging routine before the branch

위의 코드를 보면, 먼저 PEB의 ProcessHeap의 flag 정보를 가져오는 것을 확인할 수 있다. 해당 flag 값은 만약 디버깅을 당하고 있다면 대게 0x50000062로 설정이 되어 있고, 디버깅을 당하고 있지 않다면 0x2로 설정이 되어 있다. 이러한 ProcessHeap의 flag 정보를 확인하여, 값이 0x2가 아니면 쓰레드 종료 루틴으로 점프를 수행하고 0x2가 맞으면 점프 명령어의 수행을 무시하고 다음의 명령어 지점으로 흐름이 변경되는 것을 확인 할 수 있다.

이와 같이 명령어 트레이스의 비교를 통해서 만일 서로 다른 분기가 이루어지는 부분이 발견이 된다면, 해당 부분에 디버깅 여부를 판단하는 수행이 이루어지는 안티 디버깅 루틴이 존재한다는 것을 확인할 수 있다.

3.2 안티 디버깅 자동 탐지 알고리즘



(그림 3) 안티 디버깅 탐지에 관한 흐름도
(Figure 3) State machine of Anti-debugging detection

그림 3의 안티 디버깅 탐지 방법에 관한 흐름도에 대해서 살펴보면, 먼저 Pin Tool을 사용하여 정상적인 실행 환경에 대한 실행 명령어 및 API에 대한 트레이스를 추출하고, OllyDbg나 IDA Pro와 같은 디버거들을 사용하여 디버거를 통한 실행 환경에서의 실행 명령어 및 API에 대한 트레이스들을 추출한다.

그 다음, 안티 디버깅 기술을 통하여 서로 다른 분기가 이루어지기 시작하는 지점의 명령어 및 주소에 대한 정보를 얻기 위하여 추출된 트레이스 정보들 중 먼저 실행 명령어에 대한 트레이스 정보만을 비교하여 서로 다른 명령어의 주소로 이동되는 지점을 찾아낸다.

그런데 서로 다른 주소로 이동하였다 할지라도 실행 환경이나 실행 시간 또는 실행되고 있는 프로세스 등에 따라서 코드 실행 중 추가적인 루프의 발생이 이루어질 수도 있다. 그리하여 분석을 해보면 결과적으로는 같은 수행이 이루어지는 경우가 발생할 수 있기 때문에, 확실한 검증을 위하여 해당 지점 이후로 호출되는 API에 대한 비교 작업을 수행한다.

다음으로 위의 API에 대한 비교 작업의 결과가 만일 같다면 해당 지점으로부터 다시 명령어 트레이스 정보를

비교하는 작업을 수행한다. 하지만 서로 다른 API에 대한 호출이 이루어졌다면 해당 지점 이후부터는 완전히 서로 다른 흐름 및 행위가 발생되었다는 것이기 때문에, 해당 지점이 디버거 탐지로 인하여 분기가 발생된 지점이라고 볼 수 있다.

이에 먼저, 해당 분기 지점의 명령어에 대한 주소 정보를 로그로 남긴다. 그리고 분석 대상 파일에 그 외의 안티 디버깅 기술이 더 적용되어 있을 수도 있으므로, 추가적으로 적용된 안티 디버깅 기술에 대한 탐지를 위하여, 이전에 발견된 서로 다른 분기를 발생시킨 조건부 명령어 부분을 무조건적인 이동을 발생시키는 "JMP" 명령어를 사용하여 명령어의 실행 흐름이 정상 실행일 경우와 같아지도록 변경시킨다. 그 다음 해당 디버거에 대한 트레이스를 재추출하고, 이전의 비교하는 과정을 다시 한 번 수행하여 추가적인 안티 디버깅 기술이 적용되어 있을 경우 이를 탐지해낸다.

Algorithm : 안티 디버깅 탐지

```
T_d = { t_d | t_d is instruction and API trace using debuggers}
T_p = { t_p | t_p is instruction and API trace using Pin}
F_n = Debugger classification Flag
N = { n | n is each debugger's number}

# split t_d to t_d.inst and t_d.api
for i in N:
  if t_di.api in t_di:
    split t_di to (t_di.inst, t_di.api)
  else:
    t_di.inst = t_di

# split t_p to t_p.inst and t_p.api
if t_p.api in t_p:
  split t_p to (t_p.inst, t_p.api)
else:
  t_p.inst = t_p

# compare with instructions
for i in N:
  if t_p.inst == t_di.inst:
    none
  else:
    # compare with APIs
    if t_p.api == t_di.api:
      none
    else:
      log to file (t_di.inst)
      F_i = True
      # Byte patching with pin's eip address
      patch t_di.inst to ("JMP t_p.(eip)addr")
      get T_d

# Classification of debugger detection
for i in N:
  if F_i is True:
    Debugger i is detected
  else:
    none
```

위의 안티 디버깅 탐지 알고리즘에 대한 설명을 하자면, 먼저 Pin Tool과 탐지에 사용 될 디버거들을 통하여 실행 명령어 및 API에 대한 트레이스를 추출한 뒤, 추출된 전체 트레이스 결과 중 실행 명령어에 대한 정보와 API에 대한 정보를 따로 나누어서 구분 짓는다.

다음으로, 구분지어진 정보들 중 먼저 실행 명령어에 대한 정보들을 비교하는 과정을 거치는데, 만일 해당 정보들이 일치한다면 계속해서 비교를 하고, 일치하지 않는다면 해당 지점 이후로 호출되는 API 정보를 비교하게 된다.

이에 만일 서로 다른 API가 호출되었다면, 해당 지점의 실행 명령어 정보를 로그로 남기고, 어떠한 디버거에서 탐지가 이루어진 것인지를 구분하기 위하여 특정 플래그의 값을 True로 설정한다. 그리고 추가적으로 적용되어 있을 지도 모르는 안티 디버깅 기술을 탐지하기 위하여 Pin Tool의 실행 명령어 정보들 중 탐지 지점 바로 다음의 실행 명령어에 대한 주소 정보를 참고하여 디버거에서의 실행 흐름이 이와 강제적으로 같아지도록 "JMP" 명령어를 사용하여 바이트 패치를 실시하게 된다.

4. 실험 결과

4.1 기술 별 탐지 실험

안티 디버깅에는 수많은 기술들이 존재하고 있는데, 그 중 대표적인 안티 디버깅 기술[5]들에 대하여 각각의 기술 별 안티 디버깅 샘플들을 수집하고, 이를 통해 수집된 총 25개의 샘플들을 대상으로 본 논문에서 제안하고자 하는 방법을 사용하여 안티 디버깅 탐지 실험을 실시하였다. 이에 대한 탐지 결과를 보면 아래의 표 1과 같다.

(표 1) 기술 별 탐지 실험 결과
(Table 1) Experiment results on detection for each Anti-debugging technique

안티 디버깅 기술		탐지
API Based Anti-debugging	IsDebuggerPresent	O
	FindWindow	O
	CheckRemoteDebuggerPresent	O
	NtQueryInformationProcess	O
	NtQueryObject	O
	OllyDbg OllyInvisible Detect ion	O
	OllyDbg OpenProcess() String Detection	O
	OllyDbg Registry Key Detection	O
	OutputDebugString on Win 2K and WinXP	O
	TLS-CallBack + IsDebuggerP resent() Debugger Detection	O
Direct Process and Thread Block Detections	PEB ProcessHeap Flag Debugger Detection	O
	PEB BeingDebugged Flag Debugger Detection	O
	PEB NtGlobalFlag Debugger Detection	O
	LDR_MODULE	O
Hardware and Register Based Detection	Hardware Breakpoints	O
Timing Based Detections	RDTSC (Read Time Stamp Counter)	X
	GetTickCount	O
Modified Code Detections	CRC (Cyclic Redundancy Checking)	O
	OllyDbg OpenProcess() HideDebugger Detection	O
Exception Based Detections	Single Step Detection	X
	Privileged instruction	O
	INT 2D	X
	Unhandled Exception Filter	O
	OllyDbg INT3 Exception Detection	O
	Memory Breakpoint Detect ion	X

안티 디버깅 기술 별 탐지 실험을 실시해 본 결과, 위와 같이 크게 6가지로 분류된 안티 디버깅 기법들 중 Timing Based Detections 기법과 Exception Based Detections 기법을 제외한 나머지 4가지 기법들에 해당되는 총 17개의 안티 디버깅 세부 기술들에 대해서는 모두 정상적으로 탐지가 이루어졌다.

반면에 Timing Based Detections 기법과 Exception Based Detections 기법에 해당되는 총 8개의 안티 디버깅 세부 기술들에 대해서는 4가지의 세부 기술들에 대해서만 정상적으로 탐지가 이루어졌는데, 탐지가 제대로 이루어지지 않은 기술의 경우 본 논문에서 정상 실행일 경우에 해당되는 실행 명령어 트레이스를 추출하기 위한 도구로 사용한 Pin tool이 일부 안티 디버깅 기법의 탐지 방법 특성상 디버거와 동일하게 인식되어 정상 실행일 경우에 해당되는 트레이스를 추출하는 부분에 대한 문제 발생으로 인하여 탐지를 하지 못하였다.

4.2 안티 디버깅 기술이 적용된 악성코드 대상 탐지 실험

아래의 표 2는 안티 디버깅 기술이 적용된 43개의 실제 악성코드 샘플들을 대상으로, 본 논문에서 제안하고자 하는 방법을 사용하여 안티 디버깅 루틴에 대한 탐지 실험을 실시한 결과이다.

(표 2) 악성코드 탐지 실험 결과
(Table 2) Experiment results on detection in malware

안티 디버깅 기술	파일 명	탐지
API Based Anti-debugging	5c335abd297f7ac3a8f90c5df58ba3be	O
	ce7d0685ac67df0816567ae7692d15c7	O
	Worm.Win32.Donk.a	O
	Worm.Win32.Donk.b	O
	Worm.Win32.Donk.c	O
	Trojan.Win32.Agent.ee	O
	Trojan.Win32.Agent.eb	X
	Worm.Win32.Feebs.af	O
	Worm.Win32.Feebs.ah	O
	Worm.Win32.Feebs.az	O
	Worm.Win32.Feebs.bf	O
	Worm.Win32.Feebs.bg	O
	Worm.Win32.Feebs.bz	O
	Worm.Win32.Feebs.ca	O
	Worm.Win32.Feebs.cl	O
	Worm.Win32.Feebs.d	O
	Worm.Win32.Feebs.ds	O
	Worm.Win32.Feebs.du	O
	Worm.Win32.Feebs.eu	O

안티 디버깅 기술	파일 명	탐지
	Worm.Win32.Feebs.gg	O
	Worm.Win32.Feebs.hr	O
	Worm.Win32.Feebs.l	O
	Worm.Win32.Feebs.n	O
	Worm.Win32.Feebs.v	O
	Worm.Win32.Feebs.w	O
	Worm.Win32.Feebs.z	O
	Worm.Win32.Lemoor.a	O
	Worm.Win32.Lemoor.b	O
	Worm.Win32.Lemoor.c	O
	Trojan.Win32.Agent.kz	O
	Trojan.Win32.Agent.sl	O
	Trojan.Win32.Agent.vm	O
	Trojan.Win32.Dialer.dh	O
	Trojan.Win32.Dialer.ji	O
	Trojan.Win32.Dialer.le	O
	Trojan.Generic.444278	O
	34269a9fc285af11f82a87d35ecb24c3	O
Direct Process and Thread Block Detections	27c0a658c8129159d3e09140c528a4af	O
	a4ef11b40ca6db077f30c9e2b9ca4c6a	O
	d0141d592741e8ecf5ee1777dd7ff5de	O
Timing Based Detections	588573dc336b3695e9fdb890eefd26db	O
Exception Based Detections	e58a15f0c8044f09cbf592128cc6f39a	O
	97546cff895f23f49e071dbd38e0ceeb	O
	588573dc336b3695e9fdb890eefd26db	O
	Worm.Win32.Donk.b	O
	Worm.Win32.Donk.c	O

본 절에서는, 표 1의 안티 디버깅 기법들 중 Hardware and Register Based Detection 기법과 Modified Code Detections 기법이 적용된 실제 악성코드 샘플들은 구하지 못하여 실험을 실시하지 못하였고, 이를 제외한 나머지 안티 디버깅 기법들이 적용된 악성코드 샘플들을 대상으로 본 논문에서 제안하는 기법에 따라 탐지 실험을 실시하였다.

그 결과, 위의 표 2와 같이 총 43개의 악성코드 샘플들 중 API Based Anti-debugging 기법에 해당되는 1개의 악성코드 샘플을 제외한 나머지 42개의 악성코드 샘플들에 대해서는 정상적으로 탐지가 이루어졌다.

탐지가 제대로 이루어지지 않은 1개의 악성코드 샘플의 경우, 디버거를 통한 실행 환경에서 안티 디버깅 루틴 이후에 정상적인 실행 환경일 경우와 서로 다른 실행 분기는 발생되었지만, 해당 지점 이후로 같은 API들이 호출되는 것이 발견되었고 이러한 이유로 인하여 탐지가 이루어지지 않았다.

해당 악성코드의 경우 안티 디버깅 루틴 이후에 바로 해당 검사 결과에 따라서 프로그램의 행위를 결정하지 않고 추가적인 실행 조건들을 체크한 뒤 최종적으로 행위를 결정하는 실행 구조를 가진 것으로 예상이 되며, 이와 같은 실행 구조를 가진 1개의 악성코드를 제외한 나머지 악성코드들에 대해서는 정상적으로 탐지가 이루어졌다.

이와 같은 결과를 통해서 본 논문에서의 False Negative Rate와 True Positive Rate를 구해보면, 우선 False Negative는 안티 디버깅 기술이 적용되어 있는 악성코드들을 대상으로, 안티 디버깅 기술이 적용되어 있지 않다고 판단된 케이스가 발견되는 경우를 의미하며, 이러한 False Negative의 비율을 의미하는 FNR(False Negative Rate)은 0.023이다.

다음으로 True Positive는 안티 디버깅 기술이 적용되어 있는 악성코드들을 대상으로, 안티 디버깅 기술에 대한 탐지가 정확하게 이루어진 케이스가 발견되는 경우를 의미하며, 이러한 True Positive의 비율을 의미하는 TPR(True Positive Rate)은 0.977이다.

5. 결론 및 향후 연구

본 논문에서는 안티 디버깅 기술이 적용된 악성코드를 대상으로, 정상적인 흐름일 경우와 디버거의 탐지가 이루어질 경우에 대한 흐름 비교를 위하여 각각의 경우에 따른 명령어 및 API 트레이스 결과를 추출하고, 이를 토대로 실행 흐름의 비교 분석을 실시하여, 디버거의 탐지를 통해 분기가 발생되는 지점을 자동으로 탐지해내는 방안에 대하여 기술을 하였다.

그리고 이러한 제안 방식을 사용하여 탐지 실험을 실시한 결과, 알려진 25개의 안티 디버깅 기술들 중 21개의 기술들에 대해서는 정상적으로 탐지가 이루어졌으며, 안티 디버깅 기술이 적용된 43개의 실제 악성코드 샘플들을 대상으로 한 탐지 실험에서는 42개의 악성코드 샘플

들에 대하여 정상적으로 탐지가 이루어졌다.

이러한 연구 결과를 바탕으로 조금만 더 세부적인 부분까지 고려를 하여 보완을 해낸다면, 향후 디버거를 탐지해내는 여러 가지 새로운 안티 디버깅 기법들에 대해서도 활용이 가능할 것으로 예상이 된다.

참 고 문 헌 (Reference)

[1] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. Dependable Systems and Networks With FTCS and DCC (DSN 2008), pp 177-186, 2008.

[2] Pin tool. http://www.pintool.org/

[3] OllyDbg. http://www.ollydbg.de/

[4] IDA Pro. https://www.hex-rays.com/products/ida/index.shtml

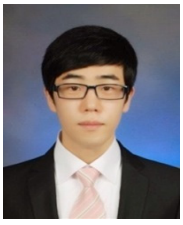
[5] Tyler Shields. Anti-Debugging – A Developers View. Whitepaper, Veracode Inc, 2009.

[6] Peidai Xie, Xicheng Lu, Yongjun Wang, Jinshu Su, and Meijian Li. An Automatic Approach to Detect Anti-debugging in Malware Analysis. International Standard Conference on Trustworthy Computing and Services (ISCTCS 2012), Volume 320, pp 436-442, 2013.

[7] JaeKeun Lee, BooJoong Kang, Eul Gyu Im. Rule-based Anti-anti-debugging System. Proceedings of the 2013 Research in Adaptive and Convergent Systems (RACS 2013), pp 353-354, 2013.

[8] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. Annual Network and Distributed System Security Symposium (NDSS 2010), 2010.

◐ 저 자 소 개 ◑



박 진 우 (Jin-woo Park)
2012년 백석대학교 정보보호학과 졸업(학사)
2012년～현재 한양대학교 대학원 전자컴퓨터통신공학과 석사과정
관심분야 : 바이너리 코드 분석, 악성코드 분석.
E-mail : jinwoo0127@gmail.com

박 용 수 (Yong-su Park)
1996년 KAIST 전산학과 졸업(학사)
1998년 서울대학교 대학원 컴퓨터공학과 졸업(석사)
2003년 서울대학교 대학원 전기컴퓨터공학부 졸업(박사)
2003년～2004년 서울대학교 자동제어특화연구센터 연수연구원
2005년～현재 한양대학교 소프트웨어전공 부교수
관심분야 : 컴퓨터 보안, 인터넷 보안.
E-mail : yongsu@hanyang.ac.kr