논문 19-45-01-13 The Journal of Korean Institute of Communications and Information Sciences '20-01 Vol.45 No.01
https://doi.org/10.7840/kics.2020.45.1.105

# 악성코드 분석을 위한 안티-디버깅의 이해와 무력화 연구를 위한 안티-안티-디버깅 연구

김 종 욱◆, 방 지 원*, 최 미 정°

# Anti-Anti-Debugging Study to Understand and Disable Anti-Debugging for Malware Analysis

Jong-Wouk Kim◆, Jiwon Bang*, Mi-Jung Choi°

요 약

최근 기업, 공무원 등으로 사칭한 이메일 공격, 스피어피싱 공격 등이 기승을 부리고 있다. 공격 기법 중 악성코드가 호스팅 된 사칭 웹사이트로 사용자를 리다이렉트시켜 사용자의 개인 및 금융 정보 등의 입력을 유도하며, 범죄에 악용하는 피싱(Phishing) 기법이 대표적으로 사용되고 있다. 이처럼 다양한 위협을 예방하기 위해 백신 프로그램 개발 업데이트가 꾸준히 이루어지고 있으나, 악성코드를 제작하는 공격자들은 다양한 기술들을 사용하여 분석가의 분석을 방해 및 지연시키고, 악성코드의 수명을 늘리기 위해 다양한 기술들을 적용하는 등 사회공학적 기법과의 결합을 통해 악성코드를 고도화시키고 있다. 악성코드를 보호하는 기술 중 대표적인 기법은 안티-디버깅이 있다. 안티-디버깅은 공격자들이 만든 악성코드를 보호하기 위해 사용된다. 본 연구에서는 위와 같이 악성코드를 은닉하는 패킹(Packing) 기법과 악성코드를 보호하는 안티-디버깅 기법에 관해 설명한다. 또한, 안티 디버깅 기법들을 무력화하는 안티-안티-디버깅 방안을 제안하고 구현한 예제 프로그램들을 적용 및 분석하여 안티-디버깅 동작 여부와 본 연구에서 제안하는 안티-안티-디버깅 방안을 검증했다.

**Key Words** : Anti-Debugging, Malicious Software, Malware, Packing, Packer

ABSTRACT

Recently, e-mail attacks and spear phishing attacks, which purported to be corporations and civil servants, are taking root. Among the attack techniques, a phishing technique for redirecting a user to a spoofed Web site hosted with malicious code to induce input of personal and financial information of the user, and to exploit a crime has been used. Although the development of vaccine programs has been constantly updated to prevent such threats, attackers who make malicious code can use various technologies to interfere with and delay analysis of analysts and apply various techniques to increase the life span of malicious code and combines it with social engineering techniques to enhance malicious code. Anti-debugging is a typical technique used for malicious code concealment. Anti-debugging is used to protect malware it-self. In this paper, we describe the packing method for concealing malicious code and the anti-debugging technique for protecting malicious code as described above. In

※2017년도 강원대학교 대학회계 학술연구조성비로 연구하였음(관리번호-520170089)
※2019년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을받아 수행된 기초연구사업임(No. NRF-2017R1A2B4010205, 표준 정보 모델 기반 NFV 통합 관리 시스템 설계 및 구현)
◆ First Author : Kangwon National University Department of Computer Science, goldbear564@kangwon.ac.kr, 학생회원
° Corresponding Author : Kangwon National University Department of Computer Science, mjchoi@kangwon.ac.kr, 종신회원
* Department of Computer Science, Kangwon National University, jiwonbang@kangwon.ac.kr
논문번호 : 201908-172-C-RU, Received August 27, 2019; Revised October 13, 2019; Accepted October 24, 2019

addition, we proposed anti-anti-debugging schemes to disable anti-debugging techniques and applied and analyzed example programs to verify the anti-debugging behavior and the anti-anti-debugging scheme proposed in this study.

## Ⅰ. 서 론

최근 바이러스, 키로거와 같은 악성코드 기반 소프트웨어로 인해 다양한 방법으로 시스템에 접근하여 사용자의 행위, 개인정보 및 금융 정보뿐만 아니라, 기업, 공공기관의 중요 데이터 탈취, 손상과 같은 치명적인 피해가 지속해서 발생하고 있다. 악성코드로 인한 피해는 사용자의 금융 정보와 같은 중요한 데이터의 손실에서부터 스턱스넷과 같은 산업시설을 감시하고 파괴하는 심각한 문제에 이르기까지 다양한 형태가 될 수 있다. Symantec 社의 'Internet Security Threat Report 2019' 보고서에 따르면 2018년도에 확산된 많은 악성코드 중 금융 트로이목마는 16%나 차지했다고 한다[1]. 또한, Trend Micro 社의 'Unseen Threats, Imminent Losses' 보고서에는 산업시스템을 구성하는 요소 중 하나인 Supervisory Control And Data Acquisition(SCADA) 시스템에 대한 취약점이 발견되었으며, 공격자들은 취약점을 공격하는 악성코드를 통해 시스템에 침투하고 특정 운영에 대한 진단 데이터를 수집 및 SCADA 시스템을 제어할 수 있다고 보고하였다[2]. 두 보고서에 따르면 악성코드로 인한 피해는 여전히 이루어지고 있으며 이에 대한 대책이 필요함을 알 수 있다.

보안 전문가들은 이러한 악성코드로 인한 피해를 최소화하기 위해 백신 프로그램, 보안 장비의 배치와 같은 다양한 방법을 통해 보안을 강화하고, 분석가들은 디버거라는 소프트웨어를 사용하여 트로이목마, 취약점을 공격하는 악성코드, 웜, 백도어, 키로거와 같은 악성 소프트웨어를 신속하게 분석하고 확산되지 않도록 분석하는 등의 큰 노력을 기울이고 있다[3]. 하지만, 악성코드를 배포 및 제작하는 해커들도 분석을 방해하고 악성코드의 수명을 늘리기 위해 안티-디버깅, 패킹과 같은 Anti-Analysis 기술 등을 활용하고 있어 악성코드 분석에 어려움을 초래한다. 상당수의 악성코드가 안티-디버깅 기술을 적재하고 있으며, 결과적으로 악성코드의 분석 및 대응 속도가 늦어진다[4]. 예를 들어, 분석 대상인 디버기(Debuggee), 즉 디버깅 중인 악성코드에 안티-디버깅 기술들이 적용되어 있다면, 백신 프로그램 또는 자동화 분석 도구로는 탐지하지 못할 가능성이 더욱 커진다. 또한, 디버거를 이용한 분석에도 많은 시간이 소모되어 악성코드로 인한 피해가 증가한다.

따라서, 본 연구에서는 윈도우즈 x86 환경에서 디버거를 사용한 악성코드를 분석할 때 문제가 되는 안티-디버깅 무력화를 연구하였다. 먼저 의심스러운 API 또는 구조체의 주소를 찾고, 구조체가 가지는 데이터를 수정하거나 API의 명령어를 수정하여 안티-디버깅을 무력화하는 안티-안티-디버깅 기술을 소개한다. 이 연구를 통해 악성코드를 분석에 소모되는 시간을 줄이고, 악성코드로 인한 피해 범위를 줄일 수 있다. 실험을 통해 본 논문에서 제안한 안티-안티-디버깅을 검증하였다.

본 연구에서는 디버거의 취약점을 공격하여 출력 및 입력 장치의 신호를 무시하거나 디버거를 종료시키는 방법과 같은 Anti-Analysis 기술 중 대표적으로 사용하는 안티-디버깅 기술들을 소개하고, 안티-디버깅 기술이 어떻게 분석을 우회하는지 설명한다. 이런 안티-디버깅 기술을 무력화하는 안티-안티-디버깅 방법을 제안하고, 제안한 방법을 예제 프로그램들을 통해 검증하고자 한다.

## Ⅱ. 관련 연구

### 2.1 Portable Executable(PE)

PE 파일은 Common Object File Format(COFF) 포맷을 변화시킨 파일 포맷이다. PE 파일을 실행시키면 하나의 독립적인 프로세스가 생성되며, 플랫폼에 상관없이 Win32 기반 시스템일 경우, 어디에서든 실행이 가능한 프로그램을 의미한다. 많은 악성코드가 PE 파일 기반으로 생성되어 네트워크를 통해 유포되고 있다. PE 파일의 종류로는 실행, 드라이버, 라이브러리, 오브젝트 총 4개의 계열이 존재한다. 실행 계열의 파일로는 EXE, SCR 파일이 있으며, 드라이버 계열은 SYS, VXD 파일이 있고, 라이브러리 계열은 DLL, DRV 등이 있다. 마지막으로 오브젝트 계열은 OBJ 파일이 있으며, 표 1은 PE 파일의 종류를 나열한 것이다.

파일을 실행시키기 위해서 운영체제는 실행 파일의 정보를 알고 있어야 하므로, 운영체제는 필요한 정보를 얻기 위해 맨 앞에 존재하는 PE 헤더 구조체에서

표 1. PE 파일 종류
Table 1. Types of PE file.

| Type | File Extension |
|---|---|
| Executable File | EXE, SCR |
| Driver File | SYS, VXD |
| Library File | DLL, DRV, OCX, CPL |
| Object File | OBJ |

읽는다. PE 파일은 'MZ(0x4D5A)'로 시작하는 IMAGE_DOS_HEADER 구조체와 도스와의 호환을 위한 코드를 담고 있는 Dos Stub으로 구성되어 있다. IMAGE_NT_HEADERS 구조체로 정의된 헤더는 'PE '(0x50450000)로 시작하여 IMAGE_FILE_HEADER와 IMAGE_OPTIONAL_HEADER 구조체로 구성된다. IMAGE_FILE_HEADER 구조체는 PE 파일 자체에 대한 정보를 나타내며 IMAGE_OPTIONAL_HEADER 구조체는 PE 파일이 실행되어 메모리에 로드될 때 필요한 정보들을 담고 있는데, 프로그램이 사용하는 라이브러리와 함수명, 함수 시작 주소 등에 대한 정보가 기술된 테이블인 Import Address Table(IAT)에서 확인할 수 있다. 이를 통해 안티-디버깅 함수의 존재 여부를 판단할 수 있으며 안티-디버깅 함수들을 수정할 때 참조할 필요가 있는 테이블이다.

다음으로 IMAGE_SECTION_HEADER 구조체들이 등장하는데, 섹션 헤더가 가리키는 각각의 섹션은 그 특성에 맞는 자체 포맷을 갖는다. 섹션 헤더에서 확인할 수 있는 정보들은 각 섹션의 이름, 메모리에 로드되었을 때 해당 섹션이 시작하는 주소, 속성을 나타내는 플래그 등을 확인할 수 있으며, IMAGE_SECTION_HEADER 구조체 다음으로 코드나 데이터들을 가지고 있는 섹션들의 배열이 등장한다.

### 2.2 패킹(Packing)

패킹이란 "Executable Compression"라고도 불리며, 실행 파일의 크기를 줄여주는 압축 기술 중 하나로, 실행 파일의 형태를 유지함과 동시에 압축하여 파일의 크기를 줄여 저장 공간을 확보하기 위해 개발된 기술이었으나, 악성코드 제작자들은 악성코드의 은닉을 위해 사용하였다[5]. WildList 社의 2006년 보고서에 따르면 92% 이상의 악성코드에 패킹 기술이 적용되어 있다는 것을 알 수 있다[6]. 패킹된 파일은 기존의 코드를 변형시킬 뿐만 아니라 안티-디버깅 기술을 새로 추가할 수 있으며 패킹된 악성코드를 분석하기 위해서는 반드시 패킹을 해제하는 언패킹 작업을 수행해야 한다. 패킹 여부를 판단하기 위해서는 Choi는 PE 파일의 헤더를 분석하여 패킹 여부를 판단하는 PHAD를 제안하였다. PHAD는 파일의 PE 헤더의 특징을 선정할 때 일반 파일과 패킹된 파일의 변수를 휴리스틱 분석을 통해 8개를 선정하고, 이를 기반으로 패킹 여부를 탐지한다[7]. Jeong은 패킹된 악성코드를 언패킹 하는 것은 압축 알고리즘을 모르더라도 동적으로 가능하며, 동적으로 언패킹 하는 과정에서 엔트로피 값의 변화량을 측정하여 압축 알고리즘을 다수의 클러스터로 분류하는 방법을 제안하였다[8].

### 2.3 안티-디버깅

디버깅(Debugging)이란 프로세스가 예기치 않게 동작하거나 충돌하여 발생하는 에러, 버그 및 오류를 감지하여 제거하는 과정을 의미하며 또한, 디버깅은 악성코드를 동적으로 분석하는 과정에서도 사용된다[9]. 안티-디버깅 기술은 디버깅을 방지하고 분석을 하지 못하도록 하는 기술로, 컴파일된 코드의 지적 재산권을 보호하기 위해 개발되었지만, 악성코드 제작자들은 악성코드의 수명을 늘리기 위해 이 기술을 악용한다. 분석가들은 디버깅을 수행할 때 Ollydbg, IDA Pro와 같은 도구를 사용하여 분석하지만, 디버거는 Self-Modifying, 키보드 및 마우스 같은 입력 장치의 이벤트를 차단하는 기술에 매우 취약하므로 안티-디버깅은 디버깅 여부를 판단 및 분석을 지연시킨다[10,11].

안티-디버깅에 관한 연구는 이전부터 지속적으로 이루어지고 있다. Tyler는 안티-디버깅 기술을 사용하여 프로그램을 개발하는 방법에 관한 연구를 소개하였으며, Branco는 안티-디버깅, Anti-Virtual Machine, 난독화 기술 등 악성코드가 사용하는 분석 방해 기술에 대한 개요를 제시하였다[12-13]. Shang은 윈도우즈의 디버깅 메커니즘에 대한 개요와 디버거가 사용하는 디버깅 방법에 따른 분류와 디버깅 함수 및 예외를 사용하는 안티-디버깅에 관하여 설명한다[14]. Ping Chen은 일반적인 악성코드와 특정 대상 시스템을 공격하기 위해 개발된 악성코드를 구분하여 얼마나 많은 안티-디버깅 기술이 사용되는지 조사하였다[4]. 대부분의 악성코드에 안티-디버깅 기술이 적용되었지만, 일반적인 악성코드가 대상 악성코드보다 더 많은 안티-디버깅 기술이 사용되었음을 연구하였다.

Lee는 Anti-Debugging Rule-Set을 정하여 분석 대상에서 부합되는 부분을 다른 명령어로 치환하는 방법을 제안하였으며 디버기에서 안티-디버깅 함수를

호출하는 부분을 직접 수정한다. Hao는 Apate라는 체제를 제안하여 안티-디버깅 기술을 17개의 카테고리로 나누어 무력화하는 연구를 진행하였다[15-17]. Xu Chen의 연구는 악성코드가 일반적으로 사용하는 역공학 방지 기법을 분류하고, 탐지 및 우회하는 방법과 실제 시스템을 보호하기 위한 새로운 체계를 제안하였다[18]. Smith는 바이너리 코드에서 안티-디버깅 기술을 정적으로 분석하기 위한 도구인 REDIR을 개발하였으며[19], 이때 난독화된 코드를 분석하기 위해 BAP(Binary Analysis Platform)를 사용하였다[20].

## Ⅲ. 안티-디버깅 기술 소개 및 우회 방법

디버거의 분석이 시작되면 디버기와 상호작용이 가능하도록 운영체제에 의해 디버기의 환경이 변경된다. 안티-디버깅 기술은 변경된 환경 데이터를 참조하여 디버깅 여부를 판단할 수 있고, 디버깅 중이라면 분석을 종료시키거나 악성 행위의 실행하지 않는다. 이와 같은 분석 방해 기법을 해결하기 위해 본 장에서는 윈도우즈에서 제공하는 함수를 사용하여 디버깅 여부를 판단하는 방법, 변경된 환경 데이터를 탐지하여 디버깅 여부를 판단하는 방법 등의 안티-디버깅 기술에 관해 기술하고, 이를 무력화하는 방법을 설명한다[21].

### 3.1Process Environment Block(PEB) 구조체

Process Environment Block(PEB) 구조체는 프로세스마다 할당되어 있으며 프로세스의 정보를 담고 있는 구조체이다. 악성코드는 함수를 사용하지 않고, 디버깅 여부를 판단할 수 있는데, 이때 사용되는 것이 PEB 구조체이다. 뿐만아니라, PECompact, ASPack, ASProtect와 같은 많은 패커들 또한 사용한다[22-24].

PEB 구조체의 많은 멤버들 중 BeingDebugged, HEAP 구조체 포인터, NtGlobalFlag 멤버를 확인하여 안티-디버깅 기능을 수행할 수 있다. BeingDebugged 멤버는 디버깅 중이 아니라면 0x0 값으로 설정되며 디버깅 중에 0x1 값으로 설정된다. NtGlobalFlag 멤버는 디버깅 중이 아닐 때 0x0 값으로 설정되지만, 디버깅 중일 때에는 0x70 값으로 설정되며 이 값의 구성은 표 2 같다[25]. 윈도우즈 XP의 경우 LDR 멤버가 사용된다. LDR 멤버는 _PEB_LDR_DATA 구조체를 가리키는 포인터로 이 구조체는 힙 메모리 영역에 생성된다. 이 구조체의 특징은 디버깅 시, 사용하지 않는 영역은 0xFEEEFEEE 또는 0xABABABAB으로 채워지는데, 이를 이용하여 안티-디버깅이 가능하다. 하지만, 윈도우즈 Vista 이후 이러한 특징이 사라졌다.

표 2. PEB 구조체 안에 있는 NtGlobalFlag 멤버의 Flag 구성
Table 2. Flag configuration of NtGlobalFlag member in PEB structure.

| Flag | Value |
|---|---|
| FLG_HEAPENABLE_TAIL_CHECK | 0x10 |
| FLG_HEAP_ENABLE_FREE_CHECK | 0x20 |
| FLG_HEAP_VALIDATE_PARAMETER | 0x40 |

또한, PEB 구조체 멤버 중 HEAP 구조체를 가리키는 멤버가 존재하는데, HEAP 구조체를 이용하여 디버깅 여부를 판단할 수 있다. 안티-디버깅에 사용되는 HEAP 구체의 멤버는 Flags, ForceFlags 멤버가 있으며, Flags 멤버는 디버깅 중이 아니라면 0x2 값을 가지며, 디버깅 중이라면 윈도우즈 XP에서는 0x50000062, 윈도우즈 7에서는 0x40000062 값을 가진다. ForceFlags 멤버는 디버깅 중이 아니라면 0x0으로, 디버깅 중이라면 0x40000060 값을 가진다.

표 3, 4는 HEAP 구조체의 Flags, ForceFlags의 구성을 각각 나타내고 있다. PEB 및 HEAP 구조체를 사용한 안티-디버깅을 우회하기 위해서는 분석을 시작하기 전에 구조체가 가지는 값을 변경하여 우회할 수 있다.

표 3. HEAP 구조체 안에 있는 Flags 멤버 구성
Table 3. Flag configuration of Flags member in HEAP structure.

| Flag | Value |
|---|---|
| HEAP_GROWABLE | 0x2 |
| HEAP_TAIL_CHECKING_ENABLED | 0x20 |
| HEAP_FREE_CHECKING_ENABLED | 0x40 |
| HEAP_SKIP_VALIDATION_CHECKS (only in Windows XP) | 0x10000000 |
| HEAP_VALIDATE_PARAMETERS_ENABLED | 0x40000000 |

표 4. PEB 구조체 안에 있는 NtGlobalFlag 멤버의 Flag 구성
Table 4. Flag configuration of NtGlobalFlag member in PEB structure.

| Flag | Value |
|---|---|
| HEAP_TAIL_CHECKING_ENABLED | 0x20 |
| HEAP_FREE_CHECKING_ENABLED | 0x40 |
| HEAP_VALIDATE_PARAMETERS_ENABLED | 0x40000000 |

### 3.2 IsDebuggerPresent() 함수

IsDebuggerPresent() 함수는 소프트웨어가 가장 쉽게 디버거의 존재를 탐지할 수 있는 함수로써, 많은 악성코드 및 일반 소프트웨어에서 사용된다. 또한, 이 함수는 UPX, PECompact와 같은 대부분의 패커에 포함된 안티-디버깅 함수이다. 이 함수는 PEB 구조체의 Being- Debugged 멤버를 참조하여 디버거가 탐지될 경우 True를 반환한다[26-28].

그림 1의 좌측 기계어 코드는 함수의 원형으로, PEB 구조체의 BeingDebugged 멤버를 확인하여 반환한다. 이 함수를 우회하기 위해서는 항상 0x0으로 반환하도록 코드를 수정하여 우회하거나, PEB 구조체가 가지고 있는 값을 변경하여 우회할 수 있다. 본 연구에서는 SUB 명령어를 이용하여 EAX 레지스터에 0x0을 저장하게 만들어 무력화를 하였다.

```
MOV EAX,DWORD PTR FS:[18]      MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+30]  SUB EAX,EAX
MOVZX EAX,BYTE PTR DS:[EAX+2]  RETN
RETN
```

그림 1. IsDebuggerPresent() 함수의 원본 코드(좌)와 수정 코드(우)
Fig. 1. The original code(left) and the modification code(right) of the IsDebuggerPresent() API.

### 3.3 CheckRemoteDebuggerPresent() 함수

본 절의 함수는 윈도우즈 NT 버전 이상에서 사용이 가능한 함수이며, 특정 프로세스의 Process Identifier(PID)를 전달하면 해당하는 프로세스의 디버깅 여부를 판단하는 함수이다. 이 함수는 내부적으로 NtQueryInformationProcess() 함수를 호출하여 안티-디버깅이 이루어지며, 악성코드 및 일반 소프트웨어뿐 아니라 몇몇 패커들이 사용하여 디버깅을 방지하기도 한다[29].

그림 2의 좌측 코드는 일부 생략된 함수의 내부 코드로, 이 함수를 우회하기 위해서는 내부에서 호출하는 NtQueryInformationProcess 함수를 우회하거나, 그 전에 EAX 레지스터에 0x0을 저장하여 함수를 반환하여 우회할 수 있다. 본 연구에서는 그림 2의 우측 코드와 같이 PUSH 명령어로 0x0을 스택에 저장하여 POP 명령어로 EAX 레지스터에 0x0을 저장한 후 RETN 명령어로 함수를 종료한다. 이때 중요한 것은 EAX 레지스터에 0x0을 저장한 후, 스택에 저장된 복귀 주소를 POP 명령어로 EBP 레지스터에 저장한 뒤, 함수를 종료하는 것이 중요하다.

```
MOV EDI,EDI                    MOV EDI,EDI
PUSH EBP                       PUSH EBP
MOV EBP,ESP                    MOV EBP,ESP
CMP DWORD PTR SS:[EBP+8],0     MOV EAX,DWORD PTR SS:[EBP+C]
PUSH ESI                       PUSH 0
JE SHORT kernel32.76C13FC1     POP DWORD PTR DS:[EAX]
MOV ESI,DWORD PTR SS:[EBP+C]   XOR EAX,EAX
TEST ESI,ESI                   POP EBP
JE SHORT kernel32.76C13FC1     RETN 8
...                            ...
POP ESI                        POP ESI
POP EBP                        POP EBP
RETN 8                         RETN 8
```

그림 2. CheckRemoteDebuggerPresent() 함수의 원본 코드(좌)와 수정된 코드(우)
Fig. 2. The original code(left) and the modification code(right) of the CheckRemoteDebuggerPresent() API.

### 3.4 Zw(Nt)QueryInformationProcess() 함수

본 절의 함수는 안티-디버깅 외, 프로세스의 경로를 찾아내는 등의 다양한 이유로 사용되는 함수이다. 그림 3은 ZwQueryInformationProcess() 함수에 대한 파라미터로, 이 중 ProcessInformationClass 파라미터는 검색할 프로세스의 정보 중 어떠한 정보를 수집할지에 대한 정보를 나타내는 값이다. 이 파라미터에 ProcessDebugPort를 의미하는 0x7 값을 설정하여 호출하면 디버깅 여부를 알 수 있는데[25], 디버깅 중이라면 0xFFFFFFFF(-1)이 반환되고, 디버깅이 이루어지지 않고 있다면 0x0이 반환된다. 이를 사용자 레벨의 디버거에서 우회하기 위해서는 가상 메모리에 추가적인 코드를 작성하여 우회할 수 있다.

그림 4의 좌측 코드는 NtQueryInformation-Process() 함수의 원본 기계어 코드이며, 우측 코드는 이 함수를 우회하기 위해 가상 메모리에 새로이 작성한 코드를 나타낸다. 이 함수를 우회하기 위해 원본 코드의 2번째 코드까지는 반드시 수행되어야 하고, 0x0을 반환하도록 유도하여야 한다. 만약,

```
NTSTATUS WINAPI ZwQueryInformationProcess(
    _In_      HANDLE                GetProcessHandle,
    _In_      PROCESSINFOCLASS      ProcessInformationClass,
    _Out_     PVOID                 ProcessInformation,
    _In_      ULONG                 ProcessInformationLength,
    _Out_opt_ PULONG                ReturnLength
);
```

그림 3. ZwQueryInformationProcess() 함수의 파라미터
Fig. 3. Parameter of ZwQueryInformationProcess() API.

```
MOV EAX,0EA                    MOV EAX,0EA
MOV EDX,7FFE0300               MOV EDX,7FFE0300
CALL DWORD PTR DS:[EDX]        CMP DWORD PTR SS:[ESP+8],7
RETN 14                        JE SHORT 003F001D
                               PUSH 775E6052
                               RETN
                               MOV EAX,DWORD PTR SS:[ESP+C]
                               PUSH 0
                               POP DWORD PTR DS:[EAX]
                               XOR EAX,EAX
                               RETN 14
```

그림 4. ZwQueryInformationProcess() 함수의 원본 코드(좌)와 수정된 코드(우)
Fig. 4. The original code(left) and the modification code(right) of the ZwQueryInformationProcess() API.

ProcessInformationClass가 0x7(ProcessDebugPort)을 가진다면 0x0을 PUSH 명령어로 스택에 저장하고, POP 명령어로 스택에 저장한 데이터를 EAX 레지스터에 가져와 반환한다면 본 절의 함수를 우회할 수 있다.

### 3.5 FindWindowW(), FindWindowA() 함수

FindWindow 계열 함수들은 특정 윈도우의 이름을 찾아내는 함수이다. 보통 윈도우의 이름을 통해 디버거가 동작 중인지를 확인하는 안티-디버깅 방법이다. 그림 5는 FindWindowA()와 FindWindowW() 함수의 파라미터를 나타낸 것으로, lpClassName은 특정 클래스 이름을, lpWindowName은 특정 윈도우 이름을 전달하여 함수를 호출한다. 만약 해당 윈도우를 찾았을 경우 윈도우 핸들을 반환하며 찾지 못했을 경우 0x0을 반환하게 된다. 이 함수를 호출할 때 lpWindowName 파라미터에 특정 디버거의 윈도우 이름을 넘겨주어 안티-디버깅이 가능하다.

```
HWND FindWindowA(
  LPCSTR lpClassName,
  LPCSTR lpWindowName
);
```

```
HWND FindWindowW(
  LPCWSTR lpClassName,
  LPCWSTR lpWindowName
);
```

그림 5. FindWindowA(좌)와 FindWindowW(우)의 파라미터
Fig. 5. Parameters of FindWindowA(left) and FindWindowW(right).

```
MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
XOR EAX,EAX
PUSH EAX
PUSH DWORD PTR SS:[EBP+C]
PUSH DWORD PTR SS:[EBP+8]
PUSH EAX
PUSH EAX
CALL USER32.752AB865
POP EBP
RETN 8
```

그림 6. (a) FindWindowW 함수의 원본 코드
Fig. 6. (a) Original code of the FindWindowW().

```
MOV EDI,EDI
PUSH EBP
MOV EBP,ESP
XOR EAX,EAX
PUSH EAX
PUSH DWORD PTR SS:[EBP+C]
PUSH DWORD PTR SS:[EBP+8]
PUSH EAX
PUSH EAX
PUSH 520000
RETN
RETN 8
```

```
CALL USER32.752AB865
XOR EAX,EAX
POP EBP
RETN 8
```

그림 6. (b) FindWindow() 함수의 무력화를 위한 수정 코드
Fig. 6. (b) Modification code of FindWindowW() to disable.

이 함수를 우회하기 위해서는 0x0을 반환하게 유도하면 가능하다. 그림 6(a)의 코드는 FindWindowW()의 원본 기계어 코드이며, 그림 6(b)의 코드는 수정된 함수이다. 0x0을 반환하기 위해 본 연구에서는 가상 주소(0x00520000)를 할당하여, 새로운 코드를 작성하여 무력화할 수 있다. 기존의 CALL 명령어를 실행한 후, XOR 명령어를 이용하여 레지스터에 0x0을 저장하고, 복귀 주소를 EBP 레지스터에 POP 명령어를 사용하여 저장한 후 함수를 끝내어 우회할 수 있다.

### 3.6 GetCurrentProcessId(), BlockInp- ut() 함수

본 절의 두 함수는 Yoda's Protector와 같은 상용 패커에서 사용되는 함수이다[30]. Yoda's Protector로 패킹된 경우, 모든 프로세스의 PID를 획득하기 위해 CreateToolHelp32Snapshot() 함수를 호출하여 실행 중인 모든 프로세스의 PID를 포함한 프로세스 정보를 획득하고, 현재 실행된 프로세스를 검색한다. 다음으로, GetCurrentProcessId() 함수를 이용하여 자신을 시작한 프로세스가 자신과 동일한 PID를 갖는지 비교 및 확인한다. 만약 자신을 시작한 프로세스가 자기 자신이 아닌 경우 프로세스를 종료시켜 분석을 방해한다. 따라서, 이와 같은 안티-디버깅 방법을 우회하기 위해서는 자기 자신을 실행시킨 PID와 일치하게 만들어야 한다. 즉, 디버기의 PID와 디버거의 PID가 같게 만듦으로써 이를 우회할 수 있다.

그림 7의 좌측 코드는 GetCurrentProcessId()의 원본 코드이며, 이 함수를 우회하기 위해 본 논문에서는 그림 7의 우측 코드와 같이 수정하였다. 디버거와 디버기의 PID를 일치시키기 위해 디버거의 PID(0x918)를 EAX 레지스터에 저장 및 반환을 통해 우회할 수 있다. 물론, PID는 실행마다 같지 않으므로 디버거의 PID를 확인 후 변경해야 할 필요가 있다.

BlockInput() 함수는 키보드와 마우스 입력 이벤트를 차단하는 함수이다. 이 함수는 분석가의 키보드와 마우스와 같은 입력 장치의 이벤트를 차단시키는 함수이다. 이 함수가 호출된다면 키보드와 마우스가 이

```
MOV EAX,DWORD PTR FS:[18]
MOV EAX,DWORD PTR DS:[EAX+20]
RETN
```

```
MOV EAX,918
NOP
NOP
NOP
RETN
```

그림 7. GetCurrentProcessId() 함수의 원본 코드(좌)와 수정 코드(우)
Fig. 7. The original code(left) and the modification code of the GetCurrentProcessId() API.

표 5. 실험 *(i) Simple-ex*에서 사용된 함수들의 정보
Table 5. Information about functions used in *(i) Simple-ex* experiment.

PEB.BeingDebugged

```
void adbg_BeingDebuggedPEB(void){
    BOOL found=FALSE;
    _asm{
        xor eax,eax
        mov eax, fs:[0x30]
        movzx eax, [eax + 0x02]
        mov found, eax
    }
    if (found)
        printf("Debugger is detected by PEB.BeingDebugged₩n");
    else
        printf("Debugger is not detected by PEB.BeingDebugged₩n");
    printf("  Return Value: 0x%08X₩n", found);
}
```

PEB.NtGlobalFlag

```
void adbg_PEBNtGlobalFlag(void){
    DWORD NtGlobalFlag;
    __asm{
        mov eax, fs:[0x30]
        mov ecx, [eax+0x68]
        mov NtGlobalFlag,ecx
    }
    if(NtGlobalFlag==0x70)
        printf("Debugger is detected by PEB.NtGlobalFlag₩n");
    else
        printf("Debugger is not detected by PEB.NtGlobalFlag₩n");
    printf("  Return Value: 0x%08X₩n", NtGlobalFlag);
}
```

PEB.HEAP.Flags

```
void adbg_PEBHEAPFlags(void){
    DWORD Flags;
    __asm{
        mov eax,fs:[0x30]
        mov eax, dword ptr [eax+0x18]
        mov ecx, dword ptr [eax+0x40]
        mov Flags, ecx
    }
    if(Flags==0x40000062)
        printf("Debugger is detected by PEB.HEAP.Flags₩n");
    else
        printf("Debugger is not detected by PEB.HEAP.Flags₩n");
    printf("  Return Value: 0x%08X₩n", Flags);
}
```

PEB.HEAP.ForceFlags

```
void adbg_PEBHEAPForceFlags(void){
    DWORD forceflags;
    __asm{
        mov eax,fs:[0x30]
        mov eax, dword ptr [eax+0x18]
        mov ecx, dword ptr [eax+0x44]
        mov forceflags, ecx
    }
    if(forceflags==0x40000060)
        printf("Debugger is detected by PEB.HEAP.ForceFlags₩n");
    else
        printf("Debugger is not detected by PEB.HEAP.ForceFlags₩n");
    printf("  Return Value: 0x%08X₩n", forceflags);
}
```

IsDebuggerPresent()

```
void adbg_IsDebuggerPresent(void){
    BOOL found = FALSE;
    found = IsDebuggerPresent();
    if (found)  {
        printf("Debugger is detected by IsDebuggerPresent()₩n");
    }
    else
        printf("Debugger is not detected by IsDebuggerPresent()₩n");
    printf("  Return Value: 0x%08X₩n", found);
}
```

CheckRemoteDebuggerPrenset()

```
void adbg_CheckRemoteDebuggerPresent(void){
    HANDLE hProcess = INVALID_HANDLE_VALUE;
    BOOL found = FALSE;
    hProcess = GetCurrentProcess();
    CheckRemoteDebuggerPresent(hProcess, &found);
    if (found)
        printf("Debugger is detected by CheckRemoteDebuggerPresent()₩n");
    else
        printf("Debugger is not detected by CheckRemoteDebuggerPresent()₩n");
    printf("  Return Value: 0x%08X₩n", found);
}
```

FindWindowW()

```
void adbg_FindWindowW(void){
    LPCWSTR dbgStr[4]={L"ollydbg",L"Immunity Debugger",L"IDA Pro(32-bit)"};
    int i=0; HWND retVal;
    for (;i<3;i++){
        retVal=FindWindow(NULL, dbgStr[i]);
        if(retVal){
            printf("Debugger is detected by FindWindowW()₩n");
            printf("  Return Value: 0x%08X₩n",retVal);
            return;
        }
    }
    printf("Debugger is not detected by FindWindowW()₩n");
    printf("  Return Value: 0x%08X₩n",retVal);
    return;
}
```

FindWindowA()

```
void adbg_FindWindowA(void){
    char *dbgStr[]={"ollydbg","Immunity Debugger","IDA Pro(32-bit)"};
    int i=0; HWND retVal;
    for (;i<3;i++){
        retVal=FindWindowA(NULL, (LPCSTR)dbgStr[i]);
        if(retVal){
            printf("Debugger is detected by FindWindowA()₩n");
            printf("  Return Value: 0x%08X₩n",retVal);
            return;
        }
    }
    printf("Debugger is not detected by FindWindowA()₩n");
    printf("  Return Value: 0x%08X₩n",retVal);
    return;
}
```

ZwQueryInformationProcess()

```
void adbg_ZwQueryInformationProcess_ProcessDebugPort(void){
    DWORD retVal;
    PFZWQUERYINFORMATIONPROCESS pfZwQueryInformationProcess;
    HMODULE h_ntdll=GetModuleHandle(TEXT("ntdll.dll"));
    pfZwQueryInformationProcess=(PFZWQUERYINFORMATIONPROCESS)GetProcAddress(h_ntdll,"ZwQueryInformationProcess");
    pfZwQueryInformationProcess(GetCurrentProcess(),0x7,&retVal,4,0);
    if(retVal==-1)
        printf("Debugger is detected by ZwQueryInformationProcess()₩n");
    else
        printf("Debugger is not detected by ZwQueryInformationProcess()₩n");
    printf("  Return Value: 0x%08X₩n", retVal);
}
```

벤트를 차단해 컴퓨터를 재시작하여야 한다. 그림 8의 좌측 코드는 BlockInput() 함수의 원본 코드이며, 우회하는 방법으로는 그림 8의 우측 코드와 같이 아무 동작도 수행하지 않는 코드인 NOP 명령어들로 수정하여 우회할 수 있다.

## Ⅳ. 실 험

이번 장에서는 제안한 방안의 동작 여부를 세 가지로 나누어 실험한 방법과 실험 결과를 설명한다. 첫 번째 실험을 위해 안티-디버깅 기법들을 포함한 간단한 예제 프로그램을 구현하였으며, 무력화 여부를 정확히 검증하기 위해 PEB 구조체 및 안티-디버깅 함수에 의해 반환되는 값을 출력하도록 구현한 예제 프로그램을 사용하였다.

두 번째로 윈도우즈 7(32bit)의 계산기 프로그램을 패킹한 데이터를 사용하여 패킹된 데이터에 두 번째로 윈도우즈 7(32bit)의 계산기 프로그램을 패킹한 데이터를 사용하여 패킹된 데이터에도 본 논문에서 제안한 방법이 제대로 동작하는지 실험하였다. 마지막으로, 실제 악성코드를 사용하여 본 논문에서 제안한 안티-안티-디버깅 방법이 일반 파일, 패킹된 파일, 실제 바이러스에서의 동작을 검증하였다.

### 4.1 실험 데이터

첫 번째 실험인 *(i) Sample-ex*에서 사용한 데이터는 표 5와 같은 방법으로 구현하였다. 안티-디버깅 기법에서 사용되는 PEB 구조체와 관련된 함수들을 사용하여 구현하였으며, 디버거 탐지 여부에 따라 출력하는 결과가 상이하도록 만들었다. 두 번째 실험인 *(ii) Packed Files-ex*에서 사용한 데이터는 윈도우즈 7(32bit)에서 제공하는 계산기 프로그램을 다섯 가지의 패커로 패킹한 데이터를 사용하였다. 사용된 패커들은 'Yoda's Protector', 'Themida', 'ACProtect', 'Armadilo', 'PECompact'를 사용하여 5개의 패킹 파일을 만들었다. 이때, 사용된 패커들의 공통적인 특징은 안티-디버깅 함수들을 적용하여 소프트웨어를 보

```
MOV EAX,1141                    NOP
MOV EDX,7FFE0300                NOP
CALL DWORD PTR DS:[EDX]         ...
RETN 4                          NOP
                                NOP
                                RETN 4
```

그림 8. BlockInput() 함수의 원본 코드(좌)와 수정 코드(우)
Fig. 8. The original code(left) and modification code of the BlockInput() API.

표 6. 실험 *(iii) Trojan.Agent-ex*에서 사용된 실제 바이러스 정보
Table 6. Information about the actual virus used in *(iii) Trojan.Agent-ex* experiment.

| File Type | Win32 EXE |
|---|---|
| Virus Type | Trojan.Agent |
| MD5 | 0566144fb91019463b0a7ee09748f4fd |
| Anti-Debugging | IsDebuggerPresent() |

호한다는 것이다. 마지막 실험인 *(iii) Trojan.Agent-ex*에서 사용한 데이터의 정보는 표 6과 같다.

### 4.2 실험 결과

실험 *(i) Sample-ex*에서 예제 프로그램을 디버거에서 분석한 결과 그림 9(a)와 같이 예제 프로그램에서 사용된 안티-디버깅 기법들은 모두 디버거를 탐지했다. 그러나, 제안한 방법을 사용하여 다시 분석한 결과, 그림 9(b)와 같이 예제 프로그램에서 사용된 안티-디버깅 기법들을 모두 무력화시켰다. 따라서, 본 논문에서 제안한 안티-안티-디버깅 기법이 성공적으로 작동함을 일차적으로 확인하였다.

다음으로, *(ii) Packed Files-ex*에서 사용된 데이터는 모두 한 개 이상의 안티-디버깅 API를 가지고 있으며, 표 7에서 각각의 패커가 가지고 있는 안티-디버깅 기법을 확인할 수 있다. 표 8은 Yoda's Protetcor로 패킹한 파일이 가진 GetCurrent-ProcessId() 함수에 안티-안티-디버깅 기법을 적용한 결과이다. 이때 0x0ABC는 실험 당시 디버거의 PID이다.

표 9는 ACprotect로 패킹한 파일이 가지는 Zw-QueryInformationProcessId() 함수에 안티-안티-디버깅을 적용한 결과를 나타내었다. 만약 본 논문에서 제안한 안티-안티-디버깅을 적용하지 않고 *(ii) Packed*

표 7. 각각의 패커가 가지고 있는 안티-디버깅 API 목록
Table 7. A list of anti-debugging APIs that each packer has.

| Packer | Anti-Debugging API |
|---|---|
| Themida | ZwQueryInformationProcess() |
| | IsDebuggerPresent() |
| Yoda's Protector | GetCurrentProcessId() |
| | BlockInput() |
| | IsDebuggerPresent() |
| ACprotect | IsDebuggerPresent() |
| | ZwQueryInformationProcess() |
| Armadilo | IsDebuggerPrenset() |
| PECompact | IsDebuggerPresent() |

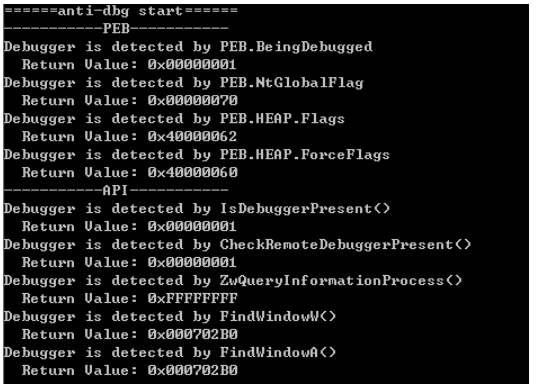


그림 9 (a) 제안한 방법을 적용하지 않고 디버거에서 분석하였을 때, (i) *Sample-ex* 실험 결과
Fig. 9 (a) The result of the (i) *Sample-ex* when analyzed in the user-level debugger without applying the proposed method to avoid anti-debugging technique.

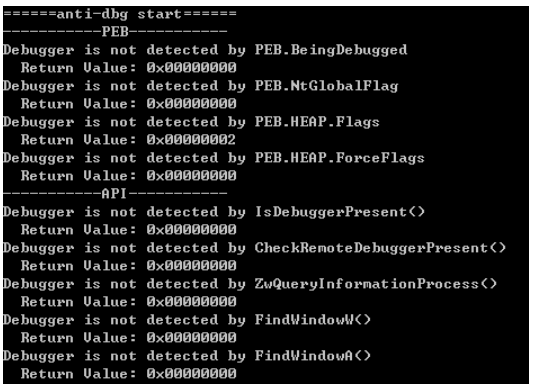


그림 9. (b) 제안한 방법을 적용하여 디버거에서 분석하였을 때, (i) *Sample-ex*의 결과
Fig. 9. (b) The result of the (i) *Sample-ex* when analyzed in the user-level debugger using proposed method to avoid anti-debugging technique.

*Files-ex*를 진행할 경우 프로세스의 흐름이 바뀌어 분석에 지연이 발생한다. 하지만 본 논문에서 제안한 방법을 사용하여 분석을 진행할 경우 문제없이 타겟 프로세스의 분석을 더욱 수월하게 진행할 수 있다.

마지막으로 실제 악성코드를 분석한 *(iii) Trojan.Agent-ex*를 실험한 결과는 표 10과 같다. 이 실험에서 사용된 악성코드의 경우 IsDebugger- Present() 함수 하나만 안티-디버깅을 위해 적재되어 있었다. 따라서, 본 논문에서 제안한 방법과 같이 IsDebuggerPresent() 함수를 표 10과 같이 우회한 결과 악성코드 분석의 시간이 줄어들 수 있었다.

표 8. 실험 *(ii) Packed Files-ex* 중 Yoda's Protector로 패킹한 파일의 GetCurrentProcessId() 함수의 무력화 실험 결과
Table 8. *(ii) Packed Files-ex* experiment result of disabling GetCurrentProcessId() API of pacekd file with Yoda's Protector.

| *Before Patching GetCurrentProcessId() API* | |
|---|---|
| Address | Disassemble Code |
| 751D68D0 | MOV EAX, DWORD PTR FS[18] |
| 751D68D6 | MOV EAX, DWORD PTR DS:[EAX+20] |
| 751D68D9 | RETN |
| *After Patching GetCurrentProcessId() API* | |
| Address | Disassemble Code |
| 751D68D0 | MOV EAX, 0ABC |
| 751D68D6 | NOP |
| 751D68D7 | NOP |
| 751D68D8 | NOP |
| 751D68D9 | RETN |

표 9. 실험 *(ii) Packed Files-ex* 중 ACprotect로 패킹한 파일의 ZwQueryInformationProcess() 함수의 무력화 실험 결과
Table 9. *(ii) Packed Files-ex* experiment result of disabling ZwQueryInformationProcess() API of pacekd file with ACprotect.

| *Before Patching ZwQueryInformationProcess() API* | |
|---|---|
| Address | Disassemble Code |
| 770E6048 | MOV EAX, 0EA |
| 770E604D | MOV EDX, 7FFE0300 |
| 770E6052 | CALL DWORD PTR DS:[EDX] (ntdll.KiFastSystemCall) |
| 770E6054 | RETN 14 |
| *After Patching ZwQueryInformationProcess() API* | |
| Address | Disassemble Code |
| 770E6048 | PUSH 200000 |
| 770E604D | RETN |
| 00200000 | MOV EAX, 0EA |
| 00200005 | MOV EDX, 4FFE0300 |
| 0020000A | CMP DWORD PTR SS:[ESP+8], 7 |
| 00200015 | JE SHORT 0020001D |
| 00200017 | PUSH 770E6052 |
| 0020001C | RETN |
| 0020002B | XOR EAX, EAX |
| 0020002D | RETN 14 |

표 10. 실험 *(iii) Trojan.Agent-ex*에서 사용된 바이러스가 가진 IsDebuggerPresent() 함수의 무력화 실험 결과
Table 10. *(iii) Trojan.Agent-ex* experiment result of disabling IsDebuggerPresent() API of malware has.

| *Before Patching IsDebuggerPresent() API* | |
|---|---|
| Address | Disassemble Code |
| 75121E2E | MOV EAX, DWORD PTR FS:[18] |
| 75121E34 | MOV EAX, DWORD PTR DS:[EAX+30] |
| 75121E37 | MOVZX EAX, BYTE PTR DS:[EAX+2] |
| 75121E3B | RETN |
| *After Patching IsDebuggerPresent() API* | |
| Address | Disassemble Code |
| 75121E2E | MOV EAX, DWORD PTR FS:[18] |
| 75121E35 | SUB EAX, EAX |
| 75121E37 | RETN |

## V. 결론 및 향후 연구

많은 악성코드와 상용 패커들은 스스로를 보호하기 위해 다양한 안티-디버깅 기술들을 채택하여 사용한다. 분석가들은 안티-디버깅 기술이 탑재된 악성코드를 분석하기 위해 진화된 디버거와 관련 전문 지식이 필요하다. 본 연구에서는 사용자 레벨의 디버거에서 안티-디버깅 기술에 대해 무력화가 가능한 방안을 제안하였으며, 실험을 통해 제안하는 방법이 효과적이라는 것을 검증하였다. 또한, 디버기를 직접적으로 수정하지 않고, 디버기가 호출하는 함수를 수정함으로써, 디버기의 수정을 최소화하였다. 안티-디버깅 방법들은 하나의 프로그램 또는 프로세스에서 여러 번 사용될 수 있다. 본 논문에서 제안한 방법을 통해 안티-디버깅의 무력화가 가능하며, 디버기가 여러 번의 안티-디버깅 기법을 사용하더라도, 매번 수정할 필요가 없으며, 프로세스 흐름을 방해하지 않는다는 것을 두 번의 실험을 통해서 확인할 수 있었다.

본 논문에서 기술한 안티-디버깅 기법들은 실제로 사용되고 있는 모든 기술을 망라한 것이 아니라 대표적인 기법을 설명한 것이다. 향후 연구로는 자동으로 안티-디버깅 기법뿐만 아니라 다양한 Anti- Analysis 기술들을 무력화시킴과 동시에 언패킹이 가능한 시스템 개발을 목표로 한다.

## References


[1] *Symantec*, *Internet Security Threat Report 2019*(2019), Retrieved Mar., 3, 2019, from https://www.symantec.com/.

[2] *Trend Micro, Unseen Threats, Imminent Losses*(2018), Retrieved Mar., 3, 2019, from https://www.trendmicro.com.

[3] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Secur. & Privacy*, vol. 5, no. 3, pp. 82-84, May 2007.

[4] P. Chen, C. Huygens, L. Desmet, and W. Joosen, "Advanced or Not? A comparative study of the use of anti-debugging and Anti-VM techniques in generic and targeted malware," in *IFIP Int. Conf. ICT Syst. Secur. and Privacy Protection*, pp. 323-336, Ghent, Belgium, May 2016.

[5] S. Cesare, Y. Xiang, and W. Zhou, "Malwise－an effective and efficient classification system for packed and polymorphic malware," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1193-1206, Jun. 2013.

[6] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Secur. and Privacy*, vol. 6, no. 5, pp. 65-69, Oct. 2008.

[7] Y. S. Choi, I. K. Kim, J. T. Oh, and J. C. Ryou, "PE file header analysis-based packed PE file detection technique (PHAD)," in *Proc. Int. Symp. Comput. Sci. and its Appl. IEEE Comput. Soc.*, pp. 28-31, ACT, Australia, Oct. 2008.

[8] G. Jeong, E. Choo, J. Lee, M. Bat-Erdene, and H. Lee, "Generic unpacking using entropy analysis," *J. Advanced Inf. Technol. and Convergence,* vol. 7, no. 1, pp. 232-238, Feb. 2009.

[9] J. W. Kim, J. W. Bang, and M. J. Choi, "A study on analysis method of malicious code with analysis avoidance technology," in *Proc. KNOM 2018*, pp. 21-24, Jeju Island, Korea, May 2018.

[10] *OllyDbg*, Retrieved Jan., 3, 2019, from www.ollydbg.de/.

[11] *IDAPro*, Retrieved Jan., 12, 2019, from https://

www.hex-rays.com/products/ida/.

[12] S. Tyler, "*Anti-debugging - a developers view*," Veracode Inc., USA, 2010.

[13] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and Anti-VM technologies," in *Proc. Black Hat Conf.*, Las Vegas, USA, Jul. 2012.

[14] S. Gao and Q. Lin, "Debugging classification and anti-debugging strategies," in *Proc. Fourth ICMV 2011*, pp. 729-734, Singapore, Singapore, Jan. 2011.

[15] J. K. Lee, B. J. Kang, and E. G. Im, "Rule-based anti-anti-debugging system," in *Proc. 2013 Res. Adaptive and Convergent Syst.*, pp. 353-354, Monteal, Canada, Oct. 2013.

[16] J. K. Lee, B. J. Kang, and E. G. Im, "Evading anti-debugging techniques with binary substitution," *Int. J. Secur. & its Appl.*, vol. 8, no. 1, pp. 183-192, Jan., 2014.

[17] H. Shi and J. Mirkovic, "Hiding debuggers from malware with apate," in *Proc. Symp. Appl. Comput.*, pp. 1703-1710, Marrakech, Morocco, Apr. 2017.

[18] X. Chen, J. Andersen, Z. M. Mao, M. Bailey and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. IEEE Int. Conf. Dependable Syst. and Netw. with FTCS and DCC(DSN)*, pp. 177-186, Anchorage, USA, Jun. 2008.

[19] A. J. Smith, R. F. Mills, A. R. Bryant, G. L. Peterson, and M. R. Grimailam "REDIR: Automated static detection of obfuscated anti-debugging techniques," in *Proc. 2014 Int. Conf. Collaboration Technol. and Syst.(CTS)*, pp. 173-180, Minnesota, USA, May 2014.

[20] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proc. Int. Conf. Comput. Aided Verification*, pp. 463-469, Utah, USA, Jul. 2011.

[21] J. W. Kim, J. W. Bang, and M. J. Choi, "A study on automatic disabling of anti-debugging in manual unpacking," in *Proc. KNOM 2019*, pp. 58-61, Daegu, Korea, May 2019.

[22] C. V. Liţă, D. Cosovan, and D. Gavriluţ, "Anti-emulation trends in modern packers: a survey on the evolution of anti-emulation techniques in UPA packers," *J. Comput. Virology and Hacking Techniques*, vol. 14, no. 2, pp. 107-126, Feb., 2018.

[23] *ASPack*, Retrieved Feb. 25, 2019, from http://www.aspack.com/.

[24] *ASProtect*, Retrieved Feb. 25, 2019, from http://www.aspack.com/.

[25] P. Ferrie, *The ultimate anti-debugging reference* (2011), Retrieved Mar. 7, 2019, from https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf.

[26] *UPX*, Retrieved Feb. 21, 2019, from https://upx.github.io/.

[27] *PECompact*, Retrieved Feb. 21, 2019, from https://bitsum.com/portfolio/pecompact/.

[28] F. Guo, P. Ferrie, and T. C. Chiueh, "A study of the packer problem and its solutions," in *Proc. Int. Wrksp. Recent Advances in Intrusion Detection*, pp. 98-115, Massachusetts, USA, Sep. 2008.

[29] P. Ferrie, *Anti-unpacker tricks - part one*(2008), Retrieved Mar. 26, 2019, from https://www.gironsec.com/code/unpackers21.pdf.

[30] Microsoft Developer Network, ZwQueryInformationProcess function(2018), Retrieved Feb, 18, 2019, from https://docs.microsoft.com/en-us/windows/desktop/procthread/zwqueryinformationprocess

**김 종 욱** (Jong-Wouk Kim)

2019년 2월 : 강원대학교 컴퓨터과학과 졸업
2019년 3월~현재 : 강원대학교 컴퓨터과학과 석사
<관심분야> 네트워크 보안, 악성코드 분석, 패킹
[ORCID:0000-0002-8180-2376]

**방 지 원** (Jiwon Bang)

2018년 2월 : 강원대학교 컴퓨터과학과 졸업
2018년 3월~현재 : 강원대학교 컴퓨터과학과 석사
<관심분야> 네트워크 관리, 정보 보안
[ORCID:0000-0002-2068-0942]

**최 미 정** (Mi-Jung Choi)

1998년 2월 : 이화여자대학교 공학사
2000년 2월 : 포항공과대학교 공학석사
2004년 2월 : 포항공과대학교 공학박사
2004년~2005년 : 프랑스 INRIA 연구소 박사 후 연구원
2005년~2006년 : 캐나다 워터루대학 박사후 연구원
2006년~2008 : 포항공과대학교 컴퓨터공학과 연구 조교수
2008년~현재 : 강원대학교 컴퓨터학부 컴퓨터과학전공 교수
<관심 분야> 네트워크 관리, 정보보안, 악성코드 언패킹
[ORCID:0000-0002-9062-4604]