한국항행학회 논문지 제16권 제3호 2012년 6월

코드은닉을 이용한 역공학 방지 악성코드 분석방법 연구

A New Analysis Method for Packed Malicious Codes

이경률*, 임강빈*

Kyung-Roul Lee*, Kang-Bin Yim*

요 약

본 논문은 악성코드가 사용하는 자기방어기법을 방식에 따라 분류하고, 악의적인 코드를 보호하는 방법의 일종인 패킹에 대해 소개하였으며, 패킹을 이용하는 악성코드를 보다 빠르게 분석할 수 있는 방안을 제시하였다. 패킹기법은 악의적인 코드를 은닉하고 실행 시에 복원하는 기술로서 패킹된 악성코드를 분석하기 위해서는 복원 후의 진입점을 찾는 것이 필요하다. 기존에는 진입점 수집을 위하여 악성코드의 패킹 관련 코드를 자세히 분석하여야만 했다. 그러나 본 논문에서는 이를 대신하여 악성코드를 생성한 표준 라이브러리 코드 일부를 탐색하는 방법을 제시하였다. 제시한 방안을 실제로 구현하여 보다 신속히 분석할 수 있음을 증명하였다.

Abstract

This paper classifies the self-defense techniques used by the malicious software based on their approaches, introduces the packing technique as one of the code protection methods and proposes a way to quickly analyze the packed malicious codes. Packing technique hides a malicious code and restore it at runtime. To analyze a packed code, it is initially required to find the entry point after restoration. To find the entry point, it has been used reversing the packing routine in which a jump instruction branches to the entry point. However, the reversing takes too much time because the packing routine is usually obfuscated. Instead of reversing the routine, this paper proposes an idea to search some features of the startup code in the standard library used to generate the malicious code. Through an implementation and a consequent empirical study, it is proved that the proposed approach is able to analyze malicious codes faster.

Key words : self-defence techniques, packing, entry point, startup code

I. 서 론

IT 기술이 발전하면서 현대사회는 다양한 서비스를 편리하게 이용할 수 있게 되었다. 하지만 그 기술의 역기능으로 공격자에 의한 피해가 드러나기 시작하였으며, 공격 유형은 과거 바이러스부터 웜, DoS [1], DDoS 등으로 이어져 더욱 다양하고 복잡하게 변화하고 있다 [2]. 공격자에 의해 피해자 컴퓨터에 설치되어 피해를 발생시키는 프로그램을 일반적으로 악성코드라 부른다. 최근 7.7 DDoS [3] 의 공격은 일반 사용자의 PC가 악성코드에 감염되는 것이 매우 일반적으로 발생하고 있음을 증명하고 있어 신속한 악성코드의 발견과 치료가 더욱 중요시되고 있다.

악성코드는 처음에는 단순한 공격을 하는 형태였

* 순천향대학교 정보보호학과(Dept. of Information Security Engineering, Soonchunhyang University)
· 제1저자 (First Author) : 이경률 · 교신저자 (Corresponding Author) : 임강빈
· 투고일자 : 2012년 4월 13일
· 심사(수정)일자 : 2012년 4월 14일 (수정일자 : 2012년 6월 24일)
· 게재일자 : 2012년 6월 30일

기 때문에 역공학을 이용한 연구를 통해 악의적인 코드들을 분석하고 이를 검출하여 시스템을 안전하게 보호할 수 있었다. 이에 대해 공격자는 악성코드가 역공학에 의해 악의적인 코드들이 분석되지 않도록 다양한 기술들을 추가하여 코드를 보호하게 되었는데 이를 자기방어기법이라 불린다. 자기방어기법은 크게 역공학 방지기법과 코드 난독화로 나뉠 수 있는데, 역공학 방지기법은 악성코드가 역공학에 의해 분석되지 않도록 사전에 방해하는 기법이며 코드 난독화는 역공학이 행해지더라도 코드를 분석하기 어렵게 만드는 기법을 말한다. 이러한 기법들은 원래 소프트웨어의 불법복제를 막기 위해 사용되는 것이었으나 악성코드들이 이들 기술을 이용함으로써 부정적인 의미를 가지게 되었다. 자기방어 기법의 기술 중의 하나로 패킹기법이 있는데 패킹이란 악성코드의 노출을 막기 위해 코드 몸채를 숨기고 차후 실행시에 언패커에 의해 이를 복원하는 기술이다. 이와 같은 패킹기법을 활용하는 악성코드는 언패커를 보호하기 위해 역공학 방지기법을 사용하며 이들을 분석하기 위해서는 언패커를 우선적으로 분석해야만 한다. 이는 상기의 역공학 방지기법을 우회해야 함을 의미하며 악성코드 분석에서 시간 소모가 매우 큰 작업 중의 하나이다. 따라서 본 논문에서는 악의적인 코드를 보호하는 패킹된 악성코드를 분석하기 위한 방법을 제시하였다.

본 논문의 구성은 다음과 같다. 제2장에서는 현재 악성코드가 사용하는 자기방어기법의 종류를 분석하고 이들을 분류하였으며, 특히 악성코드가 자기방어를 위해 주로 사용하는 패킹기법에 대해 소개하였다. 제3장에서는 패킹된 악성코드를 분석하기 위한 방안을 제시하였으며, 제4장에서는 제안한 분석 방법을 검증하기 위해 실제 실험결과에 대해 분석하고, 제5장에서 실험결과를 토대로 결론을 도출하였다.

Ⅱ. 악성코드 관련 자기방어기법

2-1 자기방어기법의 종류 및 분류

상기와 같이 악성코드가 사용하는 자기방어기법은 크게 역공학 방지기법과 코드 난독화로 분류된다. 역공학 방지기법은 디버거 감지, 브레이크 포인트와 패칭 감지, 디버거 공격, 패킹, 은닉(루트킷) 등으로 분류되며 [4], 코드 난독화는 배치 난독화, 코드 난독화, 제어 난독화, 방지 난독화로 분류된다 [5]. 이들 기법에 대한 세부적인 기술 분류를 표 1, 표 2에 나타내었다.

표 1. 코드 난독화 세부분류 [5]
Table 1. Classification of code obfuscation.

분류	기술
배치 난독화	형식 변환(Formatting Change)
	주석 제거(Remove Comment)
	식별자 변환(Scramble Identifier)
자료 난독화	저장장소 변환(Data Storage)
	인코딩 변환(Data Encoding)
	순서 변환(Data Ordering)
제어 난독화	계산 변환(Computation Transformation)
	집합 변환(Aggregation Transformation)
	순서 변환(Ordering Transformation)
방지 난독화	역어셈블 방지
	암호화
기타 난독화	가상화 등

분류된 대부분의 기술들은 이미 많은 연구에 의해 악성코드 즉, 악의적인 코드를 분석하기 위한 무력화 방안이 존재한다 [4][6][7][8][9]. 그러나 역공학 방지기법 중 패킹 기술은 방어자가 코드를 분석하지 못하도록 실행 파일의 구조를 변경하여 악의적인 코드를 보호하기도 하며 역공학 방지기법과 혼용되어 사용되기 때문에 현재 수동적, 자동적인 무력화 방안이 존재하더라도 그 종류도 다양하며, 약간의 변형만 되더라도 전혀 다른 분석방법이 필요할 수 있으므로 이를 분석하기 위한 근본적인 방법에 대한 연구가 필요하다.

2-2 악성코드의 코드 패킹 기법

상기에서 분석했듯이 공격자가 악성코드를 보호하기 위한 방법으로 패킹을 이용하곤 하는데, 원래 패킹이란 실행압축을 의미하는 말로 실행파일에서 불필요한 부분을 제거하거나 코드와 데이터를 압축해서 새로운 이미지를 생성하는 것을 의미한다. 이는 과거 실행파일의 크기를 줄여 전송속도 및 저장효율을 높이기 위한 용도로 사용되었지만 추가적으로 코드를 보호하기 위해 자기방어기법과 융합되면서 현

재는 역공학을 이용한 코드 분석을 어렵게 만드는 목적으로 이용되고 있다. 따라서 공격자는 악성코드를 패킹함으로써 악의적인 목적을 가진 코드를 보호하여 분석가로부터 분석을 어렵게 하는 변형된 악성코드를 생산할 수 있다. 이는 방어자가 패킹을 해제하지 못하면 악의적인 코드를 분석할 수 없으므로 악성코드의 생존확률을 향상시킬 수 있어 공격자에게 널리 이용되고 있는 실정이다 [10]. 현재 널리 사용되고 있는 패커는 표 3과 같으며, 알려지지 않은 많은 패커들과 알려진 패커를 수정한 변형 패커 등을 포함한다면 더욱 많은 패커들이 존재할 것으로 판단된다.

표 2. 역공학 방지기법 세부분류
Table 2. Classification of prevention of reverse engineering.

분류	기술
디버거 감지	IsDebuggerPresent()
	IsDebugged
	NtGlobalFlag
	Heap Flags
	DebugPort : CheckRemoteDebuggerPresent()/NTQueryInformationProcess()
	DebuggerInterrupt
	Time Check
	SeDebugPrivilege
	Parent Process
	DebugObject : NtQueryObject()
	Debugger Windows Search
	Debugger Process Search
	OllyDbg : Guard Pages
브레이크포인트와 패칭 감지	Software Breakpoint Detection
	Hardware Breakpont Detection
	Patchong Detection via Code Check Calculation
디버거 공격	Misdirection and Stopping Execution via Exceptions
	Blocking Input
	ThreadHideFromDebugger
	Unhandled Exception Filter
	OllyDbg : OutputDebugString()
기타	패킹
	은닉

패커의 동작방식은 그림 1과 같다. 윈도우즈에서 사용하는 실행파일은 PE 포맷을 가지며, PE 포맷은

표 3. 패커의 종류 및 코드 변형 여부
Table 3. Packer type and code transformation or not.

종류	코드 변형 여부
UPX(3.05w)	X
ASPack(2.2)	X
ASProtect(1.2.3)	X
PECompact(2.0)	X
NSPack	X
FSG(2.0)	X
RLPack(1.21)	X
Themida(2.0.3.0)	O

크게 도스 헤더, NT 헤더, .text 등의 프로그램 실행에 필요한 섹션들로 구성되어 있다. 실행파일을 분석하기 위해서는 프로그램이 실행될 때 최초로 실행되는 주소를 알아야 하는데 이를 EntryPoint라 하며, NT 헤더 내의 AddressOfEntryPoint(EP) 필드에 저장되어 있다. 따라서 EP 필드에 저장된 주소는 .text 내의 특정 번지를 가리키므로 시작주소부터 분석함으로써 전체 프로그램을 분석할 수 있다.

패커는 패킹 후 언패킹 루틴을 실행해야 하기 때문에 원본 .text 섹션을 언패킹을 준비하는 루틴이나 언패킹 루틴 자체를 포함하도록 변경해야 한다. 따라서 패커는 원본 EP 필드를 수정하여 이러한 루틴을 가리키게 함으로써 이를 가능하게 하며, 언패킹 루틴을 모두 수행한 후에는 원래 프로그램의 기능을 수행해야 하므로 원본 EP 필드가 가리키는 주소로 제어를 넘긴다.

패커에 의해 패킹된 프로그램을 분석하기 위해서는 원본 EP 필드가 가리키는 주소를 추출하는 것이 가장 근본적인 해결책이라 할 수 있으며, 이미 많은 연구 결과로 수동적, 자동적으로 추출하는 방법이 알려져 있다. 하지만 패커마다 각기 다른 방법으로 접근해야 하며, 일관적인 방법이 아니므로 패커가 약간의 변형을 한다면 새로운 추출 방안을 연구해야 하기 때문에 매우 소모적이라 할 수 있다.

패커는 크게 원본 코드를 변형하지 않는 패커와 변형하는 패커로 나누어질 수 있으며, Themida를 제외한 패커들은 코드를 변형하지 않는다. 코드를 변형하지 않는 패커는 언패킹 후 원본 코드와 매우 유사하기 때문에 프로그램이 로딩된 메모리에 접근하여 코드를 분석할 수 있다. 코드를 분석하기 위해서는

프로그램의 시작 위치를 알아야 하지만 패킹된 프로그램은 이미 패커에 의해 NT 헤더에 포함된 EP가 언패킹 루틴으로 변경되어 있기 때문에 변경 전의 EP를 찾기 위해서 언패킹 루틴을 정확히 분석해 내는 데에 많은 시간이 소모된다. 따라서 본 논문에서는 패킹된 악성코드를 분석하는 경우 언패킹 루틴을 분석하지 않고서도 효율적으로 변경 전 EP를 추출할 수 있는 방안을 제안하고자 한다.

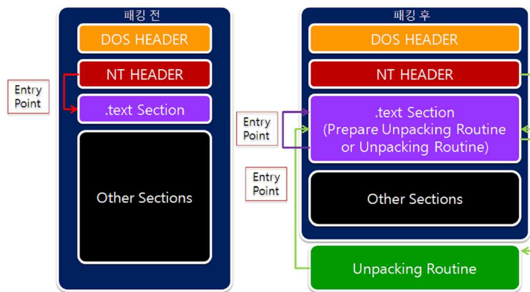

그림 1. 패커의 동작 방식
Figure 1. Operation of a general packer.

III. 패킹된 악성코드 분석 방안

EXE 및 DLL을 포함하는 윈도우 실행파일 구조로서의 PE 포맷의 각 섹션에 대해 살펴보면, 우선 섹션은 크게 .text, .data, .edata, .idata, .reloc, .rsrc 등으로 구성되어 있으며 [11], .text 섹션은 실행 코드가 저장되어 있고, .data 섹션은 전역 데이터의 정보, .edata, .idata는 익스포트/임포트 정보, .reloc 섹션은 재배치 정보, .rsrc 섹션은 리소스 데이터의 정보가 저장되어 있다. EP가 가리키는 주소가 포함된 .text 섹션은 기능에 따라 Startup Code, Code Body, Exit Code로 나뉘며, 이를 그림 2에 나타내었다.

Startup Code는 실제 실행될 코드를 준비하기 위한 기능을 수행하며, Code Body는 프로그램의 코드, Exit Code는 프로그램을 종료하기 위한 기능을 수행한다. 상기 .text 섹션의 구조에서와 같이 EP는 Startup Code의 시작주소를 가리키고 있음을 확인할 수 있다. 그림 3은 Microsoft Visual Studio의 표준 라이브러리가 제공하는 Startup Code의 전반부이며, 그림 4는 디버거를 이용해 실제 프로그램의 EP부터 디스어셈블된 코드의 결과이다.

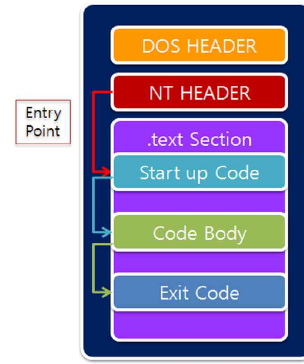

그림 2. .text 섹션 구조
Figure 2. Structure of .text section.

```
109:  void wmainCRTStartup(
110:  #else  /* WPRFLAG */
111:  void mainCRTStartup(
112:  #endif  /* WPRFLAG */
113:
114:  #endif  /* _WINMAIN_ */
115:          void
116:          )
117:
118:  {
00401190   push        ebp
00401191   mov         ebp,esp
00401193   push        0FFh
00401195   push        offset string "The value of ESP was not properl"...+0E0h (004201:
0040119A   push        offset __except_handler3 (00403514)
0040119F   mov         eax,fs:[00000000]
004011A5   push        eax
004011A6   mov         dword ptr fs:[0],esp
004011AD   add         esp,0F0h
004011B0   push        ebx
004011B1   push        esi
004011B2   push        edi
004011B3   mov         dword ptr [ebp-18h],esp
119:          int mainret;
120:
121:  #ifdef _WINMAIN_
122:          _TUCHAR *lpszCommandLine;
123:          STARTUPINFO StartupInfo;
```

그림 3. Microsoft Visual Studio 표준 라이브러리가 제공하는 Startup Code의 전반부
Figure 3. Part of startup code provided by Microsoft Visual Studio's standard library.

```
00401077  $  55             PUSH EBP
00401078  .  8BEC           MOV EBP,ESP
0040107A  .  6A FF          PUSH -1
0040107C  .  68 A8504000    PUSH VC6Conso.004050A8
00401081  .  68 641F4000    PUSH VC6Conso.00401F64
00401086  .  64:A1 000000(  MOV EAX,DWORD PTR FS:[0]
0040108C  .  50             PUSH EAX
0040108D  .  64:8925 0000(  MOV DWORD PTR FS:[0],ESP
00401094  .  83EC 10        SUB ESP,10
00401097  .  53             PUSH EBX
00401098  .  56             PUSH ESI
00401099  .  57             PUSH EDI
0040109A  .  8965 E8        MOV DWORD PTR SS:[EBP-18],ESP
0040109D  .  FF15 0450400(  CALL DWORD PTR DS:[<&KERNEL32.GetVersio
004010A3  .  33D2           XOR EDX,EDX
004010A5  .  8AD4           MOV DL,AH
004010A7  .  8915 B066400(  MOV DWORD PTR DS:[4066B0],EDX
004010AD  .  8BC8           MOV ECX,EAX
004010AF  .  81E1 FF00000(  AND ECX,0FF
004010B5  .  890D AC66400(  MOV DWORD PTR DS:[4066AC],ECX
004010BB  .  C1E1 08        SHL ECX,8
004010BE  .  03CA           ADD ECX,EDX
004010C0  .  890D A866400(  MOV DWORD PTR DS:[4066A8],ECX
004010C6  .  C1E8 10        SHR EAX,10
004010C9  .  A3 A4664000    MOV DWORD PTR DS:[4066A4],EAX
004010CE  .  6A 00          PUSH 0
004010D0  .  E8 580D0000    CALL VC6Conso.00401E2D
004010D5  .  59             POP ECX
004010D6  .  85C0           TEST EAX,EAX
004010D8  .v 75 08          JNZ SHORT VC6Conso.004010E2
004010DA  .  6A 1C          PUSH 1C
004010DC  .  E8 9A000000    CALL VC6Conso.0040117B
004010E1  .  59             POP ECX
004010E2  >  8365 FC 00     AND DWORD PTR SS:[EBP-4],0
004010E6  .  E8 970B0000    CALL VC6Conso.00401C82
004010EB  .  FF15 0050400(  CALL DWORD PTR DS:[<&KERNEL32.GetComman
004010F1  .  A3 A47B4000    MOV DWORD PTR DS:[407BA4],EAX
004010F6  .  E8 550A0000    CALL VC6Conso.00401B50
004010FB  .  A3 80664000    MOV DWORD PTR DS:[406680],EAX
00401100  .  E8 FE070000    CALL VC6Conso.00401903
00401105  .  E8 40070000    CALL VC6Conso.0040184A
0040110A  .  E8 B5040000    CALL VC6Conso.004015C4
```

그림 4. 실제 프로그램의 EP부터 디스어셈블된 결과
Figure 4. Disassembled code from the program's EP.

따라서 언패킹 후 추출된 바이너리 코드에서 Startup Code를 찾을 수 있다면 변경 전 EP를 신속하게 찾을 수 있어 패커에 의해 보호된 악의적인 코드를 분석하는 시간을 단축시킬 수 있을 것으로 판단된다. 이를 실제로 검증하기 위해서 우선 샘플 코드를 작성하여 실행시킨 후, 로드된 샘플 코드의 메모리를 추적하여 실행 코드를 추출하고 그 결과에서 Startup Code를 검색한 후 변경 전 EP를 찾아낸다. 그 과정을 그림 5, 그림 6에 나타내었다.

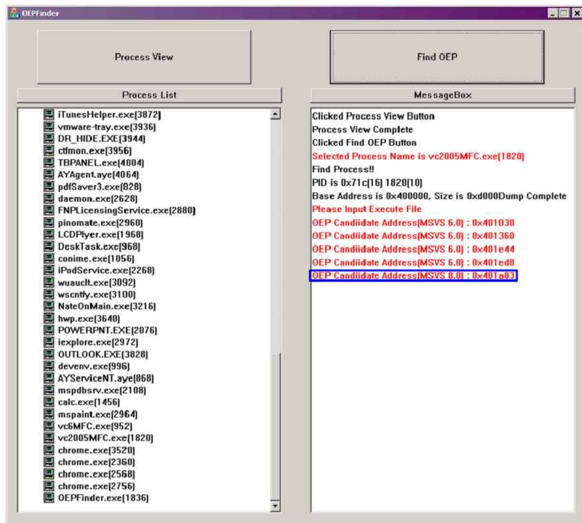

그림 5. Startup Code 검색을 통한 EP 추출
Figure 5. EP extraction by scanning the startup code.

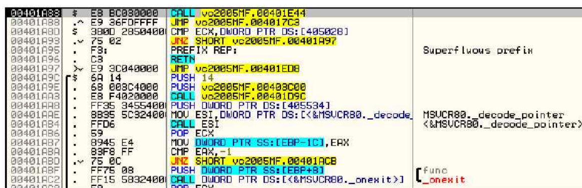

그림 6. 실제 EP
Figure 6. Real EP.

상기 실험은 패킹되지 않은 프로그램을 이용하여 제안한 방법이 옳은지 증명하기 위한 것이다. 따라서 실제 실험에서는 패킹된 프로그램을 이용하여 결과를 도출해야 할 것이며, 컴파일러마다 Startup Code가 다르기 때문에 컴파일러 별 결과를 도출해야 하며, Syntax를 다양하게 구성하여 EP 검출이 최적화된 Syntax를 찾는 실험도 필요하다.

Ⅳ. 실험결과

제3장에 서술된 시험 요구사항에 따라 우선 패킹된 프로그램을 이용하여 실험하였으며, Microsoft Visual Studio 8.0으로 컴파일된 프로그램에 대한 결과를 표 4에 나타내었다.

표 4. 패킹된 프로그램 EP 추출 결과 1
Table 4. EP extraction results for case 1.

패커	추출결과	추출 갯수
UPX	O	7개
ASPack	O	7개
ASProtect	O	7개
PECompact	O	7개
NSPack	O	7개
FSG	O	7개
RLPack	O	7개

실험결과 모두 EP 추출에 성공하였으며 추출 개수도 전체 코드에 비해 7개로 매우 효과적이라고 할 수 있지만 단순한 프로그램을 이용하였기 때문에 보다 높은 정확도를 위해 추가적으로 테스트할 필요성이 있다. 현재 테스트 프로그램은 8.0으로 생성되었지만 6.0에 해당하는 Syntax가 존재하기 때문에 발생하는 문제이므로 컴파일러 버전 별 결과를 도출하였으며, 이를 표 5에 나타내었다.

표 5. 패킹된 프로그램 EP 추출 결과 2
Table 5. EP extraction results for case 2.

패커	Syntax 3개		Syntax 4개	
	6.0	8.0	6.0	8.0
UPX	6개	1개	0개	1개
ASPack	6개	1개	0개	1개
ASProtect	6개	1개	0개	1개
PECompact	6개	1개	0개	1개
NSPack	6개	1개	0개	1개
FSG	6개	1개	0개	1개
RLPack	6개	1개	0개	1개

상기 실험결과 단순한 프로그램일 경우 Syntax가 4개면 EP를 추출할 수 있음을 확인할 수 있다. 단순한 프로그램이기 때문에 코드의 양이 적어 추출할 확률이 높을 수 있으므로 추가적으로 보다 복잡한 프로그램에 대해 결과를 도출할 필요성이 있어 컴파일러

별 결과를 도출하였으며, 이를 표 6, 표 7에 나타내었다.

표 6. 복잡한 프로그램 EP 추출 결과(6.0 프로그램)
Table 6. EP extraction results for a complicated program(6.0 program).

패커	Syntax 4개		Syntax 5개		Syntax 6개		Syntax 7개	
	6.0	8.0	6.0	8.0	6.0	8.0	6.0	8.0
UPX	73개	3개	72개	2개	72개	0개	28개	0개
ASPack	73개	3개	72개	2개	72개	0개	28개	0개
ASProtect	73개	3개	72개	2개	72개	0개	28개	0개
PECompact	73개	3개	72개	2개	72개	0개	28개	0개
NSPack	73개	3개	72개	2개	72개	0개	28개	0개
FSG	73개	3개	72개	2개	72개	0개	28개	0개
RLPack	73개	3개	72개	2개	72개	0개	28개	0개

표 7. 복잡한 프로그램 EP 추출 결과(8.0 프로그램)
Table 7. EP extraction results for a complicated program(8.0 program).

패커	Syntax 4개		Syntax 5개	
	6.0	8.0	6.0	8.0
UPX	4개	1개	0개	1개
ASPack	4개	1개	0개	1개
ASProtect	4개	1개	0개	1개
PECompact	4개	1개	0개	1개
NSPack	4개	1개	0개	1개
FSG	4개	1개	0개	1개
RLPack	4개	1개	0개	1개

상기 실험결과 Syntax가 증가함에 따라 복잡한 프로그램에 대해서는 최소 62%부터 최대 75%까지의 예상 EP를 감소시키는 것을 확인할 수 있었으나 Syntax가 7개 이상의 실험결과에 대해서는 더 이상의 EP를 감소할 수가 없었다. 이는 스레드나 기타 준비를 위해서 Startup Code와 비슷한 루틴을 사용하기 때문에 나타나는 결과라 할 수 있다. 이와 같은 루틴은 그 형태가 비슷하지만 메인으로 사용되는 루틴은 프로그램에 필요한 특별한 정보가 필요하기 때문에 이를 검색함으로써 실제 EP를 추출할 수 있을 것으로 판단된다. 실제 Startup Code가 다른 루틴과 차별화될 수 있는 특수한 기능은 내부에서 GetVersion() 함수를 호출하여 현재 운영체제의 정보를 얻기 때문에 이와 관련된 코드를 Syntax로 하여 결과를 도출하였으며, 이를 표 8, 그림 7에 나타내었다.

표 8. GetVersion()함수 Syntax 이용 EP 추출 결과 (6.0 프로그램)
Table 8. EP extraction results by syntax of GetVersion() function(6.0 program).

패커	GetVersion() 함수 관련 Syntax	
	6.0	8.0
UPX	1개	0개
ASPack	1개	0개
ASProtect	1개	0개
PECompact	1개	0개
NSPack	1개	0개
FSG	1개	0개
RLPack	1개	0개

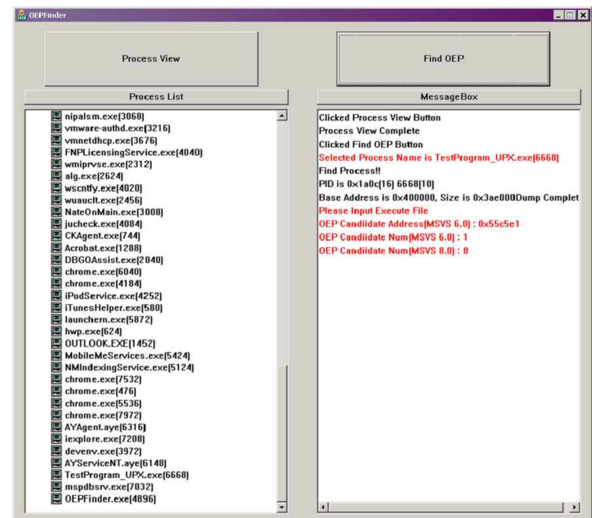

그림 7. GetVersion() 함수 Syntax 이용 EP 추출 결과(6.0 프로그램)
Figure 7. EP extraction results by syntax of GetVersion() function(6.0 program)

V. 결 론

본 논문에서는 악성코드가 사용하는 자기방어기법 중 가장 널리 사용되며 일관적인 해결책이 없는 패킹 기술에 대해 소개하였다. 소개된 내용을 토대로 패킹 기술이 가지는 취약점에 대해 분석하였으며, 취약점을 이용하여 악성코드를 보다 빠르게 분석할 수 있는 방안을 제시하였다. 제시된 내용을 다양한 실험

을 통해 증명하였으며, 실험결과 본 논문에서 제시된 방안은 악성코드 탐지 및 판단을 보다 효율적으로 할 수 있을 것으로 사료된다. 본 연구결과를 토대로 실제 악성코드가 활용하는 알려지지 않은 패커에 대한 연구 및 언패킹에 대한 연구, 다양한 컴파일러에 대한 연구도 향후 진행하고자 한다.

감사의 글

본 논문은 2011학년도 순천향대학교 교수 연구년제에 의한 연구결과임.

참 고 문 헌

[1] 천재홍, 박대우. “VoIP의 DoS 공격 차단을 위한 IPS의 동적 업데이트엔진" *한국컴퓨터정보학회 논문지,* 제10권 제5호, pp.317-226, 2006년 12월
[2] 안철수연구소, “악성코드 동향 연간 보고서”, *ASEC,* 2009년
[3] 배성훈, “‘7.7 DDoS 사고’ 대응의 문제점과 재발방지 방안”, *국회입법조사처, 현안보고서 제48호,* 2009
[4] Mark Vincent Yanson, "The Art of Unpacking", *IBM Internet Security Systems*
[5] Christian Collberg, Clark Thomborson and Duglas Low, "A Taxonomy of Obfuscating Transformations", *Department of Computer Science, The University of Auckland, Thechnical Report #148,* 1997
[6] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie and N. Tawbi, "Static Detection of Malicious Code in Executable Programs", *International Journal of Req. Eng.*, 2001
[7] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, Randal E. Bryant, "Semantic-Aware Malware Detection", *In the Proceedings of the IEEE Symposium on Security and Privacy, pp.32-46*, May 2005
[8] Gogu Balakrishnan, Thomas Reps., "Analyzing Memory Accesses in x86 Executables", In Proceedings of Compiler Construction(LNCS 2985), pp.5-23, April 2004
[9] Arun, Lakhotia and Eric Uday Kumar, "Abstracting Stack to Detect Obfuscated Calls in Binaries", *IEEE International Workshop on Source Code Analysis and Manipulation,* pp.17-26, 2004
[10] Virus Bullentin Conference, 2007
[11] "Microsoft Portable Executable and Common Object File Format Specification", *Microsoft,* 2008

이 경 률 (李庚栗)

2008년 8월: 순천향대학교 정보보호학과 (공학사)
2010년 8월: 순천향대학교 정보보호학과 (공학석사)
2010년 9월~현재: 순천향대학교 정보보호학과 박사과정
2011년 5월~12월: (미)퍼듀대학교 정보보호교육연구센터 연구원
관심분야 : vulnerability analysis, virtualized obfuscation, system security, insider threats

임 강 빈 (任綱彬)

1992년 2월: 아주대학교 전자공학과 (공학사)
1994년 2월: 아주대학교 전자공학과 (공학석사)
2001년 2월: 아주대학교 전자공학과 (공학박사)
1999년 3월~2000년 2월: (미)아리조나주립대학교 연구원
2010년 12월~2012년 2월 : (미)퍼듀대학교 정보보호교육연구센터 객원교수
2003년 3월~현재: 순천향대학교 정보보호학과 교수
2005년 3월~현재: 한국정보보호학회 이사
2009년 3월~현재: 한국인터넷정보학회 이사
관심분야 : vulnerability analysis, insider threats, secure hardware architecture, homeland security