情報保護學會論文誌
第 22 卷 第 3 號, 2012. 6

24Bit BMP 이미지를 이용한 쉘코드 은닉 기법*

금 영 준,† 최 화 재, 김 휘 강‡
고려대학교 정보보호대학원

Hiding Shellcode in the 24Bit BMP Image*

Young Jun Kum,† Hwa Jae Choi, Huy Kang Kim‡
Graduate School of Information Security, Korea University

요 약

1990년도 중반에 처음 개념이 소개된 이래, 현재까지도 가장 빈번하게 사용되고 매우 심각한 피해를 초래하는 공격기법으로 버퍼 오버플로우(buffer overflow) 취약점을 이용한 공격을 들 수 있다. 쉘코드(shellcode)는 이러한 버퍼 오버플로우 공격에 사용되는 기계어 코드(machine code)로서, 공격자는 자신이 의도한 바대로 수행되는 쉘코드를 작성하고 이를 공격 대상 호스트의 메모리에 삽입, EIP를 조작하여 시스템의 제어 흐름을 가로챌 수 있다. 따라서 버퍼오버플로우를 일으킨 후 쉘코드를 적재하려는 것을 탐지하기 위한 많은 연구들이 수행되어 왔으며, 공격자들은 이런 탐지기법들을 우회하는 기법을 끊임없이 개발해 왔다. 본 논문에서는 이러한 쉘코드 공격 기법 및 방어 기법들에 대해서 살펴보고, 24Bit BMP 이미지에 쉘코드를 은닉시킬 수 있는 새로운 기법을 제안하고자 한다. 이 기법을 통하여 쉘코드를 손쉽게 은닉할 수 있으며, 현재의 다양한 탐지 기법들을 쉽게 우회할 수 있음을 확인할 수 있었다.

ABSTRACT

Buffer overflow vulnerability is the most representative one that an attack method and its countermeasure is frequently developed and changed. This vulnerability is still one of the most critical threat since it was firstly introduced in middle of 1990s. Shellcode is a machine code which can be used in buffer overflow attack. Attackers make the shellcode for their own purposes and insert it into target host's memory space, then manipulate EIP(Extended Instruction Pointer) to intercept control flow of the target host system. Therefore, a lot of research to defend have been studied, and attackers also have done many research to bypass security measures designed for the shellcode defense. In this paper, we investigate shellcode defense and attack techniques briefly and we propose our new methodology which can hide shellcode in the 24bit BMP image. With this proposed technique, we can easily hide any shellcode executable and we can bypass the current detection and prevention techniques.

Keywords: ShellCode, Obfuscation

I. 서 론

컴퓨터 보안의 역사는 공격과 방어의 역사이다. 취약점을 악용할 수 있는 공격법이 나오면 이를 막기 위한 방어법이 연구되고, 다시 이를 우회할 수 있는 공격법이 지속적으로 개발되어 왔다. 이러한 전개는 결과적으로 보안 수준과 기법들을 크게 발전시켰고 이를 통해 과거보다 안전한 서비스 환경을 구축할 수 있게

접수일(2012년 4월 18일), 수정일(2012년 6월 12일), 게재확정일(2012년 6월 19일)
* 본 연구는 국방과학연구소의 지원을 받아 수행하였습니다 (UD110051ED).
† 주저자, 0junkum@korea.ac.kr
‡ 교신저자, cenda@korea.ac.kr

되었다. 특히 버퍼 오버플로우(buffer overflow) 기법은 이런 양상 속에서 발전해온 가장 대표적인 공격 기법 중 하나이다. 버퍼 오버플로우는 1988년 최초의 웜(worm)인 모리스 웜(morris worm)에 의해 사용되어 그 심각성이 널리 알려졌으며[1], 1996년 Aleph One (aleph1)이 "Smashing the stack for Fun and Profit"이라는 기사를 Phrack 49호[2]에 게재하면서 많은 사람들이 이에 관심을 갖게 되었다. 이로부터 지금까지 20년이 넘는 세월동안 버퍼 오버플로우와 관련된 다양한 공격 및 방어 기법들이 제안되고 연구되어져 왔다. 버퍼 오버플로우는 매우 고전적인 취약점이지만, CVE Details[3]에 따르면 최근에도 매우 빈번하게 발견되고 악용되고 있는 취약점임을 알 수 있다. 또한, "2011 CWE/SANS Top 25 Most Dangerous Software Errors"[4]에서는 3위를 기록하며 비교적 쉽게 발견되어 악용이 쉽지만, 현재까지도 매우 심각한 피해를 초래할 수 있는 위험성을 가진 취약점으로 분류되었다. [표 1]과 [그림 1]은 CVE Details에서 공개한 취약점 유형별 발생빈도에 대한 조사 자료이다. 최근에는 플랫폼이 웹에 집중되면서 SQL Injection, XSS (Cross Site Scripting)등과 같은 접근이 용이하고 공격이 수월한 웹 관련 취약점들이 많이 악용되는 추세지만, 순수하게 발견된 취약점의 수로만 보면 오히려 오버플로우가 상위권을 차지하고 있다는 것을 알 수 있다.

버퍼 오버플로우 취약점(vulnerability)을 사용한 공격은 해당 공격에 영향을 받는 취약점이 목표 시스템에 존재할 때 익스플로잇(exploit) 코드를 로컬 또는 원격에서 전송시키는 방식으로 이루어지며, 버퍼 오버플로우가 발생한 후 최종적으로 해커가 의도한 명령을 실행시키기 위하여 공격의 최종단계에서 쉘코드(shellcode)를 적재하여 보내게 된다.

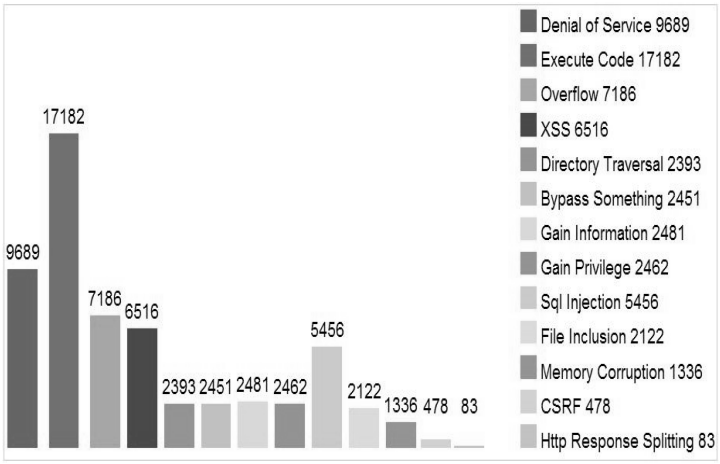
취약점은 네트워크 등 외부의 입력에 의해 코드의 실행흐름이 변경될 수 있는 소프트웨어의 결함을 이야기 한다. 익스플로잇은 취약점을 사용해 공격하는 특정 공격 패턴을 이야기 한다. 일반적으로 공격자는 하나의 취약점에 다양한 형태의 공격코드를 작성하므로, 하나의 취약점에는 다양한 형태의 익스플로잇이 있다. 마지막으로 쉘코드는 익스플로잇에 의해 전달 되게 되는 페이로드(payload)로서, 취약점을 이용해 코드의 실행 흐름을 변경한 뒤 가장 먼저 실행되는 기계어 코드(machine code)이다[5].

쉘코드는 버퍼 오버플로우 공격에 핵심으로, 공격자는 자신의 목적을 달성하기 위해 자신이 의도한 바대로 수행되는 쉘코드를 작성하고, 익스플로잇을 사용해 상대방 호스트의 취약한 프로세스에 삽입한 뒤 실행한다. 익스플로잇에 의해 실행되는 쉘코드는 특정 프로세스의 제어의 흐름을 가로챈 뒤 실행되는 코드이

[표 1] 취약점 유형별 발생빈도(1999~2012), CVE Details

Vul. : Vulnerability,

Year	# of Vul.	DoS	Code Execution	Overflow	Memory Corruption	SQL Injection	XSS	Directory Traversal	HTTP Response Splitting	Bypass Something	Gain Information	Gain Privilege	CSRF	File Inclusion
1999	894	177	112	172			2	7		25	11	102		
2000	1020	257	207	206		2	4	20		48	18	138		
2001	1677	402	402	297		7	34	123		83	35	219		2
2002	2156	497	553	435	2	41	200	103		127	74	199	2	14
2003	1526	381	476	370	2	49	129	60	1	61	69	144		16
2004	2450	580	614	409	3	148	291	110	12	145	95	134	5	38
2005	4934	838	1625	652	21	604	786	202	15	289	259	221	11	100
2006	6610	893	2719	662	91	967	1302	321	8	267	266	184	18	849
2007	6520	1100	2601	951	95	706	883	339	14	267	322	242	69	700
2008	5632	894	2310	699	128	1102	807	363	7	288	272	188	83	170
2009	5736	1035	2184	700	188	963	851	322	9	336	302	223	115	138
2010	4651	1102	1714	677	342	520	605	275	8	234	282	237	86	73
2011	4155	1221	1334	770	351	294	466	108	7	197	409	206	58	17
2012	1137	312	331	186	113	53	156	40	2	84	67	25	31	5
총계	49098	9689	17182	7186	1336	5456	6516	2393	83	2451	2481	2462	478	2122
%		19.7	35.0	14.6	2.7	11.1	13.3	4.9	0.2	5.0	5.1	5.0	1.0	4.3



〔그림 1〕 취약점 유형별 누적 발생빈도(1999~2012), CVE Details

므로 문제가 발생했을 경우 관리자 권한이 노출되는 등 매우 치명적인 결과를 초래할 수 있다. 그렇기 때문에 쉘코드를 탐지하거나 막기 위한 다양한 연구들이 진행되어왔으며, 반대로 공격자들은 이러한 연구를 통해 만들어진 보안 장치들을 우회하고자 계속적으로 시도해왔다. 쉘코드 난독화 기법은 탐지 알고리즘을 우회하고자 하는 공격자들의 노력의 대표적인 결과물이라고 할 수 있고, 지속적으로 새로운 기법들이 연구되고 있다. 본 논문에서는 쉘코드를 탐지하거나 막기 위한 연구들과 쉘코드 난독화의 발전과정에 대해서 간략하게 살펴보고, 이미지에 쉘코드를 은닉시키는 공격기법을 제안하여 새로운 공격의 형태를 제시한다. 해당 기법은 익스플로잇과 쉘코드를 분리시켜 쉘코드 부분만을 따로 이미지에 은닉하여 전달시키는 방법으로 현재의 보안기법들을 효과적으로 우회 할 수 있는 가능성을 내포하고 있다. 본 논문에서는 IA32를 대상으로 연구를 진행하였다.

II. 쉘코드 방어 기법

쉘코드 방어 기법은 크게 쉘코드 탐지, 쉘코드 제작 방해, 쉘코드 실행 방해의 3가지로 분류될 수 있다. 쉘코드 탐지의 경우 정적(static)인 방법과 동적(dynamic)인 방법이 있는데 정적인 방법의 경우 초기에는 쉘코드를 탐지하기 위해 쉘코드가 보이는 특정한 패턴이나 특징에 착안했다. 0x90으로 이루어진 NOP Sled나 원격 쉘을 띄우기 위한 "/bin/sh"라는 문자열, 또는 시스템 콜(system call) 패턴 등을 시그니처(signature)화하여 네트워크 패킷이나 프로세스에 들어오는 입력 값에 이런 패턴이 발견되는 지 검사하였다. 패턴기반 탐지는 과부하가 적어 속도가 빠르고 탐지율이 좋은 편이기 때문에 현재까지도 Snort[6]와 같은 IDS(Intrusion Detection System)에서 가장 널리 사용되고 있는 방법이다. 그러나 이를 우회하기 위해 공격자들은 난독화(obfuscation) 기법을 사용하기 시작하였는데 이는 다음 절에서 보다 자세하게 다루기로 한다. 난독화 기법이 점점 발전하고, 네트워크 역시 대량의 트래픽이 고속으로 흐르는 환경으로 진화함에 따라 문자열 검색기법에 기반을 한 패턴기반 탐지로는 한계를 가지게 되었다. 이를 보완하기 위해 시도된 방법 중 하나가 통계(statistics) 기반 탐지 기법이다. 데이터마이닝(data-mining)을 통해 해당 프로세스가 받는 정상적인 입력의 바이트 분포(byte distribution)를 학습하고 이 바이트 분포에 어긋나면 비정상적인 입력으로 간주하여 탐지를 하게 된다. 그러나 오탐(false-positive) 문제와 데이터마이닝이라는 다소 오버헤드가 큰 방법을 사용하기 때문에 현재까지도 실제로 상용 침입탐지시스템에 적용되어 사용되는 예는 극히 드물다고 할 수 있다.

한 편 동적인 탐지 방법으로는 에뮬레이션(emulation) 기법이 있다. 쉘코드는 본질적으로 헥스(hex) 값의 집합이며 이것은 일반적인 데이터도 마찬가지이다. 다만 헥스 값이 데이터로서 해석이 되느냐 아니면 EIP에 의해 명령어로서 해석이 되느냐의 차이일 뿐이다. 메모리에 코드 영역(code section)에 있지 않은 헥스 값이라도 EIP로 가리키게 되면 명령어로서 해석이 되게 된다. 따라서 이러한 점을 악용

하여 공격자는 입력 값으로 목적에 부합하는 기계코드에 해당되는 헥스 값들의 집합을 집어넣는 것이 쉘코드 공격인 것이다. 그러나 정상적인 경우라면 일반 사용자가 이런 악성행위를 하는 기계코드에 해당되는 헥스 값의 집합을 입력 값으로 넣을 리가 없다. 따라서 가상의 CPU와 레지스터로 구성되어 있는 가상 환경을 구성하고 타겟 프로세스로 가는 입력 값을 가로채어서 미리 실행을 시켜봄으로써 악성행위를 하는지 하지 않는지를 판단하는 것이다. 이 방법론은 2007년에 Polychronakis 등에 의해 처음으로 제안되었으며, 이들은 자신들의 연구에서 자신들의 방법은 직접 실행을 해보기 때문에 탐지가 어려웠던 쉘코드 유형인 폴리몰픽(polymorphic) 쉘코드도 탐지할 수 있다는 결과를 보였다[7-8]. 그러나 이들이 제안한 방법론은 단순히 가상의 환경만 구성했을 뿐 타겟 프로세스의 메모리나 레지스터 정보는 가지고 있지 않기 때문에 공격자들은 바로 이런 점을 이용하여 공격을 수행하기 시작했다. 예를 들어 쉘코드에 cmp eax, 0xbe5e(ebp), jz +10 이라는 명령어가 포함되어 있다고 하면, 0xbe5e(ebp)는 메모리에 있는 정보이기 때문에 타겟 프로세스에서만 정확한 분기를 할 수가 있다. 하지만 에뮬레이션에서는 이러한 런타임(run-time) 정보를 알 수가 없기 때문에 정확한 분기를 예측할 수가 없게 된다. 최근 연구에서 B. Gu 등[9]은 이를 개선하기 위하여 타겟 프로세스의 가상 메모리를 덤프하여 사용함으로써 이런 문제를 해결할 수 있도록 하였다. 에뮬레이션 기법의 가장 큰 장점은 직접 실행해본다는 방법론적 특성 때문에 난독화된 쉘코드에도 매우 높은 탐지율을 보인다는 것이다. 다만 오버헤드가 다소 있는 편이다.

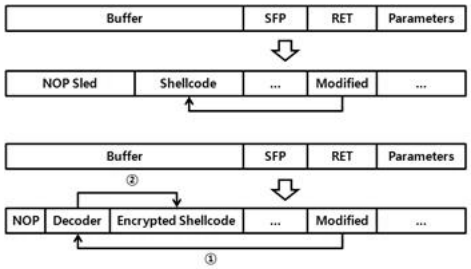
쉘코드의 제작을 어렵게 하는데 방법으로는 ASR(Address Space Randomization)[10], 입력 값 제한 등을 들 수 있다. ASR은 가상 주소 공간을 프로세스가 생성될 때 마다 랜덤하게 할당하여 공격자가 쉘코드 제작 시 복귀 주소(return address)를 가늠하기 어렵게 만드는 방법이다. 최근에 기본적으로 컴파일러들이 이 기능을 지원한다. 입력 값을 제한하는 방법은 사실상 0x00부터 0xFF까지 모든 헥스 값이 우리가 사용하는 문자나 숫자에 해당되는 것은 아니기 때문에 서비스 영역에서 사용하지 않는 헥스 값들이 존재하게 된다는 점에 착안한 것이다. 예를 들어 FTP 서비스에서는 사용자가 입력하는 명령어는 모두 대문자이며 디렉터리 경로의 경우 대소문자와 숫자, 특정 특수문자로만 이루어져 있다. 때문에 이외의 입력 값을 필터링하면 특정 쉘코드는 입력 값으로 들어올 수 없게 된다. 대소문자나 숫자 등에 해당하는 쉘코드는 들어오겠지만 못 들어오는 값이 생기기 때문에 공격을 막을 수 잇게 된다. 이를 우회하기 위한 난독화 기법인 문자숫자식 쉘코드는 다음 절에서 살펴보기로 한다.

쉘코드가 타겟 프로세스에서 메모리에 쓰이거나 실행되는 것을 어렵게 만드는 방법으로는 Stack Guard[11], Stack Shield, DEP(Data Execution Prevention), ISR(Instruction Set Randomization)[12] 등이 있다. Stack Guard는 컴파일 단계에서 RET 앞에 랜덤 값을 삽입하여 무결성을 검사하는 방법이고 DEP는 특정 영역에 실행권한 자체를 없애는 것이다. 이렇게 되면 쉘코드가 들어와도 실행이 안 되게 된다. ISR은 이름 그대로 Instruction Set을 랜덤화하는 방법으로 쉘코드 공격을 방해하는데 효과적이나 특정 작업에 대해서는 너무 비효율적인 단점을 가지고 있다.

III. 쉘코드 난독화 기법

쉘코드 난독화 기법이란 쉽게 말해 쉘코드를 암호화하는 것을 말한다. 암호화를 하게 되면 키에 따라 쉘코드의 바이트 패턴이 변하기 때문에 패턴기반 탐지를 우회할 수 있게 된다. 키 스페이스(key space)가 크면 패턴기반 탐지에서는 가능한 모든 시그니처를 가지고 있어야 하게 되는데 이것이 사실상 어렵기 때문이다. 그런데 이렇게 암호화가 될 경우 탐지는 피할 수 있지만 실행이 불가능하다는 단점이 있다. 따라서 디코더(decoder)라는 복호화 루틴을 암호화된 쉘코드에 붙여 보내 타겟 호스트에서 런타임 순간에 원래의 쉘코드로 복호화되어 실행될 수 있도록 하는 방식을 따른다. [그림 2]는 버퍼 오버플로우 시 삽입된 일반적인 쉘코드의 구조와 암호화된 쉘코드 구조를 나타낸다.

[그림 2]에서 일반적인 쉘코드의 경우 NOP Sled나 쉘코드 패턴 등으로 인해 쉽게 탐지가 가능하지만 아래의 암호화된 쉘코드의 경우 패턴이 일정하지 않기 때문에 탐지하기가 어렵게 된다. NOP Sled의 경우 초기에는 0x90을 사용하였으나 단순히 버퍼를 채우기 위한 용도이고 굳이 실행될 필요가 없는 경우에는 쓰레기 값을 넣어 패턴에 의해 탐지될 수 없도록 한다. 또는 inc eax, dec eax와 같은 결과적으로 의미가 없는 형태로 NOP Sled를 구성한다. 그런데 이 암호



〔그림 2〕 일반적인 쉘코드 구조(위)와 암호화된 쉘코드 구조(아래)

화된 쉘코드에는 큰 단점이 하나 있는데 바로 디코더가 항상 일정하다는 것이다. 디코더는 항상 실행 가능한 상태여야 하기 때문에 암호화가 불가능하다. 그렇기 때문에 디코더에 나타나는 특수한 패턴(GetPC 코드, 반복적인 읽기와 쓰기 등)이 시그니처로 활용되어 탐지가 되게 되었다. 이를 극복하기 위해 사용되는 난독화 기법이 바로 폴리몰픽(polymorphic), 메타몰픽(metamorphic) 기법이다. 이 기법은 디코더를 매번 다르게 생성하는 기법으로 기능적으로 완전히 동일한 일을 하더라도 바이트 패턴이 다른 디코더를 생성할 수 있게 된다. 사용되는 대표적인 방법들로는 쓰레기코드 삽입(dead code insertion), 레지스터 재할당(register reallocation), 인스트럭션 치환(instruction substitution), 코드 전위(code transposition) 등이 있다. 똑같은 쉘코드라도 매번 다른 디코더와 암호화된 쉘코드가 생성되기 때문에 패턴기반 방식으로는 탐지하는 것이 매우 어려우며, Metasploit〔13〕, ADMMutate〔14〕, CLET〔15〕 등 폴리몰픽 쉘코드 엔진이 공개되어 있어 쉽게 이용할 수 있다. 이러한 기법들은 기존의 탐지 방법 중 패턴 기반 탐지에 특히나 강점을 보인다. Y. Song 등〔16〕은 이런 폴리몰픽 엔진들의 성능을 평가하고 폴리몰픽 엔진이 너무나 다양한 패턴을 만들어내기 때문에, 이런 모든 케이스를 모델링하여 탐지하는 것은 매우 어렵다는 것을 보였다. 그리고 오히려 화이트리스트(white-list) 기반의 정책이 이러한 폴리몰픽 공격에 효과적으로 대응할 수 있을 것이라 말한다. CLET 폴리몰픽 엔진의 경우 앞서 언급했던 통계적인 탐지를 우회하기 위해 스펙트럼 분석(spectrum analysis)이라는 기능을 제공하는데, 이는 서비스의 정상적인 입력 값이 어떤 바이트 분포를 가지는 지를 엔진이 분석하여 패딩(padding)을 통해 공격 코드의 헥스 값이 정상 분포와 유사한 통계적 분포를 가지도록 만들어주는 방법이다. 이를 통해 앞서 언급했던 통계적 탐지 기법 또한 우회가 될 수 있음이 드러났다.

한편 2절에서 언급했던 입력 값을 필터링을 우회하기 위해서 문자숫자식 쉘코드(alphanumeric shellcode)〔17〕가 개발되었다. 이 난독화 기법은 일반적으로 사용하는 문자나 숫자에 해당되는 기계코드로만 쉘코드를 재구성하는 방식이다. 〔표 2〕는 Linux IA32 Bind Shell을 문자숫자식 쉘코드로 바꾼 예이다. '?'는 화면에 출력 불가능한 문자를 나타낸다. 두 코드는 형태는 다르지만 동일한 작업을 수행하는데 문자숫자식 쉘코드는 일반적으로 사용하는 문자로만 이루어져 있는 것을 확인할 수 있다. 문자숫자식 쉘코드 역시 패턴기반 탐지에서 탐지되지 않기 위해 개선을 거듭해 매번 다른 패턴이 생성되도록 하고 있다. 이와 같은 유사한 목적으로 개발된 쉘코드로는 Unicode〔18〕, UTF-8〔19〕 인코딩 등이 있다.

최근 J. Mason 등은 쉘코드를 영어문장으로 만드는 English Shellcode〔20〕라는 연구를 수행하였다. 이들은 만약 쉘코드가 영어 문장으로 구성될 수 있다면, 영어 문장은 어디서나 정상적인 입력 값이기 때문에 입력 값 필터링 우회는 물론 여러 탐지 방법을 우회하는데 매우 효과적일 것이라고 말한다. 쉘코드가 영어로 감싸져 있기 때문에 패턴 기반 탐지가 어려우며, 더불어 그들은 실험을 통해 자신들이 만든 영어 쉘코드가 일반적인 영어 문장들과 매우 유사한 바이트 분포를 나타낸다는 것을 보였다. 저자들은 따라서 쉘코드를 웹, 문서, 파일명, 폴더명 등 어디에나 존재시킬 수 있는 가능성이 늘어날 수 있다고 말한다. 사실상 이와 같은 쉘코드는 에뮬레이션 기법을 제외한 현재의 다른 탐지 기법들로는 탐지가 어렵다고 보인다. DEP나 ASLR 등과 같은 2차적인 보호장치들이 있으나 이들은 난독화 기법이 아닌 다른 공격 기법들로 충분히 우회가 가능하기 때문에 호스트에 쉘코드를 존재시킬 수 있다는 것만으로도 공격의 가능성은 증가하게 된다. 본 논문에서는 이와 같이 쉘코드를 정상적인 값과 같게 만드는 데에서 얻을 수 있는 장점이 많다는

〔표 2〕 IA32 Bind Shell 문자숫자식 쉘코드 예

	Hex	Ascii
일반 쉘코드	31DB5343 536A…	1?SCSj?jfX???? ??…
문자숫자식 쉘코드	37494951 5A6A…	7rrQZjJX0B….

점에 착안하여, 이미지에 쉘코드를 실행 가능한 상태로 은닉시키는 방법을 연구, 제안한다. 특히 그 중에서도 24Bit BMP 이미지에 쉘코드를 은닉시키는 방법을 연구하였다.

IV. 24Bit BMP 이미지 쉘코드 은닉 기법

4.1 쉘코드 은닉 기법의 기본 원리

24Bit의 BMP 이미지 파일에 쉘코드를 실행 가능한 상태로 은닉시키는 방법에 대한 연구를 진행하였다. BMP 이미지 파일은 비압축 포맷이기 때문에 이미지에 가장 손실이 없는 포맷 중에 하나인데 1Bit(흑백), 4Bit(4색), 8Bit(256색), 16Bit, 24Bit, 32Bit가 있다. 이 중 16, 32는 거의 사용되지 않고 24Bit BMP 파일이 가장 많이 사용되고 있다. 24Bit BMP 이미지는 RGB의 3채널로 구성되며 3개의 채널 값이 하나의 픽셀(Pixel)을 나타낸다. 1채널(R or G or B)이 1바이트로 256색을 표현하는 것이 가능하다. 이것이 본 논문에서 쉘코드를 은닉시킬 이미지로 24Bit BMP 이미지를 선택한 이유이다. 하나의 픽셀의 색을 3개의 RGB 값이 나타내기 때문에 약간 값이 바뀌어도 크게 색이 바뀌지 않기 때문이다. 다음 〔그림 3〕과 〔그림 4〕는 동일한 그림인데 압축 포맷인 jpg 파일과 24bit bmp 파일을 각 바이트 당 1bit를 1씩 증가시켜 8바이트를 수정했을 때 결과물이다. 편집도구로는 HxD라는 헥스 에디터를 사용하였다.

〔그림 4〕에서 확인할 수 있듯이 압축 포맷인 JPG는 1Bit를 1씩만 바꾸었음에도 불구하고 그림이 육안으로 구분이 될 정도로 손상이 됨을 알 수가 있다. 그러나 24Bit BMP 파일은 전혀 구분이 불가능하다. 본 논문은 바로 이런 점에 착안하여 일정 범위 내에서 헥스 값을 조정하여 쉘코드를 삽입시키는 방법을 택하였다. 24Bit BMP는 〔그림 5〕와 같은 구조를 가지고 있는데 헤더(header)는 이미지가 정상적으로 읽히기 위해서는 조작할 수 없는 부분이므로 실제 이미지를 나타내는 픽셀 데이터(pixel data) 부분에 삽입한다.

그러나 값을 약간 조정할 수 있다고 하더라도 이미지의 랜덤한 헥스 값들의 조합에서 쉘코드 헥스 값 전부를 순차적으로 삽입한다는 것은 불가능하다. 그렇기 때문에 쉘코드 헥스 값을 기계코드별로(명령어 별로) 쪼개어 삽입을 하고 쪼개어진 기계코드 사이에 있는 헥스 값들을 마찬가지로 조금씩 조작하여 결론적으로 결과에 영향을 미치지 않도록 더미코드(dummy code)를 생성하거나, 기계코드 뒤에 점프코드(jump code)를 삽입하는 방법을 택했다.

그러나 앞서 설명한 방법만으로는 쉘코드에 적용이 어렵다. 왜냐하면 쉘코드를 이루고 있는 기계코드들은 1바이트부터 4바이트이상 까지 다양하게 연속된 바이

Offset(h)	00	01	02	03	04	05	06	07
00000000	FF	D8	FF	E0	00	10	4A	46
00000010	00	60	00	00	FF	DB	00	43
00000020	07	07	07	09	09	08	0A	0C
00000030	13	0F	14	1D	1A	1F	1E	1D
00000040	22	2C	23	1C	1C	28	37	29
00000050	39	3D	38	32	3C	2E	33	34
00000060	09	0C	0B	0C	18	0D	0D	18
00000070	32	32	32	32	32	32	32	32
00000080	32	32	32	32	32	32	32	32
00000090	32	32	32	32	32	32	32	32
000000A0	00	11	08	02	00	02	00	03
000000B0	01	FF	C4	00	1F	00	00	01
000000C0	00	00	00	00	00	00	00	01
000000D0	0A	0B	FF	C4	00	B5	10	00
000000E0	05	04	04	00	00	01	7D	01
000000F0	31	41	06	13	51	61	07	22
00000100	42	B1	C1	15	52	D1	F0	24
00000110	18	19	1A	25	26	27	28	29
00000120	43	44	45	46	47	48	49	4A
00000130	63	64	65	66	67	68	69	6A

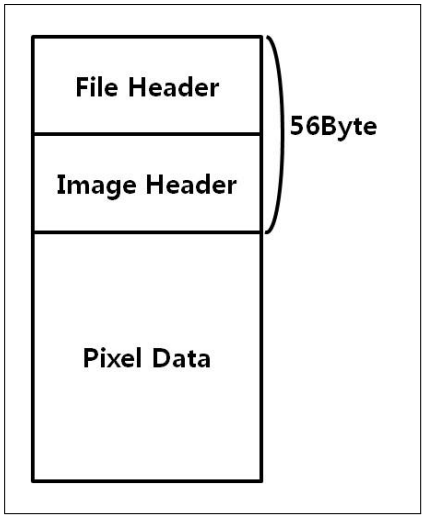
Offset(h)	00	01	02	03	04	05	06	07
00000000	FF	D8	FF	E0	00	10	4A	46
00000010	00	60	00	00	FF	DB	00	43
00000020	07	07	07	09	09	08	0A	0C
00000030	13	0F	14	1D	1A	1F	1E	1D
00000040	22	2C	23	1C	1C	28	37	29
00000050	39	3D	38	32	3C	2E	33	34
00000060	09	0C	0B	0C	18	0D	0D	18
00000070	32	32	32	32	32	32	32	32
00000080	32	32	32	32	32	32	32	32
00000090	32	32	32	32	32	32	32	32
000000A0	00	11	08	02	00	02	00	03
000000B0	01	FF	C4	00	1F	00	00	01
000000C0	00	00	00	00	00	00	00	01
000000D0	0A	0B	FF	C4	00	B5	10	00
000000E0	05	04	04	00	00	01	7D	01
000000F0	31	41	06	13	51	61	07	22
00000100	42	B1	C1	15	52	D1	F0	24
00000110	18	19	1A	25	26	27	28	29
00000120	43	44	45	46	47	48	49	4A
00000130	64	65	66	67	68	69	70	6B

〔그림 3〕 JPG 파일 수정 전(좌), 수정 후(우)

Offset(h)	00	01	02	03	04	05	06	07
00000000	42	4D	36	00	0C	00	00	00
00000010	00	00	00	02	00	00	00	02
00000020	00	00	00	00	0C	00	C4	0E
00000030	00	00	00	00	00	00	39	16
00000040	1C	5D	41	1E	61	39	15	5E
00000050	62	42	1B	5E	3A	15	59	42
00000060	57	38	79	74	58	93	80	81
00000070	BD	D9	AF	BD	DB	B3	C0	E0
00000080	DC	AE	BA	DB	A8	BA	D9	A3
00000090	7D	94	C2	65	61	AC	5B	51
000000A0	64	CC	5A	69	D4	5F	70	DB
000000B0	E5	6D	8A	E7	76	8F	E9	7B
000000C0	7C	A3	EB	79	A1	EC	81	A1
000000D0	A6	EC	7E	A2	EC	85	A2	F2
000000E0	ED	7D	A0	EC	76	9E	E5	80
000000F0	67	45	8E	60	47	8C	4C	2F
00000100	30	6F	39	16	61	42	14	51
00000110	53	3B	10	4E	48	1F	5D	46
00000120	39	0A	48	37	11	49	48	21
00000130	16	57	5F	33	71	53	2D	74

Offset(h)	00	01	02	03	04	05	06	07
00000000	42	4D	36	00	0C	00	00	00
00000010	00	00	00	02	00	00	00	02
00000020	00	00	00	00	0C	00	C4	0E
00000030	00	00	00	00	00	00	39	16
00000040	1C	5D	41	1E	61	39	15	5E
00000050	62	42	1B	5E	3A	15	59	42
00000060	57	38	79	74	58	93	80	81
00000070	BD	D9	AF	BD	DB	B3	C0	E0
00000080	DC	AE	BA	DB	A8	BA	D9	A3
00000090	7D	94	C2	65	61	AC	5B	51
000000A0	64	CC	5A	69	D4	5F	70	DB
000000B0	E5	6D	8A	E7	76	8F	E9	7B
000000C0	7C	A3	EB	79	A1	EC	81	A1
000000D0	A6	EC	7E	A2	EC	85	A2	F2
000000E0	ED	7D	A0	EC	76	9E	E5	80
000000F0	67	45	8E	60	47	8C	4C	2F
00000100	30	6F	39	16	61	42	14	51
00000110	53	3B	10	4E	48	1F	5D	46
00000120	39	0A	48	37	11	49	48	21
00000130	17	58	60	34	72	54	2E	75

〔그림 4〕 24Bit BMP 파일 수정 전(좌), 파일 수정 후(우)



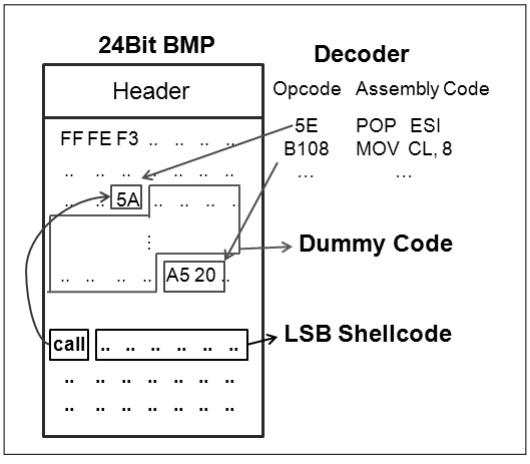
〔그림 5〕 24Bit BMP 이미지 구조

트를 이루고 있는데, 약간 값 조작이 가능하다고 하더라도 바이트가 많이 연속될수록 이미지 내 데이터에서 찾기가 어려워진다는 문제가 있기 때문이다. 예를 들어 0x49라는 기계코드가 있을 경우 이미지에 굉장히 많은 헥스 값 0x49가 존재하겠지만, 0x49 0xFE라는 기계코드가 있을 경우 0x49만 찾을 때 보다 이미지에서 찾기가 훨씬 어려워지게 된다. 게다가 쉘코드의 길이가 총 몇 바이트나 될지 알 수가 없기 때문에 더욱 어렵다. 본 연구에서는 디코더를 사용하였는데 이 샘플 디코더는 총 23바이트에 각 기계코드가 2바이트 이내로 제작되어 있다. 반면 쉘코드는 쉘코드마다 이런 조건들이 천차만별이다. 예를 들어 60바이트짜리 쉘코드가 있을 경우 각 기계코드와 유사한 부분을 찾아야 되고 사이사이에 더미코드를 넣거나 점프코드를 넣어야하는 것이 23바이트 디코더에 약 3배 이상 늘어나기 때문이다. 또한 더미코드를 생성하는 방법의 경우 이미 만들어져 있는 쉘코드는 쉘코드마다 사용하는 레지스터 등이 고정되어 있지 않기 때문에 더미코드를 생성 시 일반화하기가 어렵다는 문제점이 있다. 예를 들어 push eax, push ebx, push ecx라는 쉘코드가 있다고 가정하면 push eax와 push ebx 사이의 더미코드는 ebx를 수정해서는 안 되고 스택을 조작하는 명령어가 사용되면 안 된다. 때문에 제약이 많고 게다가 쉘코드마다 이런 경우가 너무 다양하기 때문에 더미코드를 일반적으로 생성할 수 있는 규칙을 찾아내기가 매우 어렵다. 따라서 디코더를 이용하는 접근 방법을 선택함으로써 상황이 어느정도 통제하에 있을 수 있도록 하였다.

4.2 디코더

위와 같은 이유로 디코더를 사용하였는데 쉘코드를 각 바이트의 LSB(Least Significant Bit)에 은닉시키고 디코더가 이를 복원시키는 방법을 구상하였다(〔그림 6~7〕 참조). 디코더의 기능은 매우 단순한데, 이를 통해 얻는 장점은 제작자가 의도한 바대로 디코더를 구성할 수 있다는 데에 있다. 예를 들어 본 연구에서는 eax, ecx, esi, edi 4개의 레지스터를 사용하여 디코더를 구성하였는데 이 사용되는 타이밍과 개수를 제작자가 알고 있기 때문에 더미코드를 생성 시에 이런 점을 고려할 수 있으므로 조작이 쉽다는 것이다. 또한 이런 방식을 택하면 어떤 종류의 쉘코드 던 간에 영향을 받지 않고 은닉시키는 것이 가능해진다. 문제가 매우 다양한 쉘코드들의 은닉에서 공격자가 제작한 디코더가 은닉 가능한가의 문제로 바뀌기 때문이다.

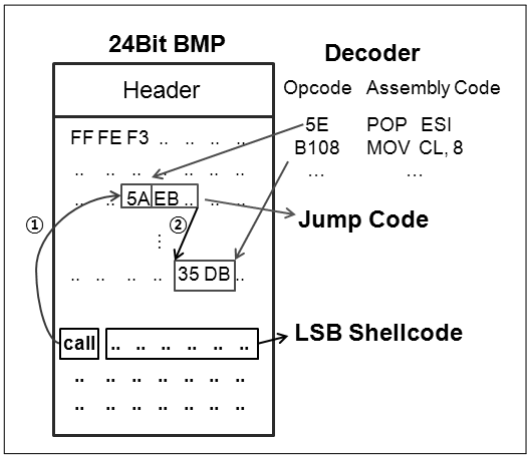
디코더의 기능은 매우 단순하다. 쉘코드가 숨겨진 영역에서 한 바이트 씩 값을 읽어와 쉘코드가 숨겨진 마지막 Bit를 모아 쉘코드를 재조립한다. 이 디코더를 실행시킬 때 쉘코드가 숨겨져 있는 시작 위치를 알아내는 것이 필요한데 CALL 명령어가 복귀 주소를 스택에 저장한다는 점을 이용하였다. CALL 0x0000xxxx(plus offset call) 또는 CALL 0xFFFFxxxx(minus offset call)를 통해 디코더 부분으로 점프하는 것이다. 그리고 디코더에서는 시작하기 전에 pop만 하면 쉘코드가 숨겨진 시작 위치를 얻을 수 있다. 연구결과 일반적으로 그림에는 연속된



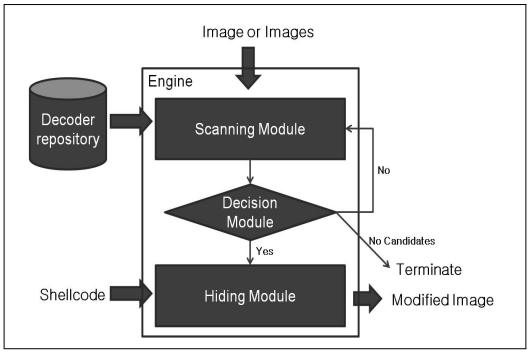
〔그림 6〕 디코더 동작 원리(더미코드)

0x00나 연속된 0xFF가 나타나는 경향이 있는데 이는 각각 흰색, 검은색 계열에 쓰이는 값이기 때문이다. 또한 우리가 필요한 값은 완전 00나 FF의 연속이 아니다. 예를 들어 10 01 FF 란 값이 있다면 각 바이트가 00와 얼마 차이가 안 나기 때문에 조작이 가능하다. 즉, CALL은 E8이므로 E80000xxxx나 E8FFFFxxxx 와 근사 패턴이 나오는 곳을 우선 찾은 뒤 이 뒤에 쉘코드를 LSB에 삽입한다. 〔그림 6〕은 이러한 디코더의 동작 원리를 도식화한 것이다. 최악의 경우에는 픽셀 데이터 첫 부분에 삽입하면 눈에 잘 띄지 않는다(이미지 가장 오른쪽 하단이기 때문에).

디코더를 은닉 시킬 때는 앞서 설명했던 분할 방법을 사용하며 더미코드를 생성한다. 다만 디코더를 작성할 시 디코더를 구성하는 각 기계코드는 2바이트를 넘지 않도록 했다. 언급했듯이 더 많은 바이트가 연속되어야 할수록 이미지 내 데이터에서 더 찾아낼 확률이 낮아지기 때문이다. 더미코드 생성은 1바이트로 된 기계코드 중에서 수정하고자 하는 헥스 값과 가장 유사한 기계코드를 사용하였다(즉 오퍼랜드 부분은 고려하지 않았다). 예를 들어 위에서처럼 POP ESI 이후 더미코드들은 ESI를 수정하는 이외에 대부분의 모든 다른 연산을 수행하는 것이 가능다. 1바이트 기계코드로만 생성 되도록 한 이유는 오퍼랜드를 고려하면 메모리를 참조해야 되는 경우가 생겨 오류 케이스가 너무 많아 사용하지 않았다. 만약 디코더를 삽입할 수 있는 부분이 여러 곳이 발견되면 각 기계코드 사이의 거리가 가장 좁아서 더미코드가 가장 적게 생성될 수 있는 디코더를 선택하였다. 왜냐하면 뒤에 실험결과에서 설명하겠지만 1바이트 기계코드로만은 완전하지



〔그림 7〕 디코더 동작 원리(점프코드)



〔그림 8〕 전체 쉘코드 은닉 프로세스

않아 최대한 더미코드의 양을 줄일 수 있는 방향으로 해야 했기 때문이다.

점프코드의 경우 〔그림 7〕과 같이 각 기계코드 뒤에 점프코드와 오프셋(offset)을 삽입하는 방식을 택했다. 이동해야 되는 오프셋의 간격에 따라 long jump와 short jump를 사용하였다.

4.3 전체 쉘코드 은닉 프로세스

〔그림 8〕은 전체 쉘코드 은닉 프로세스를 나타낸다. 총 4가지의 모듈로 구성이 되어 있으며 각 모듈이 수행하는 작업은 다음과 같다.

① Decoder Repository

디코더가 모여 있는 풀(pool)이다. 본 논문에서 제시한 복호화 방법에도 다양한 패턴의 디코더가 존재할 수 있으며, 다른 복호화 방법도 가능할 수 있다. Decoder Repository는 이런 디코더 후보들을 관리한다.

② Scanning Module

이 모듈은 24Bit 이미지를 입력 값으로 받아 Decoder Repository에 있는 디코더들 중 삽입 가능한 디코더가 있는가를 조사한다. 또한 입력받은 이미지들 중에서도 가능한 이미지가 있는가를 조사한다. Decoder Repository로부터 디코더를 받고 이미지를 검색하여 Decision Module과 통신을 통해 결정 작업을 반복한다. 만약 모든 디코더가 삽입이 불가능하다고 판단되면 프로세스를 종료한다. 디코더 삽입 후 쪼개진 기계코드를 연결하기 위해서 더미코드를 생성하거나 점프코드를 생성한다.

③ Decision Module

Scanning Module과 Hiding Module의 사이에서 통신을 돕는다. 삽입 가능한 디코더가 있다고 판단되면 Hiding Module에게 계속 프로세스를 진행하도록 명령한다. 불가능하면 프로세스를 종료한다. 만약 여러 디코더가 삽입이 가능하다면 가장 거리가 짧은 디코더를 선택한다.

④ Hiding Module

디코더와 쉘코드를 삽입하고 더미코드를 생성하거나 점프코드를 생성하는 작업을 수행한다.

V. 실험 결과 및 평가

본 논문에서 제안하는 은닉 기법은 쉘코드 부분만을 이미지에 은닉시켜 탐지를 우회하여 전달시키는 방법에 대한 것이다. 특정 취약점을 익스플로잇하는 과정에서 EIP를 조작하여 임의의 코드(쉘코드)를 실행할 수 있게 되면 사전에 탐지를 우회하여 타겟 호스트의 메모리에 적재시켜 둔 쉘코드를 실행하게 되는 것이다. 탐지의 주요 대상이 되는 쉘코드를 이미지에 은닉시켜 분리시켰기 때문에 공격자는 공격 시 탐지를 보다 효과적으로 우회할 수 있게 되며 사용하기에 따라 다양하게 응용될 수 있다(문서에 삽입 등). 이는 공격자가 공격 시에 좀 더 다양한 공격 경로를 택할 수 있는 기회를 제공하게 된다.

본 논문에서는 익스플로잇 방법이 아닌 쉘코드의 난독화 방법으로서 이미지에 쉘코드를 삽입하는 방법을 제안하고 있으며, 이에 따라, 쉘코드가 은닉된 이미지의 성능평가를 위해 다음과 같은 세가지 항목을 평가 하였다.

① 육안을 통한 식별 및 동작 테스트
② NIDS(Network based Intrusion Detection System)를 통한 시그니처 기반 쉘코드 탐지 우회
③ 에뮬레이션 기반 쉘코드 탐지 우회

5.1 육안을 통한 식별 및 동작 테스트

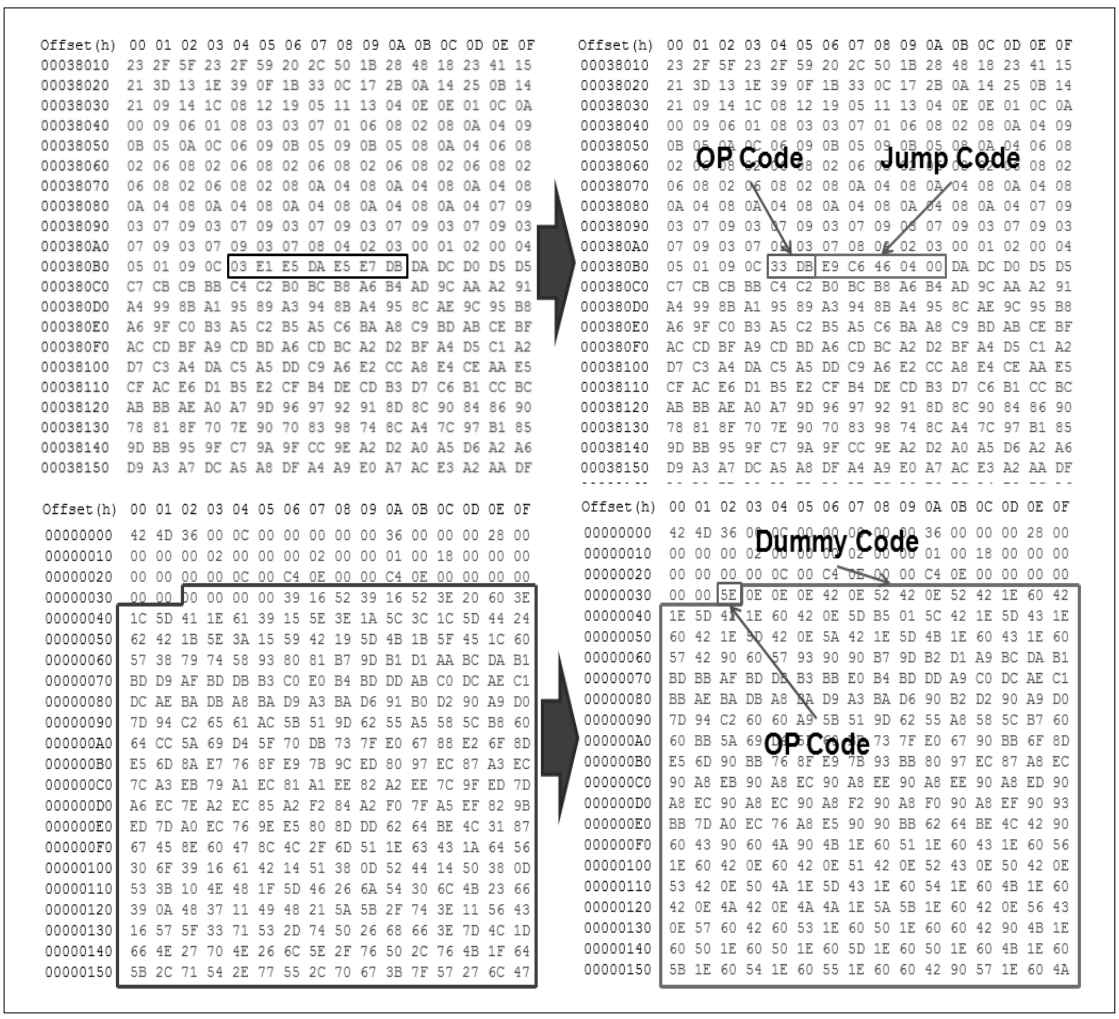
여기서는 원본 이미지와 실행가능한 상태의 쉘코드가 실제로 은닉된 이미지를 비교해본다. 실험환경은 Windows XP이며, 쉘코드를 은닉시키는 엔진은 파이썬(Python)을 사용하여 제작되었다. 쉘코드는 Metasploit에서 배포하는 270바이트 Message-Box API Call을 하는 코드로 다소 길이가 있는 쉘코드도 은닉이 가능하다는 점을 보이기 위해 사용하였다. 〔그림 9〕는 원본 이미지, 점프코드 생성 이미지, 더미코드 생성 이미지를 나타낸다.

〔그림 9〕를 유심히 보면 점프코드 생성 이미지는 점이 찍혀서 픽셀이 튀는 것을 확인할 수 있고 더미코드 생성 이미지에서는 이미지 하단에 다른 색감의 층이 생성된 것을 볼 수가 있다. 점프코드의 경우에 이동해야 되는 오프셋에 따라 short jump와 long jump를 선택하였는데 long jump를 사용하면 총 5바이트인데 헥스 값으로는 E9XXXXXXXX이다. 이 XXXXXXXXXX에 다음 기계코드까지의 오프셋이 들어가게 되어 5바이트가 수정되게 되기 때문에 색이 튀게 되는 경우가 발생한다(2바이트인 short jump에 비해서도 더 튀게 된다). 기계코드 값과 점프코드 값을 모두 고려하여 최대한 근사한 패턴을 찾도록 하였다. 색이 튀지 않는 부분은 연속된 기계코드와 점프코드가 색에 잘 녹아 들어갔기 때문이지만 색이 튄 부

〔그림 9〕 원본 이미지(왼쪽), 점프코드 생성 이미지(가운데), 더미코드 생성 이미지(오른쪽)

분은 그렇지 못한 부분이다. 디코더를 삽입할 때 마지막 기계코드부터 처음 기계코드까지 역순으로 삽입해 나가는 방식을 택하였는데 그래야 점프코드가 다음 기계코드와의 오프셋을 계산할 수 있기 때문이다. 그러나 이런 이유로 역방향으로 점프코드에 근사한 부분을 찾으면서 변화하는 오프셋에 또한 유사한 부분을 찾아야하기 때문에 검색이 어렵다. 때문에 완전히 모든 점프코드를 완벽하게 숨기기가 어렵기 때문에 색이 튀는 부분이 발생하게 되었다. 〔그림 10〕에 점프코드 삽입 예가 나타나 있다.

1바이트 더미코드 생성 시에 헥스 값 0xC0부터 0xFF까지는 대부분 특권 명령(privileged instruction) 또는 더미용으로는 사용할 수 없는 기계코드라서 사용할 수 없었다. 때문에 저 대역에 해당하는 헥스 값들은 0xB0와 같은 식으로 모두 작은 값을 사용해야만 했고, 다른 색감의 층이 생성되게 된 것이다(살색 계열의 경우 RGB값이 0xFF 근처의 값으로 이루어져 있다). 때문에 더미코드 생성의 경우 이미지에 따라 전혀 티가 나지 않기도 하고 위와 같이 층이 생성되기도 한다. 〔그림 9〕를 보면 점프코드 생성 이미지의 경우 왼쪽 하단 아래에 파란 점이 있는 것을 볼 수가 있는데 그 이외에는 튀는 부분은 보이지 않는다. 이 이미지 같은 경우 디코더의 각 기계코드와 점프코드 헥스 값이 잘 맞아 매우 결과가 잘 나온 케이스이다. 그러나 더미코드 생성의 경우 하단에 눈에 띄게 색감이 변한 것을 알 수가 있다. 따라서 이 경우에는 더미코드 생성보다 점프코드 생성 방식이 보다 유용할 것으로 보인다. 〔그림 10〕에 더미코드 삽입 예가



〔그림 10〕 점프코드 삽입(상)와 더미코드 삽입(하)의 예. 원본 이미지(좌), 수정된 이미지(우)

〔그림 11〕 쉘코드 실행 화면

나타나 있다.

〔그림 11〕은 이미지에 은닉된 쉘코드가 실행되는 화면이다. 테스트 프로그램을 작성하였으며 'CALL+쉘코드가 숨겨져 있는 곳의 시작 주소'와 이미지를 이 프로그램의 파라미터로 넣고 실행하면, 런타임에 암호화가 디코더에 의해 풀리면서 MessageBox가 실행되는 것을 확인할 수 있다.

본 논문 실험 결과에서는 원본 이미지와 비교를 해서 제시하였지만 실제로 이미지를 배포하거나 전송할 때는 해당 이미지만 사용하기 때문에 구분이 더욱 어렵다. 또한 같은 그림이라도 이미지가 작을 경우 또는 픽셀수가 많을수록(해상도가 높을수록) 구분이 더욱 어려워진다. 상대적으로 조작된 값이 미치는 영향이 더 적어지기 때문이다.

5.2 NIDS를 통한 시그니처 기반 쉘코드 탐지 우회

관련 연구에서 언급한 것과 같이, 시그니처 기반 쉘코드 탐지기법은 0x90으로 이루어진 NOP Sled나 원격 쉘을 띄우기 위한 "/bin/sh"라는 문자열, 또는 시스템 콜 패턴 등의 쉘코드가 보이는 특정한 패턴이나 특징을 시그니처화하여 네트워크 패킷이나 프로세스에 들어오는 입력 값에 이런 패턴이 발견되는 지 검사하는 방법이다. 패턴기반 탐지는 과부하가 적어 속도가 빠르고 효율이 비교적 좋은 편이기 때문에 현재까지도 널리 사용되고 있는 방법이다.

여기서는 Snort를 사용해 시그니처 기반의 탐지기법에 대한 탐지율을 확인해 보겠다. Snort는 실시간 트래픽 분석과 IP 네트워크상에서 패킷 로깅이 가능한 네트워크 기반 침입 탐지 시스템으로, 〔그림 12〕와 같은 룰(rule)을 사용하여 탐지를 수행한다. 실험에서는 Snort 공식 페이지에서 배포하는 snort-rules-snapshot-2921를 사용하였다. 해당 파일은 여러 공격을 탐지하기 위한 룰들을 담고 있으며 쉘코드 관련 탐지 룰들도 포함되어 있다.

실험을 위해 Metasploit를 사용하여 작성한 쉘코드 5개를 작성하였으며, ① 난독화를 적용하지 않은 쉘코드, ② AlphaNumeric 난독화 적용 쉘코드, ③ 이미지에 은닉된 쉘코드의 세가지 카테고리를 기준으로 각각의 탐지율을 비교하였다. AlphaNumeric 난

```
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE x86 OS agnostic unicode mixed encoder"; content:"YAZBABABABABkMAGB9u4JB"; fast_pattern:only; metadata:policy balanced-ips drop, policy security-ips drop; classtype:shellcode-detect; sid:19287; rev:1;)
```

〔그림 12〕 Snort의 쉘코드 탐지 Rule 예제

〔표 3〕 Snort와 libemu를 사용한 쉘코드 탐지 테스트 결과

탐지 횟수/샘플 수 (탐지율%)

Obfuscation	Snort	libEmu
Raw	5/5 (100%)	5/5 (100%)
Alpha_1	5/5 (100%)	5/5 (100%)
Alpha_2	0/5 (0%)	5/5 (100%)
Image obfuscation	**0/5 (0%)**	**0/5 (0%)**

독화는 Skylined's alphanumeric encoder와 Skylined's Alpha2 alphanumeric encoder를 사용하였다.

실험 수행 결과는 〔표 3〕과 같다. 난독화를 하지 않은 경우는 모두 탐지 되었으며, alphanumeric encoder의 경우도 모두 탐지 되었으나, Alpha2 alphanumeric와 이미지에 은닉된 쉘코드의 경우 탐지되지 않았다. Alpha2 alphanumeric의 경우 폴리몰픽 기술이 디코더에 적용이 되어 있기 때문에 탐지가 되지 않은 것으로 보인다.

5.3 에뮬레이션 기반 쉘코드 탐지 우회

난독화 기법을 사용한 쉘코드는 기존의 시그니처 기반의 탐지 기법을 사용한 탐지로는 한계가 있어서, 시그니처 기반의 정적인 방법이 아닌 동적인 방법이 제시 되고 있다. 본 논문에서는 동적 탐지 방법 중, CPU 에뮬레이션을 기반으로 한 쉘코드 탐지기법을 사용하여 탐지율을 테스트 하였다.

에뮬레이션 기반의 쉘코드 탐지기법 중에 Polychronakis이 제안한 방법과 Zhang이 제안한 방법〔21〕이 있다. Polychronakis이 제안한 방법은 난독화 기법이 적용된 쉘코드를 탐지하기 위해, CPU 에뮬레이션을 사용해 모든 인스트럭션 시퀀스를 실행해 보는 방법이다. 이 방법은 난독화된 쉘코드를 찾을 수 있으나, 매 바이트를 모두 실행하므로, 오버헤드가 너무 크다는 단점이 있다. 이러한 오버헤드 문제를 해결하기 위해 제안된 방법이 Zhang의 탐지 기법으로, 정적 분석을 통해 쉘코드에서 Get_PC 코드를 찾은 뒤, 그 코드를 기준으로 주변 인스트럭션을 에뮬레이션 하는 방법이다.

본 실험에서는 허니팟(honeypot) 소프트웨어인 nepenthes, dionaea에서 폴리몰픽 쉘코드 탐지를 위한 에뮬레이션에 사용하는 libEmu를 사용하였다. libEmu는 Zhang이 제안한 방법인 Get_PC 휴리스틱 탐지를 지원하며, dionaea등에서는 이 방법을 사용하여 탐지를 수행하고 있다.

실험을 위한 쉘코드 샘플은 앞의 시그니처 기반 탐지에서 사용한 것과 동일하게, 5개의 쉘코드에 난독화를 적용하여 준비하였으며, 실험 수행 결과는 〔표 3〕과 같다.

실험을 수행하기 전에 예상하기로는 에뮬레이션을 사용한 방법에는 모두 탐지가 될 것으로 예상하였으나, 예상과 다르게 Get_PC 휴리스틱 탐지에서 해당 코드를 탐지 하지 못하였으며, 쉘코드의 존재를 인식하지 못하였다. 그러나, 탐지과정에서 쉘코드를 디코딩 하는 코드의 시작 지점을 임의로 지정해준 경우, 쉘코드를 탐지하는 것을 확인 할 수 있었다. 따라서, 오버헤드의 발생이 문제가 되지 않는 상황이라면 한 바이트씩 모든 인스트럭션을 실행해 보는 방법으로 탐지가 가능할 것이다.

탐지 실험 결과에서 알 수 있듯이 이미지에 쉘코드를 은닉시킴으로써 얻을 수 있는 이득은 다음과 같다. 변조된 이미지만 가지고는 이미지가 변조되었는지 여부를 알기가 어려우며, 값이 특정부분만 변조되었기 때문에 전체 통계적으로 정상에 가깝다. 쉘코드는 LSB에 은닉되어 있고 디코더는 쪼개져 있기 때문에 이미지에서 어떤 패턴을 찾아내 패턴기반으로 탐지하는 것 역시 매우 어렵다. 다만 쉘코드를 LSB 방식으로 숨기고 디코더를 은닉시키는 방식에 있어서 스테가노그래피(steganography)적인 요소가 있기 때문에 스테가노그래피를 탐지하는 방법론으로 탐지될 우려가 있으며, 에뮬레이션 기법 등을 이용한 동적 탐지에도 탐지될 수 있다. 그러나 스테가노그래피는 숨겨진 메시지가 있는지 여부를 판단하기 위한 것이기 때문에 적합한 탐지 방법은 아니며 아무런 사전 정보 없이는 메시지가 숨겨져 있는 것을 안다고 하더라도 그것이 쉘코드인지는 알아낼 수가 없다. 게다가 현재 IDS나 WAF 등과 같은 보안 장비들에서는 스테가노그래피가 메시지를 숨기는 것일 뿐 어떤 공격의 위협요소가 되진 않기 때문에 주요하게 탐지를 하지 않는다. 에뮬레이션 기법의 경우 실험결과에서 보였듯이 우회될 수

있는 가능성 또한 있음을 알 수 있다. 게다가 단순한 패턴기반 탐지가 아닌 이런 보다 심화된 탐지 작업들은 오버헤드가 크기 때문에 쉽게 이용될 수 없는 측면도 있다. 때문에 본 논문에서 제시하는 기술이 효용성을 가질 수 있다고 판단된다.

VI. 결 론

본 논문에서는 24Bit BMP 이미지 파일에 쉘코드를 실행 가능한 상태로 은닉시키는 방법론에 대해 제안하였다. 디코더를 은닉시키면 이를 통해 쉘코드를 은닉시킬 수 있으며, 더미코드 생성 방식과 점프코드 생성 방식을 사용할 수 있다. 성능적인 면에서 보면 이미지마다 다소 차이를 보인다. 점프코드 방식의 경우 오프셋 때문에 특정 픽셀이 튀어 보이는 현상이 발생하였지만 이미지에 따라서는 육안으로 구분이 어려울 수 있다. 구분이 잘 안 되는 경우 점프코드 방식이 더 효과적이었으며, 더미코드 방식의 경우 특권 명령을 사용할 수 없어 색 층이 생기는 문제가 발생하였지만 이도 역시 분량 문제로 다 포함은 시키지 못했으나 이미지에 따라 육안으로 거의 구분이 불가능한 경우가 많았다. 특히 두 방식 모두 원본 없이 수정된 이미지만 받아 봤을 때는 더 분간이 어려웠다. 본 논문에서 제시한 은닉 기법의 성능을 기존의 쉘코드 은닉 기법과 비교하기 위해서, Snort를 사용한 시그니처 기반의 탐지 방법과, libEmu를 사용한 에뮬레이션 방법을 사용하여 탐지율을 측정 하였다. 시그니처 기반의 탐지방법은 이미지에 쉘코드를 숨기는 과정에서 쉘코드의 시그니처 정보가 사라지므로 탐지가 되지 않았다. 에뮬레이션 방법의 경우 예상과 달리 탐지가 되지 않았는데, 이는 Zhang이 제안한 방법인 Get_PC 휴리스틱 검사를 우선 수행하는 방법을 사용한 경우이며, 코드 전체에 대한 검사를 수행할 경우 방법에 따라 탐지가 가능할 것으로 보이나, 소요되는 검사시간이 길어 실시간으로 탐지하는 것은 어려울 것으로 보인다.

본 논문에서 연구된 결과들은 충분히 쉘코드가 적재된 상태로 사용자를 속여 공격에 사용될 수 있는 가능성을 제시했다는 점에서 의미가 크며, 본 논문에서 제안한 기법은 추후 알고리즘 개선을 통해 은닉 성능이 향상될 수 있다. 향후 연구로는 오퍼랜드까지 고려한 2바이트 이상의 더미코드 생성하는 방법을 계획하고 있다. 점프코드는 오프셋 때문에 픽셀이 튈 수밖에 없기 때문에 보다 더미코드를 잘 생성하는 것이 쉘코드를 보다 완벽하게 은닉시키는 방법이라고 생각된다. 2바이트 이상 더미코드를 생성이 가능해진다면 약간의 값 조작을 통해 보다 자연스러운 이미지를 생성하는 것이 가능할 것이라 예상된다.

참고문헌

〔1〕 J. Lee, "History of Buffer Overflow," Hacker School, 2008, http://www.hackerschool.org/HS_Boards/data/Lib_system/History_of_Buffer_Overflow.pdf

〔2〕 A. One, "Smashing The Stack For Fun and Profit," Phrack, Vol.7, Nov. 1996, http://www.phrack.org/issues.html?issue=49&id=14

〔3〕 "Vulnerability distribution of cve security vulnerabilities by types," http://www.cvedetails.com/vulnerabilities-by-types.php

〔4〕 B. Martin, M. Brown, A. Paller and D. Kirby, "2011 CWE/SANS Top 25 Most Dangerous Software Errorsh," Common Weakness Enumeration, Sept. 2011, http://cwe.mitre.org/top25/

〔5〕 J. Ma, J. Dunagan, H. J. Wang, S. Savage and G. M. Voelker. "Finding Diversity in Remote Code Injection Exploits," Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, pp. 53-64, Oct. 2006.

〔6〕 http://www.snort.org/snort

〔7〕 M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network-level Polymorphic Shellcode Detection using Emulation," Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment, Vol. 4064, pp. 54-73, 2006

〔8〕 M. Polychronakis, K. G. Anagnostakis and E. P. Markatos, "Emulation-based Detection of Non-self-contained Polymorphic Shellcode," Proceedings of the International Symposium on Recent Advances in Intrusion Detection, pp. 87-106, Sep. 2007

[9] B. Gu, X. Bai, Zh. Yang, A. C. Champion and D. Xuan, "Malicious Shellcode Detection with Virtual Memory Snapshots," Proceedings of the IEEE INFOCOM, pp. 974-982, 2010

[10] H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu and D. Boneh, "On the Effectiveness of Address Space Randomization," Proceedings of ACM Conference on Computer and Communications Security, pp. 298-307, Oct. 2004

[11] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, "Stackguard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks," Proceedings of the USENIX Security Symposium, pp. 63-78, Jan. 1998

[12] A. N. Sovarel, D. Evans and N. Paul, "Where's the FEEB? On the Effectiveness of Instruction Set Randomization," Proceedings of the USENIX Security Symposium, Vol. 14, pp. 10, Aug. 2005

[13] http://www.metasploit.com/about/

[14] S. Macaulay, "ADMMutate: Polymorphic Shellcode Engine", http://www.ktwo.ca/security.html

[15] T. Detristan, T. Ulenspiegel, Y. Malcom and M. S. van Underduk, "Polymorphic Shellcode Engine Using Spectrum Analysis," Vol. 11, Phrack, Aug. 2003, http://www.phrack.org/issues.html?issue=61&id=9

[16] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis and S. J. Stolfo, "On the Infeasibility of Modeling Polymorphic Shellcode," In Proceedings of ACM Conference on Computer and Communications Security, pp. 541-551, Oct. 2007.

[17] Rix, "Writing IA32 Alphanumeric Shellcode," Phrack, Vol. 11, Aug. 2001, http://www.phrack.org/issues.html?issue=57&id=15

[18] Obscou, "Building IA32 Unicode-Proof Shellcodes," Phrack, Vol. 11, Aug. 2003, http://www.phrack.org/issues.html?issue=61&id=11

[19] T. Wana, "Writing UTF-8 Compatible Shellcodes," Phrack, Vol. 11, Sep. 2004, http://www.phrack.org/issues.html?issue=62&id=9

[20] J. Mason, S. Small, "English Shellcode," Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 524-533, Nov. 2009.

[21] Q. Zhang, D.S. Reeves, P. Ning and S.P. Lyer, "Analyzing Network Traffic to Detect Self-decrypting Exploit Code," Procddeings of the ACM Symp. on Information, Computer and Commun. Security, pp. 4-12 , Mar. 2007.

〈著者紹介〉

금 영 준 (Young Jun Kum) 학생회원
2008년 2월: 서울과학기술대학교 토목공학과 졸업
2012년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 시스템 해킹, 온라인 게임 보안, 데이터 마이닝, 네트워크 포렌식

최 화 재 (Hwa Jae Choi) 학생회원
2011년 2월: 고려대학교 컴퓨터통신공학부 졸업
2011년 3월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 게임 보안, 시스템 해킹, 웹 해킹

김 휘 강 (Huy Kang Kim) 종신회원
1998년 2월: KAIST 산업경영학 학사
2000년 2월: KAIST 산업공학과 석사
2004년 5월~2010년 2월: NC소프트 정보보안실장, Technical Director
2010년 3월~현재: 고려대학교 정보보대학원 조교수
〈관심분야〉 온라인게임 보안, 네트워크 보안, 네트워크 포렌직, 침입탐지시스템, 봇넷탐지