情報保護學會論文誌
第19卷 第5號, 2009. 10

메모리 실행영역 추적을 사용한 버퍼오버플로 악성코드 탐지기법*

최 성 운,† 조 재 익, 문 종 섭‡
고려대학교 정보경영공학전문대학원

Buffer Overflow Malicious Code Detection by Tracing Executable Area of Memory*

Sung-woon Choi,† Jae-ik Cho, Jong-sub Moon‡
Graduate School of Information Management and Security

요 약

버퍼오버플로 악성코드 탐지를 위해 대부분의 안티바이러스 프로그램은 공격코드의 시그너처만 비교·탐지하고 있어 알려지지 않은 공격코드에 대해 탐지하지 못하는 문제점이 있다. 본 논문에서는 공격코드에서 필수적으로 사용하는 API의 메모리 실행영역 추적기법을 이용하여 알려지지 않은 공격코드에 대한 탐지기법을 제안한다. 제안기법 검증을 위해 7개의 샘플 공격코드를 선정하여 8개의 안티바이러스 프로그램과 비교 실험한 결과, 대부분의 안티바이러스 프로그램은 Stack영역만 감시하고 Heap영역은 감시하지 않아 제안적인 탐지만 가능하였다. 이에 대부분의 안티바이러스 프로그램에서 탐지할 수 없는 공격코드를 제안 기법을 이용하여 탐지할 수 있음을 시뮬레이션 하였다.

ABSTRACT

Most of anti-virus programs detect and compare the signature of the malicious code to detect buffer overflow malicious code. Therefore most of anti-virus programs can't detect new or unknown malicious code. This paper introduces a new way to detect malicious code traces memory executable of essentials APIs by malicious code. To prove the usefulness of the technology, 7 sample codes were chosen for compared with other methods of 8 anti-virus programs. Through the simulation, It turns out that other anti-virus programs could detect only a limited portion of the code, because they were implemented just for detecting not heap areas but stack areas. But in other hand, I was able to confirm that the proposed technology is capable to detect the malicious code.

Keywords: Buffer Overflow, API, Anti-Virus, NOP Detect

I. 서 론

악성코드의 공격과정을 살펴보면 크게 2단계로 나눌 수 있다. 첫 번째 단계는 OS 또는 어플리케이션 취약점을 이용하여 시스템 권한 획득을 위한 공격코드가 실행되는 단계이고 그 다음은 봇넷, 개인정보유출, 키로그, 악성 웜 등 시스템 제어를 위한 공격코드 실행단계이다[1]. 첫 번째 단계에서 사용하는 공격코드의 취약점 유형에는 버퍼오버플로, 힙오버플로, 자동 업데이트시 인증절차 취약점, 프로토콜 설계 오류 등이 있는데 그 중 80% 이상이 버퍼오버플로 유형의 취약점을 이용한다[2].

본 논문에서는 버퍼오버플로 악성코드가 사용하는

접수일(2009년 4월 28일), 수정일(1차: 2009년 6월 4일, 2차: 2009년 7월 17일), 게재확정일(2009년 8월 28일)
* 본 연구에 참여한 연구자(의 일부)는 '2단계 BK21사업'의 지원비를 받았음.
† 주저자, monocat2@hanmail.net
‡ 교신저자, jsmoon@korea.ac.kr

API의 메모리 실행 영역을 추적하여 공격코드 여부를 탐지하는 기법을 제안한다. 이를 위해 win32환경의 버퍼오버플로 공격코드에서 사용하는 API를 분석하여 메모리 실행 영역을 추적하는 메커니즘을 구현하고 시뮬레이션 하였다. 본 논문은 2장에서 안티바이러스 프로그램의 탐지기법과 win32환경에서의 버퍼오버플로 및 탐지 기법에 대한 기존 연구를 분석하고 3장에서는 API호출 단계의 메모리 분석을 위해 공격코드에서 필수적으로 사용되는 API를 선정하고 2단계의 메모리 실행영역 추적기법을 제시한다. 4장에서는 샘플로 선정한 7개의 악성코드를 이용하여 기존 안티바이러스 프로그램과 비교 실험하고 마지막 5장에서는 결론 및 문제점을 분석하여 향후 연구 과제를 논의한다.

II. 관련연구

2.1 안티바이러스 프로그램의 버퍼오버플로 악성코드 탐지기법

안티바이러스 프로그램에서 버퍼오버플로 악성 코드 탐지를 탐지하기 위한 기법은 정적 탐지기법과 동적 탐지기법이 있다[3]. 정적 탐지기법은 악성코드 내용 중 2~3곳의 문자열을 데이터베이스화 하여 대상 코드와 데이터베이스를 단순 비교 후 악성코드 여부를 탐지하는 기법으로 빠른 속도, 비교적 정확한 탐지능력의 장점이 있으나 우회기법을 적용한 변종 악성코드에 대한 탐지가 불가능한 단점이 존재한다. 이를 보완하기 위한 동적 탐지기법은 "웹 트래픽 분석"[4], "프로그램 은닉기법 분석"[5], "메모리 분석기법"[6], "행위탐지 기법"[7] 등을 이용해 구현되어 있으며 정적 탐지기법에 비해 알려지지 않은 악성코드에 대한 탐지가 가능한 장점이 있으나 "오탐 가능성", 실행 프로그램에 대해 신뢰여부 확인 등으로 인한 "사용 편의성 제한", "느린 속도"와 같은 단점이 있다.

2.2 Win32 환경에서의 버퍼오버플로 및 탐지 기법

버퍼오버플로 공격 기법은 프로그램 개발 시 지역변수의 경계 체크(Bounds Check)를 하지 않아 사용자가 입력한 임의의 값으로 버퍼를 넘치게 하여 함수의 리턴주소를 원하는 주소로 변경시켜 악성코드를 실행하게 하는 공격 기법이다[8]. 리턴주소를 악성코드가 있는 주소로 정확하게 변경하기 위해 Win32환경에서는 "jmp register"[9], "SEH(Structured Exception Handler) 기법"[9] 등을 이용한다.

버퍼오버플로 공격기법의 악성코드는 특정 명령수행을 위해 16진수 형태의 코드를 사용하는데 이를 쉘코드라 부른다. 쉘코드는 고급언어로 제작된 코드를 바탕으로 기계어 코드를 얻어내고 다시 그것을 16진수 형태의 명령어 코드를 만드는 방식으로 제작하기 때문에 OS와 CPU에서 제공하는 기계어에 따라 제작방법이 다르다. Win32환경에서의 쉘코드는 API의 주소를 직접 호출하는 방식으로 작성되기 때문에 쉘코드 작성의 핵심은 API 주소를 검색하는 과정이며 이 과정을 반복하여 API의 주소 검색 후 해당 API를 호출하여 사용한다[10].

2.2.1 메모리 실행영역 추적을 이용한 버퍼오버플로 탐지기법

메모리 실행영역 분석을 이용한 버퍼오버플로 탐지기법에는 "API 후킹을 이용한 차단기법"[11]과 "OOB Object를 이용한 추적기법 "[12]등이 있다. API 후킹을 이용한 차단기법은 쉘코드 동작 과정 중 "API 주소 검색" 과정에서 주소 검색을 위해 사용되는 TIB[13]와 PEB[14]주소를 수정하여 API 주소 검색을 차단하는 기법이다. 그러나 TIB와 PEB 주소의 변경 여부를 탐지하고 재검색하는 루틴을 쉘코드에 적용시키면 우회가 가능한 문제점이 있다.

OOB(Out-Of-Bounds) 오브젝트를 이용한 추적기법은 모든 변수의 메모리 할당 영역을 별도의 오브젝트 테이블에 저장하고 오브젝트 테이블에 저장된 변수 값을 실시간 감시하여 공격 여부를 판단하는 기법이다. 이 방법은 메모리 할당 빈도수가 적은 소규모 응용 프로그램에서는 효용성이 있으나 win32환경과 같이 다수의 dll과 API 호출이 발생하고 각각의 dll마다 별도의 메모리 영역을 할당하는 프로그램에서는 많은 부하가 발생하는 문제점이 있다.

2.2.2 NOP를 이용한 버퍼오버플로 탐지기법

NOP코드는 No Operation코드의 약자로 아무것도 수행하지 않는 무연산 코드를 말한다. x86 CPU 환경에서 NOP코드는 0x90로 정의되어 있는데, 버퍼오버플로 공격 시 성공률을 높이기 위해 다량의 NOP 코드를 쉘코드 주위에 저장한다. NOP를 이용한 버퍼오버플로 탐지는 Snort[4], Fnord[15]와 같은 침입탐지시스템에서 주로 사용한다. Snort는 0x90이

다량으로 사용되면 버퍼오버플로 공격으로 탐지하고, Fnord는 0x90이외에 55가지의 NOP 코드를 추가하여 SNORT와 동일한 방법으로 공격을 탐지한다. 그러나 이들 방법은 정의되지 않은 NOP코드를 사용하여 공격코드에 적용할 경우 탐지하지 못하는 문제점이 있다.

III. 버퍼오버플로 탐지를 위한 API호출 단계의 메모리 실행영역 추적기법

정상 프로그램에서 API를 호출할 경우 현재 스레드의 Code Segment 영역에서 실행되지만 버퍼오버플로 공격코드에서는 스택, 힙[16]과 같은 비실행 영역에서 API호출이 발생한다[9]. 본 논문에서는 버퍼오버플로 악성코드 탐지를 위해 2 단계로 메모리 실행 영역을 추적하여 정상 프로그램인지 공격 코드인지를 판별한다. 1 번째 단계에서 API의 실행영역을 추적하여 판별하고 1번째 단계에서 판별하지 못한 코드를 2 번째 단계에서 NOP Stamp기법을 적용하여 한번 더 공격코드 여부를 판별한다.

3.1 Win32 쉘코드에서 사용하는 필수 API

win32환경에서 대표적으로 사용되는 쉘코드는 [표 1]과 같이 1.kernel32.dll의 base주소 검색 2. 필수 API주소 검색 3.실제 API 구동을 위한 dll 적재 4.실제 동작 구현의 순서로 실행된다[10]. 4단계 과정 중 2.필수 API주소 검색과 3.실제 API구동을 위한 dll 적재과정은 dll을 현재 프로세스의 주소 공간에 적재시키는 "LoadLibraryA()"API와 DLL내 원하는 API의 실행주소를 가져오는 "GetProcAddress()"API를 이용한다.

[표 1] Admin 유저 생성 쉘코드 실행 순서 분석

단계	API 호출	설명
kernel32.dll 주소 찾기	-	PEB를 이용하여 base address 검색
kernel32.dll 내 API 주소 검색	GetProcAddress()	LoadLibraryA(), CreateProcessA(), WaitForSingleObject(), ExitProcess()의 주소 검색
netapi32.dll 적재	LoadLibraryA()	NetUserAdd()와 NetLocalGroupMembers()가 저장된 netapi32.dll 파일을 메모리에 적재
실제 동작 구현 (Admin 유저 생성)	NetUserAdd() NetLocalGroupMembers()	Admin 유저 생성을 위한 API

버퍼오버플로 유형의 악성코드에서 2가지 과정이 필수적으로 사용되며 본 논문에서는 2개의 API 중 LoadLibraryA()함수를 추적할 API로 선정하여 3.2절에서 탐지 루틴을 적용한다.

3.2 메모리 실행영역 추적기법

3.1장에서 선정한 LoadLibraryA() API의 메모리 실행영역을 추적하여 분석하는 기법은 크게 2단계로 나누어져 있으며 각 단계는 다음과 같다. 첫 번째 단계는 3.2.1장과 같이 LoadLibraryA() API가 스택 또는 힙 영역에서 실행되고 있는지 분석하여 공격코드로 판단하고 두 번째 단계에서는 3.2.2장과 같이 쉘코드 주위에 저장된 NOP코드의 빈도수를 파악하여 공격코드 여부를 탐지하는 NOP Stamp기법을 사용한다.

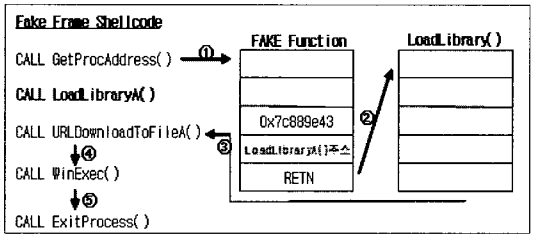
3.2.1 API 호출 메모리 실행영역 분석

실행중인 스레드에 생성된 스택영역은 OS가 제공하는 TIB(Thread Information Block)에 저장된 "Top of Stack"과 "Current Bottom of Stack"값을 이용해 파악할 수 있는데 이는 FS레지스터 내 오프셋값 0x04와 0x08을 이용하여 접근이 가능하다. 그리고 스레드에 생성된 힙 영역은 PEB(Process Environment Block)에 저장된 "Process Heaps" 값을 이용해 파악할 수 있다. PEB에 접근하기 위해선 먼저 FS레지스터 내 오프셋값 0x18을 이용하여 TIB에 접근후 오프셋값 0x30에 위치하는 PEB 구조체에 접근하고 다시 PEB 구조체의 오프셋값 0x90을 이용하여 생성된 Heap Block을 판단할 수 있다. 따라서 응용프로그램 실행시 LoadLibraryA() API의 리턴주소 메모리 영역이 현재 스레드에서 생성한 스택 또는 힙 영역에 포함될 경우 버퍼오버플로 유형의 공격코드라고 탐지할 수 있다. 기존에 연구된 "API 후킹을 이용한 차단기법"과 "OOB Object를 이용한 추적기법"은 모든 변수 또는 API에 대한 감시를 수행하지만 제안 기법은 감시할 API를 LoadLibraryA()로 한정하고 별도의 오브젝트 테이블을 생성하지 않아 기존 연구방식에 비해 메모리 사용량 및 시스템 부하를 줄일 수 있는 이점이 있다.

3.2.2 NOP Stamp 메모리 분석

일부 발전된 공격코드에서는 쉘코드가 사용하는 특정 API를 은닉하기 위해 Fake Frame 우회기법을 적용한다. Fake Frame 우회기법이란 [그림 1]에서 볼 수 있듯이 1.GetProcAddress() 2.LoadLibraryA() 3.URLDownLoafToFileA() 4.WinExec() 5.ExitProcess() 순서로 진행되는 쉘코드에서 2. LoadLibraryA()함수를 은닉시킬 경우 1. GetProcAddress()호출 후 2. LoadLibraryA()대신 Fake Function()을 호출하여 1.GetProcAddress() 2.Fake Function() 3.URLDownLoafToFileA() 4.WinExec() 5.ExitProcess() 순서로 쉘코드를 작성하는 기법이다. 이때 2.Fake Funtction()은 아무런 동작을 하지 않지만 자신이 종료될 때 사용되는 Epilog Frame의 리턴주소를 LoadLibraryA()함수 주소로 조작하여 결과적으로 LoadLibraryA()함수를 실행시킨다.

이 기법을 적용한 쉘코드는 API 후킹 시 정상적인 콜이 아니기 때문에 3.2.1장에서 제시한 메모리 탐지기법을 적용할 수 없다. 그러나 기 기법이 적용된 공격코드의 경우 공격 성공률을 높이기 위해 다량의 NOP 코드가 쉘코드 주위에 저장되는 점을 착안하여 NOP Stamp 메모리 분석기법을 제안한다. 2.2.2장에 설명한 NOP 탐지기법은 Heap Spray 공격에서 사용하는 0x0d0d, 0x0a0a, 0x0505, 0x0c0c와 같은 NOP코드는 정의되어 있지 않아 탐지하지 못한다. 이들 코드는 "XOR EAX,0505"와 같은 기능을 수행하는 코드로 쉘코드 실행에는 전혀 지장을 주지 않기 때문에 NOP코드로 대체 사용이 가능하다. 이때 사용되는 NOP코드의 공통된 특징은 동일 문자열이 연속적으로 메모리에 삽입된다는 점이다. [표 2]는 쉘코드 주위에 다량의 NOP가 존재하는지 검색하는 알고리즘으로 LoadLibraryA()함수의 리턴주소 주변의 random주소에서 동일 문자열 사용 빈도수(count변수)를 파악하는 기능을 수행한다. 사용 빈도수 파악 시 고정된 주소값에서 비교 루틴을 적용하면 공격자는 비교주소의 위치를 파악하여 우회할 가능성이 존재하여 무작위로 주소값을 추출하도록 구현하였다.



[그림 1] Fake Frame 쉘코드 개념

[표 2] NOP Stamp 기법

```
__asm {
   eax ← [esp]
   return_address ← eax
   // LoadLibraryA()의 리턴주소
   ecx ← 0x100                    // 반복 카운트
   for i ← 1 to ecx {
   // 동일 문자열 사용빈도 검색
      ebx ← random_number ( 100 ~ 1000)
      eax[i] ← [return_address - ebx]
      //리턴주소 주변의 문자열 저장
      cmp eax[i-1], eax[i]
      je count
   }
count:
   inc count
}
if (count > n)  // n = 동일 문자열 사용 빈도수
   Display "Detect NOP Stamp"
```

IV. 제안기법의 실험 및 검증

4.1 샘플데이터 선정 및 쉘코드 유형별 수정

취약점과 공격코드를 인터넷에 공개하는 밀웜싸이트[17]에서 07~08년 사이에 발표된 버퍼오버플로 유형의 샘플코드 7개를 [표 3]과 같이 선정하였다.

샘플 코드는 메모리 실행영역에 따라 2가지 형태가

[표 3] 샘플데이터 List

No	공개 일시	취약점명	메모리 실행영역
1	08. 10.26	PowerTCP FTP Module	Heap
2	08. 08.26	MS Visual Studio 6.0 (Msmask32.ocx)	Heap
3	08. 06.05	BlackIce Barcode SDK (BITiff.ocx)	Stack
4	08. 04.01	RealPlayer rmoc3260.dll	Heap
5	08. 02.03	Sejoong Namo NamoInstaller.dll	Heap
6	07. 10.29	GomPlayer 2.1.6.3499 GomWeb3.dll	Stack
7	07. 09.12	MS SQL Server DM Objects sqldmo.dll	Heap

〔표 4〕 유형별 샘플데이터에 대한 실험 결과

유형	V3	알약	Mcafee	Kaspersky	Symantec	Trendmicro	Rising	Sophos	API 호출 메모리 실행영역 추적		NOP Stamp분석
									Stack	Heap	
1	0.0	0.427	0.714	0.285	0.571	0.428	0.714	0.571	-	-	-
2	0.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

있는데 각각 스택과 힙 영역에서 쉘코드를 동작시키는 방식이다. 선정된 샘플코드를 4가지 유형으로 수정하여 실험을 진행한다. 유형1은 샘플코드 원본, 유형2는 쉘코드를 스택영역에서 동작하도록 수정, 유형3은 쉘코드를 힙영역에서 동작하도록 수정, 유형4는 쉘코드를 Fake Frame기법을 적용하여 동작하도록 수정한다. 유형1 샘플코드 원본을 제외한 유형 2~4는 공격코드의 변수 등을 수정하여 안티바이러스 프로그램의 시그너처 탐지를 회피하도록 작성한다.

4.2 메모리 실행영역 추적기법 실험

제안기법 실험은 32bit용 windows XP 영문판 서비스팩 3환경에서 진행하였으며 API 후킹은 마이크로소프트사에서 무료로 제공하는 Detours Library V1.5[18]의 DetourFunctionWithTampoline 함수를 이용하였다. 먼저 시장 점유율과 탐지율이 높은 8개의 안티바이러스 프로그램을 대상으로 〔표 4〕와 같이 실험을 진행하였다.

유형1에 대한 실험은 악성코드 자체에 대한 시그너처 비교 능력에 대한 실험으로 공격코드를 실행하여 탐지한 결과가 아니라 공격파일 자체에 대한 탐지능력에 대한 실험 결과이다. 유형1의 공격코드를 유형2형태로 수정하여 테스트한 결과 Mcafee, Kaspersky, Trendmicro 3개의 안티바이러스 프로그램이 100% 탐지 가능함을 보여준다. 이는 스택 영역에서 실행되는 유형2 형태의 공격코드에 대해 3개의 안티바이러스 프로그램은 시그너처가 없더라도 탐지할 수 있음을 나타낸다. 본 논문에서 제시한 기법은 공격코드가 실행될 때 메모리 영역을 분석하는 것으로 시그너처 비교 루틴이 구현되어 있지 않아 유형1에 대한 제안기법 실험은 생략하였다. 실험은 〔표 5〕와 같이 Detours Library를 이용하여 API 후킹 후 메모리 실행영역을 추적하는 기능을 C언어로 구현하여 시뮬레이션 하였다. 첫 번째, 스택 영역에서 실행되는 션 하에 대한 시뮬레이션은 지역변수를 이용하여 션 하를 스택에 저장하고 두 번째, 힙 영역에서 실행되는 션 하에 대한 시뮬레이션은 malloc()함수를 이용해 션 하를 힙에 저장하였다. 마지막으로 NOP Stamp기법 시뮬레이션을 위해 쉘코드를 Fake Frame기법을 적용하여 수정하고 NOP코드와 함께 스택에 저장하여 시뮬레이션을 진행하였다. 실험 결과 표4와 같이 유형2, 3은 3.2.1장에서 제안한 "API 호출 메모리 실행영역 분석"을 이용하여 탐지 가능하였고 유형4는 3.2.2장에서 제안한 NOP Stamp기법을 적용하여 탐지가 가능하였다.

〔표 5〕 시뮬레이션 코드

```
WinMain {
    LoadLibraryA("APIMonitor.dll")
    buffer ← shellcode[]
    _asm { //쉘코드 실행
        eax ← buffer
        jmp eax
    }
}
Function APIMonitor.dll { //detours
라이브러리
...
DetourFunctionWithTampoline(RealLoadLib
raryA, MyLoadLibraryA)
...
DetouredRemove(RealLoadLibraryA, MyLoa
dLibraryA)
}
Function MyLoadLibraryA{
    API Memory Search()
    // 3.2.1장 API 메모리 실행영역 분석
    NOP Stamp()
    // 3.2.2장 NOP Stamp 분석
    Call RealLoadLibraryA()
}
```

V. 결 론

본 논문에서는 버퍼오버플로 악성코드에 대해 8개의 안티바이러스 프로그램을 선정하여 유형별 탐지 여부를 실험한 결과 대부분의 안티바이러스 프로그램에서 알려지지 않은 버퍼오버플로 공격을 탐지하지 못하고 이를 구현한 일부 프로그램도 완벽하지 않았다. 이에 API호출 단계의 메모리 실행영역 분석기법과 NOP Stamp 메모리 추적기법을 동시에 적용한 메모리 실행영역 추적기법이 적합한 탐지 방식임을 증명하였다. 향후에는 시스템 부하를 최소화하면서 새로운 버퍼오버플로 공격 자체를 막을 수 있는 탐지 및 방지 기술에 대한 연구가 필요하다.

참 고 문 헌

[1] 서희석, 최중섭, 주필환, "윈도우 악성코드 분류 방법론의 설계," 정보보호학회논문지, 19(2), pp. 84-87, 2009년 4월.

[2] CERT/CC, "US-CERT Vulnerability Notes," http://www.kb.cert.org/vuls/byupdate

[3] 강태우, 조재익, 정만현, 문종섭, "API call의 단계별 복합분석을 통한 악성코드 탐지," 정보보호학회논문지, 17(6), pp. 59-64, 2007년 12월.

[4] 김재현, 강신헌, "네트워크 트래픽 특성을 이용한 스캐닝 웜 탐지기법," 정보보호학회논문지, 17(1), pp. 59-64, 2007년 2월.

[5] D. Mohanty, "Anti-Virus Evasion Techniques and Countermeasures," http://www.infosecwriters.com/text_resources/pdf/AV_Evasion.pdf, pp. 4-11, Dec. 2004.

[6] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," IEEE Software, pp. 5-6, Feb. 2002.

[7] 박남열, 김용민, 노봉남, "우회기법을 이용하는 악성코드 행위기반 탐지 방법," 정보보호학회논문지, 16(3), pp. 19-24, 2006년 6월.

[8] A. One, "Smashing The Stack For Fun And Profit," Phrack, vol. 7, issue. 49, pp. 2-6, Nov. 1996.

[9] E. Chien and P. Szor, "Blended attacks exploits, vulnerabilities and buffer-overflow techniques in computer viruses," Virus bullentin conference, pp. 19-24, Sep. 2002.

[10] Skape, "Understanding Windows Shellcode," http://www.hick.org/code/skape/papers/win32-shellcode.pdf, pp. 5-28, Dec. 2003.

[11] H.M. Sun, Y.H. Lin, and M.F. Wu, "API Monitoring System for Defeating Worms and Exploits in MS-Windows System," ACISP 2006, pp. 3-8, July 2006.

[12] O. Ruwase and M.S. Lam, "A Practical Dynamic Buffer Overflow Detector," Proceedings of the 11th Annual Network and Distributed System Security Symposium, pp. 3-5, Feb. 2004.

[13] Wikipedia, "Win32 Thread Information Block," http://en.wikipedia.org/wiki/Win32_Thread_Information_Block

[14] Microsoft, "PEB Reference," http://msdn.microsoft.com/en-us/library/dexter.functioncatall.peb.aspx

[15] Dragos Ruju, "Fnord: Multi-architecture mutated NOP sled detector," pp. 1-3, Feb. 2002.

[16] A. Sotirov, "Heap Feng Shui in Javascript," BlackHat Europe 2007, pp. 3-18, Apr. 2007.

[17] milw0rm Site, http://www.milw0rm.com

[18] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 Functions," In Proceedings of the USENIX Windows NT Workshop, pp. 135-144, July 1999.