루트킷을 이용하는 악성코드

NCSC-TR050024

국가사이버안전센터
National Cyber Security Center

루트킷을 이용하는 악성코드

안철수연구소 주임연구원 | 고흥환

1. 개 요

루트킷 설치는 SunOS, Unix, Linux 등의 루트 권한을 획득하기 위한 해커들의 가장 중요한 목적이기도 하다. '루트킷' 이라는 이름의 유래도 바로 이러한 루트 액세스를 위한 공격에서 유래된 것이다. 루트킷의 시초를 보면 공격 대상의 시스템(주로 Unix였다)에 대한 이더넷 패킷의 스니핑[1] 이었다. 공격 대상의 시스템에 루트킷 코드를 설치하여 Telnet, FTP 등의 패킷을 가로채 대상 시스템의 접속 아이디와 패스워드를 획득하는 방법을 사용하였다.

근래에 와서는 개인의 PC를 대상으로 하는 윈도우 루트킷이 등장하였다. 윈도우 루트킷 코드는 단순히 루트의 권한을 획득하기 위한 것이라기 보다는 시스템 이미지 변경과 악성코드에 대한 보호로 이어지고 있다. 개인 PC에서 루트킷의 역할은 80~90%가 악성코드(IRCBot 또는 Backdoor)에 대한 프로세스 은닉으로 나타나고 있다.

최근에는 소니 CD에 포함된 불법복사 방지기법이 루트킷으로 작동될 수 있다고 하여, 공개 리콜에 들어가기도 하는 등 루트킷에 대한 관심이 고조되고 있는 실정이다.

본 글에서는 윈도우 커널 루트킷에 한정하여 커널 단에서의 API 후킹 방법과 루트킷 탐지 방법에 대해서 설명하고 끝으로 루트킷에 대한 대응방안에 대해 논의하고자 한다.

2. 루트킷의 발전

'루트킷' 에 대한 정의를 내리자면 "일관되게 영구적으로 시스템에서 탐지되지 않도록 하는 코드와 프로그램의 집합", "공격자가 루트 권한을 획득하기 위한 작고 실용적인 프로그램으로 구성된 집합"이라 할 수 있다.

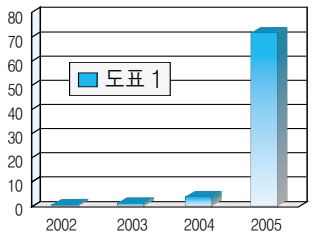
1. 스니핑(Sniffing) – Sniff(냄새를 맡다, 코를 킁킁거리다)의 어원에서 유래되어 네트워크 상의 송수신 데이터를 도청하는 행위를 말한다.

이러한 루트킷에 대한 보안 쟁점은 1994년 초기로 거슬러 올라간다. 당시 CERT/CC(Computer Emergency Response Team/Coordination Center, 카네기멜론대), 미 CIAC(Computer Incident Advisory Capability)에서는 여러 보안 권고문을 발표한 바 있다.

- CERT/CC Advisory CA-94:01, "Ongoing Network Monitoring Attacks", Feb 3, 1994
- CERT/CC Advisory CA-95-95:18, "Widespread Attacks on Internet Sites", Dec 18, 1995
- CIAC Advisory E-09, "Network Monitoring Attacks", Feb 3, 1994
- CIAC Advisory E-12, "Network Monitoring Attacks Update", Mar 18, 1994

윈도우 시스템이 개인 사용자에게 보편화되고 서버 장비의 OS로도 운용되면서 자연스럽게 루트킷의 공격 목표도 윈도우로 넘어오게 되었다. 윈도우 시스템은 Unix, Linux와 달리 OS 소스코드나 중요 API에 대해 공개를 하지 않아 윈도우가 개발된 초기에는 전체 바이러스나 악성코드가 현저히 줄어드는 현상도 나타났었다.

많은 윈도우 개발자가 등장하고 보편화 되면서 서서히 악성코드도 증가하게 되었으며, 현대의 대중적 인터넷 시스템에서는 인터넷 웜이 중요한 문제로 대두되고 있다. 또한 현재의 두드러진 특징 중 하나는 금전적 이득을 위한 트로이목마의 비중이 커지고 있다는 것이다.



옆의 [도표 1]은 안철수 연구소에 접수되어 Win-Trojan/Rootkit으로 진단된 루트킷의 연도별 추가사항을 그래프로 나타낸 것이다. 2005년 이전에는 웜 또는 트로이목마의 진단명을 그대로 부여하여 다소 차이가 나타날 수 있으나 2005년 현재 악성코드에서의 루트킷 사용 증가는 두드러지게 나타나고 있다.

- 현재 루트킷 사용 증가의 주요한 특징을 보면 다음과 같다.
  - ① 루트킷을 이용한 웜(IRCBot), 트로이목마(Backdoor)의 증가
  - ② 윈도우 커널 SSDT(System Service Descriptor Table)를 후킹하여 악성코드의 프로세스 및 파일 보호
  - ③ 루트킷 소스의 인터넷 공개에 따른 중국 제작의 루트킷 증가
  - ④ 스파이웨어에서의 루트킷 사용

3. 윈도우 커널 루트킷

루트킷 본래의 목적을 위해서는 안티바이러스나 기타 스캔도구에 탐지되지 않아야 한다. 이를 위해서는 항시 감시되고 있는 사용자모드의 애플리케이션 또는 서비스 모듈보다 커널 모드로 구성

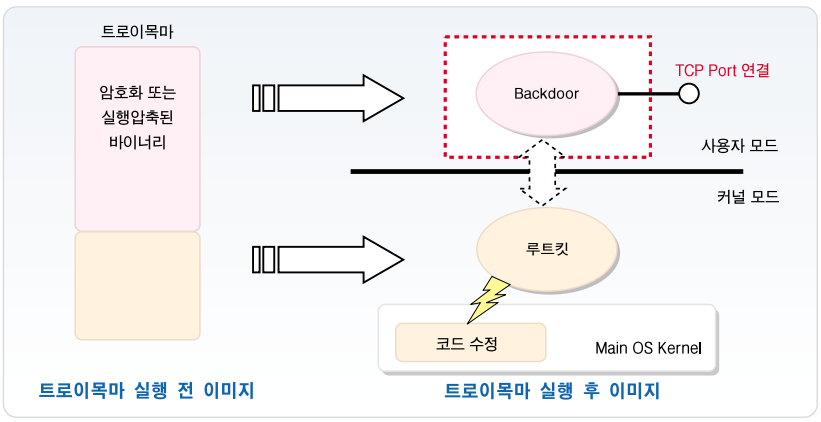
하는 것이 훨씬 효과적일 것이다. 물론 몇몇의 안티바이러스 업체나 스캔도구에서 커널단에 대해서 감시를 하기 시작하였으나 그것이 널리 보편적이지는 않다. 그리고 커널모드에서의 루트킷 구성은 다양한 방법으로 시도되고 있으며 그에 따른 탐지도 쉽지 않은 실정이다.

- 커널단의 루트킷 구성으로 성취할 수 있는 것을 보면 다음과 같이 나눌 수 있다.
  - ① 프로세스 제어 – 프로세스/스레드 은닉
  - ② 파일 및 레지스트리 제어 – 파일/폴더/레지스트리 은닉
  - ③ 보안 변경 – 프로세스의 보안설정 변경 및 제거
  - ④ 메모리 은닉 – 디버거/루트킷 탐지 프로그램에 대한 데이터 은닉
  - ⑤ 네트워크 제어 – TDI/TCPIP 드라이버 후킹에 따른 소켓/패킷 데이터 스니핑
  - ⑥ 키보드 제어 – 키보드 필터에 따른 키 데이터 스니핑

커널모드 루트킷의 제작과 동작은 사용자모드에서의 악성코드 제작보다 많은 노력과 시간이 소비된다. 이에 따른 오동작은 윈도우 시스템의 BSOD(Blue Screen Of Death)로 이어지게 마련이다. 그러나 루트킷의 소스가 인터넷에 공개되고 작은 크기의 바이너리로 시스템을 제어할 수 있는 장점으로 인하여 점점 더 많은 악성코드에서 이를 악용하는 사례가 많아 지고 있는 실정이다.

국내에서 발견된 루트킷의 대부분은 중국과 유럽 등에서 공개소스를 재 컴파일 하거나 약간의 수정을 한 상태로 나타나고 있다. 이에 대한 활용도도 악성코드 프로세스를 은닉하는 경우가 대부분이다. 그러나 점진적으로 그 사용 용도에 대한 확장과 동시에 다양한 기법들이 이용될 것으로 예상되고 있다.

4. 전형적인 루트킷과 트로이목마의 연동



[그림 1] 트로이목마와의 연동

위 [그림 1]에서는 트로이목마와 루트킷의 기본적인 연동관계를 그림으로 나타낸 것이다. 일반 사용자는 메일, 메신저, 게시판, 자료실 등에서 실행파일이 다운로드 되거나 기타 다른 악성코드에 의해 설치될 경우, [그림 1]의 왼쪽과 같이 하나의 실행파일 형태를 이루고 있다. 이러한 트로이목마가 실행될 경우에는 [그림 1]의 오른쪽과 같이 사용자모드에서 활동하는 백도어(Backdoor)와 커널 모드에서 활동하는 루트킷(Rootkit)으로 나뉘어 시스템을 장악하게 된다.

설치된 루트킷은 백도어의 프로세스와 파일, 네트워크 연결에 대한 은닉과 보호를 위해 커널 이미지 코드를 수정할 것이다. 사용자 모드의 백도어는 해커와 연결될 수 있도록 TCP 포트를 열고 기다리게 되며 이에 따라 해커는 자유자재로 사용자의 시스템을 드나들 수 있게 된다.

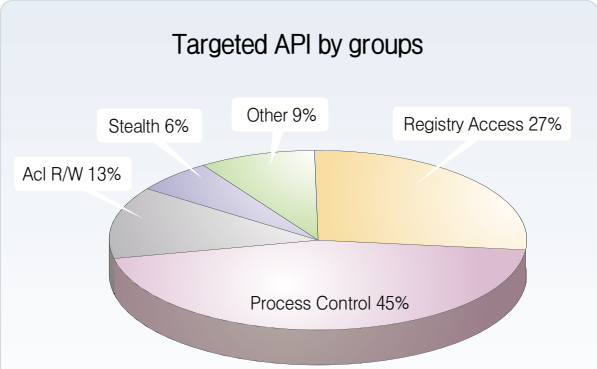
● 트로이목마로부터 루트킷의 분리와 설치 및 실행 순서를 보면 다음과 같이 나타낼 수 있다.

① E-mail, 취약점, 파일 다운로드 등에 의한 트로이목마(또는 웜)의 실행
② 트로이목마 파일 리소스(Resource)에 위치한 루트킷 분리/생성
  - LoadResource, SizeofResource, LockResource, CreateFile
③ 루트킷의 서비스 설치 및 실행
  - Service Control Manager에 의한 설치
  - Undocumented API인 SystemLoadAndCallImage에 의한 설치
④ 트로이목마/시스템 종료 시 루트킷 서비스&파일 제거

여기서 한가지 더 살펴볼 것은 '③ 루트킷의 서비스 설치 및 실행'에 대한 문제이다. 윈도우 커널 드라이버는 서비스 모듈과 마찬가지로 서비스로 등록 및 실행된다. 그러나 이와 같은 방법은 레지스트리와 서비스 등록 모듈 감시로 쉽게 탐지될 소지가 있다. 그래서 사용되는 것이 윈도우 중요 서브시스템인 NTDLL.dll의 Undocumented API이다. NTDLL.dll의 ZwSetSystemInformation의 첫 번째 인자에 SystemLoadAndCallImage 서비스 호출로서 쉽게 루트킷 드라이버를 시스템에서 실행시킬 수가 있다.

이러한 방법에도 문제는 따른다. 바로 BSOD이다. 실행된 루트킷 모듈은 커널의 페이지된 메모리(Paged Memory)에 올라가기 때문에 사용 중이지 않을 때에는 페이지 아웃(디스크로 저장)되어 직접 주소호출 시 문제가 발생한다.

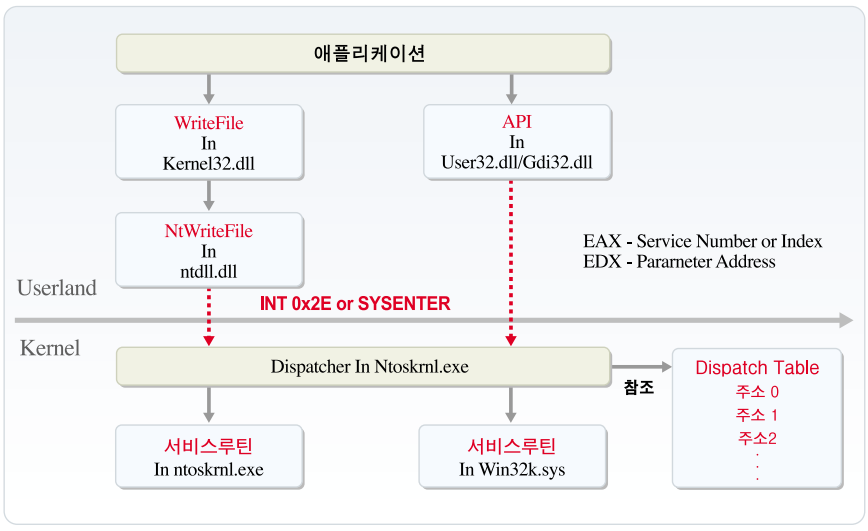
[도표 2]는 2005년 마이크로소프트 보안프로그램 매니저 Alexey Polyakov의 'Windows Rootkits' 발표 자료에서 발취한 것으로 SSDT(System Service Dispatch Table) 후킹에 의한 시스템 크레쉬(Crash) 발생 원인의 API를 그림으로 나타낸 것이다. 이후에 설명할 커널레벨 후킹기법에서 자세히 다루겠으나 SSDT 후킹은 현재 악성코드에서 사용되는 루트킷의 80~90%가 사용하고 있는 기법이다.



[도표 2] 루트킷에 의한 시스템 오류 API

5. 윈도우 커널레벨 후킹 기법

이번 장에서는 윈도우 커널 레벨에서의 다양한 후킹 방법에 대해서 논의하고자 한다. 여러 가지 방법에 대한 논의에 앞서 윈도우 시스템에서의 사용자모드와 커널모드에 대한 사전 지식이 필요하다. [그림 2]는 윈도우 시스템의 일반적인 시스템 서비스 호출에 대한 흐름이다.



[그림 2] Windows System Service

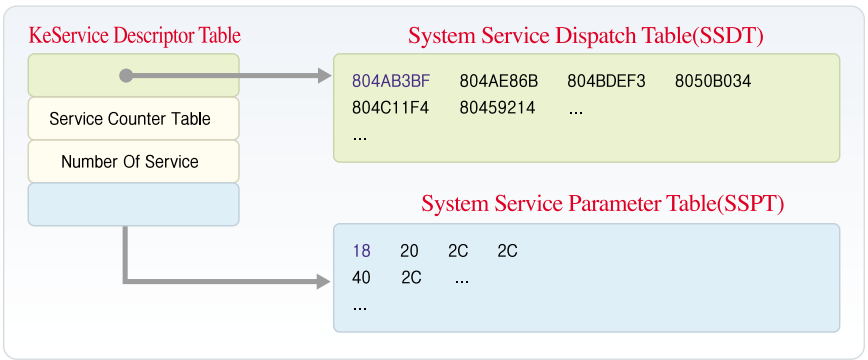
애플리케이션에서 윈도우 API인 WriteFile을 사용한 경우, Kernel32.dll의 WriteFile이 호출되고 이는 NTDLL.dll의 NtWriteFile/ZwWriteFile을 호출하게 된다. 서브시스템인 NTDLL.dll은 커널

단의 시스템 서비스 호출을 위해 인터럽트 0x2E(Windows 2000) 또는 SYSENTER[2](Windows XP)를 이용하게 되며 파라미터 인자 값으로 EAX 레지스터에 서비스 번호/인덱스, EDX 레지스터에 파라미터의 주소 값을 넘겨준다. 커널 모드의 Ntoskrnl.exe에서는 해당 서비스 번호/인덱스에 대한 디스패치 테이블을 참조하여 실제 서비스 루틴의 주소를 얻어 서비스 하게 된다.

NTDLL.dll은 Windows NT/2000/XP/2003 과 OS/2, POSIX 등에 대한 각각의 애플리케이션이 공통으로 사용될 수 있도록 설계되었으며, Kernel32.dll은 윈도우 공통 라이브러리인 NTDLL.dll을 이용하는 Win32용 진입 단계로 볼 수 있다. 그에 반해 User32.dll, Gdi32.dll[3] 같은 경우는 Windows 만의 API로 해당 DLL 자체에서 커널단으로 서비스 호출을 할 수 있도록 설계되어 있다. 이들의 고유 서비스는 Win32k.sys에서 서비스 해준다.

가. SSDT 후킹

SSDT(System Service Dispatch Table)는 서브시스템인 Ntoskrnl.exe가 관리하는 서비스 호출 주소 테이블 구조체이다. 이 주소 테이블은 각 서비스의 주소를 담고 있기 때문에 테이블 변조로 간단히 서비스 내용에 대한 조작이 가능하다.



[그림 3] SSDT 후킹

KeServiceDescriptorTable 함수는 SSDT를 참조하는 구조체의 주소(KiServiceTable)와 SSPT(System Service Parameter Table)의 구조체 주소를 포함하는 테이블을 얻을 수 있다. 여기서 SSDT의 4바이트 리스트는 서비스 번호에 따른 서비스 주소 값을 관리하며, SSPT의 1바이트 리스트는 해당 서비스에 대한 파라미터 바이트 수를 관리한다. 예를 들어, 서비스 인덱스 1번에 대한 서비스는 SSDT의 첫 번째 리스트인 0x804AB3BF 위치에 서비스 코드가 존재하며, 1번 서비스에 대한 파라미터 인자크기는 0x18 크기이다. 모든 서비스의 크기는 KeService

2. Windows XP 이상의 OS에서는 CPU의 IA32_SYSENTER_EIP 레지스터를 이용한 Fast Call 방식을 이용하여 서비스 호출을 하도록 하였다.
3. User32.dll은 Windows의 Message 처리 관련, Gdi32.dll은 Windows Graphics Device Interface 관련 API를 담당하고 있고, 서브시스템 Advapi32.dll은 레지스트리 관련 API를 담당하고 있다.

DescriptorTable의 NumberOfService로 알 수 있다. 이에 따라 SSDT 서비스 테이블의 크기가 4바이트(서비스 주소값) × NumberOfService 만큼의 크기라는 결과가 된다. 또 다른 테이블인 KeServiceDescriptorTableShadow 함수는 User32, Gdi32 서비스에 대한 서비스 참조 구조체를 관리한다. 이에 대한 서비스는 모두 Win32k.sys에서 구현될 것이다.

그럼 이제 악성코드와 연결된 루트킷에서 빈번히 사용되는 NtQuerySystemInformation에 대해서 자세히 알아보자. NtQuerySystemInformation 서비스 호출은 현재 시스템의 모든 프로세스 리스트를 얻어 알려주는 기능을 한다. 루트킷은 이 서비스를 가로채 자신이 보호하려는 악성코드(트로이목마)의 프로세스 정보를 없애 악성코드가 돌고 있는 것을 감춘다.

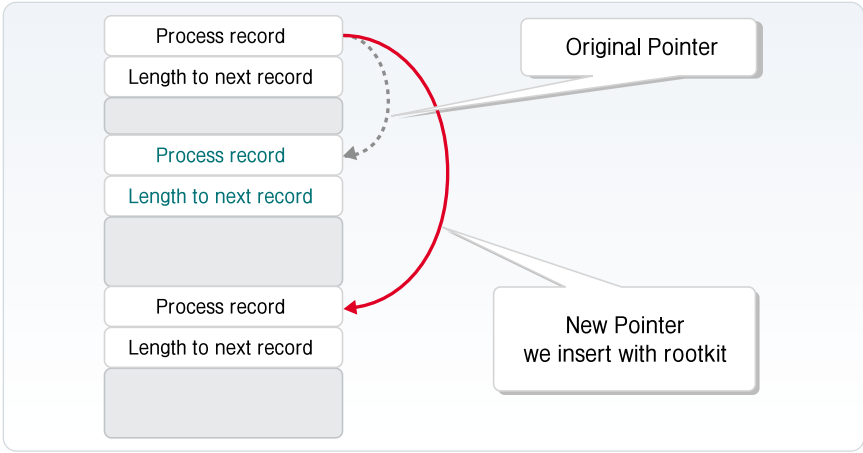
다음의 예제 코드는 SSDT의 NtQuerySystemInformation 서비스에 대한 서비스 주소 변경을 가능하게 한다.

```
#define SYSTEMSERVICE(_api)
KeServiceDescriptorTable.ServiceDescriptor[0].ServiceTable[*(DWORD*)((unsigned char*)_api + 1)]
// 원본 서비스 주소값 저장
NtQuerySystemInformation_Origin =
(NTQUERYSYSTEMINFORMATION)(SYSTEMSERVICE(NtQuerySystemInformation));
// SSDT 서비스 주소를 후킹 함수 주소로 수정
(NTQUERYSYSTEMINFORMATION)(SYSTEMSERVICE(NtQuerySystemInformation)) =
Nt                    _Hook
```

변경된 서비스 주소로 인하여 해당 서비스 호출 시 다음의 코드 주소가 불려지게 될 것이다. 이 후킹된 코드에서 실제 데이터 조작을 통해 악성코드(트로이목마)의 프로세스를 은닉시킬 수 있게 된다.

```
NTSYSAPI NTSTATUS NTAPI NtQuerySystemInformation_Hook ( … )
{
        Status = NtQuerySystemInformation_Origin( … );
        if ( Status == STATUS_SUCCESS &&
             SystemInformationClass == SystemProcessAndThreadsInformation )
   {
        얻어온 프로세스 정보 조작 …
    }
    return Status;
}
```

결과적으로, [그림 4]는 NtQuerySystemInformation 서비스를 통해서 받아온 프로세스 리스트에서 트로이목마 프로세스에 대한 리스트 연결을 다음 프로세스로 연결 시켜 줌으로 해서 프로세스 정보를 감추게 된다.



[그림 4] 프로세스 은닉 기법

나. Interrupt Descriptor Table 후킹

IDT(Interrupt Descriptor Table)는 하드웨어/소프트웨어 인터럽트에 대한 주소를 관리하며 앞서 살펴본 바와 같이 커널 서비스 호출에 대한 인터럽트 0x2E 또는 SYSENTER를 가로채서 변경할 수 있다. IDT는 CPU마다 따로 관리하므로 멀티프로세서 환경에서는 모든 IDT에 대한 변경을 해주어야 하며, 인터럽트 특성상 한번 호출되면 그에 대한 응답이나 제어가 넘어오지 않으므로 원본 API에 대한 결과값 변경이나 데이터 필터링을 할 수는 없으나 특정 소프트웨어(PC Firewall 등)의 서비스 호출을 차단하거나 알아낼 수 있다.

▶ WIN 2K, Ntdll.DLL에서의 NtWriteFile 호출에 대한 시스템코드

```
MOV  EAX, 0EDh                    // EAX  서비스 호출 인덱스 번호
LEA  EDX, DW[redacted]+4]         // EDX  스택 파라미터 주소
INT 2E
RETN 24
```

▶ WIN XP, Ntdll.DLL에서의 NtWriteFile 호출에 대한 시스템코드

```
MOV  EAX, 112h                    // EAX  서비스 호출 번호
LEA  EDX, 7[redacted]h            // EDX  현재의 스택 주소
CALL  DWORD PTR DS:[EDX] ------->  MOV  EDX, ESP
RETN 24                           SYSENTER
```

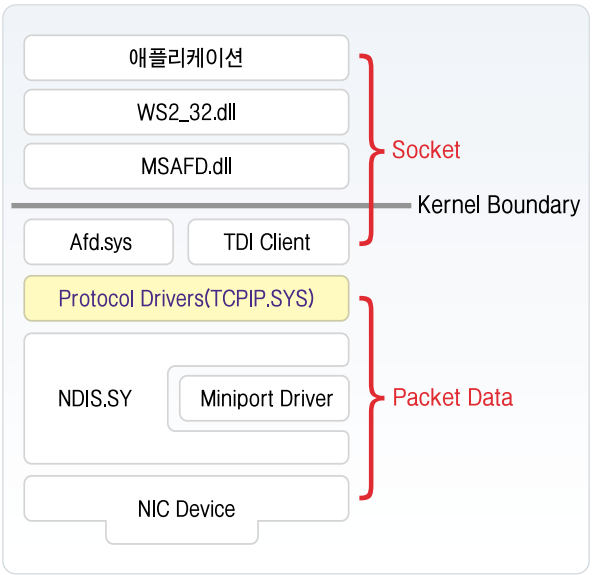
위에서와 같이 시스템 서비스에 대한 인터럽트 후킹을 위해 Windows 2000 이하에서는 INT 0x2E에 대한 IDT 변경이 필요하다. 다음의 예제는 그에 따른 코드이다.

```
__asm {
    sidt  IDTINFO                          // IDTINFO 구조체를 얻는다.
}
...                                        // INT 0x2E에 대한 IDTENTRY 주소를 얻는다.
__asm {
    cli                                    // 인터럽트 서비스를 잠시 중단 시킨다.
    lea  eax, KiSystemService_Hook         // 후킹될 모듈의 새로운 진입점을 만든다.
    mov  ebx, P[redacted]E                 // INT 2E에 대한 진입점을 얻는다.
    mov  [redacted]                        // INT 2E 호출 시, 새로운 진입점으로 변경한다.
    shr  eax, 16
    mov  [ebx+6], ax
    sti                                    // 인터럽트 서비스를 다시 시작한다.
}
```

다음으로는 Windows XP 이상의 시스템에서 시스템 서비스 호출에 대한 SYSENTER 후킹 코드이다.

```
__asm {
    mov  ecx, 0x176
    rdmsr                                  // IA32_SYSENTER_EIP 값을 eax로 읽어온다.
    mov  [redacted]                        // 원본 SYSENTER 주소를 따로 저장한다.
    mov  eax, SYSENTER_Hook                // 새로운 주소로 진입점을 변경한다.
    Wrmsr                                  // 변경된 주소를 IA32_SYSENTER_EIP에 기록한다.
 }
```

다. I/O Request Packet Function 후킹



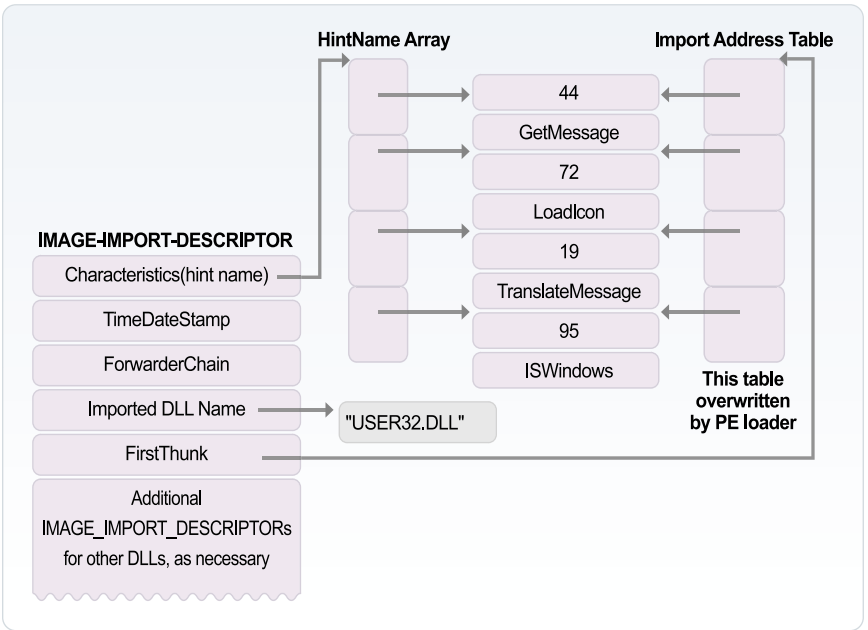
[그림 5] 네트워크 드라이버 계층도

커널단에서 동작하는 모든 드라이버들은 하드웨어 또는 다른 드라이버와의 통신을 위한 Dispatch Routine을 가지고 있다. 그리고 이러한 통신을 위해 사용되는 것이 IRP(I/O Request Packet) 구조체이다. I/O Manager에 의해서 관리되는 IRP는 계층화되어 있는 드라이버 각각에 대한 I/O Stack을 구성하여 각각의 처리에 맞는 정보를 전달하도록 하고 있다.

윈도우 네트워크도 마찬가지여서 NDIS Library를 이용하는 모든 네트워크 드라이버들도 [그림 5]와 같이 계층적 구조를 이루고 있다. NDIS[4] Library를 이용하여 이에 대한 함수 호출 만으로 데이터를 전송하는 구조는 Miniport, Protocol 드라이버 등이다. 그러나, TDI와 Protocol 드라이버에서는 IRP로 데이터를 교환한다. 루트킷은 이러한 TDI(Tdi.sys) 또는 Protocol(Tcpip.sys) 드라이버의 Device Object로부터 Driver Object 구조체를 얻고 이 구조체 내에 구성되어 있는 Dispatch Routine을 가로채서 패킷스니핑 또는 네트워크 패킷 조작이 가능하도록 할 수 있다.

이러한 방법 이외에도 네트워크, 키보드, 파일시스템등과 같은 계층적 구조의 드라이버들은 윈도우 OS에서 제공하는 필터(Filter) 기능을 이용하여 자신의 드라이버를 등록시킬 수가 있다. 안티바이러스 업체나 대부분의 보안, 모니터링 프로그램 등에서도 이러한 방법을 이용하여 각종 서비스를 제공한다.

라. Import Address Table 후킹



[그림 6] PE 파일의 .idata 섹션

모든 사용자모드 프로그램(애플리케이션)들은 시스템의 서비스를 이용하기 위해 OS에서 제공하는 라이브러리 파일을 임포트(Import)하게 된다. 라이브러리 파일들은 실행가능 파일(PE 파일)[5]

4. NDIS(Network Driver Interface Specification) Library – MS Windows OS의 계층적 커널 네트워크 구조에 대한 표준 인터페이스를 제공한다.
5. Portable Executable – Windows 실행 파일의 구조로 실행에 필요한 정보들과 프로그램 코드로 이루어져 있다.

의 .idata[6] 섹션에 자신의 익스포트(Export) 함수에 대한 정보를 담고 있으며 IMAGE_IMPORT_DESCRIPTOR 구조체의 FirstThunk에는 IAT(Import Address Table) 주소를 함유하고 있다. 이 IAT 구조체의 Import Address를 수정하여 간단하게 후킹 할 수가 있는 것이다.

이러한 IAT 후킹 방식은 사용자모드에서도 조작이 가능하다. 그러나 후킹 함수를 위치시키는데 있어 해당 프로세스의 핸들(Handle)을 여는 방식은 안티바이러스 제품이나 기타 탐지 모듈에 있어 발견되기 쉬운 단점이 존재한다. 이것을 커널모드에서 작성하여 그에 따른 탐지를 어렵게 할 수 있다. 이에 대한 설명은 다음과 같다.

```
PsSetImageLoadNotifyRoutine ( IN PLOAD_IMAGE_NOTIFY_ROUTINE NotifyRoutine );
VOID (*PLOAD_IMAGE_NOTIFY_ROUTINE)(IN PUNICODE_STRING FullImageName,
                                   IN HANDLE ProcessId,
                                   IN PIMAGE_INFO ImageInfo );
```

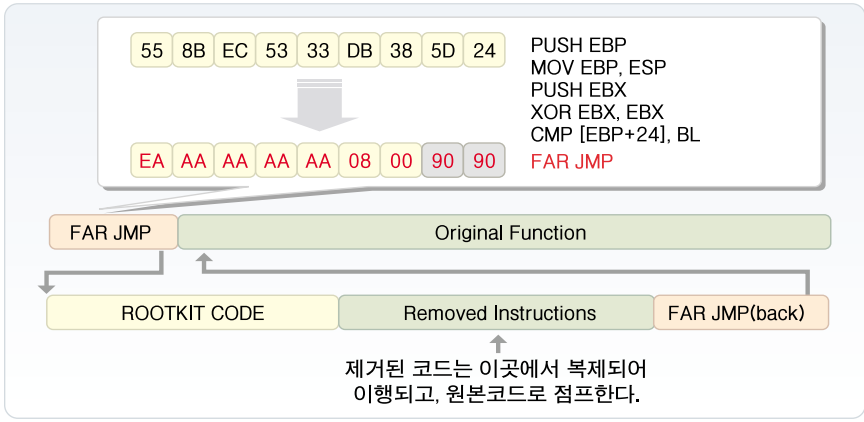
PsSetImageLoadNotifyRoutine은 커널모드 Callback API 함수이다. 새로운 프로세스, DLL 등이 메모리에 적재될 경우에 이 Callback 함수가 호출될 것이다. Callback 함수의 파라미터 인자에는 로드된 파일이름, 생성된 프로세스 ID 그리고 IMAGE_INFO 구조체가 존재한다. 이 IMAGE_INFO 구조체의 정보에는 PE 파일의 이미지 주소가 참조되어 있기 때문에 쉽게 IMAGE_IMPORT_DESCRIPTOR의 위치를 찾아낼 수 있으며, 그에 따라 IAT 위치도 알 수 있다.

문제는 실제 후킹한 모듈을 어디에 위치 시키느냐가 된다. 모든 프로세스가 참조할 수 있는 사용자 모드의 메모리 영역에 존재하여야 하지만 해당 프로세스 영역에 작성한다는 것은 위험이 따른다. 그리하여 KUSER_SHARED_DATA라고 명명된 메모리 영역을 이용한다. 이 영역은 커널단에서는 0xFFDF0000 주소로, 사용자영역에서는 0x7FFE0000 주소로 참조 가능한 물리적 페이지 메모리 영역이며, 4 Kbyte의 영역 중 1 Kbyte만을 시스템이 사용 중이므로 나머지 3 Kbyte의 공간이 비어있다.

마. Inline Function (Detour Patch) 후킹

크렉에서 사용하는 방법과 유사하게 어셈블된 코드 상에서 함수의 시작 일정코드를 자신의 루트킷 코드가 존재하는 주소로 강제 점프하여 실행되도록 하고 본래의 위치로 다시 점프되도록 하는 리버스엔지니어링(Reverse Engineering) 기법이다.

6. 실행에 필요한 DLL의 함수나 변수에 대한 임포트 정보로 구성된 PE 파일의 임포트 섹션



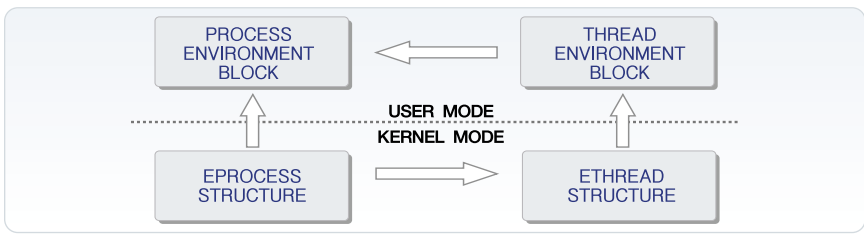
[그림 7] Inline Function Hook

[그림 7]은 Detour Patch 방식을 나타낸 그림이다. 루트킷 코드로 제어권을 넘겨받기 위해 원본 코드의 시작부분에서 FAR JMP 08:XXXXXXXX 해야만 한다. FAR JMP 코드는 7 Byte면 되지만 위 그림에서는 9 Byte를 변경했다. 7 Byte만을 JMP 코드로 변경한다면 원본 함수 어셈블리 코드의 정렬(Alignment)이 깨져 BSOD이 발생될 것이기 때문이다. 7 Byte의 FAR JMP 변경과 나머지 2 Byte에 대한 NOP[7] Instruction을 구성한 것이 중요 포인트라 할 수 있다.

FAR JMP에 의해 루트킷 코드를 이행하고 나서는 제거된 9 Byte의 원본 시작코드를 수행해 주어야 한다. 이는 원본함수 9 Byte 이후로 다시 FAR JMP 되기 때문이며, 이로 인해 원본 코드가 문제없이 수행된다.

바. Direct Kernel Object Manipulation

윈도우 커널 레벨의 Object Manager는 시스템을 구성하는 다양한 오브젝트 객체에 대한 정보를 관리한다. Device, Driver, EProcess, EThread등과 같은 오브젝트의 정보를 직접적으로 수정함으로써 후킹 방식보다 더 탐지하기 어려운 루트킷을 구성할 수가 있다. 이렇게 커널의 오브젝트를 조작하는 방식을 DKOM(Direct Kernel Object Manipulation)이라 한다.

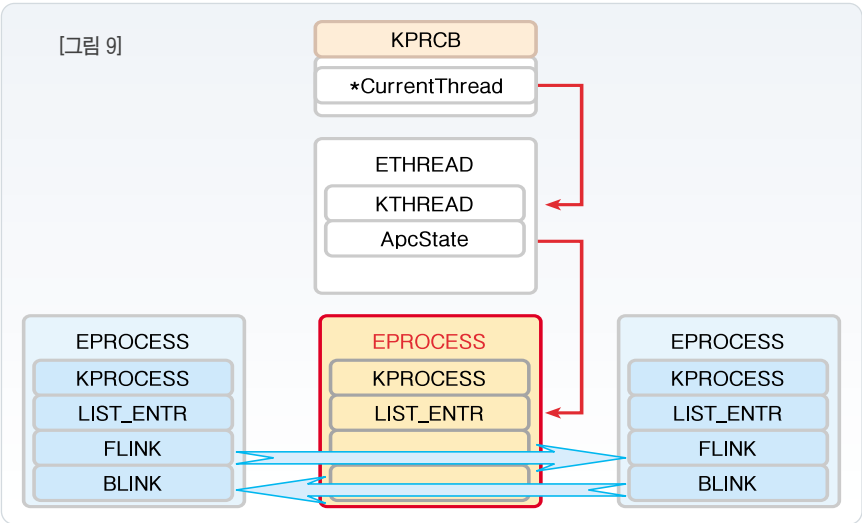


[그림 8] I프로세스와 스레드가 연결된 데이터구조

7. NOP Instruction은 어떠한 처리도 하지 않는 1Byte CPU 명령코드이다.

커널단에서는 EPROCESS라는 구조체를 하나의 객체로 관리하며 각각의 EPROCESS 구조체는 서로 이중 연결리스트로 연결되어 있다. 이에 반해 사용자 모드에서는 PEB(Process Environment Block)이라 하여 사용자 모드에서 수정되는 데이터 정보를 포함한다. 커널단에서 관리되는 EPROCESS 구조체의 정보를 획득하여 연결리스트를 조작한다면 시스템 정보에서는 조작된 EPROCESS를 참조하지 못하여 해당 프로세스(트로이목마)가 존재하지 않는 것처럼 만들 수 있을 것이다.

[그림 9]



은닉하려는 대상 프로세스의 구조체를 구하기 위해서 PsGetCurrentProcess를 이용할 수 있는데 실제적으로 PsGetCurrentProcess는 IoGetCurrentProcess를 호출한다. 다음의 코드는 PsGetCurrentProcess를 호출한 상태에서 디어셈블한 결과이다.

```
NtoskrnI!IoGetCurrentProcess
0008:804ED364    MOV    EAX, FS:[00000124]
0008:804ED36A    MOV    EAX, [EAX + 44]
0008:804ED36D    RET
```

FS 레지스터로부터 0x124번째의 위치에는 ETHREAD에 대한 주소가 위치하며, ETHREAD의 0x44의 위치가 바로 현재 EPROCESS 구조체의 주소 값이 된다. 구해진 EPROCESS 연결리스트 링크 주소를 다음의 EPROCESS 링크 주소로 변경하여 대상 프로세스를 은닉할 수 있게 된다. 그러나, 윈도우 NT/2000/XP/XP SP2/2003 OS 마다 EPROCESS 구조체에 위치한 Process ID와 연결리스트 위치가 다르기 때문에 루트킷은 디바이스 초기화 때 해당 OS의 버전 정보를 구해야 할 필요가 있다.

윈도우의 버전 정보는 다음과 같은 레지스트리 정보에서 얻을 수 있으며,

- HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\CSDVersion
- HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\CurrentBuildNumber
- HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\CurrentVersion

커널 API인 PsGetVersion 또는 RtlGetVersion을 통해서도 알 수 있다.

	Win NT	Win 2000	Win XP	Win XP	Win 2003
PID	0x94	0x9C	0x84	0x84	0x84
FLINK	0x98	0xA0	0x88	0x88	0x88

[그림 10] Offsets to the PID and FLINK within the EPROCESS Block

6. 루트킷 탐지 방법

후킹을 이용한 루트킷의 탐지 방법은 오히려 간단할 수 있다. 후킹된 API가 존재하는 모듈의 메모리 위치 시작주소와 크기를 구하여 API가 해당 모듈의 메모리 영역 내에 존재하는지를 비교하면 된다. 또한 어떤 모듈이 후킹을 하였는가에 대해서도 같은 방법으로 찾을 수 있다.

[그림 11] Softice를 사용한 루트킷 디버깅

[그림 11]은 커널 디버거인 Softice를 이용하여 SSDT 후킹에 대한 루트킷 디버깅 과정을 나타내고 있다. 붉은 라인의 코드부분이 바로 SSDT의 서비스 API인 ZwQuerySystemInformation에 대한 주소를 루트킷 자신의 모듈 주소로 바꿔 후킹하는 부분이며, 메모리 0023:F8C698D8[8] (ZwQuerySystemInformation 서비스콜) 부분의 4 Byte 주소가 루트킷 모듈(hideprocess)의

8. 해당 주소는 System Service Descriptor Table의 Index 0xAD 번째 ZwQuerySystemInformation 서비스 주소이다.

F8C72000 ~ F8C72A80 주소 내로 변경된 것을 확인할 수가 있다.

여러가지 방식으로 은닉된 악성코드의 실행 프로세스를 찾기 위한 방법으로는 시스템 프로세스인 CSRSS.exe의 HANDLE_TABLE을 이용하여 전체 프로세스 리스트를 찾을 수 있는 방법이 있으나 이 또한 DKOM(Direct Kernel Object Manipulation) 방식으로 은닉된 프로세스는 찾기 어렵다. 마지막으로 사용할 수 있는 방법은 Thread Scheduling을 이용하여 Dispatcher Ready Queue에서 대기중인 Thread를 얻어오면 이 Thread를 통해 EPROCESS 구조체 정보를 획득할 수 있으므로 은닉된 실행 프로세스의 존재를 확인할 수 있다. 이 Dispatcher Ready Queue의 구조체 정보는 KiDispatcherReadyListHead[9]에 존재한다.

7. 향후 전망

지금까지 윈도우 루트킷에 대한 다양한 방식과 트로이목마와의 연동 그리고 이러한 루트킷의 탐지 방법에 대해서 논의해 보았다. 온라인 상으로도 다양한 소스가 공개되어 있는 현재 실정에서 루트킷에 대한 악용의 소지는 날로 증가할 예정이다. 글을 작성하는 지금 이 시간에도 다양한 루트킷이 접수되고 있으며 온라인 게임에 대한 트로이목마에서도 활용되고 있는 실정이다.

최근 들어 루트킷을 탐지하는 여러 소프트웨어도 선보이고 있으며, 안티바이러스 업체에서도 루트킷에 대한 탐지 및 제거에 대한 연구가 계속되고 있고 그에 따른 결과물도 속속 나오고 있다. 이에 반해 안티디버깅과 암호화, 실행압축, 새로운 방식의 시스템 커널 이미지 변경 등 날로 발전해 가는 악성코드에 대해 보안을 담당하고 있는 기관, 기업체 등에서도 이에 대한 대비를 철저히 준비하여야 할 것이다.

Reference

1. “ROOTKITS” Greg Hoglund, James Butler
2. “Inside Windows 2000”(Third Edition) David A.Solomon, Mark E.Russinovich
3. “Windows Internals”(Fourth Edition) David A.Solomon, Mark E.Russinovich
4. “Windows Driver Model”(Second Edition) Walter Oney
5. “API로 배우는 Windows 구조와 원리” 야스무로 히로카즈
6. “Exploiting Software” Grag Hoglund, Gary McGraw
7. www.rootkit.com
8. www.pcausa.com
9. www.sysinternals.com
10. Windows Rootkits PPT (MS 보안기술발표자료) Alexey Polyakov
11. http://www.cs.wright.edu/people/faculty/pmateti/Courses/499/Fortification/obrien.html
12. http://packetstorm.security-guide.de/hitb04/hitb04-chew-keong-tan.pdf
13. http://invisiblethings.org/papers/ITUnderground2004_Win_rtks_detection.ppt

9. KiDispatcherReadyListHead는 32개의 List로 구성된 대기중인 Thread Priority Queue로 이루어져 있다.