Rootkit Scanner

[부경대학교 CERT-IS]

최종학, 신명재, 이인회, 정득교
전경휘, 김연재, 이재호

Contents

1 개요

1.1 프로젝트의 동기

악성코드의 발전은 날이 갈수록 빨라 지고 있다. 그리고 그 관심은 작년 한해 웹 해킹을 잠재울 만큼 큰 것이었다. 그 중 루트킷(Windows)은 자신을 숨기는 일을 하기 때문에 더욱 찾아내기 힘들어졌다.

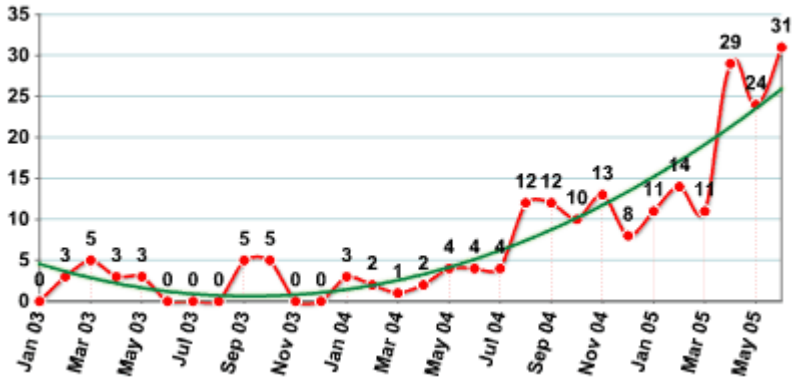


그림 *1: rootkit* 을 사용하는 악성코드의 수

[그림 1] 에서 보는 것과 같이 루트킷을 포함한 악성코드의 수는 증가하는 추세이다. 그리고 현재는 더욱 많은 수의 악성 코드들이 자신을 감추는 일을 함께 하고 있다. 이는 루트킷의 특징인 자신을 감추는 작업의 분석이 무엇보다도 중요하다는 것을 말해주고 있다. 현재 악성코드와 루트킷의 구분이 모호해졌지만 자신을 숨기는 작업의 구현은 크게 다르지 않다. 이러한 이유로 우리는 윈도우즈에서 루트킷이 어떻게 자신을 숨기는지 분석하려고 한다. 물론 그 분석을 토대로 탐지 기법을 개발 하는 것도 그 목표이다.

1.2 프로젝트 요약 및 방향

프로젝트는 크게 세 단계로 진행을 하기로 하였다.

첫째, 나를 알자. 여기서 '나'는 OS 를 비롯한 PC 의 전반적인 부분으로 이해하자. OS 나 디스크는 많은 악성 프로그램들의 목표가 되어왔다. 특히 OS 의 커널 레벨 기술은 루트킷의 핵심부분인 자신을 숨기는 작업에 많이 이용되고 있다. 물론 루트킷을 탐지

하는 데에도 쓰인다. 이런 양날의 검을 우리는 꼭 알고 넘어가야 한다.

둘째, 적을 알자. 모든 공격을 알아야 방어를 할 수 있는 것은 아니지만 공격을 분석하면 그 공격 기법을 이용한 방어기법 등 새로운 기술을 더 많이 배울 수 있다라고 생각한다. 그 때문에 루트킷에 대한 전반적인 이해와 그 분석이 이 후 탐지 작업을 위해서 필요하다.

셋째, 이제 방어를 하자. 지금까지 연구하였던 부분을 토대로 우리가 어떻게 하면 이러한 루트킷을 방어할 수 있을지 에 대한 연구와 그 결과물의 제작을 하기로 하였다.

2 루트킷의 종류

2.1 루트킷이란*?*

2.1.1 루트킷의 정의

UNIX/LINUX 시스템에서 'root'는 전면적 특권을 가진 관리자를 의미하며, 'kit'은 도구 묶음을 가리키는데 사용 된다. 따라서 rootkit 용어는 악의적인 의도를 가지고 실제 관리자에게 알려지지 않은 채로 시스템에 대한 접근권을 얻기 위해서 사용될 수 있는 도구의 묶음을 의미한다.

하지만 윈도우즈(Windows)계열의 rootkit 은 이러한 유닉스 계열의 rootkit 과 달리 DOS 스텔스 바이러스에서 파생된 것임을 명심하여야 한다. 윈도우즈에서 특정한 권한을 획득하는 목표보다 유저와 안티바이러스 프로그램으로부터 자신을 숨기는것이 윈도우즈 rootkit 의 특징 이다.

2.1.2 루트킷의 작동원리

루트킷은 변경이라는 간단한 개념으로 동작한다. 루트킷은 일반적인 흐름의 프로그램을 변경해서 잘못된 판단을 하도록 만드는 것이다. 이는 대체로 후킹기법, 인젝션 등의 흐름 제어를 변경하는 기법을 이용한다.

2.1.3 현재 루트킷 탐지/방어 시스템

Blink (www.eEye.coom)
AVG Anti-Rootkit (http://free.avg.com/us-en/download#avg-anti-rootkit-free)
gmer (http://www.gmer.net/)
icesword (http://www.antirootkit.com/software/IceSword.htm)

대다수 루트킷 탐지/방어 시스템에서는 커널 레벨 기술을 이용한 운영체제에 대한 모니터링 시스템이 그 대다수이다. 그 이유는 단지 자신을 숨기다고 해서 그것이 악성코드 혹은 루트킷이라고 단정지을수 없다는 것에 그 한계가 있다. 우리 프로젝트 역시

이러한 약점을 그대로 안고가는 경우이긴 하다. 결국 오탐의 확률이 높기 때문에 사용자의 주의가 무엇보다도 필요하다라고 생각한다.

2.2 유저 레벨 루트킷

OS 는 특정 어플리케이션을 이용하여 커널 레벨을 호출한다. 이러한 어플리케이션을 우리는 Win32 애플리케이션이라고 흔히 부른다. Win32 에서는 디렉토리 내의 파일을 나열하기 위해서 Kernel32.dll 에서 제공하는 FindFirstFile API 를 이용한다. FindFirstFile 로 반환받은 핸들을 이용하여 FindNextFile API 를 호출한다. 이러한 API 로 호출하는 파일중 Ntdll.dll 을 호출하여 커널 영역으로 접근을 할 수 있다. 이러한 일련의 작업은 [그림 2] 로 간단하게 나타낼 수 있다.

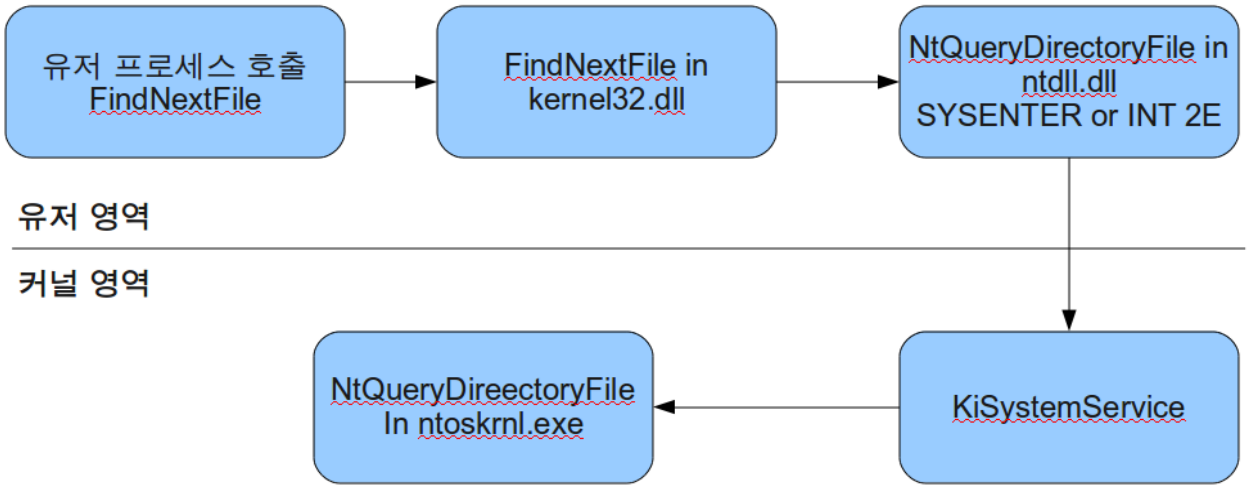


그림 *2: FindNextFile* 의 실행흐름

2.2.1 임포트 어드레스 테이블 후킹 (IAT Hooking)

애플리케이션이 다른 바이너리의 함수를 이용하려면 애플리케이션은 해당 함수의 주소를 반드시 알아야 한다. Win32 API 를 사용하는 대부분의 애플리케이션은 그러한 함수들의 주소를 저장하기 위해 ***IAT*** 를 이용한다. 애플리케이션이 사용하는 DLL 에 대한 정보는 애플리케이션 바이너리 **IMAGE_IMPORT_DESCRIPTOR 구조체** 안에 포함된다. 이 구조체 안에는 **DLL 의 이름**과 두개의 **IMAGE_IMPORT_BY_NAME 구조체의 주소**를 가지고 있다. 루트킷은 IAT 에서 후킹할 함수의 주소를 찾아 자신이 만들어 놓은 후킹함수로 교체해 놓는다. 교체 후부터는 이제 후킹된 함수가 호출될 것이다.

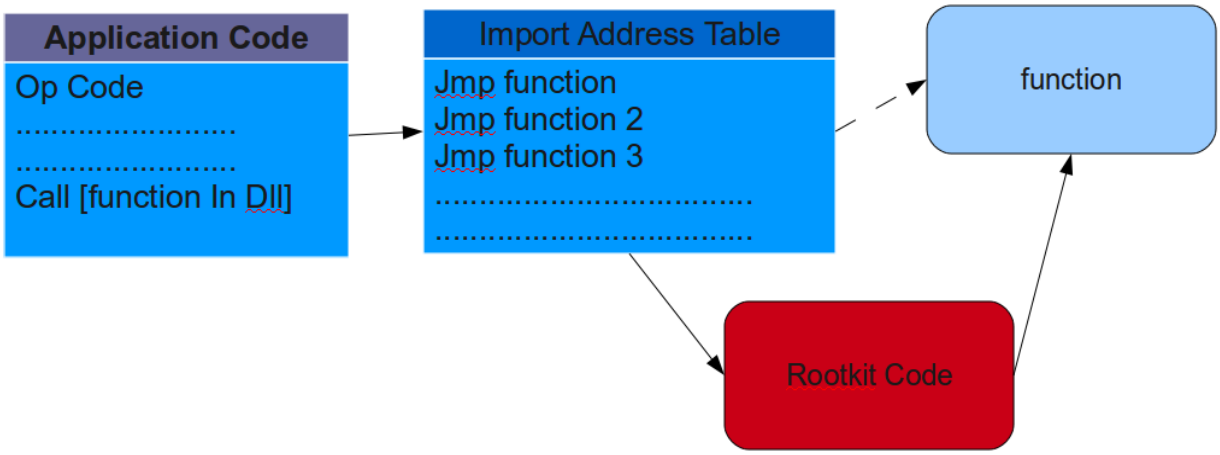


그림 *3: IAT* 후킹의 흐름

2.2.2 레지스트리를 이용한 DLL 인젝션

윈도우 NT/2000/XP/2003 에는 HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\APPInit_DLLs 라는 레지스트리 키가 존재한다. User32.dll 은 LoadLibrary 함수를 이용해서 이 키 값에 정의된 DLL 들을 로드 한다. 각 DLL 들이 로드되면 해당 DLL 들이 DllMain 함수가 호출된다. DllMain 함수에는 이 함수가 어떤 이유 때문에 호출되었는지를 나타내는 인자가 입력된다. 이 인자 중 DLL_PROCESS_ATTACH 인 경우, 즉 DllMain 함수에 DLL_PROCESS_ATTACH 인자가 입력되어 호출되는 시점에 후킹 작업을 수행해야 한다.

이 후킹 방법을 이용하기 위해 변경하는 레지스트리 값이 시스템에 적용되려면 재부팅이 필요하다. 그러나 후킹이 시작된 이후에 실행된 프로세스에 대해서는 후킹을 수행할 수 있다.

2.2.3 유저 레벨 루트킷의 한계

유저 레벨 루트킷은 결국 유저 레벨에서 동작하기 때문에 커널 레벨에서 동작하는 안티바이러스 프로그램에게 바로 발견된다. 그리고 레지스트리를 이용한 dll 인젝션 처럼 후킹이 동작한 후 실행되는 프로세스에 의해서만 그 효과가 있기때문에 실행될 가능성이 낮은 것이 사실이다.

2.3 커널 레벨 루트킷

2.3.1 커널 레벨, Ring 0

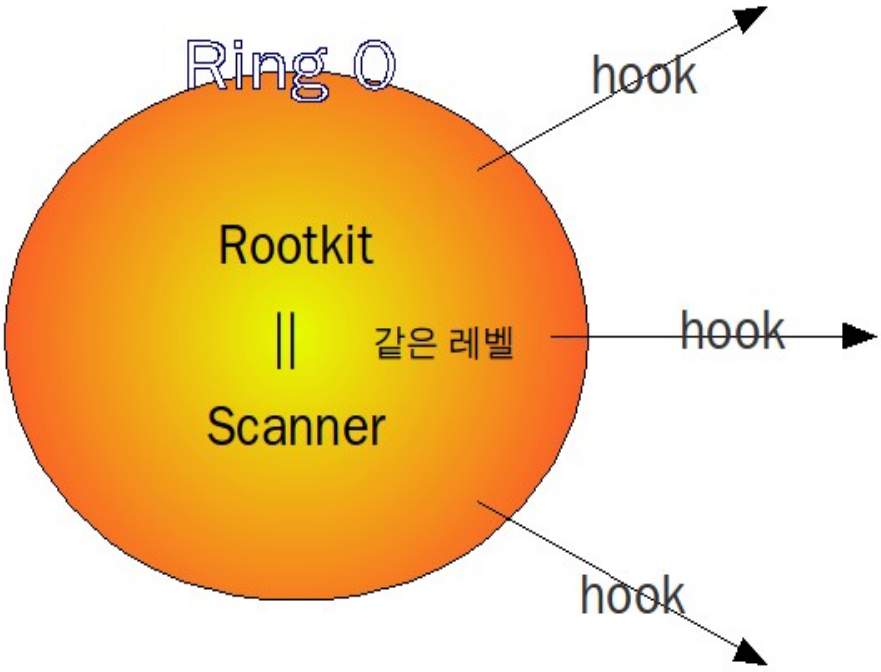


그림 *4: Ring 0*

[그림 4] 처럼 커널 레벨에서의 루트킷은 스캐너와 같은 레벨에서 실행된다. 앞에서 보았던 유저 레벨에서의 루트킷이 커널 레벨에서 동작하는 스캐너에 의해 쉽게 탐지되는 반면, 이 커널 레벨에서의 루트킷이야말로 스캐너의 천적이라고 할 수 있다. 이 곳에서는 루트킷과 스캐너는 누가 먼저 자리를 잡느냐에 따라 승패가 결정된다. 왜냐하면 스캐너 역시 커널 레벨에서 같은 방법으로 루트킷을 찾기 때문이다.

커널 레벨에서의 루트킷의 또 하나의 장점은 시스템에 전역적으로 후킹이 가능하다는 점이다. 윈도우즈에서 동작하는 프로그램들은 Native API 를 사용하게 되는데, 커널 레벨에서 이 Native API 를 후킹하게 된다면 이것을 사용하는 프로그램들에 대해 메모리에 접근할 수 있기 때문이다. 루트킷의 기본 기능인 은닉 또한 마찬가지로 가능하다.

2.3.2 SSDT

커널 레벨에서 루트킷이 동작을 하는 여러 방법 중의 하나가 SSDT 후킹이다. 앞에서

Native API 를 후킹하면 시스템에 전역적으로 후킹이 가능하다고 했었다. Nt 로 함수 이름이 시작하는 이 함수들은 메모리의 커널 영역에 자리 잡고 있는 SSDT(System Service Dispatch Table)라는 구조체에 그 주소가 저장되어 있다.

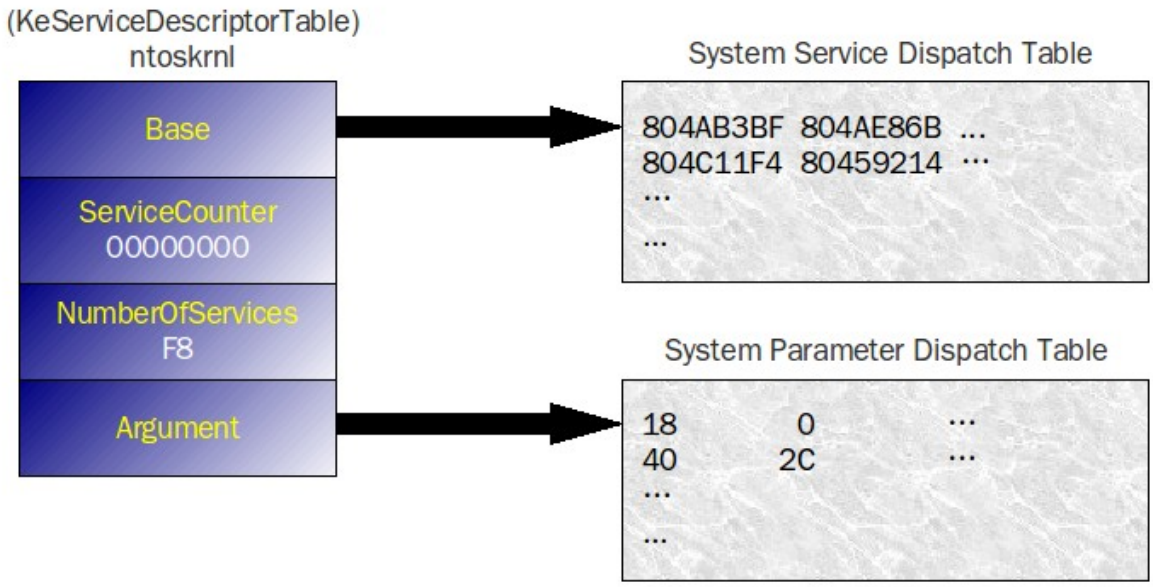


그림 *5: SSDT* 와 *SPDT*

[그림 5] 에서 ntoskrnl 은 SDE(Service Descriptor Entry)라는 구조체이다. 4 개의 SDE 가 모여서 SDT(Service Descriptor Table)를 이루는데, KeServiceDescriptorTable 의 첫 번째 SDE 가 ntoskrnl 이다.

SDE 에는 함수 테이블의 주소인 Base, 각 함수가 호출된 횟수가 저장되는 Service-Counter, 함수 테이블에 들어 있는 함수의 개수인 NumberOfServices, 각 함수의 인자의 크기를 저장하는 Argument 가 있다. 즉, SSDT 의 주소는 ntoskrnl 의 Base 에 기록되어 있다.

2.3.3 SSDT 후킹

SSDT 후킹의 원리는 간단하다. SSDT 안의 함수의 주소를 조작하여 내가 동작시키고 싶은 함수의 주소로 변경하는 것이다. 예를 들어서, 173 번 Native API 인 NtQuerySys-temInformation 의 주소를 내가 작성한 새로운 NtQuerySystemInformation 의 주소로 변경

하여 숨기고 싶은 프로세스를 프로세스 목록에서 숨길 수 있다.

SSDT 는 커널 영역에 위치하고 있고, 모든 프로세스들이 이 SSDT 를 참조하므로 모든 프로세스에 대하여 내가 원하는 작업을 할 수 있다. 이것이 SSDT 후킹이 전역적인 후킹이라고 말하는 이유이다.

하지만 SSDT 후킹이 반드시 전역적이라고 볼 수는 없다. 특정 스레드에 대하여 임의의 SDT 를 생성하여 연결하는 방법이 나왔기 때문이다. 이 방법을 사용하면 ntoskrnl 의 SSDT 를 후킹하더라도 그 스레드의 SDT 로 접근할 수 없기 때문에 어떠한 조작도 할 수 없게 된다.

2.4 MBR 루트킷

2.4.1 MBR 이란?

MBR (Master Boot Record)은 1 sector(512 bytes)의 공간으로 디스크의 0 번째 섹터 이다. Boot Record 들의 메인 격이라고 할 수 있다. BR(Boot Record)은 각 파티션의 첫 번째 섹터에 위치하며, 주된 목적은 해당 파티션에 설치된 OS 를 부팅해 주는 역할을 한다. 즉, OS 를 실행하기 위해 부트로더(Boot Loader)를 호출하는 것이다. 만약 파티션을 나누지 않은 상태라면 부트 레코드의 MBR 위치에 기록되어 있을 것이다.

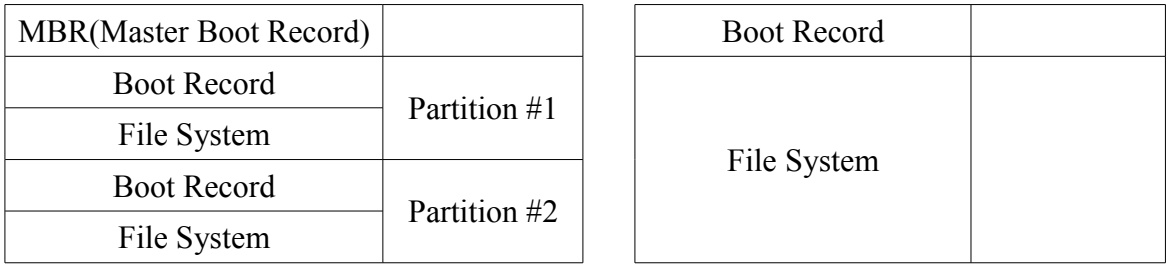


그림 *6:* 파티션이 나누어진 하드디스크*(*왼쪽*)*와 나누어지지 않은 하드디스크*(*오른쪽*)*

[그림 6] 과 같이 하드디스크의 첫번째 섹터(0 번 섹터)에 MBR 이 존재하며 파티션이 나누어 져 있지 않은 하드디스크에는 MBR 자리에 BR 이 존재한다. 파티션이 여러개 존재 할 경우에는 MBR 은 BR 을 읽어 BR 이 OS 를 작동하게 한다. 그런 이유로 MBR 영역에는 Boot Code 와 Partition Table 이 존재한다. 이런 MBR 영역은 BIOS 가 부른다.

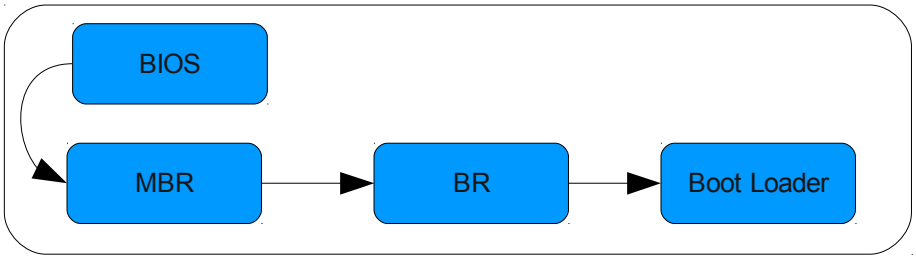


그림 *6:* 부팅 과정

즉 OS 까지 부팅하는 과정은 [그림 6] 과 같다.

2.4.2 MBR 의 구조

```
Absolute Sector 0 (Cylinder 0, Head 0, Sector 1)

       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000   33 C0 8E D0 BC 00 7C FB 50 07 50 1F FC BE 1B 7C  3.....|.P.P....|
0010   BF 1B 06 50 57 B9 E5 01 F3 A4 CB BD BE 07 B1 04  ...PW...........
0020   38 6E 00 7C 09 75 13 83 C5 10 E2 F4 CD 18 8B F5  8n.|.u..........
0030   83 C6 10 49 74 19 38 2C 74 F6 A0 B5 07 B4 07 8B  ...It.8,t.......
0040   F0 AC 3C 00 74 FC BB 07 00 B4 0E CD 10 EB F2 88  ..<.t...........
0050   4E 10 E8 46 00 73 2A FE 46 10 80 7E 04 0B 74 0B  N..F.s*.F..~..t.
0060   80 7E 04 0C 74 05 A0 B6 07 75 D2 80 46 02 06 83  .~..t....u..F...
0070   46 08 06 83 56 0A 00 E8 21 00 73 05 A0 B6 07 EB  F...V...!.s.....
0080   BC 81 3E FE 7D 55 AA 74 0B 80 7E 10 00 74 C8 A0  ..>.}U.t..~..t..
0090   B7 07 EB A9 8B FC 1E 57 8B F5 CB BF 05 00 8A 56  .......W.......V
00A0   00 B4 08 CD 13 72 23 8A C1 24 3F 98 8A DE 8A FC  .....r#..$?.....
00B0   43 F7 E3 8B D1 86 D6 B1 06 D2 EE 42 F7 E2 39 56  C..........B..9V
00C0   0A 77 23 72 05 39 46 08 73 1C B8 01 02 BB 00 7C  .w#r.9F.s......|
00D0   8B 4E 02 8B 56 00 CD 13 73 51 4F 74 4E 32 E4 8A  .N..V...sQOtN2..
00E0   56 00 CD 13 EB E4 8A 56 00 60 BB AA 55 B4 41 CD  V......V.`..U.A.
00F0   13 72 36 81 FB 55 AA 75 30 F6 C1 01 74 2B 61 60  .r6..U.u0...t+a`
0100   6A 00 6A 00 FF 76 0A FF 76 08 6A 00 68 00 7C 6A  j.j..v..v.j.h.|j
0110   01 6A 10 B4 42 8B F4 CD 13 61 61 73 0E 4F 74 0B  .j..B....aas.Ot.
0120   32 E4 8A 56 00 CD 13 EB D6 61 F9 C3 49 6E 76 61  2..V.....a..Inva
0130   6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61  lid partition ta
0140   62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E  ble.Error loadin
0150   67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74  g operating syst
0160   65 6D 00 4D 69 73 73 69 6E 67 20 6F 70 65 72 61  em.Missing opera
0170   74 69 6E 67 20 73 79 73 74 65 6D 00 00 00 00 00  ting system.....
0180   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0190   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01A0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01B0   00 00 00 00 00 2C 44 63 A8 E1 A8 E1 00 00 80 01  .....,Dc........
01C0   01 00 07 7F BF FD 3F 00 00 00 C1 40 5E 00 00 00  ......?....@^...
01D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01E0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
01F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA  ...............U.
       0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

그림 *7: MBR code in sector0*

[그림 7] 에서 주황색 부분이 446 byte 의 Boot Code 이다. Boot Code 이후 00 00 00 ..... 까지는 에러 코드 이다. 그리고 2C 44 63 A8 E1 A8 E1 부분은 Disk Signature 이다. 그 다음 80 부터 55AA 전까지 64 byte 가 파티션 테이블이다. 마지막 55AA 는 Magic Code 이다. 이로써 우리는 MBR 코드의 역할과 그 구조를 간단하게 볼수 있었다.

2.4.3 MBR 영역은 왜 위험한가?

MBR 영역에서 우리가 주의 깊게 봐야 할 영역은 처음 0x00 부터 0x012b 까지 300byte 영역 이다. 그 부분에는 실행영역으로 BR 을 읽어들이는 INT13 명령을 사용하는 부분이 존재한다.

```
06CA B80102     MOV  AX, 0201     ; INT 13,  Function 2
06CD BB007C     MOV  BX, 7C00     ; (ES):BX = Memory Buffer
06D0 8B4E02     MOV  CX, [BP+02]  ;
06D3 8B5600     MOV  DX, [BP+00]  ;
06D6 CD13       INT  13           ; "Read 1 Sector into Memory"
```

위 코드는 메모리에 로드되었던 코드를 디어셈블리 해서 나타낸 코드 부분이다. INT 13 을 이용하여 1 sector 를 메모리로 읽어 들이는 부분이다. 이 1 sector 는 OS 가 설치된 파티션의 BR 일것이다.

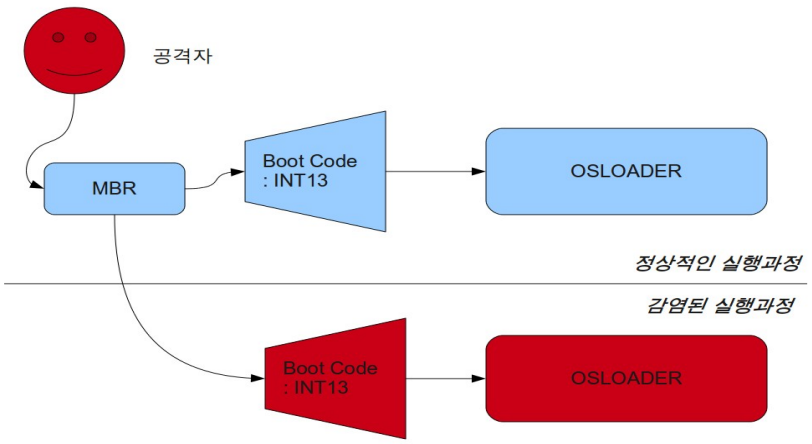


그림 *8:* 공격과정

공격자의 코드는 부팅 과정시 MBR 에 덮어 씌워진다. 이 덮어진 MBR 코드는 [그림 8] 과 같이 INT13 을 후킹하여 Windows 의 정상적인 OSLOADER 대신 공격자가 만들어 놓은 OSLOADER 를 로드하게끔한다.

공격자가 이렇게 침투할 수 있는 첫 번째 이유는 MBR 이라는 영역은 어떠한 보호도 되어있지 않기 때문이다. 그리고 두 번째 이유는 이러한 MBR 영역이 로드되는 시점은 Real Mode 로 Ring 0, 즉 최고 권한을 가진다. 이러한 이유로 공격자는 MBR 을 건드리고, 건드려진 MBR 로 운영체제의 모든 것을 제어할수 있는 권한을 얻는 것이다.

2.4.4 MBR 루트킷의 한계

하지만 이러한 MBR 의 공격은 그 공격보다는 방어가 쉬운편이다. 공격 루틴을 정확히 찾아내기는 힘들지만 우리의 MBR 은 항상 동일하다(같은 플랫폼). 그래서 뭔가가 의심스럽다면 MBR 을 다시 덮어씌워주면 되는것이다. 그 방법은 너무나도 간단한다.

```
fdisk /mbr
```

다음 명령만으로 MBR 은 정상적으로 복구가 된다. 물론 유저가 이상점을 발견하여 저 명령을 한다는 보장은 별로 안되지만 말이다.

3 루트킷의 탐색

3.1 유저 레벨 루트킷

'메모리 스캐닝'과 달리 후킹을 검색하는 것은 일반적인 접근이 가능하다. 하지만 이는 특정한 루트킷의 시그니처나 패턴을 알고 있어야 한다. 후킹을 알아내는 기본적인 알고리즘은 허용된 범위 위부로의 실행 분기를 찾아 내는 것이다. 그런 실행 분기는 call 또는 jmp 명령에 의해 이뤄진다. 프로세스의 IAT(Impoart Address Table) 안에는 Import 함수를 포함하고 있는 모듈들의 이름이 나열되어 있다. 그러한 모듈들은 모두 정의된 메모리상의 시작 주소와 크기를 가지고 있다.

1) IAT 후킹 탐지를 수행할 프로세스의 주소 공간으로 컨텍스트를 변경한다. 이를 위해 탐지를 수행하는 코드가 해당 프로세스 안에서 수행 되도록 한다.

2) 해당 프로세스가 로드한 모든 DLL 들의 리스트를 구한다. 그리고 프로세스의 IAT 를 조사해서 프로세스가 임포트해서 사용하는 함수가 어느 DLL 에 존재하는 것인지 판단하고, 그 함수의 주소가 해당 DLL 의 영역 안에 존재하는지 검사한다.

3.2 커널 레벨 루트킷

3.2.1 SSDT 후킹 탐색

SSDT 후킹은 커널 영역의 메모리를 조작한다. 만약 NtQueryServiceInformation 함수가 후킹을 당했다면, 이후에 이 함수에 의해 나오는 결과는 신뢰할 수 없다. 이미 공격자가 원하는대로 결과를 보여줄 것이기 때문이다.

그래서 SSDT 후킹을 탐색하기 위해서는 무결성이 보장되는 무언가가 필요하다. 그것이 바로 커널 이미지다. 커널 이미지는 바이너리 파일 형태로 존재하므로 단순히 메모리를 조작하는 SSDT 후킹으로는 커널 이미지 안의 SSDT 를 조작할 수 없다. Windows XP SP3 에서 커널 이미지는 C:\WINDOWS\system32\ 에 존재하는 ntoskrnl.exe 파일이다.

ntoskrnl.exe 에서 SDT 를 뽑아내어 SSDT 의 주소를 구한 후, 메모리에 있는 SSDT 와 비교를 한다. 당연히 비교한 결과가 다르다면 후킹이 된 것임을 알 수 있다. 비교를 하는 방법은 여러 가지가 있을 수 있다. 간단하게 Native API 함수의 주소가 원래의 범위를 벗어나는지를 확인해볼 수도 있고, 일일이 두 SSDT 의 함수 주소들을 직접 비교해 볼 수도 있다.

3.3 MBR 루트킷

3.3.1 MBR 의 검색

MBR 의 코드는 하드디스크의 sector 0 에 존재한다. 그렇다면 이러한 MBR 은 어떻게 복구/검사 할수 있을까? MBR 코드는 sector0 말고도 한 군데 더 존재한다. 정확히는 MBR 의 실행코드 즉 Boot Code 부분이 존재하는 파일이 있다. 이 파일은 \WINDOWS FOLDER\system32\dmadmin.exe 이다. MBR 영역은 dmadmin.exe 의 34E28h 에서 35027h 사이에 존재한다. 하지만 파티션 테이블이나 Disk Signature 의 자리에는 아무런 코드가 존재하지 않는다.

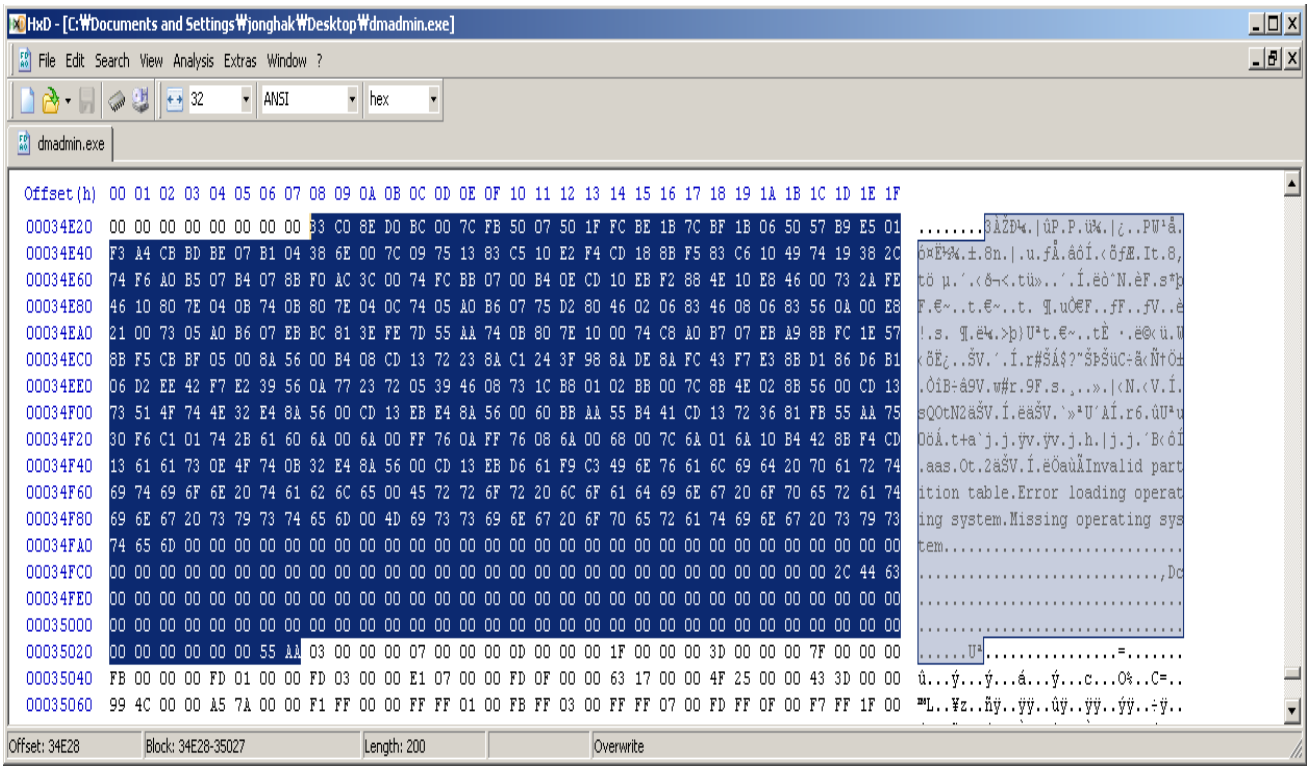


그림 *9: dmadmin.exe* 의 *MBR* 코드

우리는 [그림 9] 의 영역과 실제 MBR 코드를 비교하여 그 차이점을 발견하였을 경우 누군가의 침입이 있음을 간주할 수 있다.

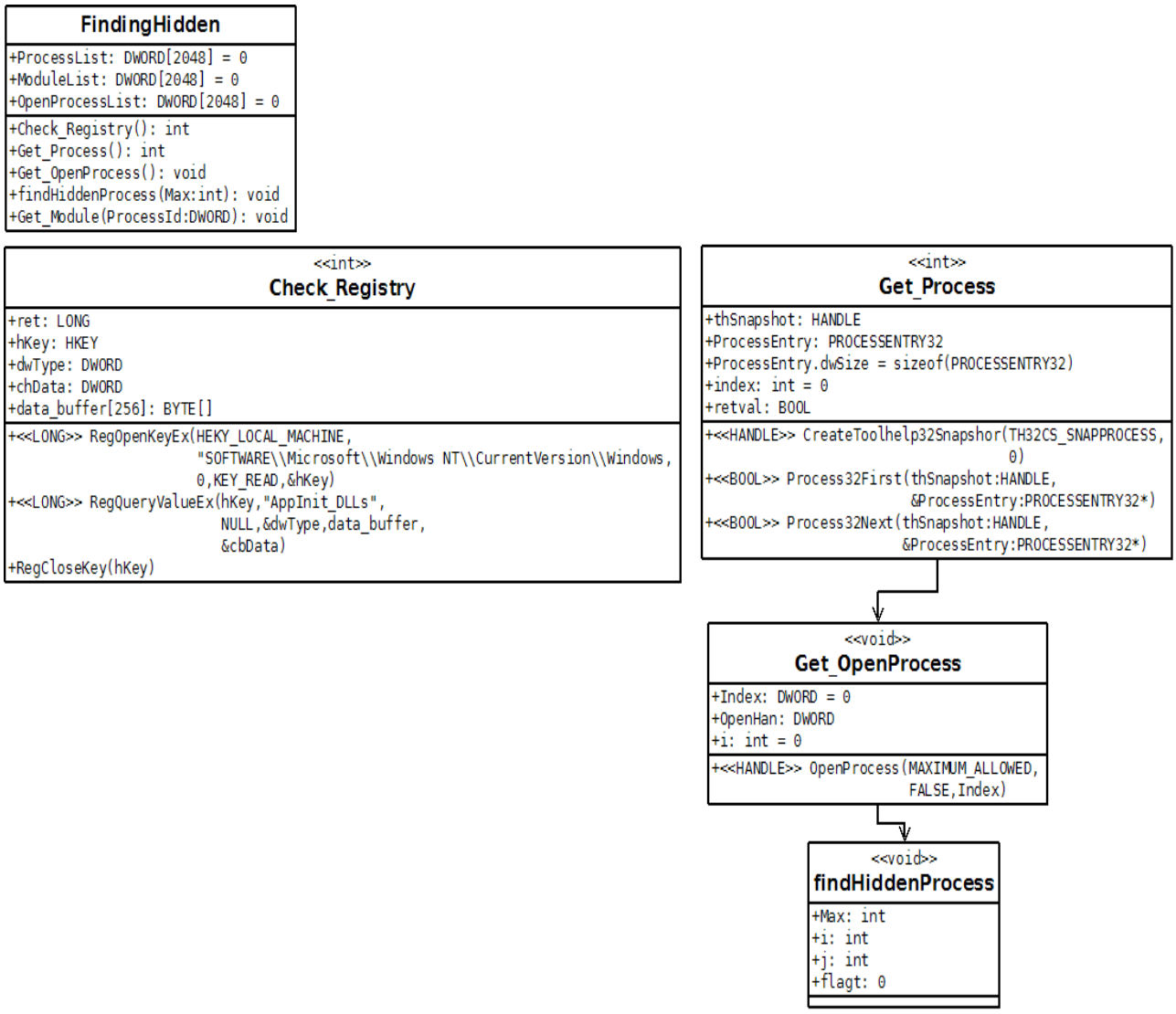
3.3.2 MBR 을 초기화 하는 방법

위에서도 언급했듯이 MBR 을 초기화하는 방법으 간단한 fdisk 명령으로 가능하다. 이 fdisk 의 명령은 새로운 MBR 코드를 덮어씌우는 것인데 공격자는 MBR 의 Boot Code 만을 이용하기 때문에 우리는 dmadmin.exe 의 Boot Code 를 MBR 에 덮어씌우는 방법으로 MBR 을 초기화 하려고 한다.

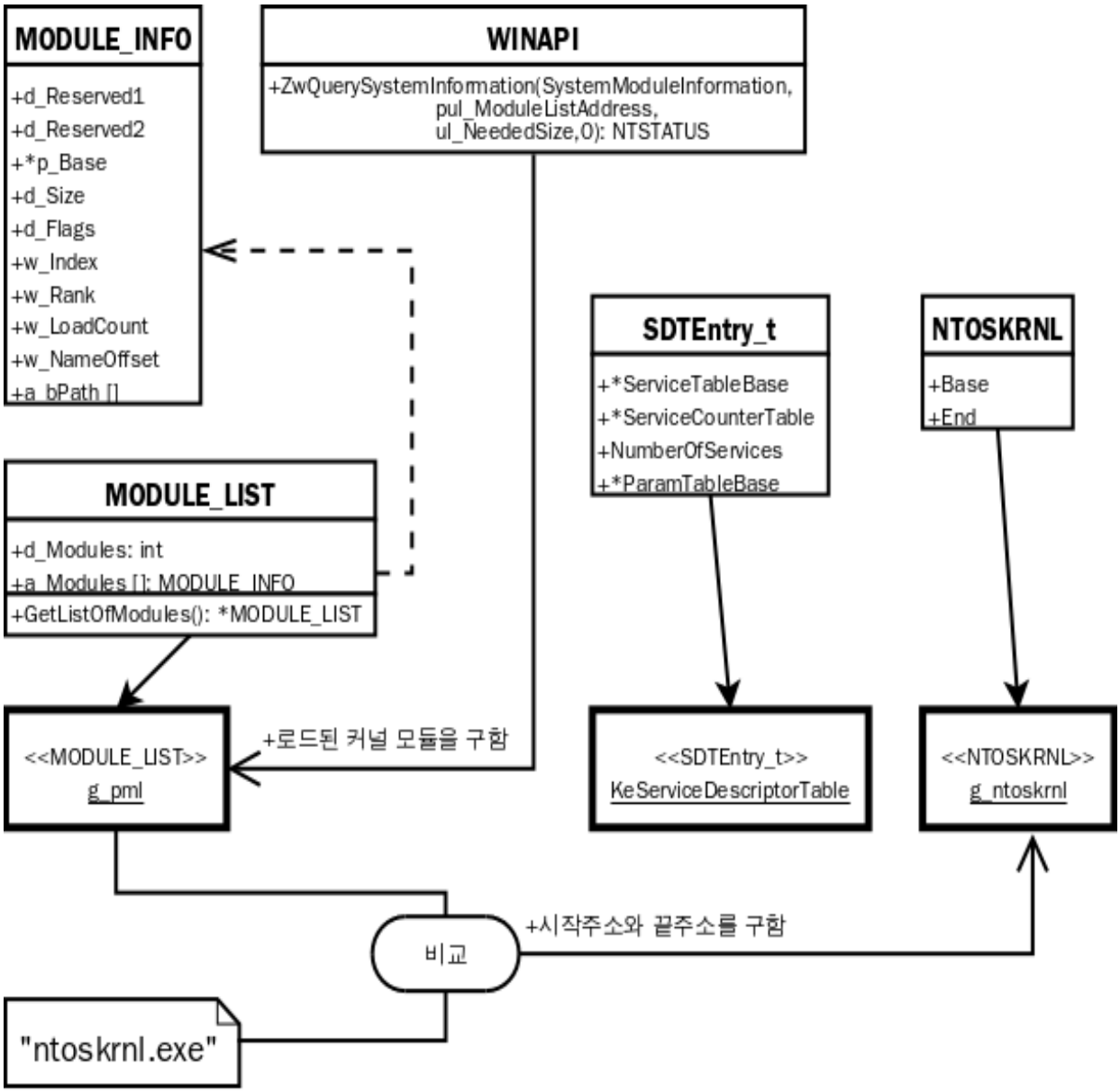
4 프로젝트 결과

4.1 설계

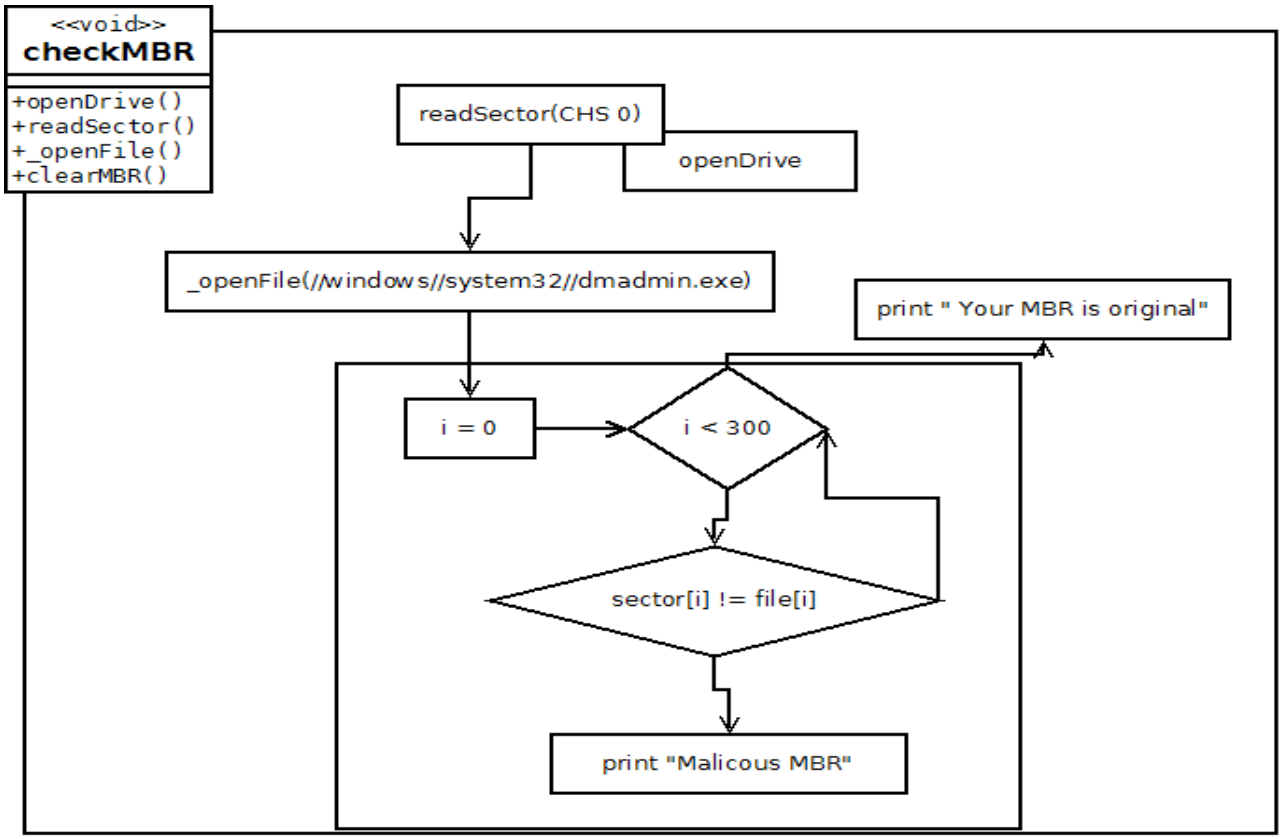
4.1.1 유저 레벨 : findingHidden

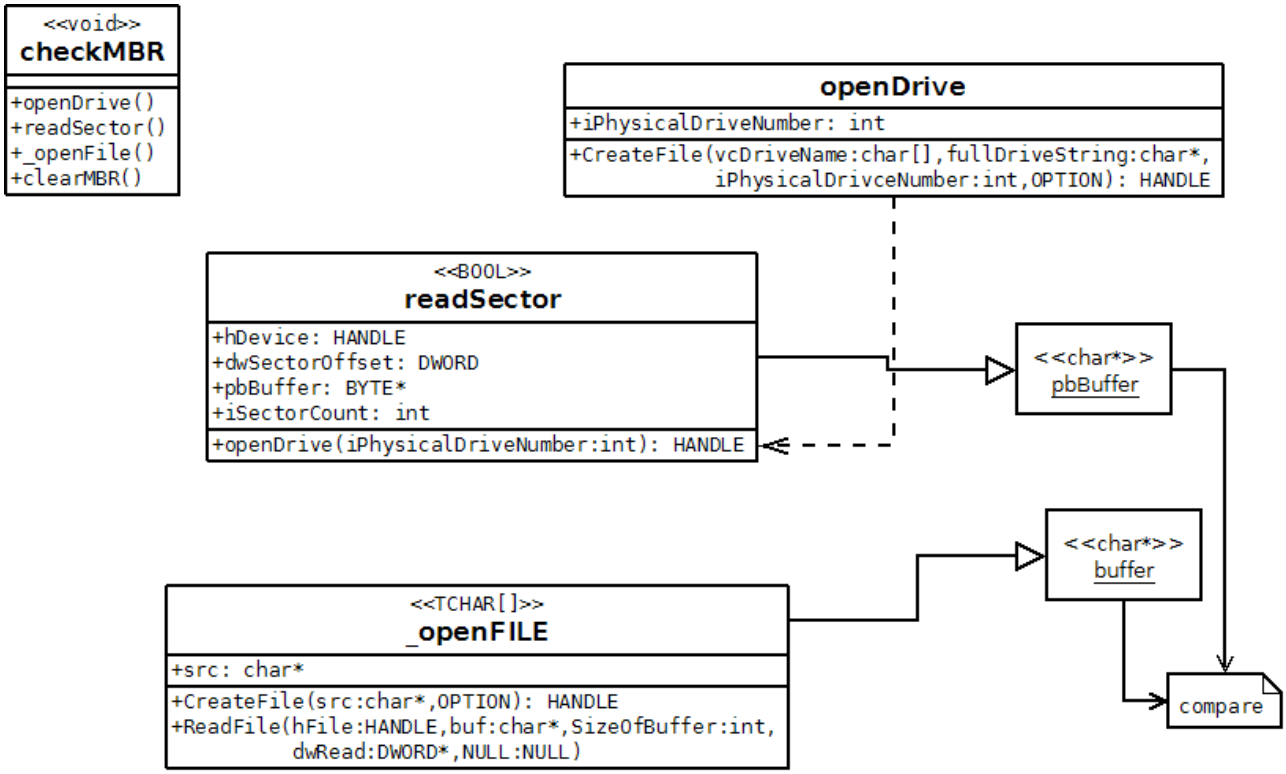


4.1.2 커널 레벨 : checkSSDT



4.1.3 디스크(MBR) : checkMBR





4.2 세부 루틴

4.2.1 유저 레벨 : findingHidden

```
/*===Function=============================================
 * 레지스트리의 AppInits 값을 체크하여 dll_injection을 검사한다.
 * Return value : success : 1, fail : 0
 *========================================================*/
int Check_Registry();

/*====Function============================================
 * API를 통해 메모리 상의 모든 프로세스 목록을 받아온다.
 * return value : 프로세스의 개수
 *========================================================*/
int Get_Process();

/*====Function============================================
 * 0x0 ~ 0x41DC(PID의 범위값)에 모든 OpenProcess()를 사용하여
 * 반환값이 NULL이 아닌 프로세스 Id목록을 생성한다.
 ========================================================*/
void Get_OpenProcess();

/*=======Function=========================================
 * 두 리스트를 비교하여 숨겨진 프로세스 목록을 출력한다.
 *========================================================*/
void findHiddenProcess(int Max);

/*
getting list of Module
but not used
*/
void Get_Module(DWORD ProcessId);
```

```
int Check_Registry(){

	LONG ret;
	HKEY hKey;
	DWORD	dwType, cbData;
	BYTE data_buffer[256];

	//success opening Registry Key
	if((ret=RegOpenKeyEx(HKEY_LOCAL_MACHINE ,
	"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Windows",
		0,
		KEY_READ,
		&hKey	))==ERROR_SUCCESS) {
		cbData = 256;
		//Reading Registry_Value
		if((ret = RegQueryValueEx(hKey, "AppInit_DLLs", NULL, &dwType, data_buffer,
&cbData))==ERROR_SUCCESS){
			if(!data_buffer){
				printf("Registry data : %S\n", data_buffer);
				printf("Please checking your System\n");
			}
			else
				printf("no value detected\n");
		}

	}

	//fail to Openning Registry key
	else
		return 0;

	//close Registry
	RegCloseKey(hKey);

	return 1;
}
```

```
void findHiddenProcess(int Max)
{
	int i,j;
	int flagt=0; //일치 여부를 검사할 플래그

	for(i=0;i<Max;++i){
		for(j=0;j<Max;++j){
			if(OpenProcessList[i] == ProcessList[j])
				flag++; // 리스트를 비교하면서 같은 값이 있으면 +1
		}

		// 리스트에 일치하지 않는 프로세스 ID 출력
		if(flag == 0){
			printf("finding Hidden Process\n");
			printf("Pid : %d\n",OpenProcessList[i]);
		}

		count = 0;
	}
}
```

```
int main (void)
{
}
```

4.2.2 커널 레벨 : checkSSDT

```
/*
 * ===  FUNCTION  ======================================================
 *         Name:  ZwQuerySystemInformation
 *  Description:  Retrieves the specified system information.
 *    Parameter:  SystemInformationClass - The type of system information
 *                *SystemInformation - A buffer that receives information
 *                SystemInformationLength - The size of the buffer
 *                *ReturnLength - [Option]
 *       Return:  NTSTATUS success or error code
 * ======================================================================
 */
NTSYSAPI NTSTATUS NTAPI ZwQuerySystemInformation (
       IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
       IN unsigned int *SystemInformation,
       IN unsigned long SystemInformationLength,
       UT unsigned long *ReturnLength
);
```

```
/*
 * ===  FUNCTION  ======================================================
 *         Name:  GetListOfModules
 *  Description:  모듈 정보를 저장하기 위해 메모리를 할당한다.
 *                모듈 정도를 구한다.
 *    Parameter:  pns - 디버깅용, NULL
 *       Return:  pml - 로드된 커널 모듈의 정보
 * ======================================================================
 */
PMODULE_LIST GetListOfModuels (PNTSTATUS pns);
```

```
/*
 * ===  FUNCTION  ======================================================
 *         Name:  IndentifySSDTHooks
 *  Description:  SSDT 안의 주소들 중에서 허용 영역 외부에 존재하는 것을 검사한다.
 *    Parameter:
```

```
 *       Return:
 * ============================================================================
 */
void IndentifySSDTHooks (void);
```

```
NTSTATUS DriverEntry (IN PDRIVER_OBJECT DriverObject,
                      IN PUNICODE_STRING RegistryPath)
{
       int count;

       g_pml = NULL;
       g_ntoskrnl.Base = 0;
       g_ntoskrnl.End = 0;
       /*-------------------------------------------------------------------------
        *  로드된 모듈 리스트를 구한다.
        *-------------------------------------------------------------------------*/
       g_pml = GetListOfModuels (NULL);

       DriverObject->DriverUnload = Unload;
       DbgPrint ("CHECKSSDT loaded.\n");

       if (!g_pml)
              return STATUS_UNSUCCESSFUL;

       /*-------------------------------------------------------------------------
        *  모듈 리스트에서 "ntoskrnl.exe"를 찾는다.
        *-------------------------------------------------------------------------*/
       for (count = 0; count < g_pml->d_Modules; count++)
       {
              if (_stricmp ("ntoskrnl.exe", g_pml->a_Modules[count].a_bPath + g_pml-
>a_Modules[count].w_NameOffset) == 0)
              {
                     g_ntoskrnl.Base = (unsigned int)g_pml->a_Modules[count].p_Base;
                     g_ntoskrnl.End = ((unsigned int)g_pml->a_Modules[count].p_Base + g_pml-
>a_Modules[count].d_Size);
              }
       }
```

```
	ExFreePool (g_pml);

	/*-----------------------------------------------------------------------
	 *  Check the SSDT hooking.
	 *-----------------------------------------------------------------------*/
	IndentifySSDTHooks ();

	if (g_ntoskrnl.Base != 0)
		return STATUS_SUCCESS;
	else
		return STATUS_UNSUCCESSFUL;
}
```

4.2.3 디스크(MBR) : checkMBR

```
HANDLE
openDrive(int iPhysicalDriveNumber)
{
	HANDLE hDevice;
	char vcDrivceName[30];

	/* HDD를 지우는 것을 방지*/
	if(iPhysicalDriveName == 0) {
		return INVAILD_HANDLE_VALUE;
	}
	sprintf(vcDriveName, "\\\\.\\PhysicalDrive%d", iPhysicalDrivceNumber);
	hDevice = CreateFile(vcDrivceName, GENERIC_READ | GENERIC_WRITE,
			FILE_SHARE_READ | FILE_SHARE_WRITE,  NULL,
			OPEN_EXISTING, 0, NULL);
	return hDevice;
}
```

```
BOOL
readSector(HANDLE hDevice, DWORD dwSectorOffset, BYTE* pbBuffer, int iSectorCount)
{
	DWORD dwLow;
	DWORD dwHigh;
	DWORD dwRet;
	DWORD dwRead;
	DWORD dwErrorCode;

	/* 1sector = 512 byte : dwSectorOffset * 512 */
	dwLow = (dwSectorOffset << 9);
	dwHigh = (dwSectorOffset >> 32);

	/* move the File pointer*/
	dwRet = SetFilePointer(hDevice, dwLow, &dwHigh, FILE_BEGIN);
	if(dwRet == INVAILD_SET_FILE_POINTER) {
		return FALSE;
```

```
	}

	/* read sector*/
	if(ReadFile(hDevice, pbBuffer, iSectorCount * SECTORSIZE, &dwRead, NULL) == FALSE) {
		dwErrorCode = GetLastError();
		return FALSE;
	}
	return TRUE;
}
```

```
TCHAR[]
_openFILE(char* src)
{
	TCHAR buf[81] = {0};
	HANDLE hFIle;

	/* ReadFile : c:\WINDOWS\system32\dmadmin.exe */
	/* read the only 80byte of execution code */
	hFile=CreateFile(src,
			GENERIC_READ,0,NULL,
			OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);

	if (hFile != INVALID_HANDLE_VALUE) {
		ReadFile(hFile,buf,1024,&dwRead,NULL);
		CloseHandle(hFile);
		return buf;
	}
	return NULL;
}
```

```
void
cleanMBR(void)
{
}
```

```
int
main(int argc, char* argv)
```

```
{
	int idx = 0;
	TCHAR buf[81] = {0};
	BYTE* pbBuffer;
	HANDLE hDevice = openDrive(1);
	DWORD dwErrorCode;

	if(!readSector(hDevice, 1, pbBuffer, 1)) {
		dwErrorCode = GetLastErro();
		return 1;
	}

	if((buf = _openFile("c:\\WINDOWS\\system32\\dmadmin.exe")) == NULL) {
		dwErrorCode = GetLastError();
		if((buf = _openFile("c:\\WINNT\\system32\\dmadmin.exe")) == NULL)  {
			dwErrorCode = GetLastError();
			return 1;
		}
		return 1;
	}

	/*check the MBR sector*/
	for(idx = 0; i < 300; i++) {
		if(buf[idx] != pbBuffer[idx]) {
			fprintf(stderr, "Malicious....\n");
			cleanMBR();
			return 1;
		}

	}
	fprintf(stdout, "MBR Sector is clean.\n");
	return 0;
}
```

5 결론

5.1 연구 결과

자기 자신을 숨기는 것과 그것을 찾는 방법은 그 방법에 있어 매우 유사하다. 루트킷의 구현과 그 탐지의 구현이 크게 다르지 않다는 것을 보여주는 하나의 예인 것 같다. 앞서 우리의 목표는 커널 레벨에서의 SSDT 후킹 탐지, 유저 레벨에서의 레지스트리/API 후킹 탐지, 마지막으로 디스크의 숨겨진 파일 및 MBR 검색 등을 그 목적으로 하였다. 루트킷이라는 이름답게 유저 레벨 기술보다 커널 레벨 기술을 이용했을 때 그 효과는 더욱 컸다. 마찬가지로 탐색 역시 유저 레벨에서의 탐색보다 커널 레벨에서의 탐지가 더욱 효과적이고 광범위한 탐색이 가능하였다. 그 외에 MBR 루트킷의 경우처럼 시스템이 시작함과 동시에 실행하여 유저 레벨도 커널 레벨도 아닌 곳에서 실행되는 것도 있었다. 이는 기본 커널과 유저 레벨 로 나뉘는 OS 레벨을 넘어서서 하드웨어나 기타 다른 장치로 부터의 루트킷이 더 많이 나올 수 있다는 것을 예상하게하는 좋은 예시였던 것 같다.

5.2 프로젝트 수행시 어려웠던 점 등 기타 의견

실제 존재하는 루트킷들을 탐지 해낼수 있는 수가 제한적이었다는 게 아쉬운 점으로 남는다. 실제 결과물은 커널 레벨에서는 SSDT 후킹 탐지, 유저 레벨에서는 레지스트리/API 후킹 탐지, 디스크에서는 MBR 검색을 구현하였다.

좀 더 보강하고 싶은 부분이 있다면 커널 레벨에서의 탐지의 확장과 그에 따른 안정성을 더욱 높이고 싶다. 커널 레벨 기술은 그 사용이 매우 조심스러워야 하기 때문에 누군가가 이 프로그램을 사용하였을 경우 시스템에 오류를 일으키지 않아야 한다. 이는 프로그램이 가져야 할 가장 기본적인 원칙으로 커널 레벨 기술을 사용하면서 이러한 원칙을 지키는 것이 쉬운 일은 아니라는게 우리 생각이다.

6 참고문헌

[1] 루트킷:커널 조작의 미학 – 그렉 호글런드, 제임스 버틀러 저

[2] 임베디드 개발자를 위한 파일시스템의 원리와 실습 – 정준석, 정원용 저

[3] 윈도우 디바이스 드라이버 - 이봉석 저

[4] 루트킷의 진화 (http://www.kaspersky.co.kr/board/bbs/board.php?bo_table=Products&wr_id=191&sca=&sfl=wr_subject||wr_content&stx=rootkit&sop=and&nca=)

[5] Boot Root - Derek Soeder, Ryan Permeh (http://research.eeye.com/html/tools/RT20060801-7.html)

[6] GMER (http://www.gmer.net/)

[7]「Windows Rootkit」by 돗가비

[8]「커널모드 루트킷 기술 – SDT 후킹의 창과 방패」by 신영진

[9]「SSDT Hooking 탐지 기법」by CRG 김범연