DEVGURU

본 컬럼에 대한 모든 저작권은 DevGuru에 있습니다.
컬럼을 타 사이트 등에 기재 및 링크 또는 컬럼 내용을 인용 시 반드시 출처를 밝히셔야 합니다.
컬럼 들을 CD나 기타 매체로 배포하고자 할 경우 DevGuru에 동의를 얻으셔야 합니다.

ⓒ DevGuru Corporation. All rights reserved

기타 자세한 질문 사항들은 웹 게시판이나 support@devguru.co.kr으로 문의하기 바랍니다.

DEVGURU

Attack Native API

written by Kwak Taejin(bluewarz@devguru.co.kr)

1부 Look around Native API

2부 Hooking Native API

"System service API가 무엇인가?"
"System service API 의 목적이 무엇인가"
"System service API 중 ZwXXX 계열과 NtXXX 계열의 함수의 차이는 무엇인가?"
"User mode Application이 Kernel service를 어떤 경로에 통해서 접근하는가?"

위의 질문에 대답할 수 있다면 System Programmer로서 어느 정도의 내공의 소유자일 것이다.

위의 질문에 대하여 답변을 하나하나 찾아가보자. 답을 찾다 보면 우린 최종 목표인 Native API를 Hooking하여 System을 감시하는 Driver의 원리를 파악 할 수 있을 것이다.

What & Purpose

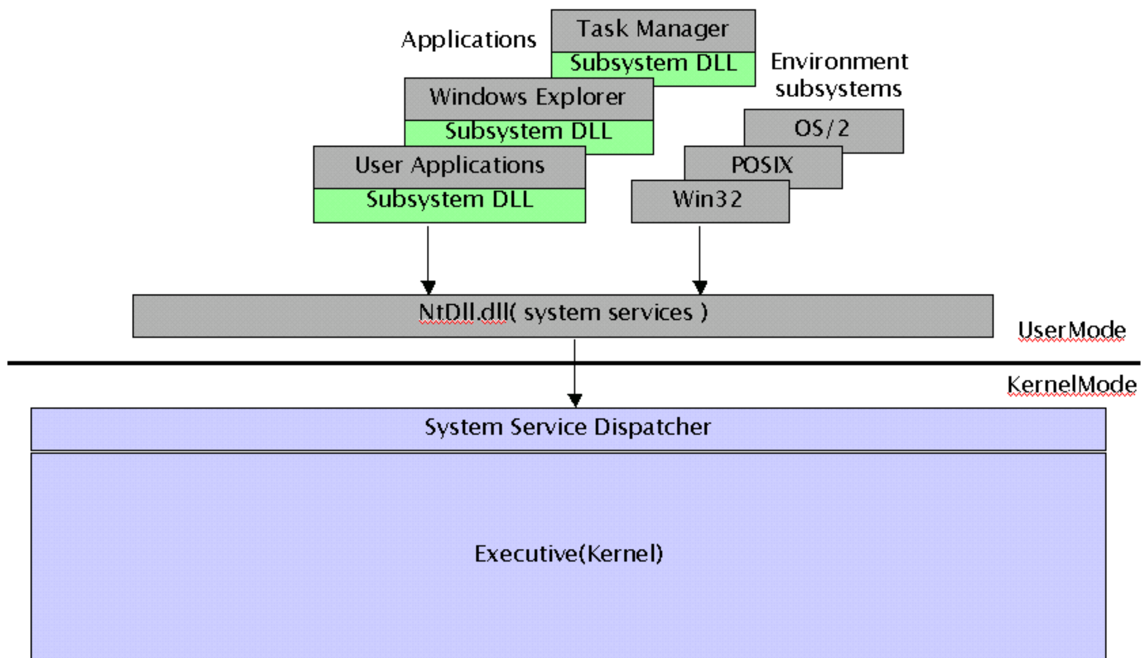
자.. 그럼 첫번째와 두번째 질문에는 무엇이라 대답 하겠는가?
System service API에 대해서 잘 모르는 사람들은 Win32 API 의 일부분 또는 비슷한 기능을 하는 부분 이라 생각을 많이 할 것이다. 그러나 System 쪽을 공부 했다면 Win32 API는 NT 계열 OS에서 지원 Subsystem의 중의 하나 인 것과, 이러한 Subsystem 들이 NT Executive(Kernel)의 도움을 받고자 할 때 System service API를 이용 하다는 것을 알 것이다. 즉, system service는 user mode application(Kernel mode application도 일부 포함) 들이 Kernel service를 받고자 할 때 사용되어지는 함수들의 모임이다. 이 API를 NT Native API 라 부른다.(이하 Native API)

NT계열 OS에서는 다른 Operation System에서 작성된 Application의 호환성을 유지하기 위하여 다양한 subsystem(Win32, POSIX, OS/2, DOS/WoW)들을 제공해준다. 각각의 subsystem은 다른 OS를 에뮬레이션 해주며 Kernel의 도움이 필요할 때는 Native API를

이용한다.

Native API 은 모든 subsystem에게 동일한 Interface와 기능을 제공하며, 이로 인하여 다른 OS의 Application들은 subsystem에 의존하여 NT계열 OS에서 실행이 가능 하다.

물론 꼭 subsystem을 이용 하여 Native API를 이용하라는 규정은 없다. 그러나 그것은 여러 가지 문제(security, handle managing 등) 가 발생 할 수 있으며, 또한 Microsoft 사에는 Native API를 end-user를 위하여 디자인 한 것이 아니어서 약 10% 정도만이 문서화 되어 공개되어 있으며 나머지는 비공개로 되어 있기 때문에 접근하기가 쉽지 않다. 하지만 많은 system programmer들은 스킬을 이용하여 원하는 기능의 Native API를 찾아서 사용하고 들 있다.



<Environment Subsystem and Subsystem DLLs>

간단히 Native API 함수를 보자.

NtCreateFile의 Prototype은 다음과 같다. 이는 ZwCreateFile과 같은 Parameters를 가지고 있으며, ZwCreateFile은 NT DDK안에서 찾아볼 수 있다.

Native API는 두가지 종류가 있다는 것인가? 맞다.

Native API는 ZwXxxx 계열과 NtXxxx 계열 두 가지가 있다. 이 둘을 자세히 알아보자.

DEVGURU

```
NTSTATUS
  NtCreateFile(
            OUT PHANDLE FileHandle,
            IN ACCESS_MASK DesiredAccess,
            IN POBJECT_ATTRIBUTES ObjectAttributes,
            OUT PIO_STATUS_BLOCK IoStatusBlock,
            IN PLARGE_INTEGER AllocationSize  OPTIONAL,
            IN ULONG FileAttributes,
            IN ULONG ShareAccess,
            IN ULONG CreateDisposition,
            IN ULONG CreateOptions,
            N PVOID EaBuffer  OPTIONAL,
            IN ULONG EaLength  );
```

위 함수는 우리가 자주 보는 CreateFile 과 크게 다르게 보이지 않을 것이다. 다만, 전달해야 할 parameters 의 type들이 생소하게 느껴지는 것이 몇 가지 있을 것이다. 사용 방법 및 인자 type은 자세히 설명을 하지 않겠다. DDK 및 참고서적을 보면 자세일 알 수 있을 것이다.

NtXxxx 함수와 ZwXxxx의 차이는 이렇게 정의가 되어 있다. NtXxxx 계열 함수는 보안을 위하여 함수의 Parameter와 access mode 유효 검사를 한다. 그리고 ZwXxxx는 이러한 부가적인 검사를 하지 않는다. 그렇기 때문에 Kernel 자신은 어떤한 제약도 없이 resource를 접근 하기위하여 ZwXxxx함수들을 대부분 이용하고 NtXxxx함수는 UserMode에서 service 요청 시에 사용되어 진다.

NtXxxx vs ZwXxxxx

좀더 자세히 Native API를 한번 파해쳐 보자. 파해치다 보면 원리를 파악할 테고 그 원리를 가지고 우리는 Native API를 공격 할 수 있을 것이다.

다음과 같은 4가지의 경우를 생각 할 수 있을 것이다.

User Mode Application에서 NtXxxxx 함수 호출
User Mode Application에서 ZwXxxxx 함수 호출
Kernel Mode Application에서 NtXxxxx 함수 호출
User Mode Application에서 NtXxxxx 함수 호출

User Mode 에서의 호출

User Mode application은 kernel의 도움을 받기 위하여 Native API를 호출한다. 호출 경로는 아마도 Application이 subsystem의 Native API를 호출할 것이다. Subsystem은 NTDLL.LIB 을 링크 하므로 우리는 Ntdll.dll을 이용하여 그 해당 코드를 볼 수 있다.

Soft-ice를 이용하여 NtReadFile의 코드를 확인해보자.

```
:u ntdll!NtReadFile
ntdll!NtReadFile
001B:77F889DC   MOV        EAX,000000A1
001B:77F889E1   LEA        EDX,[ESP+04]
001B:77F889E5   INT        2E
001B:77F889E7   RET        0024
```

위와 같은 코드가 나올 것이다.

◈ Symbol Loader 의 Load exports 이용하여 Ntdll.dll 로드 해야만 볼 수 있을 것이다.

ZwReadFile도 확인해보자.

```
:u ntdll!ZwReadFile
ntdll!NtReadFile
001B:77F889DC   MOV        EAX,000000A1
001B:77F889E1   LEA        EDX,[ESP+04]
001B:77F889E5   INT        2E
001B:77F889E7   RET        0024
```

ZwReadFile은 바로 NtReadFile로 포워딩 되어 있는 것을 확인 해볼 수 있다.
결국 User mode에서는 NtXxxx 계열만을 이용하는 것이며, 아마도 부가적인 검사(parameters, security 등의 검사)을 하기 위해서 일 것이다.

위의 형식은 EAX(001B:77F889DC) 에는 함수의 index(0xA1는 NtReadFile의 index일 것이다.)가 들어가며, EDX(001B:77xxxxxx)에는 현재 User mode의 stack의 주소가 들어간다. 한마디로 함수의 인자들과 스택을 전달하는 것이다.
그리고 INT 2E가 실행 되어질 것이고 제어권은 Kernel로 넘어 갈 것이다. INT 2E는 후반부에 자세히 살펴보기로 하자.

Kernel Mode에서 호출

Kernel Mode에서는 Ntoskrnl.lib를 이용하므로 Native API는 NTOSKRNL에 있을 것이다.
Soft-Ice를 이용하여 ZwReadFile의 코드를 보자.

```
:u ntoskrnl!zwreadfile
ntoskrnl!ZwReadFile
0008:8042EAE4   MOV     EAX,000000A1
0008:8042EAE9   LEA     EDX,[ESP+04]
0008:8042EAED   INT     2E
0008:8042EAEF   RET     0024
```

앗~!! User Mode의 Native API들과 같은 코드가 존재 하고 있다.

그럼 NtReadFile도 봐야 할 것 같다.

```
:u ntoskrnl!NtReadfile
ntoskrnl!NtReadFile
0008:804A86BA   PUSH    EBP
0008:804A86BB   MOV     EBP,ESP
0008:804A86BD   PUSH    FF
0008:804A86BF   PUSH    804017C0
0008:804A86C4   PUSH    ntoskrnl!_except_handler3
0008:804A86C9   MOV     EAX,FS:[00000000]
0008:804A86CF   PUSH    EAX
```

다행인지 불행인지 ntoskrnl에 있는 NtReadFile은 무슨 일은 하는 코드인지는 모르지만 실제 코드가 있는 것을 볼 수 있다.

우린 아무래도 INT 2E라는 놈의 정체를 밝혀야지만 될 것 같다.

그럼 INT 2E가 무엇인가?

보호모드에서는 (모른다면 컬럼 중 What's ring을 보면 알것이다.) Interrupt을 위하여 Interrupt Gate(Dos 시절의 Vector와 비슷한 기능)를 이용하며, Gate는 Interrupt에 대한 정보를 가지고 있는 Descriptor로 이루어져 있다. OS는 모든 인터럽트에 대한 Descriptor를 Table 형식으로 가지고 있다. 그것을 Interrupt Descriptor Table(IDT) 이라 한다.
Soft-ICE 에서 IDT를 보게 되면 INT 2E Handler의 시작 주소를 알 수 있다. (명령 : IDT)

002B	IntG32	0008:80463420	DPL=3	P	ntoskrnl!KiReleaseSpinLock+0FF0
002C	IntG32	0008:80463590	DPL=3	P	ntoskrnl!KiReleaseSpinLock+1160
002D	IntG32	0008:BB6D3CB4	DPL=3	P	DbgMsg!.text+09B4
002E	IntG32	0008:80462E50	DPL=3	P	ntoskrnl!KiReleaseSpinLock+0A20
002F	IntG32	0008:804664FC	DPL=0	P	ntoskrnl!Kei386EoiHelper+2BD8
0030	IntG32	0008:80462490	DPL=0	P	ntoskrnl!KiReleaseSpinLock+0060

생각 해보자.
User Mode에서 Kernel Mode로 제어권을 이양 하는 방법에는 무엇이 있을까?
아마 Fault, Interrupt, cpu 지원 명령인 SYSENTER 3가지 정도가 Mode 변환을 일을켜주는(ring transition) 명령일 것이다. 이 중에 Window2000에서는 Interrupt를 선택 한 것 같다. (XP에서는 SYSENTER를 이용한다.)
INT 2E 에서는 Ntdll!NtReadFile의 실제 수행 할 코드를 호출해주는 일을 할것이다.
실제 코드를 함 보자. 코드 중에서 중요 부분만을 거론 하겠다.

실제 INT 2E 의 코드

```
{
        PUSH all state

        0008:80462E70   MOV         ESI,[FFDFF124]
        ETHREAD curThread = gCurrentThread; // 0xFFDFF124 -
        0008:80462E76   PUSH        DWORD PTR [ESI+00000134]
        int previousMode = curTherad->previousMode; // offset 0x134 - ①

        0008:80462EB3   ADD         EDI,[ESI+000000DC]
        0008:80462EB9   MOV         EBX,EAX
        0008:80462EBB   AND         EAX,00000FFF
        0008:80462EC0   CMP         EAX,[EDI+08]
        SYSTEM_SERVICE_TABLE sst = curThread->serviceTable; // offset 0xDC -
        …
        if( eax/*함수 인덱스*/ > sst->serviceLimit )
                …․
        0008:80462EE8   INC         DWORD PTR [FFDFF5DC]
        0008:80462EEE   MOV         ESI,EDX
        0008:80462EF0   MOV         EBX,[EDI+0C]
```

```
        0008:80462EF3   XOR         ECX,ECX
        0008:80462EF5   MOV          CL,[EBX+EAX]
        0008:80462EF8   MOV          EDI,[EDI]
        0008:80462EFA   MOV          EBX,[EAX*4+EDI]
        0008:80462EFD   SUB         ESP,ECX
        0008:80462EFF   SHR         ECX,02
        0008:80462F02   MOV          EDI,ESP
        0008:80462F04   CMP         ESI,[ntoskrnl!MmUserProbeAddress]
        0008:80462F0A   JAE         804630F3
        0008:80462F10   REPZ MOVSD
        0008:80462F12   CALL        EBX
        If( previousMode != MmUserProbeAddress) - ②
                //Vaild check

        call sst->KiServiceTable[EAX] - ③
        …
        0008:8046302D   IRETD
        return;
}
```

① 현재 함수의 caller thread의 ETHREAD structure(TEB)의 주소를 가져와 몇가지의 정보를 가져온다. TEB 에는 현재 Thread에 대한 모든 정보들을 가지고 있다. (자세한 사항은 Inside2000 5장의 322 page에 structure 구조가 나와 있다.)
그 중 serviceTable을 얻어온다 이 곳에는 system service 함수 주소의 배열이 있다.

② previousMode Native API를 호출한 이전의 Mode를 말하며 OS 전역 변수인 MmUserProbeAddress와 비교하여 검사 한다. 만약 UserMode 라면 system service를 처리할 때 반듯이 parameter 들에 대해서 유효검사를 해야 한다. 그 이유는 arbitrary thread context 상에서 실행되어질 수도 있기 때문일 것이다. 또한 security를 위한 Access mode 권한을 검사 조건(이 조건은 실제 system service에서 사용되어질것이다.)을 만들 것이다.

④ 이곳에서 ServiceTable에서 EAX의 함수 Index를 이용하여 실제 처리 코드가 있는 주소를 찾아서, 그 루틴을 호출 한다.
한번 해당 주소의 코드를 Soft Ice를 이용하여 보자.

```
ntoskrnl!NtReadFile
0008:804A86BA   PUSH        EBP
0008:804A86BB   MOV         EBP,ESP
```

```
0008:804A86BD   PUSH      FF
0008:804A86BF   PUSH      804017C0
0008:804A86C4   PUSH      ntoskrnl!_except_handler3
0008:804A86C9   MOV       EAX,FS:[00000000]
0008:804A86CF   PUSH      EAX
```

오~! ntoskrnl 안에 있는 Native 함수를 호출해 준다.

결국 User Mode에서 Native API와 Kernel mode의 ZwXxxx 계열 모두 결국 NtXxxx을 호출 하게 된다.
그러나 두 가지의 경우는 행하여 지는 결과는 똑같겠지만(당연히 사용자 입장에서는 같은 함수 호출 이므로), 몇가지 차이가 있다.

User mode에서 Native AP호출

ZwXxxx와 NtXxxxx 모두 NtXxxx를 호출 한다.

Previous mode 가 user이며, valid check가 이루어진다.

Kernel mode에서 NtXxxx 경우

Kernel Mode 에서 NtXxxx 호출.

직접 system service 호출한다(실제 service code). Previous mode를 바꾸지 않는다.

그러므로 valid check 유무는 caller에 의해서 결정되어져 온다. 그리고 previous mode 로의 복구가 이루어지지 않는다.

Kernel mode에서 ZwXxxx 경우

Kernel Mode 에서 ZwXxx 호출.

INT 2E를 통하여 호출 하기 때문에 여러 가지 처리(INT 2E에서 하는 일) 코드가 실행되며 Parameters 를 EDX를 통하여 받고, EAX를 index로 이용하여 ServiceTable[EAX] 해당 system service를 호출한다.

Previous mode를 Kernel mode로 두고 처리한다. Kernel mode이기 때문에 여러 유효성 검사를 하지 않을 것이다.

여기에서 attack Native API의 looking around native API 1부를 마칠 것이다.
다음 Attack Native API 2부에서는 XP와의 차이점 과 지금 까지 우리가 찾아낸 내용들을 이용하여 Native API를 Hooking 하여 OS에서 접근하는 file들을 모니터링 해볼 예정이다.

이 컬럼을 읽고 Native API의 개념과 ZwXxxx 와 NtXxxx 의 차이 점을 이해했으면 하는 바람이며 조금이라도 system programmer로서의 내공이 업 되었으면 좋겠다.

기타 질문 사항은 이 메일을 이용해주기 바란다.

참고 서적 및 사이트

Inside Windows 2000, Microsoft Press

Undocumented Windows 2000 Secrets, Addison-wesley

Windows NT/2000 NATIVE API REFERENCE, MTP

www.osronline.com