Attack Native API 2부

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

1부 Look around Native API

2부 Hooking Native API

전회에서는 우리는 Native API가 무엇이며 Window 2000에서 Native API의 호출 경로를 알아 보았다.
이번 회에 서는 전회에서 말한 대로 XP와의 차이점을 알아보고, Native API의 경로를 탐색하면서 알아낸 정보를 이용하여 System service을 hooking 해보도록 하겠다.

XP와는 무슨 차이가 있는 것일까?

필자의 생각으로는 XP와 2000에서의 차이점의 거의 없을 것으로 보인다. 아마도 XP에서는 좀더 안정화된(?) 코드로 이루어져 있을 것이며, 동일하게 구현 되어 있을 것 같다. 이 생각이 맞는지를 확인해보도록 하자. 디버깅 툴은 Soft-Ice 위주로 사용 하였고, 때에 따라 windbg를 사용하였다.
XP에서도 2000에서와 같이 ntdll!NtReadFile 을 따라가면서 무엇이 차이점인가를 알아 보도록 하자.

User-mode에서의 이동 경로.

```
:u ntdll!ntreadfile
ntdll!NtReadFile
001B:77F5BFA8   MOV         EAX,000000B7
001B:77F5BFAD   MOV         EDX,7FFE0300
001B:77F5BFB2   CALL        EDX
001B:77F5BFB4   RET         0024

:u 7FFE0300
001B:7FFE0300   MOV         EDX,ESP
001B:7FFE0302   SYSENTER
001B:7FFE0304   RET
```

위 쪽 code는 xp에서 user-mode에서 kernel-mode 쪽으로 service를 받기 위하여 사용되어 지는 ntdll.dll의 함수 중에 하나를 보는 것이다.
EAX에 system service index(NtReadFile의 index)인 0xB7h를 넣어주고 0x7FFE0300을 호출해 준다. (0x7FFE0300 주소는 SystemCallstub 의 주소이다. Windbg의 ln 명령을 이용하여 확인 해볼 수 있을 것이다.)
SystemCallstub 함수에서는 kernel mode로의 진입을 위하여 XP에서는 SYSENTER 명령을 이용한다. (2000에서는 int 2E 사용)
SYSENTER 함수는 Intel cpu에서 지원하는 ring transition 명령이며, 실행 후 ring 0 system procedure 을 실행 시킨다. 해당 루틴에 대한 정보는 MSR(model specific registers)를 이용을 한다.

MSR	Address
SYSENTER_CS_MSR	174H
SYSENTER_ESP_MSR	175H
SYSENTER_EIP_MSR	176H

* SYSENTER와 관련된 자세한 내용은 intel cpu manual 을 참고하기 바란다.(Volume 2 : Instruction set reference, page 3-750)

SYSENTER 명령에 의해서 ring transition 과 함께 MSR의 176H(SYSENTER_EIP_MSR)의 주소의 루틴으로 분기가 될 것이다.

Soft-ice 명령인 msr 을 이용하여 아래와 같이 해당 주소들의 값들을 확인 할 수 가 있다.

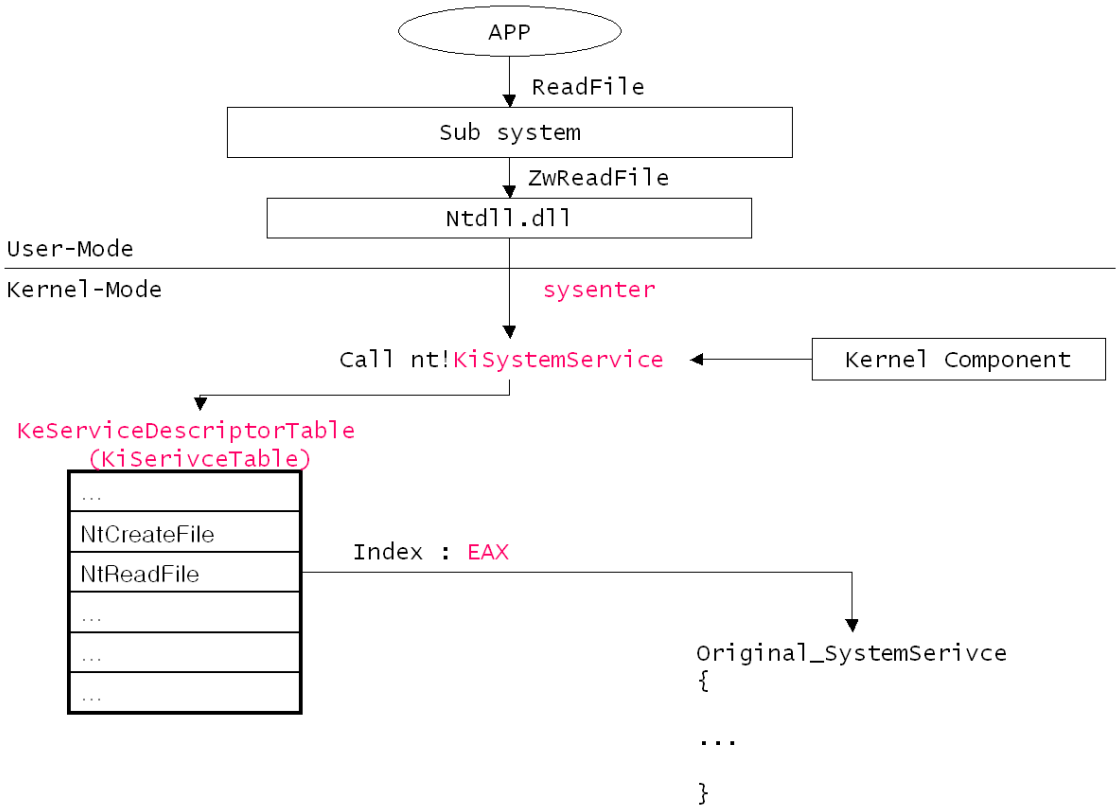
```
:msr
Reg  Value              Acc ID Name             Description
0    00000000:00000000   R  IA32_P5_MC_ADDR     Machine Check Exception Address
1    00000000:00000000   R  IA32_P5_MC_TYPE     Machine Check Exception Type
10   00000036:70D745D1  RW  IA32_TIME_STAMP_CTR Time Stamp Counter
17   000A0000:00000000   R  IA32_PLATFORM_ID    Platform ID
...
174  00000000:00000008  RW  IA32_SYSENTER_CS    CS register target for CPL 0 cod
175  00000000:00000000  RW  IA32_SYSENTER_ESP   Stack pointer for CPL 0 stack
176  00000000:8052F480  RW  IA32_SYSENTER_EIP   CPL 0 code entry point
179  00000000:000C0204   R  IA32_MCG_CAP        Machine Check Capabilities
17A  00000000:00000000   R  IA32_MCG_STATUS     Machine Check Status
...
```

MSR 의 176H 의 값의 이용하여 해당 코드를 볼 수가 있을 것이다.
(Windbg의 ln 명령을 이용하여 내부 함수 이름을 찾아보면 KiSystemService 의 이름을 찾을 수 있다.)

그러나 어떠한 이유에서인지 모르지만 해당 루틴에 break point를 잡으면 stack overflow exception이 발생을 해버린다. 그래서 필자 또한정확하게 disassembly을 해보지 못하였다. 해당 주소의 assemble code들 중에서 중요한 부분(우리에게 필요한)을 정리해보면 다음과 같다.

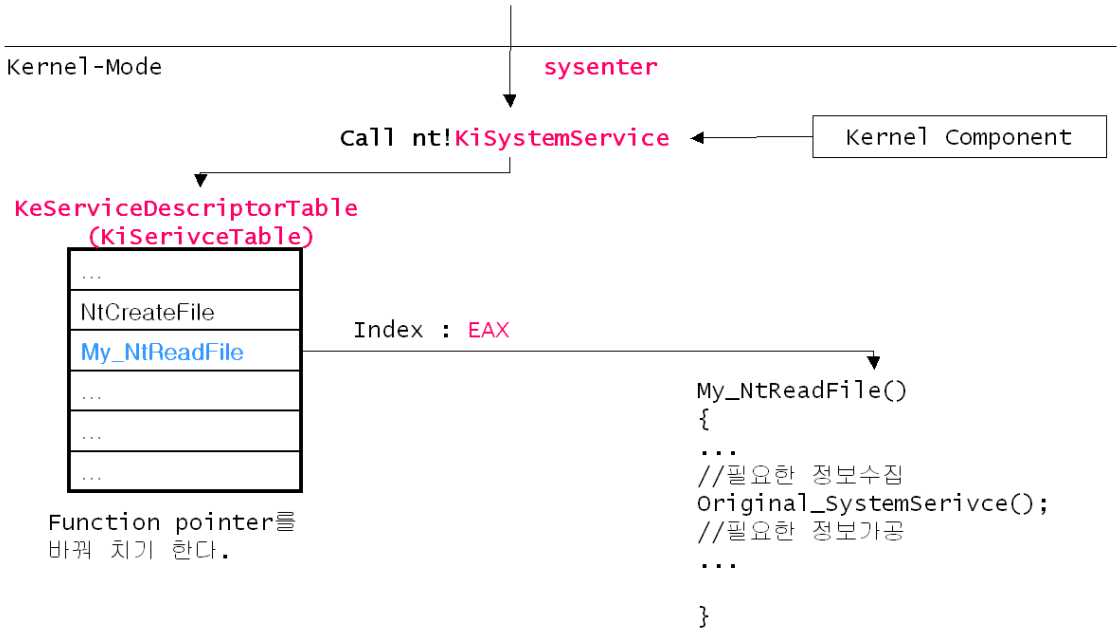
```
KiSytemService
{
        ...
        ETHREAD curThread = gCurrentThread; // 0xFFDFF124
        SYSTEM_SERVICE_TABLE sst = curThread->serviceTable;
        if( eax/*함수 인덱스*/ > sst->serviceLimit )
        {
            ...
            return;
        }
        call sst->KiServiceTable[EAX]
        ...
        return;
}
```

KiSystemService 함수는 Native Api 를 이용하여 요청된 일을 수행 할 수 있는 함수를 SystemServiceTable에서 EAX(Native Api 호출시 EAX로 넘온다)를 이용하여 해당 SystemService Routine을 호출 시켜주는 일을 해주는 것이다.



지금까지의 내용을 그림으로 정리해 보았다. 그림 처럼 User mode에서 요청이건 Kernel mode에서 요청이건 KiSystemService 함수를 호출하여 System Service 을 받게되고, KiSystemService 함수는 ServiceTable 을 이용하여 해당 Service Routine의 주소를 알아낸다.
결과적으로 XP와 2000 의 차이점의 거의 없다고 볼 수 있는 것 같다.(우리가 의도하는 Hooking 방법을 구현하기에는)

이젠 우리의 공략 포인트를 찾아야 할 것이다. Hooking 방법에는 여러 가지가 있을 것이다. 우리는 Service Table에 존재하는 function pointer를 변경 시켜 Hooking을 해볼 예정이다. Function pointer를 변경 시키게 되면 KiSystemService는 변경된 함수의 pointer가 실제 System Service의 pointer로 알고 호출을 해줄 것이다. 그렇게 되면 우리의 함수에서 stack 혹은 parameters에서 정보를 얻어 온다거나, 실제 함수를 호출 한 후 정보를 가공하는 등의 일을 할 수 있을 것이다.

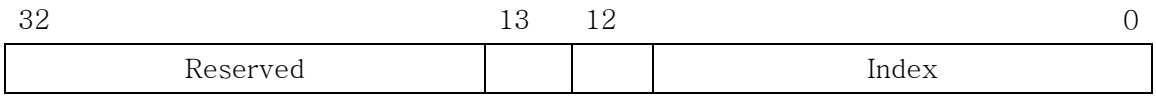


The Service Descriptor Table

모든 User-mode application들은 system service를 받기 위하여 Native API(ZwXxxx, Ntxxxx)를 호출 한다고했다. 그리고 Native API 에서는 int 2E 또는 SYSENTER 을 호출하게되고, 이 명령에 의해서 KiSystemService() 내부함수가 호출이 되어진다. 이 함수에서는 Service Descriptor Table(SDT)를 참조하여 Native API의 해당하는 System Service entry pointer을 알아온다.

KiSystemService를 호출 할 때에는 EAX에 Native API의 해당하는 service의 index가 넘어오며, 이 값을 이용하여 KiSystemService 함수는 ServiceTable의 entry를 찾는다.

EAX register의 값은 아래와 같이 구분되어 있다.

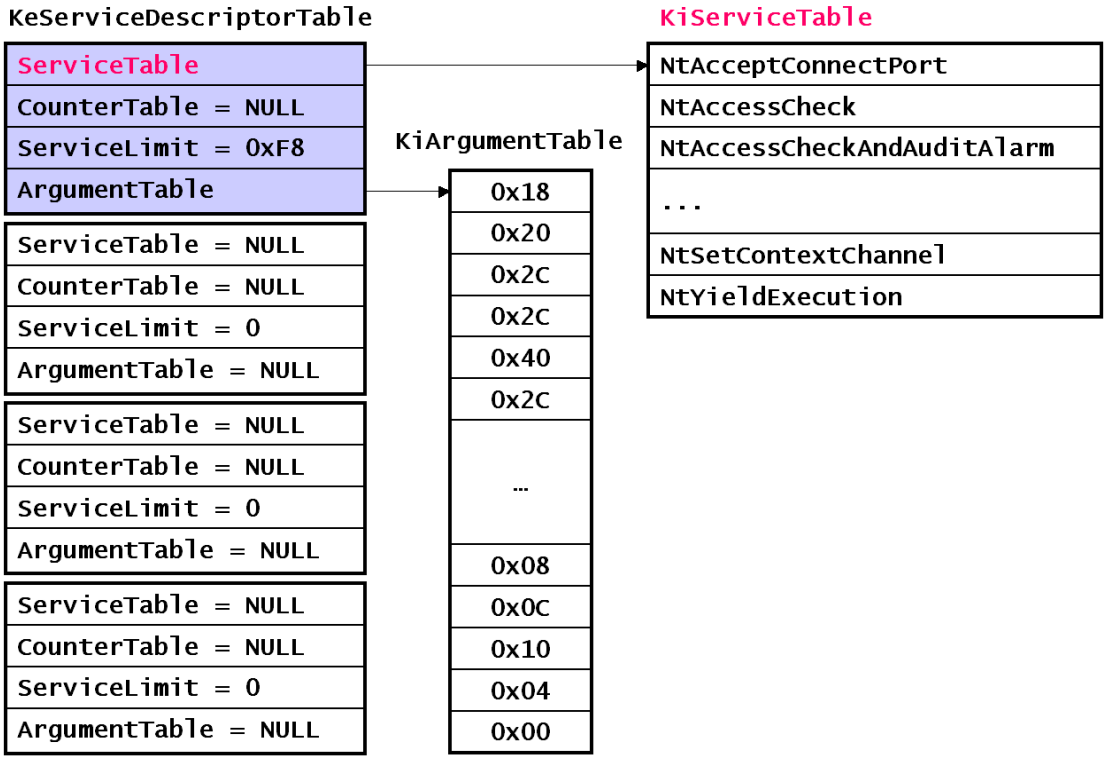


13bit와 12bit를 이용하여 두 bit의 값이 “00”일 때에는 KeServiceDescriptorTable 을 사용하며, “01”일 때에는 KeServiceDescriptorTableShawdow을 사용한다.

KeSystemDescriptorTable에는 일반적이 NativeAPI에 대한 정보를 가지고 있으며, KeServiceDescriptorTableShawdow에서는 win32k.sys(대부분 Graphic 관련)에 관련된 함수에 대한 정보를 가지고 있다. 자세한 내용은 Undocumented Windows 2000 Secrets 책을 참고하기 바란다.

ServiceTable은 ntoskrnl.exe의 KeServiceDescriptorTable이라는 export 멤버로 존재하여 이것을 이용하며 아래와 같은 구조로 선언되어져 있다.

```
Type struct _SYSTEM_SERVICE_TABLE
{
        PNTPROC         ServiceTable;
        PDWORD          ConterTable;
        DWORD           ServiceLimit;
        PBYTE           ArgumentTable;
}SYSTEM_SERVICE_TAB, *PSYSTEM_SERVICE_TAB;

extern PSYSTEM_SERVICE_TAB KeServiceDescriptorTable;
```



실전 구현

우리는 application 단에서 CreateFile의 호출을 감시 해 볼 생각이다.

CreateFile을 호출 하게되면(어떤 application이든간에) 우리가 알아본 것에 의하면 NtCreateFile이 최종적으로 호출이 되면서 KiSystemService 함수를 호출을 해주게 된다.

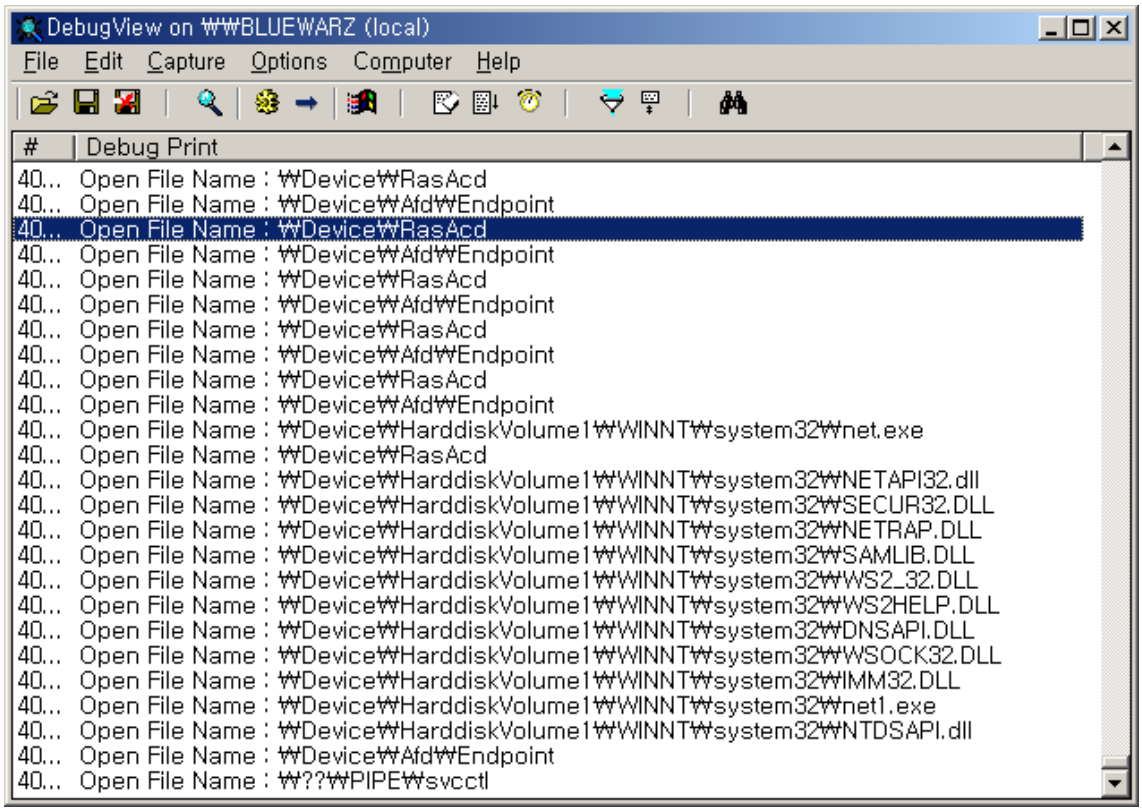
우리는 두가지를 알고 있어야 한다.
첫째는 NtCreateFile의 index number이다. 이 index 값을 알고 있어야지만 정확히 ServiceTable의 entry에 있는 주소 값을 변경 수 있다.
두번째는 실제 System Service를 대신 할 임의의 함수를 구현을 하기 위하여 NtCreateFile의 proto-type을 알고 있어야 한다.

첫번째 index를 알아내는 방법은 kernel debugger를 이용하여 쉽게 알아 낼 수 있다. 우리가 hooking 하려는 NtCreateFile에 break pointer을 잡고, EAX값을 보면 될 것이다.
NtCreateFile은 index 값은 0x20h 이다.
NtCreateFile의 proto-type은 MSDN에서 ZwCreateFile을 찾아서 똑같이 선언해주면 된다.

자 그럼 구현한 코드의 중요한 부분들을 살펴보기로하자.

```
//
// Definition for system call service table
//
 Type struct _SYSTEM_SERVICE_TABLE
{
        PNTPROC         ServiceTable;
        PDWORD          ConterTable;
        DWORD           ServiceLimit;
        PBYTE           ArgumentTable;
}SYSTEM_SERVICE_TAB, *PSYSTEM_SERVICE_TAB;

// Pointer to the image of the system service table
extern PSYSTEM_SERVICE_TAB KeServiceDescriptorTable;

// index number of NtCreatefile function in ServiceTable
#define       FN_INDEX        0x20

NTSTATUS
DriverEntry( IN PDRIVER_OBJECT pDriverObject, IN PUNICODE_STRING regPath )
{
// 실제 service routine의 주소를 저장해둔다.
g_OrginalPointer = ((PSRVTABLE) KeServiceDescriptorTable)->
                    ServiceTable[FN_INDEX];
// ServiceTable 에 우리의 함수의 주소를 넣는다.
```

```
((PSRVTABLE) KeServiceDescriptorTable)->ServiceTable[FN_INDEX] =
            MyTestRoutine;

}

NTSTATUS MyTestRoutine(   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,
   IN PVOID EaBuffer   OPTIONAL,
   IN ULONG EaLength)
{
    KdPrint (("Open File Name : %ws₩n", ObjectAttributes->ObjectName->Buffer));

//실제 함수를 호출해준다.
   return (*g_OrginalPointer)(   FileHandle,
     DesiredAccess,
     ObjectAttributes,
     IoStatusBlock,
     AllocationSize,
     FileAttributes,
     ShareAccess,
     CreateDisposition,
     CreateOptions,
     EaBuffer   OPTIONAL,
     EaLength);
}
```



위와 같이 구현하게 되면 윈도우 상에서 Create되는 파일들을 모니터링을 할 수 있을 것이며, 조금 수정을 하면 접근 불허 또는 정보 가공등에도 응용을 할 수 있을 것이다.
풀 버젼 소스(www.devguru.co.kr 에서 자료실의 소스 탭에 해당 소스가 있다)는 꼭 compile 일 해보고 디버깅을 해보기 바란다.

Build 방법

1. 해당 소스 폴더로 이동 후 build 명령 사용.
2. regini.exe를 이용하여 dglogspy.ini 를 registry에 등록 한다.
   (또는 DynamicLoader를 이용하면 된다.)
3. 재부팅 후. Net 명령을 이용하여 Driver를 loading 한다.
4. debugger를 이용하여 확인한다. (dbgview.exe를 이용하여 확인)

이것으로 Attack Native API의 컬럼을 마치겠다.
많은 정보를 전달하려고 노력하였는데, 생각대로 많은 것을 쓰지 못해서 아쉬운 점도 있으며, 필자가 전문 칼럼니스트가 아니 여서 깔끔한 의사 전달이 잘 안된 것 도 이해 바란다.
다음 주제는 좀더 재미 있고 흥미로운 것을 선정해 보도록 하겠다.

기타 질문 사항은 이 메일 또는 회사 홈페이지의 Q&A를 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