J. Edu. & Sci., Vol. (23), No. (3) 2010

Monitoring Windows Kernel's Services

Rawaa Putros Polos
Department of Computers Sciences
College of Computers and Mathematics Sciences
University of Mosul

Received 04 / 11 / 2008 **Accepted** 06 / 04 / 2009

الخلاصة

تجهز نواة نظام التشغيل ويندوز التطبيقات العالية المستوى بوظائف ذات مستوى منخفض لانجاز مهام النظام المطلوبة منها. و يشار لهذه الوظائف بخدمات النظام. فالتحكم بهذه الخـدمات يمنح إمكانية المراقبة والتحكم بفعاليات مهمة لنظام التشغيل.

يقدم هذا البحث تقنية خطف النواة كونها الأكفأ والأكثر استخداما لانجاز مراقبة خـدمات النظام.

هدف البحث هو كيف بالإمكان التحكم ومراقبة نظام التشغيل برمجيا على مستوى النظام باستخدام تقنية خطف النواة.

أنجزت هذه التقنية ضمن سواقة جهاز عن طريق الوصول إلى جدول مـوزع خـدمات النظام (SSDT) للحصول على إمكانية التعامل و التغيير في عدد من خـدمات النـواة الفعالـة لغرض السيطرة ومراقبة تنفيذ البرامج وعمليات الحذف وإيقاف المعالجات في النظام.

تم تنفيذ هذا العمل بنجاح في ويندوز XP SP2 وطور باستخدام DDK لتنفيـذ سـواقة الجهاز ولغة فيجوال سي ++ الإصدار ٦,٠ لتنفيذ التطبيق.

عند تنفيذ التطبيق يتم التحكم بجميع عمليات تنفيذ البرامج وعمليات الحذف و إنهاء المعالجات ويمنح المستخدم قابلية السماح بإجراء العمليات المذكورة أو إلغاءها.

ABSTRACT

The kernel of Windows operating system provides high-level applications with the low-level functionality needed to perform system operations. This functionality referred to as system services. So, Controlling

these services gives the ability to monitor and control important activities of the operating system.

This research presents kernel hooking technique that is one of the most efficient and used technique to achieve system services monitoring.

The aim of the research is how the operating system can be programmatically monitored and controlled on a system-wide basis by means of kernel hooking.

This technique was implemented in a device driver by accessing SSDT (System Service Descriptor Table) to gain the ability for manipulating and change number of effective kernel services for monitoring programs execution, deletion operations and processes termination in the system.

The work has been run successfully on Windows XP SP2 and developed using DDK (Driver Development Kit) for device driver implementation and Visual C++ version 6.0 for application implementation.

So, when the application is executed, programs execution, deletion, and processes termination operations have been controlled, and gives user the capability to permit performing these operations or canceling them.

1. Introduction

Operating system components (such as system services and device drivers) runs in kernel mode, which refers to a mode of execution in a processor that grants access to all system memory and all CPU (Central Processing Unit) instructions [6].

In computing, the kernel is the lowest, most central component of a computer operating system, and one of the first pieces of code to load when a system starts. Its responsibilities include managing the system's resources and the communication between hardware and software components [9].

As a basic component of a computer operating system, the kernel provides an abstraction layer for the resources (especially memory, processors and I/O devices) that applications must utilize to perform their functions. It typically makes these facilities available to application processes through inter-process communication mechanisms and system services call. It is responsible for basic operating system housekeeping tasks such as memory management, process creation and termination, and managing the data on the disk. The integrity of the kernel is instrumental to the performance and security of the computer it resides upon.

Since all other programs and many portions of the operating system itself depend on the kernel, any problems in the kernel can make those programs crash or behave in unexpected ways. The "Blue Screen of Death"

(BSoD) in Windows is the result of an error in the kernel or a kernel mode driver that is so severe that the system can't recover [1].

System services are the critical functions of the OS (Operating System) and intercepting them will enable the programmer to understand and modify the action of the OS at a deeper level than user mode techniques [8].

Kernel services monitoring using SSDT hooking technique was presented by Russinovich and Cogswell in 1997 as completely different approach to system-wide hooks for Windows NT, and SVEN B. SCHREIBER was presented it for Windows 2000 [10]. Also Chris Ries presented a brief description of this technique [1].

In this work hooking SSDT was implemented to monitor Windows programs execution, deletion operation and process termination. Hooking operation requires accessing kernel memory, so that, a device driver had been developed to achieve this task, which contains the main parts, i.e. accessing SSDT and replace the pointer of the desired service by the address of our own function.

The driver required user application to load and start it, therefore special communication between user-mode application and the kernel device driver was made to exchange important data using IRP (I/O Request Packets).

2. Introducing Code into the Kernel:

As a general rule, processes cannot access kernel's memory. Therefore, the straightforward way to introduce code into the kernel is by using a loadable module (sometimes called a device driver or kernel driver). [7][4].

As its name suggests, a device driver is typically for devices. However, any code can be introduced via a driver. Once the code running in the kernel, it can be had full access to all of the privileged memory of the kernel and system processes. With kernel-level access, modifying the code and data structures of any software on the computer can be made [2][8].

To build Windows device driver, we'll need the Driver Development Kit (DDK), which provides special header files, import libraries and different build environments for building drivers. DDKs are available from Microsoft for each version of Windows, Microsoft allows downloading the DDK from their site. After building the device driver, it must be loaded by a user-mode program [10].

A loading program typically will decompress a copy of the .sys file to the hard drive. .sys file is a single file represents the heart of device driver

and is very similar in concept to a DLL (Dynamic Link Library). After the decompression operation, commands are issued to load it into the kernel [3].

After loaded the driver into kernel memory. All the power of the OS is now under the dispose.

3. Bridging user and Kernel Mode:

A user-mode program can communicate with a kernel-level driver through a variety of means. One of the most common is the use of I/O request packets (IRP) [11].

3.1 What is an IRP?

Almost all I/O under Windows is packet-driven. Each separate I/O transaction is described by a work order that tells the driver what to do and tracks the progress of the request through the I/O subsystem. These work orders take the form of a data structure called an I/O Request Packet (IRP). The IRP is a variable sized structure includes information about the operation that is being requested [3].

In order to communicate with a user-mode program, a Windows device driver typically needs to handle IRPs. These are just data structures which contain buffers of data. A user-mode program can open a file handle and write to it. In the kernel, this write operation is represented as an IRP. So, if a user-mode program writes the string "HELLO DRIVER!" to the file handle, the kernel will create an IRP that contains the buffer and string "HELLO DRIVER!", therefore Communication can take place between the user and kernel modes via these IRPs [11].

In order to process IRPs, the kernel driver must include functions to handle the IRP.

4. Kernel System Services:

Windows provides a largely undocumented set of base system services, called the Native API which is somewhat similar to the interrupt based system. These kernel-mode base system services are used by the operating environment subsystems for the implementation of their operating environments, on top of the Windows NT micro-kernel [2][5][10].

Under Windows NT, the NT executive (part of NTOSKRNL.EXE) provides core system services. These services are rather generic and primitive. Various APIs such as Win32, OS/2, and POSIX are provided in the form of DLLs. These APIs, in turn, call services provided by the NT executive. The name of the API function to call differs for users calling from different subsystems even though the same system service is invoked. For

example, to open a file from the Win32 API, applications call CreateFile() and to open a file from the POSIX API, applications call the open() function. Both of these applications ultimately call the NtCreateFile() system service from the NT executive [6][7].

Every system service has a unique index number, which is generated automatically by a script that runs as part of the NT build process.

All of the Native APIs begin with "Nt". The export table in NTDLL.DLL also makes the Native API accessible through an alternate naming convention, one where command names begin with "Zw" instead of "Nt". Thus, **ZwCreateFile**() is an alias for **NtCreateFile**() [4].

5. The System Table

Windows kernel relies on a table of pointers to functions in order to perform system operations. This table, referred to by Microsoft as the system service descriptor table (SSDT). So, SSDT is an array of function pointers to an in-memory system services, which is implemented in the operation system [9].

This table can be indexed by system call number to locate the address of the function in memory. There are two ways a program can make a system call: by using interrupt 0x2E, or by using the SYSENTER instruction.

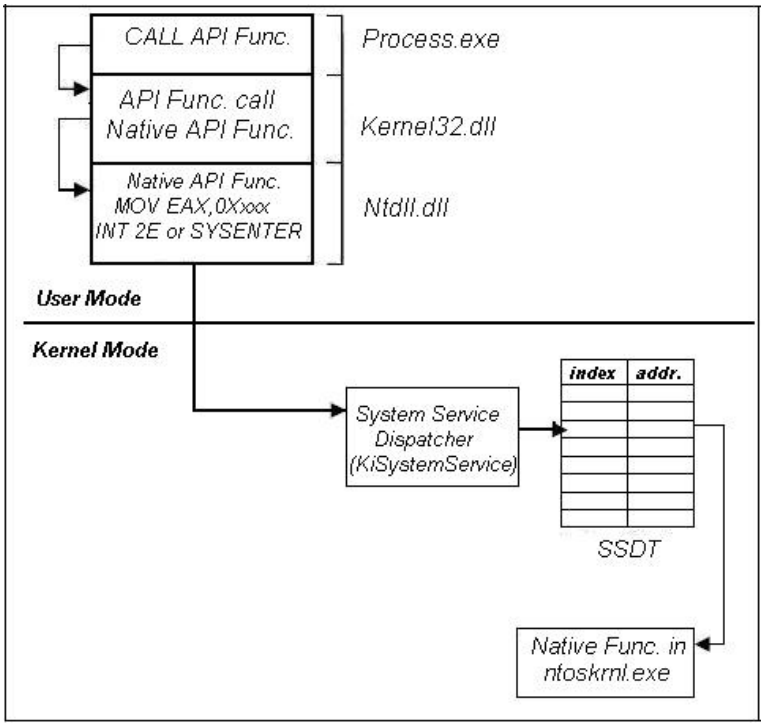

Figure (1): SSDT Table and KiSystemService [6].

On Windows XP and beyond, programs typically use the SYSENTER instruction, while older platforms use interrupt 0x2E. The two mechanisms are completely different, although they achieve the same result.

Making a system call results in the function KiSystemService being called in the kernel. This function reads the system-call number from the EAX register, and looks up the call in the SSDT as shown in Figure (1).The KeServiceDescriptorTable is exported by the kernel [1].

The table contains a pointer to the portion of the SSDT that contains the core system services implemented in Ntoskrnl.exe, which is a major piece of the kernel.

To call a specific function, the system service dispatcher, KiSystemService, simply takes the ID number of the desired function and multiplies it by 4 to get the offset into the SSDT [9].

6. Modifying Kernel Memory Protections

Before hooking kernel functions, some consideration must be taken in account. Modern Windows operating systems are capable of protecting kernel memory by making the system call table read-only. If an attempt is made to write to a read-only portion of memory, such as the SSDT, a Blue Screen of Death (BSoD) will occur.

The key to circumventing protected memory lies with the Memory Descriptor List, defined within ntddk.h of the Microsoft Windows Driver Development Kit. A MDL (Memory Descriptor List) is a system-defined structure that describes a buffer by a set of physical addresses. A driver that performs direct I/O receives a pointer to an MDL from the I/O manager, and reads and writes data through the MDL [9].

Number of related functions can be used to describe a region of memory in a MDL. MDLs contain the start address, owning process, number of bytes, and flags for the memory region [3].

7. SSDT Hooking Implementation

The efficient way to put a hook into the system services is to locate the SSDT used by the operating system and change the function pointers to point to other function provided in this work. This can be achieved only from a kernel device driver because this table is protected by the OS as mentioned earlier.

Once the hook program is loaded as a device driver, it can change the SSDT to point to my new function instead of into Ntoskrnl.exe or Win32k.sys. When a non-kernel application calls into the kernel, the request is processed by the system service dispatcher, and the hook function is called as depicts in Figure (2).

At this point, the hook function can call the original system service and modify the returned data or it can just return bogus data without calling the original code.

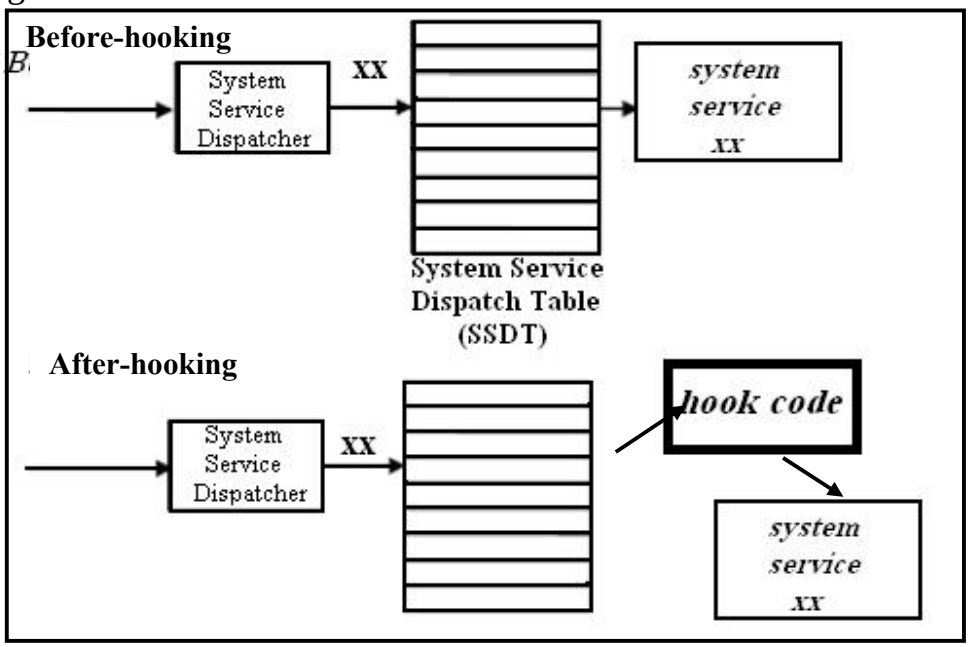

Figure (2) SSDT Hooking.

The DDK provides two different build environments: checked & free environment. Checked environment was used for developing the device driver, while the other is used for releasing code.

The source code for the driver was written in C which include the essential parts of the driver: such as driver entry function and driver dispatch function.

Driver parts contain SSDT hooking steps which can be described in the following steps:

1. Defining and implementing a new function:

   This function has the same definition as the system service to be hooked. It contains actions differ from the original one, the new actions gives the ability either to call the original service or perform another task.

2. Using MDL to change SSDT memory protection

   As mentioned previously, OS makes the SSDT read-only. In this work MDL was used to map virtual memory to physical pages, so the memory region of SSDT was described in MDL using number of related functions as follows:

- **IoAllocateMdl** function was used to allocate MDL for SSDT which accept the SSDT and its size as first and second parameters respectively.

IoAllocateMdl (IN PVOID VirtualAddress, IN ULONG Length, IN BOOLEAN SecondaryBuffer, IN BOOLEANChargeQuota, N OUT PIRP Irp OPTIONAL);

- **MmBuildMdlForNonPagedPool** function that receives the MDL allocated for the SSDT from the previous function, and updates it to describe the underlying physical page.

MmBuildMdlForNonPagedPool(IN OUT PMDL MemoryDescriptorList);

- **MmMapLockedPages** function is used to lock the MDL page.

MmMapLockedPages(IN PMDL MemoryDescriptorList, IN KPROCESS OR_MODE AccessMode);

3. Obtaining the address of the system service to be hooked in the SSDT.

To access SSDT, the driver must deal with KeServiceDescriptorTable.

The structure of this entry was defined in the driver as follows:

```
typedef struct ServiceDescriptorEntry {
        unsigned int ServiceTableBase;
        unsigned int *ServiceCounterTableBase;
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
        } SSDT_Entry;
```

Driver entry function begins with saving the address of the original system service using SYSTEMSERVICE macro:

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

It accepts the address of a function exported by ntoskrnl.exe, a Zw* function as parameters, and returns the address of the corresponding Nt* function in the SSDT.

4. Replacing SSDT entry for that service with the address of the new function.

Driver dispatch function contains the replacement operation. This operation was accomplished using the following macro:

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

Macro parameters are:

- Address for the original system service.
- Address for the new function.
- Variable to hold the address of the original service.

This macro used InterLockedExchange function that made the replacement operation and return original address:

```
LONG   InterlockedExchange(
            LPLONG Target,
            LONG Value );
```

SYSCALL_INDEX macro was used to obtain service's index:

```
#define SYSCALL_INDEX(_Function)

*(PULONG)((PUCHAR)_Function+1)
```

Which takes the address of a Zw* function and returns its corresponding index number in the SSDT.
MappedSystemCallTable is the MDL that describes SSDT table.

To terminate hooking operation **UNHOOK_SYSCALL** macro is used:

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

8. Monitoring Software Parts

Kernel monitoring software consists of two parts:

1. Kernel-mode part.
2. User-mode part.

1. Kernel-mode part:

This part includes the device driver that contains the implementation of hooking tasks as described in the previous section.

The driver performs kernel hooking by replacing the address of the selected system service in the SSDT with the address of new supported function.

2. User-mode part:

This part includes Windows application responsible for driver loading and unloading as well as exchanging data between the application and the driver.

The application program used SCM (Service Control Manager) to load the driver, through the steps below:

1. Using the ***OpenSCManager*** function to get a handle to the specified service control manager database.

```
SC_HANDLE OpenSCManager (LPCTSTR lpMachineName,
                         LPCTSTR lpDatabaseName,
                         DWORD dwDesiredAccess  )
```

2. Using the ***CreateService*** function to create a service object and add it to the specified service control manager database.

```
SC_HANDLE CreateService  (SC_HANDLE hSCManager,
                          LPCTSTR lpServiceName,
                          LPCTSTR lpDisplayName,
                          DWORD dwDesiredAccess,
                          DWORD dwServiceType,
                          DWORD dwStartType,
                          DWORD dwErrorControl,
                          LPCTSTR lpBinaryPathName,
                          LPCTSTR lpLoadOrderGroup,
                          LPDWORD lpdwTagId,
                          LPCTSTR lpDependencies,
                          LPCTSTR lpServiceStartName,
                          LPCTSTR lpPassword    )
```

3. Using ***StartService*** function to start a service.

```
BOOL StartService( SC_HANDLE hService,
                   DWORD dwNumServiceArgs,
                   LPCTSTR* lpServiceArgVectors  )
```

Now the driver becomes a service, and the hooking process was performed. So any call to any of the hooked system services by other system's parts will cause to call the new functions provided by the developer.

To unload the driver, the application program uses ***DeleteService*** function.

BOOL DeleteService (SC_HANDLE hService)

At the start of monitoring operation, user application passes the address of the exchange buffer to the driver using the function:

```
BOOL   DeviceIoControl(
          HANDLE hDevice,
          DWORD dwIoControlCode,
          LPVOID lpInBuffer,
          DWORD nInBufferSize,
          LPVOID lpOutBuffer,
          DWORD nOutBufferSize,
          LPDWORD lpBytesReturned,
          LPOVERLAPPED lpOverlapped
          );
```

The driver receives the buffer as an IRP using the function:

IoGetCurrentIrpStackLocation(Irp)

The buffer contains data that represents user decisions either to call the original service or to do anther action as discarding the call.

9. Kernel Monitoring Software

Kernel SSDT hooking method had been implemented on a number of system services:

- **ZwCreateSection** to monitor programs execution**.** It is used for process creation (program execution) with **ObjectAttributes** is setting to SEC_IMAGE and Protect field is set to PAGE_EXECUTE.

```
NTSYSAPI  NTSTATUS  NTAPI  ZwCreateSection(
                           OUT PHANDLE SectionHandle,
                           IN ACCESS_MASK DesiredAccess,
                           IN POBJECT_ATTRIBUTES,
```

IN PLARGE_INTEGER SectionSize OPTIONAL,
IN ULONG Protect,
IN ULONG Attributes,
IN HANDLE FileHandle);

- **ZwSetInformationFile** for deletion operations monitoring. A file typically deleted using the ZwSetInformationFile function with the FileInformationClass parameter set to FileDispositionInformation and the DeleteFile member of the FileDispositionInformation object set to TRUE.

NTSYSAPI NTSTATUS NTAPI ZwSetInformationFile(
IN HANDLE FileHandle,
OUT PIO_STATUS_BLOCK IoStatusBlock,
IN PVOID FileInformation,
IN ULONG FileInformationLength,
IN FILE_INFORMATION_CLASS
FileInformationClass);

- **ZwTerminateProcess** to monitor the termination of any process**.** It terminates a process and the threads that it contains when the process handle is grant PROCESS-TERMINATE access, i.e. its value is zero.

NTSYSAPI NTSTATUS NTAPI ZwTerminateProcess(
IN HANDLE ProcessHandle OPTIONAL,
IN NTSTATUS ExitStatus);

Software interface is shown in Figure (3), which gives the ability to choose which service to hook or select to hook all.

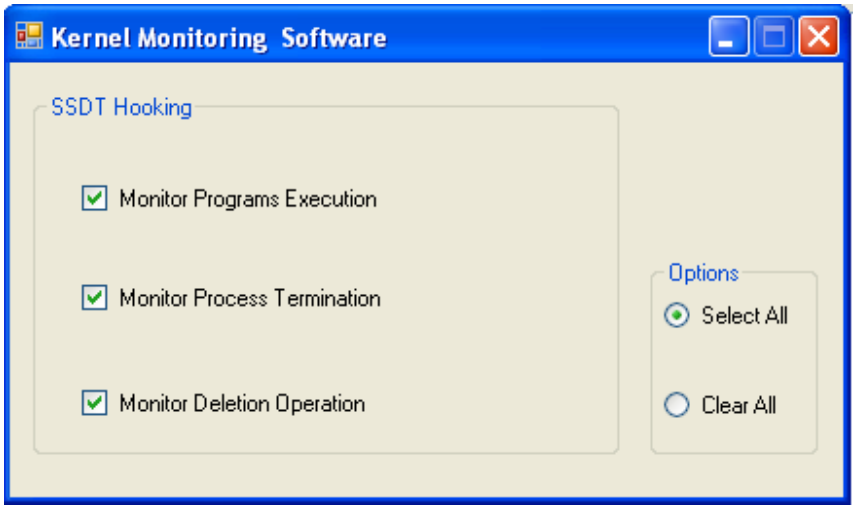

Figure (3): Software interface.

When one of the kernel hooking choices was selected the Hook&UnHook dialogue box will appear as depicted in Figure (4).

Figure (4): Hooking dialogue box.

To start service hooking, Start Hook button must be pressed from the dialogue box, at that time; the driver will be loaded and waiting for any call to the hooked system service. As the service is called, the driver sends data related to the action of that service to the application and waits a response from the user which will be sent by the application to it. Depending on user's response, the driver will either continue service call or cancel the call. To unhook, the Unhook button must be pressed.

10. Conclusion

The kernel is considered the heart of the Operating system. Therefore, in general, hooking provides a powerful way for monitoring and modifying the various actions of the OS. That is because hooking will divert the normal flow of system's control.

Despite that, kernel hooking is difficult to implement as an error in the kernel often end up in BSoD and required a device driver to access kernel memory and components.

System services are critical functions of the OS & hooking them will enable the programmer to understand and modify the action of the OS at a deeper level, but hooking implementation are sophisticated, since these services are undocumented.

Kernel hook get one central place from which it can be monitored the events or actions occurring as a result of a user-mode call or a kernel-mode call.

Reference

[1] Chris Ries, 2006, "Inside Windows Rootkits", VigilantMinds Inc.

[2] David A. Solomon and Mark E. Russinovich, 2000, "Inside Microsoft Windows 2000", Microsoft Press, 2nd Edition.

[3] DDK Documentation, 2003, Microsoft, www.microsoft.com/whdc/devtools/ddk/default-mspx.

[4] Eldad Eilam, 2005, "Reversing: Secrets of reverse Engineering", Widely publishing inc.

[5] Johnson M. Hart, 2004, "Windows System Programming", Addison Wesley Professional, 3rd edition.

[6] Mark E. Russinovich, David A. Solomon, 2004, "Microsoft Windows Internals", Microsoft Press.

[7] Dabak P., M. Borate & S. Phadke, 1999, "Undocumented Windows NT", M&T Books.

[8] Rajagopalan, M. Hittunem, M. A. Jim, T. Schlichting, 2006 "System call monitoring using Authenticated system calls", IEEE, vol. 3, Issue 3.

[9] Ric Vieler, 2007, "Professional Rootkits", Wrox Press.

[10] Sven B. Schreiber, 2001, "Undocumented Windows 2000 Secrets", Addison-wesley.

[11] Walter Oney, 2003, "Programming The Microsoft Windows Driver Model", Microsoft Press.