J Comput Virol (2010) 6:123–141
DOI 10.1007/s11416-008-0091-3

EICAR 2008 EXTENDED VERSION

User-mode memory scanning on 32-bit & 64-bit windows

Eric Uday Kumar

Received: 20 January 2008 / Revised: 7 June 2008 / Accepted: 15 June 2008 / Published online: 10 July 2008

Abstract Memory scanning is an essential component in detecting and deactivating malware while the malware is still active in memory. The content here is confined to user-mode memory scanning for malware on 32-bit and 64-bit Windows NT based systems that are memory resident and/or persistent over reboots. Malware targeting 32-bit Windows are being created and deployed at an alarming rate today. While there are not many malware targeting 64-bit Windows yet, many of the existing Win32 malware for 32-bit Windows will work fine on 64-bit Windows due to the underlying WoW64 subsystem. Here, we will present an approach to implement user-mode memory scanning for Windows. This essentially means scanning the virtual address space of all processes in memory. In case of an infection, while the malware is still active in memory, it can significantly limit detection and disinfection. The real challenge hence actually lies in fully disinfecting the machine and restoring back to its clean state. Today's malware apply complex anti-disinfection techniques making the task of restoring the machine to a clean state extremely difficult. Here, we will discuss some of these techniques with examples from real-world malware scenarios. Practical approaches for user-mode disinfection will be presented. By leveraging the abundance of redundant information available via various Win32 and Native API from user-mode, certain techniques to detect hidden processes will also be presented. Certain challenges in porting the memory scanner to 64-bit Windows and Vista will be discussed. The advantages and disadvantages of implementing a memory scanner in user-mode (rather than kernel-mode) will also be discussed.

E. U. Kumar (✉)
Authentium Inc., 7121 Fairway Drive, Suite 102,
Palm Beach Gardens, FL 33418, USA
e-mail: ekumar@authentium.com

1 Introduction

Computer malware targeting Microsoft's Windows operating system has been constantly evolving in order to remain stealthier, while still being effective in its attack. More and more complex malware are rapidly being generated and deployed each day [14]. Powered with automated malware generation tools and customized server side encryption/packing, the widely spread malware authors have plagued computer users [25]. The biggest challenge that the anti-malware industry has to face today is the sheer quantity of malware being generated on a daily basis [2]. The shift in intent of malware authors toward monitory gain has furthered the creation of stealthier and more subtle malware. This has resulted in malware that apply complex techniques to disallow detection and more so, disinfection.

Today's Windows based malware apply complex methods of anti-disinfection such as:

- Protecting its associated files on disk by disallowing access to any external program, such as an *on-demand* or *on-access* scanner.
- Protecting itself and its associated processes in memory from being terminated by using multi thread/process monitoring.
- Running as a SYSTEM process or native service to thwart termination.
- Monitoring its registry entries to thwart deletion.
- Injecting code (such as a dynamic link library) within system processes such as *winlogon.exe*, *explorer.exe*, *services.exe*, *lsass.exe*, etc.
- Patching system files.
- Hiding its associated processes in memory and/or files on disk by patching user-mode APIs, native APIs, or kernel data structures.

- Applying offensive techniques such as lowering system security and making it vulnerable for further attacks as well as terminating security applications.

Hence, while the malware and/or its components are still active in memory, it makes the task of disinfecting and restoring the machine to a clean state significantly harder. It is imperative that an anti-malware system for the Windows OS has a good implementation of both user-mode and kernel-mode memory scanning. A user-mode memory scanner purely operates in user-mode and can only access the user-space virtual memory with the privileges of the currently logged-on user. A kernel-mode memory scanner operates in kernel-mode and can access complete user-space and kernel-space virtual memory with the highest privileges. The discussion here is confined to user-mode memory scanning.

Implementing a user-mode memory scanner for Windows NT based systems involves the usage of several user-mode Win32 APIs and native APIs. These APIs allow enumeration of loaded modules and device drivers, as well as actively running processes and threads. Using these APIs, the user-mode memory scanner would take advantage of as much redundant information that is made available by the operating system and accessible from user-mode. This involves retrieving information such as enumeration of all active processes, process heaps, threads, device drivers, and loaded modules (such as DLLs). The idea behind obtaining redundant information using several methods (essentially from several different data structures maintained by the operating system), is to be able to "see" these memory components, least they may have been hidden by a malware using any of the several bypassing techniques. For example, the malware may have hooked a few of the user-mode APIs or native APIs that are used for enumeration, but may have overlooked bypassing some of the other APIs also used for enumeration. In this case, we will have a good chance of discovering the hidden malware components.

The following sections discuss related work and background information that is useful to understand memory scanning on Windows. This is followed by a discussion of enumeration techniques, disinfection techniques and an approach to combine these techniques in order to obtain useful data for memory scanning. The paper is concluded with a brief discussion about the pros and cons of implementing a memory scanner in user-mode.

2 Related work

A reliable published work related to memory scanning on 32-bit Windows NT based systems is by Ször [27]. The paper explains implementation of both user-mode and kernel-mode memory scanner, weighing in on the advantages of implementing memory scanning in kernel-mode. Several issues with real world malware detection and disinfection were also presented.

3 Background: Windows NT based operating systems

Microsoft's first 32-bit operating system, Windows NT 3.1, comprised of micro-kernel architecture, memory protection, pre-emptive multitasking scheduler, backward compatibility with 16-bit versions of Windows and Win32 API, and Windows NT File System (NTFS). With the release of Windows NT 4.0 in 1996, several major improvements were introduced in terms of efficiency, speed, reliability, scalability and security. Examples of today's Windows NT based operating systems are Windows 2000, Windows XP, Windows Server 2003, and Windows Vista, to name a few. These are all based on the same core as the Windows NT 4.0, but with newer enhancements that exploit advanced features of modern processor architectures. The Windows NT kernel is not a pure microkernel but rather a hybrid kernel that combines aspects of both microkernel and monolithic kernel architectures. This allows for most of the core kernel code to share the same memory address space. Although this improves efficiency, a pit-fall to this is that other kernel components (such as third party device drivers) could potentially compromise the integrity of the kernel. All discussions in this paper pertain to Windows operating systems that are based on the core Windows NT kernel.

3.1 Processes and threads

A process can be described to consist of the following essential components [26, pp. 4–5]:

- A process ID, which uniquely identifies the process.
- An access token, which uniquely identifies the owner, security groups, and privileges associated with the process.
- A private virtual address space, reserved by the operating system.
- Executable program (code and data) mapped into the process' virtual address space.
- At least one thread of execution.
- A list of open handles to resources allocated by the operating system that can be accessed by any thread in the process.
- Information about resources the system has allocated for it, such as files, shared memory sections, and synchronization objects.

A thread can be described to consist of the following essential components [26,pp. 4–5]:

- A thread ID, which uniquely identifies the thread.
- An access token, which uniquely identifies the owner, security groups, and privileges associated with the thread.
- A thread-local storage (TLS), which is a private storage area that can be used by subsystems, run-time libraries, and DLLs.
- Two separate stacks to use while the thread is running in user-mode and kernel-mode.
- The contents of CPU registers that represent the state of the CPU.

The *context* of a thread is defined by the contents of the CPU registers, the stacks, and the TLS. These hold all the information that is required to continue running the thread after a context switch. Every thread running inside a process has their own context but they share the process' virtual address space and resources. Hence, any thread in a process can access the memory and handles of any other thread running inside the same process. However, threads are not allowed to access the virtual address space of any other process, unless the other process specifically makes available some of its virtual address space as a file-mapping object.

3.2 Separation of kernel-mode and user-mode

The Windows NT based architecture clearly separates the user-mode code (ring 3) from the underlying kernel-mode code (ring 0). These two modes are part of the processor's hardware state. On x86 processors, this "memory access mode" is known as the IO privilege level (IOPL). Hence kernel-mode is IOPL 0 (ring 0) and user-mode is IOPL 3 (ring 3). This is to keep any buggy or malicious user-mode applications from crashing or compromising the kernel. User-mode applications are less privileged and access the system's resources like registry, file system, memory etc. via the Win32 API. Kernel-mode is the mode of execution in the processor that grants access to entire system memory and all the processor's instructions. The Windows NT architecture provides extensibility of its kernel functionality by allowing device drivers to load in the kernel. Windows will tag memory pages specifying which mode is required to access the memory, but Windows does not protect memory in kernel-mode from other threads running in kernel-mode. Windows only supports these two modes of execution today, although Intel and AMD CPUs actually support four privilege modes (or rings) in their chips to protect system code and data from being overwritten or corrupted by code of lesser privilege.

The Windows subsystem includes the Win32 subsystem service process (*csrss.exe*), the subsystem API library (e.g. *kernel32.dll*, *advapi32.dll*, *gdi32.dll*, and *ntdll.dll*), fixed processes (*winlogon.exe* and *smss.exe*), the RPC subsystem (*rpcss.exe*), the local security authority subsystem (*lsass.exe*), and service processes that run independent of user logons (example: task scheduler and spooler service). Note that *smss.exe* is the only "parentless process" as it is spawned by the INIT routine in *ntoskrnl.exe*. Windows implements the Win32 subsystem as Dynamic Link Libraries (DLLs). This provides an Application Programming Interface (API) to the system services that reside in kernel memory. By using this API, application developers can write software that will survive most operating system upgrades. Usually, these applications do not call the Windows system services directly; instead, they go through one of these implemented APIs.

When an application in user-mode requests a system service, it usually involves invoking the Win32 APIs exported by any of the Win32 subsystem DLLs. These APIs may then make a call to any of the native API functions in ntdll.dll. The native API function then invokes the corresponding system service either by executing the software interrupt '*int 0x2e*' or the SYSENTER instruction, depending on the version of Windows NT kernel. In Windows 2000 and earlier versions of NT based operating systems, software interrupts are used to call the kernel-mode code. When an interrupt occurs, the CPU checks the Interrupt Descriptor Table (IDT) to determine what function should handle that event and then executes that function. The "System Service Dispatcher" (also known as *KiSystemService*), is the code responsible for handling system service calls. In Windows XP and newer versions of NT based operating systems, the mechanism involved in invoking *KiSystemService* is different. In these operating systems, the user-mode native API function in ntdll.dll directly executes the SYSENTER instruction which is provided by the CPU's instruction set to facilitate direct execution of a system service. On execution of this instruction the CPU checks the model-specific register IA32_SYSENTER_EIP (for Intel 32-bit processors) where the address of *KiSystemService* is stored. The value of this register is loaded into the instruction pointer and the dispatcher executes. The job of *KiSystemService* is to determine the requested system service and execute it. This it does by looking up an offset in the System Service Dispatch Table (or System Service Descriptor Table, SSDT) for the address of the requested service. The SSDT contains addresses of all system services available on the system. The dispatcher gets the address of the requested kernel-mode function (which is implemented in ntoskrnl.exe) and then calls it. Note that, before the user-mode thread is allowed to enter the kernel in order to service the request, its context is switched from user-mode to kernel-mode. When the thread returns back from kernel-mode to user-mode with the results, its context is switched back to user-mode.

Some of the executing components in user-mode are: user applications, service processes and system support processes. User applications are custom user-executed programs that are not part of the operating system. Service processes execute Win32 services, such as the Workstation and Server services that can be configured to start automatically or manually and their execution is controlled by the service control manager (SCM). System support processes are loaded by the operating system but are not started by the SCM. Examples of such processes are the Logon process (*winlogon.exe*), Session manager (*smss.exe*), and the SCM (*services.exe*).

3.3 Virtual memory

Windows NT allocates each process its own virtual address space. This virtual memory is a logical view of the actual physical memory. The memory manager (software component), with assistance from hardware (CPU feature), maps the virtual addresses at run time to corresponding physical addresses [13,22]. Parts of virtual memory belonging to each process are "paged out" to a file on disk called the *pagefile*. When a paged virtual address is referenced, the memory manager loads the data back into physical memory from disk.

On 32-bit Windows NT based operating systems, the virtual memory system is based on a flat 32-bit address space, which allows each process to "see" a total of 4 GB of private virtual memory. The address space layout consists of the following four regions [26,pp. 420–428]:

- 0x00000000 to 0x0000FFFF: No-access region to aid programmers.
- 0x00010000 to 0x7FFEFFFF: Process' private address space.
- 0x7FFF0000 to 0x7FFFFFFF: No-access region that prevents threads from passing buffers across the user/system space boundary.
- 0x80000000 to 0xFFFFFFFF: System addresses space where the Windows executive, kernel, and device drivers are loaded. Only kernel-mode processes have the privilege to manipulate this portion of memory.

Usually the system address range begins at 0x80000000. However, it is not right to assume this because of the ability to boot Windows with the/3 GB boot.ini switch. In order to determine the correct system address range start address, we can use the native API call to *NtQuerySystemInformation* (exported by *ntdll.dll*) with the *SystemInformationClass* parameter set to *SystemRangeStartInformation* sub-function (whose information class number is 50).

3.3.1 Extended virtual addressing for x86 (32-bit addressing mode)

The Windows 32-bit server operating systems support the following extended virtual addressing options suitable for large Intel machines with 4 GB or more of RAM.

(a) Application Memory Tuning (/3 GB boot switch), which allows user address range to grow to a maximum of 3 GB while shrinking the system address space to 1 GB. Only applications compiled and linked with the /LARGEADDRESSAWARE compiler switch (that defines IMAGE_FILE_LARGE_ADDRESS_AWARE in the image header) can allocate a private address space larger than 2 GB. If the /3 GB switch is used, the maximum RAM addressable by any Windows version is 16 GB.

(b) Physical Address Extension (PAE), which provides support for 36-bit real addresses on Intel Xeon 32-bit processors allowing them to address as much as 64 GB of RAM, i.e. 32 bit virtual addresses can be mapped into RAM pages above the 4 GB boundary. This hardware feature is supported by Windows NT, 2000, XP, 2003 and later. This feature is activated by using the /PAE switch in the boot.ini file, but can also be automatically enabled if the processor supports hardware data execution prevention (DEP). This feature does not change the size of the virtual address space, but allows for more processes to be resident in RAM, thus reducing paging.

(c) Address windowing extensions (AWE), are API calls which permit 32-bit process address spaces access to real addresses above their 4 GB virtual address limitations. Usually, AWE is used by applications in conjunction with PAE to extend their addressing range beyond 32-bits. Note that the size of the virtual address space is not changed but different RAM pages are mapped into application specified virtual addresses. The application program has to be specifically designed to use this feature.

3.3.2 Memory management on 32-bit and 64-bit Windows

The total number of addresses available in virtual memory is determined by the width of the registers in the CPU. The bit size of a processor refers to the size of the address space it can reference. A 32-bit processor can reference 2^{32} bytes, or 4 GB of memory (in flat addressing mode). 64-bit processors are theoretically capable of referencing 2^{64} locations in memory, or 16 EB (exa-bytes), which is more than 4 billion times the number of memory locations 32-bit processors can reference. However, all 64-bit versions of Microsoft operating systems currently impose a 16 TB limit on address space (addressing limit of 44 bits out of the available 64-bits) and allow no more than 128 GB of physical memory due to the impracticality of having 16 TB of RAM. Note that the AMD 64-bit processors implement a virtual address space of 48-bits (256TB), while the Intel Itanium2 64-bit processors implement a virtual address space of the full 64-bits (16EB) [23]. Processes created on 64-bit version of Windows are allotted 8 TB of user address space and 8 TB of kernel address space, with 4 GB virtual address space added for 32-bit "large address space aware" applications. Hence, the previously mentioned extended virtual addressing are no longer needed with 64-bit Windows operating systems running on 64-bit hardware.

On 64-bit Windows operating systems, 32-bit processes are simply separate 64-bit processes with a special *thunking* layer that sets up an environment in which 32-bit applications are run. This layer is called “Wow64”, short for “Win32 on Windows 64”. A 32-bit application can detect whether it is running under WoW64 by calling the *IsWow64Process* function. The WoW64 emulator consists of the following DLLs:

- *Wow64.dll* provides thunks for the *ntoskrnl.exe* entry-point functions.
- *Wow64Win.dll* provides thunks for the *win32k.sys* entry-point functions.
- *Wow64Cpu.dll* provides x86 instruction emulation on Intel Itanium processors. This DLL is not necessary for AMD x64 processors because they execute x86-32 instructions at full clock speed.

Along with the 64-bit version of *ntdll.dll*, these are the only 64-bit binaries that can be loaded into a 32-bit process. Note that 32-bit processes cannot load 64-bit DLLs (except for the ones mentioned above), and 64-bit processes cannot load any 32-bit DLLs. The Win32 API functions *CreateProcess* and *ShellExecute* can launch 32-bit and 64-bit processes from either 32-bit or 64-bit processes. Also, 64-bit Windows operating systems (such as Windows Vista x64 Edition), will only install on 64-bit hardware, while the 32-bit versions (such as x86 edition of Windows Vista) can run on 64-bit hardware as a 32-bit operating systems. Architectural limits for 32-bit and 64-bit Windows virtual memory can be found in [18], while maximum RAM support by 32-bit and 64-bit editions of Windows can be found in [16].

4 Enumerating objects in memory

There are several Win32 and Native APIs that help enumerate processes, process heaps, threads, loaded modules, and device drivers in user-mode. Windows 9x/ME and 2000 provide a built-in implementation (i.e. implemented by *kernel32.dll*) of Tool Help Library. On the other hand Windows NT uses, for the same purpose, the PSAPI library. There are also tools available that use these methods such as *Userdump.exe* which is part of the OEM Support Tools for Windows and is a user-mode process dumper and viewer. The use of Win32 native APIs, although not recommended by Microsoft, can be extremely useful while enumerating these objects in memory. Following are the different methods (or functions) that can be adopted to enumerate various objects in memory:

- PSAPI functions (*psapi.dll*)—can be used to enumerate processes, modules (such as dynamically or statically loaded DLLs by a process) and device drivers.
- Tool Help Library (*kernel32.dll*)—can be used to enumerate processes, threads, modules, and heaps.
- ADVAPI function (*advapi32.dll*)—can be used to enumerate services installed via the SCM (Service Control Manager).
- Performance Counters (*pdh.dll*)—can be used to enumerate processes and threads.
- Windows Management Instrumentation (WMI)—can be used to enumerate processes.
- Terminal server functions (*Wtsapi32.dll*)—can be used to enumerate processes on a terminal server.
- NTVDM sub-system functions (*vdmdbg.dll*)—can be used to enumerate 16-bit processes (or tasks) within each instance of *ntvdm.exe*.
- The native API *NTQuerySystemInformation* (*ntdll.dll*)—can be used to enumerate processes, threads and establishing parent-child process relations. These relations assist in terminating malicious processes that spawn multiple child processes.
- The native API *NtQueryInformationProcess* (*ntdll.dll*)—can be used to enumerate process modules and heaps within a process. It can also be used to establish parent-child process relations. This function also allows access to the PEB (Process Environment Block) of a process.
- The native API *NtQueryInformationThread* (*ntdll.dll*)—can be used to enumerate threads within a process. This function also allows access to the TEB (Thread Environment Block) of a thread, which in turn can be used to access the PEB of the process it belongs to.
- The native API *RtlQueryProcessDebugInformation* (*ntdll.dll*)—can be used to enumerate loaded modules and heaps within a process.
- The native API *NTQuerySystemInformation* (*ntdll.dll*) in combination with *NtQueryObject* (*ntdll.dll*)—can be used to enumerate open handles system wide.
- Using direct read of kernel memory from user-mode by exploiting read access and granting write access to the \\Device\\PhysicalMemory section object—can be used to enumerate processes and loaded modules.

A brief discussion of the use of each of these functions follows.

4.1 Enumeration using NTQuerySystemInformation native API

The Win32 API layer is a high-level interface to a subsystem built on top of the native API layer. Although a Win32 application can directly access the native API, this is not officially supported by Microsoft’s developer tools. Access to the native API is possible due to the system component *ntdll.dll*. This DLL allows us to call a subset of the functions exported by the kernel module *ntoskrnl.exe* from a user-mode application. The functions exported by *ntdll.dll* are runtime functions (executed entirely in user-mode), and kernel function wrappers (that perform a switch from user-mode to kernel-mode and back). In order to call the exported functions in *ntdll.dll* it is imperative that we also have the import library *ntdll.lib* which is included with the Platform SDK. Hence, in order to interface with *ntdll.dll*, we need to write a header file that doesn't conflict with the Platform SDK header files, and contains just enough definitions (such as constant, type and prototype function definitions) to call the desired *ntdll.dll* functions. We also need to import *ntdll.lib* by means of a linker directive in the header file.

While there are the Nt* family of native APIs, there are also the Zw* family of native APIs with the same names, except for the different prefix. If called from a user-mode application, both these families of APIs point to the same location, and essentially take the same execution path. This is not true in case of kernel-mode though, i.e. each of these families of APIs when called from kernel-mode traverse different execution paths [30].

The prototype for *NtQuerySystemInformation* is as shown:

```
NTSTATUS NTAPI NtQuerySystemInformation(
__in SYSTEM_INFORMATION_CLASS
  SystemInformationClass,
__out PVOID SystemInformation,
__in DWORD SystemInformationLength,
__out_opt PDWORD ReturnLength);
```

SystemInformationClass selects the sub-function to be called i.e. the type of information to retrieve. We are interested in the *SystemProcessAndThreadInformation* sub-function (whose information class number is 5). Many of thee sub-functions are merely wrappers around the internal *ntoskrnl.exe* functions; and in this case the base function is *ExpGetProcessInformation*. *SystemInformation* is a pointer to a buffer that receives the requested information, and *SystemInformationLength* is the size of the receiving buffer. The optional *ReturnLength* argument indicates how many bytes were copied to the buffer. The required size of the buffer depends on the sub-function being called and hence requires us to call *NtQuerySystemInformation* within a loop and check for the return code STATUS_INFO_LENGTH_MISMATCH, while dynamically increasing the buffer size until a return code of STATUS_SUCCESS is received. The *SystemProcessAnd ThreadInformation* sub-function returns an array of SYSTEM_PROCESS_INFORMATION structures that contain information about each process in memory. Each of these process structures contains an array of fixed-length thread structures called the SYSTEM_THREAD_INFORMATION. Because the length of each process structure varies with the number of threads the corresponding process hosts, the process structures are linked by a member that indicates how many bytes need to be skipped to get to the next list item. Other important members of the process structure are thread-count, handle-count, process-name, process-id and parent process-id. Important members of the thread structure are start-address, thread-id and process-id. Similarly, in order to obtain a list of all loaded drivers using, we pass in the *SystemInformationClass* parameter as *SystemModuleInformation* sub-function (whose information class number is 11). Sample code can be found in [8,9,24]. The use of native APIs is not recommended by Microsoft since associated internal structures could change from one version of Windows to other.

4.2 Enumeration using PSAPI functions

The process status application programming interface (PSAPI) is a helper library that provides functions to obtain information about processes and device drivers. These functions are available in *psapi.dll*. It is preferred to link with *psapi.dll* dynamically by loading the library with *LoadLibrary* and then obtaining addresses of necessary functions using *GetProcAddress*. The functions required for enumeration are: *EnumProcesses*, *EnumProcessModules*, *GetModuleFileNameEx*, *EnumDeviceDrivers*, *GetDevice DriverFileName*.

The *EnumProcesses* function returns an array of process identifiers (PIDs) for all running processes. The size of the buffer required to hold all PIDs is allocated dynamically in an incremental manner by calling *EnumProcesses* within a loop and checking for the return code of ERROR_NOT_ENOUGH_MEMORY. After retrieving the list of process identifiers, each PID can be used with the *OpenProcess* function in order to obtain a handle to the process. *OpenProcess* is to be called with PROCESS_QUERY_INFORMATION| PROCESS_VM_READ access rights. *OpenProcess* does not allow obtaining a handle to The "System Idle" Process or the "System" Process and fails with ERROR_ACCESS_DENIED error. Certain processes that run with higher privileges (such as CSRSS.EXE that runs as a SYSTEM process) have a security descriptor set that doesn't allow opening the process with necessary access rights. This issue can be resolved by enabling the *SeDebugPrivilege* (i.e. SE_DEBUG_NAME privilege) for the enumerating process. With this privilege turned on, the calling thread can open process handles with any access rights (PROCESS_ALL_ACCESS) regardless of the security descriptor assigned to a process. This privilege is granted only to users belonging to the Administrator group.

The process handle can then be passed to *EnumProcessModules* which retrieves a list of module handles that were loaded into the processes' address space. The first module in the list is always the executable file used to create the process. Each module handle can then be passed to *GetModuleFileNameEx* to obtain complete path to the file on disk associated with the loaded module. In order to obtain additional information of a loaded module, use the *GetModuleInformation* function. This function takes a process handle and a module handle and returns a MODULEINFO structure with the load address of the module, the size of the linear address space it occupies, and a pointer to its entry point. Sample code can be found in [8].

On 64-bit Windows NT based operating systems, if *EnumProcessModules* is called from a 32-bit application running under WoW64 (x86 emulator for 64-bit), it can only enumerate the modules of a 32-bit process. If enumeration were to be implemented via a 64-bit application then it is better to use the *EnumProcessModulesEx* function which allows for better filtering of results. The filter criteria can be set to one of the following values: LIST_MODULES_32BIT (in order to list the 32-bit modules), LIST_MODULES_64BIT (in order to list the 64-bit modules), and LIST_MODULES_ALL (in order to list all modules). If this function is called by a 32-bit application running under WoW64, the filter flag option is ignored.

Using PSAPI library functions we can enumerate device drivers as well. The Windows Driver Foundation (WDF) defines a single driver model that is supported by two frameworks: a user-mode driver framework (UMDF) and a kernel-mode driver framework (KMDF). User-mode drivers do not have access to the kernel-mode address space and therefore cannot compromise the integrity of the kernel. These drivers run in a driver host process, which runs with the security credentials of a *LocalService* account, although the host process itself is not a Windows service. Thus, user-mode drivers are as secure as any other user-mode service. Native drivers on Windows 2000 and later run in user mode.

Modules within device drivers are global to the system. We can use the *EnumDeviceDrivers* function in order to retrieve the load address for each device driver. We can then use the *GetDeviceDriverFileName* function that takes the load address and returns the complete path to the device driver. On 64-bit Windows NT based operating systems, *EnumDeviceDrivers* fails if called from within a 32-bit application, and will only succeed if called from within a 64-bit application. Note that 32-bit driver support has been removed in 64-bit Windows Vista.

The PSAPI enumeration functions ultimately call the native API *NtQuerySystemInformation* (implemented in *ntdll.dll*). Hence, a malware that hooks this native API (using any of the user-mode or kernel-mode hooking techniques) can easily bypass enumeration via PSAPI functions.

4.3 Enumeration using tool help library

The tool help library functions provide the ability to take a *snapshot* (a read-only copy) of the current state of processes, threads, modules, and heaps that reside in system memory. The tool help functions are implemented in *kernel32.dll*, while the structure definitions are defined in *tlhelp32.h*. In order to take a snapshot of the system memory, the *CreateToolhelp32Snapshot* function can be used. One or more of the following values can be specified when calling this function:

- *TH32CS_SNAPHEAPLIST*—includes the heap list of the specified process
- *TH32CS_SNAPMODULE*—includes the module list of the specified process
- *TH32CS_SNAPPROCESS*—includes the list of all running processes in memory
- *TH32CS_SNAPTHREAD*—includes the list of all active threads in memory

In order to enumerate all of these, we can specify the *TH32CS_SNAPALL* value. Note that the function call fails if we try to retrieve information for a 64-bit process from within a 32-bit process. Sample code can be found in [8].

To enumerate heap nodes of a particular process, we can use the *Heap32ListFirst* and *Heap32ListNext* functions with a handle to the processes' snapshot. Both these functions fill a *HEAPLIST32* structure which contains the process-id and heap-id as its members. Blocks within the heap nodes can be enumerated by using the *Heap32First* and *Heap32Next* functions. Both these functions fill a *HEAPENTRY32* structure with information for the appropriate block of a heap such as start address and size of the heap block. This information could be used to read the contents of each heap block into a buffer (using the *ReadProcessMemory* function) and scanned by the memory scanner.

To enumerate modules loaded by a particular process, we can use the *Module32First* and *Module32Next* functions with a handle to the processes' snapshot. Both these functions fill a *MODULEENTRY32* structure. The important members of this structure are:

- The process-id of the process whose modules are being examined.
- A handle to the module in the context of the owning process.
- The base address of the module in the context of the owning process.
- The size of the module, in bytes.
- The module name.
- The module path.

This information could again be used to read the memory contents of each loaded module into a buffer (using the *ReadProcessMemory* function) and scanned using the memory scanner.

On 64-bit Windows NT based operating systems, using the *CreateToolhelp32Snapshot* function in a 32-bit application to retrieve module information will only include 32-bit modules, while using it in a 64-bit application will only include 64-bit modules. This can be overcome by using the *TH32CS_SNAPMODULE32* flag which includes all 32-bit modules when run on 64-bit Windows.

To enumerate all active processes in memory, we can use the *Process32First* and *Process32Next* functions. Both these functions fill a *PROCESSENTRY32* structure, which holds information about the executable file, such as the process-id of its corresponding process, and the process-id of the parent process. These process-ids can be used to establish parent-child relationships between different processes which are helpful while terminating a parent malicious process and all its malicious child processes. The memory contents of a specific process can be read into a buffer (using the *Toolhelp32ReadProcessMemory* function or the combination of *VirtualQueryEx* and *ReadProcessMemory* functions) and scanned using the memory scanner.

To enumerate all active threads in the system user space, we can use the *Thread32First* and *Thread32Next* functions. Both these functions fill the *THREADENTRY32* structure. Two important pieces of information retrieved are the thread-id and the process-id of the process that created that thread. The thread-id and process-id can be passed on to *OpenThread* and *OpenProcess* functions respectively in order to obtain a handle to each. The process handle in particular can be used with the following functions to retrieve more information: *GetProcessImageFileName*, *GetModuleFileNameEx*, *QueryFullProcessImageName*.

The *GetProcessImageFileName* function requires a handle to the process which must at least have the PROCESS_QUERY_INFORMATION access right and returns the name of the executable file for the specified process. The full path to the executable file is in device form, rather than drive letters. The device form name can be converted to a drive letter by using the *GetLogicalDriveStrings* and *QueryDosDevice* functions.

The *GetModuleFileNameEx* function requires a handle to the module and a handle to the process which must at least have the PROCESS_QUERY_INFORMATION and PROCESS_VM_READ access rights. It returns the fully-qualified path for the file containing the specified module. If the handle to the module parameter is NULL, then this function returns the fully-qualified path to the file associated with the process for which the handle has been specified.

The *QueryFullProcessImageName* function can be used to retrieve the full name of an executable image for both 32-bit and 64-bit processes from within a 32-bit process. This function is present only in Windows Vista and Windows Server 2008. It requires a handle to the process which must at least have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access rights.

Note that the tool help library functions are similar to the PSAPI enumeration functions in that they too ultimately call the native API *NtQuerySystemInformation* (implemented in *ntdll.dll*). Hence, a malware that hooks this native API (using any of the user-mode or kernel-mode hooking techniques) can easily bypass enumeration via tool help library functions as well.

4.4 Enumeration using performance counters

The Windows NT based operating systems provide interfaces in order to obtain system information in the form of performance counters. There are two interfaces for this purpose, namely, the registry interface and the PDH (Performance Data Helper) interface. The PDH interface is essentially a higher-level abstraction of the functionality that the registry interface provides and is much easier to use than the registry interface. It was introduced in Windows NT 4.0 as a redistributable in the Microsoft Platform SDK and later became part of the system since Windows 2000. The PDH functions are made available via *pdh.dll*. Performance data can be collected from either real-time sources or log files. For our purpose of enumerating processes we will use the real-time sources. The performance monitoring architecture defines several *objects*. Each *object* can have one or more *instances*. Each of these *instances* is associated with a set of *performance counters*. For our purpose, we would want to enumerate all *instances* of the *object* named "*Process*", "*Thread*" and "*Process Address Space*".

The "*Process*" performance object consists of counters that monitor running application programs and system processes. The counters we are interested in are: "*Creating Process ID*"—that shows the identifier of the process that created a process, and "*ID Process*"—that shows the unique identifier of a process. Note that a "*Creating Process ID*" counter may no longer identify a running process since the creating process might have terminated after it has created a process. On the other hand, the "*ID Process*" numbers are reused and only identify a process for the lifetime of that process.

The "*Thread*" performance object consists of counters that measure aspects of thread behaviour. The counters we are interested in are: "*ID Process*"—that shows the unique identifier of a process, "*ID Thread*"—that shows the unique identifier of a thread, "*Start Address*"—that shows the starting virtual address for a thread, and "*Thread State*"—that shows the current state of a thread. Just as "*ID Process*", the "*ID Thread*" numbers are reused, so they only identify a thread for the lifetime of that thread. The "*Thread State*" values can be any of: 0 (initialized), 1 (ready), 2 (running), 3 (standby), 4 (terminated), 5 (waiting), 6 (transition), and 7 (unknown).

The "*Process Address Space*" performance object consists of counters that monitor memory allocation and use for a selected process. The counter we are interested in is: "*ID Process*"—that shows the unique identifier of a process. This counter is considered "costly", meaning that it takes a long time to collect data from them.

In order to enumerate processes and threads using performance counters, we can use the *PdhEnumObjectItems* function. This function requires as arguments the object to enumerate (which could be the "*Process*", "*Thread*" or "*Process Address Space*" objects), and two buffers (along with their sizes) that would hold the counter list and instance list. In order to determine the required buffer sizes, we have to call the *PdhEnumObjectItems* function once with the buffers pointing to NULL and sizes zero, then check for the return code of PDH_MORE_DATA that will populate the sizes parameters with the required sizes for these buffers. Then allocate the buffer sizes and call *PdhEnumObjectItems* again in order to enumerate processes or threads. The enumerations are stored in the instance list. This function also takes a parameter called "data source" which can be specified as NULL to denote collecting of real time data instead of from a log file. Another parameter it takes is called "machine name" which can also be specified as NULL to denote enumeration on local machine. Consecutive calls to this function will return identical lists of counters and instances. In order to refresh the list of performance objects, we can use the *PdhEnumObjects* function, with one of its parameters, which is a "refresh flag", set to TRUE, before calling *PdhEnumObjectItems* again.

Another method using PDH functions to enumerate processes and threads involves the following steps (Fedotov, 2006a):

- Create a query using the *PdhOpenQuery* function
- Add a counter to the query using the *PdhAddCounter* function
- Collect the performance data using the *PdhCollectQueryData* function
- Display the performance data using the *PdhGetRaw CounterArray* function
- Close the query using the *PdhCloseQuery* function

Before we open a query and add a counter to it, we have to note that object names and performance counter names are localizable. This means for a non-English version of Windows NT based operating system, process object and process identifier counters are no longer called "*Process*" and "*ID Process*", but rather localized names are used. In order to get the localized names and format a full path to the performance counter, we have to use the following additional functions: *PdhLookupPerfNameByIndex*, *PdhMakeCounterPath*.

The *PdhLookupPerfNameByIndex* function requires the index of the performance object or counter to be looked up by name. Here are a few example indices on English language systems: index 230 is "*Process*", index 238 is "*Processor*", index 6 is "*% Processor Time*", and index 784 is "*ID Process*". We call this function twice, once in order to get the required buffer size, and the second time to get the data. This is done by checking for return code PDH_MORE_DATA or PDH_INSUFFICIENT_BUFFER. These localized values are set in the PDH_COUNTER_PATH_ELEMENTS structure, along with some other members, and passed onto the *PdhMakeCounterPath* function. Again, this function is called twice, once in order to get the required buffer size, and the second time to get the data (counter name). We can now open a query using the *PdhOpenQuery* function and add the obtained counter name to the query using the *PdhAddCounter* function, which returns a handle to the counter. Then use the *PdhCollectQueryData* or *PdhCollectQueryDataEx* functions to collect the query data. These functions signal the application-defined event and wait the specified time interval before returning. The next step is to call the *PdhGetRawCounterArray* function that returns an array of raw values from the specified counter. This function requires a handle to the counter (which was previously obtained via the *PdhAddCounter* function), for which current raw instance values are to be retrieved. Again, this function is called twice, once in order to get the required buffer size, and the second time to get the data. The data for the counter is locked for the duration of the call to *PdhGetRawCounterArray* in order to prevent any changes during processing of the call. This function returns the number of raw counter values and populates an output buffer with an array of PDH_RAW_COUNTER_ITEM structures for each counter. Each structure contains the *instance name* (which in our case would correspond to the process name) and *raw value* for a single counter. The *raw value* itself is a structure of type PDH_RAW_COUNTER which contains an important member called *FirstValue*, which in our case would correspond to the process-id. Note that this is the only method that allows enumerating processes on another machine, for which we have to set the machine name in the PDH_COUNTER_PATH_ELEMENTS structure before passing it the *PdhMakeCounterPath* function.

The advantage of using these APIs is that it provides a different view to obtain the list of active processes and threads. This information is maintained and retrieved from a different set of data structures than the ones used by the previously discussed methods. The disadvantage is that there are no PDH APIs to enumerate loaded modules within processes. Also, a malware could easily hook these user-mode APIs in order to return manipulated results and essentially hide its malicious processes and threads from enumeration.

4.5 Enumeration using windows management instrumentation

Windows management instrumentation (WMI) is Microsoft's implementation of Web-based enterprise management (WBEM) and common information model (CIM) standards from the distributed management task force (DMTF). It extends the windows driver model (WDM) and provides for uniform access of data from different management sources while extending existing management protocols such as the simple network management protocol (SNMP). WMI is included since Windows 2000 and Windows XP and is available as a redistributable for previous versions of Windows. The WMI interface is based on component object model (COM) technology and provides for process enumeration functions. Sample code can be found in [8]. Again, a malware could hook the WMI or COM interfaces that service these enumerations in order to hide its malicious processes.

4.6 Enumerating processes on a terminal server

In order to enumerate processes on a terminal server, we can use the functions exported by *Wtsapi32.dll*. The *WTSEnumerateProcesses* function retrieves information about the active processes on a specified terminal server. This function requires a handle to a terminal server which can be opened with the *WTSOpenServer* function. This function requires a pointer to a null-terminated string specifying the NetBIOS name of the terminal server. The *WTSCloseServer* function is used to close the handle. If the application enumerating the processes is running on the terminal server itself then no handle need be opened, rather, the constant WTS_CURRENT_SERVER_HANDLE can be used. The enumeration function returns a pointer to an array of WTS_PROCESS_INFO structures along with a count of the number of structures. Each structure in the array contains information about an active process on the specified terminal server. To free the returned buffer, we can call the *WTSFreeMemory* function. The structures consist of members such as process-id and process-name (which is the name of executable file associated with the process).

4.7 Enumerating services

Malware could install malicious system services (such as a kernel driver or file system driver or even a Win32 process service) in order to operate in an escalated state. It is hence imperative to have an understanding of what services are currently active in memory and be able to enumerate them. We can use the *EnumServicesStatusEx* function in order to enumerate services within the specified service control manager database. This function requires a valid handle to the service control manager database, which can be obtained by using the *OpenSCManager* function with the SC_MANAGER_ENUMERATE_SERVICE access rights. In order to retrieve the name and service status information for each service, SC_ENUM_PROCESS_INFO is to be provided as another parameter. We can use this function to enumerate Win32 process services and kernel or file system driver services that are active. This function needs to be called twice, once in order to find the number of bytes to allocate for an output buffer (by checking for the return code of ERROR_MORE_DATA) and the second time to retrieve enumerated services information after allocating the required buffer size. The output buffer receives an array of ENUM_SERVICE_STATUS_PROCESS structures where each structure represents a service on the system. This structure contains information about the name of the service and another structure called SERVICE_STATUS_PROCESS that holds status information about the service. From this structure we can obtain information such as the type of the service, the current state of the service and more importantly the process-id of the service. Using this process-id we can obtain a handle to the process via *OpenProcess* that can be used to scan its memory image. Also, we can obtain the complete path and name of the associated file on disk by passing this handle to *GetModuleFileNameEx* function.

4.8 Enumerating 16-bit applications

On Windows NT based operating systems, 16-bit applications are run within an instance of NTVDM (NT Virtual DOS Machine, which is the Win16 subsystem). The NTVDM subsystem provides a set of services that allow a Win32 process to debug 16-bit Windows applications. This is provided via *vdmdbg.dll*. If any of the previous documented methods were to be used to enumerate processes (as PSAPI or Tool Help library), we would still not be able to enumerate 16-bit applications but rather only instances of *ntvdm.exe*. The system DLL *vdmdbg.dll* provides functions to enumerate all 16-bit processes running under NTVDM. The primary function for enumeration is *VDMEnumProcessWOW*, which will call a callback function in the calling application for every process on the machine which is a Win16 subsystem, i.e. an instance of *ntvdm.exe*. For each instance of *ntvdm.exe*, we have to enumerate the 16-bit processes running within them, which are referred to as *tasks*. For this, we use the *VDMEnumTaskWOWEx* function which also calls a callback function in the calling application for every *task* within each instance of *ntvdm.exe*. In this callback, we receive the module name and the full path of the executable which is running the *task* (16-bit application) [20]. Note that the 16-bit subsystem has been removed in Windows Vista.

4.9 Enumerating process modules using NtQueryInformationProcess native API

The native API *NtQueryInformationProcess* retrieves information about a specified process. The prototype for this function is as shown:

NTSTATUS NTAPI *NtQueryInformationProcess* (
__in HANDLE *ProcessHandle*,
__in PROCESS_INFORMATION_CLASS *ProcessInformationClass*,
__out PVOID *ProcessInformation*,
__in ULONG *ProcessInformationLength*,
__out_opt PULONG *ReturnLength*);

This function requires a handle to the process, which can be obtained via the *OpenProcess* function if a valid process-id is available. *ProcessInformationClass* selects the sub-function to be called i.e. the type of process information to retrieve. It could be any of *ProcessBasicInformation* (0), *ProcessDebugPort* (7), *ProcessWow64Information* (26), and *ProcessImageFileName* (27). *ProcessInformation* is a pointer to a buffer that receives the requested information, and *ProcessInformationLength* is the size of the receiving buffer. The optional *ReturnLength* argument indicates how many bytes were copied to the buffer. The required size of the buffer depends on the sub-function being called and hence requires us to call *NtQueryInformationProcess* within a loop and check for the return code STATUS_INFO_LENGTH_MISMATCH, while dynamically increasing the buffer size until a return code of STATUS_SUCCESS is received.

ProcessDebugPort retrieves the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a ring 3 debugger. This information can also be obtained using the *CheckRemoteDebuggerPresent* (for a remote process) or *IsDebuggerPresent* (for the current process) functions. *ProcessWow64Information* determines whether the process is running within a WoW64 environment. This information can also be obtained using the *IsWow64Process* function. *ProcessImageFileName* retrieves the name of the file on disk associate with the process. The most important sub-function we are interested in is *ProcessBasicInformation*. This sub-function retrieves a pointer to the PROCESS_BASIC_INFORMATION structure. This structure has few important members such as the process-id of current process, process-id of parent process, and pointer to the base address of current processes' process environment block (PEB).

Each process has a PEB. Any thread within the process can access the process' PEB or an injected thread within the process can access it as well. The PEB structure contains process information. Note that the PEB structure is different on 64-bit Windows (i.e. fields are of different sizes). Three important members of the PEB structure are:

- A pointer to the PEB_LDR_DATA structure that contains information about the loaded modules for the process.
- A pointer to the RTL_USER_PROCESS_PARAMETERS structure that contains process parameter information
  such as the command line and the path of the image file for the process.
- A pointer to a pointer that lists all the heaps within the process.

An important member of PEB_LDR_DATA structure is another structure of type LIST_ENTRY which basically is the head of a doubly linked list that contains the loaded modules for the process. Each item in the list is a pointer to an LDR_DATA_TABLE_ENTRY structure which corresponds to each loaded module. This structure holds the base address of the loaded module and the full path of associate file on disk. Along with this information, the LDR_DATA_TABLE_ENTRY structure contains pointers to lists such as: *InLoadOrderModuleList*, *InMemoryOrderModuleList*, and *InInitializationOrderModuleList*. The first two lists contain the application itself as the first module, followed by needed modules (DLLs). The last list contains *ntdll.dll* as the first module followed by *kernel32.dll*. Malware sometimes enumerate this list in order to get the base address of *ntdll.dll* and resolve addresses to native APIs in order to hook them, or get the base address of *kernel32.dll* and resolve addresses to *GetProcAddress* and *LoadLibrary* in order to dynamically load (import) and inject their own DLL (code). Again, the use of native APIs is not recommended by Microsoft since associated internal structures could change from one version of Windows to other.

4.10 From TEB to PEB using NtQueryInformationThread native API

The native API *NtQueryInformationThread* retrieves information about a specified thread. The prototype for this function is as shown:

NTSTATUS NTAPI *NtQueryInformationThread* (
__in HANDLE *ThreadHandle*,
__in THREAD_INFORMATION_CLASS *ThreadInformationClass*,
__inout PVOID *ThreadInformation*,
__in ULONG *ThreadInformationLength*,
__out_opt PULONG *ReturnLength*);

This function requires a handle to the thread, which can be obtained via the *OpenThread* function if a valid thread-id is available. *ThreadInformationClass* selects the sub-function to be called i.e. the type of thread information to retrieve. It could be any of *ThreadBasicInformation* or *ThreadQuerySetWin32StartAddress*. *ThreadInformation* is a pointer to a buffer that receives the requested information, and *ThreadInformationLength* is the size of the receiving buffer. The optional *ReturnLength* argument indicates how many bytes were copied to the buffer. The required size of the buffer depends on the sub-function being called and hence requires us to call *NtQueryInformationThread* within a loop and check for the return code STATUS_INFO_LENGTH_MISMATCH, while dynamically increasing the buffer size until a return code of STATUS_SUCCESS is received.

ThreadQuerySetWin32StartAddress retrieves the start address of the thread. On versions of Windows prior to Windows Vista, the returned start address is only reliable before the thread starts running. The sub-function we are interested in is *ThreadBasicInformation*, which retrieves a pointer to the THREAD_BASIC_INFORMATION structure. This structure contains information such as the thread's base priority, its exit status, and a pointer to the CLIENT_ID structure that contains the unique thread-id and process-id (to which the current thread belongs). The most important member of the THREAD_BASIC_INFORMATION structure is a pointer to the base address of the thread's TEB (Thread Environment Block). The base address of the TEB can also be obtained using the *NtCurrentTeb* native API call.

Each thread has a TEB. The TEB structure contains thread information. Some of its important members are: a pointer to the base address of the thread's TLS (Thread Local Storage) or TLS array, a pointer to the SDT (Service Descriptor Table) which in turn points to the SSDT (System Service Dispatcher Table), and a pointer to the PEB structure of the process that it belongs to. The PEB pointer is typically located at offset 0x30 inside the current TEB and this location has been stable across 32-bit Windows NT4, 2000, XP, and 2003. The SDT pointer is typically located at offset 0xDC on 32-bit Windows 2000 and at offset 0xE0 on 32-bit Windows XP, inside the current TEB. The FS segment register is always set such that the address FS:0 points to the TEB of the thread being executed. At offset 0x18 inside the current TEB is a pointer to self (i.e. pointer to the first thread's TEB). Hence the following are valid ways of obtaining the base addresses of TEB and PEB:

```
assume fs:nothing
mov eax, fs:[18h] ; get self pointer
    from TEB
mov ebx, fs:[30h] ; get pointer to PEB
mov ebx,dword ptr [eax+0x30] ; another
    way of getting pointer to PEB
```

Typically on a 32-bit Windows NT based operating system, the TEB is located at 0x7FFDE000 and the PEB is located at 0x7FFDF000. Each new thread's TEB is assigned an address growing towards 0x00000000. If a thread exits and a new thread is created then it will get the address of the previous thread's TEB. It is not advisable to rely on such hard-coded values since the internal structures and offsets could change from one version of Windows to the other.

We can obtain the base value of the FS segment register using documented Win32 API calls. For this, we make use of the *GetThreadContext* and *GetThreadSelectorEntry* functions. Before we call *GetThreadContext*, we have to suspend the thread using the *SuspendThread* function and then set the context-flags in the CONTEXT structure that specifies which portions of the thread context are retrieved. In order to retrieve registers context, we set the flags to CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS. Also, the function requires a handle to the thread with THREAD_GET_CONTEXT access rights. A 64-bit application can retrieve the context of a WoW64 thread using the *Wow64GetThreadContext* function and would additionally require THREAD_QUERY_INFORMATION access rights. The returned CONTEXT structure from *GetThreadContext* is then passed to the *GetThreadSelectorEntry* function (which is only functional on x86-based systems). This function retrieves a descriptor table entry for the specified selector and thread. The selector we specify here is the FS segment register. The returned descriptor table entry is a pointer to the LDT_ENTRY structure (which is again valid only on x86-based systems). This information can be used to convert a segment-relative address to a linear virtual address, so it can be passed on to the *ReadProcessMemory* function. *ReadProcessMemory* only uses linear virtual addresses. In order to calculate the base address of a segment (in our case it being the FS register), we need to combine the following members of the LDT_ENTRY structure: *BaseLow*—the low-order part of the base address of the segment, *BaseMid*—middle bits (16–23) of the base address of the segment, and *BaseHi*—high bits (24–31) of the base address of the segment.

With the base value of FS segment register; we can now use *ReadProcessMemory* to read the TEB and PEB of the specified process.

4.11 Enumerating process modules and heaps using native debug APIs

In order to enumerate loaded modules within a specific process, we need to first obtain its process-id. This can be done by using any of the above discussed methods of enumerating processes. We can then make use of the native debug APIs exported by *ntdll.dll* in order to enumerate modules within that process. This involves first creating a debug buffer using the *RtlCreateQueryDebugBuffer* function and then calling the *RtlQueryProcessDebugInformation* function to populate the debug buffer with module information. This function requires a "debug information class mask" to be passed in, which in this case would be PDI_MODULES. The debug buffer is populated with structures that contain information about a module such as the base address, image-size and full path to associated file on disk. The debug buffer can be freed using the *RtlDestroyQueryDebugBuffer* function. Sample code to enumerate modules using this can be found in [31]. Note that *RtlQueryProcessDebugInformation* creates a remote thread in the process to examine and return a read-only snapshot. In order to enumerate heaps of a specific process, the *RtlQueryProcessDebugInformation* function is called with "debug information class mask" set to PDI_HEAPS | PDI_HEAP_BLOCKS. In this case, the debug buffer is filled with structures that contain information such as the base address of the node, the allocated and committed sizes of the node, number of blocks in the node, and an array of pointers to HEAP_BLOCK structures representing each block in the node. Each of these HEAP_BLOCK structures can be traversed in order to obtain each heap's base address and size. Sample code to enumerate heaps using this can be found at [28].

Note that *RtlQueryProcessDebugInformation* creates a remote thread in the process to examine and returns a read-only snapshot.

4.12 Enumeration using direct read of kernel memory from user-mode

This method is an undocumented technique (or rather a hack) to directly access kernel memory from a user-mode application. This is done by exploiting read access and granting write access to the \\Device\\PhysicalMemory section object. A section object, also called a file-mapping object, represents a block of memory that two or more processes can share. Section objects can be mapped to a page file or some other on-disk file. As far as we know, the first use of this section object for viewing physical memory was by Mark Russinovich when he created the physical memory viewer tool called *Physmem* [21]. Since then, other proof-of-concept tools and techniques have emerged that take advantage of the \\Device\\PhysicalMemory section object in order to read and write parts of kernel memory directly from user-mode. Few examples are listed below:

- A tool called Kmem that shows reading kernel memory from user-mode [19].
- A technique to set up a call gate descriptor in the GDT (Global Descriptor Table, which exists in kernel-mode), by opening the \\Device\PhysicalMemory section object using *NtOpenSection* and then mapping it using *NtMapViewOfSecton* [3].
- Techniques to read and write kernel memory from user-mode [4].
- Technique to hide processes by directly manipulating kernel memory [1].
- Technique to modify SSDT from user-mode by writing to kernel memory [29].

The above methods require cryptic techniques to obtain addresses to un-exported kernel objects and conversion of virtual addresses to actual physical addresses in memory. We could use this undocumented method to read the EPROCESS structure from kernel memory in order to enumerate processes and loaded modules.

Starting with Microsoft Windows Server 2003 Service Pack 1 (SP1), which also includes Windows XP x64 SP1, user-mode applications cannot access \\Device\\Physical Memory directly and can only access it if a kernel-mode driver is used to pass a handle to the application. This is done by a call to *MmMapViewOfSection* function from a kernel-mode driver. But again this protection was bypassed [11]. Starting with Windows Vista, access to \\Device\\PhysicalMemory from user-mode has been completely removed.

4.13 Enumerating open file handles within a process

Sometimes it is imperative to enumerate open handles within a process in order to search for a specific type of handle. For example, the infamous *W32/Sober worm* opens a "*file*" type handle to self when in memory, preventing any other external program (such as an anti-malware scanner) from accessing its malicious image on disk. In this case, the memory scanner could enumerate all open "*file*" type handles within the process and close any those are open to self, enabling access to the malicious file on disk. We can enumerate open handles (of all types) system wide by using the native API *NtQuerySystemInformation* with the sub-function *SystemHandleInformation*. The function returns a structure called SYSTEM_HANDLE_INFORMATIOM. Since the size of the output buffer that would hold SYSTEM_HANDLE_INFORMATIOM structure depends on the number of open handles in the system which varies dynamically, this requires us to call *NtQuerySystemInformation* within a loop and check for the return code STATUS_INFO_LENGTH_MISMATCH, while dynamically increasing the buffer size until a return code of STATUS_SUCCESS is received. The SYSTEM_HANDLE_INFORMATIOM structure contains an array of structures each of type SYSTEM_HANDLE and the number of such structures. Each SYSTEM_HANDLE structure represents an open handle within the system. This structure holds important information such as the process-id of the process it is associated with and the *ObjectType* (which is the type of handle and can be any of *file*, *directory*, *symbolic link*, *process*, *thread*, *token*, *device*, etc.). For our purpose we are interested in "*file*" type handles.

For each handle (say, h) associated with a process-id (say, *pid*), we want to be able to gather information about the handle (h) such as associated object name and object type.

This can be done using the native API functions *NtQueryInformationFile* and *NtQueryObject*. For this, we have to first duplicate the handle (h) using the *DuplicateHandle* function. This function requires a handle to the process (with process-id *pid*) containing the handle (h) to be duplicated. For this, *OpenProcess* is used on *pid* with PROCESS_DUP_HANDLE access rights. The handle (h) is then duplicated with DUPLICATE_SAME_ACCESS to obtain a handle object (say *hobj*). Note that sometimes querying handle objects could lead to a deadlock situation causing the application to hang indefinitely. This can be avoided by creating a new thread (using *CreateThread*) and waiting for it to complete in the parent thread (using *WaitForSingleObjectEx*). The new thread could point to code that calls *NtQueryInformationFile* on the handle object (*hobj*), by passing the sub-function *FileNameInformation*. This test helps us avoid querying objects that have the potential to cause deadlocks. If the child-thread returns back to the parent before the time-out, then the handle object can be queried safely. Alternately, if the time-out is reached, which means the child thread is hung; in which case we terminate the hung child thread and infer that querying the handle object would lead to a deadlock situation causing the application to hang indefinitely. If this test were to finish successfully, we can then call *NtQueryObject* on the handle object. In order to obtain object name, the sub-function *ObjectNameInformation* is used. This returns a Unicode string of the object name which is in device form (eg: \\Device\\HardDisk1\). In order to obtain object type, we call *NtQueryObject* with the sub-function *ObjectTypeInformation*. Note that the size of the output buffer that would hold the results depends on the sub-function being passed and hence requires us to call *NtQueryObject* within a loop and dynamically incrementing the buffer size while checking for the return code of STATUS_INFO_LENGTH_MISMATCH, until a STATUS_SUCCESS is received. The object name and object type information can be used to check if a particular process has an open *file* type handle to self (as is the case with *W32/Sober*). When such a self *file* handle is found, it could be closed using the *DuplicateHandle* function, passing DUPLICATE_CLOSE_SOURCE as one of its parameters. Closing the self *file* handle in *W32/Sober* allows read access to its image on disk allowing complete removal of the malware.

4.13.1 Protected processes

The Microsoft Windows Vista operating system introduced a new type of process known as a *protected process* in order to enhance support for Digital Rights Management functionality in Windows Vista. Although any application can attempt to create a protected process, the operating system requires that these processes be specially signed by Microsoft. There are two known protected processes on Vista—*audiodg.exe* and *mfpmp.exe*. A typical process cannot perform the following operations such as, inject a thread, access virtual address space, debug, or duplicate a handle on a protected process, nor can it get/set context information or impersonate any thread belonging to the protected process. Also, only the following access rights are allowed to be obtained for a protected process: PROCESS_QUERY_LIMITED_INFORMATION and PROCESS_TERMINATE, while the following access rights are allowed to be obtained for any thread of the protected process, THREAD_QUERY_LIMITED_INFORMATION, THREAD_SET_LIMITED_INFORMATION, and THREAD_SUSPEND_RESUME. Except for the above privileges, no other privileges can be obtained for a protected process or thread, even if *SeDebugPrivilege* is enabled. These restrictions can be circumvented by installing a kernel-mode component in order to access the memory of a protected process. A proof-of-concept tool has already been written (that uses a kernel-mode driver) to demonstrate “un-protecting” a protected process, and make any process “protected” [12]. This shows that malware authors too could use kernel components and create malicious protected processes. A user-mode memory scanner would be unable to scan the virtual address space of such a process. The scanner could still enumerate all protected processes and scan the associated files on disk. If an infection is found, then the protected process in memory can still be terminated or its threads suspended.

4.13.2 Terminating malicious processes

In order to terminate malicious processes it is best to first acquire the *SeDebugPrivilege* so that a handle can be acquired to the target process regardless of the security descriptor assigned to it [17]. The handle can be obtained (using *OpenProcess*) with the terminate access right (PROCESS_TERMINATE) or any access right (PROCESS_ALL_ACCESS). We can then use any or all of the following methods in order to terminate malicious processes and threads [5]:

- Use the *TerminateProcess* function (exported by *kernel32.dll*). This function unconditionally causes a process to exit. All of the object handles opened by the process are closed and all threads belonging to the process terminate their execution, but DLLs attached to the process are not notified that the process is terminating. Also, terminating a process does not cause child processes to be terminated, nor does it necessarily remove the process object from the system. A process object is deleted when the last handle to the process is closed.
- Use the native API function *NtTerminateProcess* (exported by *ntdll.dll*).
- Use the *EndTask* function (exported by *user32.dll*). This works only if the target process has at least one window.

- Send the WM_CLOSE message to all windows in the target process using the *SendMessage* function (exported by *user32.dll*). This works only if the target process has at least one window and it doesn't handle the WM_CLOSE message.
- Send the WM_QUIT message to all windows in the target process again using the *SendMessage* function. Above mentioned restrictions apply.
- Send the SC_CLOSE system message to all windows in the target process again using the *SendMessage* function. Above mentioned restrictions apply.
- Enumerate all threads in the target process (using any of the discussed methods in previous sections) and terminate them individually using the *TerminateThread* function (exported by *kernel32.dll*). This requires obtaining a handle to each thread by using the *OpenThread* function with THREAD_TERMINATE or THREAD_ALL_ACCESS access rights.
- Enumerate all threads in the target process and terminate them individually using the native API function *NtTerminateThread* (exported by *ntdll.dll*).
- Enumerate all threads in the target process and suspend them, either using *SuspendThread* (exported by *kernel32.dll*) or *NtSuspendThread* (exported by *ntdll.dll*). Then use the *SetThreadContext* function (exported by *kernel32.dll*) and modify the EIP register (instruction pointer) of each to point to the *ExitProcess* function in *kernel32.dll*. Then resume each thread. This again requires obtaining a handle to each thread by using the *OpenThread* function with THREAD_SUSPEND_RESUME and THREAD_SET_CONTEXT access rights or THREAD_ALL_ACCESS access right.
- Create a new thread (as suspended) in the context of the target process using the *CreateRemoteThread* function (exported by *kernel32.dll*) with its start address pointing to *ExitProcess* function in *kernel32.dll*, and then resume the remote thread.
- Attach to the target process as a debugger by using the *DebugActiveProcess* function (exported by *kernel32.dll*) and simply terminate. This causes the process being debugged (i.e. the target process) to terminate as well.
- Obtain a handle to the target process and pass it to the *DebugBreakProcess* function causing the target process to terminate because of an un-handled breakpoint exception.
- In order to terminate 16-bit applications (or *tasks*) running within *ntvdm.exe*, we can use the *VDMTerminateTaskWOW* function (exported by *vdmdbg.dll*), which requires the process-id of the *ntvdm.exe* instance and the 16-bit task-id.

In order to terminate all child processes (i.e. spawned processes) of a malicious process, we need to establish parent-child relationships and obtain process-ids of all child processes. For this, we can use the following two techniques:

- Enumerate all processes using *NtQuerySystemInformation* and then use the *InheritedFromProcessId* information to enumerate all child process-ids.
- Enumerate all processes using *CreateToolhelp32 Snapshot*, *Process32First* and *Process32Next*. Then use the *th32ParentProcessID* information to enumerate all child process IDs.

If all attempts to terminate a malicious process fail, because it may be monitored and protected by some kernel-mode driver, or if user-mode APIs and native APIs related to process termination have been hooked by the malware, then we may at least want to suspend it in order to inhibit its activities. Another case would be where a system process (such as *explorer.exe*, *winlogon.exe*, *csrss.exe*, *smss.exe*) that should not be terminated, is found to be infected (say with a malicious injected DLL). In this case as well, we would want to simply suspend the process (although *explorer.exe* and *winlogon.exe* should not be suspended anyway in order for the computer to be functional). In order to suspend the process we could use the native API function *NtSuspendProcess* (exported by *ntdll.dll*). Another way is to enumerate all threads of the target process and suspend them individually using the *SuspendThread* function (exported by *kernel32.dll*). Sufficient access rights are to be granted when handles to the threads and process are obtained.

If all attempts to terminate and suspend a malicious process fail, we could also consider forcing it to crash. This must be approached with caution since it could sometimes lead to system instability, failure of other applications, or system hang, if the malware is deeply injected in system processes or has hooked system calls and tables. Two methods to forcefully crashing the target process are [5]:

- Enumerate all commit memory pages of the target process using the *VirtualQueyEx* function and then set the access level for those memory pages to PAGE_NOACCESS using the *VirtualProtectEx* function. This effectively prevents all read, write and execute operations on those pages, eventually forcing the target process to crash due to its inability to execute code.
- Enumerate all commit memory pages of the target process using the *VirtualQueyEx* function and then use the *WriteProcessMemory* function to overwrite those pages with junk data, eventually causing the target process to crash due to attempting to execute invalid code.

Some of the system critical processes in memory should not be suspended nor terminated in order to maintain system stability and usability. Such system critical processes are:

winlogon.exe, *explorer.exe*, *services.exe*, and *csrss.exe*. If any of these processes are found to be infected in memory, then either a reboot is required in safe mode preceded by a registry cleaning routine in order to get rid of any malware that might load on system reboot, or scanning from a clean OS loaded from an alternate boot device.

If *lsass.exe* were found to be infected in memory (i.e. via remote code/DLL injection), it is safe to suspend it in order to disinfect the machine, provided we are not enumerating any processes (or modules) by escalating to *SeDebugPrivilege*. This is because if *lsass.exe* were to be suspended while we are still enumerating processes (or modules) would cause the enumerating application to hang indefinitely. This is because, when we try to escalate privileges, one of the Win32 API function used is *LookupPrivilegeValue* which basically uses the RPC server and *lsass.exe* to retrieve information. If *lsass.exe* is suspended during this time, the application will hang indefinitely for the service.

5 Summarizing user-mode memory scanning

The basic idea is to enumerate active memory components visible from user-mode such as processes, services, loaded modules, loaded drivers, etc. and scan the associated files on disk. The actual memory image associated with each component is scanned as well. The memory image of a process is read by using a combination of *VirtualQueryEx* and *ReadProcessMemory* functions. *VirtualQueryEx* enumerates all memory pages within the specified process and the information is returned in a MEMORY_BASIC_INFORMATION structure. This structure has information such as *base address* and *region size*. The base address can correspond to any of the loaded modules within the process. *Base address + region size* will point to the next region of memory. When the "maximum application address" is reached, we have a count of the number of memory pages for the process. Using this count we could iterate to find total buffer size required to store only "commit" pages (i.e. memory pages that have state as MEM_COMMIT). We can then use *ReadProcessMemory* to read each commit page and store it in a buffer. This buffer can eventually be passed to the memory scanner.

This approach can be used to detect earlier versions of the infamous *Storm Trojan's* (a.k.a. *Zelethan*, *Peacomm*) injected code into *services.exe*. The Trojan drops a malicious kernel-mode driver that has an embedded payload (as an embedded executable). The payload is injected from kernel space into the user space of *services.exe* and scheduled for execution by queuing an Asynchronous Procedure Call (APC) for it. Due to this, there is no "visible" process executing the payload if we were to use any of the enumeration techniques in order to enumerate processes. Scanning the committed memory pages of *services.exe* will reveal the injected code.

When an attempt to scan an associated file on disk for a particular process fails due to the file not being present on disk, this could imply that the file is hidden from Windows API (using rootkit like techniques) or the file is deleted from disk once it is loaded into memory. This was seen with *W32/OnlineGames.AYW* which dropped a malicious kernel-mode driver (detected as *W32/SysTrojan.A*) that existed on disk only for a very brief instance, and was deleted by the malware as soon as it was loaded as a service into memory. This ensured that the malicious driver existed only in memory and not on disk. On subsequent reboots, the malware would re-create the malicious driver file on disk again for a brief instance and delete it again once loaded in memory. In this case, try to scan the memory image of the process in question. Any failed attempt to suspend or terminate the malicious process (because another malicious process in memory could be protecting it) results in adding it to the "pending terminates list". This list is visited again after complete memory scan. If we still fail to terminate or are only able to suspend the malicious processes listed in the list, then the user is to be notified of an un-resolved infection.

When an attempt to scan an associated file on disk for a particular process fails due to access violation to open the file for reading, this could imply that the file is locked by another malicious process in memory or that the associated process has an open handle to self. In this case, the file path is added to a "pending scans list". This list is visited after complete memory scan in order to attempt to scan the file in question again. If still read access to file is denied, and an open "*file*" handle to self is found, then try to close such a handle, and if successful, try to scan the file on disk again.

When the associated file on disk is scanned for a particular process and is found to be clean, proceed to scan all loaded modules by that process. If an infection pertaining to a loaded module is found, instead of trying to terminate the process, only try to suspend the process after making sure it is not one of the critical system processes (such as *winlogon.exe* or *explorer.exe*). If critical system processes are found to be infected then the user is notified of un-resolved infections that would require a reboot in safe mode (or booting into a clean OS using alternate boot devices) and re-scanning of memory. If both the associated file on disk and loaded modules are found to be clean, then proceed to scan the memory image of the process. This is important because a memory resident malware could disinfect its associated files on disk on-access (i.e. when opened for read by an external program) and re-infect them back on close.

5.1 Scanning for hidden processes from user-mode

One of the most effective methods to scan for hidden processes (that could be hidden via a kernel-mode driver) from user-mode is to use the technique used by the *BlackLight* rootkit detection tool [10]. It basically calls the *OpenProcess* function on process-ids ranging from 0x00 to the maximum allowed process-id of 0x4E1C, while keeping track of all successful calls. A successful call to *OpenProcess* means that process-id belongs to a valid process in memory. Then use any of the high-level user-mode APIs to enumerate processes (and process-ids), and compare this list with the previously obtained list using *OpenProcess*. Any discrepancy denotes a hidden process. Note that this technique too can be thwarted by manipulating certain structures within the kernel [10].

Use all of the methods discussed before in order to enumerate processes and compare the results from each. If there is any discrepancy in the results, then it denotes the compromised state of a machine, i.e. some user-mode API or native API has been hooked or some other technique has been used to attempt to hide processes.

Another method would be to enumerate all open handles in *csrss.exe* that are of type "*process*". This is because *csrss.exe* maintains process handles to all processes currently running in memory. With this information we can determine all process names and process-ids, which can then be compared with enumerations obtained by other techniques (as described in previous sections) in order to find any discrepancies.

There are also open handles of type "*thread*" maintained by *csrss.exe* for each running process in memory. Enumerating the thread handles as well helps us determine the parent of a thread, hence being able to determine all process-ids that currently have any threads running in memory. This enumeration of process-ids can then be compared with enumerations obtained by other techniques (as described in previous sections) in order to find any discrepancies.

Using the native API *NtQuerySystemInformation* with the sub-function *SystemHandleInformation*, we can enumerate all open handles (of all types) on a system. The retrieved information provides associated process-ids with each handle. This enumeration of process-ids can then be compared with enumerations obtained by other techniques (as described in previous sections) in order to find any discrepancies.

If a malware were to hook all of the mentioned user-mode APIs and native APIs used for enumerating memory objects, in order to consistently return manipulated results, then these techniques would fail to find the malicious hidden process. There is also the possibility of false-positives with using the combined data from multiple techniques. This could happen if a process was already enumerated by a few techniques and then exited while still being enumerated by other techniques. Such type of situations must be handled gracefully.

5.2 Scanning for memory mapped files

File mapping is the association of a file's contents with a portion of the virtual address space of a process. It is an efficient way for two or more processes on the same computer to share data, while providing synchronization between the processes. This facilitates inter process communication (IPC). Malicious processes could use file mapping in order to communicate and share data from malicious files on disk. Hence it is important for the memory scanner to enumerate mapped files within the address space of each process. Whenever a process wants to map a file on disk, it first opens the file by calling the *CreateFile* function. In order to ensure that other processes do not write to the portion of the file that is mapped, the process could open the file with exclusive access by specifying *zero* in the *fdwShareMode* parameter of *CreateFile*. The memory scanner could enumerate all open file handles by a certain process by using the native API function, *NtQuerySystemInformation* with *SystemHandleInformation* and then using another native API function, *NtQueryObject* to search for the object handle "*file*". After enumerating all open file handles, each associated file on disk could be scanned for malicious content. If any such files are found, then the associated file handles could be closed within the malicious process accessing them.

6 Pros and cons of user-mode memory scanning

Due to the virtual memory address separation of user-mode and kernel-mode, the kernel-mode address space is protected from read or writes access by any user-mode component or thread. Whenever a user-mode API requests certain system information, it is serviced via a kernel-mode service, wherein, a context switch of the thread from user-mode to kernel-mode happens. The desired information is retrieved from various kernel structures or objects and transferred back to the calling user-mode API. When in user-mode, the thread context is switched back to user-mode (less privileged). Any malware that is either using a kernel-mode component, or operating fully in kernel-mode itself, has complete access to all kernel structures as well as control transfers from user-mode to kernel-mode. Hence, such malware could manipulate the retrieved information before transferring it back to user-mode consequently hiding its presence from the user-mode memory scanner. Malware could also disallow termination of malicious processes in memory and/or disallow deletion/disinfection of malicious files on disk, by using kernel-mode components. In order to combat such malware requires implementing a kernel-mode memory scanner. In particular, user-mode memory scan can be bypassed by hooking user-mode APIs and/or native APIs, hooking of kernel structures such as system service dispatch table (SSDT) or interrupt descriptor table (IDT), import address table (IAT) & export address table (EAT) hooking, SYSENTER hook, inline function hooks, driver hooks (also called IRP—IO Request Packet hooks), and hooking the memory manager. More advanced methods available to kernel malware are filter driver insertion and DKOM (Direct Kernel Object Manipulation). All these techniques are discussed in [15]. If the memory scanner were to be implemented in kernel-mode, it is less susceptible to being thwarted, as integrity of structures and APIs can be checked or monitored.

A user-mode memory scanner also has limitations enforced by the operating system depending on the privileges of the currently logged-on user running the application. If the application were to be run by a limited user with no administrative privileges, it would fail to enumerate several system processes and threads, as well as fail to read memory pages of processes.

On the other hand, a kernel-mode memory scanner (implemented as a kernel-mode driver) is complex to implement, debug and deploy. Compatibility issues with different versions of Windows NT based operating systems need to be taken into consideration as implementation details may significantly vary. For example, the introduction of kernel patch protection or "*PatchGuard*" in 64-bit versions of the Windows OS, as well as several design features to enforce security measures in Windows Vista [7], makes driver development for memory scanning quite tedious and complex [6]. Also, the stability of such a kernel-mode application depends on a variety of factors such as software and/or hardware configuration. Any faulty implementation could lead to system wide crashes such as reboots, blue screen of death (BSoD), or system freezes. Hence, extreme care must be taken while implementing a kernel-mode memory scanner. Also note that 32-bit driver support has been removed in 64-bit Windows Vista which would require a complete port of the memory scanner if written as a 32-bit kernel-mode driver.

Although a user-mode memory scanner has its limitations, it is much easier to implement, debug and deploy than its kernel-mode counterpart. It can be reliably operated without risk of causing a system wide crash. The worst case scenario could only be a single application crash. Also, the compatibility issues with different versions of Windows NT based operating systems (such as Windows XP 64-bit, Windows Vista 32-bit & 64-bit) can be easily overcome.

Both approaches have their pros and cons. In practice it is best to implement a memory scanner in both user-mode and kernel-mode. By comparing the results from both techniques (a cross-view diff approach), one could reveal any hidden process, files or registry entries determining the compromised state of a machine.

7 Conclusion

The essential components of a user-mode memory scanner for Windows NT based operating systems were presented. This involved enumerating a wide variety of active memory components; such as processes, process heaps, threads, loaded modules, loaded drivers, services, etc. The idea was to rely on the abundance of redundant information available via various internal structures active in memory, and extract this information. This information can be queried to compare results from different sources in order to detect any possible system compromise. Techniques to terminate malicious processes in memory and restoring read access to locked files on disk were also discussed. The advantages and disadvantages of implementing a memory scanner in user-mode were also discussed.

References

1. 90210: Process Hide. http://vx.netlux.org/vx.php?id=ep12 (2004)
2. Barwise, M.: Quantity of malware booms. http://www.heise-security.co.uk/news/101764 (2008)
3. Bassov, A.: Entering the kernel without a driver and getting interrupt information from APIC. http://www.codeproject.com/KB/system/soviet_kernel_hack.aspx?df=100&forumid=209018&exp=0&select=1480766&tid=1480766 (2005)
4. Crazylord: Playing with Windows /dev/(k)mem. http://www.fsl.cs.sunysb.edu/~dquigley/files/vista_security/p59-0x10_Playing_with_Windows_dev(k)mem.txt (2002)
5. Diamond CS: Advanced Process Termination. http://www.diamondcs.com.au/advancedseries/processkilltechniques.php (2005)
6. Evers, J.: Microsoft coughs up Vista APIs.http://news.zdnet.co.uk/security/0,1000000189,39285232,00.htm (2006)
7. Evers, J.: Windows PatchGuard hindering security. http://news.zdnet.co.uk/software/0,1000000121,39280753,00.htm (2006)
8. Fedotov, A.: Enumerating Windows Processes. http://www.alexfedotov.com/articles/enumproc.asp (2006)
9. Fedotov, A.: Processes and Threads Sample. http://www.alexfedotov.com/samples/threads.asp (2006)
10. Silberman, P., C.H.A.O.S.: FUTo. http://www.uninformed.org/?v=3&a=7&t=sumry (2005)
11. Ionescu, A.: Subverting Windows 2003 SP1 Kernel Integrity Protection. http://www.alex-ionescu.com/recon2k6.pdf (2006)
12. Ionescu, A.: Why protected processes are a bad idea. http://www.alex-ionescu.com/?p=34 (2007)
13. Kath, R.: The Virtual-Memory Manager in Windows NT. http://msdn2.microsoft.com/en-us/library/ms810616.aspx (1992)
14. Kerbs, B.: Microsoft releases Windows Malware stats. Retrieved 16 February, 2008, from http://blog.washingtonpost.com/securityfix/2006/06/microsoft_releases_malware_sta.html (2006)
15. Kumar, E.: Battle with the Unseen—Understanding Rootkits on Windows. http://ericuday.googlepages.com/EKumar_Rootkits.pdf (2006)
16. Microsoft MSDN documentation: Memory Limits for Windows Releases. http://msdn2.microsoft.com/en-us/library/aa366778.aspx (2008)
17. Microsoft KB Article: How to Obtain a Handle to Any Process with SeDebugPrivilege, Q131065. http://support.microsoft.com/kb/131065 (2006)
18. Microsoft KB Article: Comparison of 32-bit and 64-bit memory architecture. http://support.microsoft.com/?kbid=294418 (2007)
19. Nebbet, G.: Read kernel memory from user-mode using Kmem. http://catch22.net/source/ (2004)
20. Restrepo, T.: Enumerating 16-bit Processes under WinNT. http://www.winterdom.com/dev/ptk/16bitproc.html (1998)

21. Russinovich, M. (2006). NT's "\dev\kmem\". http://technet.microsoft.com/en-us/sysinternals/bb897446.aspx
22. Russinovich, M., Solomon, D.: Virtual to Physical address translation 32-bit and 64-bit (IA64 & x64), http://book.itzero.com/read/microsoft/0507/microsoft.press.microsoft.windows.internals.fourth.edition.dec.2004.internal.fixed.ebook-ddu_html/0735619174/ch07lev1sec5.html (2004)
23. Sanders, B.: Address space implementations in various 64 bit processors from Intel and AMD. http://members.shaw.ca/bsanders/WindowsGeneralWeb/RAMVirtualMemoryPageFileEtc.htm (2007)
24. Schreiber, S.: Interfacing the native API in Windows 2000. http://www.informit.com/articles/article.aspx?p=22442&seqNum=5 (2001)
25. Skoudis, E.: 10 emerging malware trends for 2007. http://searchfinancialsecurity.techtarget.com/tip/0,289483,sid185_gci1294544,00.html (2007)
26. Solomon, D., Russinovich, M.: Microsoft® Windows® Internals. Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000, pp. 420–428. Microsoft Press. ISBN: 0735619174 (2004)
27. Ször, P.: Memory scanning under WinNT. http://www.peterszor.com/memscannt.pdf (1999)
28. Talekar, N.: Faster Method to Enumerate Heaps on Windows. http://securityxploded.com/enumheaps.php (2007)
29. Tan, C.: Defeating kernel native API hookers by direct Service Dispatch Table restoration. http://www.security.org.sg/code/sdtrestore.html (2004)
30. Viscarola, P.: Nt vs. Zw—Clearing confusion on the native API. http://www.osronline.com/article.cfm?id=257 (2003)
31. Vizjereij, X.: Module walker. http://www.runeforge.net/node/142 (2007)