APPLYING USER-MODE MEMORY SCANNING ON WINDOWS NT

Eric Uday Kumar
Authentium Inc., 7121 Fairway Drive, Suite 102,
Palm Beach Gardens, Florida 33406, USA

Tel +1 561 575 3200 ext. 4306
Email ekumar@authentium.com

ABSTRACT

Memory-resident malware and malware persistent over reboots have for a long time been the most challenging to detect and deactivate. Such types of malware can conceal their presence (defensive techniques), thwart termination or removal (armouring techniques), lower system security and terminate or suspend security applications (offensive techniques). Memory scanning plays a vital role in detecting and deactivating these types of malware.

Implementing a memory scanner for *Windows NT*-based systems is particularly challenging due to the complexity of the executing environment. While memory scanning can be implemented in both kernel mode and user mode, this content is confined to user-mode memory scanning, for 32-bit and 64-bit *Windows NT*-based systems. Essentially this means scanning the virtual address space of each process in memory, as well as its associated objects in memory and on physical disk, as seen from user mode. Here, we will discuss examples from real-world malware scenarios that demonstrate a few anti-detection and anti-disinfection techniques, and find how memory scanning can be applied to overcome some of them. Certain techniques to detect hidden processes from user mode will also be presented. The limitations of user-mode memory scanning will also be discussed.

INTRODUCTION

The plague of computer malware targeting *Microsoft*'s *Windows* operating system has been growing rapidly over the past few years [1]. The vast majority of malware created and deployed today targets 32-bit *Windows*. Although there are not yet many malicious programs created specifically to target 64-bit *Windows*, much of the existing malware for 32-bit *Windows* will work on 64-bit *Windows* due to the underlying WoW64 subsystem. Motivated by monetary gain, malware authors are adopting a variety of anti-detection and anti-disinfection strategies in creating malware that:

- Is stealthier and more subtle.
- Uses automated malware generation tools (such as code obfuscators).
- Uses customized server-side encryption or packing.
- Monitors its associated components such as files on disk, registry entries, spawned processes, etc. and disallows access to any external program.
- Escalates and executes with highest privileges such as a SYSTEM process or native service in order to thwart termination.
- Executes disguised as a legitimate process (such as winlogon.exe, explorer.exe, services.exe, lsass.exe, etc.) by injecting malicious code directly (such as a dynamic link library or position-independent code).
- Performs in-memory infection by patching user-mode APIs, native APIs, or kernel data structures in order to hide its associated processes in memory and/or files on disk.
- Applies offensive techniques such as lowering system security and/or terminating security applications.

A piece of malware can implement several of the techniques mentioned above by remaining memory resident. This renders the task of disinfecting and restoring the machine to a clean state significantly harder. Hence, it is important for an anti-malware system to implement both user-mode and kernel-mode memory scanning. While the user-mode component operates purely in user mode and can only access the user space virtual memory with the privileges of the currently logged-on user, its kernel-mode counterpart operates in kernel mode and can access the complete user space and kernel space virtual memory with the highest privileges. The discussion here is confined to applying user-mode memory scanning to detect and deactivate memory-resident malware. This is an application of the idea and techniques described in [2].

The following sections discuss related work and the basic idea behind user-mode memory scanning, as described in [2]. This is followed by a brief discussion about the portability of 'memory objects enumeration techniques' to 64-bit *Windows* and *Vista*. Following this is a description of the application of these techniques in order to detect and deactivate real-world memory-resident malware threats. The paper concludes with a brief discussion about the limitations of user-mode memory scanning.

RELATED WORK

Memory scanning on *Windows NT*-based systems in both user mode and kernel mode is described in [3]. The paper provides a good overview of the topic, while detailing several issues with real-world malware detection and disinfection. A detailed discussion about implementing user-mode memory scanning on *Windows NT* is provided in [2]. The paper proposes an approach of retrieving as much redundant information about various memory objects as seen from user mode, in order to detect the compromised state of a machine. It also discusses certain user-mode disinfection techniques.

MEMORY OBJECTS

The idea is to leverage the abundance of redundant information available about various memory objects from user mode, via several Win32 and native APIs [2]. The memory objects of interest are processes, process heaps, threads, handles, device drivers and loaded modules (DLLs – Dynamic Link Libraries). These memory objects are associated with physical objects on disk such as files and registry hive entries. Co-relating this information will help us to 'see' the current state of a machine's memory, and if any, the compromised state of the machine. The co-related information can also be analysed to make inferences about any hidden memory objects (such as hidden processes).

A piece of memory-resident malware is an active malicious process in memory that constitutes at least one executing thread. Such malware can execute either as an independent process or from within another legitimate process. While executing independently, it could be either single-threaded or multi-threaded, or even spawn several child processes. Executing from another legitimate process is possible by injecting its malicious code in the form of a DLL or position-independent code. DLL injection techniques are achieved via *Windows* hooks, the registry, injecting a new thread into the address space of the target process, or hijacking an existing thread within the target process. A discussion of these techniques can be found in [4]. Memory-resident malware can also allocate shared memory in the form of memory-mapped files and then access that shared memory via injected code within a legitimate process. Also, memory-resident malware may or may not have associated files on disk and/or registry entries.

RELATING MEMORY OBJECTS

From the previous section, we infer the following relations that will prove helpful while scanning memory for malware:

- Enumerating processes (or process-ids), device drivers and services will lead us to their associated files on disk.
- Enumerating processes (or process-ids) can lead us to any memory-mapped files on disk. This is not the case when a memory-mapped file is created via the pagefile.
- Enumerating processes (or process-ids) will lead us to all loaded modules by each process (including the module associated with the process itself). The loaded modules' information will in turn lead us to their associated files on disk, as well as leading us to retrieve the actual memory image (virtual address space) of each module.
- Enumerating processes (or process-ids) will lead us to all thread information within each process. Alternatively, enumerating threads (or thread-ids) will lead us to their parent-process-ids (i.e. the identifiers of the processes that created the threads).
- Enumerating processes (or process-ids) will lead us to all heap information within each process. The heap information will in turn lead us to retrieve allocated heap memory.
- The information about files on disk that are associated with memory objects (such as processes and loaded modules) can lead us to any associated registry entries.
- Enumerating system-wide handle information can lead us to information about processes (or process-ids) and threads (or thread-ids), as well as files on disk, registry entries and ports being accessed.

Based on the established relations between various memory objects, we can quickly summarize an approach to user-mode memory scanning [2]:

- Enumerate as many objects in memory as possible, resulting in redundant information that can potentially reveal hidden components.
- Scan the memory image (virtual address space) of each loaded module, the allocated heap memory of suspicious processes and associated files on disk.
- Terminate or suspend malicious processes if any. Disinfect associated files and registry entries. Use a native application to clean up any unresolved infections on the next reboot.

ENUMERATING MEMORY OBJECTS

The following is a brief overview of the various Win32 and native API functions that can be used to enumerate objects in memory [2]:

- PSAPI functions (implemented in psapi.dll) can be used to enumerate processes, loaded modules and device drivers. The functions of interest are: EnumProcesses, EnumProcessModules, EnumProcessModulesEx (for 64-bit), GetModuleFileNameEx, GetModuleInformation, EnumDeviceDrivers and GetDeviceDriverFileName.
- Tool Help functions (implemented in kernel32.dll) can be used to enumerate processes, threads, heaps and loaded modules. Process information also holds parent-process-id information. The functions of interest are: CreateToolhelp32Snapshot, Process32First, Process32Next, Thread32First, Thread32Next, Module32First, Module32Next, Heap32ListFirst, Heap32ListNext, Heap32First and Heap32Next. Allocated heap memory can be read using Toolhelp32ReadProcessMemory. QueryFullProcessImageName can be used to obtain the complete process name (i.e. path and name of associated file on disk) in *Windows Vista* and *Windows Server 2008*.
- The ADVAPI function (implemented in advapi32.dll) can be used to enumerate services installed via the SCM (Service Control Manager). The function of interest is EnumServiceStatusEx.
- Performance counter functions (implemented in pdh.dll) can be used to enumerate processes and threads. The functions of interest are: PdhEnumObjectItems, PdhEnumObjects, PdhLookupPerfNameByIndex, PdhMakeCounterPath, PdhOpenQuery, PdhAddCounter, PdhCollectQueryData, PdhGetRawCounterArray and PdhCloseQuery.
- The NTQuerySystemInformation native API (implemented in ntdll.dll) can be used to enumerate processes, threads, handles and device drivers. Process information also holds parent-process-id information. Handle name and type information can be retrieved using the NtQueryObject native API.
- The NtQueryInformationProcess native API (implemented in ntdll.dll) can be used to enumerate loaded modules within a process, as well as allowing retrieval of its parent-process-id.
- The NtQueryInformationThread native API (implemented in ntdll.dll) can be used to enumerate threads within a process.
- The debug native APIs (implemented in ntdll.dll) can be used to enumerate loaded modules and heaps within a process. The functions of interest are: RtlQueryProcessDebugInformation, RtlCreateQueryDebugBuffer and RtlDestroyQueryDebugBuffer.
- Committed virtual memory pages within a process or loaded module can be enumerated via VirtualQueryEx

(or the NtQueryVirtualMemory native API) and read using ReadProcessMemory. Note that the NtSystemDebugControl native API with control code 10 can be used to dump the contents of physical memory on some versions of *Windows NT* [5].

DEALING WITH MALWARE IN MEMORY

In this section we will discuss the application of the techniques discussed in the previous section. A few real-world malware scenarios are used to demonstrate the effectiveness and limitations of these techniques.

Legitimate system process infection

Infecting a legitimate user-mode system process (such as explorer.exe, winlogon.exe, services.exe, etc.) proves advantageous to malware that intends to remain memory resident for a long period of time. In order to accomplish this some malware will allocate shared memory (via a memory-mapped file object) and then inject code within the legitimate process that accesses this shared memory [6]. Shared memory can be allocated using the CreateFileMapping and MapViewOfFile API calls. Note that a memory-mapped file does not have to be with a file on disk. A file-mapping object backed by physical memory can be created by specifying an invalid handle value to the CreateFileMapping function. This file-mapping object can be shared by name. If the malware always uses the same memory-mapped object name, then a user-mode memory scanner can scan for that object name within the virtual address space of each process in order to detect an infection. Nonetheless, the malware can randomly generate an object name at run-time.

Once such a shared memory is created, the malware writes its malicious code to the allocated memory. It then obtains the base address of its target legitimate process and gets a handle to it with read-write access. It then attempts to find a read-write section within the target legitimate process (i.e. a section in which data is already initialized) with some free space. Once such a section is found, the malware will inject a piece of loader code in the available space and then install a hook on one of the imported APIs. The purpose of the hook is to transfer control to the injected code. The purpose of the injected code is to get access to the previously allocated shared memory (via the memory-mapped file technique) from within the infected legitimate process. This is done using the OpenFileMapping and MapViewOfFile functions.

A user-mode memory scanner could scan all sections within the memory image of each process in order to detect the piece of injected loader code. In order to disinfect, the injected code must be overwritten and the hooked API properly unhooked.

Injected code in privileged processes

Popular code injection techniques in user mode use *Windows* hooks or a combination of the VirtualAllocEx (or the NtAllocateVirtualMemory native API), WriteProcessMemory, LoadLibrary and CreateRemoteThread APIs [7]. Malicious programs often use these techniques to inject their code into privileged processes such as lsass.exe, csrss.exe and winlogon.exe. In order to do so, they usually escalate their privileges (if running as a normal process) by adjusting their token privileges so that debugging is allowed. Since on *Windows XP SP2* and *Vista* CreateRemoteThread does not function properly, certain pieces of malware tend to substitute this with RtlCreatUserThread on *Windows XP SP2*, and NtCreateThreadEx on *Vista* [8]. Both of these functions are exported by ntdll.dll. As an alternative to this, certain pieces of malware will install a temporary service (via the Service Control Manager – SCM) and have that carry out the remote thread execution using the single function CreateRemoteThread.

In *Windows NT4* and up, native *Windows* threads support APC (Asynchronous Procedure Call) queuing. The APC queue gets processed whenever a thread enters an 'alert-able wait state' (which is attained when one of the SleepEx, WaitForSingleObjectEx, etc. functions is called). The QueueUserAPC function allows your own functions to be inserted into this queue. Certain pieces of malware will inject their malicious code within a legitimate process and use the QueueUserAPC function to queue a thread for its execution.

A user-mode memory scanner could scan the virtual address space of each process and loaded module in order to detect any injected code. One way of completely unmapping an injected DLL from a process is to decrement the 'dll-load-count' value, which is available from the PEB (Process Environment Block) of a process [9]. The scanner could also enumerate all installed services (using the EnumServicesStatusEx function exported by advapi32.dll) and scan their associated files on disk. If a malicious service is identified, we can use the ControlService or ControlServiceEx functions to send the 'stop' or 'pause' control code to it. If the target service has any dependent services, then we must first enumerate those using the EnumDependentServices function and try to stop/pause them. The status of the target service can be checked using the QueryServiceStatusEx function in order to find out whether our stop/pause action was successful. Alternatively, the ChangeServiceConfig function can be used to disable the target service. Finally, the DeleteService function can be used to delete the target service from the SCM. If the target service's security descriptor prevents access to it, then we can use the SetServiceObjectSecurity function to set its security descriptor in order to enable access. Nonetheless, malware could hook the user-mode API used to enumerate services (EnumServicesStatusEx) in order to conceal its presence.

Enumerating handles (system wide)

When used with the SystemHandleInformation sub-class, the NtQuerySystemInformation native API can enumerate all open handles system wide. Each handle can be identified by its handle information structure. Each handle information structure contains a member which is the process identifier of the process to which the handle belongs. Also, each handle is associated with 'type' information that specifies what type of handle it is. We are mostly interested in the 'File', 'Process' and 'Thread'-type handle information. Handle type information can be obtained using the NtQueryObject native API.

For the 'File'-type handles, associated filename information can be obtained by using the NtQueryInformationFile native API. For 'Process'-type handles, associated process information (such as process-id, parent-process-id, process-name, image-base-address, list of loaded modules, etc.) can be obtained by using the

NtQueryInformationProcess native API and reading the PEB. For 'Thread'-type handles, associated thread information (such as thread-id, parent-process-id, start address, priority, etc.) can be obtained by using the NtQueryInformationThread API. This can also be used to obtain the base address of the TEB (Thread Environment Block), which in turn contains a pointer to the base address of the PEB of the parent-process. For all other handle types (such as directory, semaphore, port, event, symboliclink, etc.), the handle name information can be obtained by using the NtQueryObject native API.

Hence this determines another way to enumerate processes and threads on a system. We can first enumerate all open handles system wide and then for each 'Process'-type handle obtain complete process information, and for each 'Thread'-type handle obtain complete thread information, as well as the process-id of the process it belongs to.

Note that csrss.exe maintains 'Process' and 'Thread'-type handles for each process currently executing on a system, with the exception of itself and smss.exe.

Detecting hidden processes

A few user-mode techniques can be used to detect processes hidden by kernel-mode rootkits. This was tested against the infamous Fu and FuTo rootkits. Both of these rootkits are publicly available for download (in source and binary) from the 'rootkit dot com' website. These rootkits use DKOM (Direct Kernel Object Manipulation) techniques to hide processes on a system. What this basically does is unlink the target process from the double-linked list of processes maintained as a structure within the kernel. Whenever a native or user-mode API requests a list of currently executing processes on the system, this internal structure is eventually referenced by kernel services in order to provide the information.

One of the techniques implemented by some rootkit detection tools is the 'brute-force PID technique'. This uses the OpenProcess API call on a range of valid process-ids (0x0000 through 0x41DC), keeping track of each successful call. The enumerated process-ids list is compared against another enumeration using any of the user-mode APIs (such as PSAPI or ToolHelp). Any discrepancies will identify the hidden process [10].

We have identified that a process hidden using the Fu rootkit can also be detected by enumerating open handles system wide from user mode. We also noticed that the hidden process has visible 'Thread'-type handles in csrss.exe and svchost.exe. With the process-id information we use the NtQueryInformationProcess native API in order to access its PEB and find all loaded modules. The Fu rootkit's driver file (msdirectx.sys) is not hidden by default and can be enumerated via the NtQuerySystemInformation native API with SystemModuleInformation as the sub-class or by using the EnumDeviceDrivers function (in psapi.dll). If the driver file were to be hidden using the Fu rootkit, its presence could still be seen via the EnumServiceStatusEx function (in advapi32.dll), since the driver is registered as a service via the SCM (Service Control Manager).

The FuTo rootkit is an enhanced version of the Fu rootkit that attempts to bypass the 'brute-force PID' detection technique. In order to hide processes, it further manipulates the internal structures that the OpenProcess API call-chain uses to validate a process-id. We have identified that even though it is able to bypass the 'brute-force PID' technique and system-wide handle enumeration, the hidden process still possesses visible 'Thread'-type handles in csrss.exe. Using this information we can still detect processes hidden by the FuTo rootkit from user mode.

Nonetheless, malware could use a kernel-mode driver to hide its open handles (by returning manipulated results to the calling native API) or hook the native API directly from user mode and bypass the technique described above. Also, some malware uses kernel-mode techniques to load its malicious driver file without touching the registry (i.e. without registering with the SCM) [11].

Detecting Storm trojan injected code

The infamous 'Storm trojan' (a.k.a. CME-711, Nuwar, Peacomm, Tibs, Zhelatin) uses stealth techniques in order to remain undetected for as long as possible. From using anti-debugging and anti-emulation techniques to polymorphic encryptors and server-side packing, the Storm trojan presents a challenge for anti-malware systems. Since its first appearance in January 2007, the Storm trojan has used a myriad of tricks to drop and execute its payload.

Some of the earlier versions injected their malicious payload into the legitimate services.exe system process. This was done by dropping a malicious driver program and registering it as a service. This driver program was instrumental in executing the trojan's payload. The payload was an embedded executable within the driver program. The driver employed stealth techniques in order to execute the payload. The payload was injected from kernel space into the user space of services.exe and scheduled for execution by queuing an APC (Asynchronous Procedure Call) for it. For this reason, there would be no 'visible' process executing the payload if we were to use tools such as *Windows Task Manager* or *Process Explorer*. A thread injected into services.exe was carrying out the trojan's malicious activity [12].

Later versions of the Storm trojan used several layers of packers, both for the dropper itself and the dropped code (which is the injected DLL). Using signature-based detection makes it difficult to detect the polymorphic packed binaries on disk. However, when the dropper loads itself in memory, it is in its unpacked state while the embedded DLL (code to be injected) is still packed within it. When it eventually injects this embedded DLL into services.exe, the injected code is in its unpacked state. By doing an easy match of bytes within services.exe, the injected code can be detected reliably, hence detecting the compromised state of a machine. This is done by reading the heap memory of services.exe and scanning it to find the injected code [13].

The more recent versions of the Storm trojan unpack the embedded DLL and load it within its own process (instead of injecting it into services.exe) [13]. This simply changes where we look for the injected code. Reading the virtual address space of each process and scanning to find the injected code will detect the infection in memory. Hence, by detecting the sequence of bytes that are constant across multiple generations of the Storm trojan, we can detect all variants generically in memory. It then doesn't matter if newer variants use different packers or encryption techniques, all those only prevent on-disk detection.

Detecting the poison (WSN)Poem

WSNPoem (a.k.a. Infostealer.Banker) is a password-stealing trojan. It injects code into winlogon.exe and uses that to drop subsequent malicious DLLs onto the system. It also injects code into svchost.exe and uses that to download additional malicious components from a malicious server. It then enumerates all processes in memory and injects them as well (except for csrss.exe). The injected code prevents deletion of its malicious files on disk and its registry entries. In order to ensure injection of its code into any newly created processes it hooks the NtCreateThread, LdrLoadDll and LdrGetProcedureAddress native APIs [14]. A user-mode memory scanner could easily detect the injected code by enumerating processes, reading their virtual address spaces (or heaps) and scanning for the injected code. The disinfection part is a bit trickier and would require the hooked APIs to be unhooked properly and a reboot for clean up.

Cracking Kraken

The Kraken trojan (a.k.a. Bobax), which is basically a spam-bot, carries its spam component as a packed embedded executable. It unpacks this onto its own heap. A user-mode memory scanner could detect the spam component by enumerating heaps within each process and looking for the unpacked PE header. In order to remain less traceable in memory, a newer variant of Kraken wipes out the unpacked code's PE header. A user-mode memory scanner could still detect this if it knows what to look for in the enumerated heaps [15]. A more recent variant goes a step further in trying to remain stealthier in memory and throwing off memory scanners. It decrypts pieces of code and data dynamically on an as-needed basis onto its own heap. After using the decrypted code/data, it frees the information. This throws off memory scanners that depend on finding a generic memory signature, since it exposes only limited parts of its memory contents for a short interval of time [16]. Such evasion would be difficult to detect with user-mode techniques.

Disinfection using a native application

A native application is a program which uses only the native API. A flag in the executable header marks it as a native application. Such pieces of software are executed while *Windows* is booting. An example of a native application is the autocheck utility which checks the hard disk for errors during boot time. Another example is the Win32 operating environment server, csrss.exe (Client-Server Runtime Subsystem). The native API sits below the Win32 sub-system and is accessible via ntdll.dll. Other sub-systems are not available at the time native applications are executed [17].

Using a native application for disinfection can prove helpful when dealing with hard-to-disinfect malware. For example, malware that infects system-critical legitimate processes such as winlogon.exe or services.exe, fails to terminate or suspend, starts up even in safe-mode after a reboot, etc. A native application can be used as a clean-up routine at boot time in order to disinfect infected files on disk, delete malicious files on disk, or clean-up malicious registry entries [3].

A native application's start-up can be defined in the registry auto-run location: [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager] by setting the value of the key 'BootExecute' [3]. A few pointers that might come in handy while implementing a native application are listed below:

- For native applications, the entry point is NtProcessStartup (much like main or winmain for Win32 applications). The source at [17] demonstrates an example native application that displays a string at boot time.
- The source at [18] demonstrates the use of a native application for file-IO.
- In order to write to registry at boot time, NtInitializeRegistry can be used to initialize the registry (smss.exe would do). This loads the registry hives and marks the registry as writable [19]. The sources at [19, 20] demonstrate the use of a native application to read and write to the system registry.
- The source at [21] demonstrates the use of a native API to create an application that can execute a native application.

LIMITATIONS

One of the limitations for a user-mode memory scanner is the privilege level of the currently logged-on user running the application. In this case, a lack of sufficient privileges would cause the scanner to fail to enumerate several system processes and threads, as well as fail to read the memory pages of processes. The following sections will discuss other limiting factors for a user-mode memory scanner.

Hook-ups lead to mess-ups

User-mode memory scanning is susceptible to hooking techniques implemented on user-mode APIs and/or native APIs. If all of the APIs used for enumerating memory objects are hooked to provide manipulated results, then the scanner will fail to identify such objects in memory for it to begin scanning their virtual address space.

This limitation can be overcome to some extent by implementing user-mode hook checks. For example, the scanner could check for in-line hooks among the Win32 and native APIs that it imports. It could also check its own IAT (Import Address Table) for any installed hooks. In-line hooks can be checked by reading the memory image of dynamically loaded modules, and checking the first few bytes at the entry point of imported functions for any relative jmp instructions. Another method is to enumerate all loaded modules (or DLLs) within a process via its PEB and compare significant bytes of their memory image with the on-disk image. Nonetheless, malware could intercept reads to memory images and conceal the in-line hooks it has placed.

PEB hooking

Some malware uses the PEB-hooking technique to hook the user-mode APIs imported by a process. This is an in-memory infection technique wherein data within the PEB of a process is modified. The PEB of a process contains a structure member that maintains a double-linked list of all modules loaded by the process. The technique relies on manipulating the following data within this linked list: base address, entry point and size of image of the module whose exported functions the malware is interested in hooking [22, 23]. To accomplish this, the malware uses any of the user-mode code injection techniques [7] and injects a DLL (we'll call this the fake DLL) within each process in memory (and any new

process that is created) that loads the DLL which the malware is interested in hooking (we'll call this the real DLL). The injected fake DLL resembles the real DLL and the base address, entry point and size of image of the real DLL are exchanged. It then modifies the IAT of all modules (expect those of the fake and real DLLs) that have exports from the real DLL to point to corresponding functions implemented in the fake DLL.

Using this technique, malware could also conceal the presence of its injected fake DLL from user-mode enumerations. One of the ways to deal with this is to implement a kernel-mode component that enumerates processes and loaded modules from kernel space.

User-mode rootkits – Hacker Defender

As mentioned previously, user-mode memory scanning is susceptible to user-mode or native API hooking. An example of this is the user-mode rootkit Hacker Defender. It hooks several Win32 and native API functions, allowing it to hide files, processes, handles, services, registry entries, etc. Hacker Defender is publicly available for download (in source and binary) from the 'rootkit dot com' website. It injects code into every running process (as well as any new processes created) by allocating memory on the heap of the remote process and writing the payload directly into it. It then installs inline hooks (by patching imported function entry points with relative jmp instructions) that point to the payload. The rootkit is able to hide processes and modules by hooking the NtQuerySystemInformation native API. It also hooks the LdrLoadDll native API so that if a process dynamically loads a DLL whose exported functions are hooked, the rootkit can still patch those functions in memory [24]. It also drops a kernel-mode component in order to help its user-mode component to hide handles (because accessing certain handles directly from user mode can cause an application to hang indefinitely). Detecting this threat again requires proper hook-checking routines or a kernel-mode component.

Kernel malware

Malware that operates in kernel mode or at least uses some kernel-mode component to aid its malicious activity could manipulate results returned to user-mode APIs, consequently hiding its presence from the user-mode memory scanner. Such malware could also disallow the termination of malicious processes in memory and/or disallow deletion/disinfection of malicious files on disk. These pieces of malware are able to do so by hooking various kernel-mode system services, or hooking internal tables such as SSDT (System Service Dispatch Table) and IDT (Interrupt Descriptor Table), or by hooking the memory manager, or directly manipulating kernel structures. They could also place hooks such as the SYSENTER hook, inline function hooks and driver hooks [4]. One of the ways to combat such malware is by implementing a kernel-mode memory scanner. Such a scanner is less susceptible to being thwarted, as the integrity of structures and APIs can be checked or monitored from within the kernel.

Misleading memory reads

A proof-of-concept kernel-mode rootkit called Shadow Walker demonstrates the technique of *Windows* memory manager subversion. It is publicly available for download (in source and binary) from the 'rootkit dot com' website. It conceals itself in memory by modifying the page table implemented by the memory manager for virtual memory management [25]. It takes advantage of the *Intel* 32-bit CPU architecture that has two address-translation caches: one for code pages called ITLB (Instruction Translation Look-aside Buffer) and the other for data pages called DTLB (Data Translation Look-aside Buffer). The rootkit intercepts the page fault handler (int 0x0e) and marks the pages of code it wants to conceal in the page table as 'missing pages'. Following this, any attempts to access those pages for read, write or execute lead to page faults, resulting in the page fault handler being invoked. The interception routine for the page fault handler checks requested access to the page. If the request was for execution, the handler loads the ITLB with an original code page. If the request was to access data, the handler loads the DTLB with a fake page. In this way, the rootkit is able to conceal memory pages of the process it is trying to hide.

Using such a rootkit, malware could easily conceal its presence in memory. Again, dealing with this requires a kernel-mode component of the memory scanner to be implemented.

The infamous Rustock

The Rustock rootkit employs complex kernel-mode techniques in order to hide its presence and malicious activity. It extracts an embedded DLL from within and injects it into winlogon.exe. The extracted DLL never exists on disk and is only present in memory. This DLL is instrumental in spreading spam from the infected computer whilst in the disguise of winlogon.exe [26]. A user-mode memory scanner could enumerate and scan all modules loaded by winlogon.exe in order to detect the injected DLL. But the authors of the Rustock rootkit hide enumeration of the injected DLL via a small piece of code injected directly into the address space of winlogon.exe. This injected code protects the memory range occupied by the injected DLL and itself, by placing hooks on the NtReadVirtualMemory, NtWriteVirtualMemory and NtProtectVirtualMemory native APIs. The injected DLL code (spam component) is executed by user-mode threads in the context of winlogon.exe. These threads are further hidden from the *Windows* API by placing hooks on the NtQuerySystemInformation, NtOpenThread and NtTerminateThread native APIs. The rootkit also hides references to its thread handles in winlogon.exe and csrss.exe by placing a hook on the NtQuerySystemInformation native API.

Detecting this threat requires a kernel-mode component to be implemented that can bypass native API hooks and detect the hidden thread handles, threads and injected code in winlogon.exe.

Pushy Pushdo and vanishing Vanquish

The Pushdo (a.k.a. Cutwail, Pandex) family of trojans use code injection and rootkit techniques to carry out their malicious activities. On an infected machine, the main process spawns the default browser application as a child process using the CreateProcess API. The child process is spawned in a suspended state so the trojan can allocate memory on its heap (using VirtualAllocEx), and then inject malicious code (using WriteProcessMemory). It then uses the

CreateRemoteThread API to execute the injected code [27]. A user-mode memory scanner could easily have enumerated all processes and scanned their virtual address space to detect the injected code within the spawned browser application. But the trojan uses a kernel-mode driver to hide the spawned process from being enumerated.

The Vanquish rootkit (which is publicly available in both binary and source form, from the 'rootkit dot com' website) applies a slightly different injection technique. It first suspends a thread in the target process (using the SuspendThread API – this is the hijacked thread), allocates memory on the heap of the target process (using the VirtualAllocEx API), writes its malicious code in the allocated memory of the target process (using the WriteProcessMemory API), and then sets the hijacked thread's context after modifying the instruction pointer (EIP register) using the SetThreadContext API to point to its injected code. It then resumes the hijacked thread using the ResumeThread API call [28]. Again, a user-mode memory scanner could easily have enumerated and scanned heaps within a process to detect the injected code, but the rootkit uses in-line hooks on several user-mode APIs to thwart this.

Again, one way of detecting such types of threat is by implementing a kernel-mode component that will bypass any user-mode and native API hooks, and be able to enumerate and scan heaps within each process.

THE BRIGHT SIDE

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 operated reliably 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 and 64-bit) are not significant.

In practice, it is best to implement a memory scanner in both user mode and kernel mode. Although a kernel-mode component is much more powerful in detecting and disinfecting malware in memory, implementing a user-mode component as well will allow for a comparison of the results from both techniques (a cross-view diff approach) in order to reveal any hidden components.

CONCLUSION

The essential components for implementing a user-mode memory scanner for *Windows NT*-based operating systems were briefed. Several real-world malware scenarios were discussed, with the view of detecting them while they are still active in memory. The limitations of user-mode memory scanning were also presented.

REFERENCES

[1] Kerbs, B. Microsoft releases Windows Malware stats. 12 June 2006. Retrieved 20 June 2008 from http://blog.washingtonpost.com/securityfix/2006/06/microsoft_releases_malware_sta.html.

[2] Kumar, E. User-mode memory scanning on 32-bit & 64-bit Windows. Proceedings of the 17th EICAR Conference. pp.389–412. Made available and retrieved 20 June 2008 from http://ericuday.googlepages.com/EICAR2008_UserMode_Memory_Scanning_3.doc.

[3] Ször, P. Memory scanning under WinNT. September 1999. Retrieved 20 June 2008 from http://www.peterszor.com/memscannt.pdf.

[4] Kumar, E. Battle with the unseen – understanding rootkits on Windows. Proceedings of the 9th AVAR International conference. 2006. pp.96–112. Made available and retrieved 20 June 2008 from http://ericuday.googlepages.com/EKumar_Rootkits.pdf.

[5] Vidstrom, A. Memory dumping with NtSystemDebugControl. 4 February 2007. Retrieved 20 June 2008 from http://www.ntsecurity.nu/onmymind/2007/2007-02-04.html.

[6] GriYo/29A. Explorer Infection. Retrieved 20 June 2008 from http://vx.netlux.org/lib/vgy02.html.

[7] Kuster, R. Three ways to inject your code into another process. 20 August 2003. Retrieved 20 June 2008 from http://www.codeproject.com/KB/threads/winspy.aspx.

[8] Ligh, M. Injecting code into privileged Win32 processes. 16 May 2007. Retrieved 20 June 2008 from http://mnin.blogspot.com/2007/05/injecting-code-into-privileged-win32.html.

[9] Talekar, N. Reference count of DLL in a process. Retrieved 20 June 2008 from http://securityxploded.com/dllrefcount.php.

[10] Silberman, P., & C.H.A.O.S. December 2005. FUTo. Retrieved 20 June 2008 from http://www.uninformed.org/?v=3&a=7&t=sumry.

[11] Brulez, N. Kernel driver backdooring. 4 May 2007. Retrieved 20 June 2008 from http://securitylabs.websense.com/content/Blogs/2730.aspx.

[12] Kumar, E. The eye of the Storm. 12 April 2007. Retrieved 20 June 2008 from http://fightmalware.blogspot.com/2007/04/eye-of-storm.html.

[13] Shevchenko, S. Piece-of-cake Storm detection. 1 April 2008. Retrieved 20 June 2008 from http://blog.threatexpert.com/2008/04/piece-of-cake-storm-detection.html.

[14] Florio, E.; Kiernan, S. InfoStealer.Banker.C. 2 April 2007. Retrieved 20 June 2008 from http://www.symantec.com/security_response/writeup.jsp?docid=2007-040208-5335-99&tabid=2.

[15] Shevchenko, S. Kraken changes tactics. 21 April 2008. Retrieved 20 June 2008 from http://blog.threatexpert.com/2008/04/kraken-changes-tactics.html.

[16] Shevchenko, S. Memory stealthiness of Kraken. 7 May 2008. Retrieved 20 June 2008 from http://blog.threatexpert.com/2008/05/memory-stealthiness-of-kraken.html.

[17] Russinovich, M. Inside native applications. 8 February 1998. Retrieved 20 June 2008 from

http://doc.sch130.nsc.ru/www.sysinternals.com/ntw2k/info/native.shtml.

[18] Going Native - Using the NT API for File I/O. The NT Insider. 15 June 1996 volume 3, issue 3. Retrieved 20 June 2008 from http://www.osronline.com/article.cfm?id=91.

[19] Rudolph, J. BootPGM. 28 Nov 2007. Retrieved 20 June 2008 from http://virtual-void.net/projects/bootpgm.

[20] Madden, D. Registry manipulation using NT native APIs. 19 June 2006. Retrieved 20 June 2008 from http://www.codeproject.com/KB/system/NtRegistry.aspx.

[21] Shedel, A. Program used to execute NT native applications. Retrieved 20 June 2008 from http://ashedel.chat.ru/source/run/nrun.c.

[22] Deroko. PEB DLL hooking – novel method to hook DLLs. 13 November 2006. Retrieved 20 June 2008 from http://www.arteam.accessroot.com/ezine/file_info/download1.php?file=ARTeam.eZine.Number2.rar.

[23] Shearer & Dreg. phook – the PEB hooker. Phrack Inc, Volume 0x0c, Issue 0x41, Phile #0x0a of 0x0f. 15 October 2007. Retrieved 20 June 2008 from http://www.phrack.org/issues.html?issue=65&id=10#article.

[24] F-Secure Rootkit Information Pages: HacDef. 9 December 2005. Retrieved 20 June 2008 from http://www.f-secure.com/v-descs/hacdef.shtml.

[25] Butler J.; Sparks S. Shadow Walker. Proceedings of Black Hat Japan 2005. October 2005. Retrieved 20 June 2008 from http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf.

[26] HolaHola a.k.a. DNY / VXHeavens. Rustock.C. 23 May 2008. Retrieved 20 June 2008 from http://www.rootkit.com/newsread.php?newsid=879.

[27] Shevchenko, S. Malware employs a memory injection technique patented by Microsoft. 19 February 2008. Retrieved 20 June 2008 from http://blog.threatexpert.com/2008/02/malware-employs-microsofts-patent-on.html.

[28] XShadow. Executing arbitrary code in a chosen process (or advanced dll injection). 20 December 2003. Retrieved 20 June 2008 from http://www.rootkit.com/newsread.php?newsid=53.