MEMORY SCANNING UNDER WINDOWS NT

Péter Ször

Symantec AntiVirus Research Center, 2500 Broadway, Suite 200, Santa Monica, CA 90404-3063, USA
Tel +1 310 453 4600 • Fax +1 310 453 0636 • Email pszor@symantec.com

ABSTRACT

The number of 32-bit viruses is growing at an alarming rate. Most of these viruses are only able to replicate under Windows 95 but more and more viruses are becoming capable of doing it under Windows NT, too. This paper provides practical information about the problems of memory scanning and their possible solutions under Windows NT. Memory scanning is a must for all operating systems. Once the virus is executed and active in memory it has the potential to hide itself from scanners by using stealth techniques. Even if the virus does not use any stealth techniques, removing the virus from the system becomes more difficult when the virus is active in memory since such a virus can infect the already previously disinfected objects again and again.

There are many viruses which use the directory stealth technique under Windows 95 and Windows NT, respectively. We have also seen the first implementation of a Windows 95 full stealth virus (Win95/Zerg.3849). While it is not an easy task to develop a memory scanner for Windows 95, the problem is much more complex under Windows NT. In my paper I am going to introduce the different ways 32-bit viruses stay in memory as a particular process and describe the possible methods of detecting and deactivating them.

At the end of 1998 we saw the first implementation of a native Windows NT virus (WinNT/RemEx) which runs as a service. While it is possible to detect such a virus in memory even from a User mode application, the problem will become more difficult with a native Windows NT virus which is implemented as a device driver running in Kernel mode. Such a virus cannot be detected in memory in User mode but only in Kernel mode only since the system address space is protected from read and write access under Windows NT unlike Windows 95. This is probably the most important reason why a memory scanner under Windows NT should be implemented as a Kernel mode driver. In my paper I'm going to introduce both User and Kernel mode implementations of a memory scanner under Windows NT.

1 INTRODUCTION

It did not take long for virus writers to realise that a virus can replicate faster if it stays active in memory intercepting operating system calls. In fact, the very first viruses, such as Brain or Jerusalem, already used TSR (terminate and stay resident) techniques. Most of the successful file and boot sector infectors used all kind of hooking strategies. A non TSR virus has a much smaller chance of becoming in the wild

under DOS. By hooking the file system functions a virus can 'see' the access to a particular program or system area easily and infect it 'on the fly'. Of course, this means that most of the important, frequently used applications and system areas become infected very quickly. Therefore, the chance that such a virus is able to pass from one system to another before it is noticed by the user is much bigger. Another advantage of a resident virus is that it can use stealth techniques to hide itself from scanners and integrity checker products very effectively. Full stealth functionality is implemented in many old viruses and it will be used in upcoming 32-bit *Windows* viruses in the future.

The Tremor virus was one of the first 16-bit DOS, full stealth, polymorphic viruses. When it is active in memory it hides itself completely. The size of an infected application, as well as its contents remains 'virtually' the same as long as the virus is active in the memory. While the virus is active in the memory virus scanners cannot detect the infected files easily. The additional problem is that on-demand virus scanners access all important applications and system areas when scanning them. Therefore the active virus can replicate to those objects during the scanning itself! It was obvious to anti virus product developers that memory scanning and disinfection had to be implemented in virus scanner products.

Memory scanning was a relatively a simple task to implement for DOS. Since DOS uses the *Intel* processors in real-mode it cannot access more then 1 MB physical memory and it does not support virtual memory at all. Furthermore DOS does not implement any protection mechanism for the operating system code. The actual DOS kernel and all the applications should share the same limited memory and can interfere (accidentally overwrite) to each other because they have the very same rights on the machine. Therefore, memory scanning was easy to develop for DOS since the full available memory can be directly addressed and accessed by a virus scanner for both read and write operations. Most scanners did not even check if the actual memory region had any active loaded code or data at all, but did a full signature scan of the full physical memory byte by byte. A few years later thousands of virus signatures had to be located in memory and therefore anti-virus products tried to search the active areas of memory for most signatures to speed up the scanning and avoid false positives. Such memory scanners walk through the MCB (memory control block) chain. In DOS, memory is allocated in arenas (a section of memory). Each arena begins with an arena header called MCB. The way to get a pointer to the first MCB is accessible only by an undocumented DOS interrupt (Int 21h/52h function). Sadly, the need to figure out the undocumented interfaces is an every day issue with *Microsoft's* systems. (Not surprisingly, many undocumented interfaces have to be discovered to implement an efficient memory scanner for *Windows NT,* also.)

While it was a relatively easy task to develop a memory scanner for DOS, it is more difficult to do it for *Windows 95* and particularly complex to implement for *Windows NT*. *Windows NT 4.0* manages 'virtually unlimited' memory. The virtual address space is 4 GB in total. Of course, today's *NT* systems use around 64MB – 128 MB of physical memory on average and the rest is virtual only, managed by the operating system by using the less costly (but much slower) storage on a hard disk. A *Windows NT* memory scanner should scan the virtual address space of all running processes. Since the virtually continuous memory is not necessarily physically continuous, a *Windows NT* memory scanner should scan by using virtual addresses instead of physical addresses as DOS memory scanners used to do. Since the *Windows NT* virus scanner used to be a port of the 'original' DOS scanning engine it could happen (in fact I have seen it happen) that a fairly good *Windows NT* programmer blindly ports the DOS memory scanning engine under *NT*. Even if it is not very obvious to implement, scanning the first physical 1 MB memory under an *NT* system for viruses is, of course, a useless task by itself. In the following section let us have a look at the basics of the virtual memory management of *Windows NT* to provide a fair understanding to help implement a memory scanner for it.

2 THE WINDOWS NT VIRTUAL MEMORY SYSTEM

2.1 WHY USE VIRTUAL MEMORY?

You could ask: why is virtual memory useful? It is certainly not necessary since many operating systems do not use virtual memory and still manage to work. DOS does not support virtual memory, but even so it survived on the market for almost two decades. However, a constant problem for developers has always been the limitations of physical memory. In fact, it seems that nothing is enough when it comes to memory. Applications are getting larger and larger and therefore a number of techniques had to be developed to handle limited physical memory situations. One of the most well known techniques is the overlay mechanism in which a particular program is divided into several chunks from which only one at a time can be actively accessed. Whenever a chunk of the program is needed, it is read into physical memory overwriting the previously loaded one in memory. The virtual memory management of the operating system is supposed to solve these problems for all running applications by dividing the memory into a set of pages. Thus, a particular application need not take care of its memory management by using the old techniques. Additionally, virtual memory has other benefits:

- Process isolation (processes have separate address spaces and therefore do not interfere with one another)
- Memory protection (the processor is used in two modes, thus the operating system is clearly separated from the user applications)
- No memory limitation (pages which are currently not in use should not be allocated, data can be shared between applications)

2.2 HOW WINDOWS NT IMPLEMENTS VIRTUAL MEMORY?

Modern processors support virtual memory (VM) management. VM could be developed without processor support but it would be very slow that way. Whenever the processor is running in 'virtual memory mode' all addresses are assumed to be virtual addresses and therefore have to be translated to physical addresses each time the processor executes a new instruction. This is why the CPU support for VM is crucial for fast system performance.

The CPU looks at a 32-bit address as if were made up of three parts: a directory offset, a page table offset and a page offset. Translating a virtual address from page directory to page frame is similar to traversing a b-tree structure, where the page directory is the root, page tables are the immediate descendants of the root, and page frames are the page table's descendants. (Figure 1 illustrates this organisation.)

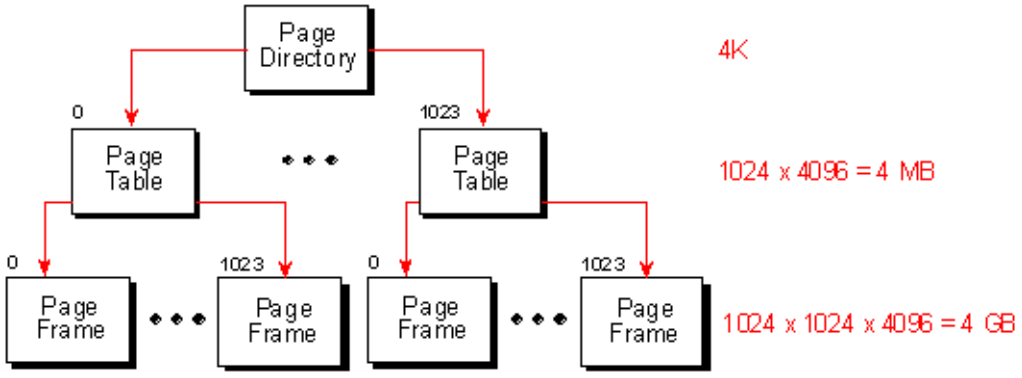


Fig 1. Translating a virtual address is similar to traversing a b-tree structure

The first step in translating the virtual address is to extract the higher-order 10 bits to serve as the first offset. This offset is used to index a 32-bit value in a page of memory called the page directory. Each process has a single, unique page directory under *Windows NT* (which is mapped to 0xc0300000 address under *Windows NT 4 Intel* platforms as Figure 5 shows). The page directory is itself a 4 K page, segmented into 1,024 4-byte values called page directory entries (PDEs). The 10 bits provide the exact number of bits necessary to index each PDE in the page directory (2^{10} bits = 1,024 possible combinations).

Each PDE is then used to identify another page of memory called a page table. The second 10-bit offset is subsequently used to index a 4-byte page-table entry (PTE) in exactly the same way as the page directory does. PTEs identify pages of memory called page frames. The remaining 12-bit offset in the virtual address is used to address a specific byte of memory in the page frame identified by the PTE. With 12 bits, the final offset can index all 4096 bytes in the page frame.

Through three layers of indirection, *Windows NT* is able to offer virtual memory that is unique to each process. A page directory has up to 1,024 PDEs or a maximum of 1,024 page tables. Each page table contains up to 1,024 PTEs with a maximum of 1,024 page frames per page table. Each page frame has its own 4,096 one-byte locations of actual data. That gives a 4 GB of address space (1,024 * 1,024 * 4,096).

2.3 VIRTUAL ADDRESS SPACES

In *Windows NT 4.0* the virtual address space of the system is divided to two parts: the low 2 GB user address space and the high 2 GB system space (Figure 2). When the CPU is running in User mode, only pages of the user address space are accessible, therefore applications cannot interfere with the operating system components which are accessible only in Kernel mode. When a User mode application (such as WINWORD.EXE, NOTEPAD.EXE, etc.) calls an API, it first calls into a subsystem DLL. The subsystem DLL API translates the documented function to an undocumented one in the native API set as part of NTDLL.DLL. When necessary, the native API calls the *Windows NT* executive and then the processor is switched to Kernel mode.

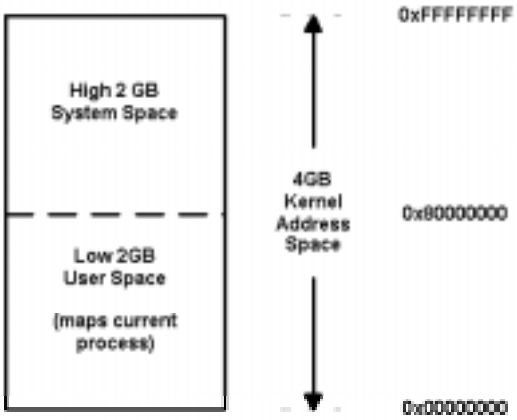


Fig 2. *Windows NT* standard 32-bit linear address space division

The 4 GB address space division can be changed by using *Windows NT* Enterprise Edition and a special BOOT.INI option. In that case the user address space is 3 GB which leaves 1 GB for the system address space. This is done to support applications which are using very large databases and can work more efficiently this way. (Figure 3)

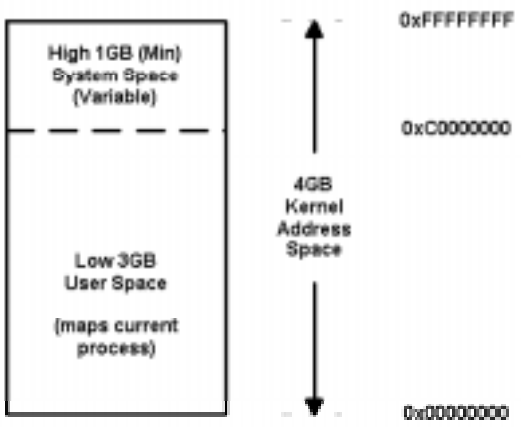


Fig 3. *Windows NT Enterprise* Edition address space division (13 GB)

One of the new features of *Windows 2000* on Alpha APX systems will be the extension of the virtual memory address space to a total of 32 GB rather than the current 4 GB.

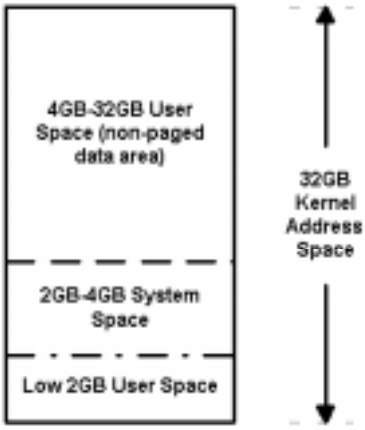


Fig 4. *Windows 2000* VLM address space division

The upper user space is not paged and can be used only for data, not for code (Figure 4).

In all these models the user address space maps a particular process at a time. Each time a User mode application is executed *Windows NT* creates a virtual address space for the new process. The same virtual address can be used by any number of applications, but the virtual address does not necessarily point to the same physical page in memory. When Process A accesses a page at 0x00400000 (usual Base Address of applications) Process B's page at 0x00400000 may not be even valid at all. Process A can not interfere with Process B by using the same address since it is valid only in its own context. On a single CPU, only one virtual-to-physical mapping can be in use. Each time a particular thread is scheduled for execution a context switch happens, changing the actual virtual to physical mappings to the process context in which the scheduled thread is running. To provide Kernel mode components (and drivers) with an environment in which they know that their memory references are always valid for the upper 2 GB of address space, *NT* provides a portion of page tables which hold the same information in each context.

The Virtual Memory Manager handles the system address space differently from the user address space (Figure 5). In that address space *Windows NT*'s code components are loaded together with all the Kernel

mode drivers. Since Kernel mode drivers have the same privilege and view of the system address space they can interfere with the operating system's code or with one another. (An example list of loaded components is available in Appendix D).

Fig 5. Normal system address space layout (*Intel*)

Probably the most demanding problem of virtual memory management is the paging mechanism. *Windows NT* has the ability to reclaim pages of memory which are no longer needed. The Memory Manager changes individual entries in the page table so that instead of pointing to the actual physical page, it is marked as invalid. If the page belongs to an executable and it is not a dirty page (a page whose context changed during execution) nothing else needs to be done, apart from marking the page to be invalid. Otherwise, the changed page has to be written into a file, most likely to the page file (pagefile.sys). When the page is accessed again a page fault will be generated. Then the actual virtual address is checked if it is available anywhere. If it is mapped in from a file (like most DLLs and applications), the page is read from the particular file in which the data exists. Otherwise, the information from a file or from a page file will be read in and then the instruction which generated the fault will be rerun by *Windows NT*.

Windows NT is able to share a physical page of memory between several processes. That means that several copies of an executable application will not reserve the very same amount of memory each time. Instead, the same physical pages are seen from all views. When the contents of a page should change, not every process context will change, but only the instance which needs the change. This is done by reserving a new physical page moving the data from the copy-on-write page to the new page and make the change in the new copy.

3 MEMORY SCANNING IN USER MODE

3.1 HOW TO ACCESS A PARTICULAR PROCESS' DATA IN MEMORY

The first question of memory scanning is the problem of accessing a particular process' data in memory. As discussed in the previous sections, Process A cannot interfere with Process B. How can a User mode scanner read the contents of all the other processes? The answer is an API called ReadProcessMemory(). This API is usually used by debuggers to control the execution of the traced application by the debugger. The ReadProcessMemory() API needs a handle to a process which can be got by the OpenProcess() API and the PROCESS_VM_READ access right. OpenProcess() needs the ID of a process. Where do we get a process ID from?

The answer was not obvious for some time since the actual DLL (PSAPI.DLL) in which documented process enumeration APIs have been placed is not part of the standard *Windows NT* environment. The lack of PSAPI.DLL and the missing documentation urged me to find out how *NT* actually does this itself. Since Task Manager and several other application can display all the running processes and their IDs, it was obvious that it was possible to do so without the use of PSAPI.DLL. In fact it turns out that most APIs in PSAPI.DLL are just wrappers around native service APIs placed in NTDLL.DLL such as NtQuerySystemInformation(). The native API set is not documented by *Microsoft* and is mostly used by subsystems. Most applications do not link to NTDLL.DLL directly because of this. In fact, *Microsoft* suggests to use the documented interfaces. However, Task Manager (TASKMGR.EXE) is linked to NTDLL.DLL directly even if the information could be obtained by using performance data.

Task Manager uses the NtQuerySystemInformation() native API to get a list of all running processes and their process IDs. A User mode application can link itself to NTDLL.DLL or simply use GetProcAddress() to get the address of the API to call it. When the process ID of a particular process is available ReadProcessMemory() can be used to read the actual address space of that particular application. Of course, a memory scanner should know the exact location of the used pages of an application in order to be able to do so. Fortunately, the VirtualQueryEx() function provides information about the range of pages within the virtual address space of a specified process. It needs an open handle to a process and returns the attributes and the sizes of regions. In addition it needs access to PROCESS_QUERY_INFORMATION for this operation. Free and reserved pages can be easily eliminated with this function and those should not be accessed, but the rest have to be checked. This can be done by using the ReadProcessMemory() API on those pages.

3.2 NTQUERYSYSTEMINFORMATION()

NtQuerySystemInformation() (NtQSI) is not documented by *Microsoft* and it would not be necessary to use it since a User mode application can link itself to PSAPI.DLL which will call NtQSI in turn. However, as we will see later on, this function can be useful in a Kernel mode implementation of a memory scanner and therefore it is better to talk about it a bit. NtQSI has four 32-bit (DWORD or ULONG) parameters. The first parameter could be named QuerySystemInformationClass. This parameter specifies the type of information to be returned by the function. It has several values from which five specify the running process list query. The second parameter is the address of the returned buffer which should be allocated by the caller. The third parameter is the allocated size in bytes. The fourth parameter is an optional value a PULONG BytesWritten. NtQSI() returns an NTSTATUS value. When the returned value is not STATUS_SUCCESS (0) it is usually STATUS_INFO_LENGTH_MISMATCH which means that the allocated buffer length does not match the length required for the specified information class. Therefore NtQSI() has to be called with bigger and bigger buffers in a loop until the information can be placed into the allocated buffer completely by the *Windows NT* executive.

On correct return the necessary information is placed in the buffer in the form of a linked list. The first DWORD value specifies the relative pointer of the next process of block information from the start of the buffer. The DWORD value at offset 0x44 of each block is the process ID. With this ID several additional APIs can be called, this is why it is the most important one. Other important information such as the loaded images (EXE and DLLs) and their base address can be examined by other native API calls by using this process ID such as RtlQueryProcessDebugInformation() (which uses allocated buffers created by RtlCreateQueryDebugBuffer() and deallocated by RtlDestroyQueryDebugBuffer() APIs). Of course these are all undocumented native APIs. Things would be so much easier to develop using correct documentation from *Microsoft*, wouldn't they?

3.3 COMMON PROCESSES AND SPECIAL SYSTEM RIGHTS

On a typical *Windows NT* machine several processes are running already even if the user has not even logged in. The most important set of these processes are the System Idle Process, The System Process, SMSS.EXE , CSRSS.EXE, WINLOGON.EXE and SERVICES.EXE. A *Windows NT* scanner should scan all of these address spaces and all the other running processes which were executed by the user.

The trick is that some of these processes cannot be opened by OpenProcess() to get a handle for the other APIs with the necessary access. In *Microsoft Press* documentation (such as the *Advanced Windows NT Third Edition*, p. 959) it is stated that some of the processes are secure processes and therefore cannot be opened for QUERY_INFORMATION or VM_WRITE operations. Such processes are WINLOGON.EXE, CLIPSRV.EXE and EVENTLOG.EXE). Well, this is true; such processes need an additional system security privilege to be adjusted first. (This is the missing information from the *Microsoft* documentation.) In particular the SeDebugPrivilege privilege value has to be adjusted to the SE_PRIVILEGE_ENABLED attribute. The SeDebugPrivilege is available to administrators and equivalent users or anyone who has been granted this privilege by an administrator. However, even under an administrator account the default attribute of this privilege is not enabled and therefore OpenProcess() will fail to open secured processes. To enable this privilege the OpenProcessToken() function has to specify TOKEN_ADJUST_PRIVILEGES, then LookupPrivilegeValue() can be used to check if the user has the privilege at all and only if the user has it, can it be set to SE_PRIVILEGE_ENABLED by the AdjustTokenPrivileges() API.

Of course, it makes very good sense to protect some of the applications this way. For instance *Windows NT* simply crashes if WINLOGON.EXE stops working. Therefore, a modification caused by any User mode application inside a random location of WINLOGON.EXE's address space could cause the system to crash by a User mode application. Of course, this would not be great. In any case, WINLOGON.EXE can even be written in memory when this privilege is enabled. However, the privilege would first have to be granted to all users to scan such applications in memory. In a situation when WINLOGON.EXE is infected, the infected process could not be detected. Giving debug privilege to all of our users would not make the system more secure for sure. This is why a memory scanner can be developed as a Kernel mode driver where PROCESS_ALL_ACCESS is gained easily because drivers are running with the highest rights on a *Windows NT* machine.

3.4 VIRUSES IN THE WIN32 SUBSYSTEM

In this section I shall introduce the different in which how viruses can become active as part of a particular process. Most 32-bit User mode applications are running in the Win32 subsystem. The Win32 subsystem is the most important subsystem of *Windows NT*. This subsystem is created and used by default and it cannot be disabled, unlike the other subsystems. This is the subsystem in which Win32 viruses can be active. The Win32 subsystem consists of the following major components: CSRSS.EXE (the environment subsystem process), the Kernel mode device driver WIN32K.SYS, subsystem DLLs (such as USER32.DLL, ADVAPI32.DLL, GDI32.DLL and KERNEL32.DLL) which translate

documented Win32 API functions into the appropriate undocumented kernel-mode system service calls to NTOSKRNL.EXE and WIN32K.SYS. There is one more very important part of the Win32 subsystem NTDLL.DLL primarily for the use of subsystem DLLs. NTDLL.DLL is used in the other subsystems of *Windows NT* also and by Native applications which do not run in a subsystem. (Appendix B shows some of the system processes, the loaded DLLs with their base addresses and sizes.)

3.4.1 Win32 viruses which allocate private pages for their own use

Some Win32 viruses allocate private pages with the PAGE_EXECUTE_READWRITE attribute for themselves. When the infected application is loaded the virus code is activated from the executed application code. Then the virus allocates new pages for its own use and moves its code into there. The write access to those pages is important for the virus since it stores data in itself which has to change and read only pages can not be written.

For instance Win32/Cabanas.3014.A allocates a 12,232 bytes block which will be represented as three pages (3*4096=12888 bytes) from the address space of the infected process (Appendix C/1). Since Win32/Cabanas uses the MEM_TOP_DOWN flag when it allocates memory with the VirtualAlloc() function the actual three pages will be available at the very end of the user address space, usually somewhere around the 0x7FFA0000 address. Win32/Cabanas hooks some of the KERNEL32.DLL APIs to itself by patching the Import table entries of the host program to its own routines. Whenever the host application calls any of the hooked APIs the virus has the chance to replicate 'on-the-fly' to another applications or call its directory stealth routines.

The Win32/Parvo.13857 virus allocates 13,2605 bytes from the address space of the infected process (more exactly 33 pages, 13,5168 bytes) since it needs a lot of memory for its polymorphic engine and for its communication modules. Win32/Parvo does not use the MEM_TOP_DOWN flag and therefore its allocated pages will be reserved from the first sufficientlyfree gap of user address space (0x002F0000 in the infected NOTEPAD.EXE as an example shown in Appendix C/2a). The virus code will be active with the name of the original infected and executed application. Only one copy of the virus is active at a time. The original host will be executed as the 'child' process of the infected application under a random name (Appendix C/2b). Since the host program will be executed almost immediately the virus can silently infect other applications from its own process and is also able to spam itself to other locations with its communication module based on the use of WSOCK32.DLL APIs.

3.4.2 Native Windows NT service viruses

A new class of *Windows NT* viruses activate by dropping a new executable image which is loaded as a native *Windows NT* service as performed by WinNT/RemEx. The WinNT/RemEx virus is running as a User mode service called IE403R.SYS (Appendix C/3). The virus sleeps for a while; then wakes up and tries to infect other applications periodically.

3.4.3 Win32 viruses which set up an active and hidden window procedure

A relatively new type of virus, {Win32,W97M}/Beast.41472.A, installs a hidden window procedure for its own use and uses a timer. Timers were available back in 16-bit *Windows* versions and they were sometimes used to simulate 'multi-threaded' functionality. This virus runs as a complete process and uses OLE APIs to inject embedded macros and executable (the binary virus code itself) into *Office 97* documents. Since the virus is able to infect *Office 97* documents from its active process a macro virus specific scanner and disinfector has a hard time removing it from documents if it is unable to detect the virus in memory first.

3.4.4 Win32 viruses which are part of the executed image itself

Win32/Heretic.1986.A was the first virus to infect KERNEL32.DLL under *Windows NT* correctly. KERNEL32.DLL is used by most applications since most of the crucial Win32 APIs are exported from it.

Once KERNEL32.DLL is infected most executed applications will be attached to it since they need to call APIs from it. Win32/Heretic patches the export address table of KERNEL32.DLL so that the CreateProcessA() and CreateProcessW() functions will point to the last section of the DLL where the virus code is placed (Appendix F/2). When these functions are called by the host program the virus has the chance to infect other applications 'on-the-fly'. The virus enlarges the last section (.reloc) of KERNEL32.DLL and puts its code there, modifying the characteristics of that section to both MEM_EXECUTE and MEM_WRITE type. (Appendix F/1 shows the virus code in memory at the end of an infected KERNEL32.DLL)

Another class of Win32 virus stays active as part of an infected executable image as performed by the Win32/Niko.5178 virus (Appendix E/1). The Win32/Niko virus is activated from an infected Portable Executable application. The virus adds itself into the last section of the PE application and modifies the characteristics of the last section to MEM_WRITE. This will allow the virus code to be modified in memory. The virus does not allocate memory for its full code but for small data blocks only whenever needed. Win32/Niko is one of the first virus to be multi-threaded. The virus creates two threads for its own use. (Appendix E/2) One of them is the trigger thread (which is supposed to display a message on a particular day), the other one is the infection thread. The host program is executed after the threads are created by the virus. As long as the host program is running, the infection thread of the virus will also be active. If the host application (main thread) terminates, all threads of the process will be killed by *Windows NT*, therefore the virus will be no longer active. The virus can replicate to other files only from those applications which are running and used for a longer time. In such a situation the infection thread will infect other applications from the background.

3.5 MEMORY SCANNING

With certain restrictions a User mode memory scanner can be developed by using the functions described earlier. The scanner should be able to eliminate the committed pages and the free pages and has to do a full scan on all the committed pages of each running processes because virus code could be placed in any of them. The actual virus code can easily be eliminated in its active and passive forms by paying attention to the used 'image' or 'private' pages. Since *Windows NT*'s memory manager reclaims unused pages and pages are not read in memory until they are accessed, the speed of the memory scanning will largely depend on the size of physical memory. The more physical memory a particular computer has, the faster the memory scanner will be because the number of page faults will be much higher if the computer has very limited physical memory. Appendix A/1a shows that unused pages, pages whose access flag was cleared by the memory manager for some time, are reclaimed from all applications. For instance WINLOGON.EXE's 'Mem usage' is 356 KB only, as shown in the example. Appendix A/2a shows how the memory usage of all running processes changed when SCANPROC.EXE (a User mode memory scanner) scanned them. WINLOGON.EXE's 'Mem usage' went up as much as 7792 K and the number of page faults caused in the process grew to a few thousand (Appendix A/1b, Appendix A/2b). This is a side effect of the memory scanning. Whenever SCANPROC.EXE accesses a new page which is not in the physical memory yet it will cause a page fault and at that point the Memory Manager will read the page into the physical memory causing the 'Mem usage' to grow also. Of course, the memory usage of a process will again be smaller and smaller since most pages will not be accessed again after some time and therefore they will be reclaimed again. *Windows NT*'s Memory Manager has several worker threads for maintaining the balance of the memory usage between each process. Fortunately memory scanning does not cause critical problems for the memory management of *Windows NT*.

3.6 MEMORY DISINFECTION – DEACTIVATION OF ACTIVE VIRUS CODE IN MEMORY

This paper would not be complete without some words about the deactivation possibilities of different virus types. A memory scanner should work closely with an on access virus scanner. A memory scanner should always know the viruses it is searching for and the same set of viruses should be known by the file

scanner components of the anti virus product. The on access *Windows NT* virus scanner can detect most known viruses even if the virus code is active in some processes. However, it cannot stop the virus replicating on the system since the active virus can infect the disinfected object again. The virus code can not be detected in the applications before the virus code is written to them. A new copy of the known virus code cannot be executed since the on access scanner will be active.

Most likely, a particular virus can become active on a machine in the following situations:

- The virus scanner has not been installed on the computer and the virus code is executed already.
- The virus is new and the scanner needs an update to be installed to detect it.

3.6.1 Terminating a particular process which contains virus code

Probably the easiest way to deactivate the virus in memory is to kill the particular task in which the virus code is detected by the memory scanner. This can be done easily by using the TerminateProcess() API and the appropriate rights (PROCESS_TERMINATE access is needed). However terminating a task is a risky procedure and should be used very carefully. Since active virus code is most likely attached to a user application, important user data could be lost if the infected process is simply killed, because any application could keep open several database files which most likely could not be kept consistent. Therefore, TerminateProcess() should be used in situations where the virus code is active as a separate process, for instance in the case of WinNT/RemEx or Win32/Parvo. *Windows NT*'s Task Manager, just like the Service Manager does not allow termination of some process, because of security reasons. For instance, the WinNT/RemEx virus cannot be terminated by the Task Manager or Service Manager, respectively. TerminateProcess() can be used in such situations to kill the active virus in memory very efficiently.

Some viruses like variants of Win32/Semisoft variants try to avoid termination by executing two different virus processes. Each time one virus process is terminated, the active copy of the virus will restart the terminated one, protecting itself that way very efficiently. This is why memory scanning should assume an on-access virus scanner in the background which will not allow the new virus task to be executed again.

3.6.2 Terminating virus threads inside an infected process

In case a virus creates its own threads in a process, the memory scanner should be able to eliminate the threads belonging to the virus itself and terminate those threads in the process. For instance, Win32/Niko (Appendix E/2) creates two threads for itself. One thread is used for the trigger routine and will terminate by itself. The infection thread will be active as long as the process (with at least one thread of its own) is running. A thread handle is needed with the necessary THREAD_TERMINATE access in order to terminate a particular thread of a process. OpenThread() is not available in the subsystem DLLs of *Windows NT* 4.0. The function is undocumented and available only from the NTDLL.DLL as NtOpenThread(). (Figure 6 shows my own declaration.) In order to eliminate the virus threads from the clean application threads the memory scanner should check the Win32StartAddress of each thread. Win32StartAddress is available in the performance data, but it is easier to get it by using another undocumented API. This API is called NtQueryInformationThread() which has five parameters. The first one is a thread handle with THREAD_QUERY_INFORMATION access. The next parameter is the QueryWin32StartAddress class value which is 9. The third parameter is the address of the return value and the fourth parameter is the size in bytes (4) of the information to be returned. The last parameter, BytesWritten, is a PULONG optional value and can be NULL. NtQueryInformaionThread() will return the correct start address of a particular thread as shown by the TLIST.EXE application (available in the *Windows NT* resource kit). (Appendix E/2 is an output of TLIST.EXE used on a process in which Win32/Niko virus is active.) In the example, the start of the two virus threads is 0x0040f021 and 0x0040f01c, respectively. Both of these addresses are pointing into the active virus image, each of them

to a jump instruction (0xe9) which will, in turn, give control to the entry points of the virus thread functions. By checking the Win32StartAddress of a thread the memory scanner is able to determine if all thread belong to a virus, since the start address of the thread will point into the active virus image in memory. Inthe case of Niko the virus, code is executed as the main thread of the host application and therefore the Win32StartAddress (0x0040f000) of the main thread (entry point) should not be terminated since that same thread is used by the host program.

```
NTSYSAPI
NTSTATUS
NTAPI
NtOpenThread (
 OUT PHANDLE ThreadHandle,
 IN ACCESS_MASK DesiredAccess,
 IN POBJECT_ATTRIBUTES ObjectAttributes,
 IN PCLIENT_ID ClientId OPTIONAL
 );
```

Fig 6. NtOpenThread API header declaration

The final step is to terminate the thread with TerminateThread() API and THREAD_TERMINATE access.

3.6.3 Patching out the virus code in the active pages

The most difficult case of deactivation is when the virus is active as part of a loaded EXE or DLL image or it allocates pages for itself on per-process basis, hooking some imports of the host application to itself. In these situations, the active virus code has to be patched in memory so that the virus is deactivated. This procedure has to be very carefully developed, because an incorrect patch of the virus code in memory could cause a new variant to be created accidentally by the memory disinfection itself!

When the virus hooks APIs to itself by patching the host applications Import address table, the Import address table should be fixed in each of the infected processes. This will remove the virus code from the API chain. This operation has to be done very quickly. Probably the safest way is to suspend each thread of an infected process during the time of this fix. When the Import address table is fixed threads can be resume again. WriteProcessMemory() can be used to write into the necessary pages in these situations. The disinfection should be performed from instance to instance of the virus. The protection flags of each page which need modification have to be checked first. If the page has PAGE_READONLY access the protection flag should be changed to PAGE_READWRITE first. VirtualProtectEx() function can be used with PROCESS_VM_OPERATION access in such cases.

A much more difficult case is when a particular subsystem DLL is infected by a virus like Win32/Heretic or the virus patches the Internet communication library (WSOCK32.DLL) as illustrated by Win32/Ska.A. In the case of Win32/Heretic, KERNEL32.DLL is infected so that the export addresses of two APIs are patched in the file itself (not in memory only). When a particular process gets the address of such an API with the GetProcAddress() function, it will get a pointer to the virus code. Since some applications determine the addresses of certain APIs during initialization, they will remember such addresses as long as they are running. This is why the Export address table of KERNEL32.DLL should not be fixed during memory disinfection since in some situations the virus could be activated again regardless of this particular fix. Instead of fixing the Export table the disinfector should patch the active virus code in memory very carefully. This can be done by modifying the virus code at the entry point of its hook routines, so the control will be given to the exit of the hook functions where the virus calls the original

API entry point. That way the virus can no longer replicate. Of course, this procedure is virus-specific and needs exact identification of the virus code.

3.6.4 How to disinfect a loaded subsystem DLLs or running applications

A loaded subsystem DLL is shared in memory and therefore it cannot be written to. The image can be disinfected in memory, but not in the file itself, since the disinfector cannot open the file for writes. The easiest solution to this particular problem is to build a list of such applications and ask the user to reboot.

For instance the disinfection can be performed by a native disinfector even from User mode. A list of native *Windows NT* application is executed even before any subsystem is loaded. This is possible by executing an application under the [HLM\SYSTEM\ CurrentControlSet\Control\Session Manager]BootExecute section. Such an application should be linked against the NTDLL.DLL only and has to take care of its own memory management and termination using an undocumented set of functions. Such a native application can open even the PAGEFILE.SYS file during boot time.

Some of the standard *Windows NT* application are native applications. For instance, AUTOCHK.EXE is such an application. The Subsystem field in the PE header of such applications is set to 1 (Native).

4 MEMORY SCANNING IN KERNEL MODE

Memory scanning in Kernel mode is very similar to User mode implementation in its basic functionality. It will be always safer to perform memory scanning from Kernel mode. Furthermore, a Kernel mode memory scanner can scan the upper 2 GB of kernel address space for viruses. Currently there is no known virus which has a Kernel mode component. However, it is very likely that such a virus will be developed as a file system filter driver in the future. In this section I am going to explain the major problems in developing a Kernel mode memory scanner for current Win32 viruses which are running in User mode. I am going to introduce the basic procedures which are important to scan the upper 2 GB address space against Kernel mode viruses of the future which will be loaded as drivers.

4.1 SCANNING THE USER ADDRESS SPACE OF PROCESSES

In Kernel mode, the user address space scanning of each process can be performed very similarly to the User mode memory scanning. In fact, many system functions can be used by adapting them in Kernel mode. There are several ways to get the process IDs of each running application. One possibility is to use the NtQuerySystemInformation() API which is exported from NTOSKRNL.EXE by name and therefore easily callable as ZwQuerySystemInformation() (ZwQSI) from a Kernel mode driver. Of course, the function is undocumented and therefore the necessary declarations have to be specified and included first, otherwise the linker cannot link the driver correctly.

4.1.1 Determination of NT service API entry points

Unfortunately, some of the important APIs needed for memory scanning are not exported by name from NTOSKRNL.EXE for the use of a Kernel mode driver. When a User mode application calls the KERNEL32.DLLÞVirtualQueryEx() API the call is redirected to the NTDLL.DLLÞNtQueryVirtualMemory() function.

This API is not available from NTOSKRNL.EXE. A driver can solve this problem in two different ways. It can be linked against NTDLL.DLL. is the easiest way. The other possibility is to develop a function similar to the User mode GetProcAddress (with some important differences) which can get the function ID of a particular *NT* service by traversing the Export table of the NTDLL.DLL in the system context. Such a function can pick up the *NT* service function ID which is placed into the EAX register with a MOV instruction at the entry point. That way the driver can specify the correct address of the function inside the *Windows NT* executive (NTOSKRNL.EXE) as KeServiceDescriptorTable+NtFunctionID.

4.1.2 The most important NT functions necessary for Kernel mode memory scanning

NtQueryVirtualMemory() queries the pages of a particular process. It is not documented but is only a translation of the VirtualQueryEx() API. ZwQueryVirtualMemory() is placed in NTOSKRNL.EXE and its name is shown by the *Windows NT* kernel debugger since the debug information contains the name of the function. However, this function, like several others, is not exported by name from NTOSKRNL.EXE.

Other useful functions are NtTerminateProcess(), NtOpenThread(), NtSuspendThread(), NtResumeThread(), NtProtectVirtualMemory(). Most of these functions are translations of their User mode equivalent versions but remain undocumented. The header declarations have to be done one by one for each of these functions.

4.1.3 Process context

In *NT*, Kernel mode drivers run in three different classes of context:

- System process context
- Specific thread (and process) context
- Arbitrary thread (and process) context

Depending on the circumstances, the lower 2 GB of virtual memory maps any user process or no user process at all. The memory scanner should be able to switch to the context of a particular process in order to map the process to the lower 2 GB of the virtual memory, by the undocumented KeAttachProccess(). The necessary header declaration of this API is:

```
VOID KeAttachProcess(
        IN PEPROCESS    Process
);
```

This kernel API needs a pointer to an EPROCESS structure first. This can be converted by another undocumented API called PsLookupProccessByProccessId() by passing a normal process ID as the first parameter:

```
NTSTATUS
PsLookupProcessByProcessId(
 IN ULONG Process_ID,
 OUT PVOID *EProcess);
```

Each time the Kernel mode memory scanner needs to read a page it should switch the context to the particular process it wants to access. KeDetachProcess() returns from any context to the System context.

```
VOID KeDetachProcess(
        VOID
);
```

The query function has to be very carefully developed to work correctly in all problematic circumstances. Since the process pages can be queried in the above way pages which are not available should not be accessed. Otherwise the memory scanning would be terribly slow since, far too many exceptions would slow down the system unacceptably.

4.2 SCANNING THE UPPER 2 GB OF ADDRESS SPACE

The upper 2 GB of the address space contains executable code such as the *NT* executive, the system drivers and third party drivers. The list of drivers can be queried by using Object Manager functions or the NtQuerySystemInformation() function 0x0b which returns the list of loaded drivers with their base addresses. Of course, none of these functions are documented and fortunately there is no need to use them yet since there are no known viruses which will be loaded in that area. However it will be very important to scan this area of the address space as soon as new viruses are developed which have Kernel mode components. It is not very easy to query the pages of that area. In fact, there seems to be no easy way to do so. Therefore, the easiest solution is to check the base address of each driver and parse their structure in memory directly. This is possible since any driver has complete access to the upper 2 GB of address space. Most likely it will be enough to identify a particular driver in memory since the virus will be developed as a complete driver itself (at least, this sounds like the easiest solution). This can be easily done by parsing the section header table of each driver in memory.

4.2.1 How to deactivate a filter driver virus? Filtering a filter driver

The former question sounds very weird since no existing virus is known to be using this approach. In any case this method is definitely possible and we can be sure that such a virus will be developed. (In this section I assume that the reader has basic knowledge of *Windows NT* drivers.)

The problem is that filter drivers cannot be unloaded. At least this is the suggestion from *Microsoft* and therefore it should be considered as a very strong opinion. File system filter drivers are attached to the Device Object of a particular file system driver (NTFS.SYS, FASTFAT.SYS, etc) or they are attached to another filter driver's Device Object, building up a chain of filter drivers. In fact, a particular filter can be attached to many Device Objects of other drivers. (Figure 7 shows an example of this.)

A filter driver can be detached from the end of the list easily, but even in this case, it is not safe to do so. An additional problem is that a filter driver in between two other filter drivers, or between a file system driver and a filter driver, cannot be detached, since this would detach all drivers after itself on the chain at the same time! Therefore another solution is necessary. After several unsuccessful attempts I found an approach which worked correctly.

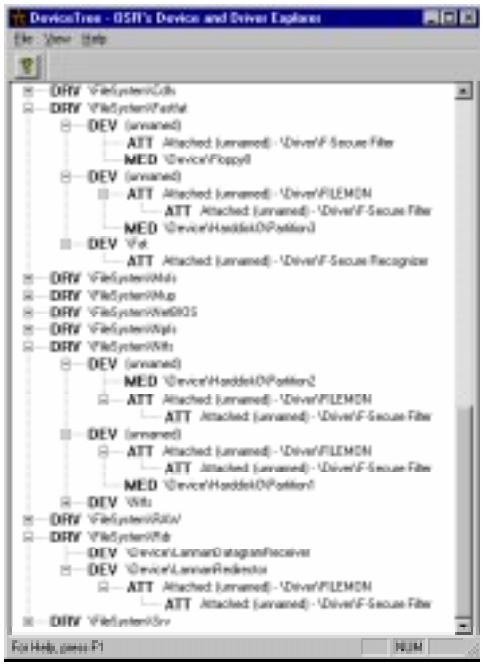


Fig 6. An example chain of filter drivers shown by OSR's DeviceTree utility

The execution of a driver begins in its DriverEntry function. Within this function, filter drivers typically create a new Device Object (a hook device) and then attach it to the Device Object of the device to be filtered by calling the IoAttachDevice(), IoAttachDeviceToDeviceStack() or AttachDeviceByPointer() function.

File system filter drivers have to support fast I/O and therefore they implement a FAST_IO_DISPATCH table with function pointers to its own fast I/O entry points. After performing the fast I/O filtering in a particular fast I/O hook routine, the filter driver has to call the original fast I/O entry point of the driver to which the filter driver's hook device was attached. Interestingly, the pointer to the lower Device Object is not saved by *Windows NT* itself. Each driver has to save these pointers by itself and it is recommended to keep this pointer in the DeviceExtension of the hook device. However, the DeviceExtension is absolutely driver-specific in structure and each driver can define it to its own preferred format or not use it at all.

It seems the only way to 'deactivate' a filter driver safely is to 'filter it' by using a non-standard method which does not let the driver receive control in any of its filtering routines. Instead the driver to which the particular filter driver was attached has to be called. In order to do, so the re-filtering driver (DeactivatorDriver) has to patch the filter driver's Driver Object (VirusDriver). All VirusDriver->MajorFunction[] entries should point to the HookDispatch routine of the DeactivatorDriver instead. Additionally, the VirusDriver->FastIoDispatch field should point to the fast I/O table of the DeactivatorDriver. When this patch is performed correctly the fast I/O entries of the DeactivatorDriver will get control instead of the VirusDriver's own. The major problem is that each fast I/O routine of the DeactivatorDriver should call the fast I/O routine under the VirusDriver by traversing the Devices object chain of the VirusDriver. The AttachedDevice field of all file system drivers' Device Object has to be checked to see if any of the VirusDriver's hook device is attached to them. When the AttachDevice field of a file system driver's Device Object is equal to any of the VirusDriver's hook Device Object pointer the Device Object pointer of the file system driver should be saved. Each time the DeactivatorDriver's fast I/O is called, the fast I/O can be redirected to the driver to which the VirusDriver was attached, because the saved Device Object pointer will point to a Device Object which has a pointer to the owner Driver Object. If that Driver Object has a fast I/O entry point for the fast I/O which has been filtered by the VirusDriver's fast I/O routine it should be called by passing the incoming parameters to it without any modification. From then on, the fast I/O of the VirusDriver is re-filtered and deactivated.

In much the same way, the Dispatch routine of the DeactivatorDriver has to complete the IRPs of the VirusDriver or pass the IRP to the corresponding Device Object with IoCallDriver() routine.

Complicated? Well, there is no doubt about it. Certainly this could be done more easily if *Windows NT*'s filter driver mechanism would be organized a bit better. However, there are no major changes regarding that even in *Windows 2000*.

5 CONCLUSION

Memory scanning and disinfection is a very challenging task under *Windows NT*. The multi-tasking, multi-threaded environment is much more complex than DOS and, therefore, most 32-bit viruses are very complex, too. As the number of Win32 viruses grows, the anti-virus world will face more difficult problems . It is extremely important to study the upcoming Win32 viruses in detail in order to deal with them correctly. Memory scanning was a few lines of code in a DOS virus scanner. It is not much different under *Windows NT*, but much more time is needed to develop it correctly. I hope my paper will help scanner developers to reach this goal a bit faster.

6 BIBLIOGRAPHY

[1] Jeffrey Richter '*Advanced Windows NT*', Microsoft Press, 1994

[2] Dr Dobb's Journal System Internals CD-ROM, 1998

[3] Peter G Viscarola & W A Mason, '*Windows NT Device Driver Development*', MTP, 1999

[4] 'Virtually Unlimited Memory', *The NT Insider*, March-April 1998

[5] 'Virtual Memory', *The NT Insider*, January-February 1999

[6] Péter Ször, 'Attacks on Win32', *Virus Bulletin Conference*, 1999

[7] Péter Ször, 'Parvo - One Sick Puppy', *Virus Bulletin*, January 1999

[8] Péter Ször, 'Happy Gets Lucky?', *Virus Bulletin*, April 1999

[9] Péter Ször, 'Beast Regards', *Virus Bulletin*, June 1999

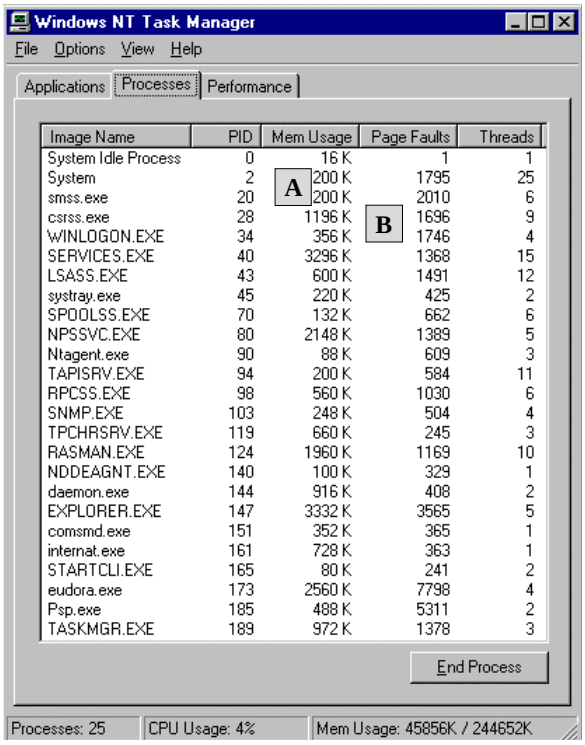
[10] Eugene Kaspersky, (*Kaspersky LAB*), personal communication.

[11] Sergey Belov, (*Kaspersky LAB*), personal communication.

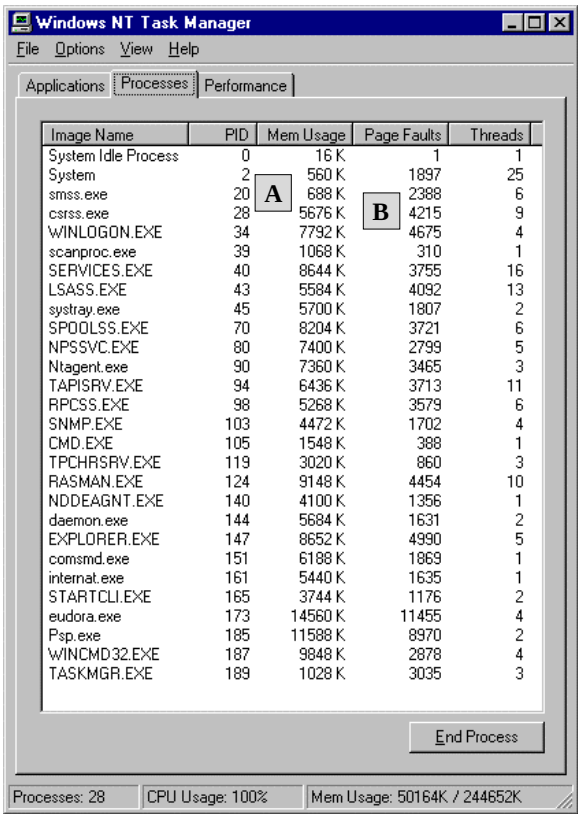
[12] Ismo Bergroth, (*Data Fellows*), personal communication.

[13] Kari Laasonen, (*Data Fellows*), personal communication.

APPENDIX A



Checking memory usage before memory scanning.



Checking memory uasage during memory scanning.

APPENDIX B

```
PID: 0x0014

BaseAddress  Size        Name
0x023a0000   0x0000c000  \SystemRoot\System32\smss.exe
0x77f60000   0x0005c000  C:\WINNT\System32\ntdll.dll

PID: 0x001c

BaseAddress  Size        Name
0x5ffe0000   0x00005000  \??\C:\WINNT\system32\csrss.exe
0x77f60000   0x0005c000  C:\WINNT\System32\ntdll.dll
0x5ff90000   0x0000b000  C:\WINNT\system32\CSRSRV.dll
0x5ffa0000   0x0000c000  C:\WINNT\system32\basesrv.dll
0x5ffb0000   0x00030000  C:\WINNT\system32\winsrv.dll
0x77e70000   0x00051000  C:\WINNT\system32\USER32.dll
0x77f00000   0x0005e000  C:\WINNT\system32\KERNEL32.dll
.
.
.
0x5f810000   0x00007000  C:\WINNT\system32\rpcltc1.dll

PID: 0x0022

BaseAddress  Size        Name
0x02880000   0x00030000  \??\C:\WINNT\SYSTEM32\winlogon.exe
0x77f60000   0x0005c000  C:\WINNT\System32\ntdll.dll
0x78000000   0x00048000  C:\WINNT\system32\MSVCRT.dll
0x77f00000   0x0005e000  C:\WINNT\system32\KERNEL32.dll
0x77dc0000   0x0003e000  C:\WINNT\system32\ADVAPI32.dll
.
.
.
0x77850000   0x0003a000  C:\WINNT\SYSTEM32\NETUI1.dll
```

List of some system executables with their DLLs.

APPENDIX C

```
PID: 0x0051

BaseAddress  Size        Name
0x01b40000   0x00010000  C:\WINNT\system32\notepad.exe
0x77f60000   0x0005b000  C:\WINNT\System32\ntdll.dll
0x77d80000   0x00032000  C:\WINNT\system32\comdlg32.dll
0x77f00000   0x0005c000  C:\WINNT\system32\KERNEL32.dll
0x77e70000   0x00053000  C:\WINNT\system32\USER32.dll
0x77ed0000   0x0002b000  C:\WINNT\system32\GDI32.dll
0x77dc0000   0x0003e000  C:\WINNT\system32\ADVAPI32.dll
0x77e20000   0x0004f000  C:\WINNT\system32\RPCRT4.dll
0x77c40000   0x0013b000  C:\WINNT\system32\SHELL32.dll
0x77bf0000   0x0004f000  C:\WINNT\system32\COMCTL32.dll
0x779f0000   0x00046000  C:\WINNT\system32\MSVCRT.dll

0x7FFA0000   PAGE_EXECUTE_READWRITE 12288 MEM_COMMIT Private
```

Win 32/Cabanas at the end the very end of the user address space.

```
PID: 0x004d

BaseAddress  Size        Name

0x002F0000  PAGE_EXECUTE_READWRITE 135168 MEM_COMMIT Private

0x01760000   0x00011000  C:\WINNT35\system32\NOTEPAD.EXE
0x77f80000   0x0004e000  C:\WINNT35\System32\ntdll.dll
0x77df0000   0x0002b000  C:\WINNT35\system32\comdlg32.dll
0x77f20000   0x00054000  C:\WINNT35\system32\KERNEL32.dll
0x77ea0000   0x00038000  C:\WINNT35\system32\USER32.dll
0x77ee0000   0x00033000  C:\WINNT35\system32\GDI32.dll
0x77e20000   0x0002c000  C:\WINNT35\system32\ADVAPI32.dll
0x77e60000   0x0003c000  C:\WINNT35\system32\RPCRT4.dll
0x77de0000   0x00010000  C:\WINNT35\system32\SHELL32.dll
0x77d70000   0x00034000  C:\WINNT35\system32\COMCTL32.dll
0x77db0000   0x00028000  C:\WINNT35\system32\CRTDLL.dll
```

Win32/Parvo (PID: 0x004d is the infected NOTEPAD.EXE).

```
PID: 0x003c

BaseAddress  Size        Name
0x01760000   0x00011000  C:\WINNT35\SYSTEM32\JWRK.EXE
0x77f80000   0x0004e000  C:\WINNT35\System32\ntdll.dll
0x77df0000   0x0002b000  C:\WINNT35\system32\comdlg32.dll
0x77f20000   0x00054000  C:\WINNT35\system32\KERNEL32.dll
0x77ea0000   0x00038000  C:\WINNT35\system32\USER32.dll
0x77ee0000   0x00033000  C:\WINNT35\system32\GDI32.dll
0x77e20000   0x0002c000  C:\WINNT35\system32\ADVAPI32.dll
0x77e60000   0x0003c000  C:\WINNT35\system32\RPCRT4.dll
0x77de0000   0x00010000  C:\WINNT35\system32\SHELL32.dll
0x77d70000   0x00034000  C:\WINNT35\SYSTEM32\COMCTL32.dll
0x77db0000   0x00028000  C:\WINNT35\system32\CRTDLL.dll
```

Win32/Parvo (original NOTEPAD.EXE running as JRWK.EXE).

```
PID: 0x0036

BaseAddress  Size        Name
0x00400000   0x0002b000  C:\WINNT\system32\drivers\ie403r.sys
0x77f60000   0x0005b000  C:\WINNT\System32\ntdll.dll
0x77f00000   0x0005c000  C:\WINNT\system32\KERNEL32.dll
0x77e70000   0x00053000  C:\WINNT\system32\USER32.dll
0x77ed0000   0x0002b000  C:\WINNT\system32\GDI32.dll
0x77dc0000   0x0003e000  C:\WINNT\system32\ADVAPI32.dll
0x77e20000   0x0004f000  C:\WINNT\system32\RPCRT4.dll
0x77720000   0x00011000  C:\WINNT\system32\MPR.dll
0x77e10000   0x00007000  C:\WINNT\system32\rpcltc1.dll
```

WinNT/RemEx running as IE403R.SYS service.

APPENDIX D

```
BaseAddr Name

0x77f60000 \WINNT\system32\NTDLL.DLL
0x80001000 Pcmcia.sys
0x8000b000 Disk.sys
0x80010000 \WINNT\System32\hal.dll
0x80100000 \WINNT\System32\ntoskrnl.exe
0x801e0000 \WINNT\System32\drivers\SCSIPORT.SYS
0x801e8000 \WINNT\System32\Drivers\CLASS2.SYS
0x801ec000 Ntfs.sys
0x80244000 TpPmPort.sys
0xa0000000 \??\C:\WINNT\system32\win32k.sys
.
0xf7000000 \SystemRoot\System32\Drivers\Cdfs.SYS
.
0xf72f0000 \SystemRoot\System32\Drivers\Cdrom.SYS
```

Partial list of loaded drivers and their base addresses.

APPENDIX E

```
0040F000 PAGE_EXECUTE_WRITECOPY 8192 MEM_COMMIT PAGE_READWRITE Image

0040f000 e9 21 00 00 00 b8 97 01 41 00 c3 b8 c1 03 41 00 T!...+ù.A.++-.A.
0040f010 c3 e9 ba 48 ff ff b8 06 00 00 00 c3 e9 bf 10 00 +T¦H  +....+T+..
0040f020 00 e9 d5 0e 00 00 e8 eb ff ff ff 50 e8 d4 ff ff .T+...Fd   PF+
.
.
00410190 d0 e9 5d ff ff ff 00 72 00 4e 49 43 4f 5f 56 49 -T]   .r.NICO_VI
004101a0 52 5f 4f 46 46 00 4b 45 52 4e 45 4c 33 32 00 47 R_OFF.KERNEL32.G
004101b0 65 74 45 6e 76 69 72 6f 6e 6d 65 6e 74 56 61 72 etEnvironmentVar
004101c0 69 61 62 6c 65 41 00 4e 49 43 4f 5f 56 49 52 5f iableA.NICO_VIR_
004101d0 43 48 49 4c 44 5f 4f 46 46 00 7b 00 00 00 43 72 CHILD_OFF.{...Cr
004101e0 65 61 74 65 54 68 72 65 61 64 00 47 6c 6f 62 61 eateThread.Globa
004101f0 6c 41 6c 6c 6f 63 00 6c 73 74 72 63 70 79 00 47 lAlloc.lstrcpy.G
00410200 6c 6f 62 61 6c 46 72 65 65 00 6c 73 74 72 63 6d lobalFree.lstrcm
00410210 70 69 00 5c 2a 2e 2a 00 6c 73 74 72 63 61 74 00 pi.\*.*.lstrcat.
00410220 46 69 6e 64 46 69 72 73 74 46 69 6c 65 41 00 2e FindFirstFileA..
```

Win32/Niko.5178 virus in an infected ASD.EXE application at page 0x0040F000.

```
117 asd.exe           Dtsactivation automatique (ASD)
   CWD:     C:\LOOK\
   CmdLine: C:\LOOK\ASD.EXE
   VirtualSize:    20152 KB   PeakVirtualSize:    20192 KB
   WorkingSetSize:  1604 KB   PeakWorkingSetSize:  1612 KB
   NumberOfThreads: 3
    122 Win32StartAddr:0x0040f000 LastErr:0x00000002 State:Waiting
     68 Win32StartAddr:0x0040f021 LastErr:0x00000002 State:Waiting
    123 Win32StartAddr:0x0040f01c LastErr:0x00000000 State:Waiting
    4.10.0.1998 shp  0x00400000  ASD.EXE
   4.0.1381.130 shp  0x77f60000  ntdll.dll
   4.0.1381.133 shp  0x77e70000  USER32.dll
   4.0.1381.133 shp  0x77f00000  KERNEL32.dll
```

Win32/Niko.5178 creates two threads (68 and 123 in this example). Entrypoint RVA is F000 and the image base is 400000. This is why thread 122 points into the virus code also.

APPENDIX F

```
0x77F5B000 PAGE_EXECUTE_WRITECOPY 16384 MEM_COMMIT Image

77f5e000 84 69 01 00 00 89 47 28 66 81 38 4d 5a 0f 85 52  äi...ëG(fü8MZ.àR
.
.
77f5e410 3f 01 75 06 3c 22 75 f6 eb 08 3c 20 74 04 0a c0 ?.u.<"u÷d.< t..+
77f5e420 75 ec c6 46 ff 00 8d 85 0c 15 40 00 89 47 08 e8 u8¦F .ìà..@.ëG.F
77f5e430 31 fb ff ff 57 ff 95 92 17 40 00 ff 95 92 17 40 1v  W òÆ.@. òÆ.@
77f5e440 00 c3 68 34 84 f1 77 9c 60 e8 0a ff ff ff 61 9d .+h4ä±w£`F.   a¥
77f5e450 c3 68 51 7f f1 77 9c 60 e8 56 ff ff ff 61 9d c3 +hQ ±w£`FV   a¥+
77f5e460 5b 48 65 72 65 74 69 63 5d 20 62 79 20 4d 65 6d [Heretic] by Mem
77f5e470 6f 72 79 20 4c 61 70 73 65 00 46 6f 72 20 6d 79 ory Lapse.For my
77f5e480 20 74 68 75 67 20 6e 69 67 67 61 7a 2e 2e 20 75  thug niggaz.. u
77f5e490 70 74 6f 77 6e 20 62 61 62 79 2c 20 75 70 74 6f ptown baby, upto
77f5e4a0 77 6e 2e 00 5c 4b 45 52 4e 45 4c 33 32 2e 44 4c wn..\KERNEL32.DL
77f5e4b0 4c 00 49 4d 41 47 45 48 4c 50 00 43 68 65 63 6b L.IMAGEHLP.Check
```

Win32/Heretic.1986.A at the end of infected KERNEL32.DLL in memory.

```
image base  77F00000

.
.
00015385    59  CreateNamedPipeA
000153FA    60  CreateNamedPipeW
00017DB6    61  CreatePipe
0005E451    62  CreateProcessA -> (77F5E451)
0005E442    63  CreateProcessW -> (77F5E442)
00004F9A    64  CreateRemoteThread
0001C893    65  CreateSemaphoreA
.
.
```

Win32/Heretic.1986.A modifies the export address of CreateProcess APIs.