CHASING GHOSTS? RETURN OF THE STEALTH MALWARE

Gergely Erdélyi
F-Secure Corporation Tammasaarenkatu 7
PL 24 00180, Helsinki, Finland

Tel +358 9 2520 0700 • Fax +358 9 2520 5001 •
Email • Gergely.Erdelyi@F-Secure.com

ABSTRACT

In the late DOS virus era stealth features in viruses were common. As most of the users – and with them virus writers – moved to Windows platforms stealth malware did not gain much room initially. This is about to change. Current malware is increasingly equipped with stealth features. Virus creators want their creations to go unnoticed, hackers want to operate on the compromised hosts without attracting attention. An active stealth malware might be able to conceal its presence by hiding processes, registry keys and open network ports. In order to hide their presence some of the recent viruses and Trojans employ sophisticated stealth components. These often operate on the kernel level to get maximum level of control. This kind of infection is often non-trivial to detect in a compromised system. This paper examines the stealth methods used by malware and possible techniques to detect their presence by manual inspection or an anti-virus program.

1. INTRODUCTION

1.1 Definition and purpose of stealth malware

Stealth is a program that deliberately tries to conceal its presence in the system. There are different types of stealth features depending on what they try to hide. Malware might try to hide changes it introduced to the system, including dropped files, file changes, running processes, registry settings and other traces of its activity.

1.2 History of stealth malware

The history of stealth computer viruses goes back to the very first PC virus: Brain. Brain was discovered in early 1986 and is the first known PC virus. Brain infects disk boot sectors and actively tries to cover up the changes it makes [1]. The DOS era of computer viruses brought a wide range of stealth viruses ranging from simple, semi-stealth viruses through very complex, full stealth ones [2]. Some of them contained really complex code to cover their tracks, even as extreme as monitoring disk I/O operations on a hardware level, like the Strange virus [3].

Windows 3.x series was never really a frequent target for viruses. The total number of viruses made specifically for *Windows 3.x* is not more than a few dozen. The platform was made obsolete by the vendor a long time ago, so it is not in the scope of this paper.

With the introduction of *Windows 95* the scene of stealth viruses changed quite a lot. *Windows 95* can use direct device drivers to access the disk – which rendered most of the stealth viruses incompatible. Incompatibility can easily reveal the presence of a virus, especially if it causes the system to hang or behave unexpectedly. It could be expected that virus writers would take this as a challenge and stealth code would be developed for *Windows 95*. For some reason, though, this did not happen. The number of stealth *Windows* malware is rather low today; we will look into some of the possible reasons in this paper.

When talking about *Windows* it is necessary to make a clear distinction between *Windows 9x* and *Windows NT*. The look of *Windows NT* – which is visually almost identical to *Windows 9x* – hides a completely different operating system. Developers of *NT* broke away from the legacy DOS base which resulted in an operating system that is similar to *Windows 9x* only in the Application Programming Interface (API) and Graphical User Interface (GUI) look-and-feel. From the stealth code developer's point of view, writing *NT*-compatible code is to rewrite the *9x* code almost from scratch.

When stealth features first appeared in computer viruses their main purpose was to make the work of anti-virus researchers and applications as difficult as possible. Today the apparent blending of virus writing and malicious hacking gives stealth code a whole new perspective. One of the most important things for any attacker after compromising a host on the Internet is to operate covertly on the host for as long as possible. This is where stealth code comes in: it allows the attacker to install different backdoors that the user will find hard to spot. Stealth backdoors allow the attacker to use the host for malicious purposes for a prolonged period.

2 SIMPLE STEALTH TECHNIQUES IN WINDOWS

2.1 Hiding behind complexity

Hiding something from the human eye in *Windows* does not always require cutting-edge code. Today's *Windows* systems are becoming exceedingly complex. A quick look at the system directory of *Windows XP* shows around 2000 files with a total size of 800 megabytes. Most of these files are

binary, executable code. This level of complexity challenges even trained techies, let alone the average user. By simply using some filename similar to system files it is likely that the malware will go unnoticed for a long time on an average user's computer. As an example, the difference between 'kernel32.exe' and 'kernel32.dll' might not be obvious at first sight, even to the trained eye. To complicate the problem further the purpose of many files in the system directory is either incompletely or not at all documented.

2.2 File system tricks

Similar techniques use different features in the standard *Windows* file manager. In order to make *Windows* more user-friendly the file manager tries to hide as much from the average user's eyes as possible. With latest versions of *Windows* even getting to the system directory is possible only after passing several annoying messages from the file manager. Even if the user has managed to reach the right folders, much of the available information is not shown by default. For example, file extensions are hidden by default and so are files that carry the 'hidden' and 'system' attribute. Whether or not these files and properties are shown is controlled by several values in the registry. These values are often modified by malware to lower the probability of being spotted. The registry key

```
HKEY_CURRENT_USER\Software\Microsoft\
Windows\CurrentVersion\Explorer\Advanced
```

contains several values to control the behaviour of File Manager with hidden files – see Table 1 (below).

One example of a virus using this technique is Nimda which modifies all these values so that *Explorer* does not show any type of hidden files [4].

Similarly, several 'optical' tricks can be used which are based on the fact that some fonts have similar-looking characters. It might be hard to spot the difference between 'kernel32.dll' and 'kerne132.dll' or 'IMPORTANT.DLL' and 'IMP0RTANT.DLL'. Whitespace characters like Tab and Space can be inserted into filenames, creating optically similar names which are, of course, still different for the operating system.

2.3 Hiding processes as services

A simple way of hiding running processes from standard tools like Task Manager is to turn the program into a service. *Windows 9x* does not make a clear distinction between regular and service processes. By calling RegisterServiceProcess() from KERNEL32.DLL the process becomes a service, it disappears from Task Manager and will not be shut down when the user logs off.

Windows NT makes this a bit more complex. Under NT services are controlled by Service Control Manager (SCM) which has a few extra requirements for services. New services have to be registered with SCM using the following code sequence:

```
OpenSCManager();
CreateService();
StartService();
CloseServiceHandle();
```

When started the service process has to contact SCM and register a control handler function to be able to receive control messages. This is done with the RegisterServiceCtrlHandlerEx() function. When a service is created with the proper parameters the operating system will automatically start and stop it when needed. Service processes are not visible in Task Manager under *Windows NT* either.

3 USER SPACE STEALTH CODE

The main idea behind stealth features is that calls to system services are diverted to the malware. This diversion is usually called 'hooking'. By diverting the calls the malware can alter the way in which the system service behaves. Altering can be filtering of certain data (like hiding filenames in a directory list) or even complete removal of functionality. This section concentrates mainly on *Windows NT*. Hooking methods used on *NT* could be applied to *Windows 9x* as well, but the ways hooks are installed into processes would be different due to lack of certain API calls in *Windows 9x* systems.

3.1 Hooking techniques

3.1.1 Import Address Table modification

In *Windows*' Portable Executable (PE) format designers decided to implement dynamic symbol loading by using

Table 1

Registry Value	It controls...
Hidden	whether files with hidden attributes are shown
ShowSuperHidden	whether files with system and hidden attributes are shown
HideFileExt	whether the file extension is shown

indirect addresses. Any call to external functions is compiled so that the CALL uses a memory address to take the call address from. When the operating system loads the executable it resolves all the external symbols and writes their addresses to these memory locations. This method makes the resolving of external symbols efficient because there is only one place to modify when one symbol is imported. Import Address Table (IAT) hooking uses this feature. By directly altering the imported addresses in IAT, API calls are redirected to the malware. There is no need for code modification as with the other methods we are going to look into shortly.

3.1.2 Dynamic code patching

A more direct way of hooking is the direct modification of the code in the API functions. Most commonly the first few bytes of the function are overwritten with a JMP instruction to the replacement function. In practice this usually requires five bytes: the JMP instruction and a 32-bit address.

Typically the original function is called from the hook. Because of that the hook has to restore the first bytes of the original function. When the original function returns the hook patches the first bytes again. The code looks rather simple:

```
ORIGINAL_Function:
JMP  Hooker
<rest of the original code>

NEW_Function()
{
Process_Arguments();
Restore_First_Bytes(Hooked_Function);
Hooked_Function();
Alter_Data();
Patch_First_Bytes(Hooked_Function);
}
```

This method is prone to errors with threads and synchronization. If the code is patched while another thread is using it, the hook might miss some of the calls or even crash.

3.1.3 Code patching with instruction analysis

Elimination of the main weakness of the dynamic code patching requires a more complex approach. The main problem with the previous method is the need of restoring

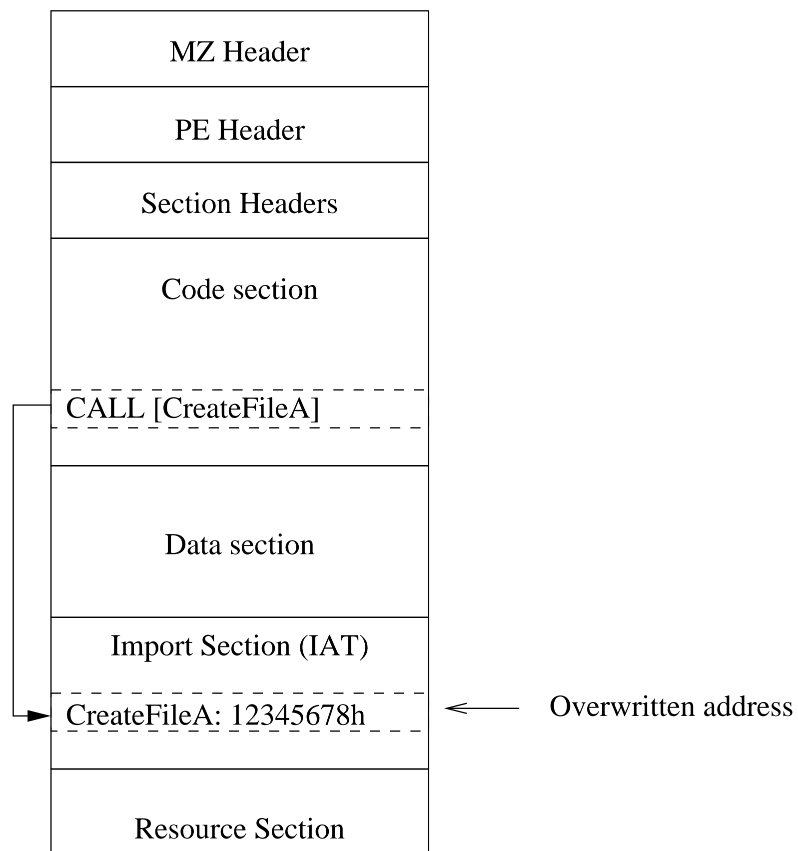

Fig 1: IAT patching in PE files

the original code before it can be called. Restoring is not necessary if full instructions are copied instead of just bytes. This requires a module that can identify the length of different instructions from the binary code. Using this module a few full instructions are copied to make space for the JMP. Let look at the hooking of FindNextFileA() as an example:

```
FindNextFileA:
000195D6: 55           PUSH EPB       ;
000195D7: 8BEC         MOV  EBP, ESP  ; These
                                        instructions
000195D9: 81EC60020000 SUB  ESP, 260  ; will be
                                        copied
Continue_Here:
000195DF: 53           PUSH EBX
000195E0: 8D85A0FDFFFF LEA  EAX, [EBP-260]
```

When the number of instructions have been determined they are copied to a buffer that ends in a JMP instruction pointing to the next instruction after the ones just copied. The code has the following structure:

```
FindNextFileA:
000195D6: E9XXXXXXXX   JMP  Hook      ;
000195DB: 90           NOP            ; These are
                                        just fill-in
000195DB: 90           NOP            ; bytes
000195DB: 90           NOP            ;
000195DB: 90           NOP            ;
Continue_Here:
000195DF: 53           PUSH EBX       ; Orignal code
000195E0: 8D85A0FDFFFF LEA  EAX, [EBP-260]
000195DF: XX           <...original code continues...>

Saved_Original:
00020000: 55           PUSH EBP
00020001: 8BEC         MOV  EBP, ESP
00020003: 81EC60020000 SUB  ESP, 260
00020009: E9XXXXXXXX   JMP  Continue_Here

Hook:
                       <process params>
                       call  Saved_Original
                       <alter data>
                       ret
```

When the hook needs to call the original function it calls the assembled ‘side track’ that first executes the first instructions of the original code then jumps to the rest which has not been patched.

3.2 Installing the hooks into processes on NT

Stealth techniques working in user space require hooks to be installed into all the processes. *Windows* has powerful features and API calls for debugging. Abusing these hooks can be installed into other processes as well. Processes having the proper privileges are capable of injecting and executing code in any other running process.

3.2.1 DLL injection

One of these API calls is CreateRemoteThread() that has the following prototype [5]:

```
HANDLE  CreateRemoteThread(
        HANDLE hProcess,
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        SIZE_T dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddress,
        LPVOID lpParameter,
        DWORD dwCreationFlags,
        LPDWORD lpThreadId );
```

CreateRemoteThread() starts a thread in the address space process specified with hProcess. The only constraint is that the function to which lpStartAddress points must already exist in the remote process. There are, however, different ways of making sure that the code is there. One of them is called ‘DLL Injection’.

To be able to load the required external symbols all processes have LoadLibrary() API in their address space. Abusing this fact lpStartAddress can point to LoadLibary(“Nasty_hook.dll”) call which will cause ‘Nasty.dll’ to be loaded to the remote process’ address space. After loading the DLL the linker will automatically call DllMain(), which is exported from every DLL. DllMain() is responsible for initializing the DLL when it is loaded and cleaning it up when unloaded. In the case of ‘Nasty_hook.dll’ it will install the API hooks using any of the methods described earlier.

3.2.2 Direct memory writing

DLL Injection is not the only way of modifying other processes. VirtualAllocEx() and WriteProcessMemory() functions help to inject code into any remote process. VirtualAllocEx() allocates memory in the address space of another process, followed by WriteProcessMemory() that copies the necessary code into the process.

The copied code is then started with CreateRemoteThread() just like with DLL Injection. Because the memory area allocated by VirtualAllocEx() can be anywhere in the remote process the hook routines have to be position-independent, which is more complicated to write properly. Position-independent code is quite common in viruses already so unfortunately it will not stop this method from being used.

4 KERNEL SPACE STEALTH CODE

In any operating system code running in the kernel space

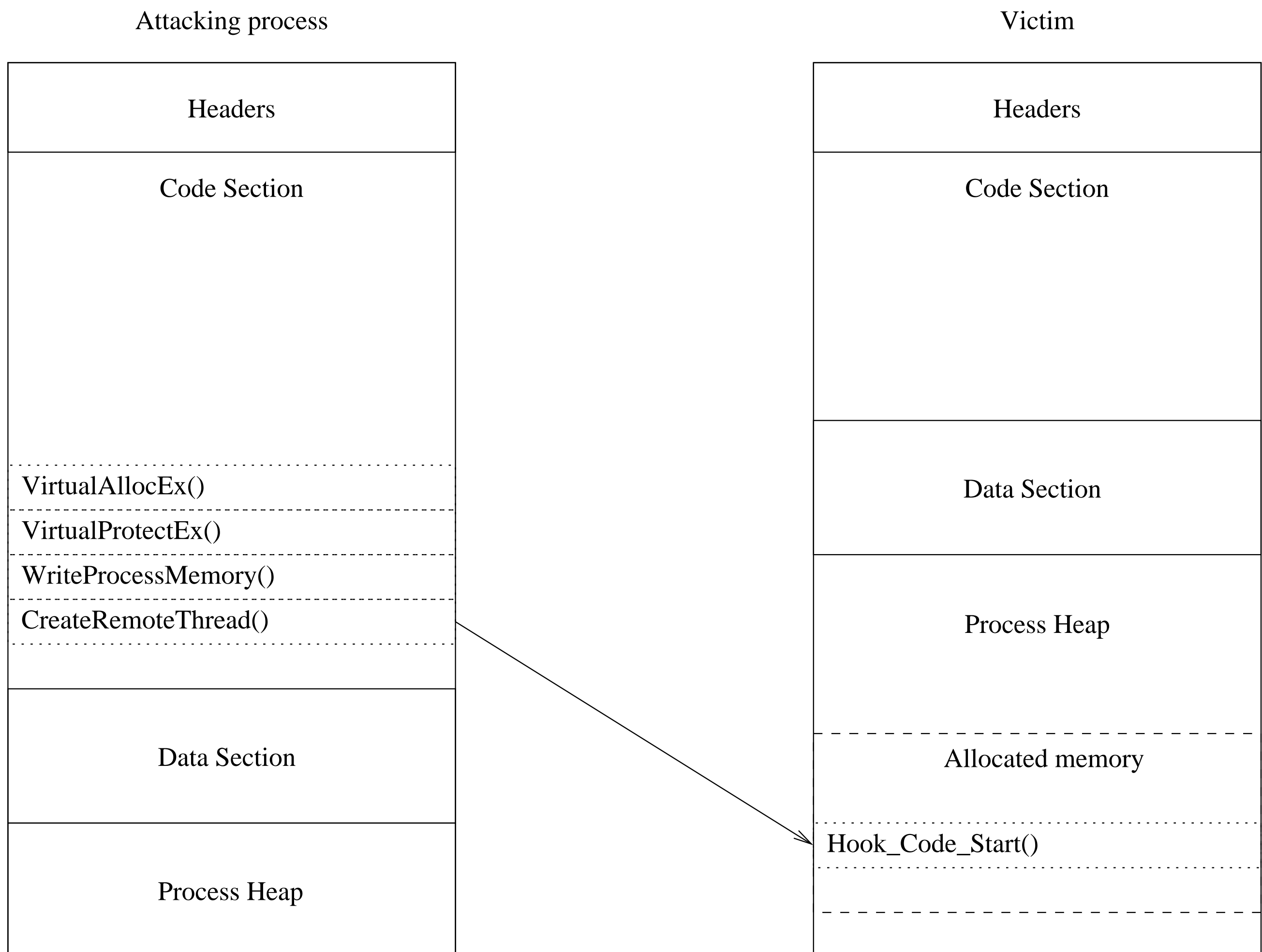

Fig 2: Patching a remote process

has more power than in user space. Kernel space hooking gives more control and possibilities to hide. Moving the hooks from the user to kernel space was a natural and anticipated step.

4.1 Kernel space hooking in Windows 9x/ME

4.1.1 Hooking the file system

Hooking the file system under *Windows 9x* is conceptually similar to hooking DOS file system calls. *Windows 9x* provides a standard mechanism for adding new file system services which is called Installable File System Services (IFS). Using IFS it is possible to add routines to the IFS chain that filter all the file operations in the system. The following service call installs a new hook into the file system service chain:

```
IFSMgr_InstallFileSystemApiHook(pIFSFileHookFunc
HookFunc)
```

where HookFunc is the address of a single function that handles all the file system service operations. The hook function is declared with the following prototype:

```
FileSystemApiHookFunction(
    pIFSFunc FSDFnAddr,
    int FunctionNum,
    int Drive,
    int ResourceFlags,
    int CodePage,
    pioreq pir
)
```

The most important parameter is FunctionNum which specifies the operation to be carried out.

Malware can handle and filter the necessary operations to hide its traces in the filesystem. The most commonly hooked operations are as follows:

- IFSFN_OPEN
- IFSFN_CLOSE
- IFSFN_READ
- IFSFN_WRITE
- IFSFN_SEEK
- IFSFN_SEARCH
- IFSFN_ENUMHANDLE
- IFSFN_FINDOPEN
- IFSFN_FINDNEXT

4.1.2 Hooking Registry and other API calls

In order to hide a malware completely, covering up the file system changes is not enough. For the malware to be hard to detect it is necessary to hide processes and registry keys as well. *Windows 9x* provides the means for this too in its Virtual Machine Manager (VMM). The VMM service Hook_Device_Service makes it possible to install hooks for VMM service calls on the kernel level. The *Windows 9x* Device Driver Kit (DDK) contains a header file called vmm.h that defines the VMM services that can be hooked.

4.2 Installing the hook drivers on Windows 9x

Just like in user space the hooks must put in place in kernel space as well. The following section will examine how it is done in *Windows 9x*.

4.2.1 Loading a VxD

The standard way of installing drivers into *Windows 9x* kernel is to create a VxD of the code and load it. Loading can be done by using CreateFile() API call with the VxD name in the form of '\\.\nasty_hook.vxd'. If the FILE_FLAG_DELETE_ON_CLOSE flag is not set when opening the VxD it will not be removed from the memory even if a CloseHandle() is issued on the VxD's handle.

4.2.2 Ring3 to Ring0 jump

Viruses usually avoid using external files for special purposes and take a more direct approach. *Windows 9x* uses two levels of privileges:

- Ring3 – User space, where the applications run
- Ring0 – Kernel space, where the kernel drivers run

Even though the idea is that user space applications do not have access to kernel space, code tricks exist to make a code jump from Ring3 to Ring0. The Zerg is one of the viruses that use such trick. Zerg was the first known *Windows 9x* full stealth virus but it did not succeed because of serious bugs in its code. This virus is capable of hiding the size change of the infected file as well as the actual infection by altering the data returned from the file system operations.

IFS routines have to run in Ring0 so the virus – which starts in user space – has to move its code to kernel level. Zerg accomplishes this by first setting up a Structured Exception Handler and modifying the IDT (Interrupt Description Table) to point Int0 (Division By Zero) to itself. Causing a 'Division By Zero' exception the virus gets the control in the kernel. After reaching Ring0 it allocates kernel memory, copies itself there and installs the IFS hook.

A similar approach is taken the Sma virus. Sma modifies the Global Descriptor Table (GDT) to have its own descriptor entry (1F0h), thereby getting access to kernel space [6].

4.3 Kernel space hooking in Windows NT/XP

Windows NT is built around a flexible, extendible microkernel architecture. The system is implemented in several layers. The lowest layers are the basic kernel services and the Hardware Abstraction Layer (HAL). The user level subsystems are built on the top of these. *NT* supports several parallel user subsystems which can run at the same time separately. These subsystems are Win32, OS/2, POSIX. Standard Win32 applications run in the Win32 subsystem and use the Win32 API. Under the hood the Win32 API calls are only wrappers around a set of lower level API called the Native NT API.

The Native API is mostly undocumented by the vendor but there is a fair amount of information available on the Internet [7]. The main reason why many people are interested in the Native NT API is the power it has. Possessing detailed knowledge on this API would make it possible to build new subsystems for *NT* which can coexist with Win32 or even replace it completely. The other interesting fact is that, by hooking the native API, a global control can be achieved, making it possible to hide something from all the subsystems

The Native NT API is – not surprisingly – quite similar to the Win32 API. Closely examining certain API calls we find that Win32 versions are basically wrappers around the Native API calls. These wrappers do some extra parameter checking/transformation and call the Native counterparts. It should be mentioned here that some Win32 calls do not expose all the functionality and parameters of the Native calls.

Figure 3 shows the call flow of an example Win32 API. From the diagram it can be seen that hooking in the kernel space is really effective because changes can be hidden

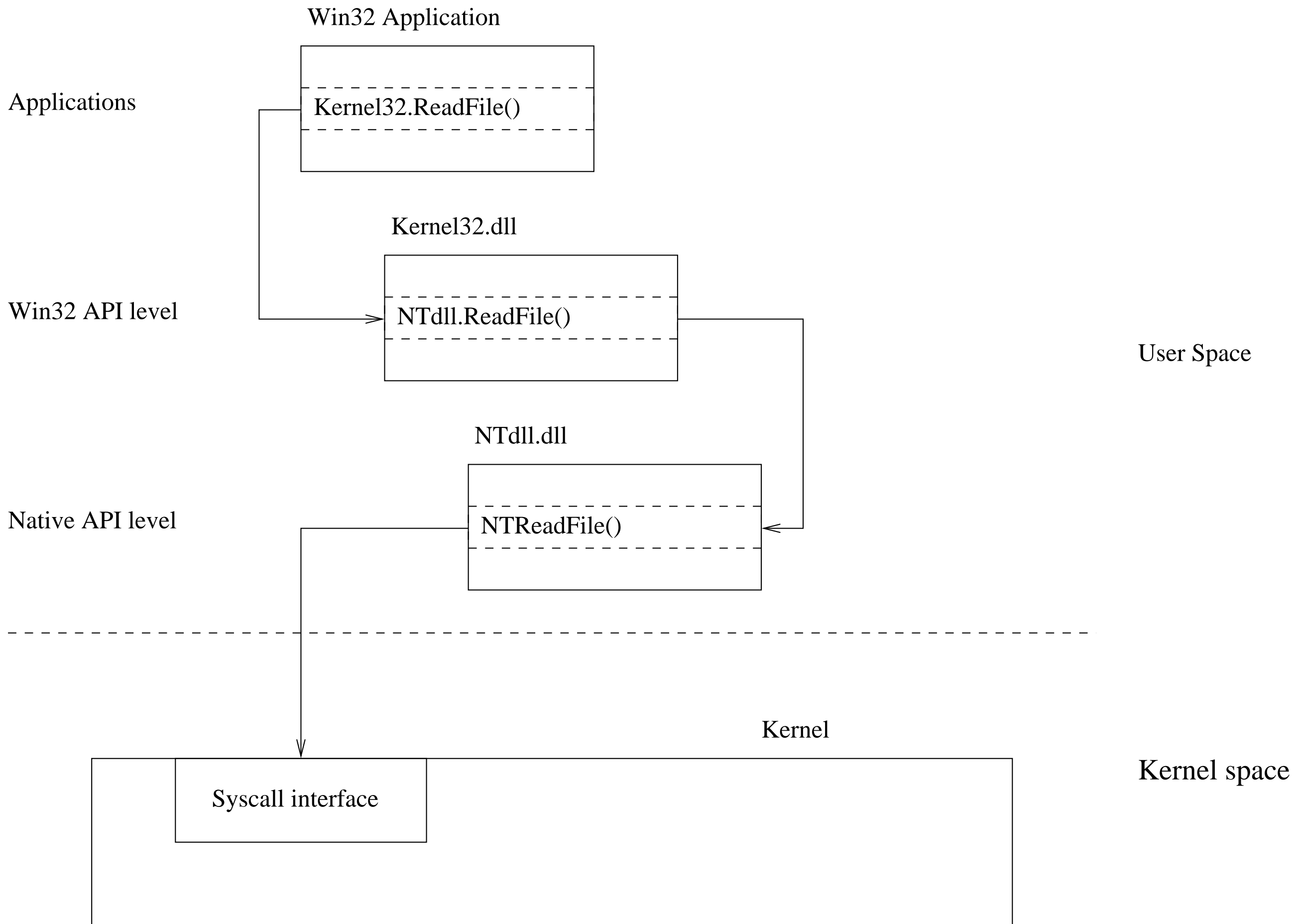

Fig 3: Example Win32 API call.

from all the subsystems. We will now look into different methods of implementing hooking in kernel space.

4.3.1 Altering the Kernel Service Table

Now we have to take a closer look into how the syscall interface in *NT* is implemented. The most important part from this point of view is the way *NT* kernel stores the routines behind different kernel services. In case of *NT* this is stored in a central place called System Service Descriptor Table (SSDT). This descriptor table is an array of special structures called System Service Descriptor (SSD).

SSD is defined as a simple structure:

```
typedef struct
{
    void     *lpfnHandlerTable; // Pointer to function
                                   pointer table
    PULONG    ulCounterTable;   //
    ULONG     cServices;        // Number of services
    void     *rguchParamTable;  // Pointer to function
                                   parameter table
} SSD;

/* System Service Descriptor Table */SSD
KeServiceDescriptorTable[4];
```

System services are divided into four groups:

- 0 – Core services (exported from NTDLL.DLL)
- 1 – GUI services
- 2 – Reserved
- 3 – Reserved

This information is sufficient to be able to hook any system service listed in SSDT [8]. The actual hooking goes as follows:

```
hook_service()
{
        /* Store the old address */
        old_addr = lpfnHandlerTable[SERVICE_NO];

        /* Insert the new one */
        lpfnHandlerTable[SERVICE_NO] = hook_function;
}
```

```
hook_function(args)
{
        /* Do argument processing */
        process_arguments();

        /* Call the original function */
        old_address(args);

        /* Filter out the data to be hidden */
        filter_data();
}
```

In order to stop this method from being used *Microsoft* made the System Descriptor Table read-only in *Windows XP*. However, their move did not stop anyone because the protection can be overridden. By disabling the the WP bit in the processor's CR0 register the write protection is rendered ineffective. Disabling and enabling of WP requires only a few assembly instructions. Hooking by modifying the Kernel Service Table is used by several publicly available packages, HE4Hook and KApiHooks for example.

4.3.2 Kernel object modification

Altering the Kernel Service Table already uses undocumented data structures in the *NT* kernel. The 'Fu' rootkit takes this one step further. Fu modifies different objects in the kernel. Altering certain kernel structures directly it is capable of hiding or give extra privileges to the caller process. Fu has a hard-coded table of offsets for different *Windows NT* versions starting from *NT 4.0* up to *Windows XP*. Since these offsets change from version to version it can be expected that Fu will stop working in new versions of *Windows* or even service packs.

4.4 Hook installation in NT kernel

Windows NT has a significantly different driver model from *Windows 9x*. This section looks into the ways how kernel drivers are installed in *Windows NT*-based systems.

4.4.1 Standard device drivers

The simplest – and only documented – way of getting the hooks into the kernel is by installing a standard driver. Driver installation goes the same way as with any other service as described in Section 2.3. The only difference is that service type parameter of CreateService() is set to SERVICE_KERNEL_DRIVER which makes the driver to be installed in the kernel. When the service is registered *Windows* will take care of loading and unloading the driver at system startup and shutdown, this way activating the stealth component.

4.4.2 Using SystemLoadAndCallImage

Using SystemLoadAndCallImage is a more obscure and completely undocumented way of loading kernel drivers in *NT*. Native API has a function call NtSetSystemInformation() which is used to set certain system parameters. One of the undocumented features of this call is to load and unload kernel drivers. The prototype of this call is rather universal:

```
NtSetSystemInformation(
   IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
   IN PVOID SystemInformation,
   IN ULONG SystemInformationLength
);
```

where SystemInformationClass specifies the parameter class to be set, SystemInformation is a pointer to the parameter and SystemInformationLength tells the length of the parameter. SystemLoadAndCallImage expects a unicode path to a driver file to be loaded. The kernel loads the specified file and initializes it as a driver in a running system.

5 HOW REAL MALWARE USES THESE TECHNIQUES

First of all it should be mentioned here that the techniques described above are not malicious by nature. Different kinds of legitimate application use these or similar methods to achieve their goals. Monitoring software (Filemon, Regmon, etc.) from http://www.Sysinternals.com/ is widely known and serves as a good example.

However – as with everything else – these techniques can be used maliciously as well. Stealth features described in this paper are mainly used in rootkits at the moment. Since rootkits (unlike most viruses) consist of many files usually they can keep the stealth functionality separately. Rootkits are standalone programs so file system stealth code hides files only, there is no need to hide parts of files or any special file content. The most apparent manifestations of the rootkits are hidden, such as:

- files and directories
- registry keys and values
- running processes
- services
- open network ports

Some backdoors come with a kernel and a separate user space component: Ierk, He4Hook. Different ones use different ways of communicating between the components. The most common ways are the usage of IOCTL calls and Mail Slots. One complex example of stealth backdoor is the Hacker Defender package which is also known as Backdoor.HacDef or Hxdef. Hxdef uses full stealth techniques, hides files, processes, services and registry keys. What makes it unique is the way it implements the backdoor functionality. Hxdef does not open any port in the system.

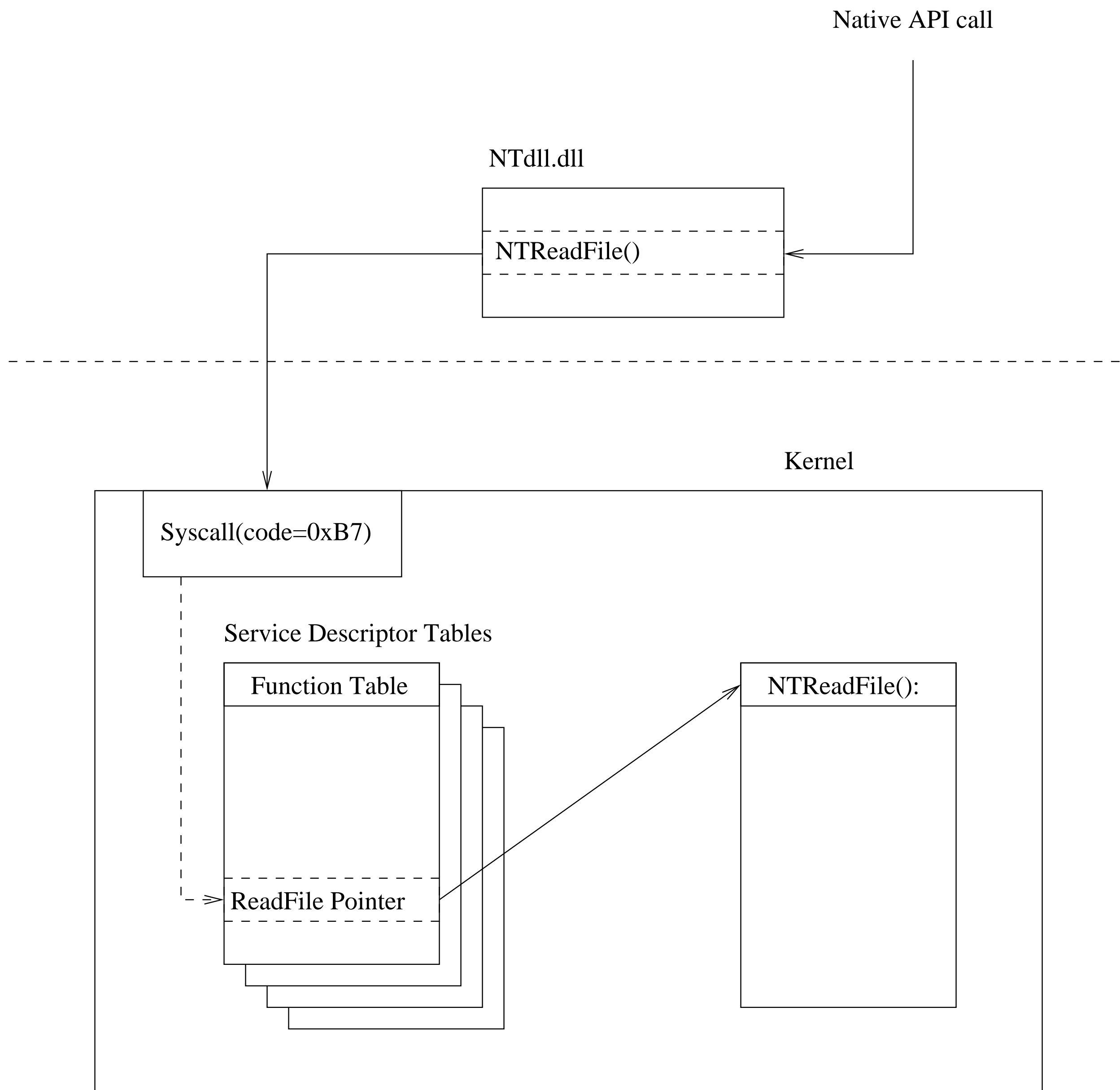

Fig 4: Usage of System Service Description Table.

Instead it captures network traffic on already open ports and looks for specially crafted packets that carry the commands to the backdoor. This makes it even more difficult to detect on the network because standard ports – especially Windows Network ports – are rarely monitored.

6 DETECTION

Detection of stealth malware with anti-virus or even manual inspection can be difficult. Depending how much of the traces the malware tries to hide and how effective tricks it uses this can range from trivial to really challenging.

6.1 User space stealth

Malware which uses user level hooking might be problematic to detect with user space tools but really easy with kernel space virus scanners.

Since user space techniques do hooking mostly in the Win32 subsystem special tools using the native API can detect them. *Microsoft* provides a collection of small utilities, called Resource Kit for administrators and advanced users. Some of those tools can be used to inspect the system in more detail and a lower level than standard tools that come with *Windows*.

6.2 Kernel space stealth

Kernel space components are more challenging. There are different scenarios depending on whether the anti-virus was loaded first or the malware. In most cases the presence of the malware can be detected by checking the presence of the communication channel the kernel component uses. This might be a device or a mail slot with a specific name. Knowing how the kernel components work a well controlled challenge-response with them can make the detection more reliable.

Trojans using kernel space components are generally hard to find when they are active. Some of them that do are not properly hidden might still be spotted using special tools. As an example the 'Fu' Trojan hiding kit can be detected even when it is active. In Resource Kit there is a small utility called 'sc.exe'. This tool allows the administrator to check services by talking directly to the Service Controller. [9]. Using this tool the kernel driver of 'Fu' can be detected:

```
c:\sc.exe query type= driver bufsize= 4096 > drivers.txt
```

By checking the resulting file which contains the list of drivers in the system, the following is found:

```
SERVICE_NAME: fu_test
DISPLAY_NAME: fu_test
                TYPE                       : 1
KERNEL_DRIVER
                STATE              : 4  RUNNING

(STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
                WIN32_EXIT_CODE    : 0  (0x0)
                SERVICE_EXIT_CODE  : 0  (0x0)
                CHECKPOINT         : 0x0
                WAIT_HINT          : 0x0
```

This clearly shows that there is something suspicious and the computer needs a closer inspection.

6.3 Clean booting

The most reliable way of detecting stealth malware is not to allow it to become stealth. This can be accomplished by clean booting the system.

In the case of *Windows 9x* it is relatively simple. These systems can be booted to DOS mode where the *Windows* drivers and application are not loaded at all. Up to and including *Windows 98* the system can be booted to DOS mode from a boot menu. In *Windows ME* this option was sadly removed. In *Windows ME* it is still possible to boot the system is DOS mode but only using external tools, like a boot floppy from older version of *Windows*.

Clean booting *NT*-based systems is even more challenging. Even if the system is booted in Safe Mode several low lever drivers are loaded that might include the malicious driver. Somewhat complicated but more reliable way of clean booting is to have an emergency copy of *NT* on a different partition or disk. For inspection the system can be booted using the emergency copy and the main partitions can be checked [10].

7 CONCLUSION

Stealth features in recent malware are quite rare. The main driving force in the development of stealth code is the area of rootkits. Considering that the knowledge and even the source code is available, stealth functionality in malware is highly expected to start showing up more frequently, some day in the – hopefully not near – future. Emerging technologies, like *Microsoft*'s *Next-Generation Secure Computing Base for Windows* (formerly known as Palladium) might change the fate of all malware, including stealth malware. The effectiveness of these remains to be seen. Until then, anti-virus and other security experts must be prepared to fight them.

REFERENCES

[1] F-Secure Antivirus Research Team, Analysis of the Brain virus, http://www.f-secure.com/v-descs/brain.shtml.

[2] Vesselin Bontchev, *Methodology of Computer Anti-Virus Research*, University of Hamburg, 1998.

[3] Kaspersky Labs, Analysis of the Strange virus, http://www.viruslist.com/eng/viruslist.html?id=2836.

[4] F-Secure Antivirus Research Team, Analysis of the Nimda worm, http://www.f-secure.com/v-descs/nimda.shtml.

[5] Microsoft Developer Network http://msdn.microsoft.com/.

[6] Péter Ször, 'Stealth Survival' – analysis of the Sma virus, *Virus Bulletin*, July 2002.

[7] Tomasz Nowak, 'Undocumented Functions for Microsoft Windows NT/2000', http://undocumented.ntinternals.net/.

[8] Tim J. Robbins, 'Windows NT System Service Table Hooking', http://www.wiretapped.net/~fyre/sst.html.

[9] Microsoft TechNet, Manual of sc.exe, http://www.microsoft.com/TechNet/prodtechnol/winxppro/proddocs/sc.asp.

[10] Lucijan Caric and Tomo Sombolac: 'Booting the Unbootable', *Proc. Int. Virus Bull. Conf.*, 2002.

[11] Microsoft 'Palladium': A Business Overview, http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp.