HIDE 'N SEEK REVISITED – FULL STEALTH IS BACK

Kimmo Kasslin, Mika Ståhlberg, Samuli Larvala and Antti Tikkanen
F-Secure Corporation, Tammasaarenkatu 7, 00181, Helsinki, Finland

Tel +358 9 2520 0700 • Fax +358 9 2520 5001 •
Email kimmo.kasslin@f-secure.com
mika.stahlberg@f-secure.com
samuli.larvala@f-secure.com
antti.tikkanen@f-secure.com

ABSTRACT

Rootkits are designed to hide information. They are no longer utilized only by highly skilled individuals targeting UNIX machines. Advanced *Windows* rootkits have emerged and are gaining popularity among intruders. The alarming news is that malware writers are adopting rootkit techniques, which allows them to create a new breed of worms, Trojans and spyware that are able to avoid detection by hiding their presence on the system.

Traditional anti-virus and intrusion detection systems are powerless against this emerging threat, since they rely on the validity of the information provided by the operating system. This information cannot be trusted if the kernel or the application programming interfaces are modified by malware.

This paper is a continuation of the academic research done by one of the authors [1]. It provides an introduction to the hiding techniques utilized by advanced *Windows* rootkits. This information is essential for understanding the threat and for fighting it. In addition, new techniques for detecting hidden objects are presented. They form the foundation for the next generation of detection tools. Finally, the paper presents and analyses a new application that brings rootkit detection onto the desktop of home users.

1. INTRODUCTION

Stealth viruses have been around for almost two decades now. Their era on *Microsoft* systems seemed to end with the introduction of *Windows 95*, but after a dormant period they are now back in the form of *Windows* rootkits.

Brain [2], the first PC virus from 1986, was also the first stealth virus. It installed hooks on disk interrupt handling in order to hide its existence. When an attempt was made to read an infected boot sector, the virus would return the original boot sector instead.

Stealth viruses almost died out completely when 32-bit *Windows* became the dominant operating system. There are probably several reasons for this, but the most important is that *Windows* users do not remember the sizes of system files [3]. Virus writers concentrated their efforts on other technologies and stealth was forgotten. Well, almost forgotten.

Cabanas [4] from 1997 is renowned because it was the first real *Windows NT*-compatible virus. Cabanas was a semi-stealth virus that hid the change in size of infected files by hooking FindFirstFile and FindNextFile API functions in the Import Address Table (IAT). In this sense Cabanas was the forefather of today's *Windows* rootkits that commonly hook these same functions.

Today, parasitic viruses – malware that infect other files – are less and less common. The most common form of malware is a stand-alone application. These come in various flavors: worms, Trojans, bots, etc. Stealth in this context means that a malicious application is able to hide itself from system monitoring tools and even anti-virus scanners. When we talk about stealth in malware today, we are talking about rootkit techniques.

Current *Windows* rootkits generally hide at least one of the following object types: processes, files and registry keys. In our experience, processes are currently the most common object type that rootkits hide. This is a little surprising since, in theory, a rootkit does not need user-mode processes. The reasons for the current situation are most likely the following:

- *Windows* has a huge number of files under system directories. No user knows them all. On the other hand, a typical *Windows* machine only has around 50 processes. System administrators are likely to notice strange new processes on their system.
- Rootkits are commonly used to hide spyware, bots, FTP-servers, etc. A rootkit that cannot hide other applications from the task manager is not very useful for real-life attackers.
- Many security applications rely on the integrity of the process list. These applications might not be able to detect malicious activities of hidden processes. For example, some anti-virus products have memory scanning functionality that is not able to scan hidden processes.

In this paper we will present some of the hiding techniques used by *Windows* rootkits. In particular, we concentrate on techniques that are used to hide malware from anti-virus software. We will also introduce a generic approach to detecting rootkits. Specifically, we will demonstrate its applicability in detecting hidden files and processes. For hidden registry key detection, the reader is advised to refer to the recently published paper by Wang *et al.* [5]. When *Windows* operating systems are discussed in this paper, we are referring to 32-bit versions of *Windows 2000*, *XP* and *Server 2003*.

This paper is not about preventing the operation of rootkits. Rootkits can be fought effectively with behaviour blocking software, but this is out of the scope of this paper. Incidentally, most of the basic technologies, such as API hooking, used by these behaviour blockers are the same technologies that rootkits use to hide themselves.

2. HIDING TECHNIQUES

2.1. General idea

Stealth malware tries to hide its presence from users of the infected computer. It does this by hiding any processes, threads, files, registry entries, handles and open ports it has on the system. This usually means that malware has to intercept requests of system information sent by administration and security tools.

Normally, the request for information originates from a user-mode application. It sends the request by calling the appropriate *Windows* API function. Later, the request is handled by the kernel, which collects the information from various data structures it maintains. Eventually, the requested information is sent back via the same route it came. This route is known as the execution path.

Stealth malware is able to hide information if it can divert the execution path to go through a special filter function. This is also known as hooking. The sole purpose of the function is to filter out the hidden information from data going through it. Depending on the situation, it can modify the data either before or after the request has been handled.

Filtering can be performed in user mode or in kernel mode. Generally, user-mode filtering is done by hooking the corresponding *Windows* or Native API functions that are responsible for retrieving the requested information. User-mode filtering is easier to implement because *Windows* API provides various documented functions that allow a malicious process to install its hooks on any process in the system. The biggest disadvantage of user-mode filtering is that usually a malicious process has to install its hooks on every process to create a system-wide filter. This is because each process has its own private memory space [6].

Kernel-mode filtering filters the information while the thread is executing in kernel mode. Since every process shares the same system address space [6], system-wide filtering can be achieved by installing only a single set of hooks. However, kernel-mode filtering is more demanding since less documentation is available and a single error can cause the whole system to crash.

2.2. Places to hook

Hooking and similar techniques have been used for decades by experienced system developers for program tracing and code instrumentation purposes. In *Windows* environments, there are several techniques that can be used to hook binary code. Below is a list of the most widely used techniques:

- Inline hooking
- System Service Table (SST) hooking
- Import Address Table (IAT) hooking
- Export Address Table (EAT) hooking
- Interrupt Descriptor Table (IDT) hooking
- I/O Request Packet (IRP) major function hooking
- Filter drivers

These techniques are quite well documented but the information is scattered around. There are, however, two academic papers [1, 7] that have collected most of the necessary information together. It is not possible to explain all these techniques in the context of this paper. Only the two most common techniques, inline hooking and SST hooking, are briefly introduced. The reader is advised to refer to the mentioned papers for more information.

2.2.1. Inline hooking

Inline hooking is the most widely used technique for intercepting function calls. It has mostly been used for user-mode hooking but it works with kernel-mode function calls as well. Inline hooking is based on patching in-memory functions to divert their execution path to a location controlled by the hooking entity. It is a powerful technique because it can intercept every function call, regardless of the way it is called. Currently, the most widely used technique is known as the *Detours* technique, which was presented by Hunt and Brubacher [8]. Their solution was special because it was the first one that preserved the logic of the original function.

The generic idea is illustrated in Figure 1. The first few instructions of the target function are replaced with a JMP instruction pointing to the user-supplied detour function. The replaced instructions are preserved in a trampoline function. The trampoline consists of the instructions removed from the target function and a JMP instruction pointing to the remainder of the target function. When execution reaches the target function, control jumps directly to the user-supplied detour function [8].

The detour function consists of three separate sections: the prolog part, the call to the trampoline function and the epilog part. First, the prolog part performs whatever preprocessing is appropriate. Then, the trampoline function is invoked with the CALL instruction. This allows the detour function to regain control after the trampoline function returns. The trampoline function runs the unpatched version of the target function. Finally, the epilog part performs appropriate post processing and returns control to the source function [8].

Inline hooking has been widely adopted by recent stealth malware as can be seen from Table 1. For example, the Haxdoor.al [9] backdoor hooks functions in several system DLLs that allow it to hook other DLLs, hide open ports and filter HTTP traffic. An interesting observation is that, currently, inline hooking is done only in user mode. The most

	Hides				
	Files	Processes	Reg. Keys	Injection Method	Hooking Method
Backdoor.Win32.Lecna.a [7]	Yes	Yes	Yes	KM Driver	SST
Backdoor.Win32.Padodor.w [11]	No	Yes	No	Physical Memory	UM Inline
Trojan-Spy.Win32.Qukart.w [11]	Yes	Yes	No	Physical Memory	UM Inline
Backdoor.Win32.Hupigon.j [24]	Yes	Yes	No	KM Driver	UM Inline
Backdoor.Win32.Haxdoor.al [8]	Yes	Yes	No	KM Driver	SST + UM Inline
Worm.Win32.Myfip.h [10]	No	Yes	No	Physical Memory	DKOM
Net-Worm.Win32.Maslan.a [9]	Yes	Yes	No	UM DLL	IAT

Table 1. Some examples of recent stealth malware and their properties.

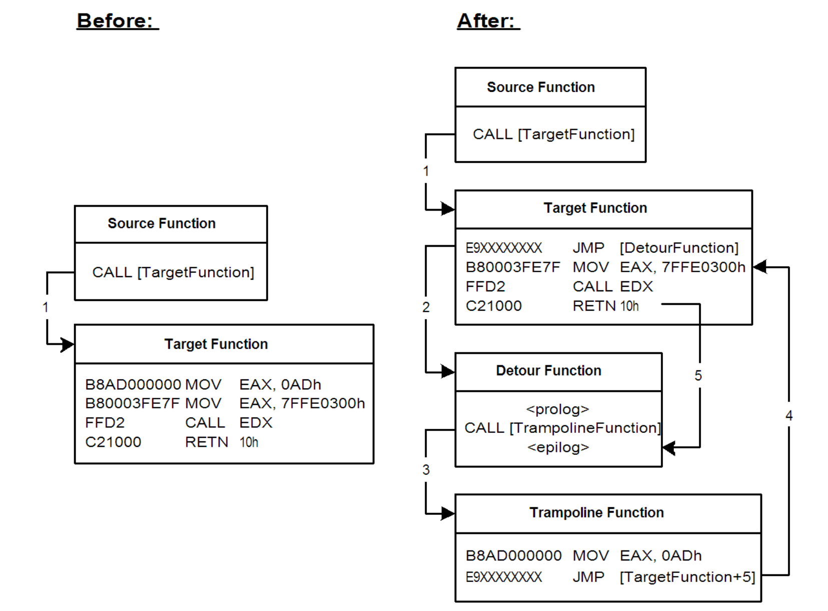


Figure 1: API function call before and after inline hooking. (Source: [8], modified by the authors.)

likely explanation is that kernel-mode inline hooking is not that well documented. Another explanation is that it has not been necessary – the other techniques have been effective enough. This might change in the future.

2.2.2. System Service Table hooking

System Service Table (SST) hooking is a powerful and widely adopted kernel-mode technique that can be used to intercept system service calls made by user-mode applications or even some kernel-mode modules. A system service call is a mechanism provided by the kernel that allows user-mode code to use its services in a controlled manner [6]. For example, whenever a user-mode application needs access to files, registry or process objects, it calls the appropriate *Windows* API call, which eventually generates a system service call that is then handled by the kernel. The concept of SST hooking was presented by Russinovich and Cogswell [10].

The idea of SST hooking is illustrated in Figure 2. The SST is a table of pointers where each entry contains the address of the internal kernel function that implements the corresponding service. If an entry in the table is replaced with one that points to code controlled by the hooking entity, it can intercept every call to the specific system service. The structure of the hook function is similar to the detour function described in the previous section. The only difference is that the hook function calls the original function instead of the trampoline function. This is because the original function was not modified in any way.

From the recent stealth malware, Lecna.a [11] and Haxdoor.al [9] use SST hooks for system-wide filtering. The installed hooks are used to hide certain files, processes and registry keys (see Table 1). Some of the hooks installed by Haxdoor.al backdoor are quite powerful. For example, if an untrusted process tries to terminate a hidden process protected by the hooks, the untrusted process will be terminated instead. In addition, the hooks enforce user-mode hook installation into every process created.

2.3. Hook installation

There are several different techniques for installing hooks on *Windows* platforms – some are well known and commonly used, whereas some are quite rare. User-mode and kernel-mode hooking requires different techniques for installation.

In order to hook a process in user mode, a rootkit usually has to inject its hook function code into its target process and then install the hooks. In the most common case this is accomplished by loading a Dynamic-Link Library (DLL) into the target process memory space. After the DLL has been loaded, hooks can be installed e.g. by running the entry point function of the DLL. See [1] for a thorough explanation on different *Windows* injection methods.

Some common user-mode code injection methods are:

- WriteProcessMemory API function
- SetWindowsHookEx API function
- AppInit_DLLs registry value

Code execution within another process can be accomplished with:

- CreateRemoteThread API function
- DLL entry point function (SetWindowsHookEx, AppInit_DLLs)
- SetThreadContext API function

As well as writing the hook function code into the target process, WriteProcessMemory can also be used for placing the hooks directly without having to execute anything within the target. This method has also been used by Haxdoor.al [9] and Hupigon.j for injecting user-mode hooks directly from kernel with the ZwWriteVirtualMemory system service call.

Injection methods that write code into the target process memory with

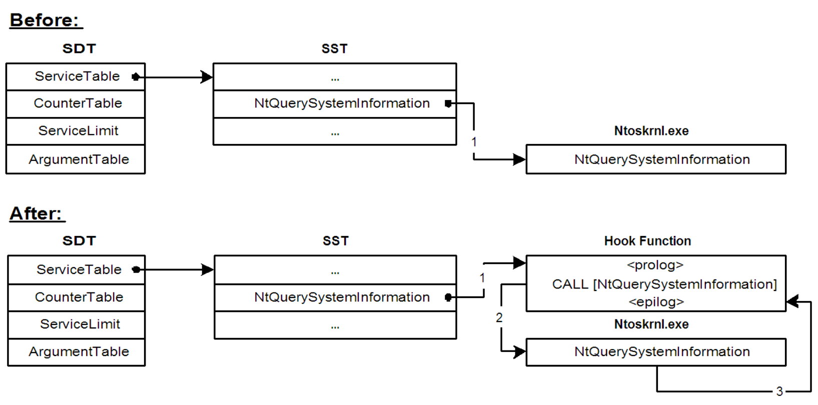


Figure 2: System Service Table before and after SST hooking.

WriteProcessMemory hook only that single process. Documented Windows GUI hook methods SetWindowsHookEx and AppInit_DLLs hook all processes that import user32.dll. Moreover, for example Padodor.W [12] uses the physical memory device to inject its hook code into all processes bypassing *Windows* copy-on-write [6] page protection. Note that *Microsoft* has denied direct access to physical memory from user-mode in recent service packs [13].

As discussed earlier, kernel-mode injection is somewhat simpler than its user-mode counterpart, as kernel mode has only a single memory space. *Windows* rootkits commonly inject their hooking code into the kernel by loading a kernel-mode driver. In addition to the documented methods, rootkits can use the ZwSetSystemInformation API function for loading drivers. However, it is also possible to write code into the kernel memory space from user-mode directly without a driver. One method is to write through the physical memory device. See [1] for more information.

2.4. Hiding without hooking

Direct kernel object manipulation was first introduced by Butler *et al.* [14]. This technique can hide selected processes and kernel-mode drivers by directly modifying executive objects used by the kernel. In theory, this method can also be used to hide other objects. However, in this paper we are going to concentrate on process objects. All the previously mentioned techniques hide information by filtering it somewhere along the execution path. This technique directly modifies the source of information.

To keep track of all processes present in the system, the *Windows* kernel maintains a doubly-linked list that links together every process object. When a user-mode application sends a request for the process list, the appropriate system service function traverses the linked list and sends the data back to the client. What Butler *et al.* [14] noticed, was that they were able to remove any entry from the list without any side effects to the actual process. The result was that they were able to hide any process from all user-mode applications without affecting its execution in any way. The process of unlinking an entry from the process list is illustrated in Figure 3.

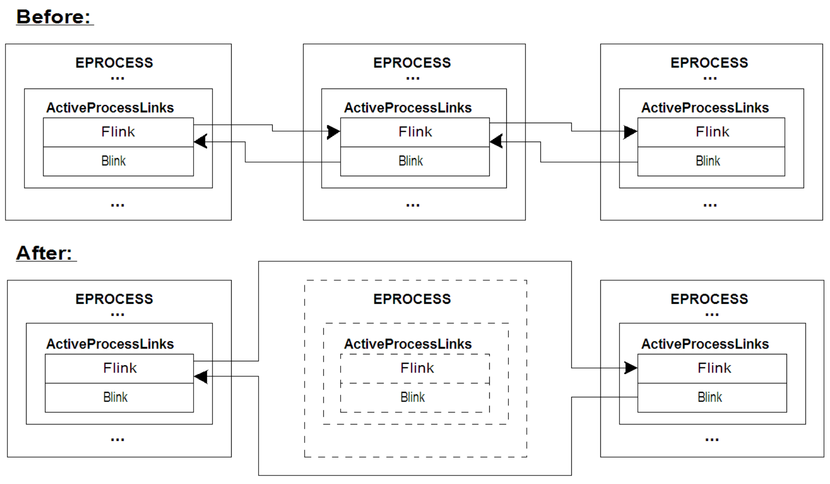


Figure 3: The ActiveProcessLinks doubly-linked list before and after entry removal. (Source: [14], modified by the authors.)

The Myfip.h [15] worm uses direct kernel object manipulation to hide its process. It does not use a driver to modify the process object. It does this directly from the user-mode code through the physical memory device. Since no drivers are used and no code modification is done, it is very hard to find the module that did the hiding.

3. DETECTING HIDDEN OBJECTS

3.1. General idea

The high-level model to detecting hidden objects is very simple, even though the actual detection techniques can be very complex. Essentially, we must acquire two views of the system: a tainted view and a trusted view. The tainted view is what the rootkit wants the user to see, with hidden files, processes and other objects out of sight. The trusted view is acquired from sources whose integrity we can trust to a reasonable degree. This view will contain everything actually present on the system, including the hidden objects. In the presence of a rootkit the tainted view and the trusted view will differ, making detection of the rootkit possible. The idea is not new – it is a traditional approach in computer forensics and has been utilized for detecting file system modifications and hidden files.

There are two key challenges. Firstly, we must acquire the trusted view. In the general case, the operating system does not provide any documented means to acquire this information. All sources can be tampered with without any noticeable side effects. Usually this means that we must replicate or manipulate operating system functionality or take advantage of undocumented data structures to acquire this view. The techniques are complicated and rarely give perfect results.

Secondly, we must define and acquire the tainted view. At first this might seem easy, but pitfalls do exist. For example, as we will show, rootkits hide files in many different ways. One API function may not see a hidden file, while another remains unaffected, making the definition of 'hidden' somewhat ambiguous. Actually acquiring the tainted view can also be hard. If a rootkit notices a detection attempt, it may temporarily reveal the objects it was hiding, making the tainted view identical to the trusted view. Detection then becomes impossible.

3.2. Hidden file detection

Comparing the trusted and the tainted views is a simple and efficient way to find hidden files. The technique has been known for several years and has at least been implemented by Philippe Bourgeois [16] and independently by Wang Jian [17].

3.2.1. Constructing the trusted view

The traditional way to construct the trusted view is to traverse through the file system using proprietary techniques. By reading the file system directly from the disk, we get a fairly reliable view of what files actually exist. The biggest problem with such an approach is that we are traversing a live file system and bypassing the standard file system API. Locking the drive while we do the scan is most likely not possible – at least for drives that contain critical system files like the OS itself or the swap file. Thus the file system can change at any time and we need to accommodate for these changes as we construct and use our trusted view.

Sector level access to the hard drives is usually possible using system API functions. If the hard drive is read using API functions, rootkits can always monitor and alter the behaviour of these functions and thus the view might not be absolutely reliable. To hide itself on this level, without interfering in a detectable way, the rootkit must know the underlying file system structure and monitor all file system activities actively in order to keep itself hidden. Such hiding techniques would be extremely hard to implement and would most likely downgrade performance dramatically on the machine, so as to alert the administrator that something is wrong.

In order to speed up the file system, most operating systems have some form of cache manager enabled. This is a problem when constructing a trusted view since the information on the disk might not be accurate and we bypass the standard file system functions where the caching happens. A new file might have been created, but it is still cached and has not been written to the disk yet. Alternatively a file might have been deleted, but since the operation is cached, the file might still be on the disk.

3.2.2. Constructing the tainted view

Constructing the tainted view is not necessarily a simple task, since different rootkits hide in different ways. However, the traditional *Windows* API way of listing the contents of a folder using the FindFirstFile and FindNextFile API functions is basically all that is needed.

A very simple approach would be just to take the files from the trusted view and ask if the appropriate API function can see the file. For example, Wang [17] parses through the raw file system and for each found file tries to locate the same file using documented system APIs. On *Windows* the most common function used for this is FindFirstFile. However, this is not sufficient since some rootkits hook only the FindNextFile function and thus FindFirstFile will find the files. This will still hide the files from applications that do not exactly know what they are looking for. They do not know the contents of a directory and thus ask FindFirstFile to find any file. The returned search handle is then passed to the FindNextFile function to search for other files that match the same pattern.

Usually the first returned files are '.' or '..' and the rest are returned by calls to FindNextFile. Since we want to know what files applications would normally see, we should just traverse through the file system and record the files that FindNextFile sees.

3.2.3. Comparing the views

Most hidden files can be found by testing whether a file in the trusted view is also in the tainted view. This type of test is not sufficient in all cases. For example, the NTFS file system has metafiles that are in the file system, but are never shown to the user. It would not be a good idea to report these to the user as hidden files that are part of a rootkit since they are legitimate files. On the other hand, excluding some files opens up the possibility for rootkits to hide their files so that we consider them to be harmless and filter them. One must be careful with such filtering.

In addition to testing whether the tainted view contains the files in the trusted view, we should compare the trusted view to what FindFirstFile sees. Maslan.a [18] worm hides its files by changing the name of the file returned by the two functions into a single dot ('.'). The files might be visible, although many applications ignore the '.' entry and even if it is visible it will be impossible to access them since the name '.' is reserved for other purposes. Thus, in a sense, we find the hidden file but it is given back with such a name that many applications would ignore it and thus the file is hidden. We need to compare that the name of the file we were asking for is the one we actually find. This test cannot be done with FindNextFile.

3.3. Hidden process detection

A simple and efficient approach to detecting hidden processes is to compare the trusted and tainted view of the system's process list. This approach works especially well on hidden process detection since normal applications have no need to hide their processes.

3.3.1. Constructing the trusted view

Constructing the trusted view is the most critical and challenging part of this technique. It is essential that the collected data has not been tampered with by any malicious code. There are basically two different approaches to this problem that have their advantages and disadvantages. The first one is to maintain a separate and private process list, whose contents are updated as new processes are created and old ones are deleted. The second approach is to collect the list of processes from some internal data structure maintained by the operating system.

Maintaining a private process list is feasible only if the code for maintaining its state, also known as the sensor, is guaranteed to be in place before a single process has been created. This is not a problem for modern anti-virus applications that already employ boot-mode drivers. In this case, there are two known kernel-mode techniques for implementing the sensor. The preferable technique is to install a driver-supplied notification routine that is called by the system whenever a process is created or deleted. This can be accomplished by using the PsSetCreateProcessNotifyRoutine, which is documented in the *Windows* DDK [19].

The other technique for implementing the sensor is quite extreme and relies on some undocumented and possibly version-dependent techniques. The basic idea is to hook the kernel scheduler's SwapContext function. Since a thread will not be able to execute before the scheduler switches its context in, the sensor will be aware of every process whose threads have received execution time. This approach was suggested by Butler *et al.* [14] and implemented by Kasslin [1]. The implementation showed that the technique was feasible – the performance impact was less than one per cent and no stability problems were encountered [1]. The only drawback is its dependability on undocumented techniques for locating the SwapContext function, which is not exported by the kernel module. This might result in compatibility problems with newer operating system versions and service packs.

To maintain a private process list, the sensor must be started before a single process has started. This is not possible for standalone applications that run their sensor only when requested by the user. Therefore, such applications have to use a different approach to collect an up-to-date process list it trusts. A solution is to gather the data from internal data structures maintained by the kernel. There are two known locations from where to fetch the data. The first one is the linked list, which links together every process through its ActiveProcessLinks field [6]. This is the list used by the ZwQuerySystemInformation system service call when a user-mode application requests a process list. The second one consists of three linked lists maintained by the kernel scheduler. Unexported symbol names KiDispatcherReadyListHead, KiWaitInListHead and KiWaitOutListHead point to their locations. On *Windows 2000* these lists can be traversed to find every thread in the system that can then be mapped to their owner processes [20].

However, both locations have their drawbacks. If malicious code uses kernel-mode components, it can remove elements from the ActiveProcessLinks linked list without side-effects [14]. This is not an issue with the linked list used by the scheduler since, if a thread is removed from any of them, it will not receive any processor time and thus will not execute. The biggest problem with using the scheduler lists is that the technique works only on *Windows 2000* and relies on unexported symbols, making it extremely version-dependent [1].

The *Windows* kernel maintains several data structures for its internal use, most of which are undocumented. Some of them contain useful data that can be used to create the trusted view. Ming *et al.* [5] have suggested the existence of such a location in the recently published *Microsoft* research paper.

3.3.2. Constructing the tainted view

Constructing the tainted view is simpler than it was with hidden file objects. Most process-hiding malware either hook the ZwQuerySystemInformation function exported by ntdll.dll or do hiding directly in kernel mode. A reasonable explanation is that, when an average user wants to know what processes are executing on his system, he launches the Task Manager application, which eventually calls the ZwQuerySystemInformation function.

Therefore, the only requirement in creating the tainted view is to make sure the execution path goes through the ZwQuerySystemInformation function. This can be done by using either the EnumProcesses function exported by psapi.dll or the CreateToolhelp32Snapshot exported by kernel32.dll. Both functions are documented in the *Windows Platform SDK* [21].

3.3.3. Comparing the views

Hidden processes can be found simply by checking whether every process in the trusted view is also present in the tainted view. If the trusted view shows more items than the tainted view, it is a clear indication of a hidden process. The operating system has no reason to hide the presence of any of its processes, as was the case with some of the internal files used by the NTFS file system. The detector should also check for the presence of any duplicate entries in the tainted view since Windows Task Manager shows only one instance of every pid. If duplicate entries are found, it would indicate that a malicious process is trying to hide by imitating a benign process.

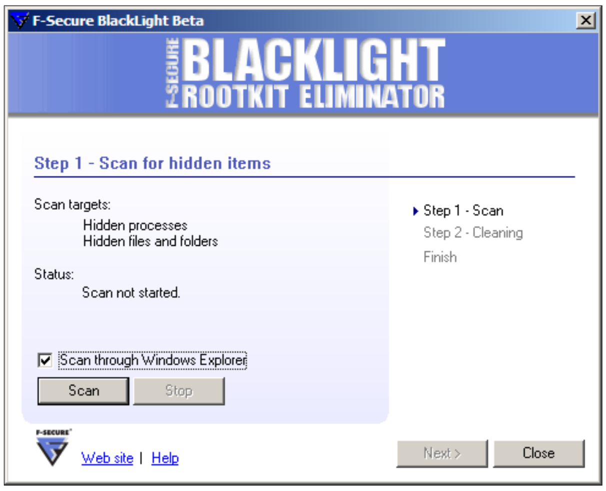
4. EXPERIENCES OF REAL-LIFE ROOTKIT DETECTION

4.1. Practical stealth malware detection

Prior to 2005, generic rootkit detection tools have presented the user with a long list of processes, registry keys, or hooked functions. These kinds of tools rely on the user to provide the detection logic and are therefore feasible only for expert users. There have also been some easier-to-use tools that detect rootkits based on different fingerprinting techniques and are therefore limited to certain rootkit implementations.

F-Secure BlackLight [22] was released as a beta on 10 March 2005. Requirements for *BlackLight* technology were the following:

- Generic solution: detection is based on behaviour (hiding) and not on binaries, services, or other fingerprints.
- Ease of use: a normal home user should be able to find rootkits with the tool.
- Accuracy: the scanner should not produce any false positives.
- Completeness: should find all current rootkits.
- Speed: use intelligent scanning of the hard disk to provide fast scans.
- Removal: users should be provided with the means to deactivate rootkits on their system.


F-Secure BlackLight Beta

F-SECURE BLACKLIGHT ROOTKIT ELIMINATOR

Step 1 - Scan for hidden items

Scan targets:
Hidden processes
Hidden files and folders

Status:
Scan not started.

[✓] Scan through Windows Explorer

BlackLight beta represents what could be called the second generation of rootkit detection tools. It is generic, but still usable even for the normal user. However, the technology is still in infancy and it will take a long time until rootkit scanner technology has matured to the level where traditional anti-virus currently is.

4.2. Rootkit threat in early 2005

Rootkit techniques are used by a number of common keyloggers and other Trojans. Perhaps the most well known of these is the Padodor/Berbew [7] family. There are also dozens of pure-breed rootkits such as Hacker Defender. Some of these rootkits are more proof-of-concept and some are clearly malicious having e.g. embedded DDoS functionality. Hackers use these rootkits in order to secure their foothold on compromised machines. Currently there is no reliable data on how extensively and to what ends rootkits are used in these cases.

Since its release in March 2005, *F-Secure* has received reports of *BlackLight* beta detecting rootkits. The number of reports is not large enough to draw decisive conclusions, but many of the more serious system compromises have been related to software piracy. It seems that the warez underground is turning compromised servers into rootkit-protected pirate FTP-sites for distribution of movies, MP3s, and cracked software.

The first rootkit-worm, Maslan.A [18] appeared in December 2004. Maslan is notorious for its DDoS attack on Chechen websites, but its rootkit capabilities are not that well known. Since Maslan, there has been at least one other stealth worm, Myfip.H [15], but the threat of a large-scale rootkit worm has not materialized.

Lately we have seen a new development in rootkits. Variants of common bots have been using open-source rootkits to hide themselves. Examples of these are Rbot [23, 24] and SdBot [25] variants that drop a recompiled FU rootkit driver, and a Sdbot variant that drops a Turkish version of the Hacker Defender rootkit. There has also been at least one Mytob worm variant, Mytob.AR [26] that uses FU rootkit's driver. In addition, we have seen some malware, such as the ProAgent 2.0 spyware Trojan that uses the open-source JiurlPortHide driver for hiding their network connections.

4.3. Rootkits can evade detection

Windows rootkit detection became a hot topic in early 2005. As an example, Hacker Defender rootkit removal was added to the *Microsoft Malicious Software Removal Tool* [27] in April. Also, a number of stand-alone tools for detecting rootkits were launched in early 2005. This resulted in rootkit authors implementing countermeasures to avoid detection. As with any security weapon-countermeasure arms race, this is an ongoing battle and it is impossible to build a rootkit detector that will detect all possible future rootkits.

Hacker Defender rootkit's public version (v1.0, source released to public on 1 January 2004) includes a functionality of adding executable file name masks to a list of trusted processes. This functionality was most likely originally intended for preventing the rootkit from hiding from the attackers' own tools on a compromised system. However, attackers soon started adding the names of rootkit detection tools to the trusted list. Most rootkit scanners are based on the concept of comparing two views. If the views are the same, the scanner will not report anything.

Not all current anti-detection methods are based on binary names. We have also seen rootkit code that evades scanner processes based on binary version strings. Moreover, commercial versions of Hacker Defender rootkits are marketed as being able to detect anti-rootkit tools with any binary name and even with packed binaries. This would suggest that there is something very close to anti-virus fingerprinting and heuristics in place.

Obviously a simple 'tainted view/clean view' comparison is not enough. Either we will have to find alternatives for the comparison approach or we will have to prevent rootkits from recognizing the scanner.

4.4. Rootkit techniques in non-malicious use

During the internal beta testing of the *F-Secure BlackLight* beta rootkit scanner we did not receive false positives and were able to detect all rootkits we tested it on. Quite soon after the public launch we began receiving reports of false positives from some users – not many, but some. It turned out that a number of harmless third-party software packages were using rootkit techniques to hide something. So far we have found three categories:

- Security software that hides its processes or files, most likely in order to avoid attacks from malware.
- Data restoration and system recovery software that uses filter drivers to hide its backups on disk.
- Commercial data-hiding software targeted for multi-user environments. One apparent target group for these applications is people who want to hide adult material from their family. Features of these programs seem very similar to common file hiding rootkits such as Vanquish or HE4Hook.

We strongly believe that if you want to prevent other people from accessing your files, you should use operating system access controls and encryption – not rootkit stealth techniques.

We have also encountered a few cases where *F-Secure BlackLight* reports some normal system files as hidden in addition to the ones actually belonging to the malware. After further investigation, the cause has always been an intentional or unintentional misconfiguration of the rootkit. For example, in one case the rootkit was configured to hide all files and directories whose name started with the string 'system'. As the reader can imagine, this resulted in thousands of hidden files.

5. CONCLUSIONS

Windows rootkit technology is maturing rapidly and new hiding techniques emerge every year. This technology has frightening possibilities. Although most current rootkits do not hide themselves effectively from anti-virus scanners, it is certainly possible to create a rootkit that, once activated, is completely hidden from anti-virus scanning engines. Also, these same technologies can be used to create backdoors that bypass personal firewalls. Stealth is certainly back and kicking!

Current research and public tools prove that it is possible to achieve generic rootkit detection – detection that is based on behaviour. However, recent events have proven that, as with any countermeasure, these detectors can always be bypassed with a new generation of rootkits. This is the same arms race that we have seen e.g. with firewalls all over again.

At the moment hidden process detection seems to be the most feasible – it is fast, produces the least false positives, and it usually detects the most interesting hidden item in the system: the rootkit process. However, in the future the significance of hidden process detection will most likely diminish as rootkits try to hide without any user-mode processes. Rootkits will most likely still have hidden files, despite some efforts on creating in-memory malware. Also, *Windows* rootkits will need to launch themselves on reboot in some way. These launch points can be detected by rootkit scanners.

Today, the rootkit threat is still rather small compared to more traditional malware technologies. On the other hand, stealth malware fits perfectly into the arsenal of network crime. We will most likely see a lot more stealth in the coming years. The anti-virus industry should make every effort to be pre-emptive and address this threat before it truly materializes – and perhaps prevent it ever really becoming a major problem.

REFERENCES

[1] Kasslin, Kimmo. (2005). Windows Rootkits: Advanced Hiding Techniques and Counter-Measures. Master's Thesis, Helsinki University of Technology.

[2] F-Secure virus descriptions: Brain. Available from: http://www.f-secure.com/v-descs/brain.shtml.

[3] Ször, Peter. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional. 744 pages. ISBN 0-321-30454-3.

[4] F-Secure virus descriptions: Cabanas. Available from: http://www.f-secure.com/v-descs/cabanas.shtml.

[5] Wang, Yi-Min; Beck, Doug; Vo, Binh; Roussev, Roussi; Verbowski, Chad. (2005). Detecting Stealth Software with Strider GhostBuster. Microsoft Research Technical Report MSR-TR-2005-25, February 21, 2005. Available from: ftp://ftp.research.microsoft.com/pub/tr/TR-2005-25.pdf.

[6] Solomon, David; Russinovich, Mark. (2005). Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000. 4th edition. Redmond, Washington. Microsoft Press. 935 pages. ISBN 0-7356-1917-4.

[7] Erdelyi, Gergely. (2004). 'Hide'n'Seek? Anatomy of Stealth Malware', *Proceedings of the 2004 Black Hat Europe*, pp.147–167.

[8] Hunt, Galen; Brubacher, Doug. (1999). 'Detours: Binary Interception of Win32 Functions', *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135–143.

[9] F-Secure virus descriptions: Haxdoor. Available from: http://www.f-secure.com/v-descs/haxdoor.shtml.

[10] Russinovich, Mark; Cogswell, Bryce. (1997). 'Windows NT System-Call Hooking', *Dr. Dobb's Journal*, no.261, January 1997.

[11] F-Secure virus descriptions: Lecna. Available from: http://www.f-secure.com/v-descs/lecna_b.shtml.

[12] F-Secure Virus Descriptions: Padodor.W. Available from: http://www.f-secure.com/v-descs/padodorw.shtml.

[13] Microsoft Platform SDK Documentation: EnumSystemFirmwareTables. Available from: http://msdn.microsoft.com/library/en-us/sysinfo/base/enumsystemfirmwaretables.asp.

[14] Butler, James; Undercoffer, Jeffrey; Pinkston, John. (2003). 'Hidden processes: the implication for intrusion dDetection', *Proceedings of the 2003 IEEE Workshop on Information Assurance*, 18-20 June, West Point, NY. Piscataway, NJ, IEEE pp.116–121.

[15] F-Secure virus descriptions: Myfip.H. Available from: http://www.f-secure.com/v-descs/myfip_h.shtml.

[16] CERT-IST Packages. Ancheck v 0.2 by Philippe Bourgeois. Available from: http://www.cert-ist.com/english/tools/outils_en.htm.

[17] Wang, Jian. (2002). An Alternative Method to Check LKM Backdoor/rootkit. *BugTraq Internet Mailing List*, April 16. Available from: http://www.securityfocus.com/archive/104/268526.

[18] F-Secure virus descriptions: Maslan.A. Available from: http://www.f-secure.com/v-descs/maslan.shtml.

[19] Microsoft Windows Driver Development Kit. Available from: http://www.microsoft.com/whdc/devtools/ddk/default.mspx.

[20] Rutkowska, Joanna. (2003). KLISTER Hidden Process Detector for Windows 2000. Available from: http://www.invisiblethings.org/tools.html.

[21] Microsoft Platform SDK Documentation. Available from: http://msdn.microsoft.com/library/en-us/sdkintro/sdkintro/devdoc_platform_software_development_kit_start_page.asp.

[22] F-Secure BlackLight. Available from: http://www.f-secure.com/blacklight.

[23] Microsoft Malicious Software Encyclopedia: Win32/Rbot. Available from: http://www.microsoft.com/security/encyclopedia/details.aspx?name=win32%2frbot.

[24] Sophos virus information W32/Rbot-ACD. Available from: http://www.sophos.com/virusinfo/analyses/w32rbotacd.html.

[25] Microsoft Malicious Software Encyclopedia: Win32/Sdbot. Available from: http://www.microsoft.com/security/encyclopedia/details.aspx?name=Win32%2fSdbot.

[26] Symantec Security Response, W32.Mytob.AR@mm. Available from: http://securityresponse.symantec.com/avcenter/venc/data/w32.mytob.ar@mm.html.

[27] Microsoft Malicious Software Removal Tool. Available from: http://www.microsoft.com/security/malwareremove/default.mspx.

[28] Symantec Security Response - Backdoor.Greybird.L. Available from: http://securityresponse.symantec.com/avcenter/venc/data/backdoor.greybird.l.html.