Taxonomy of Rootkits

A thesis
submitted in partial fulfillment of the requirements
for the award of degree
of

Master of Engineering
in
Software Engineering

Submitted By
Gaurav Bajaj
(Roll No 8043105)

Under the Supervision of
Mr. Maninder Singh
Assistant Professor
CSED, TIET, Patiala.

May 2006

Computer Science & Engineering Department
Thapar Institute of Engineering & Technology
(Deemed University), Patiala-147004

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Taxonomy of Rootkits",** submitted by me in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering at Computer Science and Engineering Department of Thapar Institute of Engineering and Technology (Deemed University), Patiala, is an authentic record of my own work carried out under the supervision and guidance of Mr. Maninder Singh.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other University.

Gaurav Bajaj

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Mr. Maninder Singh
Assistant Professor
Computer Science & Engineering Department,
Thapar Institute of Engineering & Technology,
Patiala- 147004.

Dr.(Mrs) Seema Bawa
Professor & Head
Computer Sc. & Engg. Department
Thapar Institute of Engg. & Technology
Patiala- 147004

Dr. T. P. Singh
Dean of Academic Affairs
Thapar Institute of Engg. &
Technology
Patiala- 147004

Acknowledgement

I wish to express my deep gratitude to Mr. Maninder Singh, Assistant Professor, Computer Science and Engineering Department for providing his uncanny guidance and support throughout the thesis work.

I am also thankful to Dr. (Mrs.) Seema Bawa, Head, Computer Science and Engineering Department, for the motivation and inspiration.

I would also like to thank all the staff members and my co-students who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of the thesis.

Gaurav Bajaj
(8043105)

Abstract

Breaking into a computer system involves hard work. Therefore once a hacker succeeds, he wants to maintain access into the system so that any future penetration will be a hassle-free job and effort saved could be better utilized to launch further attacks. A rootkit serves this purpose by allowing permanent or consistent, undetectable presence on a computer. Such stealth is made possible by locating and modifying the software in the target system so that it makes incorrect decision. Various techniques employed by the rootkits have been revealed through the forensic analysis of such software. The aim of my research is to classify the rootkits based on the techniques used by them in order to achieve their purpose of stealth along with the specific functions and data structures they target in each technique. By making such classifications and observing the manipulations done by rootkits in various important kernel or user space structures and functions, one can get insight into the working of rootkits and can develop key survival techniques to detect these rootkits and recover the clean system. We have attempted to discover the functions and structures for each category with the aid of reverse engineering, direct examanitation of publicly available Windows based rootkits' source code and documentation available from various sources.

Table Of Contents

List Of Figures

List Of Tables

Chapter 1: **Introduction**

Large number of new vulnerabilities are discovered each day and shared amongst people in myriad mailing lists such as Bugtraq. Such kind of networking has contributed to the growth in the number of hacking incidents over the past decade. Even people with not so good technical know-how are capable of hacking because numerous ready-made tools are available free for them on the web. Once they know which services are running on a system, it is not an uphill tasks even for a novice to compromise that system. On the other side, the administrator will try to regain his system as soon as he realizes that the system has been compromised. This can be done by eliminating the vulnerability, a full reinstall of the system, or simply by taking it offline. Therefore if attackers want to maintain their access in the system, they have to take measures not to get noticed[1]. The skilled attacker uses collection of programs known as rootkits to hide his presence and activities on the compromised system because the best way to counter detection is stealth. If administrator is not able to find any trace of compromise, no action will be taken against it.

Rootkits are typically used by attackers to open a backdoor into Windows systems, collect information on other systems on the network and mask the fact that the system is compromised. Masking implies that one can hide a process, hide a network port, redirect file writes to a different file, prevent an application from opening a handle to a particular process, and more. As a result, it becomes extremely difficult for an administrator to know whether the system under observation is clean or compromised. In order to develop a defense against such a menace, it is prudent to understand the different ways by which rootkits accomplish their purpose. By observing the manipulations done by rootkits in various important kernel structures or user space structures, one can get insight into the working of rootkits and can develop key survival techniques to detect these rootkits and recover the clean system.

1.1 History of Rootkits

Rootkits first appeared at the end of the 80's and consisted of tool collections used to manipulate UNIX log files in order to hide the presence of certain users. Later on, attackers started to substitute programs like ls, ps or netstat to hide their activities. Such programs were soon packaged together with manipulated versions of login and similar programs to secretly log passwords. *Sniffers* were used to read unencrypted passwords from the local network.

In the mid 90's *kernel rootkits* appeared on Linux systems . These were loaded as modules at runtime to manipulate the core of the operating system itself. This technique allowed very thorough obfuscation of the attacker's activities e.g. by redirecting system calls instead of exchanging single programs. By the end of the 90's kernel rootkits existed for practically all modern UNIX-like operating systems. At the same time rootkits for Microsoft Windows appeared. Although attackers had to cope with a lack of documentation they were able to replay the development of rootkits in the UNIXworld within a few years. Current Windows rootkits are kernel-based and provide similar functionality as their UNIX counterparts [2].

In recent years the methods used by attackers have been refined. New ways of code injection have been developed, allowing to patch a rootkit into the running kernel without the usage of modules. Other rootkits manipulate existing modules or kernel images on disk to install themselves. While manipulation of the kernel by the redirection of system calls is relatively easy to detect, modern rootkits have become more sophisticated and more difficult to detect. The flow of execution is diverted at different places and at deeper levels within the kernel. Furthermore, obfuscation techniques are employed in the rootkit binaries to prevent analysis if rootkits are found[2].

Aside from simple checksum based approaches new methods of detection, like runtime measurement of system calls, have been developed. Critical resources of the operating system are checked for consistency to discover the diversion of the execution flow by the rootkit.

1.2 What a Rootkit Is

A rootkit is a set of tools used by an intruder after cracking a computer system. These tools can help the attacker maintain his or her access to the system and use it for malicious purposes. Rootkits exist for a variety of operating systems such as Microsoft Windows, Linux, Solaris. To accomplish its goal, a rootkit will alter the execution flow of the operating system or manipulate the data set that the operating system relies upon for auditing and bookkeeping.

Rootkits have become very sophisticated over the past few years, and in 2005 we have seen a surge in rootkit deployments in spyware, worms, botnets, and even music CDs. Although once a computer system has been subverted by a rootkit it is extremely difficult to detect or eradicate the rootkit, there are still some different methodologies that detect the rootkit that have worked to varying degrees[3].

1.3 What A Rootkit Is Not

Rootkits may be used in conjunction with an exploit, but the rootkit itself is a fairly straightforward set of utility programs.These programs may use undocumented functions and methods, but they typically do not depend on software bugs (such as buffer overflows).

Although a rootkit is not an exploit, it may incorporate a software exploit. A rootkit usually requires access to the kernel and contains one or more programs that start when the system is booted. There are only a limited number of ways to get code into the kernel (for example, as a device driver). Many of these methods can be detected forensically.

A virus program is a self-propagating automaton. In contrast, a rootkit does not make copies of itself, and it does not have a mind of its own. A rootkit is under the full control of a human attacker, while a virus is not. Even though a rootkit is not a virus, the techniques used by a rootkit can easily be employed by a virus. When a rootkit is combined with a virus, a very dangerous technology is born.[2]

1.4 Types of Rootkits

There are broadly two types of rootkits depending on the privilege level that it operates in.

User Mode Rootkits

There are many methods by which rootkits attempt to evade detection. For example, a user-mode rootkit might intercept all calls to the Windows FindFirstFile/FindNextFile APIs, which are used by file system exploration utilities, including Explorer and the command prompt to enumerate the contents of file system directories. When an application performs a directory listing that would otherwise return results that contain entries identifying the files associated with the rootkit, the rootkit intercepts and modifies the output to remove the entries.

Kernel Mode Rootkits

Kernel-mode rootkits can be even more powerful since, not only can they intercept the native API in kernel-mode, but they can also directly manipulate kernel-mode data structures. A common technique for hiding the presence of a malware process is to remove the process from the kernel's list of active processes. Since process management APIs rely on the contents of the list, the malware process will not display in process management tools like Task Manager or Process Explorer.

1.5 Some Popular Rootkits

According to statistics from Microsoft's anti-malware engineering team, more than 20 percent of all malicious code removed from Windows XP SP2 (Service Pack 2) systems are stealth rootkits. [4]

1. According to virus hunters at F-Secure, of Helsinki, Finland, the latest Bagle.GE variant loads a kernel-mode driver to hide the processes and registry keys of itself

and other Bagle-related malware from security scanners. The use of offensive rootkits in existing virus threats signals an aggressive push by attackers to get around existing anti-virus software and maintain a persistent and undetectable presence on infected machines.

In the case of the Bagle.GE rootkit, F-Secure researcher Jarkko Turkulainen said the rootkit successfully hides processes, files and directories, registry keys and values and contains code that will prevent certain security related processes and kernel-mode modules from running.

2. F-Secure also found evidence of a rootkit in Gurong.A, a new worm that is based on the Mydoom code. Gurong.A uses a range of social engineering tricks to propagate via e-mail and also spreads through shared folders in the Kazaa peer-to-peer application. Like the Bagle rootkit, Gurong.A hides processes, files and launch points whenever the worm is active. It is also able to modify kernel-mode process structures to hide any process it specifies.

3. Microsoft lists FU among the top-five pieces of malware deleted by its free Windows malicious software removal tool. Two other stealth rootkits—WinNT/Ispro and Win32/HackDef—are also high on the list of removed malware.

1.6 An example rootkit attack

On May, 2006 news broke out that there is a rootkit embedded in online poker software. This is how it actually was distributed and carried out its attack.

A Trojan with malicious rootkit features was hidden in a legitimate software package distributed by online gaming tools vendor Check Raised and it had the ability to hijack log-in information for multiple online poker Web sites.

The spying Trojan, identified as Backdoor.Win32.Small.la, was built into a Rakeback calculator application (RBCalc.exe) distributed by Check Raised to help online poker

players keep track of scaled commission fees taken by the Web site operator. The rake calculator is offered as an executable file that players runs on their machine to calculate rake from hands they previously played (stored in hand history files or a poker tracker database). When the rake calculator was run, it silently drops several files into the Windows system directory to monitor running processes and spy on connections to several popular online poker Web sites.

The Trojan's main file comes with rootkit functionality to hide its process and the registry launch point. When the spying component is initialized, it starts a keystroke logger and connects to a remote server that is programmed to send instructions to the infected machines. The instructions range from the downloading of executable files, the uploading of stolen information, the shutdown of the Trojan and the ability to send application screenshots.

The backdoor also sent out sensitive information to remote servers, including keylogger database, computer name, and the username and password of several online poker programs.

Chapter 2: **Literature Survey**

A rootkit is a program or set of programs that an intruder uses to hide his/her presence on a computer system and to allow access to the computer system in the future. To accomplish its goal, a rootkit will alter the execution flow of the operating system or manipulate the data set that the operating system relies upon for auditing and bookkeeping.

Rootkits have historically demonstrated a co-evolutionary adaptation and response to the development of defensive technologies designed to apprehend their subversive agenda. If we trace the evolution of rootkit technology, this pattern is evident. First generation rootkits were primitive. They simply replaced / modified key system files on the victim's system [5]. The UNIX login program was a common target and involved an attacker replacing the original binary with a maliciously enhanced version that logged user passwords. Because these early rootkit modifications were limited to system files on disk, they motivated the development of file system integrity checkers such as Tripwire.

In response, rootkit developers moved their modifications off disk to the memory images of the loaded programs and, again, evaded detection. These 'second' generation rootkits were primarily based upon hooking techniques that altered the execution path by making memory patches to loaded applications and some operating system components such as the system call table. Although much stealthier, such modifications remained detectable by searching for heuristic abnormalities. For example, it is suspicious for the system service table to contain pointers that do not point to the operating system kernel.

In the following section we have described hooking in detail which represent the second generation of rootkits techniques[6]

.

2.1 Hooking

In order for a rootkit to alter the normal execution path of the operating system, one of the techniques it may employ is "hooking". In modern operating systems, there are many places to hook because the system was designed to be flexible, extendable, and backward compatible. By using a hook, a rootkit can alter the information that the original operating system function would have returned. There are many tables in the Windows operating system that can be hooked by a rootkit.

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[2]. 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. 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. [1]

Kernel-mode filtering filters the information while the thread is executing in kernel mode. Since every process shares the same system address space, 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. [1]

2.1.1 Hooking Techniques

In Windows environment, there are several techniques that can be used to hook binary code. Some of the most wide used techniques are :

IAT Hooking

Windows is designed to be largely independent of the underlying computer hardware and compatible with other operating environments such as POSIX. It also must be flexible so that upgrades to the underlying operating system do not require application developers to completely rewrite their code. Windows does this by exposing a set of environmental subsystems: the Win32 subsystem, the POSIX subsystem, and the OS/2 subsystem. Each of these environmental subsystems is implemented as a Dynamic Link Library (DLL). These subsystems provide an interface to the system services that reside in kernel memory. By using this Application Programming Interface (API), application developers can write software that will survive most operating system upgrades. Usually, these applications do not call the Windows system services directly; instead, they go through one of these subsystems. These libraries export the documented interface that the programs linked to that subsystem can call. The Win32 subsystem is the most commonly used. It is composed of Kernel32.dll, User32.dll, Gdi32.dll, and Advapi32.dll. Ntdll.dll is a special system support library that the subsystem DLLs use. It provides dispatch stubs to Windows executive system services, which ultimately pass control to the SSDT in the kernel where the real work is performed. These stubs contain architecture specific code that causes a transition into kernel mode.[2]

When a Windows binary loads in memory, the loader must parse a section of the file called the Import Address Table (IAT). The IAT lists the DLLs and the corresponding functions in the DLL that the binary will use. The loader will locate each of these DLLs on disk and map them into memory. Then, the loader puts the address of the function in the IAT of the binary that calls the function. Common entries in the IAT are functions exported by Kernel32.dll and Ntdll.dll. Other libraries provide useful functions and may appear in the IAT such as the socket functions exposed by Ws2_32.dll. Kernel device

drivers also import functions from other binaries in kernel memory such as Ntoskrnl.exe and Hal.dll.

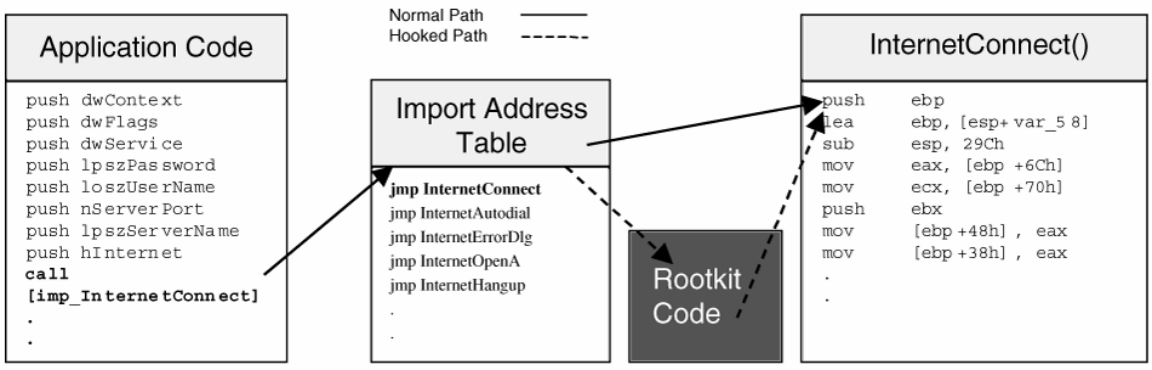


Fig 2.1 Normal execution path vs. hooked execution path for an IAT hook.
Source: Rootkits:Subverting the windows kernel.

By modifying the entries in a binary's IAT, a rootkit can alter the execution flow of the program and influence what the original function would have returned to the caller. For example, suppose an application lists all the files in a directory and performs some operation on them. This application might run in user mode as a user application or a service. Also, suppose the application is a Win32 application, which implies it will use Kernel32, User32.dll, Gui32.dll, and Advapi.dll to eventually call into kernel functions. Under Win32, to list all the files in a directory, an application first calls FindFirstFile, which is exported by Kernel32.dll. FindFirstFile returns a handle if it was successful. This handle is used in subsequent calls to FindNextFile to iterate through all the files and subdirectories in the directory. FindNextFile is also an exported function in Kernel32.dll. Since the application uses these functions, the loader will load Kernel32.dll at runtime and copy the address of these functions in memory into the application's IAT. When the application calls FindNextFile, it just jumps to a location in its import table which then jumps to the address of FindNextFile in Kernel32.dll. The same is true for FindFirstFile. FindNextFile in Kernel32.dll calls into Ntdll.dll. Ntdll.dll loads the EAX register with the system service number for FindNextFile's equivalent kernel function, which happens to be NtQueryDirectoryFile. Ntdll.dll also loads EDX with the user space address of the parameters to FindNextFile. Ntdll.dll then issues an INT 2E or a SYSENTER instruction to trap to the kernel[7].

In this example, a rootkit can overwrite the IAT in the application to point to the rootkit's function instead of Kernel32.dll's. The rootkit could have also targeted Ntdll.dll in the same manner. What the attacker does with this technique is largely up to her imagination. The rootkit may call the original function and then filter the results to hide things such as files, directories, Registry keys, processes, etc.

System Service Descriptor Table Hooking

The kernel provides the ideal place to install a hook. There are many reasons for this, but the two that are most important to remember are that kernel hooks are global (relatively speaking), and that they are harder to detect, because if our rootkit and the protection/detection software are both in Ring Zero, our rootkit has an even playing field on which to evade or disable the protection/detection software.

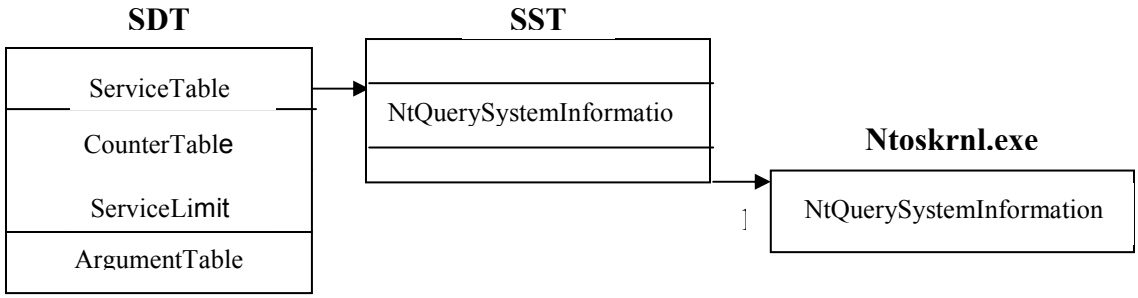
The Win32 subsystem is one place that a rootkit can hook. However, this subsystem is only a window into the kernel. The addresses of the actual implementation of the operating system functions are contained in a kernel table called the System Service Descriptor Table (SSDT) also known as the system call table. These addresses correspond to the NtXXX functions implemented in Ntoskrnl.exe. A kernel mode rootkit can alter this table directly and replace the desired NtXXX functions with pointers to the rootkit code. This is very powerful because instead of hooking a single program like an IAT hook does, this technique installs a system wide hook that affects every process. Using the example from the section on IAT hooking, the kernel rootkit could hook NtQueryDirectoryFile to hide files and directories on the local file system.

System service dispatch is triggered when an INT 2E or SYSENTER instruction is called. This causes a process to transition into kernel mode by calling the system service dispatcher. An application can call the system service dispatcher, KiSystemService, directly, or through the use of the subsystem. If the subsystem (such as Win32) is used, it calls into Ntdll.dll, which loads EAX with the system service identifier number or index of the system function requested. It then loads EDX with the address of the function parameters in user mode. The system service dispatcher verifies the number of

parameters, and copies them from the user stack onto the kernel stack. It then calls the function stored at the address indexed in the SSDT by the service identifier number in EAX.[1]

Once our rootkit is loaded as a device driver, it can change the SSDT to point to a function it provides instead of into Ntoskrnl.exe or Win32k.sys. When a non-kernel application calls into the kernel, the request is processed by the system service dispatcher, and our rootkit's function is called. At this point, the rootkit can pass back whatever bogus information it wants to the application, effectively hiding itself and the resources it uses.

Before:



After:

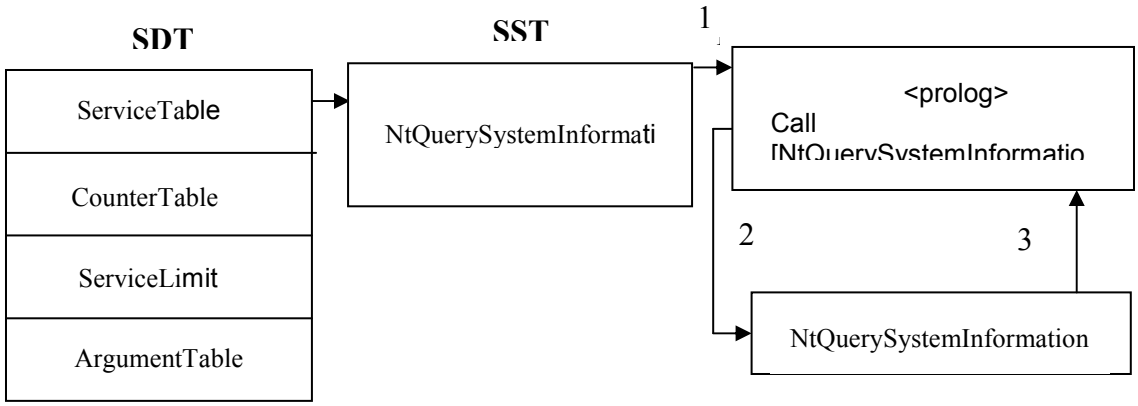


Figure 2.2 System Service Table before and after SST hooking.
Source: Hide 'n' Seek, Anatomy of Stealth Malware

The Windows operating system uses the ZwQuerySystemInformation function to issue queries for many different types of information. Taskmgr.exe, for example, uses this function to get a list of processes on the system. The type of information returned depends on the SystemInformationClass requested. Once the rootkit has replaced the NtQuerySystemInformation function in the SSDT, the hook can call the original function and filter the results[1].

Inline function hooking

Inline function hooking is more advanced than IAT or SSDT hooking. Instead of replacing pointers in a table, which we will show in a later article is easy to detect, an inline function hook replaces several bytes in the original function. Usually the rootkit adds an unconditional jump from the original function to the rootkit code. Many Windows API functions begin with a standard preamble:

Code Bytes	Assembly
8bff	mov edi, edi
55	push ebp
8bec	mov ebp, esp

For an inline function hook, the rootkit saves the original bytes in the function it is overwriting in order to preserve the same functional behavior. Then, it overwrites a portion of the original with a jump to the rootkit code. Notice that the rootkit can safely overwrite the first five bytes of the function because that is the same amount of space required for many types of jumps or for a call instruction, and it is on an even instruction boundary.[5]

Code Bytes	Assembly
e9 xx xx xx xx	jmp xxxxxxxx
…	

Here "xx xx xx xx" is the address of the rootkit. Now the rootkit can jump to the original code plus some offset and modify what the original operating system function returned.

Inline function hooking has many legitimate uses as do most rootkit techniques. Microsoft Research first documented inline function hooking at a conference. Today, Microsoft has expanded its usage far beyond just research. They have titled it "hot patching," which allows a system to be patched without rebooting [12].

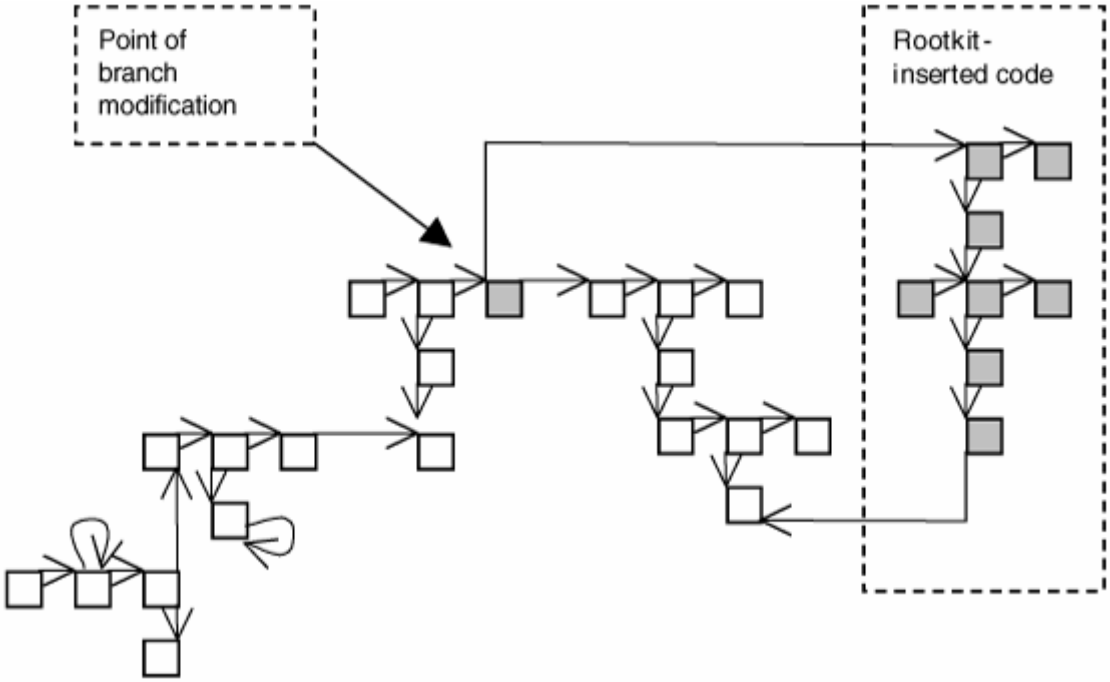


Fig 2.3 Modification of Control flow.
Source: Rootkits: Subverting the Windows Kernel

Interrupt hooking

As the name implies, the Interrupt Descriptor Table (IDT) is used to handle interrupts. Interrupts can originate from software or hardware. The IDT specifies how to process interrupts such as those fired when a key is pressed, when a page fault occurs (entry 0x0E in the IDT), or when a user process requests the attention of the System Service Descriptor Table (SSDT), which is entry 0x2E in Windows. This section will show how to install a hook on the 0x2E vector in the IDT. This hook will get called before the kernel function in the SSDT.

Two points are important to note when dealing with the IDT. First, each processor has its own IDT, which is an issue on multi-processor machines. Hooking just the processor on which our code is currently executing is not sufficient; all the IDTs on the system must be hooked. Also, execution control does not return to the IDT handler, so the typical hook technique of calling the original function, filtering the data, and then returning from the hook will not work.[2] The IDT hook is just a pass-through function and will never regain control, so it cannot filter data. However, our rootkit could identify or block requests from a particular piece of software, such as a Host Intrusion Prevention System (HIPS) or a personal firewall.

When an application needs the assistance of the operating system, NTDLL.DLL loads the EAX register with the index number of the system call in the SSDT and the EDX register with a pointer to the user stack parameters. The NTDLL.DLL then issues an INT 2E instruction. This interrupt is the signal to transfer from userland to the kernel. (Note: Newer versions of Windows use the SYSENTER instruction, as opposed to an INT 2E.

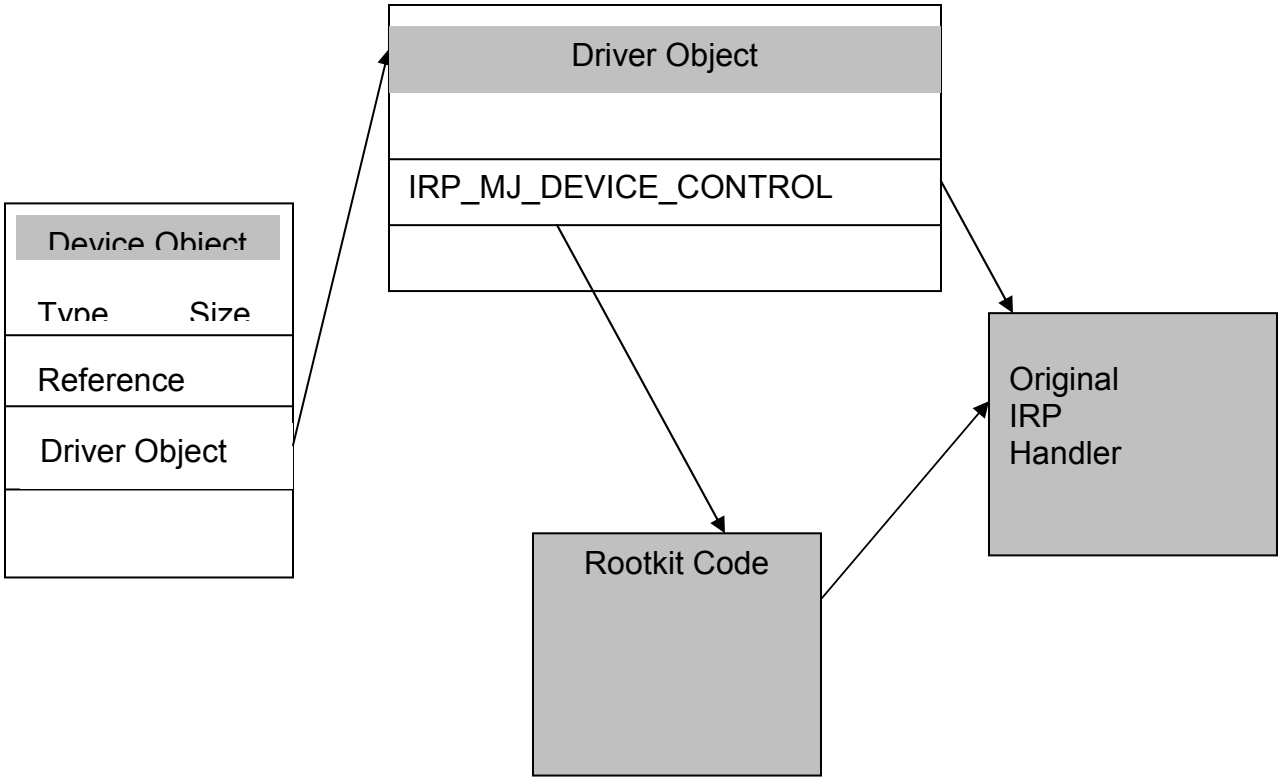


Fig 2.4 Illustration of hooking a driver's IRP table.
Source: Rootkits: Subverting the Windows Kernel

When an application needs the assistance of the operating system, NTDLL.DLL loads the EAX register with the index number of the system call in the SSDT and the EDX register with a pointer to the user stack parameters. The NTDLL.DLL then issues an INT 2E instruction. This interrupt is the signal to transfer from userland to the kernel. (Note: Newer versions of Windows use the SYSENTER instruction, as opposed to an INT 2E.

Each entry within the IDT has its own structure that is 64 bits long. The entries also display this split WORD characteristic. Every entry contains the address of the function that will handle a particular interrupt. The LowOffset and the HiOffset in the IDTENTRY structure comprise the address of the interrupt handler.

Here is the structure of each entry in the IDT:

#pragma pack(1)
typedef struct

```
{
    WORD LowOffset;
    WORD selector;
    BYTE unused_lo;
    unsigned char unused_hi:5; // stored TYPE
    unsigned char DPL:2;
    unsigned char P:1;        // vector is present
    WORD HiOffset;
} IDTENTRY;
#pragma pack()
```

2.2 Direct Kernel Object Manipulation

Direct Kernel Object Manipulation (DKOM) relies upon the fact that the operating system creates kernel objects in order to do bookkeeping and auditing. If a rootkit modifies these kernel objects, it will subvert what the operating system believes exists on the system. By modifying a token object, the rootkit can alter who the operating system believes performed a certain action, thereby subverting any logging. For example, the FU rootkit modifies the kernel object that represents the processes on the system. All the kernel process objects are linked. When a user process such as TaskMgr.exe queries the operating system for the list of processes through an API, Windows walks the linked list of process objects and returns the appropriate information. FU unlinks the process object of the process it is hiding.[1] Therefore, as far as many applications are concerned, the process does not exist. These DKOM tricks are very difficult to detect and extremely powerful! However, they also provide ample opportunity to crackers to crash the whole machine. One way to find a process is by its Process Identifier (PID). The PID is located at an offset within the EPROCESS block that varies depending on the version of the operating system in which the rootkit is running. Here is where determining the operating system version, discussed earlier, will come into play. Based upon current data as of this writing. Various operating-system versions' offsets of the PID within the EPROCESS structure.

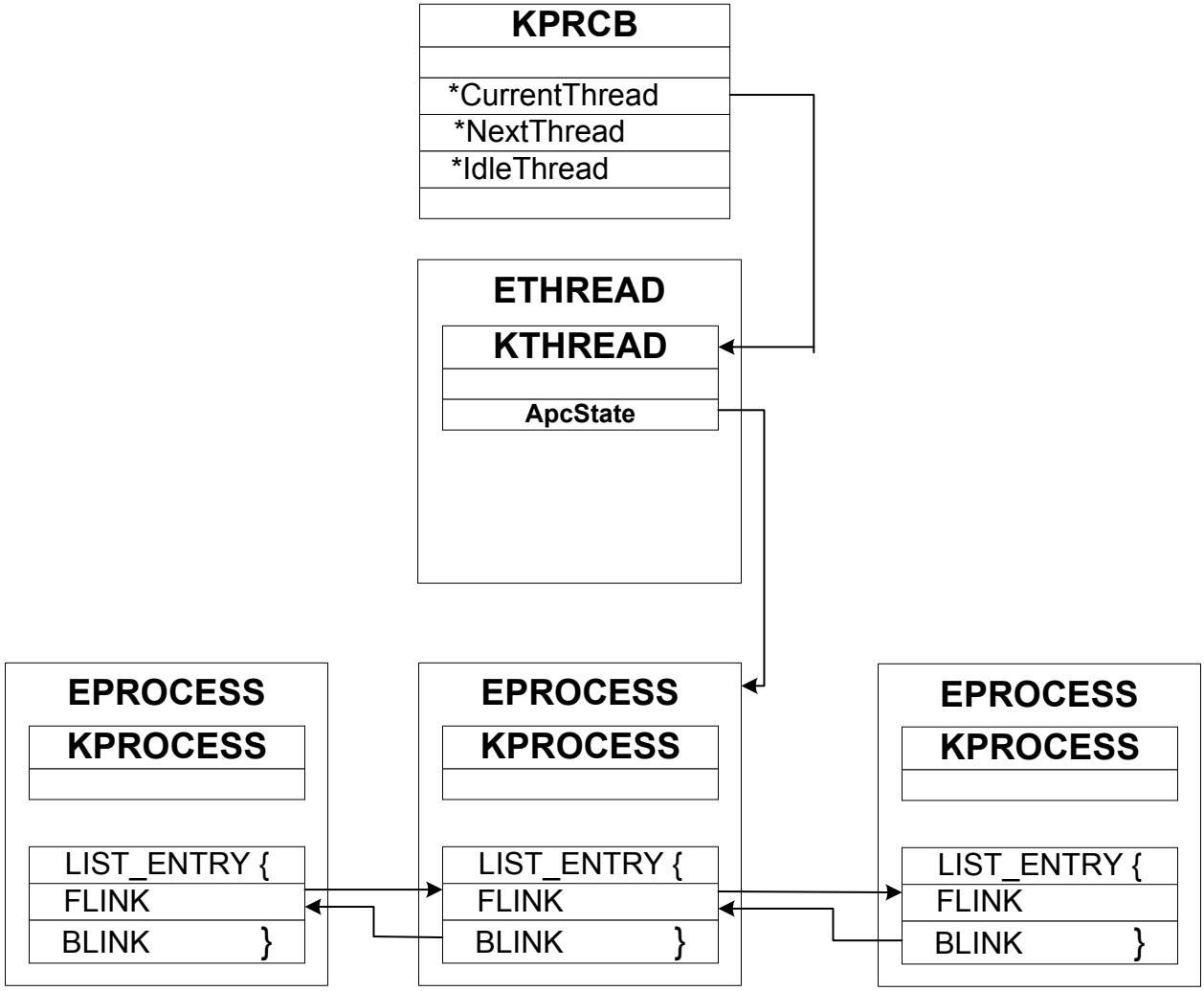


Fig 2.5 Path from KPRCB to the linked list of processes.
Source: Hide 'n' Seek revisited, Full stealth is back.

By modifying a token object, the rootkit can alter who the operating system believes performed a certain action, thereby subverting any logging. For example, the FU rootkit modifies the kernel object that represents the processes on the system.

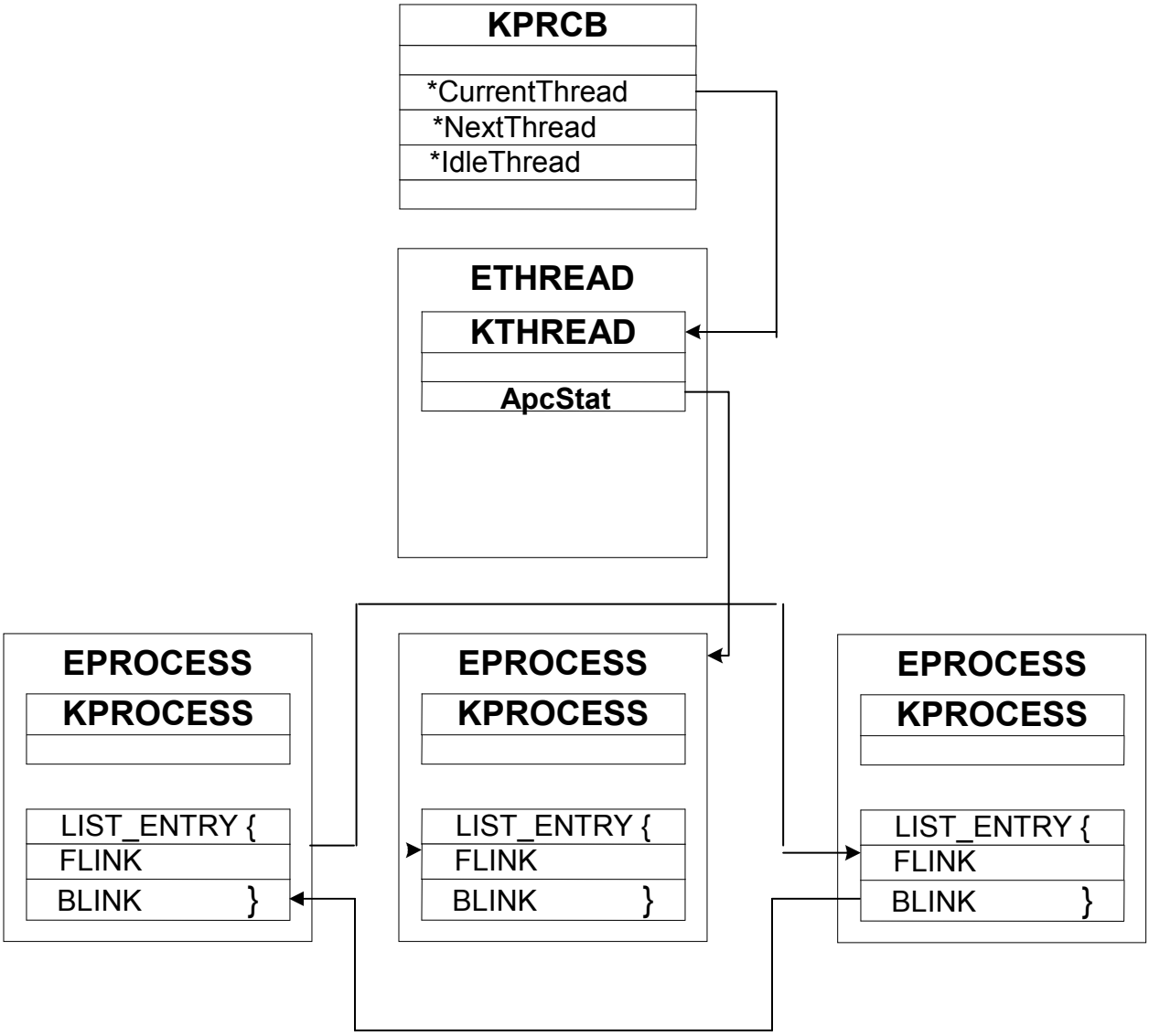


Fig 2.6 Illustration of the active-process list after hiding the current process.
Source: Hide 'n' Seek revisited, Full Stealth is back

Synchronization Issues

Walking the linked list of active processes using the EPROCESS structure directly is dangerous, as is walking the linked list of loaded modules. Processes can be created and torn down by the kernel while the rootkit is swapped out, or by another processor if the rootkit is installed on a multiprocessor system. Also, a driver can be unloaded while the rootkit that had been walking the linked list of modules is swapped out.

To walk the doubly linked list of processes in a safe manner, the rootkit grabs the appropriate mutex, PspActiveProcessMutex. This mutex is not exported by the kernel. PsLoadedModuleResource controls access to the doubly linked list of loaded modules.
One way to find these and other symbols that are not exported is to search memory for a particular pattern. This solution is not very elegant, but empirical evidence suggests it is viable. The drawback to searching memory is that the search pattern is very dynamic and differs with even minor variations in the operating system.

Walking and modifying these lists becomes dangerous only when the rootkit making the modifications is pre-empted by another thread in another process. The kernel dispatcher is responsible for pre-empting the running thread with a new one, and the dispatcher runs at an IRQL of DISPATCH_LEVEL. Therefore, if a thread is running at DISPATCH_LEVEL it should not be pre-empted. However, threads can run on other CPUs in the same computer. So, to avoid pre-emption, we must raise all processors to DISPATCH_LEVEL. The only IRQLs higher than DISPATCH_LEVEL are Device IRQLs (DIRQLs), but these are for processing device hardware interrupts; if we raise the IRQL to DISPATCH_LEVEL across all processors on the machine, we should be relatively safe.

We must be careful regarding what the rootkit does at DISPATCH_LEVEL. Certain functions cannot be called at this elevated IRQL. Also, the rootkit cannot touch any memory that is paged out. If it does, a Blue Screen of Death will occur.

The rootkit will need global variables to keep track of where it is in the process of raising all the CPUs to DISPATCH_LEVEL, and for signaling when to exit. For our purposes, we will call these AllCPURaised and NumberOfRaisedCPU. The AllCPURaised variable acts like a Boolean value. When it is equal to one, all the processors have been raised to DISPATCH_LEVEL; this will signal the individual threads that they can exit. NumberOfRaisedCPU is the total count of CPUs raised to DISPATCH_LEVEL. Use the InterlockedXXX functions to change these globals in an atomic manner.

In the primary code in the rootkit, we need to elevate the IRQL it is running at. Call KeGetCurrentIrql to determine what IRQL we are currently running at. Only if it is less than DISPATCH_LEVEL do we want to call KeRaiseIrql.

2.3 Remote Command, Control, and Exfiltration of Data

A rootkit is installed to gain remote access to a computer. This serves two primary purposes: to control computer software operation, and to copy data from the system. Examples of such command and control include shutting a computer down, enabling or disabling features, and manipulating the kernel. Taking data from a system is typically called exfiltration, or exfil for short. Exfiltration may take such arcane forms as data transmissions over electromagnetic emissions, via extra data inserted into network protocols, and in the form of time delays.

Where remote access is required, the rootkit must be able to communicate over a network. For a TCP/IP network, this could mean via a TCP connection. Once a connection has been established, commands can be issued and data can be exfiltrated.
In the hacker underground, a typical generic solution to the problem of exfil is the remote shell. A remote shell is simply a TCP session connected to the native command interpreter on the system. The command interpreter is supplied with the operating system. On an MS-Windows machine, this would be cmd.exe, and on a UNIX system it may be /bin/sh or /bin/bash.[2]

These command interpreters are actually software programs themselves. Since the command interpreters are already installed on the system before the hacker arrives, the attack program just connects the command interpreter to a network port. In other words, the hacker borrows the existing program when he attacks.

There are, however, cases where hackers have created complex remote-control software. Back Orifice 2000 is one example of a full remote-control system, with file access, screen capture, and even audio bugging.

When engaging in an activity as sensitive as remote penetration, we should be concerned about risk of exposure before anything else. Two concepts that are key to avoiding exposure are minimal footprint and unique structure.[5]

- Minimal footprint: The tools used for remote penetration should affect as little as possible on the remote system. (This is a good reason to design a rootkit that never uses the file system.) This minimizes the chance of detection. Also, fewer lines of code means less complex code, and less complex code means less chance of failure.
- Unique structure: The tools used for remote penetration should have structures and methods that are unique. Virus-detection solutions are always looking for the known. In virus-detection development, a publicly known virus is analyzed for general patterns, and these patterns are then applied to finding unknown viruses. If we attempt to download a rootkit from www.rootkit.com, for example, our virus scanner will likely quarantine the file. If they do not contain patterns found in known infections, then our tools will slip by undetected.

2.4 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 that have the priviledges are capable of injecting and executing code in any other running process.

2.4.1 DLL injection

One of these API calls is CreateRemoteThread() that has the following prototype as per MSDN [12]:

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 of the process specified by hProcess. The only constraint is that the function lpStartAddress points to 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 needed external symbols all processes have LoadLibrary() API in their address space. Abusing this fact lpStartAddress can point to LoadLibrary("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 clean it up when unloaded. In the case of "Nasty_hook.dll" it will install the API hooks using any of the methods described earlier.[8]

2.4.2 Direct memory writing

DLL injection is not the only way of modifying other processes. VirtualAllocEx() and WriteProcessMemory() functions help to inject code into any running process. VirtualAllocEx() allocates memory in the address space of another process, followmyd by WriteProcessMemory() that copies the necessary code there. 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 code must be position independent which is quite complicated to write properly. Position independent code is however quite common in viruses already so that will not stop this method from being used[8].

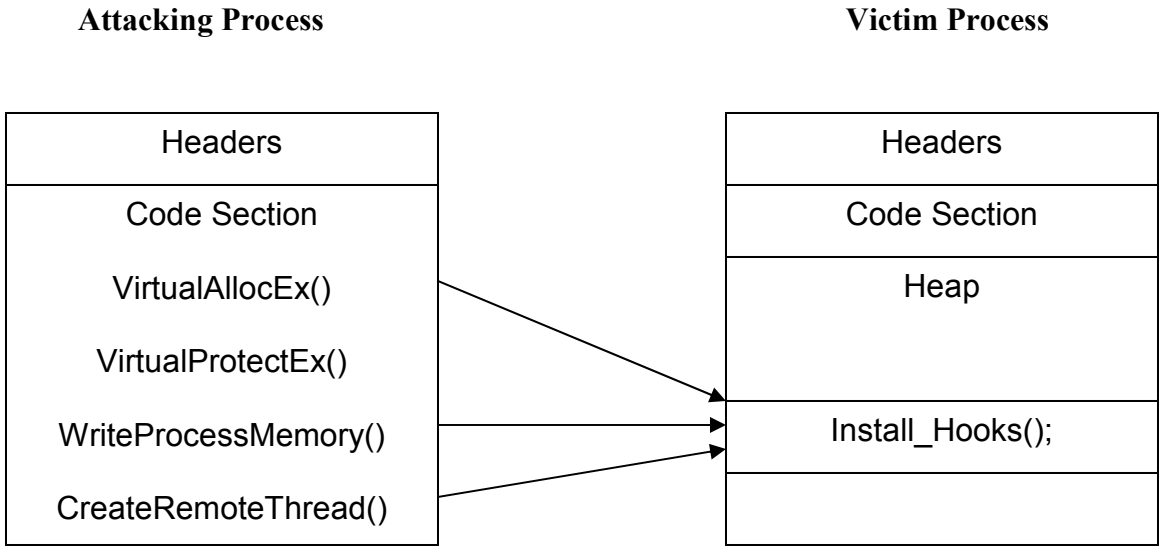


Fig 2.7 Patching another process
Source: Hide 'n' Seek, Anatomy of Stealth Malware

2.5 Hook Installation in NT kernel

Using SystemLoadAndCallImage is a more obscure and completely undocumented way of loading kenel drivers.

2.5.1 Standard device drivers

The simplest – and only documented – way of getting the hooks into the kernel is by installing the standard driver. Driver installation goes the same way as with any other service. 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.[8]

2.5.2 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 upload kernel drivers.[2] 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, SystemInformatin is a pointer to the parameter and SystemINformation Length 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.[1]

2.5.3 Abusing \Device\PhysicalMemory

Even more complex and technically challenging way is to inject code to Ring0 through \Device\PhysicalMemory. By default only SYSTEM account has write access to this device but any process running as administrator can alter that. The code can be injected through \Device\PhysicalMemory and activated with a Callgate provided by x86 processors.[8] Writing code this way is very complex and error prone but possible.

2.6 Detection Technologies

Until several months ago, rootkit detection was largely ignored by security vendors. Many mistakenly classified rootkits in the same category as other viruses and malware. Because of this, security companies continued to use the same detection methods the most prominent one being signature scans on the file system. This is only partially effective. Once a rootkit is loaded in memory is can delete itself on disk, hide its files, or even divert an attempt to open the rootkit file. In this section, we will examine more

recent advances in rootkit detection..

Signature based Detection

Signature based detection methods have been in use by antiviral products for years. The concept is simple. System files are scanned for a sequence of bytes that comprise a "fingerprint" that is unique to a particular rootkit. If the signature is found in a file on the user's system, it signals an infection. As signature scanning has traditionally been applied to the filesystem, its usefulness for rootkit detection is limited unless it is combined with some more advanced detection techniques. This is due to the rootkit's natural propensity to hide files using execution path hooking techniques.[5]

Despite their antiquity, signature based detections are worth mentioning because they may be applied with success to scanning system memory in addition to filesystem scanning. Ironically, most public kernel rootkits are susceptible to signature scans of kernel memory. As kernel drivers, they typically reside in non-paged memory and few, if any, make an effort towards any kind of polymorphic code obfuscation. Thus, a scan of kernel memory should trivially identify most public kernel rootkits regardless of their underlying "bag of tricks" (DKOM, SSDT, IDT hooking and the like). The key words in that last sentence, however, are "public rootkits" because signature based detection is, by definition, useless against malware for which a known signature does not exist.[5]

Heuristic / Behavioral detection

Where signature based detections fall short, heuristic detections take over. Their primary advantage lies in their ability to identify new, previously unidentified rootkits. They work by recognizing deviations in "normal" system patterns or behaviors. Various heuristics have been proposed for identifying rootkits based upon execution path hooking.

For example, VICE is a freeware tool written to detect hooks. It is a standalone program that installs a device driver to analyze both user mode applications and the operating system kernel. In the kernel, VICE checks the SSDT for function pointers that do not resolve to ntoskrnl.exe. Also, we can add devices to the file "driver.ini," and VICE will

check the IRP major function table of the corresponding driver. If a function pointer in the IRP major function table of a driver does not consist of an address within the driver, then the IRP has been hooked by an outside driver or piece of kernel code[2].

Cross view based detection

Cross view based detection techniques are relatively new and show a lot of promise. These techniques assume that the operating system has been subverted, but this method leverages the fact that there is usually more than one way to ask the same question. In a cross view based detection method, the detection software calls the common APIs to enumerate key elements within the computer system such as the list of files, processes, or Registry keys. However, to be successful, the detection software must also have an algorithm to generate a similar data set that does not rely upon the common APIs. Any difference in these two data sets reveals something hidden because it did not exist in the data set generated using the common APIs. Cross view based detection relies upon the fact that API hooking or manipulation of kernel data structures will taint the data returned by the operating system APIs, but the low level methods used to glean the same information will be designed in such a manner as not to be subverted by hooks or DKOM tricks.[5]

For example, Rootkit Revealer is available from Sysinternals. Sysinternals provides many free utilities for System Administrators and developers alike. Rootkit Revealer targets a subset of rootkits called persistent rootkits. Persistent rootkits are those that exist between reboots. In other words, a memory resident rootkit would not be persistent. In order to be persistent between reboots, Rootkit Revealer assumes that the rootkit will have to exist on the filesystem and in the Windows Registry. Rootkit Revealer uses a cross view based approach to detect these persistent rootkits. Rootkit Revealer calls the highest level APIs in order to enumerate the files on disk and the Registry keys. To compare against this, Rootkit Revealer parses the raw filesystem structure on disk and the bare files that comprise the Registry hives.[5]

Integrity based detection

Integrity based detection provides an alternative to both signatures and heuristics. It relies upon comparing a current snapshot of the filesystem or memory with a known, trusted baseline. Differences between the current and baseline snapshots are taken as evidence of malicious activity. Unfortunately, however, the integrity checker is usually not capable of pinpointing the origin of the activity that has caused the changes.

Tripwire is a disk based integrity checker . It creates a trusted database of unique CRC hash values for each of the system files on a user's hard disk. When the user initiates a scan for malicious activity, it recalculates the CRC's for all files and compares them to the original CRC's in the database. It is based upon observation that system files should not change (except perhaps in the rare system update or service pack), thus a mismatch indicates compromise. Tripwire was very effective against early rootkits that simply replaced system files on the disk with trojanized versions. Unfortunately, rootkits adapted by moving their modifications from disk to memory. This renders Tripwire virtually useless for modern rootkit detection. Nevertheless, where disk based integrity checking fails, memory based integrity checking may succeed.[5]

Chapter 3: **Problem Statement**

Problem Statement

The threat of rootkit is growing in the present security scenario. With the integration of rootkit in the already dangerous viruses like Bagle, it is posing even greater threat as these viruses will become harder and harder to detect due to stealthy techniques of rootkits aiding to their purpose. The recent disclosure of rootkits in the online poker gaming portals indicates that the danger posed by rootkits is far from over.

Rootkits do their primary job of bringing stealth by hiding files, processes, open ports, registry keys. The kernel provides separate data structures and functions in order to store and display the information related to each of these sub tasks. For some of these sub tasks, there are more than one data structure and functions at different privilege levels in the operating system. In other words, these points of attack can be at the user space level or at the kernel space level or at the hybrid level. With such a variance in their subversive methods, one has to know each and every target or point of attack of rootkits, if an effective avoidance or detection technology has to be developed.

This leads to the motivation to for us to study the different techniques and find out each point of attack that leads to subverting the Windows operating system. In order to find such points of subversion, we should first understand thoroughly the Windows internals, that is how Windows operating system works from inside when we instruct it to list the files, list the process, list the open ports and list the registry values. It will not only entail knowing what occurs at the user level, (that is which dll's are called for every subtask) but also deep down in the kernel, which all native api's are being referred to until the data is retrieved from the deepest level like process blocks, raw file data and so on.

This has to be followed by setting up the environment where we could run the rootkits safely without affecting any production machine and observe the behavior of the rootkits. Windows Kernel debugger like WinDbg will be play a crucial role in the reverse engineering process of rootkits because only with the debugger's help we can step

through the kernel and see it for ourselves how step by step the machine language instructions are traversed and where and when different functions are entered and exited. However, wherever rootkits source code is available publicly, we will also attempt to directly examine the source code in order to have as much idea as possible, how the task is achieved.

Lastly the fruits of our labour will only be meaningful and beneficial, if the discovered information can be utilized systematically for developing effective avoidance and detection techniques. Hence our endeavour is to develop a classification scheme in which all the targeted API's for every subtask is linked to each successive genre of rootkits.

4.1 Implementation

Following are the implementation steps carried out as part of my experiment**.**

4.1.1 Setting up an implementation environment

Setting up a remote environment is a necessity to debug something like rootkit because there is strong likelihood that a rootkits can crash the system. Even a slight mistake here and there in the rootkit program or in the rootkit loader program can potentially crash the system.

Kernel rootkits are installed as loadable modules or device drivers and provide hardware-level access to a machine. There are two ways to debug a device-driver in Windows. The first is to install the driver on our development machine and run a local kernel-mode debugger such as SoftIce. Although SoftIce is a very powerful tool, debugging a driver on our development machine is not a good idea because we will be guaranteed to destroy all our work if the driver crashes the system. We really need to install our driver on a separate machine and use remote kernel-debugging with a null-modem serial cable to protect our development box.

We will be using virtual-machine technology (Microsoft VirtualPC), and using named pipes to emulate the serial-port connections.[20]

4.1.1.1 Configure Virtual PC to be the remote host

Once we have Virtual PC, following steps mentioned below enables the virtual machine to act as a both kernel-debug and user mode debug target for WinDbg[17]. The idea is that instead of using a null-modem cable between physical machines, we can set up a named pipe and attach it to the virtual machine's COM port. WinDbg will then attach to the named pipe, and remote-debug the hosted OS inside Virtual PC. The steps of the configuration are given in Appendix A.

As a result of the successful previous operation, we were able to get the following screen at boot up time.

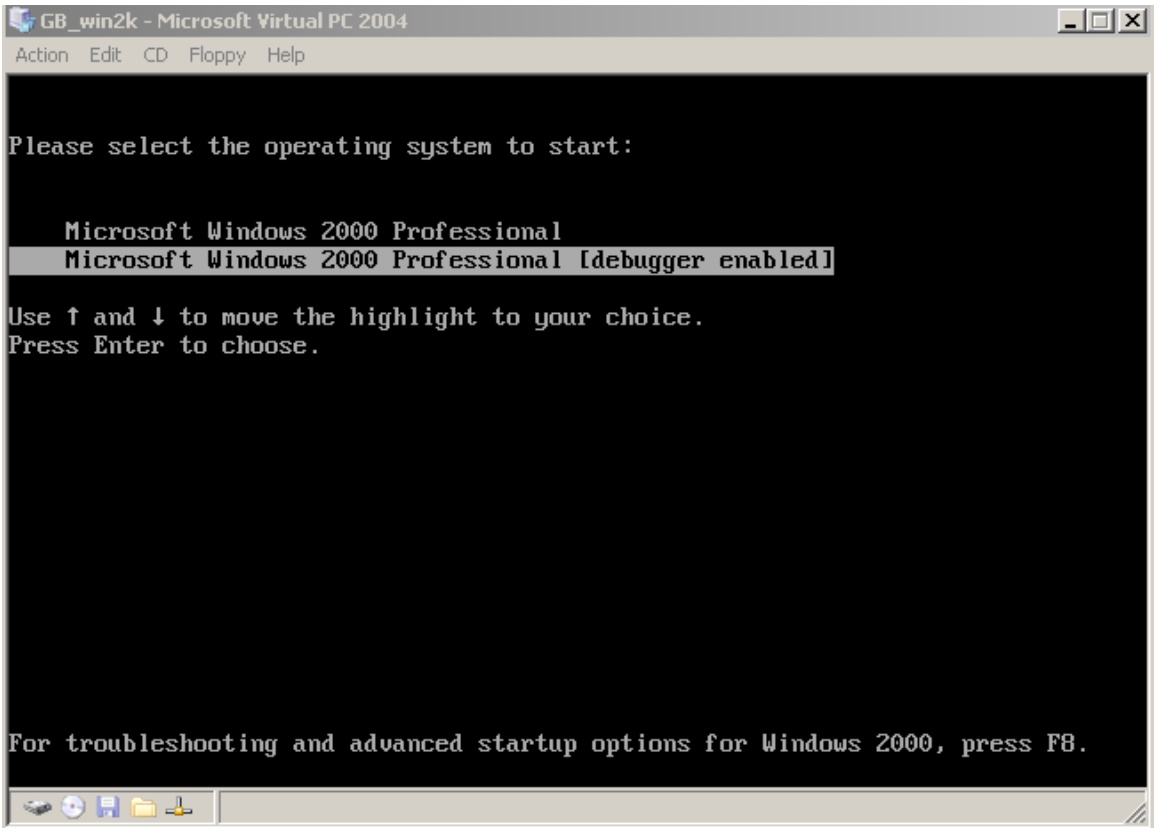


Fig 4.1 Illustration of the debugger enabled Windows 2000 OS in Virtual PC

4.1.1.2 Matching debug information by the use of symbols

Debug information in Windows environment is contained in the symbol files. Debug information (symbols) helps debuggers to analyze the internal layout of the debugged application. In particular, it helps the debugger to locate addresses of variables and functions, display values of variables (including complex structures and classes with nontrivial binary layout), and map raw addresses in the executable to the lines of the source code.

When we modify the source code and rebuild the executable, its internal layout changes. Some functions and variables can move to other locations, structures and classes can be

extended with new members while some old members can be removed, and so on. These changes should be properly reflected in the debug information, which also must be updated to correctly describe the new layout of the executable. [17]

Debug information is often stored separately from the executable, usually in a PDB or DBG file. One of the main purposes of these files is to allow a debugger or disassembler to look up the nearest symbolic name that can be attributed to a given binary address within a module. If a disassembler, for example, finds out that the next assembly language instruction to be displayed is call72A05A2Eh, it would be great to provide the user with the real name that is associated to the function entry point 0x72A05A2E, such as call_pMemAlloc@4. In the worst case, the debugger will not be able to display variables and step through the source code at all. As a result, effective debugging is not possible, and the reason is that debug information does not match the executable.

In order to match the debug symbols, all the symbols files of the debugee application along with OS symbols are required to be stored in the host environment. Otherwise any debugging exercise would have been impossible. [16]

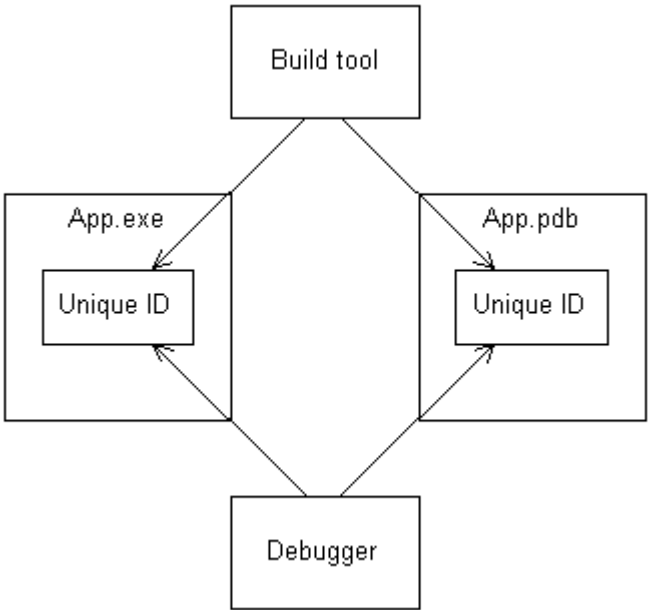


Figure 4.2 Matching Debug Information

Source: http://www.debuginfo.com/articles/debuginfomatch.html

4.1.1.3 Building the Driver

Most of the rootkits were driver programs as they work only at higher privilege levels. Using Windows Driver Development Kit's (DDK) built utility we built the drivers from the source codes of kernel level rootkits[18].

The Windows DDK includes an environment for building drivers that closely matches the build environment used internally at Microsoft®. This environment includes various tools useful in developing drivers, including build utilities (build.exe and nmake.exe), a C compiler (cl.exe), and a linker (link.exe). The Build utility automatically invokes nmake, the compiler, and the linker according to the command-line options we specify.

The driver is the part of the package that provides the I/O interface for a device. Typically, a driver is a dynamic-link library with the *.sys* filename extension. When a device is installed, Setup copies the *.sys* file to the *%windir%\system32\drivers* directory. The software required to support a particular device depends on the features of the device and the bus or port to which it connects. Microsoft ships drivers for many common devices and nearly all buses with the operating system.

The steps for building the driver are detailed in Appendix B.

4.1.1.4 Loading the driver

After building the corresponding driver of the rootkit source code, we installed the driver of with the help of a driver loader program We obtained from the www.rootkit.com site[22]. This driver loader program installed the driver in the kernel.

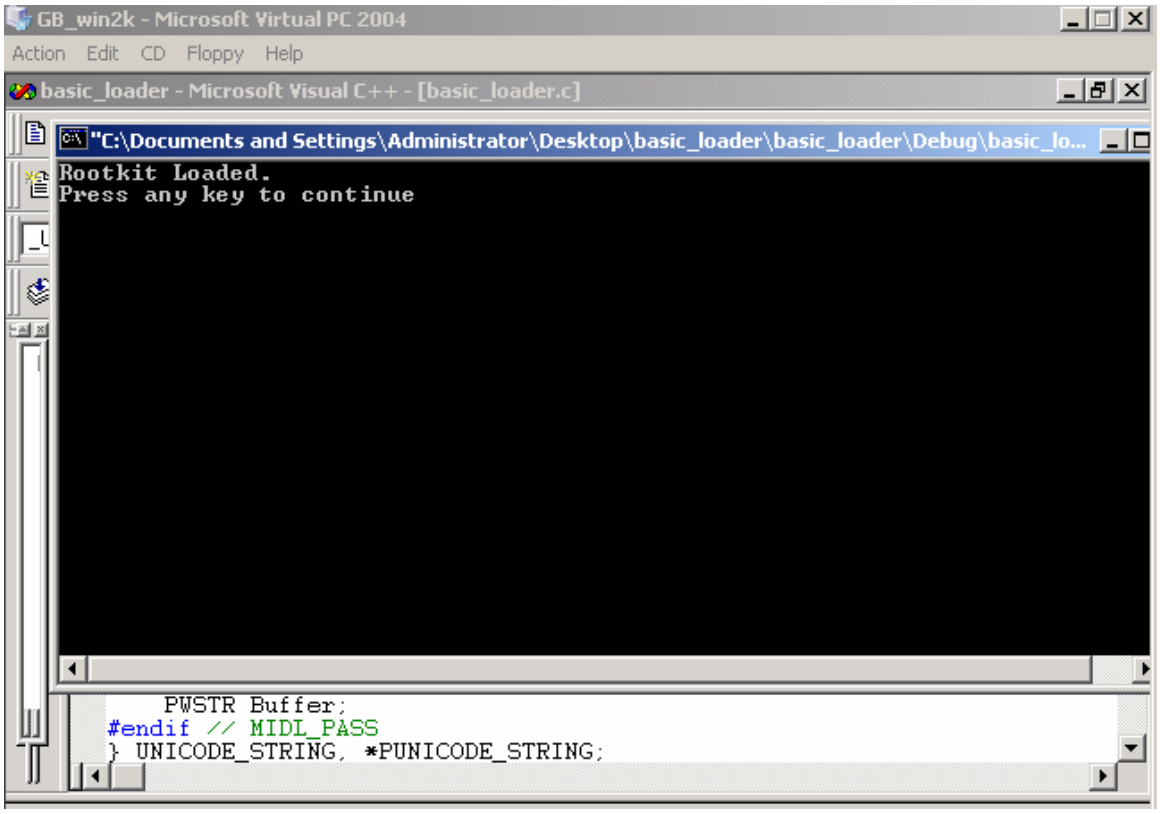


Figure 4.3 Illustration of the loading of driver.

4.1.1.5 Debugging using WinDbg

Once the OS had started booting, we simply double-click our WinDbg shortcut and WinDbg automatically attached itself to the Virtual PC session (using the named pipe "com_1") and entered the kernel-debug mode. Once we have "broken in" to the debuggee, the entire virtual machine will be stopped - not even the mouse will be active inside Virtual PC. [17]

4.2 Experimental Results

The primary functions of rootkits are to hide files, processes, registry keys, network ports. Based on these primary functions that rootkits carry out, we tried to find specific functions and structures that rootkits target to accomplish their task. We have explored

such functions for each category with the aid of debugger, direct examination of publicly available rootkits' source code and documentation available at microsoft website.

Categories of functions have been numbered and within each category we have written the names of function followed by the actual purpose of those functions and rootkits' way of manipulation.

4.2.1 Hiding files

Following are the names of the functions discovered which are typically targeted to hide files from the user. We have also written the description of these functions and how they are manipulated to achieve their objective.

- **Kernel32.dll! FindFirstFile & Kernel32.dll! FindNextFile:**

  The experiment began with the successful loading of NTIllusion rootkit[14]. Then we attached the debugger to the explorer.exe process. Thereafter whenever any explorer window was opened, the first function called by explorer.exe was **Kernel32.dll! FindFirstFile.** By stepping through this disassembled function in WinDbg, we found FindFirstFile calls another function after few instructions. This function was not of Kernel32.dll but of NTIllusion's.

Now according the MSDN, the FindFirstFile function searches a directory for a file or subdirectory whose name matches the specified name[12].

```
HANDLE FindFirstFile(
        LPCTSTR lpFileName,
        LPWIN32_FIND_DATA lpFindFileData
);
```

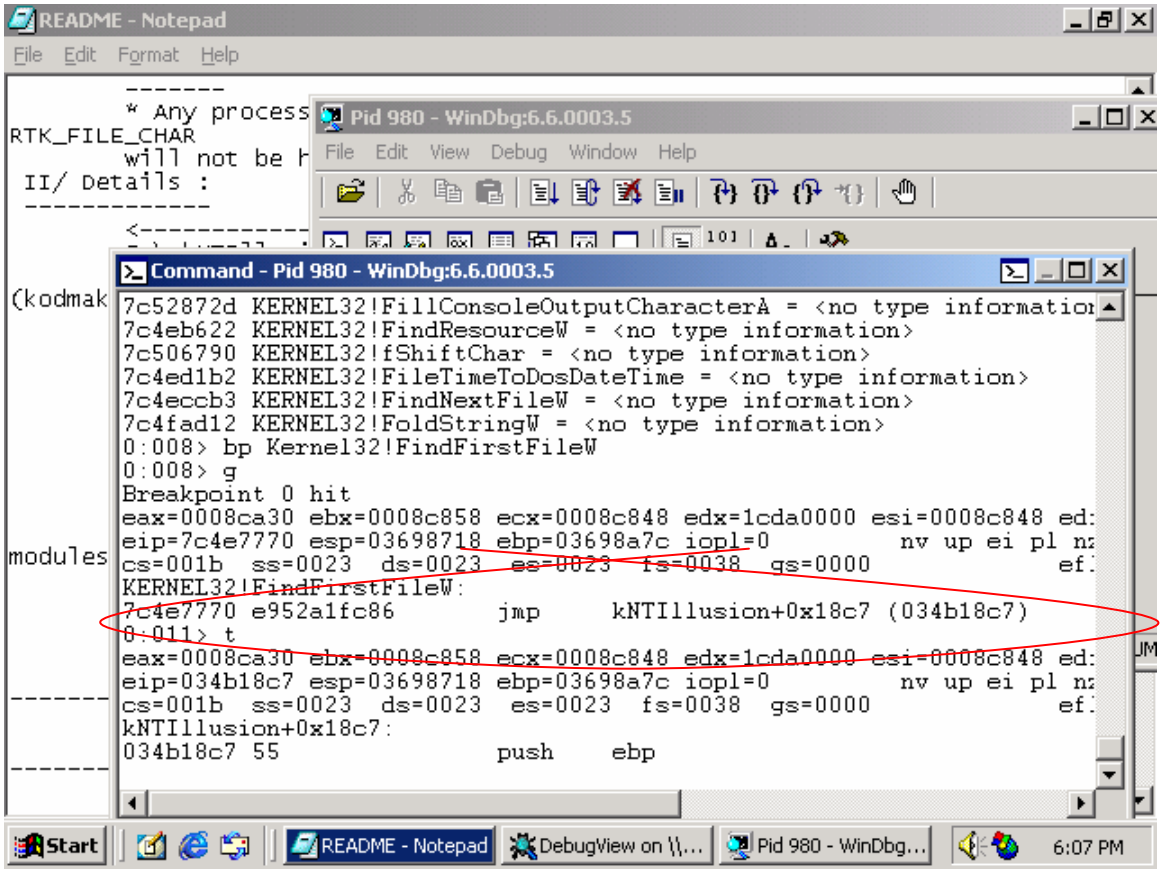


Figure 4.4 Illustration of the jump to rootkit's function from within the FindFirstFile function.

The FindFirstFile function is called to begin a file search. If it succeed, the search may be pursued by calling FindNextFile.

BOOL FindNextFile(

HANDLE hFindFile,

LPWIN32_FIND_DATA lpFindFileData

);

To perform a directory listing, an application calls FindFirstFile, and then calls FindNextFile using the returned handle, until it returns zero.

Written below is the code snippet of NTIllusion rootkit's own FindFirstFileA. We have written the comments in bold on relevant parts to show how it hides the file.

```
HANDLE WINAPI MyFindFirstFileA (LPCTSTR lpFileName, LPWIN32_FIND_DATA
lpFindFileData)
{
        HANDLE hret= (HANDLE) 1000;
        int go_on=1;
        hret = (HANDLE) fFindFirstFileA (lpFileName, lpFindFileData);
```

/* This is where filtering is done. Whenever it finds the file starting with pre-determined RTK_FILE_CHARACTER, it ignores the file and proceeds with the next file */

```
        while( go_on &&  !strnicmp(lpFindFileData->cFileName, RTK_FILE_CHAR,
        strlen(RTK_FILE_CHAR)))
        {
                go_on = fFindNextFileA(hret, lpFindFileData);
        }

        if(!go_on)
                return INVALID_HANDLE_VALUE;

        return hret;
}
```

- **NtQueryDirectory :**

Kernel level rootkits make use of this particular function for file hiding. FindNextFile in Kernel32.dll calls into Ntdll.dll. Ntdll.dll loads the EAX register with the system service number for FindNextFile's equivalent kernel function, which happens to be NtQueryDirectoryFile. Ntdll.dll also loads EDX with the user space address of the

parameters to FindNextFile. Ntdll.dll then issues an INT 2E or a SYSENTER instruction to trap to the kernel.

On examination from msdn and in WinDbg, it was discovered NtQueryDirectory is a function used for file enumeration[12].

NTSTATUS NtQueryDirectoryFile (
IN HANDLE FileHandle,
IN HANDLE Event OPTIONAL,
IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
IN PVOID ApcContext OPTIONAL,
OUT PIO_STATUS_BLOCK IoStatusBlock,
OUT PVOID FileInformation,
IN ULONG FileInformationLength,
IN FILE_INFORMATION_CLASS FileInformationClass,
IN BOOLEAN ReturnSingleEntry,
IN PUNICODE_STRING FileName OPTIONAL,
IN BOOLEAN RestartScan
);

Here FileHandle is the directory object's handle. FileInformation points to allocated memory where the function writes data. FileInformationClass determines type of record written in FileInformation.

FileInformationClass can have different types but four enumerative types are interesting.

#define FileDirectoryInformation 1
#define FileFullDirectoryInformation 2
#define FileBothDirectoryInformation 3

#define FileNamesInformation 12

Structure of record written in FileInformation for FileDirectoryInformation:

```
typedef struct _FILE_DIRECTORY_INFORMATION {
        ULONG NextEntryOffset;
        ULONG Unknown;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
        ULONG FileAttributes;
        ULONG FileNameLength;
        WCHAR FileName[1];
}FILE_DIRECTORY_INFORMATION, PFILE_DIRECTORY_INFORMATION;
```

For FileFullDirectoryInformation:

```
typedef struct _FILE_FULL_DIRECTORY_INFORMATION {
        ULONG NextEntryOffset;
        ULONG Unknown;
        LARGE_INTEGER CreationTime;
        LARGE_INTEGER LastAccessTime;
        LARGE_INTEGER LastWriteTime;
        LARGE_INTEGER ChangeTime;
        LARGE_INTEGER EndOfFile;
        LARGE_INTEGER AllocationSize;
        ULONG FileAttributes;
        ULONG FileNameLength;
```

```
        ULONG EaInformationLength;
        WCHAR FileName[1];
}FILE_FULL_DIRECTORY_INFORMATION,
PFILE_FULL_DIRECTORY_INFORMATION;
```

and for FileNamesInformation:

```
typedef struct _FILE_NAMES_INFORMATION {
        ULONG NextEntryOffset;
        ULONG Unknown;
        ULONG FileNameLength;
        WCHAR FileName[1];
} FILE_NAMES_INFORMATION, *PFILE_NAMES_INFORMATION;
```

NtQueryDirectory writes a list of these structures in FileInformation. Only three variables are important for us in any of these structure types. First item can be found on address FileInformation + 0. So the second item is on address FileInformation + NextEntryOffset of first one.

FileName is a full name of the file.
FileNameLength is a length of file name.

If file has to be hidden, four types of have to be told apart and for each returned record we need to compare its name with the one which we want to hide. If we want to hide first record, we have to move following structures by the size of the first. This will cause the first record to be rewritten.

If no record, which should be seen, was found, returned error will be STATUS_NO_SUCH_FILE.

```
#define STATUS_NO_SUCH_FILE 0xC000000F
```

- **NtVdmControl :**

NTSTATUS NtVdmControl(
IN ULONG ControlCode,
IN PVOID ControlData
);

ControlCode specifies the subfunction which is applied on data in ControlData buffer. If ControlCode equals to VdmDirectoryFile this function does the same as NtQueryDirectoryFile with FileInformationClass set on FileBothDirectoryInformation.

#define VdmDirectoryFile 6

Then ControlData is used like FileInformation. The only difference here is that we do not know the length of this buffer. So we have to count it manually. We have to add NextEntryOffset of all records and FileNameLength of the last record and 0x5E as a length of the last record excluding the name of the file. Hiding methods are the same as in NtQueryDirectoryFile then.[21]

4.2.2 Hiding Registry Keys

Windows registry is quite big tree structure containing two important types of records which needs to be hidden. First type is registry keys, second is values. Owing to registry structure, hiding registry keys is not as simple as hiding file or process.[7]

- **NtEnumerateKey:**

According to MSDN, one can get information about one key specified by its index in some part of registry. This is provided by NtEnumerateKey. By hooking this function and then altering the output, one can hide the value of a registry key. The prototype of the function is:

NTSTATUS NtEnumerateKey(

IN HANDLE KeyHandle,

IN ULONG Index,

IN KEY_INFORMATION_CLASS KeyInformationClass,

OUT PVOID KeyInformation,

IN ULONG KeyInformationLength,

OUT PULONG ResultLength

);

KeyHandle is a handle to a key in which we want to get information about subkey specified by Index. Type of returned information is specified by KeyInformationClass. Data are written to KeyInformation buffer which length is KeyInformationLength. Number of written bytes is returned in ResultLength.

- **RegEnumValueKey :**

```
LONG WINAPI MyRegEnumValue(
HKEY    hKey,
DWORD   dwIndex,
LPWSTR  lpValueName,
LPDWORD lpcValueName,
LPDWORD lpReserved,
LPDWORD lpType,
LPBYTE  lpData,
LPDWORD lpcbData)
{
        LONG lRet; /* return value */
        char buf[256];
        /* genuine API is called, only is needed hiding is done after  */
        lRet = fRegEnumValueW(hKey,dwIndex,lpValueName,lpcValueName,
         lpReserved, lpType, lpData,lpcbData);
```

```
WideCharToMultiByte(CP_ACP, 0,lpValueName, -1, buf, 255,NULL,
NULL);

/* If  hiding needs to be done*/
if(!_strnicmp((char*)buf, RTK_REG_CHAR, strlen(RTK_REG_CHAR)))
{
lRet=1; /* then return –1 (error) */
}

  return lRet;
}
```

4.2.3 Hiding Process

Following are the names of the functions that we discovered whose manipulations enable rootkits to hide processes in the system.

- **SendMessageW:**

By hooking this particular function inside task manager, one can install a hook at listbox level to prevent a process from being shown.

According to MSDN, task manager uses the row telling about system idling process and changes its name to show the newly created process by sending a LVM_SETITEMTEXT message. So, first it overwrites the content of this listBox item's line, and then adds a new "Idle process" line by sending a LVM_INSERTITEMW message.

One can control what task manager shows (at display level) by filtering these two types of messages.

```
                /* Filterevents */
if( Msg==LVM_SETITEM || Msg==LVM_INSERTITEMW ||
Msg==LVM_SETITEMTEXTW)
{
        /* Hide process, if it starts by _NT*/
        if( ((char)(pit->pszText))=='_NT' )
        {
                hWnd=Msg=wParam=lParam=NULL;
                return 0;
        }
}

        /* else call original function */
return fSendMessageW(hWnd,Msg,wParam,lParam);
```

- **NtQuerySystemInformation:**

Task manager performs direct calls to ntdll.NtQuerySystemInformation. Retrieving the processes list is done through a call to the [NtQuerySystemInformation] API[9].

```
NTSTATUS NtQuerySystemInformation(
  SYSTEM_INFORMATION_CLASS SystemInformationClass,
  PVOID SystemInformation,
  ULONG SystemInformationLength,
  PULONG ReturnLength
);
```

According to MSDN, the NtQuerySystemInformation function retrieves various kinds of system information. When specifying SystemInformationClass to be equal to SystemProcessInformation, the API returns an array of SYSTEM_PROCESS_INFORMATION structures, one for each process running in the

system. These structures contain information about the resource usage of each process, including the number of handles used by the process, the peak page-file usage, and the number of memory pages that the process has allocated. The function returns an array of SYSTEM_PROCESS_INFORMATION structures though the SystemInformation parameter.[9]

Each structure has the following layout:

```
typedef struct _SYSTEM_PROCESS_INFORMATION
{
   DWORD        NextEntryDelta;
   DWORD        dThreadCount;
   DWORD        dReserved01;
   DWORD        dReserved02;
   DWORD        dReserved03;
   DWORD        dReserved04;
   DWORD        dReserved05;
   DWORD        dReserved06;
   FILETIME     ftCreateTime;       /* relative to 01-01-1601 */
   FILETIME     ftUserTime;                 /* 100 nsec units */
   FILETIME     ftKernelTime;       /* 100 nsec units */
   UNICODE_STRING ProcessName;
   DWORD        BasePriority;
   DWORD        dUniqueProcessId;
   DWORD        dParentProcessID;
   DWORD        dHandleCount;
   DWORD        dReserved07;
   DWORD        dReserved08;
   DWORD        VmCounters;
   DWORD        dCommitCharge;
   SYSTEM_THREAD_INFORMATION  ThreadInfos[1];
```

} SYSTEM_PROCESS_INFORMATION,
**PSYSTEM_PROCESS_INFORMATION;*

A process can be hidden if we make alterations in the linked list of this particular structure. NextEntryDelta member of the structure, which represents an offset to the next SYSTEM_PROCESS_INFORMATION entry, can be made to skip the entry of desired process to be hidden.

- **System Service Descriptor Table:**

Windows executive runs in kernel mode and provides native support to all of the operating system's subsystems: Win32, POSIX, and OS/2. These native system services' addresses are listed in a kernel structure called the System Service Dispatch Table (SSDT).[5] This table can be indexed by system call number to locate the address of the function in memory. Another table, called the System Service Parameter Table (SSPT), specifies the number of bytes for the function parameters for each system service.

To call a specific function, the system service dispatcher, KiSystemService, simply takes the ID number of the desired function and multiplies it by 4 to get the offset into the SSDT.

A system service dispatch is triggered when an INT 2E or SYSENTER instruction is called. This causes a process to transition into kernel mode by calling the system service dispatcher. An application can call the system service dispatcher, KiSystemService, directly, or through the use of the subsystem. If the subsystem (such as Win32) is used, it calls into Ntdll.dll, which loads EAX with the system service identifier number or index of the system function requested. It then loads EDX with the address of the function parameters in user mode. The system service dispatcher verifies the number of parameters, and copies them from the user stack onto the kernel stack. It then calls the function stored at the address indexed in the SSDT by the service identifier number in EAX.

Once our rootkit is loaded as a device driver, it can change the SSDT to point to a function it provides instead of into Ntoskrnl.exe or Win32k.sys. When a non-kernel application calls into the kernel, the request is processed by the system service dispatcher, and the rootkit's function is called. At this point, the rootkit can pass back whatever bogus information it wants to the application, effectively hiding itself and the resources it uses.[6]

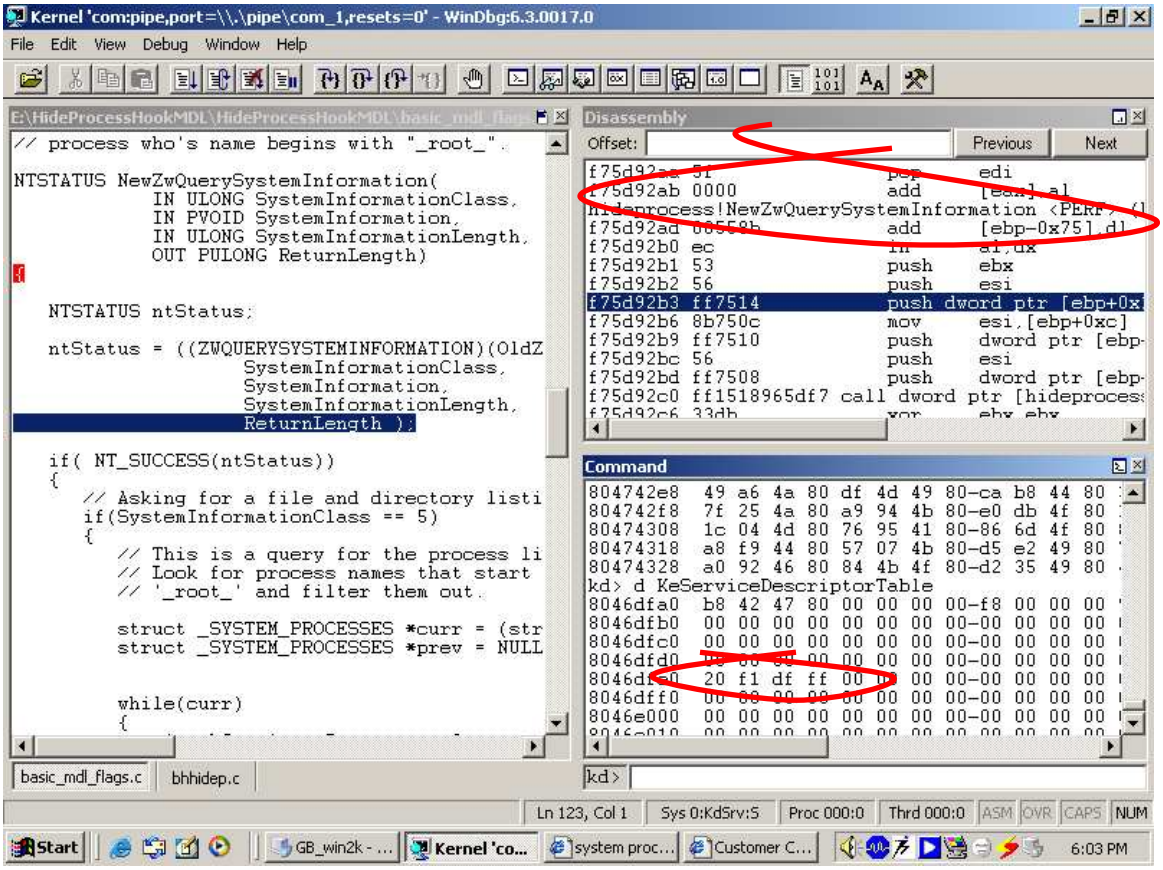


Figure 4.5: Illustration of rootkit's function being called and address falling out of range.

4.2.4 Hiding Open ports

Tools like **netstat, fport, tcpview**, in order to obtain information about network connections make call to DeviceIoControl which in turn make request and receive calls from tcp and udp drivers (/device/tcp and /device/udp).

- **GetTcpTable:**

By hooking GetTcpTable, one can change the network statistics at user space level instead of kernel level. [6]

TCP connections implemented as an array of MIB_TCPROW structures, one for each connection.

```
typedef struct _MIB_TCPROW {
        DWORD dwState;
        DWORD dwLocalAddr;
        DWORD dwLocalPort;
        DWORD dwRemoteAddr;
        DWORD dwRemotePort;
} MIB_TCPROW, *PMIB_TCPROW;
```

While the dwState describes the state of a given connection, dwLocalAddr, dwLocalPort, dwRemoteAddr, dwRemotePort inform about the source and destination of the connection.

dwLocalPort and dwRemotePort are exploited to determine if the port belongs to the secret range. If true, then that structure is wiped off and rest of the records are adjusted in the memory.

- **CharToOemBuffA:**

This function is used in netstat[11] and similar tools to perform character set translations for each line before it is written to console output. Whenever the turn is of string containing hidden port to be translated, the replaced CharToOemBuffA function calls the function with a blank buffer, so the translation results in a blank buffer, and output doesn't show anything.

```
BOOL WINAPI MyCharToOemBuff(LPCTSTR lpszSrc, LPSTR lpszDst, DWORD
cchDstLength)
{
        /* If  port range is there , drop the line. */

    if(strstr(lpszSrc,(char*)RTK_PORT_HIDE_STR)!=NULL)
```

```
        {
                return (*fCharToOemBuffA)("", lpszDst, cchDstLength);
        }
        return (*fCharToOemBuffA)(lpszSrc, lpszDst, cchDstLength);
}
```

- **WriteFile:**

Similar to the previous case, Fport processes output but character by character. If the output mode is changed from character by character to line by line, one can hide those lines that contain port numbers [15].

Rootkits occupy the pinnacle of the attacker's formidable tool set because they yield ultimate control over a target machine[mag]. The best defence against this formidable enemy can only be developed once we know about its all forms of attack. Rootkits either alter the execution path of the operating system (hooking) or directly attack the data that stores information about processes, drivers, network connections, etc(DKOM).

5.1 Conclusion

All information about processes, drivers, network connections are stored in the various dedicated data structures within the kernel space. User can avail user-level API's or even native API's directly to access this information. There are various levels through which this information passes i.e. from the low level where information is actually stored to the kernel/user space interface level space to the highest level where this information is finally displayed. Every such interface where transfer of information is taking place becomes a potential target for rootkits to replace their own function which does the job. We have summarized below the list of function which have been most likely targets for each category of process hiding, registry key hiding, file hiding, ports hiding.

Table 5.1 DKOM based rootkits with targeted objects

Process Hiding: EPROCESS object
File Hiding: MODULE_ENTRY LIST_ENTRY

Table 5.2 Hooking Based Rootkits with targeted APIs

IAT Hooking **(API targeted)**	**SSDT Hooking** **(API targeted)**
File Hiding Kernel32.dll!FindFirstFile Kernel32.dll!FindNextFile NtQueryDirectoryFile NtVdmControl	ZwQueryDirectoryFile
Process Hiding SendMessageW NtQuerySystemInformation	ZwQuerySystemInformation
RegistryKey Hiding NtEnumerateKey NtEnumerateValueKey	ZwEnumerateKey ZwEnumerateValueKey
Port Hiding GetTcpTable WriteFile CharToOemBuffA	NTdeviceControl

5.2 Summary of Contributions

1. The diversified points of attack coupled with its basic nature of stealth makes it difficult to detect the presence of rootkits, leave alone studying its characteristics. My contribution in this regard has been to collect various publicly available rootkits from various sources and then setting up an environment to study them without affecting the own system. The detailed steps for setting up a remote debugger has been given in section 4.1.

Debugging kenel level rootkit was an arduous task but it has comprehensively discussed in the section.

2. After studying various rootkits and various sources of information, we have classified the rootkits based on the techniques used by them in order to achieve their purpose of stealth along with the specific functions and data structures they target in each technique with every successive genre of rootkits. This can prove helpful in devising a strategy for detection of the rootkits once we know what are the likely points of attack.

3. We have made a reference tool, **Stealth API Reference Tool,** based on my finding. This is an extendable tool and can be revised as and when new techniques of attack comes into picture. We have given a detailed description of the tool along with the utility.

Stealth API Reference Tool

The Stealth API Reference Tool is a tool that we have designed to summarize the findings of our research in an elaborative and graphical manner. The user of this tool can be the researchers working on the detection aspects of rootkits. This will enable them to find out in no time which API's to watch out for while studying any aspect of file, registry, process and port hiding. Anyone can easily extend or modify this tool, if they discover new functions or structures with successive releases of Windows.

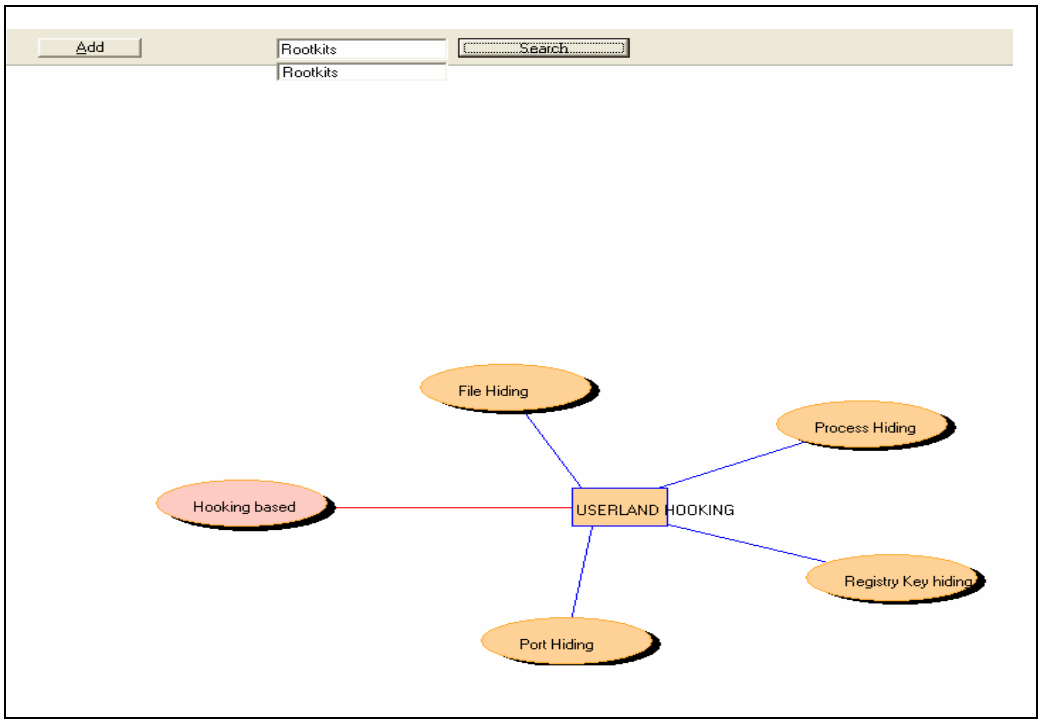


Figure 5.1 Reference tool linking hooking based rootkits to its sub-category of userland hooking.

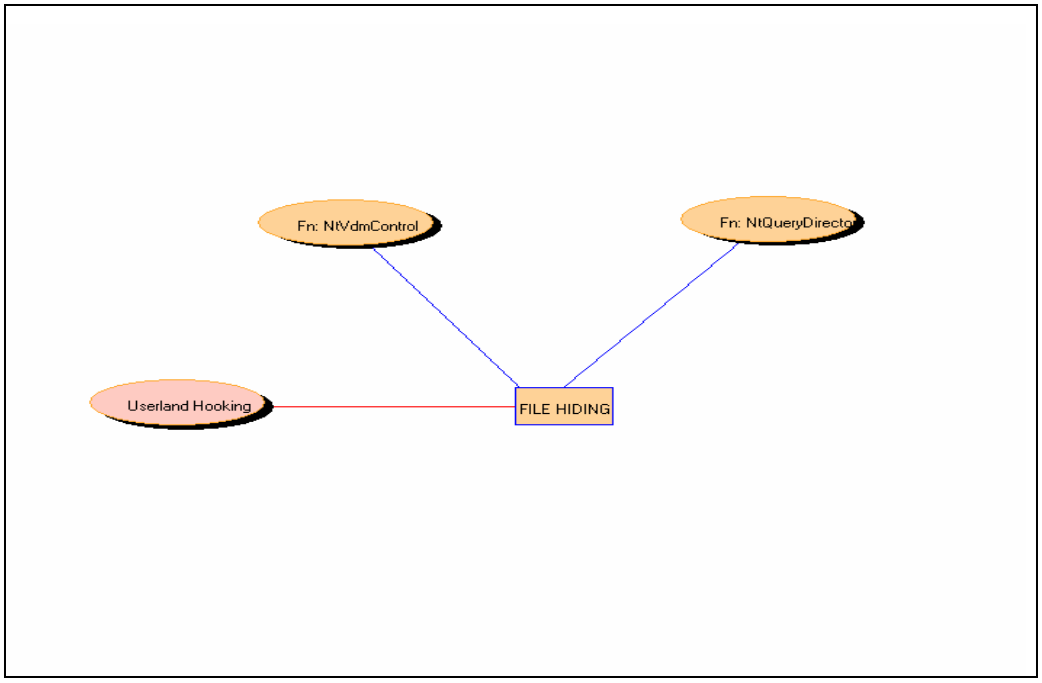


Figure 5.2: Tool showing targeted API's for File Hiding within Userland Hooking category.

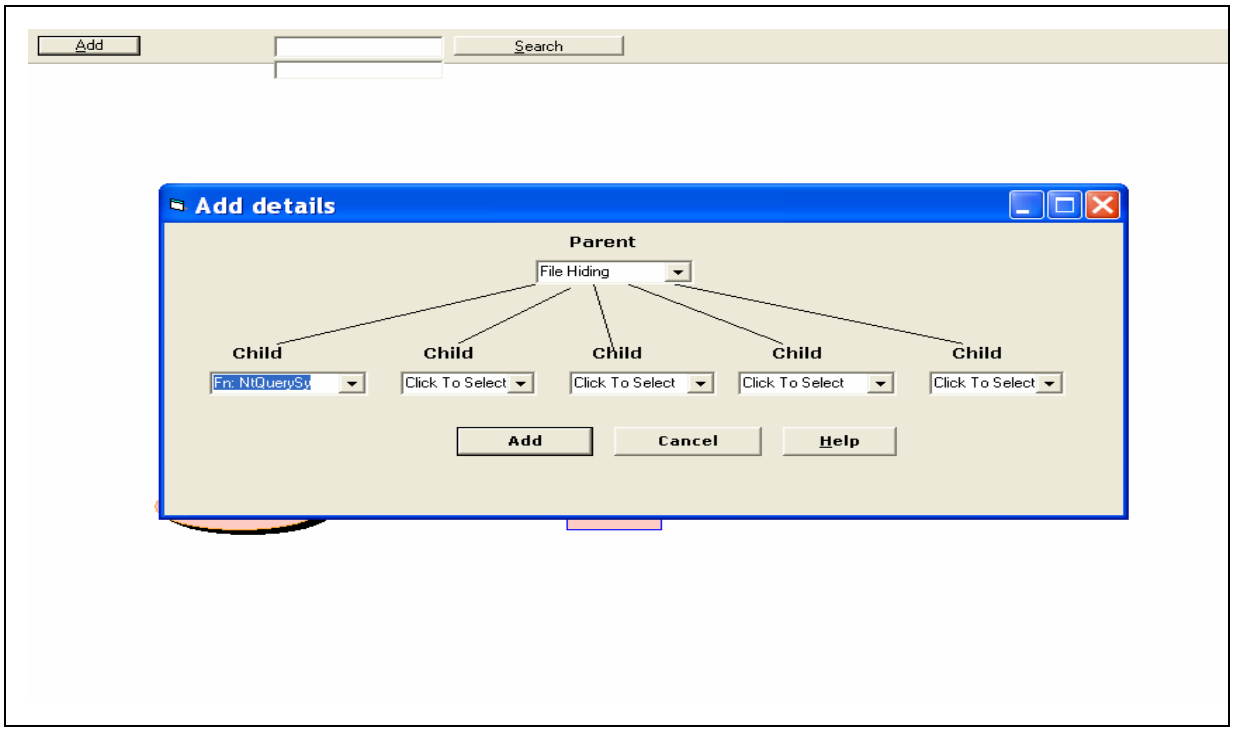


Figure 5.3: Illustration of the extendibility of the tool enabling the user to add/modify the tool based on his findings.

5.3 Future research

1. We have only studied Windows based rootkits. The study can be extended to linux based rootkits. Linux has its own mechanism of system calls analogous to API's. By using the debugger meant for linux, the rootkits can be studied.

2. Demonstration work through WinDbg can be enhanced to get further insight into the working of rootkit. This can become a very good education resource for interested students.

References

[1] Kimmo Kaslin, Mika Stahlberg, Samuli Larvala, Anti Tikkanen; "Hide 'n' Seek revisited-Full Stealth is Back", Virus Bulletin Conference October 2005.

[2] Greg Hoglund, James Butler, "Rootkits: Subverting the Windows Kernel", Publisher: Addison Wesley Professional, Pub date: July 22, 05.

[3] Ryan Naraine; "Where are rootkits coming from" http://www.eweek.com/article1/0,1852,20000012,00.asp?kc=EREDFFF423341K222212..

[4] Ryan Naraine; "Popular Rootkits", http://www.eweek.com/article2/0,1759,1944133,00.asp?kc=EWRSS03129TX1K000061.

[5] James Butler, Sherry Sparks; "Windows rootkits of 2005,part one", http://www.securityfocus.com/infocus/1851.

[6] Hacker Defender (Holy_Father 002), Phrack Volume 0x0b, Issue 0x3d Phile #0x08 of 0x14

[7] http://rootkit.host.sk/knowhow/hidingen.txt

[8] Three ways to inject your code into another process, http://www.codeguru.com/Cpp/W-P/system/processmodules/article.pho/c5767/

[9] NtQuerySystemInformation, http://msdn.microsoft.com/library/default.asp?url=/library/enus/sysinfo/base/ntquerysysteminformation.asp

[10] Netstat, http://www.sysinternals.com/files/netstatp.zip

[11] Microsoft Developers Network, http://msdn.microsoft.com/library/

[12] Detours win32 functions interception, http://research.microsoft.com/sn/detours/

[13] NTIllusion rootkit, http://www.syshell.org/?r=../phrack62/NTIllusion.rar

[14] Network Tool, http://foundstone.com/resources/freetools/fport.zip

[15] Chew Keong TAN, "Defeating Kernel Native API hookers by Direct Service Dispatch Table Restoration." Virus Bulletin Conference October 2005

[16] Matching debug information by the use of symbols, http://www.debuginfo.com.

[17] Kernel debugger help, Available at http://microsoft.public.windbg

[18] Windows Driver Development Kit, Available at http://www.microsoft.com/ddk.

[19] Microsoft Virtual PC, http://www.microsoft.com/windowsxp/virtualpc

[20] Debugging symbols, www.microsoft.com/whdc/devtools/debugging/debugstart.mspx.

[21] Rootkit-The online rootkit magazine, Available at http://www.rootkit.com.

[22] Gergely Erdelyi, "Hide' n' Seek? Anatomy of Stealth Malware", Updated version of earlier paper "Chasing Ghosts? – Return of the Stealth Malware", Virus Bulletin Conference 2003

Appendix A

Configuring Microsoft Virtual PC As Remote Host

1. Firstly Configuraton Editor from the Settings menu is opened. Then Serial Port is selected from the Add Hardware Wizard. Following this Named Pipe is selected and name of the Named pipe is changed to \\.\pipe\com_1.

2. Then “This end is the server” has to be selected which makes the virtual machine as a server and the host machine as the client. When we power up our VM image, the Devices->Serial0 menu will be available.

3. When the virtual machine boots up, boot.ini file is edited inside the VM OS's root directory.

4. Add a new OS line to tell the "virtual" Windows OS (when it boots) to enable it's kernel debugger. Two options need to be added to enable this feature:

   /debugport=com1
   /baudrate=11520

5. Edit the **Boot.ini** file and make it look like this.

   [operating systems]
   multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /fastdetect

   multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional - DEBUG"
   /fastdetect /debugport=com1 /baudrate=115200.

Appendix B

Building drivers with Windows DDK

The Windows DDK includes an environment for building drivers that closely matches the build environment used internally at Microsoft®[19]. This environment includes various tools useful in developing drivers, including build utilities (build.exe and nmake.exe), a C compiler (cl.exe), and a linker (link.exe). The Build utility automatically invokes nmake, the compiler, and the linker according to the command-line options we specify.

Step followed to make a driver

1. We named the macro with my target name, for example: TARGETNAME=myprogram
2. We edit the SOURCES macro so that it defines all the source files for the component that we are building.
3. We ran the Build utility from the command line by typing "build" to build the component

Each sample driver supplied with the Windows DDK has its own source directory and source files.

E:\NTDDK\src>build –cZ

This command created a "clean build" of the samples (by deleting all pre-existing .obj files). It also inhibits the dependency checking of source and header files.

Paper(s) Communicated/ Accepted/ Published

1. Gaurav Bajaj, Maninder Singh, "**Cyber Forensics: Taxonomy of Rootkits**", National Symposium On Cyber Forensics And Computer Crimes" **(NSCFCC 2006)**, Punjabi University**,** Patiala, 5 August 2006**. (Communicated).**