Towards Reliable Rootkit Detection in Live Response

Felix C. Freiling
Laboratory for Dependable Distributed Systems
University of Mannheim, Germany

Bastian Schwittay
Symantec (Deutschland) GmbH, Germany

Abstract: Within digital forensics investigations, the term *Live Response* refers to all activities that collect evidence on live systems. Though Live Response in general alters the state of the suspect system, it is becoming increasingly popular because it can recover valuable information that is lost in normal investigations that power down a suspect computer and perform analysis on its hard disk image. Current best practices for Live Response however fail to take into account the possibility of false information being gathered due to the presence of rootkits on the system. In this paper we propose to establish rootkit detection as a standard part of Live Response. We argue that the credibility of the recovered information can be substantially increased by regular empirical experiments using known rootkits and rootkit detectors. We present the results of such an experiment in this paper showing that a redundant combination of three tools can discover all rootkits which were publicly available as of June 2006.

1 Introduction

1.1 Motivating Live Response

After detecting a computer misuse, the classical approach during a digital investigation is to pull the plug of all computer systems involved and analyse an image of the hard drives in the respective systems. This kind of analysis is commonly called "dead" analysis. In recent years, this principle has been reconsidered, and methods of data collection on systems that are still running have become popular. In contrast to dead analysis, this is called "live" analysis, and all activities concerning data collection on live systems are summarized under the term *Live Response*.

The reason why Live Response has become an important part of digital forensic investigations is that when powering down a computer system, a lot of volatile information is lost, because the RAM is cleared. This information will not be available for further analysis, although it may contain valuable clues regarding the incident, and there is sometimes no other way to obtain it other than collecting it on the running system. The volatile information that can be collected during Live Response includes the system date and time, a list of current network connections and open TCP/UDP ports, a list of running processes, the

names of the loaded kernel modules, and a list of open files. Especially the information regarding network connections and ports, and running processes can be crucial information that allows to figure out what caused an incident or how a system was compromised.

In addition to the collection of volatile data, it has also become common to collect certain non-volatile data too, i.e. data that could also be recovered in dead analysis. Sometimes this is done out of convenience, e.g. because certain logfiles can easily be recovered from a running system with special commands, but in a dead analysis the investigator would have to parse the logfile format with large effort. In some cases file formats are even proprietary so no publicly available parsers do exist, yet it is very easy to recover the files on a running system with the right tools.

Another reason for collecting non-volatile data is that this information will already be available in a very early stage of an analysis and can improve containment of the incident. With the amount of data that can already be recovered during Live Response, performing a forensic duplication can become redundant, because enough information to explain and resolve the incident is already there. On the other hand, if Live Response yields unexplainable results or raises further suspicions, the decision to perform a forensic duplication and thorough dead analysis is strengthened.

1.2 Live Response Considerations

A severe problem with Live Response techniques is that they will generally alter the running system's state, which contradicts the general paradigm of computer forensics to never modify potential evidence. Issuing commands on an evidence system will change that system's RAM contents, create processes, and even the hard drive may be modified, e.g. by unnoticed creation of cache files by the operating system. However, if the changes that the use of a Live Response tools will make to the target system are well understood and documented, its use can be admitted even if it alters evidence slightly. This tradeoff is justified by the fact, that the information gained by conducting Live Response is often so valuable, that a small and controlled modification of the evidence is negligible.

It is well-known, that information offered by a compromised computer system cannot be trusted. Especially rootkit software can alter status information in arbitrary ways. Before performing Live Response, it is therefore important to aquire confidence that no rootkits are installed on the computer system in question. However, standard operational procedures for Live Response (see for example Jones et al. [14]) fail to acknowledge this fact.

One could argue that although a rootkit can hide certain information on a running system, a dead analysis will always be able to recover all information that the rootkit hid, and it will also reveal the presence of the rootkit. First of all, any volatile data that is manipulated is not reconstructable during a dead analysis, so the otherwise valuable information about running processes, open ports etc. is lost, unless special techniques to correct the rootkit's manipulations are used. Secondly, it is much more convenient to collect evidence about rootkits on a live system than during dead analysis since the latter often necessitates searching thousands of files for malicious traces. Thirdly, rootkits exist that leave no trace

on the file system at all, running completely in main memory; these rootkits will be undetectable on a file system image, and if their presence is not noticed during Live Response, there is a severe danger that an investigation will be using false data collected during Live Response without ever knowing.

Therefore, if a rootkit is detected, *any* information collected during Live Response must be checked against other evidence, e.g., evidence collected during dead analysis. The existence of a rootkit can additionally be the cause for further actions. For example, rootkits can initiate electronic booby traps for logical bombs, in case of which it may be advisable to freeze the system immediately to prevent any loss of valuable evidence.

1.3 Contributions

In this paper, we aim to improve the *credibility* of Live Response procedures for the Win32 family of operating systems. We contribute to the methodology of computer forensics by exhibiting a method to detect rootkits on a compromised system with high confidence. The method is based on the concept of *diversity*, i.e. redundantly using different rootkit detectors on a set of publicly available rootkits. We performed experiments with 11 publicly available rootkits and 12 available rootkit detection tools as of June 2006. By experimenting with the rootkits and detection tools, it is shown that using a combination of 3 different detection tools, 100% of all tested rootkit installations can be detected. Due to the diversity in detection techniques it can be expected that a similarly high detection rate can be achieved for unknown rootkits, allthough it is hard to precisely quantify this of course. We therefore argue that rootkit detection should be a standard part of Live Response and should follow an empirically evaluated strategy like the one presented in this paper to raise confidence in the detected evidence.

1.4 Paper Outline

This paper is structured as follows: Sect. 2 gives some background on Windows rootkits and techniques for their detection. The experimental setup is explained in Sect. 3 while the results of the experiments are reported on in Sect. 4. We conclude in Sect. 5. For lack of space we omit a discussion of standard practices for Live Response and refer the reader to Schwittay [17] for more information.

2 Windows Rootkits

We now take a detailed look at rootkits for Windows operating systems and their basic working principles and techniques. There are no novel insights in this section, but knowledge of rootkit internals is necessary to understand the different ways to detect rootkits despite their stealth capabilities. These differences are vital to the diversity of detection

techniques used in our methodology later in this paper.

A rootkit can be defined as "a set of programs and code that allows a permanent and undetectable presence on a computer" [13]. Typically rootkits are used to hide certain files, processes and other information on a running system, making them one of the most dangerous classes of malware, because they can exist on a compromised system unnoticed for years. Additional functionality like a backdoor or keyboard sniffers can often be added, turning rootkits into the ultimate attacker's tool. By examining current rootkits and detection tools and putting them to the test, in Sect. 4 it will be shown how rootkits can nevertheless be detected, resulting in a methodology to safely detect publicly available rootkits.

2.1 Background: Windows Kernel and Rootkits

To understand how rootkits work it is first of all important to understand the basics of the Microsoft Windows kernel architecture and the way the operating system works beyond the normal graphical user interface. The techniques used by rootkits to subvert the operating system are fundamentally different from the way normal software operates. Often they manipulate internal operating system tables that are used to control the execution flow of other software on the system, causing the rootkit code to be executed instead of the original code. In other cases rootkits will filter or forge the output of system calls to hide themselves or other objects and yet another class of rootkits modifies memory contents directly to fool the operating system.

Because of the complexity of the underlying technology, it will hardly be possible to explain all details of the operating system mechanism completely. Therefore this introduction will only focus on a few central concepts that allow presentation of the most common rootkit techniques, namely *Hooking* and *Direct Kernel Object Manipulation*. For detailed description of Windows operating system internals, refer to Solomon and Russinovich [18] or Hoglund and Butler [13]; the latter can also be considered to be a reference work on rootkits.

2.1.1 Windows Internals

The first important concept to understand is how *access control* is realized on the x86 processor family, which is the processor type used in most personal computers today. In this context, access control means how the hardware controls which kind of instructions may be used by a particular process, which areas of memory or files may be modified, and how hardware components may be accessed and written to. Without access control at the hardware level, any process would theoretically be allowed to access and modify every file, alter the memory contents belonging to another process and issue every CPU instruction, no matter whether that could disturb other processes or even crash the whole system.

The x86 processor type has provided the capability for access control for a long time, in the

form of four privilege levels called *rings*, where *Ring 0* is the most privileged and *Ring 3* is the least privileged (see Fig. 1). Since Windows NT, Microsoft operating systems use the two rings Ring 0 and Ring 3 for access control. All user-mode programs, i.e. most normal applications, run in Ring 3 and have only limited access to memory, and cannot access hardware directly; this will be prevented by the CPU itself. Ring 0 contains all software that runs with full system priviliges, for example the Hardware Abstraction Layer (HAL), device drivers, IO and memory management – essentially all components of the operating system kernel. Code running in Ring 0 can access the whole memory space and is able to execute privileged instructions, e.g. allowing access to hardware components like graphic cards or the keyboard. Programs running in Ring 3 are often called *userland programs* and those running at Ring 0 are called *kernel programs*.

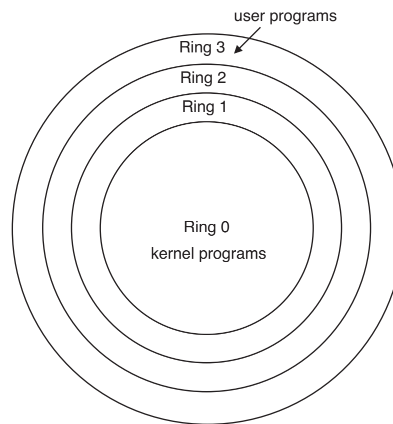

Figure 1: Ring architecture of the Intel x86 processor family.

Since even userland applications may have to access hardware from time to time – e.g. when using a system administration programm to install new device drivers for a graphics card – special mechanism exist that allow a userland program to cross the barrier to Ring 0 and use the otherwise unavailable functions in a controlled way. Some rootkits exploit this capability to plant their own device driver (a `*.sys` file) containing their malicious code in Ring 0, allowing them to run at full privilege level; these rootkits are also called *kernel rootkits*. It is important to notice that any Ring 3 detection tool will have a large disadvantage against a kernel rootkit, because the two are basically not competing with each other on equal terms. Because a Ring 0 rootkit can control the environment in which other software runs, it can achieve stealth and avoid detection in a very effective way.

The System Service Dispatch Table. Whenever a userland program wants to use a function that is only available to Ring 0 programs, it issues a so-called *system call*. A system call interrupts execution of the user program and transfers execution control to the kernel, which will then process the requested service as indicated by the system call number stored in the CPU register `EAX`, possibly using a set of input parameters received from the userland application. After the system call has been processed, execution of the userland

program is resumed, which can now use the system call's results for further operation.

On Windows, there is a large number of these service functions provided by the kernel, and each function is identified by a unique system call number. Whenever a system call is made, a special kernel routine called `KiSystemService` is executed in Ring 0, which reads the system call number from the `EAX` register and looks up the matching function in a table, the *System Service Dispatch Table (SSDT)*. The SSDT contains the memory addresses of all functions that correspond system calls, making it one of the central points of execution flow control in the kernel. Thus the SSDT naturally represents an opportunity for malicious modifications by a kernel rootkit; the technique for SSDT manipulation will be described in more detail later.

2.1.2 Hooking Techniques

Altering the execution flow of programs is one of the most prominent techniques of rootkits to subvert the Windows operating system. By having code that belongs to the rootkit execute instead of the original, a rootkit can manipulate the operating system and the programs that are run on it to achieve stealth, i.e. operate completely unnoticed and hidden from the user. The various techniques of rerouting the execution of programs are commonly referred to as *Hooking*. It should be noted that often legitimate software uses Hooking techniques too, so the presence of a hook on a system does not automatically mean that a rootkit is installed.

IAT Hooking. The easiest form of hooks manipulate the way in which individual applications are executed, in particular how they import library functions, that are typically provided on Windows operating systems in the form of DLL files. Whenever an application is run, a list of required library files included in the application executable file is used to load the matching library files into the application's memory space completely. Apart from the list of required DLL files, the executable also contains a structure that stores the specific functions inside the respective library files that the application needs. These functions' locations in memory are determined and a special table called *Import Address Table (IAT)* is loaded with the library functions' names and the corresponding address of the function in the application's memory space.

A rootkit that has access to an application's address space can manipulate the IAT in order to make its own code execute instead of the original library function. It will do this by overwriting the address of the IAT functions with the address of the rootkit function's location in memory space, allowing the rootkit to manipulate the parameters or output of the original function. There are a number of userland rootkits that will use IAT hooking to manipulate the IAT of every application running on a system, which enables them to hide their presence globally across the whole operating system. Fig. 2 shows how IAT Hooking alters the execution path of a program.

Inline Function Hooking. A slightly more elegant hooking technique does not overwrite the IAT address of the library function to be hooked, but instead modifies the func-

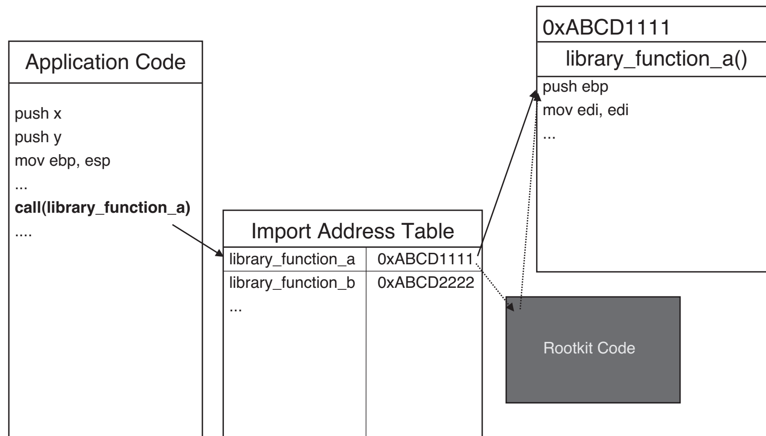

Figure 2: IAT Hooking – The solid line represents the normal execution path, the dotted line shows the hooked path.

tion's code directly in memory. Such an *inline function hook* will usually patch the first few instructions of the hooked function with a special jump instruction to the rootkit code, which will then be executed before the original code. Usually the rootkit code will then call the original function, because when execution of the original function has completed, control will return to the rootkit code, allowing to modify the results of the library function. Inline hooks have several advantages over standard IAT hooks, mainly because the technique avoids problems when the function is not called using the IAT, because no matter how a process calls the respective function, the inline hook is always effective.

SSDT Hooking. Whereas IAT and Inline Function Hooking all take place in the userland realm, there are also techniques to hook execution in the kernel. As previously explained, one of the central tables that control execution of kernel functions is the System Service Dispatch Table. Whenever a system call is performed, the kernel function `KiSystemService` takes control over the execution and looks up the memory address of the kernel function corresponding to the system call number in the SSDT.

A rootkit can subvert this mechanism by exchanging the original function's address with the address of the rootkit's hook function. The hook function can then imitate the original function but choose to filter out certain information, e.g. a function that reports a list of open ports could be replaced by a similar function that does the same but does not display a certain range of ports, which is used by a built-in backdoor of the rootkit. Overwriting the addresses in the SSDT is generally a lot harder than IAT Hooking, but the details have intentionally been left out in this context to be able to just explain the rough concepts. A detailed description of the method can be found in Hoglund and Butler [13]. Fig. 3 is a simplified representation of the technique.

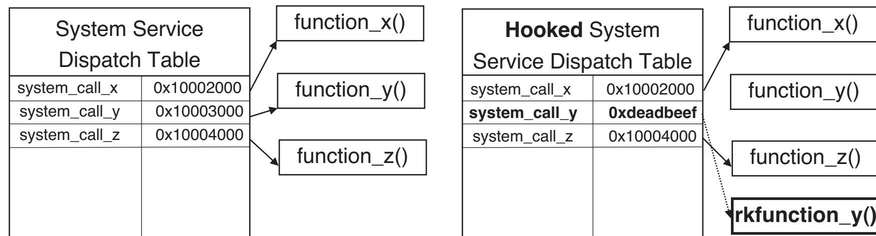

Figure 3: SSDT Hooking – left: before hooking, right: after hooking.

2.1.3 Direct Kernel Object Manipulation

Hooking techniques are a solid technique of subverting an operating system, but their main drawback is, that a hook can usually be detected and there are several tools for hook detection, some of which will also be introduced in later sections. An advanced technique for subversion, that e.g. allows to hide processes, directly manipulates the very data structures in kernel memory that keep track of the state of the operating system, therefore it is called *Direct Kernel Object Manipulation (DKOM)*.

Windows keeps a number of undocumented data structures in kernel memory which for example contain a list of running processes, of threads scheduled for execution etc. Part of DKOM techniques is therefore trying to understand the structure of these kernel objects to be able to manipulate them without making the operating system unstable or even crashing it completely. The technique then consists of careful modifications to the in-memory data structures to hide certain objects from the user. For example, processes are recorded in a doubly-linked list, so that if the offset of the forward and backward pointers to the next or previous process in the list is known, it is possible to exclude a process from the list with simple pointer reorganization. Because the scheduling of execution of processes does not depend on a process being present in that list, this technique hides the process successfully (e.g. from the Task Manager), but the process is still executed unnoticed (see figure 4 for a simplified model of the process hiding technique).

DKOM has certain benefits over Hooking, especially because it is very hard to detect, since directly modifying the raw main memory contents with a Ring 0 rootkit cannot be controlled by any built-in security mechanism in Windows. The disadvantage of DKOM is its extreme instability, since there is no documentation about the actual structure and usage of the kernel objects by the operating system, and often even minor operating system upgrades change the way they look like and how the operating system uses the kernel objects. Nevertheless DKOM has become a concept that has to be considered one of the most sophisticated and dangerous rootkit technologies that exists.

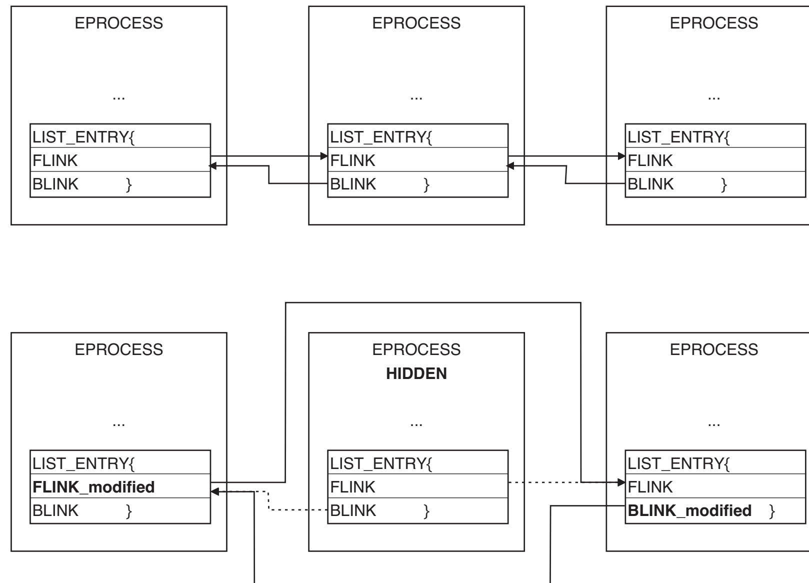

Figure 4: Process Hiding with DKOM [13]– The upper figure shows a part of the original linked list of processes, the lower figure shows how the middle process is hidden by pointer manipulation.

2.2 Overview of Rootkits and Detectors

2.2.1 Rootkits used

For the experiments on rootkit detection, a number of publicly available rootkits were used, all of which can be obtained via the rootkit.com website [8]. This includes userland rootkits as well as kernel rootkits, and rootkits which use Hooking techniques as well as some that use DKOM to hide their presence. The userland rootkits included in the experiments were AFX Rookit, Hacker Defender (also called HxDef), He4Hook, NtIllusion and Vanquish. All of these rootkits offer about the same capabilities of hiding processes and files, sometimes drivers or services; the details will be described in the next section.

Kernel rootkits included in the experiments were FU, FUTo, phide and HideProcessHook-MDL (HPHM), where the latter is more of a proof-of-concept device driver that allows to hide processes by using SSDT Hooking. It was chosen nevertheless because no other rootkit could be included that uses this technique. FU, FUTo and phide all use DKOM techniques to hide processes, FU and FUTo even offer functions to elevate a process's privileges and also alter other kernel objects than the linked list of processes.

There were also some related pieces of software, namely cfsd and Klog, which were included in the trials although they are not rootkits, because they use very similar techniques. Cfsd is a file system driver, which can be used to misrepresent the contents of a file system, for example allowing to hide certain files. Klog is a device driver that acts as a keyboard sniffer, i.e. it logs everything typed into the keyboard into a file without being noticed.

2.2.2 Detection tools used

After selecting the rootkits for the experiments an assortment of rootkit detection tools was assembled to be able to test the various technologies used by rootkits against the different kind of detection tools. Among them were two tools that explicitly look for hooks, VICE [8] and Respendence Software Rootkit Hook Analyzer (RKHA) [10]. VICE detects both userland and kernel hooks, and can discover normal IAT hooks, inline function hooks, SSDT hooks and also more advanced hooks, while Rootkit Hook Analyzer only detects kernel hooks.

Another class of detection tools directly looks for hidden objects on the running operating system, the ones used in this paper are F-Secure Blacklight [1], RKDetector [9], Rootkit Revealer [11], and UnhackMe [12]. They are generally run once and the output is a list of all hidden objects found. Blacklight can detect hidden processes and files, RKDetector and UnhackMe detect hidden processes and will sometimes also alert because of suspicious services and corresponding registry keys. Rootkit Revealer can detect hidden files and registry keys, including those belonging to a system service, making it possible to also detect hidden services indirectly.

It is known that Blacklight and Rootkit Revealer use a *cross-view detection* approach, which identifies hidden objects by comparing the output of high-level API reporting functions with results gained from parsing the appropriate structures on a very low level. For example, to find hidden files, a cross-view detection tool would first use the Windows API to request information about the contents of the file system. It would then use low-level methods of parsing the file system itself, and compare the output with the API output, uncovering any files that were hidden at API level. A cross-view approach only works if the subversion of the rootkit does not manipulate the operating system at the same or even lower level than the detection tool uses for its baseline data collection. Apart from this limitation, cross-view detection is one of the most effective ways to find rootkits, because it is "generic" in the sense that is does not scan for specific rootkit signatures or for signs of a data hiding technique like hooking.

The tools IceSword [5] and DarkSpy [2] are both freeware tools that offer the possibility to look for rootkits in an interactive fashion. For example both offer a function to view a list of currently running processes, like the Windows Task Manager, or an Explorer-like application to browse the file system or the Windows registry. The difference to the ordinary reporting and browsing tools is that they use very advanced techniques to gather the data about the respective objects, often exploiting the way in which rootkits typically hide processes or files. Although IceSword and DarkSpy may sometimes highlight hidden processes using similar techniques as the cross-view detection tools, generally they have to be used in an interactive fashion by an investigator who systematically looks for signs of a rootkit.

To complete the set of detection tools there are also a number of command line tools that cannot be classified like the tools above, these are modgreper, flister and System Virginity Verifier [6]. Flister is a tool that can be used to detect hidden files, exploiting some common bugs in rootkits when using file hiding techniques. Modgreper scans kernel memory for hidden modules, which can sometimes reveal the presence of a rootkit that uses a hid-

den module in the kernel to do its work. The System Virginity Verifier (SVV) written by Rutkowska is a unique tool, which is related to the concept of *Explicit Compromise Detection* [16], which means that the integrity of critical operating system elements is checked to detect a possible compromise. SVV compares the code sections of kernel modules in memory, i.e. drivers and DLLs, with their representation on the file system, where the files are stored. If discrepancies are detected between the stored file and its image in memory, SVV evaluates the type of the change and outputs an infection level that denotes the severity of the modification, and whether it is likely to be a malicious modification.

3 Experimental Setup

3.1 Test Platforms and Compatibility Issues

To get an idea of how rootkits work on different version of the Windows operating system, all experiments were executed on four different operating systems, namely on Windows 2000 Service Pack 4 with all official update as of June 1, 2006, Windows XP without any updates, Windows XP Service Pack 2 with all updates and Windows 2003 Server Service Pack 1 with all updates. All systems were standard installations, with a minimum of required device drivers installed to be able to run properly. After the setup, all systems were kept offline and all software was copied from a CD to the test platform, to avoid accidental compromise by a real attack.

There were significant compatibility problems with the rootkits, in particular, the only rootkits that could be installed on Windows 2003 were Hacker Defender, Klog and Vanquish. Other notable problems occured with cfsd and He4Hook, none of which could be successfully installed on any of the test platforms; it is not clear whether this was a known issue or caused by incorrect use of the supplied files, because these rootkits come with no documentation at all. On top of that, NtIllusion did not work properly on any operating system but Windows XP Service Pack 2, HideProcessHookMdl did not work on Windows 2000 and the AFX Rootkit could not be executed on Windows XP Service Pack 2.

There were also minor compatibility issues with some of the detection tools, although they appeared to be far more robust than the rootkits themselves. In particular, VICE does not execute properly on Windows XP Service Pack 2; the problem seems to be a patch to the .NET framework, which is used by VICE. In addition to these incompatibilities, SVV does not work properly on Windows XP Service Pack 2 with all patches installed as of June 1, 2006. Although it does execute normally and outputs a result, it also outputs an error message and it can be clearly seen from the resuts that it did not operate at full effectiveness. The last incompatibility was with DarkSpy and Windows 2000 SP4: when starting DarkSpy, the operating system would simply crash immediately with a Blue Screen, making it impossible to test DarkSpy on this platform.

3.2 Configuration of the rootkits

To start the experiments the rootkits were, if necessary, configured using the included documentation to use all the capabilities for stealth that they provide. Specifically, this means:

- If the rootkit offers a file hiding capability, a sample folder and files that should be hidden were created.
- If the rootkit offers a process hiding capability, an appropriate process that should be hidden was started.
- If the rootkit offers hiding of network ports, an open port was created using a listening `netcat` session.
- If the rootkit offers a registry key hiding feature, sample registry keys and values to hide were created.
- If the rootkit allows hiding of services and drivers, sample objects of the respective type to be hidden were configured.

Some rootkits hide files or processes by using a *magic string*, i.e. if a file or process name contains a specific string, it will be hidden; AFX, NtIllusion, Vanquish, and HideProcessHookMdl use this technique. Hacker Defender has a real configuration file which allows definition of different magic strings for different classes of objects to be hidden, and it allows to define network port ranges to hide. FU, FUTo and phide can hide processes by referring the process's identification number, or PID. The keyboard sniffer Klog did not have to be configured. Tab. 1 gives an overview over the hiding capabilities of the different rootkits.

	Files	Processes	Registry Keys	Ports	Services	Drivers
AFX Rootkit	X	X	X	X	X	–
FU	–	X	–	–	–	X
FUTo	–	X	–	–	–	X
Hacker Defender	X	X	X	X	X	X
Klog	–	–	–	–	–	–
NtIllusion	X	X	X	X	–	–
Vanquish	X	X	X	–	X	–
phide	–	X	–	–	–	–
HideProcessHookMdl	–	X	–	–	–	–

Table 1: Hiding capabilities of rootkits: an X denotes that the rootkit can hide objects of the respective type

3.3 Rootkit installation

After configuration the rootkits were installed using the included loader programs, which is mostly a simple EXE file that will load the rootkit into memory and start it, which results in immediate hiding of the desired objects. In this case, *hidden* means that files are no longer visible in the Windows Explorer, processes are invisible in the Task Manager, registry keys are invisible in the Registry Editor, and open network ports are not detectable with `netstat`. Drivers and services were checked using the System Information tool included in Windows.

Generally, the rootkits succeeded in hiding the respective objects they were designed to hide very well, all rootkits that have the capability the hide processes, files, registry keys or network ports worked exactly as expected, hiding the target elements from the applications mentioned above. The only exception is AFX Rootkit, which did not hide network ports as it was supposed to do. Vanquish and Hacker Defender use a special service to operate, and they both succeeded in hiding "their own" service. Driver hiding, which is offered by FU, FUTo and Hacker Defender, did not work at all; all loaded rootkit drivers were still visible with the System Information program.

3.4 Application of Detection Tools

After having installed a single rootkit and having documented its stealth properties, the rootkit detection tools were run using a small batch script to see if they would detect the hidden objects and the rootkits themselves respectively. The output was then recorded and it was checked whether all hidden objects that the detection tool can detect have actually been detected; of course it makes no sense to blame a tool for not detecting a hidden process, if it is not designed to do so. Having documented the results for the detection tool, it was then uninstalled if necessary, and any drivers it used were unloaded. After a system reboot, the state prior to testing the previous detection tool was reestablished, if necessary by restarting the rootkit and reconfiguring the sample processes and network ports. Then the next detection tool was tested in the exact same way as described above.

4 Experimental Results

4.1 Differences between operating system versions

As a first result, there were absolutely no significant differences in the detection performance between the different operating system versions used during the experiments. In particular, there was no instance, in which a detection tool detected any rootkit modification successfully on one operating system, but failed on another one. The only differences were the compatibility issues of some rootkits and detection tools that were also described in detail in the Setup section. For this reason the results will be mainly be evaluated for

the Windows XP SP2 system, which was compatible to most rootkits and detection tools and can thus be regarded as the reference operating system concerning the results of the rootkit detection experiments. For the AFX Rootkit and VICE, which were incompatible with Windows XP SP 2, the results on Windows XP without any Service Packs were used.

4.2 Detailed Results

It turned out that rootkit detection was not a "black-and-white" issue, but that in some cases detection tools would only partially detect the rootkits modifications. In this context, complete detection would mean, that the detector detected all changes that it is designed to detect. Tab. 2 gives an overview over detection tools and the kind of objects they were designed to detect.

	Files	Processes	R'stry Keys	Ports	Services	Drivers	Hooks
Darkspy	X	X	–	X	–	X	–
Flister	X	–	–	–	–	–	–
F-Sec. Blacklight	X	X	–	–	–	–	–
IceSword	X	X	X	X	X	X	X
modgreper	–	–	–	X	X	–	–
RKDetector	–	X	–	–	X	X	X
RKHA	–	–	–	–	–	–	X
RK Revealer	X	–	X	–	–	–	–
SVV	–	–	–	–	–	X	X
UnhackMe	–	X	X	–	X	–	–
VICE	–	–	–	–	–	–	X

Table 2: Rootkit Detectors: an X denotes that the detector can detect hidden objects of the respective type

A scale with three values was used to rate the amount of success that was achieved. A success variable s was used to account for the effectiveness of the detection tool as follows:

- $s = 0$: The detection tool did not detect anything.
- $s = 1$: The detection tool detected some rootkit modification or hidden objects, but not everything.
- $s = 2$: The detection tool detected all possible rootkit modifications and hidden objects.

The results can be seen in Tab. 3. A "–" means that the tool in question was not applicable, because it was not designed to detect the objects the particular rootkit hid. It is obvious, that the Klog keyboard sniffer does not really fit into this selection, because it does not

actually hide anything. This issue will be discussed after the analysis of the experimental results.

	Rootkits								
Detectors	AFX	FU	FUTo	HxDef	Klog	NtIll.	Vanq.	phide	HPHM
Darkspy	2	2	2	2	–	2	2	2	2
Flister	2	–	–	0	–	2	2	–	–
Blacklight	2	2	0	2	–	2	2	2	2
IceSword	2	2	2	2	–	2	2	2	2
modgreper	–	–	–	1	–	–	–	–	–
RKDetector	2	2	0	0	–	2	0	2	2
RKHA	–	–	–	–	–	–	–	–	2
RKRevealer	2	–	–	2	–	0	2	–	–
SVV	2	–	–	2	–	1	2	–	1
UnhackMe	2	0	0	2	–	1	2	0	2
VICE	2	–	–	2	–	–	2	–	2

Table 3: Experimental Results for s

4.3 Tests with Live Response tools

The results of the Live Response tools `netstat`, `fport` [3], `pslist` [7], `psservice` [7], `find`, and `regdmp`, included in the experiments can be summarized quickly: *none* of them reported any of the objects that were hidden by the rootkits. The only way that these tools could be used to indicated a potential rootkit installation, was when a hidden process had a non-hidden port open; in this case, the tools would detect the open port, but not the process, a discrepancy that could alert an investigator if he carefully analyzed the output of the tools.

4.4 Discussion

The experimental results show that there were several excellent tools that were able to detect all rootkit modifications, but there were also some that did not perform well. The two interactive tools DarkSpy and IceSword performed excellent, both reaching a mean score of 2, the maximum possible. F-Secure's cross-view detection tool Blacklight also detected all modifications except processes hidden by FUTo, which is an expected result, because FUTo is a modified version of the FU rootkit, specifically designed to be able to evade detection by Blacklight [4] .VICE also perfectly detected the hooks installed by the hooking rootkits, although it has previously been mentioned that identifying malicious hooks can sometimes be diffcult, because a lot of software, especially Antivirus software,

uses benign hooking.

Other tools that performed quite well were SVV with an average score of 1.6 and Rootkit Revealer (1.5), along with the file lister flister (average score of 1.5). The tools Rootkit Detector and UnhackMe did not perform satisfactorily, with scores of 1.25 and 1.125 respectively; in addition to the lower average results, these tools did not detect a number of rootkits at all, making their use questionable and the results not very reliable. The very specialized tools modgreper and Rootkit Hook Analyzer (RKHA) could not be evaluated properly based on the experiments that were conducted, because they were each only applicable in one case, which is certainly not representative of the tools' reliability. During the experiments it also became obvious that the keyboard sniffer Klog would not be detected, simply because it does not attempt to hide anything; this result was therefore expected.

The fact that the Live Response tools tested failed completely in trying to report the hidden objects does not surprise either: rootkits are specifically designed to subvert these tools' mechanisms, which generally means that they subvert at least the high level-API reporting functions used by the Live Response tools. This results shows that rootkit detection has indeed become very important in forensic analysis because the standard tools used in Live Response procedures do not work properly. Adressing this need for rootkit detection, a methodology for rootkit detection has been developed, which will be described in the next section.

5 Rootkit Detection Methodology

Based on the results of the experiments, a methodology for rootkit detection has been developed, which consists of using a combination of different detection tools to add redundancy to the detection procedures. The proposed methodology uses three tools: F-Secure Blacklight, IceSword and SVV. This combination was chosen for several reasons; first of all, all three tools achieved good average results in the experiments, reliably detecting rootkit modifications.

Secondly, these three tools represent three different approaches to rootkit detection. Blacklight uses a cross-view based detection approach, it offers a simple user interface, which allows to scan for hidden objects with a single mouse click. After checking the system, Blacklight outputs a list of hidden objects found. Usage of IceSword differs in the way that it offers information about certain operating system elements in an interactive way; it offers a function that resembles the Task Manager to review running processes, a browser for the Windows registry and the file system, which resembles the Windows Explorer, and it also includes function to view loaded device drivers, services, the SSDT and even more. IceSword does not offer a convenient scanning option such as Blacklight, but if an investigator knows what to look out for, it is a very flexible and powerful tool. SVV was chosen to be included in the methodology because it represents yet another technique to detect rootkits, by checking important operating system elements for their integrity. This technique requires the most technical knowledge about the concepts and techniques used by rootkits, but for an experienced investigator it is one of the most powerful and advanced

tools to analyze a rootkit infection.

This comparison of the three tools has also shown that the methodology is applicable for forensic examiners with different levels of understanding; Blacklight does not require any special background to be used, IceSword becomes more effective in the hand of an experienced investigator, and SVV offers detailed information which will be most helpful to a very skilled analyst. By correlating the output of all three tools, the reliability of rootkit detection is increased, because even if a rootkit manages to evade one of the tools, it can still be detected by another one that uses different methods for detection. None of the rootkits tested can evade the combination of Blacklight, IceSword and SVV, and only an extremely sophisticated rootkit could hope to avoid detection by all of the three tools together. Although it is difficult to quantify the success rate of detection when using this methodology for an arbitrary rootkit infection, it is reasonable to assume that when facing a rootkit that uses techniques similar to the ones tested in this paper, the rate of detection is close to 100%. Asssuming that the majority of rootkits used today fall into this category – which appears reasonable – the likelihood to detect an actual rootkit detection is equally high.

The results of these experiments were based on software which was available as of June 2006. Experiments like the ones performed in this paper should be repeated in regular intervals so that tools can be combined such that the detection rate can be kept high in practice.

6 Conclusions and Future Work

All experiments were planned and the test platform carefully prepared to allow objective and accurate testing of the tools, and to gain a representative view of their performance on the Windows operating system platform, multiple different operating systems were included in the tests. A selection of detection tools that use different methods for detection was chosen, among which were cross-view based detection tools and tools that use Explicit Compromise Detection (ECD) to detect rootkits.

It could be shown that there exist a number of very good rootkit detection tools, but also that some are not reliable enough to be used in a forensic investigation. By choosing a combination of three detection tools which achieved good results in the experiments and which each represent a different approach to rootkit detection, redundancy has been added to the process of rootkit detection, resulting in a reliable and accurate methodology. Using F-Secure Blacklight, IceSword and System Virginity Verifier, all publicly available rootkits can be detected safely, and depending on the skill of the investigator, this toolkit, especially IceSword and Blacklight, can also be used to analyze the rootkits in more detail.

There is one thing to keep in mind when using IceSword and several other rootkit detection tools: a special device driver, for IceSword it is called `IsPubDrv.sys`, is often copied to the systems hard drive to allow the detector to function properly. This of course contradicts the principle of not creating files on an evidence system, because this is a severe modification of the evidence. The principle problem here is, that in order to be able to detect a

rootkit's subversion, the detection software has to be able to deal with the problem on the same level, i.e. using a Ring 0 kernel component. Device drivers are the standard way to gain access to Ring 0, and that is why the creation of a single file can be tolerated in this context; otherwise the rootkit would have an implicit advantage, because it can manipulate the operating system at a lower level than the detection tool can reach.

As an avenue of future research, we wish to repeat the experiments performed in this paper in regular intervals. For example, in the time which elapsed since our study new rootkits based on virtualization have been widely discussed (see for example Rutkowska [15]) and these should be included as soon as possible.

Acknowledgments

We wish to thank Maximillian Dornseif for helpful discussions.

References

[1] F-Secure – Blacklight. `https://europe.f-secure.com/blacklight/`.

[2] DarkSpy 1.02. `http://d.hatena.ne.jp/tessy/20060417`.

[3] Foundstone, Inc. `http://www.foundstone.com/`.

[4] FUTo. `http://uninformed.org/?v=3&a=7&t=sumry`.

[5] IceSword 1.18. `http://soft.patching.net/list.asp?id=25`.

[6] Invisiblethings.org. `http://invisiblethings.org`.

[7] Sysinternals – PsTools Suite. `http://www.sysinternals.com/Utilities/PsTools.html`.

[8] rootkit.com. `http://www.rootkit.com/`.

[9] Rootkit Detector. `http://www.rootkitdetector.com/`.

[10] Resplendence Software – Rootkit Hook Analyzer. `http://www.resplendence.com/hookanalyzer`.

[11] Sysinternals – Rootkit Revealer. `http://www.sysinternals.com/Utilities/RootkitRevealer.html`.

[12] Greatis Software – UnhackMe. `http://www.greatis.com/unhackme/`.

[13] Greg Hoglund and James Butler. *Rootkits - Subverting The Windows Kernel*. Addison-Wesley, 2005.

[14] Keith J. Jones, Richard Bejtlich, and Curtis W. Rose. *Real Digital Forensics*. Addison-Wesley, 2005.

[15] Joanna Rutkowska. Subverting vista kernel for fun and profit. In *Black Hat*, Las Vegas, 2006.

[16] Joanna Rutkowska. Rootkit Hunting vs. Compromise Detection. `http://www.invisiblethings.org/papers/rutkowska_bhfederal2006.ppt`, 2005.

[17] Bastian Schwittay. Towards automating analysis in computer forensics. Master's thesis, RWTH Aachen University, Department of Computer Science, 2006.

[18] David Solomon and Mark E. Russinovich. *WIndows Internals*. Microsoft Press, 2005.