*d*Anubis – Dynamic Device Driver Analysis Based on Virtual Machine Introspection

Matthias Neugschwandtner, Christian Platzer, Paolo Milani Comparetti and Ulrich Bayer

Secure Systems Lab, Vienna University of Technology,
{mneug,cplatzer,pmilani,ulli}@seclab.tuwien.ac.at

Abstract. In the escalating arms race between malicious code and security tools designed to analyze it, detect it or mitigate its impact, malicious code running inside the operating system kernel provides an extremely powerful tool. Kernel-level code can introduce hard to detect backdoors, provide stealth by hiding files, processes or other resources and in general tamper with operating system code and data in arbitrary ways.
Under Windows, kernel-level malicious code typically takes the form of a device driver. In this work, we present *d*Anubis, a system for the real-time, dynamic analysis of malicious Windows device drivers. *d*Anubis can automatically provide a high-level, human-readable report of a driver's behavior on the system. We applied our system to a dataset of over 400 malware samples. The results of this analysis shed some light on the behavior of kernel-level malicious code that is in the wild today.

1 Introduction

Malicious code, or malware, is at the root of many security problems on the internet. Compromised computers running malware join botnets and participate in harmful activities such as spam, identity theft and distributed denial of service attacks. It is therefore no surprise that a large body of previous research has focused on collecting, detecting, analysing and mitigating the impact of malicious code.

The analysis of malicious code is an important element of current efforts to protect computer users and systems from malware. Understanding the impact of a malware sample allows to evaluate the risk it poses and helps develop detection signatures, removal tools and mitigation strategies. Because of the large number of new malware samples that appear in the wild each day, malware analysis needs to be a largely automated process.

The automatic analysis of malicious programs is complicated by the fact that malware authors can use off-the-shelf packers to make their samples extremely resistant to static code-analysis techniques. According to a recent large-scale study of current malware [1], over 40% of malware samples are packed using a known packer. Clearly, this is a lower bound to the amount of malware that is packed because malware authors may be using other, yet-unknown packers or implement their own custom solutions. While many current packers can be defeated by generic unpacking tools [2, 3], packers that use emulation-based packing can currently be fully defeated only after manually reverse-engineering their emulator [4]. Furthermore, packers based on opaque constants [5], while not yet available in the wild, can generate binaries that are provably hard to analyze for any static code analyzer.

Because of these limitations, automatic malware analysis is mostly based on a dynamic approach: Malware samples are executed in an instrumented sandbox environment, and their behavior is observed and recorded. A number of dynamic malware analysis systems are currently available that can provide a human-readable report on the malware's activities [6, 7]. The output of these tools can further be used to find clusters of samples with similar behavior [8–10], or to detect specific classes of malicious activity [11].

These systems are able to analyse the behavior of malicious code running in user-mode. The analysis of kernel-side malicious code, however, poses additional challenges. First of all, kernel-level malicious code cannot be reliably detected or analyzed unless the analysis is performed at a higher privilege level than the kernel itself. Otherwise, kernel-level malware would be able to tamper with or disable the analysis engine, in a never-ending arms race. This challenge can be overcome by using out-of-the-box, Virtual Machine Introspection techniques [12], or with more recent in-the-box monitoring techniques that leverage modern CPU features to protect the analysis engine [13]. Using such techniques, the injection and execution of code into kernel-space can be reliably detected [14, 15].

Beyond detection, however, understanding the purpose and capabilities of malicious kernel code is also useful. This is challenging because, in contrast to a user-mode process, kernel code is not restricted to its own address space and to interacting with the rest of the system through a well-defined system call interface. When monitoring the behavior of a system infected by kernel-side malicious code, it is not trivial to reliably (a) attribute an observed event to the malicious code or to the benign kernel and (b) understand the high-level semantics of an observed event. In the limit, kernel-level malware could replace the entire operating system kernel with its own implementation, making understanding the differences between the behavior of a clean system and an infected one extremely challenging. In practice, malware authors prefer to perform targeted manipulations of the operating system's behavior using hooking techniques, and to make use of functions offered by the kernel rather than re-implement existing functionality. Therefore, detecting malware hooking behavior has been the focus of a significant body of recent research [16–19].

One aspect of malicious kernel code that has received less attention is device driver behavior. That is, the malware's interaction with the system's IO driver stacks, and the interface and functionality it offers to userland processes. In this work, we attempt to provide a more complete picture of the behavior of malicious kernel code. We introduce *d*Anubis, an extension to the Anubis dynamic malware analysis system[20] for the analysis of malicious Windows device drivers. *d*Anubis can automatically generate a human-readable report of the behavior of kernel malware. In addition to providing information on the use of common rootkit techniques such as call hooking, kernel patching and Direct Kernel Object Manipulation (DKOM), *d*Anubis provides information about a malicious driver's interaction with other drivers and the interface it offers to userspace. To improve the coverage of its dynamic analysis, *d*Anubis also includes a stimulation engine that attempts to trigger rootkit functionality. Running *d*Anubis on over 400 malware samples that include kernel components allows us not only to validate our tool, but also to perform the largest study of kernel-level malware to date.

In summary, our contributions are the following.

1. We present *d*Anubis, a system for the real-time dynamic analysis of malicious Windows device drivers.
2. Using *d*Anubis, we analyzed over 400 hundred samples and present the results of the first large-scale study of Windows kernel malware. These results give insight into current kernel malware and provide directions for future research.
3. *d*Anubis will be integrated into the Anubis malware analysis service, making it available to researchers and security professionals worldwide.

2 Overview

Rootkits provide malware authors with one of their most flexible and powerful tools. The term "rootkit" derives from their original purpose of maintaining root access after exploiting a system, being a "kit" of pieces of technology with the purpose to hide the attacker's presence in the system [21]. This can include hiding files, processes, registry keys and network ports that could reveal an intruder's access to the system. Early rootkits ran entirely in user space and operated by replacing system utilities such as ls, ps and netstat with versions modified to hide the activities of an unauthorized user. Later rootkits included kernel-level code, enabling the attacker to do virtually anything on the target machine, including directly tampering with control flow and data structures of the operating system. Today, the boundaries between different classes of malware have become indistinct; many techniques originally used in rootkits are now employed in other types of malware, such as bots, worms or Trojan horses. In this paper, we will use the term *rootkit* to refer to malware that uses kernel-level code to carry out its operations.

To inject malicious code into the kernel, the attacker can either use an undetected, unpatched kernel exploit, such as a buffer overflow, or – much more convenient – load and install a device driver. The latter method has the disadvantage that it depends on hijacking an administrator account. This is in practice not much of a problem since most Windows machines are operated with Administrator privileges out of convenience for the user. While Windows Vista or 7 at least require the user to confirm administrative actions such as driver loading, Windows XP provides APIs that allow loading an unsigned driver without any user interaction. As a result, a rootkit usually comes as a user-mode executable that loads a device driver, which in turn provides all the powerful functionality.

The goal of *d*Anubis is to provide a human-readable report of a device driver's behavior. Note that detection, that is, distinguishing malicious device drivers from benign ones, is outside the scope of this work. Some behavior, such as directly patching kernel code, may give a clear indication that a sample is malicious. Many types of suspicious behavior, however, may also be exhibited by benign code, especially security tools such as antivirus or personal firewall software. The reason is that these tools may attempt to "outsmart" malware by running deep inside the operating system.

*d*Anubis analyses a driver's behavior from outside the box, using a Virtual Machine Introspection (VMI) approach [12, 22]. Our implementation is an extension of the Anubis malware analysis system, and is based on the Qemu [23] emulator. By instrumenting the emulator, we can monitor the execution of code in the guest OS, to observe events such as the execution of the malicious driver's code, invocation of kernel functions, or access to the guest's virtual hardware. Furthermore, by instrumenting the emulator's Memory Management Unit (MMU), we can observe the manipulation of kernel memory performed by the rootkit. *d*Anubis attempts to reconstruct the high level semantics of the observed events.

One focus of our analysis is to monitor all the "legitimate" communication channels between the rootkit and the rest of the system. That is, all channels provided by the OS for the driver to interact with the kernel, with other drivers and with user-space. This includes the invocation of kernel functions as well as the use of devices to participate in Windows I/O communication and to receive commands from user-space. Additionally, *d*Anubis can detect the use of a number of rootkit techniques such as hooking and runtime patching of kernel routines, and provide precise information on which routines are patched or hijacked. Overall, our tool can thus provide a comprehensive picture of the behavior of malicious kernel code.

3 System Implementation

A major drawback of any VMI-based approach is the loss of semantic information about the guest operating system. Instead of objects and well-defined data structures, only a heap of bytes is visible from the host system's point of view. To reconstruct the necessary information we extract all exported symbols and data structure layouts from the Windows OS as a preliminary step. During analysis, we utilize guest view casting of the virtual machine memory as proposed by Jiang et al. [22].

A further problem arises when comparing process-based dynamic analysis, as it is implemented in Anubis or comparable sandboxes, and driver-aware approaches. For userland processes, it is sufficient to watch and trace instructions belonging to the process in question, whose execution context is easily identifiable. Kernel-level code, however, can be triggered by multiple means, like interrupts or system calls, thus possibly running in the context of an arbitrary user-mode process. Therefore, we use the instruction pointer to determine whether the code being executed belongs to the malicious driver.

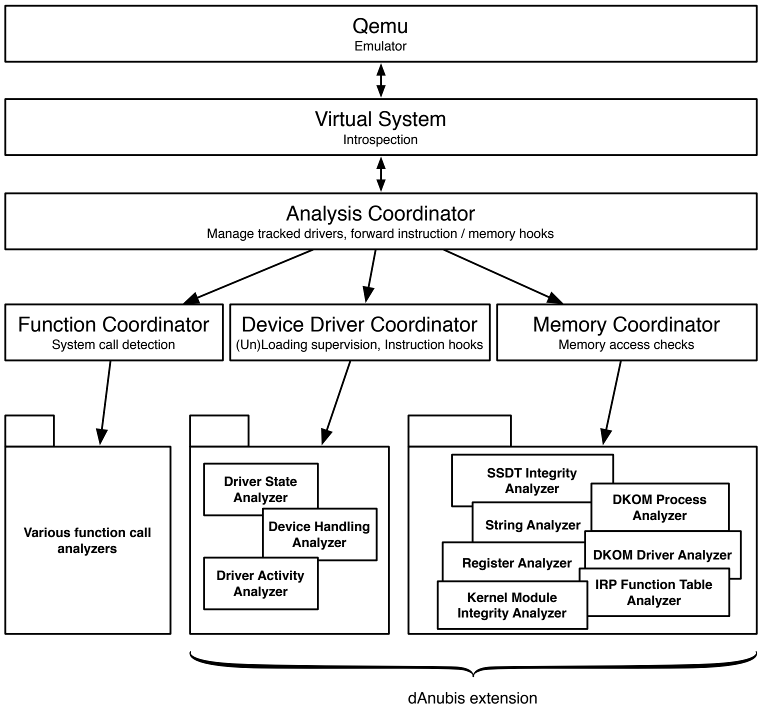

Fig. 1. Architectural overview

Our system consists of two major parts. The *Device Driver Coordinator* handles device driver-related operations while the *Memory Coordinator* is responsible for rootkit activity. Specific analysis tasks are carried out by a number of *Analyzers*. Figure 1 provides an overview of our system's architecture.

3.1 Device Driver Analysis

To analyze the behavior of device drivers, we monitor all available interfaces through which the driver can interact with the rest of the kernel, with other drivers and with userland processes [24]. The first thing to do in this respect is intercepting the low-level load- and unload mechanisms. The Windows kernel objects involved in the loading procedure provide us with import information, above all the codebase location of the driver, which allows us to track instructions belonging to the driver. Furthermore, the function addresses of the driver's entry routine (comparable to the main function of a normal program), its unload routine and its I/O dispatch routines can be gathered. Among the latter are the *major functions*, a set of well-defined functions a driver has to provide in order to participate in Windows I/O device communication. Knowing the function addresses allows us to implement a basic state supervision and relate later analysis events to the context in which they occurred.

Driver communication. The communication endpoint of a driver is the device. To receive I/O requests, a driver has to create a device and provide the aforementioned major functions to handle the requests. If requests require complex processing that can be subdivided in different parts, devices can be arranged in a stack. For example this could be the case for an encrypted file system, where the encryption is handled by the topmost driver in the stack and actual hardware access by the lowest driver. Driver stacking can be exploited for malicious purposes. For example, a rootkit may attach to the filesystem device stack to filter the results of file listings before forwarding them up the stack, while a keylogger can attach to the keyboard device to monitor all keystrokes. Therefore, we monitor whether a driver creates or attaches to a device.

Actual communication between devices and user- or kernel-mode code happens by encapsulating the request parameters in an I/O request packet (IRP) by the Windows I/O manager, which invokes the corresponding major function of the topmost driver in the stack. The IRP is then passed down and up its destined device stack. We intercept calls to attached devices, analyze these IRPs and watch for completion routines, which are invoked upon completion of a request, allowing them to filter results. Larger data amounts are not directly passed within the IRP, rather the way how it can be done – buffered I/O, direct I/O or neither of them – is specified. We also parse this information and detect strings in the data to be able to track further references to them during execution. This is accomplished using dynamic data tainting [25]. Specifically, we use techniques from [9] to detect the use of these tainted bytes in string comparison operations. This can in some cases reveal triggers for conditional behavior of the analyzed binary.

Driver activity. To get a picture of what the driver actually does when one of its functions is executed, we log calls to all exported Windows kernel functions. We expect that rootkit developers will not develop everything from scratch but make use of existing functionality. In our evaluation, we show that this assumption proves correct for most real-world samples.

We also scan the complete driver image for string occurrences and taint them to be able to log any subsequent access to such strings. As we will show in the evaluation, this simple mechanism can in practice reveal trigger conditions in the malicious code, such as the names of files and processes that are to be hidden.

3.2 Memory Analysis

Taking a look at the "standard" rootkit techniques, one similarity is obvious: they all somehow tamper with kernel memory. We achieve the goal of detecting malicious kernel memory manipulation by hooking the memory management unit (MMU) of Qemu. This allows us to detect write access independently of the instruction used so that we can put specific memory regions of the guest OS under supervision and analyse malicious changes.

Since some kernel regions are flagged read-only, rootkits often use memory descriptor lists (MDLs) to re-map a desired memory region to a new virtual address and bypass write protection. Therefore *d*Anubis also needs to analyze MDL usage.

Call table hooking. The first interesting memory region is represented by the system service dispatch table (SSDT). This table keeps a list of Windows system call handlers that can be invoked by usermode processes by issuing an interrupt or using the `sysenter` instruction. In a healthy system, the called table entry points to the beginning of the desired service routine. Malicious drivers, however, can overwrite the SSDT entry to point to arbitrary code. When called, this code typically forwards the request to the original service function, receives the response and alters it before passing it to the original caller. Again, this method provides the possibility to exclude rootkit-related information from queries like directory listings or process lists.

The same principle applies to hooks of other call tables, such as the major function dispatch table of device drivers. For incoming IRPs the Windows I/O manager normally looks up the proper handling routine in this table before invoking it. Again, the control flow can be re-routed by changing an entry in this table.

To monitor call table hooking behavior, we watch the complete memory region where a call table resides. If a manipulation occurs in one of the watched memory regions, we will know exactly which system service or major function has been hooked and monitor following calls to the hook.

DKOM. Direct kernel object manipulation (DKOM) modifies important data objects residing in Windows kernel memory. This way it can alter system behavior without changing the control flow. To hide a certain process from the process list, for example, the `EPROCESS` structure has to be altered such, that the forward and backward pointers are directed around the target entry, effectively excluding it. Although this method is more powerful and harder to detect than hooking, it has its shortcomings. It is, for instance, not feasible to mask a file from a directory listing this way.

To detect and understand DKOM activity, it is necessary to know the exact location of the kernel objects as well as their data structure and meaning. In our current implementation, DKOM detection is limited to the process and driver lists, that are frequently targeted by rootkits for the purpose of stealth.

Runtime patching. Rootkits can also affect the system by directly patching existing kernel code in memory. Usually the patch jumps to a detour containing malicious code and then back again to the original code.

To detect runtime patching, we walk through the `PsLoadedModuleList` to get the information on the codebase of the kernel modules and put them under supervision. On an integrity breach we determine exactly which kernel function has been patched by matching the patched addresses against information automatically obtained from the Windows debugging symbols.

Hardware access. In addition to manipulating kernel memory, rootkits can affect the system by directly accessing the underlying hardware. *d*Anubis monitoring of hardware access is currently limited to detecting writes to the `IA32_SYSEN-`

`TER_EIP` model specific register. This register points to the system service dispatcher routine. Making this register point to malicious code places rootkit code in the execution path of all system calls.

3.3 Stimulation

Rootkit functionality often depends on external stimuli. While the entry routine may already perform some malicious activity such as hooking or patching, many types of behavior may only be performed when triggered by specific user behavior. Without the required stimuli, such as keyboard events for keyloggers or process enumeration for process-hiding rootkits, the results of dynamic analysis are bound to be incomplete.

Our goal is to improve code coverage by simulating user activity with a stimulation engine placed in the virtual machine. To this end, we implemented a stimulator that repeatedly issues a number of Windows API calls. For example, the stimulator issues the `EnumProcesses` API call, that lists all currently running processes, triggering process hiding behavior. Similarly, it issues the `RegEnumKeyEx` call, revealing register hiding. The `FindFirstFile` and `FindNextFile` API calls are used as well to reveal file hiding behavior. Note that although the directory we are querying with these calls might not contain files to be hidden, hook code will nevertheless be executed. To trigger network hiding behavior, the `GetUdpTable` and `GetTcpTable` calls are used. Furthermore, random keypresses and mouse actions are injected to simulate user input.

4 Evaluation

To evaluate our prototype, we first verified its functionality on a set of rootkits with known behavior. For this, we chose a representative suite of six well-known rootkits that employ the popular techniques described in the previous sections and can be obtained from `www.rootkit.com`. For each of the six rootkits, *d*Anubis was able to correctly identify its characteristic behavior and present it in the human-readable report. Table 1 shows which *d*Anubis components were involved in providing information on each of the rootkits.

The first sample we selected is **TCPIRPHook**. This malware modifies the address of `DEVICE_CONTROL` in the major function table of Tcpip.sys, rerouting it to a hooking function. This allows the rootkit to hide open network ports. This hooking behavior was detected by the IRP function table analyzer.

We then selected the **HideProcessMDL** rootkit as a straightforward example of process hiding by SSDT hooking. This rootkit first creates an MDL in order to gain write access to the SSDT. This is recognized by the memory coordinator, that can thus apply the mapping upon write access to the watched memory region. This enables the SSDT integrity analyzer to report the rootkit's hooking of `NtQuerySystemInformation` as soon as the hook is placed. Once the stimulator queries for the running processes, the driver state analyzer detects the call to the hooking routine, which in turn invokes the original `NtQuerySystemInformation` function. Furthermore, The string analyzer reveals that the hooking routine accesses the string “_root_” indicating the name of the process that is to be hidden.

Klog is a key-logger based on layered filter drivers. During driver initialization it creates a log file and a virtual device, which it uses to attach to the keyboard device stack. This behavior, along with name and location of the log file, is revealed by the device handling, driver activity and string analyzer. Upon stimulation of keystrokes, the device handling analyzer further detects that the driver Kbdclass is called by Klog and dynamically adds the completion routine to the state analysis and so execution of the completion routine is subsequently logged.

Migbot uses run-time patching to modify the kernel functions `SeAccessCheck` and `NtDeviceIoControlFile`. The integrity breach along with the names of the functions is immediately reported by the kernel integrity analyzer.

The **FU** rootkit uses DKOM for process and driver hiding. However, it only performs this hiding function when it receives commands from a user-mode program through device communication. To test the DKOM analyzers we manually ordered FU to hide certain processes and drivers. These manipulations along with the corresponding filenames were immediately reported by the DKOM analyzers. Furthermore, the device handling analyzer revealed the string “msdirectx.sys” in the communication with the user-mode program. This is the name of the driver we ordered FU to hide.

Finally, we tested the **sysenter** rootkit to verify that sysenter hooks are correctly recognized.

Analyzer	TCPIRPHook	HideProcessMDL	Migbot	Klog	FU	sysenter
Device driver coordinator	√	√	√	√	√	√
Memory coordinator	-	√	-	-	-	-
Driver state analyzer	√	√	√	√	√	√
Driver activity analyzer	√	√	√	√	√	-
IRP function table analyzer	√	-	-	-	-	-
SSDT analyzer	-	√	-	-	-	-
String analyzer	-	√	-	√	√	-
Integrity analyzer	-	-	√	-	-	-
Device handling analyzer	-	-	-	√	√	-
DKOM process analyzer	-	-	-	-	√	-
DKOM driver analyzer	-	-	-	-	√	-
Register analyzer	-	-	-	-	-	√

Table 1. *d*Anubis Testing results

During the analyis of these six rootkits, we also measured the impact of *d*Anubis on the performance of the Anubis sandbox. The overhead added by *d*Anubis was between 14% and 33%. These results are consistent with our goal of integrating driver analyis into a large-scale dynamic analysis framework, because the entire analysis of a malware sample can still be performed in real time, within the six minute timeslot that Anubis typically allocates to an analysis run. This is in contrast to some previous systems, such as K-Tracer [18], that need to

Driver activity	number of samples exhibiting behavior
Device driver loaded	463
Windows kernel functions used	360
Windows device I/O used	339
Strings accessed	300
Kernel code patched	76
Kernel call tables manipulated	37
MDL allocated	34
Kernel object manipulated	3

Table 2. Global analysis statistics

perform a heavyweight analysis of detailed execution traces. For instance, K-Tracer needed over two hours to analyze the HideProcessMDL rootkit.

4.1 Quantitative results

We used *d*Anubis to conduct a large-scale study of kernel malware behavior. To obtain malware samples for this study, we leveraged the analysis results of the existing Anubis system. We first considered 64733 malware samples successfully analysed by Anubis in the month of August 2009. Among those, we selected the 463 samples (0.72%) that loaded a device driver during Anubis analysis. More precisely, we selected samples that performed the `NtLoadDeviceDriver` system call. We then repeated the analysis of these samples using *d*Anubis. Note that some malware may use different mechanisms to load kernel code, such as the undocumented `NtSetSystemInformation` system call. Therefore, the actual number of rootkit samples in the dataset may have been higher than 463. While *d*Anubis is capable of correctly analysing rootkits loaded using such methods, the legacy Anubis system does not detect and log this behavior.

All samples were automatically processed by our implementation and correctly recognized as drivers. For each test run, we defined a timeout of six minutes, during which the driver had time to carry out its operations. During the entire analysis stimuli where provided by our stimulation engine. Table 2 shows which high-level activity of the samples could be observed by *d*Anubis. Three quarters of the samples performed device I/O activity. Among the typical rootkit techniques, MDL-enabled call table hooks and runtime patching seem to be very popular compared to DKOM.

Table 3 shows an overview of device-related activity. The majority of the samples – 339 – created at least one device. In 110 cases the device was actively used for communication by a user mode program: It was at least opened, as indicated by the calls to the CREATE major function. Out of these, 86 samples carried out further communication using the device control interface. In the data buffers passed along with the IRPs, meaningful communication strings could be found in 24 cases (an example is shown in Table 7). Only two samples attached to a device stack and registered completion routines. These results allow us to draw the conclusion that devices are primarily used for communication with user mode

Device activity	number of samples
Device created	339
Driver's device accessed from user mode	110
Strings detected during communication	24
Attaches to device stack	2
Registers completion routine	2

Table 3. Device analysis statistics

	SSDT hook	runtime patching	DKOM	IRP hook	Filter driver	total
Registry	5	45	0	0	0	50
File	8	2	0	0	2	12
Process	3	2	3	0	0	8
Driver	0	0	3	0	0	3
Network port	5	0	0	1	0	6

Table 4. Hiding statistics: subject vs. technique

programs whereas hijacking device stacks seems to be far less popular. However, a significant amount of samples – 229 – register a device, but this device is never put to any use during the entire analysis run. The most likely explaination for this discrepancy is that the associated executables are merely launchers for the drivers, that in turn wait for further commands to be manually issued by the human attacker. This is the case of the FU rootkit we previously discussed. This means that some of the malicious functionality of these rootkits lies dormant, waiting for activation, and is therefore not covered by the dynamic analysis. This result highlights the need for further research in rootkit analysis. Future analysis systems might be able to automatically trigger rootkits' dormant functionality, although the problem of finding trigger conditions in arbitary code cannot be solved in general [26].

Overall, only 15% percent of the samples carried out rootkit activities. Table 4 shows the amount of samples that provided stealth broken down by the techniques employed as well as the type of object being hidden. Clearly, call table hooking and runtime patching are the more widespread techniques: only three samples used DKOM for process hiding. The same samples also used DKOM to hide their device drivers from the list of kernel modules.

Of the 19 samples that employed SSDT hooking, most hooked more than one system call. Table 5 shows the most popular system calls hooked. The idea that rootkits strive to provide stealth is confirmed by the fact that the system calls to list files, registry keys and processes are clearly favored by the attackers. In addition to stealth, another common goal of rootkits is to disable antivirus protection. Samples hooking the `NtCreateProcessEx` use this to prevent the launch of anti-malware programs. IRP function table hooks were only employed by one sample, that hooked the `DEVICE_CONTROL` major function of Tcpip.sys. Rerouting the device control interface, that is the main communication access point to the driver, allows this rootkit to hide open network ports. A less subtle method of hijacking the device control interface is to directly hook the `NtDeviceIoControlFile` system call. This technique is used by five samples, also for the purpose of port hiding. None of the samples used sysenter hooks. The StringAnalyzer, that was mainly introduced to reveal trigger conditions, shows its full potential with SSDT hooks. For example in more than half of the cases where `NtQueryDirectoryFile` or `NtQuerySystemInformation` has been hooked, the filenames to be hidden showed up in the analysis. This also demonstrates the importance of event stimulation and the effectiveness of our stimulation engine, as the strings were mainly detected during execution of hooking routines that would not have been called without stimulation.

System service	samples
NtQueryDirectoryFile	8
NtCreateProcessEx	8
NtDeviceIoControlFile	5
NtEnumerateKey	3
NtQuerySystemInformation	3
NtEnumerateValueKey	2
NtOpenKey	2
NtClose	1
NtCreateKey	1
NtSetInformationFile	1
NtSystemDebugControl	1
NtOpenProcess	1
NtOpenThread	1
NtCreateFile	1
NtOpenIoCompletion	1
NtSetValueKey	1
NtDeleteValueKey	1
NtMapViewOfSection	1

Table 5. Hooked system calls

About ten percent of the samples used runtime patching. In these cases *d*Anubis took advantage of kernel debugging symbols to automatically identify the patched kernel functions.

Table 6 shows that, as is the case for SSDT hooks, functions that take part in processing file, process and registry key queries are among the most popular kernel functions to be manipulated. The `pIofCallDriver` pointer points to the low-level kernel code implementation that invokes the major functions of a driver. Rerouting its control flow allows a rootkit to intercept and manipulate IRPs. The `KiFastCallEntry` function is the default handler of the `sysenter` instruction. By patching this function, a rootkit inserts malicious code into the code path of every system call. In this case the automatic analysis cannot tell us what types of objects are actually being hidden by this rootkit.

Kernel function	samples
NtQueryValueKey	42
NtSetValueKey	2
PsActiveProcessHead	2
NtEnumerateKey	1
IoCreateFile	1
NtQueryDirectoryFile	1
NtOpenKey	1
NtCreateKey	1
pIofCallDriver	1
KiFastCallEntry	1
ObReferenceObjectByHandle	1
KiDoubleFaultStack	1

Table 6. Patched kernel functions

4.2 Qualitative results

To provide a clearer picture of the types of behavior that can be revealed by the analysis of a kernel malware sample using *d*Anubis, we selected three interesting samples out of the dataset discussed in the previous section. For matters of space and readability the relevant information from the reports has been condensed into tables.

In Table 7 a selection of analysis results of Sample A are shown. This rootkit hooks various system calls, among them functions suitable for file and registry key hiding. Process hiding is performed using DKOM. As the hooking functions are very similarly structured, the hook of NtEnumerateKey has been chosen as an example. After calling the original function it queries an object and its name. It then performs some string operations, which is usually necessary for filtering information. Furthermore, in the course of the driver's entry function a device is created. This device is then used for user mode communication: DEVICE_CONTROL is called from user mode several times and both a registry key and a file name could be intercepted, that the driver is presumably ordered to hide. DEVICE_CONTROL itself looks up objects according to their name or ID using ObReferenceObjectByName and PsLookupProcessByProcessId – again an indication that these objects are to be hidden.

In Table 8 selected analysis results of Sample B are shown. During driver entry, this sample creates a named device (FILEMON701) for communication with user mode. This device is then used to issue commands to the driver via `FastIoDeviceControl` to install filter drivers for sr.sys and mrxsmb.sys. To this end, two unnamed devices are created and attached to the associated device stacks.

The sr.sys driver is the Windows restore filesystem filter driver, that tracks and copies system files before changes. The mrxsmb.sys is the Windows SMB Redirector, a filesystem driver that provides access to remote folders shared over the SMB/CIFS protocol. Our stimulation engine, however, does not perform operations on network shares, nor does it modify system files. Therefore, during

Driver name	syssrv	
Created devices	\Device\MyDriver	
Rootkit activity	NtOpenProcess hooked	SSDT Hook
	NtOpenThread hooked	SSDT Hook
	NtCreateFile hooked	SSDT Hook
	NtOpenIoCompletion hooked	SSDT Hook
	NtQueryDirectoryFile hooked	SSDT Hook
	NtOpenKey hooked	SSDT Hook
	NtEnumerateKey hooked	SSDT Hook
	NtEnumerateValueKey hooked	SSDT Hook
	NtSetValueKey hooked	SSDT Hook
	NtDeleteValueKey hooked	SSDT Hook
	svchost.exe hidden	DKOM process hiding
	ntoskrnl.exe: PsActiveProcessHead	Runtime patching
Invoked major functions	CREATE	called 5x from user mode
	DEVICE_CONTROL	called 5x from user mode
	CLOSE	called 5x from kernel mode
Detected strings	syssrv	in DEVICE_CONTROL IRP
	\Device\HarddiskVolume1 \WINDOWS\system32\mssrv32.exe	in DEVICE_CONTROL IRP
	SOFTWARE\Microsoft\Windows \CurrentVersion\Run\mssrv32	in DEVICE_CONTROL IRP
	\Device\%s	during entry
	MyDriver	during entry
Used kernel functions	IoCreateDevice	during entry
	KeInitializeMutex	during entry
	ObReferenceObjectByName	during DEVICE_CONTROL
	ObReferenceObjectByHandle	during DEVICE_CONTROL
	ObQueryNameString	during DEVICE_CONTROL
	KeWaitForSingleObject	during DEVICE_CONTROL
	KeReleaseMutex	during DEVICE_CONTROL
	PsLookupProcessByProcessId	during DEVICE_CONTROL
	NtEnumerateKey	during NtEnumerateKey Hook
	ObReferenceObjectByHandle	during NtEnumerateKey Hook
	ObQueryNameString	during NtEnumerateKey Hook
	wcslen, wcscpy, wcscat	during NtEnumerateKey Hook
	KeWaitForSingleObject	during NtEnumerateKey Hook
	KeReleaseMutex	during NtEnumerateKey Hook

Table 7. Analysis report, Sample A

anlysis we only observed interception of `QUERY_VOLUME_INFORMATION` of sr.sys, that is used to query free disk space or file types. This highlights the challenge of implementing a stimulation engine that is capable of activating all hooks inserted by a rootkit. Note that generic hook-detection tecniques such as Hookfinder [17] and KTracer [18] cannot detect hooks that are never activated.

The Windows system restore functionality can be used to perform a system rollback based on the information gathered by the sr.sys driver. By attaching to sr.sys, the rootkit can prevent system restore from obtaining the necesary information on system file changes and ensure that the rootkit will not be removed by a rollback. The device name and symbols included in the rootkit's executable suggest that the publicly available sources of the Filemon tool [27] were used as a basis for this rootkit.

Driver name	FILEMON701	
Created devices	\Device\Filemon701 unnamed device 1 unnamed device 2	
Attached to devices	sr MRxSmb	
Completion routine	QUERY_VOLUME_INFORMATION	for device "sr"

Invoked I/O functions	CREATE
	QUERY_VOLUME_INFORMATION
	CLEANUP
	CLOSE
	READ
	FastIoDeviceControl
		from user mode
		from kernel mode
		from kernel mode
		from kernel mode
		from kernel mode

Used kernel functions	IoCreateDevice IoCreateSymbolicLink IoGetCurrentProcess ZwCreateFile IoCreateDevice IoAttachDeviceByPointer	during entry during entry during entry during FastIoDeviceControl during FastIoDeviceControl during FastIoDeviceControl

Table 8. Analysis report, Sample B

Sample C performs SSDT hooking on the `NtQueryDirectoryFile` and `NtEnumerateValueKey` system calls to provide stealth. Furthermore, this sample calls the `PsSetLoadImageNotifyRoutine` to receive a callback whenever a process or driver image is loaded. The sting analyzer reveals that this callback accesses a number of strings hardcoded in the rootkit image, that are shown in Table 9. These strings are clearly filenames, most of them related to antivirus software or other security tools. The most logical explanation for these observations is that the rootkit uses this technique to interfere with the loading and execution of anti-malware programs. Manual analysis confirms that the callback uses the `ZwTerminateProcess` function to kill these processes. We could not directly observe this behavior during analysis because none of the listed processes are executed in our analysis environment. This attack on security software highlights the need for a secure execution context for analysis software. The watchdog.sys

vsdatant.sys	watchdog.sys	zclient.exe	bcfilter.sys	bcftdi.sys
bc_hassh_f.sys	bc_ip_f.sys	bc_ngn.sys	bc_pat_f.sys	bc_prt_f.sys
bc_tdi_f.sys	filtnt.sys	sandbox.sys	mpfirewall.sys	msssrv.exe
mcshield.exe	fsbl.exe	avz.exe	avp.exe	avpm.exe
kavsvc.exe	klswd.exe	ccapp.exe	ccevtmgr.exe	ccpxysvc.exe
issvc.exe	rtvscan.exe	savscan.exe	bdss.exe	bdmcon.exe
cclaw.exe	fsav32.exe	fsm32.exe	gcasserv.exe	icmon.exe
nod32krn.exe	nod32ra.exe	pavfnsvr.exe	kav.exe	kavss.exe
inetupd.exe	livesrv.exe	iao.exe	Windows-KB890830-V1.32.exe	

Table 9. Processes targeted by Sample C

file that is among those targeted by the rootkit is a driver that is also used by CWSandbox [7], a malware analysis system that is not based on VMI but on in-the-box monitoring using a kernel driver.

5 Related work

In this section we will discuss related research in the area of detection and analysis of malicious kernel code.

Integrity checking. In [14] the authors implement a tamper-resistant rootkit detector for Linux systems that uses VMI. To detect runtime patching, this system verifies the integrity of the kernel by hashing portions of clean memory considered critical and regularly comparing the hashes with their up-to-date counterparts. A limitation of this approach lies in the need to balance security with performance in selecting how often to perform hashing. In any case, such a system cannot guarantee that no injected code will ever be executed. To protect kernel code, an improved solution is offered by Nickle [15]. Nickle instruments the memory management unit of an emulator to redirect code fetches performed in kernel mode to a protected memory region. This way, it can detect code injected into the kernel as soon as its first instruction is executed.

Cross-view detection. Hiding an intruder's presence on a compromised system is a widespread goal of rootkits. This very behavior, however, can be exploited to detect kernel compromise. For this, cross-view detection approaches [12, 28] compare system information obtained from a high-level abstract view, e.g. the Windows API, with information extracted from a lower level view, in order to reveal hiding. [14] uses a cross-view approach to detect process, kernel module and network port hiding. To detect process hiding, the authors of [29] use Antfarm [30] to determine implicit process information, without prior knowledge of the monitored guest's OS. They then perform cross-view comparison, detecting process hiding. While OS-independence is an attractive advantage of this approach, the technique employed cannot be easily extended to other types of stealth behavior. A drawback of cross-view is that it can detect the fact that something has been hidden, but cannot provide any information on how this has been done. Moreover, for information arranged in more complex data structures cross-view soon becomes impractical. For example, it can only detect file hiding if the contents of every directory on the system are compared.

Hooking detection. Hooking is another characteristic aspect of rootkit behavior that can be exploited for detection purposes. In [16], the authors present Hookmap, a system that can systematically discover possible hooking points in the execution path of system calls, enabling detection of rootkits that use these hooking points. Hookfinder [17] uses dynamic taint propagation to monitor the impact of a rootkit on the kernel. It detects a hook when a tainted value is loaded to the instruction pointer. In [18], the authors introduce k-tracer, a system that performs a sophisticated analysis of malware hooking behavior. For this, k-tracer first records an execution trace for a system call, reaching from the `sysenter` to the `sysexit` instruction. Then, an offline analysis is applied to the trace, by performing forward slicing to identify *read access* and backward slicing to reveal *manipulation* of sensitive data. This approach is however not compatible with our performance requirements for large-scale malware analysis, since k-tracer may require hours to analyze a single rootkit.

Rootkit analysis. Closely related to *d*Anubis is recent work on the dynamic analysis of rootkit behavior. In [31], the authors present rkprofiler, a system for the analysis of Windows kernel malware that is also based on VMI using Qemu. This system can reveal which system calls have had their execution paths modified to include injected code. Both [31] and [19] address the problem of understanding the semantics of rootkit modifications to dynamically allocated kernel memory. For this, they introduce techniques to recursively infer the type of an object in memory based on the type of the pointers that are used to access it, starting from the known structure of static kernel objects and function parameters.

6 Limitations

Our evaluation demonstrates that *d*Anubis can provide a substantial amount of information on malicious drivers. Nonetheless, our system suffers from a number of limitations.

Rootkit detection. To be able to analyze a rootkit's behavior, *d*Anubis must first detect the rootkit's presence in the analysed system. That is, it must be aware that extraneous code has been inserted into kernel space. For this, *d*Anubis relies on hooking system calls used for loading drivers. Therefore, we are unable to analyse rootkits injected through kernel or device driver exploits. This is a design choice, because it allows most *d*Anubis instrumentation to remain disabled until a driver is loaded, improving performance on the majority of analysed samples that *do not* load a driver. At the cost of some performance, this limitation could be addressed by integrating techniques from [15], that can reliably detect the execution of injected code. The detection of return-oriented rootkits [32], however, remains an open problem, since these rootkits do not inject any code into kernel space.

Dynamic analysis coverage. A general limitation of dynamic approaches to code analysis is that only code that is actually executed can be analyzed. In order to cover as many code paths as possible, we strive to stimulate typical rootkit functionality. Behavior that is triggered by benign user activity can be emulated to a certain extent by our stimulator. However, our large scale study has shown that many samples waits for commands to be issued from userspace through a device interface and never receive any such commands during analysis. The rootkit behavior associated with these commands is therefore not covered by our analysis. Related work in this field [19, 18, 31] does not specifically address this issue. Future research in rootkit analysis could attempt to design a stimulator capable of automatically issuing valid commands to malicious device drivers.

Related to the problem of coverage is the issue of detection of virtual environments and of analysis environments in general. If malware can detect our analysis environment it can thwart analyis by simply refusing to run. Unfortunately, implementing an undetectable virtual environment is infeasable in practice [33], although attackers may be reluctant to make their malware not function on widely deployed virtual environments. Defeating VM detection is largely a reactive, manual process. However, recent research [34, 35] has shown that it may be possible to automatically detect previously unknown virtualization detection techniques.

Event attribution. In order to differentiate between legitimate and malicious actions, the origin of these actions has to be determined. To attribute a write access to a monitored driver we take the program counter of the instruction that carried out the manipulation and compare it with the codebase of the driver. While this technique works in practice, it can easily be fooled if the malicious driver uses a legitimate kernel function to manipulate the desired memory region. [36] introduces secure control attribution techniques based on taint tracking to tackle a similar problem in the context of (malicious) shared-memory browser extensions. Since Anubis provides tainting support, these techniques could also be adapted for integration in *d*Anubis.

7 Conclusions

The analysis of malicious code faces additional challenges when the code to be analyzed executes in kernel space. In this work, we discussed the design and implementation of *d*Anubis, a system for the dynamic analysis of Windows kernel malware. *d*Anubis can provide a comprehensive picture of a device driver's behavior and its interaction with the operating system, with other drivers and with userland processes.

We used *d*Anubis to conduct a large-scale study of kernel malware behavior that provides novel insight into current kernel-level threats. In this study, we analysed more than 400 recent rootkit samples to reveal the techniques employed to subvert the Windows kernel and, in most cases, the nefarious goals attained with these techniques. These results demonstrate that *d*Anubis can be an effective tool for security researchers and practitioners. We therefore plan to make it publicly available as part of the Anubis malware analysis service.

Acknowledgments

This work has been partially supported by the European Commission through project ICT-216026-WOMBAT funded under the 7th framework program.

References

1. Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., Kruegel, C.: Insights into current malware behavior. In: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET). (2009)
2. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: 22nd Annual Computer Security Applications Conf. (ACSAC). (2006)
3. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: ACM Workshop on Recurring malcode (WORM). (2007)
4. Rolles, R.: Unpacking virtualization obfuscators. In: 3rd USENIX Workshop on Offensive Technologies. (WOOT). (2009)
5. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection . In: 23rd Annual Computer Security Applications Conference (ACSAC). (2007)
6. Bayer, U.: Ttanalyze a tool for analyzing malware. Master's thesis, Vienna University of Technology (2005)
7. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE Security and Privacy **2**(2007) (5)
8. Bailey, M., Oberheide, J., Andersen, J., Mao, Z., Jahanian, F., Nazario, J.: Automated Classification and Analysis of Internet Malware. In: Symposium on Recent Advances in Intrusion Detection (RAID). (2007)
9. Bayer, U., Milani Comparetti, P., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, Behavior-Based Malware Clustering. In: Network and Distributed System Security Symposium (NDSS). (2009)
10. Rieck, K., Holz, T., Willems, C., Duessel, P., Laskov, P.: Learning and classification of malware behavior. In: Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). (2008)
11. Jacob, G., Debar, H., Filiol, E.: Malware behavioral detection by attribute-automata using abstraction from platform and language. In: Recent Advances in Intrusion Detection (RAID). (2009)
12. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Network and Distributed Systems Security Symposium (NDSS). (2003)
13. Sharif, M.I., Lee, W., Cui, W., Lanzi, A.: Secure in-vm monitoring using hardware virtualization. In: ACM conference on Computer and communications security (CCS). (2009)
14. Quynh, N.A., Takefuji, Y.: Towards a tamper-resistant kernel rootkit detector. In: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, ACM (2007) 276–283
15. Riley, R., Jiang, X., Xu, D.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In: RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection, Berlin, Heidelberg, Springer-Verlag (2008) 1–20
16. Wang, Z., Jiang, X., Cui, W., Wang, X.: Countering persistent kernel rootkits through systematic hook discovery. In: Recent Advances in Intrusion Detection (RAID). (2008)
17. Yin, H., Liang, Z., Song, D.: Hookfinder: Identifying and understanding malware hooking behaviors. In: Network and Distributed Systems Security Symposium (NDSS). (2008)
18. Lanzi, A., Sharif, M., Lee, W.: K-tracer: A system for extracting kernel malware behavior. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium. (2009)

19. Riley, R., Jiang, X., Xu, D.: Multi-aspect profiling of kernel rootkit behavior. In: EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems, New York, NY, USA, ACM (2009) 47–60
20. Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. Journal in Computer Virology **2**(1) (2006) 67–77
21. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley Professional (2005)
22. Jiang, X., Wang, X., Xu, D.: Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. (2007)
23. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the annual conference on USENIX Annual Technical Conference, USENIX Association (2005) 41–41
24. Orwick, P., Smith, G.: Developing Drivers with the Microsoft Windows Driver Foundation. Microsoft Press (2007)
25. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Network and Distributed Systems Security Symposium (NDSS). (2005)
26. Sharif, M.I., Lanzi, A., Giffin, J.T., Lee, W.: Impeding malware analysis using conditional code obfuscation. In: Network and Distributed System Security (NDSS). (2008)
27. Russinovich, M.: Filemon. (2010) `http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx`.
28. Beck, D., Vo, B., Verbowski, C.: Detecting stealth software with strider ghostbuster. In: Proceedings of the 2005 International Conference on Dependable Systems and Networks. (2005) 368–377
29. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Vmm-based hidden process detection and identification using lycosid. In: VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM (2007) 91–100
30. Jones, S.T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Antfarm: tracking processes in a virtual machine environment. In: ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference. (2006)
31. Xuan, C., Copeland, J., Beyah, R.: Toward revealing kernel malware behavior in virtual execution environments. In: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection. (2009)
32. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX Security Symposium. (2009)
33. Garfinkel, T., Adams, K., Warfield, A., Franklin, J.: Compatibility is not transparency: Vmm detection myths and realities. In: Proceedings of the 11th Workshop on Hot Topics in Operating Systems. (2007)
34. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In: USENIX Workshop on Offensive Technologies (WOOT). (2009)
35. Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E., Vigna, G.: Efficient detection of split personalities in malware. In: Network and Distributed System Security (NDSS). (2010)
36. Saxena, P., Sekar, R., Iyer, M.R., Puranik, V.: A practical technique for containment of untrusted plug-ins. Technical Report SECLAB08-01, Stony Brook University (2008)