

BACHELOR THESIS
COMPUTER SCIENCE



RADBOD UNIVERSITY

Identifying rootkit stealth strategies

THESIS BSc. COMPUTER SCIENCE

Author:
Egidius MYSLIWIEZ
s1000796

Supervisor:
dr. Veelasha MOONSAMY
EMAIL@VEELASHA.ORG

2020-06-20

Abstract

Rootkits provide a collection of tools allowing for low level actions on a system. With these capabilities, attackers can gain full access of a computer and even modify the way the core system itself operates. Thereby, using advanced abilities and persistence, attackers cause constantly increasing damages [1]. Current analysis focuses either on single rootkits as they are first detected or uses easy-to-spot components such as userspace programs or virtual machines¹.

This thesis explores an array of strategies used by rootkits to remain undetected and identifies the relative popularity of those techniques amongst a set of recent rootkit samples. A controlled environment based on physical machines was created to mimic a real-world scenario as closely as possible. A variety of tools have been built and adapted to stealthily collect data about changes the rootkit makes to programs, the registry or system components, both on disk and in memory. This data has been analyzed to provide statistics on the types of strategies used, which provides a model of up to date rootkits that can further be used to inform detection and prevention efforts by highlighting common vectors that need to be focused.

The diversity of rootkit strategies is fairly low, most group around a few proof of concept exploits. Attackers focus on stability, kernel changes have a higher chance to cause problems later on, so rootkit authors tend to just focus on DLLs and other less invasive strategies.

Keywords: malware; rootkit; windows; NT kernel; static; dynamic; stealth

Contributions:

- Catalog of techniques that can be used by rootkits to hide in the NT Kernel
- Quantification of usage of these techniques in current rootkits
- Implementation of tools allowing tests for and analysis of said techniques

¹see Appendix D

Contents

1	Introduction	4
1.1	Outline	5
1.2	Scope of thesis	7
2	Background	8
2.1	Rootkits	8
2.2	Rootkit evasion strategies	8
2.2.1	Permission based	8
2.2.2	Direct code modification	9
2.2.3	Direct data modification	9
2.2.3.1	Direct kernel object manipulation	10
2.2.4	Hooks	10
2.2.4.1	Import Address Table (IAT)	10
2.2.4.2	Export Address Table (EAT)	11
2.2.4.3	Interrupt Descriptor Table (IDT)	12
2.2.4.4	System Service Dispatch Table (SSDT)	12
2.2.4.5	SYSENTER_EIP	12
2.2.4.6	I/O Request Packet (IRP)	13
2.2.4.7	Hotpatching (Inline)	13
2.3	Rootkit evasion vectors	14
2.3.1	Task manager/Process listing	14
2.3.2	Resource listings (CPU)	16
2.3.3	File system	17
2.3.4	Registry	20
3	Related Work	22
4	Methodology	25
4.1	Dataset collection	25
4.2	Filtering for rootkits	25
4.3	Identifying vectors to analyze	26
4.4	Experiment setup	27
4.5	Executing rootkits	27
4.6	Collecting data	28
4.7	Quantification of data	30
5	Evaluation	31
5.1	Wapomi	31
5.2	Winnti	31
5.3	Zusy	31
5.4	Mimikatz	32
5.5	MSILPerseus	32
5.6	Bladabindi	32
5.7	Remaining samples	33
6	Discussion	34
7	Future Work	36
8	Conclusion	37

A List of Categories obtained from VirusTotal	38
B List of Samples	40
C Unclassified malware evaluation	42
D The case against VMs	45
E Kernel drivers and their use as a selection criteria for rootkits	46

1 Introduction

Computers have permeated society to the point that it is impossible to imagine the current world without them. From smart light bulbs to mobile phones and personal computers, all the way up to large data centers, they perform instructions and process data that impact and control parts of our lives. Computers perform a crucial role, be it in the form of managing the stock market, scientific equipment or private companies. These wide capabilities attract attention. Malicious actors try to make computers execute instructions that benefit the actor; instructions that are often detrimental to the machines' original purpose. Software used to illegally gain control over assets and resources is called malware. Attacks of this kind are increasing [2] and it is expected that cybercrime will cause global damages exceeding \$6 trillion by 2021 [3].

Different types of malware exist to fulfill differing goals. Some, such as ransomware encrypt important files and trade capital for their availability. Other kinds follow long term strategies: Instead of immediately profiting off of a target, they produce a larger benefit by compromising the system for a longer period of time. Cryptominers, for instance, are a type of malware that use processing resources to generate cryptocurrencies, which in turn can be exchanged for fiat value. The longer they are able to stay on a machine, the more computing hours they are able to dedicate to the task. However, an increase in CPU utilization also causes the system to become less responsive and heat up, forcing authors of cryptominers to walk a tight rope between higher returns in the short term and lower returns accumulating over a longer time.

Another family of malware, so-called Advanced Persistent Threats[4] is increasing in popularity[5]. This type of malware does not seek profit directly. Instead, it lies dormant and gathers personal data, important company assets and intel, which are quietly extracted and sent to the attacker. Long term data is much more valuable than a simple one time snapshot, so it is critical to make the machine and its owners believe that nothing has occurred, such that the malware will not be found and removed.

Although some certainly can be, most sophisticated malware is not monolithic. Like an animal, malware can be subdivided into distinct parts that serve specific functions, such as legs that carry it to its destination or claws that cause destructive damages. In the same manner, rootkits are not a type of malware in itself. Similar to camouflage, they are an optional component used by a variety of malware to evade detection.

The crucial difference is that camouflage acts passively while rootkits hide using active methods: like a parasite they burrow deep into the system and cause the detection organs to go blind. They do this by modifying the kernel, the central part of an operating system used to control the various tasks the machine needs to perform. Using these high capabilities, rootkits are on an equal footing with the operating system itself and can change the way the malicious code that accompanies them is handled.

Software can affect a computer in many ways. From fairly obvious results, such as an increase in CPU usage and subsequently heat up to subtle changes in the way external devices are handled or the way the operating system stores information about its processes, different measures can be used to see whether a piece of software is running. As a consequence, rootkits can not flip a proverbial hide-switch. They must choose which aspects are regarded as important for detection, whether it is worth to counteract them and how to do so if necessary. Given that there are many ways to solve a problem and the fact that rootkits need to solve a multitude of them, there is at least a theoretical potential for many strategies to be used, both in a single rootkit and when comparing across their populace.

Unfortunately, it is difficult to protect against rootkit infections. The differing strategies they use need to be remedied with tailored approaches, many of which come with drawbacks and performance penalties. It is uneconomical to protect a house from threats

that do not exist or are highly unlikely. In the same manner, a computer becomes less useful the more such safety systems are implemented in it. While in-depth analyses of individual rootkits exist and many such stealth techniques are described in the literature, little is known about their relative popularity. Without proper information about the probability and severity of threats, risk management has a hard time making suitable decisions. Should a specific rootkit prevention approach come with, for example, a detriment to performance of around ten percent, but for various reasons no rootkits exploit this particular weakness, it might be better to distribute resources elsewhere. If the opposite is true and it turns out that most rootkits use a very similar set of techniques, a clear motivation for improved remedies and further focus on them has been created.

This thesis will try to provide such information. Briefly summarized, the questions leading this thesis are as follows:

1. What are the detection methods a system uses to find malware?
2. How do rootkits circumvent them?
3. Which rootkits are most prevalent in the wild?
4. How can a certain detection method be identified in a rootkit attack?
5. How popular are these attacks in relation to each other?

To answer these questions, a selection of recently circulating rootkits has been executed on physical machines to closely mimic a real world infection. A number of tests have been created to identify certain strategies, which are subsequently quantified, as will be explained in the following paragraphs.

1.1 Outline of thesis

The work done for this thesis, as shown in Figure 1, was conducted as follows: A number of rootkit samples were obtained from multiple sources (Section 4.1). None of the files used were older than 3 years, most were much younger. The majority of samples pay no regard to the kernel, so a method for separating rootkits from other types of malware was devised (Section 4.2). Related work has been consulted to find reported stealth vectors (Section 4.3). Additionally, public, non-academic and my own analysis was used to find additional noteworthy characteristics. For each of the possible vectors, a test needed to be constructed in order to dynamically determine whether the vector in question is used by the respective sample (Section 4.4). Due to the suspicious nature of rootkits, low-impact kernel drivers had to be used. These tests were used in close to real world conditions on physical machines (Section 4.5) and corresponding data was collected (Section 4.6). Section 5 presents these findings.

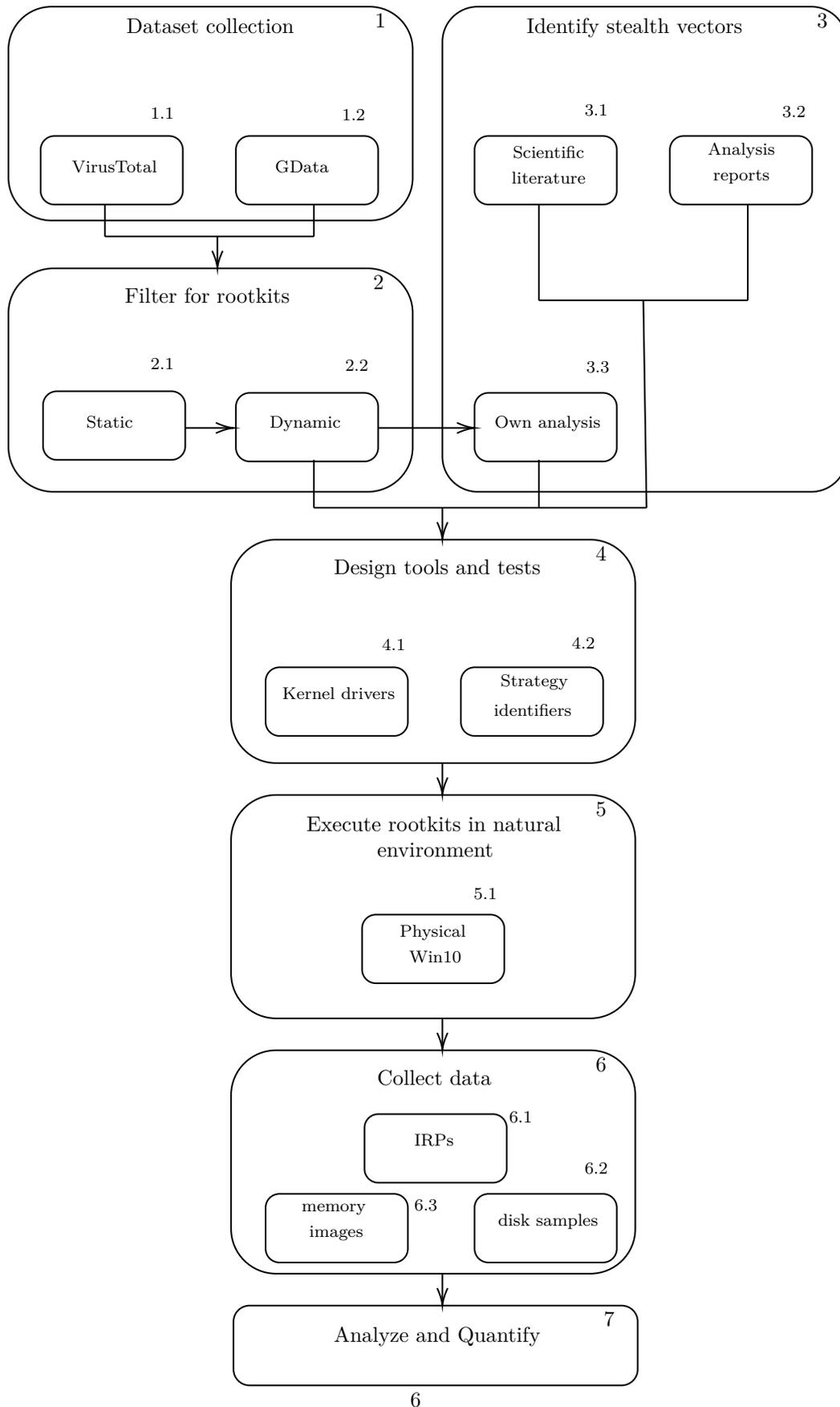


Figure 1: Research structure

This thesis will be organized in the following manner: Section 2 will cover preliminary information needed to follow the methodology. Section 3 introduces prior research done in the field and provides a context for the contributions of this thesis in relation to similar works. Section 4 explains the setup used and describes each step of this research. Section 5 presents and discusses the results obtained from the Methodology. Section 7 discusses optional additions to this research and calls for further research to be conducted to enhance or utilize these results. Finally, Section 8 ends the thesis by briefly summarizing its findings.

1.2 Scope of thesis

To prevent potential confusion, we will briefly mention some aspects this thesis has not set as a goal. Although identification of rootkits is conducted during the filtering phase 4.2, these methods operate under the assumption that a well-known system with a ground-truth good state, namely directly after a fresh installation, has been infected by a piece of malware likely to be a rootkit. The context dynamic filtering needs causes it to very likely be unsuitable to detect rootkits in real-world conditions, for example on a corporation's infected server or a personal machine. Other tools exist for such purposes, both academic, e.g. "Raide: Rootkit analysis identification elimination"[6] and commercial e.g. *GMER*[7] (incorporated into *Avast*[8], also available as stand-alone free of charge) and others[9]–[11].

Further, this work does not serve a directly defensive purpose. Even if these tools were to be used against their designations on a real-world system, they could neither prevent nor repair an infection. Instead, this thesis tries to supply information to aid these efforts. Because of this, directly malicious functionality is disregarded; the focus lies only on rootkit functionality used for stealth. Both of the above named tools are capable of removing a rootkit to some extent. Note, however, that a full reinstall should be the preferred option, as no operation within a compromised system can guarantee its correctness.

Information about executing, active rootkits is collected in a controlled environment relying as little on the infected operating system as possible. While antivirus products could potentially adapt kernel drivers or other tools utilized here, the usage of below-operating-system capabilities would complicate usage for the purpose of collecting in-system information about rootkits.

Strategies a rootkit uses to hide are dependent on the environment the rootkit occupies. This work focuses on rootkits within the Windows New Technologies (NT) Kernel². Kernel rootkits targeting different operating systems, or non-kernel rootkits such as usermode rootkits or rootkits attacking firmware, the hypervisor or the boot sector are out of scope.

²first used in Windows NT 3.1 in 1993, later in Windows 2000 in 2000, Windows XP in 2001 and every subsequent desktop version of the operating system

2 Background

2.1 Rootkits

Rootkits are commonly defined as programs that establish a persistent and undetectable presence on a computer. To some, this definition is problematic in two ways: Certain good programs need to hide themselves from other components of the operating system such that they may monitor or inhibit other malicious programs or behaviors without the latter noticing their presence. Additionally, many of these programs make it difficult to subvert or uninstall them, either by protecting their components in memory and on disk or by frequently checking their integrity and replacing them if needed, just like a rootkit would do. Examples of such software include antivirus engines or Digital rights management (DRM) protectors³. If any tool used to hide code execution is called a rootkit, such programs would be said to be rootkits or have rootkit components, while proponents of an alternative definition argue that rootkits need to be inherently malicious.

Second, so-called “user level rootkits” are programs that achieve the above goals of persistence and undetectability, although to a lesser degree, without having to resort to root privileges. Such software might for example load a malicious DLL into the memory space of a user process, such as `explorer.exe`, in order to constantly run in the background⁴. Generally, such programs are less sophisticated than their root level counterparts, as they have little protection against scans that run at root level. Still, some manage to evade detection, mostly due to novelty in their methods.

Within this thesis, a restrictive view of the word rootkit is chosen: Rootkits will be defined as programs that gain root level access to the operating system in order to compromise the system for a longer term in a way the user would consider malicious, thereby including beneficial programs but excluding user level rootkits.

2.2 Rootkit evasion strategies

Since rootkits are oftentimes malicious or unwanted, they need to hide their footprint from the user, antivirus engine and malware-detecting components of the operating system. Many strategies exist to achieve this goal. They can be grouped into the following categories:

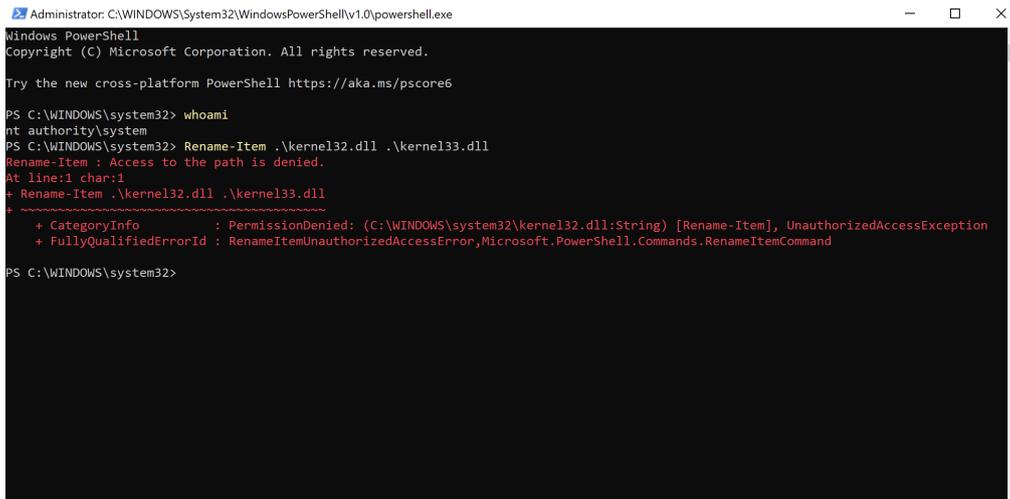
2.2.1 Permission based

A rather simple but quick and proven way for malware to deny access to files are Windows access control lists (ACL). Each file within the windows file-system has an owner who is able to modify the ACL and a set of rights associated with a number of users. Using these ACLs, access rights such as read/write/execute/modify ACL/etc can be allowed or forbidden for any given file on a per user basis. Windows uses this feature to protect critical system files, such as those present in `System32`. These kinds of files are owned by a user called `TrustedInstaller` and may only be modified exclusively by this entity, meaning that even system level users such as antivirus engines are not able to modify such a file, as can be seen in Fig. 2. Malware can use the same strategy to prevent access to its files. Note however that this method of protecting files is very explicit: the files are not hidden, they are simply inaccessible. While this may deter some

³often used to inhibit sharing of digital media, such as music, video and games. Sony famously used one such rootkit to protect its music CDs[12]

⁴`explorer.exe` is constantly being executed and automatically restarted in case of a crash by `winload`. This makes it easy to inject a dll when `explorer` is first executed, kill the process and have it restart with the malicious DLL.

(automated) malware detection, such a protected file in an unusual place will certainly attract the attention of any other sophisticated detection effort. Accessing such a file is quite trivial. In most cases, an administrator can simply claim ownership of the file from within the file explorer context menu. From there, any changes to the access control list, such as enabling read and write permissions can be made. Alternatively, any API call (those in kernel level) and any file system driver below the stage implementing the access control check are not affected by such restricted permissions.



```
Administrator: C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\WINDOWS\system32> whoami
nt authority\system
PS C:\WINDOWS\system32> Rename-Item .\kernel32.dll .\kernel33.dll
Rename-Item : Access to the path is denied.
At line:1 char:1
+ Rename-Item .\kernel32.dll .\kernel33.dll
+ ~~~~~
+ CategoryInfo          : PermissionDenied: (C:\WINDOWS\system32\kernel32.dll:String) [Rename-Item], UnauthorizedAccessException
+ FullyQualifiedErrorId : RenameItemUnauthorizedAccessError,Microsoft.PowerShell.Commands.RenameItemCommand

PS C:\WINDOWS\system32>
```

Figure 2: The failed attempt to rename `kernel32.dll` with system privileges

2.2.2 Direct code modification

This section excludes hooks and hotpatching, as they will be discussed in their own sections.

Given that all code running on a system needs to be stored in memory before being executed, rootkits may overwrite assembly instructions by accessing other processes memory regions. In contrast to hooks and hotpatching, the goal is not to make small changes to redirect control flow, but to change code in place. The focus of such a technique may be simply to change a return value or invert a conditional jump; malware may even change whole subroutines or functions. For example, a function used to enforce a policy may be overwritten by one supplied by the attacker which simply grants validation of all inputs regardless of policy. While simple, especially when used on files on disk, this method is rarely used, given that issues regarding padding, available space, etc. arise. A further complication is self-modifying code, such as encryption routines, that may render the attacker-supplied code unusable.

2.2.3 Direct data modification

Similar to the above section, malware may modify data stored on disk or in memory. Since this method does not suffer from many of the problems of the above method (given that the amount of data that needs to be written to a file within a struct is more forgiving to changes, such as moving a null-byte, or already has ample length for the required data type), it is more common in practice. A popular example is a program called **CheatEngine**, which scans memory regions of games for bytes storing information about the game, such as health, remaining ammunition, experience points, selected slots,

etc. In the same way, malware can scan memory for data structures of certain processes (such as a to-scan list within an antivirus engine) and modify them to exclude itself or redirect attention to other regions.

2.2.3.1 Direct kernel object manipulation

No clear boundaries between memory of processes in kernel mode exist, so an attack is rather easy. Given that the operating system stores a large amount of data on a system (such as running processes, memory pages, etc.) within kernel mode objects, a malicious kernel driver modify said structures to hide the malware. While such modifications are difficult to detect, given that the objects are still valid, drastic changes may cause the operating system to become unstable or even cause a Blue Screen error message, resulting in a system reboot.

2.2.4 Hooks

In contrast to direct code modification, the purpose of hooks is to patch code in order to get notified on certain events or redirect code flow, often to malware supplied code. Depending on which kind of function pointer is overwritten or which method of redirecting control flow is used, hooks may be categorized into the following sections.

2.2.4.1 Import Address Table (IAT)

Most code relies on external libraries and third party functions to run. Given that these dependencies often overlap between different programs, it would be wasteful for every program to contain its own copy of the needed external functionality. Because of this, most programs are not compiled with their dependencies but instead dynamically linked to them at runtime by the loader. Since the location of DLLs in memory changes frequently, such programs contain a table of all required external functions together with a pointer to them, which will be supplied to the program at run time. The table is of the form `jmp dword ptr ds:[addr]`, where `addr` is the memory address of the external function. Malware may overwrite an entry in this table to force the program to use a different, malicious function from the one it desires. For example, a malware scanner likely imports `ZwReadFile` in order to read and then scan files that may contain malware. By modifying the entry of this function within the import address table, malware is able to fully control the input given to the malware scanner via the malicious `ZwReadFile`-clone, which may for example only return random data, empty files or predictable and uninteresting data.

```

IDA View-A
Hex View-1
Structures
Enums
Imports
Exports

; .idata:0000000180078FC0 ; DATA XREF: ...
; .idata:0000000180078FC3 ; void __stdcall RtlTimeToTimeFields(PLARGE_INTEGER Time, PTIME_FIELDS TimeFields)
; .idata:0000000180078FC8 ; extrn RtlTimeToTimeFields:qword
; .idata:0000000180078FCB ; CODE XREF: sub_180007A8B+5A1p
; .idata:0000000180078FD0 ; FILETIME DosDate; FILETIME DosTime; FILETIME DosDateToTime; FILETIME DosTimeToTime;
; .idata:0000000180078FD5 ; BOOLEAN __stdcall RtlTimeFieldsToTime(PTIME_FIELDS TimeFields, PLARGE_INTEGER Time)
; .idata:0000000180078FD8 ; extrn RtlTimeFieldsToTime:qword
; .idata:0000000180078FDB ; CODE XREF: sub_180007A8B+481p
; .idata:0000000180078FDE ; DosDateToTimeOfFileTime+841p ...
; .idata:0000000180078FE0 ; extrn __imp_RtlUnHandledExceptionFilter:qword
; .idata:0000000180078FE3 ; DATA XREF: RtlUnHandledExceptionFilterFr
; .idata:0000000180078FE6 ; extrn __imp_NtTerminateProcess:qword
; .idata:0000000180078FE9 ; DATA XREF: NtTerminateProcessFr
; .idata:0000000180078FEF ; int __cdecl wcsncmp(const wchar_t *Str1, const wchar_t *Str2, size_t MaxCount)
; .idata:0000000180078FF0 ; extrn __imp_wcsncmp:qword
; .idata:0000000180078FF3 ; CODE XREF: sub_180016444+1E1p
; .idata:0000000180078FF6 ; wchar_t *__cdecl wcsncpy(wchar_t *Dest, const wchar_t *Source, size_t Count)
; .idata:0000000180078FF9 ; extrn wcsncpy:qword
; .idata:0000000180078FFC ; CODE XREF: sub_18003D938+D2C7p
; .idata:0000000180078FFD ; sub_18003D938+D2E7p ...
; .idata:0000000180078FF8 ; extrn LdrFindResourceEx_U:qword
; .idata:0000000180078FFB ; CODE XREF: sub_180041A30+4627p
; .idata:0000000180079000 ; DATA XREF: sub_180041A30+4627p
; .idata:0000000180079003 ; extrn RtlReadThreadProfilingData:qword
; .idata:0000000180079006 ; DATA XREF: ReadThreadProfilingData+41p
; .idata:0000000180079009 ; extrn RtlQueryThreadProfiling:qword
; .idata:000000018007900C ; CODE XREF: QueryThreadProfiling+41p
; .idata:000000018007900F ; DATA XREF: QueryThreadProfiling+41p
; .idata:0000000180079012 ; extrn RtlDisableThreadProfiling:qword
; .idata:0000000180079015 ; CODE XREF: DisableThreadProfiling+41p
; .idata:0000000180079018 ; DATA XREF: DisableThreadProfiling+41p
; .idata:000000018007901B ; extrn __imp_RtlInStatusToDosErrorNoTeb:qword
; .idata:000000018007901E ; DATA XREF: CheckElevation+11A1p ...
; .idata:0000000180079021 ; extrn RtlEnableThreadProfiling:qword
; .idata:0000000180079024 ; CODE XREF: EnableThreadProfiling+41p
; .idata:0000000180079027 ; DATA XREF: EnableThreadProfiling+41p
; .idata:000000018007902A ; extrn NtMapUserPhysicalPagesScatter:qword
; .idata:000000018007902D ; CODE XREF: MapUserPhysicalPagesScatter+41p
; .idata:0000000180079028 ; DATA XREF: MapUserPhysicalPagesScatter+41p
; .idata:000000018007902B ; extrn RtlDecodeSystemPointer:qword
; .idata:0000000180079030 ;
00077BC8.0000000180078FC3:.idata.RtlTimeToTimeFields (Synchronized with Hex View-1)

```

Figure 3: The import table inside the kernel32.dll binary

Note that values in the Import Address Table (IAT) can be double-checked by calling `GetProcAddress`, a function which queries Export Address Table (EAT) entries for the required address, but which of course might also be compromised. Given that no EAT hooks are present, a custom implementation of `GetProcAddress` may be considered adequately secure.

2.2.4.2 Export Address Table (EAT)

Similar to the import address table, programs contain an export address table containing pointers to functions they provide. This information is used by linkers in order to populate the IATs of the programs wanting to import said functions. Malware may overwrite these pointers to much the same effect as above. The main difference is that an IAT hook is able to target a single function import within a single program, while an EAT hook will affect all programs (that are loaded and linked after the hook is in place) importing a specific function. According to MalwareTech [13], EAT hooks are rare.

Name	Address	Ordinal
CloseDialogInfo	00000018003080B0	136
ClosePrivateNamespace	00000018001F0F00	137
CloseProfileUserMapping	00000018001F2200	138
CloseThreadContext	0000001800202E00	139
CloseState	000000180030C4B8	140
CloseThreadPool	0000001800305A5E	141
CloseThreadPoolCleanupGroup	0000001800309C0E	142
CloseThreadPoolCleanupGroupMembers	0000001800309C20	143
CloseThreadPool	0000001800309E12	144
CloseThreadPoolTimer	000000180030D443	145
CloseThreadPoolWaiter	000000180030956C	146
CloseThreadPoolWork	0000001800309294	147
CmDkBatchOfLocation	00000018001E6800	148
CommConfigDialogA	0000001800373A10	149
CommConfigDialogW	0000001800373A40	150
CompareCalendarDates	00000018003495F0	151
CompareFileTime	0000001800220A00	152
CompareStringA	00000018001CC100	153
CompareStringEx	0000001800162C00	154
CompareStringOrdinal	0000001800164440	155
CompareStringIf	0000001800164790	156
ConvertNamePipe	00000018001EF800	157
ConsoleMenuControl	000000180060E400	158
ConvertDefaultLocale	00000018003575A0	159
ConvertCalendarTimeToSystemTime	000000180043A4A0	160
ConvertDefaultLocale	00000018003575A0	161
ConvertFiberToThread	0000001800227100	162
ConvertLSIDayOfWeekToWin32DayOfWeek	00000018004B8900	163
ConvertSystemTimeToCalendarTime	0000001800202100	164
ConvertThreadToFiber	0000001800227200	165
ConvertThreadToFiberEx	0000001800227230	166
CopyContext	0000001800357B00	167
CopyFile2	0000001800357C00	168
CopyFileA	0000001800595E10	169
CopyFileExA	0000001800595E40	170
CopyFileIf	0000001800202D30	171
CopyFileTransactedA	0000001800595F00	172
CopyFileTransactedW	00000018005A4000	173
CopyFileW	0000001800227000	174
CopyLFile	000000180032E900	175
CreateAsCvW	00000018001F0A90	176
CreateAsCvW	00000018001F0A00	177
CreateAsCvWWorker	000000180060E900	178
CreateBoundaryDescriptorA	0000001800595900	179
CreateBoundaryDescriptorW	00000018001EECB0	180

Figure 4: The export table inside the `kernel32.dll` binary, as rendered by IDA PRO

2.2.4.3 Interrupt Descriptor Table (IDT)

In addition to regular codeflow, software interrupts can occur at any time. Should a certain event, like input from a peripheral device, a request to shut down the computer or a software exception arise, the current execution state is suspended. A table in the kernel, called the Interrupt Descriptor Table, contains addresses of interrupt routines. Should, for example, interrupt 13h be triggered, the operating system jumps to the address stored in the 13th (or 14th, if counting ordinals starts at 1) entry of the IDT, a routine used to talk to drives. A rootkit can modify the addresses contained in this table to execute every time one of these events occur.

2.2.4.4 System Service Dispatch Table (SSDT)

The System Service Dispatch Table (SSDT) is a table of pointers for various `Zw/Nt` functions, that are callable from usermode. A malicious application can replace pointers in the SSDT with pointers to its own code.

All pointers in the SSDT should point to code within `ntoskrnl`, if any pointer is pointing outside of `ntoskrnl` it is likely hooked. It is possible a rootkit could modify `ntoskrnl.exe` (or one of the related modules) in memory and slip some code into an empty space, in which case the pointer would still point to within `ntoskrnl`. Functions starting with `Zw` are intercepted by SSDT hooks, while those beginning with `Nt` are not, therefore an application should be able to detect SSDT hooks by comparing `Nt*` function addresses with the equivalent pointer in the SSDT.

A simple way to bypass SSDT hooks would be by calling only `Nt*` functions instead of the `Zw*` equivalent. It is also possible to find the original SSDT by loading `ntoskrnl.exe` (this can be done easily with `LoadLibraryEx` in usermode) then finding the export `KeServiceDescriptorTable` and using it to calculate the offset of `KiServiceTable` within the disk image (Usermode applications can use `NtQuerySystemInformation` to get the kernel base address), a kernel driver is required to replace the SSDT.

2.2.4.5 `SYSENTER_EIP`

`SYSENTER_EIP` is a variable containing the address of a legacy x86 function (usually `KiFastCallEntry`) invoked whenever a context switch from user to kernel mode is per-

formed. Hooking this function either with an inline hook or by overwriting its function pointer enables malware to be notified of any such event. Given that user mode applications do not have adequate permissions to write their own context switching routine and kernel mode drivers have no need for context switching, this kind of hook can not be circumvented. It is however possible to implement a kernel mode driver that finds the real pointer value and restores `SYSENTER_EIP` accordingly.

2.2.4.6 I/O Request Packet (IRP)

Each registered driver exposes a driver object within the kernel namespace. Among others, this object contains a table of 28 function pointers, which may be uninitialized/NULL that point to functions implementing interaction with the potential hardware/kernel object the driver is controlling, such as reading or writing data. As no boundaries exist in kernel memory space, drivers can overwrite each others IO request function pointers. Unless the affected driver restores its own value or another driver has stored the correct values before the hook occurred, it is not possible to restore such a hook. A hook may be detected by realizing that the pointer points to an address outside of the respective kernel drivers address space, but this does not protect from injecting a malicious routine into a driver or switching functions within it. For circumvention, the next lower driver in the stack may be used, which may also be compromised. Similar to Section 2.3.3.

2.2.4.7 Hotpatching (Inline)

In contrast to previous hooks that changed stored pointers to functions, hotpatching modifies the first few bytes of code to instruct the function to jump to another block of code. Often, this behavior is not malicious, it allows the operating system to replace certain functions with others that fit better into a given context without having to update the relevant address in every importing program. Because of this, DLLs such as `kernel32` were often compiled with hotpatching support in x86 versions of windows. This support is provided by a two byte no-op instruction that may be overwritten by a jump statement, also occupying two bytes, that may easily be discarded. The CPP compiler has a flag to enable such easy hotpatching [14].

```
.text:7DD712FC
.text:7DD712FC      mov     edi, edi
.text:7DD712FE      push   ebp
.text:7DD712FF      mov     ebp, esp
.text:7DD71301      push   0
.text:7DD71303      push   [ebp+lpNumberOfCharsWritten]
.text:7DD71306      push   [ebp+nNumberOfCharsToWrite]
.text:7DD71309      push   [ebp+lpBuffer]
.text:7DD7130C      push   [ebp+hConsoleOutput]
.text:7DD7130F      call   sub_7DD712D5
.text:7DD71314      test   eax, eax
.text:7DD71316      jl     loc_7DD71316
.text:7DD7131C      xor     eax, eax
.text:7DD7131E      inc    eax
.text:7DD7131F
.text:7DD7131F loc_7DD7131F:      ; CODE XREF: WriteConsoleA+3F39E↓j
.text:7DD7131F      pop    ebp
.text:7DD71320      retn   14h
.text:7DD71320 WriteConsoleA     endp
.text:7DD71320
.text:7DD71320 ; -----
```

Figure 5: A two byte no-op

Note that two single byte no-ops might cause complications due to pipelining. x64 versions of windows do not use the equivalent instruction `mov rdi, rdi`⁵. In cases where such hotpatching support is not provided, the bytes that need to be overwritten would first have to be read and stored such that they can be replaced later. By hooking the function at the beginning of execution, no half executed code can cause weird behavior and the redirected-to function is able to read arguments from stack and registers. The best way to protect against inline hooking and other forms of direct code modification of DLLs is to obtain a legitimate copy of said DLL and compare both versions. Windbg offers a feature that downloads a signed and authenticated DLL from official Microsoft sources and compares it with a DLL loaded in memory.

```

0: kd> !chkimg kernel32.dll
1 error : kernel32.dll (7ff9fbbe2160)
0: kd> U 7ff9fbbe2160
KERNEL32!BeepImplementation:
00007ff9`fbbe2160 cc                int     3

```

Figure 6: The `chkimg` function of Windbg

Read more about this in the section about memory paging.

2.3 Rootkit evasion vectors

Being a program with associated data and code, which is eventually executed, a rootkit will never be as invisible as a process that does not exist in the first place. This section will introduce the most relevant traces a rootkit (or any program) will leave within an operating system, how those traces are manifested and what a rootkit can or cannot do in order to hide this kind of trace. The last point will be elaborated in much more detail in the preceding Section 2.2.

2.3.1 Task manager/Process listing

The Windows Task Managers “Processes” and “Details” tabs provide a list of all processes running on a computer, while the tabs “App History” and “Services” provide analogous listings for apps and services, respectively. To avoid unnecessary duplications, only process listings will be focused on, as app and service listings are generated in a very similar fashion.

⁵`mov edi, edi` would zero out the upper half of `rdi` on 64bit machines

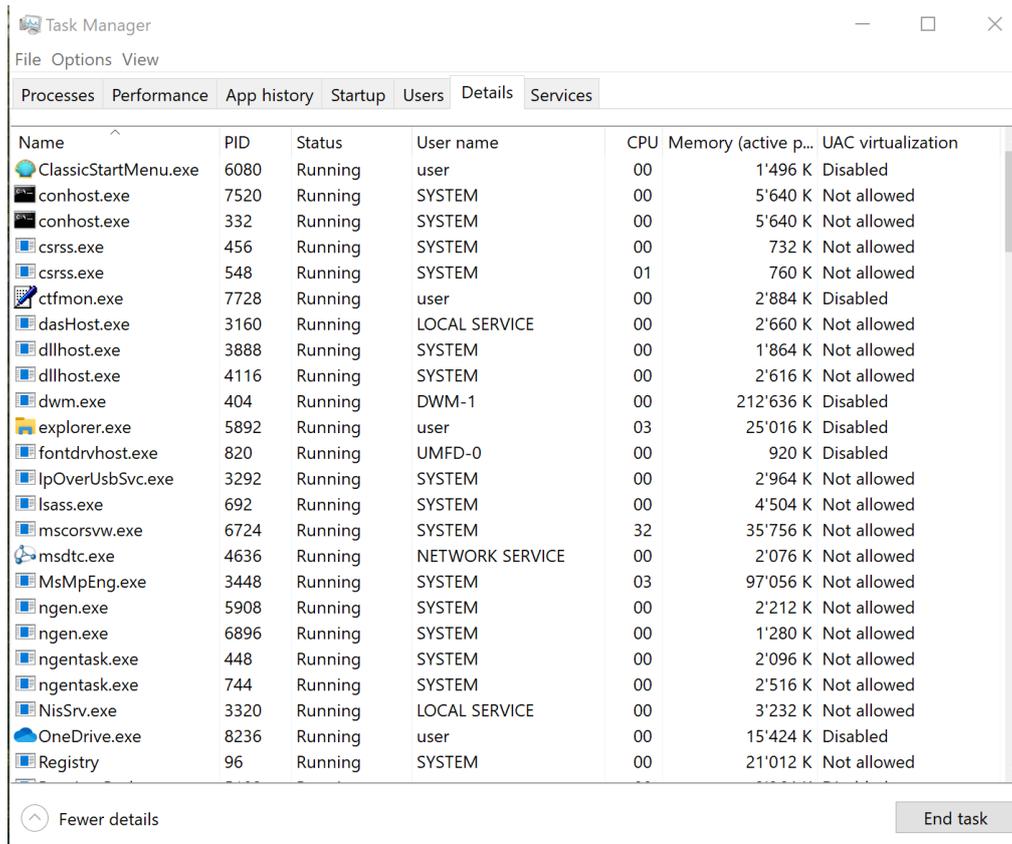


Figure 7: The Details tab of the Windows 10 Task Manager

Within the kernel, a data structure called `PsActiveProcessList` is used to contain information, such as name, PID, permissions⁶, threads, etc, about all running processes on a system. Using the function `IoGetCurrentProcess`, a pointer to a `EPROCESS` block, which contains information about one of the processes, is returned. Such a struct contains pointers to both the next as well as the previous entry of the dually linked process list, as visualized in Fig. 8. Using two simple modifications, a rootkit could change the pointers of the block pre and succeeding it to skip its own block, as no (relevant) garbage collection utility in the kernel exists to deallocate “clean up” this “removed” entry. Interestingly, the Windows Scheduling Algorithm does not solely depend upon traversing this structure, allowing even hidden entries to be allocated execution time. However, like all DKOMs, this modification is rather unstable, especially when the operating system fails to access a hidden processes block (i.e. to add/remove threads or change permissions).

⁶An attack based on this structure to elevate a processes privilege level is possible, but requires the attacker to already have kernel level access, so it is rather uninteresting

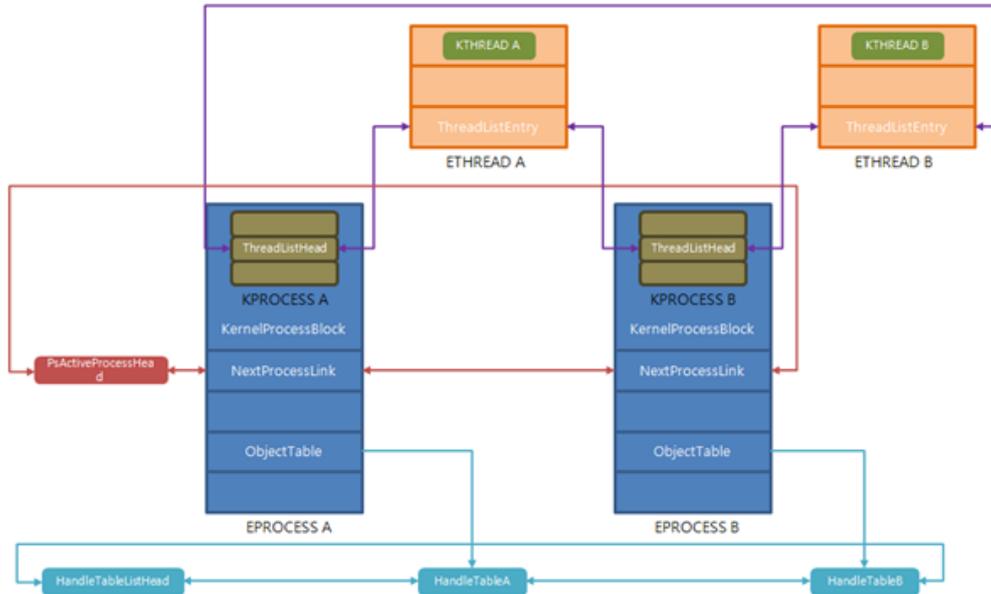


Figure 8: The PsActiveProcessList datastructure [15].

Alternatively, if protection from only user level detection is required, API functions used to query processes may be hooked, such as the `EnumProcesses` function in `kernel32.dll` [16].

2.3.2 Resource listings (CPU)

Even if a process is disguised using the above method, any running process will need to use some CPU time. The Linux Operating system uses a program called `time` to display the amount of time⁷ used by a process in user and kernel mode. Windows programs may handle similar functionality using API calls enumerated on this page: <https://docs.microsoft.com/en-us/windows/win32/sysinfo/time-functions>⁸. Since the Windows Task Manager is not open-source, details about the exact way it implements resource listings remain unknown, at least for the scope of this thesis. To make a few speculations, `taskmgr.exe` imports library functions such as `GetProcessorSystemCycleTime`, which returns the time each cpu spend on deferred procedure calls (DPCs) and interrupt service routines (ISRs)[17], so it may just provide a front end for deeper API routines⁹. In principle, since it imports scheduled timers[19], it might implement more sophisticated methods discussed below. Still, it is much more likely that timers are simply used for regular updates of the GUI.

Of course, API functions can easily be modified or hooked, so the information provided by them might not be reliable on an infected system. A more reliable against attacks but much less precise way of measuring system usage can be achieved by programs determining the values themselves solely based on their own execution. To give a trivial example, an application might just consume all available CPU usage itself for a brief period of time and compare the number of instructions it was able to execute every time-period. To be slightly stealthier, imagine a server with m execution threads. Malware

⁷As well as other quantities such as IO Accesses, etc.

⁸lower half

⁹Another relevant function is `GetSystemTimes`[18]

running on it may create a timestamps for every n higher-level, but fairly lightweight, instructions carried out on a thread. If the spacing between two timestamps suddenly increases, a heavy application has been started on that thread, which could possibly be a malware scanner and it may be wise for the malware to lie low. To not consume too many resources itself, an application might use a `sleep` call instead of executing instructions, the length of which will increase on overloaded systems. An improvement of this overly simplistic and fairly obvious technique can be made by hooking/modifying other programs, such as `explorer.exe` to see how regularly a certain repeating function, such as redrawing the screen, is being executed. It may also look at related data, such as CPU temperature (although that might be unreliable as well) or times of IO operations.

Address	Ordinal	Name	Library
00000001400BA288		GetComputerNameExW	api-ms-win-core-sysinfo-l1-1-0
00000001400BA290		GetLocalTime	api-ms-win-core-sysinfo-l1-1-0
00000001400BA298		GetSystemTime	api-ms-win-core-sysinfo-l1-1-0
00000001400BA2A0		GetTickCount	api-ms-win-core-sysinfo-l1-1-0
00000001400BA2A8		GetVersionExW	api-ms-win-core-sysinfo-l1-1-0
00000001400BA2B0		GetLogicalProcessorInformationEx	api-ms-win-core-sysinfo-l1-1-0
00000001400BA2B8		GetTickCount64	api-ms-win-core-sysinfo-l1-1-0
00000001400BA2C8		GetSystemFirmwareTable	api-ms-win-core-sysinfo-l1-2-0
00000001400BA2D0		GetNativeSystemInfo	api-ms-win-core-sysinfo-l1-2-0
00000001400BA2E0		GetPhysicallyInstalledSystemMemory	api-ms-win-core-sysinfo-l1-2-1
00000001400BA2F0		GetProcessorSystemCycleTime	api-ms-win-core-sysinfo-l1-2-2
00000001400BA300		WaitForThreadpoolTimerCallbacks	api-ms-win-core-threadpool-l1-2-0
00000001400BA308		CloseThreadpoolTimer	api-ms-win-core-threadpool-l1-2-0
00000001400BA310		SetThreadpoolTimer	api-ms-win-core-threadpool-l1-2-0
00000001400BA318		CreateThreadpoolTimer	api-ms-win-core-threadpool-l1-2-0
00000001400BA328		QueueUserWorkItem	api-ms-win-core-threadpool-legacy-l1-1-0
00000001400BA330		CreateTimerQueueTimer	api-ms-win-core-threadpool-legacy-l1-1-0
00000001400BA338		DeleteTimerQueueTimer	api-ms-win-core-threadpool-legacy-l1-1-0

Figure 9: An excerpt of the Import Table of `taskmgr.exe`

2.3.3 File system

Depending on the kind of data the malware needs to access a filesystem for, two situations may be distinguished:

user/system data In this scenario, malware wants to tamper with data, which has not been created by the malware and instead belongs either to the infected user or operating system. First, consider write changes (such as overwrite, append, modify) to files created by the user, examples of which are conducted by ransomware when encrypting user files. Due to the nature of this kind of infection, the malware explicitly makes itself known (which it has to, if it wants to collect ransom) and any stealth is counterproductive. Because of this, this behavior is out of scope for this thesis.

The more interesting case concerns read access to user created files, for purposes such as personal information gathering, or write access to programs, e.g. to infect a web browser, and system files. In these cases, the malware needs to stay hidden to prevent the user or the users' antivirus from negating malicious changes or detecting the infection. On a basic level, this can be achieved using hooks to API functions such as `ZwCreateFile`, `ZwCloseHandle`, `ZwReadFile` and `ZwWriteFile`. Alternatively, the file may be given restrictive access permissions such that it can not be accessed except by the malware¹⁰. On a more in-depth level, Windows implements its filesystem using a filesystem driver of the respective kind (usually NTFS for mass and exFAT/FAT32 for flash storage). This driver may be interacted with by kernel-mode programs using the filesystem control device object it exposes to the kernel namespace. Every physical disk in the system is directly controlled via a driver responsible solely for the port/slot the

¹⁰Again, permission based approaches are not stealthy

drive is inserted into. This port driver itself is only capable of sending and receiving bits via said port and exposes an adapter device object for said port. This object is used by another so called disk class driver, which understands the protocol used to communicate with the given disk (SATA, SCSI, USB, etc) and is aware of intricacies of the drive (min request block size, partition table size, command sequences, etc). This driver exposes a disk device object, that may be used to raw access the drive using commands, instead of the unprocessed and drive-specific bit-sequences understood by the adapter device object. The driver one level above manages the partitions within a given disk by processing the partition table located at the very start of a drive. It exposes multiple objects, namely partition0, to provide full disk raw access (the difference to the disk device object one layer below being that partition0 begins after the partition table and its padding, and does not encompass the true whole drive) and partition1 to partitionN, which give raw access to the drive space occupied by the respective partition. The filesystem driver refers to these objects and creates volume device objects used by user mode programs to interact with the given volume. In certain cases, additional drivers used to hide parts of the disk are employed at various levels. An example of this on the filesystem drive layer would be a software RAID whose filesystem driver hides parity files. In other cases, whole partitions may be hidden (something similar to the Windows boot-up partition, which may not be interacted with by user mode programs¹¹). A hardware RAID card works on an even lower level than a port driver by modifying bit streams as they traverse between the drive and port driver.

In general, the deeper the level of the driver or API function that a rootkit hooks, the more detection methods it is protected against. Unfortunately for the malware, lower drivers force it to implement a lot of functionality itself, so it is assumed (by me) that no rootkit would attempt to hook anything below a disk class driver.

Windows uses a feature called FastIO, which caches frequently accessed files to system memory. In these cases, read and write operations are intercepted by the operating system and changed to equivalent operations on the memory buffer containing the file. This file is then later written to disk. Depending on the implementation of a disk interception routine, FastIO may be problematic to a rootkit. If a rootkit hooks API functions above the implementation of FastIO, such as `ZwReadFile` it can ignore this feature. If it instead hooks into the filesystem driver stack at a lower level, such as the disk class driver, the file on disk lags behind the memory buffer. Given that a file has to be read from disk before being cached, the driver may still choose what data to show. But given that FastIO may approve a write operation that the driver would not, inconsistencies between the buffer and the file arise and blocked mem-to-disk write requests may cause unwanted errors or even crashes or unexpected behavior in applications unaware of the underlying FastIO protocol who did not implement a backtracking procedure.

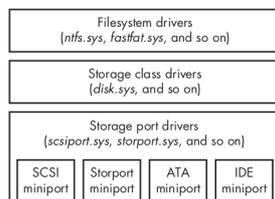


Figure 10: Storage device driver stack

¹¹Although Windows does not truly hide them using a partition driver

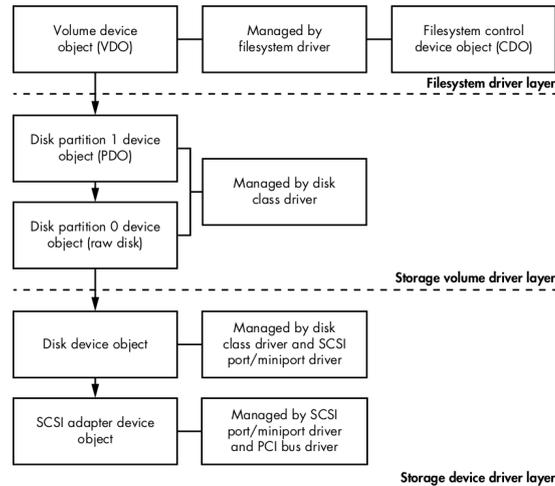


Figure 11: Example storage device driver stack based on SCSI

malware data In contrast to the scenario above, malware does not need to access files created by the operating system or the user. Instead, it desires to store its own code or data on the disk in a permanent fashion, be that to store accumulated data, configuration settings or to persist during a reboot. In this case, the malware does not necessarily need to use/implement the driver stack described above, since it does not require knowledge of partitions, volumes and filesystems in order to store some data. While being able to utilize these features is convenient, they need to either be implemented by the malware, which is time-consuming and error-prone, or the malware is forced to use already present higher level functions, which make it less stealthy. However, if there is no necessity to use higher level storage concepts, the rootkit is able to hook lower level drivers, gaining the benefit of staying hidden more easily, without much of the drawback of having to implement these higher drivers itself. One procedure, first used by a rootkit called TDL3 and later adopted by many more, is a so-called *Hidden filesystem*. When creating a single primary partition using the Windows Installer, usually not the entire remaining (non-occupied by partition table) disk is consumed but some space at the end, a few tens of kilobytes, are left unallocated for padding or performance purposes¹². Rootkits can utilize this space, usually at the end of the raw disk, to store their own data by writing to the disk itself. Depending on the way a rootkit chooses to implement its hidden filesystem, it may either just write to this unallocated section directly using a port driver, or it may create its own partition, or even register its own filesystem driver. The latter case has the benefit that malware is able to (re)use common Windows API functions for creating files, which eases malware development. In the first case, malware may simply leave its data as is, in the hope of it being in such an unusual location that no program would look for it. While this succeeds for user mode programs, it is completely exposed to a raw access operation on the relevant part of the hard drive. Because of this, many rootkits choose to not write to their space with raw operations,

¹²Depending on the type of drive used, unallocated zones can get even more interesting: Tape drives contain large unallocated sectors at the ends of the tape, which could technically be written to, but are left unallocated in software to attach the tape to its accessing mechanism. Some high performance drives leave their innermost regions unallocated, since access times on a spinning disk get much faster towards the edge of the drive. Many SSDs advise to only be filled up to a capacity of around 80%, to enable smarter caching and moving of data. While Windows does not honor this margin, it does indeed leave a small amount, but more than for hard drives of equal capacity, of space unallocated on them.

but instead use an actual filesystem implemented using higher level drivers. Oftentimes, such a filesystem has added encryption functionality implemented within drivers to deter raw disk scanning. In some cases, malware may choose to use already existing higher level drivers, which makes its code more reliable but still enables it to remain stealthy by obscuring the relevant device/partition object (e.g. using a random name such as `\Device\XXXXXXXXXXXXXXXXXX`, where X is a random digit). Additionally, it needs to hook other drivers and API functions to further hide the presence of the hidden filesystem, by returning a smaller disk size or intercepting raw read operations to the hidden filesystem part of the drive. Malware may also modify these existing drivers in such a way that they only access the hidden filesystem using a key passed to them within a parameter, e.g. a write operation of `(key|actual data)`, which gets intercepted and modified by the required driver.

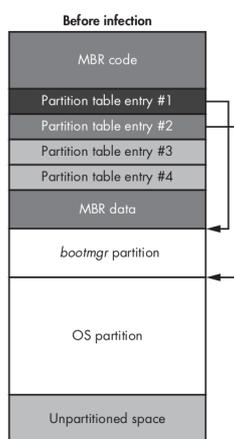


Figure 12: The contents of the primary drive before infection by a rootkit employing a hidden file system

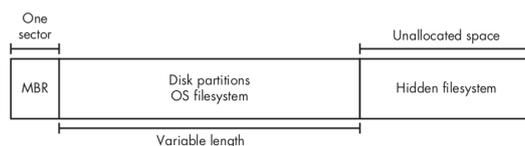


Figure 13: A high-level view of a disk infected by a hidden file system

2.3.4 Registry

Windows uses the registry to store information about DLLs loaded into any given program, so a rootkit which adds a malicious DLL into that list can easily be detected. That can be counteracted via real time DLL injection whenever a program is loaded by hooking a system call. Given that most rootkits have, as the name suggests, root level access to a system, many employ a kernel driver for additional capabilities. These kernel-mode drivers are stored within a key inside the registry to be loaded when the system starts and thus need to remain exposed inside the registry. The Windows API function used to list subkeys of any given key is called `ZwEnumerateKey`. By hooking this function, rootkits can hide their particular subkeys. Alternatively, if an antivirus engine does not rely on said system call, but is loaded after and stopped before the malware, the rootkit may remove its own registry keys as part of its startup routine

(since kernel drivers are loaded earlier, during boot) and re-add them during the system shutdown phase (e.g. by hooking `IoRegisterShutdownNotification`).

Yet another possibility: the kernel-mode function `CmRegisterCallbackEx` can be implemented by a driver and is called whenever an access operation (read or write) is about to be made by the registry. Being not only effective against basic discovery strategies (e.g. simple enumeration similar to `ZwEnumerateKey`) but also against direct access of the key via its full name by denying access to it. As discussed above, permission based approaches may only deter naive detection efforts and are certainly not stealthy. However, instead of failing the `CmRegisterCallbackEx` procedure to deny access, malware may instead forge an error stating that the requested key does not exist.

3 Related Work

General information about Rootkits A short but limited overview of rootkits, their history, abilities and usage can be obtained by referring to short summaries by Kovah [20]–[22], Thompson [23], Rankin [24], Malwarebytes [25] and Microsoft [26]. Ries [27] gives a detailed, although slightly out of date, overview of rootkits, their internal design and functioning as well as their prevention, detection and removal. Additionally, the article presents helpful software and proof-of-concept rootkits. Brendmo [28] gives an introduction into the internals of the Windows 10 kernel.

Books providing an exploration of rootkits and the kernel A technical hands-on guide to building a rootkit is presented by Høglund and Butler [29] in their book *Rootkits: Subverting the Windows kernel*. While their work is a great starting point for understanding the attacker’s point of view and gaining a complete view of a rootkit’s codebase, some approaches are unfortunately dated. *A guide to kernel exploitation* by Perla and Oldani [30] explores the internal workings of both the Linux and NT kernel from the perspective of an attacker. It is a valuable guide to both detecting the usage of and finding new stealth strategies of kernel rootkits. Blunden [31] provides a complete guide on rootkits. His book covers a high-level and societal view on rootkits, strategies for forensics, counter-forensics and counter-counter-forensics, explains the necessary tools, hardware and software systems and guides the reader through the functioning of a system from the boot process to loading a program. *The Rootkit arsenal: Escape and evasion in the dark corners of the system* covers design, stealth, forensic evasion and exploit strategies from the point of a malicious rootkit author. Matrosov, Rodionov, and Bratus [32] wrote the most recent book focusing on rootkits. While it only dedicates its starting chapters to kernel rootkits and then moves on to extensively cover bootkits, it provides detailed analyses of past boot- and rootkits, guides the reader through their analysis with hands-on explanations and real samples and gives them historic and design-related context. *Rootkits and bootkits: reversing modern malware and next generation threats* bridges the gap from strategies presented in older works to those employed by current day samples.

Works similar to this thesis The following sections are dedicated to four works regarded closest to what is proposed in this thesis and their differences to this work. They include a Masters thesis from 2011: “A comparative analysis of rootkit detection techniques” by Arnold [33]. It, too, analyzes stealth techniques of rootkits. However, instead of studying their usage in a representative and large sample of current rootkits, Arnold’s thesis uses these techniques to aid detection of a small sample of well-known rootkits¹³ by proper commercial detection engines. Although the goal and setup is different our works overlap in certain parts of the methodology. As Arnold [33] points out regarding earlier works: “*many of the current rootkits have significantly evolved to use different techniques than previous versions, and are actively subverting many of the detectors that are available [...] so the relevancy of the results is questionable*” ([33], p.4). One of these overlapping parts is the collection of data: “*Finally, once a system was infected with a rootkit, a kernel mode debugging session was performed using either the Microsoft Kernel Debugger (KD.exe) or Windbg.exe to analyze the changes to internal Windows OS structures*” ([33], p.19). This thesis emphasizes the stealth of its data collection methods. Using easy to spot tools like WinDBG[34] would likely result in undesired behavioral changes by current rootkits.

¹³Although less famous in 2011, these rootkits had already been analyzed. In contrast, the rootkits in this thesis are largely unknown or treated as unknown due to lack of clear classification by malware vendors.

The second work is called “A Brief Survey on Rootkit Techniques in Malicious Codes.”[35]. It is similar to the third work “A Catalog of Windows Local Kernel-mode Backdoor Techniques” by Skape and Skywing [36] with the main difference being that it is newer but less extensive. Both categorize a number of techniques used by rootkits, many of which can be used for stealth purposes. They provide countermeasures and discuss these methods but do not regard whether or how frequently these methods are used by attackers. While it is true that some of these techniques can be mitigated, an overhead is required to ensure this security. Knowing which areas need additional protection and which techniques to focus on provides an additional edge. Given the relative age of both papers, the conclusion “*Likewise, when these techniques are eliminated, new ones will be developed, continuing the cycle that permeates most adversarial systems.*”, presented in Skape and Skywing [36], p.25, is especially relevant. Similar categorizations of rootkit methods [37]–[39] and their prevention [40]–[42] exist.

The fourth work is a Bachelors thesis by Padakanti [43]. It analyzes functions hooked by a number of open source rootkits in order to group them into families. The goal of the thesis is to improve rootkit detection. In contrast to this thesis, their work is based on a single strategy of kernel rootkits and does not explore these characteristics in a wider, representative rootkit population.

Hasanabadi, Lashkari, and Ghorbani [44] and an earlier work by Muthumanickam and Ilavarasan [45] model the attacker-defender relationship using Game Theory. Similar to this thesis, they point out that protecting against rootkits requires tradeoffs and decisions, which in turn require information about rootkit strategies. Via the mathematical field, they inform on selecting an appropriate focus for detection and prevention efforts.

Existing rootkit detection methods Existing rootkit detection and prevention efforts, excluding few exceptions, either focus on individual components or properties of rootkits, utilize machine learning or are VM-based. Summaries of these methods [46], [47] highlight these common approaches. Amongst the first class, detection based on finding hooks[48]–[50], measuring performance impact[51]–[54], direct kernel object manipulation [55]–[57], API calls [58], process execution fingerprints [59], hidden files [60] The second class includes usage of anomaly detection in memory artifacts [61]. Tian, Ma, Jia, *et al.* [62] use machine learning to identify rootkits based on data collected using virtualization. The third class is formed by papers such as J. Cui, H. Zhang, Qi, *et al.* [63], which can find process blocks in memory samples collected on VMs. The same technique should be applicable for physical machines, although collection of memory samples is less straightforward. Grimm, Ahmed, Roussev, *et al.* [64] find variants of kernels of multiple initially identical VMs to revert changes made by rootkits. Yan, Luo, Feng, *et al.* [65] use virtualization to enforce protected memory. Korin [66] expand on the idea by adapting a hypervisor to ensure integrity and confidentiality of allocated kernel memory. Ahmed, Richard, Zoranic, *et al.* [67] check the integrity of function pointers in the heap. Hua and Y. Zhang [68] detect rootkits using memory forensics via virtual machine introspection. Korin [69] discuss attack on Windows kernel to access exclusively opened files and solves the issue by jailing kernel mode drivers in isolated memory regions via the tool *MemoryRanger*. Lengyel, Kittel, Webster, *et al.* [70] point out that virtual machine introspection based approaches come with their own risks and may not be representative.

A recent paper by the name of “Nighthawk: Transparent System Introspection from Ring-3” by Zhou, Xiao, Leach, *et al.* [71] addresses the common weakness of collecting data about rootkits from within an infected system. It, instead, uses the Intel Management Engine[72] to collect data. Note, however, that this approach does indeed offer protection against tampering by kernel rootkits but is not impervious against other va-

rieties[73]–[78]. Similarly, Soltani, Seno, Nezhadkamali, *et al.* [79] discuss architecture, protocol, type of infection, communication interval, attacks and evasion techniques of real bootkits in the wild and criticize the lack of research focusing on actual samples.

Other papers such as Tsaur [80] and Tsaur and Y.-C. Chen [81] provide proof of concept implementations of rootkits undetected by preceding tools.

4 Methodology

The research question was answered by the following seven steps. First, recent malware samples were obtained (Section 4.1). From them, rootkits were filtered out (Section 4.2). It is beyond the scope of this thesis to completely analyze the functions and intricacies of every rootkit. Instead, certain possible attack vectors a rootkit could attack were selected (Section 4.3). Tests were developed to check if these vectors were targeted (Section 4.4). Then, rootkits were executed on physical machines while tests were running (Section 4.5). The execution traces collected were further analyzed to determine the outcome of the tests (Section 4.6). Lastly, the data was quantified (Section 4.7).

4.1 Dataset collection

VirusTotal [82] provided a selection of non-specific malware samples via an academic agreement enabling downloading for the period of six months, starting on 2019-08-27. A total of 486216 such samples, accompanied by reports in json format, ranging from October 2017 to May 2019 were contained in the academic repository during the available time. Samples were grouped into a number of categories, each in its own encrypted 7z archive. The full list of these categories can be viewed in Appendix A.

As indicated, some of these categories were assumed to not contain rootkits, or contain rootkits with such low density as to be irrelevant. Non-Windows malware was disregarded as well.

Through a separate academic agreement, 129344 non-specific malware samples were obtained from GData[83] in the period from October 2019 to January 2020. No grouping exists for these files. As the detection names are unsuited¹⁴ for identification, all samples have been used for the next step.

Further, samples obtained from VirusShare [84] and Osman Arif [85] were used for testing, but are not included in the results of this thesis.

4.2 Filtering for rootkits

The remaining malware samples were analyzed statically using specific patterns, i.e. certain word or byte sequences. Using a tool called Yara [86], these patterns have been logically combined to filter out unlikely candidates for rootkits. A repository containing such rules for specific malware as well as general malware classes, such as rootkits, has been used[87].

In the subsequent step, resulting malware samples were executed in a live environment similar to 4.5 to determine whether they load a kernel driver. It is assumed that all rootkits are unable to modify the kernel directly without such a driver. A snapshot of a known good version of Windows10 was used. First, a list of loaded kernel drivers was obtained via a program called Kernel Mode Drivers Manager by NoVirusThanks, as shown in Figure 14. Next, the relevant file was executed, loaded or opened. After a period of five minutes, another list of kernel drivers and their signatures were made. If no changes of any sort (date, name, signature) occurred, the file was classified to not be a kernel rootkit and discarded. In addition to checking for a loaded kernel mode driver, the kernel memory region was compared to a known good state. This method was implemented via an automatic batch script by opening the Drivers Manager, taking a screenshot and uploading it to a remote Debian machine in the same network to check for changes in the list of displayed drivers.

¹⁴at least according to my knowledge

Image Name	Base Address	Image Size	Load Order	Publisher	Description
C:\Windows\system32\ptodkml.exe	0x00000000	0x805000	0	Microsoft Corporation	NT Kernel & System
C:\Windows\system32\hal.dll	0x00000000	0x7F000	1	Microsoft Corporation	Hardware Abstraction Layer DLL
C:\Windows\system32\kd.dll	0x00000000	0x8000	2	Microsoft Corporation	Local Kernel Debugger
C:\Windows\system32\hcupdate_GenuineIntel.dll	0x00000000	0xF7000	3	Microsoft Corporation	Intel Microcode Update Library
C:\Windows\System32\drivers\psppc.sys	0x00000000	0x61000	4	Microsoft Corporation	Kernel Remote Procedure Call Provider
C:\Windows\System32\drivers\kscoff.sys	0x00000000	0x29000	5	Microsoft Corporation	Kernel Security Support Provider Interface
C:\Windows\System32\drivers\wvkernel.sys	0x00000000	0x11000	6	Microsoft Corporation	Windows Error Reporting Kernel Driver
C:\Windows\System32\drivers\CLFS.SYS	0x00000000	0x62000	7	Microsoft Corporation	Common Log File System Driver
C:\Windows\System32\drivers\km.sys	0x00000000	0x24000	8	Microsoft Corporation	Kernel Transaction Manager Driver
C:\Windows\system32\PSHED.dll	0x00000000	0x17000	9	Microsoft Corporation	Platform Specific Hardware Error Driver
C:\Windows\system32\BOOTVID.dll	0x00000000	0x8000	10	Microsoft Corporation	VGA Boot Driver
C:\Windows\System32\drivers\FLTMCGR.SYS	0x00000000	0x68000	11	Microsoft Corporation	Microsoft Filesystem Filter Manager
C:\Windows\System32\drivers\clpp.sys	0x00000000	0x101000	12	Microsoft Corporation	CLIP Service
C:\Windows\System32\drivers\mmcxext.sys	0x00000000	0xE000	13	Microsoft Corporation	Kernel Configuration Manager Initial Configuration Extension Host Export Driver
C:\Windows\System32\drivers\ytosext.sys	0x00000000	0xC000	14	Microsoft Corporation	NTOS extension host driver
C:\Windows\system32\ci.dll	0x00000000	0x62000	15	Microsoft Corporation	Code Integrity Module
C:\Windows\System32\drivers\img.sys	0x00000000	0xAA000	16	Microsoft Corporation	Kernel Cryptography Next Generation
C:\Windows\System32\drivers\Wdf01000.sys	0x00000000	0xE3000	17	Microsoft Corporation	Kernel Mode Driver Framework Runtime
C:\Windows\system32\drivers\WDFLDR.SYS	0x00000000	0x13000	18	Microsoft Corporation	Kernel Mode Driver Framework Loader
C:\Windows\system32\drivers\WppRecorder.sys	0x00000000	0xE000	19	Microsoft Corporation	WPP Trace Recorder
C:\Windows\system32\drivers\SleepStudy\Helper.sys	0x00000000	0xF000	20	Microsoft Corporation	Sleep Study Helper
C:\Windows\System32\drivers\acpiex.sys	0x00000000	0x23000	21	Microsoft Corporation	ACPIEX Driver
C:\Windows\system32\drivers\pssectft.sys	0x00000000	0x48000	22	Microsoft Corporation	Microsoft Security Events Component file system filter driver
C:\Windows\system32\drivers\kss.sys	0x00000000	0xA000	23	Microsoft Corporation	LXSS
C:\Windows\system32\drivers\LXCORE.SYS	0x00000000	0xF7000	24	Microsoft Corporation	LX Core
C:\Windows\System32\drivers\ACPI.sys	0x00000000	0x89000	25	Microsoft Corporation	ACPI Driver for NT
C:\Windows\System32\drivers\WMI.LB.SYS	0x00000000	0x0000	26	Microsoft Corporation	WMI.LB.WMI support library DLL
C:\Windows\System32\drivers\wppeng.sys	0x00000000	0x25000	27	Microsoft Corporation	Intel Power Engine Plugin
C:\Windows\System32\drivers\WindowsTrustedRT.sys	0x00000000	0x16000	28	Microsoft Corporation	Windows Trusted Runtime Interface Driver
C:\Windows\System32\drivers\WindowsTrustedRTProxy.sys	0x00000000	0x6000	29	Microsoft Corporation	Windows Trusted Runtime Service Proxy Driver
C:\Windows\System32\drivers\pocw.sys	0x00000000	0x14000	30	Microsoft Corporation	Performance Counters for Windows Driver
C:\Windows\System32\drivers\issasdrv.sys	0x00000000	0x8000	31	Microsoft Corporation	ISA Driver
C:\Windows\System32\drivers\pci.sys	0x00000000	0x3D000	32	Microsoft Corporation	NT Plug and Play PCI Enumerator

Figure 14: A list of Windows Kernel Drivers.

4.3 Identifying vectors to analyze

As listed in Section 3, previous papers have provided extensive overviews of existing strategies. Based on information obtained from rootkit reports and both published and self-conducted disassemblies, these strategies were selected based on their suspected relevance. It is assumed that an attacker already has full access on the system, so privilege escalation or memory corruption attacks with the goal of executing code were disregarded. Further, we only focus on hiding code, not data. While rootkits can implement covert channels, it is assumed that they only become relevant if the easier to detect processes that act on the data are hidden as well; a suspicious process remains a threat regardless of whether it appears to handle data.

The following potential targets were selected as an answer for how rootkits could continuously execute code in a hidden manner¹⁵:

Processes Rootkits need to hide the execution of code in some form. Processes, instances of a program, are the common way to execute code, so we assume rootkits might have an interest in hiding them. While processes provide simple and autonomous execution, they are also fairly heavy and thus easier to detect. *Windows Services* [89], a form of background process similar to Linux daemons, are included in this section.

Threads Processes may have one or more threads which execute concurrently. While every thread is managed by a parent process, it is not sufficient to merely scan for suspicious processes. Using a technique called thread injection, malware can attach a thread to an unsuspecting otherwise harmless running program and does not need to maintain a process. This injected thread can then be hidden.

DLLs DLLs provide library functions to be used by other programs. While a function can be directly executed using `RunDLL.exe`, they are usually loaded by other programs

¹⁵While rootkits could use existing vulnerabilities such as buffer overflows to execute code, these attacks rely on previous exploits and are easier to detect

as needed. Malware could execute code in a benign process without having its own process or thread by changing one of these internal functions. Additionally, it could change the functionality of Windows DLLs like `kernel32.dll` and thereby modify the way the system is supposed to work.

System functionality Given that a rootkit is assumed to have kernel access, it can not just tamper with code executed through a program but also with code executed directly by the system. The functioning of memory allocation (e.g. via `malloc`), program scheduling, or the way the system enters ring 0 (e.g. via the `SYSENTER` fast system call) as well as other system calls might be compromised in some way.

Exceptions and Interrupts Exceptions and interrupts provide a way to temporarily suspend a program and continue execution at a pre-defined other function. Usually, this is used to handle higher priority applications, recover from unexpected program flow or alert about the inability to continue the problematic execution. Rootkits might hijack this functionality to run their own malicious code instead.

IRPs Interactions with devices are implemented via device drivers. A program that wants to make a request to a device constructs an empty I/O request packet and sends it to the corresponding top level driver. This driver does not necessarily have to directly access the device; it forwards the request to a lower driver until the device itself is reached. Then, the packet is filled with the requested data and processed one after another by each driver in the stack until the cleaned up data is passed to the requesting application. Should a rootkit add another driver into the stack or modify an existing one, it could execute arbitrary code.

The rootkits kernel driver Note that the original kernel driver loaded by the rootkit does not strictly classify as a means to execute hidden code, for reasons described in Appendix E. Briefly summarized, a kernel driver is not hidden. In addition, kernel drivers do not offer the flexibility of regular processes and may cause the system to become unstable or even crash, both of which are undesired effects for malware that should not be detected. It is assumed that rootkits only use their kernel driver for what is absolutely necessary and use some of the above means to execute further code.

4.4 Experiment setup

Two kernel drivers were created for the purposes of stealthily collecting memory from a physical machine and observing modified IRPs. To obtain memory samples, a kernel mode driver to read the `PhysicalMemory` device provided by Windows was used. A simple regularly scheduled file copy operation was used to dump the memory contents to a file on disk [90]. The IRP driver was implemented as a simple device driver that also copies the IRP content to disk[91], [92]¹⁶.

4.5 Executing rootkits

A physical machine running Windows 1903 equipped with an A12-9700p r7, 8GB DDR3 RAM and a 256GB SSD. This disk was completely wiped after each execution. Since this paper only analyzes kernel mode rootkits but not bootkits, a wiping of the disk was deemed sufficient. The device was connected to both the internet and a local Debian machine that served as a PxE boot and a ftp server. Via network boot, a verified

¹⁶Implementation details can be found at https://github.com/emysliwietz/kernel_drivers

iso of Windows 1903 including a registry tweak to enable remote command execution (`RemoteApps`) and disabled Windows Updates, Windows Defender and User Account Control was installed on the machine. This iso also already contained the memory kernel driver. Once booted up, 5 minutes after the machine would reply to a ping from the Debian server, a `xfreerdp` command was used to copy an executable rootkit sample and a batch script to the machine. This script then executed the rootkit sample, waited for 10 minutes and restarted the machine from disk in order to allow it to load the kernel driver. Once booted up the second time and after giving Windows another 5 minutes to settle, a memory sample was taken by the driver. This sample was then uploaded to the Debian remote, the disk was wiped and Windows was installed again via PxE.

Before executing any rootkits, this setup was performed five times without executing any rootkit in order to obtain known good states.

4.6 Collecting data

For each of the targets listed in Section 4.3, a reliable method had to be found to determine whether a given kernel rootkit uses them. This section explains how this decision has been made.

Processes To find processes, Volatility has been run on the full memory image taken after installation of the rookit. The plugin `psxview` compares of other volatility plugins, such as `psscan`, which identifies processes by the tag of the memory pool (`POOL_HEADER`). `pslist` iterates over the linked list of `EPROCESS` blocks given by `PsActiveProcessHead` as described in Section 2.3.1. All processes in Windows (unless modified by a rootkit) are listed by id in a table called `PspCidTable`. The kernel function `PsLookupProcessByProcessId` is used to determine whether a given process exist (e.g. to reuse the process id). Further, when Windows boots up, it executes the `client/server runtime subsystem` (`csrss.exe`), which in turn starts other processes and creates or destroys threads. In theory, this process should have handles to the pid of every other process. This is queried by the `csrss` plugin. A discrepancy between the output of these plugins has then been analyzed manually.

Given that the execution environment is controlled, the list of benign processes is already known (some irregularities may occur based on background tasks Windows decides to execute, but discrepancies are minimal) Any additional processes are also suspicious. A rootkit might use a technique called **Process hollowing** to deallocate a running benign process and use the memory space for it's own purpose. Such a process would still be managed under the old name. As counteraction, the memory space of all processes has been compared to a known good state. Moreover, similar filtering has been done for services using `svcsan`.

Threads Similar to processes, threads have been found in the memory image via the `threads` plugin, which enumerates the `ETHREAD` (or `KTHREAD`) linked list. Again, the list of expected threads is known, but might have slight inconsistencies over multiple runs. Since there are a lot more active threads then there are processes, multiple memory snapshots in a good state have been combined to compile a list of benign threads. This volatility plugin implements a number of tags, such as `ScannerOnly`, which detects unlinked threads, `AttachedProcess`, which detects threads attached to other processes, `OrphanThread`, which is able to find system threads that do not have a corresponding loaded module (because it has likely been unlinked), `DKOMExit`, which detects inconsistencies created by tampering with information stored about a given thread, such as its exit time and `HideFromDebug`, which filters for threads that explicitly say they do not want to have debuggers receive events created by this thread via

`NtSetInformationThread`.

Suspicious threads have been investigated manually via `volshell`.

DLLs DLLs included in the `LDR_DATA_TABLE_ENTRY` linked list have been extracted for each running process from memory using the plugin `dlllist` with an offset specified by the previous `psxview` run. `malfind` and `ldrmodules` have been used to find additional dlls hidden through other means.

Two stages have been used to identify hooks:

First, the dlls were dumped via `dlldump` and compared to a good copy via binary diffing. Should the DLL not differ (except for missing sections due to problems caused by dumping the dll), it was considered unaffected by the rootkit. If they do differ, a python script in a headless ghidra setup was used to disassemble all functions in the affected region. The output was dumped into a file for manual analysis.

Second, should a change in the dll have occurred as determined by binary diffing, the dll was again analyzed via volatility. We chose for this order as dumping and binary diffing happened to be more efficient than running the following plugins for the given samples. From the literature it was implied that Import Hooks were the most common in malware, so the entire interrupt descriptor table was dumped via `impscan` and compared to a known good state. The common way to analyze other types of hooks is provided by the `apihooks` plugin. However, the output of this tool is rather extensive and would have been unwieldy to use for a larger sample size of rootkits. Instead, a new recently presented plugin called `hooktracer` produces more accessible output ?? Since the exact purpose of these hooks is not the main focus of this paper, this plugin suffices to determine their existence. Note that `hooktracer` is only designed for userspace, so `apihooks` was still used in certain cases.

System functionality As described in 2.2.4.4, rootkits may perform direct kernel object modification in order to hide. To detect this, the Global Descriptor Table, responsible for storing information about memory segments has been dumped via the `gdt` plugin and inspected for oddities. The contents of the GDT are not necessarily static, so a known good state can be used for reference but not for direct comparison.

For the `ssdt`, a volatility plugin with the same name was used. While there are at least two reserved such descriptor tables, we only analyzed the native and `win32k` one. While a rootkit could install hooks in these tables, it would also need to implement functionality for them, as they are currently unused, so it would in essence only hook itself.

Additionally, a rootkit could install a callback to a system event, such as the creation of a new process which the rootkit could then load a dll into or attach a thread to. Callbacks were exported via the `callbacks` plugin and could be compared to a known good state.

Exceptions and Interrupts The interrupt descriptor table was analyzed as above using the `idt` plugin in verbose mode to include a disassembly. It, too, could be compared to a known good state.

IRPs It is infeasible for the scope of this thesis to listen to every single I/O request packet that is being sent, but a rootkit could still hide functionality by swallowing, changing the contents of or redirecting a packet. This was detected using a four stage process: First, all loaded drivers and devices were exported via `driverlist` and `devicetree`. Given that each selected rootkit installs a kernel driver, there always was at least one driver to analyze. All non-suspicious regular drivers were also included in

the next step. Next, `driverirp` in verbose mode was used to see which major functions are implemented. If a function was redirected to a valid other driver or implemented as expected in the case of regular driver as known from an uninfected scan, it was disregarded. If the legitimacy of the implementation was unknown or the redirection to another driver not obvious, the IRP kernel mode driver described in Section 4.4 was inserted into the driver stack to collect IRPs handled the suspicious major functions, once in the good and once in the infected state. Should the major function exist both in the good and the infected state and no difference could be detected, the apparent change was discarded.

4.7 Quantification of data

The 62 analyzed rootkit samples can be organized into the following families. For reference to the respective hash associated with an ID, consult Appendix B.

- 1-4: Wapomi
- 7-11: Winnti
- 14-19: Zusy
 - 26: CheatEngine
 - 27: Sofacy
- 32-34: Mimikatz
 - 39: Shadowhammer
- 43-44: Nimmul
- 47-50: MSILPerseus
- 53-58: Bladabindi

The remaining 29 samples are unclassified. The above steps were carried out for each sample. Most of those samples tend to use relatively few strategies. If a certain strategy is used, this will be noted in the following Section 5. This thesis explores what strategies tend to be the most popular. Detailed descriptions of the way a sample uses a given method are out of scope. Should a given method not be explicitly mentioned, it was not used by the respective sample.

5 Evaluation

The following section groups the largest families of rootkits used in this thesis, namely those with more than three samples present, into their own paragraphs, as the similarities between them are extensive. These families are already known, so documentation from malware databases could be used to compare our findings. The remaining rootkits have been grouped together into a single block.

5.1 Wapomi

Wapomi, as documented by Trendmicro [93], is a fairly old malware sample already known in 2014. During the last six years, many variations of it have surfaced. The four samples included here seem to have added a rootkit component to the virus. Wapomi spawns a process called `uninstall.exe` as a child of `csrss.exe`, as is common in Windows¹⁷. Samples 1 and 3 make an effort to hide the process, by unlinking the relevant `EPROCESS` block in the kernel. While this suffices to hide in the TaskManager process listing, a regular antivirus should be able to trivially identify the hidden process via `csrss`, as described in Section 4.6. All other samples of this family do not appear to hide a process or threat. The `uninstall.exe` file is copied to every attached storage device. Wapomi then creates an `AUTORUN.INF` file to execute itself automatically once the drive is inserted. The spawned process does not implement harmful functionality itself, instead it downloads and executes additional code from a remote server.

Sample 2 and 3 change the `appmgmts.dll` file. This DLL controls Application Management, which is responsible for installing and uninstalling processes according to the Group Policy. Both samples overwrite the installation routine by a stub, thereby enabling any user to install programs, regardless of administrator access.

All four samples attack `msock.dll`¹⁸. The Microsoft Windows Socket DLL enables access to network sockets. The functions `TransmitFile`, `AcceptEx`, `GetAddressByNameW` were altered. Unfortunately, documentation for this DLL is sparse and reverse engineering its functionality is out of scope for this thesis. Given that Wapomi downloads online code, we assume these alterations were made to hide this network traffic or enable it to pass through a firewall.

5.2 Winnti

Winnti, described in 2013 by Trendmicro [95], is a trojan which drops and executes an internally contained DLL file. The differences between the five samples are due to different encryption algorithms used to obscure the file. Sample 10 uses a different internal DLL while all other samples do not seem to differ significantly. Winnti executes the extracted DLL using `rundll32.exe`. The spawned process is unlinked in the `EPROCESS` list for all samples.

5.3 Zusy

A recently documented [96] trojan called Zusy was first spotted around 2018. It is a dropper, it contains an internal file called `windefender.exe` which it extracts and executes. Samples 16 and 18 do not appear to use any stealth techniques, while the other instances of this family hide their `EPROCESS` blocks.

¹⁷compare to the `init` process in Linux

¹⁸The fourth sample tends to follow Trendmicro [94]

5.4 Mimikatz

The 2019 trojan Mimikatz [97] drops a PowerShell script. It opens network interfaces and tries to connect to various network addresses to send and receive files. In all three samples analyzed, the execution of the script was not hidden. Mimikatz starts a number of scheduled tasks¹⁹. Samples 32 and 33 did not hide any of those processes, while 34 unlinked the processes from the `EPROCESS` structure and the header-list. One of these services is itself a PowerShell command `powershell -ep bypass -e base-64 string`. It is unclear why this command is hidden while the original script was not.

A number of network addresses are accessed to brute-force SMB servers from a stored word-list. Sample 33 made significant changes to `Network.dll`. The function `NetValidatePasswordPolicy` is overwritten by a stub. Should Mimikatz gain access to an SMB server, it creates a user called `k8h3d`. The given function is used to check the provided password for minimum requirements, Mimikatz overwrites this function to use a simpler password for its backdoor account. The functions `NetUserEnum`, used to enumerate user accounts on a server and `NetUserGetInfo`, which provides information on a user, have been overwritten as well. The precise alterations have not been tested, but it is assumed Mimikatz changes these functions to hide its new account. Other functions, such as `NetUserGetGroups`, can still be used to gain information about the new account, although it needs to be known by name before. For the same reason, `NetUseAdd` and `NetUseEnum` have been modified as well. These functions are responsible for adding and enumerating network connections.

All three samples of Mimikatz modify the `svhost.exe` file. The service host executable manages Windows services²⁰, such as the Windows firewall, the updating service and the Bluetooth daemon. The change made by sample 32 and 34 regards the service scheduler, it has been adapted to schedule tasks at half the usual frequency. 33 instead includes a check for the word “Defend” in the process title and does not schedule the service if a match is found, presumably to disable the Windows Defender antivirus.

5.5 MSILPerseus

MSILPerseus [98] is a 2015 trojan. Samples 47, 48 and 50 hide the `EPROCESS` block of a process spawned with a random name, while sample 49 does not seem to use any stealth techniques.

5.6 Bladabindi

Bladabindi is a 2013 remote access trojan developed by a group named *Sparclyheason*[99]. Bladabindi extracts an internal executable file named after a random non-critical Windows process. Samples 53 and 56-58 hide the `EPROCESS` block of this process, while the remaining samples instead hollow out and take over the real process.

Then, Bladabindi tries to scan for usernames and passwords contained in files. The regular Windows functions `CreateFile`, `ReadFile` and `LocalAlloc` are modified by all samples. In the case of `ReadFile`, this allows Bladabindi to read files even if they are currently written to by other programs. The reasons for changing `CreateFile` and `LocalAlloc` are less clear. The malware likely stores files such as screenshots and keylogger data on disk before sending it to a remote server, although this has not been observed.

All samples make changes to `winuser.h`. The `RAWINPUT` structure stores information about keyboard and mouse input. It, as well as `WM_INPUT`, a notification for a program

¹⁹processes that run in regular intervals

²⁰similar to Linux daemons

that new input is available, are slightly changed to re-route input towards the malicious process instead. Bladabindi likely implements a key and mouse logger this way.

A number of other functions are overwritten as well. Samples 54, 56 and 57 overwrite `GetDC`, a function to take a screenshot. Whenever some other application takes a screenshot, Bladabindi takes a copy of it. For likely the same reason, 56 and 58 modify the functions `OpenClipboard`, `EmptyClipboard` and `SetClipboardData`.

All collected data is sent to a command and control server. All samples except for 57 modify `winnetwk.h`, an API for Windows networking. `WNetGetResourceInformationW` provides information on the owner of a given network resource. Bladabindi changes this function to hide its own network usage. The `NETINFOSTRUCT` structure is modified to set the member `dwStatus` to the fixed value `ERROR_BUSY`, likely to deter other processes from attempting to close the connection.

Sample 55 strips checksums from usb IRP requests by replacing them with zeros. While the reason for this is unknown, it might possibly be used to convey some kind of hidden message or simply the presence of the rootkit on the system.

5.7 Remaining samples

A full list of techniques used by all remaining samples can be found in Appendix C. The trend to use an internal process became slightly less important, only 7 unclassified samples use this method for process execution, while 20 rootkits use threads for code execution: Of those 19 processes, seven used `EPROCESS` block hiding, while twelve hol-

DLLs	20
threads	20
processes	19
I/E-Tables	3
descriptor tables	0
IRPs	0

Figure 15: Rootkits using method of code execution

lowed an existing process, reversing the trend of the rootkit families described above. Nine processes removed their headerlist entry, three removed the `csrss` pointer, while six removed their `pcpid` trace. Changed DLLs have less overlap:

<code>comsvcs.dll</code>	COM+ devices	3
<code>rpcrt4.dll</code>	Remote procedure calls	3
<code>shlwapi.dll</code>	Light shell API	3
<code>winnetwk.h</code>	Networking	3
<code>msvbvm60.dll</code>	Visual Basic VM	2
<code>shell32.dll</code>	Shell	2

Figure 16: DLLs changed by more than one rootkit

The DLLs changed by the analyzed samples are somewhat unusual. While most of them, like `comsvcs.dll` and `winnetwk.h` are expected, the first one for spreading malware to attached devices and the second one for secure network communication, other important DLLs, such as `user32.dll`, responsible for tasks related to a logged in user, are left untouched.

6 Discussion

As enumerated in Section 4.7, about half of all rootkits analyzed belong to a known family with more than two instances. The cause of this trend might be due to the provided data sets. Alternatively, the methodology for selecting rootkits might not have been sufficiently random, since the focus has been on finding a number of rootkits, not necessarily finding a diverse number of rootkits. Still, the observation remains striking. Should it reflect an underlying trend in wild rootkits, which it to some extent certainly does, it seems to indicate that the population of active rootkits is not all that large. Quite possibly, a principle similar to the Pareto distribution applies, where a large fraction of active rootkits belong to a small fraction of families, although this thesis does not include this claim within its scope.

Of course, these observations as well as all further points are not free from bias. Rootkits pride themselves with being hard to detect, yet in order for me to be able to analyze the sample, it had to have been detected and cataloged by one of my sources. Antivirus companies tend to use signatures to find malware, it is easier for them to detect already known malicious programs. Infection numbers were not provided, neither by GData or VirusTotal. Although implausible, in theory, it might be possible that a single rootkit sample is responsible for most real-world infections. Some rootkit families, although large, might not be relevant. In addition, a sample size of 62 rootkits is certainly not representative, the high rate of grouping might just be a statistical fluke.

Section 5 shows that most categorized rootkits tend to be older. Should it be representative, the innovation rate of rootkits would not be very high. While this thesis does not include research on the profitability of rootkits, the results hint that the return of investment for a potential attacker is higher when copying and slightly modifying existing malware code instead of writing a new sample from scratch. Based on the threat encyclopedia used in Section 5, some of the sample families also include members that do not implement kernel functionality. It is conceivable that authors adapt existing code to include small rootkit features, possibly to extend the lifespan of old malware by reducing detectability.

Regarding the research question itself, the results indicate that changing the kernel is difficult. Delicate changes, such as direct kernel object manipulation or modification of jump tables cause a system to become unstable. Although rootkits using these advanced features are very hard to detect on a working system, a blue-screen will certainly alert administrators to investigate. The time needed to develop, test, and ensure stability of delicate kernel changes is likely not worth the increase in stealth.

Targeting the `EPROCESS`, `ETHREAD`, headerlist and process hollowing have been the by far most popular strategies. They are fairly easy to implement and provide the desired result of increased stealth without affecting system performance or stability. While not difficult to detect for antivirus engines, the relative ubiquity of documentation on these techniques likely implies that implementation source code is widely circulated, making its cost to utility calculation very favorable.

A large percentage of rootkits focused on simple and fairly non-intrusive changes, such as overwriting DLL functions. Interestingly, there do not seem to be any common functions that are overwritten. Likely, malware authors look at their specific use case, such as not to modify too much, which increases chances for detection. Possibly, changes here are not pre-made, like in the case of `EPROCESS` blocks, but implemented by the malware authors themselves. DLL code is much simpler than kernel code, malware authors presumably have some level of familiarity with it.

An unexpected trend was the large emphasis on network functions. There seems to be a need to bypass on-host firewalls and not appear in traffic logs. Two potential causes exist: Either, firewalls such as the one included in Windows are fairly competent and

require malware authors to bypass them, or the indented audience for malware targeting are not just private individuals but also corporations who utilize more extensive network controls.

This observation, in turn, allows making predictions about commonly installed antivirus engines. If most systems use these scans, rootkits would need to hide better. Given that there is inconsistency and a certain sloppiness in the way rootkits hide, for example by unlinking their `EPROCESS` but not their `csrss` entries²¹, most targets likely rely on no or very rudimentary antivirus engines.

Rootkits tend to favor simple techniques for hiding, the kind that has been around for longer and is well documented. It is unclear whether this is due to the education level of rootkit authors, many of which likely entered the profession because of economic hardship, or due to a return on investment calculation. Every change made on a system needs time to develop, adds additional complexity for maintaining the software and increases the overall footprint. Attackers might be hesitant to include higher level features or a large number of features for these reasons, instead focusing on the bare minimum.

Malware comes in iterations, once detected, authors make small changes and try again. Additional rootkit features can be included in later iterations, to get a head start for generating profit by releasing the malware earlier. Should the first iteration turn out to be easy to detect, including a large number of rootkit features might be devastating. The samples will likely be analyzed quickly and the features included in signature detection, nullifying previous development work.

Most rootkits follow previous proofs of concept by researchers. Most stealth techniques have been first demonstrated in an academic context before wild rootkits adapted them. Presumably, authors of commercial rootkits for targeting individuals work alone or in small teams, leaving little room or resources for innovation. While this implies that most wild rootkits will never be truly surprising, it seems to be sufficient for generating profits.

Naturally, this finding leads into a question of ethics. As shown in Section 1, rootkits cause a lot of damages, some of which could surely be prevented by not publishing new academic rootkits or parts of their implementation. This thesis does not take position on this issue. However, it itself is questionable: malware authors looking for ways to stand out might use it as a starting point. Although the thesis does not discuss the ease of detection of the presented strategies, a high emphasis on one of them by the analyzed samples hints to either a relative obscurity of that strategy or a very high chance of detection when using it.

Of course, none of the presented techniques are novel, malware authors likely already know them and are able to obtain much greater insight than this thesis provides. While this thesis is undoubtedly of some value to rootkit authors, if only to know what their competitors are doing, the benefit provided by further research to hone in on the few strategies used by rootkits outweighs this cost. Knowing about these strategies and their popularities in wild rootkits allows for improvements of antivirus engines, hopefully reducing the general number of rootkits.

²¹which wouldn't cause too much more overhead

7 Future Work

The arms race between malware authors and analysts is constantly advancing, so it is questionable how relevant the findings presented in this paper will be in the future. As researchers focus more on the most widely used stealth vectors, malware might shift to other known hiding methods. Given the complexity of an operating system, new methods to hide may be discovered and abused. Of course, future updates to Windows could change kernel functionality and both make some methods obsolete as well as open up new possibilities for rootkits. It is further possible that rootkits will shift from mostly targeting Windows to other operating systems and kernel structures.

It is difficult to determine how representative the findings of this paper are to the larger population of both recently written as well as currently circulating rootkit samples. Antivirus companies have deeper insight into malware that is currently being detected on real systems, and could therefore make stronger claims in this regard. A larger sample size from a large number of sources and the analysis of more rootkits would also provide results closer to the real distribution.

The collection and analysis steps still rely on manual guidance and intervention. Automated approaches might be used instead to build a pipeline capable of compiling information about a number of stealth vectors for any given binary. Such a solution would work in much the same way as this paper analyzed rootkits but in a single integrated package such as a web-service. Such a product would make it easier to integrate the methods outlined in this paper into other related work. Potentially, every sample uploaded to VirusTotal or similar services could be analyzed in this way, giving researchers and software providers much wider and more up to date insight into rootkit trends.

As discussed in Appendix ??, the method to separate rootkits from other malware is not perfect. Some rootkits that only briefly execute and then leave their kernel driver unloaded have not been identified in this paper. Potentially, the same method but with more frequent checks for loaded kernel drivers might catch additional samples, but we deem this to be unlikely for reasons described in the appendix. The combination of custom build tools using complementing for the purpose of identifying kernel rootkits is preferred and would help to ensure a more representative dataset.

Emphasis has been placed on a stealthy collection of data, but it is still potentially possible for rootkits to notice the presence of kernel mode drivers. Lower-than-operating-system methods to obtain memory samples, for example through a hypervisor would eliminate the chance of rootkits changing their behavior because of the presence of unknown drivers. A solution using the i2c [100] serial bus was planned for this paper but ultimately abandoned due to issues regarding the availability of needed hardware and the resulting time constraints. Using this technology, the memory control unit could potentially be directly accessed through the Intelligent Platform Management Interface [101] and would be untraceable to rootkits. As this method was not physically explored, we cannot make claims about its suitability for this research.

8 Conclusion

Rootkits are made to generate profit. While many possible stealth strategies exist, every change made to the kernel is a liability. Rootkits are meant to hide, an unstable system alerts attention. Therefore, rootkit authors favor stability over complexity. Too many changes, even if each one of them increases stealth on its own, might have unpredictable side effects. Attackers choose to use the least amount of features to be sufficient, both to save on development time and to maximize profits. Intellectual property laws do not apply to already illegal rootkits. Authors copy known and trusted changes, such as process hiding, and stay away from more complicated, riskier and less documented features. This trend reverses on a non-kernel level. Changing DLL functionality is still risky, but malware authors are likely more familiar with the code base. Tailored changes can be made in-house, suited to a given rootkit, instead of simply adapting existing code as is done for kernel changes. Malware goes through iterations, additional features may be added later on. Circulating malware is not necessarily a finished product, prototypes and forks of other projects exist as well. Rootkit families group together and implement similar functionality. Subtle differences between code are uncommon, malware instead tends to remove or add certain features one at a time. Of course, these results have been obtained with a relatively small sample size of current rootkits. It is left to future research to validate the relevancy of these findings or to document changes rootkits might make. As antivirus engines get more powerful, it is to be expected that rootkits follow.

A List of Categories obtained from VirusTotal

VirusTotal provided a zip archive with many categories of malware. Some of them have been used to filter for rootkits, while other have been discarded. The following table briefly describes all families and whether they were a part of this thesis:

Category	Description	Used
2018-06-miners	Cryptominers	No
7ZIP	7z archives	Decompressed
ACE	ACE archives (legacy)	No
Android	android apps	No
Apple software package	MacOS software	No
ASF	Microsoft ASF encoded videos	No
BMP	Bitmap images	No
BZIP	BZIP archives	No
C++	C++ source code	Scanned for inline asm, functions/routines associated with rootkits
C	C source code	See above
CAB	Cabinet archives	Decompressed
COFF	Unix executables	No
Debian Package	Debian (Linux) software	No
DOS COM	MSDOS scripts	No
DOS EXE	MSDOS executables	Yes
DZIP	NN compressed archives	No
ELF	Unix executables	No
Email	Email backups and attachments	Yes
Flash	Adobe flash	No
Fortran	Fortran source code	No
GIF	Animated image format	No
Google Chrome Extension	Browser extensions	No
GZIP	GZIP archives	No
Hangul Word Processor document	Korean documents	No
HTML	HTML documents	No
iPhone	iOS apps	No
ISO image	disk images	Yes
JAR	Java executables	Yes
Java	Java source code	Scanned for functions/routines associated with rootkits
Java Bytecode	Java byte	Yes
JPEG	image files	No
Linux RPM package	RedHat (Linux) software	No
Mach-O	Apple executable format	No
Macintosh Disk Image	MacOS disk images	No
MIDI	sound files	No
Mozilla Firefox Extension	browser extensions	No

mp3	audio files	No
MP4	video files	No
MS Excel Spreadsheet	tabular documents (VisualBasic macros)	No
MS PowerPoint Presentation	presentations (VisualBasic macros)	No
MS Word Document	text documents (VisualBasic macros)	
Network capture	Package capture files	No
Office Open XML Document	text documents	No
Office Open XML Presentation	presenations	No
Office Open XML Slide Show	presenations	No
Office Open XML Spreadsheet	tablular documents	No
Outlook	Email backups and attachments	Yes
PalmOS	PDA software	No
Pascal	Pascal source code	No
PDF	PDF documents	Yes
Perl	Perl source code	No
PHP	PHP source code	No
PNG	image files	No
PostScript	PostScript documents	No
Python	Python Source Code	No
RAR	RAR archives	Yes
Rich Text Format	text documents	No
Shell script	Unix scripts	No
TAR	TAR archives	No
Text	text files	No
TrueType Font	Fonts	Yes
unknown	various	Yes
Win16 EXE	16bit MS executables	No
Win32 DLL	32bit MS libraries	Yes
Win32 EXE	32bit MS executables	Yes
Windows Installer	MS software	Yes
Windows shortcut	MS shortcut files	No
XML	XML documents	No
ZIP	ZIP archives	Yes

B List of Samples

In total, 62 samples were identified as rootkits and further analyzed. Given the differing naming conventions of antivirus engines, the author included a best guess to identify the rootkit, if possible. This guess need not always be accurate. All samples used have been uploaded to VirusTotal²², refer to the SHA-256 hash for further information. Multiple versions of the same rootkit family were included. In most cases, changes tend to be rather small, although some of them, i.e. *Wapomi* changed their behaviour between versions. For easier reference in the main thesis body, every sample has been assigned an ID, which carries no further significance. GData seems to use the detection name **GenericKD** to refer to kernel drivers. Since GData has not released any insight into their naming scheme, this observation remains speculative. The following samples are presented in no particular order.

ID	SHA-256	Best guess of classification
1	0ee76f971a666ca9f77f456a1493da1129fac9e40ca929040967e43f37e61777	Wapomi
2	124510769217fa8bd249e115438850714c9d70db0f607bdae6dc3f3f5bac0d8c	Wapomi
3	c2781cd83f7178dcc76835e4b27bdcc789cb8470270e2576596ddd1f4a3e61e	Wapomi
4	95c4d6f12e50e5f58c35c3df6dc031d32631096d6a2381725981a2e79335ec59	Wapomi
5	dafacefe134cf66f1213bded7af2ac8ea7a23d8f6c655842ed4d18c147e7673b	?
6	60cbb8bd1e20e998ec43710347c81cf9908f532386fba8b67534a7cf68eb6f54	?
7	243e73ecb5ad479d4469baa8564aff89cd25a70de6cdc9e87817ddad2f315db3	Winnti
8	28810d2bba6b1fd8f1a6d2311e471f5723027bac1f0fb67108cc47f51c66c446	Winnti
9	8e12446077573842d0f22b1eecf470081652b1c12d84eedfdaaeeda2287d08fd	Winnti
10	44a8846db20acc12af7c4d69f0d2bf2a20a29fbf6e7c0962631ed65b42487661	Winnti
11	b3882d96d300ce7664b1e90df1a919c45e2061b5729463ff588e0534bd1456a3	Winnti
12	d1468fe1caba5ce27f4c3e93c76081c17e709d5ed2df7916db72a28990aa1ef2	?
13	69e966e730557fde8fd84317cdef1ece00a8bb3470c0b58f3231e170168af169	?
14	89a2dfd11302952582f662df52eb782c0d86276a87a8c8bb845475f94b92af02	Zusy
15	a25f4b8b96f5f6ed734568eba9ae1aeea0fbf8f4b178d61d9bce64caead3d965	Zusy
16	0b921f2820c78f1e755f6a5e2e2e49e0a8b1d26f9e91e8bfc001c829fa207630	Zusy
17	3071875f2dd48b58f0ee708e5151bac2beb179672185a5210ea7023da0764d4e	Zusy
18	0f7d3c7c5629810980b1b166c2fa9a5e44ce39d85df77930f60271adb55fb877	Zusy
19	75913f8871592331f96163952fea7fbbdbfd26fe88a152dc5db3084c7de4fbb0	Zusy
20	df935bfd5b701eda758b780769c09be5d52e02d9c02ecd934b321cbb285a72	?
21	47633ca9e319d89a2abc44c5195c47531e94032a8b3b1e8f32f8dd51f315ae5f	?
22	2df7e28dc521f7a4a1538d8baf4523cfd478256be15f88feb944bae6f484721e	?
23	530ba7074523aa39d813e087e4b99bb005b9316db61ebcae0cbaee189ede1934	?
24	0ede60d715c33a35684e80458d7b70ff987c0519767d51c56e4a30219b33cc84	?
25	e204164ca564c612ff65261d187f8fb0cfe8482176f2adb2ef5a2a08b43ae199	?
26	12ea61a67604c939ab7378764a768acc40118fab3a2035c79cad47fe03904e71	CheatEngine
27	25ff92ed61208513c0aec27cd3b4864f8f4604960a58d59310ec64a8706fb6f1	Sofacy
28	ac55ab499121685f81fff55bd39b855555e0490f5ed68aae93f864d403fa8666	?
29	ef62be4e92283a790ae872e36e659fa2db7c1d30867e34ba06c3d4f780bb5384	?
30	c242bf7553b3b3a2672183bb9c57fa06357d5740152401fb0819b50f461b0983	?
31	505c8a348169e544a9f96c754e701fd8cb0648501a36849a9109868d8c045e4f	?
32	ff855ef9b9eaedc47f23d9e6d7db972fe09ddd8ba9fe8ba2eb7574703bc481b6	Mimikatz
33	1f3448353ff7980d1ae8fbf09fdcd7471f8101cb4df30d6c117639da34e9f3	Mimikatz

²²by third parties

34	d19fb6c9b4d7b5ad3002ead50046e8999a80c95d863d647f57c8ca9ce104bd45	Mimikatz
35	d90aa414598046171ec6217a37d66797f1326fd4291bc2096e8c01a21c4a7d1c	?
36	9822e2055f8497ca24610bc34a29ffa2297a1980d2e2cddb7c97c999cab722f	?
37	30d8b1272a764d0ad45442017256f43237ec813ce508da3bdb7e55de18374e1f	?
38	6733975e736f554dfb8cacc196160f6157d5b9b704ac017cb6a78488aa35576	?
39	c2f4b5180fc4b351ec07b9d9bb745dade135e58dcfdeb96f6156684b68e80d2a	ShadowHammer
40	ff22856d7fa4d982f5587a8a60b421fc9129470aab79287db28b5cc04e6eac4b	?
41	b11d8aa43f0343572c4c7c9cb7acd416d8b17e4ed322d07df0c1ae5cc23c3625	?
42	c1e9a7d295731eab65bd6a371d885e0ad14553c79ced4a0a2aaa7d5fbd0cad	?
43	46b9a583ae9d0e403b6c8dfb19cb3cfe455a32a5cfff33d8ae4c8fb44632d41	Nimnul
44	337ac7115411094d4ec42c43b11df2355473423ef48dd5a87976b53fd10c35d0	Nimnul
45	25a3eb871341cb91845a6e3fa5991cafddedc72a6638c810ea79278e94a12a53	?
46	9b345259e76a48fb927101cecbd664fe7babb74dbb49b213e33cc737dcbd8ad7	?
47	03a17d89936a37e43bf5ddcf477d914ac156268c672900bd2a64e1835e7bd3a8	MSILPerseus
48	7dea3ff5e383d70c4788661964f0f16900faa129f93fdc7e8bb84ecce6c0ec41	MSILPerseus
49	24b69fdface2ac9f9f9c31743ae628859355ed45b5b7a92f97eaf1bbb6aa5504	MSILPerseus
50	ac865bf57c99b7d95a8f8c9b143e4abb54d897e66ae97a86b0927e6f9e80c3bb	MSILPerseus
51	c88504b4641e9798a300a203d2277100220babcb141c7abb6e88bafcdeddf4a	?
52	b3691c3193077bf43dfd48f4b30a234a3ad02c3c74a185685ebd8a1fa096524d	?
53	070b88ec36f26bbe31c15baba7fa6608ecee0670c38df6e68f7b128b62bdc67d	Bladabindi
54	a6e41d32e6807b13f546e29c8b4ee330157ee0174e216b1a07a37f6435734af3	Bladabindi
55	008ed01c46a66448de3c3eb7ae79953266371adbfe015bbf9ba65236e9ac6174	Bladabindi
56	0a76b32f57452e3121164c8ad1cd3060d386562586f87d99c3549c34efcb98f5	Bladabindi
57	0f622c328a2a04614b3e8f3267f91c771a5b0c527f0a8e0e0ca53250911464db	Bladabindi
58	3925f2cedb3ca31ff8defc630e9b7a502cade5b677b596e09eade562da7c63c9	Bladabindi
59	dafacefe134cf66f1213bded7af2ac8ea7a23d8f6c655842ed4d18c147e7673b	?
60	a2c1e614395a02f8db2bb93719aaa490b6a75db000cb6e92eda0113886d653c7	?
61	b4b74f8abd41806679ff85777cdded1dceaef0c0759156b105279b4a8f2f4cdb	?
62	c092db06e52a5477e2fb6093ae93ce2d8355389fd599381a22218b3eda30263d	?

C Unclassified malware evaluation

A summary of changes made by every unclassified malware sample:

- 5**
- Orphan thread
 - Changes to `winnetwk.h`
 - Changes to `rpcrt4.dll`
 - Changes to `shell32.dll`
- 6**
- Unlinks `ETHREAD` block
 - Orphan thread
- 12**
- Unlinks `EPROCESS` block
 - Removes headerlist entry
 - Changes to `winnetwk.h`
- 13**
- Process hollowing
- 20**
- Process hollowing
 - Removes `csrss` entry
 - IAT hook in `msvbvm60.dll`
 - Changes to `kernel32.dll`
- 21**
- Process hollowing
- 22**
- Unlinks `EPROCESS` block
 - Unlinks `ETHREAD` block
 - Removes headerlist entry
 - Removes `csrss` entry
 - Removes `pcpid` entry
 - IAT hook in `comsvcs.dll`
- Changes to `shlwapi.dll`
 - Changes to `crypt32.dll`
 - Changes to `system32.dll`
- 23**
- Nothing
- 24**
- Process hollowing
 - Changes to `winnetwk.h`
 - Changes to `shlwapi.dll`
- 25**
- Unlinks `EPROCESS` block
- 26**
- Unlinks `EPROCESS` block
 - Removes headerlist entry
 - Process hollowing
- 27**
- Unlinks `ETHREAD` block
 - Orphan thread
- 28**
- Orphan thread
 - Removes `pcpid` entry
 - Changes to `ntdll.dll`
 - Changes to `rpcrt4.dll`
- 29**
- Process hollowing
 - Unlinks `ETHREAD` block

- 30**
 - Unlinks ETHREAD block
 - Changes to rpcrt4.dll
- 31**
 - Unlinks EPROCESS block
- 35**
 - Unlinks EPROCESS block
 - Removes headerlist entry
- 36**
 - Process hollowing
 - Removes headerlist entry
 - IAT hook in comsvcs.dll
- 37**
 - Unlinks ETHREAD block
 - Orphan thread
- 38**
 - Unlinks EPROCESS block
 - Removes pcpcid entry
- 39**
 - Unlinks ETHREAD block
 - Removes headerlist entry
- 40**
 - Orphan thread
- 41**
 - Unlinks ETHREAD block
- 42**
 - Process hollowing
 - Removes headerlist entry
- 43**
 - Nothing
- 44**
 - Unlinks ETHREAD block
 - Changes to ws2_32.dll
- 45**
 - Process hollowing
 - Removes headerlist entry
 - Removes pcpcid entry
- 46**
 - Process hollowing
 - Removes headerlist entry
 - Orphan thread
 - Unlinks ETHREAD block
- 51**
 - Hook on raw input data interrupt
 - Changes to winnetwk.h
- 52**
 - Unlinks ETHREAD block
- 47**
 - Process hollowing
 - Changes to LogiLDA.dll
 - IAT hook in comsvcs.dll
 - EAT hook in comsvcs.dll
 - Changes to shlwapi.dll
- 59**
 - Unlinks ETHREAD block
- 60**
 - Hook on raw input data interrupt
 - Removes csrss entry
 - Removes pcpcid entry
 - Changes to shell32.dll

61

- Unlinks ETHREAD block
- Changes to advapi32.dll
- IAT hook in msvbvm60.dll

62

- Process hollowing
- Hook on shutdown interrupt
- Removes pepcid entry

D The case against VMs

Given the low level implementation of the features being used by rootkits, it must be assumed that rootkit authors possess an in-depth understanding of lower level system architecture and system internals. Because of this, rootkit authors are likely also capable of subverting a hardening VM guide that can be found relatively easily via a web search. As the evaluation of a VM hardening script has shown [102], even after applying all mitigations, VMs still expose three times as many indicators of being a virtual environment as physical machines do. Virtual machines also mask themselves using generic hardware and predictable specifications (generic CPU model, standard memory bandwidths, etc), which provide clues about the platform not being physical. Furthermore, the majority of related work uses virtual machines. The usage of physical machines explores the lesser known path.

E Kernel drivers and their use as a selection criteria for rootkits

Detecting a loaded kernel driver has been used as the final method to separate rootkits from other malware. The following section will discuss this choice.

Windows represents a kernel driver using a structure called `DRIVER_OBJECT`. In it, information about the driver, the devices that are supported and the main functions used to control it are provided. Additionally, an indirect member called `KLDR_DATA_TABLE_ENTRY` contains additional information, including the entry point of the driver (`PVOID EntryPoint`) as a void pointer. Drivers are initialized by calling their `DriverEntry` routine and may chose to provide an `Unload` routine[103].

As previously discussed, a kernel rootkit per definition has kernel access. This access can be realized either through a driver provided by the rootkit or an exploit in the existing Windows kernel or an unrelated third party kernel driver which happens to be loaded. For the purpose of this paper, the possibility of a zero day in the NT kernel exploited by already detected rootkits is deemed as exceedingly unrealistic. As the Windows image used for tests is minimal and does not contain additional kernel drivers besides those shipped with a default Windows installation, there are no third party drivers that could be exploited for kernel access. A kernel rootkit, not a bootsector or firmware rootkit, both of which are out of scope, thus has to load a kernel driver that has not been there before and thereby increase the number of loaded modules.

In principle, the software used to list all loaded drivers might not detect the additional driver, for one of three reasons:

1. A bug specific to the software causes it not to find certain drivers
2. The malware knows about the software and actively subverts it before loading a kernel module
3. The rootkit hides its own kernel module such that the software does not list it

Given that the software is proprietary, option one could indeed happen, but assuming the rootkit does not actively hide, it would be very unlikely for any one drivers loading procedure to differ significantly enough from those of other drivers to trigger such a bug. We perceived this risk to be small enough to accept it. In principle, the issue will persist but can be made even smaller by using a second or a third method for listing kernel drivers, but no guarantees for completeness can be made unless a formally verified implementation is presented.

Similarly, the second possibility is valid but highly unlikely. To our current knowledge, no malware using this trick has been observed. A slightly more probable approach for malware upon finding that the software is present might be to decide they are in a monitored environment and not install the kernel module. However, the software was only copied to the machine after the initial reboot, meaning that the kernel driver was already loaded before the malware could make any decision based on the existence of the software. While the malware could sit dormant and listen for this exact program until a certain number of restarts has occurred, we deemed this extremely improbable and accepted the risk.

The remaining possibility is similar. While a rootkit could potentially implement a kernel driver with significant differences, for example an initialization routine other than `DriverEntry`, by instructing the linker, a kernel driver is only useful if it can interface with the kernel. It is conceivable that the driver of a rootkit is not described though a `DRIVER_OBJECT` structure but in a custom struct accessed via some custom bridge, but then that bridge needs to communicate with the kernel. A kernel driver can theoretically

be obscured to a point that its purpose is completely unrecognizable, but some part of it needs to be loaded via standard means for it to serve any function. Since the purpose of a driver does not matter for our purposes and the number of loaded drivers does not change, our solution may still be used.

Another potential problem might be a rootkit that goes further than hiding. Assume a rootkit that loads a driver, which immediately performs some functions and promptly unloads itself, all in the span of a few milliseconds. While the usefulness of such a rootkit is debatable as it could not hide changing malware, it would be falsely classified as not a rootkit by our method.

Note that this work does not focus on providing a completely accurate method for finding all kernel rootkits in a given malware population.

References

- [1] Positive Technologies, “Putting a price on apt attacks,” 2019. [Online]. Available: <https://www.ptsecurity.com/upload/corporate/ww-en/analytics/APT-Attacks-eng.pdf>.
- [2] Accenture, *Ninth Annual Cost of Cybercrime Study*, Mar. 2019. [Online]. Available: <https://www.accenture.com/us-en/insights/security/cost-cybercrime-study>.
- [3] Herjavec Group, *The 2019 official annual cybercrime report*. [Online]. Available: <https://www.herjavecgroup.com/the-2019-official-annual-cybercrime-report/>.
- [4] Jeff Petters, *What is an Advanced Persistent Threat (APT)?* Mar. 2020. [Online]. Available: <https://www.varonis.com/blog/advanced-persistent-threat/>.
- [5] Kaspersky, *Advanced Persistent Threats in 2020: abuse of personal information and more sophisticated attacks are coming*, Nov. 2019. [Online]. Available: https://www.kaspersky.com/about/press-releases/2019_advanced_persistent_threats_in_2020_abuse_of_personal_information_and_more_sophisticated_attacks_are_coming.
- [6] J. Butler and P. Silberman, “Raide: Rootkit analysis identification elimination,” *Black Hat USA*, vol. 47, 2006.
- [7] P. Gmerek, *Gmer*, 2016. [Online]. Available: <http://www.gmer.net/>.
- [8] Avast. [Online]. Available: <https://press.avast.com/avast-gmer-technology-gets-top-score-in-rootkit-detection-tests>.
- [9] *NortonPowerEraser*, 2020. [Online]. Available: <https://us.norton.com/support/tools/npe.html>.
- [10] *UnHackMe*, 2020. [Online]. Available: <https://unhackme.en.softonic.com/>.
- [11] M. S. Live, “RootkitRevealer,” 2006. [Online]. Available: <https://docs.microsoft.com/en-us/sysinternals/downloads/rootkit-revealer>.
- [12] J. A. Halderman and E. W. Felten, “Lessons from the sony cd drm episode,” in *USENIX Security Symposium*, 2006, pp. 77–92.
- [13] MalwareTech, *Ring3 / ring0 rootkit hook detection*, 2013. [Online]. Available: <https://www.malwaretech.com/2013/10/ring3-ring0-rootkit-hook-detection-22.html>.
- [14] Microsoft, *hotpatch (create hotpatchable image)*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/cpp/build/reference/hotpatch-create-hotpatchable-image?redirectedfrom=MSDN%5C&view=vs-2019>.
- [15] Allmnet, “Eprocess and kprocess, ethread, kthread structures,” 2017. [Online]. Available: <https://asecurity.dev/2017/12/eprocess-and-kprocess-ethread-kthread-structures/>.
- [16] Microsoft, *Enumprocesses function*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/psapi/nf-psapi-enumprocesses>.
- [17] —, *Getprocessorsystemcycletime function*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getprocessorsystemcycletime>.
- [18] —, *Getsystemtimes function*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getsystentimes?redirectedfrom=MSDN>.

- [19] —, *Getsystemtimes function*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/windows/win32/api/threadpoollegacyapiset/nf-threadpoollegacyapiset-createtimerqueueetimer>.
- [20] X. Kovah. (2010). “Rootkits: What they are and how to find them, part 1,” [Online]. Available: http://opensecuritytraining.info/Rootkits_files/Rootkits-Part1.ppt.pdf.
- [21] —, (2010). “Rootkits: What they are and how to find them, part 2,” [Online]. Available: http://opensecuritytraining.info/Rootkits_files/Rootkits-Part2.ppt.pdf.
- [22] —, (2010). “Rootkits: What they are and how to find them, part 3,” [Online]. Available: http://opensecuritytraining.info/Rootkits_files/Rootkits-Part3.ppt.pdf.
- [23] E. C. Thompson, *Cybersecurity Incident Response*. Apress, 2018, pp. 120–121. DOI: 10.1007/978-1-4842-3870-7. [Online]. Available: <https://doi.org/10.1007%2F978-1-4842-3870-7>.
- [24] B. Rankin, “Rootkit prevention understanding rootkits and the role they play in malware attacks,” *lastline*, Feb. 2018. [Online]. Available: <https://www.lastline.com/blog/rootkit-prevention/>.
- [25] Malwarebytes, “Rootkits,” Jun. 2016. [Online]. Available: <https://blog.malwarebytes.com/threats/rootkits/>.
- [26] Microsoft, “Rootkits,” Apr. 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/rootkits-malware>.
- [27] C. Ries, “Inside windows rootkits,” *VigilantMinds Inc*, vol. 4736, p. 27, 2006.
- [28] H. K. Brendmo, “Live forensics on the windows 10 secure kernel,” Master’s Thesis, NTNU, 2017.
- [29] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2006.
- [30] E. Perla and M. Oldani, *A guide to kernel exploitation*. Elsevier, 2011.
- [31] B. Blunden, *The Rootkit arsenal: Escape and evasion in the dark corners of the system*, second. Jones & Bartlett Publishers, 2012.
- [32] A. Matrosov, E. Rodionov, and S. Bratus, *Rootkits and bootkits: reversing modern malware and next generation threats*. No Starch Press, 2019.
- [33] T. M. Arnold, “A comparative analysis of rootkit detection techniques,” M.S. thesis, University of Houston-Clear Lake, 2011.
- [34] Microsoft, *WinDBG*, 2019. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools>.
- [35] S. Kim, J. Park, K. Lee, I. You, and K. Yim, “A brief survey on rootkit techniques in malicious codes,” *J. Internet Serv. Inf. Secur.*, vol. 2, no. 3/4, pp. 134–147, 2012.
- [36] Skape and Skywing, “A catalog of windows local kernel-mode backdoor techniques,” *Uninformed*, vol. V8, no. A2, Aug. 2007, Authors’ names are pseudonyms; their main websites from this time seem to be abandoned or offline. The first authors real name is Matt Miller. [Online]. Available: <http://www.hick.org/~mmiller/>.
- [37] M. Landi, “The methods of windows rootkits,” *Journal of Applied Security Research*, vol. 4, no. 3, pp. 389–426, 2009.

- [38] M. Miller, “A brief history of exploitation techniques & mitigations on windows,” 2007.
- [39] K. Kortchinsky, “Real world kernel pool exploitation,” *InSyScan08 Hong Kong*, 2008.
- [40] A. Backman, “A brief tour on control-flow protection,” Bachelor’s Thesis, 2019.
- [41] P. Bravo and D. F. Garcia, “Proactive detection of kernel-mode rootkits,” in *2011 Sixth International Conference on Availability, Reliability and Security*, IEEE, 2011, pp. 515–520.
- [42] A. Ramaswamy, “Autoscopy: Detecting pattern-searching rootkits via control flow tracing,” Master’s Thesis, Citeseer, 2009.
- [43] S. Padakanti, “Rootkits detection using inline hooking,” Bachelor’s Thesis, Texas A&M University-Corpus Christi, 2012.
- [44] S. S. Hasanabadi, A. H. Lashkari, and A. A. Ghorbani, “A game-theoretic defensive approach for forensic investigators against rootkits,” *Forensic Science International: Digital Investigation*, p. 200 909, 2020.
- [45] K. Muthumanickam and E. Ilavarasan, “Optimization of rootkit revealing system resources—a game theoretic approach,” *Journal of King Saud University-Computer and Information Sciences*, vol. 27, no. 4, pp. 386–392, 2015.
- [46] Y. Liu, Z.-t. Jiang, S.-k. Gang, and K.-w. Li, “A method of doubly detecting rootkit,” *Computer and Modernization*, no. 2, p. 29, 2009.
- [47] J. Joy, A. John, and J. Joy, “Rootkit detection mechanism: A survey,” in *International Conference on Parallel Distributed Computing Technologies and Applications*, Springer, 2011, pp. 366–374.
- [48] Y. Akao and T. Yamauchi, “Krguard: Kernel rootkits detection method by monitoring branches using hardware features,” in *2016 International Conference on Information Science and Security (ICISS)*, IEEE, 2016, pp. 1–5.
- [49] Z. Wang, X. Jiang, W. Cui, and X. Wang, “Countering persistent kernel rootkits through systematic hook discovery,” in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2008, pp. 21–38.
- [50] Z. Wang, X. Jiang, W. Cui, and P. Ning, “Countering kernel rootkits with lightweight hook protection,” in *Proceedings of the 16th ACM conference on Computer and communications security*, ACM, 2009, pp. 545–554.
- [51] J. Dawson, “Rootkit detection through phase-space analysis of system call timing and power data,” Master’s thesis, 2017.
- [52] C.-W. Wang, C. K. Chen, C.-W. Wang, S. W. Shieh, *et al.*, “Mrkip: Rootkit recognition with kernel function invocation pattern,” *J. Inf. Sci. Eng.*, vol. 31, no. 2, pp. 455–473, 2015.
- [53] X. Wang and X. Guo, “Numchecker: A system approach for kernel rootkit detection and identification,” *Black Hat Asia*, 2016.
- [54] P. Lockett, J. T. McDonald, and J. Dawson, “Neural network analysis of system call timing for rootkit detection,” in *2016 Cybersecurity Symposium (CYBERSEC)*, IEEE, 2016, pp. 1–6.
- [55] M. Graziano, L. Flore, A. Lanzi, and D. Balzarotti, “Subverting operating system properties through evolutionary dkom attacks,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2016, pp. 3–24.

- [56] A. Baliga, V. Ganapathy, and L. Iftode, “Detecting kernel-level rootkits using data structure invariants,” *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 670–684, 2011.
- [57] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, “Manipulating semantic values in kernel data structures: Attack assessments and implications,” in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, 2013, pp. 1–12.
- [58] F. Mira, “A review paper of malware detection using api call sequences,” in *2019 2nd International Conference on Computer Applications & Information Security (ICCAIS)*, IEEE, 2019, pp. 1–6.
- [59] L. Zhou and Y. Makris, “Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution,” in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 1580–1585.
- [60] G. Ramani and S. Kumar, “Nonvolatile kernel rootkit detection using cross-view clean boot in cloud computing,” *Concurrency and Computation: Practice and Experience*, 2019.
- [61] M. Jones, “Automated in-memory malware/rootkit detection via binary analysis and machine learning,” 2018, Presented at CAMLIS: 2018. [Online]. Available: <https://www.youtube.com/watch?v=G2b8c5tMQZk>.
- [62] D. Tian, R. Ma, X. Jia, and C. Hu, “A kernel rootkit detection approach based on virtualization and machine learning,” *IEEE Access*, vol. 7, pp. 91 657–91 666, 2019.
- [63] J. Cui, H. Zhang, J. Qi, R. Peng, and M. Zhang, “Hidden process offline forensic based on memory analysis in windows,” *Wuhan University Journal of Natural Sciences*, vol. 22, no. 4, pp. 346–354, 2017.
- [64] J. Grimm, I. Ahmed, V. Roussev, M. Bhatt, and M. Hong, “Automatic mitigation of kernel rootkits in cloud environments,” in *International Workshop on Information Security Applications*, Springer, 2017, pp. 137–149.
- [65] G. Yan, S. Luo, F. Feng, L. Pan, and Q. G. K. Safi, “Moskg: Countering kernel rootkits with a secure paging mechanism,” *Security and Communication Networks*, vol. 8, no. 18, pp. 3580–3591, 2015.
- [66] I. Korkin, “Hypervisor-based active data protection for integrity and confidentiality of dynamically allocated memory in windows kernel,” *arXiv preprint arXiv:1805.11847*, 2018.
- [67] I. Ahmed, G. G. Richard, A. Zoranic, and V. Roussev, “Integrity checking of function pointers in kernel pools via virtual machine introspection,” in *Information Security*, Springer, 2015, pp. 3–19.
- [68] Q. Hua and Y. Zhang, “Detecting malware and rootkit via memory forensics,” in *2015 International Conference on Computer Science and Mechanical Automation (CSMA)*, IEEE, 2015, pp. 92–96.
- [69] I. Korkin, “Memoryranger prevents hijacking file_object structures in windows kernel,” *arXiv preprint arXiv:1905.09543*, 2019.
- [70] T. Lengyel, T. Kittel, G. Webster, J. Torrey, and C. Eckert, “Pitfalls of virtual machine introspection on modern hardware,” in *1st Workshop on Malware Memory Forensics (MMF)*, Citeseer, 2014.
- [71] L. Zhou, J. Xiao, K. Leach, W. Weimer, F. Zhang, and G. Wang, “Nighthawk: Transparent system introspection from ring-3,” in *European Symposium on Research in Computer Security*, Springer, 2019, pp. 217–238.

- [72] Intel, *Intel Management Engine*, 2020. [Online]. Available: <https://www.intel.com/content/www/us/en/support/products/34227/software/chipset-software/intel-management-engine.html>.
- [73] A. Algawi, M. Kiperberg, R. Leon, A. Resh, and N. Zaidenberg, "Creating modern blue pills and red pills," in *ECCWS 2019 18th European Conference on Cyber Warfare and Security*, Academic Conferences and publishing limited, 2019, p. 6.
- [74] R. Wojtczuk and C. Kallenberg, "Attacking uefi boot script," in *31st Chaos Communication Congress (31C3)*, 2014.
- [75] S. Embleton, S. Sparks, and C. C. Zou, "Smm rootkit: A new breed of os independent malware," *Security and Communication Networks*, vol. 6, no. 12, pp. 1590–1605, 2013.
- [76] A. Tereshkin and R. Wojtczuk, "Introducing ring-3 rootkits," *Black Hat USA*, 2009.
- [77] R. Wojtczuk and J. Rutkowska, "Attacking intel trusted execution technology," *Black Hat DC*, vol. 2009, pp. 1–6, 2009.
- [78] R. Wojtczuk, J. Rutkowska, and A. Tereshkin, "Another way to circumvent intel trusted execution technology," *Invisible Things Lab*, pp. 1–8, 2009.
- [79] S. Soltani, S. A. H. Seno, M. Nezhadkamali, and R. Budiarto, "A survey on real world botnets and detection mechanisms," *International Journal of Information and Network Security*, vol. 3, no. 2, p. 116, 2014.
- [80] W.-J. Tsaur, "Strengthening digital rights management using a new driver-hidden rootkit," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 2, pp. 479–483, 2012.
- [81] W.-J. Tsaur and Y.-C. Chen, "Exploring rootkit detectors' vulnerabilities using a new windows hidden driver based rootkit," in *2010 IEEE Second International Conference on Social Computing*, IEEE, 2010, pp. 842–848.
- [82] VirusTotal, *VirusTotal: Analyze suspicious files and urls to detect types of malware, automatically share them with the security community*. [Online]. Available: <https://www.virustotal.com/>.
- [83] GData, *GData: Trust in German Sicherheit*. [Online]. Available: <https://www.gdata-software.com/>.
- [84] VirusShare, *VirusShare.com - Because Sharing is Caring*. [Online]. Available: <https://virusshare.com>.
- [85] Osman Arif, *This is the list of all rootkits found so far on github and other sites*. [Online]. Available: <https://github.com/d30sa1/RootKits-List-Download>.
- [86] Victor M. Alvarez, *The pattern matching swiss knife for malware researchers (and everyone else)*. [Online]. Available: <https://virustotal.github.io/yara/>.
- [87] J. Martin, *Yara-rules*, 2020. [Online]. Available: <https://github.com/Yara-Rules/rules/tree/master/malware>.
- [88] NoVirusThanks, *NoVirusThanks*. [Online]. Available: <https://www.novirusthanks.org/>.
- [89] Microsoft, *Introduction to windows service applications*, 2017. [Online]. Available: <https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications>.
- [90] Crazylord, "Playing with windows /dev/(k)mem," 2002. [Online]. Available: <http://phrack.org/issues/59/16.html>.

- [91] Microsoft, “Kernel-mode driver architecture design guide,” 2017. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/>.
- [92] —, “Introduction to device objects,” 2017. [Online]. Available: <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-device-objects>.
- [93] Trendmicro, *Wapomi - threat encyclopedia*, 2014. [Online]. Available: <https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/wapomi>.
- [94] —, *Pe_wapomi.s-o - threat encyclopedia*, 2014. [Online]. Available: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/pe_wapomi.s-o.
- [95] —, *Troj64_winnti.sm - threat encyclopedia*, 2013. [Online]. Available: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/TROJ64_WINNTI.SM/.
- [96] —, *Troj_zusy.a - threat encyclopedia*, 2018. [Online]. Available: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/troj_zusy.a.
- [97] —, *Trojan.win32.mimikatz.adt - threat encyclopedia*, 2019. [Online]. Available: <https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/Trojan.Win32.MIMIKATZ.ADT/>.
- [98] —, *Troj_perseus.b - threat encyclopedia*, 2015. [Online]. Available: https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/troj_perseus.b.
- [99] —, *Bladabindi - threat encyclopedia*, 2019. [Online]. Available: <https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/BLADABINDI/>.
- [100] NXP, *i2c bus specification and user manual*, Apr. 2014. [Online]. Available: <https://i2c.info/i2c-bus-specification>.
- [101] Intel, *Intelligent Platform Management Interface*, 2020. [Online]. Available: <https://www.intel.com/content/www/us/en/products/docs/servers/ipmi/ipmi-home.html>.
- [102] L. Deelen, V. Moonsamy, A. Ehrenstrom, and E. Herder, “Detection of cryptominers in the wild,” 2019.
- [103] Microsoft, *DRIVER_OBJECT structure*, Apr. 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm_driver_object.