Electronic Preprint for Journal of Information Processing Vol.25

Regular Paper

The Evolution of Process Hiding Techniques in Malware – Current Threats and Possible Countermeasures

Sebastian Eresheim[1,a)] Robert Luh[1,b)] Sebastian Schrittwieser[1,c)]

Received: November 26, 2016, Accepted: June 6, 2017

Abstract: Rootkits constitute a significant threat to modern computing and information systems. Since their first appearance in the early 1990's they have steadily evolved, adapting to ever-improving security measures. The main feature rootkits have in common is the ability to hide their malicious presence and activities from the operating system and its legitimate users. In this paper we systematically analyze process hiding techniques routinely used by rootkit malware. We summarize the characteristics of different approaches and discuss their advantages and limitations. Furthermore, we assess detection and prevention techniques introduced in operating systems in response to the threat of hidden malware. The results of our assessments show that defenders still struggle to keep up with rootkit authors. At the same time we see a shift towards powerful VM-based techniques that will continue to evolve over the coming years.

Keywords: rootkit, process hiding, malware

1. Introduction

The origins of hiding processes goes back to the early days of computer science. Outside academics, stealth viruses for MS-DOS appeared in the early 1990's and already showed similar behavior like modern rootkits [55]. The term *rootkit* has its origin in the UNIX world, where "root" is the most privileged user on a system and a root-kit provides an adversary with root privileges. When rootkits evolved from UNIX to the Windows world, the name still remained the same although they did not fulfill their original purpose anymore. Today's rootkits typically use their stealth abilities in order to hide various artifacts such as processes, files, registry keys, drivers, etc. Process hiding is very often the primary hiding ability of rootkits today, therefore this paper uses the term *rootkit* as a synonym for a piece of software that is capable of hiding processes, although not all rootkits actually have this ability. For example, one of the very first rootkits for Windows NT [21], developed by Greg Hoglund, was only about privilege escalation and evading the Kernel security. Hoglund had a big impact on the evolution of the rootkit industry. Not only is the book he co-authored with Jamie Butler [22] a de-facto standard for rootkit research, he also operated rootkit.com, a popular website within the rootkit community for years.

During the early 2000's rootkits as well as anti-rootkit and anti-virus software were entirely focusing on the kernel and the control of its internal structures. In 2006 Rutkowska et al. [51] opened a new chapter in rootkit research with her works on virtualization-based malware. During the evolution of rootkits many techniques have appeared with the goal of leaving as little evidence as possible on the running machine [1], [37], [48]. This paper surveys past and present process hiding techniques and analyzes their capabilities against modern rootkit countermeasures.

[1] Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks, St. Poelten University of Applied Sciences, Austria
[a)] sebastian.eresheim@fhstp.ac.at
[b)] robert.luh@fhstp.ac.at
[c)] sebastian.schrittwieser@fhstp.ac.at

2. Process Hiding Techniques

The execution flow of a Windows API call includes multiple functions from several DLLs (Dynamic Link Library) in both user- and kernel-mode. Specifically, the API call that gathers a list of running processes includes several sub-calls, which lead down into the depths of the OS, until the process manager is called. The process manager, who stores all processes and their associated information, retrieves a list of the running processes and returns it to the function caller. The list might then be processed by all the intermediate steps in between and then handed all the way up again to the caller of the Windows API.

Several of the process hiding techniques discussed in this paper intercept this API call to retrieve all running processes. This technique is also known as a *hook* [44]. They all share the same characteristics and the same goal: filtering the data the API call returns. The required steps for this concept look like the following:

(1) Take over the execution, either the API call itself or a sub-call beneath
(2) Jump to a memory region in which a custom program has been loaded (the steps 4, 5, and 6 are executed by this program)
(3) Call the original function that was supposed to be called
(4) When the original API call returns it should give the actual true list of all running processes
(5) Modify the list in a way that all processes, which should be hidden, are removed from the list

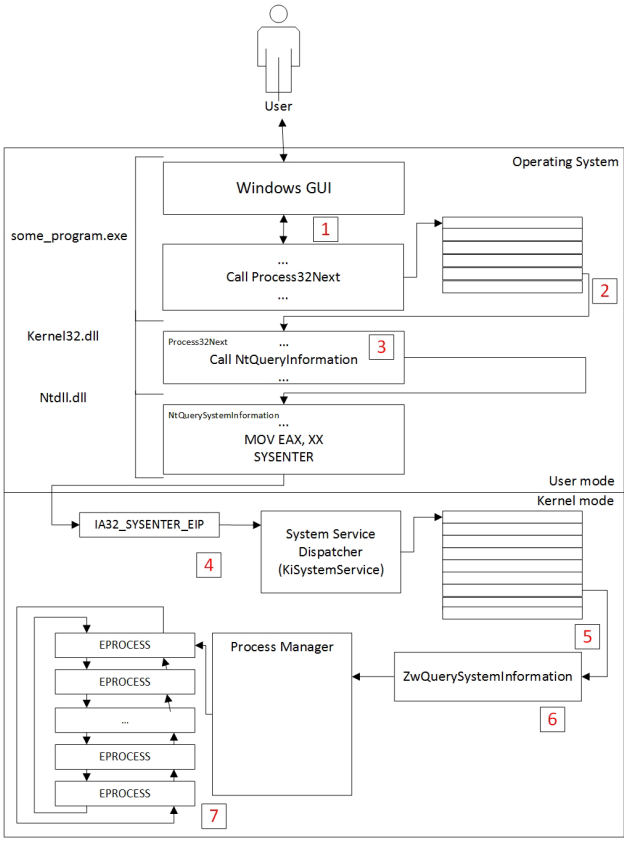

Fig. 1 Interception points of analyzed process hiding techniques within the execution sequence of a Windows API call:
1) UI-Hooking, 2) IAT-Hooking, 3) Inline Function Patching (user-mode), 4) SYSENTER-Hooking, 5) SSDT-Hooking, 6) Inline Function Patching (kernel-mode), 7) DKOM.

(6) Return the list to the caller

In order to be fully functional, a hook usually requires code that intercepts the execution and does the actual filtering. This piece of software is also known as a hook handler and needs to be loaded into the right memory location (depending on the concrete setup/technique) beforehand. Windows API calls can have multiple interception points which are introduced in the following sections. **Figure 1** shows an overview.

2.1 Static Patching

The basic idea of static patching is rather simple: 'alter the code that actually defines the habit you want to change'. This means that DLL- or EXE-files, which hold the binaries of the Windows API such as ntdll.dll, ntoskrnl.exe, etc., are replaced with malicious code [47]. An advanced level on reverse engineering is required to alter official binaries correctly without destroying necessary routines and data structures, especially when it comes to kernel binaries. Therefore this is one of the techniques with the highest required level of technical knowledge.

Detection – Since Windows 2000 Microsoft includes a system called Windows File Protection [13], which protects critical system files from being modified by static patching. Windows File Protection was superseded through Windows Resource Protection in Windows Vista. Additional third party tools, like Tripwire [30], also detect file corruptions by comparing hash values.

2.2 UI-Hooking

UI-Hooking [*1] interferes with the window manager of Windows. The main idea behind this method is to change what a program displays, in contrast to what the program logic holds on information. This means a task manager program still holds the correct list of running processes of the system. The application just simply displays something different to the user. One example of this technique exploits a characteristic of the window class. Every element in a window is a subclass of the window class and therefore has a window procedure. The idea is to overwrite the default window procedure of a window element (for example a ListView) with a custom one, that filters predefined names of processes. For example, a ListView would iterate through its rows and filter them, if they contain certain values. The element that is hooked is the handler of the queued message WM_PAINT, which usually indicates that the receiving window should repaint its content. Instead of repainting the full content, certain values will be left out by the hooked handler. The hook uses the *SetClassLong* function, which is able to set the window procedure for a window class, in order to install its message handler and is injected into processes by using a Windows system hook (*SetWindowsHookEx*). The main advantage of this technique is that it does not attack the OS itself. Unlike other process hiding techniques, no call tables or other system structures are altered. Instead the application is the focus of the attack, which makes it harder to detect.

Detection – UI-Hooking itself is relatively easy to apply, because window procedures and window messages are a fundamental concept of the Windows UI. However, UI-Hooking is highly dependent on the application that needs to be hooked, which is why the developer of an UI-Hooker needs to know the application he wants to hook beforehand. Programs with no UI, or simply a UI that is not suitable for the attack are not affected, because it affects only the UI-Layer and not the program-logic-layer. This means that on a system were UI-Hooking is applied, command line tolls like *tasklist* still show the correct list of running processes. Besides the already mentioned disadvantages, attaching an additional window message handler affects all instances of the window class probably leading to unwanted side effects in other processes with list views. Furthermore this technique only applies to applications that use the Win32 API. Applications which use WinRT, like Universal Windows Platform (UWP) Apps, are not affected anymore.

2.3 Import Address Table-Hooking

Every executable that uses the PE file format [40] and calls at some point a function that does not reside within itself, has an Import Address Table (IAT). The purpose of this table is to tell the running program where to find this function in memory during run-time. The IAT is an array of IMAGE_IMPORT_DESCRIPTOR-structures. Each element of the array holds the name of the DLL it stands for, a time-stamp and a pointer to an array of IMAGE_THUNK_DATA-structures. Before the application is launched these pointer sized structures contain

[*1] https://www.codeproject.com/articles/23686/kitkat-the-lazy-poor-mans-rootkit

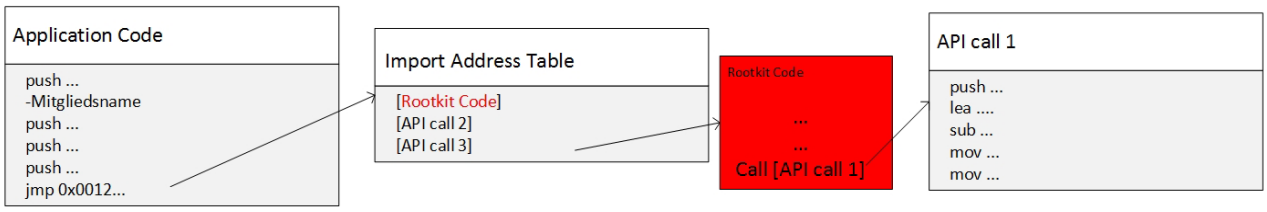

Fig. 2 Setup of an IAT-Hook.

the address of the name of the to-be-imported function. When the executable is loaded the loader overwrites the pointer with the address of the actual function.

The idea of IAT-Hooking is to change one of these addresses to point to an injected, custom function [16]. This allows the intercepting function to apply modifications to the parameters passed to the original function or the results returned by it. In most cases this function will call the original procedure, that was supposed to be called, and then intercept the data it returns [6], [15]. This setup is displayed in **Fig. 2**. As injection method *SetWindowsHookEx* is not sufficient for this technique, because it is restricted to only hook functions that possess a message queue. Alternatively there are different code injection methods to inject the hook handler into the target process. A very popular one was first introduced by Jeffrey Richter [43] and is based upon *CreateRemoteThread* and a custom DLL-file. It includes the following steps:

(1) Acquire space in the target process
(2) Write the name of the DLL-file to the just acquired space
(3) Get address of the Windows API *LoadLibrary* in memory
(4) Call *CreateRemoteThread* with the address of LoadLibrary and the address of the written DLL-name

CreateRemoteThread then creates a new thread in the target process and starts execution with *LoadLibrary*, which loads the attacker's custom library. The hook can either be done directly in the DllMain-Function, or the library sets up an inter-process communication channel and waits for any further commands.

A second way of injecting code into a target process is done with the *WriteProcessMemory* function [34]. The main steps behind this one are:

(1) Read the import address table of the target process
(2) Copy the hook handler to a buffer and patch it with the original import address
(3) Write the hook handler from the buffer to the .text section of the target process using *WriteProcessMemory*
(4) Update the import address using *WriteProcessMemory*

This second approach is less noisy than the *CreateRemoteThread* method, because it creates less artifacts (a separate DLL) and uses less calls to suspicious API-functions (*VirtualAllocEx*, *CreateRemoteThread*, *OpenProcess* with PROCESS_CREATE_THREAD) than the first one. Compared to other process hiding techniques, IAT-Hooking itself is a simple technique, because the knowledge base for this method relies on the functionality of the loader.

A unique feature of IAT-Hooking is that it can be aimed at a single target process. On the other hand, if it is necessary to hook all running processes, this method comes with a pretty high cost. Every code injection bears the risk of corrupting the running process and crash it. Therefore, the more processes need to be injected, the higher is the risk that one crashes and the user get suspicious. A restriction of this method happened with a feature that was introduced with Windows Vista and is called *session 0 isolation* [13]. Since then services do not run in the same session as user applications anymore, but instead in session 0 where they are isolated from other sessions. Processes running in this session cannot be accessed for code injection and thus services are not prone to IAT-Hooking.

Detection – One detection strategy against IAT-Hooking is to look for the exchanged address in the table which as a very different value than the others. This is because the address of the hook handler resides at a very different memory location than the rest of the imported functions. The process of detecting such an anomaly in memory addresses is also easy to automate: if the address in the IAT falls not within the range of the given DLL then the function is hooked.

2.4 Inline Function Patching

Inline Function Patching evolved from IAT-Hooking and is a more subtle approach. One major drawback of IAT-Hooking is that it can be easily tested if the pointer within the IAT points to the memory area of the DLL or not. Inline function patching moves the hook from the IAT right into the function that should be hooked. By using this approach the address within the IAT still stays valid and points to an area within the space of the DLL-file.

The most popular inline function patching method is Detours, a library created by Microsoft Research for the Windows OS [24]. Detours takes the first 5 bytes of a function and replaces them with an unconditional jump instruction. This instruction jumps to a function called the detour function and is the one which intercepts the data passing to and coming from the target function. In order to intercept the data, the target function needs to be called, but because the first few bytes of the target function have already been replaced, it cannot be called without further actions. So before this can be done, the original first 5 bytes need to be executed to create the stack frame of the target function. Therefore they have been saved beforehand in a function called the trampoline function. After the trampoline function has been executed, it jumps to the target function + 5 bytes, in order to avoid to jump into the detour function again. The target function then executes as usual and returns to the detour function. After this procedure, the list of processes is already available and the detour function is able to modify and return it to the source function. Detours, originally provided by Microsoft as a library, is very popular and used in various different projects such as Ref. [29]. **Figure 3** depicts

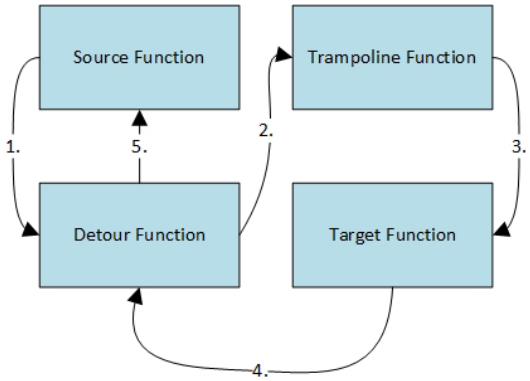

Fig. 3 Execution sequence of a Detours setup.

the execution sequence of a Detours setup.

Debugger aided Hooking, as used by Gomez et al. [18], is an alternative approach for inline function patching. It uses the breakpoints of a debugger to break within the code and lets the debugger take over the execution. These single-byte breakpoint instructions are shorter than the 5-byte jump instructions, which makes it easier to intercept the execution in the middle of a function.

Single Instruction Hooking (*SIH*) [4] takes this approach to the next level. It also makes use of debugger breakpoints, but unlike *Debugger aided Hooking* it does not need a second process to continue the execution after the breakpoint is hit. It is able to do so, because breakpoints usually generate a certain kind of exception when execution hits them. These exceptions are caught in user-mode by *KiUserExceptionDispatcher* and are then managed by the corresponding exception handler. SIH hooks this *KiUserExceptionDispatcher* (by any of the previous methods) and jumps to a custom handler if an exception was raised that corresponds to a breakpoint within the original image.

Inline function patching disguises hooks a little further than just pointing to some arbitrary code area already in a call table. It is less likely from being detected than IAT-Hooking unless someone is explicitly looking for it. Although it disguises hooks further, it does not masquerade their characteristics. It has to leave the path of known/trusted code at some point and execute arbitrary code that has been injected.

Detection – Detours can be spotted by examining the first few bytes of each imported function. If they contain an unconditional jump, then Detours has been installed. However, the jump-instruction can also be placed a little later in the function, making detection more difficult. The major drawback of *Debugger aided Hooking* is its need for a separate debugger process. While Single Instruction Hooking can overcome this drawback, it still leaves the path of trusted execution and jumps to an arbitrary code area. While Detours works in kernel-mode as well, the properties of *Debugger aided Hooking* and *SIH* make them inappropriate for being useful within the kernel.

2.5 SYSENTER-Hooking

SYSENTER-Hooking is a kernel-mode technique. This means the hook handler needs to be injected into the kernel and not into another process. Unnoticed code injection requires kernel exploits, or certain vulnerabilities in third party drivers. The legit way to get executable code into the kernel is to deploy a driver. Microsoft has put a decent amount of effort into drivers meeting certain criteria [3], which is why it is quite hard to get a non-conform driver into the kernel of later versions of Windows. The SYSENTER instruction on modern systems is the only transition from user-mode to kernel-mode. After the processor is switched from execution ring 3 to ring 0, three registers define where the processor carries on with the execution:

- IA32_SYSENTER_CS - defines the code segment in which the execution continues
- IA32_SYSENTER_EIP - defines the instruction which continues (usually the first instruction of *KiSystemService*)
- IA32_SYSENTER_ESP - defines the stack for the kernel execution

These registers can be read and modified with the *rdmsr*- and *wrmsr*-instructions [14], [20], which are privileged instructions and need to be executed from kernel-mode. If an attacker places the address of any arbitrary function in the IA32_SYSENTER_EIP-register it is called for every native API call. This comes with the cost of high responsibility, because every single native API call executes the attacker's code. If the call to the kernel is not dispatched appropriately afterwards or it takes too long, then the whole system might start getting unresponsive, freezes, or even crashes. Only brief and easy tasks should therefore be executed in such hook handlers [27]. With a single hook all of the system's native API calls can be monitored and modified at one's leisure. The advantage here is that there is no need for multiple hooks or multiple code functions if multiple API calls are hooked. In practice though the responsibility and the amount of possible unforeseen events is why this technique is probably not widely used.

Detection – SYSENTER-hooking modifies data that should never be modified. Therefore it is possible to test if it is present or not. If the address in the IA32_SYSENTER_EIP register is not the one of the function which is responsible for dispatching system calls in the current OS version, then something is very likely wrong. On the other hand, all it needs to fix a compromised system is to restore the address of this function back into the register again.

2.6 SSDT-Hooking

After a SYSENTER-instruction the execution usually continues with the system service dispatcher (*KiSystemService*) [54] in kernel-mode. Native API functions can not be called separately, therefore the system service dispatcher does it depending on the service number a user-mode process provides. Similar to the IAT in user-mode, *KiSystemService* works with a call table that contains the addresses of the functions to be executed, which is called System Service Descriptor Table (SSDT). Again, just like in user-mode, this address can be modified and therefore be hooked [2], [8], [31]. In the case of *NtQuerySystemInformation* the hook handler receives all requests for a complete process list and is able to filter the outcome for all user-mode processes. A single hook is therefore sufficient to trick all processes system wide. Inline function patches do apply to this method too, in

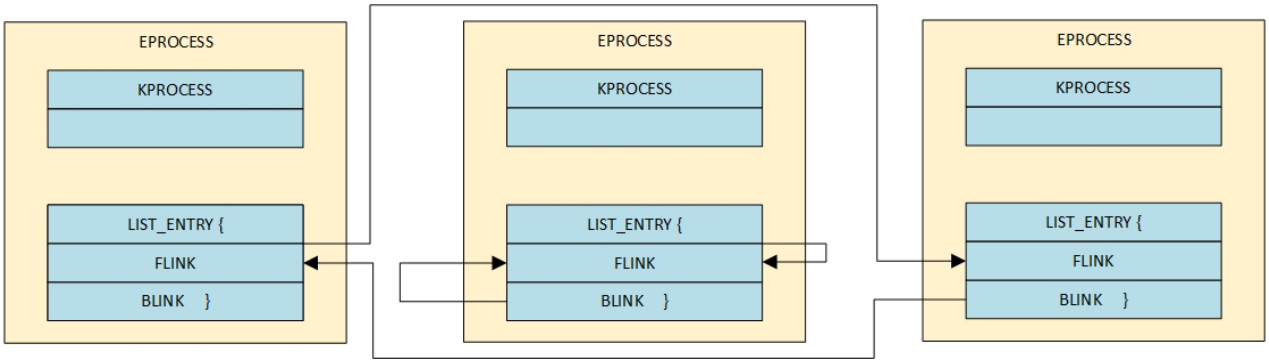

Fig. 4 EPROCESS-structures with a hidden process.

some extent, although this depends on the applied patch-method.

SSDT-hooking is global. It affects all processes alike because if a process demands a certain information from the kernel, the call ends at some point at the system service dispatcher. Therefore, SSDT-Hooking is more efficient than for example IAT-Hooking when all processes of the system should be affected by the hook. The severity of a crash, on the other hand, is more devastating than in user-mode. One of the first projects using this technique was the NTRegmon by Russinovich and Cogswell [12], which monitors all registry activity on a system. It also shows that SSDT-Hooking can be used with a non-malicious intent, for example many antivirus and malware detection systems [19] depend on this technique too.

Detection – This technique is similar to its counterpart in the user-mode, consequently it can also be detected in a similar way. The range of the correct binary can be retrieved and then checked if the address in the SSDT lies within that range. If an inline hook has been applied, a check of the first few bytes of a native API function indicates if an unconditional jump has been inserted. Moreover technologies like the Kernel Patch Protection [35] prevent kernel-mode code and -objects from unauthorized modification.

2.7 Direct Kernel Object Manipulation

Direct Kernel Object Manipulation, or short DKOM, is the art of directly modifying data structures in the kernel without crashing the system. It follows the principle of 'hiding without hooking' [28]. The difference to other techniques operating within the boundaries of the kernel is that it does not intercept the flow of the execution at some point. Instead it directly modifies the objects and data structures that hold that information.

The internal data structure for storing process information is the EPROCESS-structure [49]. Besides scheduling-related information like pointers to thread-structures, it also holds data that is relevant for the user. System calls like *NtQuerySystemInformation* gather process-specific information from this memory structure. For managing purposes, every EPROCESS-structure has two pointers to the next- and previous EPROCESS-structure, the so called forward- and backward link (FLINK/BLINK). Each of them points to the FLINK of the next/previous structure. These pointers connect the structures and line them up to a doubly-linked ring (the BLINK of the first element points to the last one, the FLINK of the last element points to the first one). A list of running processes is then built by iterating through the ring and collecting the necessary information.

As seen in **Fig. 4**, the basic concept of DKOM is to take one EPROCESS structure and unlink it from the ring [7], [9]. This means that the FLINK of the previous structure points to the structure after the to-be-hidden one and the BLINK of the next structure points to the one before it. If the process manager starts at one element and iterates through the ring, it would not pass by the unlinked process and therefore not have it in its list. This unlinked process is then hidden.

Scheduling is not affected, since it does not work with processes but is based on threads. This is why the process is still executed although it is not visible anymore.

Detection – An effective way for identification of these hidden processes is using thread information. Each thread's internal structure includes a pointer to its corresponding EPROCESS-structure. A hooked *SwapContext* function (hooked with inline function patching for example) can then compare if the KTHREAD-structure of the to-be-swapped-in thread is linked to an EPROCESS-block that is appropriately linked to the doubly-linked ring. If not, then the process was hidden on purpose.

2.8 Virtualization Based Rootkits

Virtualization based Rootkits (VMBRs) put the host OS into a virtual machine. This makes it a guest OS without noticing it and the VMBR takes its place as a host OS. Therefore processes that are created by the VMBR (the new host OS) and not the guest OS cannot be traced.

There exist two ways of developing and deploying a VMBR. The first proof of concept for such a rootkit came from Microsoft Research in cooperation with the University of Michigan and was called SubVirt [32]. It was published in 2006, a time when hardware virtualization was not yet available and the researchers focused on the software equivalent. SubVirt used a kernel module to register a *LastChanceShutdownNotification* event handler. Once it is executed, the kernel module copies the VMBR into the active partition of the disk, which causes the system to load the rootkit at the next startup. This setup requires at least administrator privileges on the system during its installation. After the VMBR is started, it loads the target OS in a VM and an attacker OS in a separate one. The attacker OS can then run different kinds of

Table 1 Overview on process hiding techniques.

	Version with built in preventives			Countermeasures			
Process hiding technique	**Windows version**	**32 bit**	**64 bit**	**Hash-based integrity checks**	**In-the-box cross-view analysis**	**Out-of-the-box cross-view analysis**	**Timing analysis**
Static patching	Windows 2k	✓	✓	✓			
UI-Hooking	Windows 7	✓	✓		✓	✓	
IAT-Hooking					✓	✓	
Inline Function Patching					✓	✓	
SYSENTER-Hooking	Windows XP		✓			✓	
SSDT-Hooking	Windows XP		✓			✓	
DKOM	Windows XP		✓			✓	
VMBR							✓

malicious services.

SubVirt was shortly followed by Blue Pill [51]. Blue Pill is based on AMDs first generation of hardware assisted virtualization, which allows hypervisors to remove a lot of the software overhead and form a thin layer between the hardware and the VMs. Blue Pill uses the Secure Virtual Machine (SVM) technology that was introduced with hardware assisted virtualization. Its core is the *vmrun* instruction which starts a new guest VM and takes the address of a 4 KB page as a parameter [14]. This page is the virtual machine control block (VMCB) and sets a few parameters for the newly created VM. Besides different configurations the VMCB also holds the address of the first instruction that is executed within the guest OS. Blue Pill creates such a VMCB and sets the pointer of the first guest OS instruction to the instruction after the call of Blue Pill (which usually would have been executed in the host OS). This way it creates a smooth transition from the host OS to a guest OS for the target.

Vitriol [60] is an equivalent of Blue Pill for the Intel VT-x technology. Vitriol and Blue Pill only differ on processor specific instructions. They are both capable of migrating the host OS into a virtualized guest OS on the fly.

Detection – Every other introduced hiding technique, the hidden process is still in the virtual memory of the system and the information can be extracted by using the right strategy. The hidden process of a VMBR runs in another VM than the user interacts with, thus the user is not able to find this hidden process. This is why the detection of VMBRs focuses entirely on whether the OS the user interacts with is running in a VM or not. If so, the hidden processes can still not be accessed from within the guest OS. To be able to find out what exactly was executed in a hidden way, a dump of the physical memory is necessary.

Software based VMBRs have non-virtualizable instructions in the x86 architecture (such as *SIDT* and *SGDT*) which show inaccuracies and allow the detection of a privileged state. But also the emulation of peripheral devices has its limitations. Therefore a lot of emulated network, SCSI and video cards look very different to real devices.

Hardware VMBRs on the other hand are not prone to discrepancies of peripheral devices due to *device pass through* where real devices are passed forward to guest systems and allow them to use them directly. But even they are affected by timing discrepancies, where certain instructions need a different amount of time depending whether they are executed in a VM or natively. For example a race between two threads executing just *NOP* instructions and the virtualization-sensitive *CPUID* instruction will show a higher ratio on virtualized environments than on physical ones [17]. Additionally, Kyte et al. [33] have shown that it is possible to detect hardware virtual machines based on timing attacks, because of their overhead on context switches between VMs and the code the hypervisor executes.

3. Countermeasures

Over the years various countermeasure-types and -systems have been introduced to the community to cope with rootkits and hidden processes. **Table 1** shows an overview. Integrity checking tools were one of the first available tools to verify the purity of a system, by checking current hash values of important system files against known good values. Tripwire [30] is, as already mentioned, one of them. Since Windows Vista Microsoft protects its critical system files with Windows Resource Protection [13] from unwanted modifications. This does not prevent hooks and modifications during run-time. Probably the most popular approach to cope with run-time modifications is cross-view analysis. For this analysis concept two views of the same data are compared against each other and if any discrepancies occur, a hidden artifact has been discovered. For example a list of all running processes is gathered in user-mode as well as in kernel-mode. If the list does not contain the same elements, then it is most likely that it has been modified somewhere in between. This approach is implemented by multiple tools like Strider Ghostbuster [57], RootkitRevealer [11] and many more [36].

In 2005 concerns were raised [50], because running tools with an in-the-box cross-view approach on a corrupted machine, might still not be able to detect all hidden artifacts successfully. Since then various virtual machine based systems have been introduced and many of them share the same characteristics. They run the corrupted system within a virtual machine and create a view within guest OS of the VM and another one in the hypervisor-layer. This is also why they are called out-of-the-box systems. KernelGuard [42] and VmDetector [58] are two cross-view detection systems using the QEMU hypervisor. Though both of them are implemented to detect hidden artifacts on Linux as a guest OS, their basic principle is also applicable to Windows. On the other hand Lycosid [26] is a typical out-of-the-box cross-view system that operates on a Xen hypervisor. Additionally VmWatcher [25] is able to operate on both hypervisors and is able to use its detection functionality on Windows as well as on Linux distributions. XenKIMONO [41] also performs a cross-view analysis but uses the older in-the-box approach, while it also performs integrity checks of certain kernel structures. It uses the Xen hypervisor

only in order to make sure that predefined processes are not killed by any malicious programs (e.g., make sure an anti-virus software keeps operating).

A different approach to prevent process hiding from happening is to already act at an earlier stage. HookMap [59], which is also using a hypervisor, focuses on not even letting code, belonging to a rootkit, run. To do so kernel level rootkits usually hook a part within the kernel, either a certain data structure (like SSDT-Hooking) or on an instruction basis (e.g., Inline Function Patching). For the first part tripwire-like techniques can be applied, but in order to make sure that no instruction gets misused, HookMap tracks and monitors all jmp- and cmp-instructions and checks whether the corresponding memory address relates to the operation that should be executed. SecVisor [52] and NICKLE [45] even take on a step before and prevent the injection of malicious code into the kernel. SecVisor focuses on the code execution and makes sure only user-approved code is executed. NICKLE uses an additional memory area called 'shadow memory'. Combined with real-time kernel code authentication the system verifies that no injected code is executed. On top of these two systems PoKeR [46] is able to profile the further execution and empowers an analyst with information on the malicious behavior. Finally another very different approach to ensure kernel integrity takes Copilot [39]. The system uses a second (co-)processor to verify the integrity of the instructions executed by the main processor. Instead of taking one step out-of-the-box in a software manner, Copilot relocates the approach to the hardware level.

4. Discussion

Although trying to prevent rootkits from accessing the kernel in the first place seems preferable over detecting it afterwards, it is not sufficient to simply prevent kernel code injections. Rootkits which are based on return-oriented programming (ROP) are capable of hiding processes without introducing any new code to the system as demonstrated in Refs. [23] and [56], which leaves SecVisor and NICKLE vulnerable against these kinds of attacks.

Another major drawback of many of the previously mentioned detection solutions is the systems overhead and its impact on performance. While some of them were already built with performance in mind, others still happen to have a negative influence. While this might be negligible on servers and desktop computers using state-of-the-art hardware, mobile devices, like smart phones, tablets, etc., still lack the capabilities of these high end computers. Mobile devices have in recent years, become a more and more attractive target for rootkits [5], as they even offer new and unique possibilities for rootkits, like spying on conversations via GSM or compromising GPS data. A few first, hypervisor-based, mobile detection systems have already been introduced to the community: XNPro [38] enforces an executable-space protection (memory is not writable and executable at the same time) for the mobile device, which is not default on Android systems, RootGuard [53] matches the origin of privileged instructions to predefined policies and therefore secures rooted Android devices from Malware and OSP [10] aims to provide isolated computing environments for security critical code in an efficient and secure manner for mobile devices. All three of them are either designed to improve the system security of Android devices or have only been tested on them. Since Windows 10 shares the same kernel over all devices and the fact that there have already emerged kernel rootkits for other platforms, a conclusion suggests, that Windows 10 for Mobile is not invulnerable to kernel rootkits either, although more research on this topic is required to bring a definite answer to this question.

Besides desktop computers, notebooks and mobile devices Windows 10 was released for another special type of devices, which are the ones of the internet-of-things (IoT). Again, because of the shared kernel, these devices might be vulnerable to kernel rootkits and process hiding, but unlike mobile devices, these probably will not have the power to run out-of-the-box cross-view detection systems anytime soon. IoT devices are usually at the low-cost end of the price scale, thus a copilot-similar approach, where every IoT device has its own second processor that verifies the integrity of the execution of the main one, might be a rather cost-effective solution. But even if this approach detects the corruption of the kernel, or hidden processes, it would still leave the end user with a corrupted device. New approaches with preventing a corruption rather than detecting it would be preferable. Therefore a certain amount of research is still required to secure IoT devices from kernel mode rootkits and process hiding.

5. Conclusion

In this paper, we presented a systematic summary of past and current process hiding techniques based on their implementations for the Windows operating system. These methods include static patching, user- and kernel-mode hooks in multiple locations across the call-stack of an API call, live patches, direct kernel object manipulation, and virtual machine based rootkits. VMBRs are of particular relevance for modern, VM-based infrastructures as they act even outside of the operating system itself.

While many of the surveyed techniques are primarily used by rootkits to hide their activities from the system, some of them are also utilized by security and malware detection software. Countermeasures range from static integrity analysis and digitally signing of executables, to kernel code injection prevention mechanisms and out-of-the-box cross-view analysis. While most of these countermeasures where designed for desktop operating systems, they fail to address the needs of mobile or IoT devices. In the future, it will be necessary to take a closer look at kernel mode rootkits and process hiding in the mobile and IoT versions of Windows 10. Furthermore, new approaches are necessary to verify the integrity of the kernel of IoT devices.

Acknowledgments The financial support by the Austrian Federal Ministry of Science, Research and Economy and the National Foundation for Research, Technology and Development is gratefully acknowledged.

References

[1] Alexander, J.S., Dean, T. and Knight, S.: Spy vs. spy: Counter-intelligence methods for backtracking malicious intrusions, *Proc. 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pp.1–14, IBM Corp. (2011).

[2] Alzaidi, M., Alasiri, A., Lindskog, D., Zavarsky, P., Ruhl, R. and Alassmi, S.: The study of SSDT Hook through a comparative anal-

ysis between live response and memory image, Information Systems Security Department, Concordia University College of Alberta, Unpublished Master Thesis (2011).
[3] Baxter, J.: Driver Signing changes in Windows 10 (2015).
[4] Berdajs, J. and Bosnić, Z.: Extending applications using an advanced approach to DLL injection and API hooking, *Software: Practice and Experience*, Vol.40, No.7, pp.567–584 (2010).
[5] Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V. and Iftode, L.: Rootkits on smart phones: attacks, implications and opportunities, *Proc. 11th Workshop on Mobile Computing Systems & Applications*, pp.49–54, ACM (2010).
[6] Bozagaç, C.D.: Ghostware and rootkit detection techniques for Windows, PhD Thesis, Bilkent University (2006).
[7] Butler, J. and Hoglund, G.: VICE–catch the hookers, *Black Hat USA*, Vol.61, pp.17–35 (2004).
[8] Butler, J. and Sparks, S.: Windows rootkits of 2005, part two, *Security Focus* (2005).
[9] Butler, J., Undercoffer, J.L. and Pinkston, J.: Hidden processes: The implication for intrusion detection, *Information Assurance Workshop, 2003. IEEE Systems, Man and Cybernetics Society*, IEEE, pp.116–121 (2003).
[10] Cho, Y., Shin, J., Kwon, D., Ham, M., Kim, Y. and Paek, Y.: Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices, *2016 USENIX Annual Technical Conference* (*USENIX ATC 16*), USENIX Association, pp.565–578 (2016).
[11] Cogswell, B. and Russinovich, M.: Rootkitrevealer v1. 71, Rootkit detection tool by Microsoft (2006).
[12] Cogswell, R. and Russinovich, M.: Windows NT System-Call Hooking, *Dr. Dobb's Journal*, Vol.261 (1997).
[13] Conover, M.: Analysis of the Windows Vista security model (2006).
[14] AMD: AMD64 architecture programmer's manual volume 2: System programming (2016).
[15] Erdelyi, G.: Hide'n'seek? anatomy of stealth malware, *Proc. 2004 Black Hat Europe*, pp.147–167 (2004).
[16] Father, H.: Hooking Windows API-Technics of hooking API functions on Windows, *CodeBreakers J.*, Vol.1, No.2 (2004).
[17] Garfinkel, T., Adams, K., Warfield, A. and Franklin, J.: Compatibility Is Not Transparency: VMM Detection Myths and Realities, *HotOS* (2007).
[18] Gomez, D.: Intelligent Debugging for Vulnerability Analysis and Exploit Development, available from ⟨https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-gomez.pdf⟩.
[19] Grégio, A.R., Fernandes Filho, D.S., Afonso, V.M., Santos, R.D., Jino, M. and de Geus, P.L.: Behavioral analysis of malicious code through network traffic and system call monitoring, *SPIE Defense, Security, and Sensing*, Int'l Society for Optics and Photonics (2011).
[20] Guide, P.: Intel® 64 and IA-32 Architectures Software Developer's Manual (2016).
[21] Hoglund, G.: A real NT Rootkit, patching the NT Kernel, *Phrack Magazine*, Vol.9, No.55, pp.55–65 (1999).
[22] Hoglund, G. and Butler, J.: *Rootkits: Subverting the Windows kernel*, Addison-Wesley Professional (2006).
[23] Hund, R., Holz, T. and Freiling, F.C.: Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms, *USENIX Security Symposium*, pp.383–398 (2009).
[24] Hunt, G. and Brubacher, D.: Detours: Binary Interception of Win32 Functions, *3rd USENIX Windows NT Symposium* (1999).
[25] Jiang, X., Wang, X. and Xu, D.: Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction, *Proc. 14th ACM Conference on Computer and Communications Security*, pp.128–138, ACM (2007).
[26] Jones, S.T., Arpaci-Dusseau, A.C. and Arpaci-Dusseau, R.H.: VMM-based hidden process detection and identification using Lycosid, *Proc. 4th ACM SIGPLAN/SIGOPS Int. Conference on Virtual Execution Environments*, pp.91–100, ACM (2008).
[27] Kapoor, A. and Sallam, A.: Rootkits Part 2: A Technical Primer, McAfee (2007).
[28] Kasslin, K., Ståhlberg, M., Larvala, S. and Tikkanen, A.: Hide'n seek revisited–full stealth is back, *Proc. 15th Virus Bulletin Int'l Conference* (2005).
[29] Ki, Y., Kim, E. and Kim, H.K.: A novel approach to detect malware based on API call sequence analysis, *Int'l Journal of Distributed Sensor Networks*, Vol.2015, p.4 (2015).
[30] Kim, G.H. and Spafford, E.H.: The design and implementation of tripwire: A file system integrity checker, *Proc. 2nd ACM Conference on Computer and Communications Security*, pp.18–29 (1994).
[31] Kim, S., Park, J., Lee, K., You, I. and Yim, K.: A brief survey on rootkit techniques in malicious codes, *Journal of Internet Services and Information Security*, Vol.3, No.4, pp.134–147 (2012).
[32] King, S.T. and Chen, P.M.: SubVirt: Implementing malware with virtual machines, *2006 IEEE Symposium on Security and Privacy*, pp.14–327, IEEE (2006).
[33] Kyte, I., Zavarsky, P., Lindskog, D. and Ruhl, R.: Enhanced side-channel analysis method to detect hardware virtualization based rootkits, *2012 World Congress on Internet Security* (*WorldCIS*), pp.192–201, IEEE (2012).
[34] Leitch, J.: IAT Hooking Revisited (2011), available from ⟨https://github.com/m0n0ph1/IAT-Hooking-Revisited⟩.
[35] Lobo, D., Watters, P., Wu, X.-W. and Sun, L.: Windows rootkits: Attacks and countermeasures, *Cybercrime and Trustworthy Computing Workshop* (*CTC*), *2010 Second*, pp.69–78, IEEE (2010).
[36] Molina, D., Zimmerman, M., Roberts, G., Eaddie, M. and Peterson, G.: Timely rootkit detection during live response, *IFIP Int'l Conference on Digital Forensics*, pp.139–148, Springer (2008).
[37] Nerenberg, D.D.: A study of rootkit stealth techniques and associated detection methods, Technical report, DTIC Document (2007).
[38] Nordholz, J., Vetter, J., Peter, M., Junker-Petschick, M. and Danisevskis, J.: Xnpro: low-impact hypervisor-based execution prevention on arm, *Proc. 5th Int'l Workshop on Trustworthy Embedded Devices*, pp.55–64, ACM (2015).
[39] Petroni Jr, N.L., Fraser, T., Molina, J. and Arbaugh, W.A.: Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor, *USENIX Security Symposium*, pp.179–194, San Diego, USA (2004).
[40] Pietrek, M.: Inside windows-an in-depth look into the Win32 portable executable file format, *MSDN magazine*, Vol.17, No.2 (2002).
[41] Quynh, N.A. and Takefuji, Y.: Towards a tamper-resistant kernel rootkit detector, *Proc. 2007 ACM Symposium on Applied Computing*, pp.276–283, ACM (2007).
[42] Rhee, J., Riley, R., Xu, D. and Jiang, X.: Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring, *Int. Conference on Availability, Reliability and Security*, pp.74–81, IEEE (2009).
[43] Richter, J.: Load your 32 bit dll into another process's address space using injlib, *Microsoft Systems Journal-US Edition*, pp.13–40 (1994).
[44] Ries, C.: Inside windows rootkits, Vol.4736, VigilantMinds Inc (2006).
[45] Riley, R., Jiang, X. and Xu, D.: Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing, *Int'l Workshop on Recent Advances in Intrusion Detection*, pp.1–20, Springer (2008).
[46] Riley, R., Jiang, X. and Xu, D.: Multi-aspect profiling of kernel rootkit behavior, *Proc. 4th ACM European Conference on Computer Systems*, pp.47–60, ACM (2009).
[47] Ring, S. and Cole, E.: Taking a lesson from stealthy rootkits, *IEEE Security & Privacy*, Vol.2, No.4, pp.38–45 (2004).
[48] Rudd, E., Rozsa, A., Gunther, M. and Boult, T.: A Survey of Stealth Malware: Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions, *IEEE Communications Surveys & Tutorials*, Vol.19, No.2, pp.1145–1172 (2016).
[49] Russinovich, M.E., Solomon, D.A. and Ionescu, A.: *Windows Internals*, 6th edition, Pearson Education (2012).
[50] Rutkowska, J.: Thoughts about cross-view based rootkit detection, *Retrieved on January*, Vol.14, p.2014 (2005).
[51] Rutkowska, J.: Subverting VistaTM kernel for fun and profit, *Black Hat Briefings* (2006).
[52] Seshadri, A., Luk, M., Qu, N. and Perrig, A.: SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, *ACM SIGOPS Operating Systems Review*, Vol.41, No.6, pp.335–350, ACM (2007).
[53] Shao, Y., Luo, X. and Qian, C.: Rootguard: Protecting rooted android phones, *Computer*, Vol.47, No.6, pp.32–40 (2014).
[54] Sun, H.-M., Wang, H., Wang, K.-H. and Chen, C.-M.: A native apis protection mechanism in the kernel mode against malicious code, *IEEE Trans. Comput.*, Vol.60, No.6, pp.813–823 (2011).
[55] Swimmer, M.: Computer Virus Catalog 1.2: Tequila Virus (1991).
[56] Vogl, S., Pfoh, J., Kittel, T. and Eckert, C.: Persistent Data-only Malware: Function Hooks without Code, *NDSS Symposium* (2014).
[57] Wang, Y.-M., Beck, D., Vo, B., Roussev, R. and Verbowski, C.: Detecting stealth software with strider ghostbuster, *2005 Int'l Conference on Dependable Systems and Networks* (*DSN'05*), pp.368–377, IEEE (2005).
[58] Wang, Y., Hu, C. and Li, B.: Vmdetector: A vmm-based platform to detect hidden process by multi-view comparison, *2011 IEEE 13th Int'l Symposium on High-Assurance Systems Engineering* (*HASE*), pp.307–312, IEEE (2011).
[59] Wang, Z., Jiang, X., Cui, W. and Wang, X.: Countering persistent kernel rootkits through systematic hook discovery, *Int. Workshop on Recent Advances in Intrusion Detection*, pp.21–38, Springer (2008).
[60] Zovi, D.: Hardware virtualization based rootkits, *Black Hat USA* (2006).

Sebastian Eresheim is a research assistant at the Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks (JRC TARGET) of St. Poelten University of Applied Sciences (UAS). He obtained his B.Sc. in IT Security at the UAS and is currently studying for his Master's degree in Information Security. His main research interests lie in systems security and machine learning.

Robert Luh is a researcher at Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks. He obtained his B.Sc. and master's degree in IT Security/Information Security at St. Poelten UAS and is currently working towards his Ph.D. in Cyber Security at De Montfort University, Leicester, UK. His research revolves around automated behavioral malware analysis and digital forensics.

Sebastian Schrittwieser is a professor at St. Poelten University of Applied Sciences as well as head of the Josef Ressel Center for Unified Threat Intelligence on Targeted Attacks. He earned his Ph.D. degree in Computer Science from TU Wien in 2014. His research interests are in software security, code security, digital forensics, and privacy.