Rootkits Survey

A concealment story

Pablo Bravo, Daniel F. García
Department of Informatics
University of Oviedo
Oviedo, Spain
{UO139758, dfgarcia}@uniovi.es

Abstract—**Computer security is an old problem, as old as computers themselves. The evolution of computer threats has also experienced an exponential complexity development, being the last example of that evolution the malware categorized as rootkits or stealth malware. A rootkit is code that is used by an attacker to keep the legitimate users and administrators of a system unaware of the code, and thus the attacker's presence on the compromised system. This paper will discuss the history of rootkits from the basic modification of system binaries to the cutting edge research being develop today. A discussion of each type of rootkit will be followed by an overview of rootkit detection techniques and how to know when a rootkit has been deployed. Finally new techniques and research directions will be discussed.**

Malware; Rootkits; Stealth software; Hooking; Binary patching

I. Introduction

The term "rootkit" has evolved over time, from early *NIX toolsets used to attack mainframes to computer worms like *Stuxnet* [1].

The best way to understand what the term rootkit means is by looking at the role it plays in the phases of a computer attack. Generally, successful targeted computer attacks involve two phases (obviating a prephase of achieving the necessary information to be able to reach the desired machine or network): breaking into the machine (or "rooting" it) and maintain continued access to it in order to perform the supposedly malicious actions with or over it.

Is in this second phase of the attack where the rootkit comes into play. A really successful attack, if not properly concealed, would be ineffective in a short period of time, generally until the system administrator notices the penetration. Of course there are kinds of attack that can accomplish its objectives without the need to conceal its actions, but except for those scenarios (such as information stealing without the need to conceal the fact that the information has been stolen, or denial of service attacks, which doesn't have even the need of rooting the machine under attack), concealing the attack is as important as penetrating the target.

Initially, the term rootkit referred to a tool or suite of tools used to maintain administrative level access on a compromised system. Something as simple as a modified configuration file or binary could be used to allow an attacker uncontrolled access to a target machine for an indeterminate amount of time. The techniques that rootkit authors employ have evolved with computing systems and networks. What began as user-land modified UNIX binaries [27][3][4] has changed over time to kernel-mode code that use complex features of the microprocessors and subvert even the operating system to achieve its goals. They can even be deployed in the form of hypervisors in virtualization-enabled systems to control the operating system itself [2].

In this paper, we will understand a rootkit as any software that enables continued privileged access to a computer while actively hiding its presence and other information from administrators by subverting standard operating system functionality or other applications. Although the definition involves the concept of machine control, the main contribution of a rootkit to its goals is the idea of operation concealment, and this is part of its operation what is generally understood when talking about rootkits [27][28]. Therefore only the techniques which allow the concealment of the rootkit will be presented. Also, when talking about rootkits, we could distinguish between the rootkit itself and an associated piece of software (possibly malicious) which the rootkit would hide. We won't make this distinction and will use the term rootkit interchangeably, meaning both the rootkit only used for the concealment of other piece of software or a malicious software with rootkit capabilities.

II. Origins

Rootkits, in the form of stealth functionality within software, have been in existence since at least 1988. The first notable piece of stealth code was the Brain virus [36]. This virus affected the boot sector of storage media formatted with the DOS File Allocation Table (FAT) system. What makes this virus interesting with regards to rootkit or stealth technologies is that this virus was the first one in existence to include code created to hide the virus from detection. The virus changed the boot sector of floppy disks to spread itself, but it hooked INT 13 so when an attempt of reading the boot sector was made, the virus would present the original one [33]. This means everything looked correct, but in fact there were an ongoing infection in the machine.

The next step were *NIX machines. In an (successful) attempt of obtaining remote root level access, system binaries were substituted by modified versions which performed as the original with subtle changes. Access was maintained by installing backdoors in net-aware applications such as telnet or ftp daemons. Especial programs could be run without noticing thanks to a patched version of *ps*, and logs were modified (or even not produced) by those modified versions of the system binaries. However, this way of attack was easily defeatable, as a simple checksum would do.

The next phase for rootkit attacks was the operating system kernel. This allowed similar results, but is far more difficult to detect. So rootkits started being deployed as kernel modules. This way, rootkits had unrestricted access to the system and they could easily subvert it. The detection of kernel-mode rootkits was not easy and detection software had to be heavily redesigned. And once opened this alternative, rootkit war became a how-deep-you-can-go game. The nearer to the hardware, the more difficult to detect. So research directions went through very specialized software such as rootkits that used System Management Mode (SMM), firmware rootkits, BIOS rootkits, Master Boot Record (MBR) rootkits and when virtualization technology came into play, virtualization rootkits.

Right now, with the rise of mobile technology and the fact that some of these mobile platforms use virtual machine technologies such as Java or .NET, there is renewed interest in the so-called managed-code rootkits.

III. Classification

There are different criteria to classify rootkits. The main ones are:

- Operation layer of the rootkit in the system architecture [6][7].
- Stealth malware taxonomy by Rutkowska [5].

A. Classification based on layers

Computer systems can be described as a set of layers where each one uses services provided by layers below it, similarly to network protocols. Hardware would be the base layer of the computer system, and typically, user programs would fit in the top layer as shown in Fig. 1.

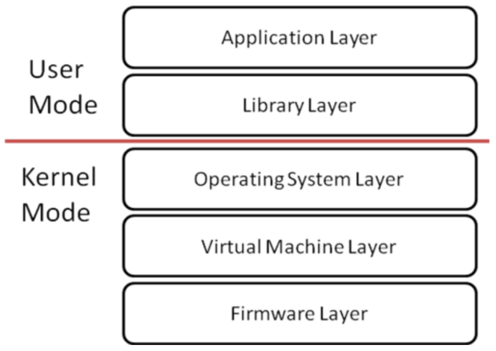

Figure 1. Computer System Layered Architecture.

Upper layers represent application software and lower layers represent the operating system, hypervisor (if the processor supports them) and hardware peripherals firmware code. The lower the layer, the harder to write the software.

Note that this classification is a generalization of a simpler one based on the operating mode of the processor under which the rootkit operates. In this classification, both application and library layer rootkits would be classified as user-mode rootkits, whereas kernel layer rootkits would be classified as kernel-mode rootkits. Usually, also virtualization and firmware layer rootkits would get classified as kernel-mode rootkits, as the processor must operate in the most privileged mode for this kind of rootkits to operate. As a side note, rootkits operating in SMM (System Managed Mode) and Master Boot Record (MBR) rootkits would also be classified as kernel-mode rootkits, and could fit both in the virtualization layer and in the firmware layer, as this kind of rootkits have characteristics of both layers in common.

1) Application layer

These rootkits consist of recompiled binaries (or patched ones) that replace the original user-mode system binaries and operate in a malicious manner. Historically, these rootkits have been also known as Trojans, although some authors disagree [7].

2) Library layer

These rootkits are conceptually equal to rootkits classified as application layer rootkits, except that they are targeted to system-wide dynamic link libraries. This means they are in fact altered versions of the original user-mode system libraries. This way, they can affect the whole system on a process basis, and thanks to dynamic linking, will affect every new process. To explain this a little further, the libraries which are the usual target of these rootkits are loaded by (almost) every process running in the system. These libraries usually control the communication between the processes and the operating system services, so subverting them is a good way to subvert the whole system.

Also, in this category, we would classify those rootkits implemented as dynamic-link libraries that use alternate methods to get loaded in every process, but does not need to subvert original operating system libraries. This kind of rootkits usually modifies the victim process data or code via binary patching to be able to perform its behavior.

3) Kernel layer

Kernel layer rootkits are implemented by replacing or writing new code directly into the running system kernel. This goal is typically achieved by writing a device driver on a Windows system or creating a Loadable Kernel Module (LKM) on a Linux system [3].

When a user-mode process request information or a resource from the operating system kernel, there is a specific path of system calls that must take place. Hooking in any of these places will result in the execution of rootkit code instead of original system requested behavior. This way the system is effectively subverted.

Technically, these rootkits could also replace the kernel image with a recompiled (or patched) one, but this is rare and very uncommon.

4) Virtualization layer

These kinds of rootkits are the most dangerous ones as detection is theoretically impossible, although some author suggest that the detection can be achieved by indirect means. They are implemented as hypervisors and basically run in a layer below the most privileged layer of execution, which is where the operating system resides. These rootkits need the existence of any virtualization technology in the processor such as Intel VT-x or AMD-V. They can virtually make everything they desire as they monitor every operation that takes place in upper levels. As an example, the rootkit could monitor, log, modify or even drop packets of a network connection and the compromised system will not notice anything.

5) Firmware layer

The general concept of firmware rootkits is that firmware can be modified by privileged code, and firmware is a type of code that sooner or later will be executed. This code also has some interesting properties, as for example, it is extremely difficult to write and to detect. Other advantage for a rootkit is that it is located very near the real hardware and can be used to subvert the system without the need to subvert (almost) the operating system. Also, if the firmware software targeted is the BIOS instead of a peripheral firmware, and as this software is executed prior to any operating system initialization, the rootkit potentially can do anything as it could control every aspect in the system.

These kinds of rootkits are very difficult to remove, as reinstallation of operating system, or reformatting the hard disk, or even installing a new hard disk, will not dispose the subversive code. The affected piece of hardware should be replaced or returned to its original state to ensure rootkit removal.

B. Stealth malware classification by Rutkowska

The approach used in Rutkowska's classification is based in the concept of system compromise [5]. According to this idea, there are four distinct types of malware, and each one compromises the system security in different degrees.

It is important to note that Rutkowska's classification is not based in the common distinction of user versus kernel mode.

1) Type 0 malware

This type of malware is not considered malware by Rutkowska's definition as it does not compromise the system in any way. It interacts with the system in a documented way, and it does not subvert the system in any way. It is presented as a legal user-mode application. Of course, the application could perform some behavior considered malicious, as deleting all the documents in a user folder (this would match the AV industry definition), but from a system perspective, the application is legal. Fig. 2 represents visually this kind of malware.

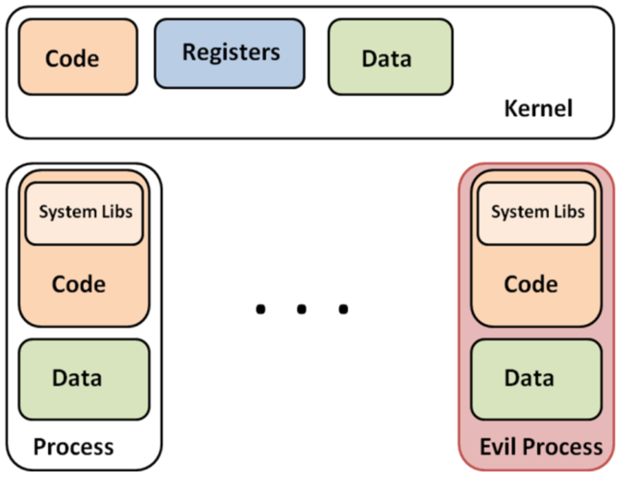

Figure 2. Type 0 malware

2) Type 1 malware

When system read-only resources, such as executable code sections, constant data tables, BIOS code or PCI expansion EEPROMS are modified by a rootkit in such a way that the rootkit achieves its goal, the rootkit is classified as type 1 malware. These rootkits performs their operation by different kinds of hooking. Fig. 3 represents visually this kind of malware.

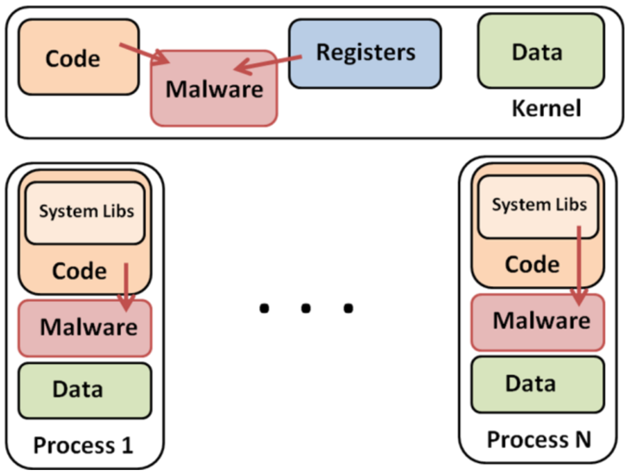

Figure 3. Type 1 malware

3) Type 2 malware

As opposed to type 1 malware, type 2 malware does not modify read-only resources, but instead does modify sections which are by nature dynamic and are subject to be changed by the operating system itself. Such sections can be dynamic data structures which the rootkit modify in such a way that it is able to achieve its goals. Fig. 4 represents visually this kind of malware. Examples of this are DKOM (Direct Kernel Object Manipulation) techniques [22].

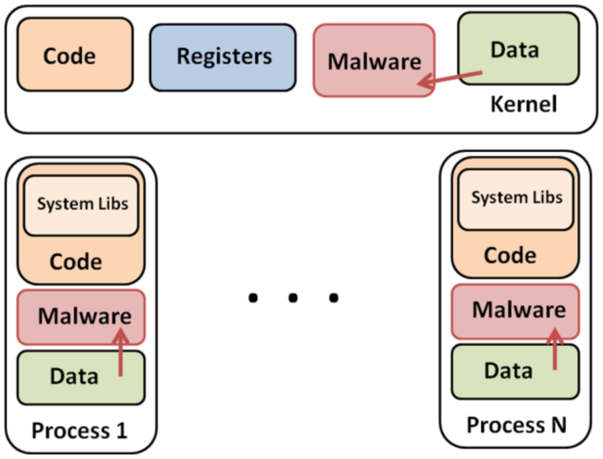

Figure 4. Type 2 malware

4) *Type 3 malware*

Type 3 malware is the more dangerous malware as it does not need to modify in any way a system to compromise it. To work, this kind of rootkits needs some kind of virtualization platform in the system. Fig. 5 represents this malware.

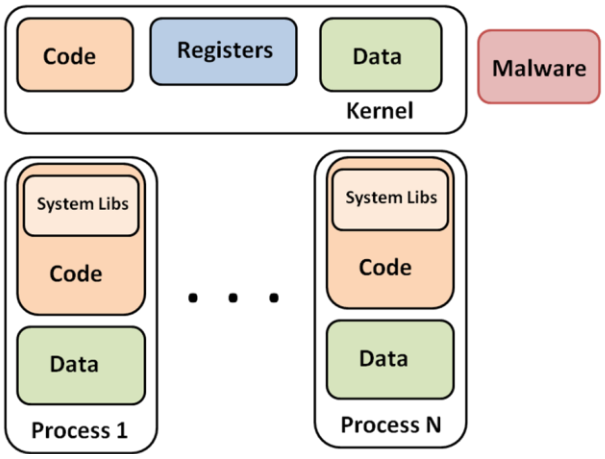

Figure 5. Type 3 malware

The basic way to subvert the system by the rootkit (which takes the form of a hypervisor) is performing its behavior when the normal system operation is intercepted by the virtualization technology.

IV. Techniques Used by Rootkits

This section discusses how rootkits operate to achieve its goals. Usually, the rootkit must be loaded in memory. In the case of user-mode library rootkits, the dynamic library must be loaded in the victim process address space. In the case of kernel-mode rootkits, it must be loaded in the kernel address space. There are a variety of techniques, some legal and some illegal or based on some form of exploit. However, techniques to achieve the rootkit loading in the desired address space are beyond the scope of this paper. Rootkit techniques, understood as a way to achieve rootkit behavior, are based on some way of binary patching, both in user and kernel mode.

A. *Common hooking techniques*

There are two basic ways of code detouring [26]:

- Table based detouring
- Inline code patching detouring

The idea behind table based detouring is that certain pieces of code will invoke API functions resident in other modules whose address is stored in a table, usually filled in the linking process. By overwriting pointers in this table, a rootkit can redirect functions to its own code. A visual explanation can be found in Fig. 6:

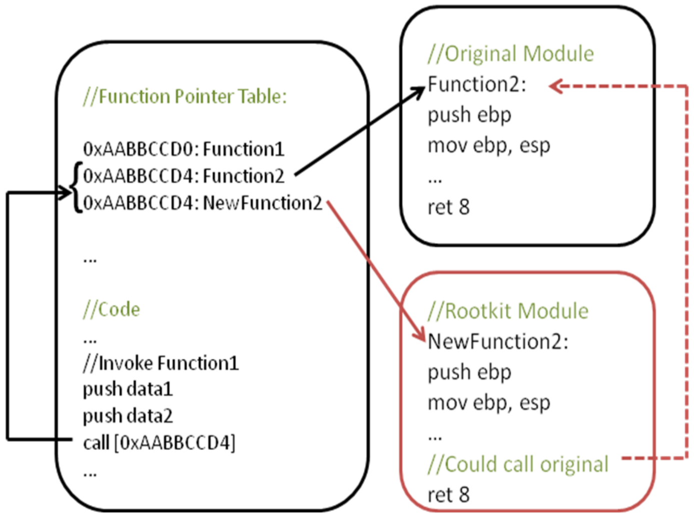

Figure 6. Table hooking

The function table contains different function pointers that are invoked indirectly. At address 0xAABBCCD4, originally resides the function pointer to *Function2*, but the rootkit installs its own function by overwriting this entry in the table. When the code invokes *Function2*, it will in fact invoke *NewFunction2*.

As table-based hooking is relatively easy to detect, the other way of detouring code is by inline hooking. As its name suggest, in-memory code will get patched so it will execute whatever a rootkit writer desires. Although any kind of patching is theoretically possible, the most common way to do it is to overwrite the function prologues of the functions that the rootkit is interested in. The new code usually is some form of jump to a rootkit function, which can return to the original function or directly to the calling code depending on the desired behavior. A visual explanation can be found in Fig. 7.

The *FunctionX* prologue is overwritten by the rootkit, changing these instructions with a jump to rootkit detouring code. The first thing that the detoured rootkit function must do is to set the original stack frame. Later it can perform its behavior and finally it can or cannot return to the original function. In any case it must correctly balance the stack.

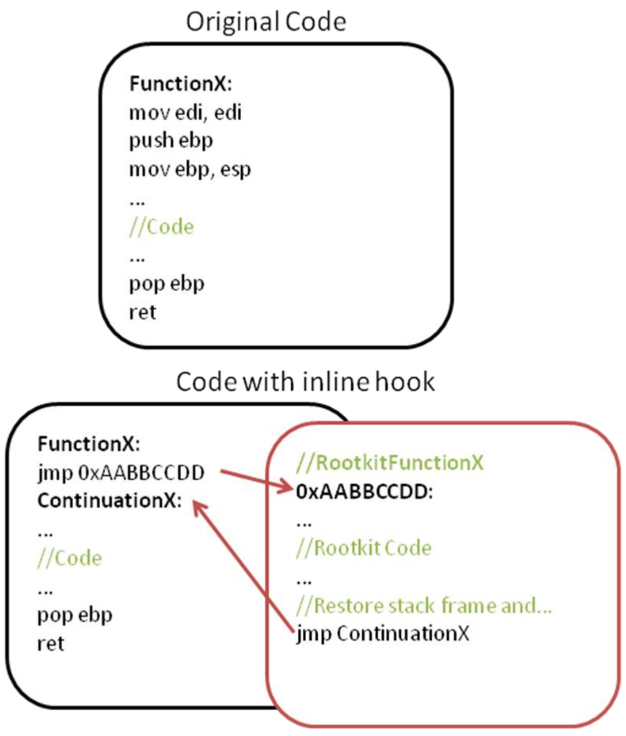

Figure 7. Inline hooking

B. *User mode techniques*

The tables that a library rootkit targets usually are the Import Address Table (IAT) and the Export Address Table (EAT) [27]. The IAT is the data structure within a PE file where the operating system loader fills the addresses of imported functions. The EAT is the table where a module reports its exported functions. This table can also be overwritten by library rootkits, although not very usual, as it will be of use only in case there were new dynamic modules loaded in the process address space. In that case, the new loaded module would obtain the addresses the rootkit overwrote in the EATs when the loader fills its IAT.

Other technique that user-mode rootkits frequently use is inline hooking. This technique achieves the same effect as table hooking, but makes a bit more involved the detection of the hook. Notice that in a table hook, the address of the imported function points into the rootkit module instead of the original one. With inline hooking, all table pointers are correct, but when the API function is invoked, it will jump to the rootkit function.

Anyway, inline hooks are not especially difficult to detect, as a checksum of the code section that takes into account the relocation addresses would be enough to detect the hook. Other detection way is disassembling the prologue of the functions that are suspect of being hooked, but although inline hooking takes place generally at the beginning of the function, nothing prevent an inline hook inside a function, which makes detection quite more difficult. In this case, inline hooking detection could be accomplished by static analysis.

C. *Kernel mode techniques*

Kernel-mode techniques can be classified in two main types coinciding with Rutkowska's classification: type 1 and type 2 techniques.

Type 1 techniques are in essence the same as presented in user-mode techniques. The main difference is that the rootkit driver can hook in many places. Fig. 8 represents the code path for serving a particular service call [26].

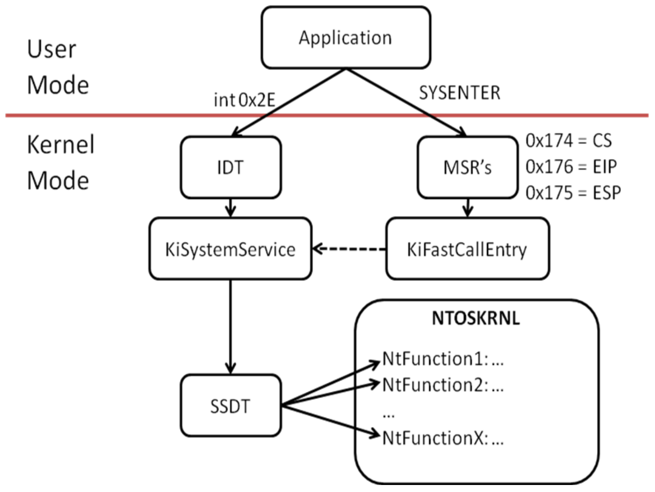

Figure 8. Windows service call path

When an application invokes a system service, it generates a 0x2E interrupt. This value is used to index a processor table denominated Interrupt Descriptor Table (IDT). The processor basically generates a jump to the code function address in the IDT entry. This function is *KiSystemService* which in turn indexes a table denominated System Service Dispatch Table (SSDT). This SSDT table contains the addresses of the services exported by the kernel. The index used in the SSDT comes in the form of a parameter passed from the application. Alternatively, in modern processors, the mechanism is basically the same, but based in the *SYSENTER* instruction, which performs the transition between user and kernel mode. This instruction uses some processor Model Specific Registers (MSRs) which perform a call to a function denominated *KiFastCallEntry*. This function in turn invokes *KiSystemService* to service the call.

1) *Type 1 Techniques*

The techniques used by type 1 rootkits are the exposed before: table hooking and inline hooking. But as can be seen in Fig. 8, there are a lot more places to hook:

- **IDT:** The Interrupt Descriptor Table is a data structure used by Intel processors to route the interrupts that take place in the system. In older systems, interrupts are the preferred portable way to invoke operating system services. The operating system reserves an interrupt number to this task. In Linux Systems, the assigned interrupt is 0x80, while in Windows Systems it is the

0x2E. A rootkit can hijack this interrupt to route the petition of a generic system service to itself. This is not very popular as it involves a lot of work for the rootkit as it must route every function in the system, even those it is not interested in, basically substituting the operating systems routing mechanism [23].

- **SYSENTER:** The SYSENTER instruction is the new path into kernel-mode in newer processors. It is logically equivalent to the interrupt mechanism, although it is implemented in a different way. From the technical point of view, it consults certain processor MSRs and requires the Global Descriptor Table (GDT) to have certain layout. The MSRs contains the kernel memory descriptor, and the EIP of the code to be invoked. A rootkit could detour the path into kernel-mode by writing into the 0x176 MSR (containing the EIP). This technique is logically equivalent to the IDT detouring one.

- **SSDT:** The System Service Descriptor Table is Windows specific data structure which routes system calls to the functions that serves them. It is a system wide table, which makes it very attractive to rootkits, as simple table hooking allows the rootkit to affect the whole system. The main disadvantage is that hooking this table is easily detected (if virtual memory hiding techniques [23] are not involved).

- **IRP Dispatch Table:** The living of a system call does not end when a servicing function in the kernel gets invoked by the SSDT. Depending on the service, it could require the system to communicate with a peripheral. If this is the case, the system will issue an Interrupt Request Packet (IRP) that will travel through the stack of drivers. A visual representation can be seen in Fig. 9 and Fig. 10. The IRP travels through a stack of drivers, each one servicing a different aspect of the request. For example, in the case of a file read, drivers near the top of the stack are the file system drivers, while lower drivers communicate physically with the disk itself. This situation will happen also if a peripheral issues an interrupt. An IRP contains the particular request of the service being invoked, and is filled through its travel with additional data by the drivers encountered in the corresponding stack. A driver contains a table of functions which responds to petitions made by the operating system in certain situations. Obviously, these functions can be detoured by a rootkit to achieve its behavior.

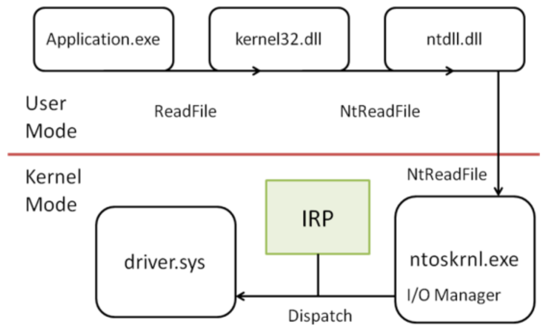

Figure 9. Windows API call serviced by a driver

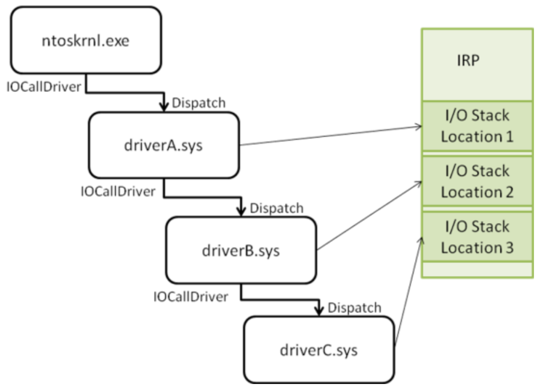

Figure 10. IRP traverse through driver stack

2) Type 2 Techniques

The basic idea behind type 2 techniques is the manipulation of operating system data structures. This way, the rootkit does not need to modify the operating system in any way, as it will operate in a normal way. The difference is in the data that the operating system handles, and, if data is missing, the operating system cannot retrieve it. There are lots of operating system data structures that can be modified, but a very common target that will serve as example is the process list.

Windows systems save the active processes in the system in a doubly-linked list. A rootkit could modify pointers in this list to hide effectively a process. A visual representation can be seen in Fig. 11.

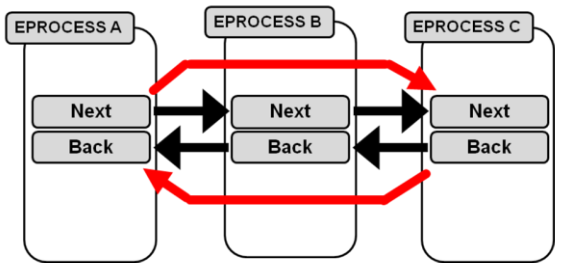

Figure 11. Processes list manipulation

A hypothetical rootkit could manipulate pointers in the EPROCESS A and EPROCESS C data structures to hide the presence of EPROCESS B. The operating system would not notice the existence of process B because it would be found when traversing the EPROCESS list.

3) *Filter drivers*

One of the common extensions supported in the I/O model of Windows Systems, is the concept of filter driver. These drivers are conventional drivers installed in the stack of drivers which can control the flow of IRPs. This means that this type of drivers are basically observers of IRPs that can alter the content of those IRPs and the call flow of the requests across the stack of drivers allowing or denying the IRP to progress.

Initially, these drivers where used on top of file system drivers in order to implement antiviral functionality. Today, a rootkit could install itself as a filter driver and control, for example, the access to its own binary image on disk, making it effectively invisible.

4) *SMM Rootkits*

One idea in rootkit research and development is always try to obtain any kind of advantage over the target software. This lead to the exploring of different alternatives such as using even more privileged processor modes. Software operating in System Management Mode (SMM) has by definition more privilege than the software operating in ring 0 in processor protected mode. Thus it is a good mode to execute a rootkit as it can subvert the operating system in an almost impossible to detect fashion. Further details on this technique can be found in [15][16][17].

5) *MBR Rootkits*

Master Boot Record rootkits are those which overwrite the system's MBR in order to gain execution before the operating system is loaded in memory, in order to subvert it [18]. Generally, these rootkits operate by hooking the INT 13h (which controls the BIOS access to disk) and control when the operating system image is loaded, and overwrite certain elements in the aforementioned operating system image. The real rootkit module usually is a normal operating system citizen (eg. a driver), but the deployment and installation process uses the MBR to allow the rootkit to be loaded in a very stealthy way [19].

6) *ACPI and PCI Rootkits*

In the race of getting unnoticed and lower-level, proof of concept rootkits have been developed which target ACPI (Advanced Configuration Power Interface) and PCI (Peripheral Component Interconnect) BIOSes.

This kind of rootkits are particularly dangerous, as the lower the rootkit is installed, the more difficult both to detect and to correctly erase it. These rootkits can even be written in other language than x86 assembler, as they target the BIOS of different expansion cards, which can use other type of processors. Nonetheless, they can correctly interface with the operating system, as AML (ACPI Machine Language), in the case of ACPI rootkits, is a high level language that allows interacting with system memory and I/O space [14]. The same holds true in case of PCI rootkits [13].

7) *Virtual Machine-based Rootkits*

These rootkits exists in two flavors: software based and hardware based.

Software virtual machine based rootkits generally operate by modifying the boot sequence and execute the operating system into some kind of virtual machine such as VirtualPC [9].

On the other hand, Hardware Virtual Machine (HVM) rootkits use new virtualization technologies such as AMD-V (Pacifica) or Intel VT-x (VanDerPool), which have introduced a new layer of execution (aka Ring -1) that can be subverted and exploited [10].

The main idea is that these rootkits are executed as hypervisor in the target machine, effectively monitoring the execution of the whole system. As those technologies are prepared for executing virtual machines, they provide full virtualization, and that makes impossible to detect directly the existence of a hypervisor from a virtual machine. This feature would make the rootkit truly undetectable [11] [38].

Also, these rootkits could install themselves on the fly, and move the system to a virtual machine without the operating system notice, making them a very dangerous kind of rootkits [12] [34].

D. *Rootkit techniques overview*

An overview of the rootkits most found in the wild and the techniques they use is presented in Table I.

TABLE I
ROOTKIT TECHNIQUES

	User		**Kernel**					
Malware	**IAT**	**Inline**	**SSDT**	**DKOM**	**Filter**	**Memory**	**IRP**	**N°**
NTillusion		X						0
BootRootkit				X				0
ShadowWalker				X		X		0
Vanquish		X						1
DigitalNames			X					1
Sony F4I			X		X			1
WinKRootkit					X			1
PWS-Gogo							X	2
PigSearch			X		X			5
CommonName			X					7
ISearch			X					8
ALI			X					9
He4Hook			X					9
FU				X				10
CoolWeb		X						11
Maddis	X							15
AFX		X						40
PWS-Progent		X						48
Mailbot.c			X					48
Qoolaid	X							58
Vanti		X		X				60
EliteBar	X	X						77
PWS-Goldun			X					223
HackerDefender		X						304
BAC		X	X					394
Feebs		X						556
CKB			X					707
Frequency	3	10	12	4	3	1	1	2595

The last column shows the number of variants existing since 2003. Previous columns represent the techniques categorized in user and kernel modes. Finally, the last line shows the frequency of the technique found in the listed rootkits. Preferred techniques in kernel mode are SSDT hooking and DKOM process hiding, while inline hooking is the most used approach in user mode. The data was extracted from [24].

V. Rootkit Detection Techniques

Current rootkit detection techniques are:

- Behavioral detection
- Integrity checking
- Signature based detection
- Difference based detection

A. Behavioral detection

The base of the behavioral detection is trying to measure the effects that a rootkit might cause in a system. This means it could theoretically detect previously unknown rootkits.

In this field, there are two main streams:

- Detecting diverted execution paths: involves detecting deviations in executed instructions [37] and detecting hooks [29][31] [32][35].
- Detecting alterations in the number, order and frequency of system calls [30].

These techniques may suffer from a high false positive rate, as it is difficult in a production system to correctly measure certain of the needed characteristics. For example, the same system call could execute a different path due to the particular system state. Also, although a hook seems always illegal, legitimate software can also use hooks to implement part of its functionality (in a very unsafe way). Even the operating system installs hooks when patching parts of the kernel.

B. Integrity checking

Integrity checking consists on detecting unauthorized changes to system files or to loaded operating system components in memory.

The operation mode of detectors implementing this technique is as follows: initially they create a baseline database of some kind of hash values. Later, and generally in a periodic fashion, the detector calculates and compares the hashes of the system being monitored against the initial trusted database.

C. Signature based detection

This detection technique is the oldest of all, and has been used to detect every type of malware since the first antivirus scanner was released.

This technique searches memory or the file system for unique byte patterns (known as signatures) that can be found in the code of the rootkit.

This technique is highly accurate, but completely ineffective against unknown rootkits or variants of a known one, or even against obfuscated or packed code.

D. Difference based detection

The idea behind this detection technique is that if a rootkit is present in a system, it is probably hiding something. It is also known as crossview detection [8][31].

To detect a rootkit then, a detector should compare the differences existing between a common system view and a real system view. If differences appear between what the common view shows and what is really going on in the system (provided by the real view), then a rootkit is operating in the system.

The principal problem of these detectors is obtaining correctly the real view of the system. Ideally they should relay only in its own code and directly access the hardware to scan files, memory, registry entries and the like. On the other hand, that is a daunting task and all detectors in this category finally use the operating system in some way. This is dangerous as if the rootkit is operating in the functionality that the detector uses, the detector won't catch the rootkit.

E. Rootkit detection techniques overview

An overview of the implementation of anti-rookit techniques in the most current representative tools is presented in Table II.

TABLE III
ANTI - ROOTKIT TECHNIQUES

Anti Rootkit Tool	**Hooking**	**Heuristic**	**Memory scanning**	**Crossview**	**Signature**
ATool	X				
Avast!		X	X	X	X
AVZ			X	X	X
CMC	X			X	
ComboFix			X	X	X
ESET				X	
F-Secure		X	X	X	X
GMER	X		X	X	
Helios	X			X	
Hidden Finder				X	
Ice Sword	X		X	X	
K X-Ray	X		X	X	
Kernel Detective	X		X	X	
Kaspersky		X	X	X	X
Malware Bytes Antimalware		X	X	X	X
Microsoft Security Essentials		X	X	X	X
McAffee	X			X	
Panda		X	X	X	X
Rootkit Revealer				X	
Rootkit Unhooker	X		X	X	
RootRepeal	X		X	X	
Sophos				X	
Spy Bot		X			X
Moosoft Cleaner		X		X	X
Trend				X	
VBA 32	X	X	X	X	
XeuTr	X		X	X	
Frequency	12	9	16	25	10

The columns referring to hooking and heuristic belong to behavior based detection techniques, while memory scanning belongs both to signature and integrity detection techniques. The most used approach is crossview detection. Surprinsingly, signature based detection is not among the preferred approaches although every antivirus company provides a rootkit detection tool within their products. More information can be found at [25].

VI. RESEARCH

The future evolution of rootkits is determined by the future platforms. In this sense, with the raise of software virtual machine systems (Android, .NET), rootkits will sooner or later target them [21].

Previous work [20] shows how managed code, as every other software, can be subverted and should be of no surprise the fact that malware is already spreading across systems that use managed code, as for example, Android mobiles.

The term Managed Code Rootkit (MCR) is the general denomination for this new type of rootkits. The principal reasons why these rootkits result attractive to attackers is that the attack surface is rapidly growing and they have a single control point because they always target the underlying virtual machine. Moreover, rootkit writers can port their creations among different platforms without rewriting any code [39]. An Android rootkit would work on every Android mobile, as well as a .NET rootkit will work on any .NET platform including desktop computers and Windows Phone 7 mobiles.

REFERENCES

[1] A. Matrosov, E. Rodionov, D. Harley and J. Malcho, “Stuxnet under the microscope”, Technical Report of ESET, 2010.

[2] Symantec, “Windows Rootkit Overview”, Symantec White Paper, 2006.

[3] R. Siles, "Linux kernel rootkits: protecting the system’s ring-zero," White paper of SANS Institute, 2004.

[4] S. Manap, “Rootkit: Attacker undercover tools,” Personal Communication, 2001.

[5] J. Rutkowska, “Introducing stealth malware taxonomy,” White paper of COSEINC Advanced Malware Labs, 2006.

[6] A. Shah, "Analysis of rootkits: Attack approaches and detection mechanisms," Technical Report of Georgia Institute of Technology, 2008.

[7] E. Skoudis and L. Zeltser, Malware: Fighting malicious code. Prentice Hall, 2003.

[8] J. Rutkowska, "Thoughts about crossview based rootkit detection", White paper of InvisibleThings, 2005.

[9] S. King et al., “SubVirt: Implementing malware with virtual machines”, Proceedings from the IEEE Symposium on Security and Privacy, pp. 314-327, 2006

[10] M. Myers and S. Youndt, “An introduction to hardware-assisted virtual machine (HVM) rootkits”, White Paper of Crucial Security, 2007

[11] D. A. D. Zovi, “Hardware Virtualization Rootkits”. [Online]. Available: http://www.theta44.org/software/ HVM_Rootkits_ddz_bh-usa-06.pdf

[12] J. Rutkowska, “Subverting Vista Kernel for Fun and Profit”. [Online]. Available: http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf

[13] J. Heasman, “Implementing and detecting a PCI rootkit,” White paper of Next Generation Security Software Ltd., 2007.

[14] J. Heasman, “Implementing and detecting an ACPI BIOS rootkit,” White paper of Next Generation Security Software Ltd., 2006.

[15] L. Duflot, “Using CPU system management mode to circumvent operating system security functions”, CanSecWest Applied Security Conference, Vancouver, Canada, 2006.

[16] F. Wecherowski, “A real SMM rootkit: Reversing and hooking BIOS SMI handlers,” Phrack Magazine, Volume 13, Issue 66, 2009.

[17] R. Wojtczuk and J. Rutkowska, “Attacking SMM memory via Intel CPU cache poisoning”, White Paper of Invisible Things Lab, 2009.

[18] N. Kumar and V. Kumar, “Vbootkit: Compromising Windows Vista security”, Black Hat USA Conference 2007, Las Vegas, 2007.

[19] E. Florio and K. Kasslin, “Your computer is now stoned (again!): The rise of MBR rootkits”, Technical Report of Symantec.

[20] F. F. Guerra and M. A. Besteiro, “Attacks on .NET – UnCLR future”, Virus Bulletin Conference 2002, Oxfordshire, England, 2002.

[21] J. Bickford et al., “Rootkits on Smart Phones: Attacks, Implications and Opportunities,” in Workshop on Mobile Computing Sys. and Appl. (HotMobile’10). ACM, 2010.

[22] P. Silberman, “FUTo”, Uninformed, Volume3, Article 7, 2006.

[23] J. Butler and S. Sparks, “ShadowWalker: Raising the bar for Windows rootkit detection,” Phrack Magazine, Volume 11, Issue 63, 2005.

[24] A. Kapoor and A. Sallam, “Rootkits Parte 2: Manual Técnico”, Technical Report of McAffee, 2007.

[25] T. M. Arnold, “A comparative analysis of rootkit detection techniques”, M. Thesis, University of Houston Clear Lake, 2011.

[26] B. Blunden, The Rootkit Arsenal: Evasion in the Dark Corners of the System. Jones & Bartlett Publishers, Massachusetts, USA, 2009.

[27] G. Hoglund and J. Butler, Rootkits: Subverting the windows kernel. Addison Wesley, Boston, USA, 2006.

[28] D. Harley and A. Lee, “The root of all evil: Rootkits revealed”, Technical Report of ESET, 2009.

[29] J. Butler and G. Hoglund, "VICE – Catch the hookers! (Plus new rootkit techniques)", Black Hat USA 2004 Conference, Las Vegas, USA, 2004.

[30] J. Rutkowska, "Detecting Windows Server compromises with Patchfinder 2”, Personal Communication, January 2004.

[31] Y.-M. Wang, "Strider Ghostbuster: Why it’s a bad idea for stealth software to hide files", Technical Report MSR-TR-2004-71 of Microsoft, 2004.

[32] H. Yin et al., "HookScout: Proactive binary-centric hook detection”, 7th Conf. on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA’10), Bonn, Germany, 2010.

[33] P. Szor, The Art of Computer Virus Research and Defense. Symantec Press, Addison Wesley, Boston, USA, 2005.

[34] D. Quist and V. Smith, “Detecting the Presence of Virtual Machines Using the Local Data Table”, White Paper of Offensive Computing, 2006

[35] J. Butler and P. Silberman, "RAIDE: Rootkit Analysis IDentification Elimination", Black Hat USA 2006 Conference, Las Vegas, USA, 2006.

[36] T. Shields, “Survey of Rootkit Technologies and Their Impact on Digital Forensics”, Personal Communication, 2008.

[37] J. Rutkowska, “System Virginity Verifier: Defining the Roadmap for Malware Detection on Windows Systems”, Hack in the Box Security Conference, Kuala Lumpur, Malaysia, 2005.

[38] J. Rutkowska and A. Tereshkin, “IsGameOver() Anyone?”, Invisible Things Lab, 2007.

[39] E. Metula, Managed Code Rootkits: Hooking into Runtime Environments. Syngress, 2010.