J Comput Virol (2008) 4:137–157
DOI 10.1007/s11416-007-0069-6

SSTIC 2007 BEST ACADEMIC PAPERS

Rootkit modeling and experiments under Linux

Éric Lacombe · Frédéric Raynal · Vincent Nicomette

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 15 September 2007 / Published online: 25 October 2007

Abstract This article deals with rootkit conception. We show how these particular malicious codes are innovative comparing to usual malware like virus, Trojan horses, etc. From that comparison, we introduce a functional architecture for rootkits. We also propose some criteria to characterize a rootkit and thus, to qualify and assess the different kinds of rootkits. We purposely adopt a global view with respect to this topic, that is, we do not restrict our study to the rootkit software. Namely, we also consider the communication between the attacker and his tool, and the induced interactions with the system. Obviously, we notice that the problems faced up during rootkit conception are close to those of steganography, while however showing the limits of such a comparison. Finally, we present a rootkit paradigm that runs in kernel-mode under Linux and also some new techniques in order to improve its stealth features.

É. Lacombe (✉) · V. Nicomette
LAAS-CNRS, University of Toulouse,
7 Avenue du Colonel Roche,
31077 Toulouse Cedex 4, France
e-mail: eric.lacombe@laas.fr; eric.lacombe@security-labs.org

V. Nicomette
e-mail: vincent.nicomette@laas.fr

F. Raynal
Sogeti ESEC, 6/8 rue Duret,
75116 Paris, France
e-mail: frederic.raynal@security-labs.org

F. Raynal
MISC Magazine, Diamond Editions,
20142, 67603 Sélestat Cedex, France

1 Introduction

Contrary to a widespread idea, an exploit[1] is not systematically required to penetrate an information system. However, once compromised, the intruder carries out some actions to use the system and often to maintain its access into the system behind the legitimate users' back: he uses a *rootkit* to achieve this goal. We call rootkit *a set of modifications that allow an attacker to maintain along the time a fraudulent control of the information system*. To achieve its goal, the rootkit combines different techniques used by different malware approaches (logic bomb, Trojan horse, virus, etc.). In that sense, a rootkit forms a new kind of malicious code even if he borrows from others (cf. Sect. 4 on page 141). In this paper, we develop the fundamental principles that characterize rootkits. We adopt the rootkit designer and the rootkit user's point of view, to identify the different strategies and the technical constraints to be considered.

Regarding the security properties of the rootkit,[2] the intruder needs to preserve at least one way or access to send instructions to the compromised system. Then, he has to make a choice. Does he need to be directly logged in to the system or is it enough for him to send instructions which are executed in an asynchronous way? Does he need to operate stealthily or can his actions be revealed? Are the rootkit internals a problem for his machinations? In order to answer those strategic questions, we detail the intruder's objectives and examine his security expectations to meet them. From that step, we propose in this paper some criteria in order to assess rootkit efficiency. Note that the attacker's objectives

[1] An exploit is a little program that uses a flaw inside a software to penetrate the host system.

[2] Let us stress on the fact that since we adopt the intruder's point of view, the security properties are the attacker's ones.

can greatly vary and hence the capabilities of his rootkit. Nonetheless, some common features must be shared by any rootkit. Indeed, the intruder often needs some means to hide his activity or to carry out operations inside the compromised system.

This paper finally proposes a new method to subvert and divert the Linux kernel. Our approach is focused on invisibility inside the compromised system. Thus, we hide our malicious code into the kernel space and make it a parasite upon a process, which is generally sufficient to compromise the whole system.

This article is divided into five parts. First, we recall in Sect. 2 the technical background required to make this paper self-contained. Then, we summarise in Sect. 3 the evolution of rootkits and recall the injection and diversion mechanisms used by kernel rootkits on Linux. Section 4 deals with the objectives that conduct to rootkit design. We adopt the attacker's point of view in order to figure out how the rootkit can change according to the attacker's objectives and constraints. We propose an analogy with the *dissimulation of information* in order to assess rootkit's efficiency. But we limit ourselves to the invisibility criteria. We show in Sect. 5 the method that we have elaborated to corrupt a Linux kernel and stay as stealth as possible. Finally, we conclude in Sect. 6 and expose the previous contributions in the rootkit field as well as the limits of our approach.

2 Technical background

In this part, we recall some kernel internals and some features which are specific to the x86 architecture.

2.1 Operating system kernel

An operating system kernel is a software that handles the computer hardware (memories, processors, disks, devices, etc.) and that provides an interface to the user, dedicated to easily interact with it. Different kinds of kernel have been designed. Among them, the most widespread are monolithic kernels and micro-kernels. The latter contains what needs to be executed in a privileged mode, only. All the other services are supplied made at the user space level. Thus, the different subsystems of the operating system (virtual memory manager, etc.) are isolated one from the other and communicate with messages conveyed through the micro kernel. At the opposite, in monolithic kernels, the main part of the critical services are implemented at the kernel space level: hardware management (hardware interruption, Input/Output, etc.), memory management, process scheduling, system calls supplied to the user space, etc.

In the rest of this paper, we focus on the Linux kernel which is a modular monolithic kernel. It offers to users many services without enforcing a policy, whenever possible.

2.2 The x86 architecture

The primary goal for the operating system is to manage the hardware upon which it is executed. Thus, the operating system depends on this hardware. However, if the design of operating system is decomposed into layers, hardware specific features are managed by the lower layers while being invisible from the highest ones. The Linux kernel considers that approach and implements most of its services in a hardware-independent way.

We focus on the x86 architecture, which is widespreaded. Although each architecture has its own characteristics, however they share some common features: memory management (less typical for embedded system), processor's privilege levels, communication between the different hardware parts and the software (often through interrupts), etc. On x86, memory management is operated through a segmentation unit (mandatory) and a paging unit (optional). Contrary to the segmentation unit, the paging one is very common to all kind of architectures. As Linux is a multi-platform kernel, the segmentation unit is only used in its bare mode (i.e., the flat mode[3]). This enables to easily cut oneself off from it, to eventually use the paging mechanism only (cf. Fig. 1). The x86 architecture is designed in a 4-ring structure, and each of them represents a processor's specific execution mode. A privilege level is associated to each mode. The most privileged ring is the 0 one—the kernel execution mode—while the least privileged mode is the ring 3 which is limited to user space applications.

The communication between kernel and user space—i.e., switching from ring 0 to ring 3 and conversely—occurs from different events. Among them, interrupts are the most frequent: asynchronous signals (requests) from hardware. They are divided into exceptions (i.e., interrupts from the processor whenever a division by zero or a page fault occurs, etc.), hardware interrupts (i.e., those which are triggered by devices, such as hitting the keyboard for example) and finally software interrupts (i.e., interrupts that are triggered by the software as an application from user space which invokes a system call).

On x86 architecture, those interrupts are numbered from 0 to 255. Each of them is associated to a handler if it has actually been set by the kernel. That handler is a function that is executed when the interruption is raised. All these functions are accessible from a specific table in memory: the IDT (Interrupt Descriptor Table). The kernel fills this table and then loads its address into the processor via the `lidt` instruction.

[3] A single memory segment is set up from physical address 0 to 4 GB.

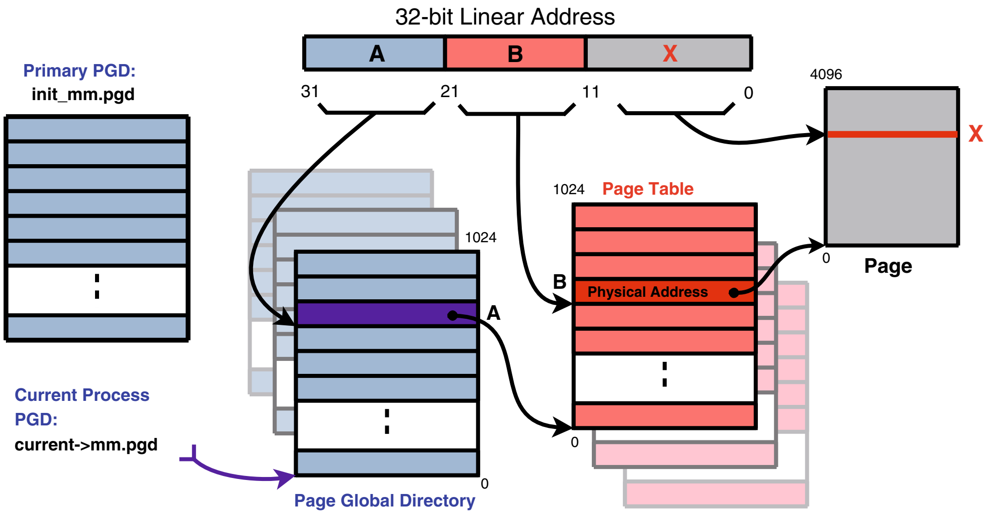


Fig. 1 Paging mechanism

Hardware interrupts or processor's exceptions interrupt user space execution and trap them into the kernel. Hardware interruptions occur asynchronously whereas processor's exceptions trigger synchronously. The kernel handles the interruption or exception and then hands over to the user space. However, before that, the kernel can decide to carry out more urgent tasks. Particularly, in the Linux case, the scheduler verifies whether there exists a more priority process that needs to be executed.

Software interrupts are typically used within system call implementation. Once a user has raised the software interrupt defined by the operating system kernel (`0x80` for Linux), the processor switches to ring 0 and hands over to the kernel entry point in charge of servicing the user request. In order to get an optimized transition, the x86 architecture defines the `sysenter` instruction (which is not an interrupt). Its purpose is to provide an efficient technical mean to achieve system calls. Our work takes that technology into account. Appendix briefly explains how that instruction operates.

The next section gets back those notions in the best practices of rootkit methods.

3 State of the art

We introduce afterward a brief review of rootkits, and show how they have evolved to become more and more efficient.

3.1 Rootkit evolution

When an awkward event occurs in his information system, administrator's first habit is to consult system's logs, to call the `last` command to see who was recently logged in to the system, to call `netstat` to examine the network connections, and so on. A pirate that knows administrators' habits, may try to hide himself within the system by replacing those typical commands: it is the first, trivial form of rootkits. The administrator that uses these modified commands does not detect anything abnormal or unusual.

However, this approach is not reliable for the pirate:

– on former Unix systems, upgrading the system often required to recompile sources as the bandwidth did not allow to download Giga-Bytes of binary; as a consequence, the intruder had to substitute each time his programs with the original ones;
– it is easy to obtain the same information from different paths (e.g., list files with `ls`, `find`, `grep -r`, ...). Thus, the risk for the attacker is to miss one of them;
– usually and especially inside secured environment, *checksums* are generated from hash functions to detect program modifications during forensic analyses for instance.

In order to partially solve these problems and especially the two first ones, rootkits evolved to corrupt a maximal number of programs with fewer efforts. With dynamic shared libraries, intruders have now a good way to regulate their concerns: a modification of functions inside one library reflects itself to all the programs that use that library. Nevertheless, the problems are the same to a lesser extent.

That factorization way (modifying less and corrupting more) went on to the last resource shared by all elements on the system: the kernel. In charge of hardware management, tasks scheduling, memory management, the kernel is the compulsory stage for all operating system's elements. We respectively expose in Sects. 3.2 and 3.3 some injection and diversion methods in kernel space.

New trends come with hardware-assisted virtualization inside general public processors (`VT-x` for Intel and `SVM` for AMD). A hypervisor, at the bottom level, deals with virtual machine management on which operating systems are executed. It is an opportunity for a rootkit to introduce itself at the hypervisor level so to take over the guest systems (which are executed on the host) without infects them [1]. Although, a hypervisor can be integrated to a host system kernel (e.g., KVM for Linux), it owns some interception and modification means conducted by hardware which are out of host kernel bounds. Besides, thanks to hardware-assisted virtualization, the development of a hypervisor is easier and its size has become smaller. This latter characteristic allows a more stealth transmission of a malicious hypervisor from the attacker's system to the system to compromise.

Finally, a stealth-malware categorization has been carried out. Indeed, J. Rutkowska proposes a taxonomy [2] which distinguishes malwares through their corruption type. Thus, three groups of increasing invisibility are set apart:

1. malwares that corrupt fixed elements (e.g., code),
2. those that corrupt non-fixed elements (e.g., data) and,
3. those that act beyond the bounds of the operating system or its applications, without altering them (e.g., hypervisor-rootkits).

3.2 Kernel space injection methods employed by Linux rootkits

We distinguish four different ways to inject code and data into the Linux kernel. The first one uses kernel modules which are dynamically added to Linux [3]. This method is only allowed if Linux kernel module (LKM) support is enabled. In that case, protection mechanisms exist however. To detect malicious kernel modules, it is possible to set up an automatic verification of modules through cryptographic signatures [4]. A functional alternative carries out a static analysis of kernel module behaviors. This is processed before module loading to verify that there is no rootkit inside. That approach of static analysis of binaries has been implemented in [5].

The second approach consists in corrupting and subverting the kernel through the access to the `/dev/kmem` virtual device [6,7]. However, some protection mechanisms like *Grsecurity* can be set up to forbid write or read access to this device. Nonetheless, these typical settings prevent X server to be executed (unless a discretionnary policy on each binary is enforced through the `rsbac` program). Thus, this kind of protection mechanisms is typically disabled on personal computers, work stations, etc.

The third one consists in kernel flaws exploitations. Some allow code injection while others are much more limited. The problem with this approach lies in the fact that it depends on the kernel versions that are affected by the specific exploited flaw.

The last approach uses the specificities of devices that can access to the memory management unit without involving the processor (i.e., DMA, Direct Memory Access). Thus, for instance, the Firewire bus can be used to read or inject data in physical memory without the operating system consent [8,9]. However, J. Rutkowska shows that physical memory reading through DMA can be tricked at software level [10].

The injection method used in our work is based on code injection through `/dev/kmem` (see next section). This approach appears to us to be a good compromise:[4] a workstation with LKM being disabled, remains totally usable, while disabling `/dev/kmem` prevents the execution of typical programs on that kind of system. Finally, kernel flaw exploitation is not enough reliable in time.

3.3 Diversion methods employed by Linux kernel rootkits

The first methods are based on system call table alteration [6,42]. The attacker diverts kernel services towards its malicious functions which have been injected first. However, straightforward detection mechanisms counter this approach. Some compare the current system call addresses with a recording of them which have been made during the system installation. Others verify the relative position of the code between the different system calls.

To mitigate this problem, the attacker goes up and alter the system call handler [6]. First, the attacker copies the system call table and modifies the copy to include pointers to his malicious functions. Then, he modifies the reference of this table used by the system call handler. Thus, the previous detection methods are not efficient anymore since the original table is still in place and is not modified. To counter this approach, it is possible to memorize and check the table's address used by the system call handler.

To execute a system call, a software interrupt at user space level is raised. It triggers the switching into kernel mode. This interrupt can be intercepted. Indeed, the reference on its handler inside the IDT can be replaced by a reference to a malicious one [12]. To detect this fraudulent activity, the reference inside the current IDT must be compared to a backup previously memorized during system installation. Moreover, if the kernel uses the `sysenter` instruction only, this approach cannot be performed.

The IDT also includes the address of the page fault handler. This latter is executed when the processor raises an exception

[4] The last injection mechanism presented above has not been studied at the time of the development of our proof of concept.

due to a forbidden access to a page, or when a page is not present in memory. This interrupt can be intercepted through IDT modification in order to inject malicious code into any process' virtual address space [13]. Generally, any interrupt can be diverted from its original purpose of function.

By going up the execution path, the attacker, this time, copies the IDT. Then, he modifies this copy to eventually load its address into the processor (replacing the previous one) [12]. However, it is easy to retrieve the current address of the IDT with the `sidt` instruction. Thus, to counter this approach, it is sufficient to compare again this address with a backup made at the system installation. No glaring innovation is on the way yet.

In this cat and mouse game, now the attacker comes down inside the low level kernel parts. Especially, the Virtual File System (VFS) functions are diverted thanks to hooking (i.e., the substitution of function pointers) when considering the the *adore-ng* rootkit [14]. Thus, the attacker hides the files and the processes he wishes. However, the same kind of detection methods than before can still be deployed.

In order to put an end to this problem, the attacker can first inject the malicious code into memory. Then, he calls it inside the system call handler before the spot where the system services are called. To achieve this last step, the attacker uses for instance Silvio Cesare's techniques [11,15]. To counter them a more complicated approach than before can be set up. It is an integrity control system which verifies the kernel code integrity by hardware means (TPM, etc.) for instance.

The detection methods that we briefly explained before and others more sophisticated [16] are bypassed by some rootkits. Indeed, what is read by a detection program can be filtered by the rootkit (taking over the `read` and `write` system calls it can trick user space programs). Consequently, it returns information that does not compromise its stealth, only. To have an effective detection, some mechanisms are implemented at the kernel space level to prevent easy reading interceptions. Some products use kernel modules like *Saint Jude* [17].

The Shadow Walker rootkit [18] is one step further in the improvement process of the attacker technology. This rootkit succeeds in hiding its data to the whole system. That is at the user space or the kernel space level, these data are known by the attacker only. This rootkit uses a similar method to the memory protection Linux patch: `PaX`. We come back later on this technique in Sect. 5.4.

The diversion types that we have described so far, take place inside the system and especially inside the kernel. Now, let's see two relative new technologies that are at the system bounds. The first one injects the rootkit inside the master boot record. Thus, it takes over the computer before the operating system. This idea is implemented for instance in the Boot-Root [19] or the Boot Kit [20] rootkits. The second innovation comes from hypervisor rootkits. They implement a malicious hypervisor assisted by hardware. Blue Pill [21] is an example of a such hypervisor rootkit that benefits from AMD's Secure Virtual Machine (SVM) technology. We suppose that no use of hardware virtualization is made by the current operating system. Then, the rootkit declares itself as a hypervisor of the processor and switch the current operating system into a virtual machine without the system being aware of that change. Hence, it takes over the operating system without corrupting it.

We do not describe, in this state of the art, all the services that rootkits are accustomed to provide. However, we explain our approaches, in a more detailed way in Sect. 5, and show how to achieve some of these services. Before going on with that part, we present some issues about rootkit design, in the next section.

4 Rootkit architecture and design

This section presents the fundamental elements that an attacker has to take into account when designing a rootkit. This section is divided into four subsections. In Sect. 4.1, we propose a definition of a rookit, in order to compare this kind of malware to virus, worms, Trojans, etc. The typical architecture of a rootkit is then presented in Sect. 4.2. We introduce the essential components that constitute a rootkit. A rootkit makes it possible for an attacker to durably keep the control over a computer. Thus, the attacker must be able to communicate and interact with the rootkit.

In Sect. 4.3, we detail the possible communication means between the attacker and his rootkit. Finally, in Sect. 4.4, we propose to discuss the evaluation of rootkits. As a matter of fact, an attacker must choose the most appropriate rootkit techniques, according to his objectives. So that he can make this choice, it is important that an attacker can use some objective criteria, that do not exist so far. Thus, we introduce three criteria in order to evaluate a rootkit. Let us note that these criteria may of course be used by the defenders that try to protect their system from rootkits. Better evaluating rootkits enables defenders to better detect and eliminate them.

4.1 Definition of rootkits and comparison with other Malwares

Malware are usually classified into two families [22, Chapts. 3, 4]:

– the self-replicating codes, including virus or worms, that are able to duplicate themselves, into a precise type of file;
– the basic infections, like logical bombs and trojans, that are unable to replicate themselves.

Rookits do not directly belong to this classification of malware. Thus, we do not have a clear and precise characterization of what is actually a rootkit. That is why we propose the following definition:

Definition 1 A *rootkit* is a set of modifications that allows an attacker to durably keep a fraudulent control over an information system.

Several elements characterize a rookit:

– *a set of modifications*: this is a first difference compared to other malware. On one hand, a rootkit is rarely an isolated program, but is in general constituted of several components. On the other hand, the rootkit components are rarely autonomous programs, but rather some modifications made on other components of the system (user space programs, some parts of the kernel, etc.). This kind of modifications is very close to the concept of "program parasitism", which is usually associated to malware that propagate their payload according to nature of their potential targets.
– *a durable control*: other malware do not have a real relationship with respect to time—except in a few cases of logical bombs that execute their payload at a precise date. As for a rootkit, an attacker takes the control over a system in such a way that he can execute operations (information theft, rebound, denial of service, etc.). In order to do so, the attacker must ensure that his access to the system is reliable during a given period of time.
– *a fraudulent control*: it means that the attacker must possess particular privileges to execute his operations whereas he should not be authorized to use the system. Most of the time, the attacker tries to keep the control over the system without the knowledge of the other legitimate users, but this is not necessarily the case. In addition, this control requires interactions and thus, communications, between the attacker and the compromised system.

The usual classification of malware did take into account the inherent characteristics of rootkits and thus is inappropriate. A rootkit is not supposed to replicate itself, so it cannot be included in the set of self-replicating codes. On the other hand, a rookit is supposed to propagate itself within the compromised system. In order to do that, a rootkit very often modifies several parts of the system. For example, the first generation of rootkits modified several binary programs in order to hide their own system and network activity. While a virus is often dedicated to a precise target (binaries only or documents only for example), a rookit may modify any part of the system.

A *logical bomb* is composed of a trigger and a payload. For example, the payload may be activated at a given date or when a user executes a particular operation. Once installed on the system, there is generally no interaction between the owner of the bomb and the bomb itself. This characteristic clearly makes a logical bomb different from a rootkit. Nevertheless, a rookit may include logical bombs (for example, to destroy the system if administrators try to analyse the rootkit).

Trojan horses are often made of a pair of client and server programs. The server is involuntarily installed on the system by a user who believes he is just installing a legitimate program. Spyware are probably the most typical example: when installing a game, a user also installs, without being aware of that, a spying program responsible for collecting information on the system. There are two conceptual differences between a rootkit and a Trojan horse. First of all, the rootkit is voluntarily installed by the attacker. Secondly, a Trojan horse is a "simple" and monolithic program whereas a rootkit is not a program in itself, but rather a set of modifications made on several components of the system.

In conclusion, rookit are closer to simple infections than to self-replicating codes. Nevertheless, this classification is not accurate enough since a rookit includes functionalities of any of the other malicious codes.

4.2 Rootkit architecture

In this section, we identify the elements that compose a rootkit. In order to use a rootkit, the attacker must first install it and then prevent it from being removed. Thus, a rootkit necessarily includes a protection module. Once installed in the system, the rootkit must be able to communicate with the attacker. To this end, the rootkit includes a central module that constitutes an interface for the attacker (i.e., a *backdoor*). Once the intruder is able to communicate with the rootkit, he can "ask" it to execute some operations on the system. A rootkit includes thus one or several modules implementing these services.

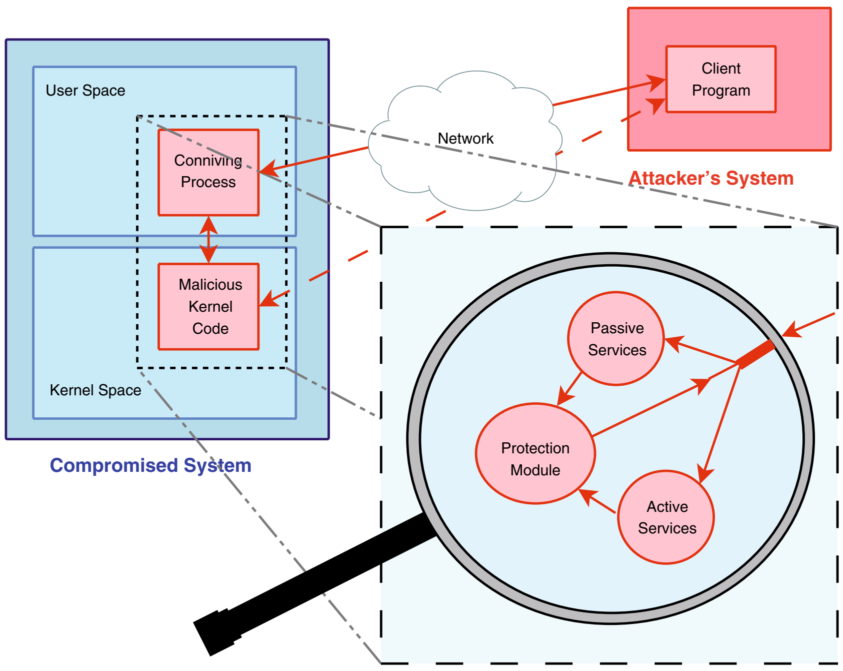
These different modules are described below (cf. Fig. 2).

The Injector. The injector is the mechanism used by the attacker in order to install the rootkit in the system (kernel modules infection, code injection via `/dev/kmem`, etc.). As a matter of fact, whatever the vulnerability exploited by the attacker is (weak passwords or software bug), he has to insert his rootkit in the system. Whatever the rootkit is (user or kernel space, or even hypervisor), it needs an injector to modify only once some structures of the system.[5]

The Protection Module. The protection module makes the rootkit "robust" on the system all along the period required

[5] In the case of our rootkit, as described in Sect. 5, this modification is a bootstrap that is used as a starting point to execute more complex and deep modifications in the system.

Fig. 2 Functional architecture



by the attacker. Several strategies are possible and may be mixed together:

- *Dissimulating the rootkit*:
  Two cases must be distinguished. On the one hand, the rootkit must be hidden while the system is running. On the other hand, if the rootkit is persistent (goes on being active after a system reboot), the rootkit code that needs to be injected in non-volatile memory of the system must be hidden.
- *Making the rootkit resistant*:
  Let us suppose that the rootkit has been detected. A *resistant* rootkit must be able to resist to removal attempts from the administrator. For example, once he is detected, the rootkit may threaten the administrator by pretending to make the bios unusable in case of removal attempts. The purpose is to make the administrator believe (whether this is true or not) that it more dangerous for himself and the users of the system to remove the rootkit than to live with it.
- *Making the rootkit persistent*:
  Making the rootkit *persistent* consists in making it survive even in case of a system reboot. This means that at least a part of the rootkit code must be injected on stable elements of the system.
- *Dissimulating the activity of the attacker*:
  It consists in dissimulating the processes, network *sockets* and files which are used by the attacker. It also consists in cleaning the event log files on the compromised system.

The Backdoor. The backdoor enables the attacker to keep the control over the system and to connect to the services delivered by the rootkit. The backdoor is the central point of the rootkit and it is the interface between the attacker and the rootkit services. The backdoor is characterized by an interaction vector with the system, that we divide into two distinct parts:

- *From the attacker system to the compromised system*
  It consists in the channel used by the attacker from his machine to communicate with the rootkit itself. It may be as simple as a connection to a broken account and as complex as a communication through covert channels. A discussion about this channel is developped in Sect. 4.3.
- *From the compromised system to the rootkit services*
  It characterizes the path used by the rootkit in the compromised system to executed the operations requested by the attacker (such as hooking of the system call table, hooking of the *Virtual File System*, functions hijacking, etc.).

The backdoor is also associated to a mechanism that activates it. We call it the *backdoor-actuator*.

The Services. A rootkit provides services used by the attacker to perform malicious operations in the compromised system. Two categories of services are distinguished:

- *Passive services or spying*:
  These services are used by the attacker to obtain sensitive information that may be found on the compromised system or that may cross the compromised system. A *keylogger* is a typical example of such a service: it intercepts all the key hits on the keyboard of the system.

– *Active services*:
  There are services used by the attacker in order to execute malicious operations on other systems, such as denial of service, information or software removal, etc. They also correspond to services used by the attacker to rebound to other systems in order to go further in his intrusion process.

4.3 Communicating with the rootkit

We identify three different phases of communication during the compromission of a system by a rootkit.

1. during the intrusion phase itself which allows the intruder to enter the target system;
2. once the system is under control, during the downloading of the rootkit;
3. when the attacker uses the services of the rootkit by sending its instructions and receiving the corresponding results (cf. Fig. 2).

This section focuses on the two latest points only, since intrusion technics are out of the scope of this paper. It first should be noted that these communication phases only occur if the intrusion is a remote one. If the attack is a local one, it is obvious that the attacker can install the rookit without any communication; in the same way, he can use the services of the rootkit without any communication. We consider the communications as an external activity for the compromised system. In fact, this is not really the case since these communications provoke modifications on the compromised system itself: sockets are created, input/output statistics of network interfaces are updated, etc. A discussion about these local modifications is held in Sect. 4.3.

The Rootkit Downloading. Once an attacker has successfully broken into a target system, one of his very first tasks is to make his access to the system durable. Two methods can be distinguished:

– *the usual* method: the attacker downloads his rootkit from a remote site using applications like `ftp, sftp, http, etc.` and then installs it;
– *all in memory* method: the attacker downloads his rootkit without writting any byte on the disk: every modification are made in the memory of the compromised process or other processes of the system [23–26].

In both cases, the attacker must connect to an outside host, then transfer data. Whatever the bytes are stored in the file system or in memory does not change the problem from a network point of view: there is a connection to the intruder base, thus, the attacker tries to protect it. In order to do so, he has to face the following problems:

– the base, i.e., the place from which the intruder downloads his tools, may be discovered when used;
– if the network flow is sniffed, it becomes possible for the defendor to rebuild the rootkit and as a consequence, to precisely identify the operations performed by the attacker on the system.

Authors of [27] have proved that these risks are far from being hypothetic. From a network traffic capture, they were able to identify several bases of intruders, to connect to them and then collect several tools. Moreover, the analysis of the rootkit revealed a detailed explication of the compromission process.

In order to protect his base, the attacker may choose to use proxies or other anonymity methods but the required effort seems a little disproportionate compared to the other choice: using the Internet vastness. As a matter of fact, the attacker has a lot of possibilities to store his data on the Internet: newsgroups, ftp sites, P2P networks, etc. A conscientious attacker will store his rootkit into one of these places, just before his attack, and cipher it with a key, as we explain below.

To face the problem of rootkit decoding and flow rebuilding in case of interception, the usal method consists in ciphering. If the network flow is correctly ciphered, even an interception will prevent any possibility of rebuilding, except if the key is also intercepted. This is not a trivial probleme because the key must also be sent through the network and finally stored on the target system to decipher the rootkit.

A solution to this problem has been presented in the context of the Bradley proof-of-concept virus exposed by Filiol [28,29]. This virus uses several ciphering layers, the key to decrypt one layer being calculated from the previous layer. In this way, it is not possible to analyze the virus while the first layer has not calculated the correct key. Moreoever, the whole plaintext code of the virus is never stored in memory, only pieces are, and they remove themselves just as their actions are over. The cryptographic protocol used to obtain this result is based on the notion of *environmental keys* [30]. The authors explain that a ciphered mobile code evolves in a hostile environment and must protect its encryption key. Thus, it must not carry the encryption key but rebuild it from different pieces of information of the environment. Bradley proposes several methods to rebuild the key. In the case of a rootkit, the key must be dependent of a piece of information of the target system and of an external and private secret of the attacker.

Figure 3 illustrates this mechanism. The EVP_i blocks are ciphered with a key calculated by the previous layer. The

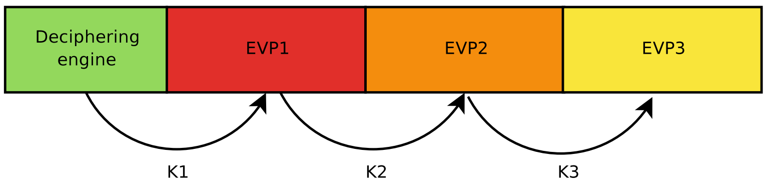


Fig. 3 Structure of Bradley virus

first key is calculated for example thanks to a piece of information characterizing the target system (its IP address or a username) and a private secret (like the hash of a web page or the RR field of a DNS record, etc.).

Thus, even if the rootkit is captured, its analysis is impossible without knowing all the elements to decrypt it. It was prooved in [28,29], that, if the cryptographic protocol is correctly implemented, the complexity of the cryptanalysis is exponential with respect to the length of the key.

The Command Channel. In the command channel, two components need to be protected. The contents of the communication may be protected thanks to ciphering protocols (ssl, ssh, ipsec for example). But this is not sufficient. As a matter of fact, an administrator could be suspicious if his network is abnormaly used during the night for example. Even if ciphered, the unusual quantity of traffic may be detected. Solutions to this problem exist and consist in using covert channels [31–34] or statistical simulability techniques [28,35].

If the rootkit is able to execute operations in kernel mode, the whole TCP/IP stack can be used. If IP datagrams or TCP segments are used, some network equipments may modify the packets headings: *load balancer*, *traffic shaper*, or even *proxies*. Conversely, more and more network equipements try to intercept network flows in order to analyse them and check their conformity to standards. However, these methods are still hardly reliable. For example, a very few network equipments correctly check the TCP session number,[6] which allows the attacker to desynchronize the network flows.

The attacker may also choose to use application protocols, such as DNS or HTTP(S), which are commonly authorized by firewalls. Moreover, using such protocols from the kernel mode will not be detected by local network analysis tools. As a matter of fact, on the sender site, the data are rebuilt in user space, before entering kernel space to finally be sent on the network by the corresponding driver. Conversly, on the recepter side, data are extracted by the driver, pass through the network stack in kernel space and are finally sent to the corresponding application in user space. A code which operates in kernel space is able to bypass all the network analysis mechanisms—such as anti-virus and HIPS—since the majority of these tools only operates in userspace. Thus, a rootkit can become parasitic upon the communication and manipulate it:

– when the data are transmitted; it can add its own bytes to the packets just before transfering them to the driver;
– when the data are received by the driver; it can extract the bytes previously inserted then transfer the packets after cleaning them, to the corresponding application.

Let us note that our rootkit operates in kernel mode and executes its operations before any filter rule. Thus, once a network interface is active, and even if the firewall blocks all input and output connections, our rootkit is able to go on communicating with outside components.

Network Activity Traces in the System. If a rootkit operates in kernel mode (such as our rootkit), it does not use network structures (sockets and ports), thus commands in user space, such as `netstat`, will not detect its presence. Nevertheless, examples such as Sebek [36] show that it is not so easy to dissimulate a kernel rootkit which uses the network [37]. In the first version of Sebek, the statistics regarding packets sent and received increased, even if a sniffer directly connected to the interface did not capture anything.

We do not enter into details regarding this point. Anyway, it is important to keep it in mind.

4.4 Towards rootkit evaluation

Whatever the nature of the attacker is (opportunist, hacktivist, gangster, etc.), his objectives when he uses a rootkit are:

1. gathering information from the compromised system;
2. provoking a denial of service of the system (system including network in that case);
3. taking the control over the system before using it for spying, for making it participate to a distributed denial of service or for turning it into a server with illegal contents;
4. rebounding towards other systems.

Most of time, the attacker tries to dissimulate his intrusion to legitimate users. Thus, it seems obvious that an essential criterion characterizing a rootkit is its degree of *invisibility*

[6] Checking the consistency of sequence and acknowledge numbers of all packets, in addition to many other verifications, may result in to a congestion of the equipment that makes these checks.

(we detail this notion below). Nevertheless, the attacker may not necessarily try to dissimulate his rootkit but, on the other hand, he may have built it in such as way that it is very difficult to remove it without making the whole system in danger. This notion puts the emphasis on a second criterion: the robustness of a rootkit. Finally, whatever its level of invisibility and robustness is, a rootkit modifies the compromised system so that the attacker can keep the control over the system. A third criterion expressing this modification of the compromised system seems relevant in order to evaluate a rootkit.

Let us try to give a definition of these three criteria. A comparison with steganographic systems may be very useful for that [38] purpose. As a matter of fact, the conception of a rookit is close to the conception of a steganographic system: a safe support, called *stegano-medium* is modified in order to dissimulate a secret message. In this analogy, the stegano-medium is the system and the secret message that is dissimulated is a set of data and actions bound to the rootkit. The usual criteria used in steganography are *invisibility* (the secret message must of course be dissimulated as much as possible and it must not be possible to detect the communication itself), *robustness* (a modification of the stegano-medium must not deteriorate the secret message) and *capacity* (which expresses the quantity of information dissimulated in the stegano-medium).

The first two criteria can easily be adapted in the context of the rootkits, as we already explained above. On the other hand, the analogy with steganography stops here because the third criterion (capacity) does not seem to be adapted to the notion of rootkit. Indeed, the information dissimulated in a stegano-medium are passive data whereas the information dissimulated in a system by a rootkit are both active and passive: they are of course modifications of files but also modifications of data in memory as well as activities such a processes. These modifications are executed during the insertion of the rootkit itself and during the execution of all the rootkit malicious activities. Thus, even if a rootkit does dissimulate information on the system, the nature of this information makes difficult to use the capacity criteria as it is.

To characterize a rootkit, we propose to introduce as third criteria a notion representing the “mischievousness” of the rootkit, i.e., representing to what extent the rootkit corrupts the system. This criterion seems close to the notion of virulence as defined by Filiol [22] and that usually applies to virus. This notion is defined as follows:

Definition 2

virulence $= I_v^0 \times I_v^1$

where:

– I_v^0 represents the division of the number of infected files by the virus v by the total number of files in the system.

– I_v^1 represents the division of the number of infected files by the virus v by the number of infectable files by the virus.

In the case of rootkit, this notion must be adapted, because, as we already explained, modifications made by a rootkit are not file modifications only. This notion of virulence seems more adequate than the notion of capacity because it also indicates to what extent the system is corrupted. However, as defined in medical terms, the virulence is the ability of microbes to spread in the organism. The notion of *infestation*, which is defined as the invasion of an organism by a parasite, seems more appropriate.

Let us define the three criteria as follows:

Definition 3 The *invisibility* of a rootkit expresses how difficult it is for a legitimate user of the compromised system to detect the rootkit itself as well as the malicious activities executed by the rootkit.

Definition 4 The *robustness* of a rootkit expresses how difficult it is to remove the rootkit from the infected system.

Definition 5 The *infestation power* of a rootkit expresses the degree of spread of the rootkit in the system, i.e., the quantity of elements of the system that are affected by the rootkit.

Let us note that the notion of invisibility includes the activities that the attacker executes on the compromised system thanks to the rootkit as well as the rootkit itself. Moreover, it includes at the same time the dissimulation of passive objects (files) but also dissimulation of activities (such as processes). Filiol in [28] proposes the definitions of *camouflage* (dissimulation of passive objects) and *furtivity* (dissimulation of activities) [38]. Our concept of invisibility includes both notions of camouflage and furtivity.

The robustness expresses how difficult it is to remove the rootkit while the system is running but also when the system is halted and possibily rebooted later. The two abovedmentioned notions are defined in the general context of the malware removal, in particular for virus and worms. They respectively are the *resistance* and the *persistance*. Our definition of robustness includes these two notions.

The *infestation power* of a rootkit expresses to what extent the system, in which the rootkit is installed, is compromised. It indicates how the rootkit has spread in the system and gives an evaluation of the spread of damage. Indeed, as we already explained in Sect. 4.1, a rootkit is not an autonomous program but rather a set of modifications made on the system.

We can imagine several strategies for a rootkit: it may realize limited modifications, hardly detectable but easy to fix (and thus, focusing on invisibility) or may modify all the components of the system, making it very difficult to clean (and thus, focusing on robustness). A measure must be associated to these three criteria. Indeed, if we are able to associate a measure to each criterion, we are able to evaluate each rootkit and to compare it with others. Cachin in [39] proposes a measure for the security of steganographic systems. He suggests to use the information theory and statistic tests. He introduces a formal definition of a steganographic system and proposes to measure the reliability of such a system by a probabilistic calculation. This approach has been used to define program stealth and address the critical problem of stealth detection in [38].

Finally, to measure the infestation power of a rootkit, integrity tests on files can be a first step. However, usual hash functions enforce the avalanche principle,[7] and they are both sensitive to the least modification to the system.[8] The Levenshtein distance seems to be a better tool: it measures the similarity between two strings (for example the contents of two binary files). Unfortunately, the infestation power of a rootkit is not only due to file modifications. Process modification must also be evaluated, which is more difficult because these modifications are made on data in memory.

In the rest of this article, we focus one the invisibility property.

5 An example of stealth-rootkit design

In this section, we present our stealth rootkit. First, we introduce our subversion approach based on a 2.6 Linux kernel upon an `x86` architecture (this technique constitutes our interaction vector with the system). We focus next on rootkit dissimulation questions and on its backdoor installation. We finish with the deployed methods that dissimulate the attacker's activity[9] that is a part of the protection module inside the functional rootkit architecture presented Sect. 4.2 on page 142.

5.1 Main principle of the kernel part of the backdoor

Our approach consists in corrupting only one process/thread in the system. What is original in this approach is that no other process running on the system see any modification inside its own environment. That is not the case of any of the other previously presented approaches.

[7] A modified input bit of a good hash function must provoke on average a modification of half of the output bits.

[8] They are not adapted to evaluate rootkit. However, from a administrator viewpoint, a hash function is useful to detect that the system has been compromised if it reveals that a file that should not have been modified has actually been modified.

[9] Note that we do not discuss on file hiding in this article.

The per-process syscall hooking technique [40] is not comparable with ours as it acts only at the user space level:[10] no malicious kernel code execution is feasible. Moreover, the modifications carried out on the infected process thread, affect all the threads of this process while the granularity of our approach is the thread.

Our approach subverts the system call 0. It is usually employed by the kernel to restart some interrupted system calls with new parameters and with user space transparency. That is the case for instance of an asleep process (after a `sys_nanosleep` call) that needs to be waken up in order to execute a signal handler. Next to this signal handling, the process must be put to sleep again (if needed) during a shorter period of time: the initial duration minus the execution time of the handler. The `sys_nanosleep` function is thus restarted with this new parameter through the system call 0 mechanism. Given that the system call to restart is specific to each process (or thread) and can change along the time, a reference to this call is saved for each thread. Thus, when a system call (among those that need a restart) is carried out, its address is stored temporarily inside the caller process descriptor. More precisely, this address is stored inside a `thread_info` structure linked to the descriptor (Fig. 4).

Our subversion technique consists in modifying this address. This technique permits to run at the ring 0 level any kernel space function (or arbitrary code injected previously inside the kernel space) from the user space, and that modification is only perceptible from the modified process.

5.2 Design of our rootkit

According to the functional architecture introduced in Sect. 4.2 (cf. Fig. 2), we present the design of our rootkit. It first consists in a kernel space backdoor that is protected with a concealment strategy (cf. Sect. 5.4). This kernel backdoor includes:

1. the code that modifies the system call 0 semantic in order to call any kernel function (cf. Sect. 5.1), and
2. some code that allows the attacker to inject new features if they are not provided by the kernel.

Then, the attacker constructs his operations from the kernel functions and the injected code. He then acts either from within the compromised system, or from another computer outside the network of the compromised system. In the first case, his operations (consisting in sequences of system calls 0) are launched from its session on the compromised system.

[10] The technique changes the instruction employed to call the system in the *glibc* system call wrappers, from `int 0x80` to `int 3`. Thus, when a process executes a system call, it receives a `sigtrap` signal instead of. So, this signal is intercepted by a malicious handler.

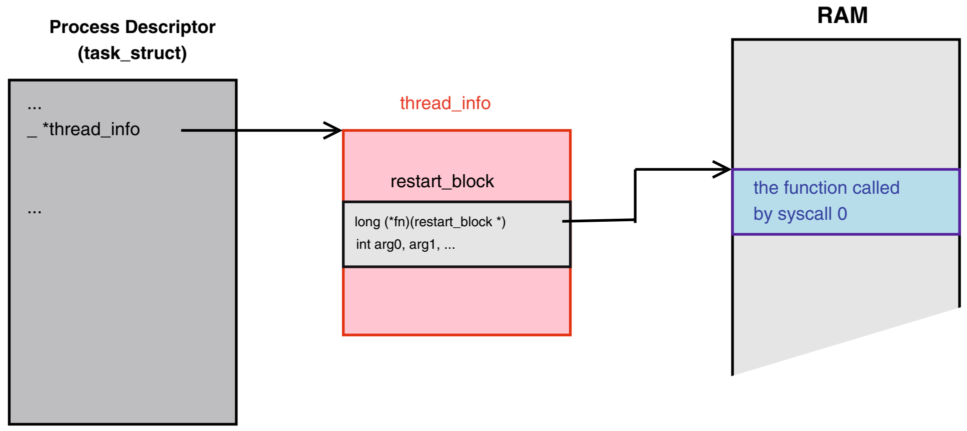


Fig. 4 The process descriptor and the system call 0

Then, the process that supports this session needs to be activated. It is presented in Sect. 5.5. In the last case, a conniving process in the compromised system is set up to relay the commands (i.e., the system calls 0) which has been sent by attacker, to the kernel backdoor.

In the next section, we describe the basic installation mode of our rootkit from the attacker's process on the compromised system. A more advanced mode hides this process before installing the rootkit. However, we do not explain this concealment in the next section but in Sect. 5.7 that deals with stealth.

5.3 Preliminary installation step of the kernel backdoor

From the attacker's process, the `/dev/kmem` virtual device which must access to the kernel address space, is opened. This operation requires appropriate privileges (we suppose that these privileges were obtained by the attacker before the installation of the rootkit). Then, we look for the `get_zeroed_page` primitive (especially its address) thanks to pattern matching through `/dev/kmem`. This primitive consists in the most low-level call of the kernel memory allocator. It books a physical memory page and returns its address. Next, we look for the stack of our process (as well as its decriptor). As the employed technique is quite complex, we explain it in the Appendix.

The next step consists in injecting some code into the kernel stack of our process. This code is only a call to the previous function (`get_zeroed_page`) and returns the allocated page address. It is run through the attacker process by the subversion of the system call 0. For that purpose, we first replace the address of the function called by the system call 0 by the address of the code we have just injected. Then, we run from our process the system call 0 with no parameters. The code we have injected runs itself in ring 0 and then we get the address of the memory page that the kernel has just allocated for us. We inject in this page (through `/dev/kmem` writing) a "trampoline" code that allows to run any kernel function from the user space. We then replace the address used by the system call 0 (that referenced our code inside the stack) by the address of this new code.

From now on, the system call 0 has a new semantic. When we call it, the following parameters must be passed: the address of the kernel primitive (or the address of an arbitrary code we inject in the kernel space) that we want to execute in ring 0, then the parameters (if there are) to pass to it. Then, the trampoline code fetches the parameters transmitted to the system call 0 from the kernel stack of the caller process. Finally, it calls the requested function with the adequate parameters.

Once this diversion mechanism is installed, we can now deploy the logic of our rootkit in the user space. We can for instance create new kernel threads (i.e., ring 0 execution) that run any code of our choice. However, before using the rootkit services (provided by the client program), we conceal it. That is the issue addressed in the next subsection.

5.4 Kernel backdoor concealment

Data and code concealment has to be considered in running state, but also when the system is offline. In this article, we only develop the first case by introducing two methods.

VMALLOC Subversion. This section presents one of our memory concealment approaches that depends on the Memory management Unit (MMU) paging mechanism and its implementation in Linux.

A Page Global Directory (PGD) is associated to each process whose address is loaded into the MMU at each context switch. This mechanism allows processes to be isolated from each others. Each one has its own address space. On `x86` architecture, the 3 to 4 GB interval of the linear address space is reserved to the kernel space and is only accessible in

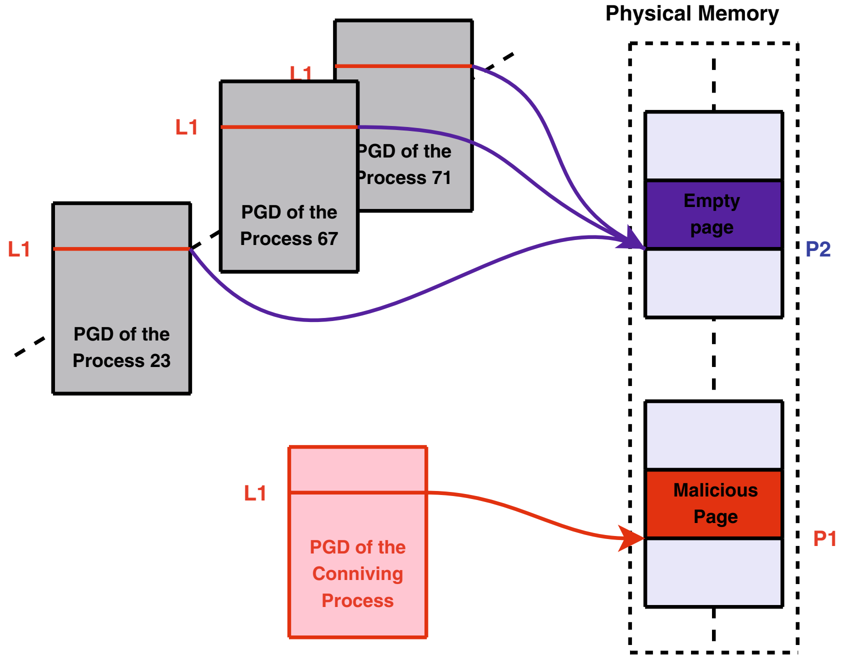


Fig. 5 Exploitation of VMALLOC lazy behavior

ring 0. The linear addresses in this interval are all associated to the same physical addresses whatever the process is.

We use in our approach the VMALLOC non-contiguous memory allocator to allocate a memory page in the kernel space. This page is used to store the malicious code. Its linear address is located inside the VMALLOC-reserved address space area. In this area, contiguous pages of the linear address space (all 4 KB-size) correspond to physical pages that are not necessarily contiguous. So there is no constraint in this area with regard to the association between a linear address and a physical address. That is not the case of the remaining kernel address space where pages are 4 MB in size and are mapped to the same physical pages up to a constant.

The use of the VMALLOC allocator solely provokes a modification of the primary PGD—accessible from the `init_mm` structure (cf. Fig. 1 on page 139)—during a memory allocation. It is a mechanism that is *lazy* in order to improve system performance. Thus, after a process allocate memory with VMALLOC, the kernel does not update the caller's PGD but the primary PGD. Nonetheless, it returns the kernel linear address—that maps in the primary PGD, the beginning of the allocated physical memory—to the caller process. Then, when this process first accesses to this linear address, a page fault is raised and the page fault handler runs and updates the PGD of this process by synchronizing it with the primary PGD.

Our approach (cf. Fig. 5) exploits the lazy behavior of VMALLOC. It consists in booking two memory pages through VMALLOC: one that will contain malicious code and another that will be empty. Let the malicious page have L_1 for its linear address and P_1 for its physical address. Likewise, let the empty page have L_2, P_2 for its linear and physical addresses, respectively. Then the conniving process—which has allocated these pages—looks for the address L_1 inside the primary PGD to get the associated physical address P_1. It does the same for L_2. Then, it changes the primary PGD entry that contains P_1 with P_2. It then updates its own PGD with the mapping $L_1 \leftrightarrow P_1$. In this way, it can accesses the malicious page while the other processes in the system cannot.

Indeed, when they first access to the VMALLOC allocated area at linear address L_1, their PGD is updated with the mapping $L_1 \leftrightarrow P_2$, and so they access to the empty page. Let us notice that the linear addresses of the VMALLOC area can be associated to any physical address without restriction. Thus, whenever a process looks for the physical page we use, it needs to go through all the physical memory.

Modification of the MMU Control Bits. In this subsection, we describe the *Shadow Walker* rootkit [18] because it is a relevant alternative to our approach. (However, it depends on a hardware specificity that we find nevertheless in the vast majority of the `x86` type processors.) The employed technique benefits from the TLB division (Translation Lookaside Buffer—ITLB for instructions and DTLB for data) in order to hide its data from the system. We assume that its data are written into a memory page. Shadow Walker sets this page as non-present (in the corresponding page table) and the associated TLB entry is cleared, which produces a page fault at the first access attempt. Then, the rootkit verifies whether it is an execution or a read/write access or not. In the former case, it loads the ITLB with the malicious page. In the latter case, it loads either an empty page or another page of its choice in the DTLB. Thus, the reading at the malicious page address corresponds to an ordinary page reading, whereas the execution from this address triggers the malicious code.

The paramount stage for a rootkit is to install a backdoor. The next section introduces our various approaches.

5.5 Backdoor-actuators for a process

We present how a process that uses our method, with ordinary user privileges, can interact with the kernel space, that is, how it can activate the backdoor. A benefit of our technique is that we can execute any kernel primitive from an ordinary user process (i.e., not *root*).

Let us now present the two kinds of backdoor-actuators we propose. They are based on kernel thread creation. We assume in these two approaches that the attacker connects to the compromise system to recover its control.

First Backdoor-Actuator. It consists in partially hiding a kernel thread. For that purpose, we unlink it from the thread list to prevent it from being present in the `/proc` filesystem and so from being caught by system activity tools like `ps` or `top`. Our thread is only partially hidden because it is also referenced in a hash table that is used when signals are exchanged between processes (through the `sys_kill` system call) or when a process is traced by another (through the `sys_ptrace` system call).

In this mode, we communicate with our kernel thread from the user space by signals. Thus, our thread sets a signal handler and waits for a specific signal. The purpose of this handler is to answer to a signal emitted from the user space.[11] Once the signal is emitted, it is then handled by the kernel thread which reads through the user processes list until it finds the transmitter. Whenever it finds it, it changes the address used by the system call 0 to the trampoline code address. An improvement to this approach consists in using a signal sequence (rather than only one) with different emission temporal gaps. This improvement guarantees a better authentication to the attacker.

Nevertheless, it is easy to detect this kernel thread concealment, by sending a termination signal to it, that is neither maskable nor interceptable (`SIGKILL`) (e.g., by sending this signal to all the processes). More sophisticated techniques—many threads are considered, each of them restoring the other ones whenever they are killed—would make this active defense fail.

[11] So that a signal sent by the non-root attacker's process is accepted by a kernel thread, we simply need to change the kernel thread UID to the attacker's UID.

Second Backdoor-Actuator. In this approach, we also unlink the created kernel thread from the hash table and so cut off all communication means with it (signals and IPC System V cannot work anymore). However, we cannot totally conceal it. Indeed, so that a process can be executed, it still has to be in the scheduler lists. Nevertheless, we limit in time its stay in these lists by putting it to temporarily and periodically in sleep mode. These concealment cautions being taken, interaction from user space is no longer possible. The thread has then to periodically read through the whole set of the process descriptors[12] to find the one that corresponds to the attacker's identifier (i.e., the UID used by the attacker).

Then, whenever it finds this process descriptor, it changes the system call 0 to the trampoline code address. Then, we inject inside the process the code of the *system call proxy* and make the process run it.

5.6 User-land part of the backdoor

As exposed in Sect. 5.5, the modification required to divert the system call 0 (i.e., the change of an address) was done inside the attacker's process in the compromise system. The user-land part of the backdoor is used when the attacker remotely operates without connecting to the system through a regular user account. Let us note that in this scenario, the backdoor-actuator is limited to an authentication mean between the attacker and the user-land backdoor. We will not further develop this topic.

First Scheme for the User-Land Backdoor. In this scheme, most of the rootkit logic with respect to malicious operations carries out at the client program side, located on the attacker computer. The commands (i.e., some system calls 0) that the attacker wants the compromise machine to execute are relayed from his computer through a system call proxy [23] that is injected in a local process whose system call 0 has been diverted. As only system calls 0 need to be relayed, this mechanism is totally convenient. However, it is also possible to adopt remote userland-execve approaches [26] if programs that are not present in the compromised machine need to be executed (e.g., *nmap*).

Alternative Scheme for the User-Land Backdoor. In this scheme, the previous system call proxy is now executed through a mobile parasitic technique across the running processes in the system. We briefly explain our mobile parasitism algorithm in Sect. 5.7. Likewise, the modification of the system call 0 address follows this strategy. To figure out the benefits of parasitism methods with regard to rootkit activity concealment, we call the reader back to Sect. 5.7. Notice that

[12] In the rest of the paper, we use the term *process descriptor* to also describe a thread descriptor, as it is the same structure in Linux.

now, the backdoor-actuator consists in a specific communication between the attacker and the infested process.

Once the kernel backdoor has been hidden and the userland part of the backdoor has been installed, we can come back to the compromise system and keep control over it. In what follows, we take care of rootkit services to conceal its system activity produced by the attacker (cf. Sect. 5.7).

5.7 System activity concealment

In this section, we introduce three methods to dissimulate the attacker's system activity. We begin with the method that we expose in Sect. 5.5.

Hiding a Process. To hide the attacker's system activity, we have to conceal his tasks from the administrators' view. Thus, we have to hide the processes that are executed by the attacker, if any. In Sect. 5.5, we have exposed a method to hide a process by untying it from the double linked list that goes all over the processes—as well as from the hash table—but also by leaving it temporarily inside the scheduler lists. This method is used by our demonstrator to hide itself in memory. However, the attacker has to run programs within the system and these ones have to be kept hidden.

A naive method is to break the child process links after our process calls `sys_fork` (in order to create a clone of itself) and then to load the program we want to execute through the `sys_execve` system call. However, between the time the process is created and the time we break its links, another process can take over the system and detect it.

To cope with this problem, we duplicate in memory the `sys_fork` system call code and the code of the `copy_process` function (that is called by the former one) that carries out the link operations of the new clone process. It is then sufficient to delete these link operations. Our new modified `sys_fork` function can be used instead of the original system call through the system call 0 diversion, as explained previously. By proceeding in this way, the new process is not linked to its creation time.

An alternative to this last approach is to create another process having the highest priority and which is in charge of breaking the links. In this way, when we create a new process with our demonstrator through the call to `sys_fork`, the third party process takes over the system right after its creation and goes all over the process list to find the new one and to unlink it.[13]

In the next section we use the same method to dissimulate the lineage of a process.

[13] In order to achieve that we indeed verify the relationship between each process and our demonstrator.

Hiding Process' offspring. The hidden programs which are executed by the attacker can also create themselves new processes. We introduce here a method to dynamically hide the offspring of any process. We create a hidden thread that goes periodically all over the process list of the system and checks for each one its relationship with the targeted process so to verify whether it is one of its descendants or not. When it is the case, the process is hidden, otherwise we go on to the next process. The algorithm stops himself to go up the relationship links of a process when it reaches the *idle task* of PID 0 that is the first created task (i.e., the father of all the processes). To improve the task stealth, we put it to sleep by removing it temporarily from the scheduler lists. Indeed, we avoid to monopolize the processor and so to wake up the administrator's suspicion.

Mobile Parasitism of Kernel Threads. Stealth execution methods we have previously proposed are problematic because of the many links that exist between the structures describing the process descriptor. As a consequence, we plan to get rid of that descriptor. Thus, we develop a mobile parasitic algorithm upon kernel threads that run in the compromised system.[14] In what follows, we briefly explain the basics of this algorithm.

It consists in stealing execution cycles to at least two threads. The principle is as follows:

1. we first inject some code in kernel memory,
2. we then divert the thread execution to make it execute this code which,
3. finally executes itself in a loop that makes coming and going all over the infected threads.[15]

This algorithm takes care of keeping malicious code execution as stealth as possible. Thus, we do not want to alter the initial work of the infested threads. Next, we briefly explain how it is accomplished in the two-thread scenario.

Our code is composed of two blocks, each one being associated to an infected thread (depicted by their descriptor in Fig. 6). These blocks are similar and divided up into three parts: the prologue (that recovers the initial state of the thread that runs the other block), the malicious code and the epilogue (that initiates the execution of the other block inside its associated thread).

We show how it operates in Fig. 7. The first-block executes itself on thread 1 behalf, then hands over to the thread 2 that executes the second-block. This latter hands over back to

[14] The implementation is easier with kernel thread than with regular user space process. Indeed, the kernel address space is the same for all its threads.

[15] The instruction pointer of a process waiting to run is stored in its descriptor. Therefore we can change it to the address of our parasitic code.

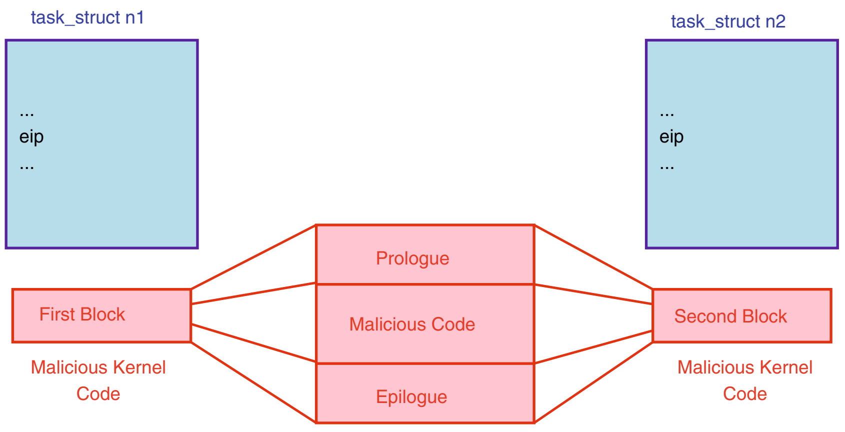


Fig. 6 Execution cycle theft 1/2

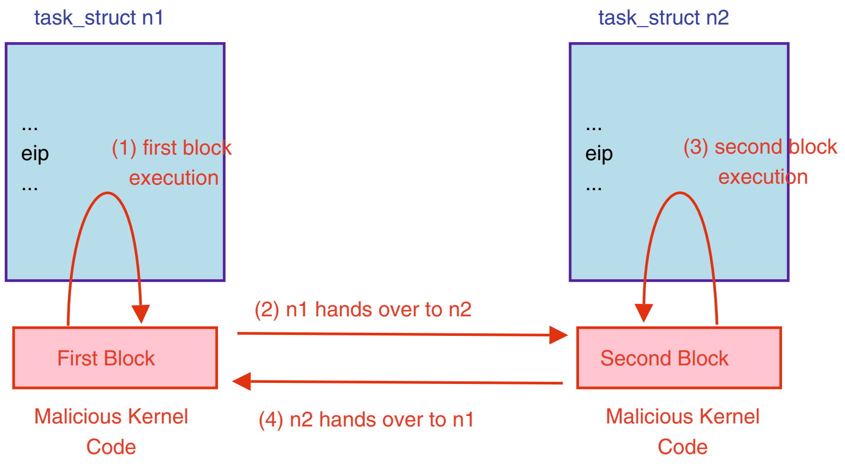


Fig. 7 Execution cycle theft 2/2

the thread 1 that goes on with first-block execution and so on. Thus, we obtain a mobile parasitism that goes from one process to another and conversely. These processes are temporarily infested so they can do the work for what they were originally created. In this way, we hamper the detection of the malicious activity.

6 Experiment synthesis

Let us recall that in this paper we consider x86 architectures, without any hardware extension for virtualization. We do not consequently consider hypervisor-rootkits in our synthesis.

6.1 Compatibility and protection of the rootkit

In order to make our rootkit compatible with many kernel versions, we can take advantage of its *stable sections* of code. They are the code sections that have been sustained and are rarely modified. In addition, the *critical sections* of the kernel are also of interest. They are the ones (i.e., the essential parts) that have a great impact on global system efficiency. Thus, these code sections are implemented with a great care and are seldom modified along their life-cycle. It follows that the majority of these sections are also stable. We explain next why it is a benefit for our rootkit.

To favor the concealment of our rootkit or of its activities (i.e., to favor the invisibility criterion) it is relevant to act upon the environment of the kernel critical sections (i.e., the data the kernel manipulates or uses) without modifying them. Indeed, as we have just seen, the addition of code in these sections can be catastrophic for the system performance. Thus, administrators have to cope with a painful choice. In case of they want to implement some detection or prevention mechanisms like data filtering in these sections, it results in a totally unusable system.[16] We illustrate that with our original interaction approach with the compromised system kernel. We divert a kernel critical section (the system call 0) without modifying it (cf. Sect. 5.1). We only act upon the data it uses.

By acting upon kernel critical sections, we make the implementation of protection mechanisms difficult and in this way

[16] The impact on the system greatly depends on the critical section that is modified. However, the modification of many of them results in system unusability.

we favor rootkit *robustness*. Moreover, the rootkit may find it very beneficial to work with these sections in which its activity will not significantly alter its *invisibility*.

6.2 Contribution

In this section we go all over the contributions of our work and compare them with the existing approaches.

Kernel Part of the Backdoor. In our approach, the necessary steps to install the rootkit are only read and write accesses to the `/dev/kmem` device. That constitutes the preliminary operations to settle a *bootstrap*. Thereafter, the remaining part of the installation is carried out through the system call 0, including the future injections into the kernel space. Thus, the attacker's activity is hidden from the beginning, i.e., from the rootkit installation.

We now detail the differences between our approach and the usual techniques employed in known rootkits:

- *Local visibility of modifications*:
  Our approach modifies the system behavior only for the process from which we work. The system call 0 operation is unaffected for all other processes on the system. Thus, we name it *local* diversion, to be compared with *hooking* and *hijacking* methods employed by the rootkits that affect the system globally. Let us recall that the *per-process syscall hooking* technique [40] is not comparable with our method (cf. Sect. 5.1) since it does not allow any kind of ring 0 execution at all (it is a user space level technique that does not deal with kernel space).
- *Data corruption, not code corruption*:
  Many rootkits alter the kernel code. Our approach only acts upon the code environment, i.e., upon the data it manipulates. From the J. Rutkowska taxonomy [2], our rootkit is type II: the kernel corruption is carried out inside non-fixed section (for instance, inside data areas). Relatively to the type II rootkits, we only modify one variable: a function pointer inside a thread descriptor. Then we add some code sections to the kernel inside areas that we allocate before through kernel mechanisms.
- *Arbitrary code execution in ring 0*:
  The system call 0 diversion allows to execute the code of our choice in ring 0. But it is only after we settle our "trampoline" code that we can execute any kernel functions or some code that we have injected before.
- *Use of the kernel-provided mechanisms in the compromised system*:
  The known approaches do not set up a trampoline code which allows to take benefit from all the kernel mechanisms. Indeed, the majority of the rootkit malicious-operations are written completely by the attacker whereas the kernel implements a lot of mechanisms that could make easier the attacker's duty.
  The use of kernel services in our rootkit allows us to deploy the majority of its logical in a client-side program located in a remote machine. Thus, the contribution of our approach compared to the "all in memory" current attack techniques [24–26] lies in the fact that the compromised system memory does not contain any comprehensive part of our rootkit, at any moment. Only actions that cannot be implemented with the help of kernel services are designed and implemented to be executed directly in memory within the compromised system. In this way, we hamper online forensic-analyse mechanisms by leaving them some partial clues only, that are not sufficient to figure out or rebuild the malicious activities the attacker carried out.[17]

Kernel Backdoor Concealment. In order to hide the code and the data of our kernel backdoor, we propose a method that benefits from the characteristics of the Linux-kernel non-contiguous memory allocator (cf. Sect. 5.4).

- *Use of Kernel Mechanisms*: In order to hide a kernel backdoor we do not create any additional mechanism, but we just exploit the characteristics of the VMALLOC memory allocator. Through it, our kernel backdoor can thus be concealed. Once again, we try to benefit as much as we can from the kernel features any of kernel subsystem does.
- *Efficiency*: The method proposed by the Shadow Walker rootkit [18] (cf. Sect. 5.4) in order to dissimulate a rootkit is enforced by the hardware. Thus, as it works at a lower level than our technique, it is technically more reliable. However, the price to be paid is its implementation complexity and its strong dependency on the hardware architecture, contrary to our approach.

User-Land Part of the Backdoor. These advantages are only relevant for the case of an the attacker which remotely operates without connecting himself to the system through a regular user account. We proposed two approaches that both use a proxy mechanism (cf. Sect. 5.6).

- *Use of a proxy mechanism*: Our backdoors use a system call proxy mechanism. Thus, they are triggered from the user mode. It seems less interesting than approaches that interact with the attacker directly from the kernel. However, in this way, we are less dependent on kernel internals.

[17] We only relay system call 0 what furthermore hampers the rebuilding of the attack.

- *Alternative scheme for the user-land backdoor*:
  The alternative envisioned scheme for the user-land part of the backdoor—that is not implemented yet—includes an innovation compared to fixed parasitism approaches. Indeed, contrary to the backdoors which is parasitic upon a process, ours moves from one process to another (the collection of infected processes has to be for now established before the execution of the algorithm). The backdoor execution alters the execution of the infected-processes only temporarily. In this way, we can say that our approach is stealthier than fixed parasitism. However, it acts upon several processes instead of only one. Thus, relatively to the implemented detection type on the compromised system, the stealth of our backdoor can change.

Rootkit Services. Among the services provided by a rootkit, we focus on those which cover the attacker's activities concealment (actually, the last component of the protection module of the rootkit architecture proposed in Sect. 4.2).

- *A posteriori concealment*: The majority of rootkits hide *a posteriori* (i.e., after they are started) the processes they create by unlinking them from the system. We proposed a converse approach (cf. Sect. 5.7) by preventing the linking at the process creation (duplication and modification of `do_fork`). In this way, our processes have a limited interactivity from the beginning with kernel internal structures (except with those of the scheduler).
- *Dynamic Concealment*: We proposed a technique that hides a process and its offspring each time it evolves (cf. Sect. 5.7). The approaches that systematically hide all UID-specific processes reach a nearby outcome.
- *Mobile parasitism*: Our algorithm goes all over (or a subset of) the processes or the kernel threads of the system (cf. Sect. 5.7 for a brief explanation of the principle). The current parasitic techniques usually infect a single target only. The main contribution of our technique comes from our purpose on which we focused during its conception: the work of the infested processes must not be continuously altered. Thus, we improve malicious execution stealth by temporary process corruption.

6.3 Limitations

Kernel Part of the Backdoor. The main limitation of our interaction vector lies in the fact that the administrator may trace us; he can thus observe that the system call 0 is called from the user space. Since this kind of behavior is a priori suspicious, the administrator is likely to worry about the fact that the system may have been compromised.

Another problem is about the compatibility of kernel internals with respect to different versions. Indeed, the functions we execute through the system call 0 may change from one kernel version to another. Although these changes are not very common for critical or stable primitives, they are more likely than the *libc* API modification or than the kernel external ABI modification. Thus, the *remote userland-execve* attacks [26] have an undeniable advantage that is an ascertained compatibility whatever the kernel version which is used in the compromised system.

Kernel Backdoor Concealment. We show here the limitations of our concealment technique for rootkit code.

- *Full reading of physical memory*:
  Our method, which is based on the non-contiguous Linux kernel memory allocator, does not hold out on a full reading of physical memory. However, this kind of action takes long time to operate while greatly stressing the processor. Therefore this kind of detection mechanisms is usually not conceivable.[18]
- *Safety versus Concealment*: The area descriptor which creates VMALLOC, can betray our dissimulation attempts. We can of course delete it but then we have no longer the guarantee that our code could not be overwritten when a kernel module is inserted, for instance.

Backdoor-Actuators for a Process. The two proposed backdoor-actuators (cf. Sect. 5.5) use a kernel thread which is then hidden by breaking the majority of its links with the compromised system. They are only relevant for the case of an the attacker who logs on to the system through a regular user account. In the case of an attacker who cannot reconnect to the system like a regular user, these backdoor-actuators do not work.

User-Land Part of the Backdoor. These limitations are only relevant for the case of the attacker who remotely operates without logging on to the system through a regular user account.

- *Use of a proxy mechanism*: As the backdoor depends on a user-land process, hence it depends on its ability to survive within the system.
- *Alternative scheme for the user-land backdoor*: Our alternative scheme suffers from the same limitations that our mobile parasitic technique does. We address this issue in the remaining part of this section.

Rootkit Services. We present here the limitations of our concealment techniques of the attacker's activities.

The concealment of a process is breakable as soon as its descriptor is present and visible in memory. Indeed, we can

[18] Stealth is also the prerogative of defence mechanisms as a rootkit can detect the actions triggered against it and hence act accordingly.

reveal a process by exploiting many relationships between the `task_struct` and the `thread_info` structures (that, in part, makes up the process descriptor). Thus, going all over the physical memory to find out hidden processes, is utterly conceivable.[19] The observation of these limitations has carried us on with the elaboration of our mobile parasitism. However, this approach is not without drawbacks.

- *Limited robustness*:
  With our mobile parasitism, when the malicious code is executed by a given process, its survival (i.e., the fact that it can continue to execute itself or to move to another process) depends on this process upon which it is parasitic. Thus, if this process dies, our malicious code disappears.
- *Implementation difficulties of the malicious payload*: The design of the malicious payload has to be specifically designed to work with our algorithm. Thus, for instance, for the two-processes case, the malicious code has to be cut out in two independent chunks.
- *Detection risk*: The corruption of several processes is a limitation by itself. Indeed, the bigger this number is, the more flawed the malicious code stealth is. Indeed, some of the infected processes could be deception codes settled by the administrator to detect a mobile parasitic activity.

7 Conclusion and prospects

We have highlighted two main issues with respect to rootkit technology in this paper: on the one hand, the principles that allow us to model the rootkits, and on the other hand, the stealth-rootkit approach through malicious diversion of kernel subsystems. In addition, we discussed on a usually main kernel feature: hiding attacker's activity. Namely, we think that the longer we legitimately act before operating fraudulently, the less is obvious for detection mechanisms to succeed. Indeed, the attacker takes time to draw up its environment to make its malicious activity run as well as it can (the quicker, etc.). Thus, the attacker takes less risks to be detected when acting maliciously at the latest.

These observations stimulate us to characterize the rootkits in order to better evaluate them. We brought to light three criteria that qualify them. The next step is to define some relevant measures on these criteria in order to eventually get an unbiased comparator on the rootkits.

In addition, we introduced rootkit's functional architecture that we set up through its definition. Then, it will be interesting to put together this architecture and the criteria that we brought to light in order to figure out the essential spots. For that purpose, the formalization of both the architecture and the evaluation criteria is essential.

We only focused during our experimental study on the Linux kernel. Therefore it seems essential to deem the other current kernels to imagine many diversion ways. This study would help as an experimental base to determine factors that favor the diversion potentiality of kernel features.

Acknowledgement This work has been partially carried out by Éric Lacombe and Frédéric Raynal while at EADS France - Innovation Works, Suresnes, France.

Appendix: Searching for the current process descriptor in Linux 2.6

In order to find our process descriptor, we first look for the associated kernel stack. Indeed, a `thread_info` structure is at the bottom of this stack and it includes a reference to our descriptor.

So to find the location of our kernel stack, we have to know the `esp` stack pointer value whenever our process execute itself in kernel mode. To this end, we base our approach on the system call internals of x86 Linux since its version 2.6. The machinery operates thanks to the `sysenter` hardware instruction [41].

Let us now briefly describe the internals (Fig. 8). From the user space, `sysenter` is executed. At this time, `eip` (the instruction pointer register) et `esp` (the stack pointer register) are set up to compile-time predefined values (these values are loaded in machine specific registers during the system initialization) and the processor switch to *ring 0* (i.e., the kernel mode).

So, the `esp` value is always the same whenever the kernel mode is switched. But each process has its own kernel stack. Actually, the first instruction that is executed after the switch to *ring 0* consists in loading `esp` with the `esp` value of the scheduler-chosen process. This value is stored by the scheduler within the `tss_struct` that builds up the *Task State Segment* employed by the *x86* architecture. Linux 2.6 uses one of them in memory, for each processor only.

Once the `sysenter` execution is achieved, the `esp` register is loaded with the address of this structure. Therefore, the first instruction can get the kernel stack address of the executed process with only the help of `esp`.

Thus, in order to get the address of our kernel stack back, we just have to read at the location pointed by `esp` plus a fixed offset, at the time of the `sysenter` execution.[20] The problem is to know how to get the value that is affected to

[19] Moreover, we have implemented this approach in our demonstrator. False positives which occur due to the presence of dead process descriptors, are suppressed after a step which checks whether descriptors are fixed in time or not.

[20] When our process read the memory at this location, it indeed reads its `esp0` value; no other process is involved in this memory read access.

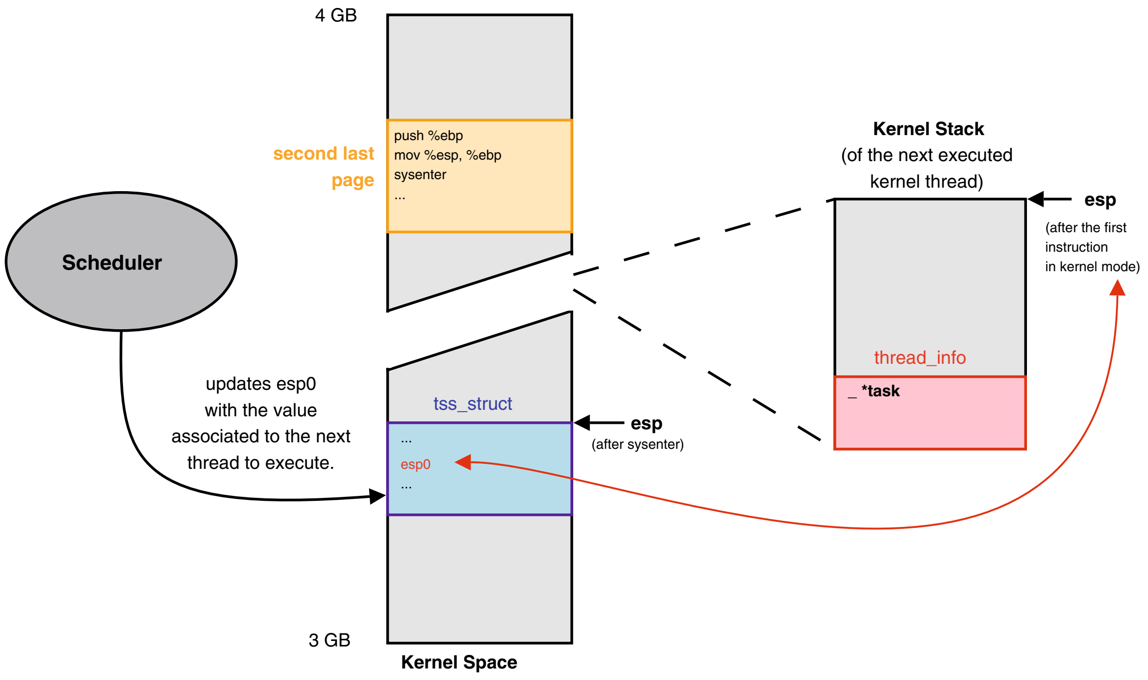


Fig. 8 Relationship between the process descriptor and the `sysenter` instruction

`esp` at the `sysenter` time. To this end, a hardware instruction allows us to get the value back, which is stored inside a specific register of the processor. However, this instruction is only runnable in *ring 0*. Therefore, we chose to find the kernel function in charge of the specific register initialization (`enable_sep_cpu`) to find out the value. To this end we employ pattern matching techniques through the `/dev/kmem` virtual device. From now on, we know the location of our kernel stack and hence the location of our process descriptor.

References

1. King, S.T., et al.: Subvirt: Implementing malware with virtual machines. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006)
2. Rutkowska, J.: Stealth malware taxonomy (2006)
3. truff. Infecting loadable kernel modules. Phrack 61 (2003)
4. Microsoft Corporation.: Digital signatures for kernel modules on systems running windowsǎvista. Technical report, Microsoft Corporation (2006)
5. Kruegel, C., Robertson, W., Vigna, G.: Detecting kernel-level rootkits through binary analysis (2004)
6. sd and devik. Linux on-the-fly kernel patching without l km. Phrack 58 (2001)
7. c0de. Reverse symbol lookup in linux kernel. Phrack 61 (2003)
8. Dornseif, M., et al.: Firewire—all your memory are belong to us. In: CanSecWest/core05 (2005)
9. Boileau, A.: Hit by a bus: physical access attacks with firewire. In: Ruxcon 2006 (2006)
10. Rutkowska, J.: Beyond the cpu: defeating hardware based ram acquisition tools (part i: Amd case). In: Black Hat DC 2007 (2007)
11. Cesare, S.: Syscall redirection without modifying the syscall table (1999)
12. kad. Handling interrupt descriptor table for fun and profit. Phrack 59 (2002)
13. buffer. Hijacking linux page fault handler. Phrack 61 (2003)
14. stealth. Kernel rootkit experience. Phrack 61 (2003)
15. Cesare, S.: Kernel function hijacking (1999)
16. Rutkowski, J.K.: Execution path analysis: finding kernel based rootkits. Phrack 59 (2002)
17. Lawless, T.: On intrusion resiliency (2002)
18. Sparks, S., Butler, J.: Raising the bar for windows rootkit detection. Phrack 63 (2005)
19. Soeder, D., Permeh, R.: Eeye bootroot: a basis for bootstrap-based windows kernel code (2005)
20. Kumar, N., Kumar, V.: Boot kit (2006)
21. Rutkowska, J.: Subverting vista kernel for fun and profit. In: Black Hat in Las Vegas 2006 (2006)
22. Filiol, É.: Computer Viruses: from Theory to Applications. IRIS International Series. Springer, France (2005)
23. Maximiliano Caceres. Syscall proxying—simulating remote execution (2002)
24. grugq. Remote exec. Phrack 62 (2004)
25. Pluf and Ripe. Advanced antiforensics: self. Phrack, 63 (2005)
26. Dralet, S., Gaspard, F.: Corruption de la Mémoire lors de l'Exploitation. In: Symposium sur la Sécurité des Technologies de l'Information et des Communications 2006, pp. 362–399. École Supérieure et d'Application des Transmissions (2006)
27. Raynal, F., Berthier, Y., Biondi, P., Kaminsky, D.: Honeypot forensics: analyzing system and files. IEEE Secur. Priv. J., aovt (2004)
28. Filiol, É.: Techniques virales avancTes. Collection IRIS. Springer, France (2007)
29. Filiol, E.: Strong cryptography armoured computer viruses forbidding code analysis: the bradley virus. In: 14th EICAR Conference, StJuliens/Valletta - Malta (2005)
30. Riordan, J., Schneier, B.: Environmental key generation towards clueless agents. Lect. Notes Comput. Sci. **1419**, 15–24 (1998)

31. Girling, C.G.: Covert channels in lan's. IEEE Trans. Softw. Eng. février (1987)
32. Wolf, M.: Covert channels in lan protocols. In: LANSEC '89: Proceedings on the Workshop for European Institute for System Security on Local Area Network Security, pp. 91–101, London, UK, 1989. Springer, Heidelberg
33. Rowland, C.H.: Covert channels in the tcp/ip protocol suite. First Monday, mars (1996)
34. Raynal, F.: Les canaux cachTs. Techniques de l'ingTnieur, dTcembre (2003)
35. Filiol, E., Josse, S.: A statistical model for viral detection undecidability. In: Broucek, V. (ed.) J. Comput. Virol., EICAR 2007 Special Issue, **3**(2) (2007)
36. The Honeynet Project Staff. Know your enemy: Sebek—a kernel based data capture tool (2003)
37. bioforge. Hacking the linux kernel network stack. Phrack 61 (2003)
38. Filiol, E.: Formal model proposal for (malware) program stealth. In: Proceedings of the 17th Virus Bulletin Conference (2007)
39. Cachin, C.: An information-theoretic model for steganography. In: Proceedings of the International Workshop on Information Hiding (1998)
40. 7a69ezine Staff. Linux per-process syscall hooking (2006)
41. Intel. IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference (2003)
42. Pragmatic and THC.: (nearly) Complete Linux Loadable Kernel Modules. The definitive guide for hackers, virus coders and system administrators (1999). http://newdata.box.sk/raven/skm.html