# PROACTIVE DETECTION OF KERNEL-MODE ROOTKITS

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

*Abstract*—**The sophistication of malicious software (malware) used to break the computer security has increased exponentially in the last years. Frequently, malware is hidden into a computer by software components called rootkits. Therefore, early detection of rootkits is of primary importance to avoid the uncontrolled operation of malware. Most of current techniques for rootkit detection only allow a late detection after the malware has already been hidden by a rootkit. In this paper, a new technique is presented that enables the proactive detection of rootkits while they are hiding malware, and therefore, allowing that hiding can be avoided. The technique has been designed for rootkits that operate in kernel-mode. This rootkits are particularly difficult to detect because both the detector and the rootkit are executed with the same privileges. This technique can be used to improve the detection capabilities of intrusion detection and prevention systems.**

***Keywords-Malware; Rootkits; Stealth software; Hooking***

## I. INTRODUCTION

Rootkits, or more generically stealth malware, are software components used to hide objects inside a computer system. Generally, the objects hidden by rootkits are processes and files.

The nature of rootkit operation implies that they are usually detected by crossview detection. This kind of detection method cross checks the information obtained from a raw object enumeration with the vision that a program using the operating system API would get. This implies solid knowledge of the functionality of the operating system and its data structures. This technique detects the presence of rootkits into the system but does not stop them from operating. In addition, the time period in which the malware has executed its actions is undetermined.

This paper presents a new technique that allows the detection of a particular modification in data or code of the operating system. The paper focuses on Windows NT family over Intel architecture, although the same ideas are applicable to UNIX systems and even to other processor architectures if they operate with paged memory.

## II. TECHNOLOGICAL BACKGROUND

One of the latest technologies used for malware aid is rootkit technology. The rootkits hide system objects such as files, processes, drivers, registry keys, etc., which are generally related to a malware element.

Originally, rootkits were understood to be toolkits that allowed an attacker to subvert an UNIX system and maintain root privileges over it without being detected by the system administrator. On UNIX systems, after a successful exploitation of a bug which allowed the attacker to obtain root privileges, the attacker usually made use of this kind of tools. They allowed him to hide the processes or TCP channels so he could continue making use of the attacked computer. These tools were trojanized versions of the original programs installed in the system. For example, the toolkit could include a version of the `ps` program which omitted the processes the attacker uses to his own benefit [1].

However, there have been recent changes in the malware scenario. Current stealth software presents a more technical profile so the definition must be expanded to include code that executes information hiding tasks in a compromised system [2]. Thus, the current definition of a rootkit is any software that gives continued privileged access to a computer while actively hiding its presence and other information from administrators by subverting standard operating system functionality or other applications.

Currently, the complexity achieved by both malware and anti-malware software is very significant. This can be seen in [3] for PCI rootkits, in [4] for SMM rootkits, in [5] for MBR rootkits, and in [6] for rootkits that manipulates the memory page tables of the processor.

We must distinguish between the rootkit and a possible associated malware element which is hidden by the rootkit. Rootkit technology, as we understand it nowadays, is not malicious in itself, but generally its application is. However, the hidden object can be the rootkit itself, which probably implies some kind of malware rootkit.

Rootkits are mainly classified in terms of their execution mode on the processor [7], [8]. Those modes are:

- User mode: The rootkit usually affects a single user process. If system wide hiding is desired, all the user mode processes in the system are subverted. To achieve its goals, the rootkit is loaded in the address space of the victim process. The mechanism used to infect the processes usually implies some kind of code

injection (legal or illegal) and once the code is loaded, it hooks system API functions to perform its malicious actions. The two main hooking techniques are: Import Address Table (IAT) deviation and detouring via code overwriting. The rootkit can also take the form of a trojanized binary module which executes additional actions to those expected by the user.

- Kernel mode: The rootkit usually takes the form of a dynamic kernel module, which is loaded in the kernel memory. Once loaded, it can operate by detouring system calls (type 1) or modifying the data structures of the operating system directly (type 2) [9].

## III. Related Work on Rootkit Detection

There are two main types of techniques for detecting a rootkit: those which detect the presence of the rootkit and those which detect the behavior of the rootkit.

To detect the presence the rootkit, several elements in the system should be inspected. The rootkit must use the system resources so there may be inconsistencies that the rootkit does not take into account. A rootkit detector should inspect the following elements:

- File system: The detector must look for a byte pattern that identifies a known rootkit (or malware in the most general case). This is the traditional antivirus approach.
- System mechanisms that allow a binary module to be loaded into memory: Functions such as NtLoadDriver or NtOpenSection allow code to be loaded into memory. Unfortunately, there are a lot of functions which can be used to load code into memory [2], and there are also other methods like registry keys which can load modules into memory. These registry keys are process dependent, which further complicate the inspection. This technique follows a "protecting the gate" approach.
- Virtual memory modules: A rootkit detector can walk through the list of modules loaded into memory in order to find a byte pattern which identifies a rootkit [10]. This approach is the same as the detection technique in a file system, but applied to memory modules.
- Hooks: A type 1 rootkit must modify somehow the execution flow of system code. This approach is used in the anti-rootkit VICE system introduced in [11] and in its evolution RAIDE, introduced in [12]. Typical elements to inspect are:
  - IAT (Import Address Table)
  - SSDT (System Service Dispatch Table)
  - IDT (Interrupt Descriptor Table)
  - Drivers' I/O Request Packet (IRP) handler
  - Inline hooks.
- Tracing execution: This is another technique to look for hooks in the system. It is presented in [13]. The idea is to instrument a clean system to obtain the number of instructions that a given code path should execute. The premise is that hooks will increment the number of instructions executed for a given code path. This technique is questionable as the same code path executes a different number of instructions depending on different factors. Although in [13] is affirmed that the difference is appreciable for known rootkits, this could depend upon the sophistication of the attack.

The main disadvantage of all the techniques based on presence detection is that as new attack techniques are developed, the detection process must protect more elements that the rootkit could use to its benefit.

To detect rootkit behavior, the main technique is crossview detection [14]. This technique compares a set of objects obtained by a high level process with a raw enumeration, performed almost at hardware level. Any differences imply that there is a rootkit in the system. The main disadvantage of this technique is that, although evidence of intrusion in the system is detected, neither the rootkit nor its associated malware are identified. One advanced antirootkit system using this technique is Strider GhostBuster developed by Microsoft Research and introduced in [15].

There are also other complex techniques that are focused on the detection of the hooking process [16].

It is important to notice that none of the aforementioned techniques prevent the installation of the rootkit in the system. They only allow detection at a later stage when the rootkit is already installed in the system.

However, one of the newest antimalware defenses consists in components that anticipate the presence and operation of user-mode malware. They generally do so by monitoring user processes from kernel modules and observing their behavior. Then, they can infer if the observed behavior is suspicious and act consequently. These techniques can work because of the different protection levels of execution modes in Intel processors. User-mode processes always run in a less privileged mode than the kernel. This way, the kernel can monitor and modify user-mode processes if it is required.

But when malware executes in kernel-mode, the privileges of malware and operating system are the same. Kernel-mode antiviral modules have no privileges over kernel-mode malware modules, so malware cannot be monitored and the required information to detect malicious behavior dynamically cannot be collected.

In fact, current detection techniques for kernel-mode rootkits are the already explained ones. The most important problem they suffer is that the detection cannot be anticipated, and rootkits can only be detected at a later stage, when they have already been operating for an undetermined time. It would be very interesting if rootkit operation could be monitored in real-time so an antimalware engine could not only detect but even prevent the rootkit actions. The new technique presented in this paper achieves this goal.

## IV. THE NEW TECHNIQUE FOR ROOTKIT DETECTION

This paper proposes a new technique to detect any software module (rootkit) which patches the System Service Description Table (SSDT) or manipulates the process list in Windows systems in order to hide processes. This technique is based on gaining execution at the moment that the code or data of the operating system is being patched (modified). This is achieved by hooking the page fault handler in the Interrupt Descriptor Table (IDT), and then, manipulating the kernel page tables to set the memory pages to be monitored as "not-present".

In this way, the page where the SSDT resides can be hidden and write accesses over it can be detected. In case of a writing trial to the SSDT, the module responsible for the modification can be identified analyzing the stack. The identity of the responsible module can be used to decide whether to allow or deny the modification. This is important as the technique of SSDT detouring is also used by legal software such as any antivirus.

The same concept can be applied for monitoring accesses to the process list. In this particular case another problem arises because the nodes of the process list are obtained from a non-paged memory pool in a dynamic fashion. Therefore, a technique to hook system memory allocations is used. Then, when a node of the process list is allocated, the memory allocated is detoured to a controlled memory page which can be monitored. In this manner, the allocation and deallocation of nodes of the process list can be controlled. Once allocated to a controlled memory page, it is possible to detect when the pointers to the nodes of the process list are manipulated. Once again, stack walk is used to determine which module is deleting the node. If it is not the operating system itself, it is probably due to an attempt to hide the process.

There are three major components involved in the proposed mechanism:

- The IDT hook and page fault handler.
- The memory allocation and deallocation handler.
- The finder of the memory writer modules.

Each component is examined in depth in the following sections.

### *A. The IDT hook and the page fault handler*

To control the page fault handling mechanism, the code responsible for resolving page faults must be hooked. The best way to do this is modifying the 0x0E entry (page fault exception) in the IDT. Firstly, the IDT address must be located executing the SIDT instruction. There is an IDT per processor. Once located in the IDT, the page fault handler entry is overwritten with our own handler, and the overwritten entry corresponding to the system page fault handler is saved. A debug trap handler (0x01 exception) must also be installed in order to restore the presence-bit state of the pages monitored in order that they can continue being monitored.

One often subverted Windows structure is the SSDT. The SSDT is used to resolve API system calls [2]. It contains a pointer to an array of function pointers (denominated services array) and its length, among other information. Every user-mode system function has an index in this array of function pointers. In this way, when a system function is invoked, the kernel uses this table with the appropriate index and transfers the execution flow to the function responsible for serving the call. Remind that there are two important elements: the SSDT and the services array.

Once our handlers are installed, the presence bit of almost any memory page can be turned off and the accesses to memory pages are monitored, in particular to the page where the SSDT resides. The monitoring process is shown in figure 1. In the left column shows the software modules involved on a purple background. On the right, are the memory pages accessed by the modules on an orange background.

The following points describe the monitoring process. The point numbers correspond to the numbers in the circles of figure 1.

(1) A rootkit tries to hook a function of the operating system denominated NtOriginalFunction. It performs the hooking by overwriting the entry of SSDT table at the 0xXXXXX0FF address, which is where the pointer to NtOriginalFunction resides in the services array of the SSDT.

(2) The rootkit tries to write on the memory page containing the services array. This page is being monitored, and therefore, it appears as not present in the physical memory because its presence-bit has been set to 0 in its entry of the page table.

(3) The system triggers a page fault trap which invokes the page fault handler. As our page fault handler was installed previously, it gains execution. Then, our page fault handler determines if the faulting address belongs to a monitored memory page. In this case, it looks for the code that tries to access the page, presumably belonging to a rootkit module.

(4) Our page fault handler sets the presence bit of the offending page to 1, making it visible for further memory accesses.

(5) Our page fault handler also changes the trap flag of the thread that is executing the rootkit code. This forces to run the rootkit code in debug mode, triggering a debug exception after the execution of each instruction.

(6) When our page fault handler has finished, the offending instruction is re-executed successfully.

(7) As the instruction execution was successful, the 0xXXXXX0FF address is written with the address of the rootkit function. Thus, the rootkit has effectively hooked the NtOriginalFunction of the system.

(8) As our page fault handler has put the thread executing the rootkit code in debug mode, just after the execution of each instruction a debug trap is executed, and our handler gains execution again.

(9) Our debug trap handler restores the data if this is the desired action. Remember that the offending code has been identified as belonging to a rootkit module.

(10) Next, our debug trap handler hides the page again by setting its presence bit to 0.

(11) Finally, our debug trap handler deactivates the debug mode in the offending thread.

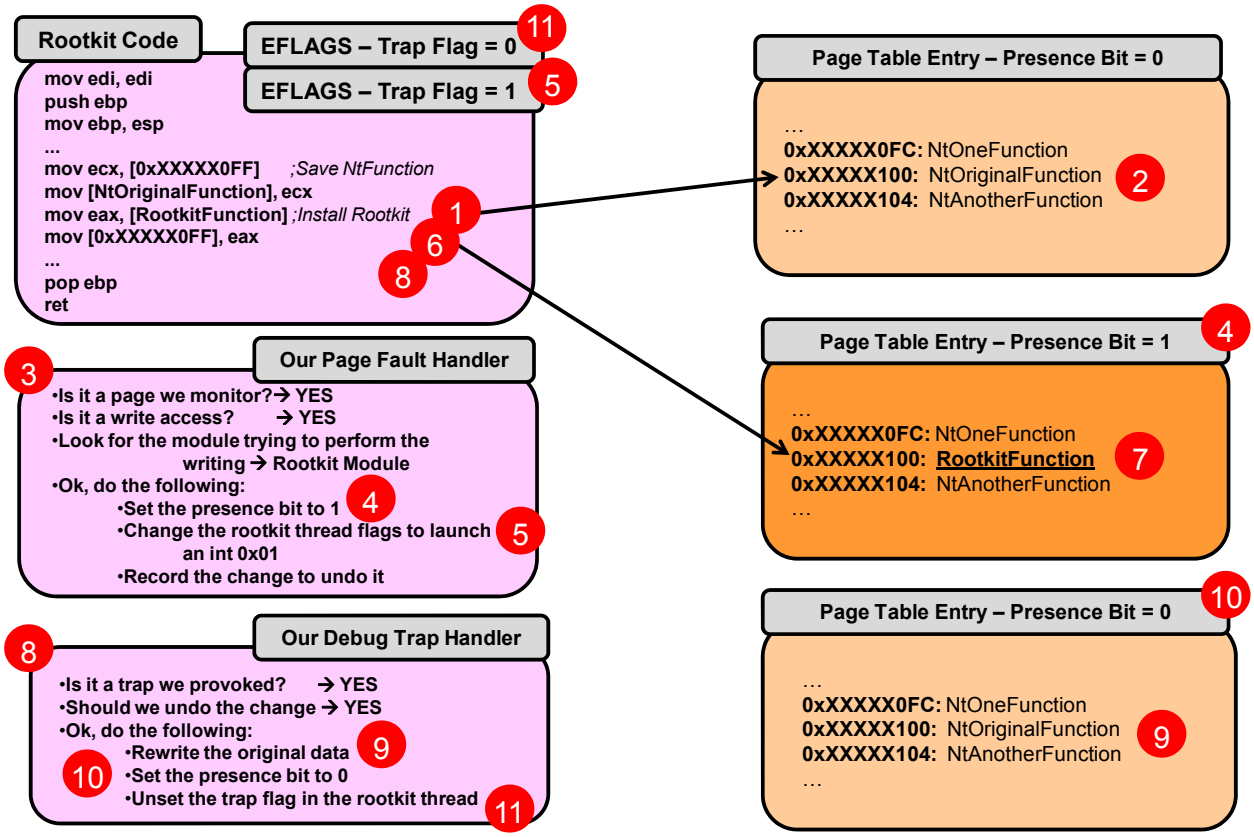


Fig 1. Page Monitoring Technique

### B. *The memory allocation and deallocation handler*

One technique to hide processes in Windows is by manipulating the process list directly. This is an example of Direct Kernel Object Manipulation (DKOM), also known as type 2 malware.

Windows processes are represented by a record called EPROCESS. If the EPROCESS records are in a page which can be monitored, accesses to EPROCESS records can be detected and the hiding of a process can be monitored. Figure 2 shows the technique to allocate EPROCESS records in monitored pages. The upper area shows the execution path followed when the system is allocating memory. The lower area shows the memory given to the system in two cases: allocating an EPROCESS and another memory.

The following points describe this technique. The point numbers correspond to the numbers in circles in figure 2.

(1) The operating system creates a new process and allocates memory for the new EPROCESS record.

(2) The memory allocation function has been hooked in order to inspect every memory allocation in the system.

(3) If memory is not allocated for a new EPROCESS, the execution is returned to the original function.

(4) In this case, the original function finishes its execution by giving a block of memory to the calling code which is then allocated in a system page.

(5) On the other hand, if the memory allocation is for an EPROCESS (which can be seen in the "tag" parameter), a previously allocated page which can be monitored is used to host the EPROCESS record.

With this technique, accesses to the linking fields in an EPROCESS record as well as the behavior shown in figure 2 can be monitored. A complementary mechanism has been developed for memory deallocation, because when the system deletes an EPROCESS, the execution flow must be detoured to avoid a system inconsistency.

### C. *The finder of the memory writer modules*

Determining which module has generated a modification is as important as the modification itself. The operation of the finder is shown in figure 3. The left part shows the stack of the thread performing the write operation. The stack grows from bottom to top in figure 3 towards low addresses. The right part shows the modules (addresses and their content). The CR2 register of an Intel processor contains the address of the instruction that provoked a page fault.

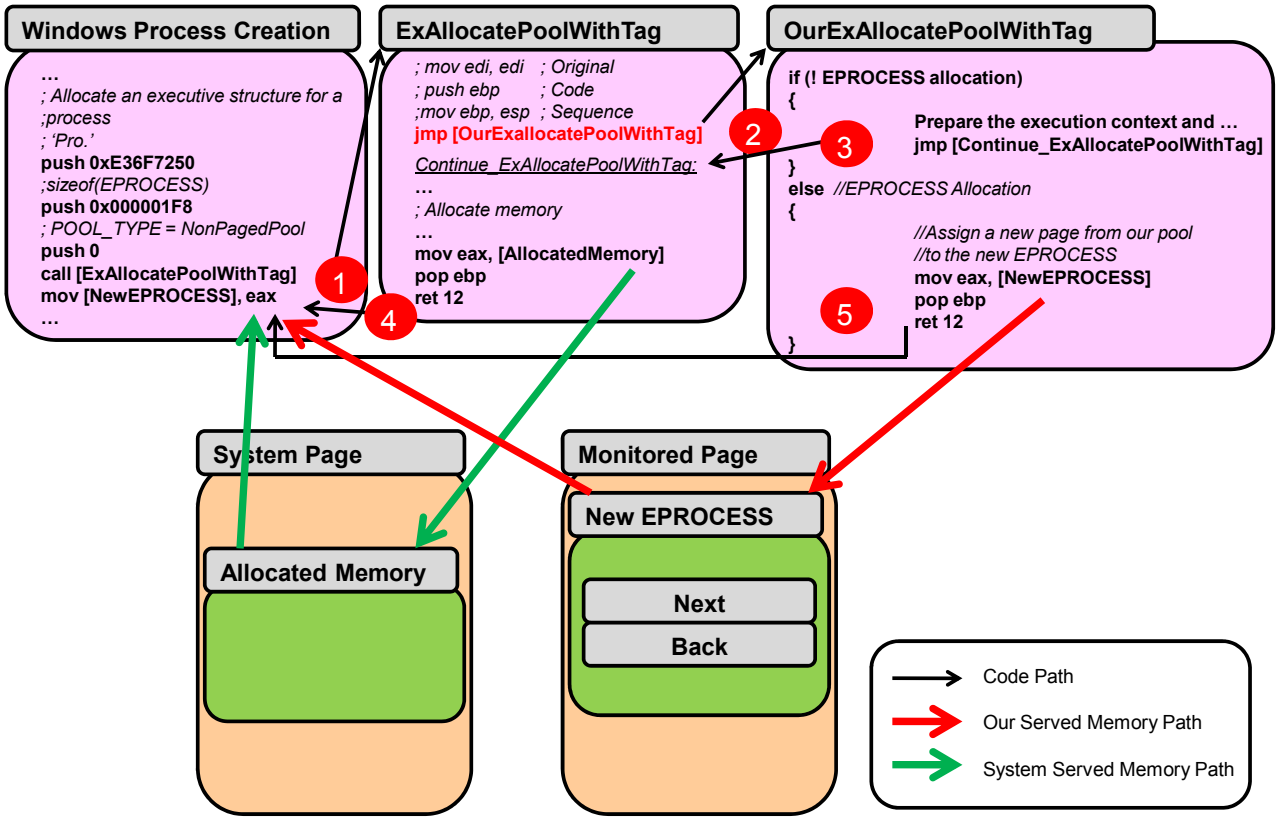


Fig 2. Memory Allocation Handler in action

In the first scenario, the exception record placed in the stack, in a green box in figure 3, shows that the write access on a monitored page was originated by the rootkit module.

In fact, the finder can obtain the exact address where the write access is performed. In this scenario is straightforward determine the responsible module of the modification, because, as figure 3 shows, the finder can infer the writer module directly from the faulting EIP stored in the stack, as this virtual address belongs to the rootkit module.

In the second scenario, find the writer module is a little more complicated. The write access is performed by a helper function located in NtosKrnl module. This function is called by the rootkit code and performs an atomic write on the memory address indicated by its first parameter with the value of the second parameter.

By examining only the exception record, surrounded by a green box in figure 3, the finder would suppose that the modification was originated by the kernel itself. But this is not correct, because the real responsible of the write access is the rootkit that called the helper system function.

So the finder must analyze the stack in order to find the true writer module. As the rootkit code performed a function call, the stack must contain an activation record corresponding to this call. Figure 3 shows the activation record surrounded by a blue box, which has the two parameters passed to the function, the return address and locals. Using that return address, which belongs to the address range of the rootkit module, the finder can infer that the module which performed the memory write was, in fact, the rootkit module.

## V. Evaluation of the Technique

The evaluation of the proposed technique was based on black-box testing. The evaluation method consisted of installing a kernel module containing the implementation of the proposed technique, and then executing several software which is known to hook Windows SSDT and hide processes manipulating the process list. The software used in the test was primarily malware and antivirus. The tests were performed in a Windows XP with Service Pack 2 running on an Intel IA-32 machine with 1GB of RAM.

The SSDT hooking tests have shown, without prior knowledge, that Karspersky Antivirus hooks almost all SSDT functions, and also that rootkits usually hook five or six system functions, all of them directly related with its purpose, such as NtEnumProcesses.

Other successful test was the monitoring of an encrypted rootkit (Mitglieder.OM). The application of reverse engineering to this rootkit in order to determine the system function that it modifies is extremely difficult. But once the rootkit was running, the proposed technique discovered the SSDT functions hooked by this rootkit.

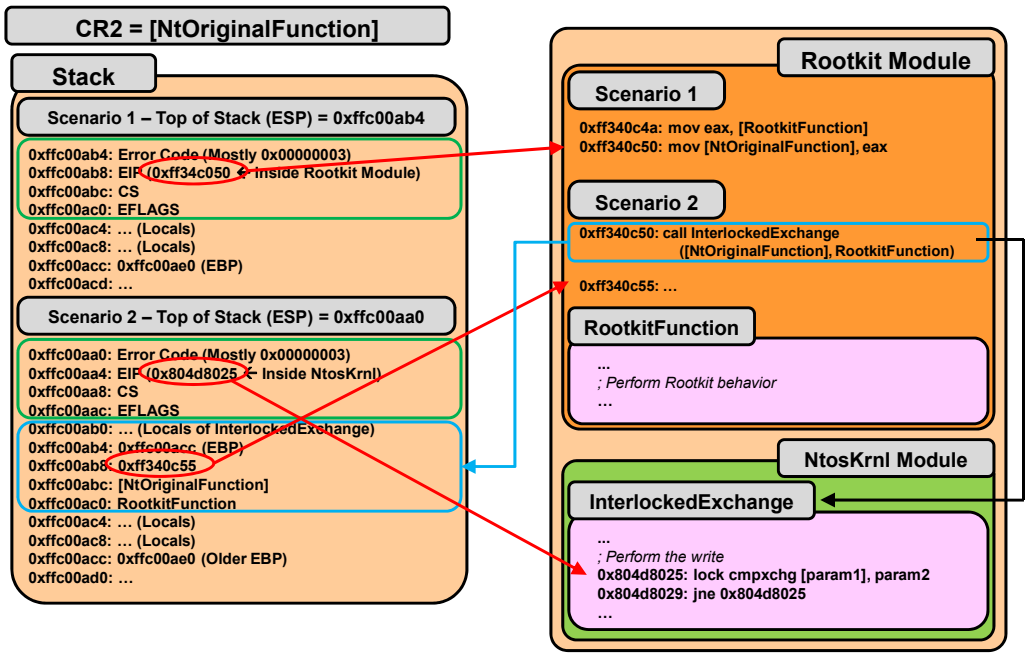


Fig 3. Analysis of the stack to find the memory writer module

In the case of processes hiding, the test used the proof-of-concept "Fu" rootkit for hiding processes which uses the aforementioned technique involving process unlinking, verifying that the process hiding trial is detected and that the module responsible is correctly identified.

To further validate the obtained results, the tests should be performed on additional platforms, primarily different operating system versions with different customizations, service packs and hardware.

## VI. Conclusions and Future Work

A new technique to detect modifications on parts of the operating system kernel has been developed. It has been applied it to monitor the parts most often modified by kernel-mode rootkits and its usefulness has been verified. This technique detects patching trials of the operating system by unknown modules on the fly.

There are various ways to evade our technique, for example, modifying the IDT page fault entry again, thus disabling the mechanism. Furthermore, the page containing the IDT cannot be protected by our technique as it must be a present page in memory to avoid a processor triple fault.

Future work includes extending the technique to handle different paging schemas such as large pages. Also, a more robust mechanism to analyze the stack, probably based on frame walk using the EBP processor register, should be developed.

## References


[1] S. Manap, "Rootkit: Attacker undercover tools," Personal Communication, 2001.

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

[3] J. Heasman, "Implementing a PCI rootkit," White paper of Next Generation Security Software Ltd., 2006.

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

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

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

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

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

[9] J. Rutkowska, "Introducing stealth malware taxonomy," White paper of COSEINC Advanced Malware Labs, 2006.

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

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

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

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

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

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

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