

Windows rootkits in 2005, part two

James Butler, Sherri Sparks,

1. Introduction

In our previous article, we discussed current rootkit development techniques. In this article, we take it a step further and focus upon upcoming, cutting edge trends in rootkit technologies. Then the third and final article in this series will discuss various methods of rootkit detection and countermeasures that can be used by security professionals.

The methods described in this article were presented in our proof of concept rootkit named Shadow Walker at Black Hat 2005. These methods make it possible for an attacker to hide both known and unknown malicious code from a security scanner by controlling its memory reads at the hardware level. Although we focus upon rootkits, the underlying implications are alarming because the technology can be applied to all forms of malicious code, ranging from worms to spyware.

1.1 Persistent versus memory-based rootkits

Generally speaking, there are two types of rootkits: *persistent rootkits* and *memory-based rootkits*. The primary difference between these two types of rootkits lies in their "persistence" on an infected machine after a reboot. Persistent rootkits are capable of surviving a system reboot whereas memory-based rootkits are not. In order to survive a reboot, two conditions must be met. First, they must have some means of permanently storing their code on the victim system (such as on the hard disk). Second, they must place a hook in the system boot sequence so that they can be loaded from disk into memory and begin execution.

Unlike persistent rootkits, in-memory rootkits make no effort to permanently store their code on disk or hook into the boot sequence. Their code exists only in volatile memory and they may be installed covertly via a software exploit. This makes them stealthier than their "persistent" brethren and confers anti-forensic advantages. While it may seem that an inability to survive a reboot would undermine the usefulness of these rootkits, server systems frequently remain online for days, weeks, or months at a time. In practice, the potential for losing the rootkit infection may be counter-balanced by an attacker's need for untraceability.

1.2 Hiding a rootkit's presence

Rootkit writers have developed a number of clever techniques for hiding their rootkit's presence on a system. These techniques range from various hooking tricks to direct kernel object manipulation (DKOM). Nevertheless, even the most sophisticated kernel rootkits like FU have an inherent flaw. [ref 1] Although these rootkits are experts at controlling the execution path, for the most part they have not demonstrated an ability to control the view of memory that is seen by other applications. Thus, these rootkits must address two primary issues if they are to remain undetected. First, they must be able to conceal the presence of their own executable code. Second, they must be able to conceal their memory-based modifications (i.e., hooks) in operating system components. Without these capabilities, even the most sophisticated public kernel rootkits are "sitting ducks" for primitive in-memory signature detection scans - the same type of scans anti-virus products have been using for the past 20 years. Persistent rootkits must furthermore deal with hiding their code on a long term storage medium and concealing a permanent hook in the system boot sequence. In this article, we will be addressing the first two issues and ignoring the third. Practically speaking, this confines our discussion to memory-based rootkits.

The problem of hiding code and/or changes in memory is reminiscent of the problem early virus writers faced when attempting to hide their code on the file system. Virus writers reacted to file system signature scanners by developing polymorphic and metamorphic techniques. Polymorphism attempts to vary the superficial appearance of a block of code while maintaining functional equivalence. As a simple analogy, we can consider English synonyms (words that are spelled differently yet have exactly the same meaning). A polymorphic virus replaces instructions (words) with different opcodes (synonyms) that perform the same functionality. In this manner, the mutated virus superficially "looks different" and becomes immune to simple pattern-based detection.

Very few public rootkits have made any substantial effort to integrate viral polymorphic techniques. Though polymorphism could be effective for disguising a rootkit's code body from signature scans, it is not ideal because it does not lend itself well to hiding the changes a rootkit makes to existing binary code in other system components. In other words, hijacked system components remain vulnerable to in-memory integrity checking. A better solution, therefore, is not to alter the rootkit's code, but to alter how other system components "see" it. In the following sections, we show how the current architecture permits subversion of virtual memory management so that a non-polymorphic kernel mode rootkit is capable of controlling the memory reads of the operating system and other processes. In section 2, we review basic architectural and operating support for virtual memory. In section 3, we discuss how the Shadow Walker proof of concept rootkit subverts the virtual memory subsystem to hide executing code from a security scanner. Finally, in section 4, we discuss the implications of this technology for both security professionals and hackers.

2. Virtual memory concepts

Most modern architectures make a distinction between "virtual" and "physical" memory. Oftentimes, a system will have a great deal more virtual memory than physical memory. Consider a 32-bit system with 256 MB of installed RAM. On this machine we have 4 GB of virtual memory, even though our physical memory size is only 256 MB. In short, physical memory size is defined by the amount of physically installed RAM, and virtual memory size is defined by the width of the processor's address bus. Thus, with a 32 bit processor we are capable of addressing a maximum of 2³² bytes, or 4 gigabytes of virtual memory. With a 64-bit machine, we would be capable of addressing 2⁶⁴ bytes, or over 16 exabytes of memory!

Virtual memory implementations come in a couple of different flavors including segmentation and paging schemes. The x86 architecture supports both; however, this article focuses on paging since that is the part Shadow Walker subverts. The basic idea behind paging is that the virtual and physical address spaces are divided into fixed size blocks. Virtual memory blocks are referred to as "pages" and they are mapped to blocks of physical memory known as "frames." Page tables and page directories hold the mapping information necessary to link virtual pages with their corresponding frames. They also hold protection and status information. One key point is that when paging is enabled, every memory access must be looked up to determine the physical frame to which it maps and whether that frame is present in main memory. This incurs a substantial performance overhead, especially when the architecture is based upon a two-level page table scheme like that found in the Intel Pentium.

By making a distinction between virtual and physical memory, the hardware and operating system are able to provide processes with the illusion that there is more memory than is actually, physically available. Paging is invisible to application processes. From the viewpoint of an application, it has 4 GB of virtual memory available for its personal use. It does not need to know how much RAM is actually installed or how the virtual addresses it uses map to physical memory. Since the virtual address space may be larger than the physical address space, it is possible that

a process' demand for memory may exceed the amount of memory that is physically available. If this happens, the operating system will need to temporarily swap some of the data in physical memory to disk in order to make room for current memory demands. It does this by copying some frames to the pagefile and marking their corresponding page table entries as "not present." When these pages are accessed again, they will not be present in main memory and page faults will occur. A page fault will invoke the operating system's page fault handler, causing it to issue the I/O request necessary to bring in the requested page from the page file. If all of the available physical frames are still full, the handler may have to swap another page out before it can bring in the requested page.

In a two-level paging scheme, a memory access potentially involves the following sequence of steps.

1. Lookup in the page directory to determine if the page table for the address is present in main memory.
2. If not present, an I/O request is issued to bring in the page table from disk.
3. Lookup in the page table to determine if the requested page is present in main memory.
4. If not present, an I/O request is issued to bring in the page from disk.
5. Lookup the requested byte (offset) in the page.

Figure 1 illustrates the process of x86 address translation. From the above steps we can see that, in the worst case, a single memory access may actually require three memory accesses plus two disk I/Os. Hardware designers developed the Translation Lookaside Buffer (TLB) to help with this problem. The TLB is a high speed cache that is used to hold frequently used virtual-to-physical mappings. When a memory access occurs, the TLB is searched first for the virtual-to-physical translation information before consulting the page directory / table. If the translation is found, it is termed a "hit". Otherwise, it is a "miss". Because the TLB can be searched much faster than performing an access to the page tables, memory accesses resolved via the TLB avoid most of the aforementioned performance penalty.

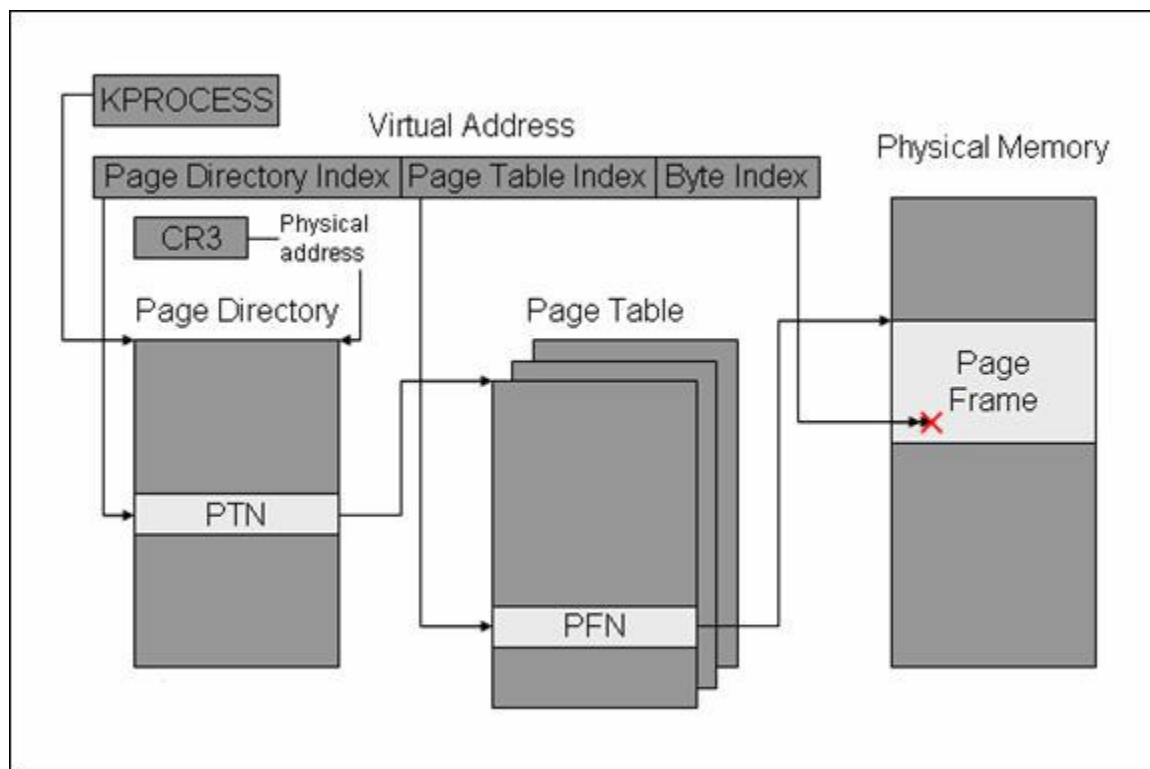


Figure 1. x86 virtual To physical address translation.

3. Shadow Walker: how it works

The Shadow Walker proof of concept currently consists of 2 drivers, a modified FU rootkit driver and a memory hook driver that is used to hide the FU rootkit. As Shadow Walker is intended only to be a proof of concept demonstration, it makes no effort to hide the memory hook driver or otherwise conceal the page fault handler and corresponding interrupt hook. It also possesses a number of implementational limitations, including lack of PAE (Physical Address Extension) and multiprocessor support. Shadow Walker is not intended to be a fully functional or weaponized rootkit and, in its current implementation, it is detectable from a variety of angles. Rather, it is intended to provide a chilling glimpse into the future of rootkit technology. A fully weaponized rootkit, worm, or spyware application based upon Shadow Walker is within reach of a skilled attacker and given the state of existing malicious code detection technology, it is a scary thought indeed. These methods make it possible for an attacker to hide both known and unknown malicious code from signature scanners, heuristic detectors, and integrity checkers. In the next paragraphs, we discuss the implementational details behind the virtual memory subversion techniques used by Shadow Walker.

3.1 Background information

Although it is common knowledge that there are three basic types of memory access (read, execute, and write), it is less common knowledge that there are, in fact, only two types supported for most of the 32-bit x86 processors (read/execute and write/execute). Execute access is implied, meaning that all memory is executable and there is no direct, supported means of marking it otherwise. This quirk of the architecture has been the bane of buffer overflow intrusion detection systems because it meant that there was no easy way to make the stack non-executable . [ref 2] Not to be outdone, a group of software security developers discovered that another quirk of the architecture made it possible to implement "no-execute" memory with additional software support under UNIX. [ref 3] This implementation became known as PaX. Shadow Walker takes advantage of some of the PaX team's research to hide executing code on Windows systems.

To recap: modern rootkits need to address two additional issues if they are to remain undetected.

1. They must be able to conceal the presence of their own executable code from in memory signature scans.
2. They must be able to conceal their memory based modifications (typically, hooks) in operating system components from heuristic detection (with tools like VICE) and integrity checkers. [ref 4]

In order to accomplish these goals, the rootkit must be capable of controlling the data returned from the raw memory reads performed by other applications, such as security scanners. Clearly, if a rootkit detects a read access to its own executable code section it is a reasonable heuristic that a scanner may be looking for it!

3.2 Shadow Walker's subversion of virtual memory

Shadow Walker must address three issues in its subversion of virtual memory. First, it must be able to differentiate and filter execute, read, and write accesses to certain memory ranges (i.e. the memory where its own executable code resides or the memory of some subverted operating system APIs). Second, it must be able to "fake" the read accesses when they are detected. Lastly, it must ensure that the performance of the victim system is not adversely affected.

The first issue is solved by marking the PTEs for the hidden pages "non present" and hooking the page fault (PF) handler. This ensures that Shadow Walker will be able to trap accesses to these pages and subsequently filter them in the page fault handler. In the page fault handler, we have access to the saved instruction pointer and the faulting address. If the instruction pointer is the same as the faulting address, we can conclude that the memory access was due to an execute. Otherwise, it is a read / write. One other point to note is that Shadow Walker needs to differentiate between page faults due to the memory hook and normal page faults which need to be serviced by the OS. It currently addresses this issue by ensuring that all hidden pages are in non-paged memory. This isn't a problem since Shadow Walker is currently designed to hide only driver pages that are usually non-paged anyway.

The second issue, that of faking read accesses, is somewhat tricky and is tied in with the third issue of not degrading performance. Once the page fault handler has been invoked and we have detected and verified a read access to a hidden page, it is possible for us to modify the PTE and alter the physical frame address "on the fly." Thus, we can ensure that execute accesses translate to our "subverted" frames while read / write accesses translate to "clean" frames. Remember, however, that the TLB is actually the first entity on the memory access path and that performing a memory access loads the TLB with the corresponding translation. The Pentium actually uses a "split" TLB architecture and Shadow Walker uses this fact to devious advantage. A split TLB architecture means that there are actually two TLBs, one to hold the translations for execute accesses (ITLB) and the other to hold the translations for read / write accesses (DTLB). Normally, the two TLBs are synchronized and maintain the same mapping information. It is, however, possible to de-synchronize the TLBs such that they hold different virtual-to-physical mapping information.

A rootkit like Shadow Walker can use this desynchronization trick to hide executing code. In other words, it loads the ITLB with the mapping information for the subverted pages, and the DTLB with the mapping information for a "clean" copy. The rootkit code runs, but all attempts to read this region of memory result in "clean" non-rootkit data being returned to the security scanner. TLB loading is performed in the page fault handler in response to faults generated by memory accesses to hidden pages. Figure 2 illustrates this concept. All other faults are passed down to the operating system page fault handler for service.

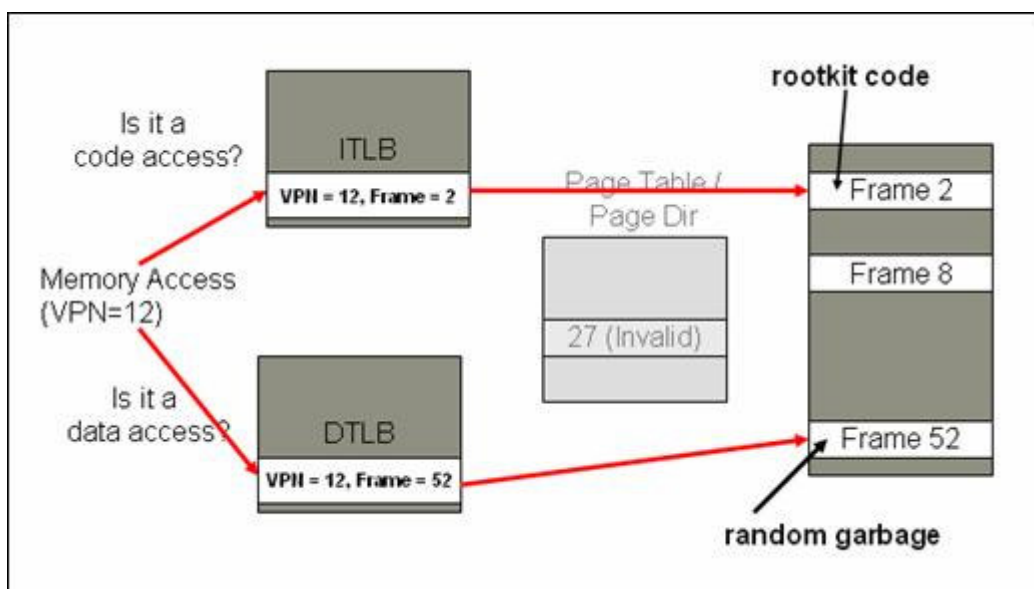


Figure 2: TLB Desynchronization

Once loaded, most memory references will be resolved via the TLB path and will not generate page faults. Faults, however, will occur on the first execute and data accesses

to the page, on TLB cache line evictions, on context switches, and explicit TLB flushes. These few faults do not pose any problems and will simply result in the page fault handler being invoked to re-desynchronize the TLBs. Finally, this brings us to our last issue of maintaining system performance. The relatively low increase in the page fault rate caused by Shadow Walker coupled with the extremely high hit rates of modern TLBs results in virtually no noticeable performance impact.

4. Concluding part two

Shadow Walker provides an example of an ironic, however, common yin/yang theme in computer security. It takes an originally defensive solution to a security problem (with the PaX-based buffer overflow protection) and inverts it into an offensive exploitation technique. The recent controversy surrounding Sony's usage of rootkit technology to provide Digital Rights Management is yet another compelling example. The lines between protection and exploitation between the "hacker" and the "security professional" are not as clearly defined as many would like to believe.

In the first article of the series, we showed that security applications can't trust the integrity of operating system APIs. Now, we show that the violation of trust runs much deeper. Shadow Walker bids user mode malware scanners a final goodbye and places rootkit detection strictly within the kernel realm. In the third and final article of this series we will discuss rootkit detection and threat mitigation.

5. References

- [ref 1] Fuzen, FU Rootkit. <http://www.rootkit.com/project.php?id=12>
- [ref 2] Hardware support has subsequently been added for non-executable memory in 64-bit systems as well as some AMD Sempron and Intel Pentium 4 processors. Windows also provides limited software support in the form of Data Execution Prevention (DEP) as of Windows XP Service Pack 2 and Windows 2003 Server.
- [ref 3] PaX. <http://pax.grsecurity.net/docs/pax.txt>
- [ref 4] Butler, James, "VICE - Catch the hookers!" Black Hat, Las Vegas, July, 2004. www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf

5.1 Further reading

Rutkowska, Joanna. "Concepts For The Stealth Windows Rootkit", Sept 2003
http://www.invisiblethings.org/papers/chameleon_concepts.pdf

Russinovich, Mark and Solomon, David. Windows Internals, Fourth Edition.

6. About the authors

Jamie Butler is the Director of Engineering at HBGary, Inc. specializing in rootkits and other subversive technologies. He is the co-author and a teacher of "Aspects of Offensive Rootkit Technologies" and co-author of the newly released bestseller "[Rootkits: Subverting the Windows Kernel](#)".

Sherri Sparks is a PhD student at the University of Central Florida. Currently, her research interests include offensive/defensive malicious code technologies and related issues in digital forensic applications.

Copyright © 2005, SecurityFocus

[Privacy Statement](#)
Copyright 2005, SecurityFocus