October 2015

Kernel Integrity Analysis

Caleb Martin Stepanian
Worcester Polytechnic Institute

Project CS2 AAVR

Kernel Integrity Analysis

Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Computer Science
By

Caleb Stepanian
cmstepanian@wpi.edu

Submitted On: October 27, 2015

Project Advisor:
Professor Craig Shue
cshue@cs.wpi.edu

Abstract

Rootkits are dangerous and hard to detect. A rootkit is malware specifically designed to be stealthy and maintain control of a computer without alerting users or administrators. Existing detection mechanisms are insufficient to reliably detect rootkits, due to fundamental problems with the way they do detection.

To gain control of an operating system kernel, a rootkit edits certain parts of the kernel data structures to route execution to its code or to hide files that it has placed on the file system. Each of the existing detector tools only monitors a subset of those data structures.

This MQP has two major contributions. The first contribution is a Red Team analysis of WinKIM, a rootkit detection tool. The analysis shows my attempts to find flaws in WinKIM's ability to detect rootkits. WinKIM monitors a particular set of Windows data structures; I attempt to show that this set is insufficient to detect all possible rootkits. The second is the enumeration of data structures in the Windows kernel which can possibly be targeted by a rootkit. These structures are those which a detector would have to measure in order to detect any rootkit. This should facilitate future improvement of rootkit-detection tools.

Contents

1 Introduction

Rootkits are extremely dangerous and threatening, and their defining characteristic is stealth [6]. A rootkit is malware that tries to maintain administrator/superuser privileges on a computer, while hiding the fact that the computer is compromised. Rootkits do this by manipulating the kernel of the infected machine so that it reports incorrectly about parts of the kernel where the malware resides. The state of the art in detection of this sort of stealth is insufficient and the situation is biased strongly in favor of the attackers. A clever rootkit author can create rootkits that are practically impossible to detect except with great resources and the manual effort of experts.

Most currently-deployed detection techniques use signatures. This is where software scans the contents of the computer, searching for traces of *already known* malware, which has been previously reported and studied. This is a major flaw, because it is then impossible to detect so-called "zero-days", when a rootkit is first released into the wild. Rootkits are not prevented from infecting machines the first time; the scanners can only detect them once someone has found the rootkit and reported it to the anti-malware vendors.

Additionally, those detection methods run at exactly the same privilege level as the malware they are trying to detect. The scanner has kernel-level privileges (and can read and write to core parts of the kernel at will), but so do the rootkits it is trying to detect. The rootkit author can anticipate this and write special cases into their code to manipulate scanners as well. This behavior is commonly observed in real rootkits.

Rootkit infections are severely damaging. They can involve stealing valuable secrets and sabotaging industrial processes, and are very difficult to clean up because it is difficult to be sure that all traces have been removed, especially on a corporate network of even average size.

The lack of reliable detection methods leaves users and administrators in a situation where they must always suspect that their computers are compromised. Better techniques are required.

There is research in better techniques for rootkit detection. WinKIM, a MITRE research project, detects rootkits by comparing kernel data structures against known goods and developer-specified invariants, from the vantage point of a hypervisor. Copilot [9] is similar in that it compares the kernel with known goods, but it gets its privilege separation by running in a PCI-attached peripheral device and accessing physical memory via DMA (Direct Memory Access). Gibraltar [4] is another rootkit detector which is novel in that it uses the Daikon invariant inference engine to infer invariants of the kernel under inspection, and then enforces those invariants using a PCI-attached device like Copilot does.

My MQP was completed with the WinKIM team, where my task was to improve WinKIM by performing a Red-Team analysis of it.

2 Background

This section explains important background concepts relating to computer security in general, the WinKIM rootkit detector, the Windows operating system, the practice of Red Teaming, assembly code, and self-modifying code. This section concludes with a survey of related works.

2.1 Security

Design and engineering can be challenging, but they are at least straightforward when all one wants to do is make a product that essentially works most of the time. Things get much more interesting when one wants the product to be reliable in the face of random failures and environmental interference. Writing code that is meant to be reliable is described by Ross Anderson [2] to be like programming Murphy's computer. As with Murphy's Law, one must expect that everything will go wrong. Every error code must be checked, no matter how rarely it is a problem. The code has to try its hardest to run or at least fail gracefully in every possible circumstance.

Programming for security involves completely different considerations. One must write the code as if a malicious part of the computer could provide intentionally wrong answers to some kinds of query, specifically with the goal of making code behave in a way beneficial to the attacker. Anderson calls this programming Satan's computer [2]. The attacker has transformed from the mere bad luck of the universe, to a malicious intelligent agent who will do everything in their power to cause bugs to reveal secrets or redirect execution.

It is for this reason that designing, engineering, and implementing secure products is very hard and requires large investments to be effective.

2.1.1 Attestation

A major topic in security is attestation, the process of validating the integrity of a computer. This means making sure that it is running a correct copy of the code that was originally intended by the designers to run on it, with a valid internal state. Attestation somehow compares the running state of the computer with a "known good" state, determining whether the computer is functioning correctly and should therefore be trusted.

One important example of attestation is the Trusted Platform Module (TPM) that is installed on many motherboards in both PC and Mac systems. It is a small chip that is capable of cryptographic processing and reading configuration information about the computer. TPMs were designed to support a feature known as Trusted Computing, where the TPM can attest that the computer booted in a particular trusted configuration, and that there is an unmodified copy of software running, such as Windows. TPMs were initially intended to enforce Digital Rights Management (DRM), by making it much harder to copy copyrighted media without permission [1]. They can also support disk encryption and various cryptography-related features.

The purpose of rootkit detectors like WinKIM, Copilot and Gibraltar is a form of attestation: they attest that the machine they measure is in a non-compromised state, and is running without interference by malware. All three of these rootkit detectors have a form of privilege separation, just like the TPM. WinKIM's privilege separation is due to its placement in a hypervisor, Copilot and Gibraltar both make use of a trusted PCI-attached coprocessor, and a TPM is also a trusted coprocessor, but has even more access to the system than PCI devices do.

2.1.2 Forensics

A discussion of rootkits would not be complete without mention of forensics, the science of collecting evidence. Rootkit behavior is entirely about anti-forensics, the destruction or hiding of evidence [6]. Practitioners of computer forensics often must use specialized tools for data recovery and search to obtain anything useful from a computer system. Depending on the situation, they might take memory dumps from a running system that is currently under attack or suspected of being compromised, or they may have access to the hard drives of a shutdown computer. Their goal is to discover whether attackers have manipulated the computer, and if so, who the attackers are, when they performed the attack, what they did and how, and what their motivations were. The evidence obtained by computer forensics teams is used to help administrators and developers debrief on what went wrong and how to fix it in the future, and to prosecute the attackers in court if possible.

One of the biggest challenges for a computer forensics team is rootkits, because they are specifically designed to be difficult for forensics professionals to find [6]. Forensics professionals have only limited time to search for problems, and the harder it is to discover how a rootkit is keeping its hold on a system, the more likely it is that the team will give up. Rootkits can be so stealthy that they are very hard to notice, and even harder to detect with certainty. But if the rootkit author makes any mistakes in the code, the rootkit might leak information that leads to its detection. Rootkit authors are thus in the business of anti-forensics. They need to be sure to destroy data as soon as it is no longer needed, and hide anything else very carefully.

2.2 WinKIM

WinKIM (Windows Kernel Integrity Monitor) is an NSA-funded MITRE project that started in the late 2000s, that attempts to solve two fundamental issues in current rootkit detection, those being signature-based detection and lack of privilege separation. WinKIM is based on the hypothesis that: there exists a set of *key data structures* in the Windows kernel, some of which *must* be modified by any rootkit. By measuring and appraising all of these key data structures, one can detect any rootkit, even those created in the future with knowledge of how WinKIM detects rootkits.

This means that WinKIM detects rootkits by what they are, by definition, rather than by a mere symptom of their presence. To avoid detection by WinKIM, a rootkit would have to lose the ability to hook or modify anything useful, if indeed it could still hide itself at all.

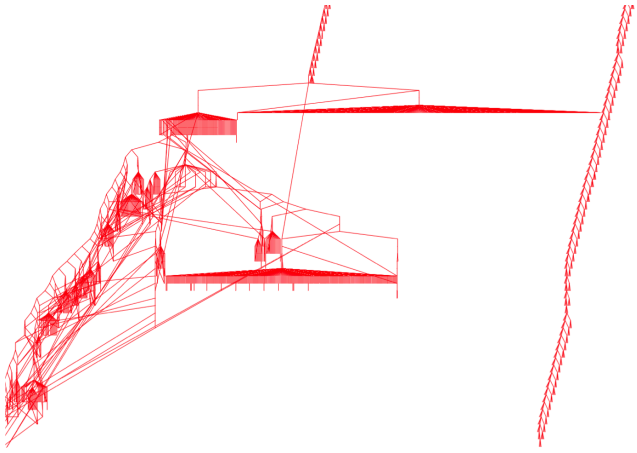

Figure 1: A measurement graph generated by WinKIM [5]

In concrete terms, WinKIM runs in a Xen hypervisor as a privileged Domain, which is just Xen's term for Virtual Machine (VM). The Xen hypervisor is a small operating system that runs directly on a machine's hardware, and supports running multiple guest operating systems in isolated virtual machines. A virtual machine is a simulation of an entire computer, including a CPU, motherboard, peripheral devices, and disks. Software running inside a virtual machine may be able to detect that it is being virtualized, but has no way of escaping or manipulating the hypervisor or other virtual machines on the same hypervisor. Privileged guest VMs are allowed to control the hypervisor and read/write to other guest VMs. WinKIM monitors a Windows guest running under the same hypervisor. The guest Windows machine is an unprivileged VM, so it is unable to access WinKIM or Xen. WinKIM calls functions in the Xen libraries to gain access to the (virtualized) physical memory and execution context of the Windows machine, and periodically performs a *measurement*. In contrast to some other detection tools, WinKIM's measurements are periodic, not continuous, and the Windows system is not being watched in any way by WinKIM between measurements. During a measurement, WinKIM locates the base of the Windows kernel in memory, then recursively parses its data structures, creating a *measurement graph* containing information about the contents of and relationships between all the important data structures in the kernel (see Figure 1). Then, WinKIM performs an *appraisal*, where it compares the measurement graph with known-goods and invariants. Many data structures afford a simple equality check to confirm that they are correct, but others are a bit more dynamic, requiring the specification of rules that tell WinKIM how the

structures should look. The appraisal is WinKIM's output, indicating whether the Windows kernel is corrupted.

WinKIM's primary mode of operation is as just described, with a Xen hypervisor. However, as a way of testing WinKIM more easily, there is a "Hosted Mode" version which runs directly in Windows as a mere device driver. This can be installed very easily on existing installations of Windows without uprooting everything to insert a hypervisor as an intermediate layer. Hosted Mode WinKIM works in largely the same way as Xen WinKIM, except that there are some abilities of Xen WinKIM that are hard to simulate when running inside Windows, like access to the pagefile (there are locks enforced by the filesystem API) and the ability to pause the entire machine.

2.2.1 Threat Model

WinKIM was designed with a particular threat model in mind. A threat model states what kind of adversary the system is intended to protect from, and what the adversary's abilities are. The designers of WinKIM want it to be able to withstand attacks from a knowledgeable threat actor capable of providing an Advanced Persistent Threat (APT). "Knowledgeable" means that the person or group attacking WinKIM will posess expert knowledge about the Windows kernel, possibly including original source code, and about the design and source code of WinKIM. "Persistent" means that the attacker(s) will keep trying, even for years, until they succeed. The attackers will also try to keep control of the system for as long as possible. "Advanced" means that the attacker is also presumed to have expert knowledge in the creation of rootkits and would be able to write a custom rootkit to attack WinKIM.

The threat model also specifies what the attackers want: it is presumed that the attackers want to *stealthily* gain persistent root privileges on Windows computers without disturbing Service Level Agreements (SLA) or alarming administrators or security personnel. "Stealth" here means that their attacks must remain undetected and it must appear that nothing is wrong on the machines. A Service Level Agreement defines a service, including specifications of acceptable amounts of downtime or failures. If an attacker made the system even temporarily unable to perform its normal services, technicians and administrators might notice and then investigate, ruining their stealth. Additionally, the attackers are presumed to need the system running in order to complete their mission or exfiltrate data.

WinKIM's threat model allows that the attacker can only access the Windows VM, not the hypervisor or WinKIM, as they have only remote access to the Windows machine. The attacker is expected to be able to get their own code to run in the kernel, at least briefly, perhaps by exploiting a privilege escalation attack or installing a Trojan driver that is signed with a stolen certificate. The attacker will want to hide their code as soon as possible to avoid detection by WinKIM.

2.2.2 Xen

Xen is a Type-1 hypervisor, meaning that it runs directly on physical hardware and allows virtual machines, called Domains in Xen parlance, to be run. This is in contrast to a Type-2 hypervisor, like VirtualBox or VMWare Player, which run on top of a conventional operating system as a program (see Figure 2). Xen itself contains only the kernel necessary to manage VM guests. It plays the role of an operating system, but does nothing but manage virtual machines. Xen allows a special Domain to be present, which has privileges over the other VM guests. This special Domain is called the Dom0. WinKIM runs in the Dom0 because it needs the extra privileges to read directly from the contents of other virtual machines, and to pause and unpause them. The Windows machine under test by WinKIM runs in an unprivileged Domain called a DomU.

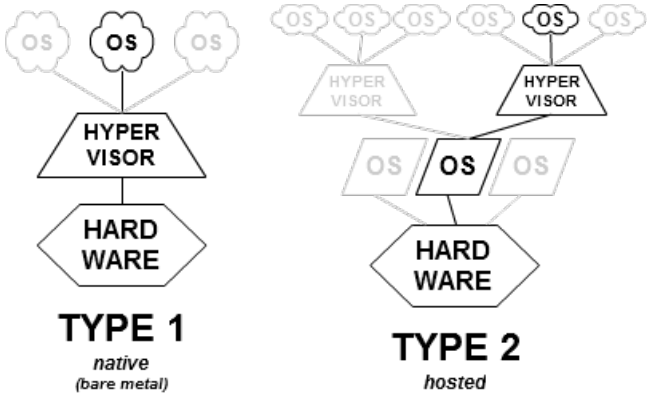

Figure 2: Type 1 and Type 2 hypervisors [13]

2.2.3 PDB files and symbols

Microsoft provides debugging symbols for their programs and the kernel in files with the `.pdb` suffix, meaning Program Database. PDB files document the offsets into the binary code where particular functions are implemented, without revealing the source code. WinKIM greatly benefits from this symbol information, because it is the best information available on how Windows operates, without actually obtaining the source code. The symbols also enable WinKIM to compute the addresses of structures in a running kernel, relative to other known addresses.

2.3 Windows Internals

Detailed understanding of the internals of Windows is important when trying to create rootkits or software to detect rootkits. Windows is not very well

documented by Microsoft itself; these internals can be learned from the Windows Internals book [12], hacker publications like Uninformed, and the blogs of researchers. It is also sometimes necessary to talk directly to experts.

2.3.1 Pagefile

The Windows pagefile is a file on disk where Windows can temporarily store the contents of memory, if it runs out of real physical memory. This process is transparent to programs, because of Virtual Memory. Programs see a flat address space of memory they can use (called virtual address space), but it is translated by the operating system and the processor into physical memory addresses using the page tables. Part of this translation involves checking whether a given virtual address points to memory that is currently in physical memory, or if it is "paged-out" and inside the pagefile. Unlike with UNIX kernels, parts of the Windows kernel get paged out from time to time.

Since rootkits have full control of the kernel, and the kernel can choose how it pages-out memory, it is expected that rootkits could force their own memory pages to be paged-out at any time. This could be done by making sure the pagefile contains a copy of the data (which takes some time because disks are slow) and then manipulating the page tables to claim that the data is not in physical memory and only available in the pagefile.

2.3.2 System Calls

Rootkits often perform their job by hooking system calls, because they are the interface that the kernel exposes to both userspace programs and device drivers. If a rootkit can make the kernel provide modified results for some queries, it can make everything look completely normal, as if there is no rootkit at all. The Windows kernel has hundreds of system calls, of various categories. For example, process management functions have the prefix `Ps`, security functions have the prefix `Se`, and `Io` is for I/O functions. Some system calls are privileged, and can only be called from kernel-mode code, like in device drivers. These system calls are prefixed with either `Nt` or `Zw`.

Windows handles system calls with multiple layers of abstraction, so there are multiple locations in which hooking can be performed. Userspace Windows programs and device drivers link with DLLs like `ntdll.dll`, which either implement the code themselves or just wrap the real system calls contained in `ntoskrnl.exe` [12].

2.3.3 Tables

There are two tables in the kernel of major significance, those being the System Service Descriptor Table (SSDT) and the Interrupt Dispatch Table (IDT). The SSDT informs the kernel where it has implementations of system calls, so that it knows where to jump to when a user-mode program tries to use a system call. The IDT, similarly, contains pointers to code implementing handlers for hardware interrupts [12]. Hardware interrupts are generated by attached peripheral

devices, signalling to the CPU that, for example, a buffer is now ready to be copied out of the device.

Both of these tables are prime targets for rootkits because it is possible to simply overwrite an entry in either of these such that it points to the attacker's code. Then, the attacker's code will be called every time those events occur instead of the real kernel code. This is useful for staying persistent and for hooking interesting behaviors in the kernel for stealth. For example, a rootkit may want to hide files on the filesystem, and it could do this by hooking the system calls that involve reading from the filesystem, and inject special cases into the code that specifically ignore the files placed by the rootkit.

It is relatively straightforward to detect hooking of table entries, though, because they are almost entirely static for a given version of Windows (and are thus predictable). A tool can read the contents of the table, and compare it byte-by-byte with the copy of the table found in a pristine installation of Windows. Any differences are corruption, possibly caused by a rootkit. WinKIM is very successful at checking these tables.

2.3.4 PatchGuard

PatchGuard (also called Kernel Patch Protection, or KPP) is a part of the kernel which periodically reads important parts of the kernel data structures, and ensures that they hold their integrity. PatchGuard was first added in Windows 7 64 bit, to strongly encourage third-party device driver and antivirus developers not to hook the kernel in order to operate, but rather use other, safer alternatives. If it detects hooks, it immediately crashes the system with Bug Check 0x109, `CRITICAL_STRUCTURE_CORRUPTION`. PatchGuard operates by caching known-good copies or checksums of the protected structures, and periodically checking the in-memory versions for correctness. It checks at a random interval that is roughly five to ten minutes [15]. PatchGuard checks many things, but the most important ones are the System Service Descriptor Table (SSDT), Global Descriptor Table (GDT), Interrupt Dispatch Table (IDT), the system images (`ntoskrnl.exe`, `ndis.sys`, `hal.dll`), and the processor Model Specific Registers (MSRs) relating to handling the `syscall` instruction. It also tries very hard to hide itself and make sure that tampering with PatchGuard itself is difficult. There are no Microsoft-provided symbols that document where PatchGuard is implemented in the kernel binary, but researchers have had mixed success in reverse-engineering it and disabling it [15]. As it runs inside the Windows kernel, trying to protect the kernel, it has no privilege separation to protect itself from device drivers.

2.3.5 Driver Signing

Microsoft has configured Windows to only run code from drivers that have been signed by a driver-signing certificate. This makes it fairly difficult for malware to cause code to run in kernel space, because their driver needs to be signed by a legitimate signing certificate.

Digital certificates are a way to prove that a particular message (in this case the entire contents of the device driver) was created by a trusted authority (in this case an authentic company which creates legitimate device drivers). This is achieved by creating a pair of keys, one "public" and one "private". The company "signs" the driver by computing a function of the driver and their secret private key, and including this "signature" with the driver. Then, anyone who has the signed driver (simply the signature and the driver together) and the public key can verify that the signature is valid. Any tampering with the signature or driver would make it fail the check. The signatures are computed in such a way that it is extremely difficult (in terms of requiring huge amounts of computing power for brute-force searching) to fake a certificate without actually posessing the private key.

The certificate model for driver signing is similar to that of SSL CAs. Microsoft provides signing certificates (private keys) to companies which want to publish Windows drivers, who then sign their drivers with the provided private key. The kernel will only load a driver if it has a valid certificate. Driver signature validation can be disabled, by setting the operating system into "Test signing mode", which exists so that driver developers can easily test their changes. This mode puts obvious watermarks on the Desktop indicating that the system is in test-signing mode [12]. Since many companies develop drivers, and Microsoft cannot practically do a thorough investigation of every individual company to whom it provides signing certificates, it is plausible that an attacker could obtain legitimate-appearing certificates on malicious drivers. This could be done by masquerading as a legitimate company, or by stealing a copy of a signing certificate from a company's systems.

2.3.6 Process List and Thread List

Rootkits commonly try to hide processes from users and monitoring tools, so that it looks like no malware is running at all. This hiding is done by manipulating the process and thread lists.

Windows has separate notions of "process" and "thread", similar to UNIX. A process is in control of one or more threads. Scheduling is done at the resolution of threads. Programs that are meant to display which programs are running generally only ask the kernel about running processes, whereas the kernel internally only cares which threads exist and need to run at a given time. Process information and thread information are stored separately, so this means that it is possible for rootkits to manipulate only the process information, like hiding a process, but leave the thread-related datastructure alone so that the kernel will still schedule it.

The process and thread lists are both implemented as doubly-linked lists, so it is quite straightforward to arbitrarily insert and delete items without moving memory around.

On the other hand, it is not difficult to detect that the process and thread lists are inconsistent, which would point to malicious manipulation. This can be checked because the process list contains pointers to all the threads contained

in a given process. A tool like WinKIM can compare the two lists and report whether there are threads in the thread list which have no corresponding owner process in the process list, which would indicate that a rootkit is trying to hide the process that owns those threads.

2.3.7 System Threads

Just like with user-mode processes and threads, system threads are a target of rootkits. They are one of the ways that kernel code can be scheduled to run repeatedly.

Windows, like UNIX, has the concept of threads running in kernel mode, doing various system tasks that must be executed periodically. These are called "system threads" by Microsoft, and they can be started by an API call accessible only to other kernel code. System threads are attached to the System process by default, which is a catch-all process in the kernel used merely to organize most kernel threads together. System threads are periodically scheduled by the scheduler, and are visible to the scheduler's data structures.

2.3.8 Deferred Procedure Calls (DPCs)

Deferred Procedure Calls are somewhat like system threads in that they are a way of scheduling code to run with kernel privileges. Rootkits would find them just as useful for getting their code to run.

Windows, like any operating system, has interrupt handlers, which are pieces of code that handle events triggered by external hardware. Due to real-time requirements, such code needs to be highly optimized and not perform any operations that could reasonably be done at a lower scheduling priority. But, the interrupt handlers need some way of scheduling the lower-priority work to be completed eventually. There are multiple solutions, but Microsoft implemented Deferred Procedure Calls, which are designed to allow interrupt handlers to schedule work to be done eventually. DPCs are not tied to any particular process, thread, or kernel thread, and their scheduling system is independent of those.

2.4 Red Teaming

Most non-trivial software created by humans has bugs [1], because as new features interact with old features, it can become more and more difficult for the programmer to keep all possible interactions actively in mind. Sometimes bugs are exploitable by hackers. Such exploitable bugs include buffer overflows, logic errors in special cases, and command injection. While standard testing procedures used in software engineering can be used to get some assurance that there are few to no bugs affecting functionality, it is very difficult to test to assure that the software has no exploitable vulnerabilities [3]. Because vulnerabilities are discovered and exploited by hackers, one way to find vulnerabilities in one's own software is to pay hackers to find them. This practice is known as Red

Teaming or Penetration Testing, where the "Red Team" is the team of hackers hired to spend time trying to exploit the software. Red Teaming/Penetration testing can only be effective when used throughout the Software Development Life Cycle (SDLC), and not just as a final checklist item [3].

2.5 Assembly and machine code

When discussing hooks, it is necessary to understand how the processor interprets programs at the lowest level. The lowest level of abstraction in a computer is machine code, which is a binary packed format that is not meant to be human-readable. For example, the machine code for the instruction `inc rax` is `48 ff c0`. But, humans find it easier to write in higher-level languages like C, which are compiled down to the machine language that the processor can directly execute. There is also an intermediate language called assembly language, which is human readable and has an almost one-to-one mapping with machine code. It is thus very easy to write a straightforward program that converts assembly code to machine code or machine code to assembly code. These are called assemblers and disassemblers, respectively.

However, it is not at all straightforward to write a program that consumes machine code and generates the C source code that created it. Such a program is called a decompiler, and they are hard to make because the mapping between C and machine code is many-to-many, in that there are many possible machine code programs for a given C source file (due to differences in optimization, or arbitrary choices by the compiler), and there are also many possible C sources that could have generated a given piece of machine code, because C is a relatively expressive language and there are multiple ways to write the same thing.

Microsoft's Windows operating system, like most proprietary products, is distributed to users in an entirely binary form, which is synonymous with precompiled machine code. The intention of this is that competitors to Microsoft will have a difficult time reverse-engineering the operating system in order to copy its appearance or behavior. It also makes it difficult for attackers to learn how Windows is vulnerable, or for defenders to learn how to protect Windows. When developing rootkits or anti-rootkit technology for Windows, then, one must be able to reverse-engineer parts of Windows by examining the machine code using a debugger and disassembler. WinDBG provides the `u` command (for Unassemble) which will disassemble the bytes at a given address as if they are machine code.

2.6 Self-modifying Code

Malware often uses self-modifying code as a technique for obfuscation, because the effect of a piece of such code is not obvious from looking at it. To reverse engineer it, one would have to run it on a real or virtual machine to inspect what its code finally becomes, after it computes new values for its own machine code and rewrites itself. This takes a large amount of work, so it discourages forensics professionals from trying to understand the code.

There are two main ways of organizing memory in a computer, the Von Neumann architecture and the Harvard architecture. In the Von Neumann computer architecture, there is only one kind of memory, and parts of it can be data or code, depending on interpretation. This is much more powerful than the Harvard architecture, which strictly separates code and data into two memories, usually without easy ways to modify code memory while running. Since modern computers use the Von Neumann architecture, not the Harvard architecture, code can be written such that it writes to the memory containing its own code, changing its own behavior in future runs. Self-modifying code allows for very useful tricks, like Just-In-Time compilation (JIT) and tuning algorithms at runtime to suit inputs. It is also fairly dangerous, in that it makes debugging very difficult, and opens up more attack surface for malware, because it requires parts of memory to be marked both writable and executable. This would allow an exploit to write code to that memory which would subsequently get executed.

2.7 Related Work

There are research projects that are similar to WinKIM in various ways. XenKIMONO [10] is a kernel integrity monitor very similar in scope to WinKIM's sister project LKIM, which protects Linux kernels. XenKIMONO also virtualizes the protected kernel in a Xen VM, and also periodically parses the data structures of the virtualized kernel and compares them against known goods. Copilot [9] is another rootkit detector which executes on a PCI-connected device, using the PCI protocol's ability to read directly from physical memory to scan the kernel. This provides the same sort of privileged execution that WinKIM gains by running in a Xen hypervisor. This has the added cost of requiring additional hardware to be purchased and installed in the machines to be tested, and this will only work on desktop-size computers with user-customizable peripheral slots. Rhee et al demonstrate a Virtual Memory-Manager-based rootkit detector [11], which hooks the translation of virtual addresses to physical addresses, and preemptively detects malicious accesses to parts of memory. This rootkit detector virtualizes the protected kernel in the QEMU VM for its privilege separation.

There have also been attempts at using machine learning to help identify rootkits; Gibraltar [4] uses the Daikon invariant inference tool to automatically recognize invariants in kernel code, and then automatically enforces that those inferences hold true. Limbo [17] is a driver loader for Windows which runs drivers in a sandbox, measuring patterns in their execution, and using a Bayes classifier to identify malicious looking execution patterns. It will only load drivers which do not appear malicious. If machine learning became popular as a way to detect rootkits, rootkit authors would just start making their rootkits "similar" to code marked benevolent, which would make it very hard for machine learning alone to properly detect the difference.

There are other methods of blocking malware, such as Bit9's Endpoint whitelisting. This software is installed with kernel privileges, and will only

run programs that match a whitelist generated by Bit9 and the administrators of the machine. This can be very effective, but it can be very difficult to administrate the maintenance of whitelists and still allow work to be accomplished. WinKIM does not require administrators to manage anything.

3 Approach

My Red-Teaming goal was to create a rootkit for the Windows kernel, and that rootkit also needed to be undetectable by WinKIM. Since WinKIM only tries to measure certain specific parts of the kernel, I wondered whether it is possible to operate maliciously inside the kernel while not touching the parts measured by WinKIM. I thought it might be possible to violate WinKIM's core assumptions, by showing that more data structures need to be measured than currently are, or that it is possible to create a useful rootkit that resides only in writable (and therefore not appraised) memory.

There are some rootkit techniques that are definitely unavailable, on account of WinKIM's scanning, such as directly hooking the SSDT or IDT, or inline-hooking functions in `ntoskrnl.exe`. But perhaps those techniques merely need to be augmented to bypass detection. I surveyed the workings of the Windows kernel, noting important data structures, and how they work in normal operation. I also found how rootkits hook and manipulate these structures, and learned what WinKIM does in particular to protect the structures. I particularly noted where a structure needs protection but WinKIM's current abilities are insufficient. With some structures, I found hypothetical attacks that might be effective against WinKIM. I will discuss those attacks in detail and what WinKIM should do about them.

3.1 Quality Assurance by Red Team

Red Teaming is used here as a form of quality assurance testing, to make sure that the tool behaves as intended when installed and running on a real system, defending from real attacks. Red Teaming is most often seen as a team of experts who have a large kit of tools, like scanners, fuzzers, and weaponized exploits, which they simply deploy to test the system for vulnerabilities to already-known attacks. This is often good enough for clients who have a standard common setup, like a web site with a database and a network of office PCs. A standard Red Team attack is not appropriate for WinKIM, as there do not exist (known to the author at time of writing) tools for automatically generating rootkits to test with a rootkit detector. Also, a significant part of this effort was to determine whether there are design issues in WinKIM; there do not currently exist automated tools to assist humans with any security design work. Therefore, if one wants to test the effectiveness of their rootkit detector, they will have to do it all manually.

3.2 My Background

Creative Red Teaming is a big thing for one person to do, but I have an unusual amount of experience that prepared me for this. I took an assembly course in which I learned how to reverse-engineer a binary "bomb" using a disassembler and debugger, all while preventing it from "exploding" (contacting the grading server). The binary was a Linux x86 executable. I am a member of the Cyber Security Club at my university, and our team has competed in many Capture The Flag (CTF) competitions. These competitions are one of the best ways to get into the hacker mindset, of thinking always how systems can be broken, and subverted, and turned to the hacker's will. Programmers and engineers normally only think about how their systems will work in the expected cases. A hacker thinks about it the opposite way; they have to think about how the system might fail to work. I also took a course in Software Security Engineering, which gave me experience in Red Teaming/Penetration Testing. In the course, we were provided with a vulnerable PHP web application, and instructed to crack it in as many different ways as possible, documenting our findings. We had to provide exact details of how we subverted the system, and what that would allow an attacker to access or modify.

3.3 Discussion with Team

I asked the WinKIM team about how the tool works, and specifically how it might be incomplete in its detection. It is known by the team that WinKIM does not measure or appraise certain dynamic parts of the kernel, such as the list of kernel threads running, or the DPC buffers. WinKIM also did not (at the time) support reading from the Windows pagefile. The team intends for WinKIM to support measuring all important parts of the kernel.

3.4 Research on Rootkits

As WinKIM is designed to stop rootkits, I needed to learn what a rootkit is. I found the book *The Rootkit Arsenal* by Bill Blunden [6], which gives an overview of what they do and why. In particular, it explains how 32 bit Windows operates and how to subvert its datastructures, with complete code examples. It was very useful but I had to do a lot of research to translate its 32 bit explanations to modern 64 bit Windows.

I researched two well-known rootkits, Stuxnet and Uroburos. Both of them are well-publicized for their real-world usage and effectiveness. There is documentation about how they work and how they were used in the form of dossiers [8, 14]. From what I read about both Stuxnet and Uroburos, they hooked parts of the Windows kernel that WinKIM should be able to detect. Accordingly, I explored actually running them.

The team has access to an isolated malware-testing laptop, meaning it is not allowed to be connected to the internal network or any writable media. The way to get data on the machine is by burning CDs and then inserting them into

the laptop. The CDs must then never be inserted into a non-isolated machine. This strict protocol makes it extremely unlikely that any malware could escape the isolated machine.

The isolated machine is set up with WinKIM and a guest Windows VM. I copied our sample of Stuxnet onto the Windows VM, and ran it, but unfortunately, Stuxnet is 32-bit only, while WinKIM supports only 64-bit Windows.

Uroburos is a much more recently-made piece of malware, and therefore it does support 64-bit Windows. It actually supports both 32-bit and 64-bit Windows, due to a well-funded development. Uroburos is a rootkit whose origins have been traced to Russia, and it is called "crimeware". It takes control of Windows computers and copies data out of them, sending it to the creators of the rootkit.

The team also tried running Uroburos on the isolated laptop and successfully got it to infect the machine. WinKIM was able to detect some of its behavior (it inline-hooks some system calls, like `ZwQueryKey`, `ZwReadFile`, and `ZwQuerySystemInformation`) but did not notice that it had modified some kernel memory that was paged-out. WinKIM also did not notice Uroburos's use of Deferred Procedure Calls (DPCs), because WinKIM does not currently appraise DPC scheduling data structures.

3.5 Pagefile

The team had mentioned to me that the pagefile is currently something which WinKIM cannot measure. The Windows pagefile is a file on the disk which is occasionally used by the kernel to store data which it cannot fit in memory. In contrast to UNIX, parts of the Windows kernel are actually marked pageable; the kernel is allowed to free some memory by temporarily moving such parts of the kernel code into the pagefile on disk. At the time I started this project, WinKIM was only capable of reading from the memory space of the Windows guest. Therefore, some parts of the kernel which would be desirable to measure were beyond reach. The WinKIM team would naturally like the tool to have as wide a view as possible on general principles, but a more compelling reason was that the Uroburos rootkit modified some parts of paged-out memory.

It is plausible that the pagefile could be used as part of a rootkit's stealth; if the rootkit could detect when WinKIM was about to perform a measurement (as the measurements are periodic, not continuous), it could force its own code to be paged-out, temporarily evading detection. This would have to be combined with other techniques to be a complete rootkit.

I fixed this entire issue by adding pagefile-reading support to WinKIM. In WinKIM's Hosted Mode, it appears impossible to read from the pagefile, because the kernel maintains locks on it. These locks prevent concurrent reads, even with administrator privileges. It turns out, though, that such locks are not enforced when the code directly accesses the disk and parses the filesystem itself. The Sleuth Kit, an open-source library for digital forensics, is just the tool for the job. It can parse NTFS, the primary filesystem format used by Windows. It was quite straightforward to use the Sleuth Kit API to open the

`C:` drive, parse the file system, and search for the file named `pagefile.sys` located in the root of the directory tree:

```
TSK_TCHAR *images[] = {L"\\\\.\\c:"};
TSK_IMG_INFO *img = tsk_img_open(1, images, TSK_IMG_TYPE_DETECT, 0);
TSK_FS_INFO *fs = tsk_fs_open_img(img, 0, TSK_FS_TYPE_DETECT);
tsk_fs_ifind_path(fs, L"pagefile.sys", &inum);
TSK_FS_FILE *pagefile = tsk_fs_file_open_meta(fs, NULL, inum);
tsk_fs_file_read(pagefile, offset, buf, len, TSK_FS_FILE_READ_FLAG_NONE);
```

Then, once I had access to the raw bytes of the pagefile, I modified WinKIM's page fault handler (WinKIM does all the virtual address translation itself). I changed it so that it reads from the pagefile when the Page Table Entry (PTE) has the flags that mean the page is not in physical memory, but rather on the disk. The format of page tables on the x86 architecture is mostly specified by Intel in their documentation, but the operating system is free to choose the format of the PTE when the Valid bit is not set, meaning the page is not in memory. This allows each operating system to perform its own handling of paging to disk, whether that involves a pagefile or a swap partition. To find out how Windows formats its PTEs, I opened WinDBG in kernel debugging mode and typed the command `dt _MMPTE_SOFTWARE`. This prints out the definition of the internal structure representing PTEs:

```
kd> dt _MMPTE_SOFTWARE
nt!_MMPTE_SOFTWARE
   +0x000 Valid            : Pos 0, 1 Bit
   +0x000 PageFileLow      : Pos 1, 4 Bits
   +0x000 Protection       : Pos 5, 5 Bits
   +0x000 Prototype        : Pos 10, 1 Bit
   +0x000 Transition       : Pos 11, 1 Bit
   +0x000 UsedPageTableEntries : Pos 12, 10 Bits
   +0x000 InStore          : Pos 22, 1 Bit
   +0x000 Reserved         : Pos 23, 9 Bits
   +0x000 PageFileHigh     : Pos 32, 32 Bits
```

I learned from an article written by the authors of The Sleuth Kit [7] that the `PageFileLow` member indicates which of multiple pagefiles contains this page (most systems have only one pagefile), and the `PageFileHigh` member is the offset into that pagefile, in pages. On the versions of Windows under test, a page is always 4096 bytes. Using the lower 20 bits of the virtual address (the offset within the page), combined with the offset into the pagefile, my code handles requests to paged-out virtual memory as if it was any other kind of memory. This enabled WinKIM to greatly expand its view of the kernel, almost doubling the number of measurement nodes in some cases. This had been a long-desired feature of WinKIM.

3.6 Inline-hooking RWX sections

As part of the search for interesting vectors of attack, I used a PE (Portable Executable) dissector tool called `readpe`, and looked at the various sections in `ntoskrnl.exe`. This file is the main executable for the Windows kernel. I discovered that the `INIT` section has the WRITE and EXECUTE flags set, meaning that this section is allowed to have self-modifying code:

```
Section
    Name:                           INIT
    Virtual Address:                0x556000
    Physical Address:               0x576ce
    Size:                           0x57800 (358400 bytes)
    Pointer To Data:                0x4bb400
    Relocations:                    0
    Characteristics:                0xe2000020
    Characteristic Names
                                        IMAGE_SCN_CNT_CODE
                                        IMAGE_SCN_MEM_DISCARDABLE
                                        IMAGE_SCN_MEM_EXECUTE
                                        IMAGE_SCN_MEM_READ
                                        IMAGE_SCN_MEM_WRITE
```

The processor will allow running code to edit this memory, but the processor will also be able to jump to addresses within that range and execute the data there as code. This can be useful, but it is generally quite dangerous. Self-modifying code is most often a concern for stability reasons; it is difficult for programmers to reason about the workings of such code, and so it sometimes hides bugs. Here, though, this is a security concern. By design, WinKIM does not appraise sections of memory marked writable. But here, there is code, inside a writable section, so WinKIM will definitely ignore any hooks added by a rootkit.

Relatedly, there was a bug in certain versions of Visual Studio which caused compiled device drivers to include WRX INIT sections [16]. This broken version lasted for long enough that many device drivers created for Windows Vista through Windows 8.1 have this problem. It would normally require several difficult things at once for a WRX section to be a problem, such as a write-what-where vulnerability, a leaked pointer, and the ability to trigger a driver load from userspace. In the case of WinKIM, this is a different sort of problem, as it is a matter of whether a rootkit can hide useful hooks inside the RWX section.

As its name suggests, the `INIT` section of any PE file is used for one-time initialization. According to Microsoft documentation, this section is discarded after that initialization. I could not find any information on how aggressively the kernel discards such sections; it could be that they stay in memory lazily until the space is needed. I tried looking myself: I used WinDBG on a running Windows system and tried to dump the memory allocated to the `INIT` section. It appeared to be completely deallocated. I tentatively conclude that the only

way to access the `INIT` section would be during boot. Since the `INIT` section is discarded by the time the system is booted, it may be very difficult to abuse this in practice.

To discover whether the INIT section was worth hooking, I needed to find out whether the code in this section was called frequently enough, due to event handling for example. A hook is only useful if the hooked function does something interesting and other code calls it. With the help of IDA Pro, a commercial disassembly tool, I analyzed `ntoskrnl.exe` and listed the functions implemented in the `INIT` section. I had hoped to find any that were important enough to be called by system calls, which would allow me to edit code that was, in some way, executed in response to events. If the functions were called only once, they would not be very useful for persistence.

I expected to eventually find code to hook, so I studied inline hooking. This is where the machine code, in memory, is rewritten to contain other code. Since this involves *overwriting* single instructions, and not *inserting* code, the easiest way is to find a short instruction (2 bytes) in the function preamble, and overwrite it with a short relative jump to nearby unused memory. A short relative jump is encoded by the byte `0xeb` followed by a signed byte offset. In the nearby memory, execute the instruction that was replaced, and then do whatever is desired, followed by another short jump back to the instruction just after the hook. The final effect is that, when the function is executed, it executes the new code just before it does any of its work. This sort of hook is called a prolog detour [6]. These allow an attacker to filter arguments or do special handling in order to implement stealth, like only returning secret information when a special code is supplied by the caller.

I implemented a demonstration hook, by selecting `NtSetValueKey`, a system call that sets a value in the Windows Registry, which is called very often by many programs. The code in memory appeared like this before my tampering:

```
fffff800‘02959dfc 90              nop
fffff800‘02959dfd 90              nop
fffff800‘02959dfe 90              nop
fffff800‘02959dff 90              nop
nt!NtSetValueKey:
fffff800‘02959e00 4c8bdc          mov     r11,rsp
fffff800‘02959e03 45894b20        mov     dword ptr [r11+20h],r9d
fffff800‘02959e07 45894318        mov     dword ptr [r11+18h],r8d
fffff800‘02959e0b 45894b08        mov     qword ptr [r11+8],rcx
fffff800‘02959e0f 53              push    rbx
fffff800‘02959e10 56              push    rsi
fffff800‘02959e11 57              push    rdi
fffff800‘02959e12 4154            push    r12
fffff800‘02959e14 4155            push    r13
fffff800‘02959e16 4156            push    r14
fffff800‘02959e18 4157            push    r15
```

And, after using WinDBG to edit the bytes of memory to contain different machine code, it looked like this:

```
fffff800‘02959dfc 4154            push    r12
fffff800‘02959dfe eb14            jmp     nt!NtSetValueKey+0x14 (fffff800‘02959e14)
nt!NtSetValueKey:
fffff800‘02959e00 4c8bdc          mov     r11,rsp
fffff800‘02959e03 45894b20        mov     dword ptr [r11+20h],r9d
fffff800‘02959e07 45894318        mov     dword ptr [r11+18h],r8d
fffff800‘02959e0b 45894b08        mov     qword ptr [r11+8],rcx
fffff800‘02959e0f 53              push    rbx
fffff800‘02959e10 56              push    rsi
fffff800‘02959e11 57              push    rdi
fffff800‘02959e12 ebe8            jmp     nt!ObpInsertDirectoryEntry+0x8 (fffff800‘02959dfc)
fffff800‘02959e14 4155            push    r13
fffff800‘02959e16 4156            push    r14
fffff800‘02959e18 4157            push    r15
```

The modified instructions and the target addresses of the jumps are highlighted for improved comprehension.

This tiny detour worked flawlessly, though it was completely useless, as my code simply jumps away, performs the push of `r12` which I overwrote with a jump, and jumps back to the original code. This manipulation of read-only memory was only possible because I was using WinDBG to edit the memory. The Windows kernel allows edits by the debugger, to facilitate debugging. Normally, PatchGuard would have noticed that I edited part of the read-only code section of `ntoskrnl.exe`.

I tried to be more ambitious and jump farther away, so that I would have space to implement a real hook, with some form of parameter filtering. The nearby memory just above `NtSetValueKey` (with the `nop`s) only had enough space to do another, longer jump to more distant memory. In the distant memory, I just executed the instructions that had been replaced, and did a long jump back to the next instruction that should be executed in the original implementation. This detour worked, for the most part. I confirmed that it was executed, by placing breakpoints in the original code and my detours, but the kernel kept crashing eventually due to a “Watchdog Timeout” error. Breakpoints sometimes cause the system to misbehave, so I tried it again without inserting any breakpoints at all. Still, the system crashed. I searched the Internet, and asked my colleagues, but I could not figure out how my tiny patch was causing an unrelated-seeming crash. I must have been making some mistake in the setup, because this should clearly be possible.

As discussed in the article about the Visual Studio bug [16], the device drivers shipped with the incorrectly-marked INIT sections did not actually need writable executable memory. The author of that article manually changed the flag in the device drivers to make the INIT sections read-only, and they still functioned as normal. This means that it should be possible to automatically scan for device drivers with RWX INIT sections, and fix them in bulk.

3.7 System Threads and DPCs

Windows has a set of system threads running at all times, with kernel privileges, to complete important jobs. The set of threads running at a given time may not be predictable, and there is no particular table in any static part of the kernel on disk that would indicate which threads should be running. They can start and stop at any time. System threads can do anything the kernel can do, including reading from and writing to anywhere in the kernel address space, user address space, or physical memory. System threads are scheduled in an analogous manner to user-mode threads, in that there are mutable data structures which are read by the scheduler to determine which code to run at a given time. Code running in kernel space can manipulate these data structures at any time, or it can simply call the documented `PsCreateSystemThread` system call to create a new thread.

A system thread simply runs continuously until it terminates itself with `PsTerminateSystemThread`. A system thread cannot respond to events and intercept them, as with traditional rootkit behavior, but it can implement polling behavior to repeatedly check for conditions or read from buffers. For example, a system thread rootkit could act as a keylogger by repeatedly querying the keyboard buffer and copying it out of kernel space, by writing it to disk or sending it over the network.

WinKIM does not currently measure or appraise anything to do with the data structures involved in system threads. This is because WinKIM was deliberately designed to only measure and appraise read-only memory. Read-only memory is marked as such in the page tables by setting a flag on the page frame.

A first approach at appraising system threads might be to whitelist them, perhaps by name or by what code they point to. One way might be to simply check that the pointer to the thread code points into memory that will be appraised. This would ensure that the kernel could only run system threads whose code came from an appraised part of the kernel.

3.8 Inline-hooking PatchGuard

PatchGuard is completely undocumented by Microsoft, by design; if it became easy to understand it and bypass it, driver developers might be encouraged to use such exploits and go back to the hooking techniques they once used. PatchGuard is, in fact, very deeply obfuscated. It modifies itself during runtime, including parts of code that reside in read-only memory, ignoring the access rules that are supposed to apply. This fact has been a cause of frustration to the WinKIM team, as read-only sections are supposed to remain constant, but PatchGuard causes false alarms in WinKIM because it is detected as a corruption. This has forced the WinKIM team to special-case the sections of memory in which PatchGuard resides, simply ignoring changes made to them, considering the changes impossible to predict.

Since WinKIM has been forced to ignore PatchGuard, it may be effective to hijack PatchGuard itself. A rootkit could rewrite code in PatchGuard's mem-

ory space, redirecting execution to other attacker-provided code. If the attacker was clever enough to reverse-engineer PatchGuard, they could repurpose PatchGuard to operate as a rootkit and also selectively ignore patches made by the rootkit.

3.9 Device Driver Hooking

Windows allows third parties to install code that will run with kernel privileges, in the form of device drivers. As the name implies, they are usually interface code that allows an end user to plug in some peripheral device to their Windows machine and access it via software. As mentioned in the section on PatchGuard, Windows follows the policy that third party code cannot hook core parts of the kernel. However, a driver can still hook the code of *other drivers* [12]. And since there are very many drivers in the Windows ecosystem, it is totally impractical for WinKIM to successfully appraise all of them to make sure they have not been manipulated. An attacker could figure out how to exploit a device driver that is not appraised by WinKIM, then simply hook that driver to install their rootkit. They can actually install the driver on the system themselves, if it wasn't already there. WinKIM would notice that there was another driver installed, but it currently has no policy on whether that is considered a problem for the machine's health.

WinKIM cannot reasonably cope with an arbitrary set of device drivers present on a machine it needs to appraise. In a real-world deployment of WinKIM, it will be necessary to enforce a whitelist of allowed device drivers.

4 Conclusions

I investigated a variety of paths an attacker might take in attempting to bypass WinKIM. In the case of the pagefile, I successfully modified WinKIM to be entirely immune to that potential problem. With the RWX hooking, it remains an open question whether this could be practically exploited, and what WinKIM should do to mitigate it.

Effectively detecting stealth and anti-forensics techniques is one of the toughest challenges in computer security. Windows, being a complex closed-source operating system with a long history of backwards-compatibility, is probably the most difficult kind of operating system to protect from rootkits. There is a very large space of feature mis-interactions which might cause an apparently-safe system to actually be vulnerable. Being closed source (and also obfuscated in the case of PatchGuard) makes Windows very difficult to protect with a tool like WinKIM. WinKIM's effectiveness depends on a deep understanding of obscure parts of the kernel, and lack of source code or documentation make this extremely challenging.

Since WinKIM is still actively developed and improved, it will likely become better at detecting the possible attacks described here. WinKIM will need to be able to appraise more parts of the kernel that are writable and change over

time, but that follow predictable invariants. In particular, the scheduling data structures for system threads and DPCs are currently a big hole in its view of the kernel.

References

[1] Ross Anderson. *Security Engineering*. John Wiley & Sons, 2008. ISBN: 978-0-470-06852-6. URL: https://www.cl.cam.ac.uk/~rja14/book.html.

[2] Ross Anderson and Roger Needham. "Programming Satan's computer". In: *Computer Science Today*. Springer, 1995, pp. 426–440.

[3] Brad Arkin, Scott Stender, and Gary McGraw. "Software Penetration Testing". In: *IEEE Security & Privacy* 1 (2005), pp. 84–87.

[4] A. Baliga, V. Ganapathy, and L. Iftode. "Automatic Inference and Enforcement of Kernel Data Structure Invariants". In: *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*. Dec. 2008, pp. 77–86. DOI: 10.1109/ACSAC.2008.29.

[5] Erik Beckstrom. *winkim.png*. Personal correspondence. MITRE Corporation. 2015.

[6] Bill Blunden. *Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Wordware Publishing, Inc., 2009. ISBN: 978-1-59822-061-2.

[7] Michael Cohen. *Windows Virtual Address Translation and the Pagefile*. 2014. URL: http://www.rekall-forensic.com/posts/2014-10-25-pagefile.html.

[8] Nicolas Falliere, Liam O Murchu, and Eric Chien. "W32.Stuxnet Dossier". In: *White paper, Symantec Corp., Security Response* 5 (2011).

[9] Nick L Petroni Jr et al. "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor." In: *USENIX Security Symposium*. San Diego, USA. 2004, pp. 179–194.

[10] Nguyen Anh Quynh and Yoshiyasu Takefuji. "Towards a Tamper-resistant Kernel Rootkit Detector". In: *Proceedings of the 2007 ACM Symposium on Applied Computing*. SAC '07. Seoul, Korea: ACM, 2007, pp. 276–283. ISBN: 1-59593-480-4. DOI: 10.1145/1244002.1244070. URL: http://doi.acm.org/10.1145/1244002.1244070.

[11] Junghwan Rhee et al. "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring". In: *Availability, Reliability and Security, 2009. ARES '09. International Conference on*. Mar. 2009, pp. 74–81. DOI: 10.1109/ARES.2009.116.

[12] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals*. Pearson Education, 2012. ISBN: 978-0-7356-4873-9.

[13] Scsami. *Hyperviseur.png*. 2011. URL: https://en.wikipedia.org/wiki/File:Hyperviseur.png.

[14] G Data SecurityLabs. *Uroburos, Highly complex espionage software with Russian roots*. Tech. rep. G Data SecurityLabs, 2014. URL: https://blog.gdatasoftware.com/blog/article/uroburos-highly-complex-espionage-software-with-russian-roots.html.

[15] skape and Skywing. *Bypassing PatchGuard on Windows x64*. 2005. URL: http://uninformed.org/?v=3&a=3.

[16] Graham Sutherland. *W^X policy violation affecting all Windows drivers compiled in Visual Studio 2013 and previous*. 2015. URL: https://codeinsecurity.wordpress.com/2015/09/03/wx-policy-violation-affecting-all-windows-drivers-compiled-in-visual-studio-2013-and-previous/.

[17] Jeffrey Wilhelm and Tzi-cker Chiueh. "A Forced Sampled Execution Approach to Kernel Rootkit Identification". In: *Recent Advances in Intrusion Detection*. Ed. by Christopher Kruegel, Richard Lippmann, and Andrew Clark. Vol. 4637. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 219–235. ISBN: 978-3-540-74319-4. DOI: 10.1007/978-3-540-74320-0_12. URL: http://dx.doi.org/10.1007/978-3-540-74320-0_12.