Protecting Commodity Kernels from Execution of Unauthorized Code*

Glenn Wurster, Paul C. van Oorschot, Trent Jaeger

Abstract

Motivated by the goal of hardening operating system kernels against rootkits and related malware, we provide an overview of the common interfaces and methods which can be used to modify (either legitimately or maliciously) the kernel which is run on a commodity desktop computer. We also give an overview of how these interfaces can be restricted or disabled. While we concentrate mainly on Linux, many of the methods for modifying kernel code also exist on other operating systems, some of which are discussed.

Keywords: Kernel, Rootkit, Swap, Kernel Modules, Memory, Kernel Protection

1 Introduction and Overview

Modern kernels are tasked with many important activities, including providing process isolation, enforcing access controls, and mediating access to resources. All these tasks rely on proper isolation between the kernel and all user-space applications. Any application (even those running as the super-user) must request that the kernel perform certain operations on their behalf, and the kernel has the option of either granting or denying the request. This privilege separation increases both the security and stability of a system – both are directly related to the inability for user-space processes to run arbitrary code with kernel-level control. This paper concentrates on the functionality a kernel intentionally (i.e., by design) makes available to user-space applications; this functionality can then be used by those user-space applications to modify the running kernel and thereby cross process boundaries (even when those processes are run as different users), disable or subvert access control mechanisms, and perform other operations not typically allowed of user-space processes.

We differentiate between the kernel and user-space applications by the same discriminator modern processors use: whether or not the code runs with supervisor level processor control. We do not consider libraries or applications installed at the same time as the kernel to be part of the kernel, since they do not run with supervisor processor control. We focus on protecting the kernel against malicious modifications, allowing the kernel to better enforce non-bypassable protection for user-space applications.

We examine the methods through which applications running with user-level control can elevate their privileges, with the result that they are able to either modify or insert additional code which runs with elevated processor privileges (in effect, modifying the kernel). The

[1]Version: March 15, 2011. Contact author: gwurster@ccsl.carleton.ca

ability for applications to modify the running kernel is well-known – there is rich literature attempting to detect and limit malicious modifications to the running kernel [6, 9, 17, 21, 36, 43, 44, 54, 65, 66]. Our objective is to provide a comprehensive list of the kernel interfaces which must be protected in order to prevent (as opposed to detect) arbitrary code being inserted or modified in the running kernel by user-space applications. Attacks requiring physical access to the target hardware (e.g., [22]), or "live CD" (boot CD) approaches which introduce external bootable operating systems through storage media, are beyond our main focus of standard interfaces that user-space applications may exploit to gain kernel level control. Understanding these avenues of kernel modification available to user-space applications provides knowledge to better protect the kernel against attack.

The remainder of this paper is organized as follows. Section 2 discusses various methods for modifying kernel code. Section 3 discusses restrictions which have been, and can be implemented to protect against each of the methods in Section 2. Section 4 discusses advanced methods which can be used to protect the kernel. We conclude in Section 5.

2 Methods of Modifying Kernel Code

We first provide a brief overview of the different methods of modifying the kernel on a typical desktop, focusing primarily on Linux. Many of the interfaces are also available on other operating systems, with some briefly mentioned.

2.1 Loading Kernel Modules

The easiest way of expanding the *operating system* (OS) kernel with new code that is considered privileged is through loading a kernel module [25, 37, 48]. Kernel modules allow new code to be inserted into the running kernel. This provides a structured, stable way of expanding the functionality of the OS kernel (as opposed to modifying the kernel through the swap file or physical memory access as discussed below, which tends to be fragile).

2.2 Modifying Swap on Disk

In a modern kernel, the swap (or page) area on disk is designated for excess virtual memory allocated by a process (or the kernel) which is not currently being stored in physical memory [58]. The contents of physical memory are written (or paged) to disk and the physical memory reassigned by the OS kernel. The swap area on disk can either be a partition (as is commonly used on Linux), or a file (as is common on Windows and Mac OS). Regardless, if elements of the kernel can be swapped out to disk, and that area of the disk is writable by user-space applications, then there exists the potential to change kernel code or data arbitrarily. Applications such as the Bluepill installer [49] can modify the kernel though forcing kernel pages to be swapped out, modifying those pages on disk, and then causing the pages to be pulled back into physical memory and the code on them run. While Linux, in contrast to Windows Vista, does not appear to swap kernel code out to disk, it still allows kernel data to be swapped, and proposals exist for swapping kernel code [10].

2.3 Memory Access Interfaces

The kernel may export to applications an interface allowing arbitrary access to the physical address space of the machine [48, 51, 63, 64]. This allows an application to:

1. Talk to hardware mapped into memory: On many systems, reads and writes to certain ranges in the physical address space are used for communication with hardware (as opposed to the read/write being serviced by the RAM controller and underlying physical memory) [47]. Hardware devices such as the video card are controlled by accessing specific areas of the physical address space. In Linux, the X server is one such application that relies on being able to access areas of physical memory assigned to the video card.

2. Read from and write to memory allocated to another application already running on the system: While each process on a modern desktop is assigned its own virtual address space, physical memory is divided up across all processes on the system. By accessing the physical address space corresponding to RAM regions used by other processes, a process can modify or examine another process' state on the system.

3. Read from and write to memory allocated to the kernel: Similar to point 2, a process may be able to read and write to physical memory currently allocated to the kernel. While such functionality may be useful for kernel debugging, and entertaining for Russian roulette,[1] the feature is most commonly used by rootkits.

A user-space application will typically open the related device interface exported by the kernel, and then use seek, read, and write operations to modify the contents of the physical address space. Typically, permission to open/use this interface is restricted by the kernel to only applications running with the highest privileges (super user). On Linux, applications gain access to physical memory through the `/dev/mem` device node, with the `/dev/kmem` device node allowing access to kernel memory specifically (other device nodes such as `/dev/mergemem` may also provide access to physical memory). On Windows, the device node for modifying physical memory is `\Device\PhysicalMemory` [13].

2.4 Hardware Interfaces to Kernel Memory

Certain hardware configurations may provide methods allowing a user-space application write access to kernel memory through functionality of the underlying hardware. For example, user-space applications may have access to a device on the FireWire bus. By sending specific commands to this device, it may be possible to have the device read and write to arbitrary areas of physical memory, with the result that the kernel can be modified.

In this paper, we do not provide a complete overview of all hardware which may allow write access to kernel code, but do note that technologies such as direct memory access (or bus mastering) [5, 24], FireWire [18], and ACPI [23] may provide mechanisms for updating physical memory. Furthermore, *memory type range registers* (MTRRs) might be used in modifying

[1] A game where two or more players write random bytes to random areas of kernel memory and the one who crashes the computer looses - `http://lwn.net/Articles/322149/`

the kernel [67]. Because the kernel mediates access to the underlying hardware, applications wishing to make use of hardware in modifying the kernel must request such access through the kernel. This is illustrated in Figure 1, where ultimately the request to modify the kernel travels through the kernel on the way to the hardware.

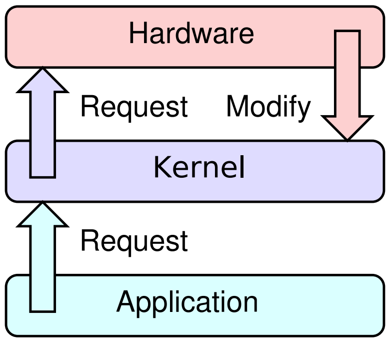

Figure 1: Request flow for modifying the kernel via hardware.

2.5 Kernel Tuning Parameters

Many aspects of the running kernel can be tuned at run-time. While the ability to tune the kernel through modifying values made available to user-space applications does not necessarily result in a compromise, care must be taken in preventing vulnerabilities caused by incorrect parsing of untrusted values provided by user-space applications. Care must also be taken to avoid exporting functionality which can be used to modify the kernel in arbitrary ways [67].

On Linux, the operation of the kernel can be modified through reading and writing to files in `/dev`, as well as to files in the virtual file systems procfs, sysfs [40] and configfs. Parameters passed to the kernel when loading a module or booting the kernel can also be modified. While some kernel parameters exported to user space do not directly depend on specific hardware, others do (e.g., MTRRs [67]).

2.6 Boot Process Mechanisms

During the course of booting a system, typically firmware embedded on the commodity desktop will be responsible for loading and running the boot loader. The boot loader, in turn, is responsible for loading and running the OS kernel.[2] If a user-space application has the ability to modify the firmware, boot loader, or kernel run during the boot process, the application can modify the kernel which is run when the system is next booted. Since the kernel is loaded and executed by the boot loader, the opportunity exists for a modified boot loader to modify the kernel after it is loaded from disk, but before it is run (modified firmware can also alter the kernel [24]). Depending on the firmware used during boot, the boot loader responsible

[2]In the case of Linux, the boot loader may also be responsible for loading an initial RAM disk, which contains kernel modules required in order for the kernel to be able to access storage hardware and continue booting. We discuss protecting the loading of kernel modules in Section 3.1.

for loading and executing the kernel may exist in EEPROM [55], in the boot sector [2], or in a partition [62, 69].

For BIOS based machines, the BIOS will typically load and run the boot loader, which is stored in the first few sectors of the drive, prior to the sectors allocated to the file-system. The boot loader, in turn, is again responsible for loading and executing the kernel. For EFI based machines, a FAT-based [19] file-system is used to store code used during machine boot-up. The EFI approach allows both scripts and executables to be run by the firmware during the boot process, prior to the kernel being loaded and run.

Replacing the running kernel image without rebooting is possible in Linux through the `kexec` system call [45]. This functionality allows the booting of an unauthorized new kernel on a system, effectively allowing arbitrary kernel execution. To protect the kernel, we must also therefore disable the booting of arbitrary kernels through `kexec`.

2.7 Kernel Security Vulnerabilities

Barring the ability to modify the kernel through any of the above methods, the only way left is apparently through discovering and exploiting a security vulnerability in the kernel. Typical application vulnerabilities (e.g., buffer and integer overflow exploits) can exist in the kernel as well, facilitating its compromise. The kernel can additionally be compromised through exploiting unchecked references to application memory (e.g., null pointers, and pointers to application data) [14, 15].

While this paper focuses on the insertion of unauthorized code into the running kernel other than through exploiting software vulnerabilities, many different actions can be performed once access is obtained [26, 34], e.g., hooking system calls [28], hiding processes [66], spawning a virtual machine monitor (e.g., SubVirt [33] and Bluepill [49]) and extracting authentication credentials [32].

3 Basic Protection Mechanisms

In order to secure the kernel, and prevent subversion of kernel access controls by user-space applications, the methods of modifying the kernel as discussed in Section 2 need to be limited. This also prevents applications from destabilizing the entire system (i.e., assuming there are no bugs in the kernel, a user-space application should not be capable of causing system-wide instability). We now discuss, for each method of modifying the kernel, some steps which have been taken to prevent a particular modification technique.

3.1 Through Kernel Modules

To prevent arbitrary code being inserted into the kernel, the loading of kernel modules must be restricted. The commonly accepted way of doing so involves kernel module signing [35, 48]. While not currently accepted into the mainline Linux kernel, a patch does exist to implement this feature [35]. It enforces that only modules signed with the private key corresponding to the public key embedded in the OS kernel can be loaded to extend the kernel. Each kernel module has embedded within it a signature which can be verified using the public verification key embedded in the running kernel. The public key is embedded into the core OS kernel during compile time. Only someone with access to the private key can create a kernel module which verifies and is loaded into the running kernel. Windows Vista introduced a similar approach, preventing arbitrary kernel modules from being loaded unless they are signed by a key recognized by the Windows kernel [35, 48]. While disabling kernel modules entirely is a possible, this requires all hardware device drivers be built directly into the kernel. Typically, kernels without module support are custom-built for a specific hardware configuration.

3.2 Through Swap

The OS kernel is responsible for mediating all access to underlying system hardware, including storage devices. By restricting access to those areas of the disk being used by swap, writes to swap can be prevented (and hence the integrity of kernel memory can be protected). Arbitrary writes to sectors of the disk occupied by swap need to be restricted (e.g., even a root level process must not be allowed to perform arbitrary writes to either `/dev/hda` or `/dev/hda1` if a swap file or partition exists on the drive).

While the Linux kernel does not currently attempt to restrict root's ability to write to arbitrary blocks on disk, Windows Vista does [41]. Vista prevents applications (even those with super-user privilege) from performing raw writes to volumes where the file-system is currently mounted. On Linux, a similar protection mechanism could be added. This protection rule would suffice for protecting both file and partition-based swap against arbitrary modification at the block (or raw) level. To protect against file-based swap being modified, the kernel would need to additionally limit the ability to write to the swap file. In Linux, the ability to modify the swap file is not normally protected by any additional access controls beyond those protecting other (non-swap) files on the system. The Linux kernel would therefore need to be modified to treat the swap file as special and disallow write attempts (even those by root). Windows Vista already employs such an approach to prevent modifications to the page file [48].

3.3 Through Memory Access

If write access to physical pages containing kernel code (or data) is allowed, the potential exists for an application to modify the running kernel. If only read access is allowed to kernel memory, the threat of being able to modify kernel code is mitigated (although the threat of a user-space process gaining access to sensitive kernel or application information is not). On Linux, the kernel can restrict write access to `/dev/mem` and `/dev/kmem`. These restrictions have been configurable since version 2.6.26 of the main-line Linux kernel, and need only be enabled when building the kernel [63]. Before being introduced in the mainline Linux kernel, the options had been used in Fedora and other Red Hat kernels for 4 years without any known problems [64].

The potential exists to restrict some of the actions of user-space applications on physical memory without completely disabling the interface. The kernel can selectively allow access to memory mapped hardware without allowing access to those areas of physical memory associated with the kernel. This was the approach taken for restricting access to `/dev/mem` in

Linux version 2.6.26. Only those areas of physical memory associated with I/O (e.g., the graphics card) can be accessed through `/dev/mem`. Of course, the abilities made available through memory mapped I/O still need to be limited such that hardware can not be used to modify the kernel (see Section 2.4). No areas of the physical address space associated with RAM can be written to through `/dev/mem` when the option is enabled. `/dev/mem` cannot be disabled entirely as X (the graphical display manager typically used in Linux) uses it to communicate with the video card.

In Windows, writes directly to physical memory are done using the `\Device\PhysicalMemory` device [13]. Any attempt to access this device node by a user-space application, however, has been denied since Windows 2003 SP1 [39, 48].

3.4 Through Hardware Access

For user-space software to gain write access to kernel memory through the underlying hardware, the kernel would first have to allow the user-space applications sufficient access to the hardware device. The kernel is typically responsible for mediating all access to the underlying hardware, and we therefore see no fundamental reason why the kernel would not be able to sufficiently limit allowed hardware requests in such a way to prevent user-space applications from modifying the kernel. This may involve restricting both the hardware interface and allowed operations exposed to applications. The exact method for ensuring that applications cannot modify the kernel via hardware is highly dependent on the exact hardware in question, and therefore beyond the scope of this paper.

The hardware I/O memory management unit (IOMMU) [3, 4] presents the equivalent of virtualized memory to hardware devices, restricting the physical memory which can be accessed by a hardware device to only those areas which are assigned to it by the kernel through the mapping stored in the IOMMU page tables. The IOMMU can be used to prevent devices from writing to arbitrary memory (and hence modifying the kernel).

3.5 Through Kernel Tuning Parameters

From Section 2.5, the kernel can be tuned by modifying parameters made available through virtual file systems, as well as tuning parameters passed to the kernel during boot and module loading. While kernel authors limit the ability to modify many kernel parameters to those processes running as super-user, we are not aware of any specific focus on preventing the super-user from arbitrarily modifying the kernel through available tuning parameters. This is not surprising, since in the past much of the focus was on preventing unprivileged users from gaining kernel level privileges.

3.6 Through Changing the Boot Process

Though it requires a machine reboot, updating of the kernel image on disk remains an avenue through which the OS kernel can be modified. There are two methods for preventing a modified kernel from being run on a system reboot: (1) preventing arbitrary modification to disk blocks containing the boot loader, kernel, or kernel modules; or (2) detecting and refusing to execute a modified boot loader, kernel, or kernel module. Approach 1 was attempted for boot-sector virus protection, but was limited to those applications using the BIOS to overwrite the boot sector [38]. A more comprehensive implementation of approach 1 uses the currently running kernel to enforce that disk blocks containing the boot loader, kernel, and kernel modules cannot be written to arbitrarily (compare to Section 3.2). Approach 2 has been used on more specialized hardware such as video-game consoles [27], but as-of-yet has not been deployed on standard desktops to our knowledge. We discuss approach 2 further in Section 4.1.

First discussed above in Section 2.6, `kexec` provides a mechanism for booting a new kernel without rebooting the machine. To prevent booting an arbitrary kernel by using `kexec`, one must either limit the kernels that can be run through the `kexec` system call (similar to verification of kernels during the boot process), or disable the call entirely. Limiting the kernels allowed prevents arbitrary code from running with kernel level control of the system. In current (unmodified) Linux kernels, `kexec` is by default disabled, and set by the compile option `CONFIG_KEXEC` (it is also only available for X86 processors). `kexec` could, however, be modified to only boot signed kernel images which can be validated by the currently running kernel.

3.7 Through Exploiting a Vulnerability

There exist countless tools for detecting and mitigating software vulnerabilities (e.g., static analysis [16, 20] and address space layout randomization [12, 53]). Many of these tools may apply equally well to mitigating kernel vulnerabilities. Some approaches for mitigating kernel level vulnerabilities have been developed specifically for the kernel. Since Linux version 2.6.23, the feature has existed to prevent null pointer dereference vulnerabilities in the kernel from causing arbitrary code execution. The approach works by preventing an application from allocating memory at address 0 [46]. While super-user processes can disable this mechanism, most user-space processes are now prevented from exploiting such a vulnerability (except for denial of service).

4 Advanced Protection Mechanisms

Many of the kernel protection methods in Section 3 follow the approach of simply denying all access. While such an approach may be appropriate in some environments, more thorough protection mechanism proposals attempt to address short-comings of a blanket deny policy.

4.1 Using the Trusted Computing Platform

The trusted computing platform [1, 61] can be used to verify the boot loader and operating system as it is loaded [31, 60]. By having each component in the boot process verify the next before it is executed, the ability to run arbitrarily modified boot-loaders, kernels, and kernel modules is restricted (since any modification to the boot process is detected and the boot halted). Without disabling other methods of modifying the kernel, however, a cryptographic hash of the kernel at boot time alone is insufficient to verify the integrity of the currently running kernel. In order to do the latter, the integrity verification must include all applications which have ever run on the system since boot along with the untrusted input they consumed [50], or all other methods of modifying the kernel, as discussed in Sections 3.1 through 3.7 above, must be restricted.

4.2 Using Fine-Grained Mandatory Access Control

AppArmor [7, 11] and SELinux [30, 56] are two approaches which provide additional fine-grained access control for the Linux kernel. Both use the *Linux Security Module* (LSM) hooks available within the kernel, but base their protection mechanisms on different sources of information. While SELinux uses security contexts (labels) in making security decisions, AppArmor confines applications based on their paths and the paths of the resources they want to access.

AppArmor confines applications only if there is a profile defined. Applications without associated profiles run unconfined, limited only by the *discretionary access control* (DAC) permissions on the system. For each application, a profile will specify the permissions on files that application is allowed to access. If a permission is not specified, then the access is denied. AppArmor can also meditate use of POSIX capabilities [29], and network access. While profiles may vary across installations, AppArmor has the ability to prevent modification of the kernel by blocking write access to devices exposed through the file-system. An application's access to devices exposed under `/dev` can be denied, as can access to `/boot` and `/lib/modules`. AppArmor has the potential to limit, but does not provide blanket protection against any application modifying the running kernel.

SELinux provides a mechanism for administrators to label OS objects associated with kernel inputs and data structures. Access decisions are made based on the labels assigned to the object. Most deployments of SELinux use the reference policy [52], which provides (1) labels for some processes that may run on the system, including the kernel and multiple well known applications; (2) labels for file-systems and files, including kernel relevant files; and (3) the rules that specify access control (what subjects have access to what objects, and what kind of access). The reference policy does not attempt to restrict the permission of many services, including package managers and virtualization tools – these services are allowed to perform raw disk IO and write to kernel images. Other applications, such as the X server, are granted access to kernel memory as a side-effect of being granted access to video card memory (SELinux does not require that the kernel be built with limited access to `/dev/mem`, as discussed in Section 3.3). While the loading of kernel modules is restricted to the two programs normally tasked with performing the loading operations – `insmod` and `modprobe` – other applications are allowed to call `insmod` and `modprobe` to perform the module loading on their behalf. In practice, SELinux ends up mainly being used to confine network-facing daemons, a policy goal first proposed by AppArmor.

4.3 Preventing Hard Drive Writes

Many proposals exist which attempt to either restrict the writing to hard drive sectors, or detect malicious writes. Butler et al. [8] proposed a method where regions of disk were marked as requiring a specific USB key to be inserted before they could be updated. The approach works at the block level, underneath both the file-system and kernel. Blocks on disk become marked as associated with a USB key when they are updated while the key is installed, and can subsequently only be updated when the USB key is inserted. Pennington et al. [42] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications. Strunk et al. [59] proposed logging all file modifications for a period of time to assist in the recovery after malware infection.

4.4 Detecting Kernel Modifications

By far, the most common approach for dealing with OS kernel level malware is to have a detection mechanism installed outside the kernel that observes the kernel. Copilot [43], for example, operates as a distinct hardware device on the system. It monitors the OS kernel in an attempt to detect changes to static kernel elements such as privileged processor code. It also provides a mechanism for partial restoration of changes made by malicious kernel rootkits. Petroni et al. [44] likewise use a system device to detect changes in the OS kernel, but concentrate on protecting dynamic data structures.

Carbone et al. [9] take a snapshot of memory allocated to the running kernel and attempt to map all dynamic data contained within it. Using the memory snapshot and the corresponding kernel source code, they create a directed graph of memory usage within the snapshot. They then check function pointers and detect hidden objects as a method for detecting kernel rootkits. The approach operates offline, and works to detect rather than prevent rootkits.

There is also extensive literature suggesting various approaches leveraging virtual machine monitor technology to detect kernel malware (e.g., see recent papers [57, 68]).

5 Concluding Remarks

Some experts have abandoned hope of securing the kernel without additional support from virtual machines, hardware, or other external supports. Others, however, continue to rely on the kernel to enforce mandatory access control policies (e.g., using SELinux) on user-space applications. We provide a novel (and to our knowledge, complete[3]) identification of standard interfaces which must be restricted to ensure privilege separation between user-space processes and the kernel (root level privilege). The main approach pursued to date is to deny access methods through a few simple mechanisms which can be (or already have been, in some cases) deployed.

Acknowledgement

We thank Sandra Reuda, Hayawardh Vijayakumar, Josh Schiffman, David Lie, Andy Warfield, and Mohammad Mannan who provided feedback and input for this work. The second author is a Canada Research Chair in Authentication and Software Security, and acknowledges NSERC

[3]This list is complete with respect to standard interfaces and access methods available on commodity operating systems to user-space applications. As noted, this scope excludes from discussion other attack vectors such as those requiring physical access.

for funding the chair and a discovery grant; partial funding from NSERC ISSNet is also acknowledged.

References

[1] *Trusted Computing Platforms: TCPA Technology in Context.* Prentice Hall, 2002.

[2] *Modern Operating Systems*, chapter Chapter 1.3.7. Prentice Hall, 3rd edition, 2008.

[3] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 2006. `http://www.intel.com/technology/itj/2006/v10i3/`.

[4] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*, revision 1.26 edition, Feb 2009.

[5] J. L. Baer. *Computer Systems Architecture.* Computer Science Press, 1980.

[6] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of rootkit attacks. Technical Report DCS-TR-593, Rutgers Univ. Dpt. of Computer Science, January 2006.

[7] M. Bauer. Paranoid penguin: an introduction to Novell AppArmor. In *Linux Journal*, number 148, page 13. Aug 2006.

[8] K. R. B. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-resistant disks. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 403–415, Oct 2008.

[9] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systemic integrity checking. In *Proc. 16th ACM Conference on Computer and Communications Security*, Oct 2009.

[10] D. Chanet, J. Cabezas, E. Morancho, N. Navarro, and K. D. Bosschere. Linux kernel compaction through cold code swapping. In *Transactions on High-Performance Embedded Architectures and Compilers II*, pages 173–200, 2009.

[11] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *Proc. LISA '00: 14th Systems Administration Conference*, pages 341–354, 2000.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo*, pages 119–129, Jan 2000.

[13] crazylord. Playing with Windows /dev/(k)mem. In *Phrack*, volume 0x0b (0x3b), chapter 0x10. Jul 2002. `http://www.phrack.com/issues.html?issue=59&id=16`.

[14] Vulnerability summary for cve-2008-0010 - copy_from_user_mmap_sem. CVE, Feb 2008. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0010`.

[15] Vulnerability summary for cve-2010-3081 - compact_alloc_user_space. CVE, Sep 2010. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3081`.

[16] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[17] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proc. 16th ACM Conference on Computer and Communications Security*, Oct 2009.

[18] M. Dornseif. Owned by an iPod. In *PacSec*, 2004.

[19] ECMA. *ECMA 107: Volume and File Structure of Disk Cartridges for Information Interchange*, 2nd edition, Jun 1995. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-107.pdf`.

[20] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. In *IEEE Software*, number 1 in 19, pages 42–51. IEEE Computer Society, Jan 2002.

[21] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. 2003 Network and Distributed Systems Security Symposium*, pages 191–206. Internet Society, February 2003.

[22] A. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold boot attacks on encryption keys. In *Proc. 17th USENIX Security Symposium*, pages 45–60, August 2008.

[23] J. Heasman. Implementing and detecting an ACPI BIOS rootkit. In *Proc. Blackhat Federal*, 2006.

[24] J. Heasman. Implementing and detecting a PCI rootkit. In *Proc. Blackhat Federal*, 2007.

[25] B. Henderson. *Linux Loadable Kernel Module HowTo*, v1.06 edition, Jan 2005.

[26] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[27] A. Huang. *Hacking the Xbox*. No Starch Press, Inc., 2003.

[28] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proc. 3rd USENIX Windows NT Symposium*, Jul 1999.

[29] Draft standard for information technology - portable operating system interface (POSIX) part 1: System application program interface (API). Technical Report IEEE Std 1003.1e, IEEE Computer Society, 1997.

[30] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proc. 12th USENIX Security Symposium*, pages 59–74, Aug 2003.

[31] B. Kauer. OSLO: Improving the security of trusted computing. In *Proc. 16th USENIX Security Symposium*, pages 229–237, Aug 2007.

[32] How to: Building your own kernel space keylogger. Web Page, Jun 2010. `http://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html`.

[33] S. King, P. Chen, Y.-M. Wang, C. Verbowski, H. Wang, and J. Lorch. Subvirt: Implementing malware with virtual machines. In *Proc. 2006 IEEE Symposium on Security and Privacy*, pages 314–327, May 2006.

[34] J. Kong. *Designing BSD Rootkits: An Introduction to Kernel Hacking*. No Starch Press, 2007.

[35] G. Kroah-Hartman. Signed kernel modules. *Linux Journal*, 117:48–53, January 2004.

[36] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 91–100. IEEE Computer Society, 2004.

[37] R. Love. *Linux Kernel Development*. Novell Press, second edition, 2005.

[38] Microsoft. *Built-In Anti-Virus Support in Windows 95*, 1.1 edition, Nov. 2006. `http://support.microsoft.com/kb/q143281/`.

[39] Microsoft Corporation. Device\PhysicalMemory object. TechNet Article (viewed 20 Feb 2010). `http://technet.microsoft.com/en-us/library/cc787565(WS.10).aspx`.

[40] P. Mochel. The sysfs filesystem. In *Proc. Ottawa Linux Symposium*, 2005.

[41] WriteFileEx function. Web Page, Nov 2008. http://msdn.microsoft.com/en-us/library/aa365748(VS.85).aspx.

[42] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proc. 12th USENIX Security Symposium*, pages 137–151, August 2003.

[43] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. 13th USENIX Security Symposium*, pages 179–194, August 2004.

[44] N. L. Petroni Jr., T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. 15th USENIX Security Symposium*, pages 289–304, August 2006.

[45] A. Pfiffer. *Reducing System Reboot Time With kexec.* Open Source Development Labs, Inc., Apr 2003.

[46] Red Hat, Inc. How to mitigate against null pointer dereference vulnerabilities? Web Page, Jun 2010. https://access.redhat.com/kb/docs/DOC-20536.

[47] E. D. Reilly. *Encyclopedia of Computer Science*, page 1152. John Wiley and Sons, Ltd., 4th edition edition, 2003.

[48] N. Ruff. Windows memory forensics. *Journal in Computer Virology*, 4(2):83–100, May 2008.

[49] J. Rutkowska. Subverting Vista kernel for fun and profit. Blackhat Presentation, August 2006. http://blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

[50] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proc. 13th USENIX Security Symposium*, pages 223–238, Aug 2004.

[51] sd and devik. Linux on-the-fly kernel patching without LKM. In *Phrack*, volume 0x0b (0x3a), chapter 0x07. Dec 2001. http://www.phrack.org/issues.html?issue=58&id=7.

[52] SELinux reference policy. Web Page (accessed 31 Dec 2010). http://oss.tresys.com/projects/refpolicy.

[53] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 298–307, Oct 2004.

[54] M. Sharif, W. Lee, and W. Cui. Secure in-VM monitoring using hardware virtualization. In *Proc. 16th ACM Conference on Computer and Communications Security*, Oct 2009.

[55] R. Siemsen. *The NetWinder Firmware HOWTO*, Dec 2001. http://www.netwinder.org/howto/Firmware-HOWTO-all.html.

[56] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Technical Report 01-043, NAI Labs, 2002.

[57] A. Srivastava and J. Giffin. Efficient monitoring of untrusted kernel-mode execution. In *Proc. 2011 Network and Distributed Systems Security Symposium*. Internet Society, February 2011.

[58] W. Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, 4th edition, 2001.

[59] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, Oct 2000.

[60] Trusted boot open source project. Web site (viewed 13 Mar 2011). http://sourceforge.net/projects/tboot/.

[61] Trusted Computing Group, Inc. *TCG Specification Architecture Overview*, version 1.4 edition, 2003.

[62] Unified EFI, Inc. *Unified Extensible Firmware Interface Specification*, version 2.3, errata b edition, Feb 2010.

[63] A. van de Ven. make /dev/kmem a config option. GIT Commit, Apr 2008. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=b781ecb6a379f155568ef7093e38c6c1d857fe53`.

[64] A. van de Ven. x86: Introduce /dev/mem restrictions with a config option. GIT Commit, Apr 2008. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=ae531c26c5c2a28ca1b35a75b39b3b256850f2c8`.

[65] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 368–377, June 2005.

[66] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. 16th ACM Conference on Computer and Communications Security*, Nov 2009.

[67] R. Wojtczuk and J. Ruthowska. *Attacking SMM Memory via Intel CPU Cache Poisoning*. Invisible Things Lab, 2009. `http://www.invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf`.

[68] X. Xiong, D. Tian, and P. Liu. Practical protection of kernel integrity for commodity OS from untrusted extensions. In *Proc. 2011 Network and Distributed Systems Security Symposium*. Internet Society, February 2011.

[69] V. Zimmer, M. Rothman, and R. Hale. *Beyond BIOS: Implementing the Unified Extensible Firmware Interface with Intel's Framework*. Intel Press, 2006.